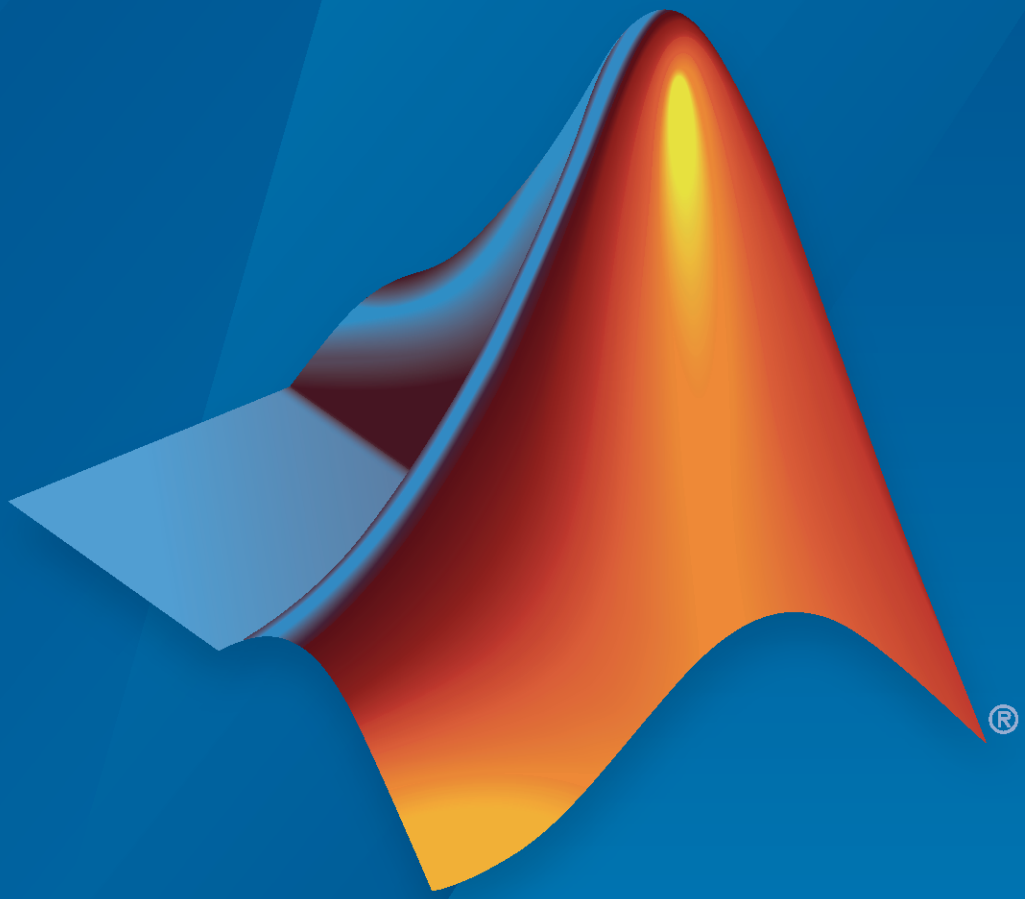
DSP System Toolbox™

User's Guide



MATLAB® & SIMULINK®

R2020a

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

DSP System Toolbox™ User's Guide

© COPYRIGHT 2011–2020 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

April 2011	First printing	Revised for Version 8.0 (R2011a)
September 2011	Online only	Revised for Version 8.1 (R2011b)
March 2012	Online only	Revised for Version 8.2 (R2012a)
September 2012	Online only	Revised for Version 8.3 (R2012b)
March 2013	Online only	Revised for Version 8.4 (R2013a)
September 2013	Online only	Revised for Version 8.5 (R2013b)
March 2014	Online only	Revised for Version 8.6 (R2014a)
October 2014	Online only	Revised for Version 8.7 (R2014b)
March 2015	Online only	Revised for Version 9.0 (R2015a)
September 2015	Online only	Revised for Version 9.1 (R2015b)
March 2016	Online only	Revised for Version 9.2 (R2016a)
September 2016	Online only	Revised for Version 9.3 (R2016b)
March 2017	Online only	Revised for Version 9.4 (R2017a)
September 2017	Online only	Revised for Version 9.5 (R2017b)
March 2018	Online only	Revised for Version 9.6 (R2018a)
September 2018	Online only	Revised for Version 9.7 (R2018b)
March 2019	Online only	Revised for Version 9.8 (R2019a)
September 2019	Online only	Revised for Version 9.9 (R2019b)
March 2020	Online only	Revised for Version 9.10 (R2020a)

Contents

1 DSP Tutorials

Introduction to Streaming Signal Processing in MATLAB 1-2

Filter Frames of a Noisy Sine Wave Signal in MATLAB 1-6

Filter Frames of a Noisy Sine Wave Signal in Simulink 1-8
- Open Model 1-8
- Inspect Model 1-8
- Compare Original and Filtered Signal 1-10

Lowpass Filter Design in MATLAB 1-12

Lowpass IIR Filter Design in Simulink 1-20
- filterBuilder 1-20
- Butterworth Filter 1-21
- Chebyshev Type I Filter 1-26
- Chebyshev Type II Filter 1-27
- Elliptic Filter 1-29
- Minimum-Order Designs 1-31
- Lowpass Filter Block 1-34
- Variable Bandwidth IIR Filter Block 1-35

Design Multirate Filters 1-36
- Implement an FIR Decimator in MATLAB 1-36
- Implement an FIR Decimator in Simulink 1-39
- Sample Rate Conversion 1-41

Tunable Lowpass Filtering of Noisy Input in Simulink 1-45
- Open Lowpass Filter Model 1-45
- Simulate the Model 1-47

Signal Processing Algorithm Acceleration in MATLAB 1-51
- FIR Filter Algorithm 1-51
- Accelerate the FIR Filter Using codegen 1-52
- Accelerate the FIR Filter Using dspunfold 1-53
- Kalman Filter Algorithm 1-54
- Accelerate the Kalman Filter Using codegen 1-56
- Accelerate the Kalman Filter Using dspunfold 1-57

Signal Processing Acceleration through Code Generation 1-59
- FIR Filter Algorithm 1-59
- Accelerate the FIR Filter Using codegen 1-60
- Accelerate the FIR Filter Using dspunfold 1-61

Multithreaded MEX File Generation **1-64**
Use dspunfold with a MATLAB Function Containing a Stateless Algorithm **1-64**
Using dspunfold with a MATLAB Function Containing a Stateful Algorithm **1-66**
Detecting State Length Automatically **1-67**
Verify Generated Multithreaded MEX Using the Generated Analyzer . . . **1-68**

Fixed-Point Filter Design in MATLAB **1-70**

Visualizing Multiple Signals Using Logic Analyzer **1-76**
Model Programmable FIR Filter **1-76**
Simulation **1-77**
Use the Logic Analyzer **1-78**
Modify the Display **1-79**

Signal Visualization and Measurements in MATLAB **1-84**

Filter Frames of a Noisy Sine Wave Signal using Testbench Generator **1-96**

Create Composite System object **1-100**
Create Multi-Notch Filter **1-100**
Set Up the Multi-Notch Filters **1-100**
Contain System Objects as Private Properties **1-100**
Work with Dependent Properties **1-100**
Use the Multi-Notch Filter - Initialization **1-101**
Use the Multi-Notch Filter - Streaming **1-101**

2 Input, Output, and Display

Discrete-Time Signals **2-2**
Time and Frequency Terminology **2-2**
Recommended Settings for Discrete-Time Simulations **2-3**
Simulink Tasking Modes **2-4**
Other Settings for Discrete-Time Simulations **2-5**
Cross-Rate Operations **2-5**

Continuous-Time Signals **2-8**
Continuous-Time Source Blocks **2-8**
Continuous-Time Nonsource Blocks **2-8**

Create Signals for Sample-Based Processing **2-9**
Create Signals Using Constant Block **2-9**
Create Signals Using Signal From Workspace Block **2-11**

Create Signals for Frame-Based Processing **2-13**
Create Signals Using Sine Wave Block **2-14**
Create Signals Using Signal From Workspace Block **2-15**

Create Multichannel Signals for Sample-Based Processing **2-18**
Multichannel Signals for Sample-Based Processing **2-18**
Create Multichannel Signals by Combining Single-Channel Signals **2-19**
Create Multichannel Signals by Combining Multichannel Signals **2-20**

Create Multichannel Signals for Frame-Based Processing **2-23**
Multichannel Signals for Frame-Based Processing **2-24**
Create Multichannel Signals Using Concatenate Block **2-24**

Deconstruct Multichannel Signals for Sample-Based Processing **2-27**
Split Multichannel Signals into Individual Signals **2-27**
Split Multichannel Signals into Several Multichannel Signals **2-29**

Deconstruct Multichannel Signals for Frame-Based Processing **2-32**
Split Multichannel Signals into Individual Signals **2-33**
Reorder Channels in Multichannel Signals **2-35**

Import and Export Signals for Sample-Based Processing **2-38**
Import Vector Signals for Sample-Based Processing **2-38**
Import Matrix Signals for Sample-Based Processing **2-40**
Export Signals for Sample-Based Processing **2-43**

Import and Export Signals for Frame-Based Processing **2-47**
Import Signals for Frame-Based Processing **2-48**
Export Frame-Based Signals **2-50**

3 Data and Signal Management

Sample- and Frame-Based Concepts **3-2**
Sample- and Frame-Based Signals **3-2**
Model Sample- and Frame-Based Signals in MATLAB and Simulink **3-2**
What Is Sample-Based Processing? **3-3**
What Is Frame-Based Processing? **3-3**

Inspect Sample and Frame Rates in Simulink **3-6**
Sample Rate and Frame Rate Concepts **3-6**
Inspect Signals Using the Probe Block **3-7**
Inspect Signals Using Color Coding **3-9**

Convert Sample and Frame Rates in Simulink **3-13**
Rate Conversion Blocks **3-13**
Rate Conversion by Frame-Rate Adjustment **3-14**
Rate Conversion by Frame-Size Adjustment **3-15**
Frame Rebuffering Blocks **3-17**
Buffer Signals by Preserving the Sample Period **3-19**
Buffer Signals by Altering the Sample Period **3-21**

Buffering and Frame-Based Processing **3-24**
Buffer Input into Frames **3-24**
Buffer Signals into Frames with Overlap **3-26**
Buffer Frame Inputs into Other Frame Inputs **3-28**

Buffer Delay and Initial Conditions 3-30
Unbuffer Frame Signals into Sample Signals 3-31

Delay and Latency 3-35
Computational Delay 3-35
Algorithmic Delay 3-36
Zero Algorithmic Delay 3-36
Basic Algorithmic Delay 3-38
Excess Algorithmic Delay (Tasking Latency) 3-40
Predict Tasking Latency 3-41

Variable-Size Signal Support DSP System Objects 3-46
Variable-Size Signal Support Example 3-46
DSP System Toolbox System Objects That Support Variable-Size Signals 3-46

4 Filter Analysis, Design, and Implementation

Design a Filter in Fdesign — Process Overview 4-2
Process Flow Diagram and Filter Design Methodology 4-2

Use Filter Designer with DSP System Toolbox Software 4-9
Design Advanced Filters in Filter Designer 4-9
Access the Quantization Features of Filter Designer 4-11
Quantize Filters in Filter Designer 4-13
Analyze Filters with a Noise-Based Method 4-18
Scale Second-Order Section Filters 4-22
Reorder the Sections of Second-Order Section Filters 4-25
View SOS Filter Sections 4-28
Import and Export Quantized Filters 4-32
Generate MATLAB Code 4-35
Import XILINX Coefficient (.COE) Files 4-35
Transform Filters Using Filter Designer 4-36
Design Multirate Filters in Filter Designer 4-42
Realize Filters as Simulink Subsystem Blocks 4-50

FIR Nyquist (L-th band) Filter Design 4-53

Digital Frequency Transformations 4-60
Details and Methodology 4-60
Frequency Transformations for Real Filters 4-65
Frequency Transformations for Complex Filters 4-74

Digital Filter Design Block 4-83
Overview of the Digital Filter Design Block 4-83
Select a Filter Design Block 4-84
Create a Lowpass Filter in Simulink 4-85
Create a Highpass Filter in Simulink 4-85
Filter High-Frequency Noise in Simulink 4-86

Filter Realization Wizard **4-90**
Overview of the Filter Realization Wizard **4-90**
Design and Implement a Fixed-Point Filter in Simulink **4-90**
Set the Filter Structure and Number of Filter Sections **4-97**
Optimize the Filter Structure **4-97**

Digital Filter Implementations **4-100**
Using Digital Filter Blocks **4-100**
Implement a Lowpass Filter in Simulink **4-100**
Implement a Highpass Filter in Simulink **4-101**
Filter High-Frequency Noise in Simulink **4-102**
Specify Static Filters **4-105**
Specify Time-Varying Filters **4-106**
Specify the SOS Matrix (Biquadratic Filter Coefficients) **4-106**

Removing High-Frequency Noise from an ECG Signal **4-108**

5 Adaptive Filters

Overview of Adaptive Filters and Applications **5-2**
Adaptive Filters in DSP System Toolbox **5-2**
Choosing an Adaptive Filter **5-4**
Mean Squared Error Performance **5-5**
Common Applications **5-5**

System Identification of FIR Filter Using LMS Algorithm **5-9**

System Identification of FIR Filter Using Normalized LMS Algorithm . . **5-17**

Compare Convergence Performance Between LMS Algorithm and Normalized LMS Algorithm **5-20**

Noise Cancellation Using Sign-Data LMS Algorithm **5-22**

Compare RLS and LMS Adaptive Filter Algorithms **5-26**

Inverse System Identification Using RLS Algorithm **5-29**

Signal Enhancement Using LMS and NLMS Algorithms **5-34**

Noise Cancellation in Simulink Using Normalized LMS Adaptive Filter **5-41**
Create an Acoustic Environment in Simulink **5-41**
LMS Filter Configuration for Adaptive Noise Cancellation **5-42**
Modify Adaptive Filter Parameters During Model Simulation **5-45**

6 Multirate and Multistage Filters

Multirate Filters **6-2**
Why Are Multirate Filters Needed? **6-2**
Overview of Multirate Filters **6-2**

Multistage Filters **6-5**
Why Are Multistage Filters Needed? **6-5**
Optimal Multistage Filters in DSP System Toolbox **6-5**

Compare Single-Rate/Single-Stage Filters with Multirate/Multistage Filters **6-6**

Design of Decimators/Interpolators **6-9**

Filter Banks **6-21**
Dyadic Analysis Filter Banks **6-21**
Dyadic Synthesis Filter Banks **6-23**

Multirate Filtering in Simulink **6-27**

7 Dataflow

Dataflow Domain **7-2**
Specifying Dataflow Domains **7-2**
Simulation of Dataflow Domains **7-2**
Dataflow Parameters **7-3**
Unsupported Simulink Software Features in Dataflow Domains **7-8**

Model Multirate Signal Processing Systems Using Dataflow **7-10**

Multicore Simulation and Code Generation of Dataflow Domains **7-12**
Simulation of Dataflow Domains **7-12**
Code Generation of Dataflow Domains **7-12**
Types of Parallelism **7-12**
Improve Simulation Throughput with Multicore Simulation **7-14**
Generate Multicore Code from a Dataflow Subsystem **7-16**

Multicore Execution using Dataflow Domain **7-19**

Multicore Code Generation for Dataflow Domain **7-27**

8 Simulink HDL Optimized Block Examples in DSP System Toolbox

Implement CIC Decimation Filter for HDL 8-2

Fully Parallel Systolic FIR Filter Implementation 8-5

Partly Serial Systolic FIR Filter Implementation 8-9

Automatic Delay Matching for the Latency of FFT HDL Optimized Block 8-13

9 Simulink Block Examples in Scopes and Data Logging Category

Obtain Measurement Data Programmatically for dsp.SpectrumAnalyzer System object 9-2

Obtain Measurements Data Programmatically for Spectrum Analyzer Block 9-5

10 DSP System Toolbox Simulink block Examples in Signal Input and Output Category

Write and Read Binary Files in Simulink 10-2

Write and Read Matrix Data from Binary Files in Simulink 10-6

Write and Read Fixed-Point Data from Binary Files in Simulink 10-8

Write and Read Character Data from Binary Files in Simulink 10-10

Change the Endianness of the Data in Simulink 10-11

11 Simulink Block Examples in Signal Generation and Operations Category

Delay Signal Using Multitap Fractional Delay 11-2

Bidirectional Linear Sweep **11-7**

Unidirectional Linear Sweep **11-10**

When Sweep Time Is Greater than Target Time **11-12**

Sweep with Negative Frequencies **11-14**

Aliased Sweep **11-17**

Generate Discrete Impulse with Three Channels **11-19**

Generate Unit-Diagonal and Identity Matrices **11-20**

Generate Five-Phase Output from the Multiphase Clock Block **11-21**

Count Down Through Range of Numbers **11-23**

Import Frame-Based Signal From Workspace **11-25**

Import 3-D Array From Workspace **11-26**

Generate Sample-Based Sine Waves **11-27**

Generate Frame-Based Sine Waves **11-28**

Design an NCO Source Block **11-29**

Generate Constant Ramp Signal **11-32**

Averaged Power Spectrum of Pink Noise **11-33**

Downsample a Signal **11-35**

Sample and Hold a Signal **11-38**

Generate and Apply Hamming Window **11-41**

Convert Sample Rate of Speech Signal **11-44**

Unwrap Signal **11-46**

Convolution of Two Inputs **11-48**

Select Rows or Columns from Matrices **11-50**

Convert 2-D Matrix to 1-D Array **11-51**

12 Simulink Block Examples in DSP System Toolbox

Why Does Reading Data from the dsp.AsyncBuffer Object Give a Dimension Mismatch Error in the MATLAB Function Block? 12-2

Why Does the dsp.AsyncBuffer Object Error When You Call read Before write? 12-7

Buffering Input with Overlap 12-9

13 Simulink Block Examples in DSP System Toolbox

Synthesize and Channelize Audio 13-2

Filter input with Butterworth Filter in Simulink 13-9

14 Simulink Block Examples in DSP System Toolbox

Compute the Maximum 14-2

Compute the Running Maximum 14-4

Compute the Minimum 14-6

Compute the Running Minimum 14-8

Compute the Mean 14-10

Compute the Running Mean 14-12

Compute the Histogram of Real and Complex Data 14-14

Extract Submatrix from Input Signal 14-19

Compute Difference of a Matrix 14-21

Compute Maximum Column Sum of Matrix 14-22

Convert Amplitude Signal to dB 14-23

Extract Diagonal of Matrix 14-24

Generate Diagonal Matrix from Vector Input 14-25

Permute Matrix by Row or Column **14-26**

LDL Factorization of 3-by-3 Hermitian Positive Definite Matrix **14-27**

15 Simulink Block Examples in Transforms and Spectral Analysis Category

Analyze a Subband of Input Frequencies Using Zoom FFT **15-2**

16 Transforms, Estimation, and Spectral Analysis

Transform Time-Domain Data into Frequency Domain **16-2**

Transform Frequency-Domain Data into Time Domain **16-4**

Linear and Bit-Reversed Output Order **16-6**
FFT and IFFT Blocks Data Order **16-6**
Find the Bit-Reversed Order of Your Frequency Indices **16-6**

Calculate Channel Latencies Required for Wavelet Reconstruction **16-8**
Analyze Your Model **16-8**
Calculate the Group Delay of Your Filters **16-9**
Reconstruct the Filter Bank System **16-10**
Equalize the Delay on Each Filter Path **16-11**
Update and Run the Model **16-12**
References **16-13**

Estimate the Power Spectrum in MATLAB **16-14**
Estimate the Power Spectrum Using dsp.SpectrumAnalyzer **16-14**
Convert the Power Between Units **16-21**
Estimate the Power Spectrum Using dsp.SpectrumEstimator **16-23**

Estimate the Power Spectrum in Simulink **16-27**
Estimate the Power Spectrum Using the Spectrum Analyzer **16-27**
Convert the Power Between Units **16-36**
Estimate Power Spectrum Using the Spectrum Estimator Block **16-38**

Estimate the Transfer Function of an Unknown System **16-43**
Estimate the Transfer Function in MATLAB **16-43**
Estimate the Transfer Function in Simulink **16-46**

View the Spectrogram Using Spectrum Analyzer **16-50**
Colormap **16-51**
Display **16-52**
Resolution Bandwidth (RBW) **16-52**
Time Resolution **16-55**

Convert the Power Between Units 16-55
Scale Color Limits 16-57

Spectral Analysis 16-59
Welch’s Algorithm of Averaging Modified Periodograms 16-59
Filter Bank 16-62

17 Fixed-Point Design

Fixed-Point Signal Processing 17-2
Fixed-Point Features 17-2
Benefits of Fixed-Point Hardware 17-2
Benefits of Fixed-Point Design with System Toolboxes Software 17-2

Fixed-Point Concepts and Terminology 17-4
Fixed-Point Data Types 17-4
Scaling 17-5
Precision and Range 17-6

Arithmetic Operations 17-8
Modulo Arithmetic 17-8
Two's Complement 17-8
Addition and Subtraction 17-9
Multiplication 17-10
Casts 17-12

System Objects in DSP System Toolbox that Support Fixed-Point 17-15
Get Information About Fixed-Point System Objects 17-15
Set System Object Fixed-Point Properties 17-18
Full Precision for Fixed-Point System Objects 17-19

Simulink Blocks in DSP System Toolbox that Support Fixed-Point 17-20

System Objects Supported by Fixed-Point Converter App 17-26

Convert dsp.FIRFilter Object to Fixed-Point Using the Fixed-Point Converter App 17-27
Create DSP Filter Function and Test Bench 17-27
Convert the Function to Fixed-Point 17-28

Specify Fixed-Point Attributes for Blocks 17-33
Fixed-Point Block Parameters 17-33
Specify System-Level Settings 17-35
Inherit via Internal Rule 17-35
Specify Data Types for Fixed-Point Blocks 17-42

Quantizers 17-49
Scalar Quantizers 17-49
Vector Quantizers 17-53

Create an FIR Filter Using Integer Coefficients **17-58**
Define the Filter Coefficients **17-58**
Build the FIR Filter **17-58**
Set the Filter Parameters to Work with Integers **17-59**
Create a Test Signal for the Filter **17-60**
Filter the Test Signal **17-60**
Truncate the Output WordLength **17-62**
Scale the Output **17-64**
Configure Filter Parameters to Work with Integers Using the set2int Method **17-67**

Fixed-Point Precision Rules for Avoiding Overflow in FIR Filters **17-70**
Output Limits for FIR Filters **17-70**
Fixed-Point Precision Rules **17-72**
Polyphase Interpolators and Decimators **17-73**

18 C Code Generation

Functions and System Objects Supported for C Code Generation **18-2**

Understanding C Code Generation in DSP System Toolbox **18-8**
Generate C and C++ code from MATLAB code **18-8**
Generate C and C++ Code from a Simulink Model **18-8**
Shared Library Dependencies **18-9**
Generate C Code for ARM Cortex-M and ARM Cortex-A Processors . . . **18-10**
Generate Code for Mobile Devices **18-10**

Generate C Code from MATLAB Code **18-12**
Set Up the Compiler **18-12**
Break Out the Computational Part of the Algorithm into a MATLAB Function **18-12**
Make Code Suitable for Code Generation **18-13**
Compare the MEX Function with the Simulation **18-15**
Generate a Standalone Executable **18-15**
Read and Verify the Binary File Data **18-17**
Relocate Code to Another Development Environment **18-18**

Relocate Code Generated from MATLAB Code to Another Development Environment **18-19**
Package the Code **18-19**
Prebuilt Dynamic Library Files (.dll) **18-19**

Generate C Code from Simulink Model **18-21**
Open the Model **18-21**
Configure Model for Code Generation **18-22**
Simulate the Model **18-22**
Generate Code from the Model **18-23**
Build and Run the Generated Code **18-24**

Relocate Code Generated from a Simulink Model to Another Development Environment **18-26**
Package the Code **18-26**
Prebuilt Dynamic Library Files (.dll) **18-28**

How To Run a Generated Executable Outside MATLAB **18-29**

Use Generated Code to Accelerate an Application Deployed with MATLAB Compiler **18-32**

How Is dspunfold Different from parfor? **18-43**
DSP Algorithms Involve States **18-43**
dspunfold Introduces Latency **18-43**
parfor Requires Significant Restructuring in Code **18-43**
parfor Used with dspunfold **18-43**

Workflow for Generating a Multithreaded MEX File using dspunfold . **18-45**
Workflow Example **18-45**

Why Does the Analyzer Choose the Wrong State Length? **18-49**
Reason for Verification Failure **18-50**
Recommendation **18-50**

Why Does the Analyzer Choose a Zero State Length? **18-51**
Recommendation **18-51**

Array Plot with Android Devices **18-52**

19 HDL Code Generation

HDL Code Generation Support for DSP System Toolbox **19-2**
Blocks **19-2**
System Objects **19-3**

Find Blocks and System Objects Supporting HDL Code Generation **19-5**
Blocks **19-5**
System Objects **19-6**

High Throughput HDL Algorithms **19-7**
Blocks with HDL Support for Frame Input **19-7**

HDL Filter Architectures **19-9**
Fully Parallel Architecture **19-9**
Serial Architectures **19-10**
Frame-Based Architecture **19-11**

Subsystem Optimizations for Filters **19-14**
Sharing **19-14**
Streaming **19-14**
Pipelining **19-14**
Area Reduction of Filter Subsystem **19-15**

Area Reduction of Multichannel Filter Subsystem 19-17

20 Links to Category Pages

Signal Management Library 20-2

Sinks Library 20-3

Math Functions Library 20-4

Filtering Library 20-5

21 Designing Lowpass FIR Filters

Lowpass FIR Filter Design 21-2

Controlling Design Specifications in Lowpass FIR Design 21-7

Designing Filters with Non-Equiripple Stopband 21-12

Minimizing Lowpass FIR Filter Length 21-16

22 Filter Designer: A Filter Design and Analysis App

Using Filter Designer 22-2
Choosing a Response Type 22-3
Choosing a Filter Design Method 22-3
Setting the Filter Design Specifications 22-4
Computing the Filter Coefficients 22-6
Analyzing the Filter 22-7
Editing the Filter Using the Pole/Zero Editor 22-11
Converting the Filter Structure 22-14
Exporting a Filter Design 22-16
Generating a C Header File 22-19
Generating MATLAB Code 22-20
Managing Filters in the Current Session 22-21
Saving and Opening Filter Design Sessions 22-22

Importing a Filter Design 22-24
Import Filter Panel 22-24
Filter Structures 22-24

23 Designing a Filter in the Filter Builder GUI

Filter Builder Design Process **23-2**
- Introduction to Filter Builder **23-2**
- Design a Filter Using Filter Builder **23-2**
- Select a Response **23-2**
- Select a Specification **23-4**
- Select an Algorithm **23-5**
- Customize the Algorithm **23-6**
- Analyze the Design **23-7**
- Realize or Apply the Filter to Input Data **23-7**

24 Visualize Data and Signals

Display Time-Domain Data **24-2**
- Configure the Time Scope Properties **24-3**
- Use the Simulation Controls **24-6**
- Modify the Time Scope Display **24-7**
- Inspect Your Data (Scaling the Axes and Zooming) **24-8**
- Manage Multiple Time Scopes **24-10**

Display Frequency-Domain Data in Spectrum Analyzer **24-13**

Visualize Central Limit Theorem in Array Plot **24-16**

Configure Spectrum Analyzer **24-19**
- Signal and Spectrum Computation Information **24-19**
- Generate a MATLAB Script **24-22**
- Spectral Masks **24-24**
- Measurements Panels **24-26**
- Customize Visualization **24-36**
- Zoom and Pan **24-37**

Configure Array Plot MATLAB Object **24-38**
- Signal Display **24-38**
- Multiple Signal Names and Colors **24-39**
- Configure Plot Settings **24-39**
- Use dsp.ArrayPlot Measurements **24-40**
- Share or Save the Array Plot **24-42**
- Scale Axes **24-43**

Configure Array Plot Block **24-44**
- Signal Display **24-44**
- Multiple Signal Names and Colors **24-45**
- Array Plot Measurement Panels **24-46**
- Configuration Properties **24-50**
- Style Dialog Box **24-51**
- Axes Scaling Properties **24-52**

Configure Time Scope **24-53**
Signal Display **24-53**
Display Multiple Signals **24-55**
Time Scope Measurement Panels **24-57**
Style Dialog Box **24-81**
Axes Scaling Properties **24-81**
Sources — Streaming Properties **24-82**

Common Scope Block Tasks **24-83**
Connect Multiple Signals to a Scope **24-83**
Save Simulation Data Using Scope Block **24-85**
Pause Display While Running **24-87**
Copy Scope Image **24-87**
Plot an Array of Signals **24-89**
Scopes in Referenced Models **24-89**
Scopes Within an Enabled Subsystem **24-92**
Show Signal Units on a Scope Display **24-92**
Select Number of Displays and Layout **24-95**
Dock and Undock Scope Window to MATLAB Desktop **24-95**

Display Frequency Input on Spectrum Analyzer **24-97**

Use Peak Finder to Find Heart Rate from ECG Input **24-99**

25 Logic Analyzer

Inspect and Measure Transitions Using the Logic Analyzer **25-2**
Open a Simulink Model **25-2**
Open the Logic Analyzer **25-2**
Configure Global Settings and Visual Layout **25-3**
Set Stepping Options **25-4**
Run Model **25-5**
Configure Individual Wave Settings **25-5**
Inspect and Measure Transitions **25-5**
Step Through Simulation **25-7**
Save Logic Analyzer Settings **25-7**

Configure Logic Analyzer **25-8**

26 Statistics and Linear Algebra

What Are Moving Statistics? **26-2**

Sliding Window Method and Exponential Weighting Method **26-5**
Sliding Window Method **26-5**
Exponential Weighting Method **26-7**

Measure Statistics of Streaming Signals **26-14**
Compute Moving Average Using Only MATLAB Functions **26-14**
Compute Moving Average Using System Objects **26-15**

How Is a Moving Average Filter Different from an FIR Filter? **26-17**
Frequency Response of Moving Average Filter and FIR Filter **26-17**

Energy Detection in the Time Domain **26-21**
Detect Signal Energy **26-21**

Remove High-Frequency Noise from Gyroscope Data **26-24**

Measure Pulse and Transition Characteristics of Streaming Signals . . **26-26**

Linear Algebra and Least Squares **26-34**
Linear Algebra Blocks **26-34**
Linear System Solvers **26-34**
Matrix Factorizations **26-35**
Matrix Inverses **26-36**

27 Bibliography

References — Advanced Filters **27-2**

References — Frequency Transformations **27-3**

28 Audio I/O User Guide

Run Audio I/O Features Outside MATLAB and Simulink **28-2**

29 Block Example Repository

Decrease Underrun **29-2**

DSP Tutorials

- “Introduction to Streaming Signal Processing in MATLAB” on page 1-2
- “Filter Frames of a Noisy Sine Wave Signal in MATLAB” on page 1-6
- “Filter Frames of a Noisy Sine Wave Signal in Simulink” on page 1-8
- “Lowpass Filter Design in MATLAB” on page 1-12
- “Lowpass IIR Filter Design in Simulink” on page 1-20
- “Design Multirate Filters” on page 1-36
- “Tunable Lowpass Filtering of Noisy Input in Simulink” on page 1-45
- “Signal Processing Algorithm Acceleration in MATLAB” on page 1-51
- “Signal Processing Acceleration through Code Generation” on page 1-59
- “Multithreaded MEX File Generation” on page 1-64
- “Fixed-Point Filter Design in MATLAB” on page 1-70
- “Visualizing Multiple Signals Using Logic Analyzer” on page 1-76
- “Signal Visualization and Measurements in MATLAB” on page 1-84
- “Filter Frames of a Noisy Sine Wave Signal using Testbench Generator” on page 1-96
- “Create Composite System object” on page 1-100

Introduction to Streaming Signal Processing in MATLAB

This example shows how to use System objects to do streaming signal processing in MATLAB. The signals are read in and processed frame by frame (or block by block) in each processing loop. You can control the size of each frame.

In this example, frames of 1024 samples are filtered using a notch-peak filter in each processing loop. The input is a sine wave signal that is streamed frame by frame from a `dsp.SineWave` object. The filter is a notch-peak filter created using a `dsp.NotchPeakFilter` object. To ensure smooth processing as each frame is filtered, the System objects maintain the state of the filter from one frame to the next automatically.

Initialize Streaming Components

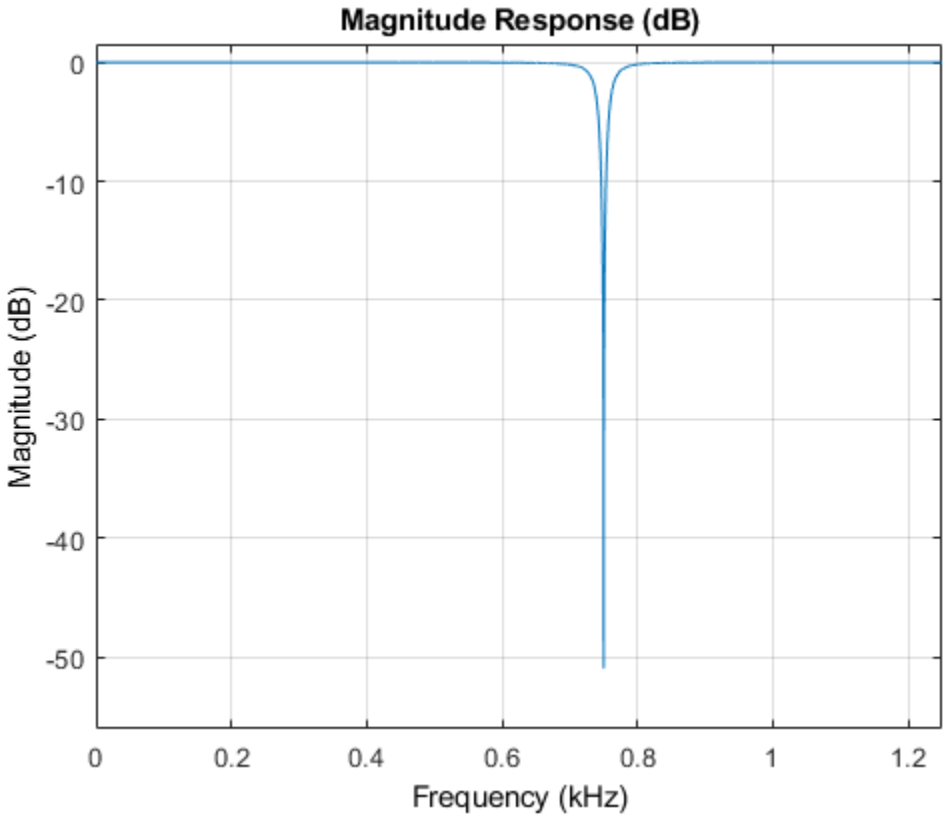
Initialize the sine wave source to generate the sine wave, the notch-peak filter to filter the sine wave, and the spectrum analyzer to show the filtered signal. The input sine wave has two frequencies: one at 100 Hz, and the other at 1000 Hz. Create two `dsp.SineWave` objects, one to generate the 100 Hz sine wave, and the other to generate the 1000 Hz sine wave.

```
Fs = 2500;
Sineobject1 = dsp.SineWave('SamplesPerFrame',1024,...
                      'SampleRate',Fs,'Frequency',100);
Sineobject2 = dsp.SineWave('SamplesPerFrame',1024,...
                      'SampleRate',Fs,'Frequency',1000);

SA = dsp.SpectrumAnalyzer('SampleRate',Fs,'NumInputPorts',2,...
    'PlotAsTwoSidedSpectrum',false,...
    'ChannelNames',{'SinewaveInput','NotchOutput'},'ShowLegend',true);
```

Create Notch-Peak Filter

Create a second-order IIR notch-peak filter to filter the sine wave signal. The filter has a notch at 750 Hz and a Q-factor of 35. A higher Q-factor results in a narrower 3-dB bandwidth of the notch. If you tune the filter parameters during streaming, you can see the effect immediately in the spectrum analyzer output.

```
Wo = 750;
Q  = 35;
BW = Wo/Q;
NotchFilter = dsp.NotchPeakFilter('Bandwidth',BW,...
    'CenterFrequency',Wo, 'SampleRate',Fs);
fvtool(NotchFilter);
```

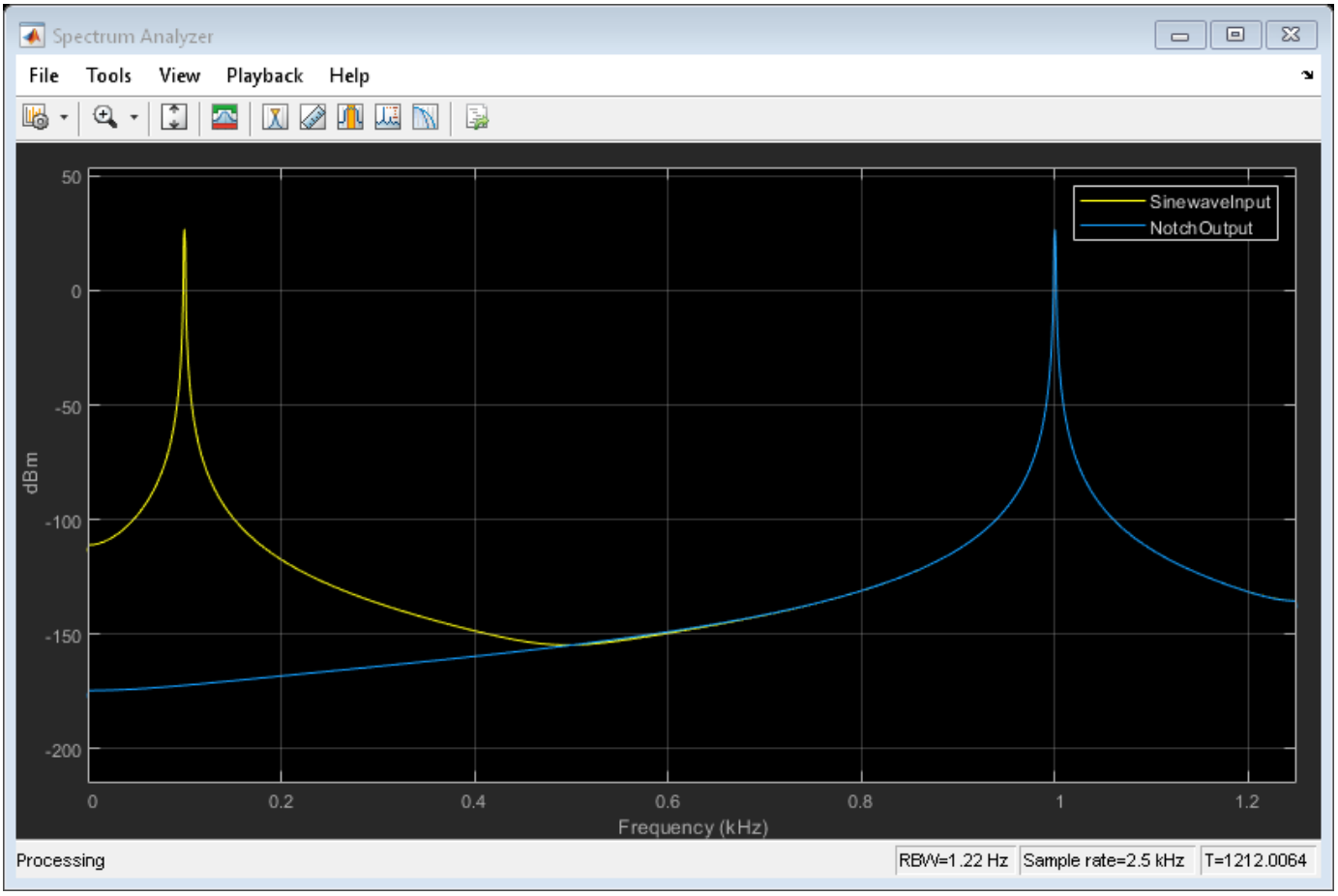


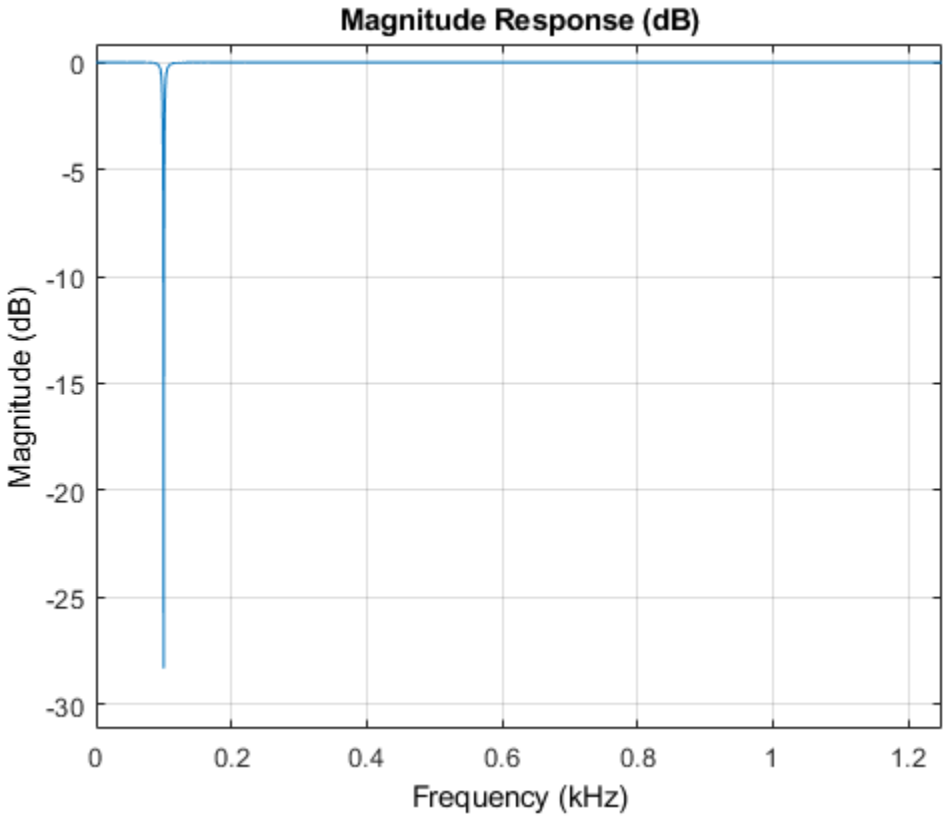
Stream In and Process Signal

Construct a for-loop to run for 3000 iterations. In each iteration, stream in 1024 samples (one frame) of the sinewave and apply a notch filter on each frame of the input signal. To generate the input signal, add the two sine waves. The resultant signal is a sine wave with two frequencies: one at 100 Hz and the other at 1000 Hz. The notch of the filter is tuned to a frequency of 100, 500, 750, or 1000 Hz, based on the value of `VecIndex`. The filter bandwidth changes accordingly. When the filter parameters change during streaming, the output in the spectrum analyzer gets updated accordingly.

```
FreqVec = [100 500 750 1000];
VecIndex = 1;
VecElem = FreqVec(VecIndex);
for Iter = 1:3000
    Sinewave1 = Sineobject1();
    Sinewave2 = Sineobject2();
    Input = Sinewave1 + Sinewave2;
    if (mod(Iter,350)==0)
        if VecIndex < 4
            VecIndex = VecIndex+1;
        else
            VecIndex = 1;
        end
        VecElem = FreqVec(VecIndex);
    end
    NotchFilter.CenterFrequency = VecElem;
    NotchFilter.Bandwidth = NotchFilter.CenterFrequency/Q;
    Output = NotchFilter(Input);
```

```
        SA(Input,Output);
    end
    fvtool(NotchFilter)
```





At the end of the processing loop, the `CenterFrequency` is at 100 Hz. In the filter output, the 100 Hz frequency is completely nulled out by the notch filter, while the frequency at 1000 Hz is unaffected.

See Also

“Filter Frames of a Noisy Sine Wave Signal in MATLAB” on page 1-6 | “Filter Frames of a Noisy Sine Wave Signal in Simulink” on page 1-8 | “Lowpass IIR Filter Design in Simulink” on page 1-20 | “Design Multirate Filters” on page 1-36

Filter Frames of a Noisy Sine Wave Signal in MATLAB

This example shows how to lowpass filter a noisy signal in MATLAB and visualize the original and filtered signals using a spectrum analyzer. For a Simulink version of this example, see “Filter Frames of a Noisy Sine Wave Signal in Simulink” on page 1-8.

Specify Signal Source

The input signal is the sum of two sine waves with frequencies of 1 kHz and 10 kHz. The sampling frequency is 44.1 kHz.

```
Sine1 = dsp.SineWave('Frequency',1e3,'SampleRate',44.1e3);
Sine2 = dsp.SineWave('Frequency',10e3,'SampleRate',44.1e3);
```

Create Lowpass Filter

The lowpass FIR filter, `dsp.LowpassFilter`, designs a minimum-order FIR lowpass filter using the generalized Remez FIR filter design algorithm. Set the passband frequency to 5000 Hz and the stopband frequency to 8000 Hz. The passband ripple is 0.1 dB and the stopband attenuation is 80 dB.

```
FIRLowPass = dsp.LowpassFilter('PassbandFrequency',5000,...
    'StopbandFrequency',8000);
```

Create Spectrum Analyzer

Set up the spectrum analyzer to compare the power spectra of the original and filtered signals. The spectrum units are dBm.

```
SpecAna = dsp.SpectrumAnalyzer('PlotAsTwoSidedSpectrum',false, ...
    'SampleRate',Sine1.SampleRate, ...
    'NumInputPorts',2,...
    'ShowLegend',true, ...
    'YLimits',[-145,45]);

SpecAna.ChannelNames = {'Original noisy signal','Low pass filtered signal'};
```

Specify Samples per Frame

This example uses frame-based processing, where data is processed one frame at a time. Each frame of data contains sequential samples from an independent channel. Frame-based processing is advantageous for many signal processing applications because you can process multiple samples at once. By buffering your data into frames and processing multisample frames of data, you can improve the computational time of your signal processing algorithms. Set the number of samples per frame to 4000.

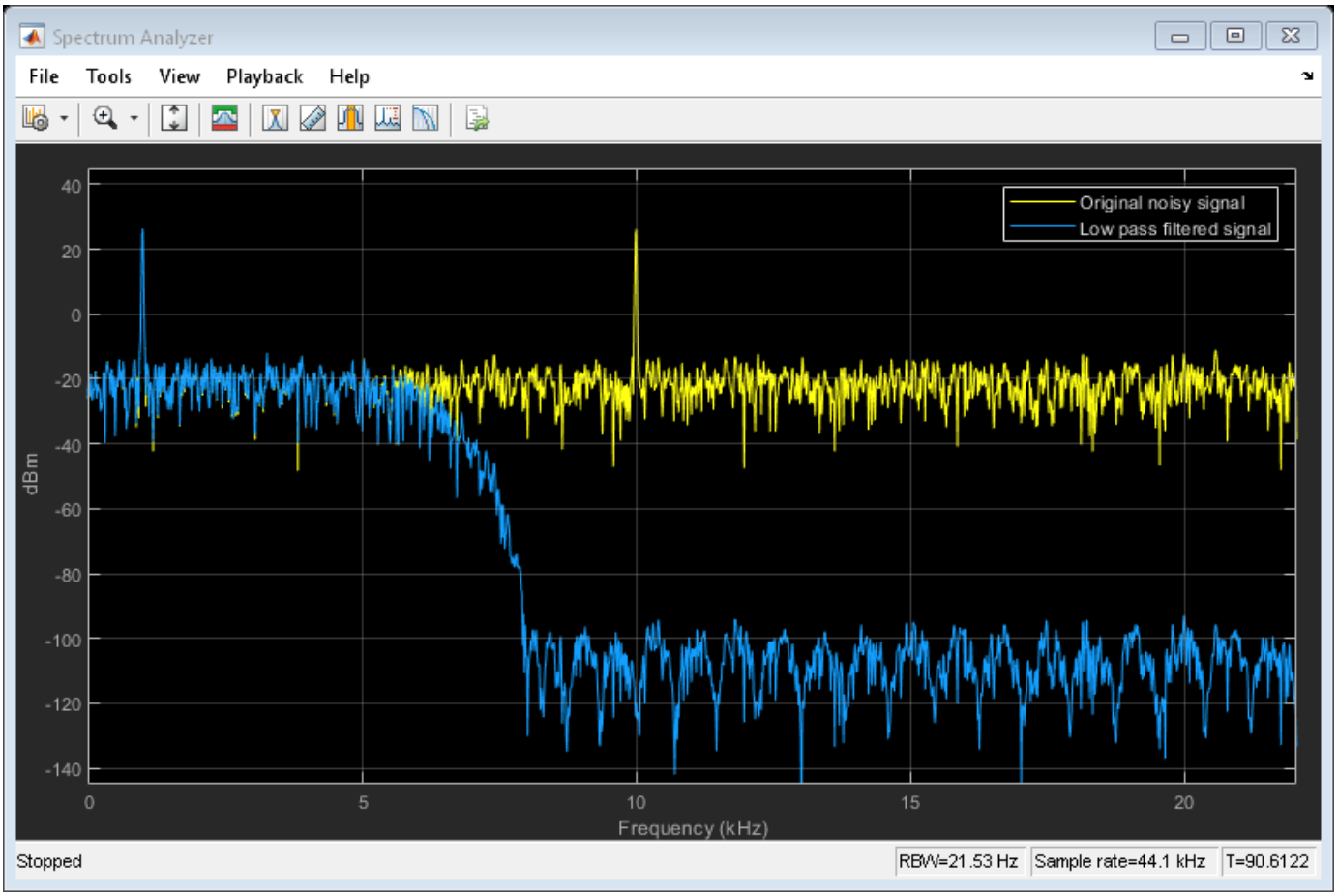
```
Sine1.SamplesPerFrame = 4000;
Sine2.SamplesPerFrame = 4000;
```

Filter the Noisy Sine Wave Signal

Add zero-mean white Gaussian noise with a standard deviation of 0.1 to the sum of sine waves. Filter the result using the FIR filter. While running the simulation, the spectrum analyzer shows that frequencies above 8000 Hz in the source signal are attenuated. The resulting signal maintains the peak at 1 kHz because it falls in the passband of the lowpass filter.

```
for i = 1 : 1000
    x = Sine1()+Sine2()+0.1.*randn(Sine1.SamplesPerFrame,1);
```

```
    y = FIRLowPass(x);
    SpecAna(x,y);
end
release(SpecAna)
```



See Also

“Lowpass Filter Design in MATLAB” on page 1-12 | “Filter Frames of a Noisy Sine Wave Signal in Simulink” on page 1-8 | “Introduction to Streaming Signal Processing in MATLAB” on page 1-2 | “Design Multirate Filters” on page 1-36

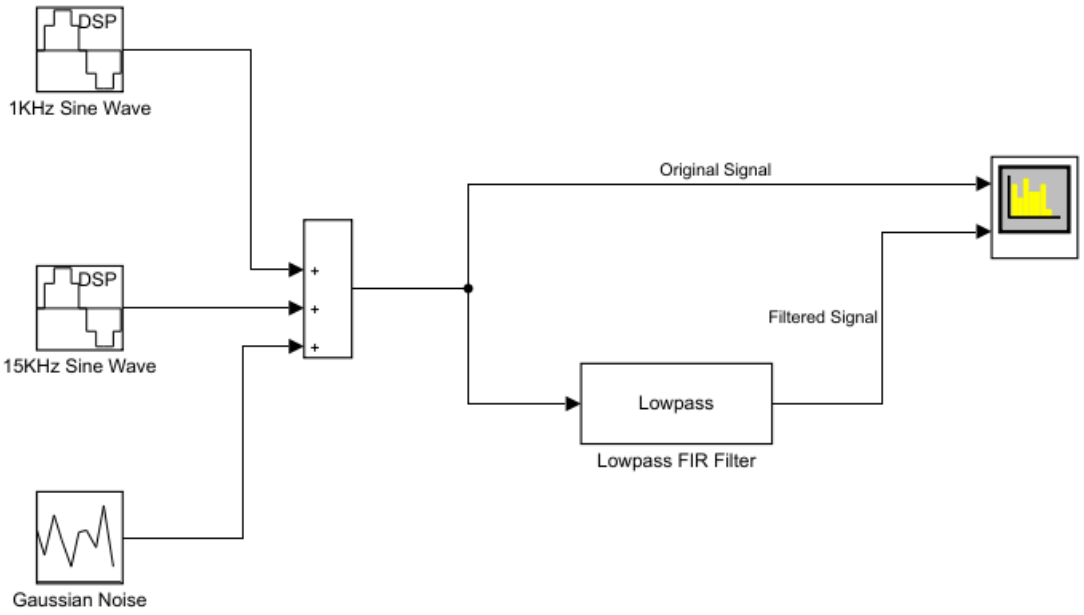
Filter Frames of a Noisy Sine Wave Signal in Simulink

This example shows how to lowpass filter a noisy signal in Simulink® and visualize the original and filtered signals with a spectrum analyzer. For a MATLAB® version of this example, see “Filter Frames of a Noisy Sine Wave Signal in MATLAB” on page 1-6.

Open Model

To create a new blank model and open the library browser:

1. On the MATLAB **Home** tab, click **Simulink**, and choose the **Basic Filter** model template.
2. Click **Create Model** to create a basic filter model opens with settings suitable for use with DSP System Toolbox. To access the library browser, in the **Simulation** tab, click **Library Browser** on the model toolstrip.

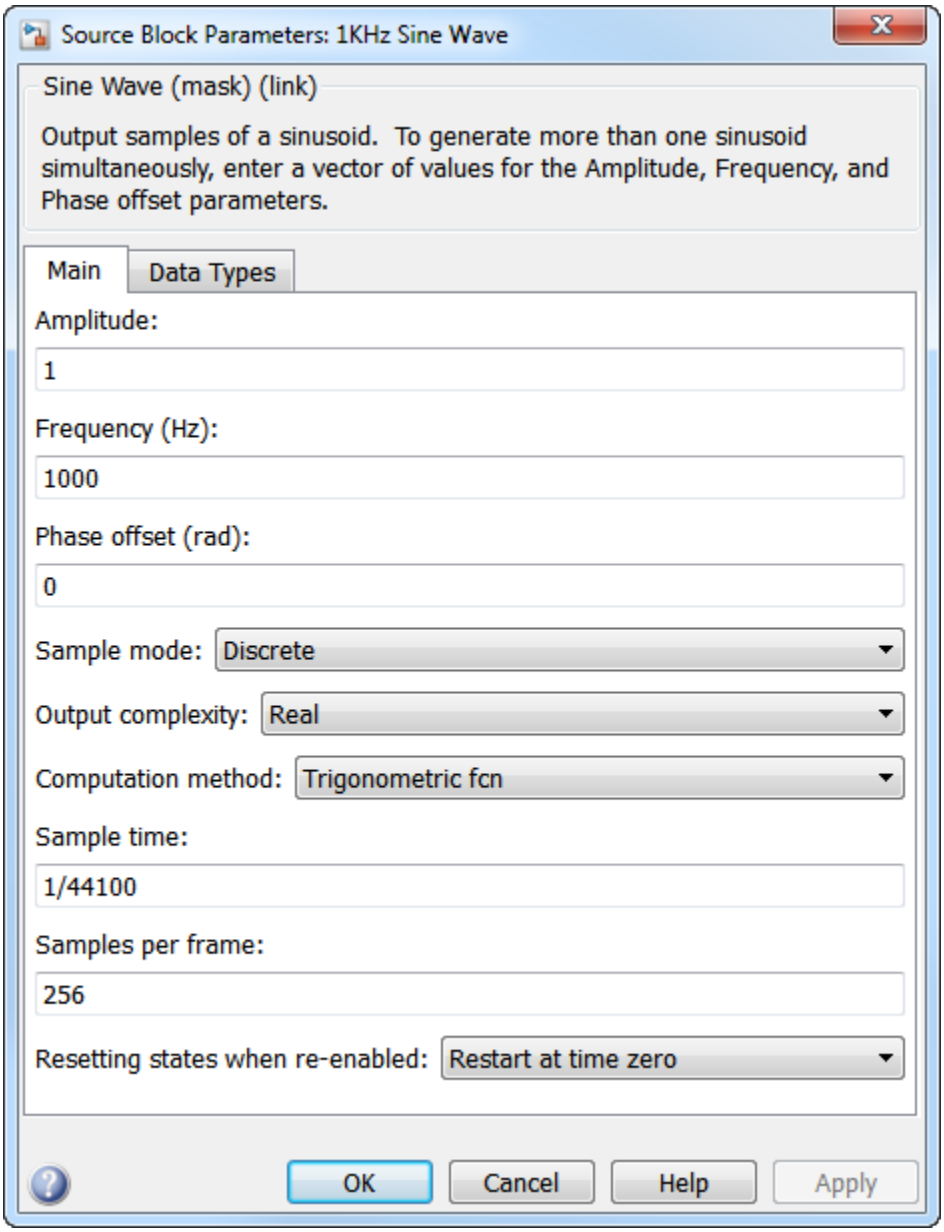


The new model using the template settings and contents appears in the Simulink Editor. The model is only in memory until you save it.

Inspect Model

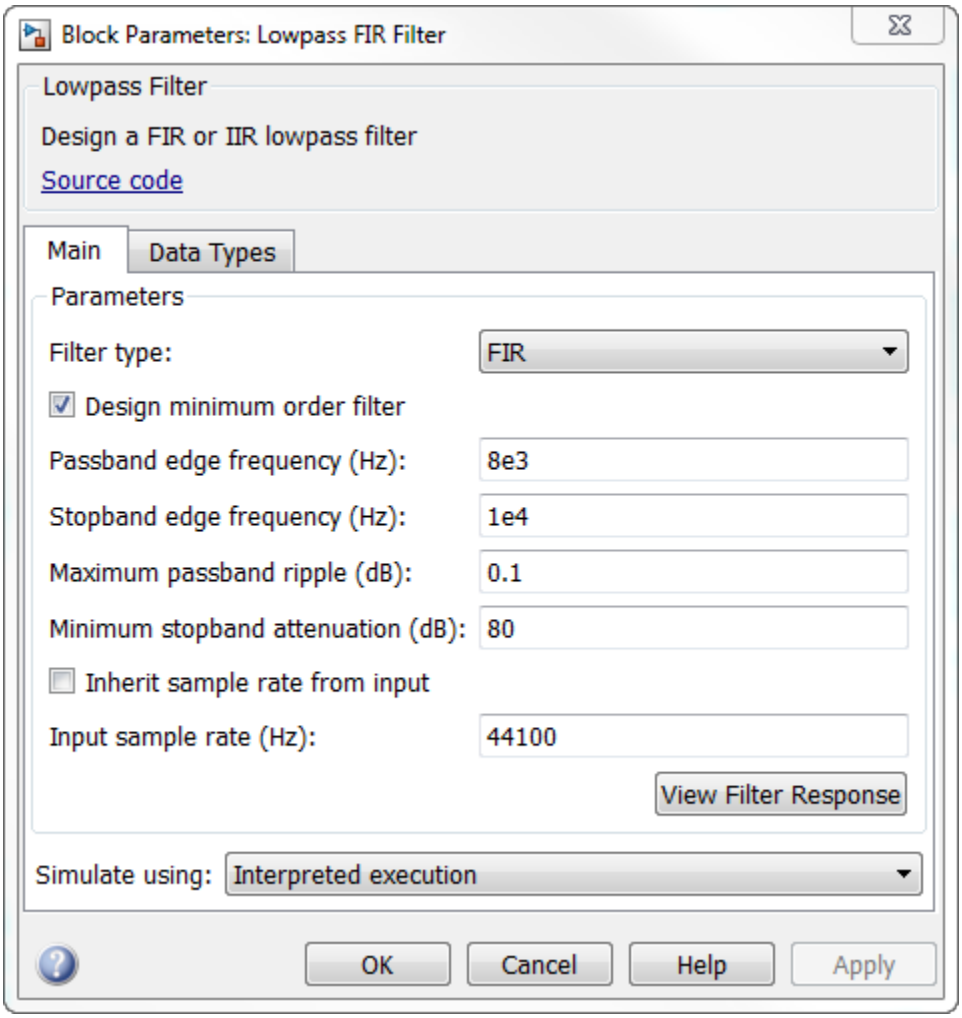
Input Signal

Three source blocks comprise the input signal. The input signal consists of the sum of two sine waves and white Gaussian noise with mean 0 and variance 0.05. The frequencies of the sine waves are 1 kHz and 15 kHz. The sampling frequency is 44.1 kHz. The dialog box shows the block parameters for the 1 kHz sine wave.



Lowpass Filter

The lowpass filter is modeled using a Lowpass Filter block. The example uses a generalized Remez FIR filter design algorithm. The filter has a passband frequency of 8000 Hz, a stopband frequency of 10,000 Hz, a passband ripple of 0.1 dB, and a stopband attenuation of 80 dB.



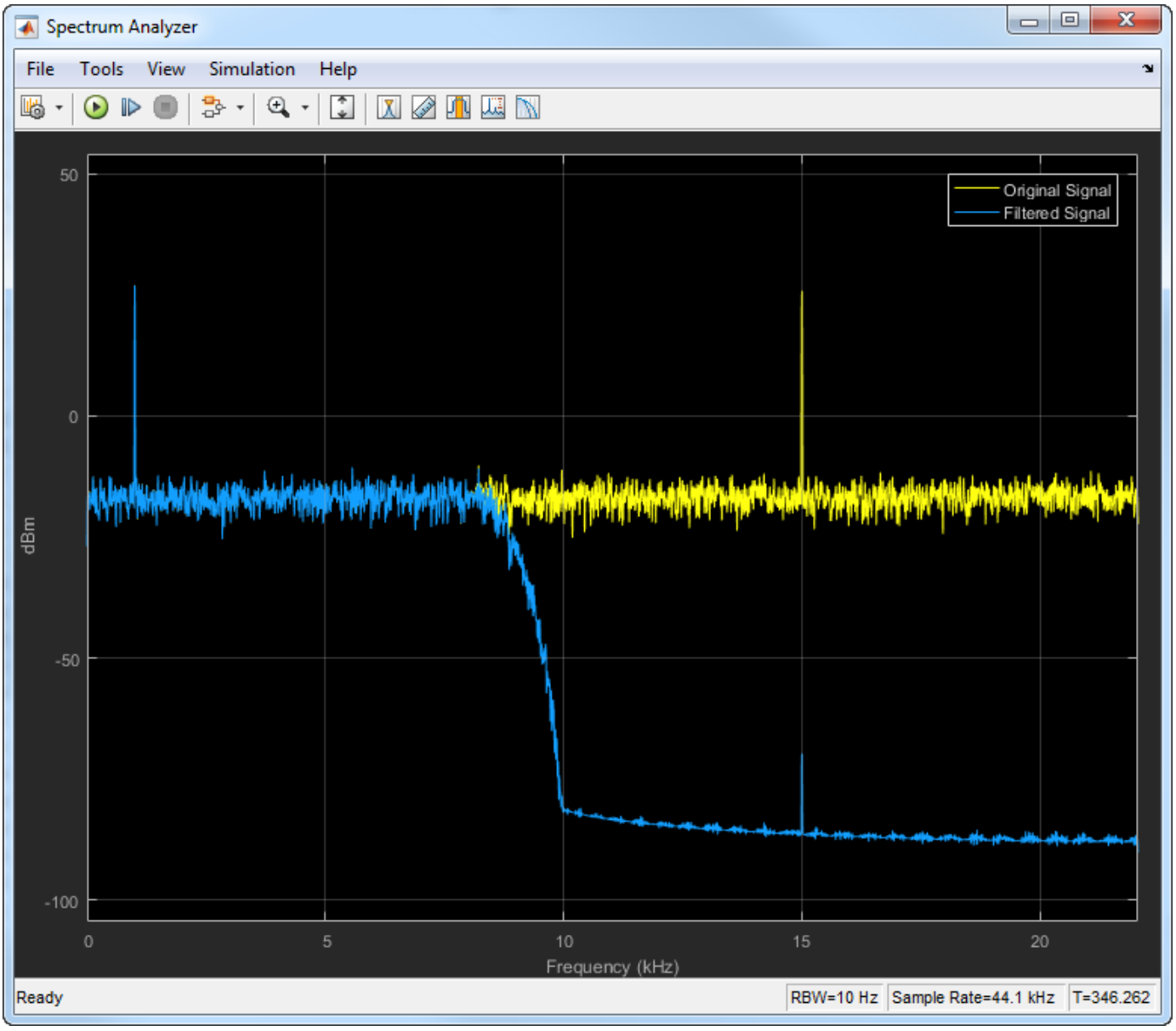
The Lowpass Filter block uses frame-based processing to process data one frame at a time. Each frame of data contains sequential samples from an independent channel. Frame-based processing is advantageous for many signal processing applications because you can process multiple samples at once. By buffering your data into frames and processing multisample frames of data, you can improve the computational time of your signal processing algorithms.

Compare Original and Filtered Signal

Use a Spectrum Analyzer to compare the power spectra of the original and filtered signals. The spectrum units are in dBm.

To run the simulation, in the model, click **Run**. To stop the simulation, in the Spectrum Analyzer block, click **Stop**. Alternatively, you can execute the following code to run the simulation for 200 frames of data.

```
set_param(model,'StopTime','256/44100 * 200')
sim(model);
```



Frequencies above 10 kHz in the source signal are attenuated. The resulting signal maintains the peak at 1 kHz because it falls in the passband of the lowpass filter.

Lowpass Filter Design in MATLAB

This example shows how to design lowpass filters. The example highlights some of the most commonly used command-line tools in the DSP System Toolbox™. Alternatively, you can use the Filter Builder app to implement all the designs presented here. For more design options, see “Designing Low Pass FIR Filters”.

Introduction

When designing a lowpass filter, the first choice you make is whether to design an FIR or IIR filter. You generally choose FIR filters when a linear phase response is important. FIR filters also tend to be preferred for fixed-point implementations because they are typically more robust to quantization effects. FIR filters are also used in many high-speed implementations such as FPGAs or ASICs because they are suitable for pipelining. IIR filters (in particular biquad filters) are used in applications (such as audio signal processing) where phase linearity is not a concern. IIR filters are generally computationally more efficient in the sense that they can meet the design specifications with fewer coefficients than FIR filters. IIR filters also tend to have a shorter transient response and a smaller group delay. However, the use of minimum-phase and multirate designs can result in FIR filters comparable to IIR filters in terms of group delay and computational efficiency.

FIR Lowpass Designs - Specifying the Filter Order

There are many practical situations in which you must specify the filter order. One such case is if you are targeting hardware which has constrained the filter order to a specific number. Another common scenario is when you have computed the available computational budget (MIPS) for your implementation and this affords you a limited filter order. FIR design functions in the Signal Processing Toolbox (including `fir1`, `firpm`, and `firls`) are all capable of designing lowpass filters with a specified order. In the DSP System Toolbox, the preferred function for lowpass FIR filter design with a specified order is `firceqrip`. This function designs optimal equiripple lowpass/highpass FIR filters with specified passband/stopband ripple values and with a specified passband-edge frequency. The stopband-edge frequency is determined as a result of the design.

Design a lowpass FIR filter for data sampled at 48 kHz. The passband-edge frequency is 8 kHz. The passband ripple is 0.01 dB and the stopband attenuation is 80 dB. Constrain the filter order to 120.

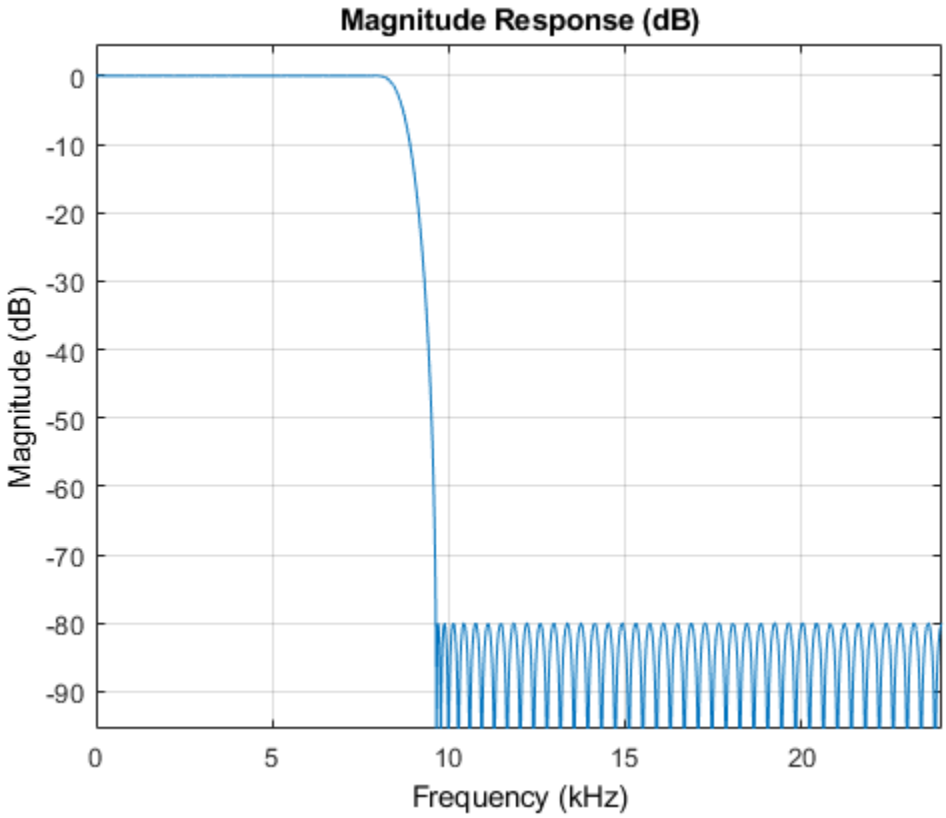
```
N   = 120;
Fs  = 48e3;
Fp  = 8e3;
Ap  = 0.01;
Ast = 80;
```

Obtain the maximum deviation for the passband and stopband ripples in linear units.

```
Rp  = (10^(Ap/20) - 1)/(10^(Ap/20) + 1);
Rst = 10^(-Ast/20);
```

Design the filter using `firceqrip` and view the magnitude frequency response.

```
NUM = firceqrip(N,Fp/(Fs/2),[Rp Rst],'passedge');
fvtool(NUM,'Fs',Fs)
```



The resulting stopband-edge frequency is about 9.64 kHz.

Minimum-Order Designs

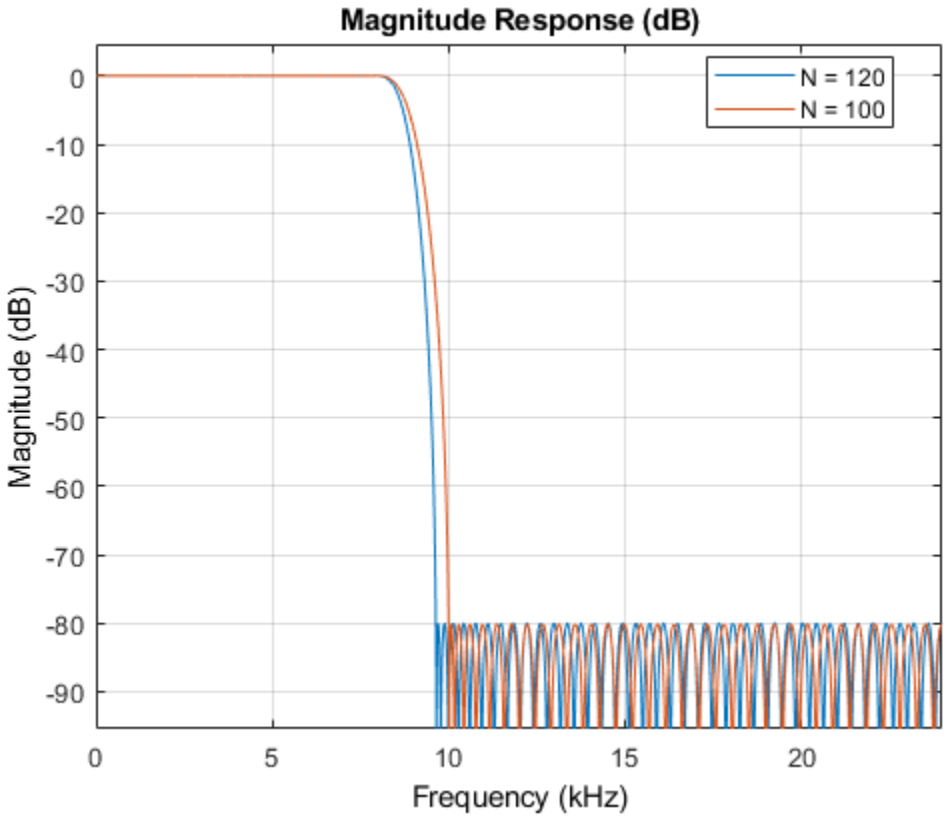
Another design function for optimal equiripple filters is `firgr`. `firgr` can design a filter that meets passband/stopband ripple constraints as well as a specified transition width with the smallest possible filter order. For example, if the stopband-edge frequency is specified as 10 kHz, the resulting filter has an order of 100 rather than the 120th-order filter designed with `firceqrip`. The smaller filter order results from the larger transition band.

Specify the stopband-edge frequency of 10 kHz. Obtain a minimum-order FIR filter with a passband ripple of 0.01 dB and 80 dB of stopband attenuation.

```
Fst     = 10e3;
NumMin = firgr('minorder',[0 Fp/(Fs/2) Fst/(Fs/2) 1], [1 1 0 0],[Rp,Rst]);
```

Plot the magnitude frequency responses for the minimum-order FIR filter obtained with `firgr` and the 120th-order filter designed with `firceqrip`. The minimum-order design results in a filter with order 100. The transition region of the 120th-order filter is, as expected, narrower than that of the filter with order 100.

```
hvft = fvtool(NUM,1,NumMin,1,'Fs',Fs);
legend(hvft,'N = 120','N = 100')
```

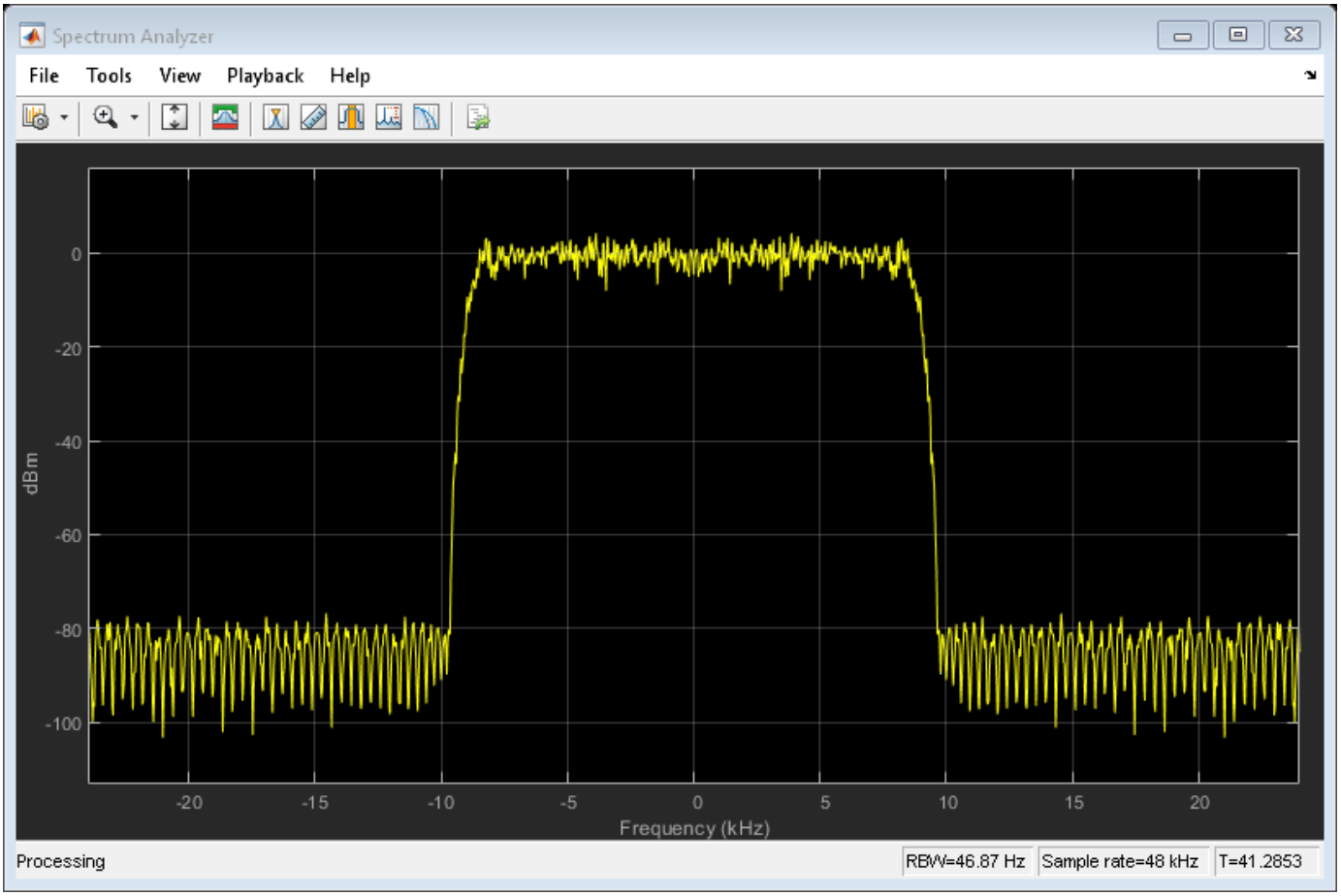


Filtering Data

To apply the filter to data, you can use the `filter` command or you can use `dsp.FIRFilter`. `dsp.FIRFilter` has the advantage of managing state when executed in a loop. `dsp.FIRFilter` also has fixed-point capabilities and supports C code generation, HDL code generation, and optimized code generation for ARM® Cortex® M and ARM Cortex A.

Filter 10 seconds of white noise with zero mean and unit standard deviation in frames of 256 samples with the 120th-order FIR lowpass filter. View the result on a spectrum analyzer.

```
LP_FIR = dsp.FIRFilter('Numerator',NUM);
SA     = dsp.SpectrumAnalyzer('SampleRate',Fs,'SpectralAverages',5);
tic
while toc < 10
    x = randn(256,1);
    y = LP_FIR(x);
    step(SA,y);
end
```



Using `dsp.LowpassFilter`

`dsp.LowpassFilter` is an alternative to using `firceqrip` and `firgr` in conjunction with `dsp.FIRFilter`. Basically, `dsp.LowpassFilter` condenses the two step process into one. `dsp.LowpassFilter` provides the same advantages that `dsp.FIRFilter` provides in terms of fixed-point support, C code generation support, HDL code generation support, and ARM Cortex code generation support.

Design a lowpass FIR filter for data sampled at 48 kHz. The passband-edge frequency is 8 kHz. The passband ripple is 0.01 dB and the stopband attenuation is 80 dB. Constrain the filter order to 120. Create a `dsp.FIRFilter` based on your specifications.

```
LP_FIR = dsp.LowpassFilter('SampleRate',Fs,...
    'DesignForMinimumOrder',false,'FilterOrder',N,...
    'PassbandFrequency',Fp,'PassbandRipple',Ap,'StopbandAttenuation',Ast);
```

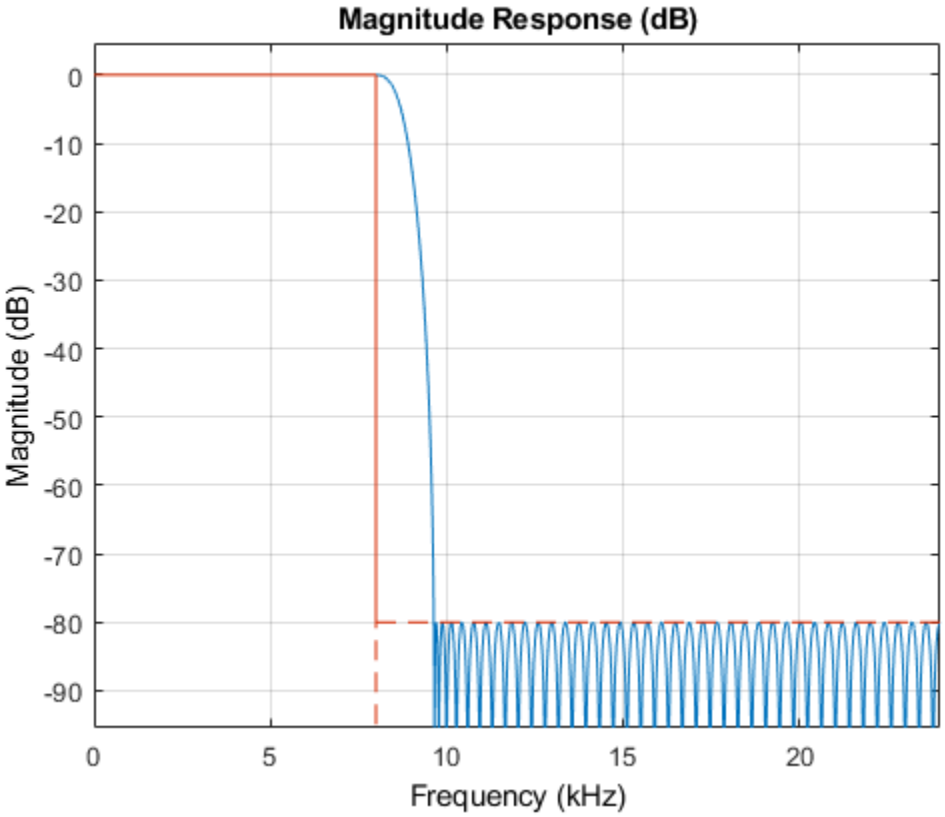
The coefficients in `LP_FIR` are identical to the coefficients in `NUM`.

```
NUM_LP = tf(LP_FIR);
```

You can use `LP_FIR` to filter data directly, as shown in the preceding example. You can also analyze the filter using FVTool or measure the response using `measure`.

```
fvtool(LP_FIR,'Fs',Fs);
measure(LP_FIR)
```

```
ans =

Sample Rate      : 48 kHz
Passband Edge    : 8 kHz
3-dB Point       : 8.5843 kHz
6-dB Point       : 8.7553 kHz
Stopband Edge    : 9.64 kHz
Passband Ripple  : 0.01 dB
Stopband Atten.  : 79.9981 dB
Transition Width : 1.64 kHz
```



Minimum-Order Designs with dsp.LowpassFilter

You can use `dsp.LowpassFilter` to design minimum-order filters and use `measure` to verify that the design meets the prescribed specifications. The order of the filter is again 100.

```
LP_FIR_minOrd = dsp.LowpassFilter('SampleRate',Fs,...
    'DesignForMinimumOrder',true,'PassbandFrequency',Fp,...
    'StopbandFrequency',Fst,'PassbandRipple',Ap,'StopbandAttenuation',Ast);
measure(LP_FIR_minOrd)
Nlp = order(LP_FIR_minOrd)
```

```
ans =

Sample Rate      : 48 kHz
```

```
Passband Edge    : 8 kHz
3-dB Point       : 8.7136 kHz
6-dB Point       : 8.922 kHz
Stopband Edge    : 10 kHz
Passband Ripple  : 0.0098641 dB
Stopband Atten.  : 80.122 dB
Transition Width : 2 kHz


Nlp =

   100
```

Designing IIR Filters

Elliptic filters are the IIR counterpart to optimal equiripple FIR filters. Accordingly, you can use the same specifications to design elliptic filters. The filter order you obtain for an IIR filter is much smaller than the order of the corresponding FIR filter.

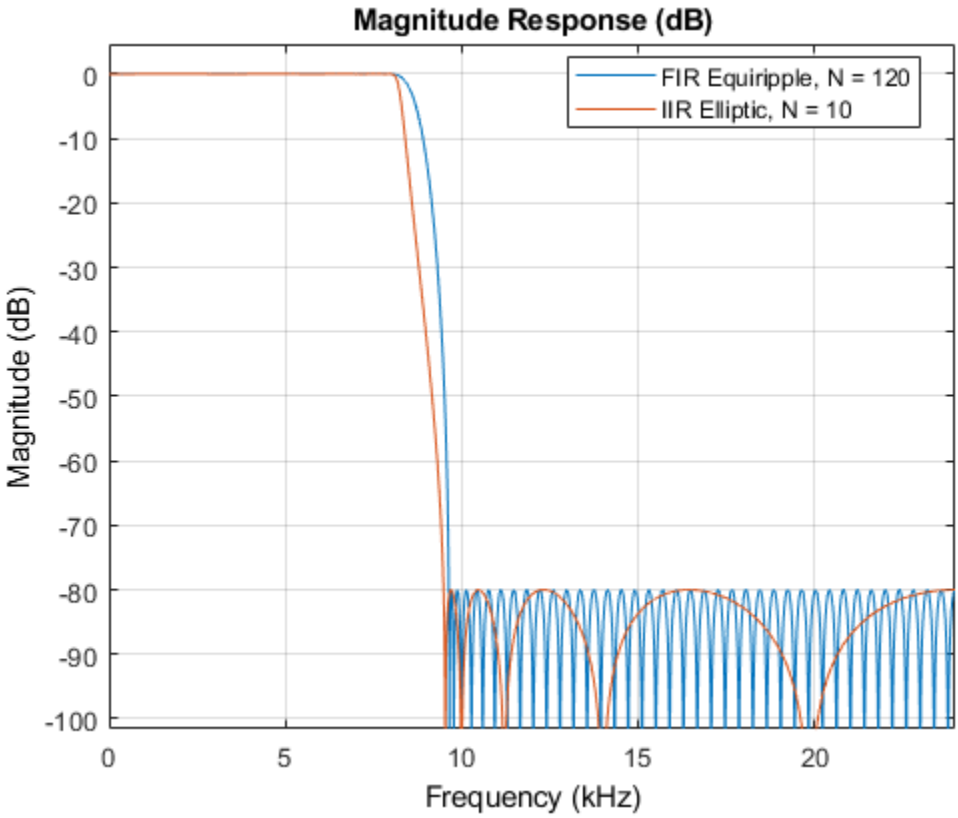
Design an elliptic filter with the same sampling frequency, cutoff frequency, passband-ripple constraint, and stopband attenuation as the 120th-order FIR filter. Reduce the filter order for the elliptic filter to 10.

```
N = 10;
LP_IIR = dsp.LowpassFilter('SampleRate',Fs,'FilterType','IIR',...
    'DesignForMinimumOrder',false,'FilterOrder',N,...
    'PassbandFrequency',Fp,'PassbandRipple',Ap,'StopbandAttenuation',Ast);
```

Compare the FIR and IIR designs. Compute the cost of the two implementations.

```
hfvt = fvtool(LP_FIR,LP_IIR,'Fs',Fs);
legend(hfvt,'FIR Equiripple, N = 120', 'IIR Elliptic, N = 10');
cost_FIR = cost(LP_FIR)
cost_IIR = cost(LP_IIR)
```

```
cost_FIR =

  struct with fields:

                 NumCoefficients: 121
                      NumStates: 120
    MultiplicationsPerInputSample: 121
          AdditionsPerInputSample: 120


cost_IIR =

  struct with fields:

                 NumCoefficients: 25
                      NumStates: 20
    MultiplicationsPerInputSample: 25
          AdditionsPerInputSample: 20
```



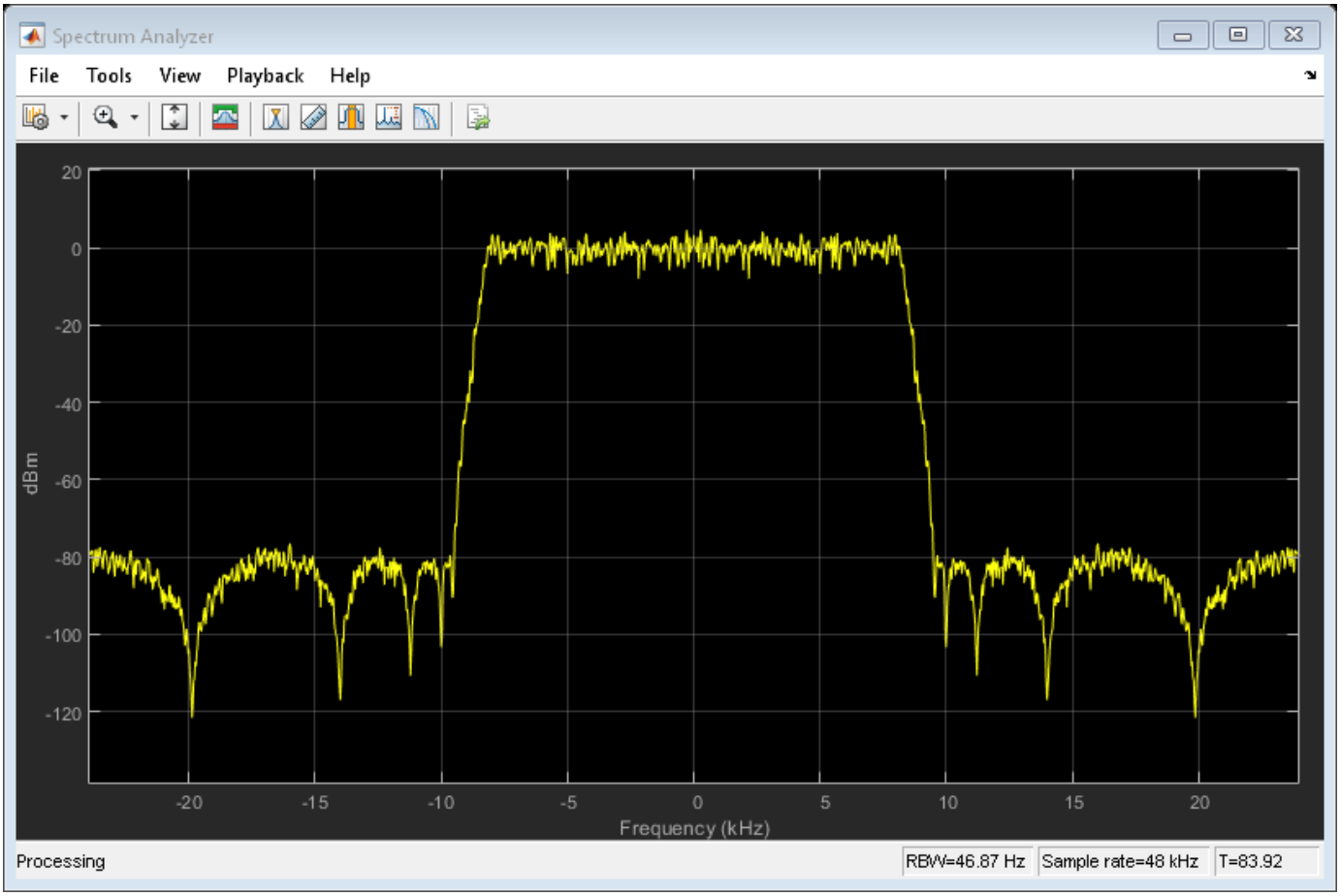
The FIR and IIR filters have similar magnitude responses. The cost of the IIR filter is about 1/6 the cost of the FIR filter.

Running the IIR Filters

The IIR filter is designed as a biquad filter. To apply the filter to data, use the same commands as in the FIR case.

Filter 10 seconds of white Gaussian noise with zero mean and unit standard deviation in frames of 256 samples with the 10th-order IIR lowpass filter. View the result on a spectrum analyzer.

```
SA = dsp.SpectrumAnalyzer('SampleRate',Fs,'SpectralAverages',5);
tic
while toc < 10
    x = randn(256,1);
    y = LP_IIR(x);
    SA(y);
end
```



Variable Bandwidth FIR and IIR Filters

You can also design filters that allow you to change the cutoff frequency at run-time. `dsp.VariableBandwidthFIRFilter` and `dsp.VariableBandwidthIIRFilter` can be used for such cases.

See Also

Related Examples

- “Filter Frames of a Noisy Sine Wave Signal in MATLAB” on page 1-6
- “Lowpass IIR Filter Design in Simulink” on page 1-20
- “Tunable Lowpass Filtering of Noisy Input in Simulink” on page 1-45
- “Design Multirate Filters” on page 1-36

Lowpass IIR Filter Design in Simulink

In this section...
“filterBuilder” on page 1-20
“Butterworth Filter” on page 1-21
“Chebyshev Type I Filter” on page 1-26
“Chebyshev Type II Filter” on page 1-27
“Elliptic Filter” on page 1-29
“Minimum-Order Designs” on page 1-31
“Lowpass Filter Block” on page 1-34
“Variable Bandwidth IIR Filter Block” on page 1-35

This example shows how to design classic lowpass IIR filters in Simulink.

The example first presents filter design using `filterBuilder`. The critical parameter in this design is the cutoff frequency, the frequency at which filter power decays to half (`-3` dB) the nominal passband value. The example shows how to replace a Butterworth design with either a Chebyshev or elliptic filter of the same order and obtain a steeper roll-off at the expense of some ripple in the passband and/or stopband of the filter. The example also explores minimum-order designs.

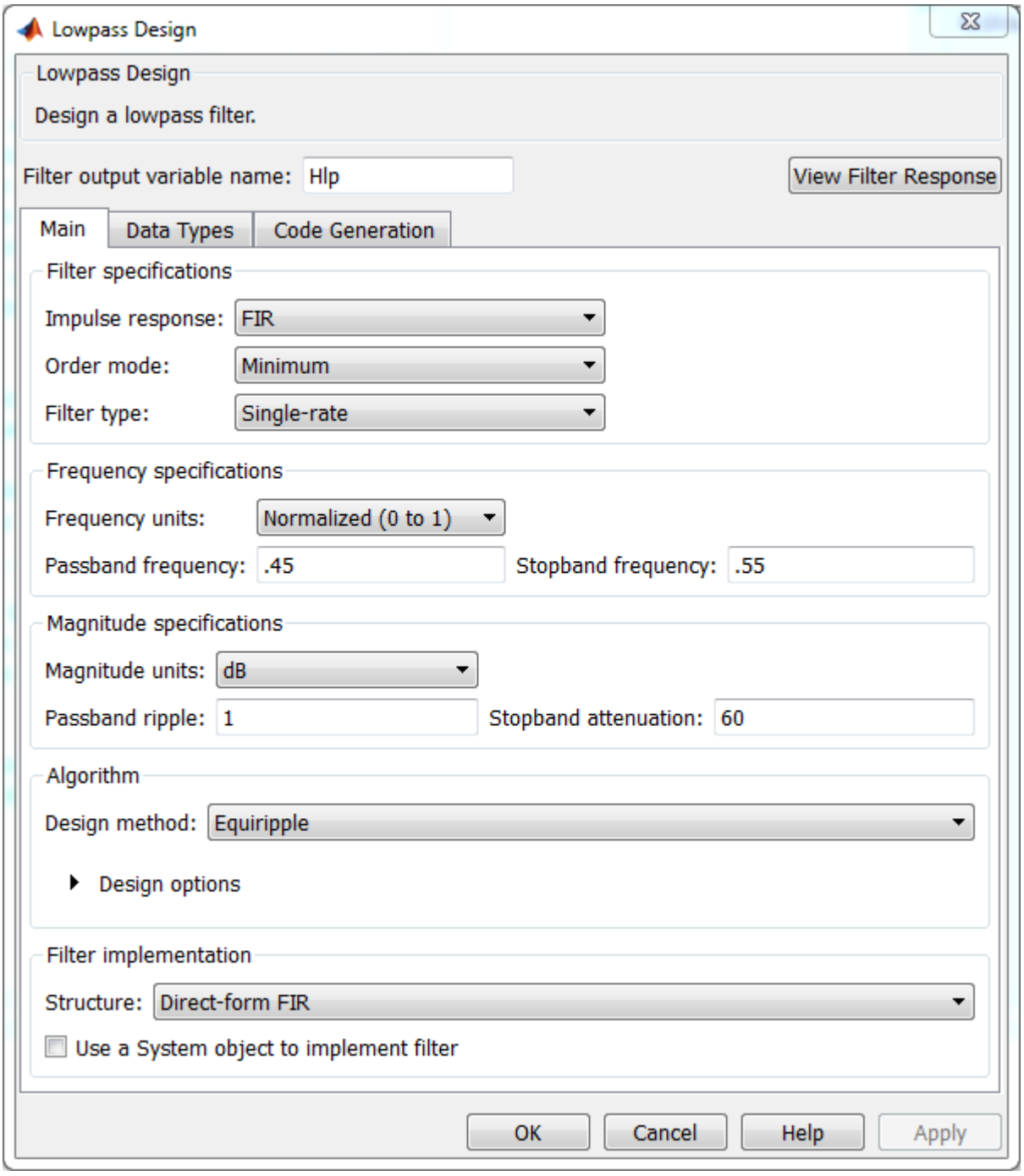
The example then shows how to design and use lowpass filters in Simulink using the interface available from the Lowpass Filter block..

Finally, the example showcases the Variable Bandwidth IIR Filter, which enables you to change the filter cutoff frequency at run time.

filterBuilder

`filterBuilder` starts user interface for building filters. `filterBuilder` uses a specification-centered approach to find the best algorithm for the desired response. It also enables you to create a Simulink block from the specified design.

To start designing IIR lowpass filter blocks using `filterBuilder`, execute the command `filterBuilder('lp')`. A Lowpass Design dialog box opens.

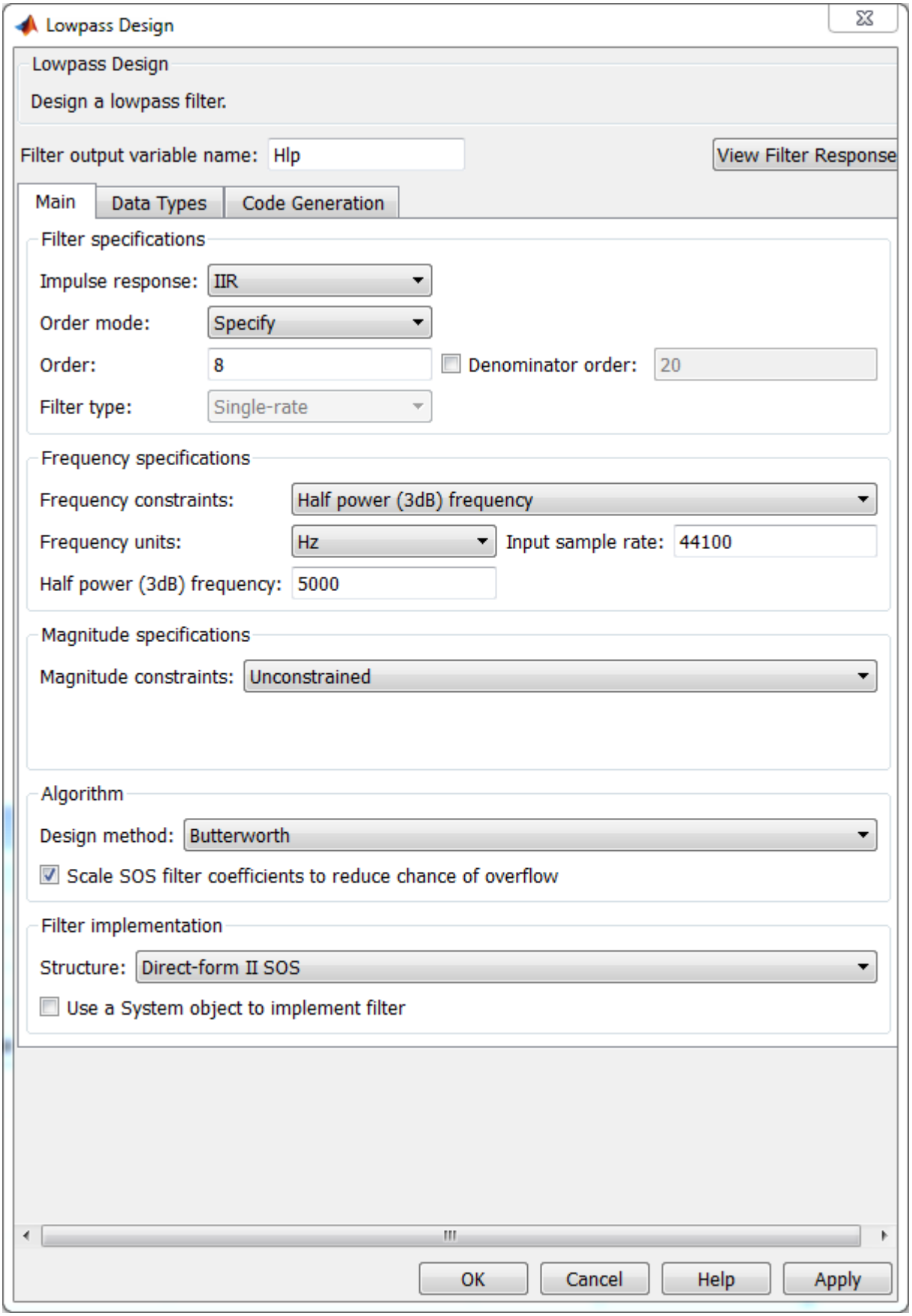


Butterworth Filter

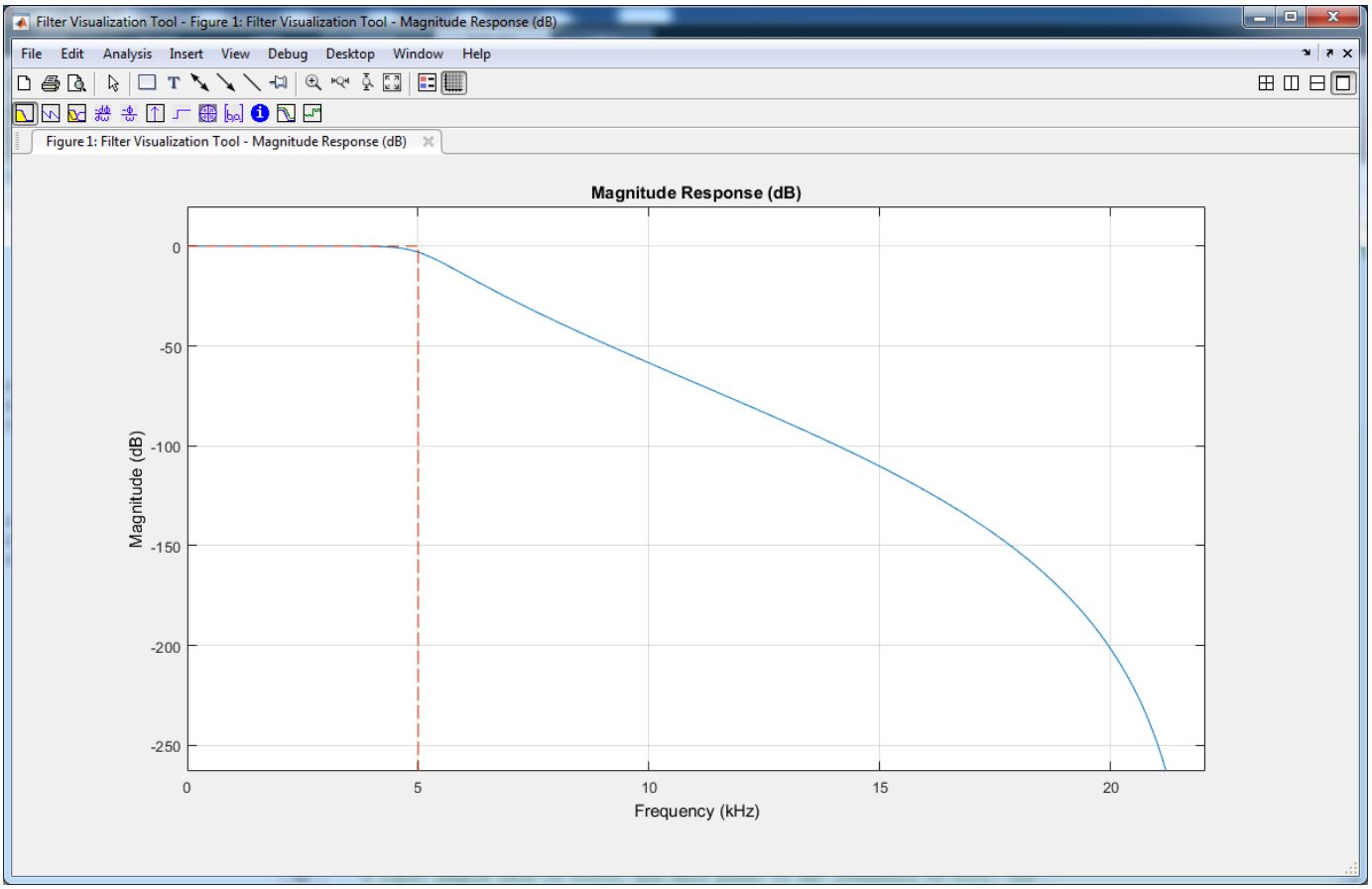
Design an eighth order Butterworth lowpass filter with a cutoff frequency of 5 kHz, assuming a sample rate of `44.1` KHz.

Set the **Impulse response** to `IIR`, the **Order mode** to `Specify`, and the **Order** to 8. To specify the cutoff frequency, set **Frequency constraints** to `Half power (3 dB) frequency`. To specify the

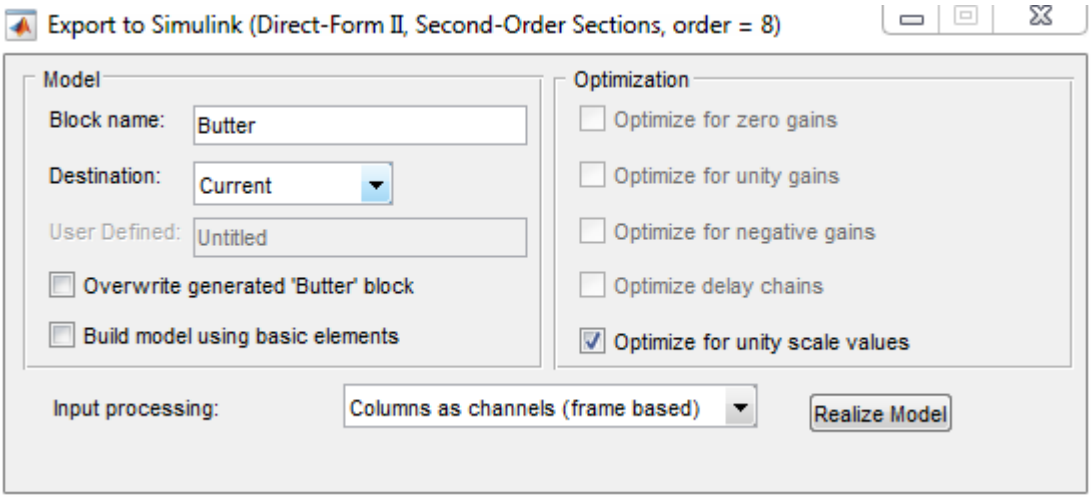
frequencies in Hz, set **Frequency units** to `Hz`, **Input sample rate** to `44100`, and **Half power (3 dB) frequency** to `5000`. Set the **Design method** to `Butterworth`.


Lowpass Design
Lowpass Design
Design a lowpass filter.
Filter output variable name: Hlp
View Filter Response
Main
Data Types
Code Generation
Filter specifications
Impulse response: IIR
Order mode: Specify
Order: 8
Denominator order: 20
Filter type: Single-rate
Frequency specifications
Frequency constraints: Half power (3dB) frequency
Frequency units: Hz
Input sample rate: 44100
Half power (3dB) frequency: 5000
Magnitude specifications
Magnitude constraints: Unconstrained
Algorithm
Design method: Butterworth
Scale SOS filter coefficients to reduce chance of overflow
Filter implementation
Structure: Direct-form II SOS
Use a System object to implement filter
OK
Cancel
Help
Apply

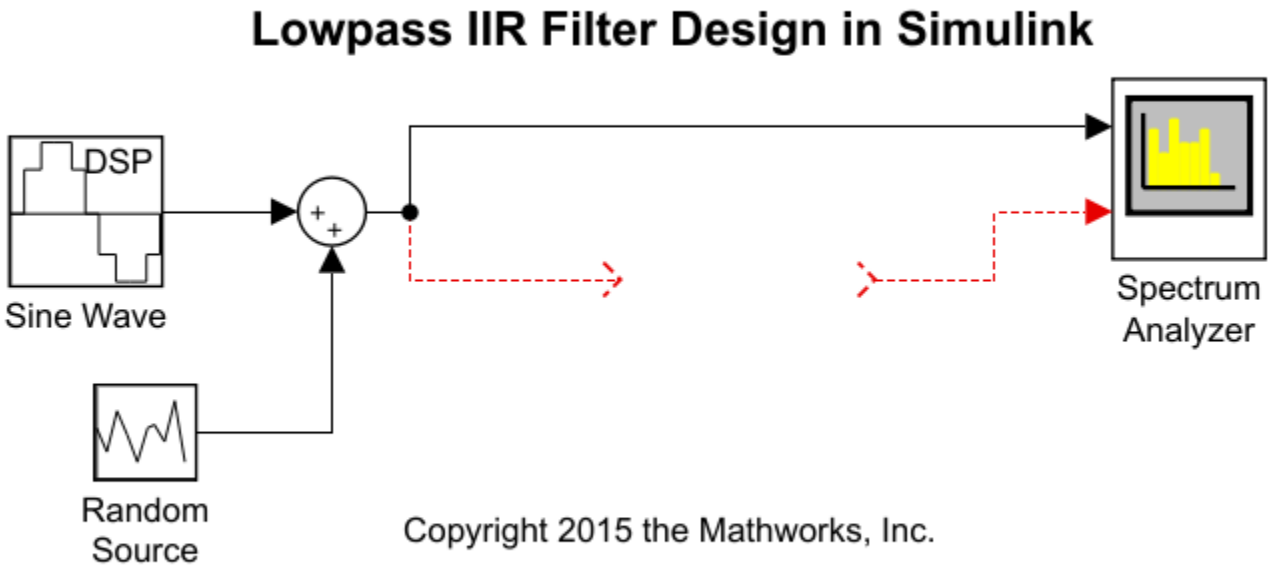
Click **Apply**. To visualize the filter's frequency response, click **View Filter Response**. The filter is maximally flat. There is no ripple in the passband or in the stopband. The filter response is within the specification mask (the red dotted line).



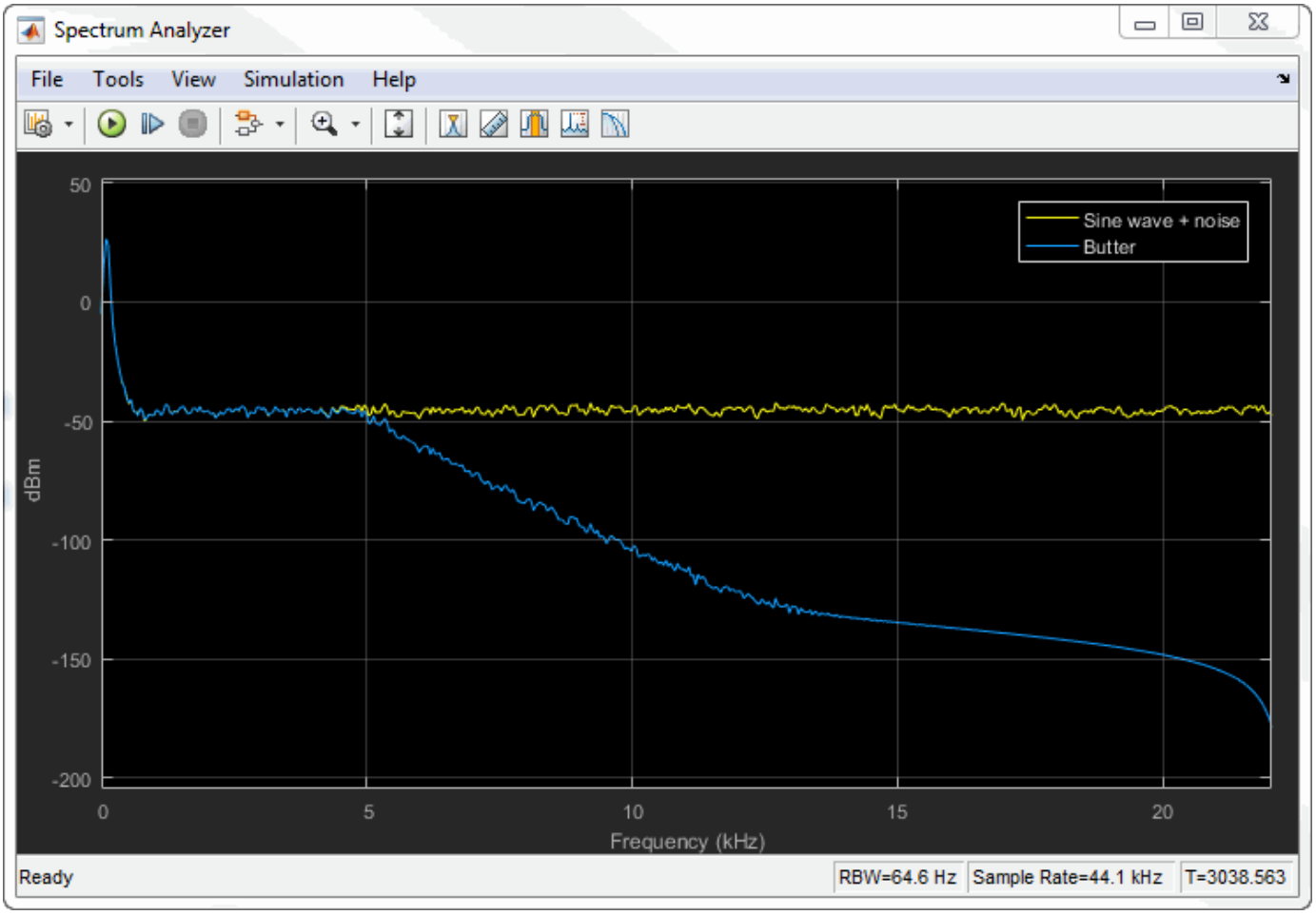
Generate a block from this design and use it in a model. Open the model `ex_iir_design`. In **Filter Builder**, on the **Code Generation** tab, click **Generate Model**. In the Export to Simulink window, specify the **Block name** as `Butter` and **Destination** as `Current`. You can also choose to build the block using basic elements such as delays and gains, or use one of the DSP System Toolbox filter blocks. This example uses the filter block.



Click **Realize model** to generate the Simulink block. You can now connect the block's input and output ports to the source and sink blocks in the `ex_iir_design` model.



In the model, a noisy sine wave sampled at `44.1` kHz passes through the filter. The sine wave is corrupted by Gaussian noise with zero mean and a variance of 10^{-5}. Run the model. The view in the Spectrum Analyzer shows the original and filtered signals.



Chebyshev Type I Filter

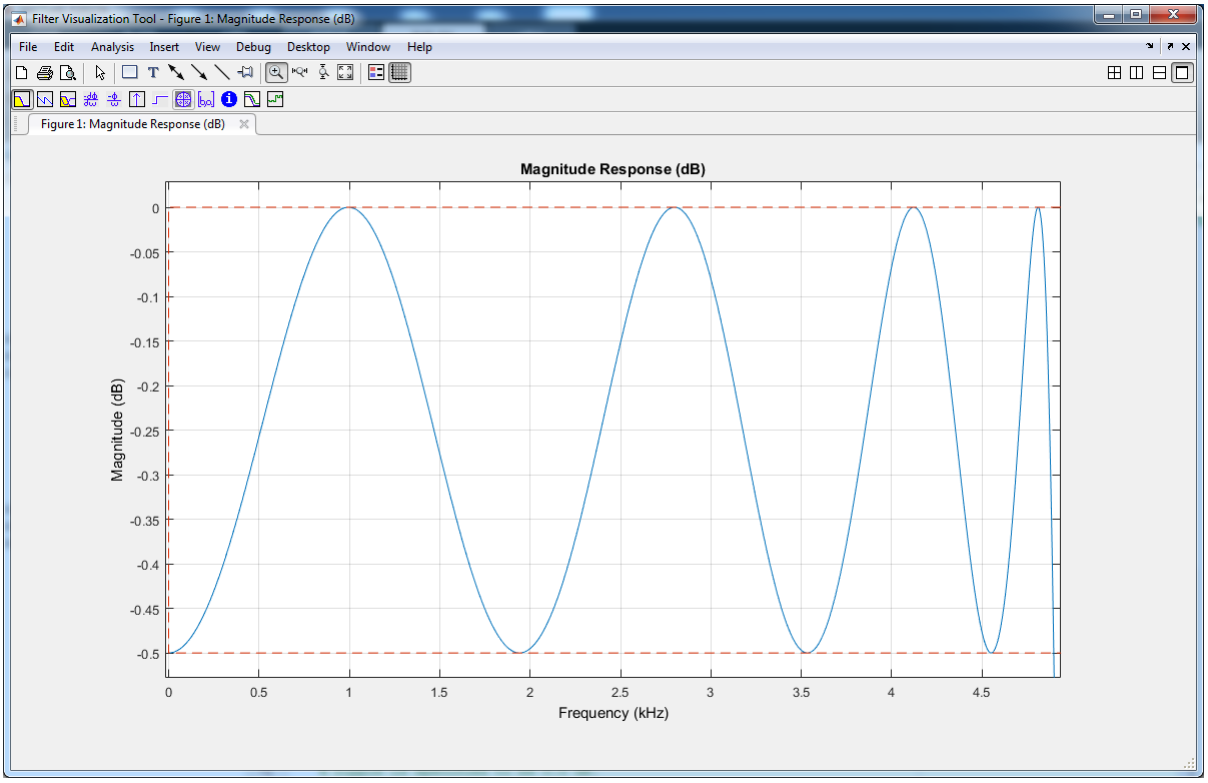
Now design a Chebyshev Type I filter. A Chebyshev type I design allows you to control the passband. There are still no ripples in the stopband. Larger ripples enable a steeper roll-off. In this model, the peak-to-peak ripple is specified as 0.5 dB.

In the **Main** tab of **Filter Builder**, set the

1. **Magnitude Constraints** to `Passband ripple`.
2. **Passband ripple** to `0.5`.
3. **Design method** to `Chebyshev type I`.

Click **Apply** and then click **View Filter Response**.

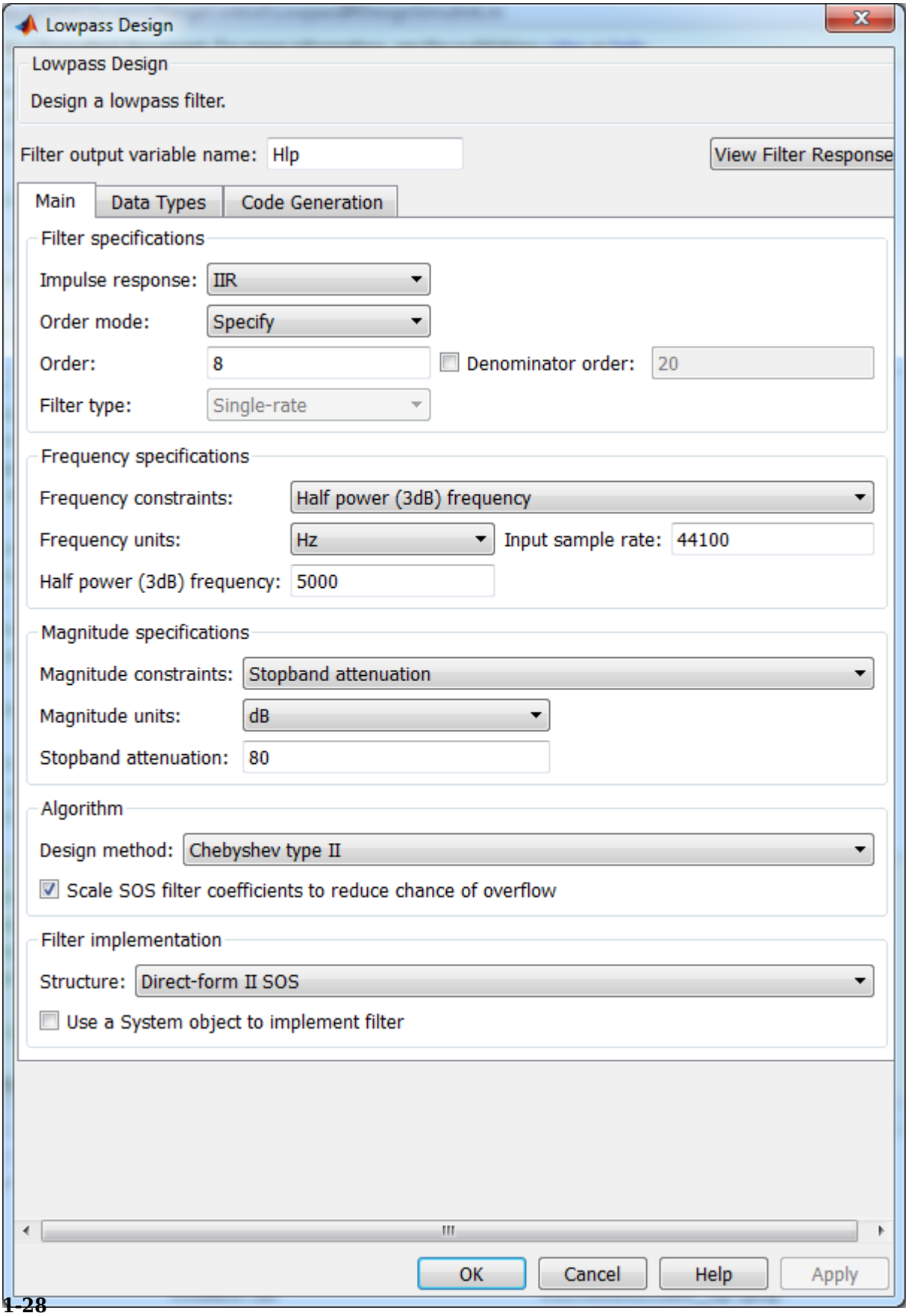
Zooming in on the passband, you can see that the ripples are contained in the range [-0.5, 0] dB.



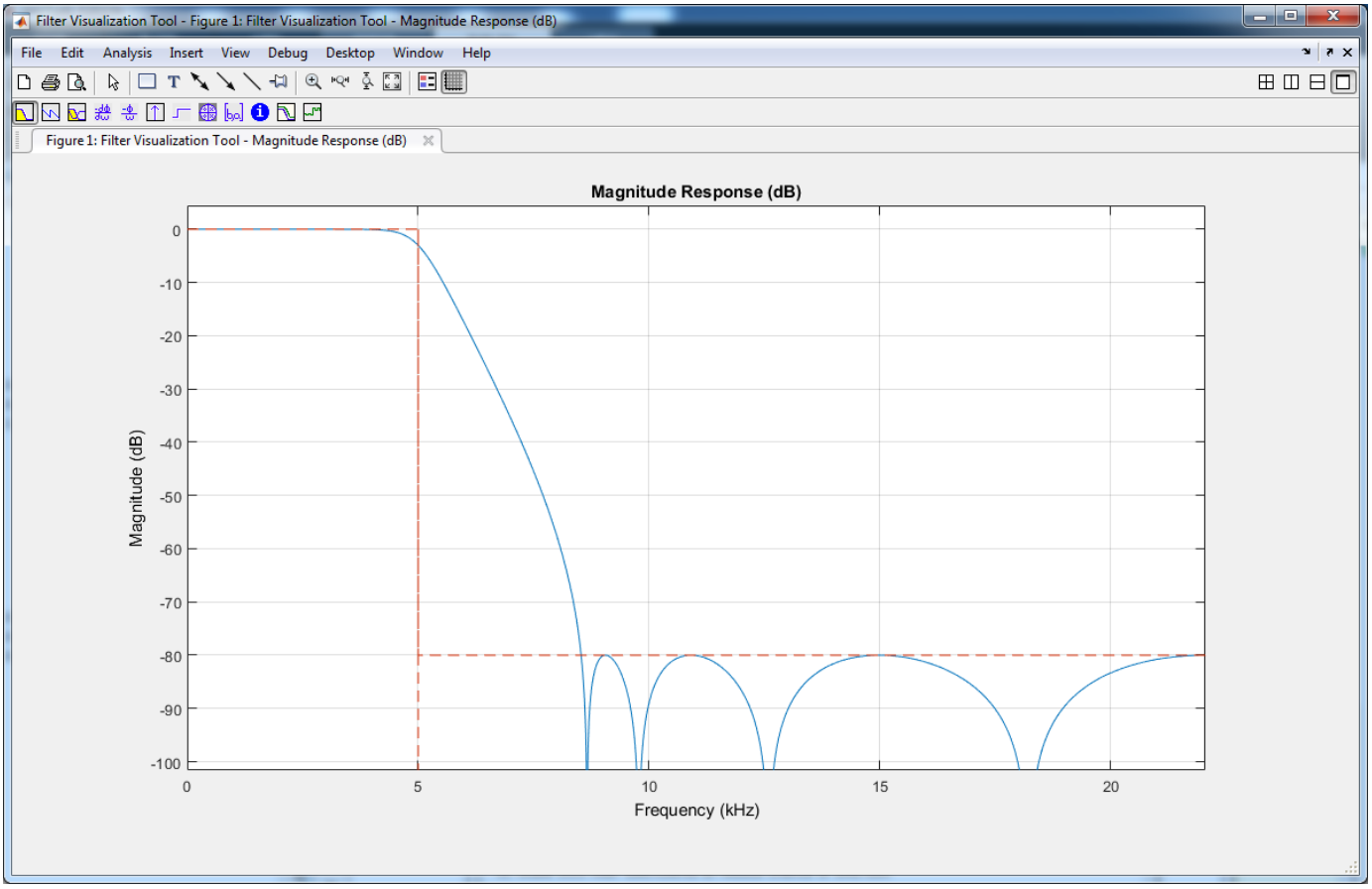
Similar to the Butterworth filter, you can generate a block from this design by clicking **Generate Model** on the **Code Generation** tab, and then clicking **Realize model**.

Chebyshev Type II Filter

A Chebyshev type II design allows you to control the stopband attenuation. There are no ripples in the passband. A smaller stopband attenuation enables a steeper roll-off. In this example, the stopband attenuation is `80` dB. Set the **Filter Builder Main** tab as shown, and click **Apply**.


Lowpass Design
Lowpass Design
Design a lowpass filter.
Filter output variable name: Hlp
View Filter Response
Main
Data Types
Code Generation
Filter specifications
Impulse response: IIR
Order mode: Specify
Order: 8
Denominator order: 20
Filter type: Single-rate
Frequency specifications
Frequency constraints: Half power (3dB) frequency
Frequency units: Hz
Input sample rate: 44100
Half power (3dB) frequency: 5000
Magnitude specifications
Magnitude constraints: Stopband attenuation
Magnitude units: dB
Stopband attenuation: 80
Algorithm
Design method: Chebyshev type II
Scale SOS filter coefficients to reduce chance of overflow
Filter implementation
Structure: Direct-form II SOS
Use a System object to implement filter
OK
Cancel
Help
Apply

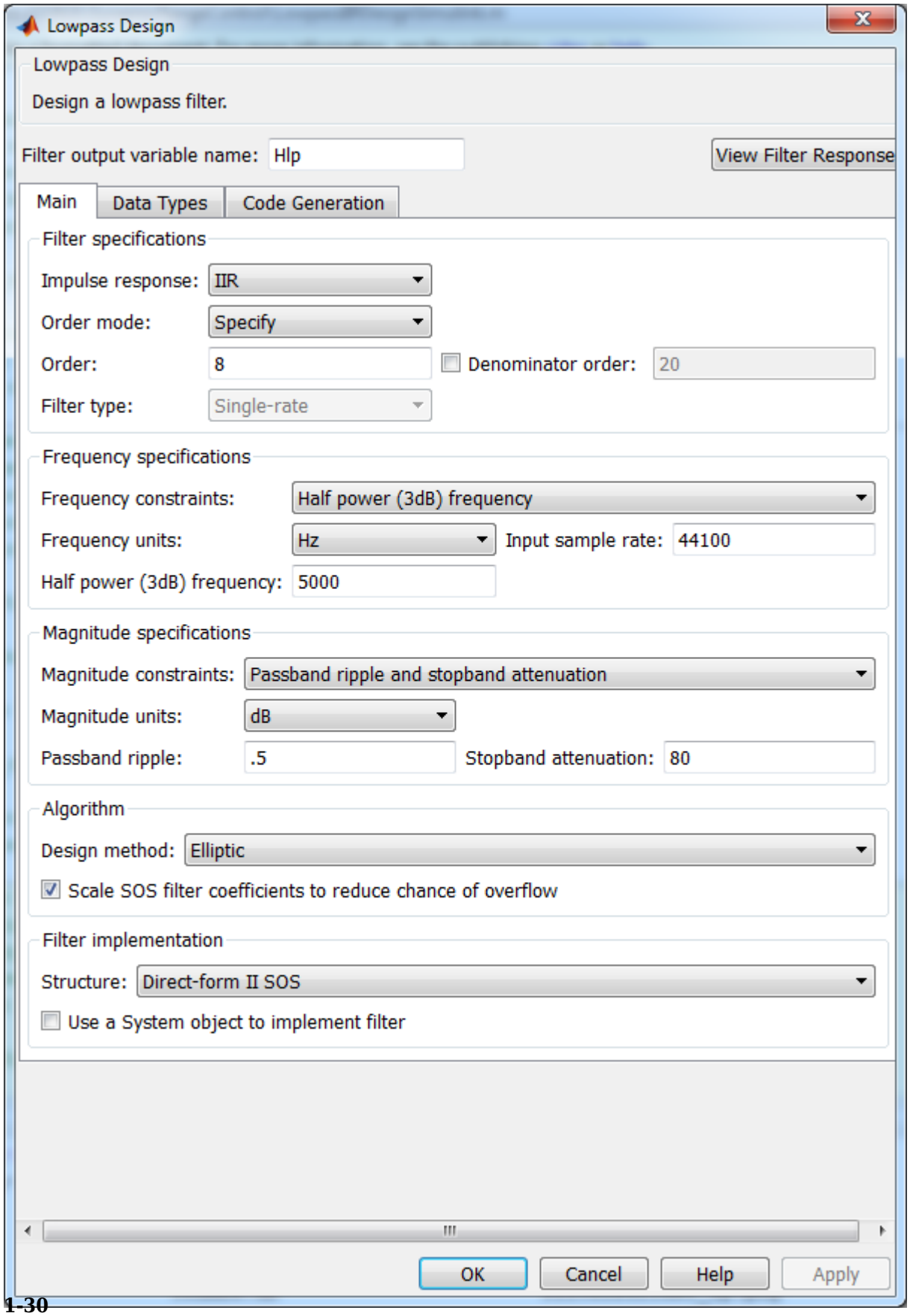
Click **View Filter Response**.

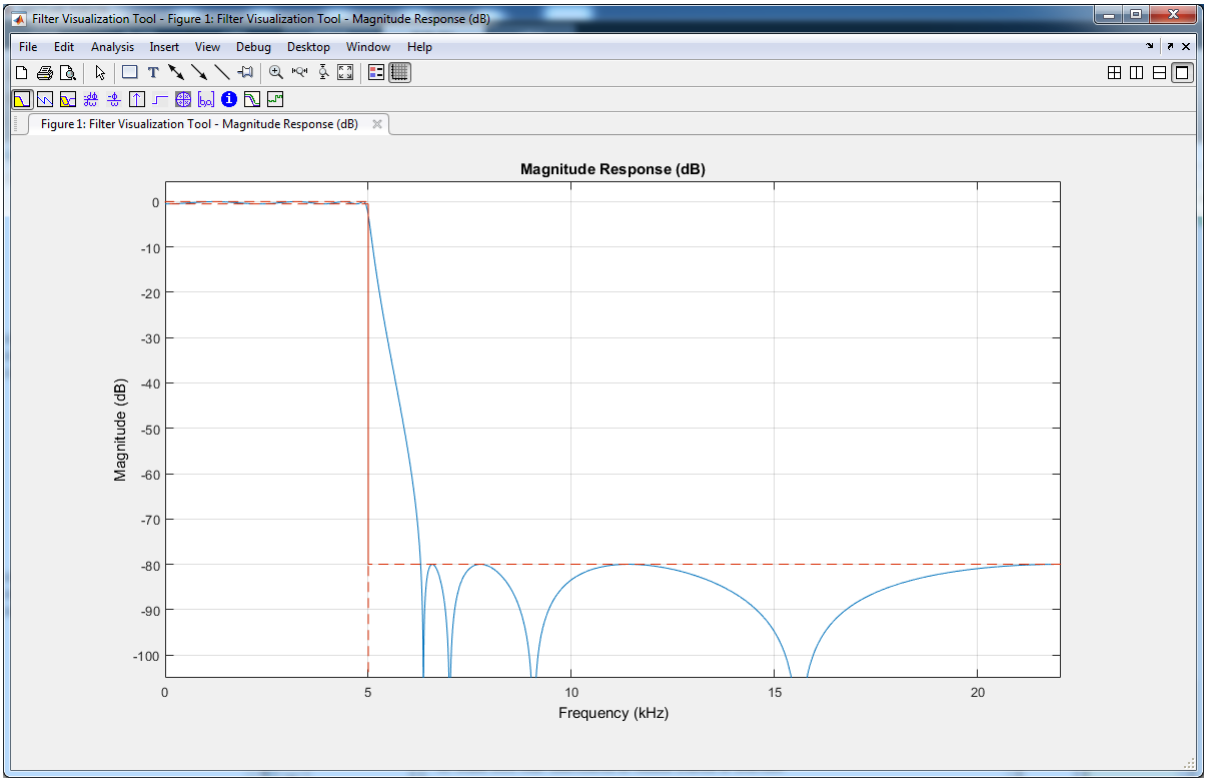


To generate a block from this design, on the **Code Generation** tab, click **Generate Model**, and then click **Realize model**.

Elliptic Filter

An elliptic filter can provide steeper roll-off compared to previous designs by allowing ripples in both the stopband and passband. To illustrate this behavior, use the same passband and stopband characteristics specified in the Chebyshev designs. Set the **Filter Builder Main** tab as shown, and click **Apply**.

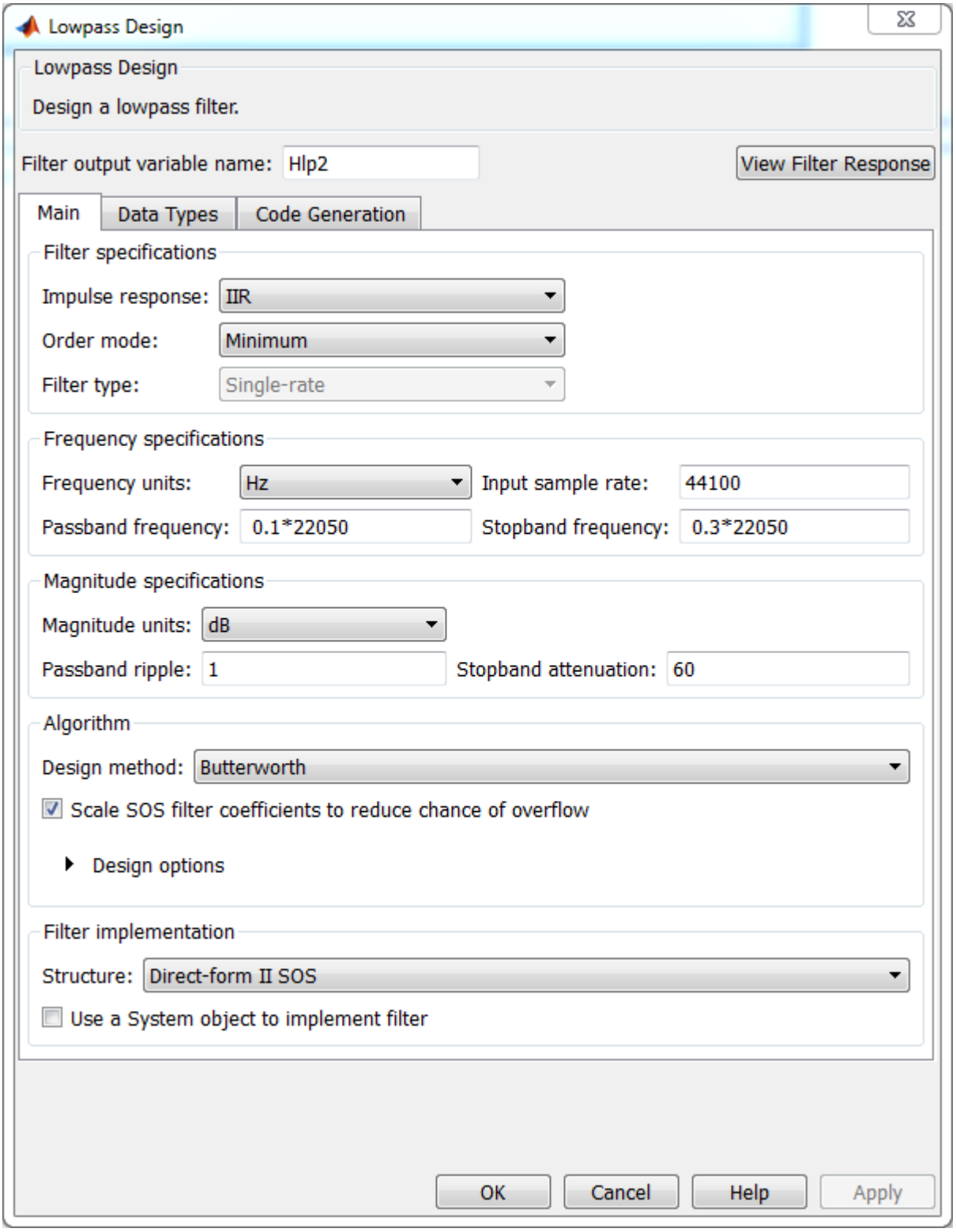

Lowpass Design
Lowpass Design
Design a lowpass filter.
Filter output variable name: Hlp
View Filter Response
Main
Data Types
Code Generation
Filter specifications
Impulse response: IIR
Order mode: Specify
Order: 8
Denominator order: 20
Filter type: Single-rate
Frequency specifications
Frequency constraints: Half power (3dB) frequency
Frequency units: Hz
Input sample rate: 44100
Half power (3dB) frequency: 5000
Magnitude specifications
Magnitude constraints: Passband ripple and stopband attenuation
Magnitude units: dB
Passband ripple: .5
Stopband attenuation: 80
Algorithm
Design method: Elliptic
Scale SOS filter coefficients to reduce chance of overflow
Filter implementation
Structure: Direct-form II SOS
Use a System object to implement filter
OK
Cancel
Help
Apply



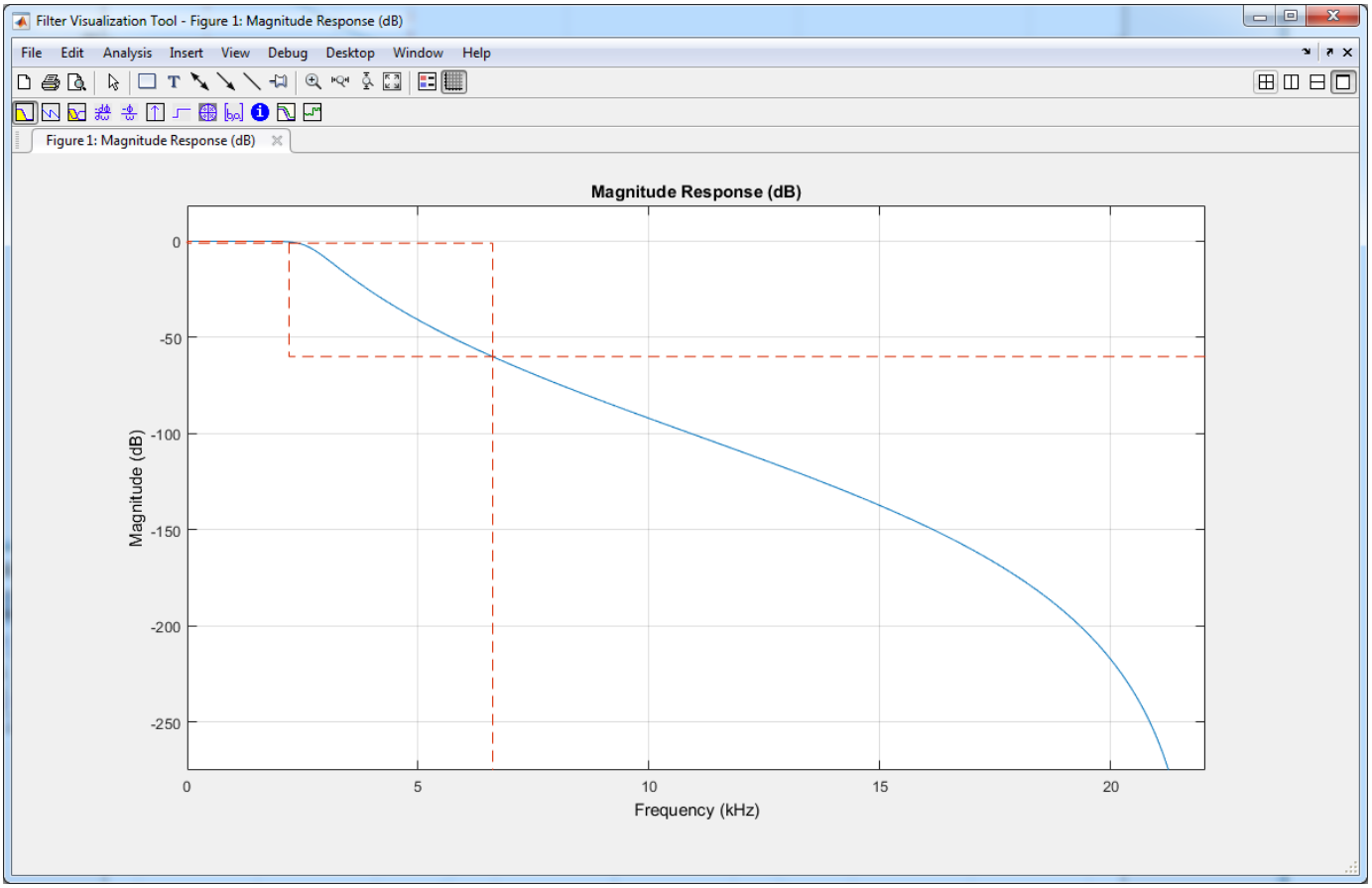
To generate a block from this design, on the **Code Generation** tab, click **Generate Model**, and then click **Realize model**.

Minimum-Order Designs

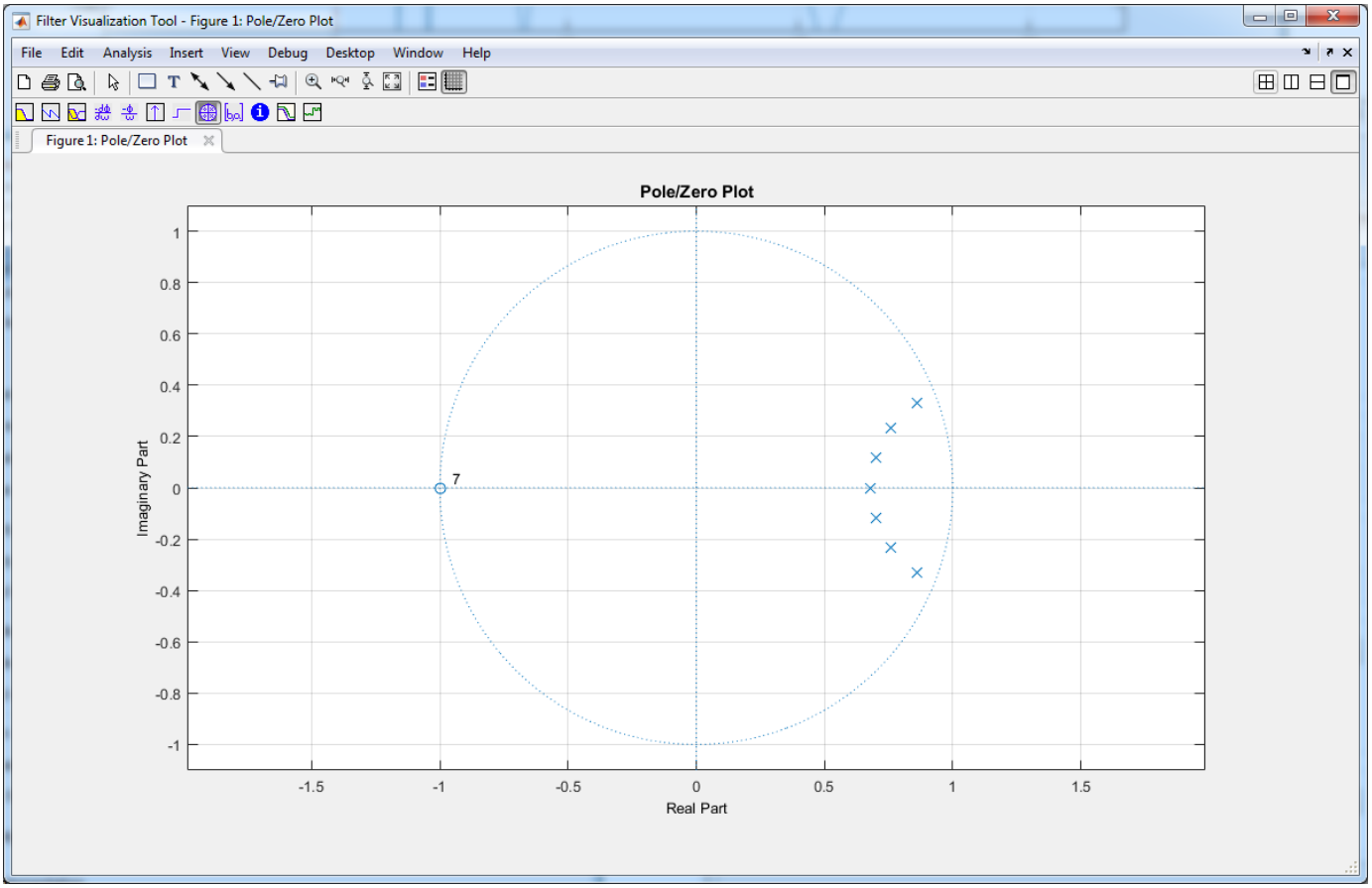
To specify the passband and stopband in terms of frequencies and the amount of tolerable ripple, use a minimum-order design. As an example, verify that the **Order mode** of the Butterworth filter is set to `Minimum`, and set **Design method** to `Butterworth`. Set the passband and stopband frequencies to `0.1*22050` Hz and `0.3*22050` Hz, and the passband ripple and the stopband attenuation to `1` dB and `60` dB, respectively. A seventh-order filter is necessary to meet the specifications with a Butterworth design. By following the same approach for other design methods, you can verify that a fifth-order filter is required for Chebyshev type I and type II designs. A fourth-order filter is sufficient for the elliptic design.



This figure shows the magnitude response for the seventh-order Butterworth design.



The pole-zero plot for the seventh-order Butterworth design shows the expected clustering of 7 poles around an angle of zero radians on the unit circle and the corresponding 7 zeros at an angle of π radians.

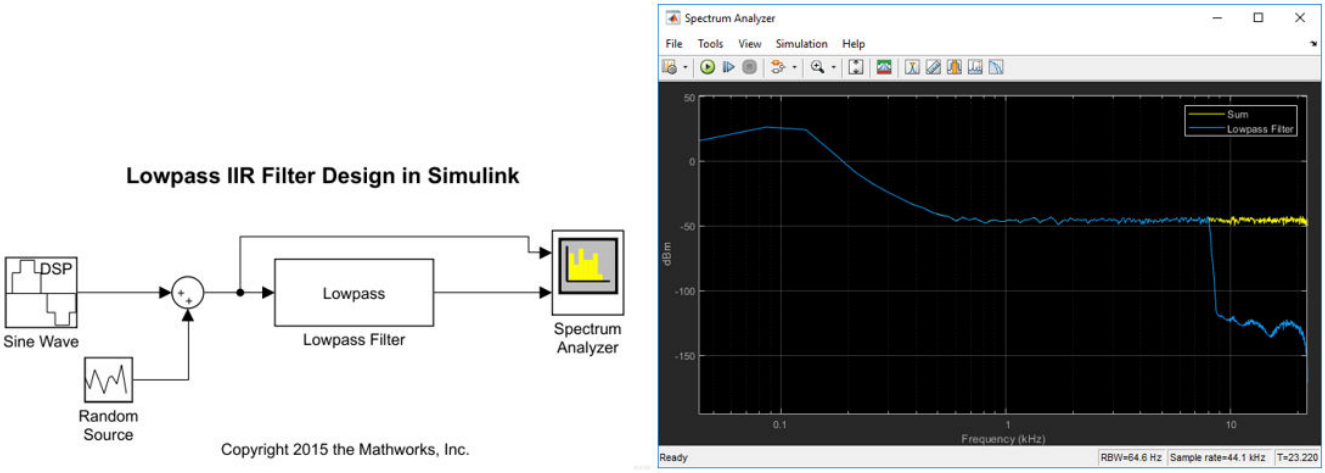


Lowpass Filter Block

As an alternative to **Filter Builder**, you can use the Lowpass Filter block in your Simulink model. The Lowpass Filter block combines the design and implementation stages into one step. The filter designs its coefficients using the elliptical method, and allows minimum order and custom order designs.

The Lowpass Filter block is used in the model `ex_lowpass` to filter a noisy sine wave signal sampled at `44.1` kHz. The original and filtered signals are displayed in a spectrum analyzer.

```
model = 'ex_lowpass';
open_system(model);
set_param(model,'StopTime','1024/44100 * 1000')
sim(model);
```



The Lowpass Filter block allows you to design filters that approximate arbitrarily close to Butterworth and Chebyshev filters. To approximate a Chebyshev Type I filter, make the stopband attenuation arbitrarily large, for example, `180` dB. To approximate a Chebyshev Type II filter, make the passband ripple arbitrarily small, for example, `1e-4`. To approximate a Butterworth filter, make the stopband attenuation arbitrarily large and the passband ripple arbitrarily small.

Variable Bandwidth IIR Filter Block

You can also design filters that allow you to change the cutoff frequency at run time. The Variable Bandwidth IIR Filter block can be used for such cases. Refer to the “Tunable Lowpass Filtering of Noisy Input in Simulink” on page 1-45 example for a model that uses this block.

See Also

Related Examples

- “Tunable Lowpass Filtering of Noisy Input in Simulink” on page 1-45
- “Lowpass Filter Design in MATLAB” on page 1-12
- “Design Multirate Filters” on page 1-36

Design Multirate Filters

Note If you are using R2016a or an earlier release, replace each call to the object with the equivalent step syntax. For example, `obj(x)` becomes `step(obj,x)`.

Multirate filters are filters in which different parts of the filter operate at different rates. Such filters are commonly used when the input and output sample rates differ, such as during decimation, interpolation, or a combination of both. However, multirate filters are often used in designs where the input sample rate and output sample rate are the same. In such filters, there is an internal decimation and interpolation occurring in a series of filters. Such filters can achieve both greatly reduced filter lengths and computational rates as compared to standard single-rate filter designs.

The most basic multirate filters are interpolators, decimators, and rate converters. These filters are building components of more advanced filter technologies such as filter banks and Quadrature Mirror Filter (QMF). You can design these filters in MATLAB and Simulink using the `designMultirateFIR` function.

The function uses the FIR Nyquist filter design algorithm to compute the filter coefficients. To implement these filters in MATLAB, use these coefficients as inputs to the `dsp.FIRDecimator`, `dsp.FIRInterpolator`, and `dsp.FIRRateConverter` System objects. In Simulink, compute these coefficients using `designMultirateFIR` in the default **Auto** mode of the FIR Decimation, FIR Interpolation, and FIR Rate Conversion blocks.

The inputs to the `designMultirateFIR` function are the interpolation factor and the decimation factor. Optionally, you can provide the half-polyphase length and stopband attenuation. The interpolation factor of the decimator is set to 1. Similarly, the decimation factor of the interpolator is set to 1.

These examples show how to implement an FIR decimator in MATLAB and Simulink. The same workflow can apply to an FIR interpolator and FIR rate converter as well.

Implement an FIR Decimator in MATLAB

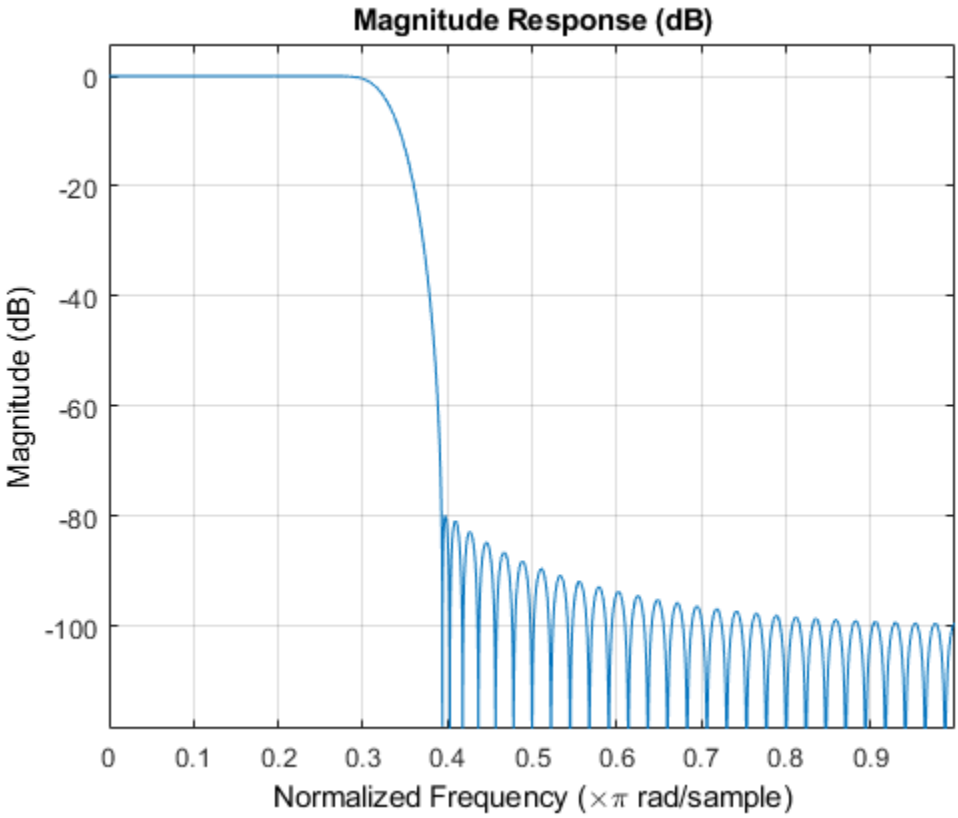
To implement an FIR Decimator, you must first design it by using the `designMultirateFIR` function. Specify the decimation factor of interest (usually greater than 1) and an interpolation factor equal to 1. You can use the default half-polyphase length of 12 and the default stopband attenuation of 80 dB. Alternatively, you can also specify the half-polyphase length and stopband attenuation values.

Design an FIR decimator with the decimation factor set to 3 and the half-polyphase length set to 14. Use the default stopband attenuation of 80 dB.

```
b = designMultirateFIR(1,3,14);
```

Provide the coefficients vector, `b`, as an input to the `dsp.FIRDecimator` System object?.

```
FIRDecim = dsp.FIRDecimator(3,b);
fvtool(FIRDecim);
```



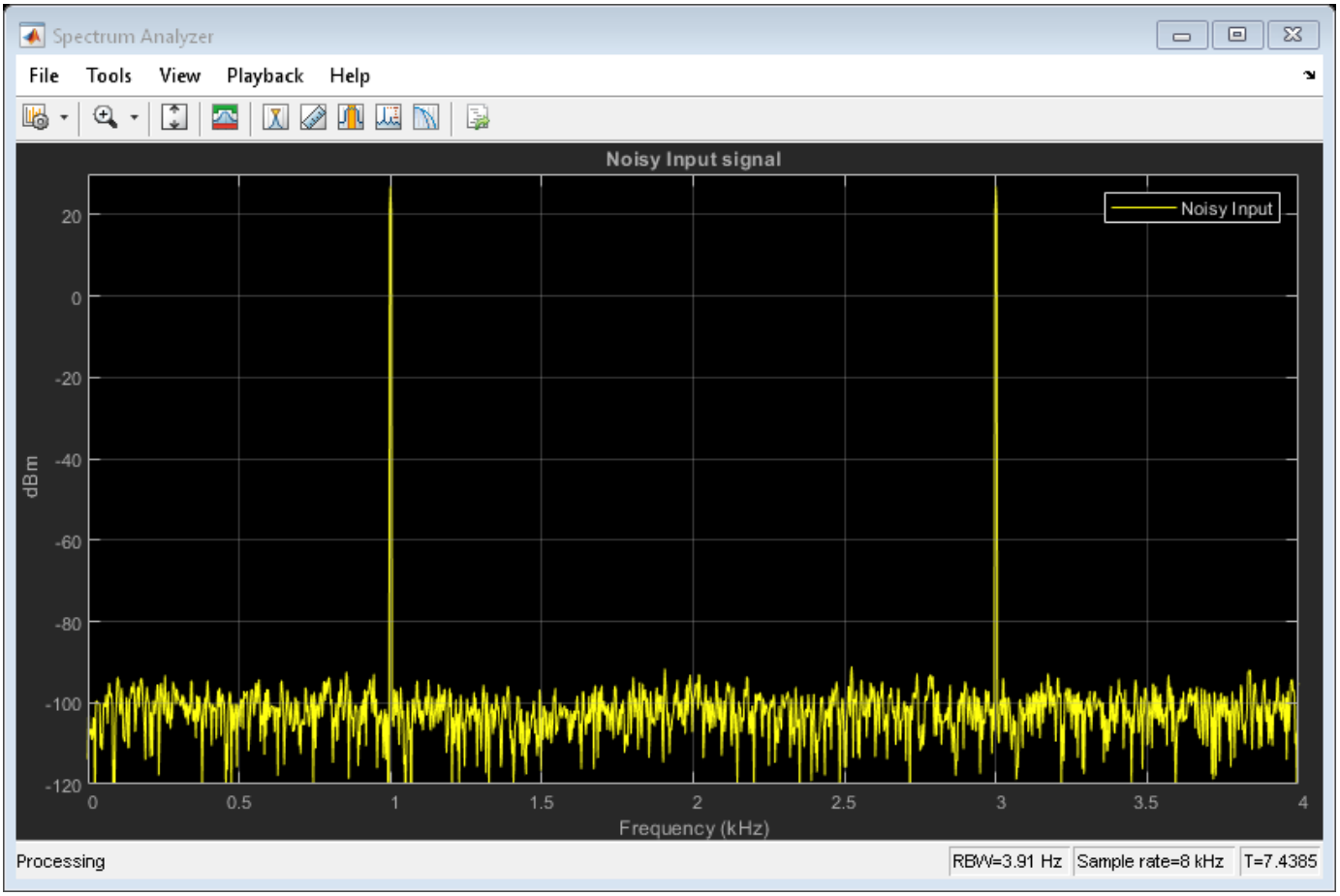
By default, the `fvtool` shows the magnitude response. Navigate through the `fvtool` toolbar to see the phase response, impulse response, group delay, and other filter analysis information.

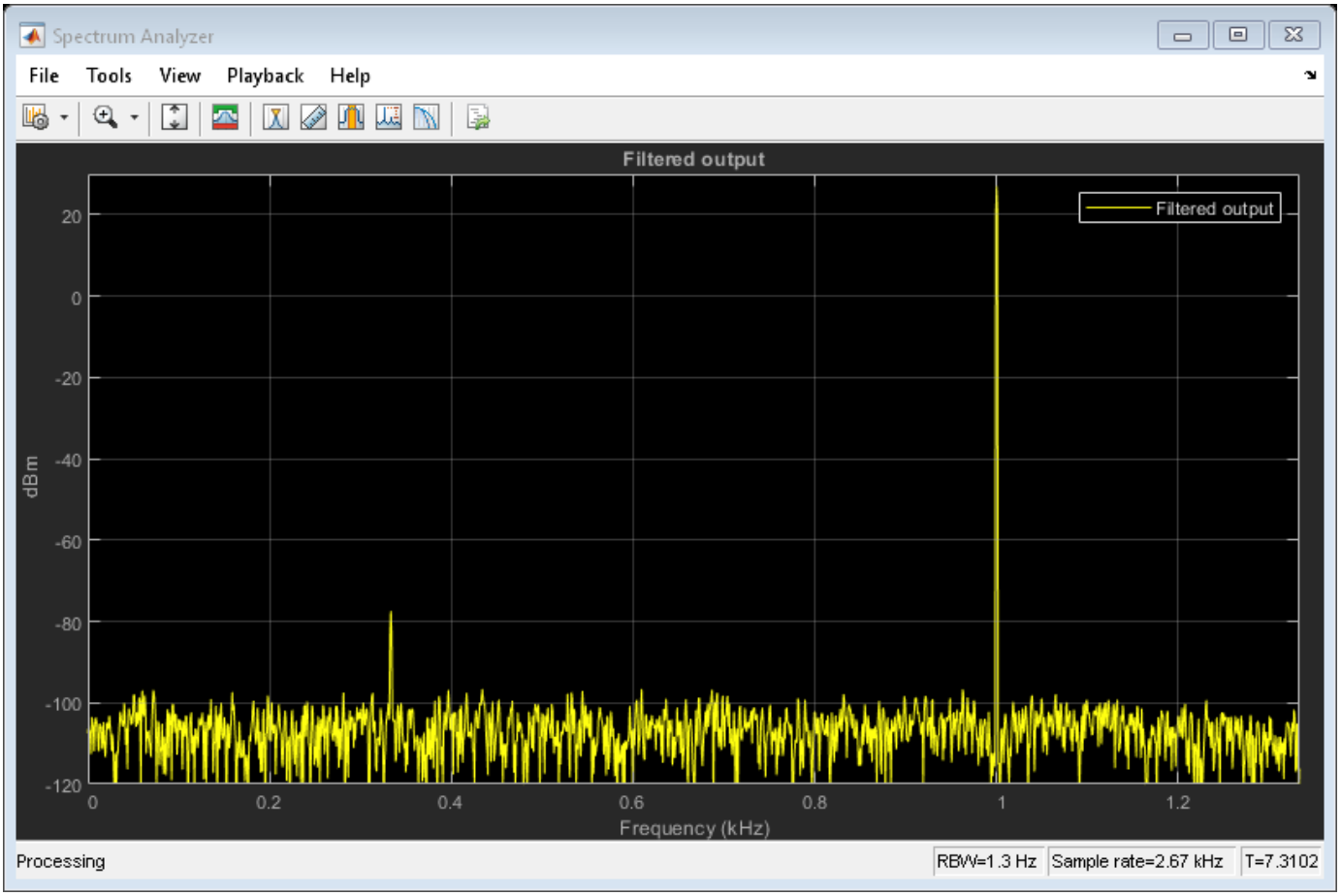
Filter a noisy sine wave input using the `FIRDecim` object. The sine wave has frequencies at 1000 Hz and 3000 Hz 3000 Hz. The noise is a white Gaussian noise with mean zero and standard deviation 1e-5. The decimated output will have one-third the sample rate as input. Initialize two `dsp.SpectrumAnalyzer` System objects, one for the input and the other for the output.

```
f1 = 1000;
f2 = 3000;
Fs = 8000;
source = dsp.SineWave('Frequency',[f1,f2],'SampleRate',Fs,...
    'SamplesPerFrame',1026);

specanainput = dsp.SpectrumAnalyzer('SampleRate',Fs,...
    'PlotAsTwoSidedSpectrum',false,...
    'ShowLegend',true,'YLimits',[-120 30],...
    'Title','Noisy Input signal',...
    'ChannelNames', {'Noisy Input'});
specanaoutput = dsp.SpectrumAnalyzer('SampleRate',Fs/3,...
    'PlotAsTwoSidedSpectrum',false,...
    'ShowLegend',true,'YLimits',[-120 30],...
    'Title','Filtered output',...
    'ChannelNames', {'Filtered output'});
```

Stream in the input and filter the signal in a processing loop.

```
for Iter = 1:100
    input = sum(source(),2);
    noisyInput = input + (10^-5)*randn(1026,1);
    output = FIRDecim(noisyInput);
    specanainput(noisyInput)
    specanaoutput(output)
end
```



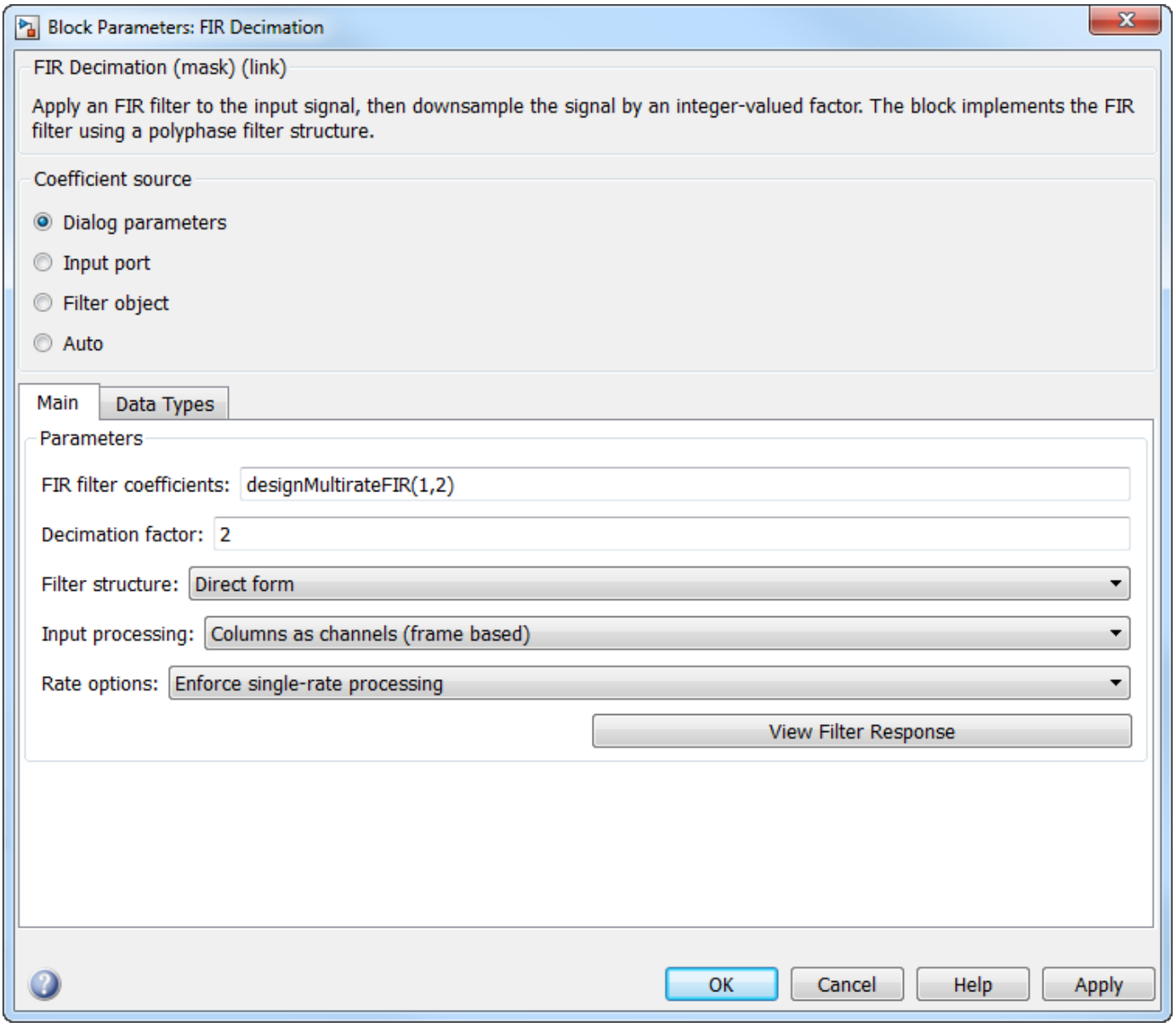


The input has two peaks: one at 1000 Hz and the other at 3000 Hz. The filter has a lowpass response with a passband frequency of 0.3*pi rad/sample. With a sampling frequency of 8000 Hz, that is a passband frequency of 1200 Hz. The tone at 1000 Hz is unattenuated, because it falls in the passband of the filter. The tone at 3000 Hz is filtered out.

Similarly, you can design an FIR interpolator and FIR rate Converter by providing appropriate inputs to the `designMultirateFIR` function. To implement the filters, pass the designed coefficients to the `dsp.FIRInterpolator` and `dsp.FIRRateConverter` objects.

Implement an FIR Decimator in Simulink

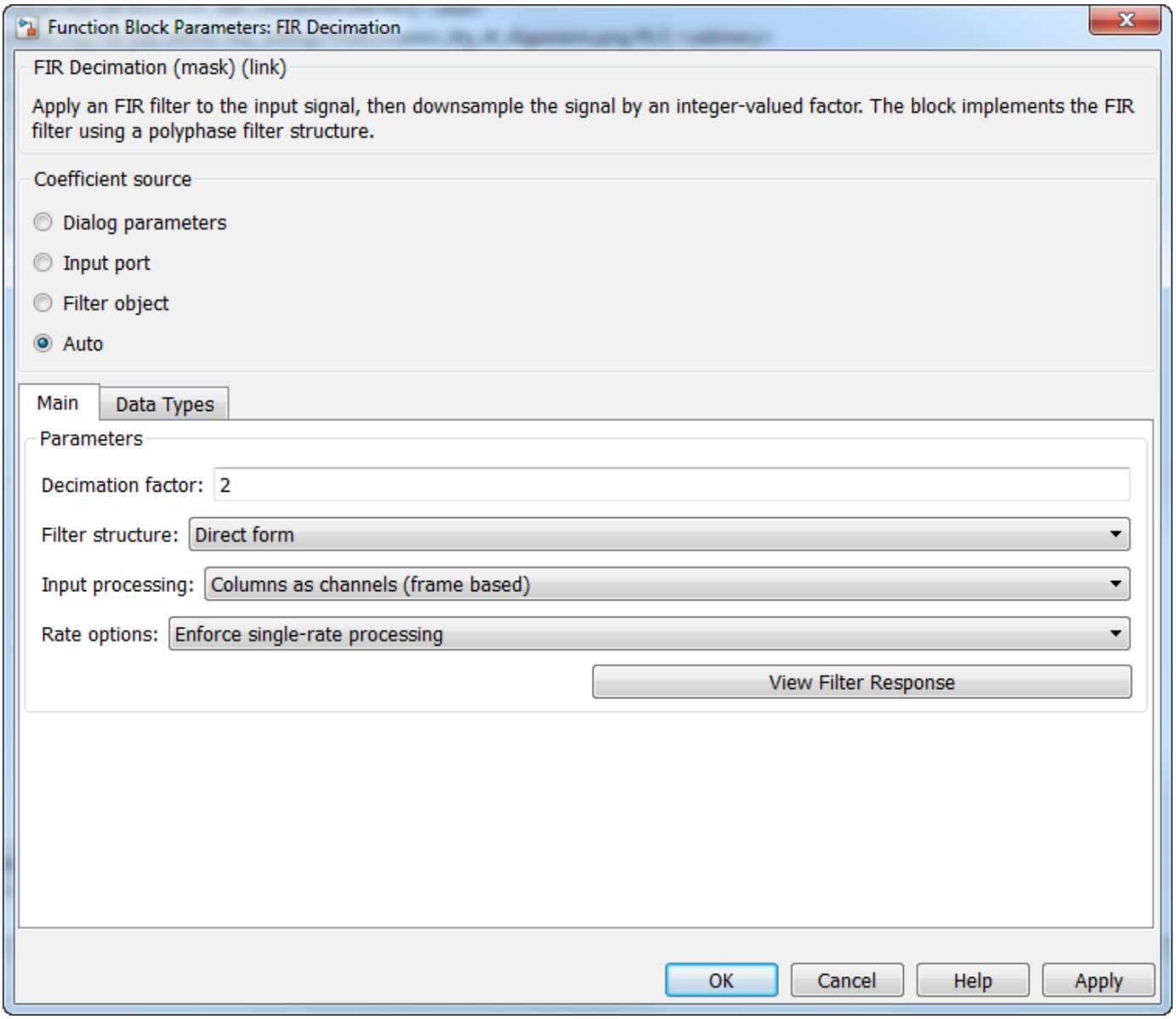
You can design and implement the FIR multirate filters in Simulink using the FIR Decimation, FIR Interpolation, and FIR Rate Conversion blocks. When you set **Coefficient source** to **Dialog parameters**, you can provide `designMultirateFIR(1,2)` as a parameter to specify the filter coefficients. To design the Decimator using the `designMultirateFIR` function, you must specify the decimation factor of interest (usually greater than 1) and an interpolation factor equal to 1. You can use the default half-polyphase length of 12 and the default stopband attenuation of 80 dB. Alternatively, you can also specify the half-polyphase length and stopband attenuation values.



The block chooses the coefficients computed using the `designMultirateFIR` function.

Similarly, you can design an FIR interpolator and FIR rate converter by providing appropriate inputs to the `designMultirateFIR` function.

When you set **Coefficient source** to **Auto**, the block computes the coefficients using the `designMultirateFIR` function. The function uses the decimation factor specified in the block dialog box.



You can design an FIR Interpolator and an FIR Rate Converter using a similar approach in the corresponding blocks.

Sample Rate Conversion

Sample rate conversion is a process of converting the sample rate of a signal from one sampling rate to another sampling rate. Multistage filters minimize the amount of computation involved in a sample rate conversion. To perform an efficient multistage rate conversion, the `dsp.SampleRateConverter` object:

1. accepts input sample rate and output sample rate as inputs.
2. partitions the design problem into optimal stages.

3 designs all the filters required by the various stages.

4 implements the design.

The design makes sure that aliasing does not occur in the intermediate steps.

In this example, change the sample rate of a noisy sine wave signal from an input rate of 192 kHz to an output rate of 44.1 kHz. Initialize a sample rate converter object.

```
SRC = dsp.SampleRateConverter;
```

Display the filter information.

```
info(SRC)
```

```
ans =

    'Overall Interpolation Factor    : 147
     Overall Decimation Factor       : 640
     Number of Filters               : 3
     Multiplications per Input Sample: 27.667188
     Number of Coefficients          : 8631
     Filters:
        Filter 1:
        dsp.FIRDecimator     - Decimation Factor   : 2
        Filter 2:
        dsp.FIRDecimator     - Decimation Factor   : 2
        Filter 3:
        dsp.FIRRateConverter - Interpolation Factor: 147
                             - Decimation Factor   : 160
     '
```

SRC is a three-stage filter: two FIR decimators followed by an FIR rate converter.

Initialize the sine wave source. The sine wave has two tones: one at 2000 Hz and the other at 5000 Hz.

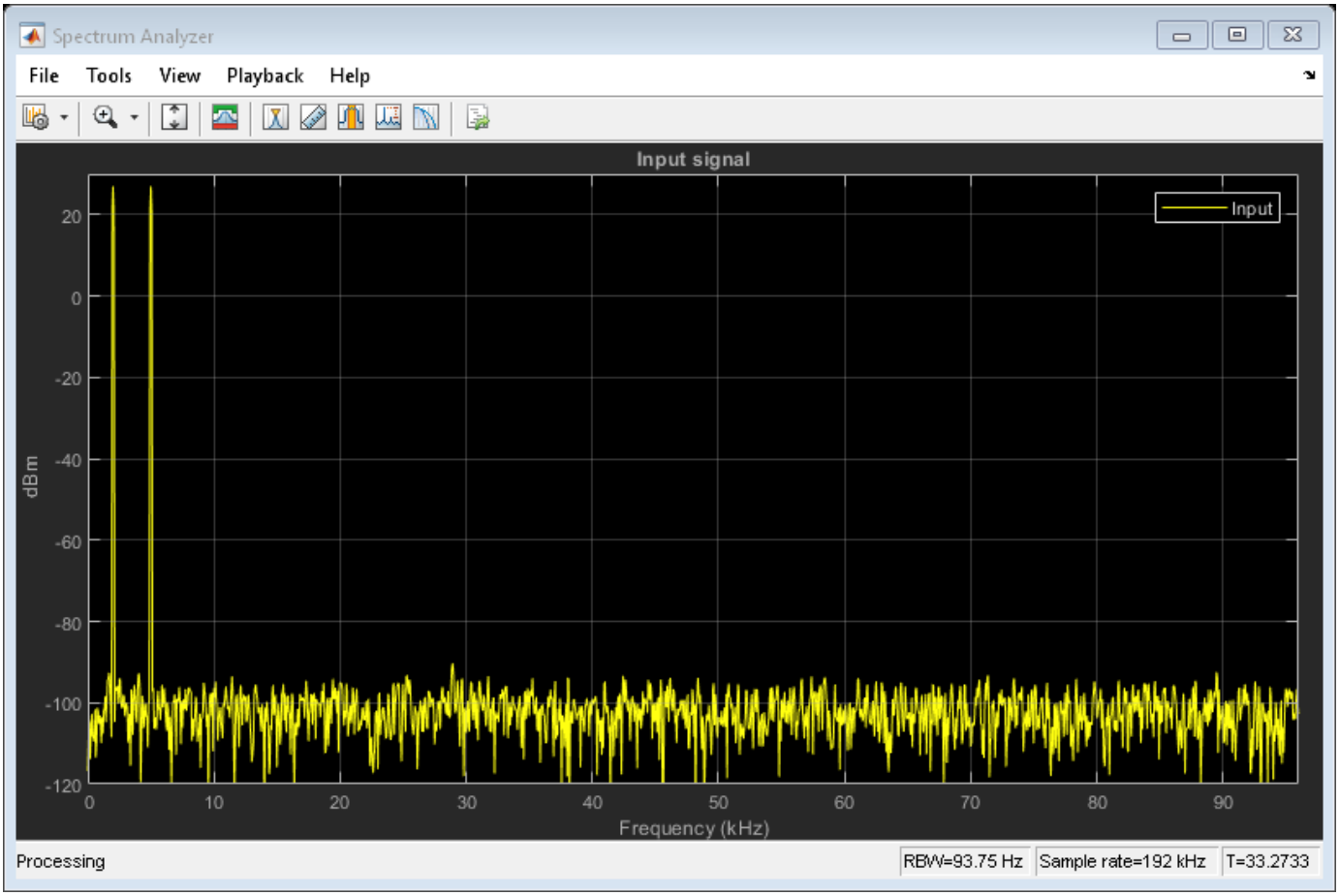
```
source = dsp.SineWave ('Frequency',[2000 5000],'SampleRate',192000,...
    'SamplesPerFrame',1280);
```

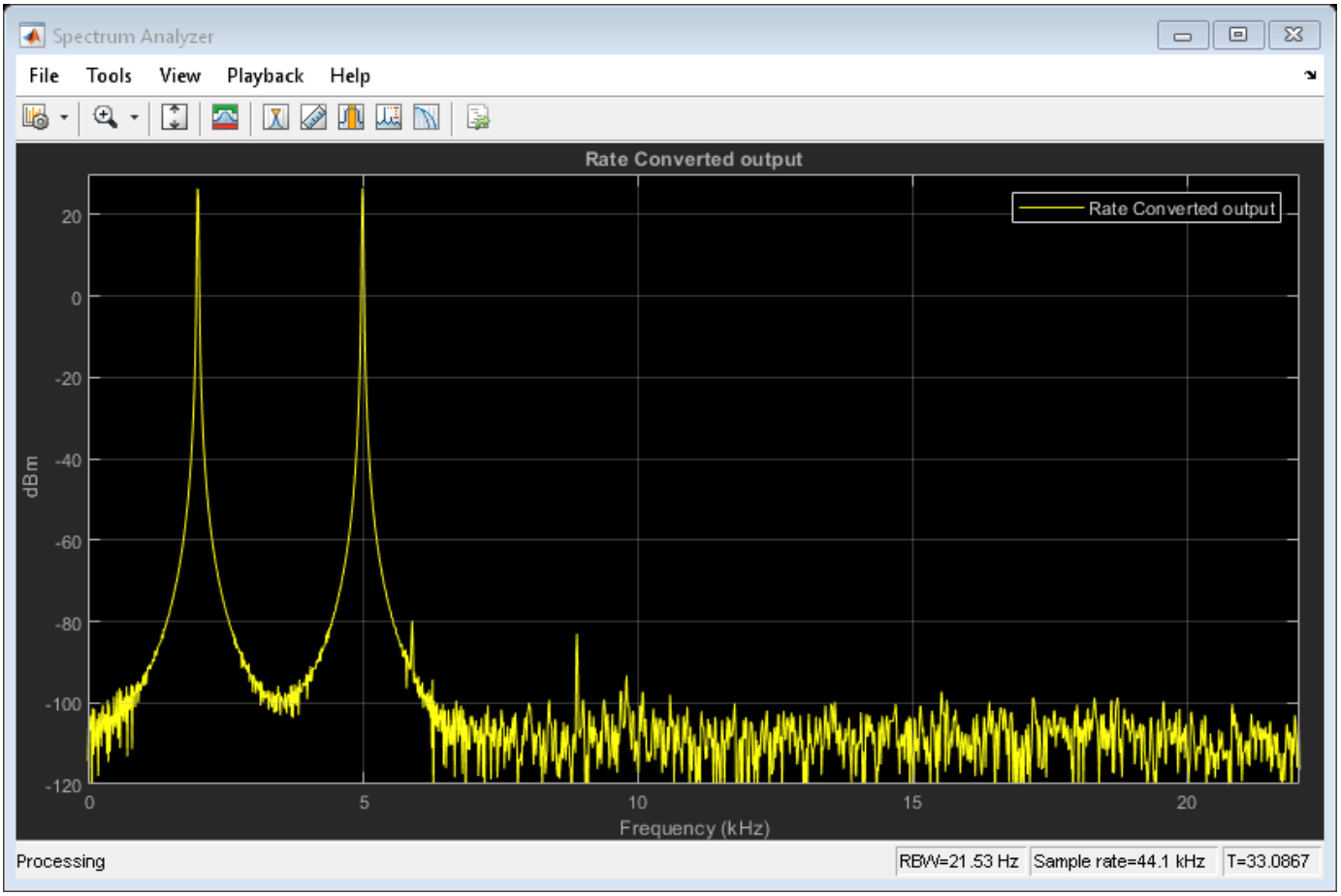
Initialize the spectrum analyzer to see the input and output signals.

```
Fsin = SRC.InputSampleRate;
Fsout = SRC.OutputSampleRate;
specanainput = dsp.SpectrumAnalyzer('SampleRate',Fsin,...
    'PlotAsTwoSidedSpectrum',false,...
    'ShowLegend',true,'YLimits',[-120 30],...
    'Title','Input signal',...
    'ChannelNames', {'Input'});
specanaoutput = dsp.SpectrumAnalyzer('SampleRate',Fsout,...
    'PlotAsTwoSidedSpectrum',false,...
    'ShowLegend',true,'YLimits',[-120 30],...
    'Title','Rate Converted output',...
    'ChannelNames', {'Rate Converted output'});
```

Stream in the input signal and convert the signal's sample rate.

```
for Iter = 1 : 5000
    input = sum(source(),2);
    noisyinput = input + (10^-5)*randn(1280,1);
    output = SRC(noisyinput);
    specanainput(noisyinput);
    specanaoutput(output);
end
```





The spectrum shown is one-sided in the range [0 Fs/2]. For the spectrum analyzer showing the input, Fs/2 is 192000/2. For the spectrum analyzer showing the output, Fs/2 is 44100/2. Hence, the sample rate of the signal changed from 192 kHz to 44.1 kHz.

References

[1] Harris, Fredric J. *Multirate Signal Processing for Communication Systems*. Prentice Hall PTR, 2004.

See Also

More About

- “Multirate Filters” on page 6-2
- “Multistage Filters” on page 6-5
- “Filter Banks” on page 6-21
- “Design of Decimators/Interpolators” on page 6-9

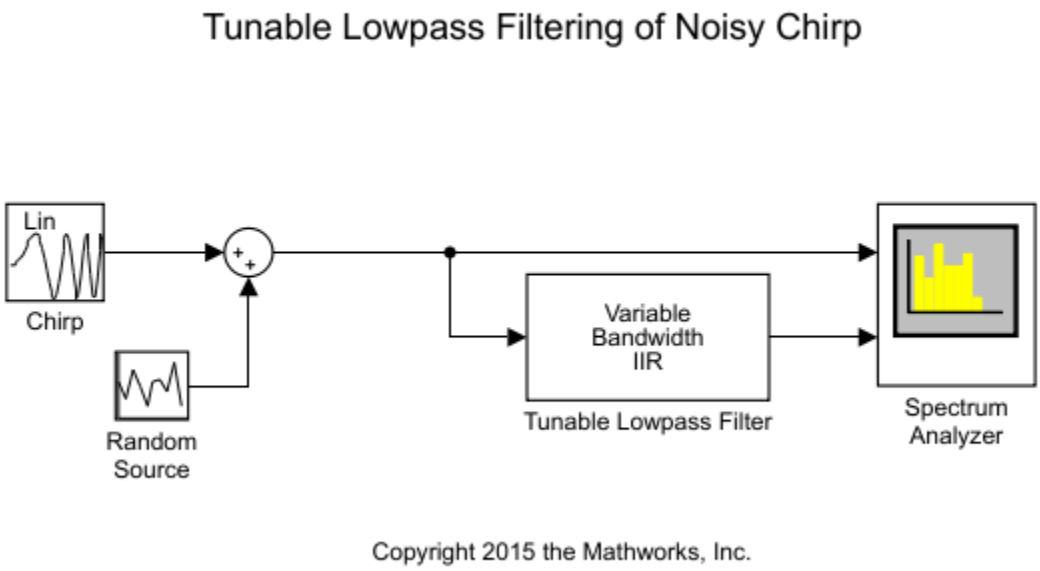
Tunable Lowpass Filtering of Noisy Input in Simulink

In this section...
“Open Lowpass Filter Model” on page 1-45 “Simulate the Model” on page 1-47

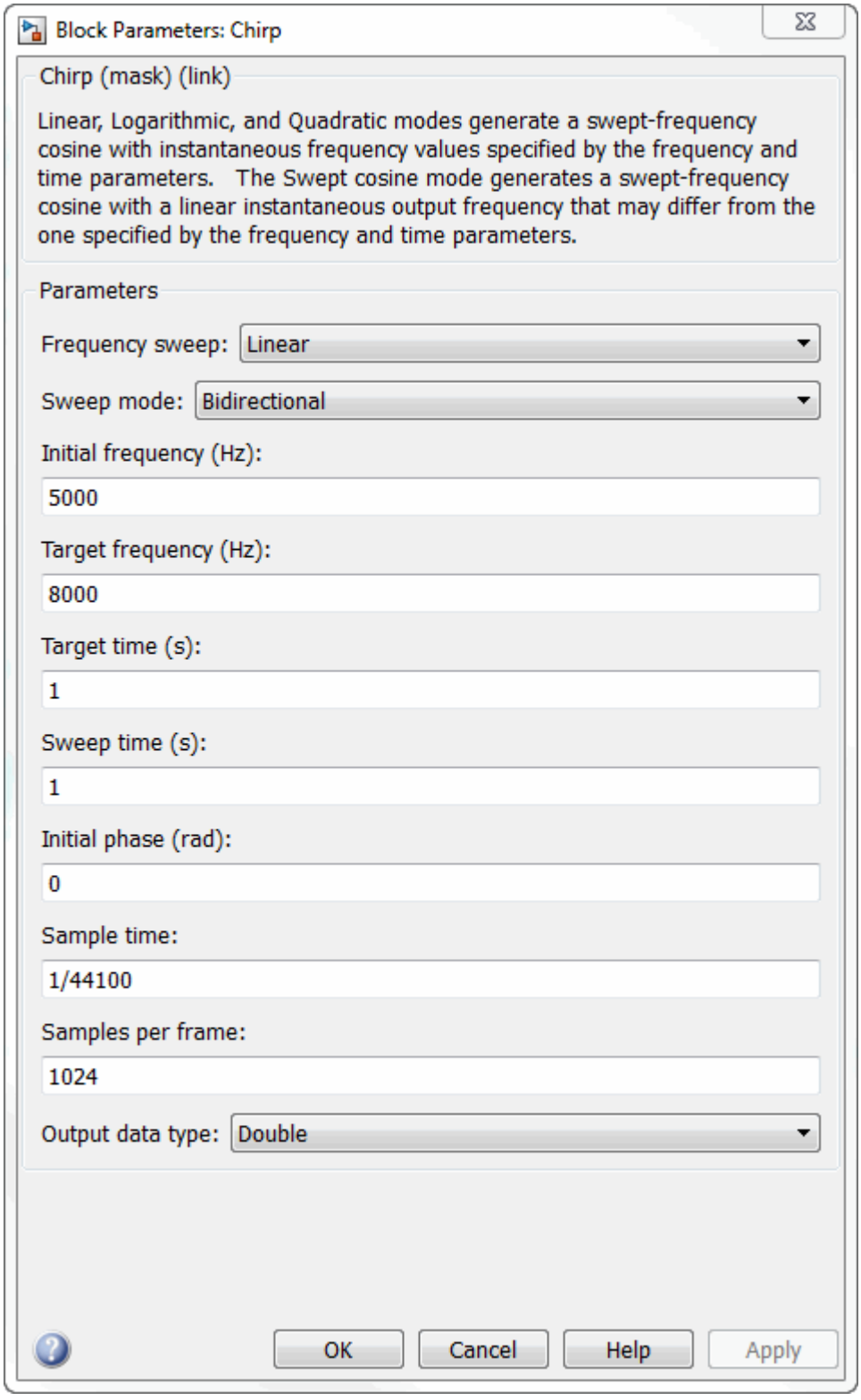
This example shows how to filter a noisy chirp signal with a lowpass filter that has a tunable passband frequency. The filter is a Variable Bandwidth IIR Filter block with **Filter type** set to `Lowpass`. This type of filter enables you to change the passband frequency during simulation without having to redesign the whole filter. The filter algorithm recomputes the filter coefficients whenever the passband frequency changes.

Open Lowpass Filter Model

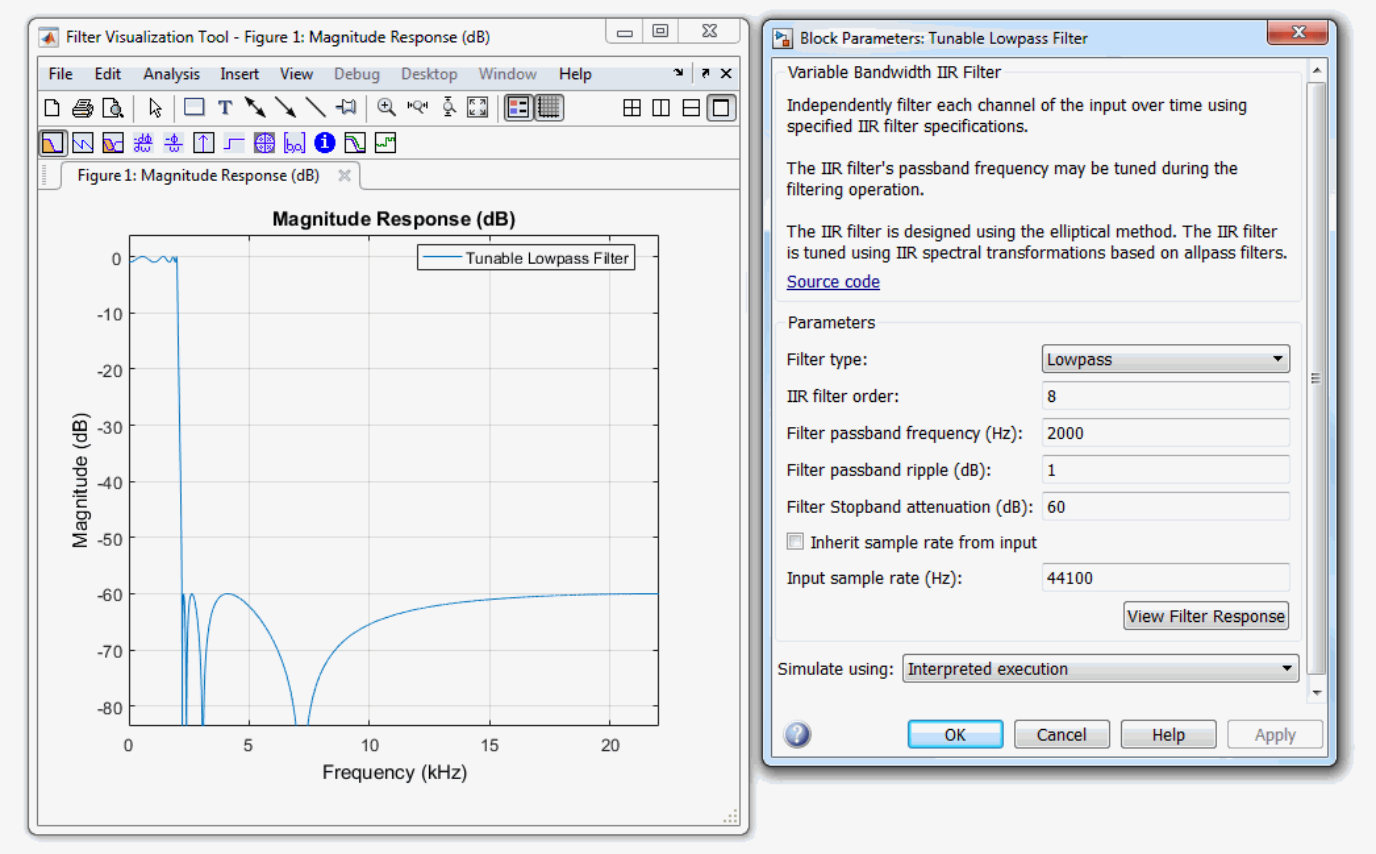
```
model = 'ex_tunable_chirp_lowpass';
open_system(model);
```



The input signal is a noisy chirp sampled at 44.1 kHz. The chirp has an initial frequency of 5000 Hz and a target frequency of 8000 Hz.

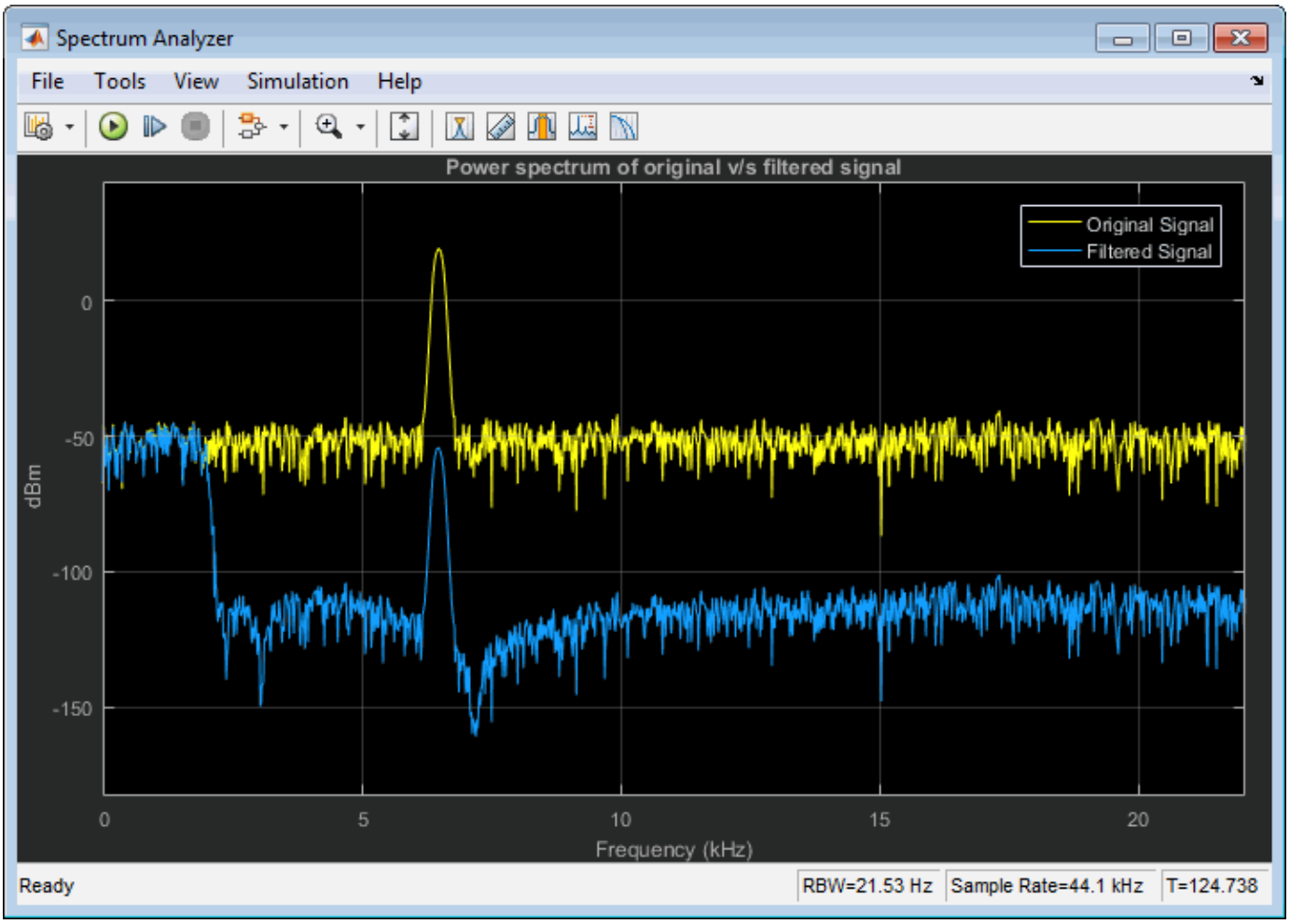


The Variable Bandwidth IIR Filter block has a lowpass frequency response, with the passband frequency set to 2000 Hz.



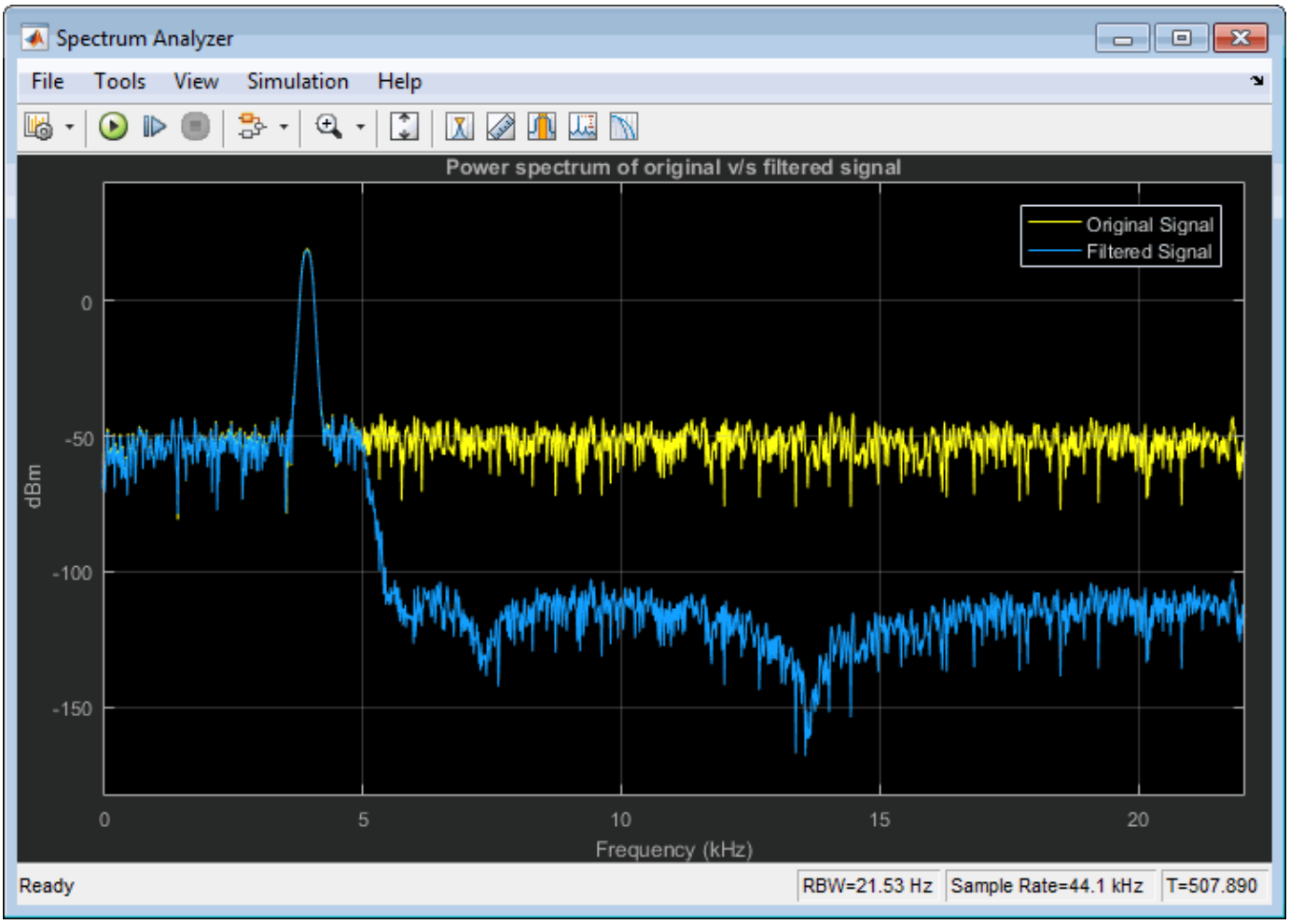
Simulate the Model

After you configure the block parameters, simulate the model. In the initial configuration, the chirp sweeps from 5000 Hz to 8000 Hz which falls in the stopband of the filter. When the chirp input passes through this filter, the filter attenuates the chirp.

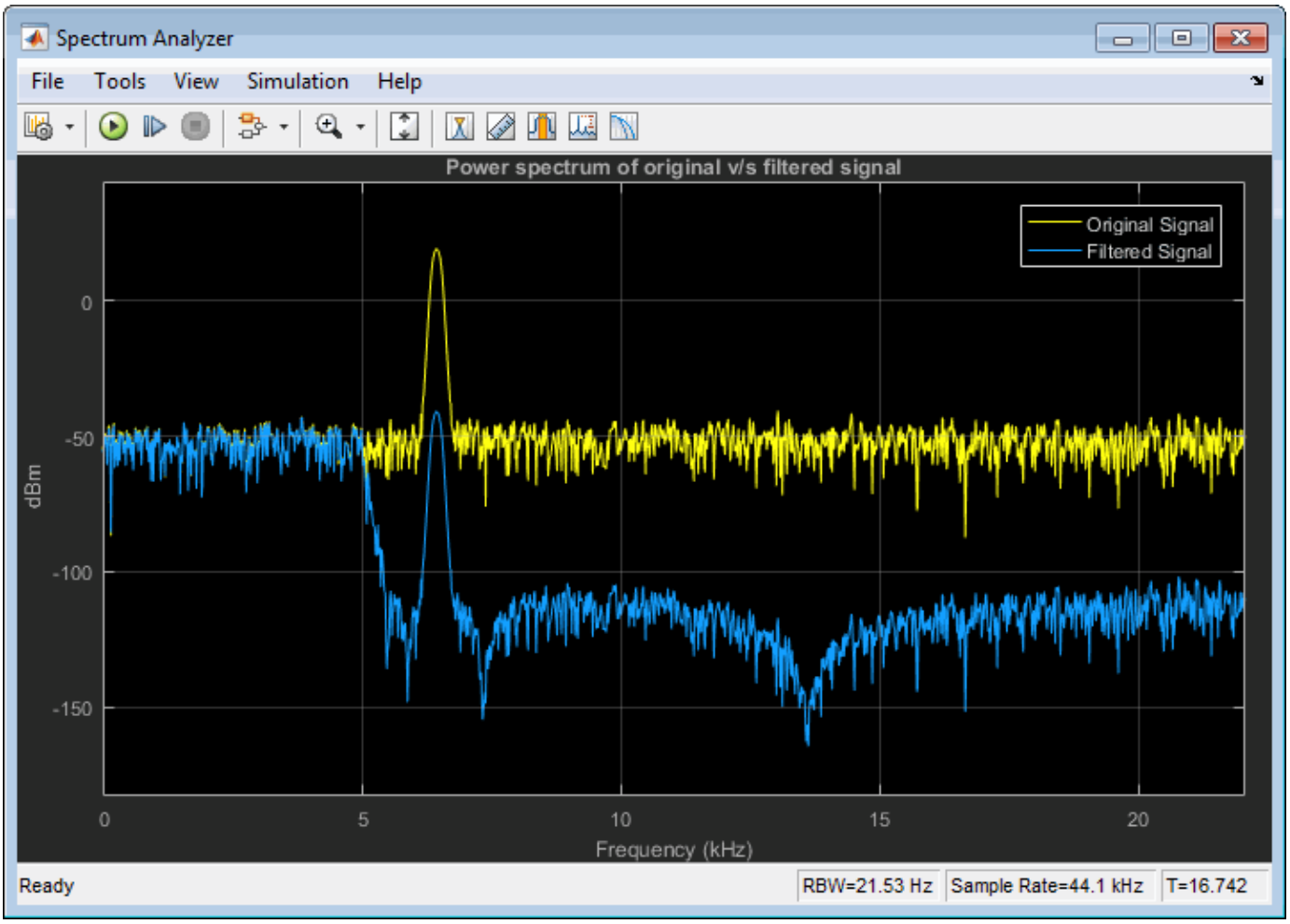


To tune the Passband frequency of the filter, in the Variable Bandwidth IIR Filter block dialog box, change **Filter passband frequency (Hz)** to 6000 Hz. Click **Apply** and the output of the Spectrum Analyzer changes immediately.

The chirp's sweep frequency ranges from 5000 to 8000 Hz. Part of this frequency range is in the passband and the remaining part is in the stopband. While in the filter's passband frequency, the chirp is unaffected.



While in the filter's stopband frequency, the chirp is attenuated.



During simulation, you can tune any of the tunable parameters in the model and see the effect on the filtered output real time.

See Also

"Lowpass IIR Filter Design in Simulink" on page 1-20 | "Design Multirate Filters" on page 1-36 | "Filter Frames of a Noisy Sine Wave Signal in MATLAB" on page 1-6 | "Filter Frames of a Noisy Sine Wave Signal in Simulink" on page 1-8 | "Introduction to Streaming Signal Processing in MATLAB" on page 1-2

Signal Processing Algorithm Acceleration in MATLAB

In this section...
“FIR Filter Algorithm” on page 1-51
“Accelerate the FIR Filter Using codegen” on page 1-52
“Accelerate the FIR Filter Using dspunfold” on page 1-53
“Kalman Filter Algorithm” on page 1-54
“Accelerate the Kalman Filter Using codegen” on page 1-56
“Accelerate the Kalman Filter Using dspunfold” on page 1-57

Note The benchmarks in this example have been measured on a machine with four physical cores.

This example shows how to accelerate a signal processing algorithm in MATLAB using the `codegen` and `dspunfold` functions. You can generate a MATLAB executable (MEX function) from an entire MATLAB function or specific parts of the MATLAB function. When you run the MEX function instead of the original MATLAB code, simulation speed can increase significantly. To generate the MEX equivalent, the algorithm must support code generation.

To use `codegen`, you must have MATLAB Coder installed. To use `dspunfold`, you must have MATLAB Coder and DSP System Toolbox installed.

To use `dspunfold` on Windows and Linux, you must use a compiler that supports the Open Multi-Processing (OpenMP) application interface. See `https://www.mathworks.com/support/compilers/current_release/`.

FIR Filter Algorithm

Consider a simple FIR filter algorithm to accelerate. Copy the `firfilter` function code into the `firfilter.m` file.

```
function [y,z1] = firfilter(b,x)
% Inputs:
%   b - 1xNTaps row vector of coefficients
%   x - A frame of  noisy input

% States:
%   z, z1 - NTapsx1 column vector of states

% Output:
%   y - A frame of filtered output

persistent z;

if (isempty(z))
    z = zeros(length(b),1);
end
Lx = size(x,1);
y = zeros(size(x),'like',x);

z1 = z;
```

```
for m = 1:Lx
    % Load next input sample
    z1(1,:) = x(m,:);

    % Compute output
    y(m,:) = b*z1;

    % Update states
    z1(2:end,:) = z1(1:end-1,:);
    z = z1;
end
```

The `firfilter` function accepts a vector of filter coefficients, *b*, a noisy input signal, *x*, as inputs. Generate the filter coefficients using the `fir1` function.

```
NTaps = 250;
Fp = 4e3/(44.1e3/2);
b = fir1(NTaps-1,Fp);
```

Filter a stream of a noisy sine wave signal by using the `firfilter` function. The sine wave has a frame size of 4000 samples and a sample rate of 192 kHz. Generate the sine wave using the `dsp.SineWave` System object™. The noise is a white Gaussian with a mean of 0 and a variance of 0.02. Name this function `firfilter_sim`. The `firfilter_sim` function calls the `firfilter` function on the noisy input.

```
function totVal = firfilter_sim(b)
% Create the signal source
Sig = dsp.SineWave('SamplesPerFrame',4000,'SampleRate',19200);
totVal = zeros(4000,500);
R  = 0.02;

clear firfilter;

% Iteration loop. Each iteration filters a frame of the noisy signal.
for i = 1 : 500
    trueVal = Sig();                        % Original sine wave
    noisyVal = trueVal + sqrt(R)*randn;     % Noisy sine wave
    filteredVal = firfilter(b,noisyVal);    % Filtered sine wave
    totVal(:,i) = filteredVal;              % Store the entire sine wave
end
```

Run `firfilter_sim` and measure the speed of execution. The execution speed varies depending on your machine.

```
tic;totVal = firfilter_sim(b);t1 = toc;
fprintf('Original Algorithm Simulation Time: %4.1f seconds\n',t1);
```

```
Original Algorithm Simulation Time:  7.8 seconds
```

Accelerate the FIR Filter Using codegen

Call `codegen` on `firfilter`, and generate its MEX equivalent, `firfilter_mex`. Generate and pass the filter coefficients and the sine wave signal as inputs to the `firfilter` function.

```
Ntaps = 250;
Sig = dsp.SineWave('SamplesPerFrame',4000,'SampleRate',19200); % Create the Signal Source
R = 0.02;
trueVal = Sig();                    % Original sine wave
noisyVal = trueVal + sqrt(R)*randn; % Noisy sine wave
```

```
Fp = 4e3/(44.1e3/2);
b = fir1(Ntaps-1,Fp);               % Filter coefficients

codegen firfilter -args {b,noisyVal}
```

In the `firfilter_sim` function, replace `firfilter(b,noisyVal)` function call with `firfilter_mex(b,noisyVal)`. Name this function `firfilter_codegen`.

```
function totVal = firfilter_codegen(b)
% Create the signal source
Sig = dsp.SineWave('SamplesPerFrame',4000,'SampleRate',19200);
totVal = zeros(4000,500);
R  = 0.02;

clear firfilter_mex;

% Iteration loop. Each iteration filters a frame of the noisy signal.
for i = 1 : 500
    trueVal = Sig();                          % Original sine wave
    noisyVal = trueVal + sqrt(R)*randn;       % Noisy sine wave
    filteredVal = firfilter_mex(b,noisyVal);  % Filtered sine wave
    totVal(:,i) = filteredVal;                % Store the entire sine wave
end
```

Run `firfilter_codegen` and measure the speed of execution. The execution speed varies depending on your machine.

```
tic;totValcodegen = firfilter_codegen(b);t2 = toc;
fprintf('Algorithm Simulation Time with codegen: %5f seconds\n',t2);
fprintf('Speedup factor with codegen: %5f\n',(t1/t2));
```

```
Algorithm Simulation Time with codegen: 0.923683 seconds
Speedup factor with codegen: 8.5531
```

The speedup gain is approximately `8.5`.

Accelerate the FIR Filter Using dspunfold

The `dspunfold` function generates a multithreaded MEX file which can improve the speedup gain even further.

`dspunfold` also generates a single-threaded MEX file and a self-diagnostic analyzer function. The multithreaded MEX file leverages the multicore CPU architecture of the host computer. The single-threaded MEX file is similar to the MEX file that the `codegen` function generates. The analyzer function measures the speedup gain of the multithreaded MEX file over the single-threaded MEX file.

Call `dspunfold` on `firfilter` and generate its multithreaded MEX equivalent, `firfilter_mt`. Detect the state length in samples by using the `-f` option, which can improve the speedup gain further. `-s auto` triggers the automatic state length detection. For more information on using the `-f` and `-s` options, see `dspunfold`.

```
dspunfold firfilter -args {b,noisyVal} -s auto -f [false,true]
```

```
State length: [autodetect] samples, Repetition: 1, Output latency: 8 frames, Threads: 4
Analyzing: firfilter.m
Creating single-threaded MEX file: firfilter_st.mexw64
Searching for minimal state length (this might take a while)
Checking stateless ... Insufficient
Checking 4000 samples ... Sufficient
Checking 2000 samples ... Sufficient
Checking 1000 samples ... Sufficient
Checking 500 samples ... Sufficient
Checking 250 samples ... Sufficient
Checking 125 samples ... Insufficient
Checking 187 samples ... Insufficient
Checking 218 samples ... Insufficient
```

```
Checking 234 samples ... Insufficient
Checking 242 samples ... Insufficient
Checking 246 samples ... Insufficient
Checking 248 samples ... Insufficient
Checking 249 samples ... Sufficient
Minimal state length is 249 samples
Creating multi-threaded MEX file: firfilter_mt.mexw64
Creating analyzer file: firfilter_analyzer.p
```

The automatic state length detection tool detects an exact state length of 259 samples.

Call the analyzer function and measure the speedup gain of the multithreaded MEX file with respect to the single-threaded MEX file. Provide at least two different frames for each input argument of the analyzer. The frames are appended along the first dimension. The analyzer alternates between these frames while verifying that the outputs match. Failure to provide multiple frames for each input can decrease the effectiveness of the analyzer and can lead to false positive verification results.

```
firfilter_analyzer([b;0.5*b;0.6*b],[noisyVal;0.5*noisyVal;0.6*noisyVal]);
```

```
Analyzing multi-threaded MEX file firfilter_mt.mexw64. For best results,
please refrain from interacting with the computer and stop other processes until the
analyzer is done.
Latency = 8 frames
Speedup = 3.2x
```

`firfilter_mt` has a speedup gain factor of `3.2` when compared to the single-threaded MEX file, `firfilter_st`. To increase the speedup further, increase the repetition factor using the `-r` option. The tradeoff is that the output latency increases. Use a repetition factor of 3. Specify the exact state length to reduce the overhead and increase the speedup further.

```
dspunfold firfilter -args {b,noisyVal} -s 249 -f [false,true] -r 3
```

```
State length: 249 samples, Repetition: 3, Output latency: 24 frames, Threads: 4
Analyzing: firfilter.m
Creating single-threaded MEX file: firfilter_st.mexw64
Creating multi-threaded MEX file: firfilter_mt.mexw64
Creating analyzer file: firfilter_analyzer.p
```

Call the analyzer function.

```
firfilter_analyzer([b;0.5*b;0.6*b],[noisyVal;0.5*noisyVal;0.6*noisyVal]);
```

```
Analyzing multi-threaded MEX file firfilter_mt.mexw64. For best results,
please refrain from interacting with the computer and stop other processes
until the analyzer is done.
Latency = 24 frames
Speedup = 3.8x
```

The speedup gain factor is `3.8`, or approximately 32 times the speed of execution of the original simulation.

For this particular algorithm, you can see that `dspunfold` is generating a highly optimized code, without having to write any C or C++ code. The speedup gain scales with the number of cores on your host machine.

The FIR filter function in this example is only an illustrative algorithm that is easy to understand. You can apply this workflow on any of your custom algorithms. If you want to use an FIR filter, it is recommended that you use the `dsp.FIRFilter` System object in DSP System Toolbox. This object runs much faster than the benchmark numbers presented in this example, without the need for code generation.

Kalman Filter Algorithm

Consider a Kalman filter algorithm, which estimates the sine wave signal from a noisy input. This example shows the performance of Kalman filter with `codegen` and `dspunfold`.

The noisy sine wave input has a frame size of 4000 samples and a sample rate of 192 kHz. The noise is a white Gaussian with mean of 0 and a variance of 0.02.

The function `filterNoisySignal` calls the `kalmanfilter` function on the noisy input.

```
type filterNoisySignal
```

```
function totVal = filterNoisySignal
% Create the signal source
Sig = dsp.SineWave('SamplesPerFrame',4000,'SampleRate',19200);
totVal = zeros(4000,500);
R  = 0.02;
clear kalmanfilter;
% Iteration loop to estimate the sine wave signal
for i = 1 : 500
    trueVal = Sig();                    % Actual values
    noisyVal = trueVal + sqrt(R)*randn; % Noisy measurements
    estVal = kalmanfilter(noisyVal);    % Sine wave estimated by Kalman filter
    totVal(:,i) = estVal;               % Store the entire sine wave
end
```

```
type kalmanfilter
```

```
function [estVal,estState] = kalmanfilter(noisyVal)
% This function tracks a noisy sinusoid signal using a Kalman filter
%
% State Transition Matrix
A = 1;
stateSpaceDim = size(A,1);

% Measurement Matrix
H = 1;
measurementSpaceDim = size(H,1);
numTsteps = size(noisyVal,1)/measurementSpaceDim;

% Containers to store prediction and estimates for all time steps
zEstContainer = noisyVal;
xEstContainer = zeros(size(noisyVal));

Q = 0.0001; % Process noise covariance
R = 0.02;   % Measurement noise covariance
persistent xhat P xPrior PPrior;

% Local copies of discrete states
if isempty(xhat)
    xhat = 5; % Initial state estimate
end

if isempty(P)
    P = 1; % Error covariance estimate
end

if isempty(xPrior)
    xPrior = 0;
end

if isempty(PPrior)
    PPrior = 0;
end

% Loop over all time steps
for n=1:numTsteps

    % Gather chunks for current time step
    zRowIndexChunk = (n-1)*measurementSpaceDim + (1:measurementSpaceDim);
    stateEstsRowIndexChunk = (n-1)*stateSpaceDim + (1:stateSpaceDim);

    % Prediction step
    xPrior = A * xhat;
    PPrior =  A * P * A' + Q;

    % Correction step. Compute Kalman gain.
    PpriorH   = PPrior * H';
    HPpriorHR  = H * PpriorH + R;
    KalmanGain = (HPpriorHR \ PpriorH')';
    KH  = KalmanGain * H;

    % States and error covariance are updated in the
```

```
    % correction step
    xhat = xPrior + KalmanGain * noisyVal(zRowIndexChunk,:) - ...
        KH * xPrior;
    P = PPrior - KH * PPrior;

    % Append estimates
    xEstContainer(stateEstsRowIndexChunk, :) = xhat;
    zEstContainer(zRowIndexChunk,:) = H*xhat;

end

% Populate the outputs
estVal = zEstContainer;
estState = xEstContainer;

end
```

Run `filterNoisySignal.m` and measure the speed of execution.

```
tic;totVal = filterNoisySignal;t1 = toc;
fprintf('Original Algorithm Simulation Time: %4.1f seconds\n',t1);
```

```
Original Algorithm Simulation Time: 21.7 seconds
```

Accelerate the Kalman Filter Using codegen

Call the `codegen` function on `kalmanfilter`, and generate its MEX equivalent, `kalmanfilter_mex`.

The `kalmanfilter` function requires the noisy sine wave as the input.

```
Sig = dsp.SineWave('SamplesPerFrame',4000,'SampleRate',19200); % Create the Signal Source
R = 0.02;                           % Measurement noise covariance
trueVal = step(Sig);                % Actual values
noisyVal = trueVal + sqrt(R)*randn; % Noisy measurements
codegen -args {noisyVal} kalmanfilter.m
```

Replace `kalmanfilter(noisyVal)` in `filterNoisySignal` function with `kalmanfilter_mex(noisyVal)`. Name this function as `filterNoisySignal_codegen`

```
function totVal = filterNoisySignal_codegen
% Create the signal source
Sig = dsp.SineWave('SamplesPerFrame',4000,'SampleRate',19200);
totVal = zeros(4000,500);
R  = 0.02;
clear kalmanfilter_mex;
% Iteration loop to estimate the sine wave signal
for i = 1 : 500
    trueVal = Sig();                        % Actual values
    noisyVal = trueVal + sqrt(R)*randn;     % Noisy measurements
    estVal = kalmanfilter_mex(noisyVal);    % Sine wave estimated by Kalman filter
    totVal(:,i) = estVal; % Store the entire sine wave
end
```

Run `filterNoisySignal_codegen` and measure the speed of execution.

```
tic; totValcodegen = filterNoisySignal_codegen; t2 = toc;
fprintf('Algorithm Simulation Time with codegen: %5f seconds\n',t2);
fprintf('Speedup with codegen is %0.1f',t1/t2);
```

```
Algorithm Simulation Time with codegen: 0.095480 seconds
Speedup with codegen is 227.0
```

The Kalman filter algorithm implements several matrix multiplications. `codegen` uses the Basic Linear Algebra Subroutines (BLAS) libraries to perform these multiplications. These libraries generate a highly optimized code, hence giving a speedup gain of 227.

Accelerate the Kalman Filter Using dspunfold

Generate a multithreaded MEX file using `dspunfold` and compare its performance with `codegen`.

```
Sig = dsp.SineWave('SamplesPerFrame',4000,'SampleRate',19200);
% Create the signal source
R = 0.02;                           % Measurement noise covariance
trueVal = step(Sig);                % Actual values
noisyVal = trueVal + sqrt(R)*randn; % Noisy measurements
dspunfold kalmanfilter -args {noisyVal} -s auto
```

```
State length: [autodetect] frames, Repetition: 1, Output latency: 8 frames, Threads: 4
Analyzing: kalmanfilter.m
Creating single-threaded MEX file: kalmanfilter_st.mexw64
Searching for minimal state length (this might take a while)
Checking stateless ... Insufficient
Checking 1 frames ... Sufficient
Minimal state length is 1 frames
Creating multi-threaded MEX file: kalmanfilter_mt.mexw64
Creating analyzer file: kalmanfilter_analyzer.p
```

Call the analyzer function.

```
kalmanfilter_analyzer([noisyVal;0.01*noisyVal;0.05*noisyVal;0.1*noisyVal]);
```

```
Analyzing multi-threaded MEX file kalmanfilter_mt.mexw64. For best results,
please refrain from interacting with the computer and stop other processes until
the analyzer is done.
Latency = 8 frames
Speedup = 0.7x
```

`kalmanfilter_mt` has a speedup factor of `0.7`, which is a performance loss of `30%` when compared to the single-threaded MEX file, `kalmanfilter_st`. Increase the repetition factor to `3` to see if the performance increases. Also, detect the state length in samples.

```
dspunfold kalmanfilter -args {noisyVal} -s auto -f true -r 3
```

```
State length: [autodetect] samples, Repetition: 3, Output latency: 24 frames, Threads: 4
Analyzing: kalmanfilter.m
Creating single-threaded MEX file: kalmanfilter_st.mexw64
Searching for minimal state length (this might take a while)
Checking stateless ... Insufficient
Checking 4000 samples ... Sufficient
Checking 2000 samples ... Sufficient
Checking 1000 samples ... Sufficient
Checking 500 samples ... Sufficient
Checking 250 samples ... Insufficient
Checking 375 samples ... Sufficient
Checking 312 samples ... Sufficient
Checking 281 samples ... Sufficient
Checking 265 samples ... Sufficient
Checking 257 samples ... Insufficient
Checking 261 samples ... Sufficient
Checking 259 samples ... Sufficient
Checking 258 samples ... Insufficient
Minimal state length is 259 samples
Creating multi-threaded MEX file: kalmanfilter_mt.mexw64
Creating analyzer file: kalmanfilter_analyzer.p
```

Call the analyzer function.

```
kalmanfilter_analyzer([noisyVal;0.01*noisyVal;0.05*noisyVal;0.1*noisyVal]);
```

```
Analyzing multi-threaded MEX file kalmanfilter_mt.mexw64. For best results,
please refrain from interacting with the computer and stop other processes until the
analyzer is done.
Latency = 24 frames
Speedup = 1.4x
```

`dspunfold` gives a speedup gain of `40%` when compared to the highly optimized single-threaded MEX file. Specify the exact state length and increase the repetition factor to `4`.

```
dspunfold kalmanfilter -args {noisyVal} -s 259 -f true -r 4
```

```
State length: 259 samples, Repetition: 4, Output latency: 32 frames, Threads: 4
Analyzing: kalmanfilter.m
Creating single-threaded MEX file: kalmanfilter_st.mexw64
Creating multi-threaded MEX file: kalmanfilter_mt.mexw64
Creating analyzer file: kalmanfilter_analyzer.p
```

Invoke the analyzer function to see the speedup gain.

```
kalmanfilter_analyzer([noisyVal;0.01*noisyVal;0.05*noisyVal;0.1*noisyVal]);
```

```
Analyzing multi-threaded MEX file kalmanfilter_mt.mexw64. For best results, please refrain
from interacting with the computer and stop other processes until the analyzer is done.
Latency = 32 frames
Speedup = 1.5x
```

The speedup gain factor is `50%` when compared to the single-threaded MEX file.

The performance gain factors `codegen` and `dspunfold` give depend on your algorithm. `codegen` provides sufficient acceleration for some MATLAB constructs. `dspunfold` can provide additional performance gains using the cores available on your machine to distribute your algorithm via unfolding. As shown in this example, the amount of speedup that `dspunfold` provides depends on the particular algorithm to accelerate. Use `dspunfold` in addition to `codegen` if your algorithm is well-suited for distributing via unfolding, and if the resulting latency cost is in line with the constraints of your application.

Some MATLAB constructs are highly optimized with MATLAB interpreted execution. The `fft` function, for example, runs much faster in interpreted simulation than with code generation.

See Also

More About

- “Multithreaded MEX File Generation” on page 1-64
- “Generate a Multi-Threaded MEX File from a MATLAB Function using DSP Unfolding”
- “Accelerated Polyphase Synthesis Filter Bank”
- “Workflow for Accelerating MATLAB Algorithms” (MATLAB Coder)
- “Accelerate MATLAB Algorithms” (MATLAB Coder)

Signal Processing Acceleration through Code Generation

In this section...
“FIR Filter Algorithm” on page 1-59
“Accelerate the FIR Filter Using codegen” on page 1-60
“Accelerate the FIR Filter Using dspunfold” on page 1-61

Note The benchmarks in this example have been measured on a machine with four physical cores.

This example shows how to accelerate a signal processing algorithm in MATLAB using the `codegen` and `dspunfold` functions. You can generate a MATLAB executable (MEX function) from an entire MATLAB function or specific parts of the MATLAB function. When you run the MEX function instead of the original MATLAB code, simulation speed can increase significantly. To generate the MEX equivalent, the algorithm must support code generation.

To use `codegen`, you must have MATLAB Coder installed. To use `dspunfold`, you must have MATLAB Coder and DSP System Toolbox installed.

To use `dspunfold` on Windows and Linux, you must use a compiler that supports the Open Multi-Processing (OpenMP) application interface. See `https://www.mathworks.com/support/compilers/current_release/`.

FIR Filter Algorithm

Consider a simple FIR filter algorithm to accelerate. Copy the `firfilter` function code into the `firfilter.m` file.

```
function [y,z1] = firfilter(b,x)
% Inputs:
%   b - 1xNTaps row vector of coefficients
%   x - A frame of  noisy input

% States:
%   z, z1 - NTapsx1 column vector of states

% Output:
%   y - A frame of filtered output

persistent z;

if (isempty(z))
    z = zeros(length(b),1);
end
Lx = size(x,1);
y = zeros(size(x),'like',x);

z1 = z;
for m = 1:Lx
    % Load next input sample
    z1(1,:) = x(m,:);
```

```
    % Compute output
    y(m,:) = b*z1;

    % Update states
    z1(2:end,:) = z1(1:end-1,:);
    z = z1;
end
```

The `firfilter` function accepts a vector of filter coefficients, *b*, a noisy input signal, *x*, as inputs. Generate the filter coefficients using the `fir1` function.

```
NTaps = 250;
Fp = 4e3/(44.1e3/2);
b = fir1(NTaps-1,Fp);
```

Filter a stream of a noisy sine wave signal by using the `firfilter` function. The sine wave has a frame size of 4000 samples and a sample rate of 192 kHz. Generate the sine wave using the `dsp.SineWave` System object. The noise is a white Gaussian with a mean of 0 and a variance of 0.02. Name this function `firfilter_sim`. The `firfilter_sim` function calls the `firfilter` function on the noisy input.

```
function totVal = firfilter_sim(b)
% Create the signal source
Sig = dsp.SineWave('SamplesPerFrame',4000,'SampleRate',19200);
totVal = zeros(4000,500);
R  = 0.02;

clear firfilter;

% Iteration loop. Each iteration filters a frame of the noisy signal.
for i = 1 : 500
    trueVal = Sig();                        % Original sine wave
    noisyVal = trueVal + sqrt(R)*randn;     % Noisy sine wave
    filteredVal = firfilter(b,noisyVal);    % Filtered sine wave
    totVal(:,i) = filteredVal;              % Store the entire sine wave
end
```

Run `firfilter_sim` and measure the speed of execution. The execution speed varies depending on your machine.

```
tic;totVal = firfilter_sim(b);t1 = toc;
fprintf('Original Algorithm Simulation Time: %4.1f seconds\n',t1);
```

```
Original Algorithm Simulation Time:  7.8 seconds
```

Accelerate the FIR Filter Using codegen

Call `codegen` on `firfilter`, and generate its MEX equivalent, `firfilter_mex`. Generate and pass the filter coefficients and the sine wave signal as inputs to the `firfilter` function.

```
Ntaps = 250;
Sig = dsp.SineWave('SamplesPerFrame',4000,'SampleRate',19200); % Create the Signal Source
R = 0.02;
trueVal = Sig();                    % Original sine wave
noisyVal = trueVal + sqrt(R)*randn; % Noisy sine wave
Fp = 4e3/(44.1e3/2);
b = fir1(Ntaps-1,Fp);               % Filter coefficients

codegen firfilter -args {b,noisyVal}
```

In the `firfilter_sim` function, replace `firfilter(b,noisyVal)` function call with `firfilter_mex(b,noisyVal)`. Name this function `firfilter_codegen`.

```
function totVal = firfilter_codegen(b)
% Create the signal source
Sig = dsp.SineWave('SamplesPerFrame',4000,'SampleRate',19200);
totVal = zeros(4000,500);
R  = 0.02;

clear firfilter_mex;

% Iteration loop. Each iteration filters a frame of the noisy signal.
for i = 1 : 500
    trueVal = Sig();                          % Original sine wave
    noisyVal = trueVal + sqrt(R)*randn;       % Noisy sine wave
    filteredVal = firfilter_mex(b,noisyVal);  % Filtered sine wave
    totVal(:,i) = filteredVal;                % Store the entire sine wave
end
```

Run `firfilter_codegen` and measure the speed of execution. The execution speed varies depending on your machine.

```
tic;totValcodegen = firfilter_codegen(b);t2 = toc;
fprintf('Algorithm Simulation Time with codegen: %5f seconds\n',t2);
fprintf('Speedup factor with codegen: %5f\n',(t1/t2));
```

```
Algorithm Simulation Time with codegen: 0.923683 seconds
Speedup factor with codegen: 8.5531
```

The speedup gain is approximately `8.5`.

Accelerate the FIR Filter Using dspunfold

The `dspunfold` function generates a multithreaded MEX file which can improve the speedup gain even further.

`dspunfold` also generates a single-threaded MEX file and a self-diagnostic analyzer function. The multithreaded MEX file leverages the multicore CPU architecture of the host computer. The single-threaded MEX file is similar to the MEX file that the `codegen` function generates. The analyzer function measures the speedup gain of the multithreaded MEX file over the single-threaded MEX file.

Call `dspunfold` on `firfilter` and generate its multithreaded MEX equivalent, `firfilter_mt`. Detect the state length in samples by using the `-f` option, which can improve the speedup gain further. `-s auto` triggers the automatic state length detection. For more information on using the `-f` and `-s` options, see `dspunfold`.

```
dspunfold firfilter -args {b,noisyVal} -s auto -f [false,true]
```

```
State length: [autodetect] samples, Repetition: 1, Output latency: 8 frames, Threads: 4
Analyzing: firfilter.m
Creating single-threaded MEX file: firfilter_st.mexw64
Searching for minimal state length (this might take a while)
Checking stateless ... Insufficient
Checking 4000 samples ... Sufficient
Checking 2000 samples ... Sufficient
Checking 1000 samples ... Sufficient
Checking 500 samples ... Sufficient
Checking 250 samples ... Sufficient
Checking 125 samples ... Insufficient
Checking 187 samples ... Insufficient
Checking 218 samples ... Insufficient
Checking 234 samples ... Insufficient
Checking 242 samples ... Insufficient
Checking 246 samples ... Insufficient
Checking 248 samples ... Insufficient
Checking 249 samples ... Sufficient
Minimal state length is 249 samples
```

```
Creating multi-threaded MEX file: firfilter_mt.mexw64
Creating analyzer file: firfilter_analyzer.p
```

The automatic state length detection tool detects an exact state length of 259 samples.

Call the analyzer function and measure the speedup gain of the multithreaded MEX file with respect to the single-threaded MEX file. Provide at least two different frames for each input argument of the analyzer. The frames are appended along the first dimension. The analyzer alternates between these frames while verifying that the outputs match. Failure to provide multiple frames for each input can decrease the effectiveness of the analyzer and can lead to false positive verification results.

```
firfilter_analyzer([b;0.5*b;0.6*b],[noisyVal;0.5*noisyVal;0.6*noisyVal]);
```

```
Analyzing multi-threaded MEX file firfilter_mt.mexw64. For best results,
please refrain from interacting with the computer and stop other processes until the
analyzer is done.
Latency = 8 frames
Speedup = 3.2x
```

`firfilter_mt` has a speedup gain factor of `3.2` when compared to the single-threaded MEX file, `firfilter_st`. To increase the speedup further, increase the repetition factor using the `-r` option. The tradeoff is that the output latency increases. Use a repetition factor of 3. Specify the exact state length to reduce the overhead and increase the speedup further.

```
dspunfold firfilter -args {b,noisyVal} -s 249 -f [false,true] -r 3
```

```
State length: 249 samples, Repetition: 3, Output latency: 24 frames, Threads: 4
Analyzing: firfilter.m
Creating single-threaded MEX file: firfilter_st.mexw64
Creating multi-threaded MEX file: firfilter_mt.mexw64
Creating analyzer file: firfilter_analyzer.p
```

Call the analyzer function.

```
firfilter_analyzer([b;0.5*b;0.6*b],[noisyVal;0.5*noisyVal;0.6*noisyVal]);
```

```
Analyzing multi-threaded MEX file firfilter_mt.mexw64. For best results,
please refrain from interacting with the computer and stop other processes
until the analyzer is done.
Latency = 24 frames
Speedup = 3.8x
```

The speedup gain factor is `3.8`, or approximately 32 times the speed of execution of the original simulation.

For this particular algorithm, you can see that `dspunfold` is generating a highly optimized code, without having to write any C or C++ code. The speedup gain scales with the number of cores on your host machine.

The FIR filter function in this example is only an illustrative algorithm that is easy to understand. You can apply this workflow on any of your custom algorithms. If you want to use an FIR filter, it is recommended that you use the `dsp.FIRFilter` System object in DSP System Toolbox. This object runs much faster than the benchmark numbers presented in this example, without the need for code generation.

See Also

More About

- “Multithreaded MEX File Generation” on page 1-64
- “Generate a Multi-Threaded MEX File from a MATLAB Function using DSP Unfolding”

- “Accelerated Polyphase Synthesis Filter Bank”
- “Workflow for Accelerating MATLAB Algorithms” (MATLAB Coder)
- “Accelerate MATLAB Algorithms” (MATLAB Coder)

Multithreaded MEX File Generation

This example shows how to use the `dspunfold` function to generate a multithreaded MEX file from a MATLAB function using unfolding technology. The MATLAB function can contain an algorithm which is stateless (has no states) or stateful (has states).

NOTE: The following example assumes that the current host computer has at least two physical CPU cores. The presented screenshots, speedup, and latency values were collected using a host computer with eight physical CPU cores.

Required MathWorks® products:

- DSP System Toolbox
- MATLAB Coder

Use dspunfold with a MATLAB Function Containing a Stateless Algorithm

Consider the MATLAB function `dspunfoldDCTExample`. This function computes the DCT of an input signal and returns the value and index of the maximum energy point.

```
function [peakValue,peakIndex] = dspunfoldDCTExample(x)
% Stateless MATLAB function computing the dct of a signal (e.g. audio), and
% returns the value and index of the highest energy point

% Copyright 2015 The MathWorks, Inc.

X = dct(x);
[peakValue,peakIndex] = max(abs(X));
```

To accelerate the algorithm, a common approach is to generate a MEX file using the `codegen` function. This example shows how to do so when using an input of 4096 doubles. The generated MEX file, `dspunfoldDCTExample_mex`, is singlethreaded.

```
codegen dspunfoldDCTExample -args {(1:4096)'}
```

To generate a multithreaded MEX file, use the `dspunfold` function. The argument `-s 0` indicates that the algorithm in `dspunfoldDCTExample` is stateless.

```
dspunfold dspunfoldDCTExample -args {(1:4096)'} -s 0
```

This command generates these files:

- Multithreaded MEX file `dspunfoldDCTExample_mt`
- Single-threaded MEX file `dspunfoldDCTExample_st`, which is identical to the MEX file obtained using the `codegen` function
- Self-diagnostic analyzer function `dspunfoldDCTExample_analyzer`

Additional three MATLAB files are also generated, containing the help for each of the above files.

To measure the speedup of the multithreaded MEX file relative to the single-threaded MEX file, see the example function `dspunfoldBenchmarkDCTExample`.

```
function dspunfoldBenchmarkDCTExample
% Function used to measure the speedup of the multi-threaded MEX file
% dspunfoldDCTExample_mt obtained using dspunfold vs the single-threaded MEX
```

```
% file dspunfoldDCTExample_st.

% Copyright 2015 The MathWorks, Inc.

clear dspunfoldDCTExample_mt;  % for benchmark precision purpose
numFrames = 1e5;
inputFrame = (1:4096)';

% exclude first run from timing measurements
dspunfoldDCTExample_st(inputFrame);
tic;  % measure execution time for the single-threaded MEX
for frame = 1:numFrames
    dspunfoldDCTExample_st(inputFrame);
end
timeSingleThreaded = toc;

% exclude first run from timing measurements
dspunfoldDCTExample_mt(inputFrame);
tic;  % measure execution time for the multi-threaded MEX
for frame = 1:numFrames
    dspunfoldDCTExample_mt(inputFrame);
end
timeMultiThreaded = toc;
fprintf('Speedup = %.1fx\n',timeSingleThreaded/timeMultiThreaded);
```

`dspunfoldBenchmarkDCTExample` measures the execution time taken by `dspunfoldDCTExample_st` and `dspunfoldDCTExample_mt` to process `numFrames` frames. Finally, it prints the speedup, which is the ratio between the multithreaded MEX file execution time and single-threaded MEX file execution time. Run the example.

```
dspunfoldBenchmarkDCTExample;
```

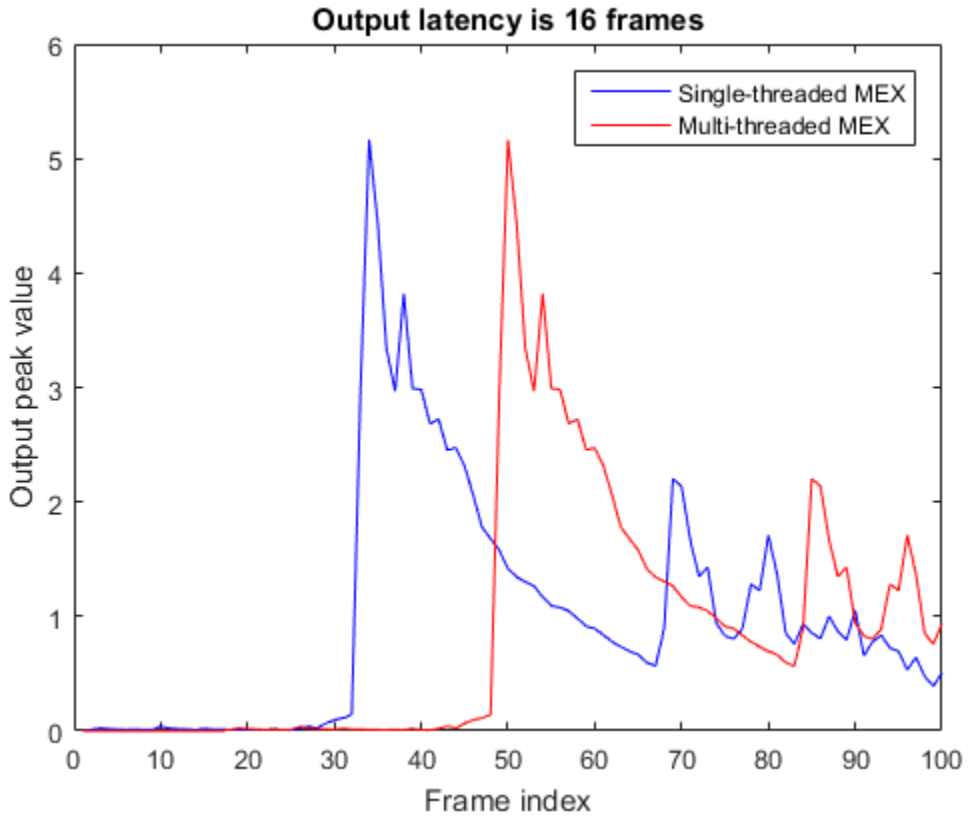
```
Speedup = 4.7x
```

To improve the speedup even more, increase the repetition value. To modify the repetition value, use the `-r` flag. For more information on the repetition value, see the `dspunfold` function reference page. For an example on how to specify the repetition value, see the section 'Using `dspunfold` with a MATLAB Function Containing a Stateful Algorithm'.

`dspunfold` generates a multithreaded MEX file, which buffers multiple signal frames and then processes these frames simultaneously, using multiple cores. This process introduces some deterministic output latency. Executing `help dspunfoldDCTExample_mt` displays more information about the multithreaded MEX file, including the value of the output latency. For this example, the output of the multithreaded MEX file has a latency of 16 frames relative to its input, which is not the case for the single-threaded MEX file.

Run `dspunfoldShowLatencyDCTExample` example. The generated plot displays the outputs of the single-threaded and multithreaded MEX files. Notice that the output of the multithreaded MEX is delayed by 16 frames, relative to that of the single-threaded MEX.

```
dspunfoldShowLatencyDCTExample;
```



Using dspunfold with a MATLAB Function Containing a Stateful Algorithm

The MATLAB function `dspunfoldFIRExample` executes two FIR filters.

```
type dspunfoldFIRExample
```

```
function y = dspunfoldFIRExample(u,c1,c2)
% Stateful MATLAB function executing two FIR filters

% Copyright 2015 The MathWorks, Inc.

persistent FIRSTFIR SECONDFIR
if isempty(FIRSTFIR)
    FIRSTFIR = dsp.FIRFilter('NumeratorSource','Input port');
    SECONDFIR = dsp.FIRFilter('NumeratorSource','Input port');
end
t = FIRSTFIR(u,c1);
y = SECONDFIR(t,c2);
```

To build the multithreaded MEX file, you must provide the state length corresponding to the two FIR filters. Specify 1s to indicate that the state length does not exceed 1 frame.

```
firCoeffs1 = fir1(127,0.8);
firCoeffs2 = fir1(256,0.2,'High');
dspunfold dspunfoldFIRExample -args {(1:2048)',firCoeffs1,firCoeffs2} -s 1
```

Executing this code generates:

- Multithreaded MEX file `dspunfoldFIRExample_mt`
- Single-threaded MEX file `dspunfoldFIRExample_st`
- Self-diagnostic analyzer function `dspunfoldFIRExample_analyzer`
- The corresponding MATLAB help files for these three files

The output latency of the multithreaded MEX file is 16 frames. To measure the speedup, execute `dspunfoldBenchmarkFIRExample`.

```
dspunfoldBenchmarkFIRExample;
```

```
Speedup = 3.9x
```

To improve the speedup of the multithreaded MEX file even more, specify the exact state length in samples. To do so, you must specify which input arguments to `dspunfoldFIRExample` are frames. In this example, the first input is a frame because the elements of this input are sequenced in time. Therefore it can be further divided into subframes. The last two inputs are not frames because the FIR filters coefficients cannot be subdivided without changing the nature of the algorithm. The value of the `dspunfoldFIRExample` MATLAB function state length is the sum of the state length of the two FIR filters (127 + 256 = 383). Using the `-f` argument, mark the first input argument as true (frame), and the last two input arguments as false (nonframes)

```
dspunfold dspunfoldFIRExample -args {(1:2048)',firCoeffs1,firCoeffs2} -s 383 -f [true,false,false]
```

Again, measure the speedup for the resulting multithreaded MEX using the `dspunfoldBenchmarkFIRExample` function. Notice that the speedup increased because the exact state length was specified in samples, and `dspunfold` was able to subdivide the frame inputs.

```
dspunfoldBenchmarkFIRExample;
```

```
Speedup = 6.3x
```

Oftentimes, the speedup can be increased even more by increasing the repetition (`-r`) provided when invoking `dspunfold`. The default repetition value is 1. When you increase this value, the multithreaded MEX buffers more frames internally before the processing starts. Increasing the repetition factor increases the efficiency of the multi-threading, but at the cost of a higher output latency.

```
dspunfold dspunfoldFIRExample -args {(1:2048)',firCoeffs1,firCoeffs2} ...
-s 383 -f [true,false,false] -r 5
```

Again, measure the speedup for the resulting multithreaded MEX, using the `dspunfoldBenchmarkFIRExample` function. Speedup increases again, but the output latency is now 80 frames. The general output latency formula is `2*Threads*Repetition` frames. In these examples, the number of `Threads` is equal to the number of physical CPU cores.

```
dspunfoldBenchmarkFIRExample;
```

```
Speedup = 7.7x
```

Detecting State Length Automatically

To request that `dspunfold` autodetect the state length, specify `-s auto`. This option generates an efficient multithreaded MEX file, but with a significant increase in the generation time, due to the extra analysis that it requires.

```
dspunfold dspunfoldFIRExample -args {(1:2048)',firCoeffs1,firCoeffs2} ...
-s auto -f [true,false,false] -r 5

State length: [autodetect] samples, Repetition: 5, Output latency: 40 frames, Threads: 4
Analyzing: dspunfoldFIRExample.m
Creating single-threaded MEX file: dspunfoldFIRExample_st.mexw64
Searching for minimal state length (this might take a while)
Checking stateless ... Insufficient
Checking 2048 samples ... Sufficient
Checking 1024 samples ... Sufficient
Checking 512 samples ... Sufficient
Checking 256 samples ... Insufficient
Checking 384 samples ... Sufficient
Checking 320 samples ... Insufficient
Checking 352 samples ... Insufficient
Checking 368 samples ... Insufficient
Checking 376 samples ... Insufficient
Checking 380 samples ... Insufficient
Checking 382 samples ... Insufficient
Checking 383 samples ... Sufficient
Minimal state length is 383 samples
Creating multi-threaded MEX file: dspunfoldFIRExample_mt.mexw64
Creating analyzer file: dspunfoldFIRExample_analyzer.p
```

`dspunfold` checks different state lengths, using as inputs the values provided with the `-args` option. The function aims to find the minimum state length for which the outputs of the multithreaded MEX and single-threaded MEX are the same. Notice that it found 383, as the minimal state length value, which matches the expected value, manually computed before.

Verify Generated Multithreaded MEX Using the Generated Analyzer

When creating a multithreaded MEX file using `dspunfold`, the single-threaded MEX file is also created along with an analyzer function. For the stateful example in the previous section, the name of the analyzer is `dspunfoldFIRExample_analyzer`.

The goal of the analyzer is to provide a quick way to measure the speedup of the multithreaded MEX relative to the single-threaded MEX, and also to check if the outputs of the multithreaded MEX and single-threaded MEX match. Outputs usually do not match when an incorrect state length value is specified.

Execute the analyzer for the multithreaded MEX file, `dspunfoldFIRExample_mt`, generated previously using the `-s auto` option.

```
firCoeffs1_1 = fir1(127,0.8);
firCoeffs1_2 = fir1(127,0.7);
firCoeffs1_3 = fir1(127,0.6);
firCoeffs2_1 = fir1(256,0.2,'High');
firCoeffs2_2 = fir1(256,0.1,'High');
firCoeffs2_3 = fir1(256,0.3,'High');
dspunfoldFIRExample_analyzer((1:2048*3)',[firCoeffs1_1;firCoeffs1_2;firCoeffs1_3],...
[firCoeffs2_1;firCoeffs2_2;firCoeffs2_3]);
```

```
Analyzing multi-threaded MEX file dspunfoldFIRExample_mt.mexw64  ...
Latency = 80 frames
Speedup = 7.8x
```

Each input to the analyzer corresponds to the inputs of the `dspunfoldFIRExample_mt` MEX file. Notice that the length (first dimension) of each input is greater than the expected length. For example, `dspunfoldFIRExample_mt` expects a frame of 2048 doubles for its first input, while 2048*3 samples were provided to `dspunfoldFIRExample_analyzer`. The analyzer interprets this input as 3 frames of 2048 samples. The analyzer alternates between these 3 input frames circularly while checking if the outputs of the multithreaded and single-threaded MEX files match.

The table shows the inputs used by the analyzer at each step of the numerical check. The total number of steps invoked by the analyzer is 240 or `3*latency`, where `latency` is 80 in this case.

	Input 1	Input 2	Input 3
Step 1	`(1:2048)'`	`firCoeffs1_1`	`firCoeffs2_1`
Step 2	`(2049:4096)'`	`firCoeffs1_2`	`firCoeffs2_2`
Step 3	`(4097:6144)'`	`firCoeffs1_3`	`firCoeffs2_3`
Step 4	`(1:2048)'`	`firCoeffs1_1`	`firCoeffs2_1`
...	...	...	...

NOTE: For the analyzer to correctly check for the numerical match between the multithreaded MEX and single-threaded MEX, provide at least two frames with different values for each input. For inputs that represent parameters, such as filter coefficients, the frames can have the same values for each input. In this example, you could have specified a single set of coefficients for the second and third inputs.

References

[1] Unfolding (DSP implementation)

See Also

“Generate a Multi-Threaded MEX File from a MATLAB Function using DSP Unfolding” | “Workflow for Generating a Multithreaded MEX File using dspunfold” on page 18-45 | “Why Does the Analyzer Choose the Wrong State Length?” on page 18-49 | “How Is dspunfold Different from parfor?” on page 18-43 | `dspunfold`

Fixed-Point Filter Design in MATLAB

This example shows how to design filters for use with fixed-point input. The example analyzes the effect of coefficient quantization on filter design. You must have the Fixed-Point Designer software™ to run this example.

Introduction

Fixed-point filters are commonly used in digital signal processors where data storage and power consumption are key limiting factors. With the constraints you specify, DSP System Toolbox software allows you to design efficient fixed-point filters. The filter for this example is a lowpass equiripple FIR filter. Design the filter first for floating-point input to obtain a baseline. You can use this baseline for comparison with the fixed-point filter.

FIR Filter Design

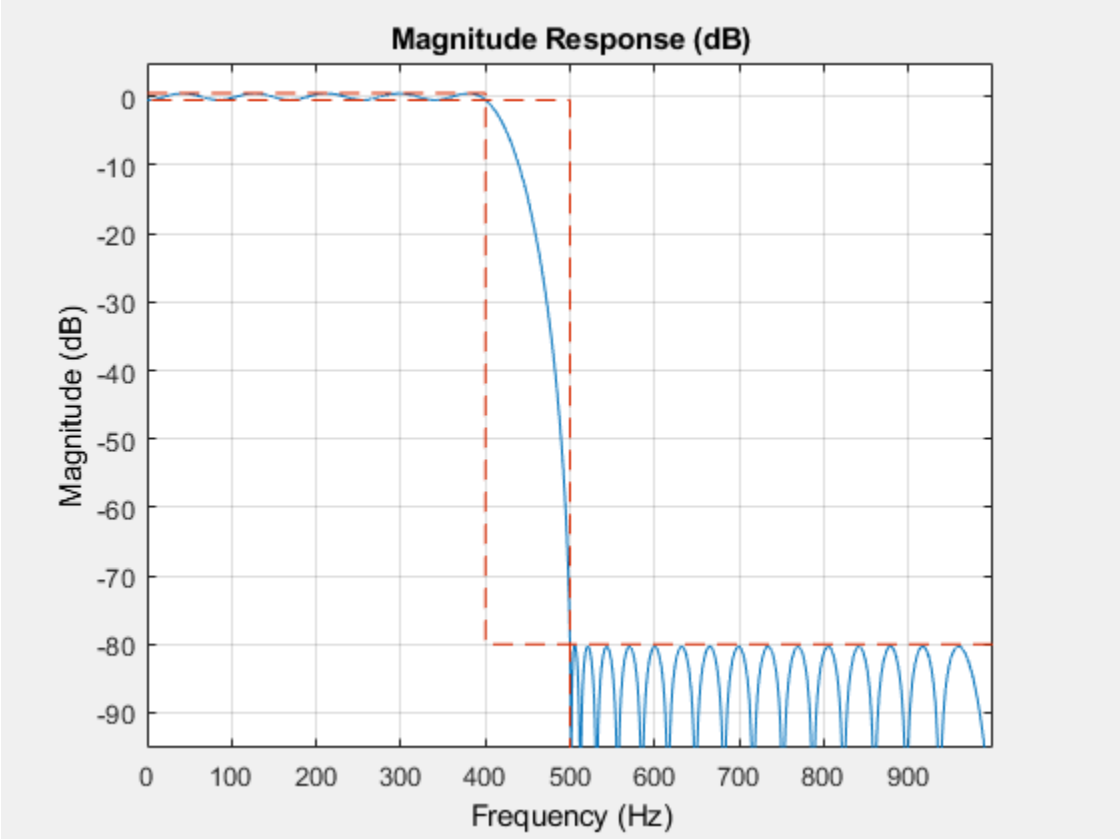
The lowpass FIR filter has the following specifications:

- Sample rate: 2000 Hz
- Center frequency: 450 Hz
- Transition width: 100 Hz
- Equiripple design
- Maximum 1 dB of ripple in the passband
- Minimum 80 dB of attenuation in the stopband

```
samplingFrequency = 2000;
centerFrequency = 450;
transitionWidth = 100;
passbandRipple = 1;
stopbandAttenuation = 80;

designSpec = fdesign.lowpass('Fp,Fst,Ap,Ast',...
    centerFrequency-transitionWidth/2, ...
    centerFrequency+transitionWidth/2, ...
    passbandRipple,stopbandAttenuation, ...
    samplingFrequency);
LPF = design(designSpec,'equiripple','SystemObject',true)

LPF =
  dsp.FIRFilter with properties:

            Structure: 'Direct form'
      NumeratorSource: 'Property'
            Numerator: [1x52 double]
    InitialConditions: 0

  Show all properties
```

View the baseline frequency response. The dotted red lines show the design specifications used to create the filter.

```
fvtool(LPF)
```



Full-Precision Fixed-Point Operation

The fixed-point properties of the filter are contained in the `Fixed-point properties` section in the display of the object. By default, the filter uses full-precision arithmetic to deal with fixed-point inputs. With full-precision arithmetic, the filter uses as many bits for the product, accumulator, and output as needed to prevent any overflow or rounding. If you do not want to use full-precision arithmetic, you can set the `FullPrecisionOverride` property to `false` and then set the product, accumulator, and output data types independently.

```
rng default
inputWordLength = 16;
fixedPointInput = fi(randn(100,1),true,inputWordLength);
floatingPointInput = double(fixedPointInput);
floatingPointOutput = LPF(floatingPointInput);

release(LPF)
fullPrecisionOutput = LPF(fixedPointInput);
norm(floatingPointOutput-double(fullPrecisionOutput),'inf')
```

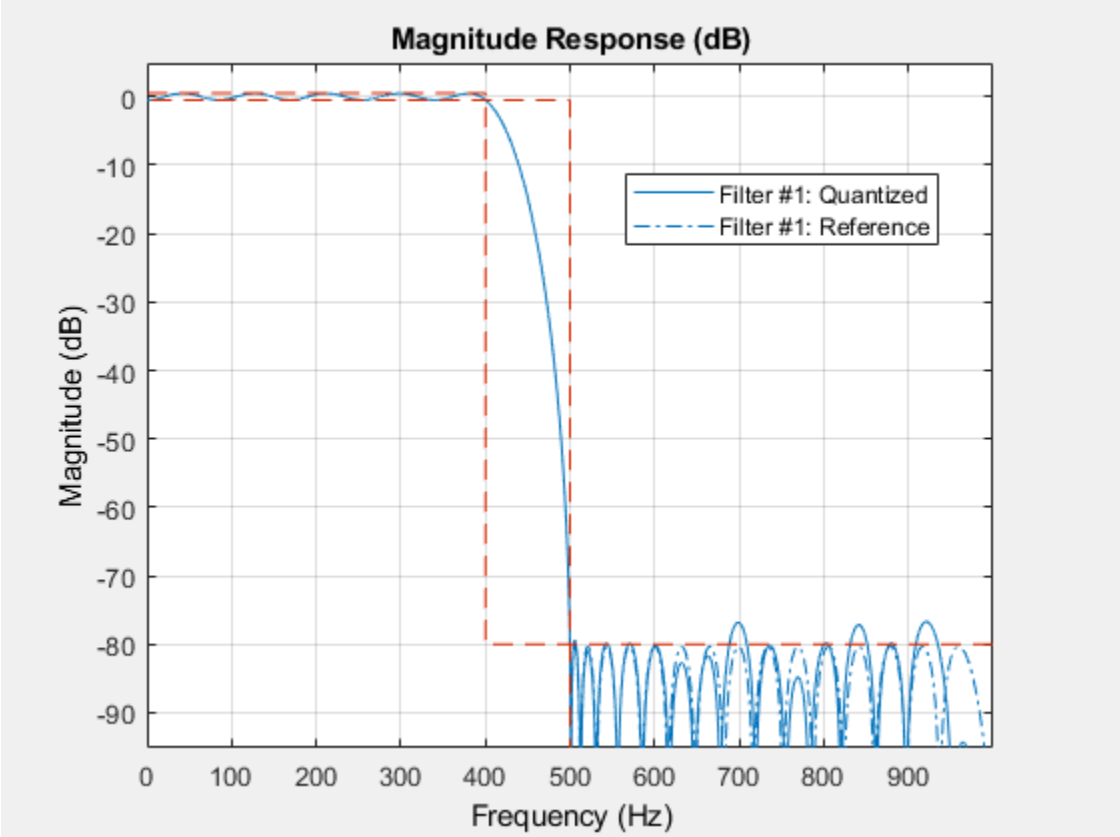
```
ans = 6.8994e-05
```

The result of full-precision fixed-point filtering comes very close to floating point, but the results are not exact. The reason for this is coefficient quantization. In the fixed-point filter, the `CoefficientsDataType` property has the same word length (16) for the coefficients and the input. The frequency response of the filter in full-precision mode shows this more clearly. The `measure` function shows that the minimum stopband attenuation of this filter with quantized coefficients is 76.6913 dB, less than the 80 dB specified for the floating-point filter.

```
LPF.CoefficientsDataType
```

```
ans =
'Same word length as input'
```

```
fvtool(LPF)
```




```
measure(LPF)
```

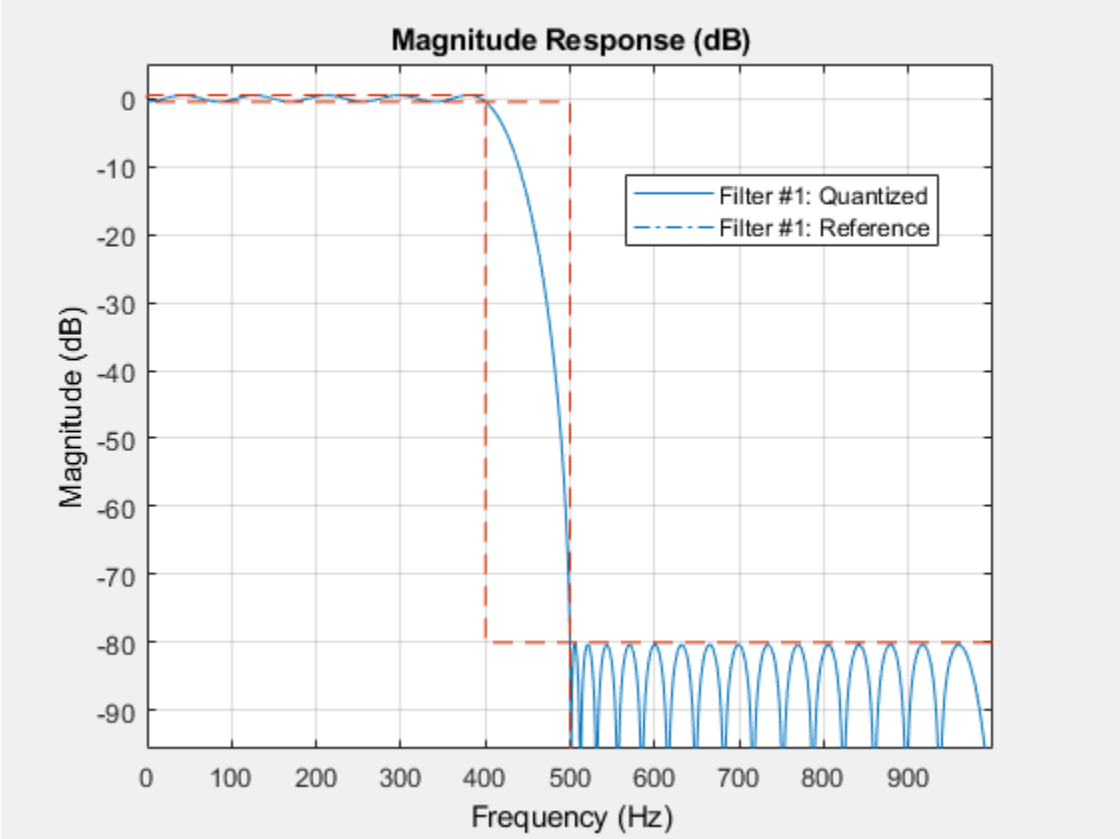
```
ans =
Sample Rate      : 2 kHz
Passband Edge    : 400 Hz
3-dB Point       : 416.2891 Hz
6-dB Point       : 428.1081 Hz
Stopband Edge    : 500 Hz
Passband Ripple  : 0.96325 dB
Stopband Atten.  : 76.6913 dB
Transition Width : 100 Hz
```

The filter was last used with fixed-point input and is still in a locked state. For that reason, `fvtool` displays the fixed-point frequency response. The dash-dot response is that of the reference floating-point filter, and the solid plot is the response of the filter that was used with fixed-point input. The desired frequency response cannot be matched because the coefficient word length has been restricted to 16 bits. This accounts for the difference between the floating-point and fixed-point designs. Increasing the number of bits allowed for the coefficient word length makes the quantization error smaller and enables you to match the design requirement for 80 dB of stopband attenuation. Use a coefficient word length of 24 bits to achieve an attenuation of 80.1275 dB.

```
LPF24bitCoeff = design(designSpec,'equiripple','SystemObject',true);
LPF24bitCoeff.CoefficientsDataType = 'Custom';
coeffNumerictype = numerictype(fi(LPF24bitCoeff.Numerator,true,24));
LPF24bitCoeff.CustomCoefficientsDataType = numerictype(true, ...
            coeffNumerictype.WordLength,coeffNumerictype.FractionLength);
fullPrecisionOutput32bitCoeff = LPF24bitCoeff(fixedPointInput);
norm(floatingPointOutput-double(fullPrecisionOutput32bitCoeff),'inf')
```

```
ans = 4.1077e-07
```

```
fvtool(LPF24bitCoeff)
```




```
measure(LPF24bitCoeff)
```

```
ans =
Sample Rate      : 2 kHz
Passband Edge    : 400 Hz
3-dB Point       : 416.2901 Hz
6-dB Point       : 428.1091 Hz
Stopband Edge    : 500 Hz
Passband Ripple  : 0.96329 dB
Stopband Atten.  : 80.1275 dB
Transition Width : 100 Hz
```

Design Parameters and Coefficient Quantization

In many fixed-point design applications, the coefficient word length is not flexible. For example, supposed you are restricted to work with 14 bits. In such cases, the requested minimum stopband attenuation of 80 dB cannot be reached. A filter with 14-bit coefficient quantization can achieve a minimum attenuation of only 67.2987 dB.

```
LPF14bitCoeff = design(designSpec,'equiripple','SystemObject',true);
coeffNumerictype = numerictype(fi(LPF14bitCoeff.Numerator,true,14));
LPF14bitCoeff.CoefficientsDataType = 'Custom';
LPF14bitCoeff.CustomCoefficientsDataType = numerictype(true, ...
            coeffNumerictype.WordLength,coeffNumerictype.FractionLength);
measure(LPF14bitCoeff,'Arithmetic','fixed')
```

```
ans =
Sample Rate      : 2 kHz
Passband Edge    : 400 Hz
3-dB Point       : 416.2939 Hz
6-dB Point       : 428.1081 Hz
Stopband Edge    : 500 Hz
Passband Ripple  : 0.96405 dB
Stopband Atten.  : 67.2987 dB
Transition Width : 100 Hz
```

For FIR filters in general, each bit of coefficient word length provides approximately 5 dB of stopband attenuation. Accordingly, if your filter's coefficients are always quantized to 14 bits, you can expect the minimum stopband attenuation to be only around 70 dB. In such cases, it is more practical to design the filter with stopband attenuation less than 70 dB. Relaxing this requirement results in a design of lower order.

```
designSpec.Astop = 60;
LPF60dBStopband = design(designSpec,'equiripple','SystemObject',true);
LPF60dBStopband.CoefficientsDataType = 'Custom';
coeffNumerictype = numerictype(fi(LPF60dBStopband.Numerator,true,14));
LPF60dBStopband.CustomCoefficientsDataType = numerictype(true, ...
            coeffNumerictype.WordLength,coeffNumerictype.FractionLength);
measure(LPF60dBStopband,'Arithmetic','fixed')
```

```
ans =
Sample Rate      : 2 kHz
Passband Edge    : 400 Hz
3-dB Point       : 419.3391 Hz
6-dB Point       : 432.9718 Hz
Stopband Edge    : 500 Hz
Passband Ripple  : 0.92801 dB
Stopband Atten.  : 59.1829 dB
Transition Width : 100 Hz
```

```
order(LPF14bitCoeff)
```

```
ans = 51
```

```
order(LPF60dBStopband)
```

```
ans = 42
```

The filter order decreases from 51 to 42, implying that fewer taps are required to implement the new FIR filter. If you still want a high minimum stopband attenuation without compromising on the number of bits for coefficients, you must relax the other filter design constraint: the transition width. Increasing the transition width might enable you to get higher attenuation with the same coefficient word length. However, it is almost impossible to achieve more than 5 dB per bit of coefficient word length, even after relaxing the transition width.

```
designSpec.Astop = 80;
transitionWidth = 200;
designSpec.Fpass = centerFrequency-transitionWidth/2;
designSpec.Fstop = centerFrequency+transitionWidth/2;
LPF300TransitionWidth = design(designSpec,'equiripple', ...
                               'SystemObject',true);
LPF300TransitionWidth.CoefficientsDataType = 'Custom';
coeffNumerictype = numerictype(fi(LPF300TransitionWidth.Numerator, ...
                                  true, 14));
LPF300TransitionWidth.CustomCoefficientsDataType = numerictype(true, ...
           coeffNumerictype.WordLength,coeffNumerictype.FractionLength);
measure(LPF300TransitionWidth,'Arithmetic','fixed')
```

```
ans =
Sample Rate      : 2 kHz
Passband Edge    : 350 Hz
3-dB Point       : 385.4095 Hz
6-dB Point       : 408.6465 Hz
Stopband Edge    : 550 Hz
Passband Ripple  : 0.74045 dB
Stopband Atten.  : 74.439 dB
Transition Width : 200 Hz
```

As you can see, increasing the transition width to 200 Hz allows 74.439 dB of stopband attenuation with 14-bit coefficients, compared to the 67.2987 dB attained when the transition width was set to 100 Hz. An added benefit of increasing the transition width is that the filter order also decreases, in this case from 51 to 27.

```
order(LPF300TransitionWidth)
```

```
ans = 27
```

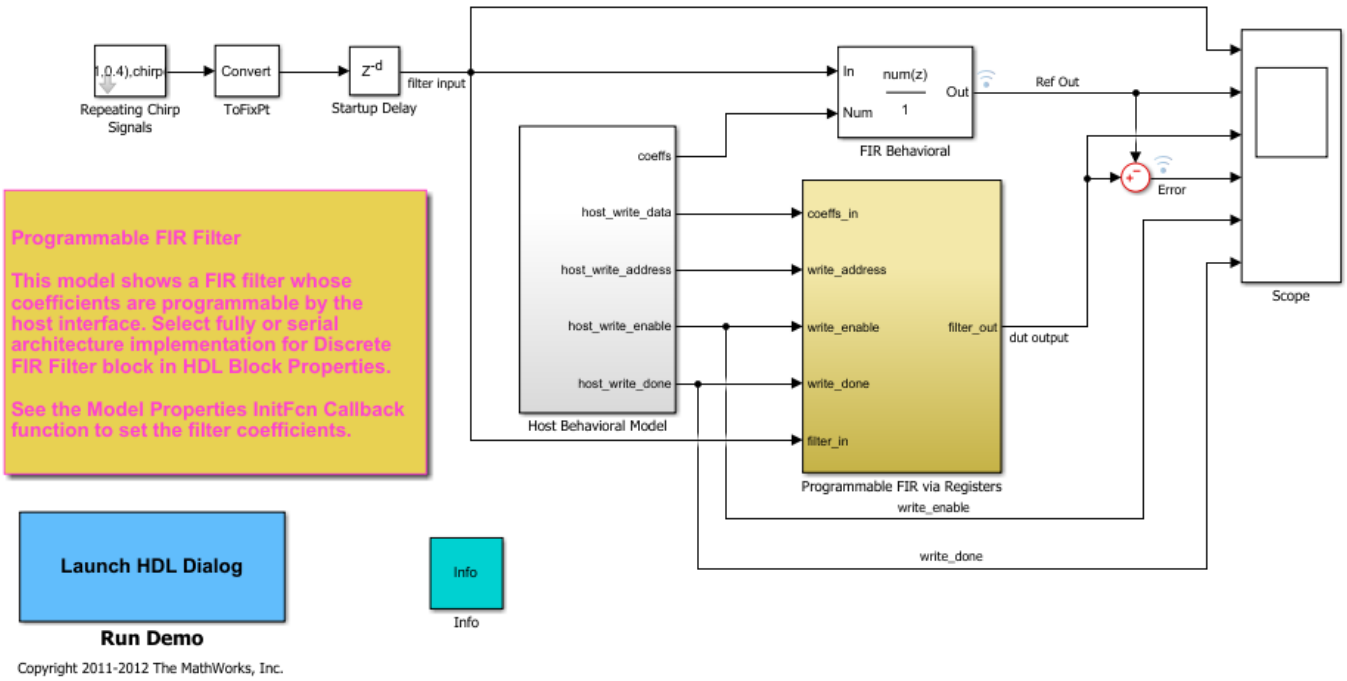
Visualizing Multiple Signals Using Logic Analyzer

Visualize multiple signals of a programmable FIR filter by using a logic analyzer. For more information on the model used in this example and how to configure the model to generate HDL code, see “Programmable FIR Filter for FPGA”.

Model Programmable FIR Filter

Open the example model.

```
modelname = 'dspprogfirhdl';
open_system(modelname);
```

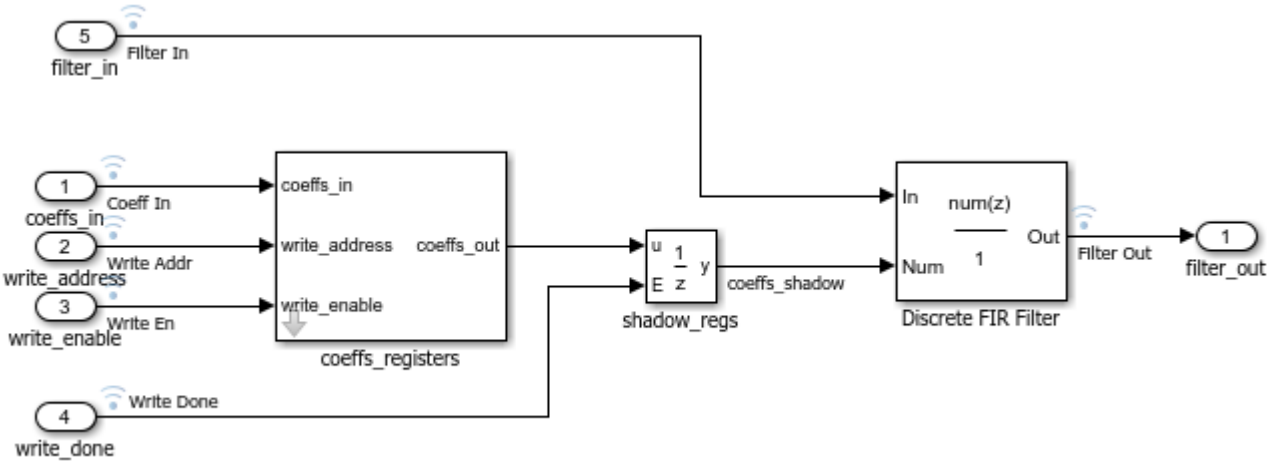


Consider two FIR filters, one with a lowpass response and the other with a highpass response. The coefficients can be specified using the **InitFcn*** callback function. To specify the callback, select **File > Model Properties > Model Properties**. In the dialog box, in the **Callbacks** tab, select **InitFcn***.

The Programmable FIR via Registers block loads the lowpass coefficients from the Host Behavioral Model block and processes the input chirp samples first. The block then loads the highpass coefficients and processes the same chirp samples again.

Open the Programmable FIR via Registers block.

```
systemname = [modelname '/Programmable FIR via Registers'];
open_system(systemname);
```



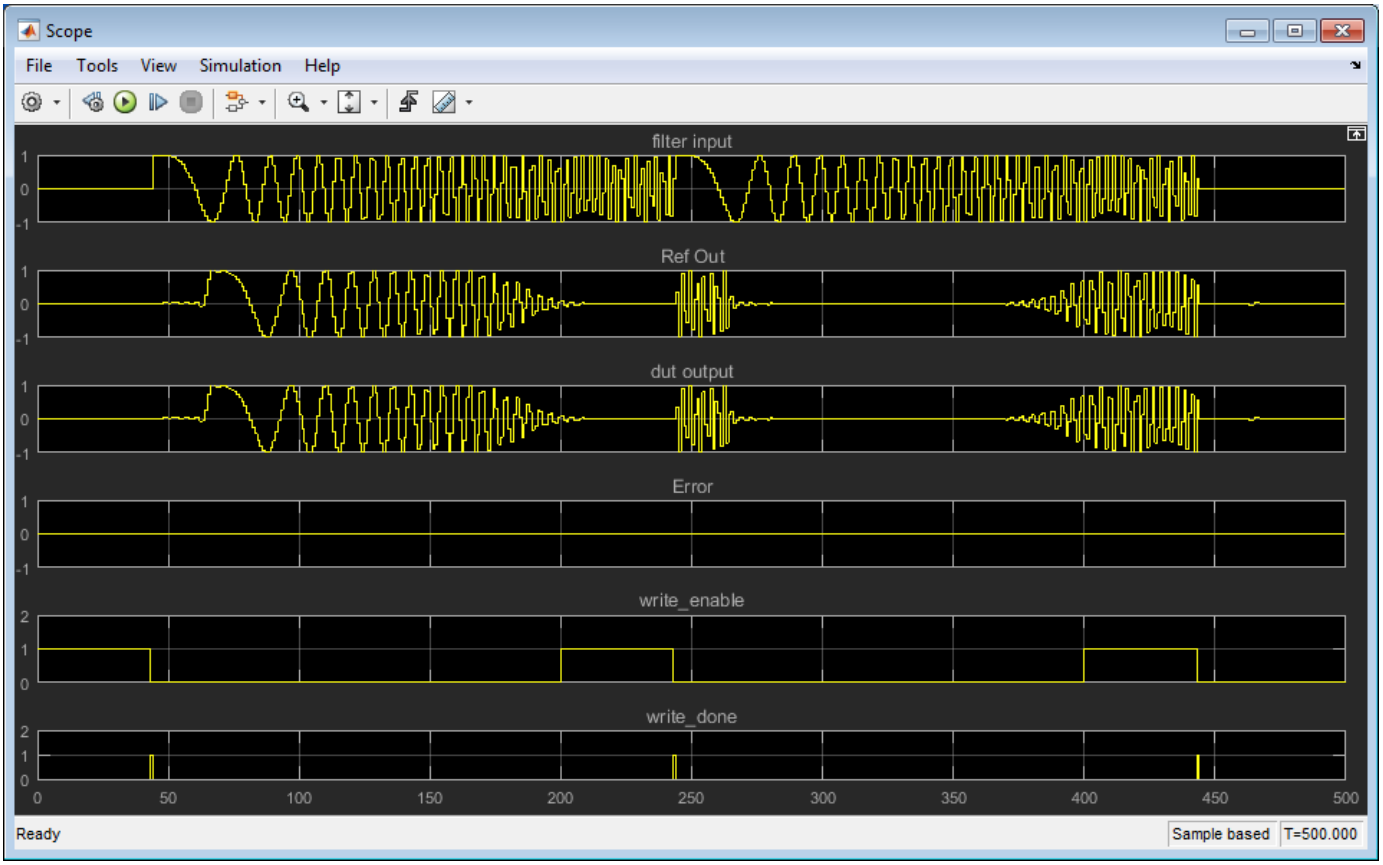
Simulation

Run the example model.

```
sim(modelname)
```

Open the scope.

```
open_system([modelname '/Scope']);
```

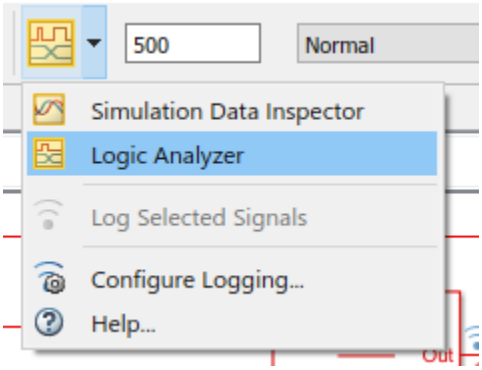


Compare the DUT (Design under Test) output with the reference output.

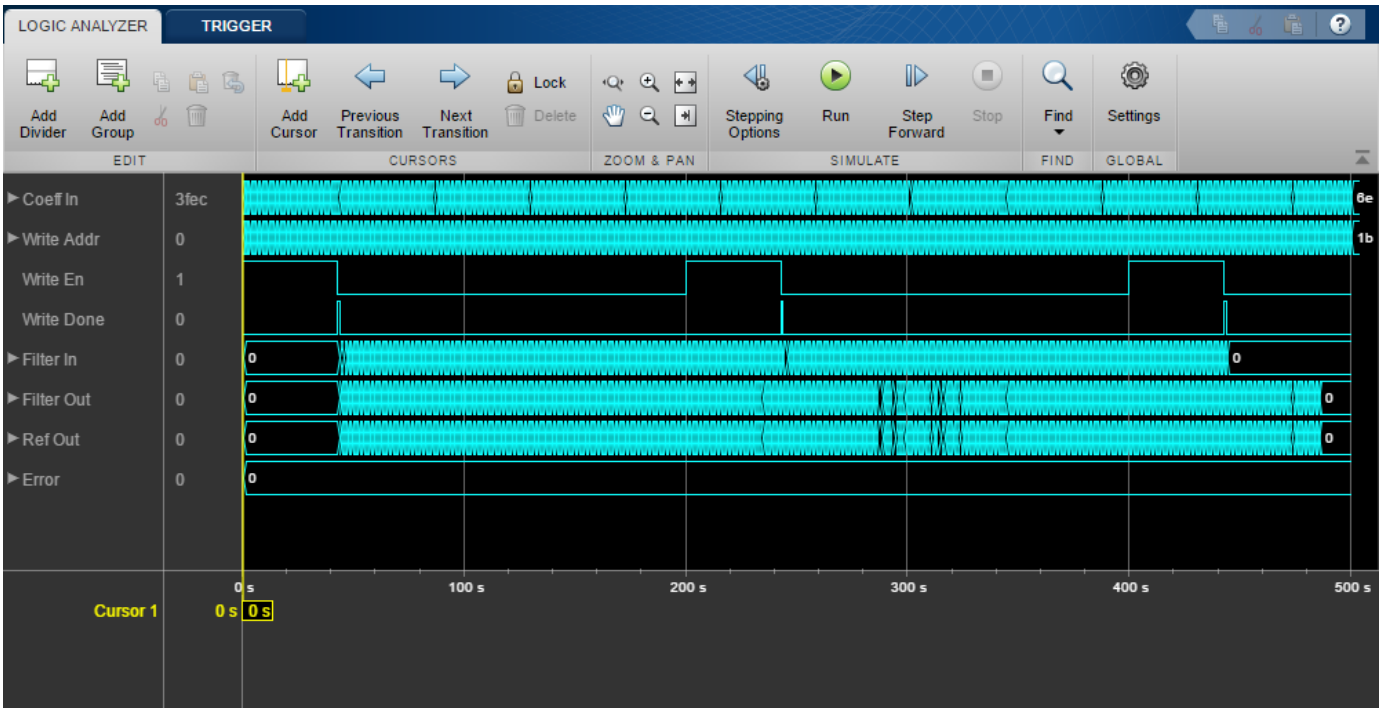
Use the Logic Analyzer

The Logic Analyzer enables you to view multiple signals in one window. It also makes it easy to detect signal transitions.

The signals of interest (input coefficient, write address, write enable, write done, filter in, filter out, reference out, and error) have been marked for streaming in the model. Click the streaming button in the toolbar and select **Logic Analyzer**.

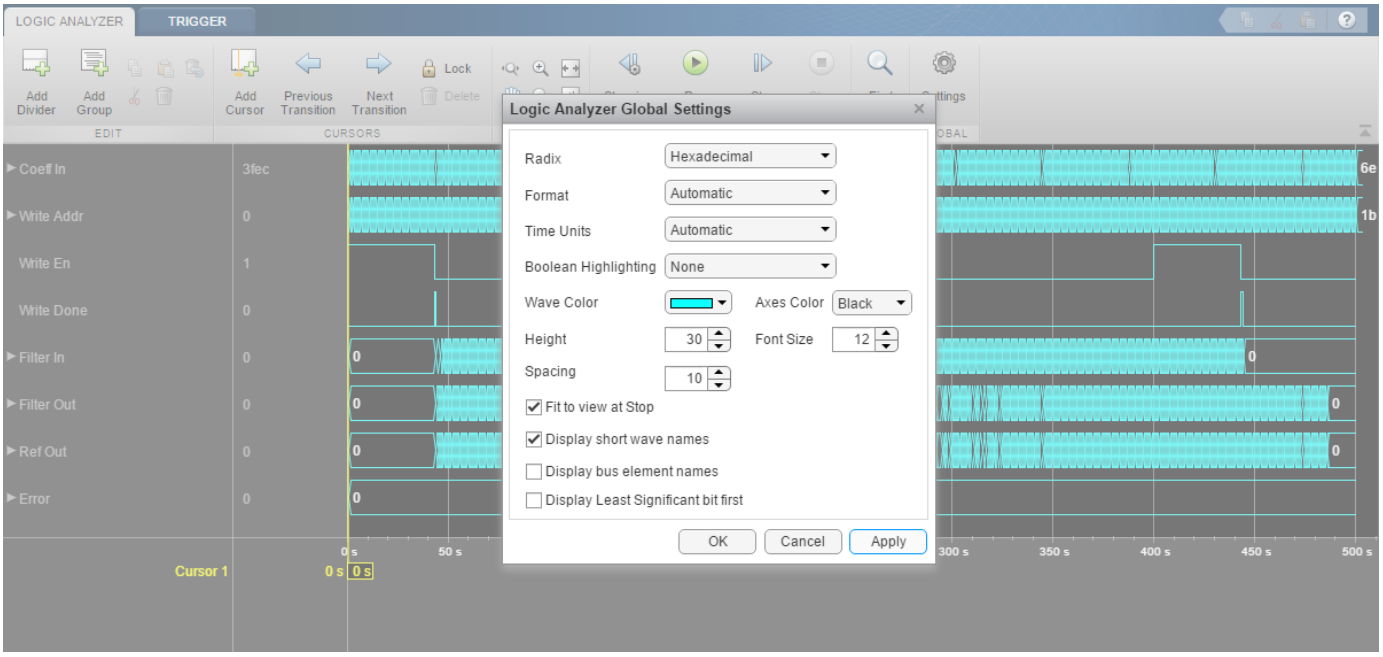


The Logic Analyzer displays waveforms of the selected signals.

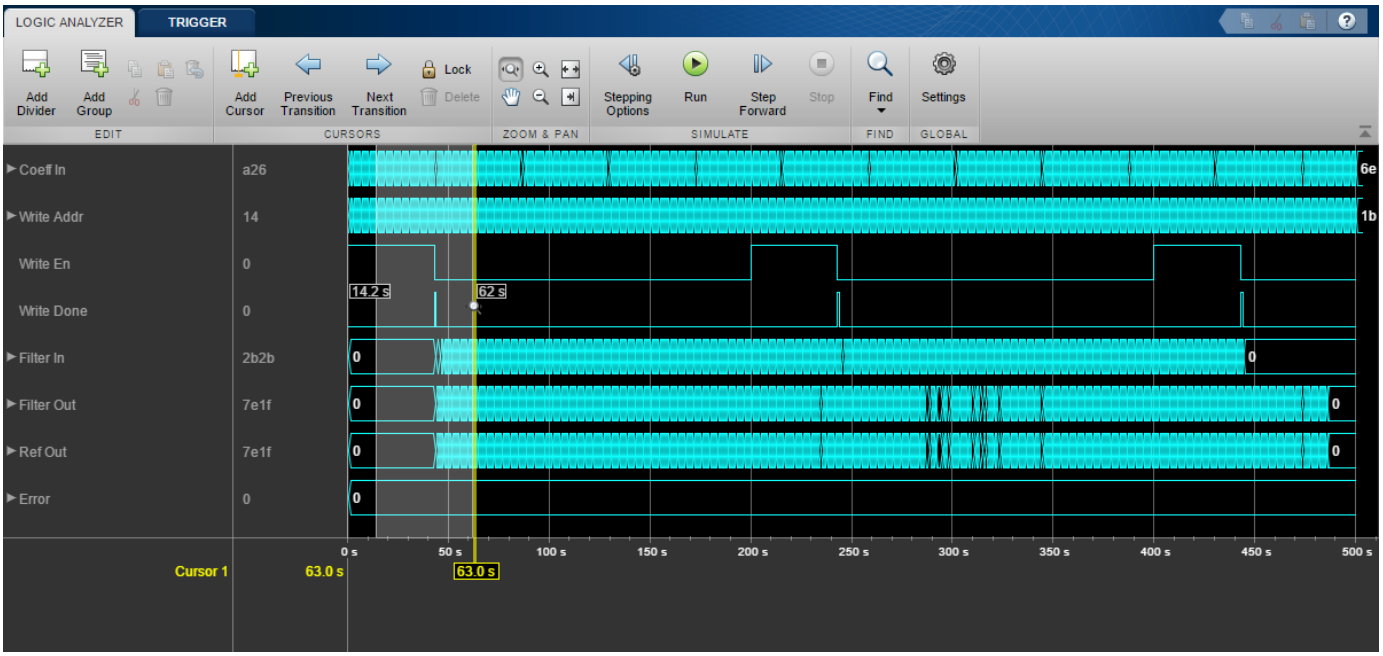


Modify the Display

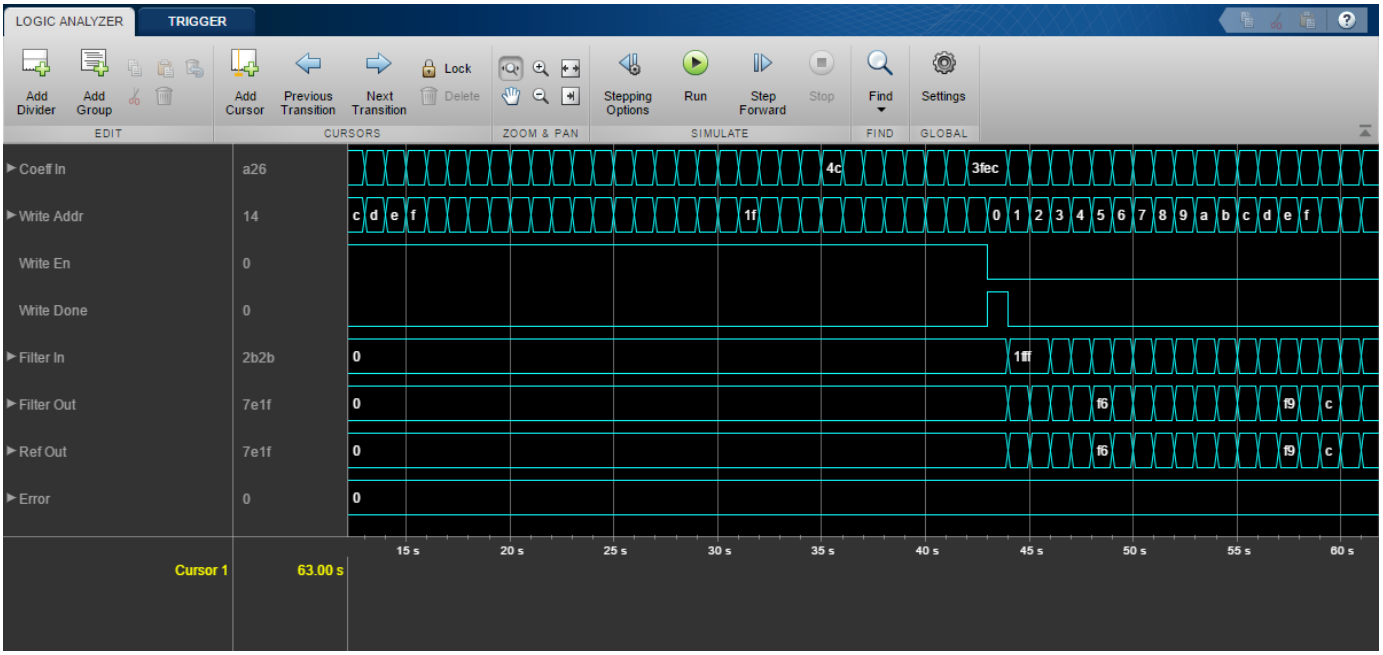
In the Logic Analyzer, you can modify the height of all the displayed channels, and the spacing between the channels. Click the **Settings** button. Then, modify the default height and spacing for each wave. Click **Apply** to show the new dimensions in the background.



To zoom in on the waveform, click the **Zoom In Time** button in the **ZOOM & PAN** section of the toolbar. Your cursor becomes a magnifying glass. Then click and drag to select an area on the waveform.

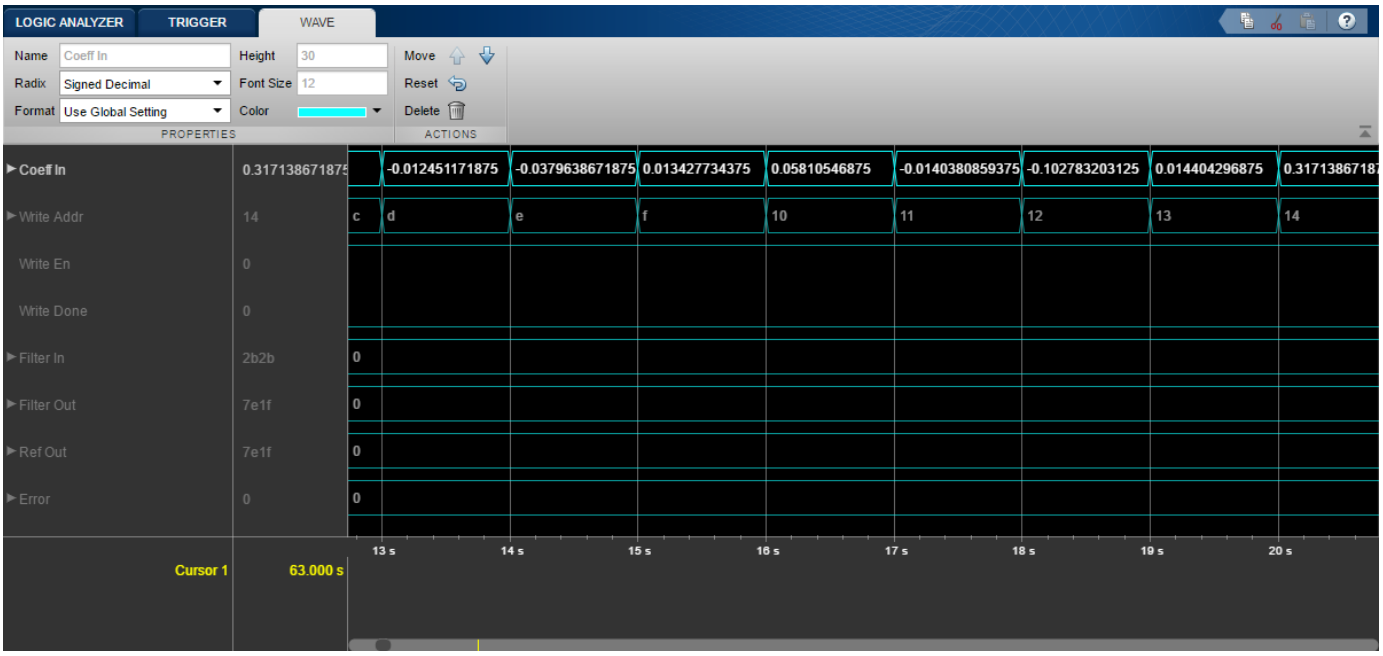


The Logic Analyzer now displays the time span you selected.

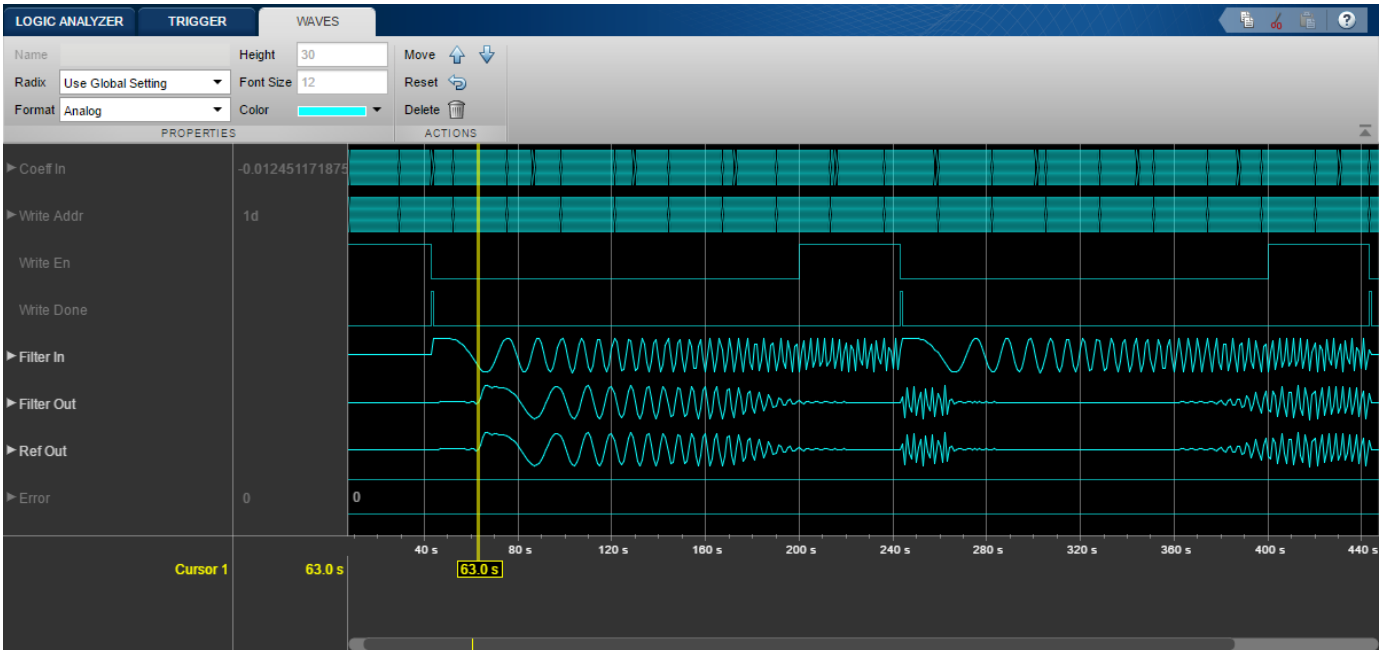


You can also control the display on a per-waveform basis. To modify an individual waveform, double-click the signal, select the signal, then click the **WAVE** tab to modify its settings.

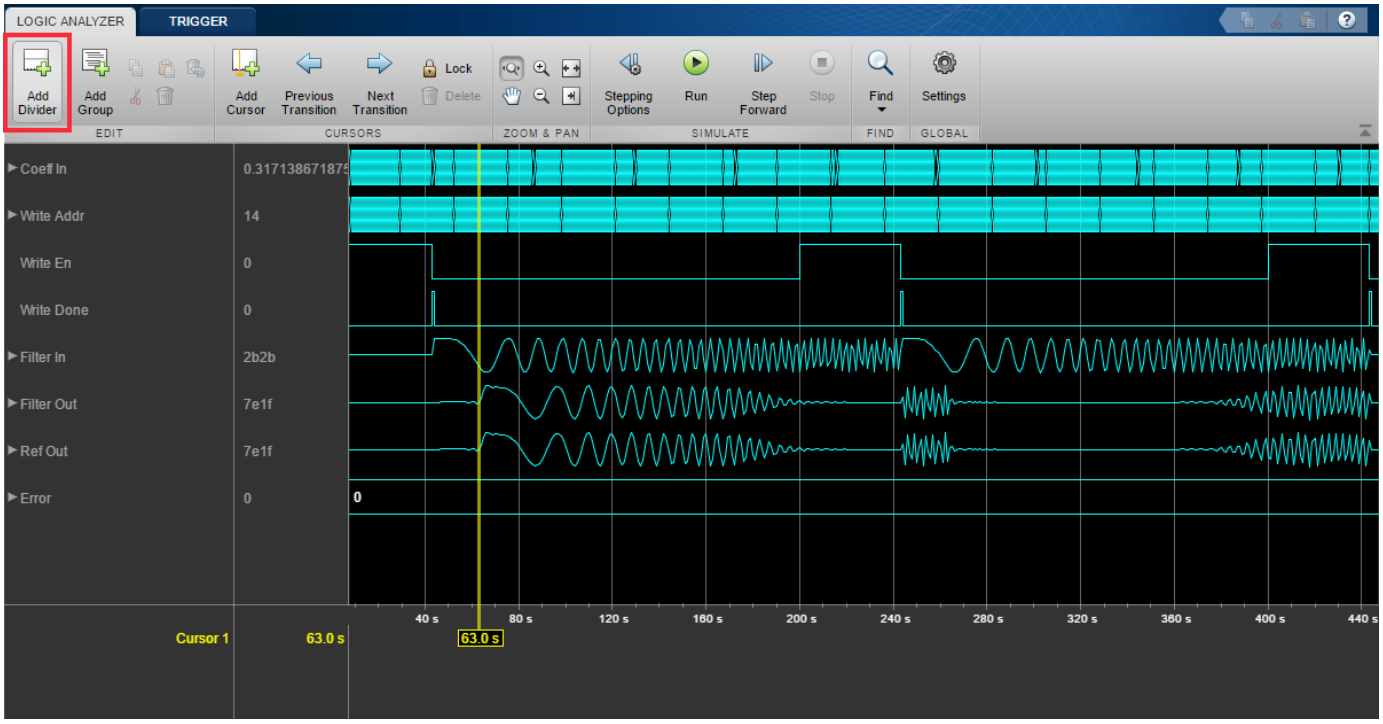
Display the `CoeffIn` signal in signed decimal mode. The conversion uses the fractional and integer bits as defined for this signal in your model.



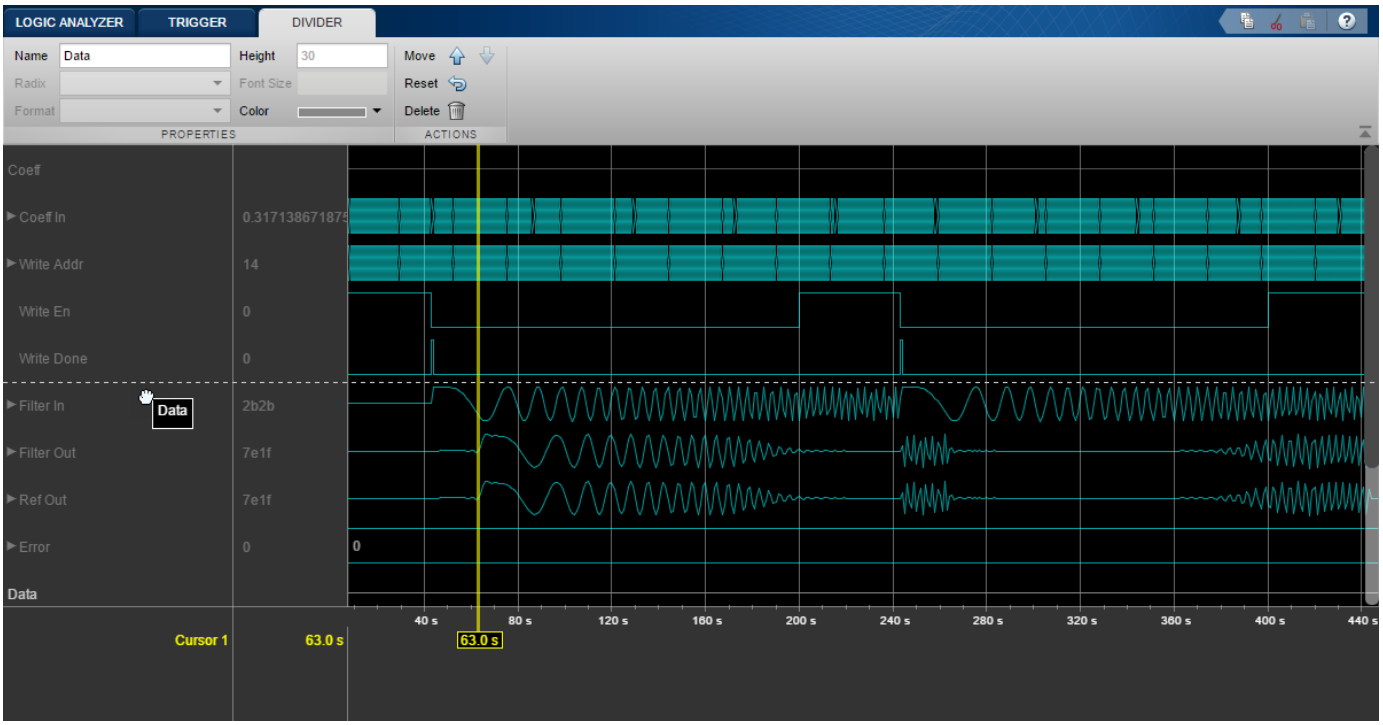
Another useful mode of visualization in the Logic Analyzer is the analog format. View the `Filter In`, `Filter Out`, and `Ref Out` signals in analog format.



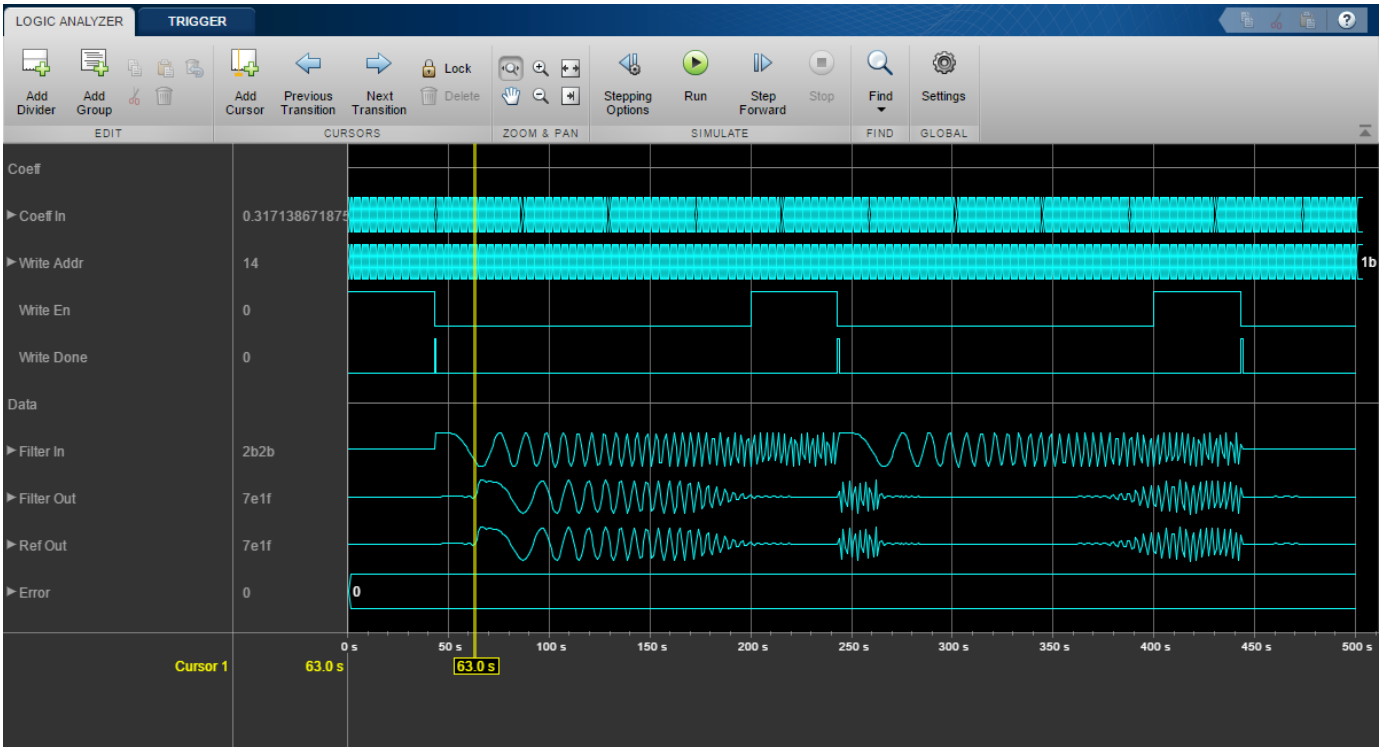
You can also add dividers to the display. Click the **Add Divider** button in the toolbar.



Then specify a name for your divider on the **DIVIDER** tab. Add a second divider. A divider is added underneath the selected wave. If no wave is selected, it is added at the bottom of the display. To move the divider, click on the divider name and drag it to a new position. Alternatively, use the **Move** arrows on the **DIVIDER** tab.



Note the divider in its new position.



For more instructions on using the waveform display tool, see **Logic Analyzer**.

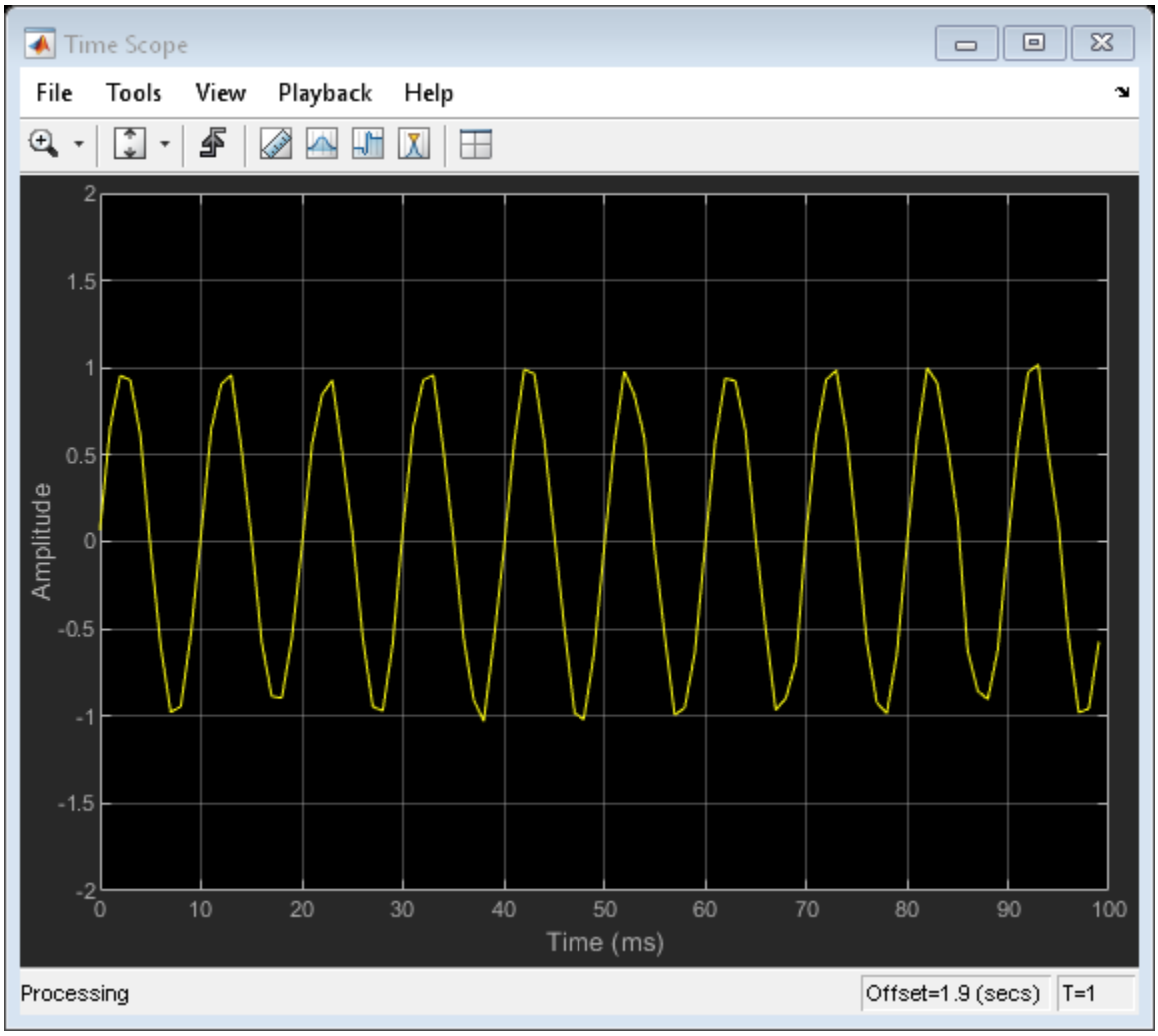
Signal Visualization and Measurements in MATLAB

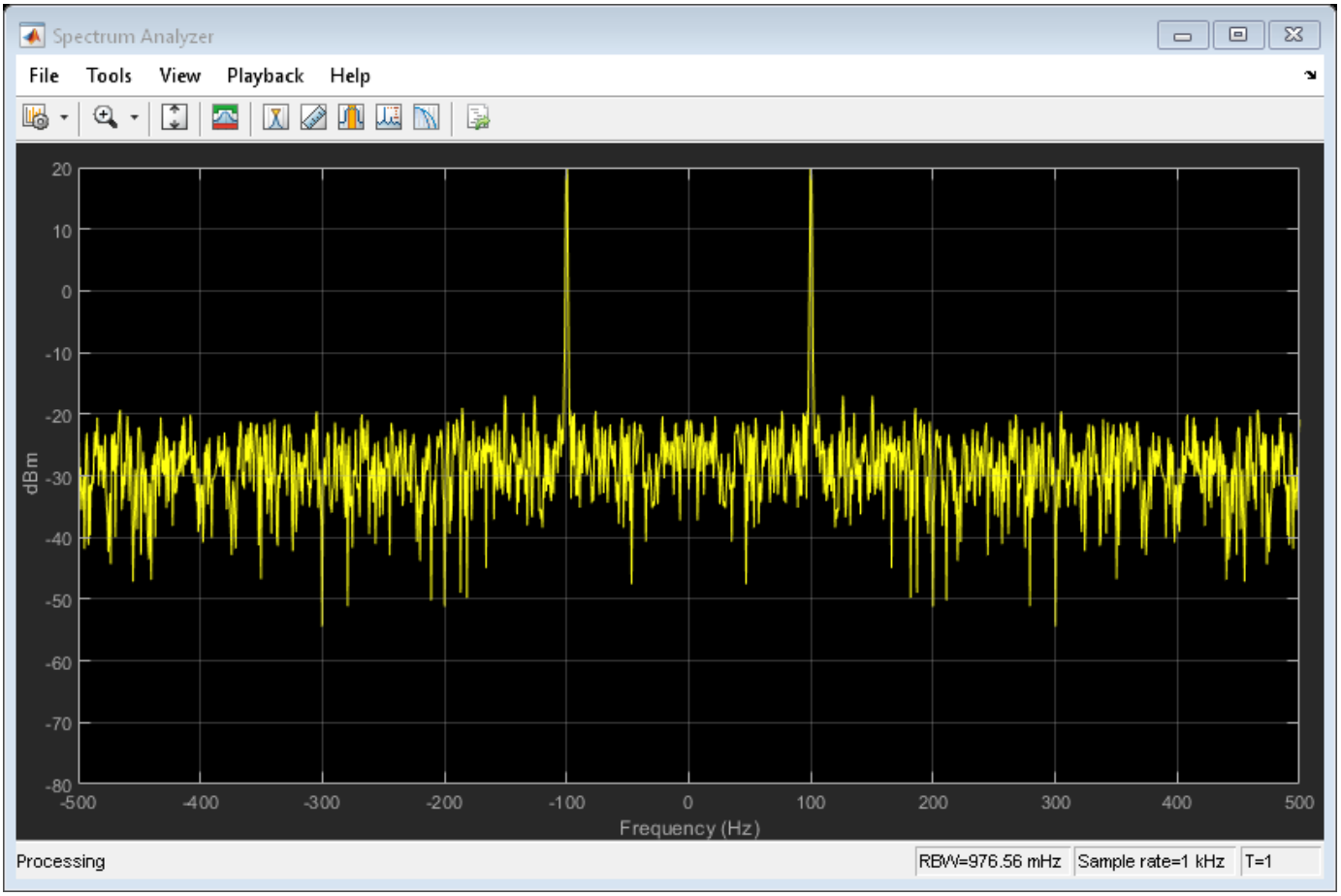
This example shows how to visualize and measure signals in the time and frequency domain in MATLAB using a time scope and spectrum analyzer.

Signal Visualization in Time and Frequency Domains

Create a sine wave with a frequency of 100 Hz sampled at 1000 Hz. Generate five seconds of the 100 Hz sine wave with additive $N(0, 0.0025)$ white noise in one-second intervals. Send the signal to a time scope and spectrum analyzer for display and measurement.

```
SampPerFrame = 1000;
Fs = 1000;
SW = dsp.SineWave('Frequency', 100, ...
  'SampleRate', Fs, 'SamplesPerFrame', SampPerFrame);
TS = dsp.TimeScope('SampleRate', Fs, 'TimeSpan', 0.1, ...
    'YLimits', [-2, 2], 'ShowGrid', true);
SA = dsp.SpectrumAnalyzer('SampleRate', Fs);
tic;
while toc < 5
  sigData = SW() + 0.05*randn(SampPerFrame,1);
  TS(sigData);
  SA(sigData);
end
```


Time Scope
File
Tools
View
Playback
Help
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
Amplitude
0
10
20
30
40
50
60
70
80
90
100
Time (ms)
Processing
Offset=1.9 (secs)
T=1



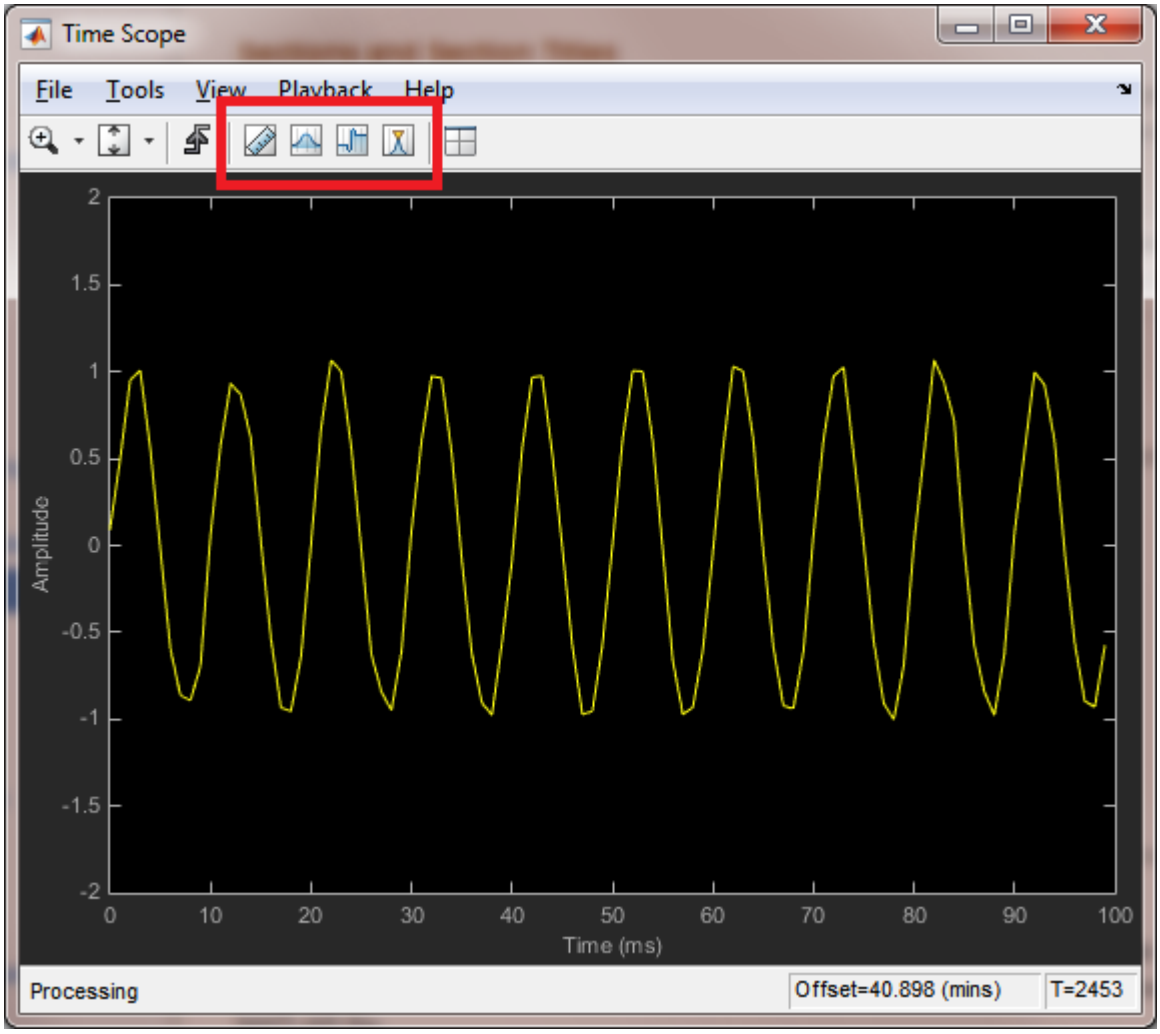
Time-Domain Measurements

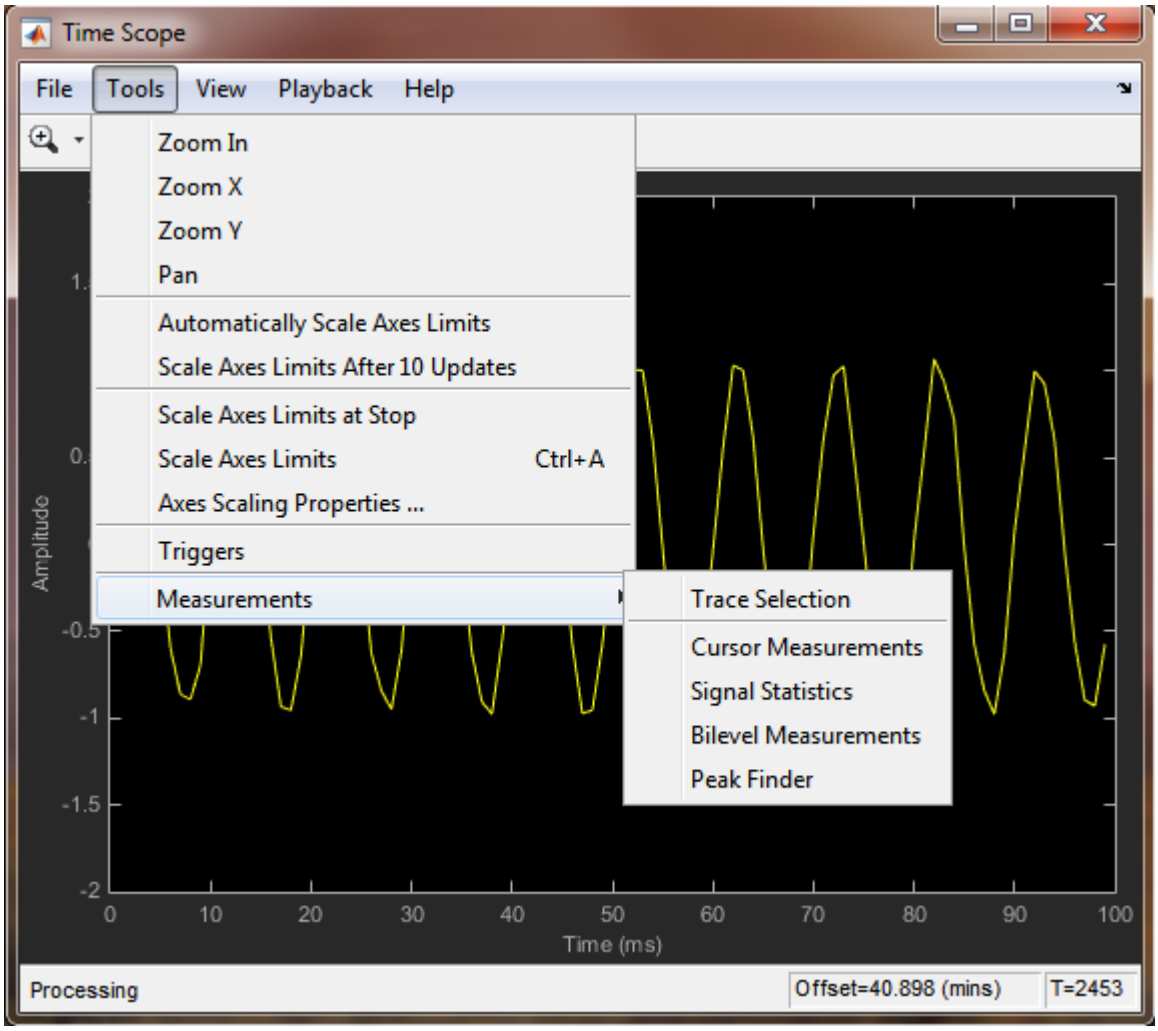
Using the time scope, you can make a number of signal measurements.

The following measurements are available:

- **Cursor Measurements** - puts screen cursors on all scope displays.
- **Signal Statistics** - displays maximum, minimum, peak-to-peak difference, mean, median, RMS values of a selected signal, and the times at which the maximum and minimum occur.
- **Bilevel Measurements** - displays information about a selected signal's transitions, overshoots or undershoots, and cycles.
- **Peak Finder** - displays maxima and the times at which they occur.

You can enable and disable these measurements from the time scope toolbar or from the **Tools > Measurements** menu.


Time Scope
File Tools View Playback Help
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
Amplitude
0 10 20 30 40 50 60 70 80 90 100
Time (ms)
Processing
Offset=40.898 (mins)
T=2453

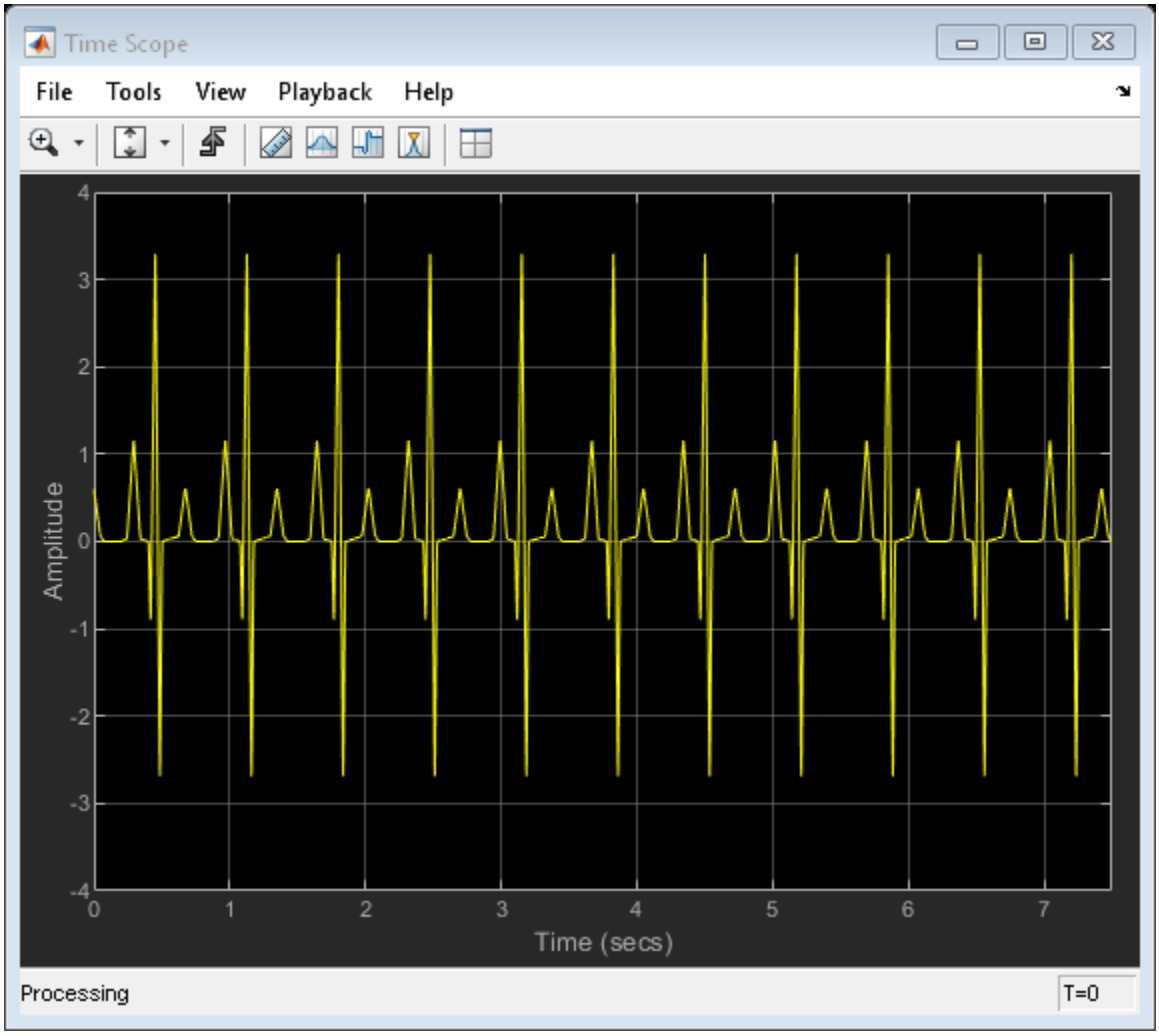


To illustrate the use of measurements in the time scope, simulate an ECG signal. Use the `ecg` function to generate 2700 samples of the signal. Use a Savitzky-Golay filter to smooth the signal and periodically extend the data to obtain approximately 11 periods.

```
x = 3.5*ecg(2700).';
y = repmat(sgolayfilt(x,0,21),[1 13]);
sigData = y((1:30000) + round(2700*rand(1))).';
```

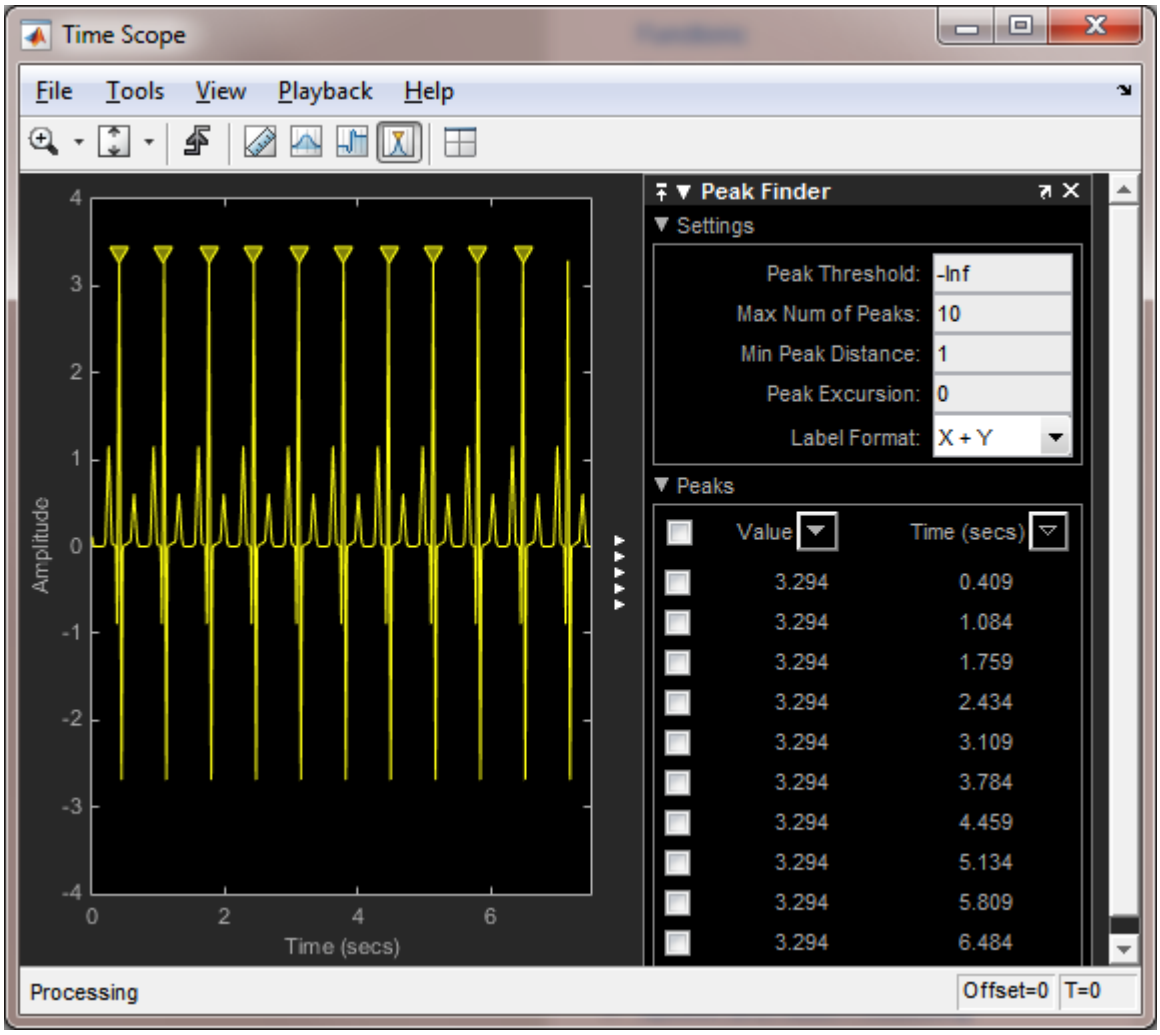
Display the signal in the time scope and use the Peak Finder, Cursor and Signal Statistics measurements. Assume a sample rate of 4 kHz.

```
TS_ECG = dsp.TimeScope('SampleRate', 4000, ...
    'TimeSpanSource', 'Auto', 'ShowGrid', true);
TS_ECG(sigData);
TS_ECG.YLimits = [-4, 4];
```



Peak Measurements

Enable **Peak Measurements** by clicking the corresponding toolbar icon or by clicking the **Tools > Measurements > Peak Finder** menu item. Click **Settings** in the Peak Finder panel to expand the Settings pane. Enter `10` for **Max Num of Peaks** and press Enter. The time scope displays in the Peaks pane a list of 10 peak amplitude values and the times at which they occur.

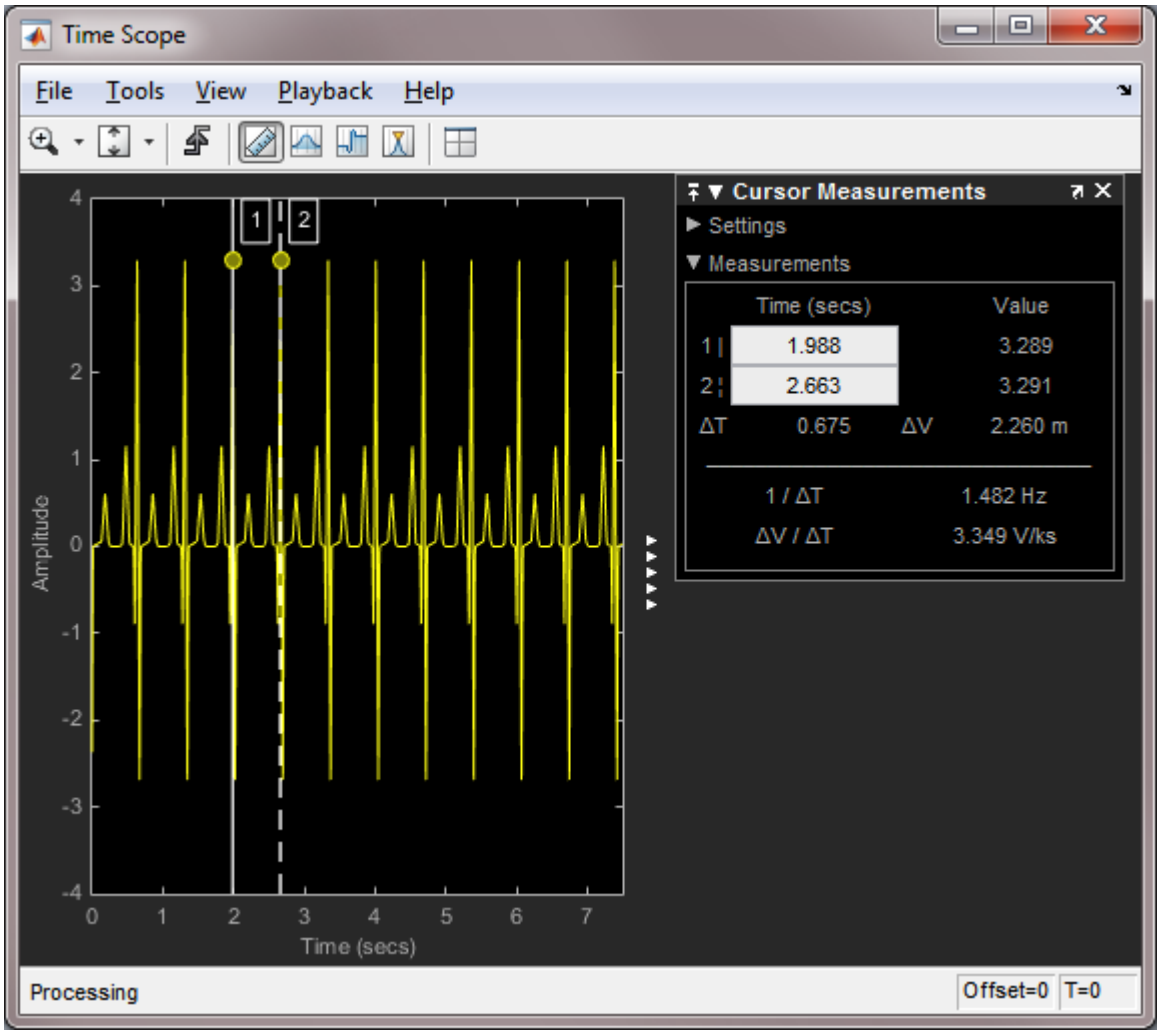


There is a constant time difference of 0.675 seconds between each heartbeat. Therefore, the heart rate of the ECG signal is given by the following equation:

$$\frac{60\,\text{sec/min}}{0.675\,\text{sec/beat}} = 88.89\,\text{beats/min (bpm)}$$

Cursor Measurements

Enable **Cursor Measurements** by clicking the corresponding toolbar icon or by clicking the **Tools > Measurements > Cursor Measurements** menu item. The Cursor Measurements panel opens and displays two cursors in the time scope. You can drag the cursors and use them to measure the time between events in the waveform. In the following figure, cursors are used to measure the time interval between peaks in the ECG waveform. The ΔT measurement in the Cursor Measurements panel demonstrates that the time interval between the two peaks is 0.675 seconds corresponding to a heart rate of 1.482 Hz or 88.9 beats/min.



Signal Statistics and **Bilevel Measurements**

You can also select **Signal Statistics** and **Bilevel Measurements** from the **Tools > Measurements** menu. Signal Statistics can be used to determine the signal's minimum and maximum values as well as other metrics like the peak-to-peak, mean, median, and RMS values. Bilevel Measurements can be used to determine information about rising and falling transitions, transition aberrations, overshoot and undershoot information, pulse width, and duty cycle. To read more about these measurements, see the Time Scope Measurements tutorial example.

Frequency-Domain Measurements

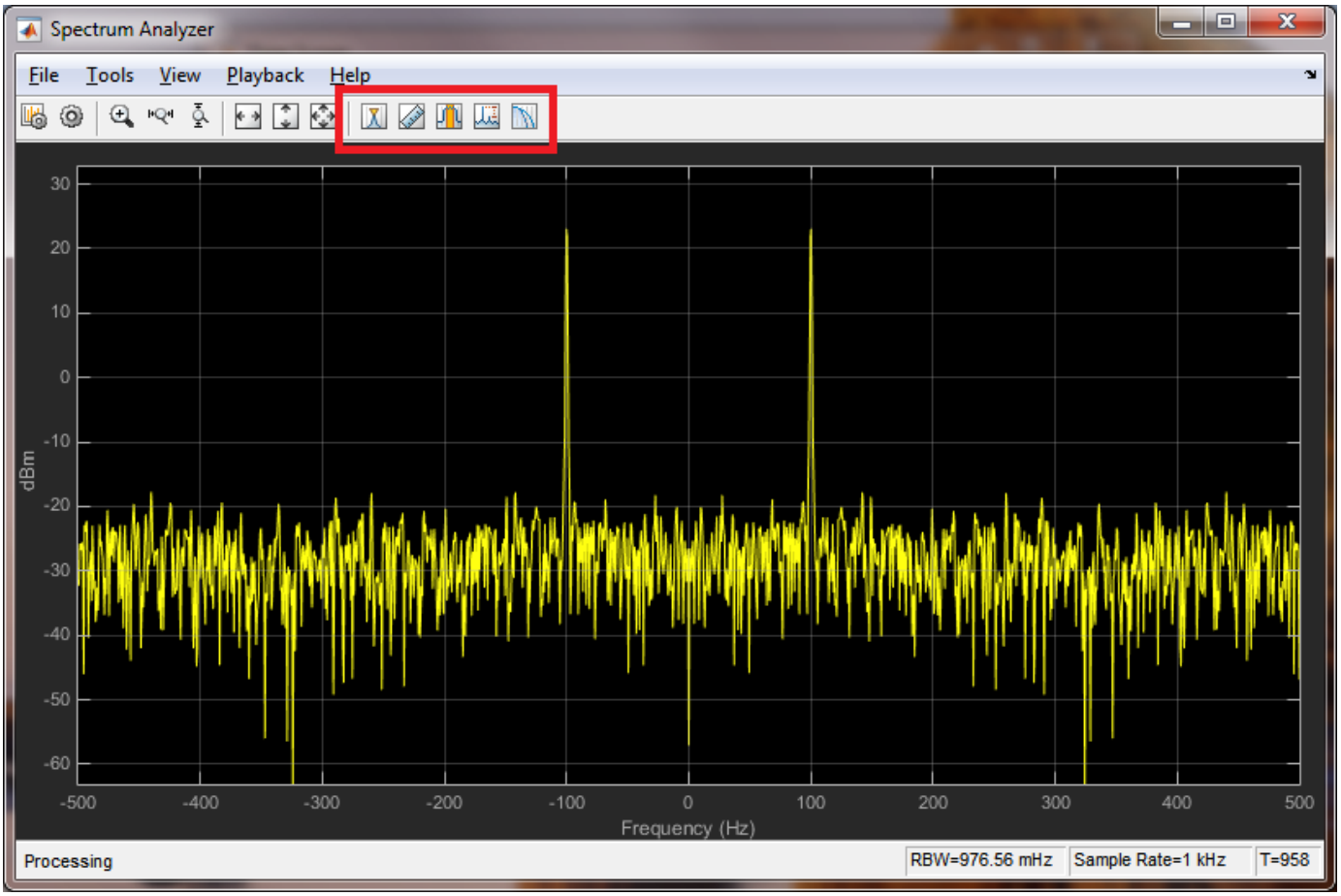
This section explains how to make frequency domain measurements with the spectrum analyzer.

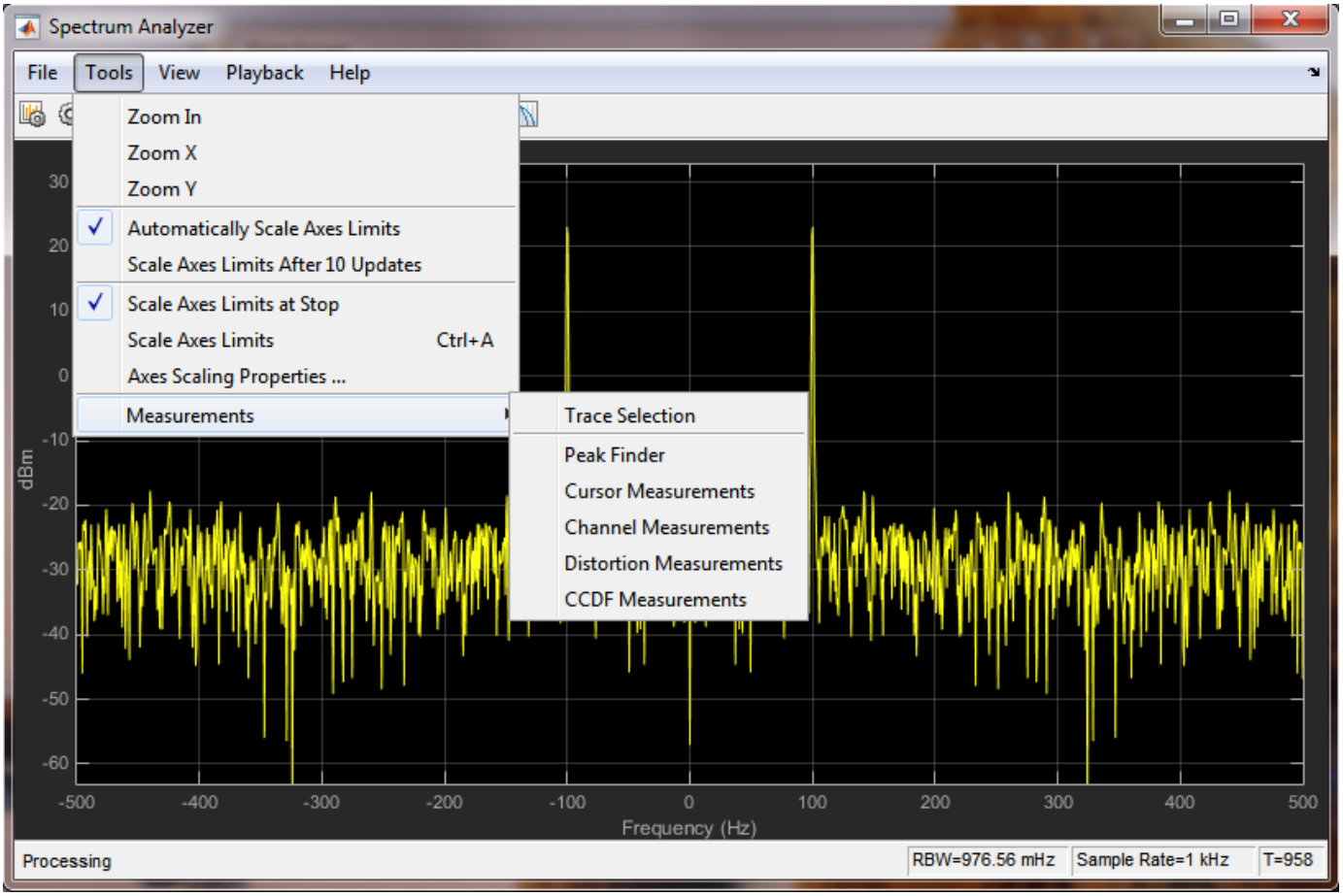
The spectrum analyzer provides the following measurements:

- **Cursor Measurements** - places cursors on the spectrum display.
- **Peak Finder** - displays maxima and the frequencies at which they occur.
- **Channel Measurements** - displays occupied bandwidth and ACPR channel measurements.

- **Distortion Measurements** - displays harmonic and intermodulation distortion measurements.
- **CCDF Measurements** - displays complimentary cumulative distribution function measurements.

You can enable and disable these measurements from the spectrum analyzer toolbar or from the **Tools > Measurements** menu.





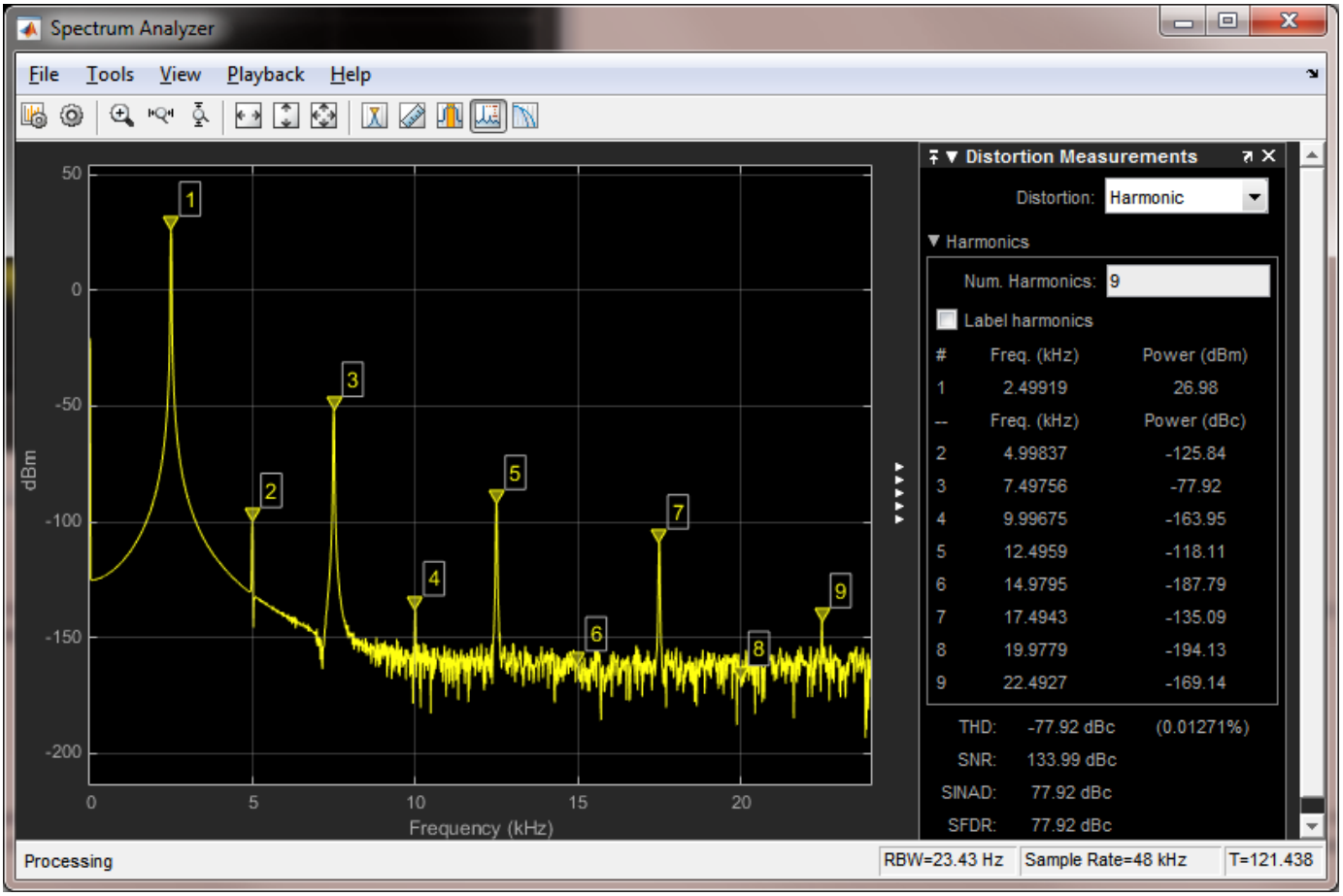
Distortion Measurements

To illustrate the use of measurements with the spectrum analyzer, create a 2.5 kHz sine wave sampled at 48 kHz with additive white Gaussian noise. Evaluate a high-order polynomial (9th degree) at each signal value to model non-linear distortion. Display the signal in a spectrum analyzer.

```
Fs = 48e3;
SW = dsp.SineWave('Frequency', 2500, ...
  'SampleRate', Fs, 'SamplesPerFrame', SampPerFrame);
SA_Distortion = dsp.SpectrumAnalyzer('SampleRate', Fs, ...
    'PlotAsTwoSidedSpectrum', false);
y = [1e-6 1e-9 1e-5 1e-9 1e-6 5e-8 0.5e-3 1e-6 1 3e-3];
tic;
while toc < 5
  x = SW() + 1e-8*randn(SampPerFrame,1);
  sigData = polyval(y, x);
  SA_Distortion(sigData);
end
clear SA_Distortion;
```

Enable the harmonic distortion measurements by clicking the corresponding icon in the toolbar or by clicking the **Tools > Measurements > Distortion Measurements** menu item. In the Distortion Measurements, change the value for **Num. Harmonics** to 9 and check the **Label Harmonics** checkbox. In the panel, you see the value of the fundamental close to 2500 Hz and 8 harmonics as

well as their SNR, SINAD, THD and SFDR values, which are referenced with respect to the fundamental output power.



Peak Finder

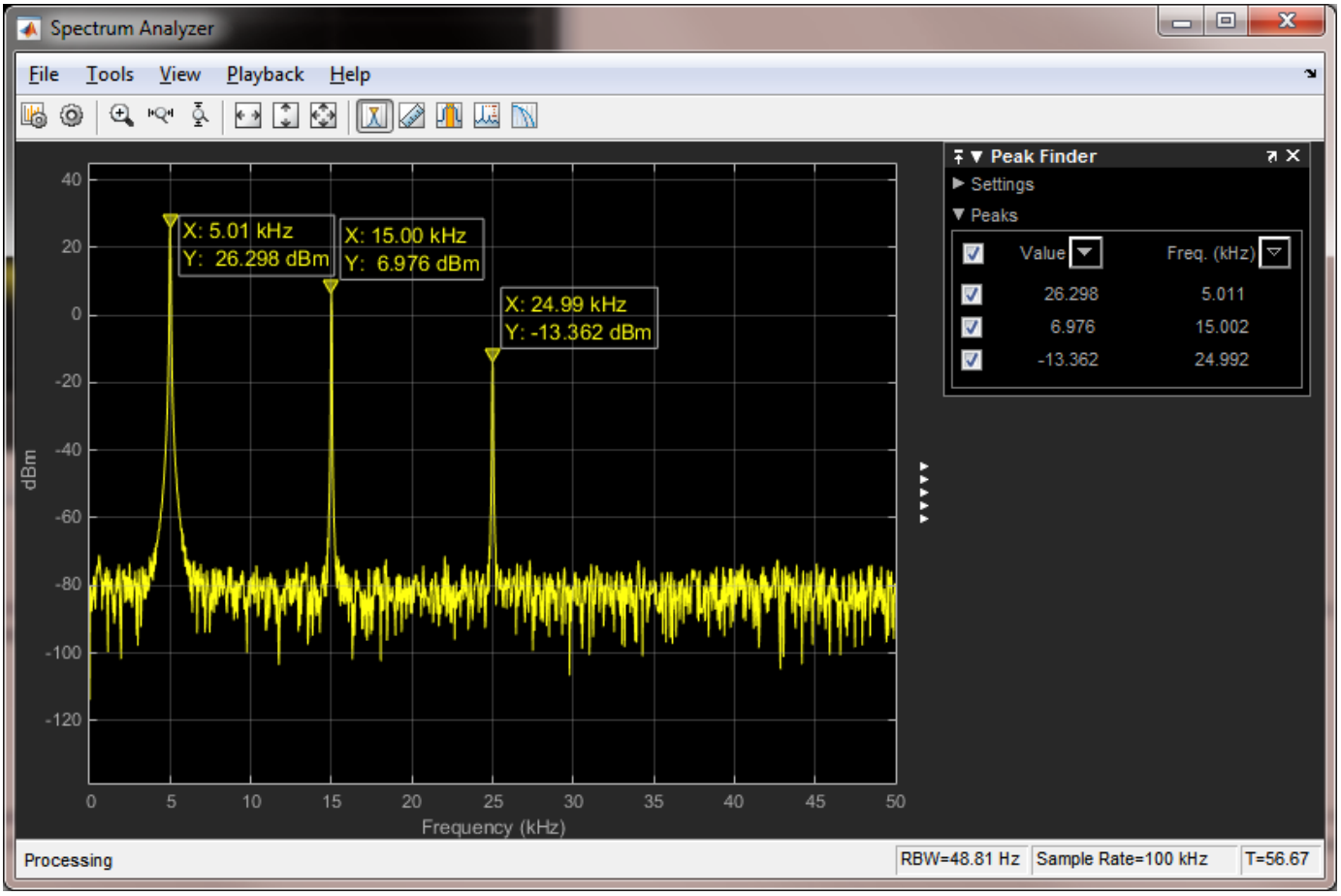
You can track time-varying spectral components by using the Peak Finder measurement dialog. You can show and optionally label up to 100 peaks. You can invoke the Peak Finder dialog from the **Tools > Measurements > Peak Finder** menu item, or by clicking the corresponding icon in the toolbar.

To illustrate the use of **Peak Finder**, create a signal consisting of the sum of three sine waves with frequencies of 5, 15, and 25 kHz and amplitudes of 1, 0.1, and 0.01 respectively. The data is sampled at 100 kHz. Add $N(0, 10^{-8})$ white Gaussian noise to the sum of sine waves and display the one-sided power spectrum in the spectrum analyzer.

```
Fs = 100e3;
SW1 = dsp.SineWave(1e0,   5e3, 0, 'SampleRate', Fs, 'SamplesPerFrame', SampPerFrame);
SW2 = dsp.SineWave(1e-1, 15e3, 0, 'SampleRate', Fs, 'SamplesPerFrame', SampPerFrame);
SW3 = dsp.SineWave(1e-2, 25e3, 0, 'SampleRate', Fs, 'SamplesPerFrame', SampPerFrame);
SA_Peak = dsp.SpectrumAnalyzer('SampleRate', Fs, 'PlotAsTwoSidedSpectrum', false);
tic;
while toc < 5
    sigData = SW1() + SW2() + SW3() + 1e-4*randn(SampPerFrame,1);
    SA_Peak(sigData);
```

```
end
clear SA_Peak;
```

Enable the **Peak Finder** to label the three sine wave frequencies. The frequency values and powers in dBm are displayed in the **Peak Finder** panel. You can increase or decrease the maximum number of peaks, specify a minimum peak distance, and change other settings from the **Settings** pane in the Peak Finder Measurement panel.



To learn more about the use of measurements with the spectrum analyzer, see the Spectrum Analyzer Measurements example.

Filter Frames of a Noisy Sine Wave Signal using Testbench Generator

This example shows how to use the Streaming Testbench Generator app to generate DSP algorithm testbenches. The DSP algorithm generated in this example is similar to the algorithm in the "Filter Frames of a Noisy Sine Wave Signal in MATLAB" on page 1-6 example. That example filters a noisy sine wave signal using a FIR lowpass filter and displays the power spectrum using a spectrum analyzer.

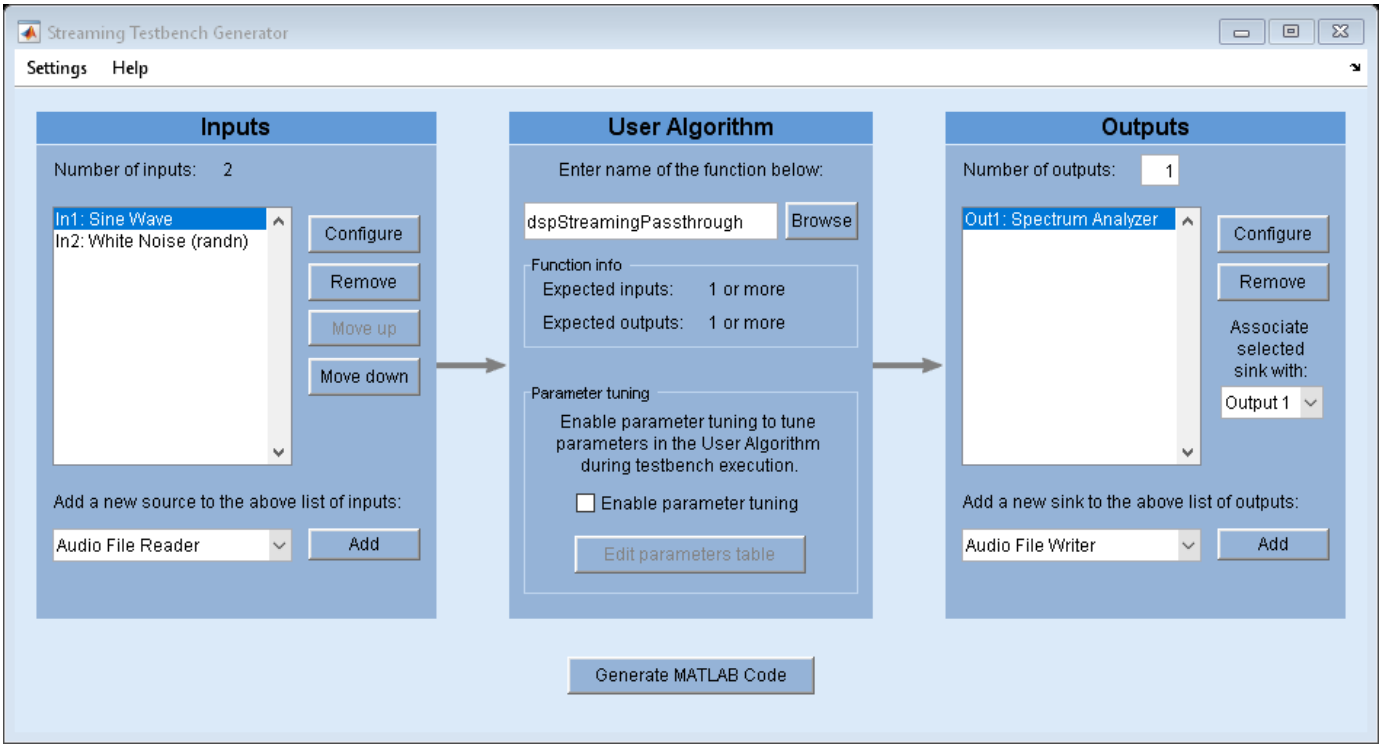
Streaming Testbench Generator Example App

The Streaming Testbench Generator app helps you develop and test streaming signal processing algorithms by enabling you to quickly generate testbenches. To launch the Testbench Generator, enter `testbenchGeneratorExampleApp` at the MATLAB command prompt. The command launches an interface through which you can:

1. Select a set of **sources** and **sinks**.
2. Enter the function name of your custom **User Algorithm**.
3. Customize the properties of each of the added sources and sinks.

Each source is treated as a separate input to your algorithm, but you can associate more than one sink with the same output from your algorithm.

`testbenchGeneratorExampleApp`



Inputs - Sine Waves and White Noise

By default, the testbench generator selects a two-channel sine wave source and a white Gaussian noise source. The two channels of the sine wave source have frequencies of 1 kHz and 10 kHz. The

sampling frequency is 44.1 kHz. The white Gaussian noise input has mean 0 and standard deviation 0.1. The data is processed in frames of 1024 samples. To add more sources, use the list under **Add a new source to the above list of inputs** to select one of the supported sources. Alternatively, you can add your custom System object source by selecting **Custom System object** from the list and clicking **Add**. The added source appears in the list of inputs.

After adding a source, you can select it and click **Configure** to change the selected source's properties.

User Algorithm - Lowpass Filter

The default user algorithm `dspStreamingPassthrough` is a generic function that just passes through the inputs to the outputs. The user algorithm used in this example is a more meaningful function `hTestbenchLowpass`. You can view the code for this function by entering

```
edit hTestbenchLowpass
```

at the MATLAB command prompt. `hTestbenchLowpass` accepts two inputs, lowpass filters the sum of those two inputs, and returns the filtered signal. It uses a constrained equiripple FIR filter design with a cutoff frequency of 5 kHz. The ripples in the passband and stopband are equal to 0.05 and 0.001. Filtering is performed using `dsp.FIRFilter`, which is optimized for streaming.

Type `hTestbenchLowpass` in the User Algorithm text box replacing the default `dspStreamingPassthrough`. Alternatively, you can bring up a new testbench generator session by entering `testbenchGeneratorExampleApp('hTestbenchLowpass')` at the MATLAB command prompt.

Output

The power spectrum of the output is displayed on a spectrum analyzer in dBm. You can add more sinks to visualize or post-process the outputs. Similar to inputs, you can use the list under **Add a new sink to the above list of outputs** to add a new sink, and click **Configure** to modify the properties of the selected sink.

You can associate a single output from the user algorithm with one or more sinks. For example, you can visualize the same output signal with both a time scope and spectrum analyzer. To do this, add the required sinks and make sure you associate all of the sinks to desired output from the user algorithm by changing the value under the **Associate selected sink with** list.

Generate Code and Simulate

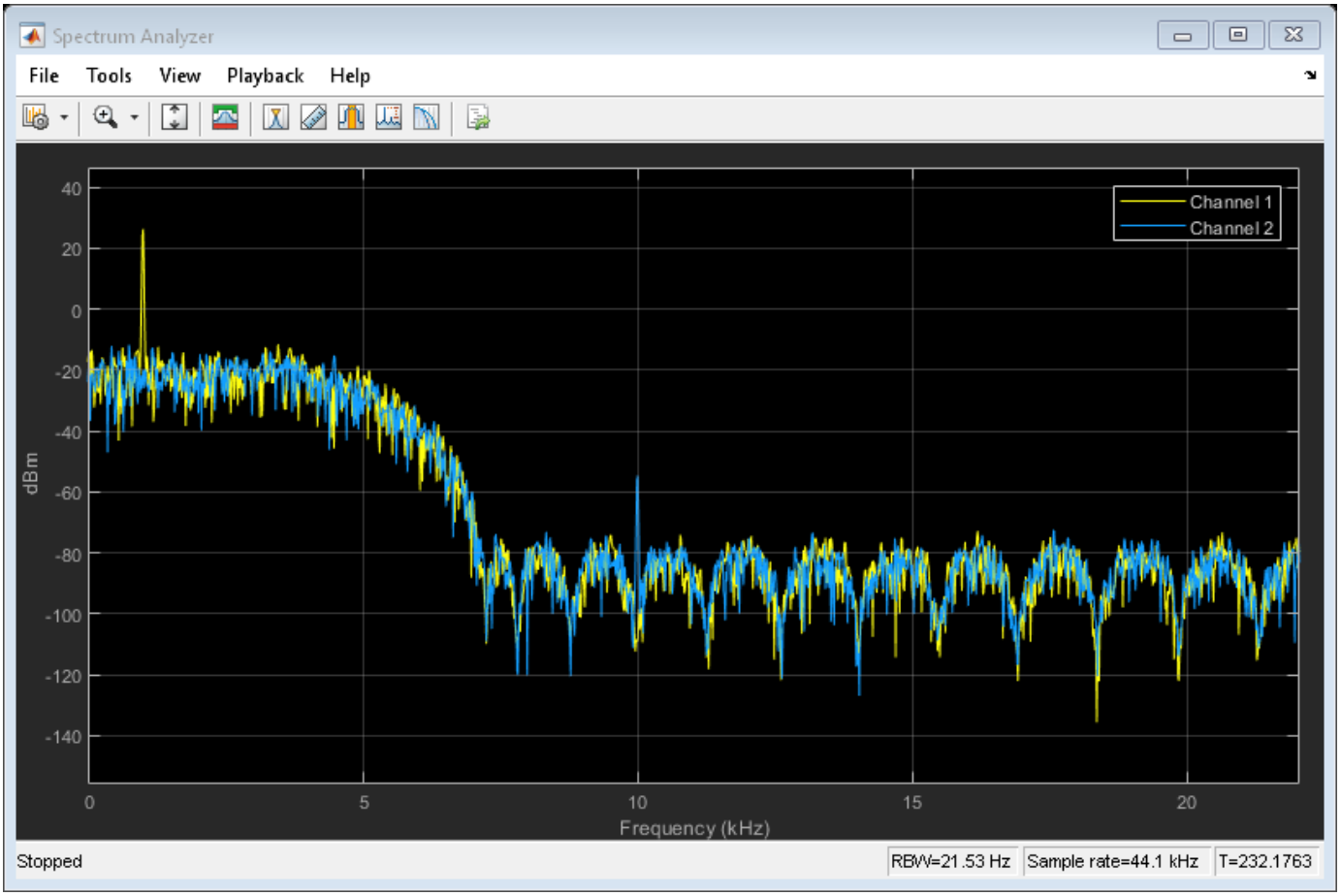
After you add and configure the sources and sinks and enter a function name in the **User Algorithm** text box, the testbench generator is ready to generate testbench MATLAB code. To generate code, click on the **Generate MATLAB Code** button. A new untitled document opens in the MATLAB editor containing the generated testbench code.

You can edit the generated code to customize it before executing it. For the default example, the generated code is included below. Executing this testbench code, you see in the spectrum analyzer that the frequencies above 4 kHz in the source signal are attenuated. The resulting signal maintains the peak at 1 kHz because 1 kHz falls in the passband of the lowpass filter.

Streaming testbench script Generated by Streaming Testbench Generator

```
% Initialization
numIterations = 10000;
```

```
% Construct sources (for all inputs)
src1 = dsp.SineWave('Frequency',[1000 10000], ...
    'SampleRate',44100, ...
    'SamplesPerFrame',1024);

% Construct sinks (for all outputs)
sink1 = dsp.SpectrumAnalyzer('SampleRate',44100, ...
    'PlotAsTwoSidedSpectrum',false, ...
    'ShowLegend',true);

% Stream processing loop
clear hTestbenchLowpass;
for i = 1:numIterations
    % Sources
    in1 = src1();
    in2 = 0.1*randn(1024,2);

    % User Algorithm
    out1 = hTestbenchLowpass(in1,in2);

    % Sinks
    sink1(out1);
end

% Clean up
release(src1);
release(sink1);
```



More Customizations in Testbench Generator

The testbench generator offers additional top-level customizations, which you can configure using the **Testbench Generator Settings** dialog box. To open this dialog box, select **Settings > Testbench Generator Settings ...**.

You can also tune some of the parameters used in your algorithm during testbench execution. To use the **Parameter Tuning UI**, check the **Enable parameter tuning** check box under the **User Algorithm** and click **Edit parameters table** to add the details of your tunable parameters before you generate testbench code. Also, make sure that your user algorithm handles parameter tuning during execution. See the MATLAB code for `hTestbenchVariableBandwithFIR` for an example of how to make your user algorithm work with parameter tuning.

Create Composite System object

This example shows how to create a System object composed of other System objects. In this example, build a multi-notch filter using two `dsp.NotchPeakFilter` System objects. Multi-notch filters are used in many applications. Examples include audio phasers and applications that require the removal of multiple interfering tones.

Create Multi-Notch Filter

"Create Moving Average System object" (MATLAB) explains in detail how to write a System object using a template file. In this example, the entire System object is provided for convenience in `dspdemo.CompositeObj_MultiNotch`. To view the MATLAB code, at the command prompt enter:

```
edit dspdemo.CompositeObj_MultiNotch
```

`dspdemo.CompositeObj_MultiNotch` has five public properties. Public properties are accessible to every user. The first property, `SampleRate`, is a nontunable property. Nontunable properties cannot change when the filter is locked (after you use `step` on the filter). The remaining properties, `CenterFrequency1`, `CenterFrequency2`, `QualityFactor1`, and `QualityFactor2` control settings in the two notch filters contained in `dspdemo.CompositeObj_MultiNotch`. These four properties are tunable. You can change the values of these properties while streaming data.

Set Up the Multi-Notch Filters

`dspdemo.CompositeObj_MultiNotch` uses `dsp.NotchPeakFilter` to design the two notch filters. The notch filters are set up in the `setupImpl` method.

```
methods (Access=protected)
        function setupImpl(obj,~)
            % Construct two notch filters with default values
            obj.NotchFilter1 = dsp.NotchPeakFilter(...
                'Specification', 'Quality factor and center frequency',...
                'CenterFrequency',400);
            obj.NotchFilter2 = dsp.NotchPeakFilter(...
                'Specification', 'Quality factor and center frequency',...
                'CenterFrequency',800);
        end
end
```

Contain System Objects as Private Properties

The ability to create more than one instance of a System object and having each instance manage its own state is one of the biggest advantages of using System objects over functions. The private properties `NotchFilter1` and `NotchFilter2` are used to store the two notch filters.

```
properties (Access=private)
        % This example class contains two notch filters (more can be added
        % in the same way)
        NotchFilter1
        NotchFilter2
end
```

Work with Dependent Properties

The `SampleRate` property as well as the remaining four public properties are implemented as *dependent* properties in this example. Whenever you assign a value to one of the dependent properties, the value is set in the corresponding single-notch filter. When you read one of the

dependent properties, the value is read from the corresponding single-notch filter. Find the following code block in `dspdemo.CompositeObj_MultiNotch`.

```
properties (Dependent)
        %CenterFrequency1 Center frequency of first notch
        %   Specify the first notch center frequency as a finite positive
        %   numeric scalar in Hertz. The default is 400 Hz. This property
        %   is tunable.
        CenterFrequency1;
        %QualityFactor1 Quality factor of first notch
        %   Specify the quality factor (Q factor) for the first notch
        %   filter. The default value is 5. This property is tunable.
        QualityFactor1;
        %CenterFrequency2 Center frequency of second notch
        %   Specify the second notch center frequency as a finite positive
        %   numeric scalar in Hertz. The default is 800 Hz. This property
        %   is tunable.
        CenterFrequency2;
        %QualityFactor2 Quality factor of second notch
        %   Specify the quality factor (Q factor) for the first notch
        %   filter. The default value is 5. This property is tunable.
        QualityFactor2;
end
```

Use the Multi-Notch Filter - Initialization

To use `dspdemo.CompositeObj_MultiNotch`, initialize the filter and any other required components. In this example, initialize an audio file reader, a multi-notch filter, a transfer function estimator, an array plotter, and an audio player.

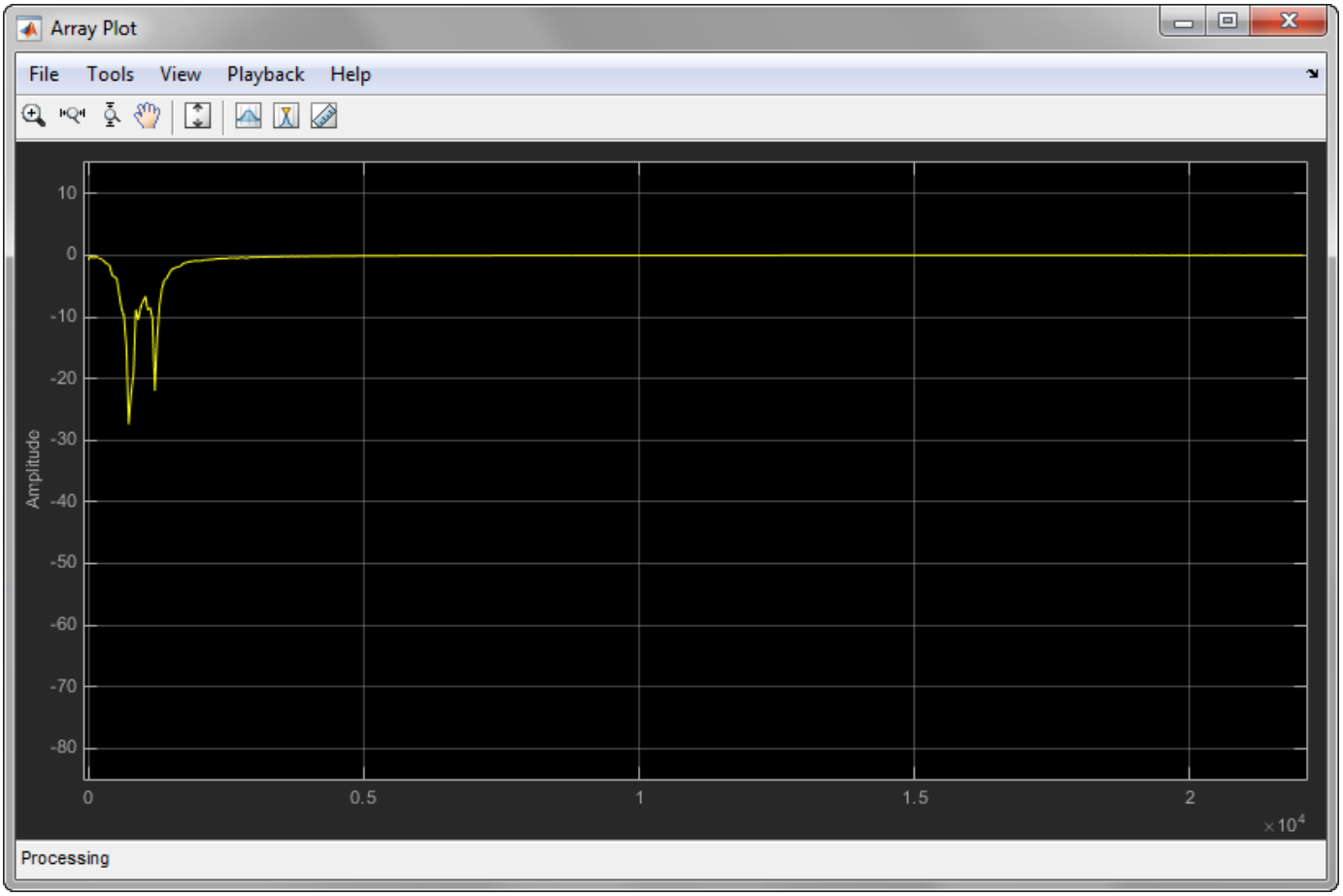
```
FrameSize = 1024;
AFR = dsp.AudioFileReader('guitar10min.ogg','SamplesPerFrame',FrameSize);
Fs  = AFR.SampleRate;

MNF = dspdemo.CompositeObj_MultiNotch('SampleRate',Fs);

TFE = dsp.TransferFunctionEstimator(...
    'FrequencyRange','onesided','SpectralAverages',5);
AP  = dsp.ArrayPlot('PlotType','Line','YLimits',[-85 15],...
    'SampleIncrement',Fs/FrameSize);
P   = audioDeviceWriter;
```

Use the Multi-Notch Filter - Streaming

To illustrate the tunability of the two notch filter center frequencies, vary the center frequencies sinusoidally in a loop. The starting center frequencies are 500 and 2000 Hz. The first center frequency oscillates over the range [100, 900] Hz with a frequency of 0.2 Hz. The second center frequency oscillates over the range [1200, 2800] Hz with a frequency of 0.5 Hz. Because `CenterFrequency1` and `CenterFrequency2` are dependent properties, modifying their values in the loop changes the center frequencies in the two notch filters contained in `dspdemo.CompositeObj_MultiNotch`. To visualize the multi-notch filter, estimate and plot the transfer function continuously. The quality factors remain constant. The simulation runs for 20 seconds.

```
MNF.QualityFactor1 = .5;
MNF.QualityFactor2 = 1;

f0 = 0.2;
f1 = 0.5;

k = 0;
tic,
while toc < 20
    x = AFR();
    t = k*FrameSize/Fs;
```

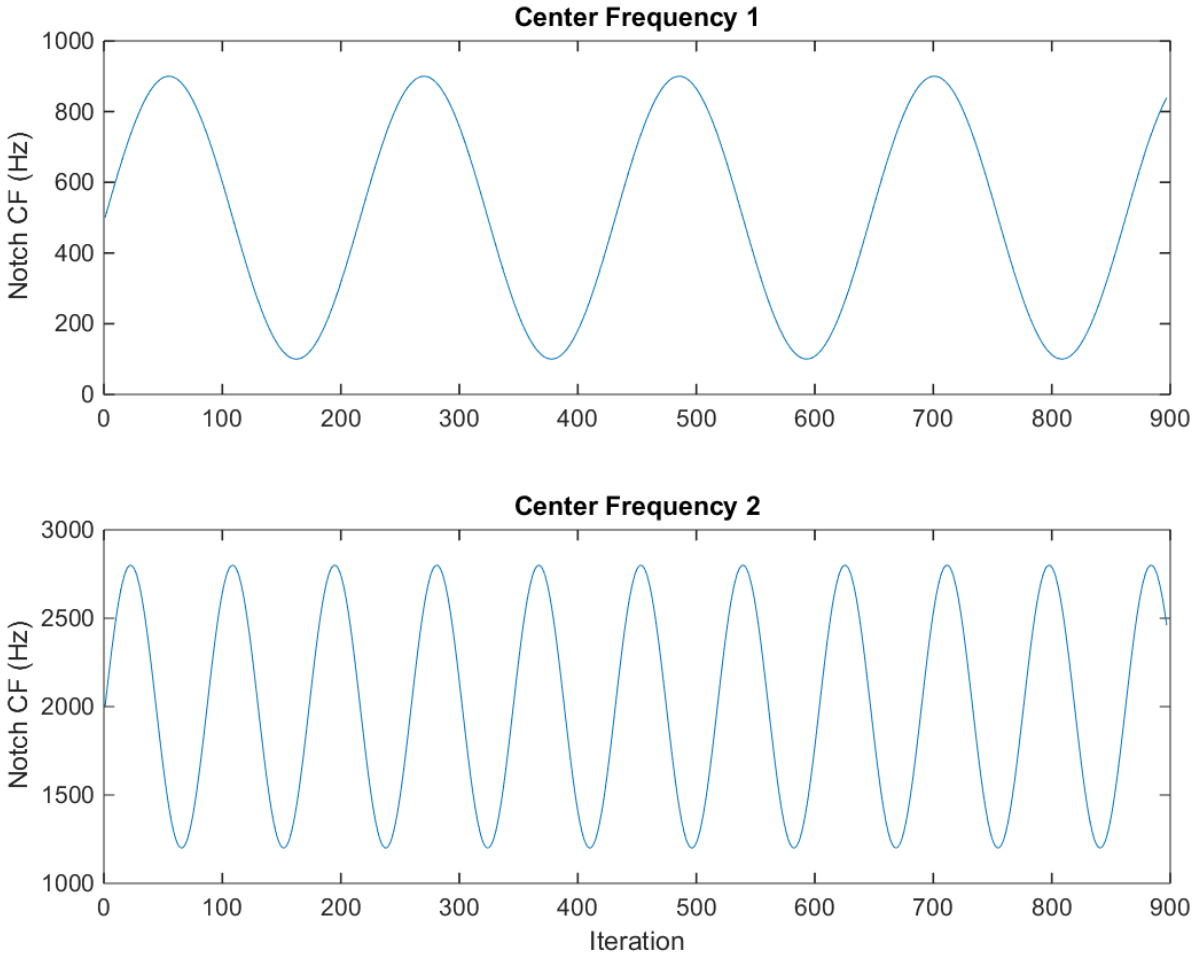
```
        k = k+1;

        MNF.CenterFrequency1 =  500 + 400*sin(2*pi*f0*t);
        MNF.CenterFrequency2 = 2000 + 800*sin(2*pi*f1*t);
        CF1(k) = MNF.CenterFrequency1;
        CF2(k) = MNF.CenterFrequency2;

        y = MNF(x);

        H = TFE(x(:,1),y(:,1));
        magdB = 20*log10(abs(H));
        AP(magdB);
        P(y);
    end
```



Execute this code to show how the two notch center frequencies varied over the simulation.

```
subplot(2,1,1)
plot(CF1);
title('Center Frequency 1');
ylabel('Notch CF (Hz)');
subplot(2,1,2)
plot(CF2);
title('Center Frequency 2');
```

```
ylabel('Notch CF (Hz)');
xlabel('Iteration')
```



Input, Output, and Display

Learn how to input, output and display data and signals with DSP System Toolbox.

- “Discrete-Time Signals” on page 2-2
- “Continuous-Time Signals” on page 2-8
- “Create Signals for Sample-Based Processing” on page 2-9
- “Create Signals for Frame-Based Processing” on page 2-13
- “Create Multichannel Signals for Sample-Based Processing” on page 2-18
- “Create Multichannel Signals for Frame-Based Processing” on page 2-23
- “Deconstruct Multichannel Signals for Sample-Based Processing” on page 2-27
- “Deconstruct Multichannel Signals for Frame-Based Processing” on page 2-32
- “Import and Export Signals for Sample-Based Processing” on page 2-38
- “Import and Export Signals for Frame-Based Processing” on page 2-47

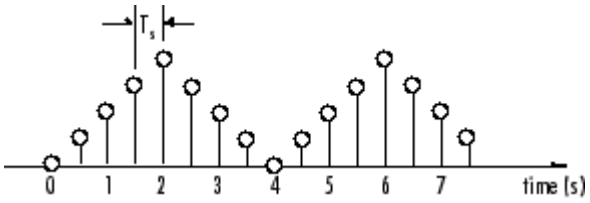
Discrete-Time Signals

In this section...
“Time and Frequency Terminology” on page 2-2
“Recommended Settings for Discrete-Time Simulations” on page 2-3
“Simulink Tasking Modes” on page 2-4
“Other Settings for Discrete-Time Simulations” on page 2-5
“Cross-Rate Operations” on page 2-5

Time and Frequency Terminology

Simulink models can process both discrete-time and continuous-time signals. Models built with the DSP System Toolbox are intended to process discrete-time signals only. A discrete-time signal is a sequence of values that correspond to particular instants in time. The time instants at which the signal is defined are the signal's *sample times*, and the associated signal values are the signal's *samples*. Traditionally, a discrete-time signal is considered to be undefined at points in time between the sample times. For a periodically sampled signal, the equal interval between any pairs of consecutive sample times is the signal's *sample period* T_s. The *sample rate* F_s is the reciprocal of the sample period, or $1/T_s$. The sample rate is the number of samples in the signal per second.

This 7.5-second triangle wave segment has a sample period of 0.5 seconds, and sample times of 0.0, 0.5, 1.0, 1.5, ...,7.5. The sample rate of the sequence is therefore 1/0.5, or 2 Hz.



A number of different terms are used to describe the characteristics of discrete-time signals found in Simulink models. This table lists terms that are frequently used to describe how various blocks operate on sample-based and frame-based signals.

Term	Symbol	Units	Notes
Sample period	T_s T_{si} T_{so}	Seconds	The time interval between consecutive samples in a sequence, as the input to a block (T_{si}) or the output from a block (T_{so}).
Frame period	T_f T_{fi} T_{fo}	Seconds	The time interval between consecutive frames in a sequence, as the input to a block (T_{fi}) or the output from a block (T_{fo}).
Signal period	T	Seconds	The time elapsed during a single repetition of a periodic signal.
Sample frequency	F_s	Hz (samples per second)	The number of samples per unit time, $F_s = 1/T_s$.
Frequency	f	Hz (cycles per second)	The number of repetitions per unit time of a periodic signal or signal component, $f = 1/T$.

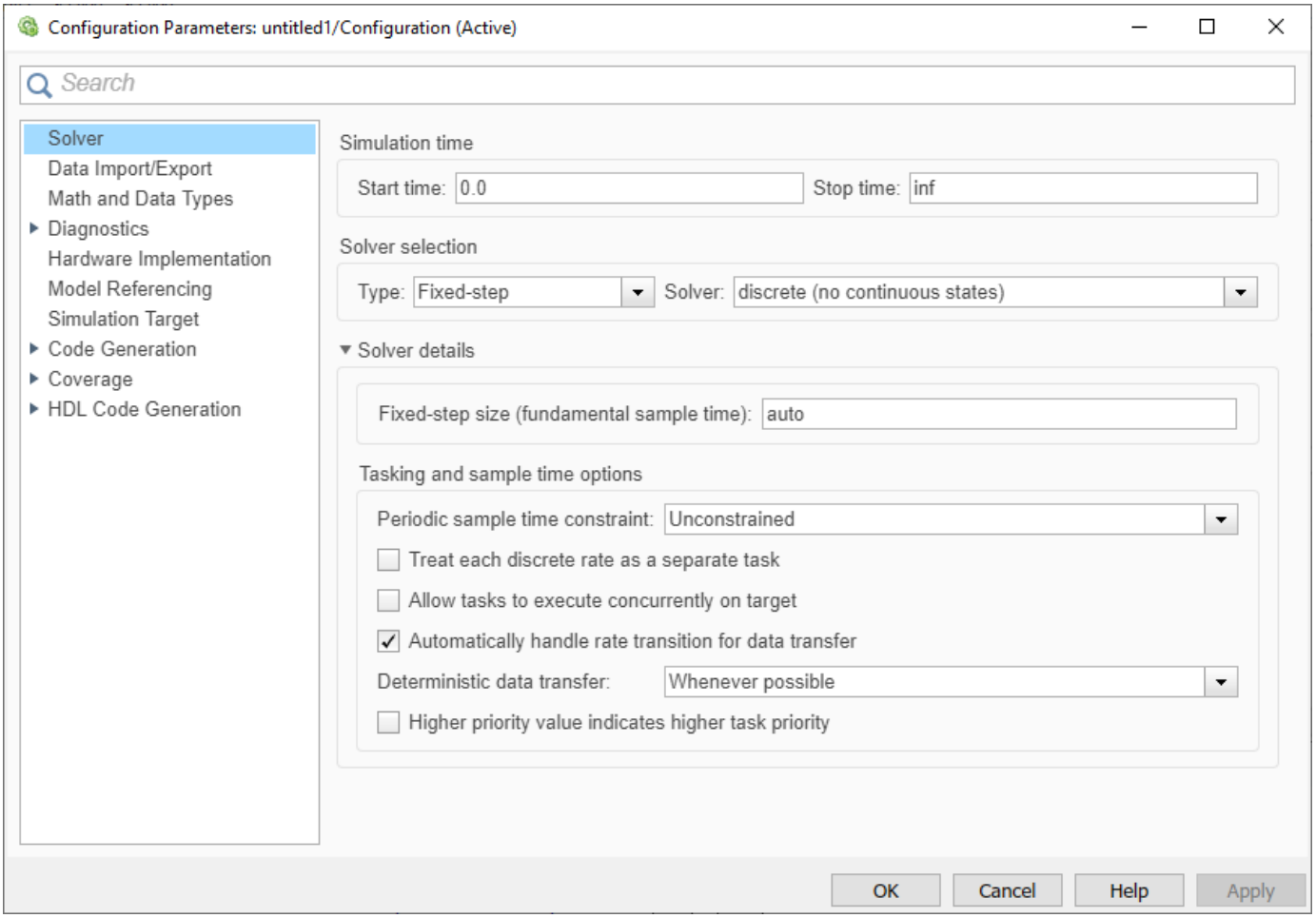
Term	Symbol	Units	Notes
Nyquist rate		Hz (cycles per second)	The minimum sample rate that avoids aliasing, usually twice the highest frequency in the signal being sampled.
Nyquist frequency	f_{nyq}	Hz (cycles per second)	Twice the highest frequency present in the signal.
Normalized frequency	f_n	Two cycles per sample	Frequency (linear) of a periodic signal normalized to half the sample rate, $f_n = \omega/\pi = 2f/F_s$.
Angular frequency	Ω	Radians per second	Frequency of a periodic signal in angular units, $\Omega = 2\pi f$.
Digital (normalized angular) frequency	ω	Radians per sample	Frequency (angular) of a periodic signal normalized to the sample rate, $\omega = \Omega/F_s = \pi f_n$.

Note In the block dialogs, the term *sample time* is used to refer to the *sample period* T_s. For example, the **Sample time** parameter in the Signal From Workspace block specifies the imported signal's sample period.

Recommended Settings for Discrete-Time Simulations

Simulink allows you to select from several different simulation solver algorithms. You can access these solver algorithms from a Simulink model:

1. On the **Modeling** tab, click **Model Settings**. The **Configuration Parameters** dialog box opens.
2. The selections you make in the **Solver** pane determine how discrete-time signals are processed in Simulink. The recommended **Solver** settings for signal processing simulations are:
   - **Type**: `Fixed-step`
   - **Solver**: `Discrete (no continuous states)`
   - **Fixed-step size (fundamental sample time)**: `auto`
   - **Treat each discrete rate as a separate task**: `Off`



You can automatically set these solver options for all new models by using the DSP Simulink model templates. For more information, see “Configure the Simulink Environment for Signal Processing Models”.

Simulink Tasking Modes

When the solver type is set to `Fixed-step`, Simulink operates in two tasking modes:

- Single-tasking mode
- Multitasking mode

On the **Modeling** tab, click **Model Settings**. The **Configuration Parameters** dialog box opens. In the **Solver** pane, select **Type** > `Fixed-step`. Expand **Solver details**. To specify the multitasking mode, select **Treat each discrete rate as a separate task**. To specify the single-tasking mode, clear **Treat each discrete rate as a separate task**.

If you select the **Treat each discrete rate as a separate task** parameter, the single-tasking mode is still used in these cases:

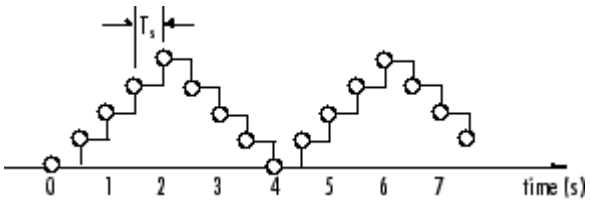
- If your model contains one sample time

- If your model contains a continuous and a discrete sample time, and the fixed-step size is equal to the discrete sample time

For a typical model that operates on a single rate, Simulink selects the single-tasking mode.

Fixed-step single-tasking mode

In the fixed-step, single-tasking mode, discrete-time signals differ from the prototype described in “Time and Frequency Terminology” on page 2-2 by remaining defined between sample times. For example, the representation of the discrete-time triangle wave looks like this.



This signal's value at $t = 3.112$ seconds is the same as the signal's value at $t = 3$ seconds. In the fixed-step, single-tasking mode, a signal's sample times are the instants where the signal is allowed to change values rather than where the signal is defined. Between sample times, the signal takes on the value at the previous sample time.

As a result, in the fixed-step, single-tasking mode, Simulink permits cross-rate operations such as the addition of two signals of different rates. This is explained further in “Cross-Rate Operations” on page 2-5.

Other Settings for Discrete-Time Simulations

It is useful to know how the other solver options available in Simulink affect discrete-time signals. In particular, you should be aware of the properties of discrete-time signals under these settings:

- **Type:** `Fixed-step`, select **Treat each discrete rate as a separate task** to enable the multitasking mode.

  When the fixed-step, multitasking solver is selected, discrete signals in Simulink are undefined between sample times. Simulink generates an error when operations attempt to reference the undefined region of a signal, as, for example, when signals with different sample rates are added.

- **Type:** `Variable-step` (the Simulink default solver)

  When the `Variable-step` solver is selected, discrete-time signals remain defined between sample times, just as in the fixed-step, single-tasking case described in “Recommended Settings for Discrete-Time Simulations” on page 2-3. When the `Variable-step` solver is selected, cross-rate operations are allowed by Simulink.

For a typical model containing multiple rates, Simulink selects the multitasking mode.

Cross-Rate Operations

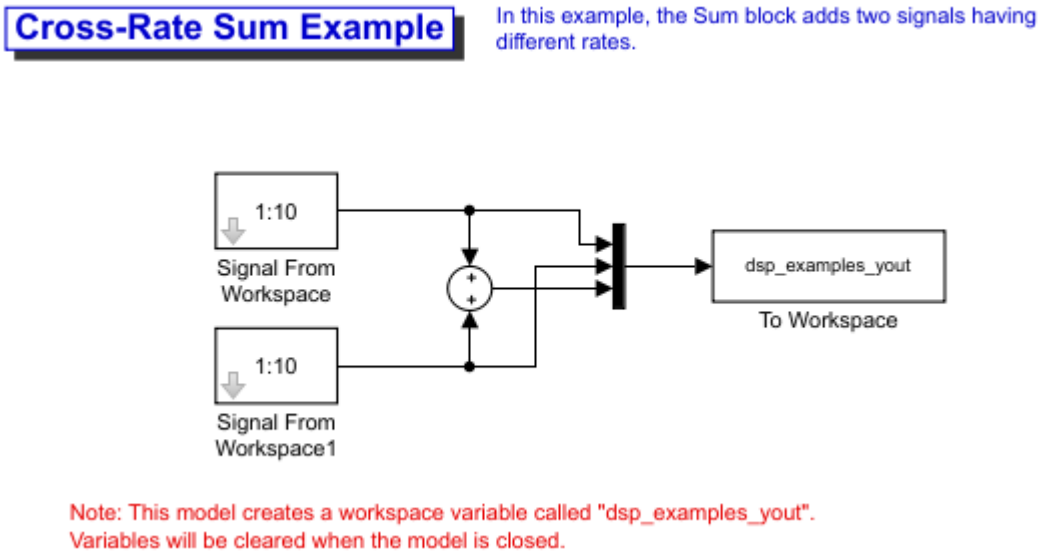
When the fixed-step, multitasking solver is selected, discrete signals in Simulink are undefined between sample times. Therefore, to perform cross-rate operations like the addition of two signals with different sample rates, you must convert the two signals to a common sample rate. Several

blocks in the Signal Operations and Multirate Filters libraries can accomplish this task. See “Convert Sample and Frame Rates in Simulink” on page 3-13 for more information. Rate change can occur implicitly depending on the diagnostic settings. However, this is not recommended. See “Multitask rate transition” (Simulink), “Single task rate transition” (Simulink). By requiring explicit rate conversions for cross-rate operations in discrete mode, Simulink helps you identify sample rate conversion issues early in the design process.

When the `Variable-step` solver or fixed-step, single-tasking solver is selected, discrete-time signals remain defined between sample times. Therefore, if you sample the signal with a rate or phase that is different from the signal's own rate and phase, you will still measure meaningful values:

1 At the MATLAB command line, type `ex_sum_tut1`.

The Cross-Rate Sum Example model opens. This model adds two signals with different sample periods.



2 Double-click the upper Signal From Workspace block. The **Block Parameters: Signal From Workspace** dialog box opens.

3 Set the **Sample time** parameter to 1.

This creates a fast signal (T_s=1) with sample times 1, 2, 3, ...

4 Double-click the lower Signal From Workspace block.

5 Set the **Sample time** parameter to 2.

This creates a slow signal (T_s=2) with sample times 1, 3, 5, ...

6 On the **Debug** tab, select **Information Overlays > Colors**.

Selecting **Colors** allows you to see the different sampling rates in action. For more information about the color coding of the sample times, see “View Sample Time Information” (Simulink).

7 Run the model.

Note Using the DSP Simulink model templates with cross-rate operations generates errors even though a fixed-step, single-tasking solver is selected. This is due to the fact that **Single task rate**

transition is set to `error` in the **Sample Time** pane of the **Diagnostics** section of the **Configuration Parameters** dialog box.

8 At the MATLAB command line, type `dsp_examples_yout`.

The following output is displayed:

```
dsp_examples_yout =
     1     1     2
     2     1     3
     3     2     5
     4     2     6
     5     3     8
     6     3     9
     7     4    11
     8     4    12
     9     5    14
    10     5    15
     0     6     6
```

The first column of the matrix is the fast signal (T_s=1). The second column of the matrix is the slow signal (T_s=2). The third column is the sum of the two signals. As expected, the slow signal changes once every 2 seconds, half as often as the fast signal. Nevertheless, the slow signal is defined at every moment because Simulink holds the previous value of the slower signal during time instances that the block doesn't run.

In general, for `Variable-step` and the fixed-step, single-tasking modes, when you measure the value of a discrete signal between sample times, you are observing the value of the signal at the previous sample time.

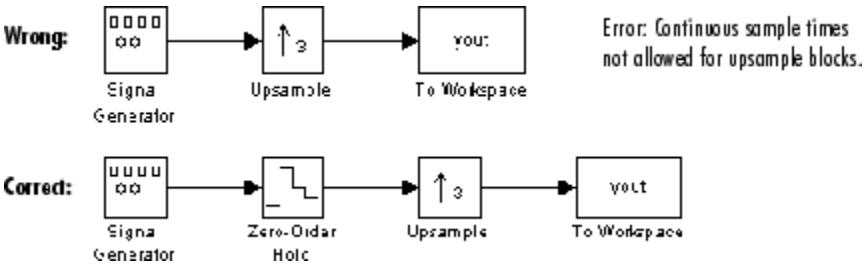
Continuous-Time Signals

In this section...
“Continuous-Time Source Blocks” on page 2-8
“Continuous-Time Nonsource Blocks” on page 2-8

Continuous-Time Source Blocks

Most signals in a signal processing model are discrete-time signals. However, many blocks can also operate on and generate continuous-time signals, whose values vary continuously with time. Source blocks are those blocks that generate or import signals in a model. Most source blocks appear in the Sources library. The sample period for continuous-time source blocks is set internally to zero. This indicates a continuous-time signal. The Simulink Signal Generator and Constant blocks are examples of continuous-time source blocks. To render continuous-time signals in black when, in the **Debug** tab, select **Information Overlays > Colors**.

When connecting continuous-time source blocks to discrete-time blocks, you might need to interpose a Zero-Order Hold block to discretize the signal. Specify the desired sample period for the discrete-time signal in the **Sample time** parameter of the Zero-Order Hold block.



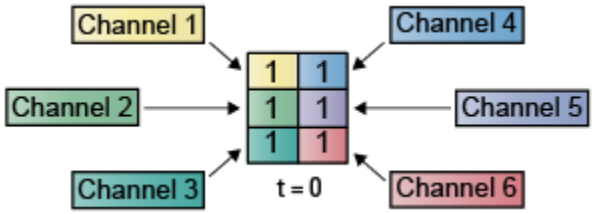
Continuous-Time Nonsource Blocks

Most nonsource blocks in DSP System Toolbox software accept continuous-time signals, and all nonsource blocks inherit the sample period of the input. Therefore, continuous-time inputs generate continuous-time outputs. Blocks that are not capable of accepting continuous-time signals include the Biquad Filter, Discrete FIR Filter, FIR Decimation, and FIR Interpolation blocks.

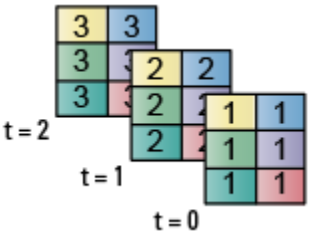
Create Signals for Sample-Based Processing

In this section...
“Create Signals Using Constant Block” on page 2-9
“Create Signals Using Signal From Workspace Block” on page 2-11

In sample-based processing, blocks process signals one sample at a time. Each element of the input signal represents one sample in a distinct channel. For example, from a sample-based processing perspective, the following 3-by-2 matrix contains the first sample in each of six independent channels.



When you configure a block to perform sample-based processing, the block interprets scalar input as a single-channel signal. Similarly, the block interprets an *M*-by-*N* matrix as multichannel signal with *M***N* independent channels. For example, in sample-based processing, blocks interpret the following sequence of 3-by-2 matrices as a six-channel signal.



For more information about the recent changes to frame-based processing, see the “Frame-based processing changes” section of the *DSP System Toolbox Release Notes*.

This page discusses creating signals for sample-based processing using the Constant block and the Signal From Workspace block. Note that the block receiving this signal implements sample-based processing or frame-based processing on the signal based on the parameters set in the block dialog box.

Create Signals Using Constant Block

1. Create a new Simulink model.
2. From the Sources library, click-and-drag a Constant block into the model.
3. From the Sinks library, click-and-drag a Display block into the model.
4. Connect the two blocks.
5. Double-click the Constant block, and set the block parameters as follows:
   - **Constant value** = `[1 2 3; 4 5 6]`

- **Interpret vector parameters as 1-D** = Clear this check box
- **Sample time** = 1

Based on these parameters, the Constant block outputs a constant, discrete-valued, 2-by-3 matrix signal with a sample period of 1 second.

The Constant block's **Constant value** parameter can be any valid MATLAB variable or expression that evaluates to a matrix.

6 Save these parameters and close the dialog box by clicking **OK**.

7 In the **Debug** tab of the model toolstrip, select **Information Overlays > Signal Dimensions**.

8 Run the model and expand the Display block so you can view the entire signal.

You have now successfully created a six-channel signal with a sample period of 1 second.

To view the model you just created, and to learn how to create a 1–D vector signal from the block diagram you just constructed, continue to the next section.

Create an Unoriented Vector Signal

You can create an unoriented vector by modifying the block diagram you constructed in the previous section:

1 To add another signal to your model, copy the block diagram you created in the previous section and paste it below the existing signal in your model.

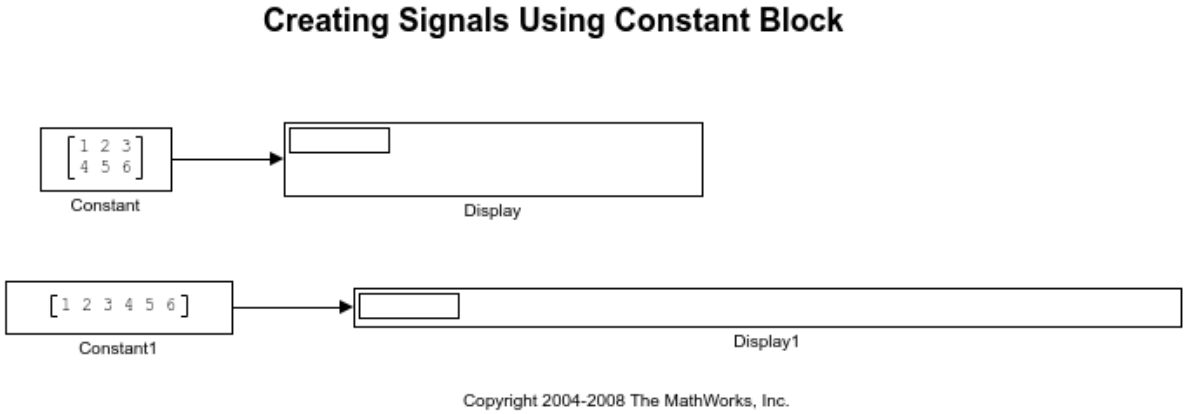
2 Double-click the Constant1 block, and set the block parameters as follows:

- **Constant value** = `[1 2 3 4 5 6]`
- **Interpret vector parameters as 1-D** = Check this box
- **Sample time** = 1

3 Save these parameters and close the dialog box by clicking **OK**.

4 Run the model and expand the Display1 block so you can view the entire signal.

Your model should now look similar to the following figure. You can also open this model by typing `ex_usingcnstblksb` at the MATLAB command line.



The Constant1 block generates a length-6 unoriented vector signal. This means that the output is not a matrix. However, most nonsource signal processing blocks interpret a length-M unoriented vector as an M-by-1 matrix (column vector).

Create Signals Using Signal From Workspace Block

This topic discusses how to create a four-channel signal for sample-based processing with a sample period of 1 second using the Signal From Workspace block:

1. Create a new Simulink model.
2. From the Sources library, click-and-drag a Signal From Workspace block into the model.
3. From the Simulink Sinks library, click-and-drag a To Workspace block into the model.
4. Connect the two blocks.
5. Double-click the Signal From Workspace block, and set the block parameters as follows:

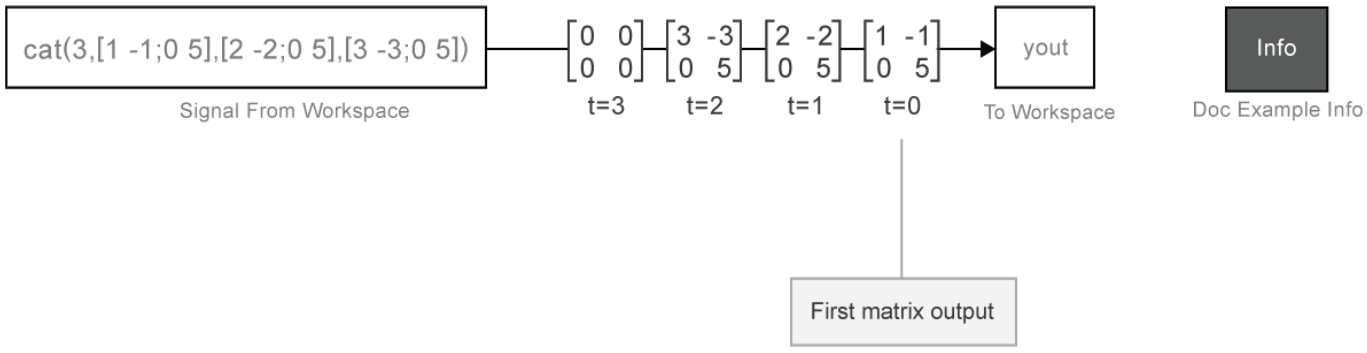
   - **Signal** = `cat(3,[1 -1;0 5],[2 -2;0 5],[3 -3;0 5])`
   - **Sample time** = `1`
   - **Samples per frame** = `1`
   - **Form output after final data value by** = `Setting to zero`

   Based on these parameters, the Signal From Workspace block outputs a four-channel signal with a sample period of 1 second. After the block has output the signal, all subsequent outputs have a value of zero. The four channels contain the following values:

   - Channel 1: 1, 2, 3, 0, 0,...
   - Channel 2: -1, -2, -3, 0, 0,...
   - Channel 3: 0, 0, 0, 0, 0,...
   - Channel 4: 5, 5, 5, 0, 0,...
6. Save these parameters and close the dialog box by clicking **OK**.
7. In the **Debug** tab of the model toolstrip, select **Information Overlays > Signal Dimensions**.
8. Run the model.

   The following figure is a graphical representation of the model's behavior during simulation. You can also open the model by typing `ex_usingsfwblksb` at the MATLAB command line.

Creating Sample-Based Signals Using the Signal From Workspace Block



9 At the MATLAB command line, type `yout`.

The following is a portion of the output:

```
yout(:,:,1) =

1    -1
0     5

yout(:,:,2) =

2    -2
0     5

yout(:,:,3) =

3    -3
0     5

yout(:,:,4) =

0     0
0     0
```

You have now successfully created a four-channel signal with sample period of 1 second using the Signal From Workspace block. This signal is used for sample-based processing.

See Also

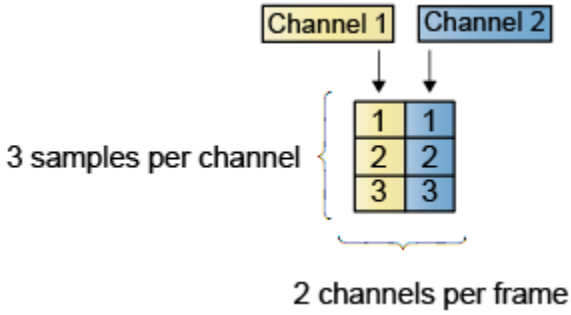
More About

- “Sample- and Frame-Based Concepts” on page 3-2
- “Create Signals for Frame-Based Processing” on page 2-13
- “Create Multichannel Signals for Sample-Based Processing” on page 2-18
- “Create Multichannel Signals for Frame-Based Processing” on page 2-23
- “Deconstruct Multichannel Signals for Sample-Based Processing” on page 2-27
- “Deconstruct Multichannel Signals for Frame-Based Processing” on page 2-32

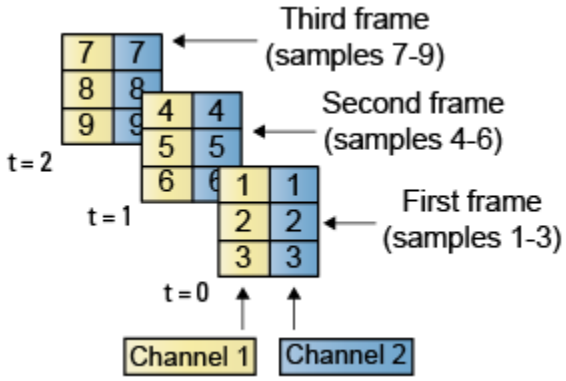
Create Signals for Frame-Based Processing

In this section...
“Create Signals Using Sine Wave Block” on page 2-14
“Create Signals Using Signal From Workspace Block” on page 2-15

In frame-based processing, blocks process data one frame at a time. Each frame of data contains sequential samples from an independent channel. Each channel is represented by a column of the input signal. For example, from a frame-based processing perspective, the following 3-by-2 matrix has two channels, each of which contains three samples.



When you configure a block to perform frame-based processing, the block interprets an *M*-by-1 vector as a single-channel signal containing *M* samples per frame. Similarly, the block interprets an *M*-by-*N* matrix as a multichannel signal with *N* independent channels and *M* samples per channel. For example, in frame-based processing, blocks interpret the following sequence of 3-by-2 matrices as a two-channel signal with a frame size of 3.



Using frame-based processing is advantageous for many signal processing applications because you can process multiple samples at once. By buffering your data into frames and processing multisample frames of data, you can often improve the computational time of your signal processing algorithms. To perform frame-based processing, you must have a DSP System Toolbox license.

For more information about the recent changes to frame-based processing, see the “Frame-based processing changes” section of the *DSP System Toolbox Release Notes*.

This page discusses creating signals for frame-based processing using the Sine Wave block and the Signal From Workspace block. Note that the block receiving this signal implements sample-based

processing or frame-based processing on the signal based on the parameters set in the block dialog box.

Create Signals Using Sine Wave Block

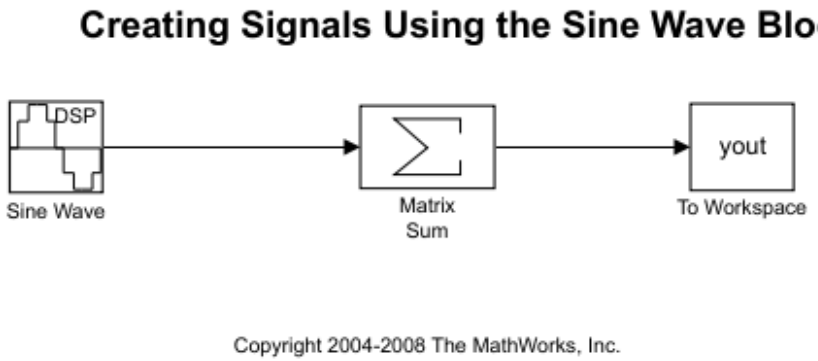
1. Create a new Simulink model.
2. From the Sources library, click-and-drag a Sine Wave block into the model.
3. From the Matrix Operations library, click-and-drag a Matrix Sum block into the model.
4. From the Simulink Sinks library, click-and-drag a To Workspace block into the model.
5. Connect the blocks in the order in which you added them to your model.
6. Double-click the Sine Wave block, and set the block parameters as follows:

   - **Amplitude** = `[1 3 2]`
   - **Frequency** = `[100 250 500]`
   - **Sample time** = `1/5000`
   - **Samples per frame** = `64`

   Based on these parameters, the Sine Wave block outputs three sinusoids with amplitudes `1`, `3`, and `2` and frequencies `100`, `250`, and `500` Hz, respectively. The sample period, 1/5000, is 10 times the highest sinusoid frequency, which satisfies the Nyquist criterion. The frame size is 64 for all sinusoids, and, therefore, the output has 64 rows.
7. Save these parameters and close the dialog box by clicking **OK**.

   You have now successfully created a three-channel signal, with `64` samples per each frame, using the Sine Wave block. The rest of this procedure describes how to add these three sinusoids together.
8. Double-click the Matrix Sum block. Set the **Sum over** parameter to `Specified dimension`, and set the **Dimension** parameter to `2`. Click **OK**.
9. In the **Debug** tab of the model toolstrip, select **Information Overlays** > **Signal Dimensions**.
10. Run the model.

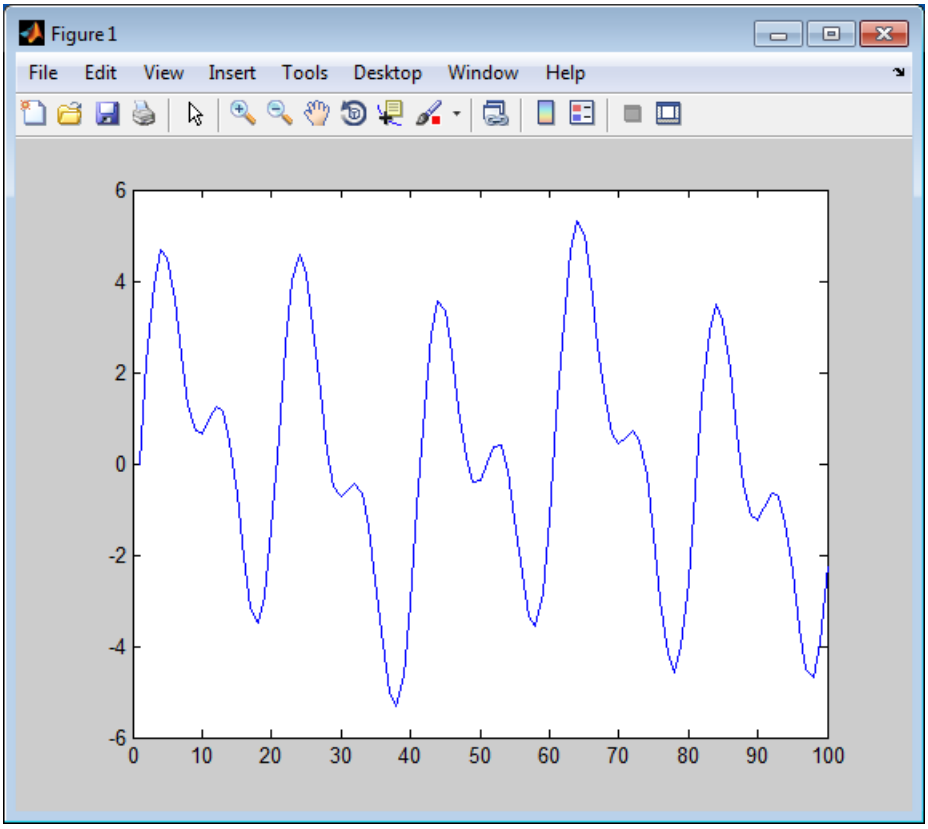
    Your model should now look similar to the following figure. You can also open the model by typing `ex_usingsinwaveblkfb` at the MATLAB command line.



The three signals are summed point-by-point by a Matrix Sum block. Then, they are exported to the MATLAB workspace.

11 At the MATLAB command line, type `plot(yout(1:100))`.

Your plot should look similar to the following figure.



This figure represents a portion of the sum of the three sinusoids. You have now added the channels of a three-channel signal together and displayed the results in a figure window.

Create Signals Using Signal From Workspace Block

Frame-based processing can significantly improve the performance of your model by decreasing the amount of time it takes your simulation to run. This topic describes how to create a two-channel signal with a sample period of 1 second, a frame period of 4 seconds, and a frame size of 4 samples using the Signal From Workspace block.

1 Create a new Simulink model.

2 From the Sources library, click-and-drag a Signal From Workspace block into the model.

3 From the Simulink Sinks library, click-and-drag a To Workspace block into the model.

4 Connect the two blocks.

5 Double-click the Signal From Workspace block, and set the block parameters as follows.

- **Signal** = `[1:10; 1 1 0 0 1 1 0 0 1 1]'`

- **Sample time** = 1
- **Samples per frame** = 4
- **Form output after final data value by** = `Setting to zero`

Based on these parameters, the Signal From Workspace block outputs a two-channel signal with a sample period of 1 second, a frame period of 4 seconds, and a frame size of four samples. After the block outputs the signal, all subsequent outputs have a value of zero. The two channels contain the following values:

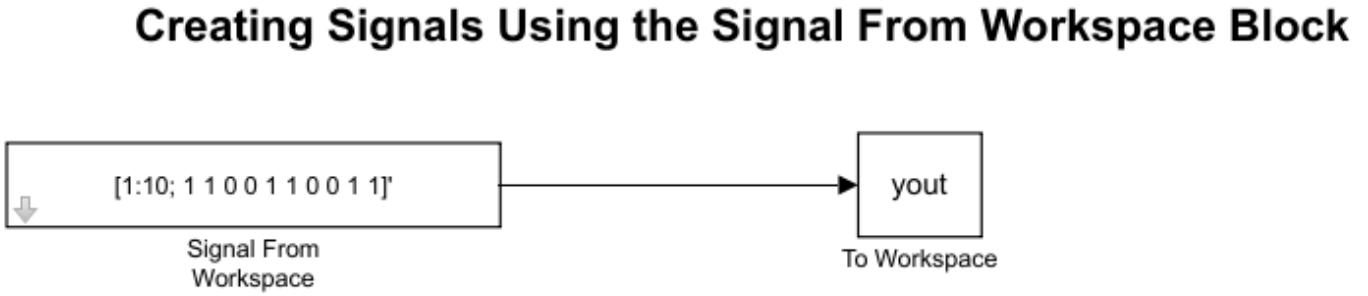
- Channel 1: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 0, 0,...
- Channel 2: 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0,...

6 Save these parameters and close the dialog box by clicking **OK**.

7 In the **Debug** tab of the model toolstrip, select **Information Overlays** > **Signal Dimensions**.

8 Run the model.

The following figure is a graphical representation of the model's behavior during simulation. You can also open the model by typing `ex_usingsfwblkfb` at the MATLAB command line.



9 At the MATLAB command line, type `yout`.

The following is the output displayed at the MATLAB command line.

```
yout =

     1     1
     2     1
     3     0
     4     0
     5     1
     6     1
     7     0
     8     0
     9     1
    10     1
     0     0
     0     0
```

Note that zeros were appended to the end of each channel. You have now successfully created a two-channel signal and exported it to the MATLAB workspace.

See Also

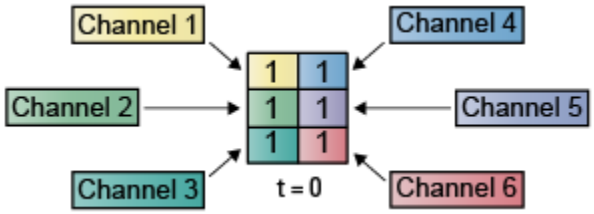
More About

- “Sample- and Frame-Based Concepts” on page 3-2
- “Create Signals for Sample-Based Processing” on page 2-9
- “Create Multichannel Signals for Sample-Based Processing” on page 2-18
- “Create Multichannel Signals for Frame-Based Processing” on page 2-23
- “Deconstruct Multichannel Signals for Sample-Based Processing” on page 2-27
- “Deconstruct Multichannel Signals for Frame-Based Processing” on page 2-32

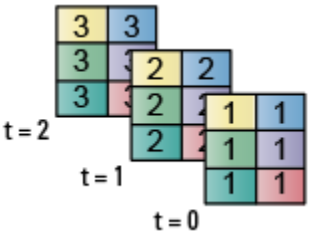
Create Multichannel Signals for Sample-Based Processing

In this section...
“Multichannel Signals for Sample-Based Processing” on page 2-18
“Create Multichannel Signals by Combining Single-Channel Signals” on page 2-19
“Create Multichannel Signals by Combining Multichannel Signals” on page 2-20

In sample-based processing, blocks process signals one sample at a time. Each element of the input signal represents one sample in a distinct channel. For example, from a sample-based processing perspective, the following 3-by-2 matrix contains the first sample in each of six independent channels.



When you configure a block to perform sample-based processing, the block interprets scalar input as a single-channel signal. Similarly, the block interprets an *M*-by-*N* matrix as multichannel signal with *M***N* independent channels. For example, in sample-based processing, blocks interpret the following sequence of 3-by-2 matrices as a six-channel signal.



For more information about the recent changes to frame-based processing, see the “Frame-based processing changes” section of the *DSP System Toolbox Release Notes*.

Multichannel Signals for Sample-Based Processing

When you want to perform the same operations on several independent signals, you can group those signals together as a multichannel signal. For example, if you need to filter each of four independent signals using the same direct-form II transpose filter, you can combine the signals into a multichannel signal, and connect the signal to a single Biquad Filter block. The block decides to treat each element of the input as a channel when you set the block's **Input processing** parameter to `Elements as channels (sample based)`. The block then applies the filter to each channel independently.

Multiple independent signals can be combined into a single multichannel signal using the Concatenate block. In addition, several multichannel signals can be combined into a single multichannel signal using the same technique.

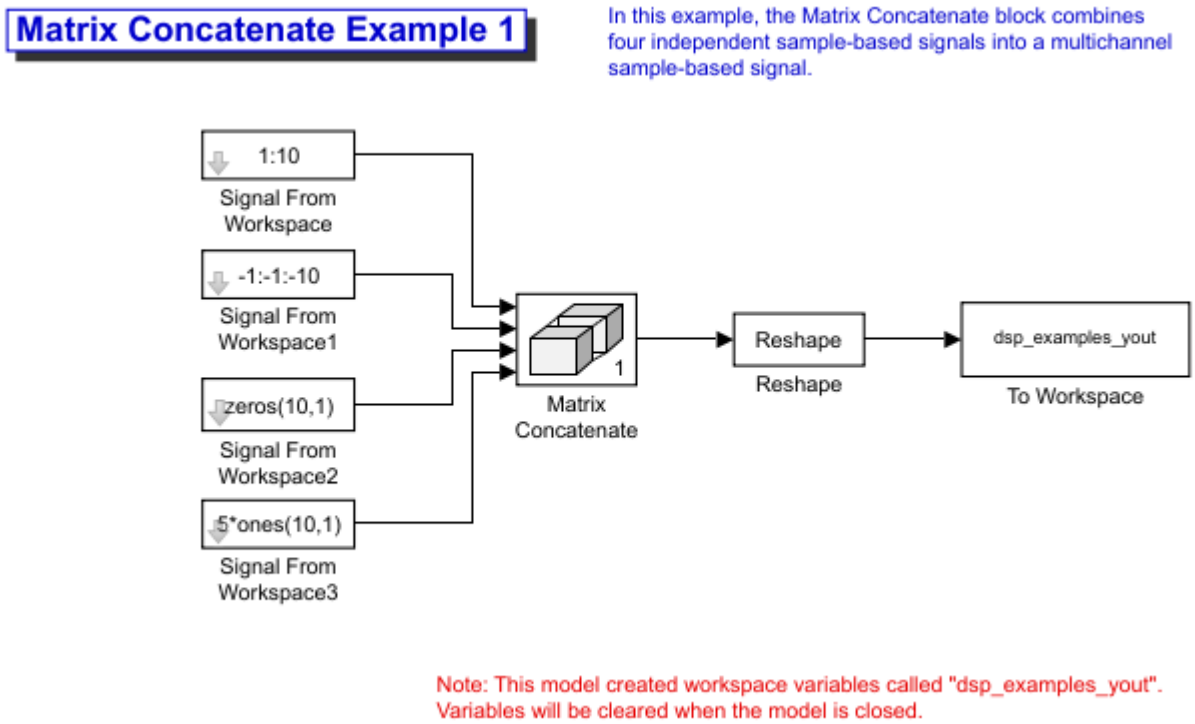
Create Multichannel Signals by Combining Single-Channel Signals

You can combine individual signals into a multichannel signal by using the Matrix Concatenate block in the Simulink Math Operations library:

1 Open the Matrix Concatenate Example 1 model by typing

```
ex_cmbsnglchsbsigs
```

at the MATLAB command line.



2 Double-click the Signal From Workspace block, and set the **Signal** parameter to `1:10`. Click **OK**.

3 Double-click the Signal From Workspace1 block, and set the **Signal** parameter to `-1:-1:-10`. Click **OK**.

4 Double-click the Signal From Workspace2 block, and set the **Signal** parameter to `zeros(10,1)`. Click **OK**.

5 Double-click the Signal From Workspace3 block, and set the **Signal** parameter to `5*ones(10,1)`. Click **OK**.

6 Double-click the Matrix Concatenate block. Set the block parameters as follows, and then click **OK**:

- **Number of inputs** = 4
- **Mode** = `Multidimensional array`
- **Concatenate dimension** = 1

7 Double-click the Reshape block. Set the block parameters as follows, and then click **OK**:

- **Output dimensionality** = `Customize`

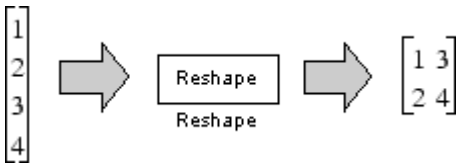
- **Output dimensions** = `[2,2]`

8 Run the model. In the **Simulation** tab, click **Run**.

Four independent signals are combined into a 2-by-2 multichannel matrix signal.

Each 4-by-1 output from the Matrix Concatenate block contains one sample from each of the four input signals at the same instant in time. The Reshape block rearranges the samples into a 2-by-2 matrix. Each element of this matrix is a separate channel.

Note that the Reshape block works column wise, so that a column vector input is reshaped as shown below.



The 4-by-1 matrix output by the Matrix Concatenate block and the 2-by-2 matrix output by the Reshape block in the above model represent the same four-channel signal. In some cases, one representation of the signal may be more useful than the other.

9 At the MATLAB command line, type `dsp_examples_yout`.

The four-channel signal is displayed as a series of matrices in the MATLAB Command Window. Note that the last matrix contains only zeros. This is because every Signal From Workspace block in this model has its **Form output after final data value by** parameter set to `Setting to Zero`.

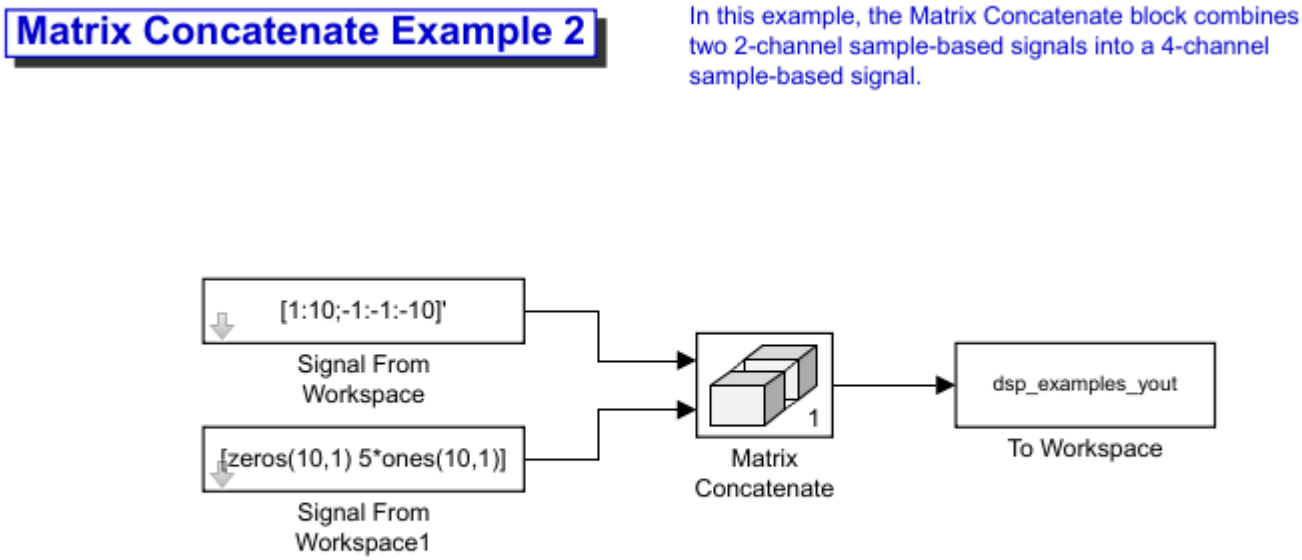
Create Multichannel Signals by Combining Multichannel Signals

You can combine existing multichannel signals into larger multichannel signals using the Simulink Matrix Concatenate block:

1 Open the Matrix Concatenate Example 2 model by typing

```
ex_cmbmltichsbsigs
```

at the MATLAB command line.



2. Double-click the Signal From Workspace block, and set the **Signal** parameter to `[1:10;-1:-1:-10]'`. Click **OK**.
3. Double-click the Signal From Workspace1 block, and set the **Signal** parameter to `[zeros(10,1) 5*ones(10,1)]`. Click **OK**.
4. Double-click the Matrix Concatenate block. Set the block parameters as follows, and then click **OK**:
   - **Number of inputs** = 2
   - **Mode** = `Multidimensional array`
   - **Concatenate dimension** = 1
5. Run the model.

   The model combines both two-channel signals into a four-channel signal.

   Each 2-by-2 output from the Matrix Concatenate block contains both samples from each of the two input signals at the same instant in time. Each element of this matrix is a separate channel.

See Also

More About

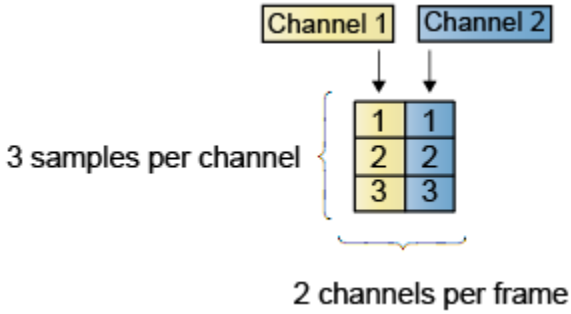
- “Sample- and Frame-Based Concepts” on page 3-2
- “Create Signals for Sample-Based Processing” on page 2-9
- “Create Signals for Frame-Based Processing” on page 2-13
- “Create Multichannel Signals for Frame-Based Processing” on page 2-23

- “Deconstruct Multichannel Signals for Sample-Based Processing” on page 2-27
- “Deconstruct Multichannel Signals for Frame-Based Processing” on page 2-32
- “Sample- and Frame-Based Concepts” on page 3-2

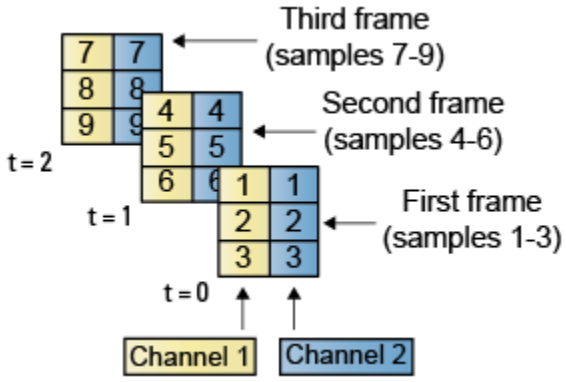
Create Multichannel Signals for Frame-Based Processing

In this section...
“Multichannel Signals for Frame-Based Processing” on page 2-24
“Create Multichannel Signals Using Concatenate Block” on page 2-24

In frame-based processing, blocks process data one frame at a time. Each frame of data contains sequential samples from an independent channel. Each channel is represented by a column of the input signal. For example, from a frame-based processing perspective, the following 3-by-2 matrix has two channels, each of which contains three samples.



When you configure a block to perform frame-based processing, the block interprets an *M*-by-1 vector as a single-channel signal containing *M* samples per frame. Similarly, the block interprets an *M*-by-*N* matrix as a multichannel signal with *N* independent channels and *M* samples per channel. For example, in frame-based processing, blocks interpret the following sequence of 3-by-2 matrices as a two-channel signal with a frame size of 3.



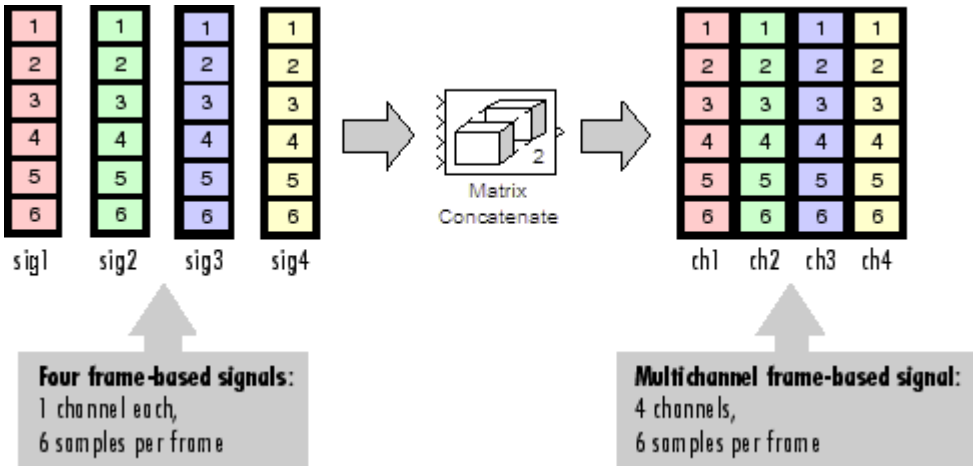
Using frame-based processing is advantageous for many signal processing applications because you can process multiple samples at once. By buffering your data into frames and processing multisample frames of data, you can often improve the computational time of your signal processing algorithms. To perform frame-based processing, you must have a DSP System Toolbox license.

For more information about the recent changes to frame-based processing, see the “Frame-based processing changes” section of the *DSP System Toolbox Release Notes*.

Multichannel Signals for Frame-Based Processing

When you want to perform the same operations on several independent signals, you can group those signals together as a multichannel signal. For example, if you need to filter each of four independent signals using the same direct-form II transposed filter, you can combine the signals into a multichannel signal, and connect the signal to a single Biquad Filter block. The block decides to treat each column of the input as a channel when you set the block's **Input processing** parameter to `Columns as channels (frame based)`. The block then applies the filter to each channel independently.

A signal with N channels and frame size M is represented by a matrix of size *M*-by-*N*. Multiple individual signals with the same frame rate and frame size can be combined into a single multichannel signal using the Simulink Matrix Concatenate block. Individual signals can be added to an existing multichannel signal in the same way.



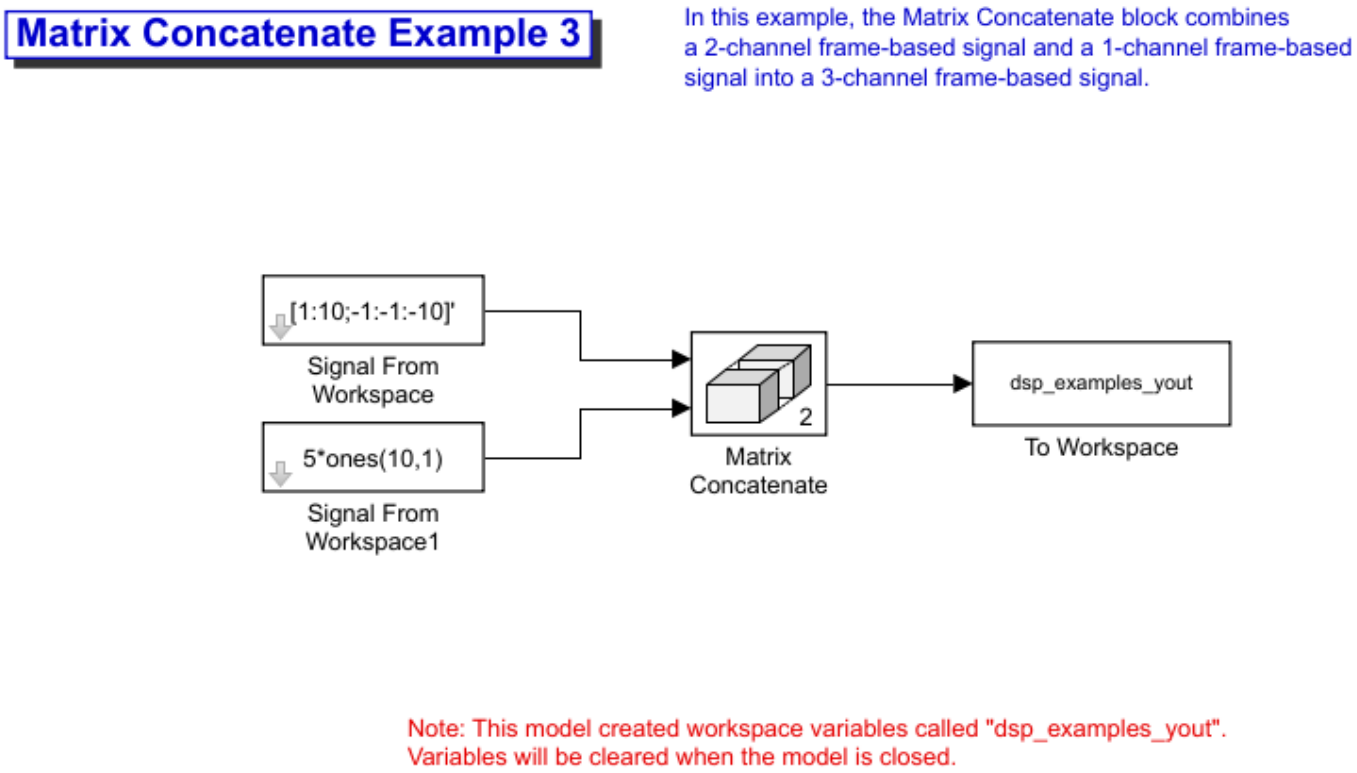
Create Multichannel Signals Using Concatenate Block

You can combine independent signals into a larger multichannel signal by using the Simulink Concatenate block. All signals must have the same frame rate and frame size. In this example, a single-channel signal is combined with a two-channel signal to produce a three-channel signal:

1. Open the Matrix Concatenate Example 3 model by typing

   ```
   ex_combiningfbsigs
   ```

   at the MATLAB command line.



2 Double-click the Signal From Workspace block. Set the block parameters as follows:

- **Signal** = `[1:10;-1:-1:-10]'`
- **Sample time** = `1`
- **Samples per frame** = `4`

Based on these parameters, the Signal From Workspace block outputs a signal with a frame size of four.

3 Save these parameters and close the dialog box by clicking **OK**.

4 Double-click the Signal From Workspace1 block. Set the block parameters as follows, and then click **OK**:

- **Signal** = `5*ones(10,1)`
- **Sample time** = `1`
- **Samples per frame** = `4`

The Signal From Workspace1 block has the same sample time and frame size as the Signal From Workspace block. To combine single-channel signals into a multichannel signal, the signals must have the same frame rate and the same frame size.

5 Double-click the Matrix Concatenate block. Set the block parameters as follows, and then click **OK**:

- **Number of inputs** = `2`
- **Mode** = `Multidimensional array`

- **Concatenate dimension** = 2

6 Run the model.

The 4-by-3 matrix output from the Matrix Concatenate block contains all three input channels, and preserves their common frame rate and frame size.

See Also

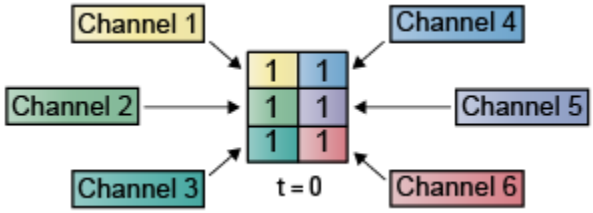
More About

- “Sample- and Frame-Based Concepts” on page 3-2
- “Create Signals for Sample-Based Processing” on page 2-9
- “Create Signals for Frame-Based Processing” on page 2-13
- “Create Multichannel Signals for Sample-Based Processing” on page 2-18
- “Deconstruct Multichannel Signals for Sample-Based Processing” on page 2-27
- “Deconstruct Multichannel Signals for Frame-Based Processing” on page 2-32
- “Sample- and Frame-Based Concepts” on page 3-2

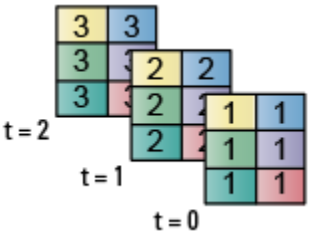
Deconstruct Multichannel Signals for Sample-Based Processing

In this section...
“Split Multichannel Signals into Individual Signals” on page 2-27
“Split Multichannel Signals into Several Multichannel Signals” on page 2-29

In sample-based processing, blocks process signals one sample at a time. Each element of the input signal represents one sample in a distinct channel. For example, from a sample-based processing perspective, the following 3-by-2 matrix contains the first sample in each of six independent channels.



When you configure a block to perform sample-based processing, the block interprets scalar input as a single-channel signal. Similarly, the block interprets an *M*-by-*N* matrix as multichannel signal with *M***N* independent channels. For example, in sample-based processing, blocks interpret the following sequence of 3-by-2 matrices as a six-channel signal.



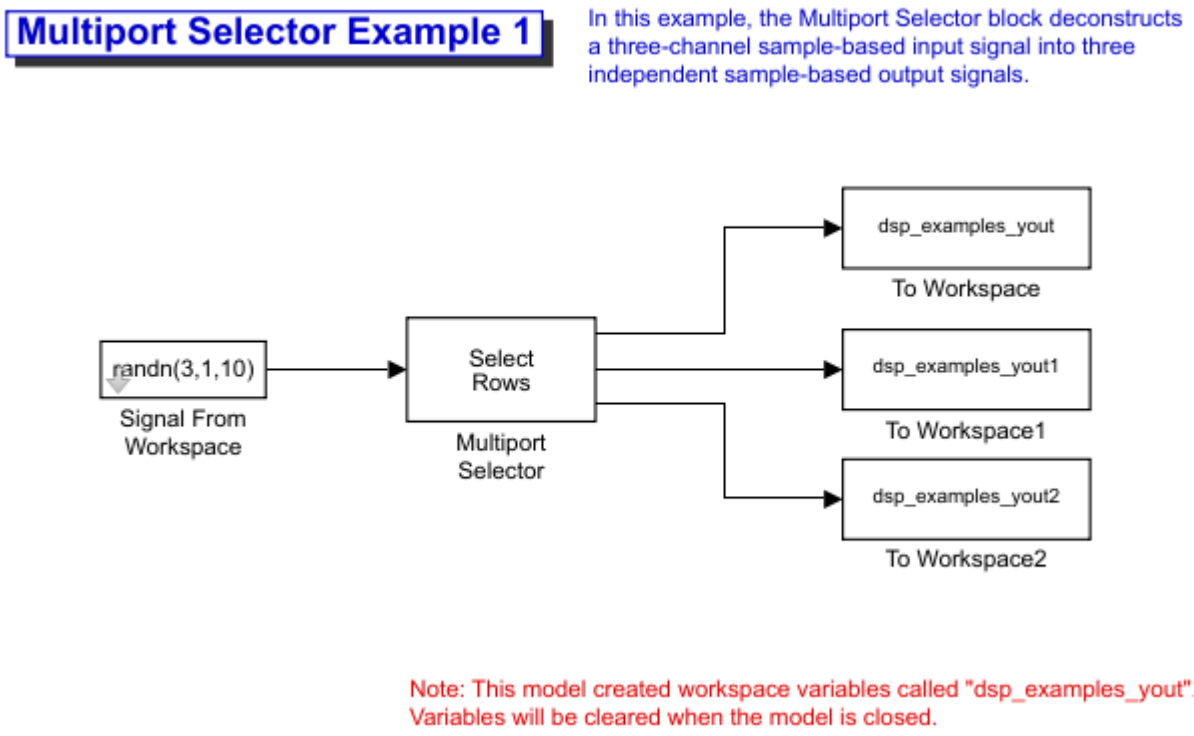
For more information about the recent changes to frame-based processing, see the “Frame-based processing changes” section of the *DSP System Toolbox Release Notes*.

Split Multichannel Signals into Individual Signals

Multichannel signals, represented by matrices in the Simulink environment, are frequently used in signal processing models for efficiency and compactness. Though most of the signal processing blocks can process multichannel signals, you may need to access just one channel or a particular range of samples in a multichannel signal. You can access individual channels of the multichannel signal by using the blocks in the Indexing library. This library includes the Selector, Submatrix, Variable Selector, Multiport Selector, and Submatrix blocks.

You can split a multichannel based signal into single-channel signals using the Multiport Selector block. This block allows you to select specific rows and/or columns and propagate the selection to a chosen output port. In this example, a three-channel signal of size 3-by-1 is deconstructed into three independent signals of sample period 1 second.

1 Open the Multiport Selector Example 1 model by typing `ex_splitmltichsbsigsind` at the MATLAB command line.



2 Double-click the Signal From Workspace block, and set the block parameters as follows:

- **Signal** = `randn(3,1,10)`
- **Sample time** = `1`
- **Samples per frame** = `1`

Based on these parameters, the Signal From Workspace block outputs a three-channel signal with a sample period of 1 second.

3 Save these parameters and close the dialog box by clicking **OK**.

4 Double-click the Multiport Selector block. Set the block parameters as follows, and then click **OK**:

- **Select** = `Rows`
- **Indices to output** = `{1,2,3}`

Based on these parameters, the Multiport Selector block extracts the rows of the input. The **Indices to output** parameter setting specifies that row 1 of the input should be reproduced at output 1, row 2 of the input should be reproduced at output 2, and row 3 of the input should be reproduced at output 3.

5 Run the model.

6 At the MATLAB command line, type `dsp_examples_yout`.

The following is a portion of what is displayed at the MATLAB command line. Because the input signal is random, your output might be different than the output show here.

```
dsp_examples_yout(:,:,1) =
```

```
   -0.1199

dsp_examples_yout(:,:,2) =

   -0.5955

dsp_examples_yout(:,:,3) =

   -0.0793
```

This signal is the first row of the input to the Multiport Selector block. You can view the other two input rows by typing `dsp_examples_yout1` and `dsp_examples_yout2`, respectively.

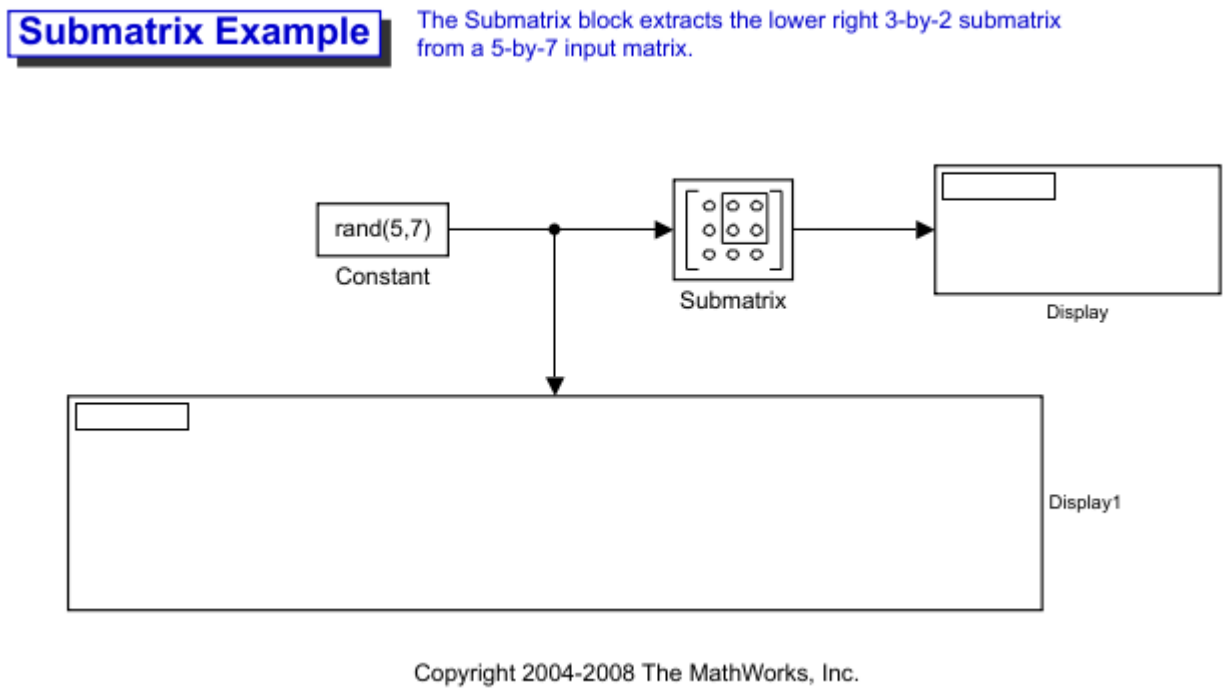
You have now successfully created three single-channel signals from a multichannel signal using a Multiport Selector block.

Split Multichannel Signals into Several Multichannel Signals

Multichannel signals, represented by matrices in the Simulink environment, are frequently used in signal processing models for efficiency and compactness. Though most of the signal processing blocks can process multichannel signals, you may need to access just one channel or a particular range of samples in a multichannel signal. You can access individual channels of the multichannel signal by using the blocks in the Indexing library. This library includes the Selector, Submatrix, Variable Selector, Multiport Selector, and Submatrix blocks.

You can split a multichannel signal into other multichannel signals using the Submatrix block. The Submatrix block is the most versatile of the blocks in the Indexing library because it allows arbitrary channel selections. Therefore, you can extract a portion of a multichannel signal. In this example, you extract a six-channel signal from a 35-channel signal (a matrix of size 5-by-7). Each channel contains one sample.

1 Open the Submatrix Example model by typing `ex_splitmltichsbsigsev` at the MATLAB command line.



2 Double-click the Constant block, and set the block parameters as follows:

- **Constant value** = `rand(5,7)`
- **Interpret vector parameters as 1-D** = Clear this check box
- **Sample Time** = `1`

Based on these parameters, the Constant block outputs a constant-valued signal.

3 Save these parameters and close the dialog box by clicking **OK**.

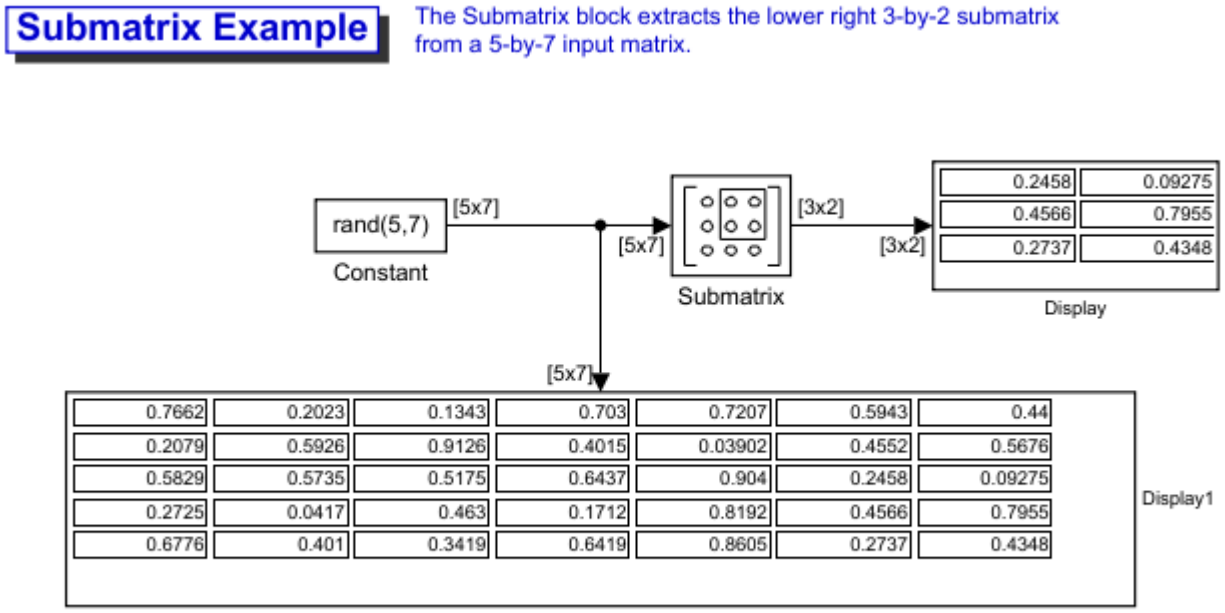
4 Double-click the Submatrix block. Set the block parameters as follows, and then click **OK**:

- **Row span** = `Range of rows`
- **Starting row** = `Index`
- **Starting row index** = `3`
- **Ending row** = `Last`
- **Column span** = `Range of columns`
- **Starting column** = `Offset from last`
- **Starting column offset** = `1`
- **Ending column** = `Last`

Based on these parameters, the Submatrix block outputs rows three to five, the last row of the input signal. It also outputs the second to last column and the last column of the input signal.

5 Run the model.

The model should now look similar to the following figure.



Notice that the output of the Submatrix block is equivalent to the matrix created by rows three through five and columns six through seven of the input matrix.

You have now successfully created a six-channel signal from a 35-channel signal using a Submatrix block. Each channel contains one sample.

See Also

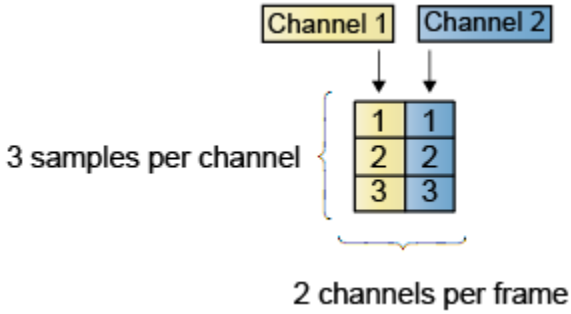
More About

- “Sample- and Frame-Based Concepts” on page 3-2
- “Create Signals for Sample-Based Processing” on page 2-9
- “Create Signals for Frame-Based Processing” on page 2-13
- “Create Multichannel Signals for Sample-Based Processing” on page 2-18
- “Create Multichannel Signals for Frame-Based Processing” on page 2-23
- “Deconstruct Multichannel Signals for Frame-Based Processing” on page 2-32
- “Import and Export Signals for Sample-Based Processing” on page 2-38
- “Import and Export Signals for Frame-Based Processing” on page 2-47
- “Inspect Sample and Frame Rates in Simulink” on page 3-6
- “Convert Sample and Frame Rates in Simulink” on page 3-13

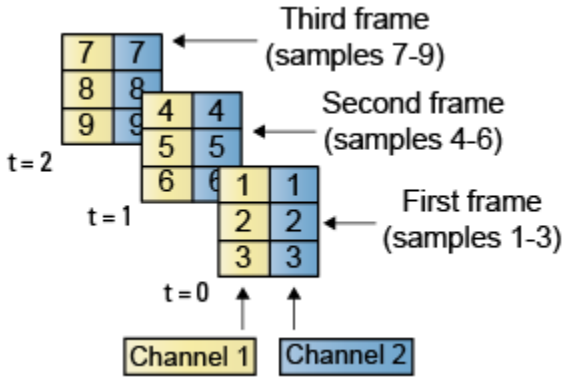
Deconstruct Multichannel Signals for Frame-Based Processing

In this section...
“Split Multichannel Signals into Individual Signals” on page 2-33
“Reorder Channels in Multichannel Signals” on page 2-35

In frame-based processing, blocks process data one frame at a time. Each frame of data contains sequential samples from an independent channel. Each channel is represented by a column of the input signal. For example, from a frame-based processing perspective, the following 3-by-2 matrix has two channels, each of which contains three samples.



When you configure a block to perform frame-based processing, the block interprets an *M*-by-1 vector as a single-channel signal containing *M* samples per frame. Similarly, the block interprets an *M*-by-*N* matrix as a multichannel signal with *N* independent channels and *M* samples per channel. For example, in frame-based processing, blocks interpret the following sequence of 3-by-2 matrices as a two-channel signal with a frame size of 3.



Using frame-based processing is advantageous for many signal processing applications because you can process multiple samples at once. By buffering your data into frames and processing multisample frames of data, you can often improve the computational time of your signal processing algorithms. To perform frame-based processing, you must have a DSP System Toolbox license.

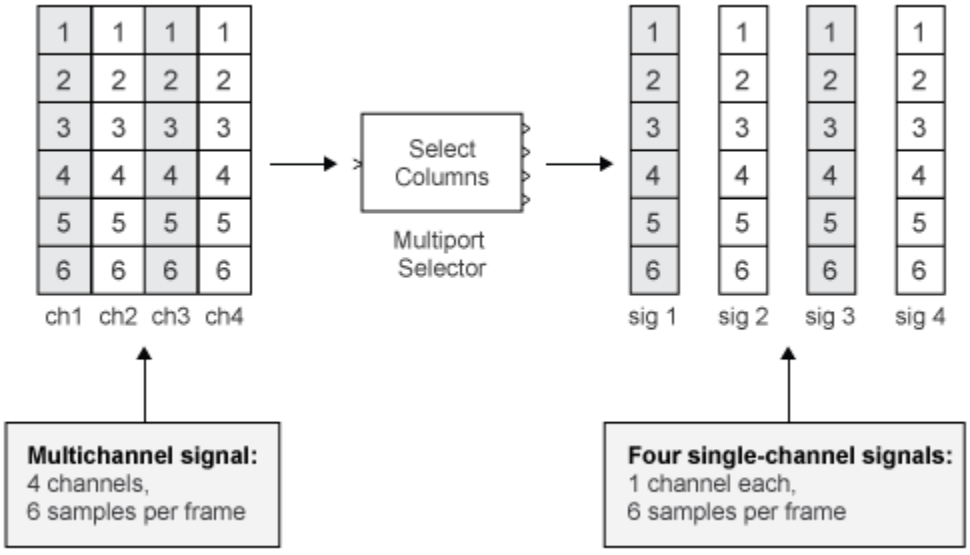
For more information about the recent changes to frame-based processing, see the “Frame-based processing changes” section of the *DSP System Toolbox Release Notes*.

Split Multichannel Signals into Individual Signals

Multichannel signals, represented by matrices in the Simulink environment, are frequently used in signal processing models for efficiency and compactness. Though most of the signal processing blocks can process multichannel signals, you may need to access just one channel or a particular range of samples in a multichannel signal. You can access individual channels of the multichannel signal by using the blocks in the Indexing library. This library includes the Selector, Submatrix, Variable Selector, Multiport Selector, and Submatrix blocks. It is also possible to use the Permute Matrix block, in the Matrix operations library, to reorder the channels of a frame-based signal.

You can use the Multiport Selector block in the Indexing library to extract the individual channels of a multichannel signal. These signals form single-channel signals that have the same frame rate and frame size of the multichannel signal.

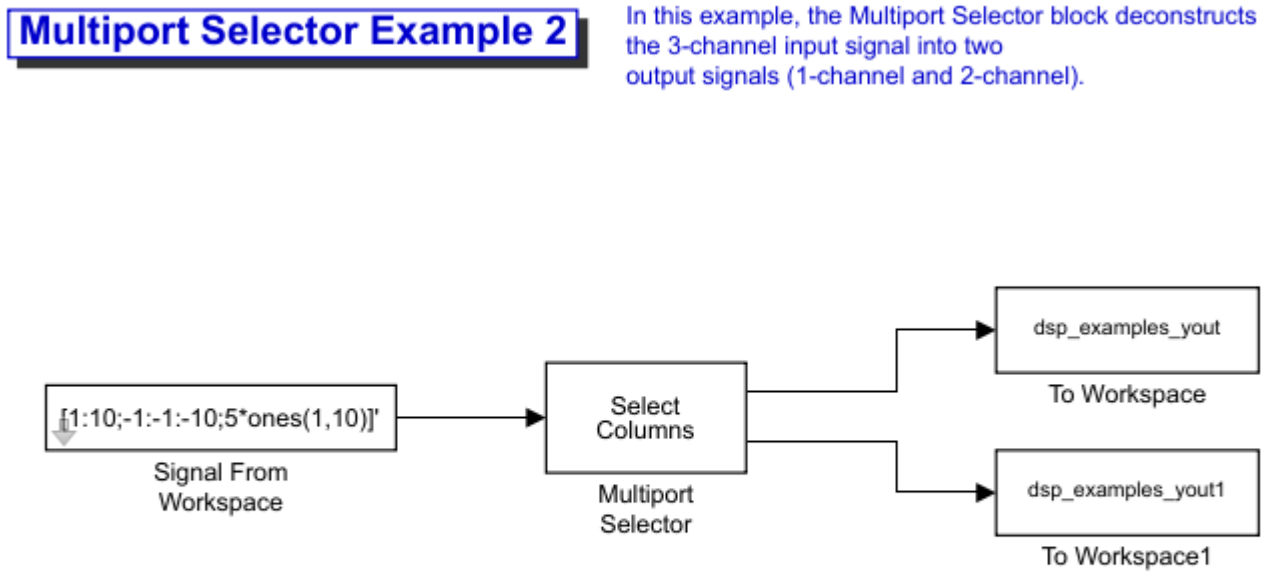
The figure below is a graphical representation of this process.



In this example, you use the Multiport Selector block to extract a single-channel signal and a two channel signal from a multichannel signal. Each channel contains four samples.

1 Open the Multiport Selector Example 2 model by typing `ex_splitmltichfbsigsind`

at the MATLAB command line.



2 Double-click the Signal From Workspace block, and set the block parameters as follows:

   - **Signal** = `[1:10;-1:-1:-10;5*ones(1,10)]'`
   - **Samples per frame** = 4

   Based on these parameters, the Signal From Workspace block outputs a three-channel signal with a frame size of four.

3 Save these parameters and close the dialog box by clicking **OK**.

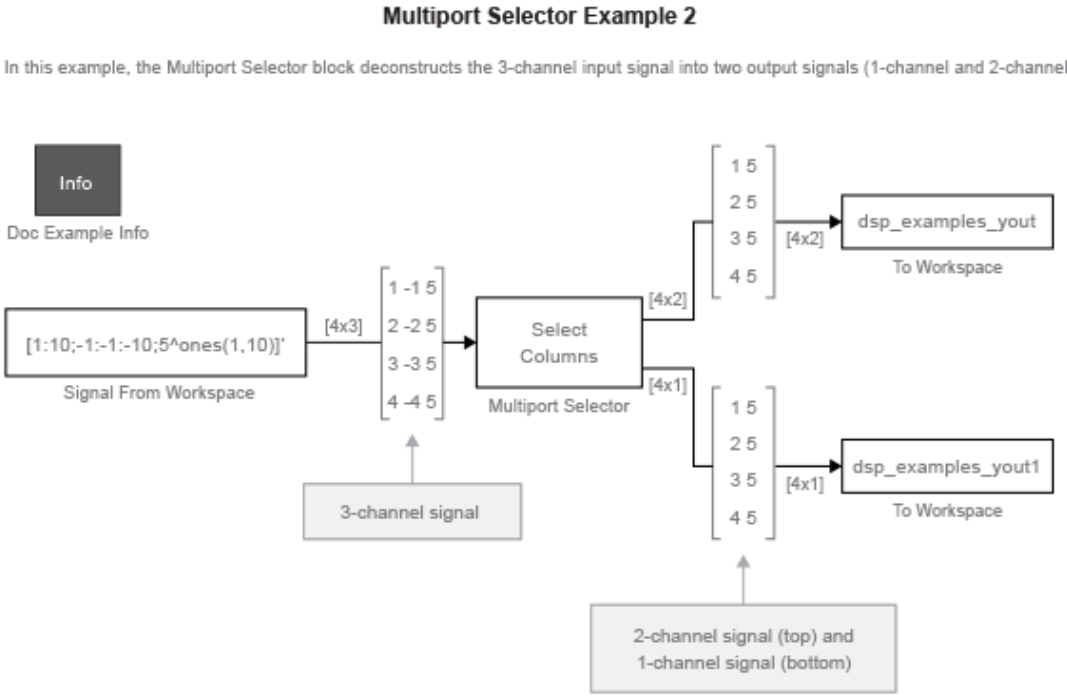
4 Double-click the Multiport Selector block. Set the block parameters as follows, and then click **OK**:

   - **Select** = `Columns`
   - **Indices to output** = `{[1 3],2}`

   Based on these parameters, the Multiport Selector block outputs the first and third columns at the first output port and the second column at the second output port of the block. Setting the **Select** parameter to `Columns` ensures that the block preserves the frame rate and frame size of the input.

5 Run the model.

   The figure below is a graphical representation of how the Multiport Selector block splits one frame of the three-channel signal into a single-channel signal and a two-channel signal.



Note: This model creates workspace variables called "dsp_examples_yout" and "dsp_examples_yout1".

The Multiport Selector block outputs a two-channel signal, comprised of the first and third column of the input signal, at the first port. It outputs a single-channel comprised of the second column of the input signal, at the second port.

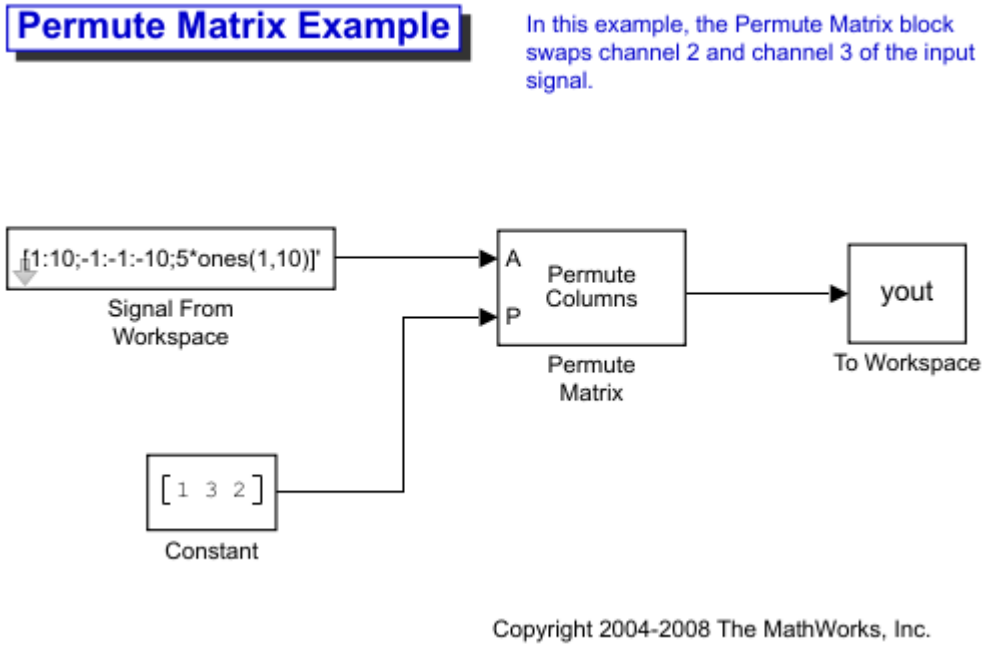
You have now successfully created a single-channel signal and a two-channel signal from a multichannel signal using the Multiport Selector block.

Reorder Channels in Multichannel Signals

Multichannel signals, represented by matrices in Simulink, are frequently used in signal processing models for efficiency and compactness. Though most of the signal processing blocks can process multichannel signals, you may need to access just one channel or a particular range of samples in a multichannel signal. You can access individual channels of the multichannel signal by using the blocks in the Indexing library. This library includes the Selector, Submatrix, Variable Selector, Multiport Selector, and Submatrix blocks. It is also possible to use the Permute Matrix block, in the Matrix operations library, to reorder the channels of a frame signal.

Some DSP System Toolbox blocks have the ability to process the interaction of channels. Typically, DSP System Toolbox blocks compare channel one of signal A to channel one of signal B. However, you might want to correlate channel one of signal A with channel three of signal B. In this case, in order to compare the correct signals, you need to use the Permute Matrix block to rearrange the channels of your signals. This example explains how to accomplish this task.

1 Open the Permute Matrix Example model by typing `ex_reordermltichfbsigs` at the MATLAB command line.



2 Double-click the Signal From Workspace block, and set the block parameters as follows:

- **Signal** = `[1:10;-1:-1:-10;5*ones(1,10)]'`
- **Sample time** = `1`
- **Samples per frame** = `4`

Based on these parameters, the Signal From Workspace block outputs a three-channel signal with a sample period of 1 second and a frame size of 4. The frame period of this block is 4 seconds.

3 Save these parameters and close the dialog box by clicking **OK**.

4 Double-click the Constant block. Set the block parameters as follows, and then click **OK**:

- **Constant value** = `[1 3 2]`
- **Interpret vector parameters as 1-D** = Clear this check box
- **Sample time** = `4`

The discrete-time vector output by the Constant block tells the Permute Matrix block to swap the second and third columns of the input signal. Note that the frame period of the Constant block must match the frame period of the Signal From Workspace block.

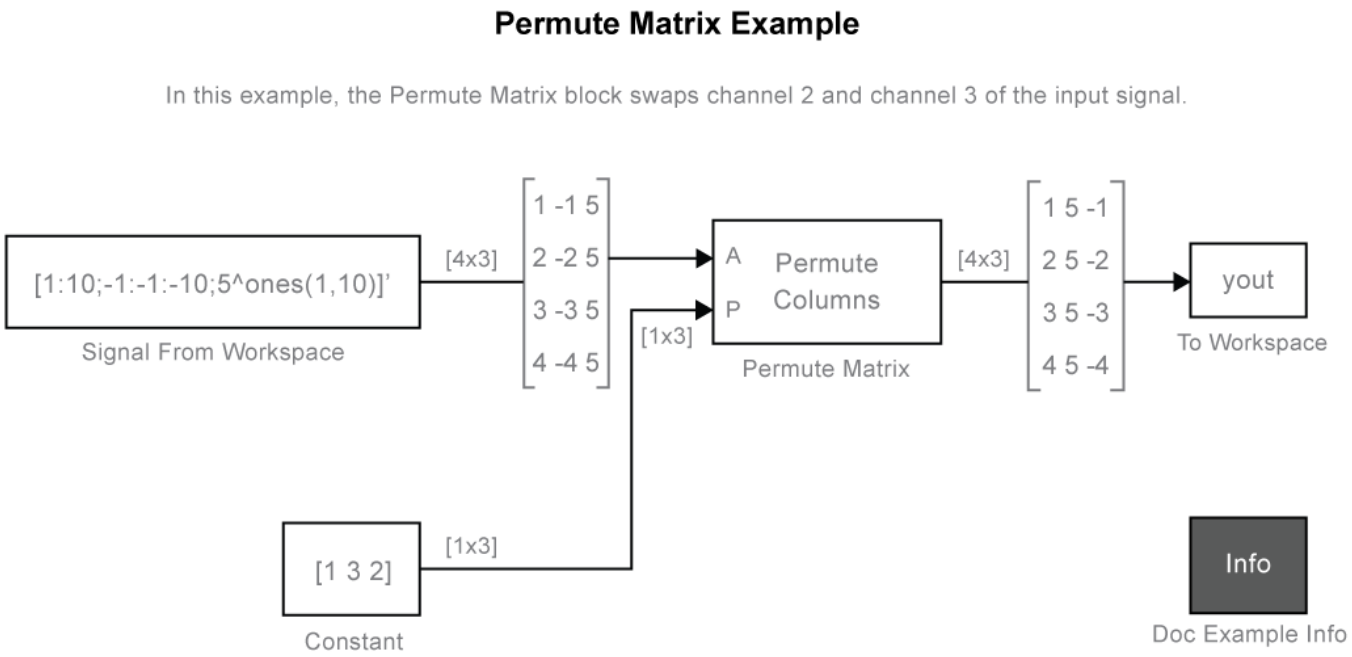
5 Double-click the Permute Matrix block. Set the block parameters as follows, and then click **OK**:

- **Permute** = `Columns`
- **Index mode** = `One-based`

Based on these parameters, the Permute Matrix block rearranges the columns of the input signal, and the index of the first column is now one.

6 Run the model.

The figure below is a graphical representation of what happens to the first input frame during simulation.



The second and third channel of the input signal are swapped.

7 At the MATLAB command line, type `yout`.

You can now verify that the second and third columns of the input signal are rearranged.

You have now successfully reordered the channels of a frame signal using the Permute Matrix block.

See Also

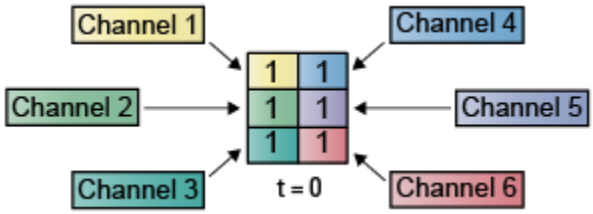
More About

- “Sample- and Frame-Based Concepts” on page 3-2
- “Create Signals for Sample-Based Processing” on page 2-9
- “Create Signals for Frame-Based Processing” on page 2-13
- “Create Multichannel Signals for Sample-Based Processing” on page 2-18
- “Create Multichannel Signals for Frame-Based Processing” on page 2-23
- “Deconstruct Multichannel Signals for Sample-Based Processing” on page 2-27
- “Import and Export Signals for Sample-Based Processing” on page 2-38
- “Import and Export Signals for Frame-Based Processing” on page 2-47
- “Inspect Sample and Frame Rates in Simulink” on page 3-6
- “Convert Sample and Frame Rates in Simulink” on page 3-13

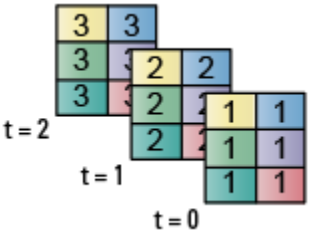
Import and Export Signals for Sample-Based Processing

In this section...
“Import Vector Signals for Sample-Based Processing” on page 2-38
“Import Matrix Signals for Sample-Based Processing” on page 2-40
“Export Signals for Sample-Based Processing” on page 2-43

In sample-based processing, blocks process signals one sample at a time. Each element of the input signal represents one sample in a distinct channel. For example, from a sample-based processing perspective, the following 3-by-2 matrix contains the first sample in each of six independent channels.



When you configure a block to perform sample-based processing, the block interprets scalar input as a single-channel signal. Similarly, the block interprets an *M*-by-*N* matrix as multichannel signal with *M***N* independent channels. For example, in sample-based processing, blocks interpret the following sequence of 3-by-2 matrices as a six-channel signal.

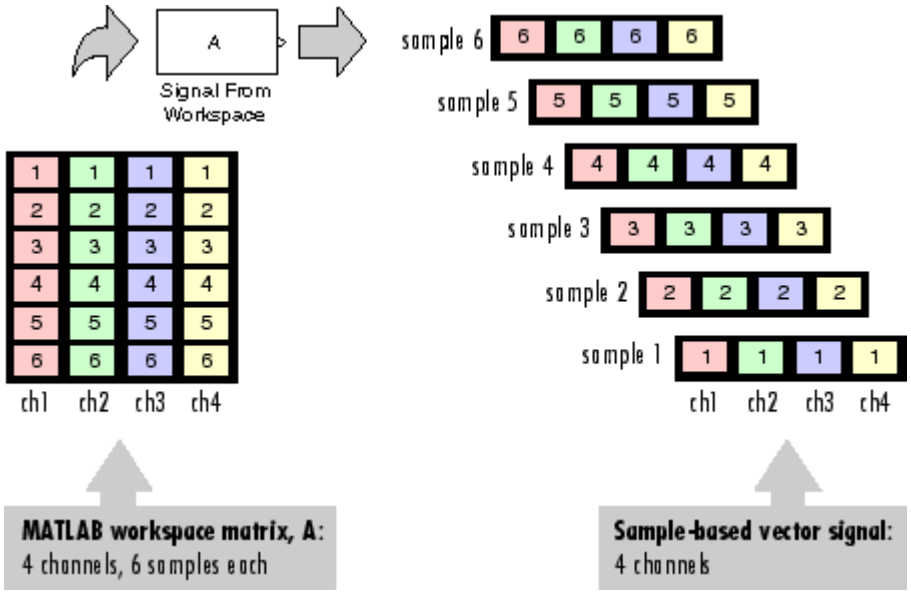


For more information about the recent changes to frame-based processing, see the “Frame-based processing changes” section of the *DSP System Toolbox Release Notes*.

Import Vector Signals for Sample-Based Processing

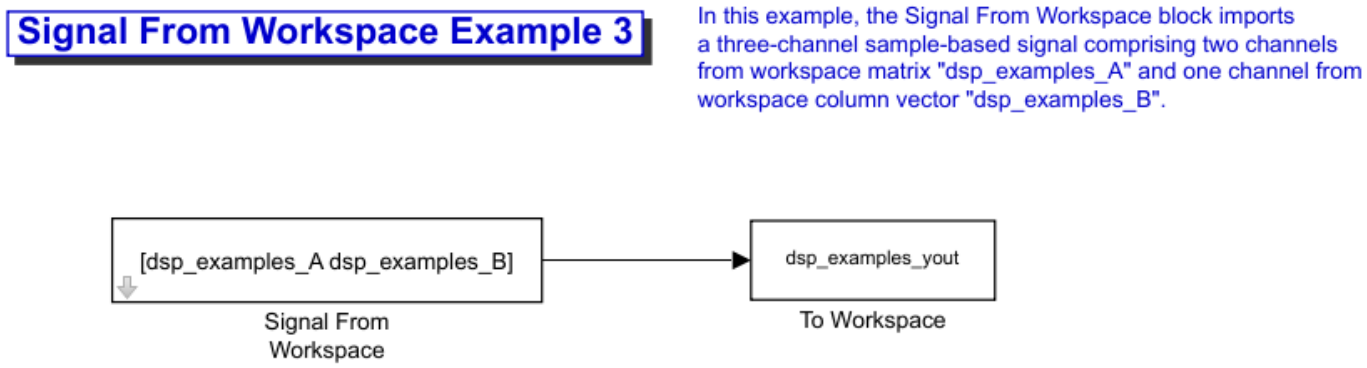
The Signal From Workspace block generates a vector signal for sample-based processing when the variable or expression in the **Signal** parameter is a matrix and the **Samples per frame** parameter is set to 1. Each column of the input matrix represents a different channel. Beginning with the first row of the matrix, the block outputs one row of the matrix at each sample time. Therefore, if the **Signal** parameter specifies an M-by-N matrix, the output of the Signal From Workspace block is M 1-by-N row vectors representing N channels.

The figure below is a graphical representation of this process for a 6-by-4 workspace matrix, A.



In the following example, you use the Signal From Workspace block to import the vector signal into your model.

1 Open the Signal From Workspace Example 3 model by typing `ex_importsbvectorsigs` at the MATLAB command line.



2 At the MATLAB command line, type `A = [1:100;-1:-1:-100]';`

The matrix `A` represents a two column signal, where each column is a different channel.

3 At the MATLAB command line, type `B = 5*ones(100,1);`

The vector `B` represents a single-channel signal.

4 Double-click the Signal From Workspace block, and set the block parameters as follows:

- **Signal** = `[A B]`

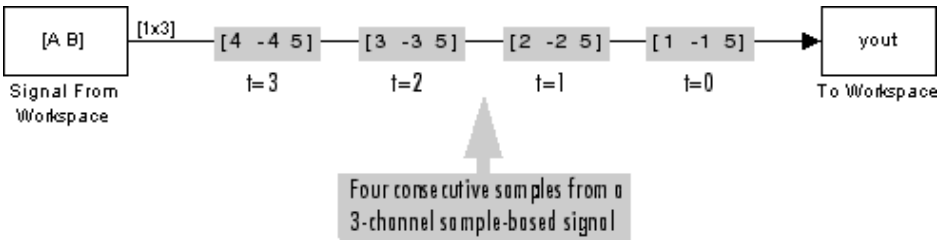
- **Sample time** = 1
- **Samples per frame** = 1
- **Form output after final data value** = `Setting to zero`

The **Signal** expression `[A B]` uses the standard MATLAB syntax for horizontally concatenating matrices and appends column vector `B` to the right of matrix `A`. The Signal From Workspace block outputs a signal with a sample period of 1 second. After the block has output the signal, all subsequent outputs have a value of zero.

5 Save these parameters and close the dialog box by clicking **OK**.

6 Run the model.

The following figure is a graphical representation of the model's behavior during simulation.



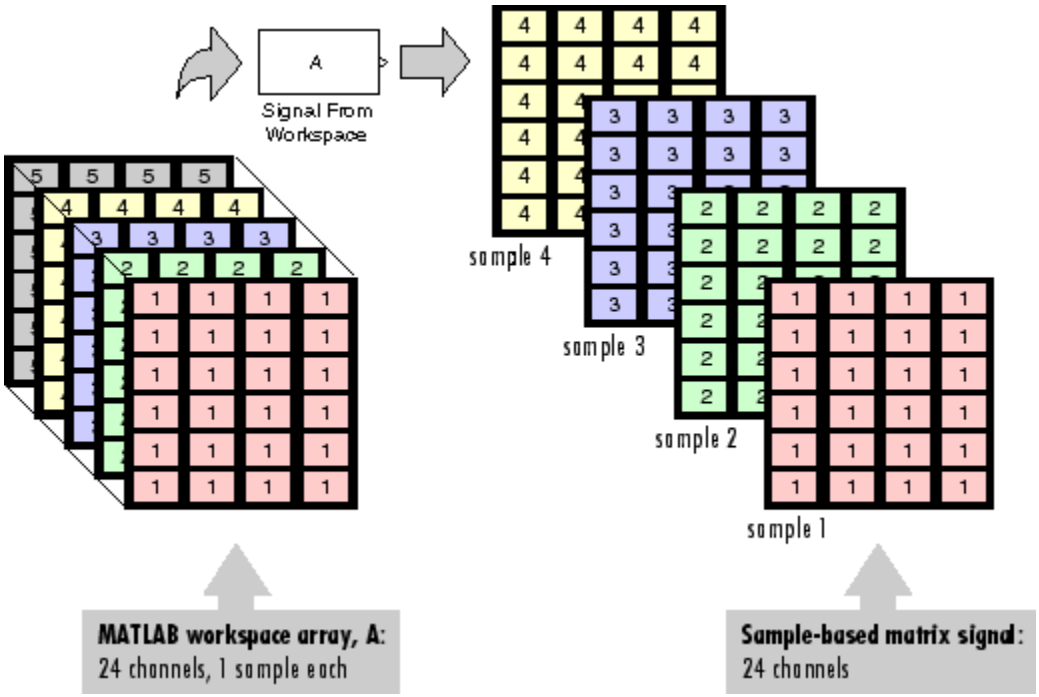
The first row of the input matrix `[A B]` is output at time `t=0`, the second row of the input matrix is output at time `t=1`, and so on.

You have now successfully imported a vector signal with three channels into your signal processing model using the Signal From Workspace block.

Import Matrix Signals for Sample-Based Processing

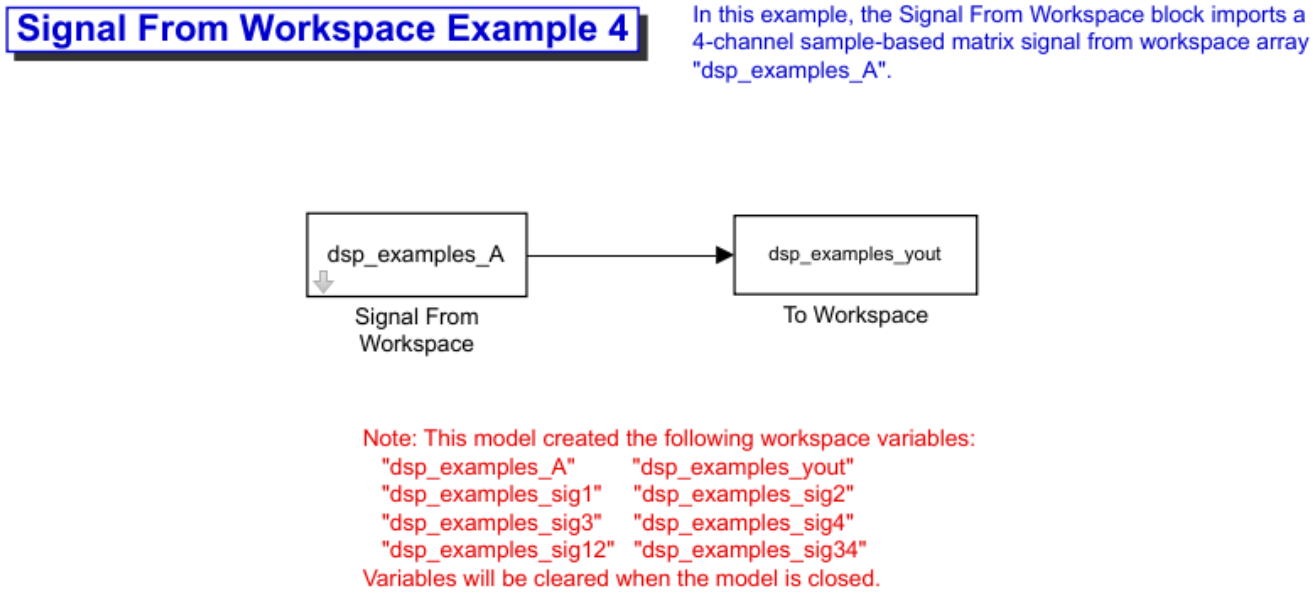
The Signal From Workspace block generates a matrix signal that is convenient for sample-based processing. Beginning with the first page of the array, the block outputs a single page of the array to the output at each sample time. Therefore, if the **Signal** parameter specifies an M-by-N-by-P array, the output of the Signal From Workspace block is P M-by-N matrices representing M*N channels. The block receiving this signal does sample-based processing or frame-based processing on the signal based on the parameters set in the block dialog box.

The following figure is a graphical illustration of this process for a 6-by-4-by-5 workspace array `A`.



In the following example, you use the Signal From Workspace block to import a four-channel matrix signal into a Simulink model.

1 Open the Signal From Workspace Example 4 model by typing `ex_importsbmatrixsigs` at the MATLAB command line.



Also, the following variables are loaded into the MATLAB workspace:

```
Fs                      1x1             8           double array
```

```
dsp_examples_A          2x2x100       3200        double array
dsp_examples_sig1       1x1x100       800         double array
dsp_examples_sig12      1x2x100       1600        double array
dsp_examples_sig2       1x1x100       800         double array
dsp_examples_sig3       1x1x100       800         double array
dsp_examples_sig34      1x2x100       1600        double array
dsp_examples_sig4       1x1x100       800         double array
mtlb                    4001x1        32008       double array
```

2 Double-click the Signal From Workspace block. Set the block parameters as follows, and then click **OK**:

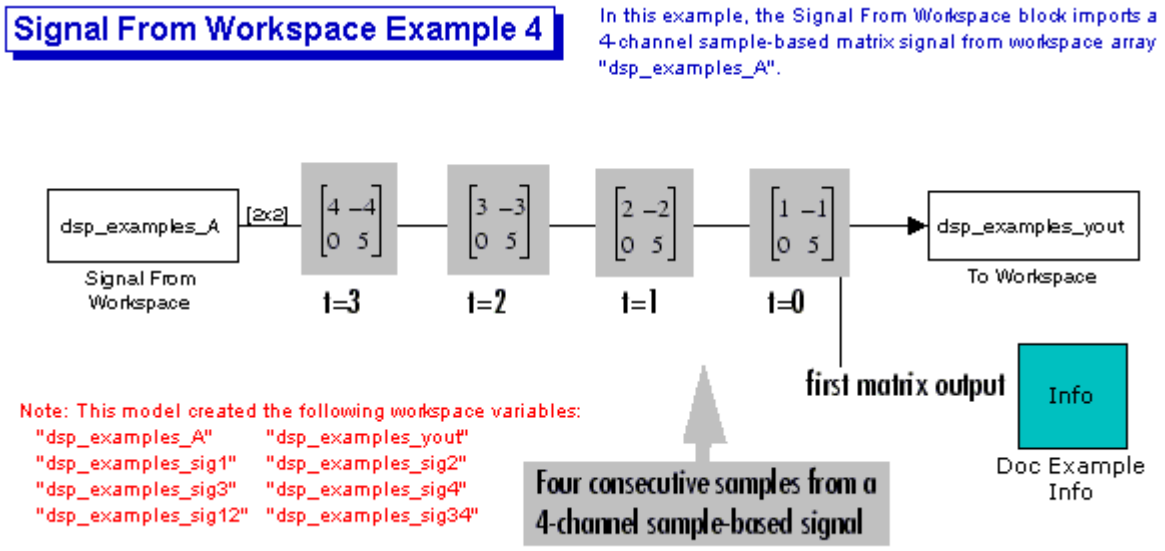
- **Signal** = `dsp_examples_A`
- **Sample time** = 1
- **Samples per frame** = 1
- **Form output after final data value** = `Setting to zero`

The `dsp_examples_A` array represents a four-channel signal with 100 samples in each channel. This is the signal that you want to import, and it was created in the following way:

```
dsp_examples_sig1 = reshape(1:100,[1 1 100])
dsp_examples_sig2 = reshape(-1:-1:-100,[1 1 100])
dsp_examples_sig3 = zeros(1,1,100)
dsp_examples_sig4 = 5*ones(1,1,100)
dsp_examples_sig12 = cat(2,sig1,sig2)
dsp_examples_sig34 = cat(2,sig3,sig4)
dsp_examples_A = cat(1,sig12,sig34)    % 2-by-2-by-100 array
```

3 Run the model.

The figure below is a graphical representation of the model's behavior during simulation.



The Signal From Workspace block imports the four-channel signal from the MATLAB workspace into the Simulink model one matrix at a time.

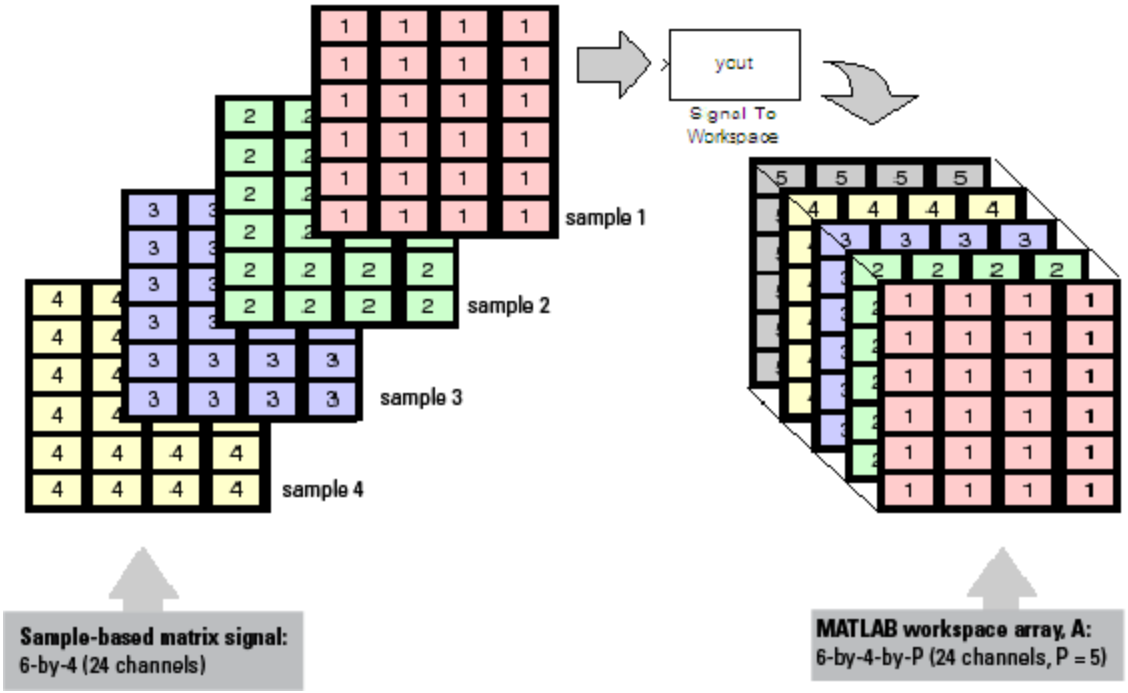
You have now successfully imported a 4-channel matrix signal into your model using the Signal From Workspace block.

Export Signals for Sample-Based Processing

The To Workspace and Triggered To Workspace blocks are the primary blocks for exporting signals of all dimensions from a Simulink model to the MATLAB workspace.

A signal with M*N channels, is represented in Simulink as a sequence of M-by-N matrices. When the input to the To Workspace block is a signal created for sample-based processing, the block creates an M-by-N-by-P array in the MATLAB workspace containing the P most recent samples from each channel. The number of pages, P, is specified by the **Limit data points to last** parameter. The newest samples are added at the end of the array.

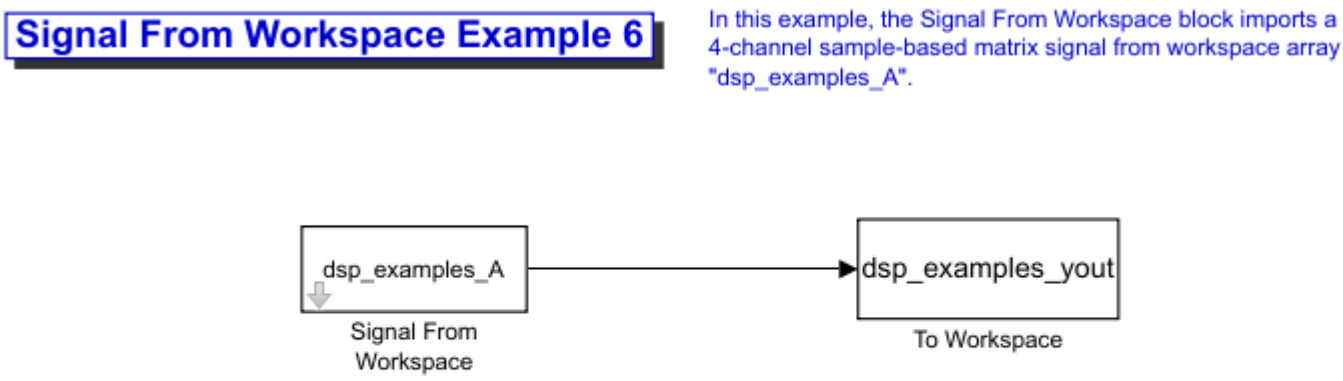
The following figure is the graphical illustration of this process using a 6-by-4 signal exported to workspace array A.



The workspace array always has time running along its third dimension, P. Samples are saved along the P dimension whether the input is a matrix, vector, or scalar (single channel case).

In the following example you use a To Workspace block to export a matrix signal to the MATLAB workspace.

1. Open the Signal From Workspace Example 6 model by typing `ex_exportsbsigs` at the MATLAB command line.



Also, the following variables are loaded into the MATLAB workspace:

```
dsp_examples_A          2x2x100      3200       double array
dsp_examples_sig1       1x1x100      800        double array
dsp_examples_sig12      1x2x100      1600       double array
dsp_examples_sig2       1x1x100      800        double array
dsp_examples_sig3       1x1x100      800        double array
dsp_examples_sig34      1x2x100      1600       double array
dsp_examples_sig4       1x1x100      800        double array
```

In this model, the Signal From Workspace block imports a four-channel matrix signal called `dsp_examples_A`. This signal is then exported to the MATLAB workspace using a To Workspace block.

2 Double-click the Signal From Workspace block. Set the block parameters as follows, and then click **OK**:

- **Signal** = `dsp_examples_A`
- **Sample time** = 1
- **Samples per frame** = 1
- **Form output after final data value** = `Setting to zero`

Based on these parameters, the Signal From Workspace block outputs a signal with a sample period of 1 second. After the block has output the signal, all subsequent outputs have a value of zero.

3 Double-click the To Workspace block. Set the block parameters as follows, and then click **OK**:

- **Variable name** = `dsp_examples_yout`
- **Limit data points to last** parameter to `inf`
- **Decimation** = 1

Based on these parameters, the To Workspace block exports its input signal to a variable called `dsp_examples_yout` in the MATLAB workspace. The workspace variable can grow indefinitely large in order to capture all of the input data. The signal is not decimated before it is exported to the MATLAB workspace.

4. Run the model.
5. At the MATLAB command line, type `dsp_examples_yout`.

   The four-channel matrix signal, `dsp_examples_A`, is output at the MATLAB command line. The following is a portion of the output that is displayed.

```
dsp_examples_yout(:,:,1) =

     1    -1
     0     5

dsp_examples_yout(:,:,2) =

     2    -2
     0     5

dsp_examples_yout(:,:,3) =

     3    -3
     0     5

dsp_examples_yout(:,:,4) =

     4    -4
     0     5
```

Each page of the output represents a different sample time, and each element of the matrices is in a separate channel.

You have now successfully exported a four-channel matrix signal from a Simulink model to the MATLAB workspace using the To Workspace block.

See Also

More About

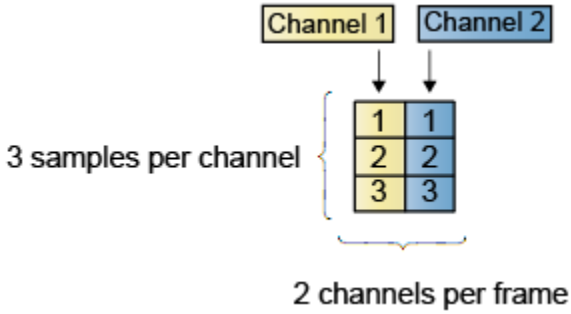
- “Sample- and Frame-Based Concepts” on page 3-2
- “Create Signals for Sample-Based Processing” on page 2-9
- “Create Signals for Frame-Based Processing” on page 2-13
- “Create Multichannel Signals for Sample-Based Processing” on page 2-18
- “Create Multichannel Signals for Frame-Based Processing” on page 2-23
- “Deconstruct Multichannel Signals for Sample-Based Processing” on page 2-27
- “Deconstruct Multichannel Signals for Frame-Based Processing” on page 2-32

- “Import and Export Signals for Frame-Based Processing” on page 2-47
- “Inspect Sample and Frame Rates in Simulink” on page 3-6
- “Convert Sample and Frame Rates in Simulink” on page 3-13

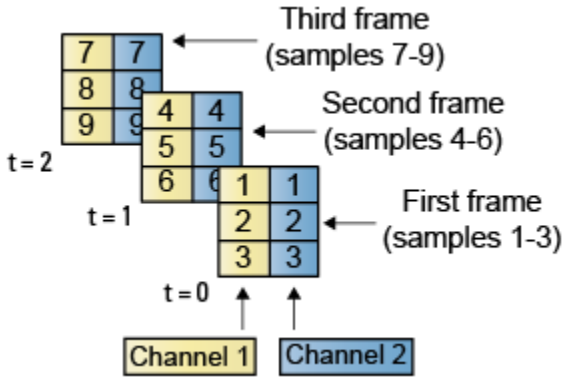
Import and Export Signals for Frame-Based Processing

In this section...
“Import Signals for Frame-Based Processing” on page 2-48
“Export Frame-Based Signals” on page 2-50

In frame-based processing, blocks process data one frame at a time. Each frame of data contains sequential samples from an independent channel. Each channel is represented by a column of the input signal. For example, from a frame-based processing perspective, the following 3-by-2 matrix has two channels, each of which contains three samples.



When you configure a block to perform frame-based processing, the block interprets an *M*-by-1 vector as a single-channel signal containing *M* samples per frame. Similarly, the block interprets an *M*-by-*N* matrix as a multichannel signal with *N* independent channels and *M* samples per channel. For example, in frame-based processing, blocks interpret the following sequence of 3-by-2 matrices as a two-channel signal with a frame size of 3.



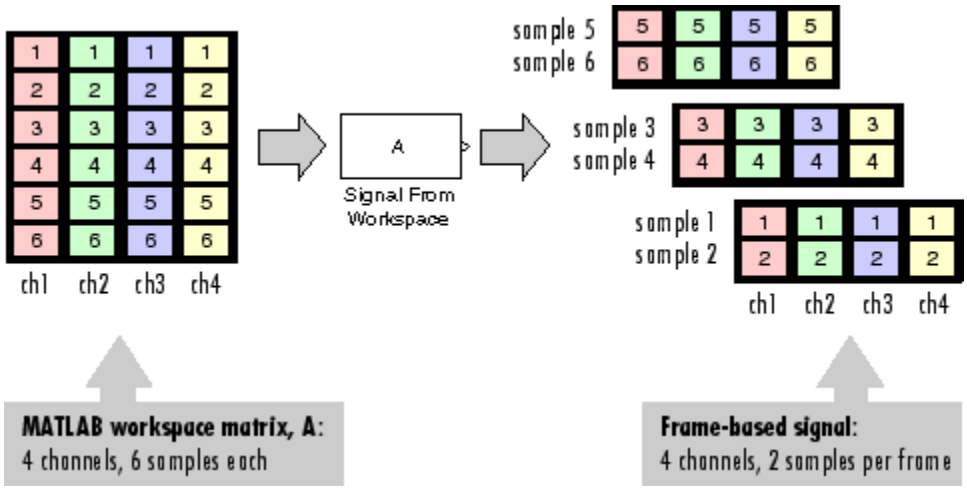
Using frame-based processing is advantageous for many signal processing applications because you can process multiple samples at once. By buffering your data into frames and processing multisample frames of data, you can often improve the computational time of your signal processing algorithms. To perform frame-based processing, you must have a DSP System Toolbox license.

For more information about the recent changes to frame-based processing, see the “Frame-based processing changes” section of the *DSP System Toolbox Release Notes*.

Import Signals for Frame-Based Processing

The Signal From Workspace block creates a multichannel signal for frame-based processing when the **Signal** parameter is a matrix, and the **Samples per frame** parameter, *M*, is greater than 1. Beginning with the first *M* rows of the matrix, the block releases *M* rows of the matrix (that is, one frame from each channel) to the output port every $M*T_s$ seconds. Therefore, if the **Signal** parameter specifies a *W*-by-*N* workspace matrix, the Signal From Workspace block outputs a series of *M*-by-*N* matrices representing *N* channels. The workspace matrix must be oriented so that its columns represent the channels of the signal.

The figure below is a graphical illustration of this process for a 6-by-4 workspace matrix, A, and a frame size of 2.



Note Although independent channels are generally represented as columns, a single-channel signal can be represented in the workspace as either a column vector or row vector. The output from the Signal From Workspace block is a column vector in both cases.

In the following example, you use the Signal From Workspace block to create a three-channel frame signal and import it into the model:

1. Open the Signal From Workspace Example 5 model by typing

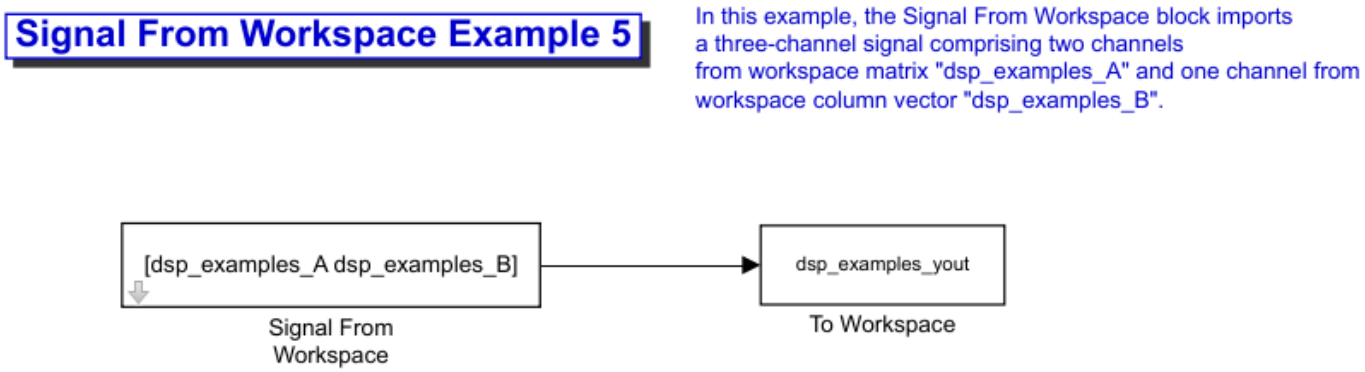
   ```
   ex_importfbsigs
   ```

   at the MATLAB command line.

   ```
   dsp_examples_A = [1:100;-1:-1:-100]';  % 100-by-2 matrix
   dsp_examples_B = 5*ones(100,1);         % 100-by-1 column vector
   ```

   The variable called dsp_examples_A represents a two-channel signal with 100 samples, and the variable called dsp_examples_B represents a one-channel signal with 100 samples.

   Also, the following variables are defined in the MATLAB workspace:



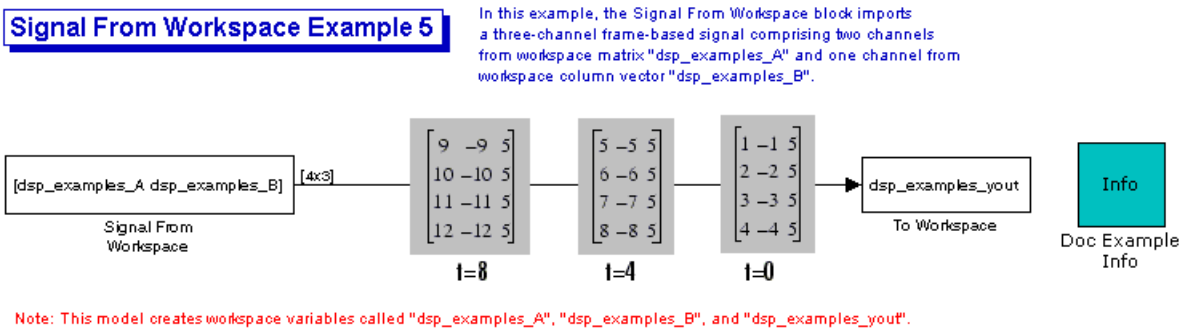
2 Double-click the Signal From Workspace block. Set the block parameters as follows, and then click **OK**:

- **Signal** parameter to `[dsp_examples_A dsp_examples_B]`
- **Sample time** parameter to `1`
- **Samples per frame** parameter to 4
- **Form output after final data value** parameter to `Setting to zero`

Based on these parameters, the Signal From Workspace block outputs a signal with a frame size of 4 and a sample period of 1 second. The signal's frame period is 4 seconds. The **Signal** parameter uses the standard MATLAB syntax for horizontally concatenating matrices to append column vector `dsp_examples_B` to the right of matrix `dsp_examples_A`. After the block has output the signal, all subsequent outputs have a value of zero.

3 Run the model.

The figure below is a graphical representation of how your three-channel frame signal is imported into your model.



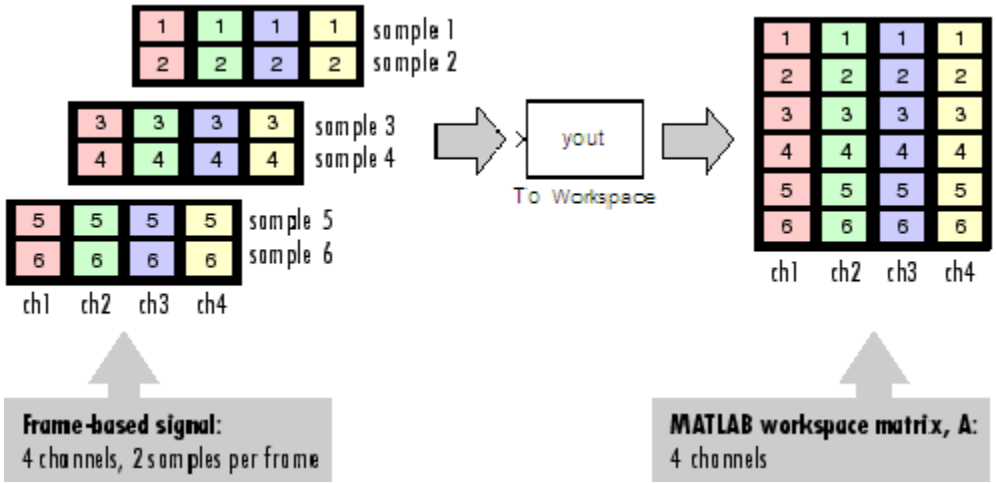
You have now successfully imported a three-channel frame signal into your model using the Signal From Workspace block.

Export Frame-Based Signals

The To Workspace and Triggered To Workspace blocks are the primary blocks for exporting signals of all dimensions from a Simulink model to the MATLAB workspace.

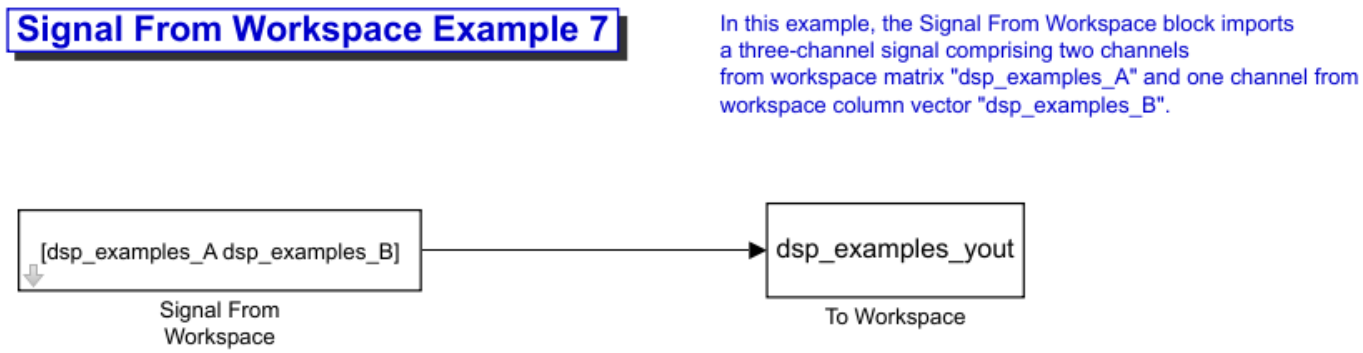
A signal with *N* channels and frame size M is represented by a sequence of *M*-by-*N* matrices. When this signal is input to the To Workspace block, the block creates a *P*-by-*N* array in the MATLAB workspace containing the *P* most recent samples from each channel. The number of rows, *P*, is specified by the **Limit data points to last** parameter. The newest samples are added at the bottom of the matrix.

The following figure is a graphical illustration of this process for three consecutive frames of a signal with a frame size of 2 that is exported to matrix A in the MATLAB workspace.



In the following example, you use a To Workspace block to export a three-channel signal with four samples per frame to the MATLAB workspace.

1. Open the Signal From Workspace Example 7 model by typing `ex_exportfbsigs` at the MATLAB command line.



Also, the following variables are defined in the MATLAB workspace:

The variable called dsp_examples_A represents a two-channel signal with 100 samples, and the variable called dsp_examples_B represents a one-channel signal with 100 samples.

```
dsp_examples_A = [1:100;-1:-1:-100]';    % 100-by-2 matrix
dsp_examples_B = 5*ones(100,1);            % 100-by-1 column vector
```

2 Double-click the Signal From Workspace block. Set the block parameters as follows, and then click **OK**:

- **Signal** = `[dsp_examples_A dsp_examples_B]`
- **Sample time** = `1`
- **Samples per frame** = `4`
- **Form output after final data value =** `Setting to zero`

Based on these parameters, the Signal From Workspace block outputs a signal with a frame size of 4 and a sample period of 1 second. The signal's frame period is 4 seconds. The **Signal** parameter uses the standard MATLAB syntax for horizontally concatenating matrices to append column vector `dsp_examples_B` to the right of matrix `dsp_examples_A`. After the block has output the signal, all subsequent outputs have a value of zero.

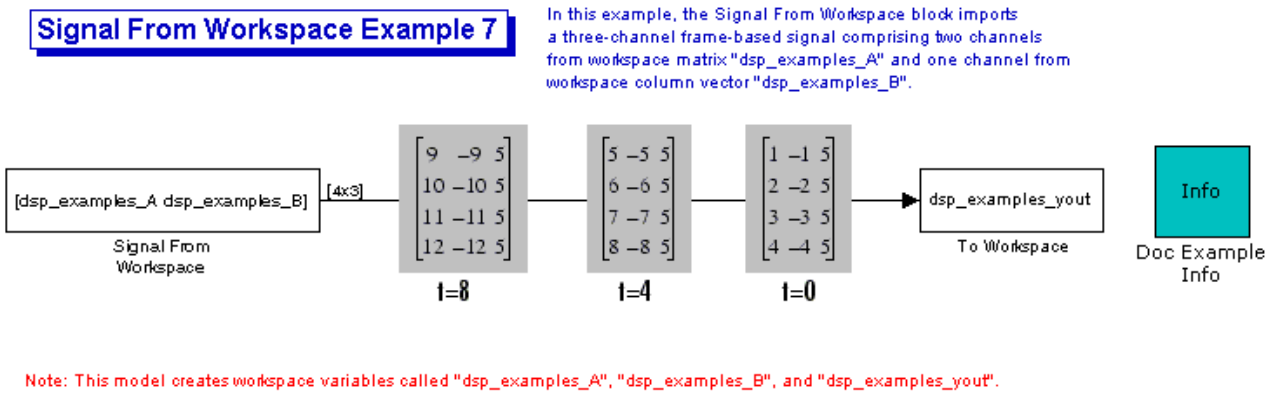
3 Double-click the To Workspace block. Set the block parameters as follows, and then click **OK**:

- **Variable name** = `dsp_examples_yout`
- **Limit data points to last** = `inf`
- **Decimation** = `1`
- **Frames** = `Concatenate frames (2-D array)`

Based on these parameters, the To Workspace block exports its input signal to a variable called `dsp_examples_yout` in the MATLAB workspace. The workspace variable can grow indefinitely large in order to capture all of the input data. The signal is not decimated before it is exported to the MATLAB workspace, and each input frame is vertically concatenated to the previous frame to produce a 2-D array output.

4 Run the model.

The following figure is a graphical representation of the model's behavior during simulation.



5 At the MATLAB command line, type `dsp_examples_yout`.

The output is shown below:

```
dsp_examples_yout =

     1    -1     5
     2    -2     5
     3    -3     5
     4    -4     5
     5    -5     5
     6    -6     5
     7    -7     5
     8    -8     5
     9    -9     5
    10   -10     5
    11   -11     5
    12   -12     5
```

The frames of the signal are concatenated to form a two-dimensional array.

You have now successfully output a frame signal to the MATLAB workspace using the To Workspace block.

See Also

More About

- “Sample- and Frame-Based Concepts” on page 3-2
- “Create Signals for Sample-Based Processing” on page 2-9
- “Create Signals for Frame-Based Processing” on page 2-13
- “Create Multichannel Signals for Sample-Based Processing” on page 2-18
- “Create Multichannel Signals for Frame-Based Processing” on page 2-23
- “Deconstruct Multichannel Signals for Sample-Based Processing” on page 2-27

- “Deconstruct Multichannel Signals for Frame-Based Processing” on page 2-32
- “Import and Export Signals for Sample-Based Processing” on page 2-38
- “Inspect Sample and Frame Rates in Simulink” on page 3-6
- “Convert Sample and Frame Rates in Simulink” on page 3-13

Data and Signal Management

Learn concepts such as sample- and frame-based processing, sample rate, delay and latency.

- “Sample- and Frame-Based Concepts” on page 3-2
- “Inspect Sample and Frame Rates in Simulink” on page 3-6
- “Convert Sample and Frame Rates in Simulink” on page 3-13
- “Buffering and Frame-Based Processing” on page 3-24
- “Delay and Latency” on page 3-35
- “Variable-Size Signal Support DSP System Objects” on page 3-46

Sample- and Frame-Based Concepts

In this section...
“Sample- and Frame-Based Signals” on page 3-2
“Model Sample- and Frame-Based Signals in MATLAB and Simulink” on page 3-2
“What Is Sample-Based Processing?” on page 3-3
“What Is Frame-Based Processing?” on page 3-3

Sample- and Frame-Based Signals

Sample-based signals are the most basic type of signal and are the easiest to construct from a real-world (physical) signal. You can create a sample-based signal by sampling a physical signal at a given sample rate, and outputting each individual sample as it is received. In general, most Digital-to-Analog converters output sample-based signals.

You can create frame-based signals from sample-based signals. When you buffer a batch of N samples, you create a frame of data. You can then output sequential frames of data at a rate that is $1/N$ times the sample rate of the original sample-based signal. The rate at which you output the frames of data is also known as the frame rate of the signal.

Frame-based data is a common format in real-time systems. Data acquisition hardware often operates by accumulating a large number of signal samples at a high rate. The hardware then propagates those samples to the real-time system as a block of data. Doing so maximizes the efficiency of the system by distributing the fixed process overhead across many samples. The faster data acquisition is suspended by slower interrupt processes after each frame is acquired, rather than after each individual sample. See “Benefits of Frame-Based Processing” on page 3-4 for more information.

DSP System Toolbox Source Blocks	Create Sample-Based Signals	Create Frame-Based Signals
Chirp	X	X
Constant	X	X
Colored Noise	X	X
Discrete Impulse	X	X
From Multimedia File	X	X
Identity Matrix	X	
Multiphase Clock	X	X
N-Sample Enable	X	X
Random Source	X	
Signal From Workspace	X	X
Sine Wave	X	X
UDP Receive	X	

Model Sample- and Frame-Based Signals in MATLAB and Simulink

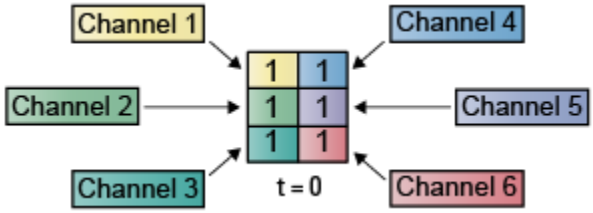
When you process signals using DSP System Toolbox software, you can do so in either a sample- or frame-based manner. When you are working with blocks in Simulink, you can specify, on a block-by-

block basis, which type of processing the block performs. In most cases, you specify the processing mode by setting the **Input processing** parameter. When you are using System objects in MATLAB, only frame-based processing is available. The following table shows the common parameter settings you can use to perform sample- and frame-based processing in MATLAB and Simulink.

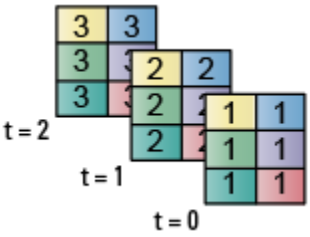
	Sample-Based Processing	Frame-Based Processing
Simulink — Blocks	**Input processing** = `Elements as channels (sample based)`	**Input processing** = `Columns as channels (frame based)`

What Is Sample-Based Processing?

In sample-based processing, blocks process signals one sample at a time. Each element of the input signal represents one sample in a distinct channel. For example, from a sample-based processing perspective, the following 3-by-2 matrix contains the first sample in each of six independent channels.



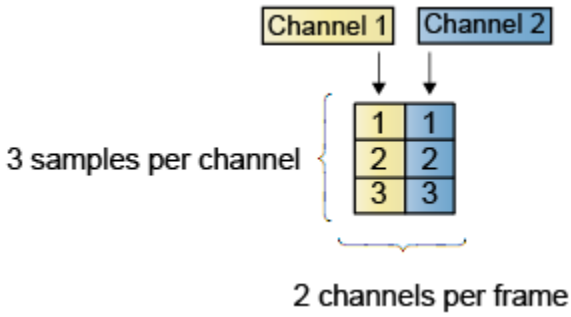
When you configure a block to perform sample-based processing, the block interprets scalar input as a single-channel signal. Similarly, the block interprets an *M*-by-*N* matrix as multichannel signal with *M***N* independent channels. For example, in sample-based processing, blocks interpret the following sequence of 3-by-2 matrices as a six-channel signal.



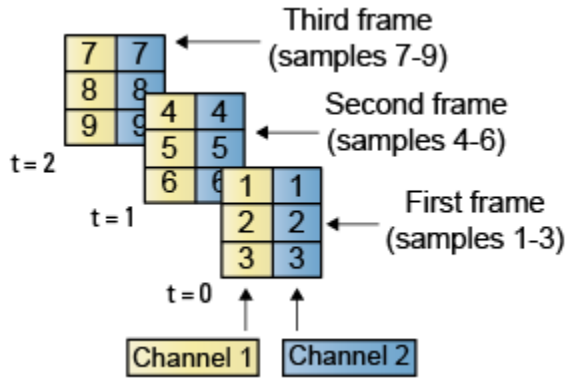
For more information about the recent changes to frame-based processing, see the “Frame-based processing changes” section of the *DSP System Toolbox Release Notes*.

What Is Frame-Based Processing?

In frame-based processing, blocks process data one frame at a time. Each frame of data contains sequential samples from an independent channel. Each channel is represented by a column of the input signal. For example, from a frame-based processing perspective, the following 3-by-2 matrix has two channels, each of which contains three samples.



When you configure a block to perform frame-based processing, the block interprets an *M*-by-1 vector as a single-channel signal containing *M* samples per frame. Similarly, the block interprets an *M*-by-*N* matrix as a multichannel signal with *N* independent channels and *M* samples per channel. For example, in frame-based processing, blocks interpret the following sequence of 3-by-2 matrices as a two-channel signal with a frame size of 3.



Using frame-based processing is advantageous for many signal processing applications because you can process multiple samples at once. By buffering your data into frames and processing multisample frames of data, you can often improve the computational time of your signal processing algorithms. To perform frame-based processing, you must have a DSP System Toolbox license.

For more information about the recent changes to frame-based processing, see the “Frame-based processing changes” section of the *DSP System Toolbox Release Notes*.

Benefits of Frame-Based Processing

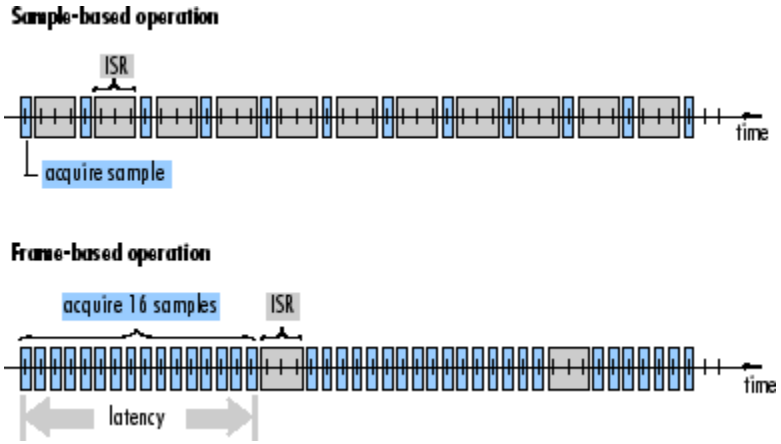
Frame-based processing is an established method of accelerating both real-time systems and model simulations.

Accelerate Real-Time Systems

Frame-based data is a common format in real-time systems. Data acquisition hardware often operates by accumulating a large number of signal samples at a high rate, and then propagating those samples to the real-time system as a block of data. This type of propagation maximizes the efficiency of the system by distributing the fixed process overhead across many samples; the faster data acquisition is suspended by slower interrupt processes after each frame is acquired, rather than after each individual sample is acquired.

The following figure illustrates how frame-based processing increases throughput. The thin blocks each represent the time elapsed during acquisition of a sample. The thicker blocks each represent the time elapsed during the interrupt service routine (ISR) that reads the data from the hardware.

In this example, the frame-based operation acquires a frame of 16 samples between each ISR. Thus, the frame-based throughput rate is many times higher than the sample-based alternative.



Be aware that frame-based processing introduces a certain amount of latency into a process due to the inherent lag in buffering the initial frame. In many instances, however, you can select frame sizes that improve throughput without creating unacceptable latencies. For more information, see “Delay and Latency” on page 3-35.

Accelerate Model Simulations

The simulation of your model also benefits from frame-based processing. In this case, you reduce the overhead of block-to-block communications by propagating frames of data rather than individual samples.

See Also

More About

- “Inspect Sample and Frame Rates in Simulink” on page 3-6
- “Convert Sample and Frame Rates in Simulink” on page 3-13

Inspect Sample and Frame Rates in Simulink

In this section...
“Sample Rate and Frame Rate Concepts” on page 3-6
“Inspect Signals Using the Probe Block” on page 3-7
“Inspect Signals Using Color Coding” on page 3-9

Sample Rate and Frame Rate Concepts

Sample rates and frame rates are important issues in most signal processing models. This is especially true with systems that incorporate rate conversions. Fortunately, in most cases when you build a Simulink model, you only need to set sample rates for the source blocks. Simulink automatically computes the appropriate sample rates for the blocks that are connected to the source blocks. Nevertheless, it is important to become familiar with the sample rate and frame rate concepts as they apply to Simulink models.

The *input frame period* (T_{fi}) of a frame signal is the time interval between consecutive vector or matrix inputs to a block. Similarly, the *output frame period* (T_{fo}) is the time interval at which the block updates the frame vector or matrix value at the output port.

In contrast, the sample period, T_s, is the time interval between individual samples in a frame, this value is shorter than the frame period when the frame size is greater than 1. The sample period of a frame signal is the quotient of the frame period and the frame size, M:

$$T_s = T_f/M$$

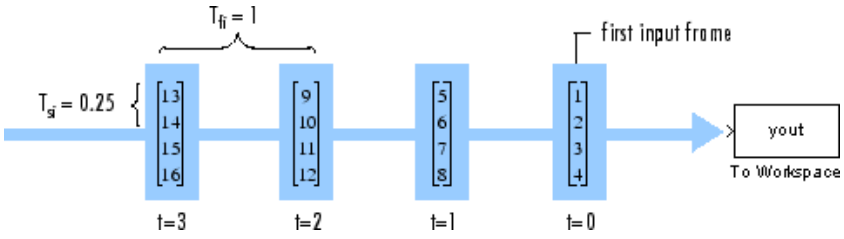
More specifically, the sample periods of inputs (T_{si}) and outputs (T_{so}) are related to their respective frame periods by

$$T_{si} = T_{fi}/M_i$$

$$T_{so} = T_{fo}/M_o$$

where M_i and M_o are the input and output frame sizes, respectively.

The illustration below shows a single-channel, frame signal with a frame size (M_i) of 4 and a frame period (T_{fi}) of 1. The sample period, T_{si}, is therefore 1/4, or 0.25 second.



The frame rate of a signal is the reciprocal of the frame period. For instance, the input frame rate would be $1/T_{fi}$. Similarly, the output frame rate would be $1/T_{fo}$.

The sample rate of a signal is the reciprocal of the sample period. For instance, the sample rate would be $1/T_s$.

In most cases, the sequence sample period T_{si} is most important, while the frame rate is simply a consequence of the frame size that you choose for the signal. For a sequence with a given sample period, a larger frame size corresponds to a slower frame rate, and vice versa.

The block decides whether to process the signal one sample at a time or one frame at a time depending on the settings in the block dialog box. For example, a Biquad filter block with **Input processing** parameter set to `Columns as channels (frame based)` treats a 3-by-2 input signal as a two-frame signal with three samples in each frame. If **Input processing** parameter is set to `Elements as channels (sample based)`, the 3-by-2 input signal is treated as a six-channel signal with one sample in each channel.

Inspect Signals Using the Probe Block

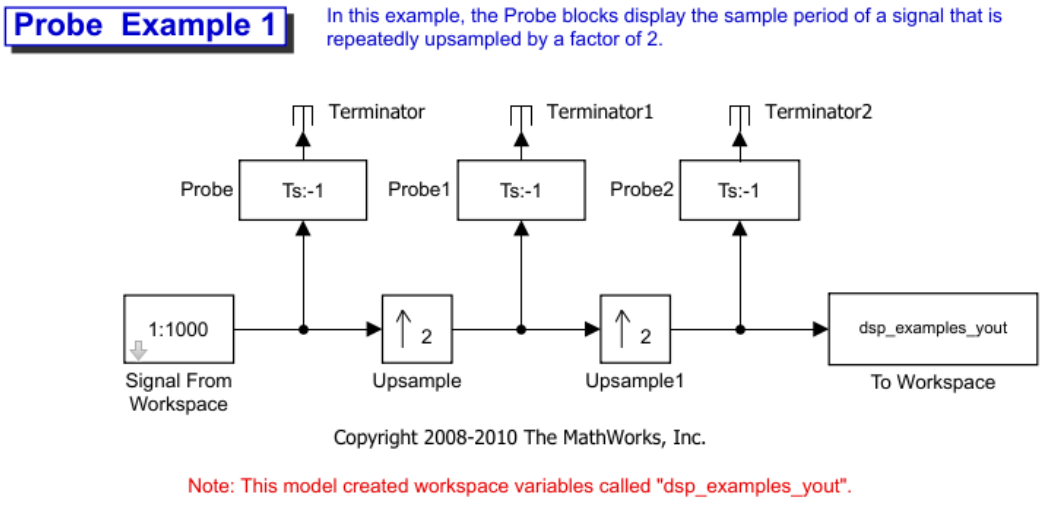
You can use the Probe block to display the sample period or the frame period of a signal. The Probe block displays the label `Ts`, the sample period or frame period of the sequence, followed by a two-element vector. The left element is the period of the signal being measured. The right element is the signal's sample time offset, which is usually 0.

Note Simulink offers the ability to shift the sample time of a signal by an arbitrary value, which is equivalent to shifting the signal's phase by a fractional sample period. However, sample-time offsets are rarely used in signal processing systems, and DSP System Toolbox blocks do not support them.

Display the Sample Period of a Signal Using the Probe Block

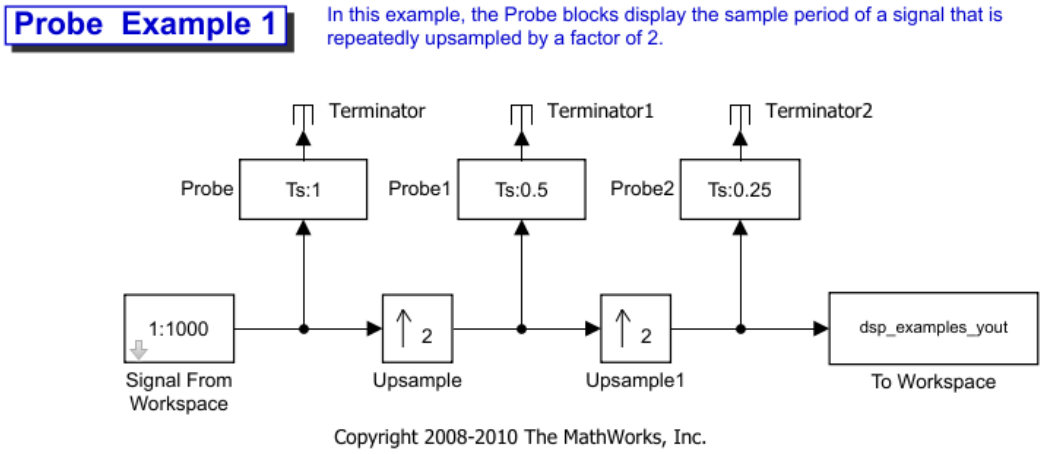
1. At the MATLAB command prompt, type `ex_probe_tut1`.

   The Probe Example 1 model opens. Double-click the Signal From Workspace block. Note that the **Samples per frame** parameter is set to 1.



2. Run the model.

   The figure below illustrates how the Probe blocks display the sample period of the signal before and after each upsample operation.

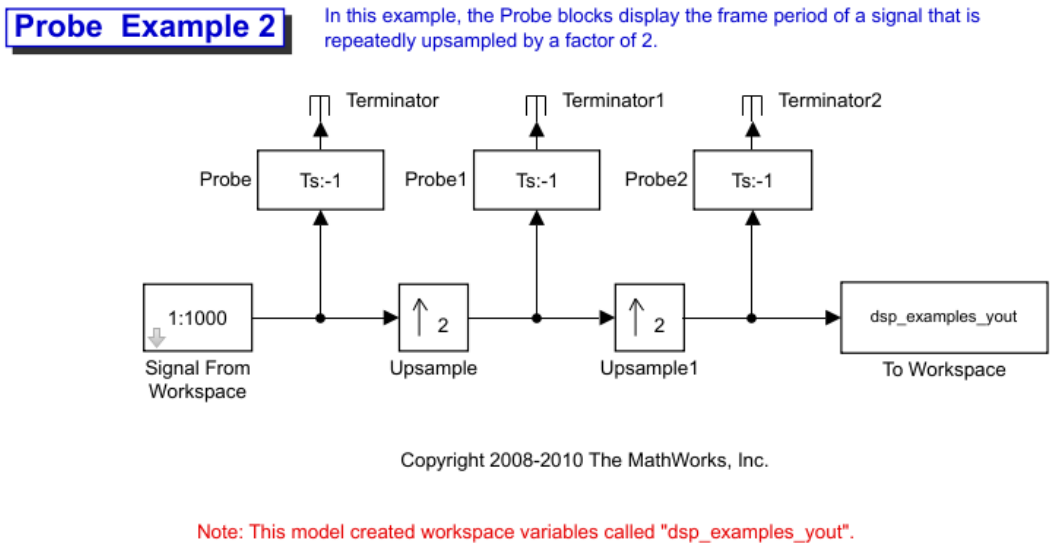


As displayed by the Probe blocks, the output from the Signal From Workspace block is a signal with a sample period of 1 second. The output from the first Upsample block has a sample period of 0.5 second, and the output from the second Upsample block has a sample period of 0.25 second.

Display the Frame Period of a Signal Using the Probe Block

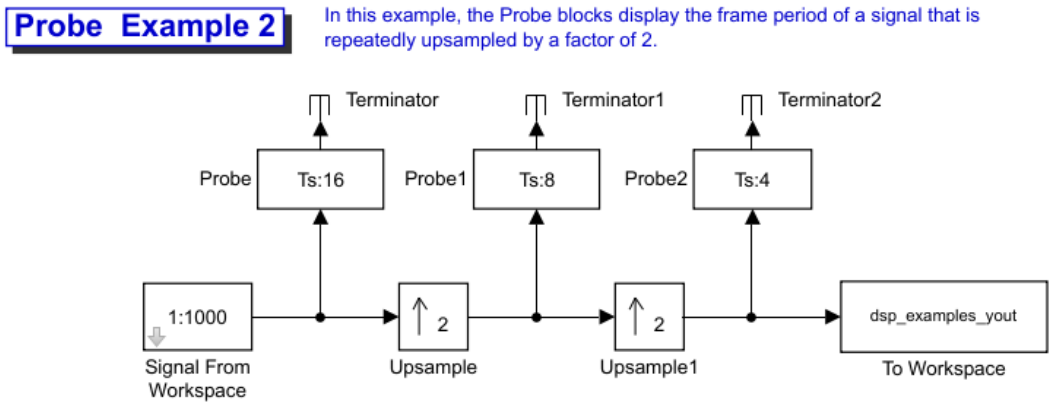
1 At the MATLAB command prompt, type `ex_probe_tut2`.

The Probe Example 2 model opens. Double-click the Signal From Workspace block. Note that the **Samples per frame** parameter is set to 16. Each frame in the signal contains 16 samples.



2 Run the model.

The figure below illustrates how the Probe blocks display the frame period of the signal before and after each upsample operation.



As displayed by the Probe blocks, the output from the Signal From Workspace block has a frame period of 16 seconds. The output from the first Upsample block has a frame period of 8 seconds, and the output from the second Upsample block has a frame period of 4 seconds.

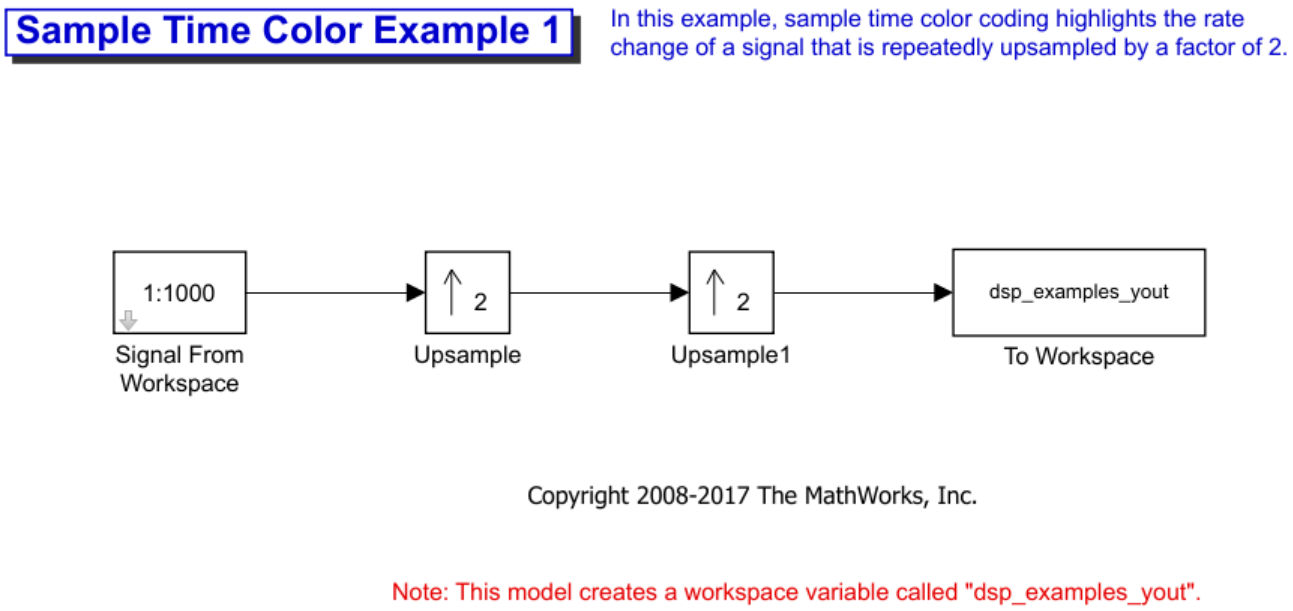
Note that the sample rate conversion is implemented through a change in the frame period rather than the frame size.

Inspect Signals Using Color Coding

View the Sample Rate of a Signal Using the Sample Time Color Coding

1 At the MATLAB command prompt, type `ex_color_tut1`.

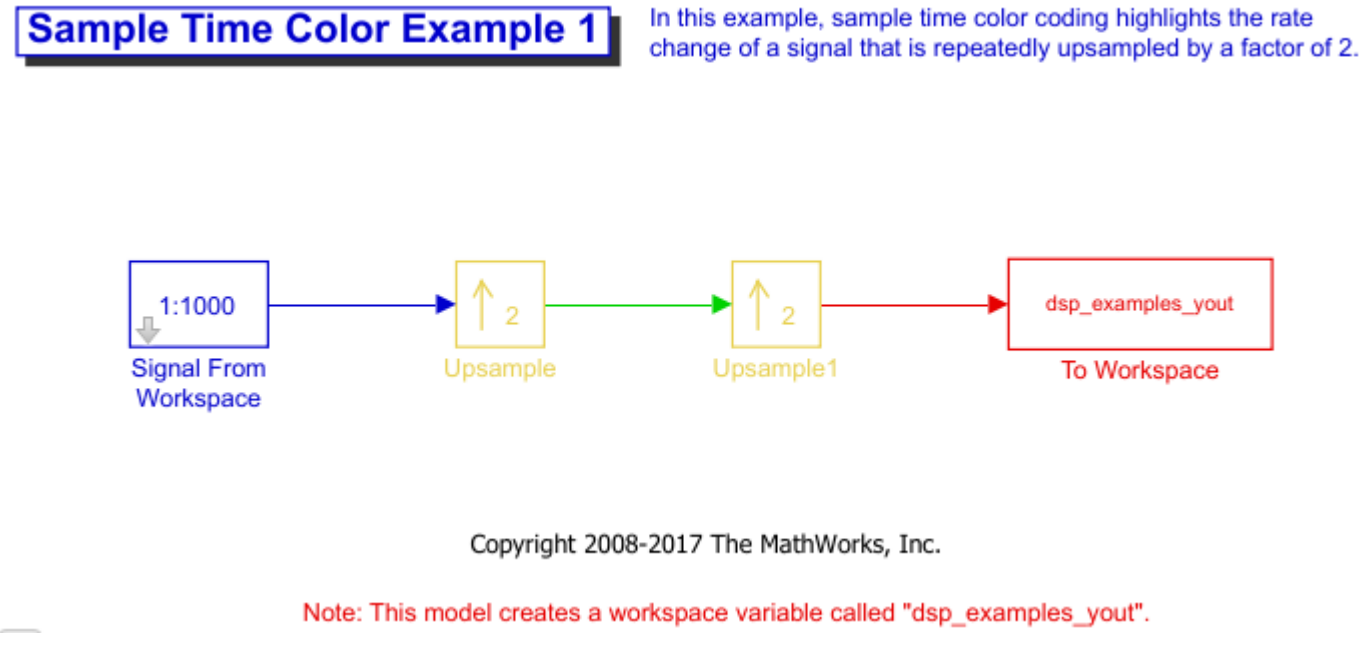
The Sample Time Color Example 1 model opens. Double-click the Signal From Workspace block. Note that the **Samples per frame** parameter is set to `1`.



2 In the **Debug** tab, select **Information Overlays > Colors**. This selection turns on sample time color coding. Simulink now assigns each sample rate a different color.

3 Run the model.

The model should now look similar to the following figure:

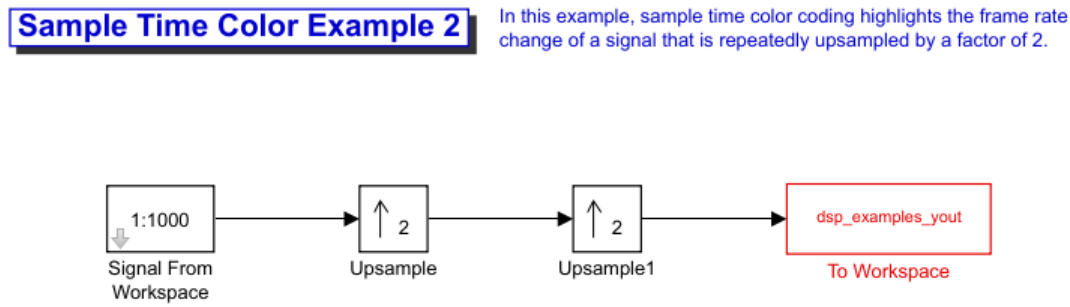


Every signal in this model has a different sample rate. Therefore, each signal is assigned a different color.

View the Frame Rate of a Signal Using the Sample Time Color Coding

1 At the MATLAB command prompt, type `ex_color_tut2`.

The Sample Time Color Example 2 model opens. Double-click the Signal From Workspace block. Note that the **Samples per frame** parameter is set to 16. Each frame in the signal contains 16 samples.

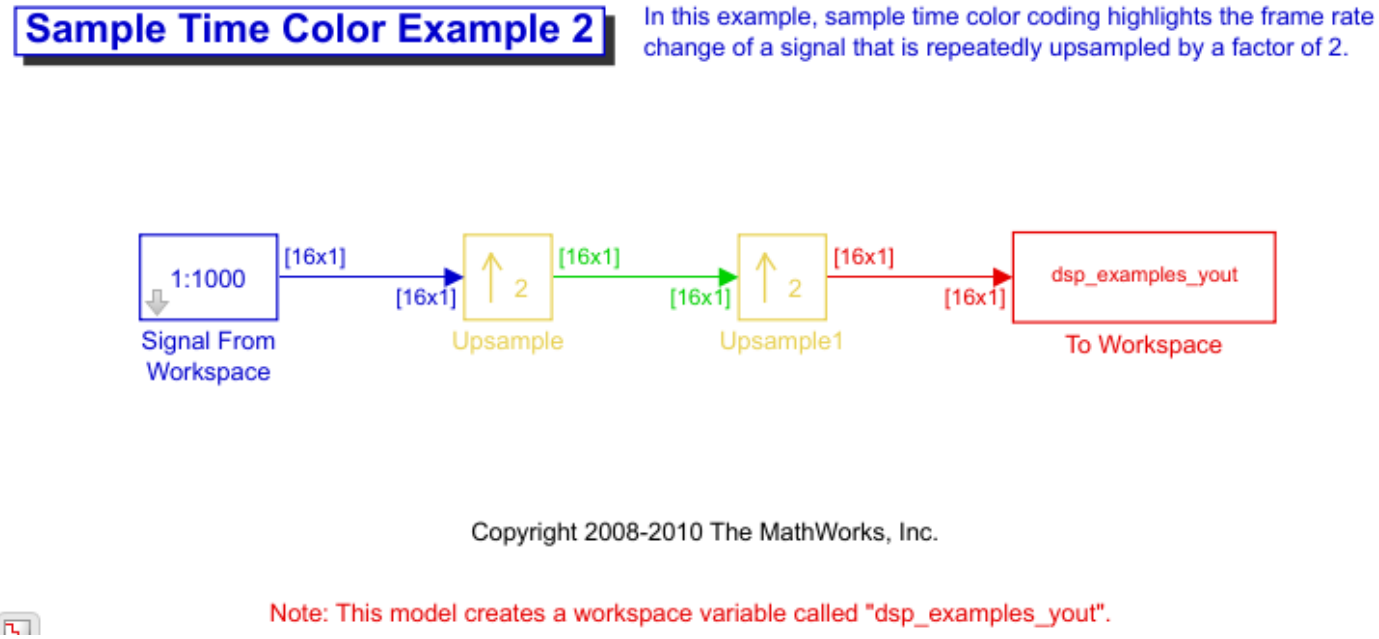


2 To turn on sample time color coding, in the **Debug** tab, select **Information Overlays** > **Colors**.

Simulink now assigns each frame rate a different color.

3 Run the model.

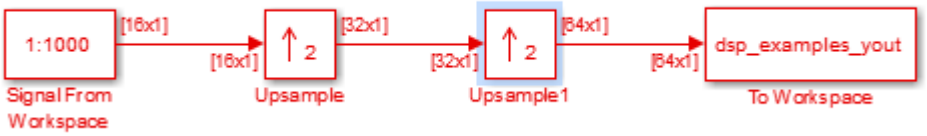
The model should now look similar to the following figure:



Because the **Rate options** parameter in the Upsample blocks is set to `Allow multirate processing`, each Upsample block changes the frame rate. Therefore, each frame signal in the model is assigned a different color.

4. Double-click on each Upsample block and change the **Rate options** parameter to `Enforce single-rate processing`.
5. Run the model.

   Every signal is coded with the same color. Therefore, every signal in the model now has the same frame rate.



For more information about sample time color coding, see “View Sample Time Information” (Simulink).

See Also

More About

- “Convert Sample and Frame Rates in Simulink” on page 3-13
- “Sample- and Frame-Based Concepts” on page 3-2

Convert Sample and Frame Rates in Simulink

In this section...
“Rate Conversion Blocks” on page 3-13
“Rate Conversion by Frame-Rate Adjustment” on page 3-14
“Rate Conversion by Frame-Size Adjustment” on page 3-15
“Frame Rebuffering Blocks” on page 3-17
“Buffer Signals by Preserving the Sample Period” on page 3-19
“Buffer Signals by Altering the Sample Period” on page 3-21

Rate Conversion Blocks

There are two common types of operations that impact the frame and sample rates of a signal: direct rate conversion and frame rebuffering. Direct rate conversions, such as upsampling and downsampling, can be implemented by altering either the frame rate or the frame size of a signal. Frame rebuffering which is used to alter the frame size of a signal in order to improve simulation throughput, usually changes either the sample rate or frame rate of the signal as well.

The following table lists the principal rate conversion blocks in DSP System Toolbox software. Blocks marked with an asterisk (*) offer the option of changing the rate by either adjusting the frame size or frame rate.

Block	Library
Downsample *	Signal Operations
Dyadic Analysis Filter Bank	Filtering / Multirate Filters
Dyadic Synthesis Filter Bank	Filtering / Multirate Filters
FIR Decimation *	Filtering / Multirate Filters
FIR Interpolation *	Filtering / Multirate Filters
FIR Rate Conversion	Filtering / Multirate Filters
Repeat *	Signal Operations
Upsample *	Signal Operations

Direct Rate Conversion

Rate conversion blocks accept an input signal at one sample rate, and propagate the same signal at a new sample rate. Several of these blocks contain a **Rate options** parameter offering two options for multirate versus single-rate processing:

- `Enforce single-rate processing`: When you select this option, the block maintains the input sample rate.
- `Allow multirate processing`: When you select this option, the block downsamples the signal such that the output sample rate is K times slower than the input sample rate.

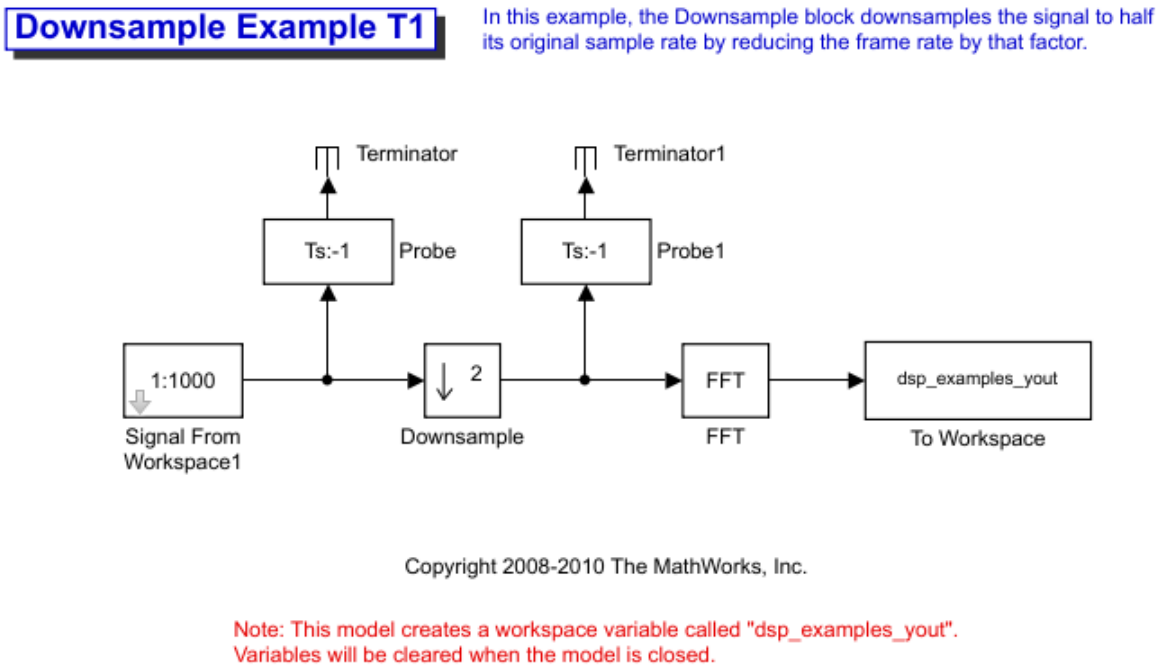
Note When a Simulink model contains signals with various frame rates, the model is called *multirate*. You can find a discussion of multirate models in “Excess Algorithmic Delay (Tasking Latency)” on page 3-40. Also see “Time-Based Scheduling and Code Generation” (Simulink Coder).

Rate Conversion by Frame-Rate Adjustment

One way to change the sample rate of a signal, $1/T_{so}$, is to change the output frame rate ($T_{fo} \neq T_{fi}$), while keeping the frame size constant ($M_o = M_i$). Note that the sample rate of a signal is defined as $1/T_{so} = M_o/T_{fo}$:

1. At the MATLAB command prompt, type `ex_downsample_tut1`.

   The Downsample Example T1 model opens.



2. In the **Debug** tab, select **Information Overlays** > **Signal Dimensions**.

   When you run the model, the dimensions of the signals appear next to the lines connecting the blocks.

3. Double-click the Signal From Workspace block. The **Source Block Parameters: Signal From Workspace** dialog box opens.

4. Set the block parameters as follows:

   - **Sample time** = `0.125`
   - **Samples per frame** = 8

   Based on these parameters, the Signal From Workspace block outputs a signal with a sample period of 0.125 second and a frame size of 8.

5. Save these parameters and close the dialog box by clicking **OK**.

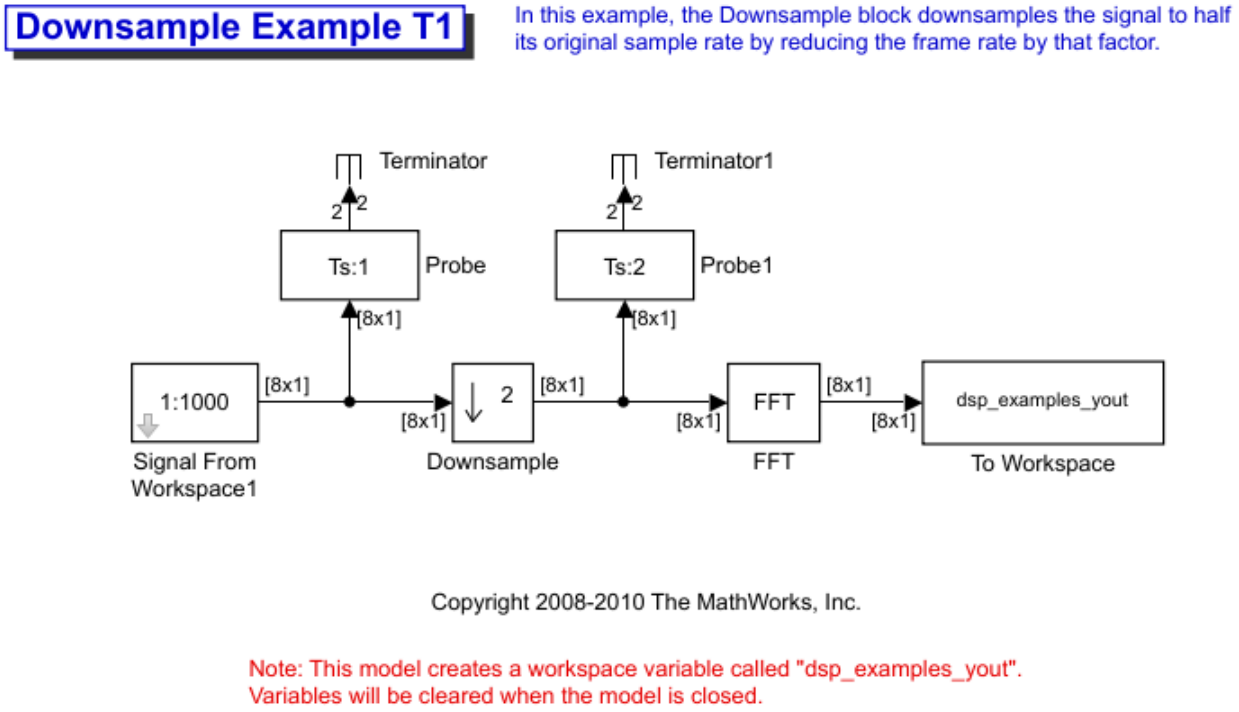
6. Double-click the Downsample block. The **Function Block Parameters: Downsample** dialog box opens.

7. Set the **Rate options** parameter to `Allow multirate processing`, and then click **OK**.

The Downsample block is configured to downsample the signal by changing the frame rate rather than the frame size.

8 Run the model.

After the simulation, the model should look similar to the following figure.



Because $T_{fi} = M_i \times T_{si}$, the input frame period, T_{fi}, is $T_{fi} = 8 \times 0.125 = 1$ second. This value is displayed by the first Probe block. Therefore the input frame rate, $1/T_{fi}$, is also 1 frame per second.

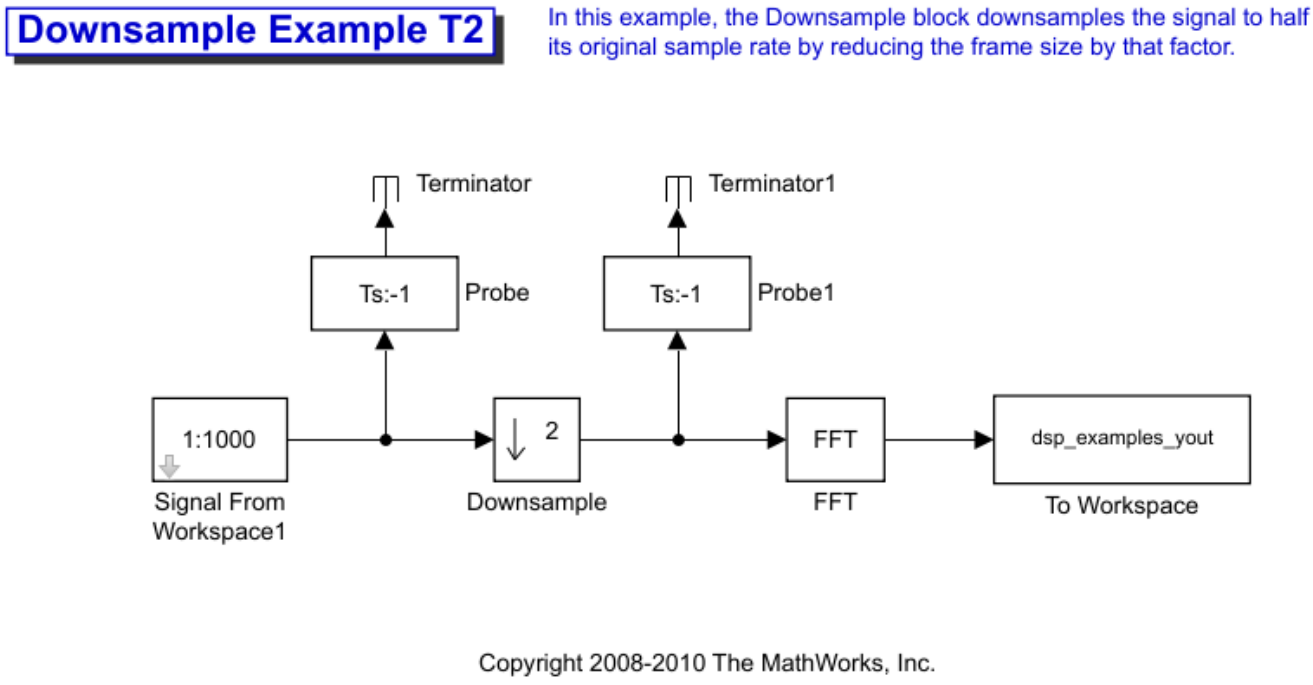
The second Probe block in the model verifies that the output from the Downsample block has a frame period, T_{fo}, of 2 seconds, twice the frame period of the input. However, because the frame rate of the output, $1/T_{fo}$, is 0.5 frames per second, the Downsample block actually downsampled the original signal to half its original rate. As a result, the output sample period, $T_{so} = T_{fo}/M_o$, is doubled to 0.25 second without any change to the frame size. The signal dimensions in the model confirm that the frame size did not change.

Rate Conversion by Frame-Size Adjustment

One way to change the sample rate of a signal is by changing the frame size (that is $M_o \neq M_i$), but keep the frame rate constant ($T_{fo} = T_{fi}$). Note that the sample rate of a signal is defined as $1/T_{so} = M_o/T_{fo}$:

1 At the MATLAB command prompt, type `ex_downsample_tut2`.

The Downsample Example T2 model opens.



2 In the **Debug** tab, select **Information Overlays** > **Signal Dimensions**.

When you run the model, the dimensions of the signals appear next to the lines connecting the blocks.

3 Double-click the Signal From Workspace block. The **Source Block Parameters: Signal From Workspace** dialog box opens.

4 Set the block parameters as follows:

- **Sample time** = `0.125`
- **Samples per frame** = 8

Based on these parameters, the Signal From Workspace block outputs a signal with a sample period of 0.125 second and a frame size of 8.

5 Save these parameters and close the dialog box by clicking **OK**.

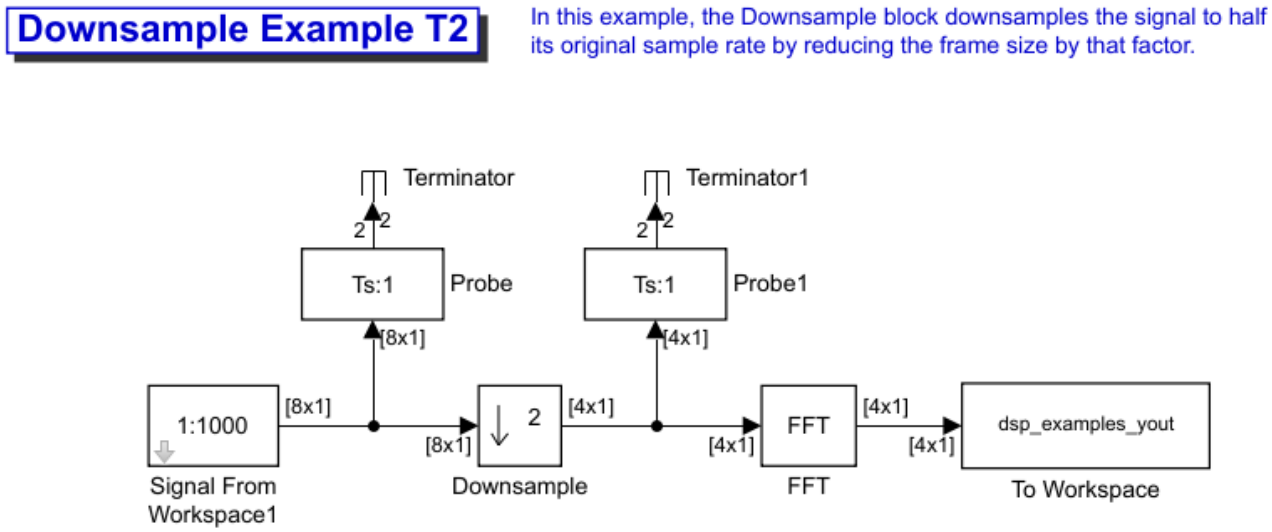
6 Double-click the Downsample block. The **Function Block Parameters: Downsample** dialog box opens.

7 Set the **Rate options** parameter to `Enforce single-rate processing`, and then click **OK**.

The Downsample block is configured to downsample the signal by changing the frame size rather than the frame rate.

8 Run the model.

After the simulation, the model should look similar to the following figure.



Because $T_{fi} = M_i \times T_{si}$, the input frame period, T_{fi}, is $T_{fi} = 8 \times 0.125 = 1$ second. This value is displayed by the first Probe block. Therefore the input frame rate, $1/T_{fi}$, is also 1 frame per second.

The Downsample block downsampled the input signal to half its original frame size. The signal dimensions of the output of the Downsample block confirm that the downsampled output has a frame size of 4, half the frame size of the input. As a result, the sample period of the output, $T_{so} = T_{fo}/M_o$is 0.25 second. This process occurred without any change to the frame rate ($T_{fi} = T_{fo}$).

Frame Rebuffering Blocks

There are two common types of operations that impact the frame and sample rates of a signal: direct rate conversion and frame rebuffering. Direct rate conversions, such as upsampling and downsampling, can be implemented by altering either the frame rate or the frame size of a signal. Frame rebuffering, which is used alter the frame size of a signal in order to improve simulation throughput, usually changes either the sample rate or frame rate of the signal as well.

Sometimes you might need to rebuffer a signal to a new frame size at some point in a model. For example, your data acquisition hardware may internally buffer the sampled signal to a frame size that is not optimal for the signal processing algorithm in the model. In this case, you would want to rebuffer the signal to a frame size more appropriate for the intended operations without introducing any change to the data or sample rate.

The following table lists the principal DSP System Toolbox buffering blocks.

Block	Library
Buffer	Signal Management/ Buffers
Delay Line	Signal Management/ Buffers
Unbuffer	Signal Management/ Buffers
Variable Selector	Signal Management/ Indexing

Blocks for Frame Rebuffering with Preservation of the Signal

Buffering operations provide another mechanism for rate changes in signal processing models. The purpose of many buffering operations is to adjust the frame size of the signal, M, without altering the signal's sample rate T_s. This usually results in a change to the signal's frame rate, T_f, according to the following equation:

$$T_f = MT_s$$

However, the equation above is only true if no samples are added or deleted from the original signal. Therefore, the equation above does not apply to buffering operations that generate overlapping frames, that only partially unbuffer frames, or that alter the data sequence by adding or deleting samples.

There are two blocks in the Buffers library that can be used to change a signal's frame size without altering the signal itself:

- Buffer — redistributes signal samples to a larger or smaller frame size
- Unbuffer — unbuffers a signal with frame size M and frame period T_f to a signal with frame size `1` and frame period T_s

The Buffer block preserves the signal's data and sample period only when its **Buffer overlap** parameter is set to `0`. The output frame period, T_{fo}, is

$$T_{fo} = \frac{M_o T_{fi}}{M_i}$$

where T_{fi} is the input frame period, M_i is the input frame size, and M_o is the output frame size specified by the **Output buffer size (per channel)** parameter.

The Unbuffer block unbuffers a frame signal and always preserves the signal's data and sample period

$$T_{so} = T_{fi}/M_i$$

where T_{fi} and M_i are the period and size, respectively, of the frame signal.

Both the Buffer and Unbuffer blocks preserve the sample period of the sequence in the conversion ($T_{so} = T_{si}$).

Blocks for Frame Rebuffering with Alteration of the Signal

Some forms of buffering alter the signal's data or sample period in addition to adjusting the frame size. This type of buffering is desirable when you want to create sliding windows by overlapping consecutive frames of a signal, or select a subset of samples from each input frame for processing.

The blocks that alter a signal while adjusting its frame size are listed below. In this list, T_{si} is the input sequence sample period, and T_{fi} and T_{fo} are the input and output frame periods, respectively:

- The Buffer block adds duplicate samples to a sequence when the **Buffer overlap** parameter, L, is set to a nonzero value. The output frame period is related to the input sample period by

  $$T_{fo} = (M_o - L)T_{si}$$

  where M_o is the output frame size specified by the **Output buffer size (per channel)** parameter. As a result, the new output sample period is

  $$T_{so} = \frac{(M_o - L)T_{si}}{M_o}$$

- The Delay Line block adds duplicate samples to the sequence when the **Delay line size** parameter, M_o, is greater than 1. The output and input frame periods are the same, $T_{fo} = T_{fi} = T_{si}$, and the new output sample period is

  $$T_{so} = \frac{T_{si}}{M_o}$$

- The Variable Selector block can remove, add, and/or rearrange samples in the input frame when **Select** is set to `Rows`. The output and input frame periods are the same, $T_{fo} = T_{fi}$, and the new output sample period is

  $$T_{so} = \frac{M_i T_{si}}{M_o}$$

  where M_o is the length of the block's output, determined by the **Elements** vector.

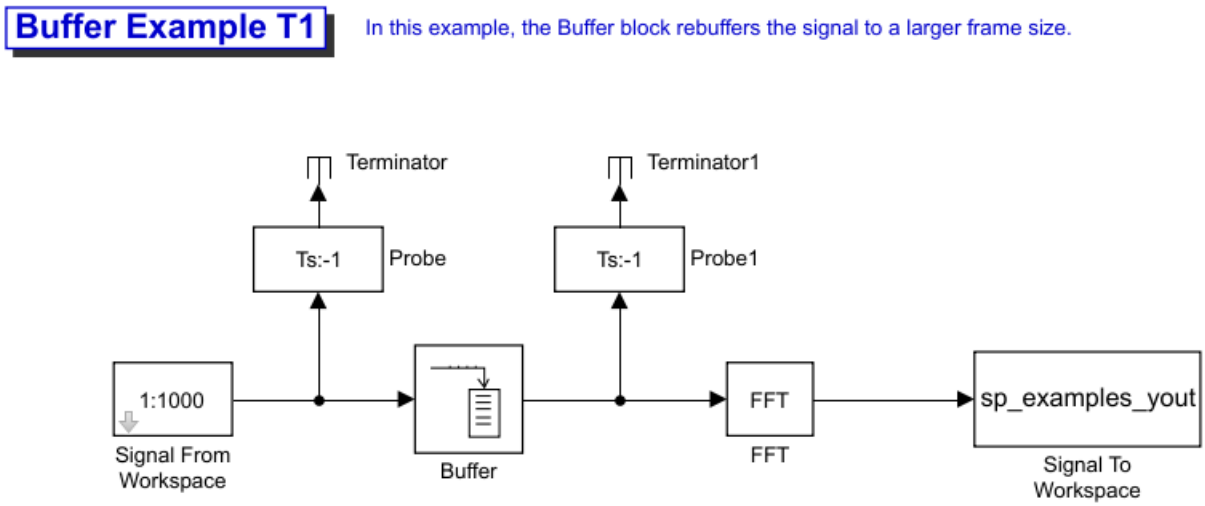
In all of these cases, the sample period of the output sequence is *not* equal to the sample period of the input sequence.

Buffer Signals by Preserving the Sample Period

In the following example, a signal with a sample period of 0.125 second is rebuffered from a frame size of 8 to a frame size of 16. This rebuffering process doubles the frame period from 1 to 2 seconds, but does not change the sample period of the signal ($T_{so} = T_{si} = 0.125$). The process also does not add or delete samples from the original signal:

1 At the MATLAB command prompt, type `ex_buffer_tut1`.

The Buffer Example T1 model opens.



2 Double-click the Signal From Workspace block. The **Source Block Parameters: Signal From Workspace** dialog box opens.

3 Set the parameters as follows:

- **Signal** = `1:1000`
- **Sample time** = `0.125`
- **Samples per frame** = `8`
- **Form output after final data value** = `Setting to zero`

Based on these parameters, the Signal from Workspace block outputs a signal with a sample period of 0.125 second. Each output frame contains eight samples.

4 Save these parameters and close the dialog box by clicking **OK**.

5 Double-click the Buffer block. The **Function Block Parameters: Buffer** dialog box opens.

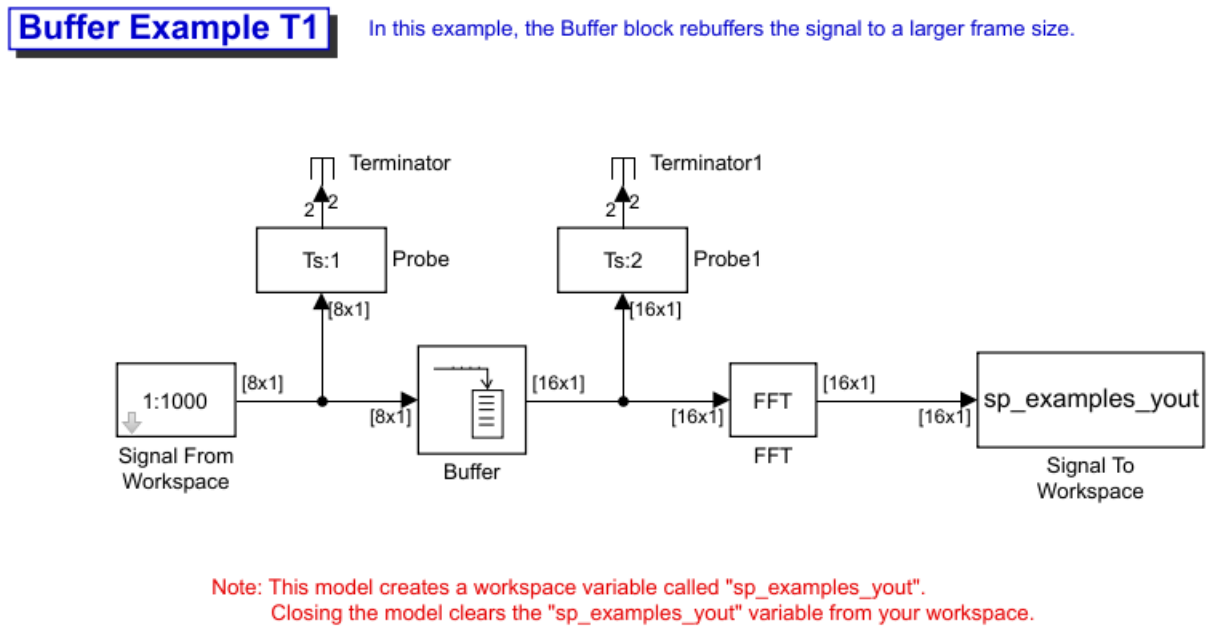
6 Set the parameters as follows, and then click **OK**:

- **Output buffer size (per channel)** = `16`
- **Buffer overlap** = `0`
- **Initial conditions** = `0`

Based on these parameters, the Buffer block rebuffers the signal from a frame size of 8 to a frame size of 16.

7 Run the model.

The following figure shows the model after simulation.



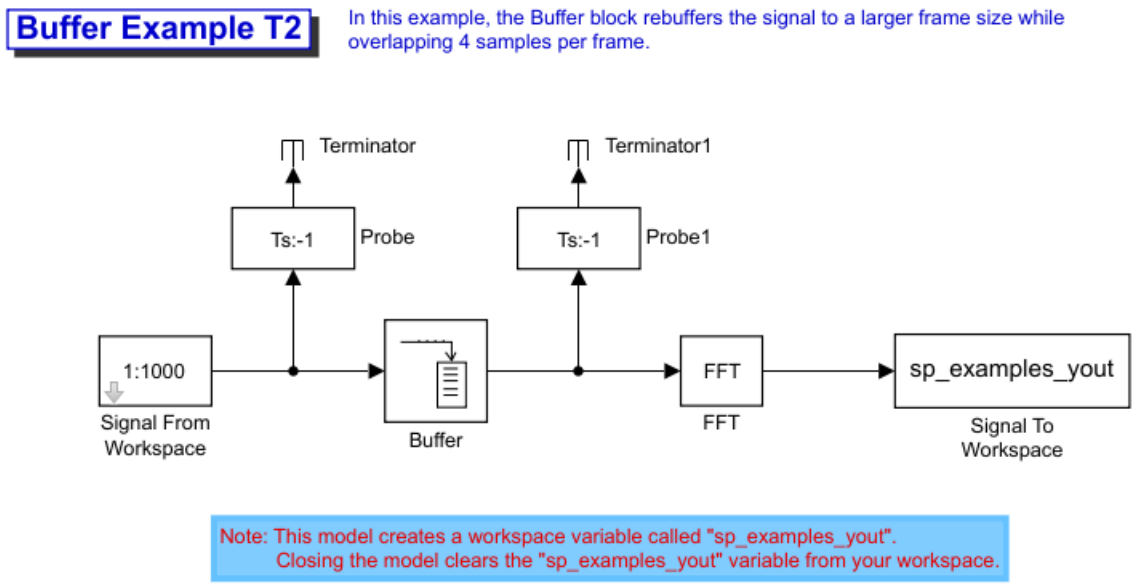
Note that the input to the Buffer block has a frame size of 8 and the output of the block has a frame size of 16. As shown by the Probe blocks, the rebuffering process doubles the frame period from 1 to 2 seconds.

Buffer Signals by Altering the Sample Period

Some forms of buffering alter the signal's data or sample period in addition to adjusting the frame size. In the following example, a signal with a sample period of 0.125 second is rebuffered from a frame size of 8 to a frame size of 16 with a buffer overlap of 4:

1 At the MATLAB command prompt, type `ex_buffer_tut2`.

The Buffer Example T2 model opens.



2 Double-click the Signal From Workspace block. The **Source Block Parameters: Signal From Workspace** dialog box opens.

3 Set the parameters as follows:

- **Signal** = `1:1000`
- **Sample time** = `0.125`
- **Samples per frame** = `8`
- **Form output after final data value** = `Setting to zero`

Based on these parameters, the Signal from Workspace block outputs a signal with a sample period of 0.125 second. Each output frame contains eight samples.

4 Save these parameters and close the dialog box by clicking **OK**.

5 Double-click the Buffer block. The **Function Block Parameters: Buffer** dialog box opens.

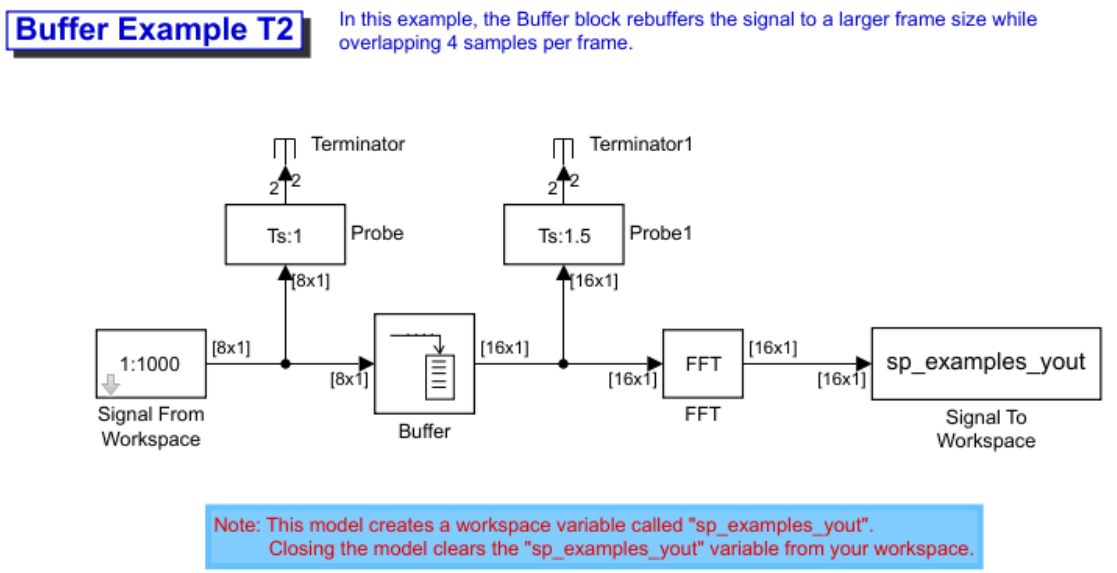
6 Set the parameters as follows, and then click **OK**:

- **Output buffer size (per channel)** = `16`
- **Buffer overlap** = `4`
- **Initial conditions** = `0`

Based on these parameters, the Buffer block rebuffers the signal from a frame size of 8 to a frame size of 16. Also, after the initial output, the first four samples of each output frame are made up of the last four samples from the previous output frame.

7 Run the model.

The following figure shows the model after the simulation has stopped.



Note that the input to the Buffer block has a frame size of 8 and the output of the block has a frame size of 16. The relation for the output frame period for the Buffer block is

$$T_{fo} = (M_o - L)T_{si}$$

T_{fo} is (16-4)*0.125, or 1.5 seconds, as confirmed by the second Probe block. The sample period of the signal at the output of the Buffer block is no longer 0.125 second. It is now $T_{so} = T_{fo}/M_o$ = 1.5/16 = 0.0938 second. Thus, both the signal's data and the signal's sample period have been altered by the buffering operation.

See Also

More About

- “Sample- and Frame-Based Concepts” on page 3-2
- “Inspect Sample and Frame Rates in Simulink” on page 3-6

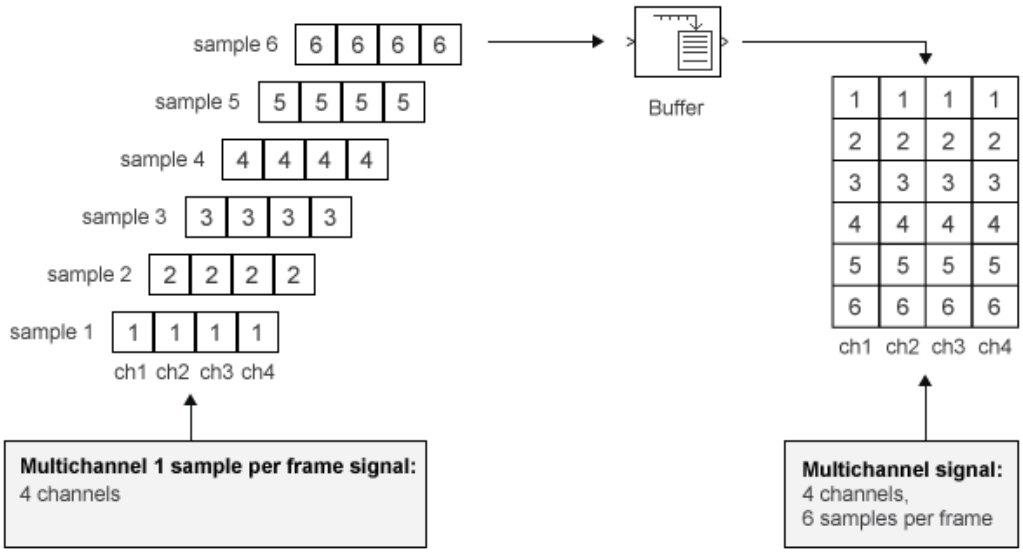
Buffering and Frame-Based Processing

In this section...
“Buffer Input into Frames” on page 3-24
“Buffer Signals into Frames with Overlap” on page 3-26
“Buffer Frame Inputs into Other Frame Inputs” on page 3-28
“Buffer Delay and Initial Conditions” on page 3-30
“Unbuffer Frame Signals into Sample Signals” on page 3-31

Buffer Input into Frames

Multichannel signals of frame size 1 can be buffered into multichannel signals of frame size *L* using the Buffer block. *L* is greater than 1.

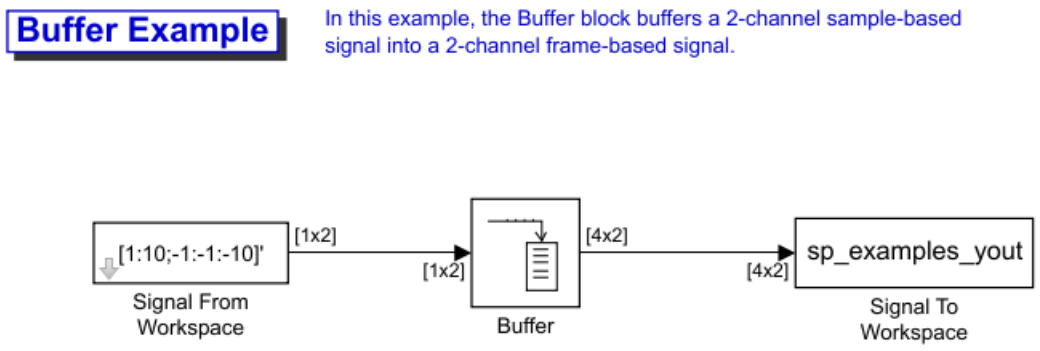
The following figure is a graphical representation of a signal with frame size 1 being converted into a signal of frame size *L* by the Buffer block.



In the following example, a two-channel 1 sample per frame signal is buffered into a two-channel 4 samples per frame signal using a Buffer block:

1 At the MATLAB command prompt, type `ex_buffer_tut`.

The Buffer Example model opens.



2. Double-click the Signal From Workspace block. The **Source Block Parameters: Signal From Workspace** dialog box opens.
3. Set the parameters as follows:

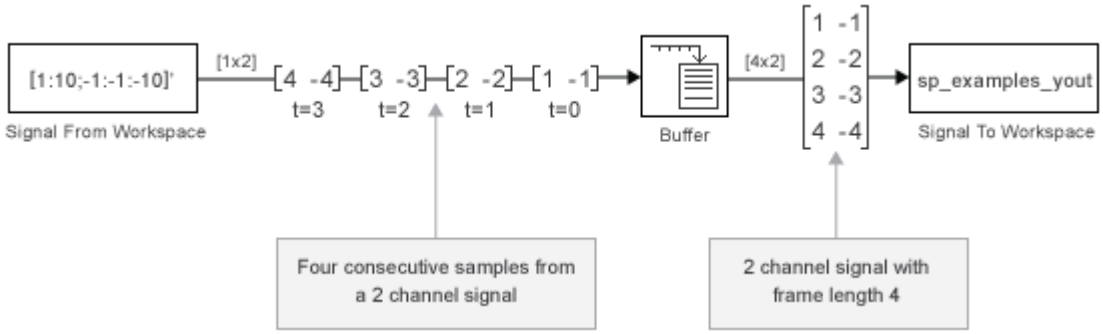
   - **Signal** = `[1:10;-1:-1:-10]'`
   - **Sample time** = `1`
   - **Samples per frame** = `1`
   - **Form output after final data value** = `Setting to zero`

   Based on these parameters, the Signal from Workspace block outputs a signal with a frame length of 1 and a sample period of 1 second. Because you set the **Samples per frame** parameter setting to 1, the Signal From Workspace block outputs one two-channel sample at each sample time.
4. Save these parameters and close the dialog box by clicking **OK**.
5. Double-click the Buffer block. The **Function Block Parameters: Buffer** dialog box opens.
6. Set the parameters as follows:

   - **Output buffer size (per channel)** = `4`
   - **Buffer overlap** = `0`
   - **Initial conditions** = `0`

   Because you set the **Output buffer size** parameter to 4, the Buffer block outputs a frame signal with frame size 4.
7. Run the model.

   The figure below is a graphical interpretation of the model behavior during simulation.



Note Alternatively, you can set the **Samples per frame** parameter of the Signal From Workspace block to 4 and create the same signal shown above without using a Buffer block. The Signal From Workspace block performs the buffering internally, in order to output a two-channel frame.

Buffer Signals into Frames with Overlap

In some cases it is useful to work with data that represents overlapping sections of an original signal. For example, in estimating the power spectrum of a signal, it is often desirable to compute the FFT of overlapping sections of data. Overlapping buffers are also needed in computing statistics on a sliding window, or for adaptive filtering.

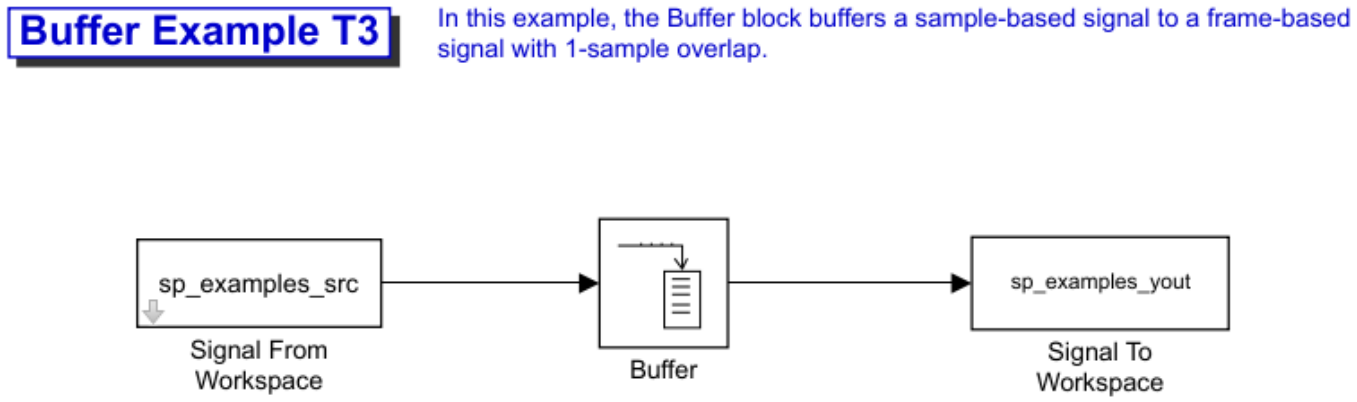
The **Buffer overlap** parameter of the Buffer block specifies the number of overlap points, L. In the overlap case ($L > 0$), the frame period for the output is $(M_o\text{-}L)*T_{si}$, where T_{si} is the input sample period and M_o is the **Buffer size**.

Note Set the **Buffer overlap** parameter to a negative value to achieve output frame rates *slower* than in the nonoverlapping case. The output frame period is still $T_{si}*(M_o\text{-}L)$, but now with $L < 0$. Only the M_o newest inputs are included in the output buffers. The previous L inputs are discarded.

In the following example, a four-channel signal with frame length 1 and sample period 1 is buffered to a signal with frame size 3 and frame period 2. Because of the buffer overlap, the input sample period is not conserved, and the output sample period is 2/3:

1. At the MATLAB command prompt, type `ex_buffer_tut3`.

   The Buffer Example T3 model opens.



Also, the variable `sp_examples_src` is loaded into the MATLAB workspace. This variable is defined as follows:

```
sp_examples_src=[1 1 5 -1; 2 1 5 -2; 3 0 5 -3; 4 0 5 -4; 5 1 5 -5; 6 1 5 -6];
```

2. Double-click the Signal From Workspace block. The **Source Block Parameters: Signal From Workspace** dialog box opens.
3. Set the block parameters as follows:

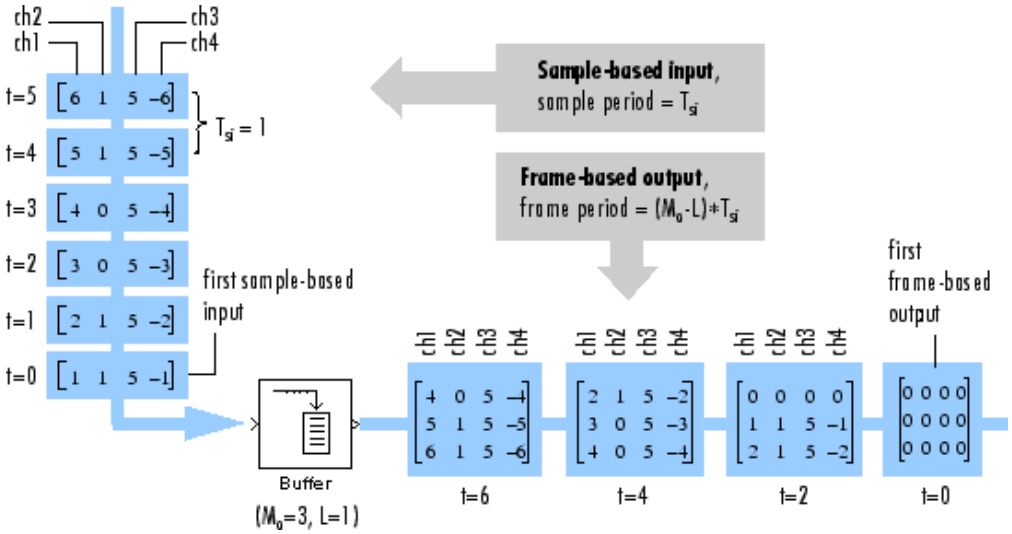
   - **Signal** = `sp_examples_src`
   - **Sample time** = `1`
   - **Samples per frame** = `1`
   - **Form output after final data value by** = `Setting to zero`

   Based on these parameters, the Signal from Workspace block outputs a signal with a sample period of 1 second. Because you set the **Samples per frame** parameter setting to 1, the Signal From Workspace block outputs one four-channel sample at each sample time.
4. Save these parameters and close the dialog box by clicking **OK**.
5. Double-click the Buffer block. The **Function Block Parameters: Buffer** dialog box opens.
6. Set the block parameters as follows, and then click **OK**:

   - **Output buffer size (per channel)** = `3`
   - **Buffer overlap** = `1`
   - **Initial conditions** = `0`

   Because you set the **Output buffer size** parameter to 3, the Buffer block outputs a signal with frame size 3. Also, because you set the **Buffer overlap** parameter to 1, the last sample from the previous output frame is the first sample in the next output frame.
7. Run the model.

   The following figure is a graphical interpretation of the model's behavior during simulation.



8 At the MATLAB command prompt, type `sp_examples_yout`.

The following is displayed in the MATLAB Command Window.

```
sp_examples_yout =

     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
     1     1     5    -1
     2     1     5    -2
     2     1     5    -2
     3     0     5    -3
     4     0     5    -4
     4     0     5    -4
     5     1     5    -5
     6     1     5    -6
     6     1     5    -6
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
```

Notice that the inputs do not begin appearing at the output until the fifth row, the second row of the second frame. This is due to the block's latency.

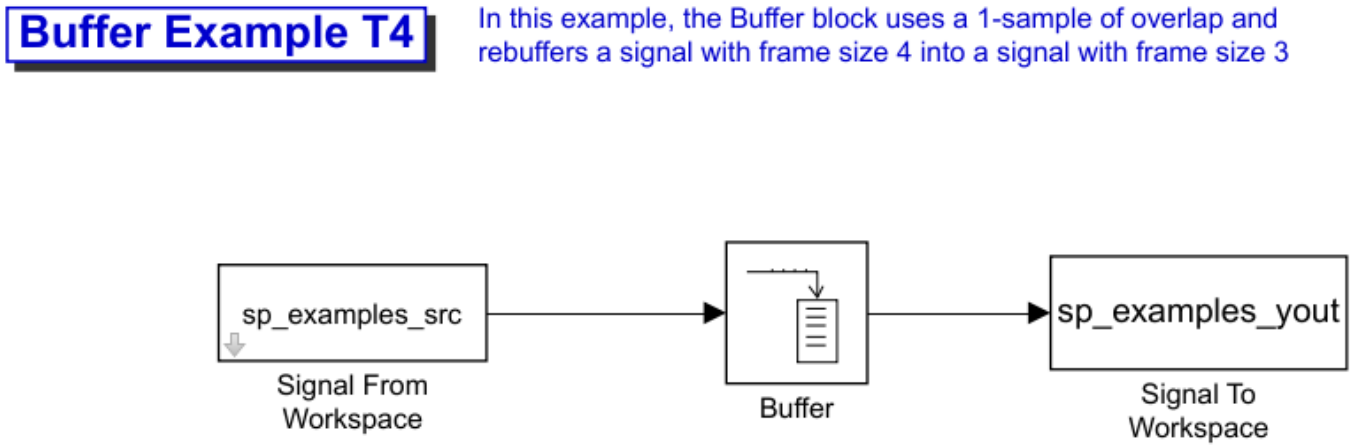
See “Excess Algorithmic Delay (Tasking Latency)” on page 3-40 for general information about algorithmic delay. For instructions on how to calculate buffering delay, see “Buffer Delay and Initial Conditions” on page 3-30.

Buffer Frame Inputs into Other Frame Inputs

In the following example, a two-channel signal with frame size 4 is rebuffered to a signal with frame size 3 and frame period 2. Because of the overlap, the input sample period is not conserved, and the output sample period is 2/3:

1 At the MATLAB command prompt, type `ex_buffer_tut4`.

The Buffer Example T4 model opens.



Also, the variable `sp_examples_src` is loaded into the MATLAB workspace. This variable is defined as

```
sp_examples_src = [1 1; 2 1; 3 0; 4 0; 5 1; 6 1; 7 0; 8 0]
```

2 Double-click the Signal From Workspace block. The **Source Block Parameters: Signal From Workspace** dialog box opens.

3 Set the block parameters as follows:

- **Signal** = `sp_examples_src`
- **Sample time** = 1
- **Samples per frame** = 4

Based on these parameters, the Signal From Workspace block outputs a two-channel frame signal with a sample period of 1 second and a frame size of 4.

4 Save these parameters and close the dialog box by clicking **OK**.

5 Double-click the Buffer block. The **Function Block Parameters: Buffer** dialog box opens.

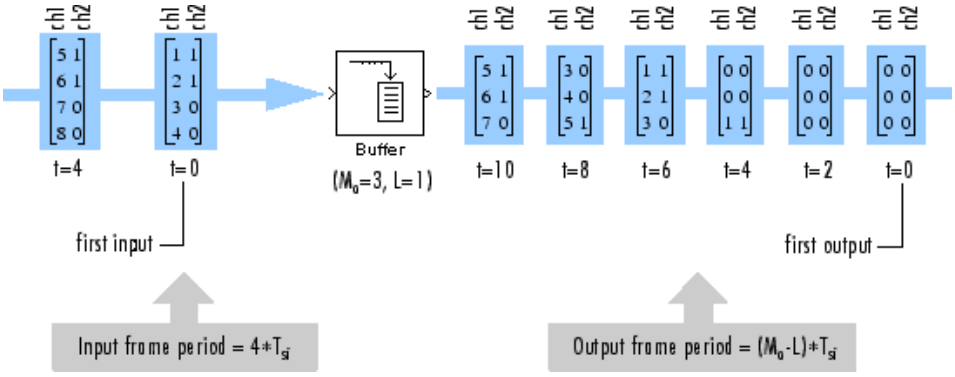
6 Set the block parameters as follows, and then click **OK**:

- **Output buffer size (per channel)** = 3
- **Buffer overlap** = 1
- **Initial conditions** = 0

Based on these parameters, the Buffer block outputs a two-channel frame signal with a frame size of 3.

7 Run the model.

The following figure is a graphical representation of the model's behavior during simulation.



Note that the inputs do not begin appearing at the output until the last row of the third output matrix. This is due to the block's latency.

See “Excess Algorithmic Delay (Tasking Latency)” on page 3-40 for general information about algorithmic delay. For instructions on how to calculate buffering delay, and see “Buffer Delay and Initial Conditions” on page 3-30.

Buffer Delay and Initial Conditions

In the examples “Buffer Signals into Frames with Overlap” on page 3-26 and “Buffer Frame Inputs into Other Frame Inputs” on page 3-28, the input signal is delayed by a certain number of samples. The initial output samples correspond to the value specified for the **Initial condition** parameter. The initial condition is zero in both examples mentioned above.

Under most conditions, the Buffer and Unbuffer blocks have some amount of delay or latency. This latency depends on both the block parameter settings and the Simulink tasking mode. You can use the `rebuffer_delay` function to determine the length of the block's latency for any combination of frame size and overlap.

The syntax `rebuffer_delay(f,n,v)` returns the delay, in samples, introduced by the buffering and unbuffering blocks during multitasking operations, where `f` is the input frame size, `n` is the **Output buffer size** parameter setting, and `v` is the **Buffer overlap** parameter setting.

For example, you can calculate the delay for the model discussed in the “Buffer Frame Inputs into Other Frame Inputs” on page 3-28 using the following command at the MATLAB command line:

```
d = rebuffer_delay(4,3,1)
d = 8
```

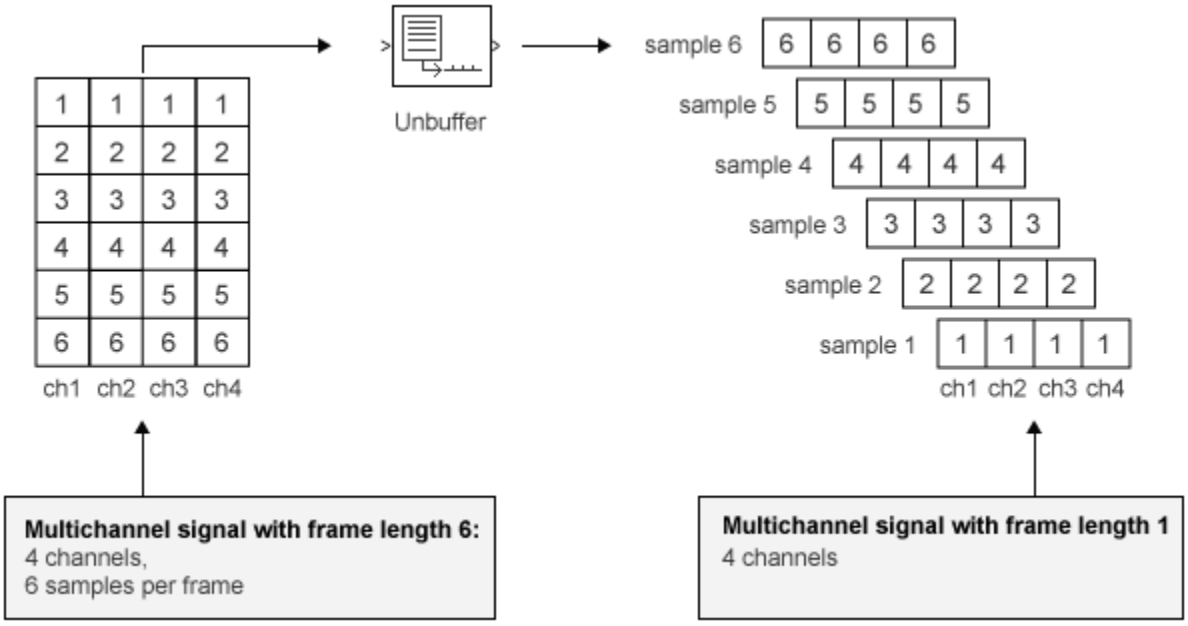
This result agrees with the block's output in that example. Notice that this model was simulated in Simulink multitasking mode.

For more information about delay, see “Excess Algorithmic Delay (Tasking Latency)” on page 3-40. For delay information about a specific block, see the “Latency” section of the block reference page. For more information about the `rebuffer_delay` function, see `rebuffer_delay`.

Unbuffer Frame Signals into Sample Signals

You can unbuffer multichannel signals of frame length greater than 1 into multichannel signals of frame length equal to 1 using the Unbuffer block. The Unbuffer block performs the inverse operation of the Buffer block's buffering process, where signals with frame length 1 are buffered into a signal with frame length greater than 1. The Unbuffer block generates an N-channel output containing one sample per frame from an N-channel input containing multiple channels per frame. The first row in each input matrix is always the first output.

The following figure is a graphical representation of this process.



The sample period of the output, T_{so}, is related to the input frame period, T_{fi}, by the input frame size, M_i.

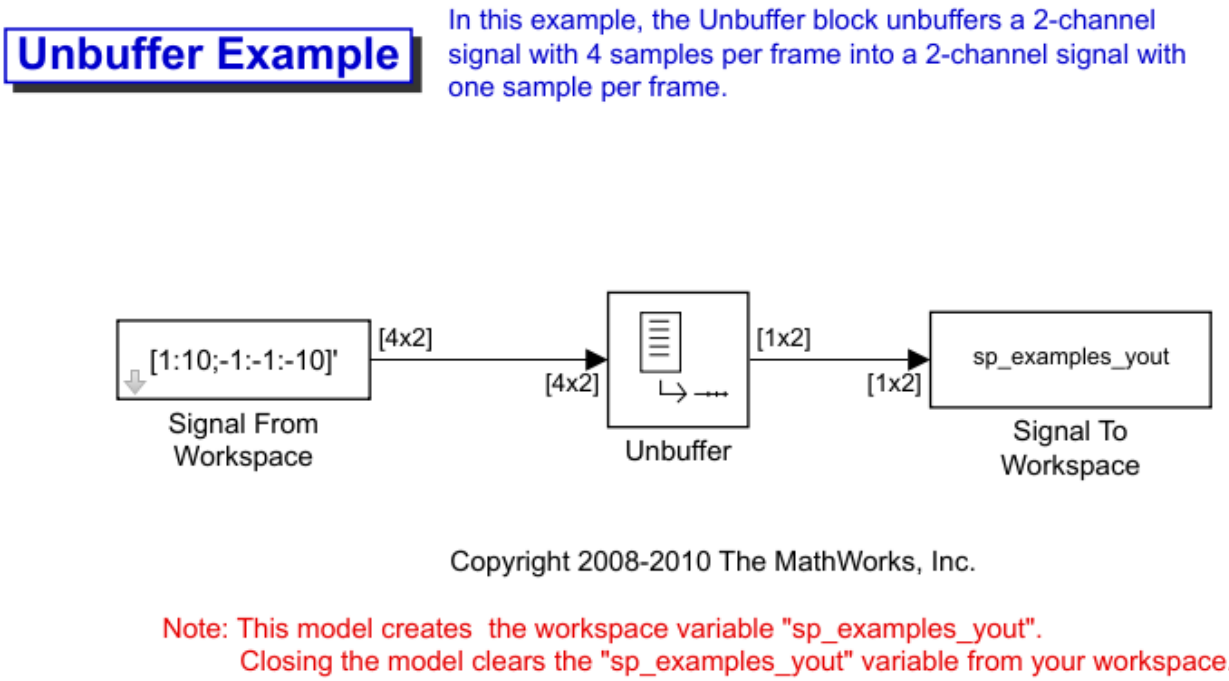
$$T_{so} = T_{fi}/M_i$$

The Unbuffer block always preserves the signal's sample period ($T_{so} = T_{si}$). See “Convert Sample and Frame Rates in Simulink” on page 3-13 for more information about rate conversions.

In the following example, a two-channel signal with four samples per frame is unbuffered into a two-channel signal with one sample per frame:

1 At the MATLAB command prompt, type `ex_unbuffer_tut`.

The Unbuffer Example model opens.



2 Double-click the Signal From Workspace block. The **Source Block Parameters: Signal From Workspace** dialog box opens.

3 Set the block parameters as follows:

- **Signal** = `[1:10;-1:-1:-10]'`
- **Sample time** = `1`
- **Samples per frame** = `4`
- **Form output after final data value by** = `Setting to zero`

Based on these parameters, the Signal From Workspace block outputs a two-channel signal with frame size 4.

4 Save these parameters and close the dialog box by clicking **OK**.

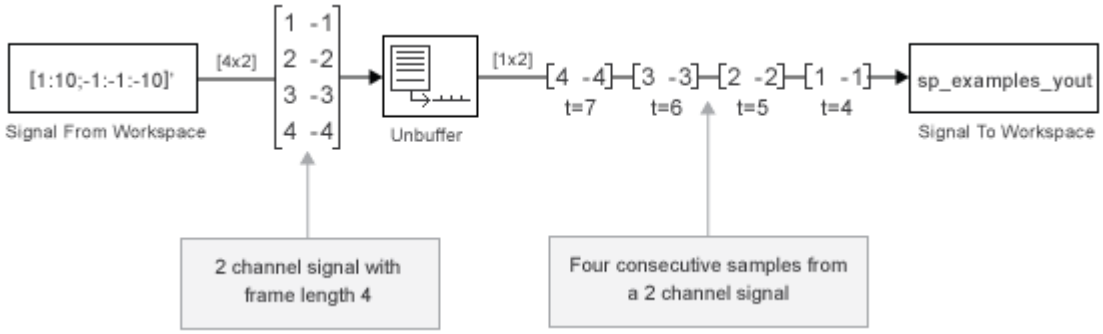
5 Double-click the Unbuffer block. The **Function Block Parameters: Unbuffer** dialog box opens.

6 Set the **Initial conditions** parameter to `0`, and then click **OK**.

The Unbuffer block unbuffers a two-channel signal with four samples per frame into a two-channel signal with one sample per frame.

7 Run the model.

The following figures is a graphical representation of what happens during the model simulation.



Note The Unbuffer block generates initial conditions not shown in the figure below with the value specified by the **Initial conditions** parameter. See the Unbuffer reference page for information about the number of initial conditions that appear in the output.

8 At the MATLAB command prompt, type `sp_examples_yout`.

The following is a portion of the output.

```
sp_examples_yout(:,:,1) =

     0     0


sp_examples_yout(:,:,2) =

     0     0


sp_examples_yout(:,:,3) =

     0     0


sp_examples_yout(:,:,4) =

     0     0


sp_examples_yout(:,:,5) =

     1    -1


sp_examples_yout(:,:,6) =

     2    -2


sp_examples_yout(:,:,7) =

     3    -3
```

The Unbuffer block unbuffers the signal into a two-channel signal. Each page of the output matrix represents a different sample time.

See Also

More About

- “Sample- and Frame-Based Concepts” on page 3-2
- “Delay and Latency” on page 3-35

Delay and Latency

In this section...
“Computational Delay” on page 3-35
“Algorithmic Delay” on page 3-36
“Zero Algorithmic Delay” on page 3-36
“Basic Algorithmic Delay” on page 3-38
“Excess Algorithmic Delay (Tasking Latency)” on page 3-40
“Predict Tasking Latency” on page 3-41

Computational Delay

The computational delay of a block or subsystem is related to the number of operations involved in executing that block or subsystem. For example, an FFT block operating on a 256-sample input requires Simulink software to perform a certain number of multiplications for each input frame. The actual amount of time that these operations consume depends heavily on the performance of both the computer hardware and underlying software layers, such as the MATLAB environment and the operating system. Therefore, computational delay for a particular model can vary from one computer platform to another.

The simulation time represented on a model's status bar, which can be accessed via the Simulink Digital Clock block, does not provide any information about computational delay. For example, according to the Simulink timer, the FFT mentioned above executes instantaneously, with no delay whatsoever. An input to the FFT block at simulation time t=25.0 is processed and output at simulation time t=25.0, regardless of the number of operations performed by the FFT algorithm. The Simulink timer reflects only algorithmic delay, not computational delay.

Reduce Computational Delay

There are a number of ways to reduce computational delay without actually running the simulation on faster hardware. To begin with, you should familiarize yourself with “Manual Performance Optimization” (Simulink) which describes some basic strategies. The following information discusses several options for improving performance.

A first step in improving performance is to analyze your model, and eliminate or simplify elements that are adding excessively to the computational load. Such elements might include scope displays and data logging blocks that you had put in place for debugging purposes and no longer require. In addition to these model-specific adjustments, there are a number of more general steps you can take to improve the performance of any model:

- Use frame-based processing wherever possible. It is advantageous for the entire model to be frame based. See “Benefits of Frame-Based Processing” on page 3-4 for more information.
- Use the DSP Simulink model templates to tailor Simulink for digital signal processing modeling. For more information, see Configure the Simulink Environment for Signal Processing Models.
- Turn off the Simulink status bar. In the **Modeling** tab, deselect **Environment > Status Bar**. Simulation speed will improve, but the time indicator will not be visible.
- Run your simulation from the MATLAB command line by typing

  ```
  sim(gcs)
  ```

This method of starting a simulation can greatly increase the simulation speed, but also has several limitations:

- You cannot interact with the simulation (to tune parameters, for instance).
- You must press **Ctrl+C** to stop the simulation, or specify start and stop times.
- There are no graphics updates in MATLAB S-functions.

- Use Simulink Coder code generation software to generate generic real-time (GRT) code targeted to your host platform, and run the model using the generated executable file. See the Simulink Coder documentation for more information.

Algorithmic Delay

Algorithmic delay is delay that is intrinsic to the algorithm of a block or subsystem and is independent of CPU speed. In this guide, the algorithmic delay of a block is referred to simply as the block's delay. It is generally expressed in terms of the number of samples by which a block's output lags behind the corresponding input. This delay is directly related to the time elapsed on the Simulink timer during that block's execution.

The algorithmic delay of a particular block may depend on both the block parameter settings and the general Simulink settings. To simplify matters, it is helpful to categorize a block's delay using the following categories:

- “Zero Algorithmic Delay” on page 3-36
- “Basic Algorithmic Delay” on page 3-38
- “Excess Algorithmic Delay (Tasking Latency)” on page 3-40

The following topics explain the different categories of delay, and how the simulation and parameter settings can affect the level of delay that a particular block experiences.

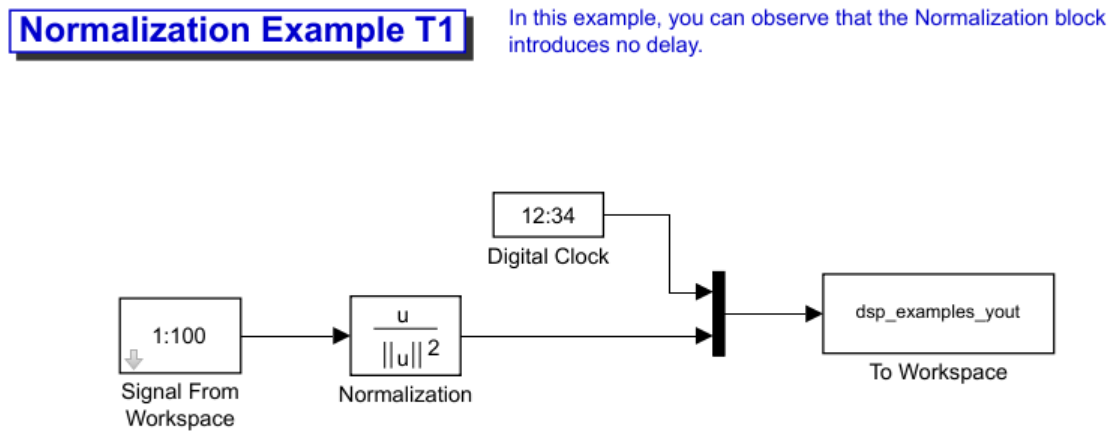
Zero Algorithmic Delay

The FFT block is an example of a component that has no algorithmic delay. The Simulink timer does not record any passage of time while the block computes the FFT of the input, and the transformed data is available at the output in the same time step that the input is received. There are many other blocks that have zero algorithmic delay, such as the blocks in the Matrices and Linear Algebra libraries. Each of those blocks processes its input and generates its output in a single time step.

The Normalization block is an example of a block with zero algorithmic delay:

1. At the MATLAB command prompt, type `ex_normalization_tut`.

   The Normalization Example T1 model opens.



2. Double-click the Signal From Workspace block. The **Source Block Parameters: Signal From Workspace** dialog box opens.
3. Set the block parameters as follows:
   - **Signal** = `1:100`
   - **Sample time** = `1/4`
   - **Samples per frame** = 4
4. Save these parameters and close the dialog box by clicking **OK**.
5. Run the model.

   The model prepends the current value of the Simulink timer output from the Digital Clock block to each output frame.

   The Signal From Workspace block generates a new frame containing four samples once every second (T_{fo} = π*4). The first few output frames are:

```
(t=0)        [ 1  2  3  4]'
(t=1)        [ 5  6  7  8]'
(t=2)        [ 9 10 11 12]'
(t=3)        [13 14 15 16]'
(t=4)        [17 18 19 20]'
```

6. At the MATLAB command prompt, type `squeeze(dsp_examples_yout)'`.

   The normalized output, `dsp_examples_yout`, is converted to an easier-to-read matrix format. The result, `ans`, is shown in the following figure:

```
ans =

         0    0.0333    0.0667    0.1000    0.1333
    1.0000    0.0287    0.0345    0.0402    0.0460
    2.0000    0.0202    0.0224    0.0247    0.0269
    3.0000    0.0154    0.0165    0.0177    0.0189
```

```
    4.0000    0.0124    0.0131    0.0138    0.0146
    5.0000    0.0103    0.0108    0.0113    0.0118
```

The first column of `ans` is the Simulink time provided by the Digital Clock block. You can see that the squared 2-norm of the first input,

```
[1 2 3 4]' ./ sum([1 2 3 4]'.^2)
```

appears in the first row of the output (at time t=0), the same time step that the input was received by the block. This indicates that the Normalization block has zero algorithmic delay.

Zero Algorithmic Delay and Algebraic Loops

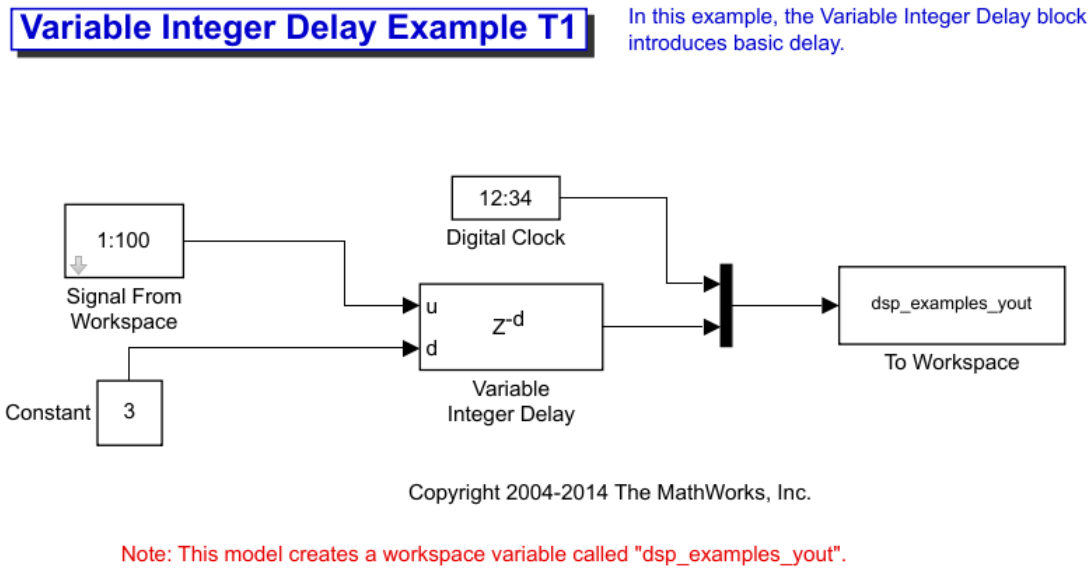
When several blocks with zero algorithmic delay are connected in a feedback loop, Simulink may report an algebraic loop error and performance may generally suffer. You can prevent algebraic loops by injecting at least one sample of delay into a feedback loop , for example, by including a Delay block with **Delay** > 0. For more information, see “Algebraic Loop Concepts” (Simulink).

Basic Algorithmic Delay

The Variable Integer Delay block is an example of a block with algorithmic delay. In the following example, you use this block to demonstrate this concept:

1 At the MATLAB command prompt, type `ex_variableintegerdelay_tut`.

The Variable Integer Delay Example T1 opens.



2 Double-click the Signal From Workspace block. The **Source Block Parameters: Signal From Workspace** dialog box opens.

3 Set the block parameters as follows:

- **Signal** = `1:100`
- **Sample time** = `1`

- **Samples per frame** = 1

4 Save these parameters and close the dialog box by clicking **OK**.

5 Double-click the Constant block. The **Source Block Parameters: Constant** dialog box opens.

6 Set the block parameters as follows:

- **Constant value** = 3
- **Interpret vector parameters as 1-D** = Clear this check box
- **Sample time** = 1

Click **OK** to save these parameters and close the dialog box.

The input to the `Delay` port of the Variable Integer Delay block specifies the number of sample periods that should elapse before an input to the `In` port is released to the output. This value represents the block's algorithmic delay. In this example, since the input to the `Delay` port is `3`, and the sample period at the In and Delay ports is 1, then the sample that arrives at the block's In port at time t=0 is released to the output at time t=3.

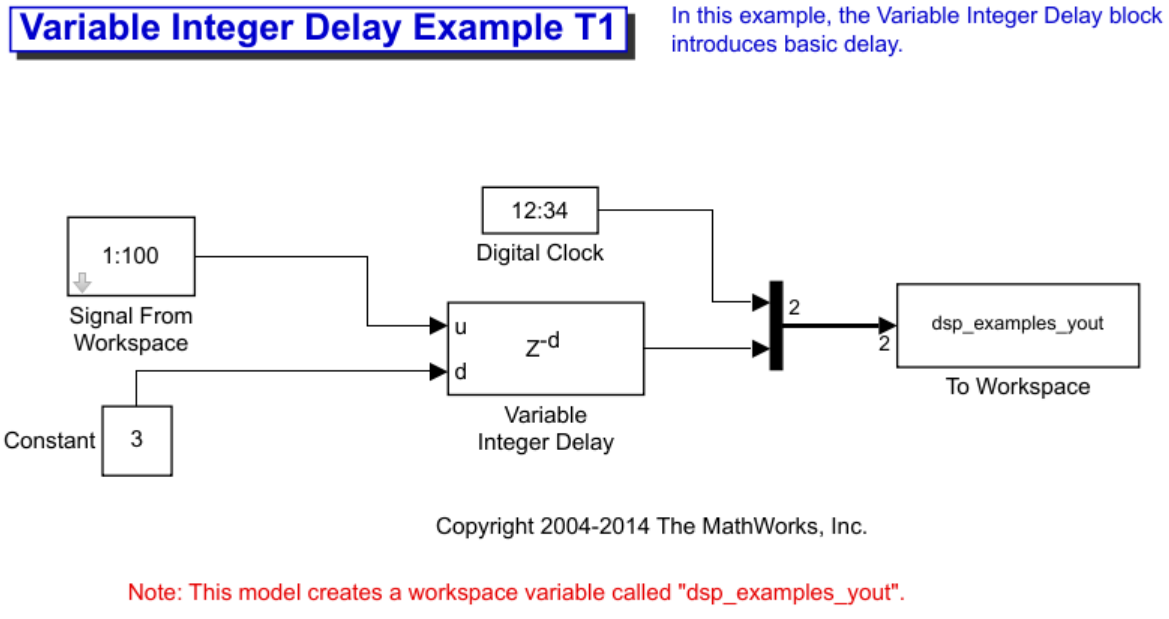
7 Double-click the Variable Integer Delay block. The **Function Block Parameters: Variable Integer Delay** dialog box opens.

8 Set the **Initial conditions** parameter to `-1`, and then click **OK**.

9 In the **Debug** tab, select **Information Overlays** > **Signal Dimensions** and **Nonscalar Signals**.

10 Run the model.

The model should look similar to the following figure.



11 At the MATLAB command prompt, type `dsp_examples_yout`

The output is shown below:

```
dsp_examples_yout =
```

```
0    -1
1    -1
2    -1
3     1
4     2
5     3
```

The first column is the Simulink time provided by the Digital Clock block. The second column is the delayed input. As expected, the input to the block at t=0 is delayed three samples and appears as the fourth output sample, at t=3. You can also see that the first three outputs from the Variable Integer Delay block inherit the value of the block's **Initial conditions** parameter, `-1`. This period of time, from the start of the simulation until the first input is propagated to the output, is sometimes called the *initial delay* of the block.

Many DSP System Toolbox blocks have some degree of fixed or adjustable algorithmic delay. These include any blocks whose algorithms rely on delay or storage elements, such as filters or buffers. Often, but not always, such blocks provide an **Initial conditions** parameter that allows you to specify the output values generated by the block during the initial delay. In other cases, the initial conditions are internally set to 0.

Consult the block reference pages for the delay characteristics of specific DSP System Toolbox blocks.

Excess Algorithmic Delay (Tasking Latency)

Under certain conditions, Simulink may force a block to delay inputs longer than is strictly required by the block's algorithm. This excess algorithmic delay is called tasking latency, because it arises from synchronization requirements of the Simulink tasking mode. A block's overall algorithmic delay is the sum of its basic delay and tasking latency.

Algorithmic delay = Basic algorithmic delay + Tasking latency

The tasking latency for a particular block may be dependent on the following block and model characteristics:

- “Simulink Tasking Mode” on page 3-40
- “Block Rate Type” on page 3-41
- “Model Rate Type” on page 3-41
- “Block Input Processing Mode” on page 3-41

Simulink Tasking Mode

Simulink has two tasking modes:

- Single-tasking
- Multitasking

In the **Modeling** tab, click **Model Settings**. In the **Solver** pane, select **Type** > `Fixed-step`. Expand **Solver details**. To specify multitasking mode, select **Treat each discrete rate as a separate task**. To specify single-tasking mode, clear **Treat each discrete rate as a separate task**.

Note Many multirate blocks have reduced latency in the Simulink single-tasking mode. Check the “Latency” section of a multirate block's reference page for details. Also see “Time-Based Scheduling and Code Generation” (Simulink Coder).

Block Rate Type

A block is called single-rate when all of its input and output ports operate at the same frame rate. A block is called multirate when at least one input or output port has a different frame rate than the others.

Many blocks are permanently single-rate. This means that all input and output ports always have the same frame rate. For other blocks, the block parameter settings determine whether the block is single-rate or multirate. Only multirate blocks are subject to tasking latency.

Note Simulink may report an algebraic loop error if it detects a feedback loop composed entirely of multirate blocks. To break such an algebraic loop, insert a single-rate block with nonzero delay, such as a Unit Delay block. For more information, see “Algebraic Loop Concepts” (Simulink).

Model Rate Type

When all ports of all blocks in a model operate at a single frame rate, the model is called single-rate. When the model contains blocks with differing frame rates, or at least one multirate block, the model is called multirate. Note that Simulink prevents a single-rate model from running in multitasking mode by generating an error.

Block Input Processing Mode

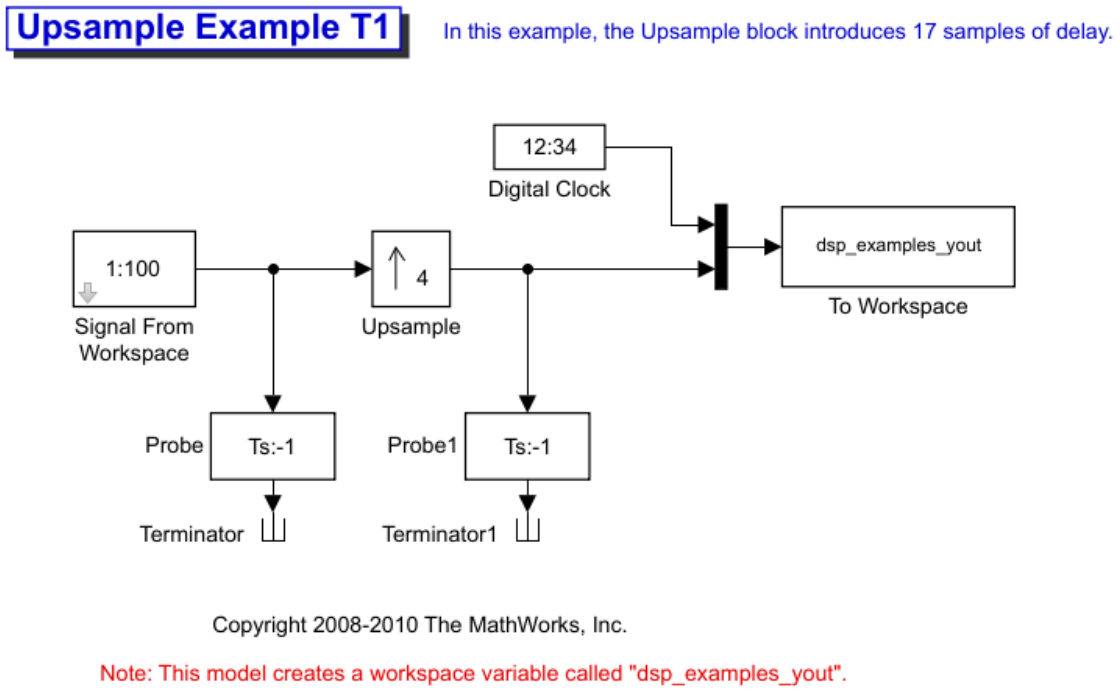
Many blocks can operate in either sample-based or frame-based processing modes. To choose, you can set the **Input processing** parameter of the block to `Columns as channels (frame based)` or `Elements as channels (sample based)`.

Predict Tasking Latency

The specific amount of tasking latency created by a particular combination of block parameter and simulation settings is discussed in the “Latency” section of a block's reference page. In this topic, you use the Upsample block's reference page to predict the tasking latency of a model:

1. At the MATLAB command prompt, type `ex_upsample_tut1`.

   The Upsample Example T1 model opens.



2. In the **Modeling** tab, click **Model Settings**.
3. In the **Solver** pane, from the **Type** list, select `Fixed-step`. From the **Solver** list, select `discrete (no continuous states)`.
4. Expand **Solver details**. Select **Treat each discrete rate as a separate task** and click **OK**.

   Most multirate blocks experience tasking latency only in the Simulink multitasking mode.
5. Double-click the Signal From Workspace block. The **Source Block Parameters: Signal From Workspace** dialog box opens.
6. Set the block parameters as follows, and then click **OK**:

   - **Signal** = `1:100`
   - **Sample time** = `1/4`
   - **Samples per frame** = `4`
   - **Form output after final data value by** = `Setting to zero`
7. Double-click the Upsample block. The **Function Block Parameters: Upsample** dialog box opens.
8. Set the block parameters as follows, and then click **OK**:

   - **Upsample factor, L** = `4`
   - **Sample offset (0 to L-1)** = `0`
   - **Input processing** = `Columns as channels (frame based)`
   - **Rate options** = `Allow multirate processing`
   - **Initial condition** = `-1`

   The **Rate options** parameter makes the model multirate, since the input and output frame rates will not be equal.

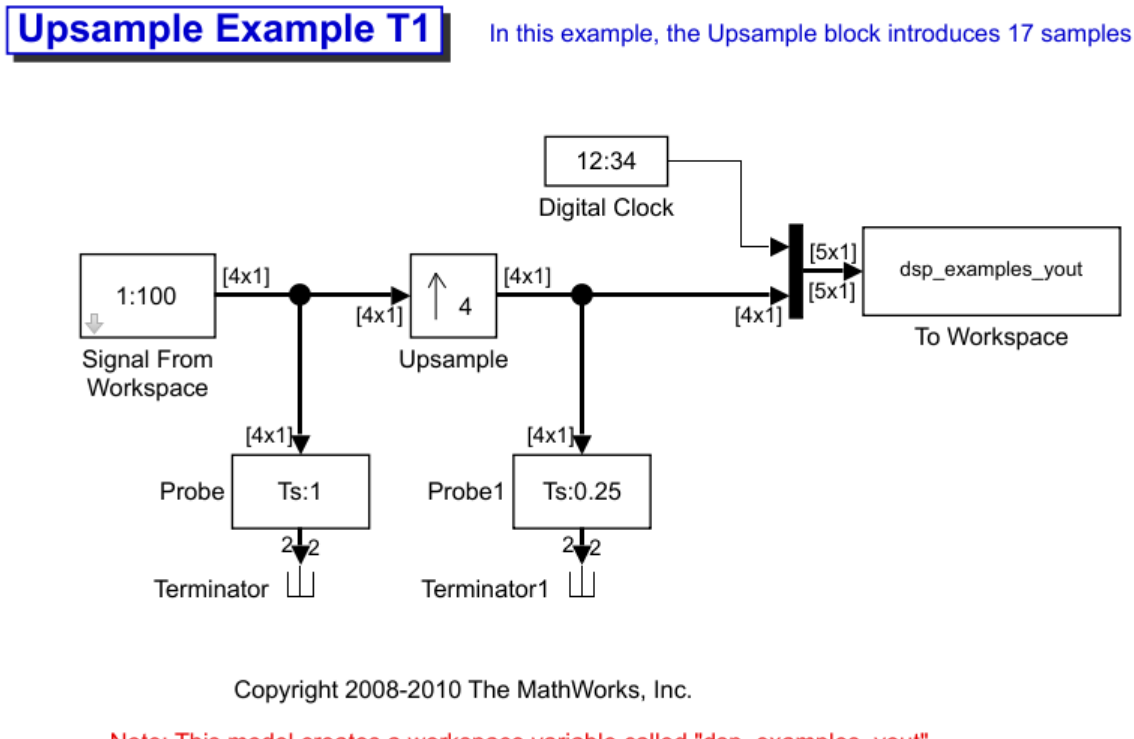
9 Double-click the Digital Clock block. The **Source Block Parameters: Digital Clock** dialog box opens.

10 Set the **Sample time** parameter to `0.25`, and then click **OK**.

This matches the sample period of the Upsample block's output.

11 Run the model.

The model should now look similar to the following figure.



The model prepends the current value of the Simulink timer, from the Digital Clock block, to each output frame.

In the example, the Signal From Workspace block generates a new frame containing four samples once every second (T_{fo} = π*4). The first few output frames are:

```
(t=0)        [ 1  2  3  4]
(t=1)        [ 5  6  7  8]
(t=2)        [ 9 10 11 12]
(t=3)        [13 14 15 16]
(t=4)        [17 18 19 20]
```

The Upsample block upsamples the input by a factor of 4, inserting three zeros between each input sample. The change in rates is confirmed by the Probe blocks in the model, which show a decrease in the frame period from T_{fi} = 1 to T_{fo} = 0.25.

12 At the MATLAB command prompt, type `squeeze(dsp_examples_yout)'`.

The output from the simulation is displayed in a matrix format. The first few samples of the result, `ans`, are:

```
ans =
```

0	-1.0000	0	0	0	1st output frame
0.2500	-1.0000	0	0	0	
0.5000	-1.0000	0	0	0	
0.7500	-1.0000	0	0	0	
1.0000	1.0000	0	0	0	5th output frame
1.2500	2.0000	0	0	0	
1.5000	3.0000	0	0	0	
1.7500	4.0000	0	0	0	
2.0000	5.0000	0	0	0	
time					

“Latency and Initial Conditions” in the Upsample block's reference page indicates that when Simulink is in multitasking mode, the first sample of the block's input appears in the output as sample M_iL+D+1, where M_i is the input frame size, L is the **Upsample factor**, and D is the **Sample offset**. This formula predicts that the first input in this example should appear as output sample 17 (that is, 4*4+0+1).

The first column of the output is the Simulink time provided by the Digital Clock block. The four values to the right of each time are the values in the output frame at that time. You can see that the first sample in each of the first four output frames inherits the value of the Upsample block's **Initial conditions** parameter. As a result of the tasking latency, the first input value appears as the first sample of the 5th output frame (at t=1). This is sample 17.

Now try running the model in single-tasking mode.

13 In the **Modeling** tab, click **Model Settings**.

14 In the **Solver** pane, from the **Type** list, select `Fixed-step`. From the **Solver** list, select `Discrete (no continuous states)`.

15 Clear the **Treat each discrete rate as a separate task** parameter.

16 Run the model.

The model now runs in single-tasking mode.

17 At the MATLAB command prompt, type `squeeze(dsp_examples_yout)'`.

The first few samples of the result, `ans`, are:

```
ans =
```

0	1.0000	0	0	0	1st output frame
0.2500	2.0000	0	0	0	
0.5000	3.0000	0	0	0	
0.7500	4.0000	0	0	0	
1.0000	5.0000	0	0	0	5th output frame
1.2500	6.0000	0	0	0	
1.5000	7.0000	0	0	0	
1.7500	8.0000	0	0	0	
2.0000	9.0000	0	0	0	
time					

“Latency and Initial Conditions” in the Upsample block's reference page indicates that the block has zero latency for all multirate operations in the Simulink single-tasking mode.

The first column of the output is the Simulink time provided by the Digital Clock block. The four values to the right of each time are the values in the output frame at that time. The first input value appears as the first sample of the first output frame (at t=0). This is the expected behavior for the zero-latency condition. For the particular parameter settings used in this example, running `upsample_tut1` in single-tasking mode eliminates the 17-sample delay that is present when you run the model in multitasking mode.

You have now successfully used the Upsample block's reference page to predict the tasking latency of a model.

See Also

More About

- “Sample- and Frame-Based Concepts” on page 3-2
- “Buffering and Frame-Based Processing” on page 3-24

Variable-Size Signal Support DSP System Objects

In this section...
“Variable-Size Signal Support Example” on page 3-46
“DSP System Toolbox System Objects That Support Variable-Size Signals” on page 3-46

Several DSP System Toolbox System objects support variable-size input signals. In these System objects, you can change the frame size (number of rows) of the input matrix even when the object is locked. The number of channels (number of columns) of the input matrix must remain constant. The System object locks when you call the object to run its algorithm.

Variable-Size Signal Support Example

Note: If you are using R2016a or an earlier release, replace each call to the object with the equivalent step syntax. For example, `obj(x)` becomes `step(obj,x)`.

Create a `dsp.FIRHalfbandDecimator` System object™. The input signal contains 10 channels, with 1000 samples in each channel.

```
FIRHalfband = dsp.FIRHalfbandDecimator;
input = randn(1000,10);
```

Lock the object by running the algorithm.

```
FIRHalfband(input);
isLocked(FIRHalfband)
```

```
ans = logical
   1
```

Change the frame size of the input to 800 without releasing the object.

```
input = randn(800,10);
FIRHalfband(input);
```

The System object runs without error.

DSP System Toolbox System Objects That Support Variable-Size Signals

Sources
`dsp.UDPReceiver`
Sinks
`dsp.SpectrumAnalyzer`
`dsp.UDPSender`
Adaptive Filters
`dsp.AdaptiveLatticeFilter`

`dsp.AffineProjectionFilter`
`dsp.FastTransversalFilter`
`dsp.FilteredXLMSFilter`
`dsp.FrequencyDomainAdaptiveFilter`
`dsp.KalmanFilter`
`dsp.LMSFilter`
`dsp.RLSFilter`
Filter Designs
`dsp.Channelizer`
`dsp.ChannelSynthesizer`
`dsp.Differentiator`
`dsp.FilterCascade` (if the cascaded filters support variable-size signals)
`dsp.FIRHalfbandDecimator`
`dsp.FIRHalfbandInterpolator`
`dsp.HampelFilter`
`dsp.HighpassFilter`
`dsp.IIRHalfbandDecimator`
`dsp.IIRHalfbandInterpolator`
`dsp.LowpassFilter`
`dsp.NotchPeakFilter`
`dsp.VariableBandwidthFIRFilter`
`dsp.VariableBandwidthIIRFilter`
Filter Implementations
`dsp.AllpassFilter`
`dsp.AllpoleFilter`
`dsp.BiquadFilter`
`dsp.CoupledAllpassFilter`
`dsp.FIRFilter`
Multirate Filters
`dsp.FIRDecimator`
`dsp.FIRInterpolator`
Transforms
`dsp.FFT`
`dsp.IFFT`
Measurements and Statistics
`dsp.MovingAverage`
`dsp.MovingMaximum`

dsp.MovingMinimum
dsp.MovingRMS
dsp.MovingStandardDeviation
dsp.MovingVariance
dsp.MedianFilter
dsp.PeakToPeak
dsp.PeakToRMS
dsp.PulseMetrics
dsp.StateLevels
Signal Operations
dsp.DCBlocker
dsp.Delay
dsp.VariableFractionalDelay
dsp.PhaseExtractor
Matrix Operations
dsp.ArrayVectorAdder
dsp.ArrayVectorDivider
dsp.ArrayVectorMultiplier
dsp.ArrayVectorSubtractor
Signal Management
dsp.AsyncBuffer

For a list of DSP System Toolbox blocks that support variable-size signals, open the block data type support table from the MATLAB command prompt:

```
showsignalblockdatatypetable
```

See the blocks with an X in the **Variable-Size Support** column of the block data type support table.

Filter Analysis, Design, and Implementation

- “Design a Filter in Fdesign — Process Overview” on page 4-2
- “Use Filter Designer with DSP System Toolbox Software” on page 4-9
- “FIR Nyquist (L-th band) Filter Design” on page 4-53
- “Digital Frequency Transformations” on page 4-60
- “Digital Filter Design Block” on page 4-83
- “Filter Realization Wizard” on page 4-90
- “Digital Filter Implementations” on page 4-100
- “Removing High-Frequency Noise from an ECG Signal” on page 4-108

Design a Filter in Fdesign — Process Overview

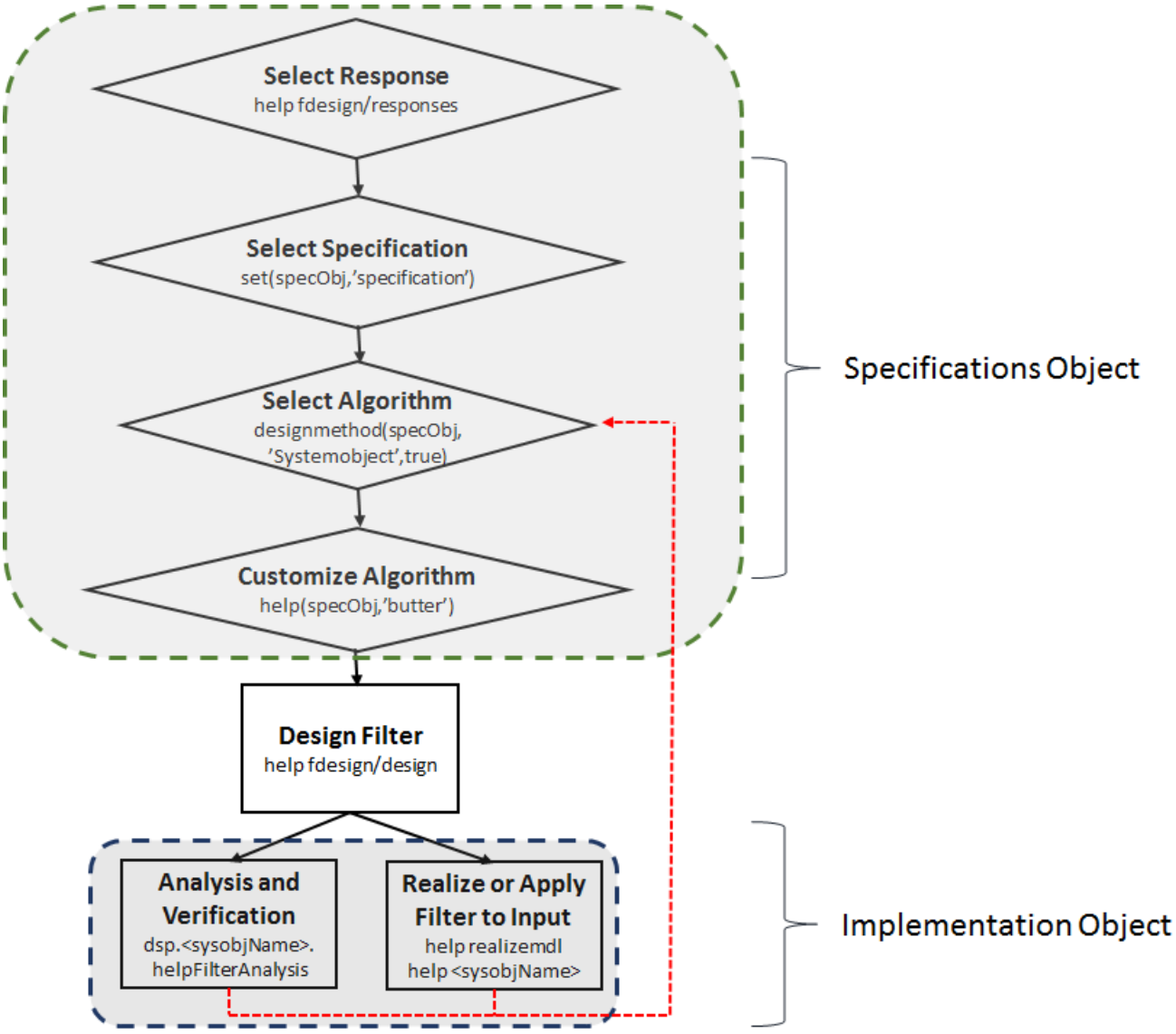
Process Flow Diagram and Filter Design Methodology

- “Exploring the Process Flow Diagram” on page 4-2
- “Select a Response” on page 4-4
- “Select a Specification” on page 4-4
- “Select an Algorithm” on page 4-5
- “Customize the Algorithm” on page 4-6
- “Design the Filter” on page 4-7
- “Design Analysis” on page 4-7
- “Realize or Apply the Filter to Input Data” on page 4-8

Note

Exploring the Process Flow Diagram

The process flow diagram shown in the following figure lists the steps and shows the order of the filter design process.



The first four steps of the filter design process relate to the filter Specifications Object, while the last two steps involve the filter Implementation Object. Both of these objects are discussed in more detail in the following sections. Step 5 - the design of the filter, is the transition step from the filter Specifications Object to the Implementation object. The analysis and verification step is completely optional. It provides methods for the filter designer to ensure that the filter complies with all design criteria. Depending on the results of this verification, you can loop back to steps 3 and 4, to either choose a different algorithm, or to customize the current one. You may also wish to go back to steps 3 or 4 after you filter the input data with the designed filter (step 7), and find that you wish to tweak the filter or change it further.

The diagram shows the help command for each step. Enter the help line at the MATLAB command prompt to receive instructions and further documentation links for the particular step. Not all of the steps have to be executed explicitly. For example, you could go from step 1 directly to step 5, and the interim three steps are done for you by the software.

The following are the details for each of the steps shown above.

Select a Response

If you type:

```
help fdesign/responses
```

at the MATLAB command prompt, you see a list of all available filter responses.

You must select a response to initiate the filter. In this example, a bandpass filter Specifications Object is created by typing the following:

```
d = fdesign.bandpass
```

Select a Specification

A *specification* is an array of design parameters for a given filter. The specification is a property of the Specifications Object.

Note A specification is not the same as the Specifications Object. A Specifications Object contains a specification as one of its properties.

When you select a filter response, there are a number of different specifications available. Each one contains a different combination of design parameters. After you create a filter Specifications Object, you can query the available specifications for that response. Specifications marked with an asterisk require the DSP System Toolbox.

```
d = fdesign.bandpass; % step 1 - choose the response
set (d, 'specification')

ans =

  16×1 cell array

    'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2'
    'N,F3dB1,F3dB2'
    'N,F3dB1,F3dB2,Ap'
    'N,F3dB1,F3dB2,Ast'
    'N,F3dB1,F3dB2,Ast1,Ap,Ast2'
    'N,F3dB1,F3dB2,BWp'
    'N,F3dB1,F3dB2,BWst'
    'N,Fc1,Fc2'
    'N,Fc1,Fc2,Ast1,Ap,Ast2'
    'N,Fp1,Fp2,Ap'
    'N,Fp1,Fp2,Ast1,Ap,Ast2'
    'N,Fst1,Fp1,Fp2,Fst2'
    'N,Fst1,Fp1,Fp2,Fst2,C'
    'N,Fst1,Fp1,Fp2,Fst2,Ap'
    'N,Fst1,Fst2,Ast'
    'Nb,Na,Fst1,Fp1,Fp2,Fst2'

d = fdesign.arbmag;
set(d,'specification')

ans =
```

```
  7×1 cell array

    'N,F,A'
    'F,A,R'
    'Nb,Na,F,A'
    'N,B,F,A'
    'N,B,F,A,C'
    'B,F,A,R'
    'Nb,Na,B,F,A'
```

The `set` command can be used to select one of the available specifications as follows:

```
d = fdesign.lowpass;
% step 1: get a list of available specifications
set (d, 'specification')

ans =

  18×1 cell array

    'Fp,Fst,Ap,Ast'
    'N,F3dB'
    'Nb,Na,F3dB'
    'N,F3dB,Ap'
    'N,F3dB,Ap,Ast'
    'N,F3dB,Ast'
    'N,F3dB,Fst'
    'N,Fc'
    'N,Fc,Ap,Ast'
    'N,Fp,Ap'
    'N,Fp,Ap,Ast'
    'N,Fp,F3dB'
    'N,Fp,Fst'
    'N,Fp,Fst,Ap'
    'N,Fp,Fst,Ast'
    'N,Fst,Ap,Ast'
    'N,Fst,Ast'
    'Nb,Na,Fp,Fst'

% step 2: set the required specification
 set (d, 'specification', 'N,Fc')
```

If you do not perform this step explicitly, `fdesign` returns the default specification for the response you chose in “Select a Response” on page 4-4, and provides default values for all design parameters included in the specification.

Select an Algorithm

The availability of algorithms depends the chosen filter response, the design parameters, and the availability of the DSP System Toolbox. In other words, for the same lowpass filter, changing the specification entry also changes the available algorithms. In the following example, for a lowpass filter and a specification of `'N, Fc'`, only one algorithm is available—`window`.

```
% step 2: set the required specification
set (d, 'specification', 'N,Fc')
% step 3: get available algorithms
designmethods (d,'Systemobject',true)
```

```
Design Methods that support System objects for class fdesign.lowpass (N,Fc):


window
```

However, for a specification of `'Fp,Fst,Ap,Ast'`, a number of algorithms are available.

```
set (d, 'specification', 'Fp,Fst,Ap,Ast')
designmethods(d,'Systemobject',true)
```

```
Design Methods that support System objects for class fdesign.lowpass (Fp,Fst,Ap,Ast):


butter
cheby1
cheby2
ellip
equiripple
ifir
kaiserwin
multistage
```

The user chooses a particular algorithm and implements the filter with the `design` function.

```
filt = design(d,'butter','Systemobject',true)
```

```
filt =

  dsp.BiquadFilter with properties:

                  Structure: 'Direct form II'
            SOSMatrixSource: 'Property'
                  SOSMatrix: [13×6 double]
                ScaleValues: [14×1 double]
          InitialConditions: 0
    OptimizeUnityScaleValues: true

  Show all properties
```

The preceding code creates the filter, where `filt` is the filter Implementation Object. This concept is discussed further in the next step.

If you do not perform this step explicitly, `design` automatically selects the optimum algorithm for the chosen response and specification.

Customize the Algorithm

The customization options available for any given algorithm depend not only on the algorithm itself, selected in “Select an Algorithm” on page 4-5, but also on the specification selected in “Select a Specification” on page 4-4. To explore all the available options, type the following at the MATLAB command prompt:

```
help (d, 'algorithm-name')
```

where `d` is the Filter Specification Object, and `algorithm-name` is the name of the algorithm in single quotes, such as `'butter'` or `'cheby1'`.

The application of these customization options takes place while “Design the Filter” on page 4-7, because these options are the properties of the filter Implementation Object, not the Specification Object.

If you do not perform this step explicitly, the optimum algorithm structure is selected.

Design the Filter

To create a filter, use the `design` command:

```
% Design filter without specifying the algorithm
filt = design(d,'Systemobject',true);
```

where `filt` is the filter object and `d` is the Specifications Object. This code creates a filter without specifying the algorithm. When the algorithm is not specified, the software selects the best available one.

To apply the algorithm chosen in “Select an Algorithm” on page 4-5, use the same `design` command, but specify the algorithm as follows:

```
filt = design(d,'butter','Systemobject',true)
```

where `filt` is the new filter object, and `d` is the specifications object.

To obtain help and see all the available options, type:

```
help fdesign/design
```

This help command describes not only the options for the `design` command itself, but also options that pertain to the method or the algorithm. If you are customizing the algorithm, you apply these options in this step. In the following example, you design a bandpass filter, and then modify the filter structure:

```
filt = design(d, 'butter', 'filterstructure', 'df2sos','Systemobject',true)

filt =

  dsp.BiquadFilter with properties:

                  Structure: 'Direct form II'
            SOSMatrixSource: 'Property'
                  SOSMatrix: [13×6 double]
                ScaleValues: [14×1 double]
          InitialConditions: 0
    OptimizeUnityScaleValues: true

  Show all properties
```

The filter design step, just like the first task of choosing a response, must be performed explicitly. A filter object is created only when `design` is called.

Design Analysis

After the filter is designed, you may wish to analyze it to determine if the filter satisfies the design criteria. Filter analysis is broken into these main sections:

- Frequency domain analysis — Includes the frequency response, group delay, pole-zero plots, and phase response through the functions `freqz`, `grpdelay`, `zplane`, and `phasez`.
- Time domain analysis — Includes impulse and step response through the functions `impz` and `stepz`.

- Implementation analysis — Includes cost estimate for implementing the filter, power spectral density of the filter output due to roundoff noise, and frequency response estimate of the filter through the functions `cost`, `noisepsd`, and `freqrespest`.

For a list of analysis methods for a discrete-time filter, enter the following in the MATLAB command prompt:

```
dsp.<sysobjName>.helpFilterAnalysis
```

Replace *<sysobjName>* with the name of the System object. Alternatively, you can see the list of analysis methods under the “Filter Analysis” category.

To analyze your filter, you must explicitly perform this step.

Realize or Apply the Filter to Input Data

After the filter is designed and optimized, it can be used to filter actual input data.

```
y = filt(x)
```

This step is never automatically performed for you. To filter your data, you must explicitly execute this step.

Note `y = filt(x)` runs only in R2016b or later. If you are using an earlier release, replace `y = filt(x)` with `y = step(filt,x)`.

Note If you have Simulink, you have the option of exporting this filter to a Simulink block using the `realizemdl` command. To get help on this command, type:

```
help realizemdl
```

Use Filter Designer with DSP System Toolbox Software

In this section...
“Design Advanced Filters in Filter Designer” on page 4-9
“Access the Quantization Features of Filter Designer” on page 4-11
“Quantize Filters in Filter Designer” on page 4-13
“Analyze Filters with a Noise-Based Method” on page 4-18
“Scale Second-Order Section Filters” on page 4-22
“Reorder the Sections of Second-Order Section Filters” on page 4-25
“View SOS Filter Sections” on page 4-28
“Import and Export Quantized Filters” on page 4-32
“Generate MATLAB Code” on page 4-35
“Import XILINX Coefficient (.COE) Files” on page 4-35
“Transform Filters Using Filter Designer” on page 4-36
“Design Multirate Filters in Filter Designer” on page 4-42
“Realize Filters as Simulink Subsystem Blocks” on page 4-50

Design Advanced Filters in Filter Designer

- “Overview of Filter Designer Features” on page 4-9
- “Use Filter Designer with DSP System Toolbox Software” on page 4-10
- “Design a Notch Filter” on page 4-10

Overview of Filter Designer Features

DSP System Toolbox software adds new dialog boxes and operating modes, and new menu selections, to the filter designer provided by Signal Processing Toolbox software. From the additional dialog boxes, one titled **Set Quantization Parameters** and one titled **Frequency Transformations**, you can:

- Design advanced filters that Signal Processing Toolbox software does not provide the design tools to develop.
- View Simulink models of the filter structures available in the toolbox.
- Quantize double-precision filters you design in this app using the design mode.
- Quantize double-precision filters you import into this app using the import mode.
- Analyze quantized filters.
- Scale second-order section filters.
- Select the quantization settings for the properties of the quantized filter displayed by the tool:
  - Coefficients — select the quantization options applied to the filter coefficients
  - Input/output — control how the filter processes input and output data
  - Filter Internals — specify how the arithmetic for the filter behaves
- Design multirate filters.

- Transform both FIR and IIR filters from one response to another.

After you import a filter into filter designer, the options on the quantization dialog box let you quantize the filter and investigate the effects of various quantization settings.

Options in the frequency transformations dialog box let you change the frequency response of your filter, keeping various important features while changing the response shape.

Use Filter Designer with DSP System Toolbox Software

Adding DSP System Toolbox software to your tool suite adds a number of filter design techniques to filter designer. Use the new filter responses to develop filters that meet more complex requirements than those you can design in Signal Processing Toolbox software. While the designs in filter designer are available as command line functions, the graphical user interface of filter designer makes the design process more clear and easier to accomplish.

As you select a response type, the options in the right panes in filter designer change to let you set the values that define your filter. You also see that the analysis area includes a diagram (called a *design mask*) that describes the options for the filter response you choose.

By reviewing the mask you can see how the options are defined and how to use them. While this is usually straightforward for lowpass or highpass filter responses, setting the options for the arbitrary response types or the peaking/notching filters is more complicated. Having the masks leads you to your result more easily.

Changing the filter design method changes the available response type options. Similarly, the response type you select may change the filter design methods you can choose.

Design a Notch Filter

Notch filters aim to remove one or a few frequencies from a broader spectrum. You must specify the frequencies to remove by setting the filter design options in filter designer appropriately:

- Response Type
- Design Method
- Frequency Specifications
- Magnitude Specifications

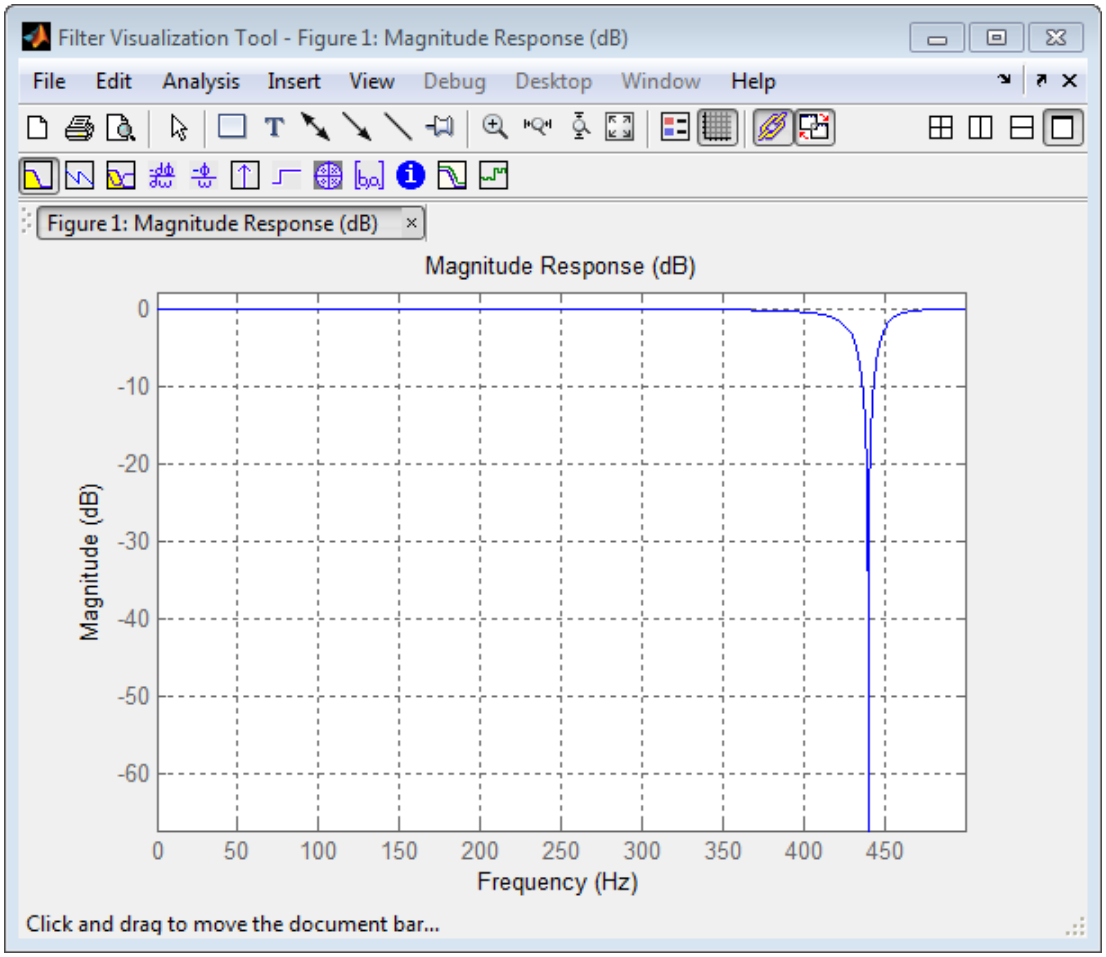
Here is how you design a notch filter that removes concert A (440 Hz) from an input musical signal spectrum.

1. Select `Notching` from the **Differentiator** list in **Response Type**.
2. Select **IIR** in **Filter Design Method** and choose `Single Notch` from the list.
3. For the **Frequency Specifications**, set **Units** to `Hz` and **Fs**, the full scale frequency, to `1000`.
4. Set the location of the center of the notch, in either normalized frequency or Hz. For the notch center at 440 Hz, enter `440`.
5. To shape the notch, enter the **bandwidth**, `bw`, to be `40`.
6. Leave the **Magnitude Specification** in `dB` (the default) and leave **Apass** as `1`.
7. Click Design Filter.

   filter designer computes the filter coefficients and plots the filter magnitude response in the analysis area for you to review.

When you design a single notch filter, you do not have the option of setting the filter order — the **Filter Order** options are disabled.

Your filter should look about like this:



For more information about a design method, refer to the online Help system. For instance, to get further information about the **Q** setting for the notch filter in filter designer, enter

```
doc iirnotch
```

at the command line. This opens the Help browser and displays the reference page for function `iirnotch`.

Designing other filters follows a similar procedure, adjusting for different design specification options as each design requires.

Any one of the designs may be quantized in filter designer and analyzed with the available analyses on the **Analysis** menu.

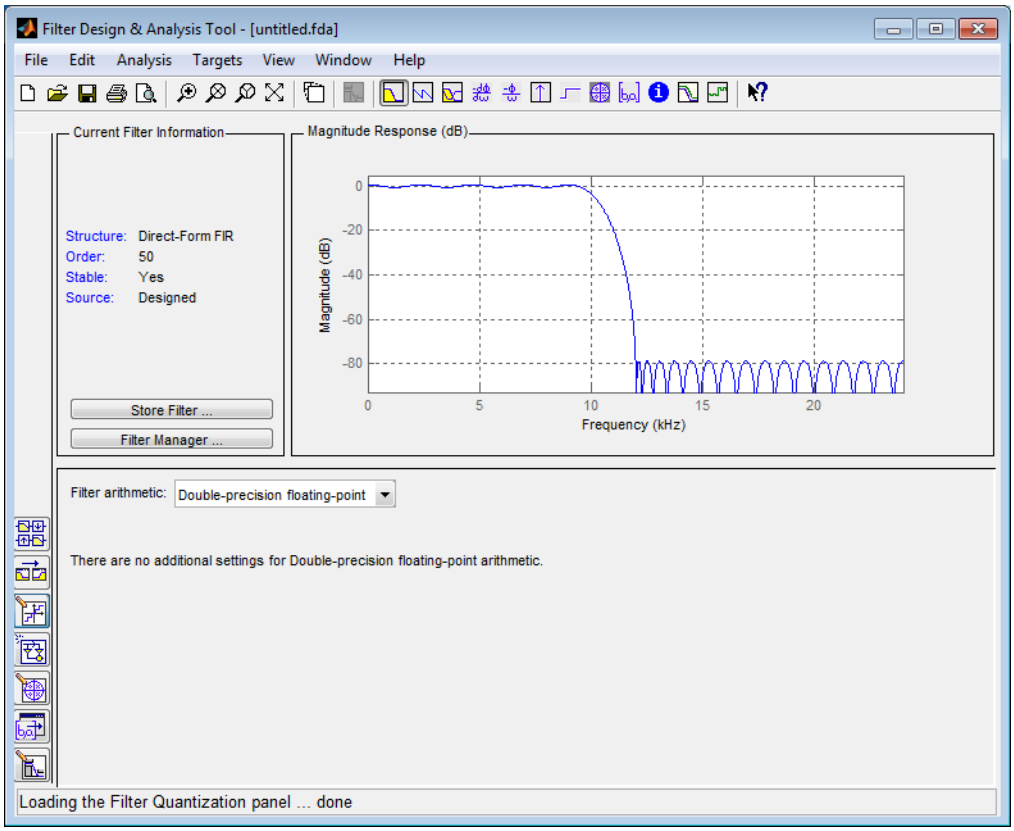
Access the Quantization Features of Filter Designer

You use the quantization panel in filter designer to quantize filters. Quantization represents the fourth operating mode for filter designer, along with the filter design, filter transformation, and import

modes. To switch to quantization mode, open filter designer from the MATLAB command prompt by entering

```
filterDesigner
```

When filter designer opens, click the **Set Quantization Parameters** button on the side bar. Filter designer switches to quantization mode and you see the following panel at the bottom of filter designer, with the default double-precision option shown for **Filter arithmetic**.



The **Filter arithmetic** option lets you quantize filters and investigate the effects of changing quantization settings. To enable the quantization settings in filter designer, select `Fixed-point` from the **Filter Arithmetic**.

The quantization options appear in the lower panel of filter designer. You see tabs that access various sets of options for quantizing your filter.

You use the following tabs in the dialog box to perform tasks related to quantizing filters in filter designer:

- **Coefficients** provides access the settings for defining the coefficient quantization. This is the default active panel when you switch filter designer to quantization mode without a quantized filter in the tool. When you import a fixed-point filter into filter designer, this is the active pane when you switch to quantization mode.
- **Input/Output** switches filter designer to the options for quantizing the inputs and outputs for your filter.

- **Filter Internals** lets you set a variety of options for the arithmetic your filter performs, such as how the filter handles the results of multiplication operations or how the filter uses the accumulator.
- **Apply** — applies changes you make to the quantization parameters for your filter.

Quantize Filters in Filter Designer

- "Set Quantization Parameters" on page 4-13
- "Coefficients Options" on page 4-13
- "Input/Output Options" on page 4-14
- "Filter Internals Options" on page 4-15
- "Filter Internals Options for CIC Filters" on page 4-17

Set Quantization Parameters

Quantized filters have properties that define how they quantize data you filter. Use the **Set Quantization Parameters** dialog box in filter designer to set the properties. Using options in the **Set Quantization Parameters** dialog box, filter designer lets you perform a number of tasks:

- Create a quantized filter from a double-precision filter after either importing the filter from your workspace, or using filter designer to design the prototype filter.
- Create a quantized filter that has the default structure (Direct form II transposed) or any structure you choose, and other property values you select.
- Change the quantization property values for a quantized filter after you design the filter or import it from your workspace.

When you click **Set Quantization Parameters**, and then change **Filter arithmetic** to `Fixed-point`, the quantized filter panel opens in filter designer, with the coefficient quantization options set to default values.

Coefficients Options

To let you set the properties for the filter coefficients that make up your quantized filter, filter designer lists options for numerator word length (and denominator word length if you have an IIR filter). The following table lists each coefficients option and a short description of what the option setting does in the filter.

Option Name	When Used	Description
Numerator Word Length	FIR filters only	Sets the word length used to represent numerator coefficients in FIR filters.
Numerator Frac. Length	FIR/IIR	Sets the fraction length used to interpret numerator coefficients in FIR filters.
Numerator Range (+/-)	FIR/IIR	Lets you set the range the numerators represent. You use this instead of the **Numerator Frac. Length** option to set the precision. When you enter a value x, the resulting range is -x to x. Range must be a positive integer.

Option Name	When Used	Description
Coefficient Word Length	IIR filters only	Sets the word length used to represent both numerator and denominator coefficients in IIR filters. You cannot set different word lengths for the numerator and denominator coefficients.
Denominator Frac. Length	IIR filters	Sets the fraction length used to interpret denominator coefficients in IIR filters.
Denominator Range (+/-)	IIR filters	Lets you set the range the denominator coefficients represent. You use this instead of the **Denominator Frac. Length** option to set the precision. When you enter a value x, the resulting range is -x to x. Range must be a positive integer.
Best-precision fraction lengths	All filters	Directs filter designer to select the fraction lengths for numerator (and denominator where available) values to maximize the filter performance. Selecting this option disables all of the fraction length options for the filter.
Scale Values frac. length	SOS IIR filters	Sets the fraction length used to interpret the scale values in SOS filters.
Scale Values range (+/-)	SOS IIR filters	Lets you set the range the SOS scale values represent. You use this with SOS filters to adjust the scaling used between filter sections. Setting this value disables the **Scale Values frac. length** option. When you enter a value x, the resulting range is -x to x. Range must be a positive integer.
Use unsigned representation	All filters	Tells filter designer to interpret the coefficients as unsigned values.
Scale the numerator coefficients to fully utilize the entire dynamic range	All filters	Directs filter designer to scale the numerator coefficients to effectively use the dynamic range defined by the numerator word length and fraction length format.

Input/Output Options

The options that specify how the quantized filter uses input and output values are listed in the table below.

Option Name	When Used	Description
Input Word Length	All filters	Sets the word length used to represent the input to a filter.
Input fraction length	All filters	Sets the fraction length used to interpret input values to filter.

Option Name	When Used	Description
Input range (+/-)	All filters	Lets you set the range the inputs represent. You use this instead of the **Input fraction length** option to set the precision. When you enter a value x, the resulting range is -x to x. Range must be a positive integer.
Output word length	All filters	Sets the word length used to represent the output from a filter.
Avoid overflow	All filters	Directs the filter to set the fraction length for the input to prevent the output values from exceeding the available range as defined by the word length. Clearing this option lets you set **Output fraction length**.
Output fraction length	All filters	Sets the fraction length used to represent output values from a filter.
Output range (+/-)	All filters	Lets you set the range the outputs represent. You use this instead of the **Output fraction length** option to set the precision. When you enter a value x, the resulting range is -x to x. Range must be a positive integer.
Stage input word length	SOS filters only	Sets the word length used to represent the input to an SOS filter section.
Avoid overflow	SOS filters only	Directs the filter to use a fraction length for stage inputs that prevents overflows in the values. When you clear this option, you can set **Stage input fraction length**.
Stage input fraction length	SOS filters only	Sets the fraction length used to represent input to a section of an SOS filter.
Stage output word length	SOS filters only	Sets the word length used to represent the output from an SOS filter section.
Avoid overflow	SOS filters only	Directs the filter to use a fraction length for stage outputs that prevents overflows in the values. When you clear this option, you can set **Stage output fraction length**.
Stage output fraction length	SOS filters only	Sets the fraction length used to represent the output from a section of an SOS filter.

Filter Internals Options

The options that specify how the quantized filter performs arithmetic operations are listed in the table below.

Option	Equivalent Filter Property (Using Wildcard *)	Description
Round towards	`RoundMode`	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). Choose from one of: • `ceil` - Round toward positive infinity. • `convergent` - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • `fix/zero` - Round toward zero. • `floor` - Round toward negative infinity. • `nearest` - Round toward nearest. Ties round toward positive infinity. • `round` - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.
Overflow Mode	`OverflowMode`	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either `saturate` (limit the output to the largest positive or negative representable value) or `wrap` (set overflowing values to the nearest representable value using modular arithmetic.
Filter Product (Multiply) Options		
Product Mode	`ProductMode`	Determines how the filter handles the output of product operations. Choose from full precision (`FullPrecision`), or whether to keep the most significant bit (`KeepMSB`) or least significant bit (`KeepLSB`) in the result when you need to shorten the word length. `Specify all` lets you set the fraction length applied to the results of product operations.
Product word length	`*ProdWordLength`	Sets the word length applied to interpret the results of multiply operations.
Num. fraction length	`NumProdFracLength`	Sets the fraction length used to interpret the results of product operations that involve numerator coefficients.
Den. fraction length	`DenProdFracLength`	Sets the fraction length used to interpret the results of product operations that involve denominator coefficients.
Filter Sum Options		

Option	Equivalent Filter Property (Using Wildcard *)	Description
Accum. mode	`AccumMode`	Determines how the accumulator outputs stored values. Choose from full precision (`FullPrecision`), or whether to keep the most significant bits (`KeepMSB`) or least significant bits (`KeepLSB`) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set this to `Specify all`.
Accum. word length	`*AccumWordLength`	Sets the word length used to store data in the accumulator/buffer.
Num. fraction length	`NumAccumFracLength`	Sets the fraction length used to interpret the numerator coefficients.
Den. fraction length	`DenAccumFracLength`	Sets the fraction length the filter uses to interpret denominator coefficients.
Cast signals before sum	`CastBeforeSum`	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams for each filter structure) before performing sum operations.
Filter State Options		
State word length	`*StateWordLength`	Sets the word length used to represent the filter states. Applied to both numerator- and denominator-related states
Avoid overflow	None	Prevent overflows in arithmetic calculations by setting the fraction length appropriately.
State fraction length	`*StateFracLength`	Lets you set the fraction length applied to interpret the filter states. Applied to both numerator- and denominator-related states

Note When you apply changes to the values in the Filter Internals pane, the plots for the **Magnitude response estimate** and **Round-off noise power spectrum** analyses update to reflect those changes. Other types of analyses are not affected by changes to the values in the Filter Internals pane.

Filter Internals Options for CIC Filters

CIC filters use slightly different options for specifying the fixed-point arithmetic in the filter. The next table shows and describes the options.

Quantize Double-Precision Filters

When you are quantizing a double-precision filter by switching to fixed-point or single-precision floating point arithmetic, follow these steps.

1. Click **Set Quantization Parameters** to display the **Set Quantization Parameters** pane in filter designer.
2. Select `Single-precision floating point` or `Fixed-point` from **Filter arithmetic**.

   When you select one of the optional arithmetic settings, filter designer quantizes the current filter according to the settings of the options in the Set Quantization Parameter panes, and changes the information displayed in the analysis area to show quantized filter data.
3. In the quantization panes, set the options for your filter. Set options for **Coefficients**, **Input/Output**, and **Filter Internals**.
4. Click **Apply**.

   Filter designer quantizes your filter using your new settings.
5. Use the analysis features in filter designer to determine whether your new quantized filter meets your requirements.

Change the Quantization Properties of Quantized Filters

When you are changing the settings for the quantization of a quantized filter, or after you import a quantized filter from your MATLAB workspace, follow these steps to set the property values for the filter:

1. Verify that the current filter is quantized.
2. Click **Set Quantization Parameters** to display the **Set Quantization Parameters** panel.
3. Review and select property settings for the filter quantization: **Coefficients**, **Input/Output**, and **Filter Internals**. Settings for options on these panes determine how your filter quantizes data during filtering operations.
4. Click **Apply** to update your current quantized filter to use the new quantization property settings from Step 3.
5. Use the analysis features in filter designer to determine whether your new quantized filter meets your requirements.

Analyze Filters with a Noise-Based Method

- “Analyze Filters with the Magnitude Response Estimate Method” on page 4-18
- “Compare the Estimated and Theoretical Magnitude Responses” on page 4-21
- “Select Quantized Filter Structures” on page 4-21
- “Convert the Structure of a Quantized Filter” on page 4-21
- “Convert Filters to Second-Order Sections Form” on page 4-22

Analyze Filters with the Magnitude Response Estimate Method

After you design and quantize your filter, the **Magnitude Response Estimate** option on the **Analysis** menu lets you apply the noise loading method to your filter. When you select **Analysis > Magnitude Response Estimate** from the menu bar, filter designer immediately starts the Monte Carlo trials that form the basis for the method and runs the analysis, ending by displaying the results in the analysis area in filter designer.

With the noise-based method, you estimate the complex frequency response for your filter as determined by applying a noise- like signal to the filter input. **Magnitude Response Estimate** uses

the Monte Carlo trials to generate a noise signal that contains complete frequency content across the range 0 to Fs. The first time you run the analysis, magnitude response estimate uses default settings for the various conditions that define the process, such as the number of test points and the number of trials.

Analysis Parameter	Default Setting	Description
Number of Points	`512`	Number of equally spaced points around the upper half of the unit circle.
Frequency Range	`0 to Fs/2`	Frequency range of the plot x-axis.
Frequency Units	`Hz`	Units for specifying the frequency range.
Sampling Frequency	`48000`	Inverse of the sampling period.
Frequency Scale	`dB`	Units used for the y-axis display of the output.
Normalized Frequency	`Off`	Use normalized frequency for the display.

After your first analysis run ends, open the **Analysis Parameters** dialog box and adjust your settings appropriately, such as changing the number of trials or number of points.

To open the **Analysis Parameters** dialog box, use either of the next procedures when you have a quantized filter in filter designer:

- Select **Analysis > Analysis Parameters** from the menu bar
- Right-click in the filter analysis area and select **Analysis Parameters** from the context menu

Whichever option you choose opens the dialog box. Notice that the settings for the options reflect the defaults.

Noise Method Applied to a Filter

To demonstrate the magnitude response estimate method, start by creating a quantized filter. For this example, use filter designer to design a sixth-order Butterworth IIR filter.

To Use Noise-Based Analysis in Filter Designer

1. Enter `filterDesigner` at the MATLAB prompt to launch filter designer.
2. Under **Response Type**, select **Highpass**.
3. Select `IIR` in **Design Method**. Then select `Butterworth`.
4. To set the filter order to 6, select **Specify order** under **Filter Order**. Enter `6` in the text box.
5. Click **Design Filter**.

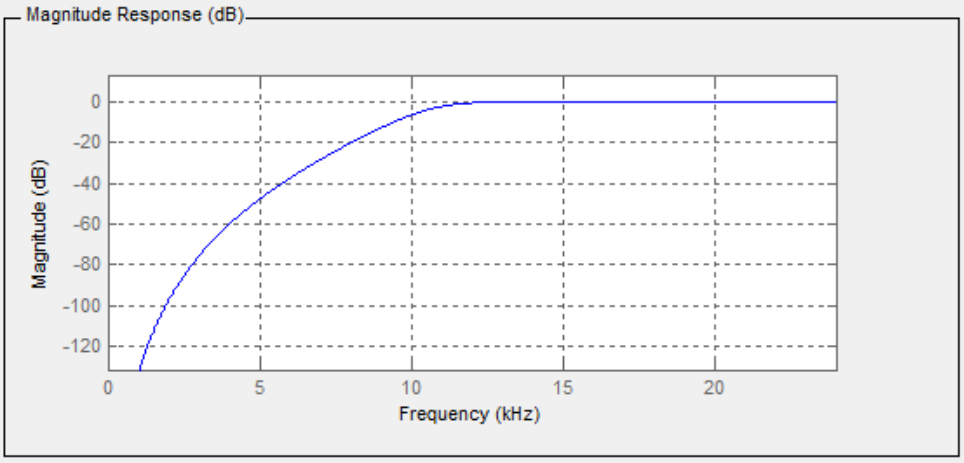
   In filter designer, the analysis area changes to display the magnitude response for your filter.
6. To generate the quantized version of your filter, using default quantizer settings, click on the side bar.

   Filter designer switches to quantization mode and displays the quantization panel.
7. From **Filter arithmetic**, select `fixed-point`.

   Now the analysis areas shows the magnitude response for both filters — your original filter and the fixed-point arithmetic version.

8 Finally, to use noise-based estimation on your quantized filter, select **Analysis > Magnitude Response Estimate** from the menu bar.

Filter designer runs the trial, calculates the estimated magnitude response for the filter, and displays the result in the analysis area as shown in this figure.



In the above figure you see the magnitude response as estimated by the analysis method.

View the Noise Power Spectrum

When you use the noise method to estimate the magnitude response of a filter, filter designer simulates and applies a spectrum of noise values to test your filter response. While the simulated noise is essentially white, you might want to see the actual spectrum that filter designer used to test your filter.

From the **Analysis** menu bar option, select **Round-off Noise Power Spectrum**. In the analysis area in filter designer, you see the spectrum of the noise used to estimate the filter response. The details of the noise spectrum, such as the range and number of data points, appear in the **Analysis Parameters** dialog box.

For more information, refer to McClellan, et al., Computer-Based Exercises for Signal Processing Using MATLAB 5, Prentice-Hall, 1998. See Project 5: Quantization Noise in Digital Filters, page 231.

Change Your Noise Analysis Parameters

In “Noise Method Applied to a Filter” on page 4-19, you used synthetic white noise to estimate the magnitude response for a fixed-point highpass Butterworth filter. Since you ran the estimate only once in filter designer, your noise analysis used the default analysis parameters settings shown in “Analyze Filters with the Magnitude Response Estimate Method” on page 4-18.

To change the settings, follow these steps after the first time you use the noise estimate on your quantized filter.

1 With the results from running the noise estimating method displayed in the filter designer analysis area, select **Analysis > Analysis Parameters** from the menu bar.

To give you access to the analysis parameters, the **Analysis Parameters** dialog box opens (with default settings).

2 To use more points in the spectrum to estimate the magnitude response, change **Number of Points** to `1024` and click **OK** to run the analysis.

Filter designer closes the **Analysis Parameters** dialog box and reruns the noise estimate, returning the results in the analysis area.

To rerun the test without closing the dialog box, press **Enter** after you type your new value into a setting, then click **Apply**. Now filter designer runs the test without closing the dialog box. When you want to try many different settings for the noise-based analysis, this is a useful shortcut.

Compare the Estimated and Theoretical Magnitude Responses

An important measure of the effectiveness of the noise method for estimating the magnitude response of a quantized filter is to compare the estimated response to the theoretical response.

One way to do this comparison is to overlay the theoretical response on the estimated response. While you have the Magnitude Response Estimate displaying in filter designer, select **Analysis** > **Overlay Analysis** from the menu bar. Then select **Magnitude Response** to show both response curves plotted together in the analysis area.

Select Quantized Filter Structures

Filter designer lets you change the structure of any quantized filter. Use the **Convert structure** option to change the structure of your filter to one that meets your needs.

To learn about changing the structure of a filter in filter designer, refer to “Converting the Filter Structure” on page 22-14.

Convert the Structure of a Quantized Filter

You use the **Convert structure** option to change the structure of filter. When the **Source** is **Designed(Quantized)** or **Imported(Quantized)**, **Convert structure** lets you recast the filter to one of the following structures:

- “Direct Form II Transposed Filter Structure”
- “Direct Form I Transposed Filter Structure”
- “Direct Form II Filter Structure”
- “Direct Form I Filter Structure”
- “Direct Form Finite Impulse Response (FIR) Filter Structure”
- “Direct Form FIR Transposed Filter Structure”
- “Lattice Autoregressive Moving Average (ARMA) Filter Structure”
- “Direct Form Antisymmetric FIR Filter Structure (Any Order)”

Starting from any quantized filter, you can convert to one of the following representation:

- Direct form I
- Direct form II
- Direct form I transposed
- Direct form II transposed
- Lattice ARMA

Additionally, filter designer lets you do the following conversions:

- Minimum phase FIR filter to Lattice MA minimum phase
- Maximum phase FIR filter to Lattice MA maximum phase
- Allpass filters to Lattice allpass

Refer to “FilterStructure” for details about each of these structures.

Convert Filters to Second-Order Sections Form

To learn about using filter designer to convert your quantized filter to use second-order sections, refer to “Converting to Second-Order Sections” on page 22-15. You might notice that filters you design in filter designer, rather than filters you imported, are implemented in SOS form.

View Filter Structures in Filter Designer

To open the demonstration, click **Help > Show filter structures**. After the Help browser opens, you see the reference page for the current filter. You find the filter structure signal flow diagram on this reference page, or you can navigate to reference pages for other filter.

Scale Second-Order Section Filters

- “Use the Reordering and Scaling Second-Order Sections Dialog Box” on page 4-22
- “Scale an SOS Filter” on page 4-23

Use the Reordering and Scaling Second-Order Sections Dialog Box

Filter designer provides the ability to scale SOS filters after you create them. Using options on the Reordering and Scaling Second-Order Sections dialog box, filter designer scales either or both the filter numerators and filter scale values according to your choices for the scaling options.

Parameter	Description and Valid Value
Scale	Apply any scaling options to the filter. Select this when you are reordering your SOS filter and you want to scale it at the same time. Or when you are scaling your filter, with or without reordering. Scaling is disabled by default.
No Overflow — High SNR slider	Lets you set whether scaling favors reducing arithmetic overflow in the filter or maximizing the signal-to-noise ratio (SNR) at the filter output. Moving the slider to the right increases the emphasis on SNR at the expense of possible overflows. The markings indicate the P-norm applied to achieve the desired result in SNR or overflow protection. For more information about the P-norm settings, refer to `norm` for details.
Maximum Numerator	Maximum allowed value for numerator coefficients after scaling.

Parameter	Description and Valid Value
Numerator Constraint	Specifies whether and how to constrain numerator coefficient values. Options are `none`, `normalize`, `power of 2`, and `unit`. Choosing `none` lets the scaling use any scale value for the numerators by removing any constraints on the numerators, except that the coefficients will be clipped if they exceed the **Maximum Numerator**. With `Normalize` the maximum absolute value of the numerator is forced to equal the **Maximum Numerator** value (for all other constraints, the **Maximum Numerator** is only an upper limit, above which coefficients will be clipped). The `power of 2` option forces scaling to use numerator values that are powers of 2, such as 2 or 0.5. With `unit`, the leading coefficient of each numerator is forced to a value of 1.
Overflow Mode	Sets the way the filter handles arithmetic overflow situations during scaling. Choose from either `saturate` (limit the output to the largest positive or negative representable value) or `wrap` (set overflowing values to the nearest representable value using modular arithmetic.
Scale Value Constraint	Specify whether to constrain the filter scale values, and how to constrain them. Valid options are `unit`, `power of 2`, and `none`. Choosing `unit` for the constraint disables the **Max. Scale Value** setting and forces scale values to equal 1. `Power of 2` constrains the scale values to be powers of 2, such as 2 or 0.5, while `none` removes any constraint on the scale values, except that they cannot exceed the **Max. Scale Value**.
Max. Scale Value	Sets the maximum allowed scale values. SOS filter scaling applies the **Max. Scale Value** limit only when you set **Scale Value Constraint** to a value other than `unit` (the default setting). Setting a maximum scale value removes any other limits on the scale values.
Revert to Original Filter	Returns your filter to the original scaling. Being able to revert to your original filter makes it easier to assess the results of scaling your filter.

Various combinations of settings let you scale filter numerators without changing the scale values, or adjust the filter scale values without changing the numerators. There is no scaling control for denominators.

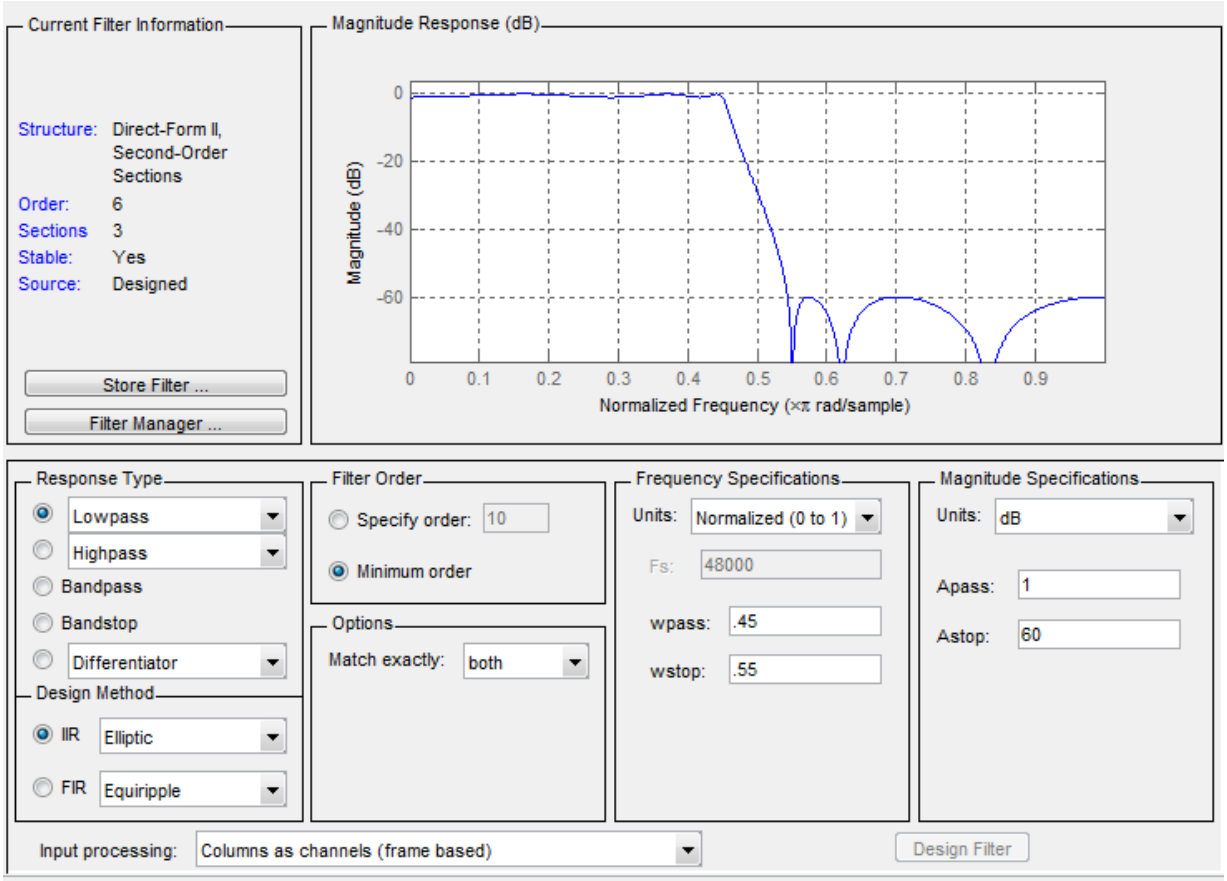
Scale an SOS Filter

Start the process by designing a lowpass elliptical filter in filter designer.

1. Launch filter designer.
2. In **Response Type**, select **Lowpass**.
3. In Design Method, select **IIR** and `Elliptic` from the IIR design methods list.
4. Select **Minimum Order** for the filter.
5. Switch the frequency units by choosing `Normalized(0 to 1)` from the **Units** list.
6. To set the passband specifications, enter `0.45` for **wpass** and `0.55` for **wstop**. Finally, in **Magnitude Specification**s, set **Astop** to `60`.

7 Click **Design Filter** to design the filter.

After filter designer finishes designing the filter, you see the following plot and settings in the tool.



You kept the **Options** setting for **Match exactly** as `both`, meaning the filter design matches the specification for the passband and the stopband.

8 To switch to scaling the filter, select **Edit > Reorder and Scale Second-Order Sections** from the menu bar.

9 To see the filter coefficients, return to filter designer and select **Filter Coefficients** from the **Analysis** menu. Filter designer displays the coefficients and scale values in filter designer.

With the coefficients displayed you can see the effects of scaling your filter directly in the scale values and filter coefficients.

Now try scaling the filter in a few different ways. First scale the filter to maximize the SNR.

1 Return to the **Reordering and Scaling Second-Order Sections** dialog box and select **None** for **Reordering** in the left pane. This prevents filter designer from reordering the filter sections when you rescale the filter.

2 Move the **No Overflow—High SNR** slider from **No Overflow** to **High SNR**.

3 Click **Apply** to scale the filter and leave the dialog box open.

After a few moments, filter designer updates the coefficients displayed so you see the new scaling.

All of the scale factors are now 1, and the SOS matrix of coefficients shows that none of the numerator coefficients are 1 and the first denominator coefficient of each section is 1.

4 Click **Revert to Original Filter** to restore the filter to the original settings for scaling and coefficients.

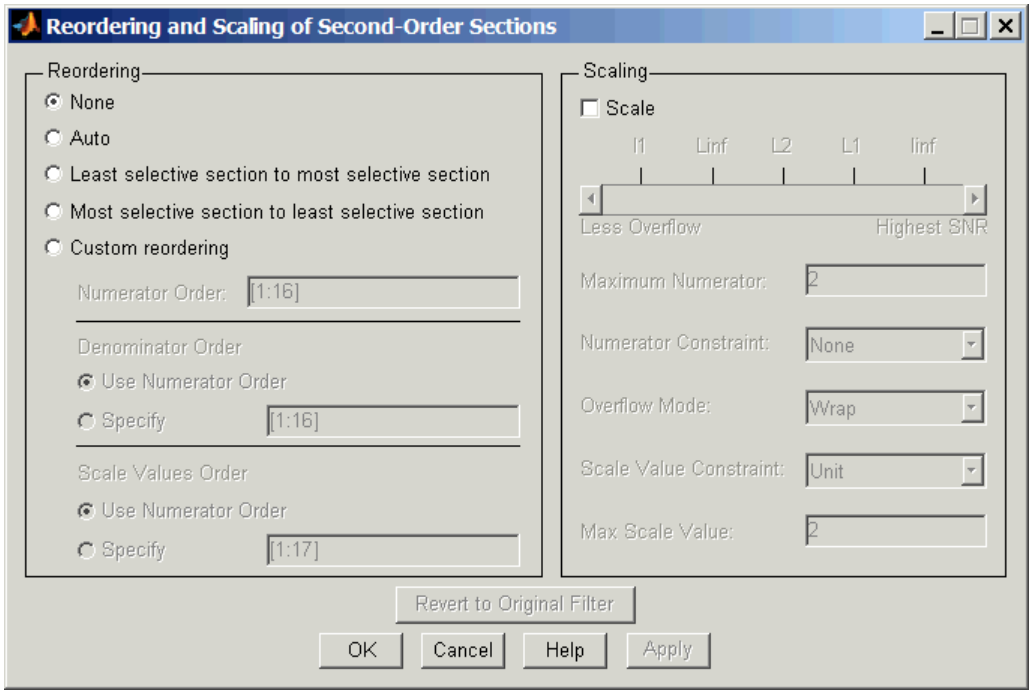
Reorder the Sections of Second-Order Section Filters

Reorder Filters Using Filter Designer

Filter Designer designs most discrete-time filters in second-order sections. Generally, SOS filters resist the effects of quantization changes when you create fixed-point filters. After you have a second-order section filter in filter designer, either one you designed in the tool, or one you imported, filter designer provides the capability to change the order of the sections that compose the filter. Any SOS filter in filter designer allows reordering of the sections.

To reorder the sections of a filter, you access the Reorder and Scaling of Second-Order Sections dialog box in filter designer.

With your SOS filter in filter designer, select **Edit > Reorder and Scale** from the menu bar. filter designer returns the reordering dialog box shown here with the default settings.



Controls on the Reordering and Scaling of Second-Order Sections dialog box

In this dialog box, the left-hand side contains options for reordering SOS filters. On the right you see the scaling options. These are independent — reordering your filter does not require scaling (note the **Scale** option) and scaling does not require that you reorder your filter (note the **None** option under

Reordering). For more about scaling SOS filters, refer to “Scale Second-Order Section Filters” on page 4-22 and to `scale` in the reference section.

Reordering SOS filters involves using the options in the **Reordering and Scaling of Second-Order Sections** dialog box. The following table lists each reorder option and provides a description of what the option does.

Control Option	Description
Auto	Reorders the filter sections to minimize the output noise power of the filter. Note that different ordering applies to each specification type, such as lowpass or highpass. Automatic ordering adapts to the specification type of your filter.
None	Does no reordering on your filter. Selecting **None** lets you scale your filter without applying reordering at the same time. When you access this dialog box with a current filter, this is the default setting — no reordering is applied.
Least selective section to most selective section	Rearranges the filter sections so the least restrictive (lowest Q) section is the first section and the most restrictive (highest Q) section is the last section.
Most selective section to least selective section	Rearranges the filter sections so the most restrictive (highest Q) section is the first section and the least restrictive (lowest Q) section is the last section.
Custom reordering	Lets you specify the section ordering to use by enabling the Numerator Order and Denominator Order options
Numerator Order	Specify new ordering for the sections of your SOS filter. Enter a vector of the indices of the sections in the order in which to rearrange them. For example, a filter with five sections has indices 1, 2, 3, 4, and 5. To switch the second and fourth sections, the vector would be [1,4,3,2,5].
Use Numerator Order	Rearranges the denominators in the order assigned to the numerators.
Specify	Lets you specify the order of the denominators, rather than using the numerator order. Enter a vector of the indices of the sections to specify the order of the denominators to use. For example, a filter with five sections has indices 1, 2, 3, 4, and 5. To switch the second and fourth sections, the vector would be [1,4,3,2,5].
Use Numerator Order	Reorders the scale values according to the order of the numerators.
Specify	Lets you specify the order of the scale values, rather than using the numerator order. Enter a vector of the indices of the sections to specify the order of the denominators to use. For example, a filter with five sections has indices 1, 2, 3, 4, and 5. To switch the second and fourth sections, the vector would be [1,4,3,2,5].
Revert to Original Filter	Returns your filter to the original section ordering. Being able to revert to your original filter makes comparing the results of changing the order of the sections easier to assess.

Reorder an SOS Filter

With filter designer open a second-order filter as the current filter, you use the following process to access the reordering capability and reorder you filter. Start by launching filter designer from the command prompt.

1. Enter `filterDesigner` at the command prompt to launch filter designer.
2. Design a lowpass Butterworth filter with order 10 and the default frequency specifications by entering the following settings:
   - Under **Response Type** select `Lowpass`.
   - Under **Design Method**, select **IIR** and `Butterworth` from the list.
   - Specify the order equal to 10 in **Specify order** under **Filter Order**.
   - Keep the default **Fs** and **Fc** values in **Frequency Specifications**.
3. Click **Design Filter**.

   Filter designer designs the Butterworth filter and returns your filter as a Direct-Form II filter implemented with second-order sections. You see the specifications in the **Current Filter Information** area.

   With the second-order filter in filter designer, reordering the filter uses the **Reordering and Scaling of Second-Order Sections** feature in filter designer (also available in Filter Visualization Tool, `FVTool`).
4. To reorder your filter, select **Edit > Reorder and Scale Second-Order Sections** from the filter designer menus.

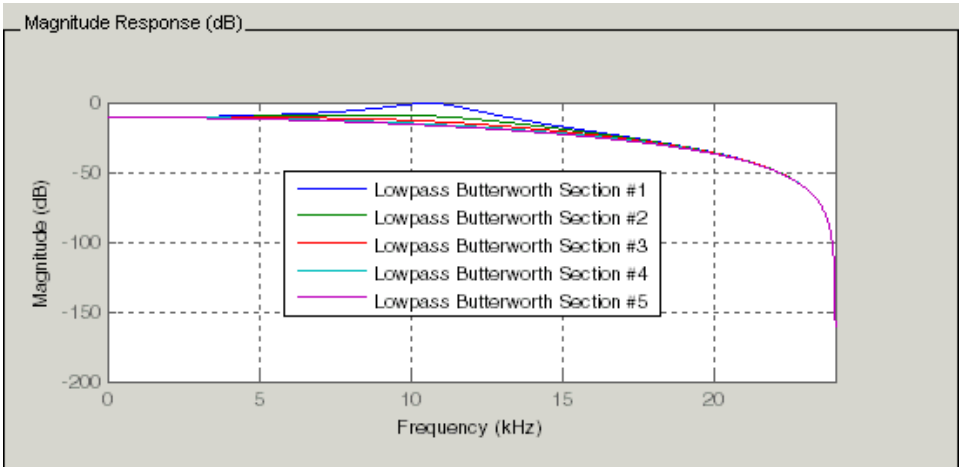
Now you are ready to reorder the sections of your filter. Note that filter designer performs the reordering on the current filter in the session.

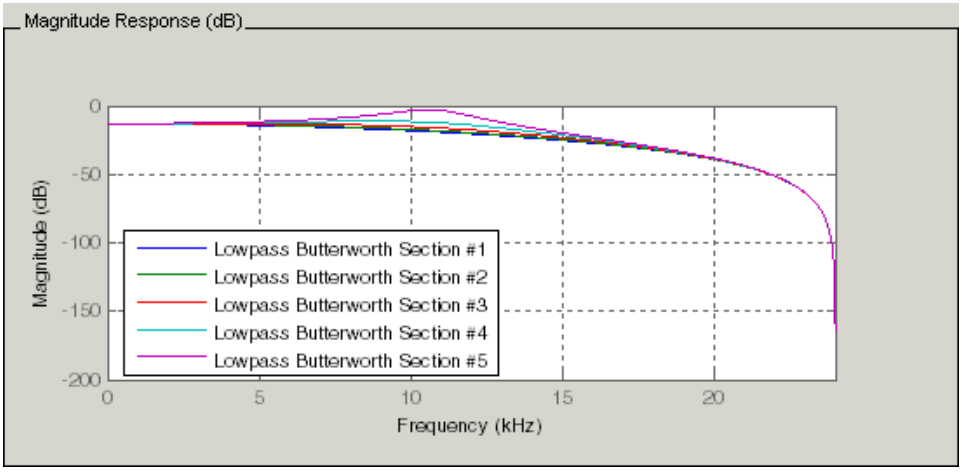
Use Least Selective to Most Selective Section Reordering

To let filter designer reorder your filter so the least selective section is first and the most selective section is last, perform the following steps in the **Reordering and Scaling of Second-Order Sections** dialog box.

1. In **Reordering**, select **Least selective section to most selective section**.
2. To prevent filter scaling at the same time, clear **Scale** in **Scaling**.
3. In filter designer, select **View > SOS View Settings** from the menu bar so you see the sections of your filter displayed in filter designer.
4. In the **SOS View Settings** dialog box, select **Individual sections**. Making this choice configures filter designer to show the magnitude response curves for each section of your filter in the analysis area.
5. Back in the **Reordering and Scaling of Second-Order Sections** dialog box, click **Apply** to reorder your filter according to the Qs of the filter sections, and keep the dialog box open. In response, filter designer presents the responses for each filter section (there should be five sections) in the analysis area.

   In the next two figures you can compare the ordering of the sections of your filter. In the first figure, your original filter sections appear. In the second figure, the sections have been rearranged from least selective to most selective.





You see what reordering does, although the result is a bit subtle. Now try custom reordering the sections of your filter or using the most selective to least selective reordering option.

View SOS Filter Sections

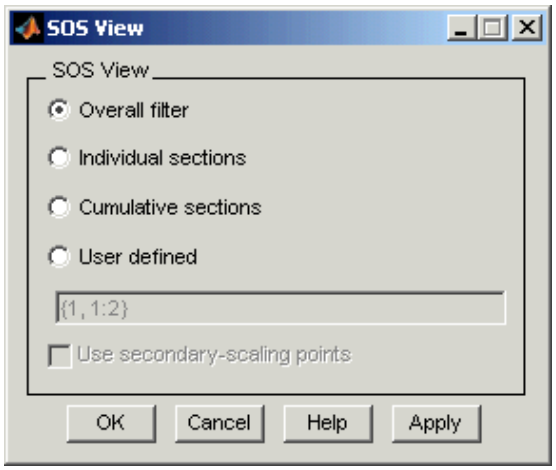
- “Using the SOS View Dialog Box” on page 4-28
- “View the Sections of SOS Filters” on page 4-30

Using the SOS View Dialog Box

Since you can design and reorder the sections of SOS filters, filter designer provides the ability to view the filter sections in the analysis area — SOS View. Once you have a second-order section filter as your current filter in filter designer, you turn on the SOS View option to see the filter sections individually, or cumulatively, or even only some of the sections. Enabling SOS View puts filter designer in a mode where all second-order section filters display sections until you disable the SOS View option. SOS View mode applies to any analysis you display in the analysis area. For example, if you configure filter designer to show the phase responses for filters, enabling SOS View means filter designer displays the phase response for each section of SOS filters.

Controls on the SOS View Dialog Box

SOS View uses a few options to control how filter designer displays the sections, or which sections to display. When you select **View > SOS View** from the filter designer menu bar, you see this dialog box containing options to configure SOS View operation.



By default, SOS View shows the overall response of SOS filters. Options in the SOS View dialog box let you change the display. This table lists all the options and describes the effects of each.

Option	Description
Overall Filter	This is the familiar display in filter designer. For a second-order section filter you see only the overall response rather than the responses for the individual sections. This is the default configuration.
Individual sections	When you select this option, filter designer displays the response for each section as a curve. If your filter has five sections you see five response curves, one for each section, and they are independent. Compare to **Cumulative sections**.
Cumulative sections	When you select this option, filter designer displays the response for each section as the accumulated response of all prior sections in the filter. If your filter has five sections you see five response curves: • The first curve plots the response for the first filter section. • The second curve plots the response for the combined first and second sections. • The third curve plots the response for the first, second, and third sections combined. And so on until all filter sections appear in the display. The final curve represents the overall filter response. Compare to **Cumulative sections** and **Overall Filter**.

Option	Description
User defined	Here you define which sections to display, and in which order. Selecting this option enables the text box where you enter a cell array of the indices of the filter sections. Each index represents one section. Entering one index plots one response. Entering something like {1:2} plots the combined response of sections 1 and 2. If you have a filter with four sections, the entry {1:4} plots the combined response for all four sections, whereas {1,2,3,4} plots the response for each section. Note that after you enter the cell array, you need to click **OK** or **Apply** to update the filter designer analysis area to the new SOS View configuration.
Use secondary-scaling points	This directs filter designer to use the secondary scaling points in the sections to determine where to split the sections. This option applies only when the filter is a `df2sos` or `df1tsos` filter. For these structures, the secondary scaling points refer to the scaling locations between the recursive and the nonrecursive parts of the section (the "middle" of the section). By default, secondary-scaling points is not enabled. You use this with the **Cumulative sections** option only.

View the Sections of SOS Filters

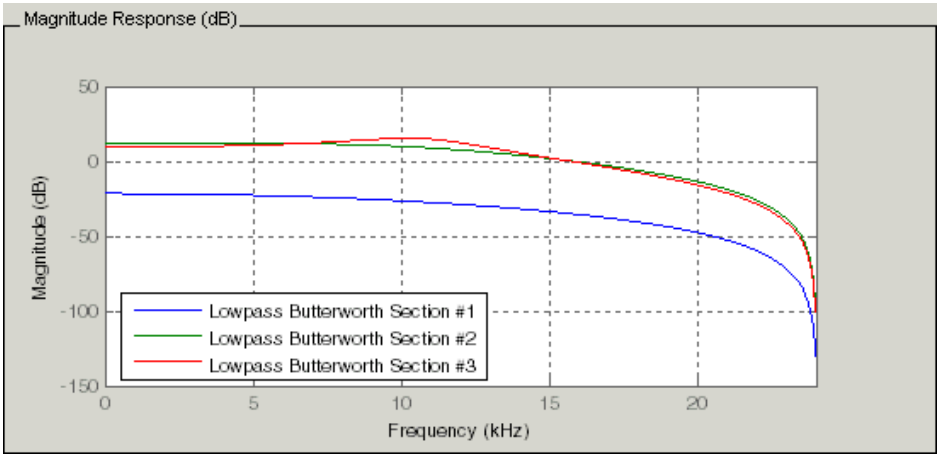
After you design or import an SOS filter in to filter designer, the SOS view option lets you see the per section performance of your filter. Enabling SOS View from the View menu in filter designer configures the tool to display the sections of SOS filters whenever the current filter is an SOS filter.

These next steps demonstrate using SOS View to see your filter sections displayed in filter designer.

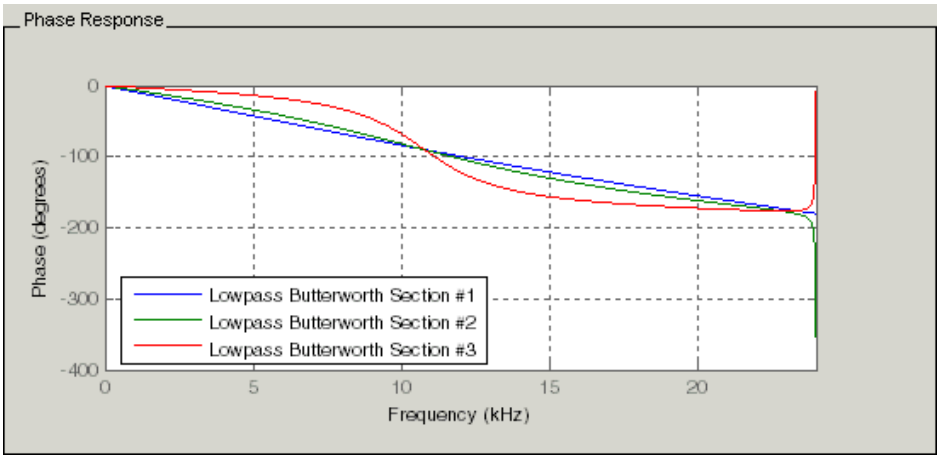
1. Launch filter designer.
2. Create a lowpass SOS filter using the Butterworth design method. Specify the filter order to be 6. Using a low order filter makes seeing the sections more clear.
3. Design your new filter by clicking **Design Filter**.

   filter designer design your filter and show you the magnitude response in the analysis area. In Current Filter Information you see the specifications for your filter. You should have a sixth-order Direct-Form II, Second-Order Sections filter with three sections.
4. To enable SOS View, select **View > SOS View** from the menu bar.

   By default the analysis area in filter designer shows the overall filter response, not the individual filter section responses. This dialog box lets you change the display configuration to see the sections.
5. To see the magnitude responses for each filter section, select **Individual sections**.
6. Click **Apply** to update filter designer to display the responses for each filter section. The analysis area changes to show you something like the following figure.



If you switch filter designer to display filter phase responses (by selecting **Analysis > Phase Response**), you see the phase response for each filter section in the analysis area.

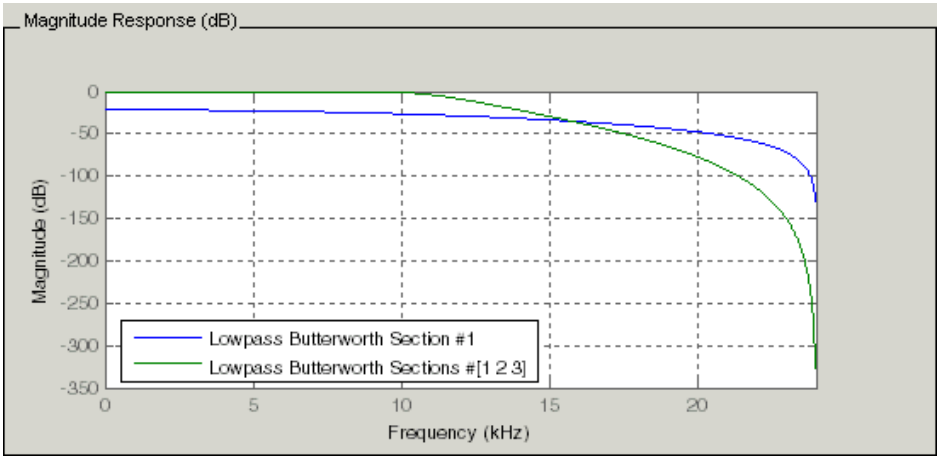


7 To define your own display of the sections, you use the **User defined** option and enter a vector of section indices to display. Now you see a display of the first section response, and the cumulative first, second, and third sections response:

- Select **User defined** to enable the text entry box in the dialog box.
- Enter the cell array `{1,1:3}` to specify that filter designer should display the response of the first section and the cumulative response of the first three sections of the filter.

8 To apply your new SOS View selection, click **Apply** or **OK** (which closes the **SOS View** dialog box).

In the filter designer analysis area you see two curves — one for the response of the first filter section and one for the combined response of sections 1, 2, and 3.



Import and Export Quantized Filters

- “Overview and Structures” on page 4-32
- “Import Quantized Filters” on page 4-33
- “To Export Quantized Filters” on page 4-34

Overview and Structures

When you import a quantized filter into filter designer, or export a quantized filter from filter designer to your workspace, the import and export functions use objects and you specify the filter as a variable. This contrasts with importing and exporting nonquantized filters, where you select the filter structure and enter the filter numerator and denominator for the filter transfer function.

You have the option of exporting quantized filters to your MATLAB workspace, exporting them to text files, or exporting them to MAT-files.

For general information about importing and exporting filters in filter designer, refer to “Importing a Filter Design” on page 22-24, and “Exporting a Filter Design” on page 22-16.

Filter designer imports quantized filters having the following structures:

- Direct form I
- Direct form II
- Direct form I transposed
- Direct form II transposed
- Direct form symmetric FIR
- Direct form antisymmetric FIR
- Lattice allpass
- Lattice AR
- Lattice MA minimum phase
- Lattice MA maximum phase
- Lattice ARMA
- Lattice coupled-allpass

- Lattice coupled-allpass power complementary

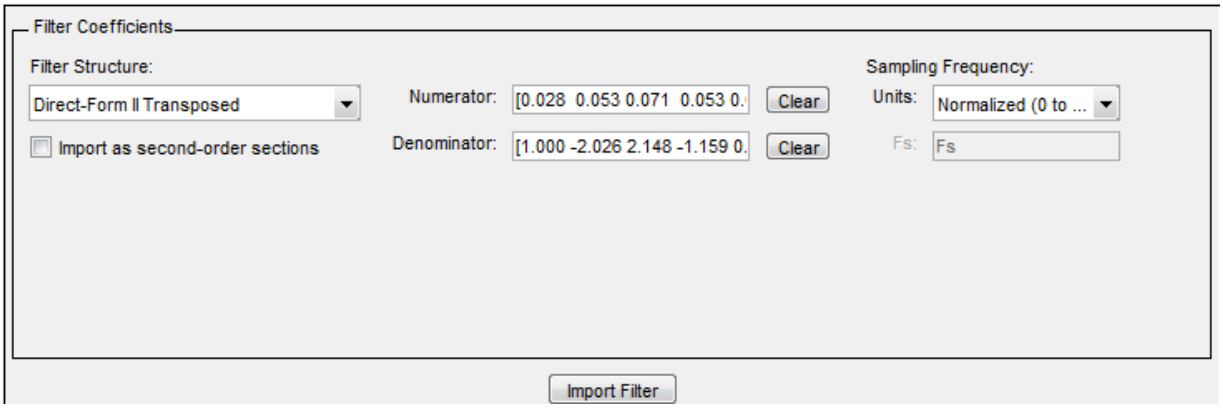
Import Quantized Filters

After you design or open a quantized filter in your MATLAB workspace, filter designer lets you import the filter for analysis. Follow these steps to import your filter in to filter designer:

1. Open filter designer.
2. Select **File > Import Filter from Workspace** from the menu bar, or choose the **Import Filter from Workspace** icon in the side panel:

   .

   In the lower region of filter designer, the **Design Filter** pane becomes **Import Filter**, and options appear for importing quantized filters, as shown.



3. From the **Filter Structure** list, select `Filter object`.

   The options for importing filters change to include:

   - **Discrete filter** — Enter the variable name for the discrete-time, fixed-point filter in your workspace.
   - **Frequency units** — Select the frequency units from the **Units** list under **Sampling Frequency**, and specify the sampling frequency value in **Fs** if needed. Your sampling frequency must correspond to the units you select. For example, when you select `Normalized (0 to 1)`, **Fs** defaults to one. But if you choose one of the frequency options, enter the sampling frequency in your selected units. If you have the sampling frequency defined in your workspace as a variable, enter the variable name for the sampling frequency.
4. Click **Import** to import the filter.

   Filter designer checks your workspace for the specified filter. It imports the filter if it finds it, displaying the magnitude response for the filter in the analysis area. If it cannot find the filter it returns an **Filter Designer Error** dialog box.

Note If, during any filter designer session, you switch to quantization mode and create a fixed-point filter, filter designer remains in quantization mode. If you import a double-precision filter, filter

designer automatically quantizes your imported filter applying the most recent quantization parameters.
When you check the current filter information for your imported filter, it will indicate that the filter is **Source:** `imported (quantized)` even though you did not import a quantized filter.

To Export Quantized Filters

To save your filter design, filter designer lets you export the quantized filter to your MATLAB workspace (or you can save the current session in filter designer). When you choose to save the quantized filter by exporting it, you select one of these options:

- Export to your MATLAB workspace on page 4-34
- Export to a text file on page 4-34
- Export to a MAT-file on page 4-35

Export Coefficients, Objects, or System Objects to the Workspace

You can save the filter as filter coefficients variables or filter System object variables.

To save the filter to the MATLAB workspace:

1. Select **Export** from the **File** menu. The **Export** dialog box appears.
2. Select `Workspace` from the **Export To** list.
3. From the **Export As** list, select one of the following options:

   - Select `Coefficients` to save the filter coefficients.
   - Select `System Objects` to save the filter in a filter System object.

   The `System Objects` option does not appear in the drop-down list when the current filter structure is not supported by System objects.
4. Assign a variable name:

   - For coefficients, assign variable names using the **Numerator** and **Denominator** options under **Variable Names**.
   - For System objects, assign the variable name in the **Discrete Filter** option.

   If you have variables with the same names in your workspace and you want to overwrite them, select the **Overwrite Variables** box.
5. Click **Export**.

   Do not try to export the filter to a variable name that exists in your workspace without selecting **Overwrite existing variables**, in the previous step. If you do so, filter designer stops the export operation. The tool returns a warning that the variable you specified as the quantized filter name already exists in the workspace.

   - To continue to export the filter to the existing variable, click **OK** to dismiss the warning.
   - Then select the **Overwrite existing variables** check box and click **Export**.

Export Filter Coefficients as a Text File

To save your quantized filter as a text file, follow these steps:

1. Select **Export** from the **File** menu.
2. Select `Text-file` under **Export to**.
3. Click **OK** to export the filter and close the dialog box. Click **Apply** to export the filter without closing the **Export** dialog box. Clicking **Apply** lets you export your quantized filter to more than one name without leaving the **Export** dialog box.

   The **Export Filter Coefficients to Text-file** dialog box appears. This is the standard Microsoft Windows® save file dialog box.
4. Choose or enter a folder and filename for the text file, and click **OK**.

   Filter designer exports your quantized filter as a text file with the name you provided, and the MATLAB editor opens, displaying the file for editing.

Export Filter Coefficients as a MAT-File

To save your quantized filter as a MAT-file, follow these steps:

1. Select **Export** from the **File** menu.
2. Select `MAT-file` under **Export to**.
3. Assign a variable name for the filter.
4. Click **OK** to export the filter and close the dialog box. Click **Apply** to export the filter without closing the **Export** dialog box. Clicking **Apply** lets you export your quantized filter to more than one name without leaving the **Export** dialog box.

   The **Export Filter Coefficients to MAT-file** dialog box appears. This dialog box is the standard Microsoft Windows save file dialog box.
5. Choose or enter a folder and filename for the text file, and click **OK**.

   Filter designer exports your quantized filter as a MAT-file with the specified name.

Generate MATLAB Code

You can generate MATLAB code using the **File > Generate MATLAB Code** menu. This menu has these options:

- **Filter Design Function (with System Objects)**

  This option generates a System object. The option is disabled when the current filter is not supported by system objects.
- **Data Filtering Function (with System Objects)**

  This option generates MATLAB code that filters input data with the current filter design. The MATLAB code is ready to be converted to C/C++ code using the `codegen` command. This option is disabled when the current filter is not supported by system objects.

Import XILINX Coefficient (.COE) Files

Import XILINX .COE Files into Filter Designer

You can import XILINX coefficients (`.coe`) files into filter designer to create quantized filters directly using the imported filter coefficients.

To use the import file feature:

1. Select **File > Import Filter From XILINX Coefficient (.COE) File** in filter designer.
2. In the **Import Filter From XILINX Coefficient (.COE) File** dialog box, find and select the `.coe` file to import.
3. Click **Open** to dismiss the dialog box and start the import process.

   Filter designer imports the coefficient file and creates a quantized, single-section, direct-form FIR filter.

Transform Filters Using Filter Designer

- "Filter Transformation Capabilities of Filter Designer" on page 4-36
- "Original Filter Type" on page 4-37
- "Frequency Point to Transform" on page 4-39
- "Transformed Filter Type" on page 4-40
- "Specify Desired Frequency Location" on page 4-40

Filter Transformation Capabilities of Filter Designer

The toolbox provides functions for transforming filters between various forms. When you use filter designer with the toolbox installed, a side bar button and a menu bar option enable you to use the **Transform Filter** panel to transform filters as well as using the command line functions.

From the selection on the filter designer menu bar — **Transformations** — you can transform lowpass FIR and IIR filters to a variety of passband shapes.

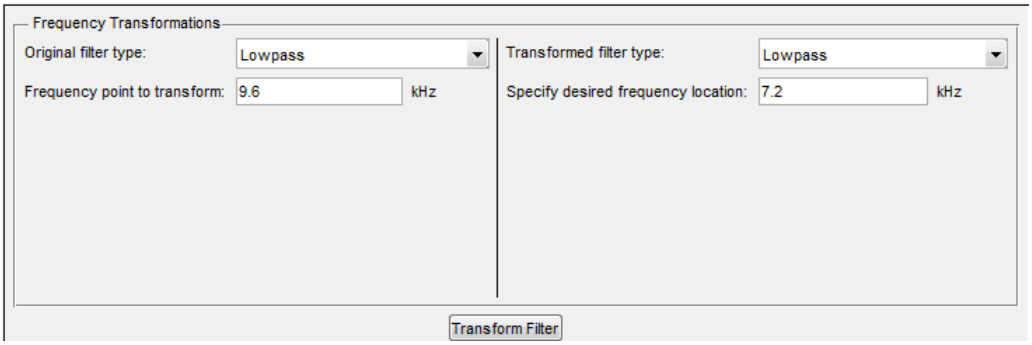
You can convert your FIR filters from:

- Lowpass to lowpass.
- Lowpass to highpass.

For IIR filters, you can convert from:

- Lowpass to lowpass.
- Lowpass to highpass.
- Lowpass to bandpass.
- Lowpass to bandstop.

When you click the **Transform Filter** button, , on the side bar, the **Transform Filter** panel opens in filter designer, as shown here.



Your options for **Original filter type** refer to the type of your current filter to transform. If you select lowpass, you can transform your lowpass filter to another lowpass filter or to a highpass filter, or to numerous other filter formats, real and complex.

Note When your original filter is an FIR filter, both the FIR and IIR transformed filter type options appear on the **Transformed filter type** list. Both options remain active because you can apply the IIR transforms to an FIR filter. If your source is as IIR filter, only the IIR transformed filter options show on the list.

Original Filter Type

Select the magnitude response of the filter you are transforming from the list. Your selection changes the types of filters you can transform to. For example:

- When you select **Lowpass** with an IIR filter, your transformed filter type can be
  - **Lowpass**
  - **Highpass**
  - **Bandpass**
  - **Bandstop**
  - **Multiband**
  - **Bandpass (complex)**
  - **Bandstop (complex)**
  - **Multiband (complex)**
- When you select **Lowpass** with an FIR filter, your transformed filter type can be
  - **Lowpass**
  - **Lowpass (FIR)**
  - **Highpass**
  - **Highpass (FIR) narrowband**
  - **Highpass (FIR) wideband**
  - **Bandpass**
  - **Bandstop**
  - **Multiband**

- **Bandpass (complex)**
- **Bandstop (complex)**
- **Multiband (complex)**

In the following table you see each available original filter type and all the types of filter to which you can transform your original.

Original Filter	Available Transformed Filter Types
Lowpass FIR	• **Lowpass** • **Lowpass (FIR)** • **Highpass** • **Highpass (FIR) narrowband** • **Highpass (FIR) wideband** • **Bandpass** • **Bandstop** • **Multiband** • **Bandpass (complex)** • **Bandstop (complex)** • **Multiband (complex)**
Lowpass IIR	• **Lowpass** • **Highpass** • **Bandpass** • **Bandstop** • **Multiband** • **Bandpass (complex)** • **Bandstop (complex)** • **Multiband (complex)**
Highpass FIR	• **Lowpass** • **Lowpass (FIR) narrowband** • **Lowpass (FIR) wideband** • **Highpass (FIR)** • **Highpass** • **Bandpass** • **Bandstop** • **Multiband** • **Bandpass (complex)** • **Bandstop (complex)** • **Multiband (complex)**

Original Filter	Available Transformed Filter Types
Highpass IIR	• **Lowpass** • **Highpass** • **Bandpass** • **Bandstop** • **Multiband** • **Bandpass (complex)** • **Bandstop (complex)** • **Multiband (complex)**
Bandpass FIR	• **Bandpass** • **Bandpass (FIR)**
Bandpass IIR	**Bandpass**
Bandstop FIR	• **Bandstop** • **Bandstop (FIR)**
Bandstop IIR	**Bandstop**

Note also that the transform options change depending on whether your original filter is FIR or IIR. Starting from an FIR filter, you can transform to IIR or FIR forms. With an IIR original filter, you are limited to IIR target filters.

After selecting your response type, use **Frequency point to transform** to specify the magnitude response point in your original filter to transfer to your target filter. Your target filter inherits the performance features of your original filter, such as passband ripple, while changing to the new response form.

For more information about transforming filters, refer to “Frequency Transformations for Real Filters” on page 4-65 and “Frequency Transformations for Complex Filters” on page 4-74.

Frequency Point to Transform

The frequency point you enter in this field identifies a magnitude response value (in dB) on the magnitude response curve.

When you enter frequency values in the **Specify desired frequency location** option, the frequency transformation tries to set the magnitude response of the transformed filter to the value identified by the frequency point you enter in this field.

While you can enter any location, generally you should specify a filter passband or stopband edge, or a value in the passband or stopband.

The **Frequency point to transform** sets the magnitude response at the values you enter in **Specify desired frequency location**. Specify a value that lies at either the edge of the stopband or the edge of the passband.

If, for example, you are creating a bandpass filter from a highpass filter, the transformation algorithm sets the magnitude response of the transformed filter at the **Specify desired frequency location** to be the same as the response at the **Frequency point to transform** value. Thus you get a bandpass filter whose response at the low and high frequency locations is the same. Notice that the passband

between them is undefined. In the next two figures you see the original highpass filter and the transformed bandpass filter.

For more information about transforming filters, refer to “Digital Frequency Transformations” on page 4-60.

Transformed Filter Type

Select the magnitude response for the target filter from the list. The complete list of transformed filter types is:

- **Lowpass**
- **Lowpass (FIR)**
- **Highpass**
- **Highpass (FIR) narrowband**
- **Highpass (FIR) wideband**
- **Bandpass**
- **Bandstop**
- **Multiband**
- **Bandpass (complex)**
- **Bandstop (complex)**
- **Multiband (complex)**

Not all types of transformed filters are available for all filter types on the **Original filter types** list. You can transform bandpass filters only to bandpass filters. Or bandstop filters to bandstop filters. Or IIR filters to IIR filters.

For more information about transforming filters, refer to “Frequency Transformations for Real Filters” on page 4-65 and “Frequency Transformations for Complex Filters” on page 4-74.

Specify Desired Frequency Location

The frequency point you enter in **Frequency point to transform** matched a magnitude response value. At each frequency you enter here, the transformation tries to make the magnitude response the same as the response identified by your **Frequency point to transform** value.

While you can enter any location, generally you should specify a filter passband or stopband edge, or a value in the passband or stopband.

For more information about transforming filters, refer to “Digital Frequency Transformations” on page 4-60.

Transform Filters

To transform the magnitude response of your filter, use the **Transform Filter** option on the side bar.

1. Design or import your filter into filter designer.
2. Click **Transform Filter**, , on the side bar.

   Filter designer opens the **Transform Filter** panel in filter designer.

3 From the **Original filter type** list, select the response form of the filter you are transforming.

When you select the type, whether is **lowpass**, **highpass**, **bandpass**, or **bandstop**, filter designer recognizes whether your filter form is FIR or IIR. Using both your filter type selection and the filter form, filter designer adjusts the entries on the **Transformed filter type** list to show only those that apply to your original filter.

4 Enter the frequency point to transform value in **Frequency point to transform**. Notice that the value you enter must be in kHz; for example, enter 0.1 for 100 Hz or 1.5 for 1500 Hz.

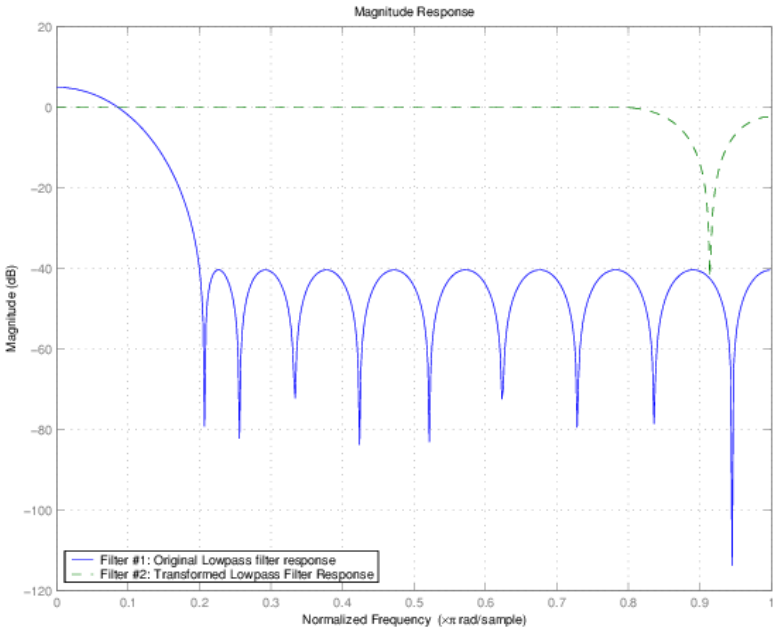
5 From the **Transformed filter type** list, select the type of filter you want to transform to.

Your filter type selection changes the options here.

- When you pick a lowpass or highpass filter type, you enter one value in **Specify desired frequency location**.
- When you pick a bandpass or bandstop filter type, you enter two values — one in **Specify desired low frequency location** and one in **Specify desired high frequency location**. Your values define the edges of the passband or stopband.
- When you pick a multiband filter type, you enter values as elements in a vector in **Specify a vector of desired frequency locations** — one element for each desired location. Your values define the edges of the passbands and stopbands.

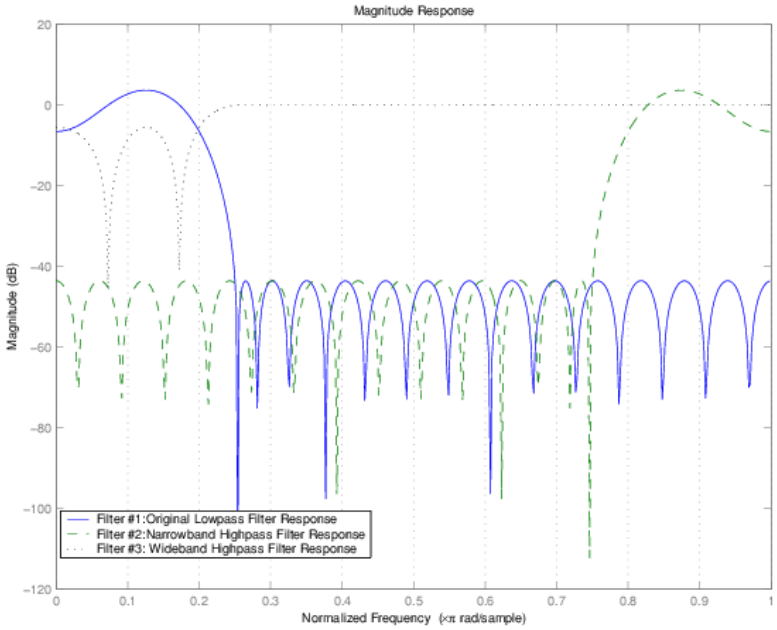
After you click **Transform Filter**, filter designer transforms your filter, displays the magnitude response of your new filter, and updates the **Current Filter Information** to show you that your filter has been transformed. In the filter information, the **Source** is **Transformed**.

For example, the figure shown here includes the magnitude response curves for two filters. The original filter is a lowpass filter with rolloff between 0.2 and 0.25. The transformed filter is a lowpass filter with rolloff region between 0.8 and 0.85.



- To demonstrate the effects of selecting **Narrowband Highpass** or **Wideband Highpass**, the next figure presents the magnitude response curves for a source lowpass filter after it is

transformed to both narrow- and wideband highpass filters. For comparison, the response of the original filter appears as well.



For the narrowband case, the transformation algorithm essentially reverses the magnitude response, like reflecting the curve around the *y*-axis, then translating the curve to the right until the origin lies at 1 on the *x*-axis. After reflecting and translating, the passband at high frequencies is the reverse of the passband of the original filter at low frequencies with the same rolloff and ripple characteristics.

Design Multirate Filters in Filter Designer

- “Introduction” on page 4-42
- “Switch Filter Designer to Multirate Filter Design Mode” on page 4-42
- “Controls on the Multirate Design Panel” on page 4-43
- “Quantize Multirate Filters” on page 4-48
- “Export Individual Phase Coefficients of a Polyphase Filter to the Workspace” on page 4-50

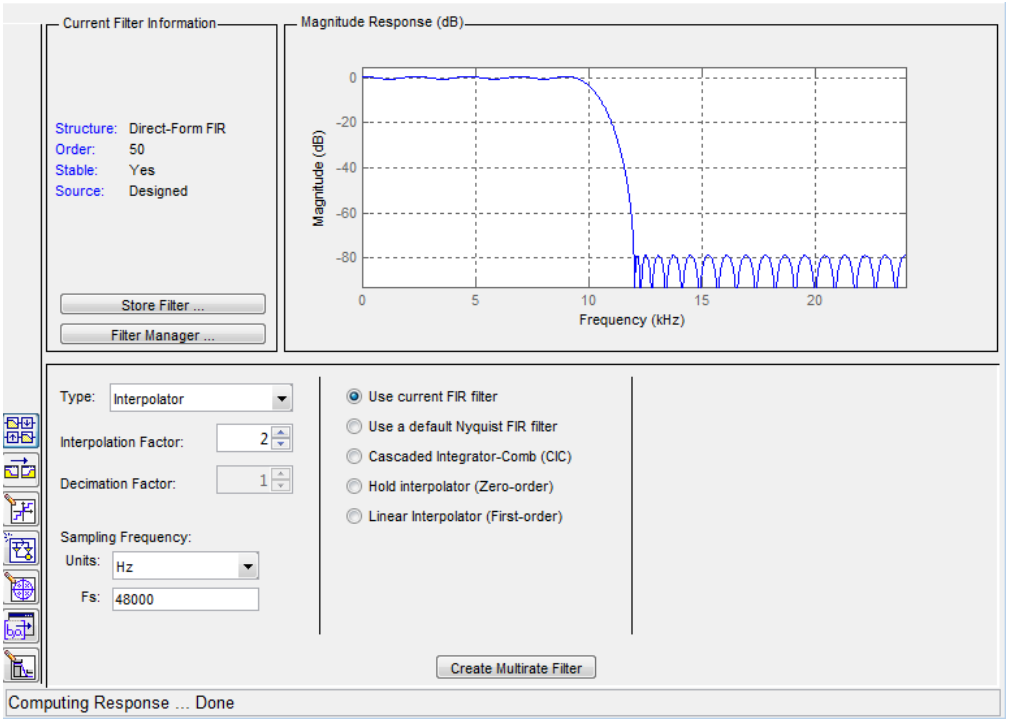
Introduction

Not only can you design multirate filters from the MATLAB command prompt, filter designer provides the same design capability in a graphical user interface tool. By starting filter designer and switching to the multirate filter design mode you have access to all of the multirate design capabilities in the toolbox — decimators, interpolators, and fractional rate changing filters, among others.

Switch Filter Designer to Multirate Filter Design Mode

The multirate filter design mode in filter designer lets you specify and design a wide range of multirate filters, including decimators and interpolators.

With filter designer open, click **Create a Multirate Filter**, , on the side bar. You see filter designer switch to the design mode showing the multirate filter design options. Shown in the following figure is the default multirate design configuration that designs an interpolating filter with an interpolation factor of 2. The design uses the current FIR filter in filter designer.



When the current filter in filter designer is not an FIR filter, the multirate filter design panel removes the **Use current FIR filter** option and selects the **Use default Nyquist FIR filter** option instead as the default setting.

Controls on the Multirate Design Panel

You see the options that allow you to design a variety of multirate filters. The Type option is your starting point. From this list you select the multirate filter to design. Based on your selection, other options change to provide the controls you need to specify your filter.

Notice the separate sections of the design panel. On the left is the filter type area where you choose the type of multirate filter to design and set the filter performance specifications.

In the center section filter designer provides choices that let you pick the filter design method to use.

The rightmost section offers options that control filter configuration when you select **Cascaded-Integrator Comb (CIC)** as the design method in the center section. Both the `Decimator` type and `Interpolator` type filters let you use the **Cascaded-Integrator Comb (CIC)** option to design multirate filters.

Here are all the options available when you switch to multirate filter design mode. Each option listed includes a brief description of what the option does when you use it.

Select and Configure Your Filter

Option	Description
Type	Specifies the type of multirate filter to design. Choose from `Decimator`, `Interpolator`, or `Fractional-rate convertor`. • When you choose `Decimator`, set **Decimation Factor** to specify the decimation to apply. • When you choose `Interpolator`, set **Interpolation Factor** to specify the interpolation amount applied. • When you choose `Fractional-rate convertor`, set both **Interpolation Factor** and **Decimation Factor**. Filter designer uses both to determine the fractional rate change by dividing **Interpolation Factor** by **Decimation Factor** to determine the fractional rate change in the signal. You should select values for interpolation and decimation that are relatively prime. When your interpolation factor and decimation factor are not relatively prime, filter designer reduces the interpolation/decimation fractional rate to the lowest common denominator and issues a message in the status bar in filter designer. For example, if the interpolation factor is 6 and the decimation factor is 3, filter designer reduces 6/3 to 2/1 when you design the rate changer. But if the interpolation factor is 8 and the decimation factor is 3, filter designer designs the filter without change.
Interpolation Factor	Use the up-down control arrows to specify the amount of interpolation to apply to the signal. Factors range upwards from 2.
Decimation Factor	Use the up-down control arrows to specify the amount of decimation to apply to the signal. Factors range upwards from 2.
Sampling Frequency	No settings here. Just **Units** and **Fs** below.
Units	Specify whether **Fs** is specified in `Hz`, `kHz`, `MHz`, `GHz`, or `Normalized (0 to 1)` units.
Fs	Set the full scale sampling frequency in the frequency units you specified in **Units**. When you select `Normalized` for **Units**, you do not enter a value for **Fs**.

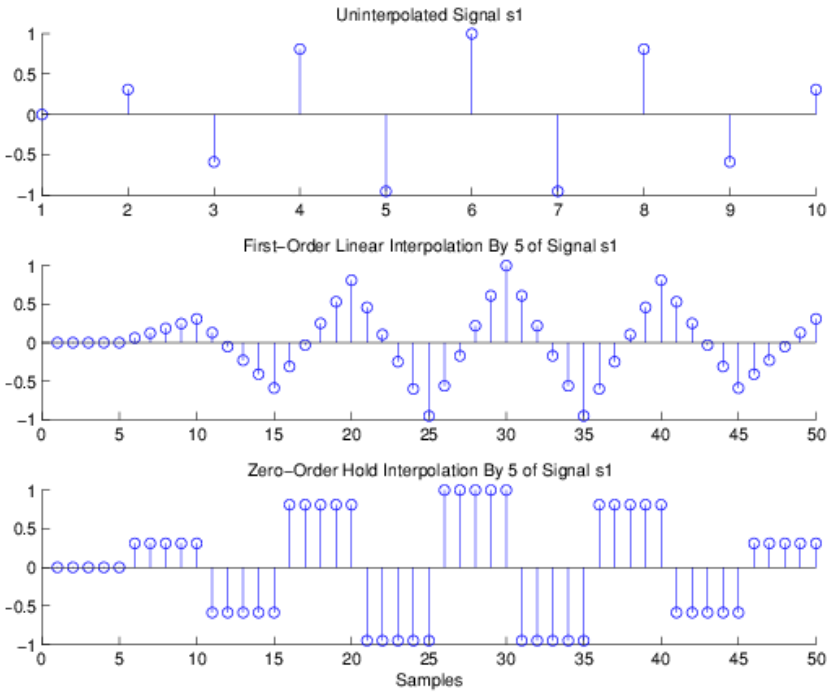
Design Your Filter

Option	Description
Use current FIR filter	Directs filter designer to use the current FIR filter to design the multirate filter. If the current filter is an IIR form, you cannot select this option. You cannot design multirate filters with IIR structures.
Use a default Nyquist Filter	Tells filter designer to use the default Nyquist design method when the current filter in filter designer is not an FIR filter.
Cascaded Integrator-Comb (CIC)	Design CIC filters using the options provided in the right-hand area of the multirate design panel.
Hold Interpolator (Zero-order)	When you design an interpolator, you can specify how the filter sets interpolated values between signal values. When you select this option, the interpolator applies the most recent signal value for each interpolated value until it processes the next signal value. This is similar to sample-and-hold techniques. Compare to the **Linear Interpolator** option.
Linear Interpolator (First-order)	When you design an interpolator, you can specify how the filter sets interpolated values between signal values. When you select this option, the interpolator applies linear interpolation between signal value to set the interpolated value until it processes the next signal value. Compare to the **Linear Interpolator** option.

To see the difference between hold interpolation and linear interpolation, the following figure presents a sine wave signal s1 in three forms:

- The top subplot in the figure presents signal s1 without interpolation.
- The middle subplot shows signal s1 interpolated by a linear interpolator with an interpolation factor of 5.
- The bottom subplot shows signal s1 interpolated by a hold interpolator with an interpolation factor of 5.

You see in the bottom figure the sample and hold nature of hold interpolation, and the first-order linear interpolation applied by the linear interpolator.



Options for Designing CIC Filters	Description
Differential Delay	Sets the differential delay for the CIC filter. Usually a value of one or two is appropriate.
Number of Sections	Specifies the number of sections in a CIC decimator. The default number of sections is 2 and the range is any positive integer.

Design a Fractional Rate Convertor

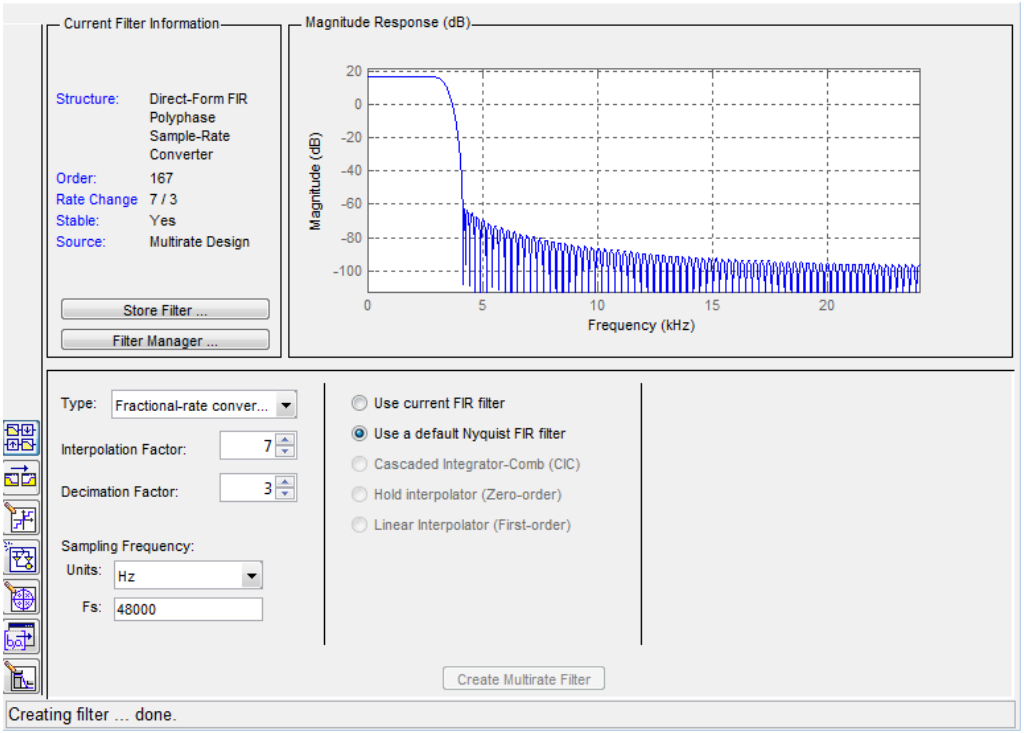
To introduce the process you use to design a multirate filter in filter designer, this example uses the options to design a fractional rate convertor which uses 7/3 as the fractional rate. Begin the design by creating a default lowpass FIR filter in filter designer. You do not have to begin with this FIR filter, but the default filter works fine.

1. Launch filter designer.
2. Select the settings for a minimum-order lowpass FIR filter, using the `Equiripple` design method.
3. When filter designer displays the magnitude response for the filter, click in the side bar. filter designer switches to multirate filter design mode, showing the multirate design panel.
4. To design a fractional rate filter, select `Fractional-rate convertor` from the **Type** list. The **Interpolation Factor** and **Decimation Factor** options become available.
5. In **Interpolation Factor**, use the up arrow to set the interpolation factor to 7.
6. Using the up arrow in **Decimation Factor**, set 3 as the decimation factor.
7. Select `Use a default Nyquist FIR filter`. You could design the rate convertor with the current FIR filter as well.

8 Enter 24000 to set **Fs**.

9 Click **Create Multirate Filter**.

After designing the filter, filter designer returns with the specifications for your new filter displayed in **Current Filter Information**, and shows the magnitude response of the filter.



You can test the filter by exporting it to your workspace and using it to filter a signal. For information about exporting filters, refer to “Import and Export Quantized Filters” on page 4-32.

Design a CIC Decimator for 8 Bit Input/Output Data

Another kind of filter you can design in filter designer is Cascaded-Integrator Comb (CIC) filters. Filter designer provides the options needed to configure your CIC to meet your needs.

1 Launch filter designer and design the default FIR lowpass filter. Designing a filter at this time is an optional step.

2 Switch filter designer to multirate design mode by clicking on the side bar.

3 For **Type**, select `Decimator`, and set **Decimation Factor** to 3.

4 To design the decimator using a CIC implementation, select **Cascaded-Integrator Comb (CIC)**. This enables the CIC-related options on the right of the panel.

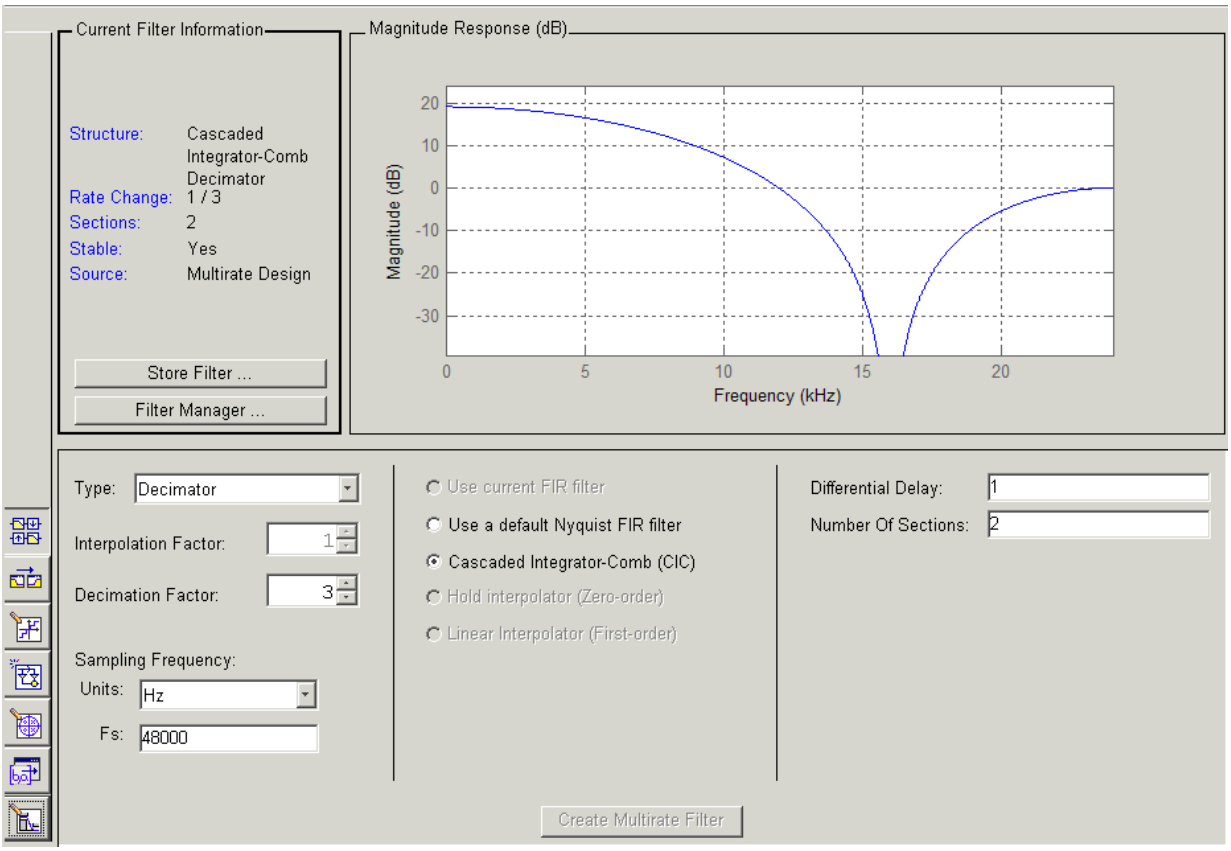
5 Set Differential Delay to 2. Generally, 1 or 2 are good values to use.

6 Enter 2 for the **Number of Sections**.

7 Click **Create Multirate Filter**.

Filter Designer designs the filter, shows the magnitude response in the analysis area, and updates the current filter information to show that you designed a tenth-order cascaded-integrator comb decimator with two sections. Notice the source is Multirate Design, indicating

you used the multirate design mode in filter designer to make the filter. Filter Designer should look like this now.



Designing other multirate filters follows the same pattern.

To design other multirate filters, do one of the following depending on the filter to design:

- To design an interpolator, select one of these options.
  - **Use a default Nyquist FIR filter**
  - **Cascaded-Integrator Comb (CIC)**
  - **Hold Interpolator (Zero-order)**
  - **Linear Interpolator (First-order)**
- To design a decimator, select from these options.
  - **Use a default Nyquist FIR filter**
  - **Cascaded-Integrator Comb (CIC)**
- To design a fractional-rate convertor, select **Use a default Nyquist FIR filter**.

Quantize Multirate Filters

After you design a multirate filter in filter designer, the quantization features enable you to convert your floating-point multirate filter to fixed-point arithmetic.

Note CIC filters are always fixed-point.

With your multirate filter as the current filter in filter designer, you can quantize your filter and use the quantization options to specify the fixed-point arithmetic the filter uses.

Quantize and Configure Multirate Filters

Follow these steps to convert your multirate filter to fixed-point arithmetic and set the fixed-point options.

1. Design or import your multirate filter and make sure it is the current filter in filter designer.
2. Click the **Set Quantization Parameters** button on the side bar.
3. From the **Filter Arithmetic** list on the Filter Arithmetic pane, select `Fixed-point`. If your filter is a CIC filter, the `Fixed-point` option is enabled by default and you do not set this option.
4. In the quantization panes, set the options for your filter. Set options for **Coefficients**, **Input/Output**, and **Filter Internals**.
5. Click **Apply**.

When you current filter is a CIC filter, the options on the **Input/Output** and **Filter Internals** panes change to provide specific features for CIC filters.

Input/Output

The options that specify how your CIC filter uses input and output values are listed in the table below.

Option Name	Description
Input Word Length	Sets the word length used to represent the input to a filter.
Input fraction length	Sets the fraction length used to interpret input values to filter.
Input range (+/-)	Lets you set the range the inputs represent. You use this instead of the **Input fraction length** option to set the precision. When you enter a value x, the resulting range is -x to x. Range must be a positive integer.
Output word length	Sets the word length used to represent the output from a filter.
Avoid overflow	Directs the filter to set the fraction length for the input to prevent the output values from exceeding the available range as defined by the word length. Clearing this option lets you set **Output fraction length**.
Output fraction length	Sets the fraction length used to represent output values from a filter.
Output range (+/-)	Lets you set the range the outputs represent. You use this instead of the **Output fraction length** option to set the precision. When you enter a value x, the resulting range is -x to x. Range must be a positive integer.

The available options change when you change the **Filter precision** setting. Moving from `Full` to `Specify all` adds increasing control by enabling more input and output word options.

Filter Internals

With a CIC filter as your current filter, the **Filter precision** option on the **Filter Internals** pane includes modes for controlling the filter word and fraction lengths.

There are four usage modes for this (the same mode you select for the `FilterInternals` property in CIC filters at the MATLAB prompt).

- `Full` — All word and fraction lengths set to B_{max} + 1, called B_{accum}. This is the default.
- `Minimum section word lengths` — Set the section word lengths to minimum values that meet roundoff noise and output requirements.
- `Specify word lengths` — Enables the **Section word length** option for you to enter word lengths for each section. Enter either a scalar to use the same value for every section, or a vector of values, one for each section.
- `Specify all` — Enables the **Section fraction length** option in addition to **Section word length**. Now you can provide both the word and fraction lengths for each section, again using either a scalar or a vector of values.

Export Individual Phase Coefficients of a Polyphase Filter to the Workspace

After designing a polyphase filter in the filter designer app, you can obtain the individual phase coefficients of the filter by:

1. Exporting the filter to an object in the MATLAB workspace.
2. Using the polyphase method to create a matrix of the filter's coefficients.

Export the Polyphase Filter to an Object

To export a polyphase filter to an object in the MATLAB workspace, complete the following steps.

1. In filter designer, open the **File** menu and select **Export...**. This opens the dialog box for exporting the filter coefficients.
2. In the Export dialog box, for **Export To**, select **Workspace**.
3. For **Export As**, select **Object**.
4. (Optional) For **Variable Names**, enter the name of the **Multirate Filter** object that will be created in the MATLAB workspace.
5. Click the **Export** button. The multirate filter object, `Hm` in this example, appears in the MATLAB workspace.

Create a Matrix of Coefficients Using the polyphase Method

To create a matrix of the filter's coefficients, enter `p=polyphase(Hm)` at the command line. The `polyphase` method creates a matrix, `p`, of filter coefficients from the filter object, `Hm`. Each row of `p` consists of the coefficients of an individual phase subfilter. The first row contains to the coefficients of the first phase subfilter, the second row contains those of the second phase subfilter, and so on.

Realize Filters as Simulink Subsystem Blocks

- “Introduction” on page 4-50
- “About the Realize Model Panel in Filter Designer” on page 4-51

Introduction

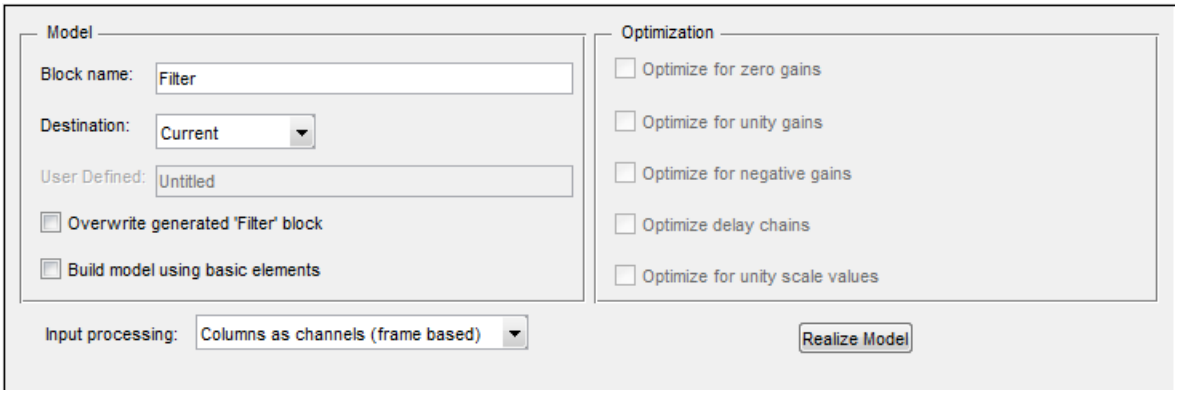
After you design or import a filter in filter designer, the realize model feature lets you create a Simulink subsystem block that implements your filter. The generated filter subsystem block uses either digital filter blocks from the DSP System Toolbox library, or the Delay, Gain, and Sum blocks in

Simulink. If you do not have a Fixed-Point Designer™ license, filter designer still realizes your model using blocks in fixed-point mode from Simulink, but you cannot run any model that includes your filter subsystem block in Simulink.

About the Realize Model Panel in Filter Designer

To access to the Realize Model panel and the options for realizing your quantized filter as a Simulink subsystem block, switch filter designer to realize model mode by clicking on the sidebar.

The following panel shows the options for configuring how filter designer implements your filter as a Simulink block.



For information on these parameters, see the descriptions on the Filter Realization Wizard block reference page.

Realize a Filter Using Filter Designer

After your quantized filter in filter designer is performing the way you want, with your desired phase and magnitude response, and with the right coefficients and form, follow these steps to realize your filter as a subsystem that you can use in a Simulink model.

1. Click **Realize Model** on the sidebar to change filter designer to realize model mode.
2. From the **Destination** list under **Model**, select either:
   - `Current model` — to add the realized filter subsystem to your current model
   - `New model` — to open a new Simulink model window and add your filter subsystem to the new window
3. Provide a name for your new filter subsystem in the **Name** field.
4. Decide whether to overwrite an existing block with this new one, and select or clear the **Overwrite generated 'Filter' block** check box.
5. Select the **Build model using basic elements** check box to implement your filter as a subsystem block that consists of Sum, Gain, and Delay blocks.
6. Select or clear the optimizations to apply.
   - **Optimize for zero gains** — removes zero gain blocks from the model realization
   - **Optimize for unity gains** — replaces unity gain blocks with direct connections to adjacent blocks

- **Optimize for negative gains** — replaces negative gain blocks by a change of sign at the nearest sum block
- **Optimize delay chains** — replaces cascaded delay blocks with a single delay block that produces the equivalent gain
- **Optimize for unity scale values** — removes all scale value multiplications by 1 from the filter structure

7 Click **Realize Model** to realize your quantized filter as a subsystem block according to the settings you selected.

If you double-click the filter block subsystem created by filter designer, you see the filter implementation in Simulink model form. Depending on the options you chose when you realized your filter, and the filter you started with, you might see one or more sections, or different architectures based on the form of your quantized filter. From this point on, the subsystem filter block acts like any other block that you use in Simulink models.

See Also

More About

- “Filter Builder Design Process” on page 23-2
- “Using Filter Designer” on page 22-2

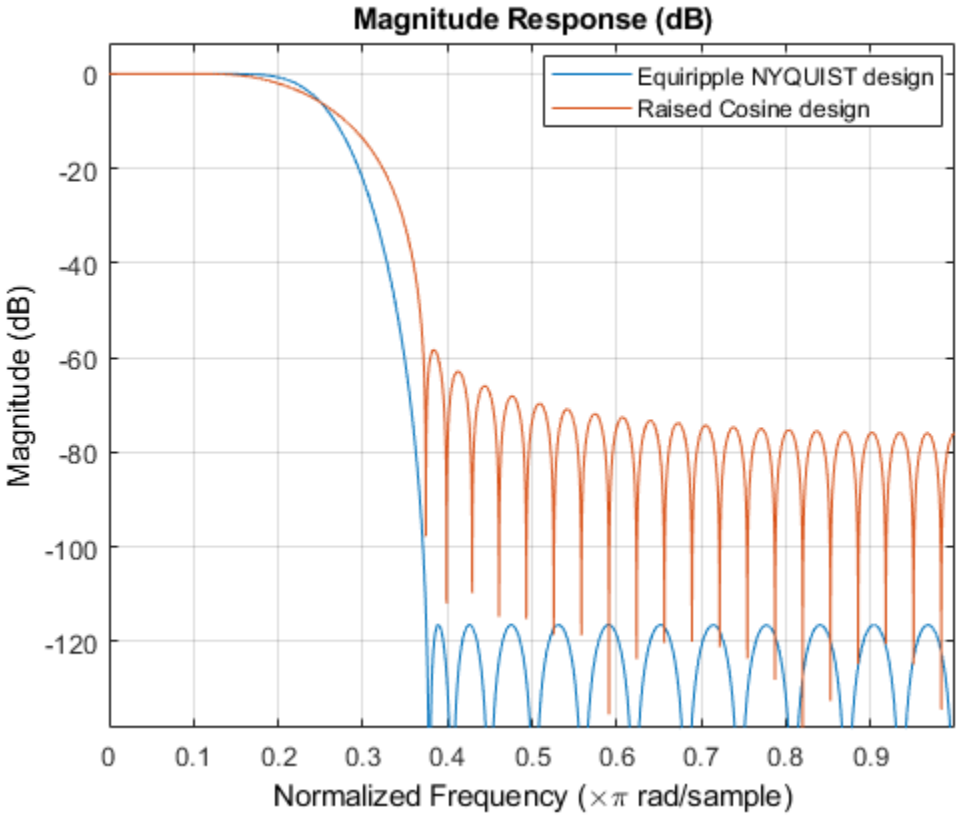
FIR Nyquist (L-th band) Filter Design

This example shows how to design lowpass FIR Nyquist filters. It also compares these filters with raised cosine and square root raised cosine filters. These filters are widely used in pulse-shaping for digital transmission systems. They also find application in interpolation/decimation and filter banks.

Magnitude Response Comparison

The plot shows the magnitude response of an equiripple Nyquist filter and a raised cosine filter. Both filters have an order of 60 and a rolloff-factor of 0.5. Because the equiripple filter has an optimal equiripple stopband, it has a larger stopband attenuation for the same filter order and transition width. The raised-cosine filter is obtained by truncating the analytical impulse response and it is not optimal in any sense.

```
NBand = 4;
N = 60;           % Filter order
R = 0.5;          % Rolloff factor
TW = R/(NBand/2); % Transition Bandwidth
f1 = fdesign.nyquist(NBand,'N,TW',N,TW);
eq = design(f1,'equiripple','Zerophase',true,'SystemObject',true);
coeffs = rcosdesign(R,N/NBand,NBand,'normal');
coeffs = coeffs/max(abs(coeffs))/NBand;
rc     = dsp.FIRFilter('Numerator',coeffs);
fvt = fvtool(eq,rc,'Color','white');
legend(fvt,'Equiripple NYQUIST design','Raised Cosine design');
```

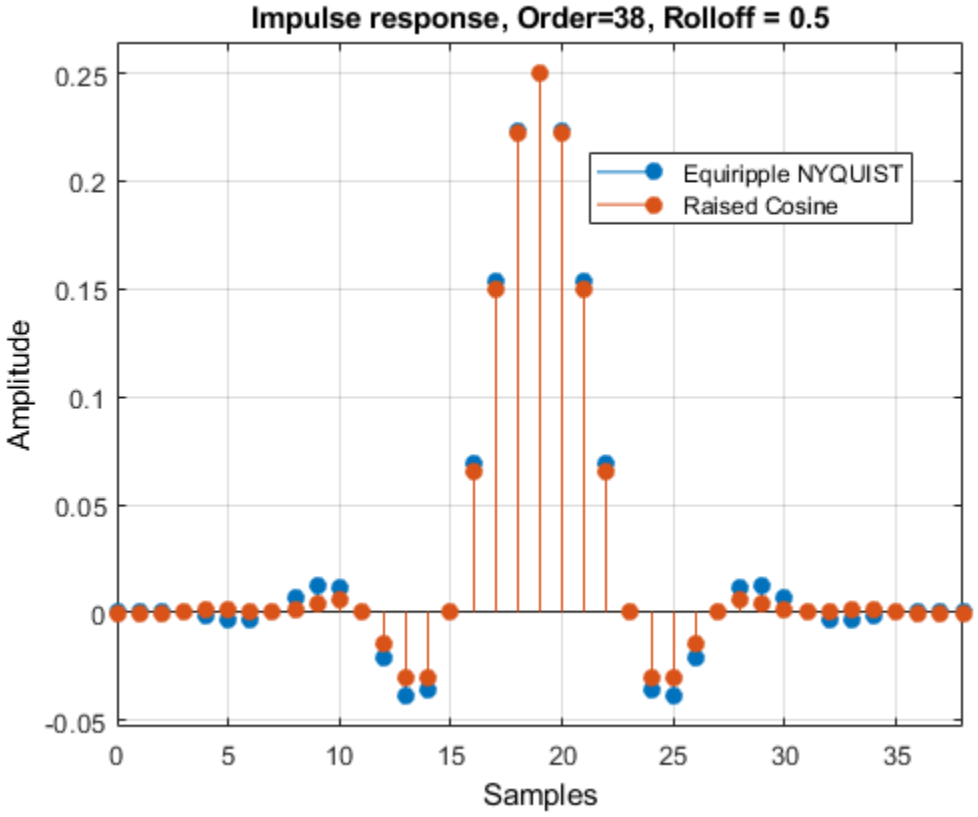


In fact, in this example it is necessary to increase the order of the raised-cosine design to about 1400 in order to attain similar attenuation.

Impulse Response Comparison

Here we compare the impulse responses. Notice that the impulse response in both cases is zero every 4th sample (except for the middle sample). Nyquist filters are also known as L-th band filters, because the cutoff frequency is Pi/L and the impulse response is zero every L-th sample. In this case we have 4th band filters.

```
f1.FilterOrder = 38;
eq1 = design(f1,'equiripple','Zerophase',true,'SystemObject',true);
coeffs = rcosdesign(R,f1.FilterOrder/NBand,NBand,'normal');
coeffs = coeffs/max(abs(coeffs))/NBand;
rc1 = dsp.FIRFilter('Numerator',coeffs);
fvt = fvtool(eq1,rc1,'Color','white','Analysis','Impulse');
legend(fvt,'Equiripple NYQUIST','Raised Cosine');
title('Impulse response, Order=38, Rolloff = 0.5');
```

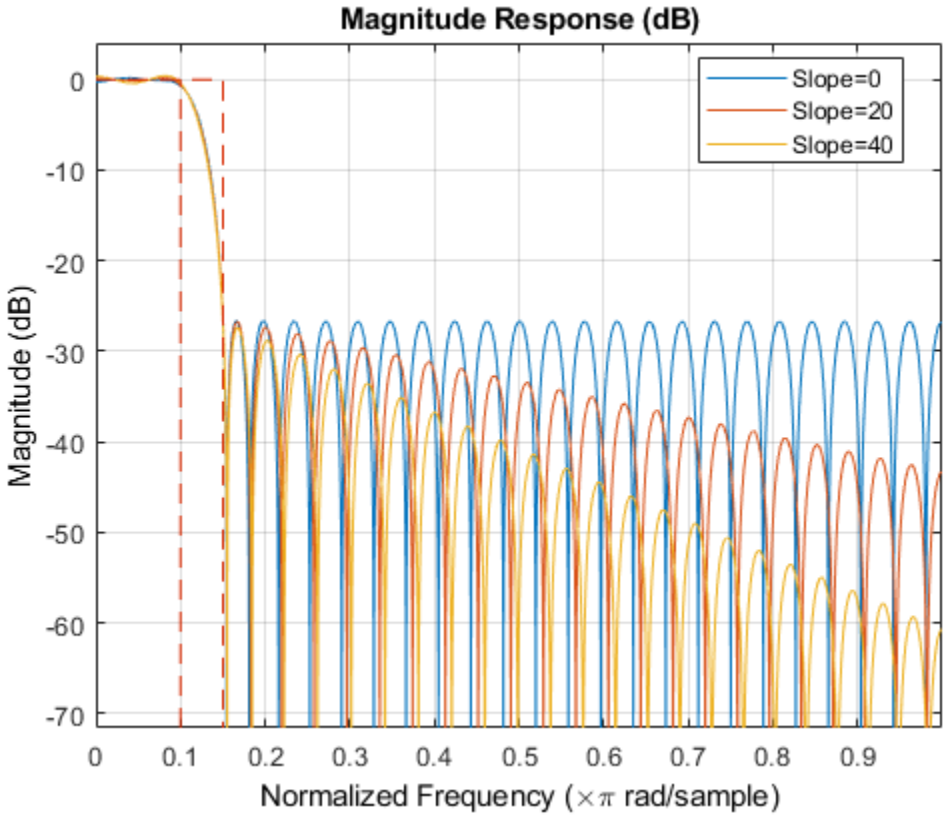


Nyquist Filters with a Sloped Stopband

Equiripple designs allow for control of the slope of the stopband of the filter. For example, the following designs have slopes of 0, 20, and 40 dB/(rad/sample)of attenuation:

```
f1.FilterOrder = 52;
f1.Band = 8;
f1.TransitionWidth = .05;
```

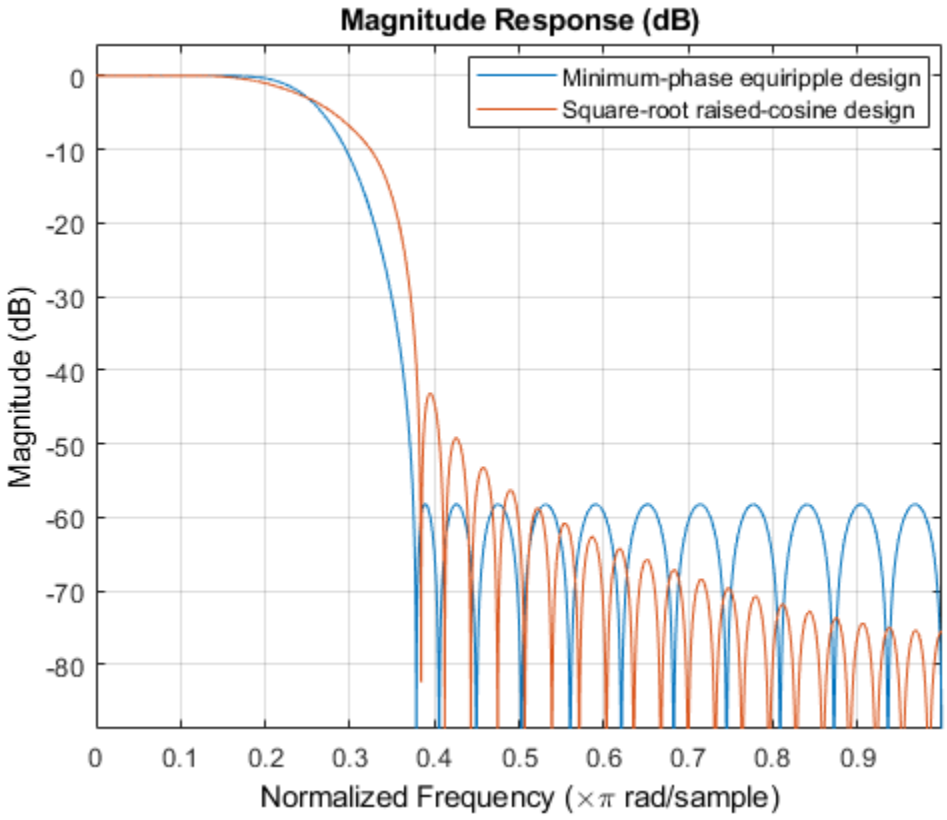
```
eq1 = design(f1,'equiripple','SystemObject',true);
eq2 = design(f1,'equiripple','StopbandShape','linear',...
    'StopbandDecay',20,'SystemObject',true);
eq3 = design(f1,'equiripple','StopbandShape','linear',...
    'StopbandDecay',40,'SystemObject',true);
fvt = fvtool(eq1,eq2,eq3,'Color','white');
legend(fvt,'Slope=0','Slope=20','Slope=40')
```



Minimum-Phase Design

We can design a minimum-phase spectral factor of the overall Nyquist filter (a square-root in the frequency domain). This spectral factor can be used in a similar manner to the square-root raised-cosine filter in matched filtering applications. A square-root of the filter is placed on the transmiter's end and the other square root is placed at the receiver's end.

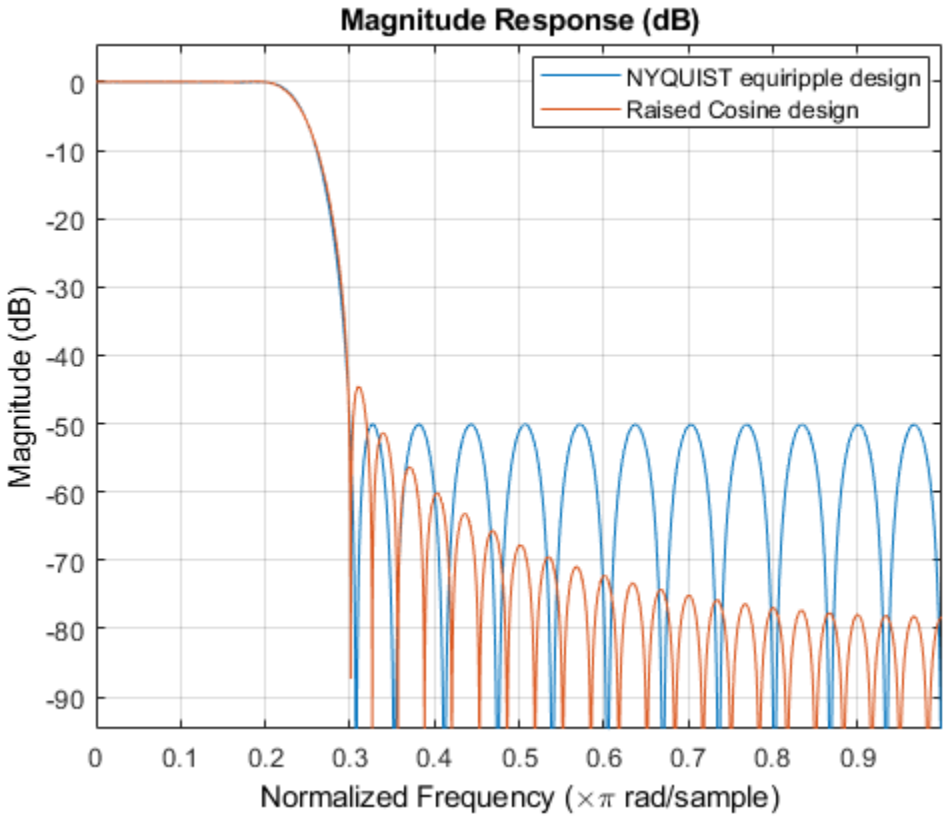
```
f1.FilterOrder = 30;
f1.Band = NBand;
f1.TransitionWidth = TW;
eq1 = design(f1,'equiripple','Minphase',true,'SystemObject',true);
coeffs = rcosdesign(R,N/NBand,NBand);
coeffs = coeffs / max(coeffs) * (-1/(pi*NBand) * (pi*(R-1) - 4*R));
srrc   = dsp.FIRFilter('Numerator',coeffs);
fvt = fvtool(eq1,srrc,'Color','white');
legend(fvt,'Minimum-phase equiripple design',...
    'Square-root raised-cosine design');
```



Decreasing the Rolloff Factor

The response of the raised-cosine filter improves as the rolloff factor decreases (shown here for rolloff = 0.2). This is because of the narrow main lobe of the frequency response of a rectangular window that is used in the truncation of the impulse response.

```
f1.FilterOrder = N;
f1.TransitionWidth = .1;
eq1 = design(f1,'equiripple','Zerophase',true,'SystemObject',true);
R = 0.2;
coeffs = rcosdesign(R,N/NBand,NBand,'normal');
coeffs = coeffs/max(abs(coeffs))/NBand;
rc1   = dsp.FIRFilter('Numerator',coeffs);
fvt = fvtool(eq1,rc1,'Color','white');
legend(fvt,'NYQUIST equiripple design','Raised Cosine design');
```



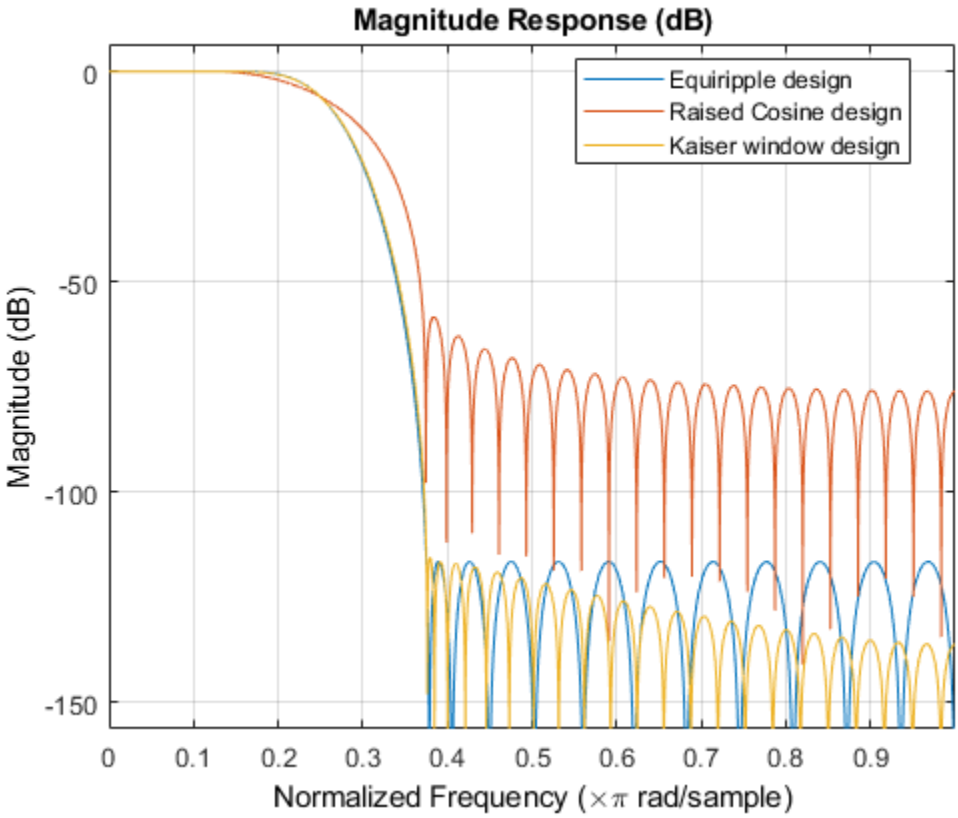
Windowed-Impulse-Response Nyquist Design

Nyquist filters can also be designed using the truncated-and-windowed impulse response method. This can be another alternative to the raised-cosine design. For example we can use the Kaiser window method to design a filter that meets the initial specs:

```
f1.TransitionWidth = TW;
kaiserFilt = design(f1,'kaiserwin','SystemObject',true);
```

The Kaiser window design requires the same order (60) as the equiripple design to meet the specs. (Remember that in contrast we required an extraordinary 1400th-order raised-cosine filter to meet the stopband spec.)

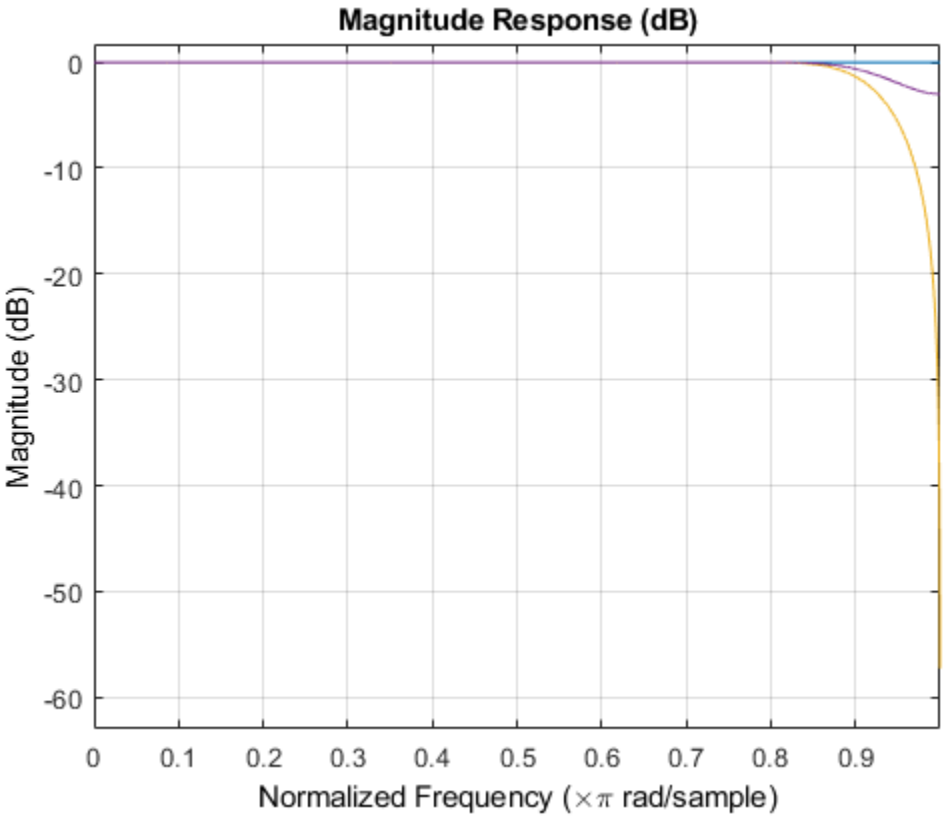
```
fvt = fvtool(eq,rc,kaiserFilt,'Color','white');
legend(fvt,'Equiripple design',...
    'Raised Cosine design','Kaiser window design');
```



Nyquist Filters for Interpolation

Besides digital data transmission, Nyquist filters are attractive for interpolation purposes. The reason is that every L samples you have a zero sample (except for the middle sample) as mentioned before. There are two advantages to this, both are obvious by looking at the polyphase representation.

```
fm = fdesign.interpolator(4,'nyquist');
kaiserFilt = design(fm,'kaiserwin','SystemObject',true);
fvt = fvtool(kaiserFilt,'Color','white');
fvt.PolyphaseView = 'on';
```



The polyphase subfilter #4 is an allpass filter, in fact it is a pure delay (select impulse response in FVTool, or look at the filter coefficients in FVTool), so that: 1. All of its multipliers are zero except for one, leading to an efficient implementation of that polyphase branch. 2. The input samples are passed through the interpolation filter without modification, even though the filter is not ideal.

See Also

More About

- “Filter Builder Design Process” on page 23-2
- “Using Filter Designer” on page 22-2

Digital Frequency Transformations

In this section...
“Details and Methodology” on page 4-60
“Frequency Transformations for Real Filters” on page 4-65
“Frequency Transformations for Complex Filters” on page 4-74

Details and Methodology

- “Overview of Transformations” on page 4-60
- “Select Features Subject to Transformation” on page 4-62
- “Mapping from Prototype Filter to Target Filter” on page 4-64
- “Summary of Frequency Transformations” on page 4-65

Overview of Transformations

Converting existing FIR or IIR filter designs to a modified IIR form is often done using allpass frequency transformations. Although the resulting designs can be considerably more expensive in terms of dimensionality than the prototype (original) filter, their ease of use in fixed or variable applications is a big advantage.

The general idea of the frequency transformation is to take an existing prototype filter and produce another filter from it that retains some of the characteristics of the prototype, in the frequency domain. Transformation functions achieve this by replacing each delaying element of the prototype filter with an allpass filter carefully designed to have a prescribed phase characteristic for achieving the modifications requested by the designer.

The basic form of mapping commonly used is

$$H_T(z) = H_o[H_A(z)]$$

The $H_A(z)$ is an Nth-order allpass mapping filter given by

$$H_A(z) = S\frac{\sum_{i=0}^{N} \alpha_i z^{-i}}{\sum_{i=0}^{N} \alpha_i z^{-N+i}} = \frac{N_A(z)}{D_A(z)}$$

$$\alpha_0 = 1$$

where

$H_o(z)$ — Transfer function of the prototype filter

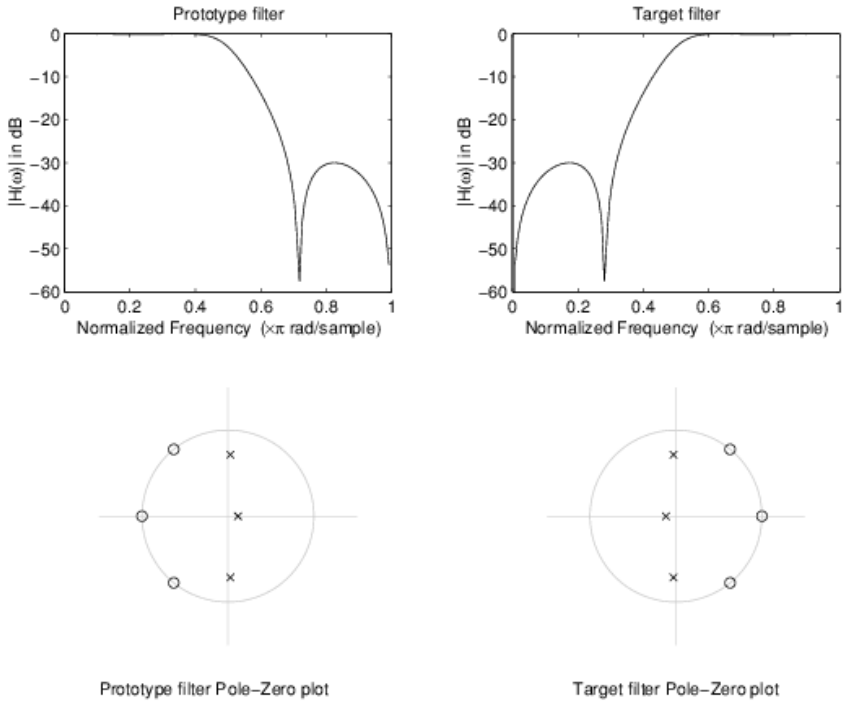
$H_A(z)$ — Transfer function of the allpass mapping filter

$H_T(z)$ — Transfer function of the target filter

Let's look at a simple example of the transformation given by

$$H_T(z) = H_o(-z)$$

The target filter has its poles and zeroes flipped across the origin of the real and imaginary axes. For the real filter prototype, it gives a mirror effect against 0.5, which means that lowpass $H_o(z)$ gives rise to a real highpass $H_T(z)$. This is shown in the following figure for the prototype filter designed as a third-order halfband elliptic filter.



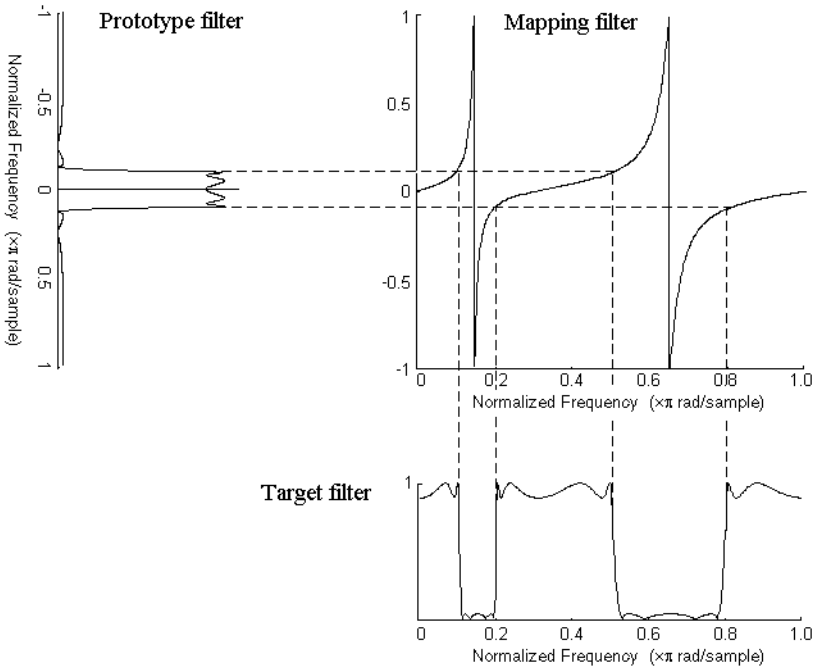
Example of a Simple Mirror Transformation

The choice of an allpass filter to provide the frequency mapping is necessary to provide the frequency translation of the prototype filter frequency response to the target filter by changing the frequency position of the features from the prototype filter without affecting the overall shape of the filter response.

The phase response of the mapping filter normalized to π can be interpreted as a translation function:

$$H(w_{new}) = \omega_{old}$$

The graphical interpretation of the frequency transformation is shown in the figure below. The complex multiband transformation takes a real lowpass filter and converts it into a number of passbands around the unit circle.



Graphical Interpretation of the Mapping Process

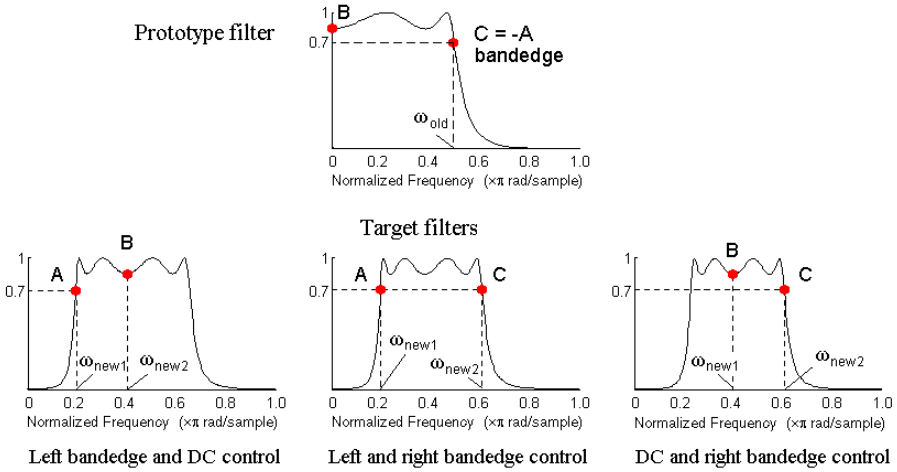
Most of the frequency transformations are based on the second-order allpass mapping filter:

$$H_A(z) = \pm \frac{1 + \alpha_1 z^{-1} + \alpha_2 z^{-2}}{\alpha_2 + \alpha_1 z^{-1} + z^{-2}}$$

The two degrees of freedom provided by α_1 and α_2 choices are not fully used by the usual restrictive set of “flat-top” classical mappings like lowpass to bandpass. Instead, any two transfer function features can be migrated to (almost) any two other frequency locations if α_1 and α_2 are chosen so as to keep the poles of $H_A(z)$ strictly outside the unit circle (since $H_A(z)$ is substituted for z in the prototype transfer function). Moreover, as first pointed out by Constantinides, the selection of the outside sign influences whether the original feature at zero can be moved (the minus sign, a condition known as “DC mobility”) or whether the Nyquist frequency can be migrated (the “Nyquist mobility” case arising when the leading sign is positive).

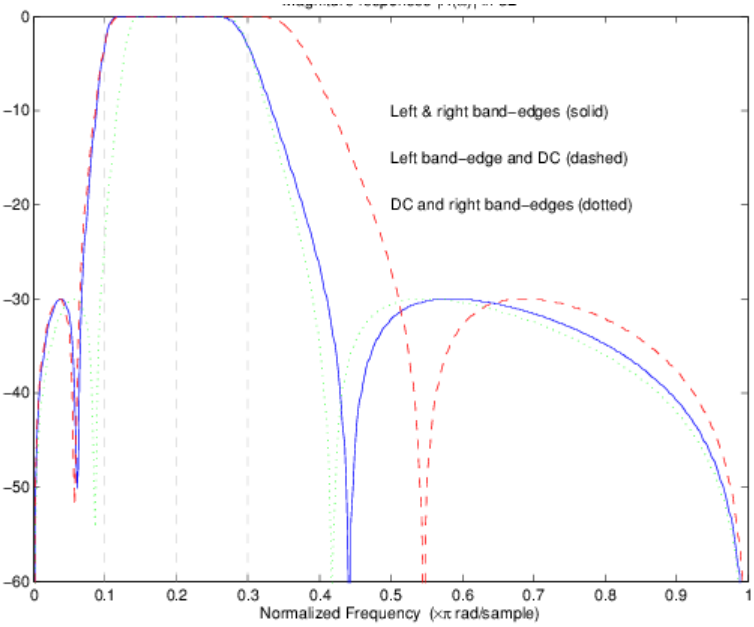
Select Features Subject to Transformation

Choosing the appropriate frequency transformation for achieving the required effect and the correct features of the prototype filter is very important and needs careful consideration. It is not advisable to use a first-order transformation for controlling more than one feature. The mapping filter will not give enough flexibility. It is also not good to use higher order transformation just to change the cutoff frequency of the lowpass filter. The increase of the filter order would be too big, without considering the additional replica of the prototype filter that may be created in undesired places.



Feature Selection for Real Lowpass to Bandpass Transformation

To illustrate the idea, the second-order real multipoint transformation was applied three times to the same elliptic halfband filter in order to make it into a bandpass filter. In each of the three cases, two different features of the prototype filter were selected in order to obtain a bandpass filter with passband ranging from 0.25 to 0.75. The position of the DC feature was not important, but it would be advantageous if it were in the middle between the edges of the passband in the target filter. In the first case the selected features were the left and the right band edges of the lowpass filter passband, in the second case they were the left band edge and the DC, in the third case they were DC and the right band edge.



Result of Choosing Different Features

The results of all three approaches are completely different. For each of them only the selected features were positioned precisely where they were required. In the first case the DC is moved toward the left passband edge just like all the other features close to the left edge being squeezed

there. In the second case the right passband edge was pushed way out of the expected target as the precise position of DC was required. In the third case the left passband edge was pulled toward the DC in order to position it at the correct frequency. The conclusion is that if only the DC can be anywhere in the passband, the edges of the passband should have been selected for the transformation. For most of the cases requiring the positioning of passbands and stopbands, designers should always choose the position of the edges of the prototype filter in order to make sure that they get the edges of the target filter in the correct places. Frequency responses for the three cases considered are shown in the figure. The prototype filter was a third-order elliptic lowpass filter with cutoff frequency at 0.5.

The MATLAB code used to generate the figure is given here.

The prototype filter is a halfband elliptic, real, third-order lowpass filter:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

In the example the requirements are set to create a real bandpass filter with passband edges at 0.1 and 0.3 out of the real lowpass filter having the cutoff frequency at 0.5. This is attempted in three different ways. In the first approach both edges of the passband are selected, in the second approach the left edge of the passband and the DC are chosen, while in the third approach the DC and the right edge of the passband are taken:

```
[num1,den1] = iirlp2xn(b, a, [-0.5, 0.5], [0.1, 0.3]);
[num2,den2] = iirlp2xn(b, a, [-0.5, 0.0], [0.1, 0.2]);
[num3,den3] = iirlp2xn(b, a, [ 0.0, 0.5], [0.2, 0.3]);
```

Mapping from Prototype Filter to Target Filter

In general the frequency mapping converts the prototype filter, $H_o(z)$, to the target filter, $H_T(z)$, using the N_Ath-order allpass filter, $H_A(z)$. The general form of the allpass mapping filter is given in “Overview of Transformations” on page 4-60. The frequency mapping is a mathematical operation that replaces each delayer of the prototype filter with an allpass filter. There are two ways of performing such mapping. The choice of the approach is dependent on how prototype and target filters are represented.

When the Nth-order prototype filter is given with pole-zero form

$$H_o(z) = \frac{\sum_{i=1}^{N}(z - z_i)}{\sum_{i=1}^{N}(z - p_i)}$$

the mapping will replace each pole, p_i, and each zero, z_i, with a number of poles and zeros equal to the order of the allpass mapping filter:

$$H_o(z) = \frac{\sum_{i=1}^{N}\left(S\sum_{k=0}^{N-1}\alpha_k z^k - z_i \cdot \sum_{k=0}^{N-1}\alpha_k z^{N-k}\right)}{\sum_{i=1}^{N}\left(S\sum_{k=0}^{N-1}\alpha_k z^k - p_i \cdot \sum_{k=0}^{N-1}\alpha_k z^{N-k}\right)}$$

The root finding needs to be used on the bracketed expressions in order to find the poles and zeros of the target filter.

When the prototype filter is described in the numerator-denominator form:

$$H_T(z) = \left.\frac{\beta_0 z^N + \beta_1 z^{N-1} + \ldots + \beta_N}{\alpha_0 z^N + \alpha_1 z^{N-1} + \ldots + \alpha_N}\right|_{z = H_A(z)}$$

Then the mapping process will require a number of convolutions in order to calculate the numerator and denominator of the target filter:

$$H_T(z) = \frac{\beta_1 N_A(z)^N + \beta_2 N_A(z)^{N-1} D_A(z) + \ldots + \beta_N D_A(z)^N}{\beta_1 N_A(z)^N + \beta_2 N_A(z)^{N-1} D_A(z) + \ldots + \beta_N D_A(z)^N}$$

For each coefficient α_i and β_i of the prototype filter the N_Ath-order polynomials must be convolved N times. Such approach may cause rounding errors for large prototype filters and/or high order mapping filters. In such a case the user should consider the alternative of doing the mapping using via poles and zeros.

Summary of Frequency Transformations

Advantages

- Most frequency transformations are described by closed-form solutions or can be calculated from the set of linear equations.
- They give predictable and familiar results.
- Ripple heights from the prototype filter are preserved in the target filter.
- They are architecturally appealing for variable and adaptive filters.

Disadvantages

- There are cases when using optimization methods to design the required filter gives better results.
- High-order transformations increase the dimensionality of the target filter, which may give expensive final results.
- Starting from fresh designs helps avoid locked-in compromises.

Frequency Transformations for Real Filters

- “Overview” on page 4-66
- “Real Frequency Shift” on page 4-66
- “Real Lowpass to Real Lowpass” on page 4-67
- “Real Lowpass to Real Highpass” on page 4-68
- “Real Lowpass to Real Bandpass” on page 4-69
- “Real Lowpass to Real Bandstop” on page 4-70
- “Real Lowpass to Real Multiband” on page 4-71
- “Real Lowpass to Real Multipoint” on page 4-73

Overview

This section discusses real frequency transformations that take the real lowpass prototype filter and convert it into a different real target filter. The target filter has its frequency response modified in respect to the frequency response of the prototype filter according to the characteristic of the applied frequency transformation.

Real Frequency Shift

Real frequency shift transformation uses a second-order allpass mapping filter. It performs an exact mapping of one selected feature of the frequency response into its new location, additionally moving both the Nyquist and DC features. This effectively moves the whole response of the lowpass filter by the distance specified by the selection of the feature from the prototype filter and the target filter. As a real transformation, it works in a similar way for positive and negative frequencies.

$$H_A(z) = z^{-1} \cdot \frac{1 - \alpha z^{-1}}{\alpha - z^{-1}}$$

with α given by

$$\alpha = \begin{cases} \dfrac{\cos\frac{\pi}{2}(\omega_{old} - 2\omega_{new})}{\cos\frac{\pi}{2}\omega_{old}} & for \quad \left|\cos\frac{\pi}{2}(\omega_{old} - 2\omega_{new})\right| < 1 \\ \dfrac{\sin\frac{\pi}{2}(\omega_{old} - 2\omega_{new})}{\sin\frac{\pi}{2}\omega_{old}} & otherwise \end{cases}$$

where

ω_{old} — Frequency location of the selected feature in the prototype filter

ω_{new} — Position of the feature originally at ω_{old} in the target filter

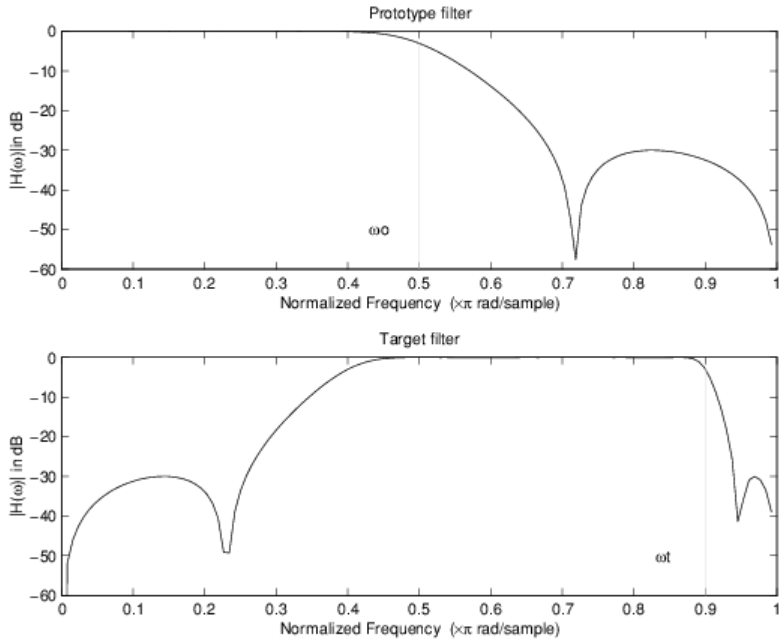
The following example shows how this transformation can be used to move the response of the prototype lowpass filter in either direction. Please note that because the target filter must also be real, the response of the target filter will inherently be disturbed at frequencies close to Nyquist and close to DC. Here is the MATLAB code for generating the example in the figure.

The prototype filter is a halfband elliptic, real, third-order lowpass filter:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

The example transformation moves the feature originally at 0.5 to 0.9:

```
[num,den] = iirshift(b, a, 0.5, 0.9);
```



Example of Real Frequency Shift Mapping

Real Lowpass to Real Lowpass

Real lowpass filter to real lowpass filter transformation uses a first-order allpass mapping filter. It performs an exact mapping of one feature of the frequency response into the new location keeping DC and Nyquist features fixed. As a real transformation, it works in a similar way for positive and negative frequencies. It is important to mention that using first-order mapping ensures that the order of the filter after the transformation is the same as it was originally.

$$H_A(z) = -\left(\frac{1-\alpha z^{-1}}{\alpha - z^{-1}}\right)$$

with α given by

$$\alpha = \frac{\sin\frac{\pi}{2}(\omega_{old} - \omega_{new})}{\sin\frac{\pi}{2}(\omega_{old} - \omega_{new})}$$

where

ω_{old} — Frequency location of the selected feature in the prototype filter

ω_{new} — Frequency location of the same feature in the target filter

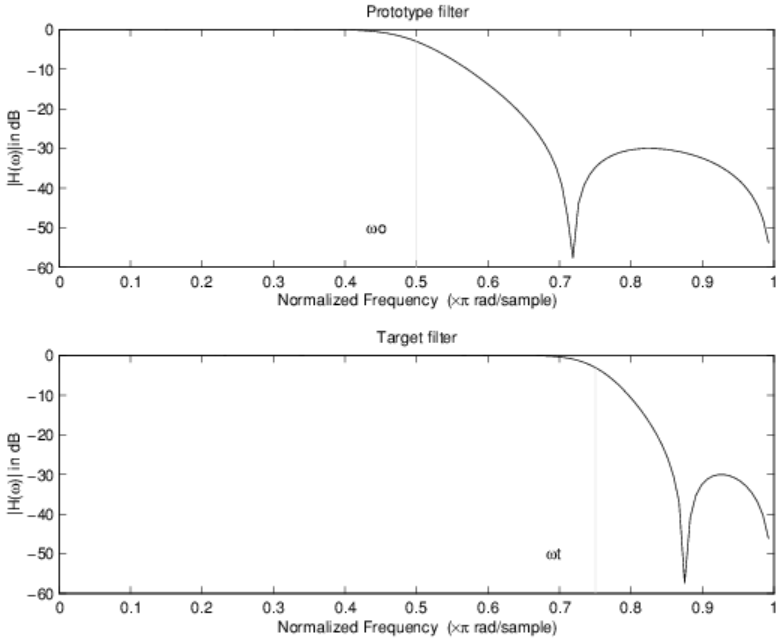
The example below shows how to modify the cutoff frequency of the prototype filter. The MATLAB code for this example is shown in the following figure.

The prototype filter is a halfband elliptic, real, third-order filter:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

The cutoff frequency moves from 0.5 to 0.75:

```
[num,den] = iirlp2lp(b, a, 0.5, 0.75);
```



Example of Real Lowpass to Real Lowpass Mapping

Real Lowpass to Real Highpass

Real lowpass filter to real highpass filter transformation uses a first-order allpass mapping filter. It performs an exact mapping of one feature of the frequency response into the new location additionally swapping DC and Nyquist features. As a real transformation, it works in a similar way for positive and negative frequencies. Just like in the previous transformation because of using a first-order mapping, the order of the filter before and after the transformation is the same.

$$H_A(z) = -\left(\frac{1+\alpha z^{-1}}{\alpha + z^{-1}}\right)$$

with α given by

$$\alpha = -\left(\frac{\cos\frac{\pi}{2}(\omega_{old}+\omega_{new})}{\cos\frac{\pi}{2}(\omega_{old}-\omega_{new})}\right)$$

where

ω_{old} — Frequency location of the selected feature in the prototype filter

ω_{new} — Frequency location of the same feature in the target filter

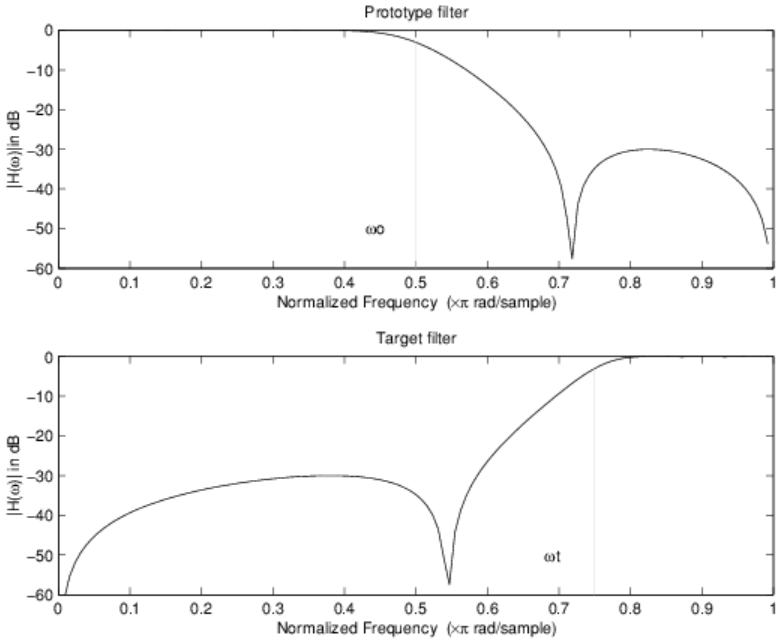
The example below shows how to convert the lowpass filter into a highpass filter with arbitrarily chosen cutoff frequency. In the MATLAB code below, the lowpass filter is converted into a highpass with cutoff frequency shifted from 0.5 to 0.75. Results are shown in the figure.

The prototype filter is a halfband elliptic, real, third-order filter:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

The example moves the cutoff frequency from 0.5 to 0.75:

```
[num,den] = iirlp2hp(b, a, 0.5, 0.75);
```



Example of Real Lowpass to Real Highpass Mapping

Real Lowpass to Real Bandpass

Real lowpass filter to real bandpass filter transformation uses a second-order allpass mapping filter. It performs an exact mapping of two features of the frequency response into their new location additionally moving a DC feature and keeping the Nyquist feature fixed. As a real transformation, it works in a similar way for positive and negative frequencies.

$$H_A(z) = -\left(\frac{1-\beta(1+\alpha)z^{-1}-\alpha z^{-2}}{\alpha-\beta(1+\alpha)z^{-1}+z^{-2}}\right)$$

with α and β given by

$$\alpha = \frac{\sin\frac{\pi}{4}(2\omega_{old}-\omega_{new,2}+\omega_{new,1})}{\sin\frac{\pi}{4}(2\omega_{old}+\omega_{new,2}-\omega_{new,1})}$$

$$\beta = \cos\frac{\pi}{2}(\omega_{new,1}+\omega_{new,2})$$

where

ω_{old} — Frequency location of the selected feature in the prototype filter

$\omega_{new,1}$ — Position of the feature originally at ($-\omega_{old}$) in the target filter

$\omega_{new,2}$ — Position of the feature originally at ($+\omega_{old}$) in the target filter

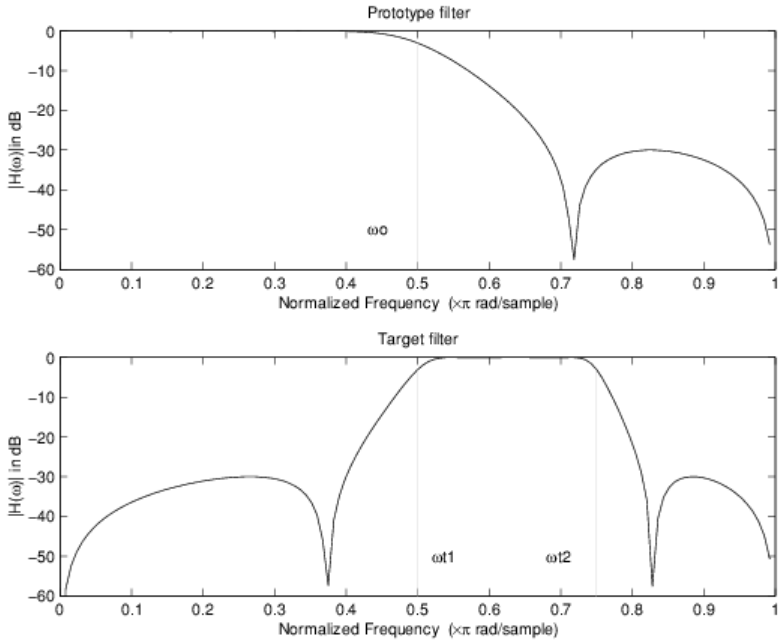
The example below shows how to move the response of the prototype lowpass filter in either direction. Please note that because the target filter must also be real, the response of the target filter will inherently be disturbed at frequencies close to Nyquist and close to DC. Here is the MATLAB code for generating the example in the figure.

The prototype filter is a halfband elliptic, real, third-order lowpass filter:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

The example transformation creates the passband between 0.5 and 0.75:

```
[num,den] = iirlp2bp(b, a, 0.5, [0.5, 0.75]);
```



Example of Real Lowpass to Real Bandpass Mapping

Real Lowpass to Real Bandstop

Real lowpass filter to real bandstop filter transformation uses a second-order allpass mapping filter. It performs an exact mapping of two features of the frequency response into their new location additionally moving a Nyquist feature and keeping the DC feature fixed. This effectively creates a stopband between the selected frequency locations in the target filter. As a real transformation, it works in a similar way for positive and negative frequencies.

$$H_A(z) = \frac{1 - \beta(1+\alpha)z^{-1} + \alpha z^{-2}}{\alpha - \beta(1+\alpha)z^{-1} + z^{-2}}$$

with α and β given by

$$\alpha = \frac{\cos\frac{\pi}{4}(2\omega_{old} + \omega_{new,2} - \omega_{new,1})}{\cos\frac{\pi}{4}(2\omega_{old} - \omega_{new,2} + \omega_{new,1})}$$

$$\beta = \cos\frac{\pi}{2}(\omega_{new,1} + \omega_{new,2})$$

where

ω_{old} — Frequency location of the selected feature in the prototype filter

$\omega_{new,1}$ — Position of the feature originally at ($-\omega_{old}$) in the target filter

$\omega_{new,2}$ — Position of the feature originally at ($+\omega_{old}$) in the target filter

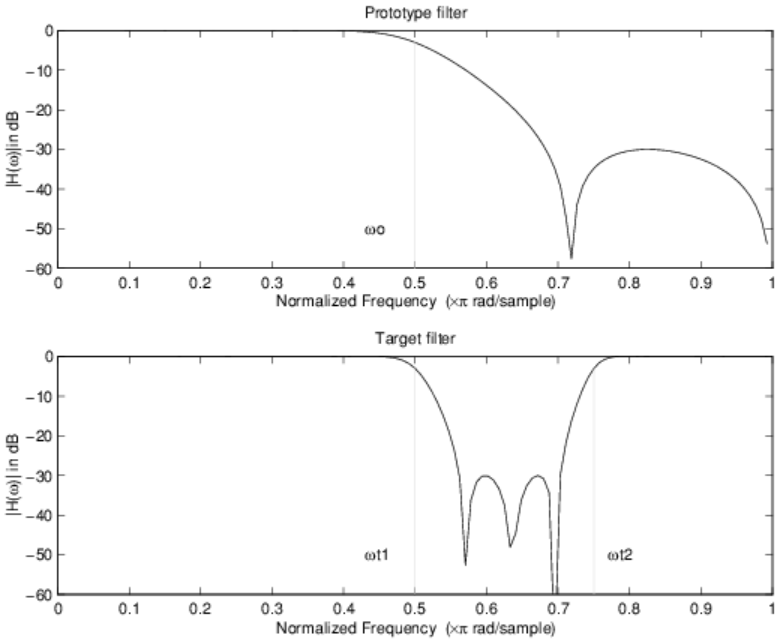
The following example shows how this transformation can be used to convert the prototype lowpass filter with cutoff frequency at 0.5 into a real bandstop filter with the same passband and stopband ripple structure and stopband positioned between 0.5 and 0.75. Here is the MATLAB code for generating the example in the figure.

The prototype filter is a halfband elliptic, real, third-order lowpass filter:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

The example transformation creates a stopband from 0.5 to 0.75:

```
[num,den] = iirlp2bs(b, a, 0.5, [0.5, 0.75]);
```



Example of Real Lowpass to Real Bandstop Mapping

Real Lowpass to Real Multiband

This high-order transformation performs an exact mapping of one selected feature of the prototype filter frequency response into a number of new locations in the target filter. Its most common use is to

convert a real lowpass with predefined passband and stopband ripples into a real multiband filter with N arbitrary band edges, where N is the order of the allpass mapping filter.

$$H_A(z) = S\frac{\sum_{i=0}^{N}\alpha_i z^{-i}}{\sum_{i=0}^{N}\alpha_i z^{-N+i}}$$

$$\alpha_0 = 1$$

The coefficients α are given by

$$\begin{cases}\alpha_0 = 1 \;\; k = 1, \ldots, N \\ \alpha_k = -S\dfrac{\sin\frac{\pi}{2}(N\omega_{new} + (-1)^k\omega_{old})}{\sin\frac{\pi}{2}((N-2k)\omega_{new} + (-1)^k\omega_{old})}\end{cases}$$

where

$\omega_{old,k}$ — Frequency location of the first feature in the prototype filter

$\omega_{new,k}$ — Position of the feature originally at $\omega_{old,k}$ in the target filter

The mobility factor, S, specifies the mobility or either DC or Nyquist feature:

$$S = \begin{cases} 1 & Nyquist \\ -1 & DC \end{cases}$$

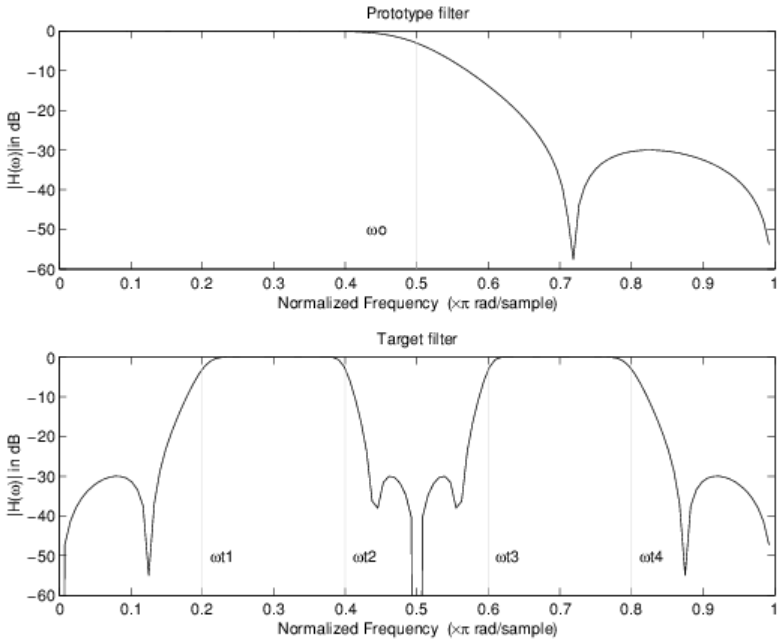
The example below shows how this transformation can be used to convert the prototype lowpass filter with cutoff frequency at 0.5 into a filter having a number of bands positioned at arbitrary edge frequencies 1/5, 2/5, 3/5 and 4/5. Parameter S was such that there is a passband at DC. Here is the MATLAB code for generating the figure.

The prototype filter is a halfband elliptic, real, third-order lowpass filter:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

The example transformation creates three stopbands, from DC to 0.2, from 0.4 to 0.6 and from 0.8 to Nyquist:

```
[num,den] = iirlp2mb(b, a, 0.5, [0.2, 0.4, 0.6, 0.8], `pass');
```



Example of Real Lowpass to Real Multiband Mapping

Real Lowpass to Real Multipoint

This high-order frequency transformation performs an exact mapping of a number of selected features of the prototype filter frequency response to their new locations in the target filter. The mapping filter is given by the general IIR polynomial form of the transfer function as given below.

$$H_A(z) = S\frac{\sum_{i=0}^{N}\alpha_i z^{-i}}{\sum_{i=0}^{N}\alpha_i z^{-N+i}}$$

$$\alpha_0 = 1$$

For the Nth-order multipoint frequency transformation the coefficients α are

$$\begin{cases} \sum_{i=1}^{N}\alpha_{N-i} z_{old,k}\cdot z_{new,k}^{i} - S\cdot z_{new,k}^{N-i} = -z_{old,k} - S\cdot z_{new,k} \\ z_{old,k} = e^{j\pi\omega_{old,k}} \\ z_{new,k} = e^{j\pi\omega_{new,k}} \\ k = 1,\ldots,N \end{cases}$$

where

$\omega_{old,k}$ — Frequency location of the first feature in the prototype filter

$\omega_{new,k}$ — Position of the feature originally at $\omega_{old,k}$ in the target filter

The mobility factor, S, specifies the mobility of either DC or Nyquist feature:

$$S = \begin{cases} 1 & Nyquist \\ -1 & DC \end{cases}$$

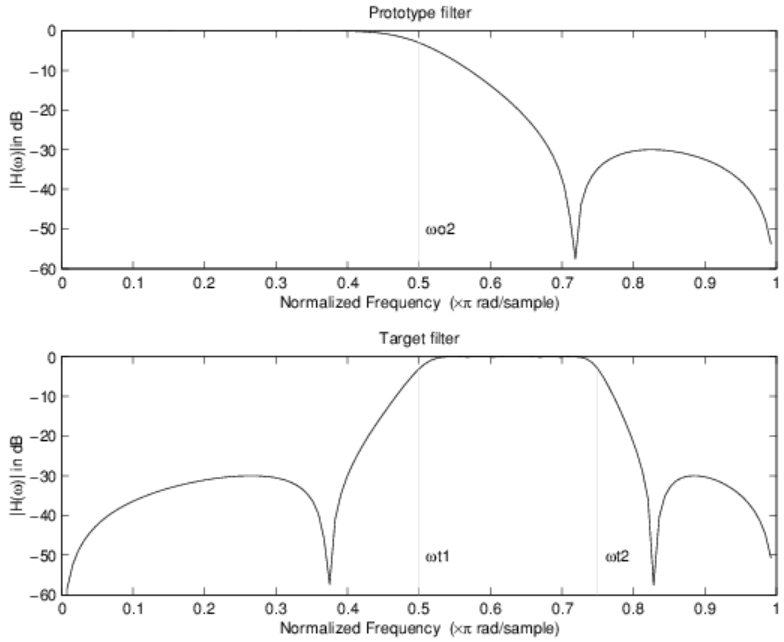
The example below shows how this transformation can be used to move features of the prototype lowpass filter originally at -0.5 and 0.5 to their new locations at 0.5 and 0.75, effectively changing a position of the filter passband. Here is the MATLAB code for generating the figure.

The prototype filter is a halfband elliptic, real, third-order lowpass filter:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

The example transformation creates a passband from 0.5 to 0.75:

```
[num,den] = iirlp2xn(b, a, [-0.5, 0.5], [0.5, 0.75], `pass');
```



Example of Real Lowpass to Real Multipoint Mapping

Frequency Transformations for Complex Filters

- “Overview” on page 4-74
- “Complex Frequency Shift” on page 4-75
- “Real Lowpass to Complex Bandpass” on page 4-76
- “Real Lowpass to Complex Bandstop” on page 4-77
- “Real Lowpass to Complex Multiband” on page 4-78
- “Real Lowpass to Complex Multipoint” on page 4-80
- “Complex Bandpass to Complex Bandpass” on page 4-81

Overview

This section discusses complex frequency transformation that take the complex prototype filter and convert it into a different complex target filter. The target filter has its frequency response modified

in respect to the frequency response of the prototype filter according to the characteristic of the applied frequency transformation from:

Complex Frequency Shift

Complex frequency shift transformation is the simplest first-order transformation that performs an exact mapping of one selected feature of the frequency response into its new location. At the same time it rotates the whole response of the prototype lowpass filter by the distance specified by the selection of the feature from the prototype filter and the target filter.

$$H_A(z) = \alpha z^{-1}$$

with α given by

$$\alpha = e^{j2\pi(\nu_{new} - \nu_{old})}$$

where

ω_{old} — Frequency location of the selected feature in the prototype filter

ω_{new} — Position of the feature originally at ω_{old} in the target filter

A special case of the complex frequency shift is a, so called, Hilbert Transformer. It can be designed by setting the parameter to $|\alpha|=1$, that is

$$\alpha = \begin{cases} 1 & forward \\ -1 & inverse \end{cases}$$

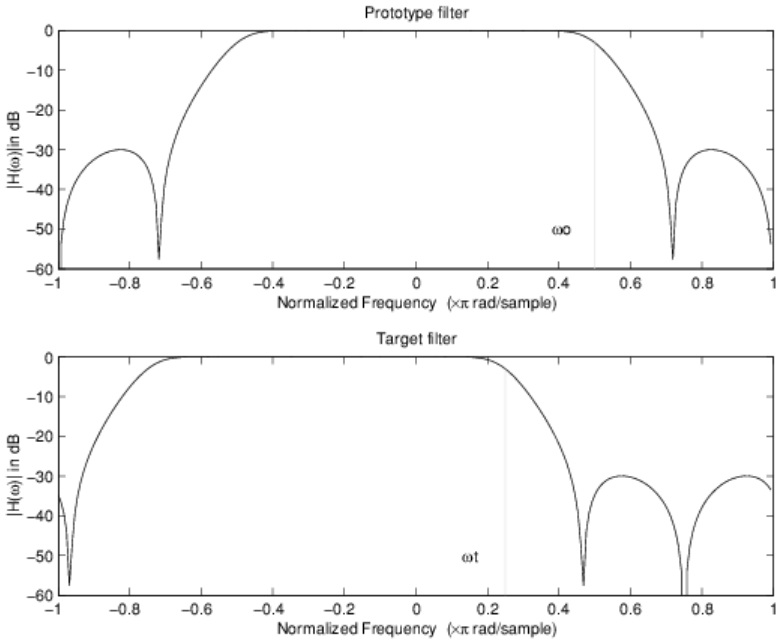
The example below shows how to apply this transformation to rotate the response of the prototype lowpass filter in either direction. Please note that because the transformation can be achieved by a simple phase shift operator, all features of the prototype filter will be moved by the same amount. Here is the MATLAB code for generating the example in the figure.

The prototype filter is a halfband elliptic, real, third-order lowpass filter:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

The example transformation moves the feature originally at 0.5 to 0.3:

```
[num,den] = iirshiftc(b, a, 0.5, 0.3);
```



Example of Complex Frequency Shift Mapping

Real Lowpass to Complex Bandpass

This first-order transformation performs an exact mapping of one selected feature of the prototype filter frequency response into two new locations in the target filter creating a passband between them. Both Nyquist and DC features can be moved with the rest of the frequency response.

$$H_A(z) = \frac{\beta - \alpha z^{-1}}{z^{-1} - \alpha\beta}$$

with α and β are given by

$$\alpha = \frac{\sin\frac{\pi}{4}(2\omega_{old} - \omega_{new,2} + \omega_{new,1})}{\sin\pi(2\omega_{old} + \omega_{new,2} - \omega_{new,1})}$$

$$\beta = e^{-j\pi(\omega_{new} - \omega_{old})}$$

where

ω_{old} — Frequency location of the selected feature in the prototype filter

$\omega_{new,1}$ — Position of the feature originally at ($-\omega_{old}$) in the target filter

$\omega_{new,2}$ — Position of the feature originally at ($+\omega_{old}$) in the target filter

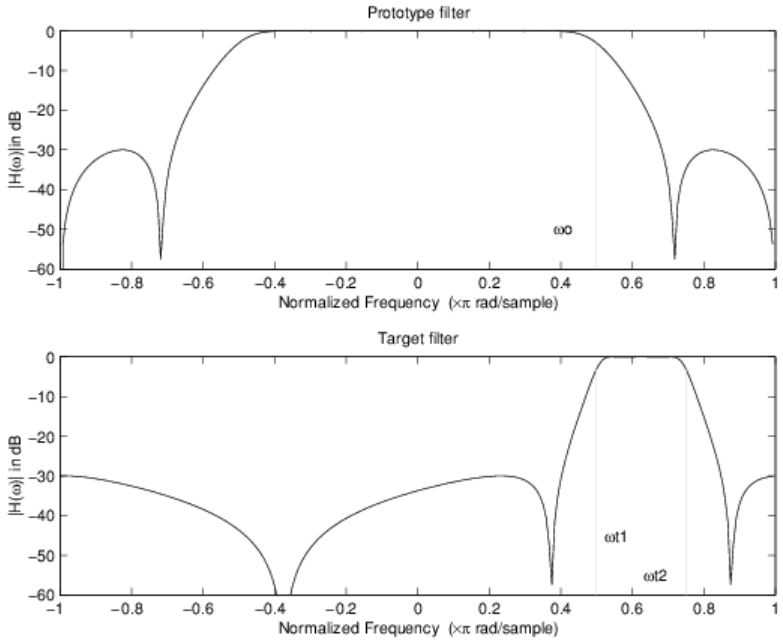
The following example shows the use of such a transformation for converting a real halfband lowpass filter into a complex bandpass filter with band edges at 0.5 and 0.75. Here is the MATLAB code for generating the example in the figure.

The prototype filter is a half band elliptic, real, third-order lowpass filter:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

The transformation creates a passband from 0.5 to 0.75:

```
[num,den] = iirlp2bpc(b, a, 0.5, [0.5 0.75]);
```



Example of Real Lowpass to Complex Bandpass Mapping

Real Lowpass to Complex Bandstop

This first-order transformation performs an exact mapping of one selected feature of the prototype filter frequency response into two new locations in the target filter creating a stopband between them. Both Nyquist and DC features can be moved with the rest of the frequency response.

$$H_A(z) = \frac{\beta - \alpha z^{-1}}{\alpha\beta - z^{-1}}$$

with α and β are given by

$$\alpha = \frac{\cos\pi(2\omega_{old} + \nu_{new,2} - \nu_{new,1})}{\cos\pi(2\omega_{old} - \nu_{new,2} + \nu_{new,1})}$$

$$\beta = e^{-j\pi(\omega_{new} - \omega_{old})}$$

where

ω_{old} — Frequency location of the selected feature in the prototype filter

$\omega_{new,1}$ — Position of the feature originally at ($-\omega_{old}$) in the target filter

$\omega_{new,2}$ — Position of the feature originally at ($+\omega_{old}$) in the target filter

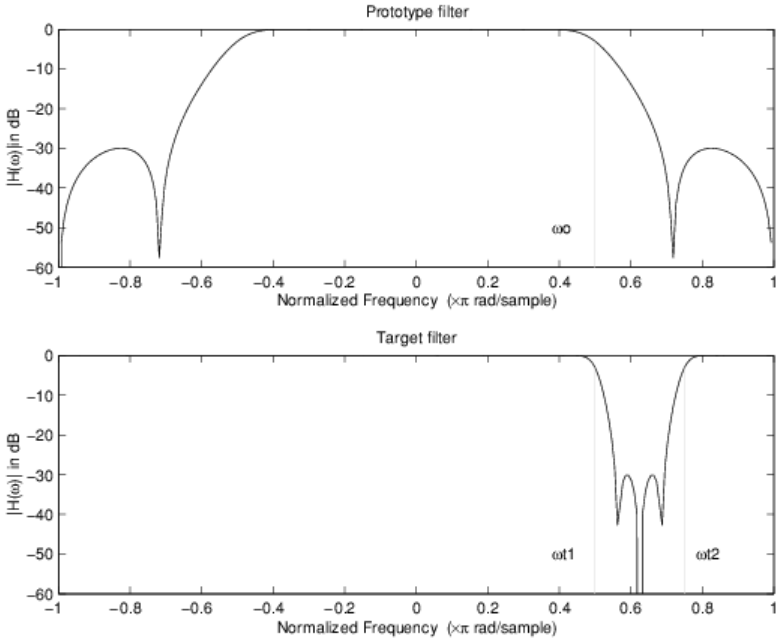
The example below shows the use of such a transformation for converting a real halfband lowpass filter into a complex bandstop filter with band edges at 0.5 and 0.75. Here is the MATLAB code for generating the example in the figure.

The prototype filter is a halfband elliptic, real, third-order lowpass filter:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

The transformation creates a stopband from 0.5 to 0.75:

```
[num,den] = iirlp2bsc(b, a, 0.5, [0.5 0.75]);
```



Example of Real Lowpass to Complex Bandstop Mapping

Real Lowpass to Complex Multiband

This high-order transformation performs an exact mapping of one selected feature of the prototype filter frequency response into a number of new locations in the target filter. Its most common use is to convert a real lowpass with predefined passband and stopband ripples into a multiband filter with arbitrary band edges. The order of the mapping filter must be even, which corresponds to an even number of band edges in the target filter. The *N*th-order complex allpass mapping filter is given by the following general transfer function form:

$$H_A(z) = S\frac{\sum_{i=0}^{N}\alpha_i z^{-i}}{\sum_{i=0}^{N}\alpha_i \pm z^{-N+i}}$$

$$\alpha_0 = 1$$

The coefficients α are calculated from the system of linear equations:

$$\begin{cases} \sum_{i=1}^{N} \Re(\alpha_i) \cdot \left[\cos\beta_{1,k} - \cos\beta_{2,k}\right] + \Im(\alpha_i) \cdot \left[\sin\beta_{1,k} + \sin\beta_{2,k}\right] = \cos\beta_{3,k} \\ \sum_{i=1}^{N} \Re(\alpha_i) \cdot \left[\sin\beta_{1,k} - \sin\beta_{2,k}\right] - \Im(\alpha_i) \cdot \left[\cos\beta_{1,k} + \cos\beta_{2,k}\right] = \sin\beta_{3,k} \\ \beta_{1,k} = -\pi\left[\nu_{old} \cdot (-1)^k + \nu_{new,k}(N-k)\right] \\ \beta_{2,k} = -\pi\left[\Delta C + \nu_{new,k} k\right] \\ \beta_{3,k} = -\pi\left[\nu_{old} \cdot (-1)^k + \nu_{new,k} N\right] \\ k = 1 \ldots N \end{cases}$$

where

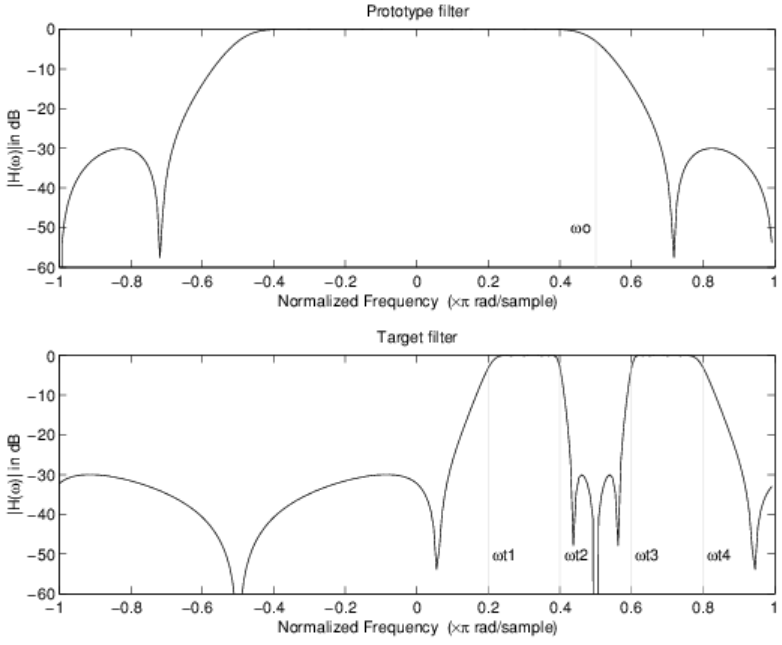
ω_{old} — Frequency location of the selected feature in the prototype filter

$\omega_{new,i}$ — Position of features originally at $\pm\omega_{old}$ in the target filter

Parameter S is the additional rotation factor by the frequency distance ΔC, giving the additional flexibility of achieving the required mapping:

$$S = e^{-j\pi\Delta C}$$

The example shows the use of such a transformation for converting a prototype real lowpass filter with the cutoff frequency at 0.5 into a multiband complex filter with band edges at 0.2, 0.4, 0.6 and 0.8, creating two passbands around the unit circle. Here is the MATLAB code for generating the figure.



Example of Real Lowpass to Complex Multiband Mapping

The prototype filter is a halfband elliptic, real, third-order lowpass filter:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

The example transformation creates two complex passbands:

```
[num,den] = iirlp2mbc(b, a, 0.5, [0.2, 0.4, 0.6, 0.8]);
```

Real Lowpass to Complex Multipoint

This high-order transformation performs an exact mapping of a number of selected features of the prototype filter frequency response to their new locations in the target filter. The *N*th-order complex allpass mapping filter is given by the following general transfer function form.

$$H_A(z) = S\frac{\sum_{i=0}^{N}\alpha_i z^{-i}}{\sum_{i=0}^{N}\alpha_i \pm z^{-N+i}}$$

$$\alpha_0 = 1$$

The coefficients α can be calculated from the system of linear equations:

$$\begin{cases} \sum_{i=1}^{N} \Re(\alpha_i)\cdot\left[\cos\beta_{1,k} - \cos\beta_{2,k}\right] + \Im(\alpha_i)\cdot\left[\sin\beta_{1,k} + \sin\beta_{2,k}\right] = \cos\beta_{3,k} \\ \sum_{i=1}^{N} \Re(\alpha_i)\cdot\left[\sin\beta_{1,k} - \sin\beta_{2,k}\right] - \Im(\alpha_i)\cdot\left[\cos\beta_{1,k} + \cos\beta_{2,k}\right] = \sin\beta_{3,k} \\ \beta_{1,k} = -\frac{\pi}{2}\left[\omega_{old,k} + \omega_{new,k}(N-k)\right] \\ \beta_{2,k} = -\frac{\pi}{2}\left[2\Delta C + \omega_{new,k}k\right] \\ \beta_{3,k} = -\frac{\pi}{2}\left[\omega_{old,k} + \omega_{new,k}N\right] \\ k = 1\ldots N \end{cases}$$

where

$\omega_{old,k}$ — Frequency location of the first feature in the prototype filter

$\omega_{new,k}$ — Position of the feature originally at $\omega_{old,k}$ in the target filter

Parameter S is the additional rotation factor by the frequency distance ΔC, giving the additional flexibility of achieving the required mapping:

$$S = e^{-j\pi\Delta C}$$

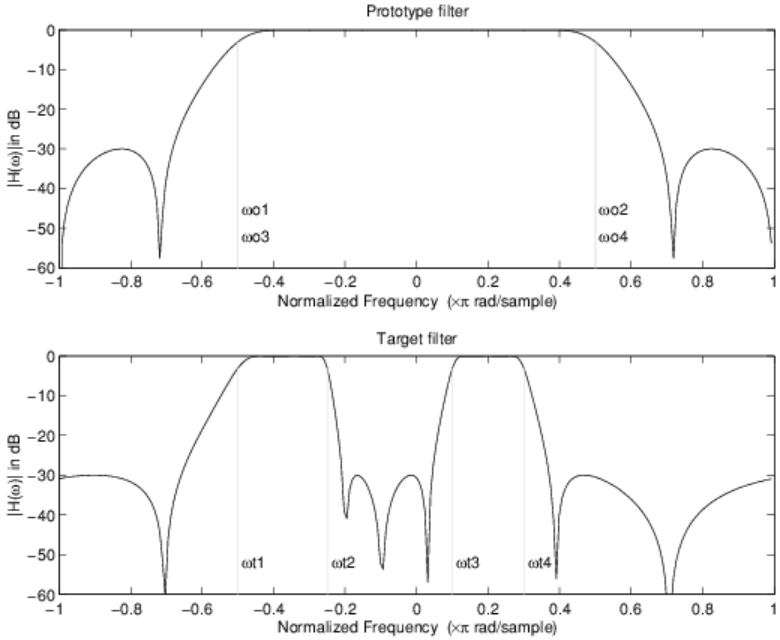
The following example shows how this transformation can be used to move one selected feature of the prototype lowpass filter originally at -0.5 to two new frequencies -0.5 and 0.1, and the second feature of the prototype filter from 0.5 to new locations at -0.25 and 0.3. This creates two nonsymmetric passbands around the unit circle, creating a complex filter. Here is the MATLAB code for generating the figure.

The prototype filter is a halfband elliptic, real, third-order lowpass filter:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

The example transformation creates two nonsymmetric passbands:

```
[num,den] = iirlp2xc(b,a,0.5*[-1,1,-1,1], [-0.5,-0.25,0.1,0.3]);
```



Example of Real Lowpass to Complex Multipoint Mapping

Complex Bandpass to Complex Bandpass

This first-order transformation performs an exact mapping of two selected features of the prototype filter frequency response into two new locations in the target filter. Its most common use is to adjust the edges of the complex bandpass filter.

$$H_A(z) = \frac{\alpha(\gamma - \beta z^{-1})}{z^{-1} - \beta\gamma}$$

with α and β are given by

$$\alpha = \frac{\sin\frac{\pi}{4}((\omega_{old,2} - \omega_{old,1}) - (\omega_{new,2} - \omega_{new,1}))}{\sin\frac{\pi}{4}((\omega_{old,2} - \omega_{old,1}) + (\omega_{new,2} - \omega_{new,1}))}$$

$$\alpha = e^{-j\pi(\omega_{old,2} - \omega_{old,1})}$$

$$\gamma = e^{-j\pi(\omega_{new,2} - \omega_{new,1})}$$

where

$\omega_{old,1}$ — Frequency location of the first feature in the prototype filter

$\omega_{old,2}$ — Frequency location of the second feature in the prototype filter

$\omega_{new,1}$ — Position of the feature originally at $\omega_{old,1}$ in the target filter

$\omega_{new,2}$ — Position of the feature originally at $\omega_{old,2}$ in the target filter

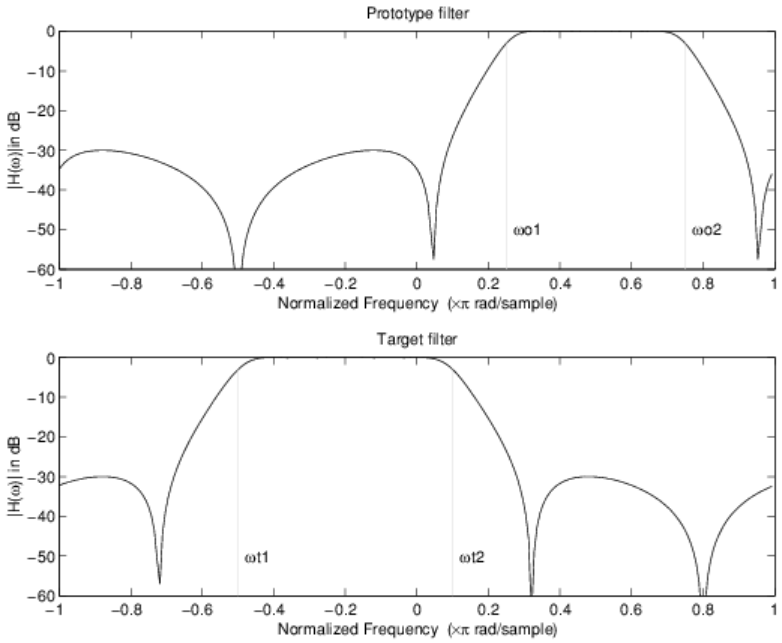
The following example shows how this transformation can be used to modify the position of the passband of the prototype filter, either real or complex. In the example below the prototype filter passband spanned from 0.5 to 0.75. It was converted to having a passband between -0.5 and 0.1. Here is the MATLAB code for generating the figure.

The prototype filter is a halfband elliptic, real, third-order lowpass filter:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

The example transformation creates a passband from 0.25 to 0.75:

```
[num,den] = iirbpc2bpc(b, a, [0.25, 0.75], [-0.5, 0.1]);
```



Example of Complex Bandpass to Complex Bandpass Mapping

Digital Filter Design Block

In this section...
“Overview of the Digital Filter Design Block” on page 4-83
“Select a Filter Design Block” on page 4-84
“Create a Lowpass Filter in Simulink” on page 4-85
“Create a Highpass Filter in Simulink” on page 4-85
“Filter High-Frequency Noise in Simulink” on page 4-86

Overview of the Digital Filter Design Block

You can use the Digital Filter Design block to design and implement a digital filter. The filter you design can filter single-channel or multichannel signals. The Digital Filter Design block is ideal for simulating the numerical behavior of your filter on a floating-point system, such as a personal computer or DSP chip. You can use the Simulink Coder product to generate C code from your filter block.

Filter Design and Analysis

You perform all filter design and analysis within the filter designer app, which opens when you double-click the Digital Filter Design block. Filter designer provides extensive filter design parameters and analysis tools such as pole-zero and impulse response plots.

Filter Implementation

Once you have designed your filter using filter designer, the block automatically realizes the filter using the filter structure you specify. You can then use the block to filter signals in your model. You can also fine-tune the filter by changing the filter specification parameters during a simulation. The outputs of the Digital Filter Design block numerically match the outputs of the equivalent filter System object, when you pass the same input.

Saving, Exporting, and Importing Filters

The Digital Filter Design block allows you to save the filters you design, export filters (to the MATLAB workspace, MAT-files, etc.), and import filters designed elsewhere.

To learn how to save your filter designs, see “Saving and Opening Filter Design Sessions” on page 22-22. To learn how to import and export your filter designs, see “Import and Export Quantized Filters” on page 4-32.

Note You can use the Digital Filter Design block to design and implement a filter. To implement a pre-designed filter, use the Discrete FIR Filter or Biquad Filter blocks. Both methods implement a filter design in the same manner and have the same behavior during simulation and code generation.

See the Digital Filter Design block reference page for more information. For information on choosing between the Digital Filter Design block and the Filter Realization Wizard, see “Select a Filter Design Block” on page 4-84.

Select a Filter Design Block

This section explains the similarities and differences between the Digital Filter Design and Filter Realization Wizard blocks.

Similarities

The Digital Filter Design block and Filter Realization Wizard are similar in the following ways:

- Filter design and analysis options — Both blocks use the filter designer app for filter design and analysis.
- Output values — If the output of both blocks is double-precision floating point, single-precision floating point, or fixed point, the output values of both blocks numerically match the output values of the equivalent System objects, when you pass the same input.

Differences

The Digital Filter Design block and Filter Realization Wizard handle the following things differently:

- Supported filter structures — Both blocks support many of the same basic filter structures, but the Filter Realization Wizard supports more structures than the Digital Filter Design block. This is because the block can implement filters using Sum, Gain, and Delay blocks. See the Filter Realization Wizard and Digital Filter Design block reference pages for a list of all the structures they support.
- Data type support — The Filter Realization Wizard block supports single- and double-precision floating-point computation for all filter structures and fixed-point computation for some filter structures. The Digital Filter Design block only supports single- and double-precision floating-point computation.
- Block versus Wizard — The Digital Filter Design block is the filter itself, but the Filter Realization Wizard block just enables you to create new filters and put them in an existing model. Thus, the Filter Realization Wizard is not a block that processes data in your model, it is a wizard that generates filter blocks (or subsystems) which you can then use to process data in your model.

When to Use Each Block

The following are specific situations where only the Digital Filter Design block or the Filter Realization Wizard is appropriate.

- Digital Filter Design
  - Use to simulate single- and double-precision floating-point filters.
  - Use to generate highly optimized ANSI® C code that implements floating-point filters for embedded systems.
- Filter Realization Wizard
  - Use to simulate numerical behavior of fixed-point filters in a DSP chip, a field-programmable gate array (FPGA), or an application-specific integrated circuit (ASIC).
  - Use to simulate single- and double-precision floating-point filters with structures that the Digital Filter Design block does not support.
  - Use to visualize the filter structure, as the block can build the filter from Sum, Gain, and Delay blocks.

- Use to rapidly generate multiple filter blocks.

See “Filter Realization Wizard” on page 4-90 and the Filter Realization Wizard block reference page for information.

Create a Lowpass Filter in Simulink

You can use the Digital Filter Design block to design and implement a digital FIR or IIR filter. In this topic, you use it to create an FIR lowpass filter:

1. Open Simulink and create a new model file.
2. From the DSP System Toolbox Filtering library, and then from the Filter Implementations library, click-and-drag a Digital Filter Design block into your model.
3. Double-click the Digital Filter Design block.

   The filter designer app opens.
4. Set the parameters as follows, and then click **OK**:
   - **Response Type** = `Lowpass`
   - **Design Method** = `FIR`, `Equiripple`
   - **Filter Order** = `Minimum order`
   - **Units** = `Normalized (0 to 1)`
   - **wpass** = `0.2`
   - **wstop** = `0.5`
5. Click **Design Filter** at the bottom of the app to design the filter.

   Your Digital Filter Design block now represents a filter with the parameters you specified.
6. From the **Edit** menu, select **Convert Structure**.

   The **Convert Structure** dialog box opens.
7. Select **Direct-Form FIR Transposed** and click **OK**.
8. Rename your block `Digital Filter Design - Lowpass`.

The Digital Filter Design block now represents a lowpass filter with a Direct-Form FIR Transposed structure. The filter passes all frequencies up to 20% of the Nyquist frequency (half the sampling frequency), and stops frequencies greater than or equal to 50% of the Nyquist frequency as defined by the **wpass** and **wstop** parameters. In the next topic, “Create a Highpass Filter in Simulink” on page 4-85, you use a Digital Filter Design block to create a highpass filter. For more information about implementing a pre-designed filter, see “Digital Filter Implementations” on page 4-100.

Create a Highpass Filter in Simulink

In this topic, you create a highpass filter using the Digital Filter Design block:

1. If the model you created in “Create a Lowpass Filter in Simulink” on page 4-85 is not open on your desktop, you can open an equivalent model by typing

   ```
   ex_filter_ex4
   ```

   at the MATLAB command prompt.

2. From the DSP System Toolbox Filtering library, and then from the Filter Implementations library, click-and-drag a second Digital Filter Design block into your model.
3. Double-click the Digital Filter Design block.

   The filter designer app opens.
4. Set the parameters as follows:
   - **Response Type** = `Highpass`
   - **Design Method** = `FIR`, `Equiripple`
   - **Filter Order** = `Minimum order`
   - **Units** = `Normalized (0 to 1)`
   - **wstop** = `0.2`
   - **wpass** = `0.5`
5. Click the **Design Filter** button at the bottom of the app to design the filter.

   Your Digital Filter Design block now represents a filter with the parameters you specified.
6. In the **Edit** menu, select **Convert Structure**.

   The **Convert Structure** dialog box opens.
7. Select **Direct-Form FIR Transposed** and click **OK**.
8. Rename your block `Digital Filter Design - Highpass` .

The block now implements a highpass filter with a direct form FIR transpose structure. The filter passes all frequencies greater than or equal to 50% of the Nyquist frequency (half the sampling frequency), and stops frequencies less than or equal to 20% of the Nyquist frequency as defined by the **wpass** and **wstop** parameters. This highpass filter is the opposite of the lowpass filter described in “Create a Lowpass Filter in Simulink” on page 4-85. The highpass filter passes the frequencies stopped by the lowpass filter, and stops the frequencies passed by the lowpass filter. In the next topic, “Filter High-Frequency Noise in Simulink” on page 4-86, you use these Digital Filter Design blocks to create a model capable of removing high frequency noise from a signal. For more information about implementing a pre-designed filter, see “Digital Filter Implementations” on page 4-100.

Filter High-Frequency Noise in Simulink

In the previous topics, you used Digital Filter Design blocks to create FIR lowpass and highpass filters. In this topic, you use these blocks to build a model that removes high frequency noise from a signal. In this model, you use the highpass filter, which is excited using a uniform random signal, to create high-frequency noise. After you add this noise to a sine wave, you use the lowpass filter to filter out the high-frequency noise:

1. If the model you created in “Create a Highpass Filter in Simulink” on page 4-85 is not open on your desktop, you can open an equivalent model by typing

   ```
   ex_filter_ex5
   ```

   at the MATLAB command prompt.
2. Click-and-drag the following blocks into your model.

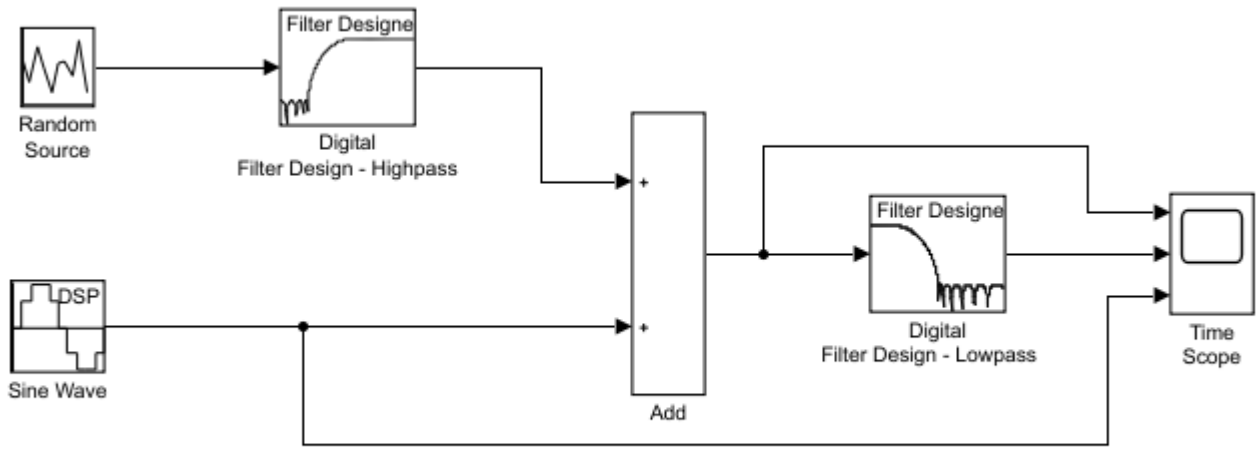
Block	Library	Quantity
Add	Simulink Math Operations library	1
Random Source	Sources	1
Sine Wave	Sources	1
Time Scope	Sinks	1

3 Set the parameters for these blocks as indicated in the following table. Leave the parameters not listed in the table at their default settings.

Parameter Settings for the Other Blocks

Block	Parameter Setting
Add	• **Icon shape** = `rectangular` • **List of signs** = `++`
Random Source	• **Source type =** = `Uniform` • **Minimum** = `0` • **Maximum** = `4` • **Sample mode** = `Discrete` • **Sample time** = `1/1000` • **Samples per frame** = `50`
Sine Wave	• **Frequency (Hz)** = `75` • **Sample time** = `1/1000` • **Samples per frame** = `50`
Time Scope	• **File** > **Number of Input Ports** > **3** • **View** > **Configuration Properties** • Open the **Time** tab and set **Time span** = `One frame period`

4 Connect the blocks as shown in the following figure. You might need to resize some of the blocks to accomplish this task.



5 In the **Modeling** tab, click **Model Settings**. The **Configuration Parameters** dialog box opens.

6 In the **Solver** pane, set the parameters as follows, and then click **OK**:

- **Start time** = `0`
- **Stop time** = `5`
- **Type** = `Fixed-step`
- **Solver** = `Discrete (no continuous states)`

7 In the **Simulation** tab, select **Run**.

The model simulation begins and the scope displays the three input signals.

8 After simulation is complete, select **View > Legend** from the Time Scope menu. The legend appears in the Time Scope window. You can click-and-drag it anywhere on the scope display. To change the channel names, double-click inside the legend and replace the default channel names with the following:

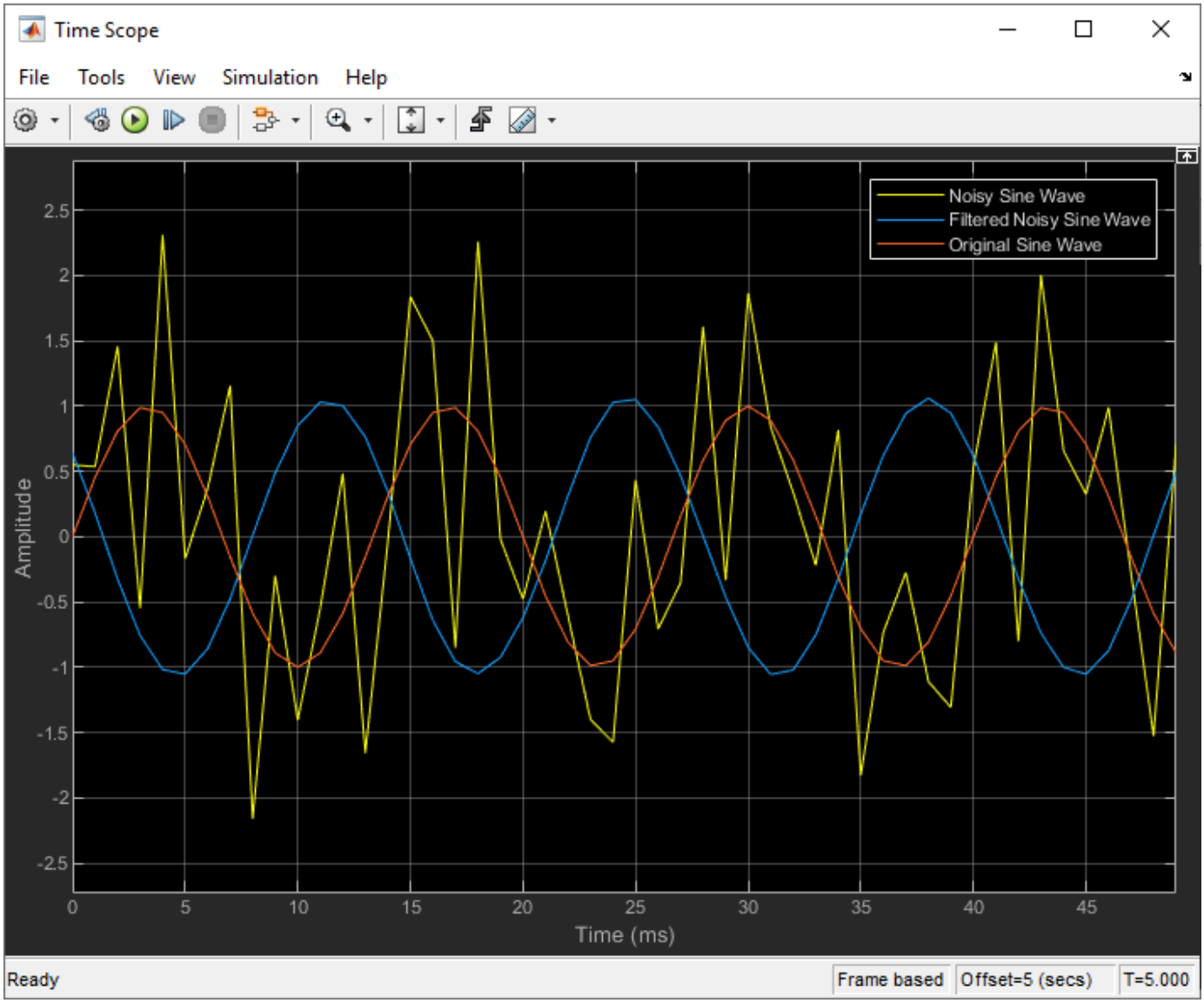
- Add = `Noisy Sine Wave`
- Digital Filter Design – Lowpass = `Filtered Noisy Sine Wave`
- Sine Wave = `Original Sine Wave`

In the next step, you will set the color, style, and marker of each channel.

9 In the Time Scope window, select **View > Style**, and set the following:

Signal	Line	Marker	Color
Noisy Sine Wave	**-**	**None**	**Black**
Filtered Noisy Sine Wave	**-**	**diamond**	**Red**
Original Sine Wave	**None**	*	**Blue**

10 The **Time Scope** display should now appear as follows:



You can see that the lowpass filter filters out the high-frequency noise in the noisy sine wave.

You have now used Digital Filter Design blocks to build a model that removes high frequency noise from a signal. For more information about these blocks, see the Digital Filter Design block reference page. For information on another block capable of designing and implementing filters, see “Filter Realization Wizard” on page 4-90. To learn how to save your filter designs, see “Saving and Opening Filter Design Sessions” on page 22-22. To learn how to import and export your filter designs, see “Import and Export Quantized Filters” on page 4-32.

Filter Realization Wizard

In this section...
“Overview of the Filter Realization Wizard” on page 4-90
“Design and Implement a Fixed-Point Filter in Simulink” on page 4-90
“Set the Filter Structure and Number of Filter Sections” on page 4-97
“Optimize the Filter Structure” on page 4-97

Overview of the Filter Realization Wizard

The Filter Realization Wizard is another DSP System Toolbox block that can be used to design and implement digital filters. You can use this tool to filter single-channel floating-point or fixed-point signals. Like the Digital Filter Design block, double-clicking a Filter Realization Wizard block opens filter designer. Unlike the Digital Filter Design block, the Filter Realization Wizard starts filter designer with the **Realize Model** panel selected. This panel is optimized for use with DSP System Toolbox software.

For more information, see the Filter Realization Wizard block reference page. For information on choosing between the Digital Filter Design block and the Filter Realization Wizard, see “Select a Filter Design Block” on page 4-84.

Design and Implement a Fixed-Point Filter in Simulink

In this section, a tutorial guides you through creating a fixed-point filter with the Filter Realization Wizard. You will use the Filter Realization Wizard to remove noise from a signal. This tutorial has the following parts:

- “Part 1 — Create a Signal with Added Noise” on page 4-90
- “Part 2 — Create a Fixed-Point Filter with the Filter Realization Wizard” on page 4-91
- “Part 3 — Build a Model to Filter a Signal” on page 4-95
- “Part 4 — Examine Filtering Results” on page 4-96

Part 1 — Create a Signal with Added Noise

In this section of the tutorial, you will create a signal with added noise. Later in the tutorial, you will filter this signal with a fixed-point filter that you design with the Filter Realization Wizard.

1. Type

   ```
   load mtlb
   soundsc(mtlb,Fs)
   ```

   at the MATLAB command line. You should hear a voice say “MATLAB.” This is the signal to which you will add noise.
2. Create a noise signal by typing

   ```
   noise = cos(2*pi*3*Fs/8*(0:length(mtlb)-1)/Fs)';
   ```

   at the command line. You can hear the noise signal by typing

   ```
   soundsc(noise,Fs)
   ```

3 Add the noise to the original signal by typing

```
u = mtlb + noise;
```

at the command line.

4 Scale the signal with noise by typing

```
u = u/max(abs(u));
```

at the command line. You scale the signal to try to avoid overflows later on. You can hear the scaled signal with noise by typing

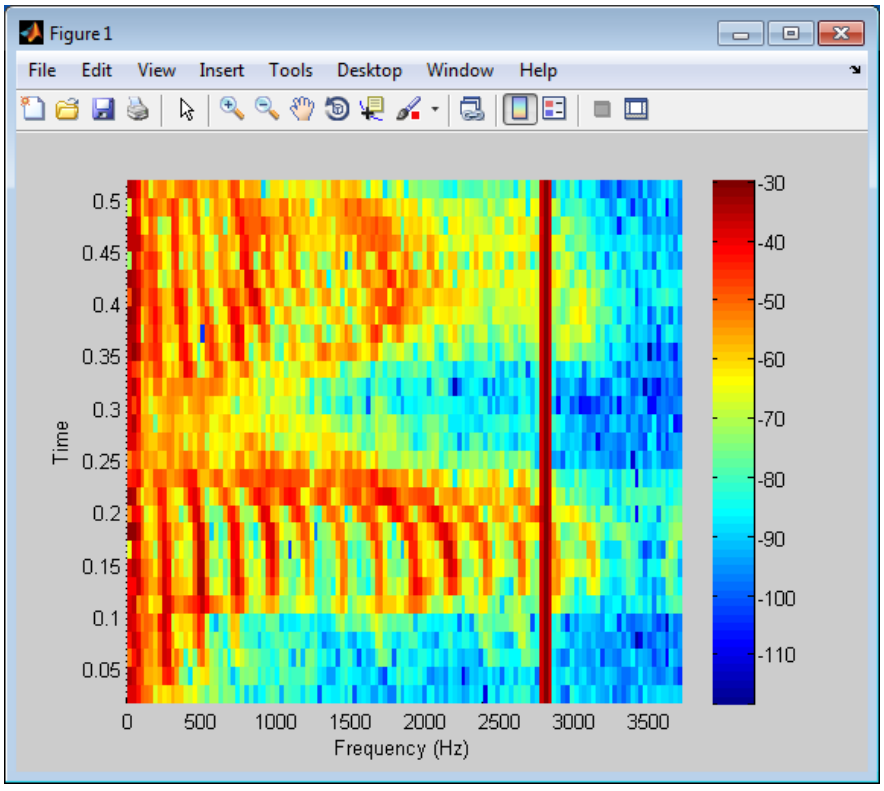
```
soundsc(u,Fs)
```

5 View the scaled signal with noise by typing

```
spectrogram(u,256,[],[],Fs);colorbar
```

at the command line.

The spectrogram appears as follows.



In the spectrogram, you can see the noise signal as a line at about 2800 Hz, which is equal to `3*Fs/8`.

Part 2 — Create a Fixed-Point Filter with the Filter Realization Wizard

Next you will create a fixed-point filter using the Filter Realization Wizard. You will create a filter that reduces the effects of the noise on the signal.

1. Open a new Simulink model, and drag-and-drop a Filter Realization Wizard block from the Filtering / Filter Implementations library into the model.

   Note You do not have to place a Filter Realization Wizard block in a model in order to use it. You can open the app from within a library. However, for purposes of this tutorial, we will keep the Filter Realization Wizard block in the model.

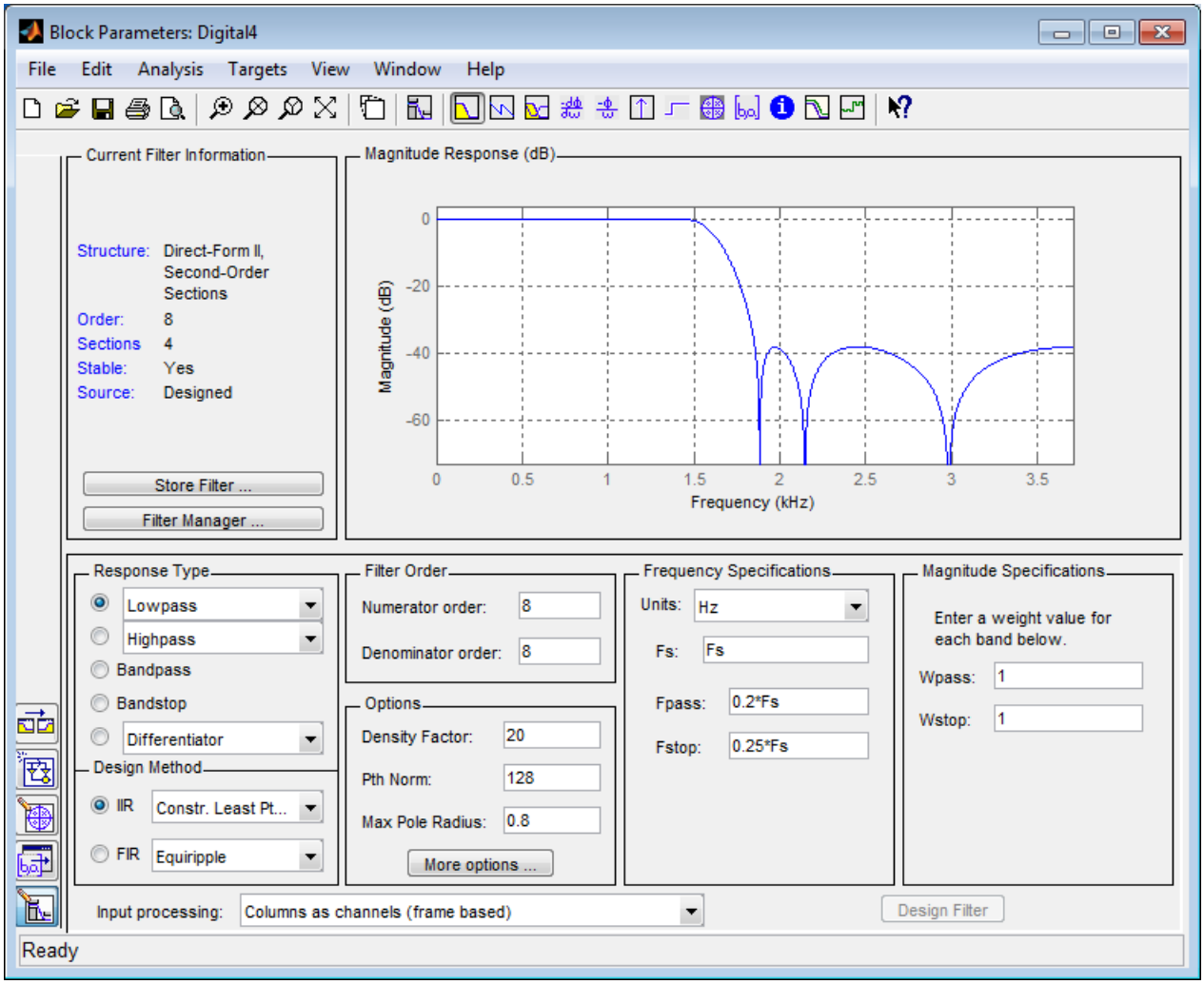
2. Double-click the Filter Realization Wizard block in your model. The **Realize Model** panel of the filter designer appears.

3. Click the Design Filter button ( ) on the bottom left of filter designer. This brings forward the **Design filter** panel of the tool.

4. Set the following fields in the **Design filter** panel:

   - Set **Design Method** to `IIR -- Constrained Least Pth-norm`
   - Set **Fs** to `Fs`
   - Set **Fpass** to `0.2*Fs`
   - Set **Fstop** to `0.25*Fs`
   - Set **Max pole radius** to `0.8`
   - Click the **Design Filter** button

   The **Design filter** panel should now appear as follows.

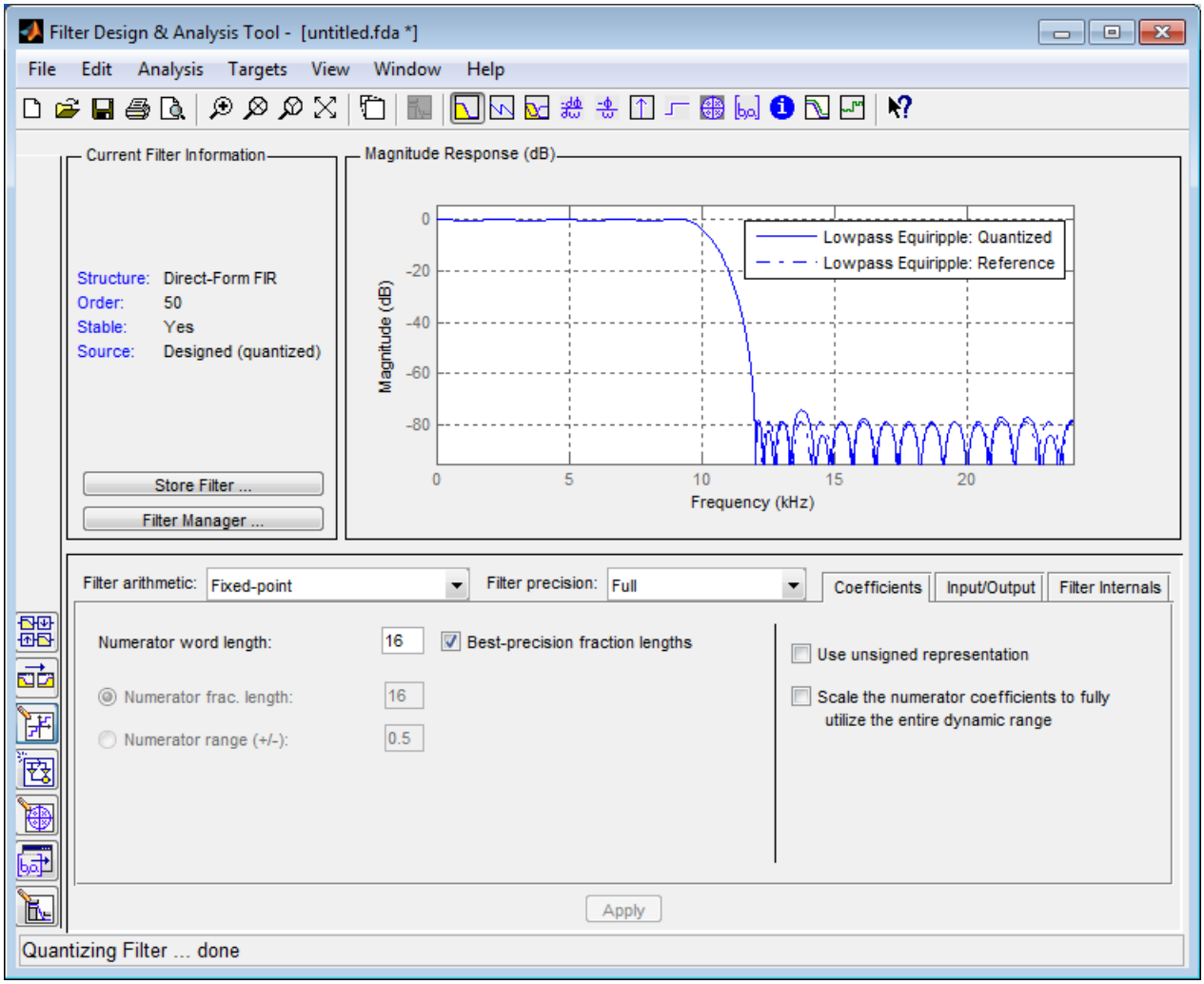


5 Click the **Set quantization parameters** button on the bottom left of filter designer ( ). This brings forward the **Set quantization parameters** panel of the tool.

6 Set the following fields in the **Set quantization parameters** panel:

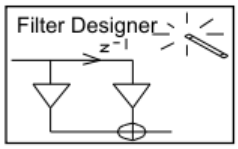
- Select `Fixed-point` for the **Filter arithmetic** parameter.
- Make sure the **Best precision fraction lengths** check box is selected on the **Coefficients** pane.

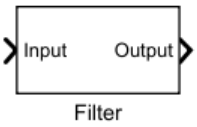
The **Set quantization parameters** panel should appear as follows.



7 Click the Realize Model button on the left side of filter designer ( ). This brings forward the **Realize Model** panel of the tool.

8 Select the **Build model using basic elements** check box, then click the **Realize Model** button on the bottom of filter designer. A subsystem block for the new filter appears in your model.





Note You do not have to keep the Filter Realization Wizard block in the same model as the generated Filter block. However, for this tutorial, we will keep the blocks in the same model.

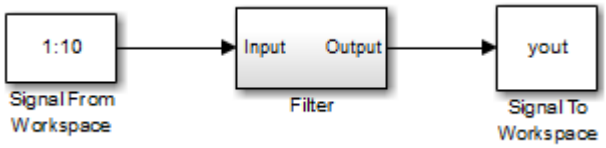
9 Double-click the `Filter` subsystem block in your model to view the filter implementation.

Part 3 — Build a Model to Filter a Signal

In this section of the tutorial, you will filter noise from a signal in your Simulink model.

1 Connect a Signal From Workspace block from the Sources library to the input port of your filter block.

2 Connect a To Workspace block from the Sinks library to the output port of your filter block. Your blocks should now be connected as follows.



3 Open the Signal From Workspace block dialog box and set the **Signal** parameter to `u`. Click **OK** to save your changes and close the dialog box.

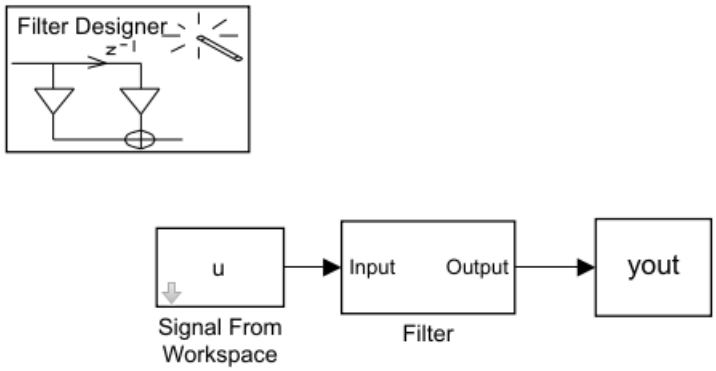
4 In the **Modeling** tab, select **Model Settings**. In the **Solver** pane of the dialog, set the following fields:

- **Stop time** = `length(u)-1`
- **Type** = `Fixed-step`

Click **OK** to save your changes and close the dialog box.

5 Run the model.

6 In the **Debug** tab, select **Information Overlays > Port Data Type**. You can now see that the input to the Filter block is a signal of type `double` and the output of the Filter block has a data type of `sfix16_En11`.



Part 4 — Examine Filtering Results

Now you can listen to and look at the results of the fixed-point filter you designed and implemented.

1. Type

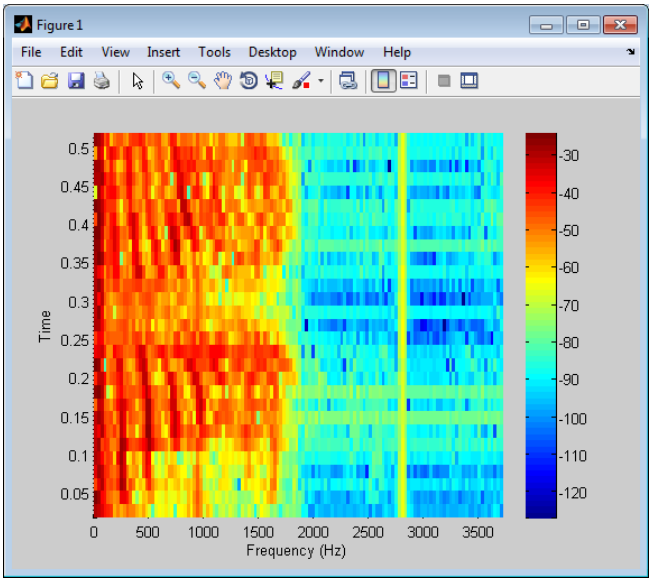
   ```
   soundsc(yout,Fs)
   ```

   at the command line to hear the output of the filter. You should hear a voice say “MATLAB.” The noise portion of the signal should be close to inaudible.

2. Type

   ```
   figure
   spectrogram(yout,256,[],[],Fs);colorbar
   ```

   at the command line.



From the colorbars at the side of the input and output spectrograms, you can see that the noise has been reduced by about 40 dB.

Set the Filter Structure and Number of Filter Sections

The **Current Filter Information** region of filter designer shows the structure and the number of second-order sections in your filter.

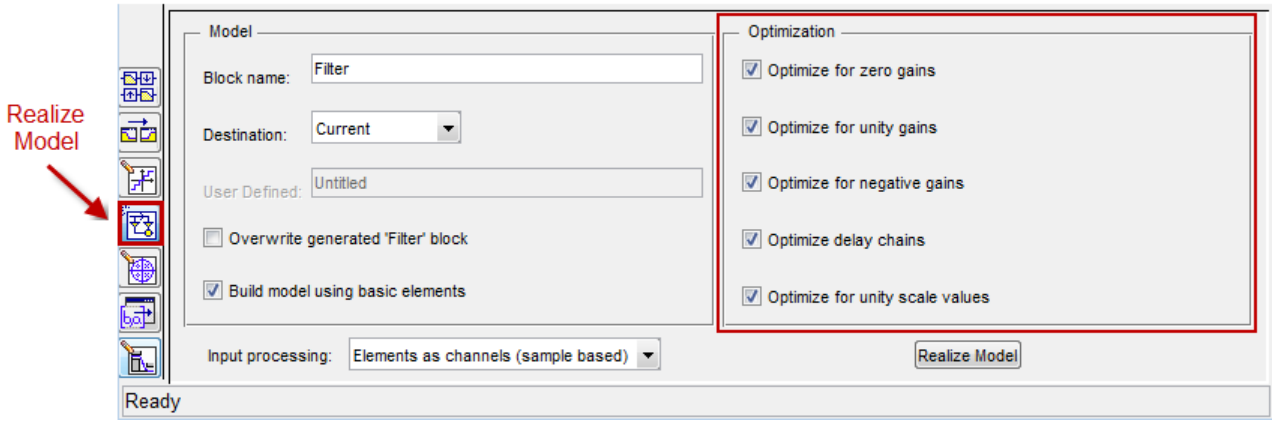
Change the filter structure and number of filter sections of your filter as follows:

- Select **Convert Structure** from the **Edit** menu to open the **Convert Structure** dialog box. For details, see “Converting to a New Structure” in the Signal Processing Toolbox documentation.
- Select **Convert to Second-Order Sections** from the **Edit** menu to open the **Convert to SOS** dialog box. For details, see “Converting to Second-Order Sections” in the Signal Processing Toolbox documentation.

Optimize the Filter Structure

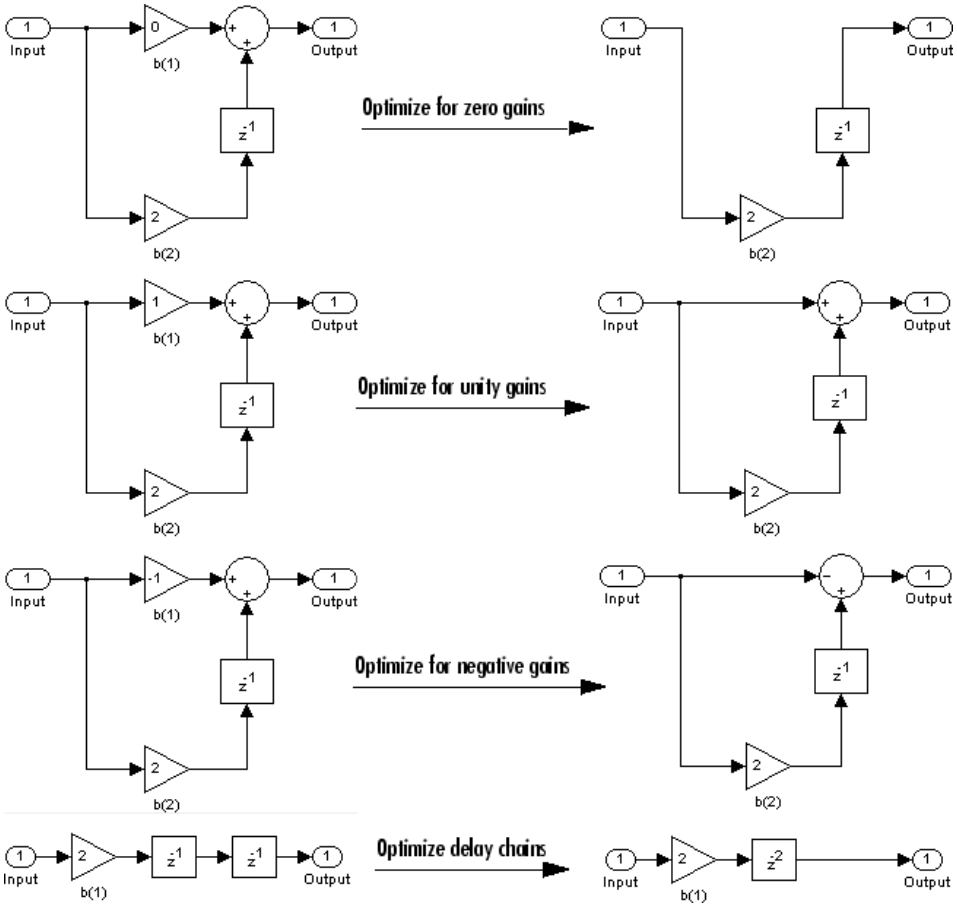
The Filter Realization Wizard can implement a digital filter using either digital filter blocks from the DSP System Toolbox library or by creating a subsystem (Simulink) block that implements the filter using Sum, Gain, and Delay blocks. The following procedure shows you how to optimize the filter implementation:

1. Open the **Realize Model** pane of filter designer by clicking the Realize Model button in the lower-left corner of filter designer.
2. Select the desired optimizations in the **Optimization** region of the **Realize Model** pane. See the following descriptions and illustrations of each optimization option.



- **Optimize for zero gains** — Remove zero-gain paths.
- **Optimize for unity gains** — Substitute gains equal to one with a wire (short circuit).
- **Optimize for negative gains** — Substitute gains equal to `-1` with a wire (short circuit), and change the corresponding sums to subtractions.
- **Optimize delay chains** — Substitute any delay chain made up of *n* unit delays with a single delay by *n*.
- **Optimize for unity scale values** — Remove all scale value multiplications by 1 from the filter structure.

The following diagram illustrates the results of each of these optimizations.


Input
Output
b(1)
b(2)
Optimize for zero gains
Optimize for unity gains
Optimize for negative gains
Optimize delay chains

Digital Filter Implementations

In this section...
“Using Digital Filter Blocks” on page 4-100
“Implement a Lowpass Filter in Simulink” on page 4-100
“Implement a Highpass Filter in Simulink” on page 4-101
“Filter High-Frequency Noise in Simulink” on page 4-102
“Specify Static Filters” on page 4-105
“Specify Time-Varying Filters” on page 4-106
“Specify the SOS Matrix (Biquadratic Filter Coefficients)” on page 4-106

Using Digital Filter Blocks

DSP System Toolbox provides several blocks implementing digital filters, such as Discrete FIR Filter and Biquad Filter.

Use these blocks if you have already performed the design and analysis and know your desired filter coefficients. You can use these blocks to filter single-channel and multichannel signals, and to simulate floating-point and fixed-point filters. Then, you can use the Simulink Coder product to generate highly optimized C code from your filters.

To implement a filter, you must provide the following basic information about the filter:

- The desired filter structure
- The filter coefficients

Note Use the Digital Filter Design block to design and implement a filter. Use the Discrete FIR Filter and Biquad Filter blocks to implement a pre-designed filter. Both methods implement a filter in the same manner and have the same behavior during simulation and code generation.

Implement a Lowpass Filter in Simulink

Use the Discrete FIR Filter block to implement a lowpass filter:

1. Define the lowpass filter coefficients in the MATLAB workspace by typing

   ```
   lopassNum = [-0.0021 -0.0108 -0.0274 -0.0409 -0.0266 0.0374 0.1435 0.2465
   0.2896 0.2465 0.1435 0.0374 -0.0266 -0.0409 -0.0274 -0.0108 -0.0021];
   ```

2. Open Simulink and create a new model file.
3. From the DSP System Toolbox Filtering>Filter Implementations library, click-and-drag a Discrete FIR Filter block into your model.
4. Double-click the Discrete FIR Filter block. Set the block parameters as follows, and then click **OK**:

   - **Coefficient source** = `Dialog parameters`
   - **Filter structure** = `Direct form transposed`

- **Coefficients** = `lopassNum`
- **Input processing** = `Columns as channels (frame based)`
- **Initial states** = `0`

Note that you can provide the filter coefficients in several ways:

- Type in a variable name from the MATLAB workspace, such as `lopassNum`.
- Type in filter design commands from Signal Processing Toolbox software or DSP System Toolbox software, such as `fir1(5, 0.2, 'low')`.
- Type in a vector of the filter coefficient values.

5 Rename your block Digital Filter - Lowpass.

The Discrete FIR Filter block in your model now represents a lowpass filter. In the next topic, “Implement a Highpass Filter in Simulink” on page 4-101, you use a Discrete FIR Filter block to implement a highpass filter. For more information about the Discrete FIR Filter block, see the Discrete FIR Filter block reference page. For more information about designing and implementing a new filter, see “Digital Filter Design Block” on page 4-83.

Implement a Highpass Filter in Simulink

In this topic, you implement a highpass filter using the Discrete FIR Filter block:

1 If the model you created in “Implement a Lowpass Filter in Simulink” on page 4-100 is not open on your desktop, you can open an equivalent model by typing

```
ex_filter_ex1
```

at the MATLAB command prompt.

2 Define the highpass filter coefficients in the MATLAB workspace by typing

```
hipassNum = [-0.0051 0.0181 -0.0069 -0.0283 -0.0061 ...
0.0549 0.0579 -0.0826 -0.2992 0.5946 -0.2992 -0.0826 ...
0.0579 0.0549 -0.0061 -0.0283 -0.0069 0.0181 -0.0051];
```

3 From the DSP System Toolbox Filtering library, and then from the Filter Implementations library, click-and-drag a Discrete FIR Filter block into your model.

4 Double-click the Discrete FIR Filter block. Set the block parameters as follows, and then click **OK**:

- **Coefficient source** = `Dialog parameters`
- **Filter structure** = `Direct form transposed`
- **Coefficients** = `hipassNum`
- **Input processing** = `Columns as channels (frame based)`
- **Initial states** = `0`

You can provide the filter coefficients in several ways:

- Type in a variable name from the MATLAB workspace, such as `hipassNum`.
- Type in filter design commands from Signal Processing Toolbox software or DSP System Toolbox software, such as `fir1(5, 0.2, 'low')`.

- Type in a vector of the filter coefficient values.

5 Rename your block Digital Filter - Highpass.

You have now successfully implemented a highpass filter. In the next topic, “Filter High-Frequency Noise in Simulink” on page 4-102, you use these Discrete FIR Filter blocks to create a model capable of removing high frequency noise from a signal. For more information about designing and implementing a new filter, see “Digital Filter Design Block” on page 4-83.

Filter High-Frequency Noise in Simulink

In the previous topics, you used Discrete FIR Filter blocks to implement lowpass and highpass filters. In this topic, you use these blocks to build a model that removes high frequency noise from a signal. In this model, you use the highpass filter, which is excited using a uniform random signal, to create high-frequency noise. After you add this noise to a sine wave, you use the lowpass filter to filter out the high-frequency noise:

1 If the model you created in “Implement a Highpass Filter in Simulink” on page 4-101 is not open on your desktop, you can open an equivalent model by typing

```
ex_filter_ex2
```

at the MATLAB command prompt.

2 If you have not already done so, define the lowpass and highpass filter coefficients in the MATLAB workspace by typing

```
lopassNum = [-0.0021 -0.0108 -0.0274 -0.0409 -0.0266 ...
0.0374 0.1435 0.2465 0.2896 0.2465 0.1435 0.0374 ...
-0.0266 -0.0409 -0.0274 -0.0108 -0.0021];
hipassNum = [-0.0051 0.0181 -0.0069 -0.0283 -0.0061 ...
0.0549 0.0579 -0.0826 -0.2992 0.5946 -0.2992 -0.0826 ...
0.0579 0.0549 -0.0061 -0.0283 -0.0069 0.0181 -0.0051];
```

3 Click-and-drag the following blocks into your model file.

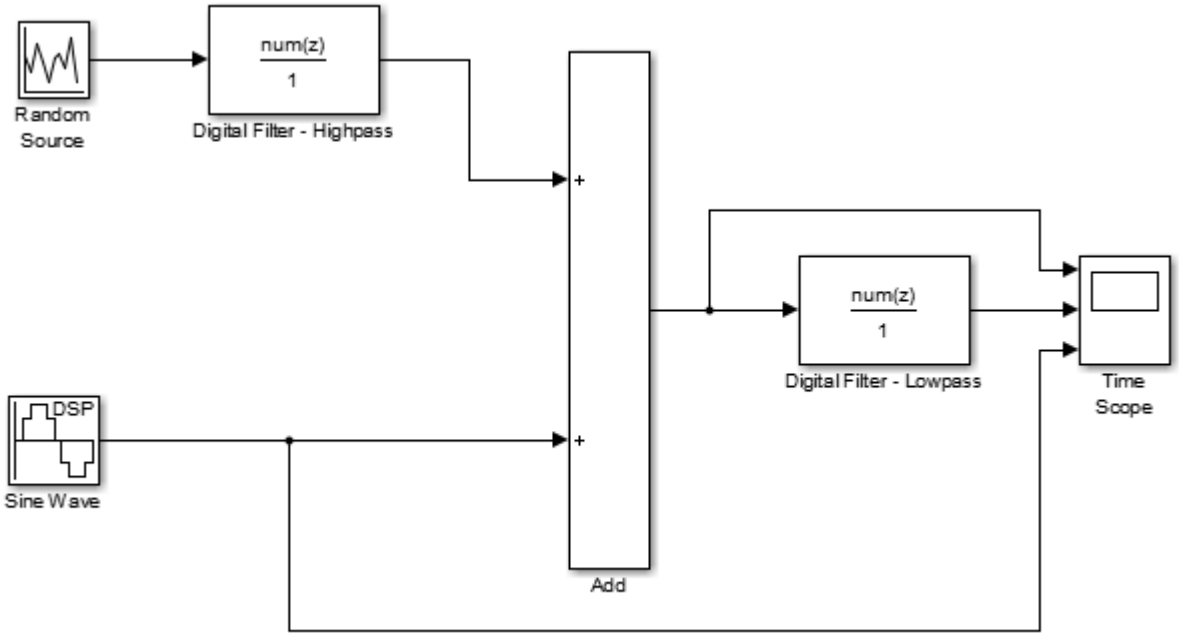
Block	Library	Quantity
Add	Simulink / Math Operations library	1
Random Source	Sources	1
Sine Wave	Sources	1
Time Scope	Sinks	1

4 Set the parameters for the rest of the blocks as indicated in the following table. For any parameters not listed in the table, leave them at their default settings.

Block	Parameter Setting
Add	• **Icon shape** = `rectangular` • **List of signs** = ++

Block	Parameter Setting
Random Source	• **Source type** = `Uniform` • **Minimum** = `0` • **Maximum** = `4` • **Sample mode** = `Discrete` • **Sample time** = `1/1000` • **Samples per frame** = `50`
Sine Wave	• **Frequency (Hz)** = `75` • **Sample time** = `1/1000` • **Samples per frame** = `50`
Time Scope	• **File > Number of Input Ports > 3** • **File > Configuration ...** • Open the **Visuals:Time Domain Options** dialog and set **Time span** = `One frame period`

5 Connect the blocks as shown in the following figure. You may need to resize some of your blocks to accomplish this task.



6 In the **Modeling** tab, click **Model Settings**. The **Configuration Parameters** dialog opens.

7 In the **Solver** pane, set the parameters as follows, and then click **OK**:

- **Start time** = `0`
- **Stop time** = `5`
- **Type** = `Fixed-step`
- **Solver** = `discrete (no continuous states)`

8 In the **Simulation** tab of the model toolstrip, click **Run**.

The model simulation begins and the Scope displays the three input signals.

9 After simulation is complete, select **View > Legend** from the Time Scope menu. The legend appears in the Time Scope window. You can click-and-drag it anywhere on the scope display. To change the channel names, double-click inside the legend and replace the current numbered channel names with the following:

- Add = `Noisy Sine Wave`
- Digital Filter – Lowpass = `Filtered Noisy Sine Wave`
- Sine Wave = `Original Sine Wave`

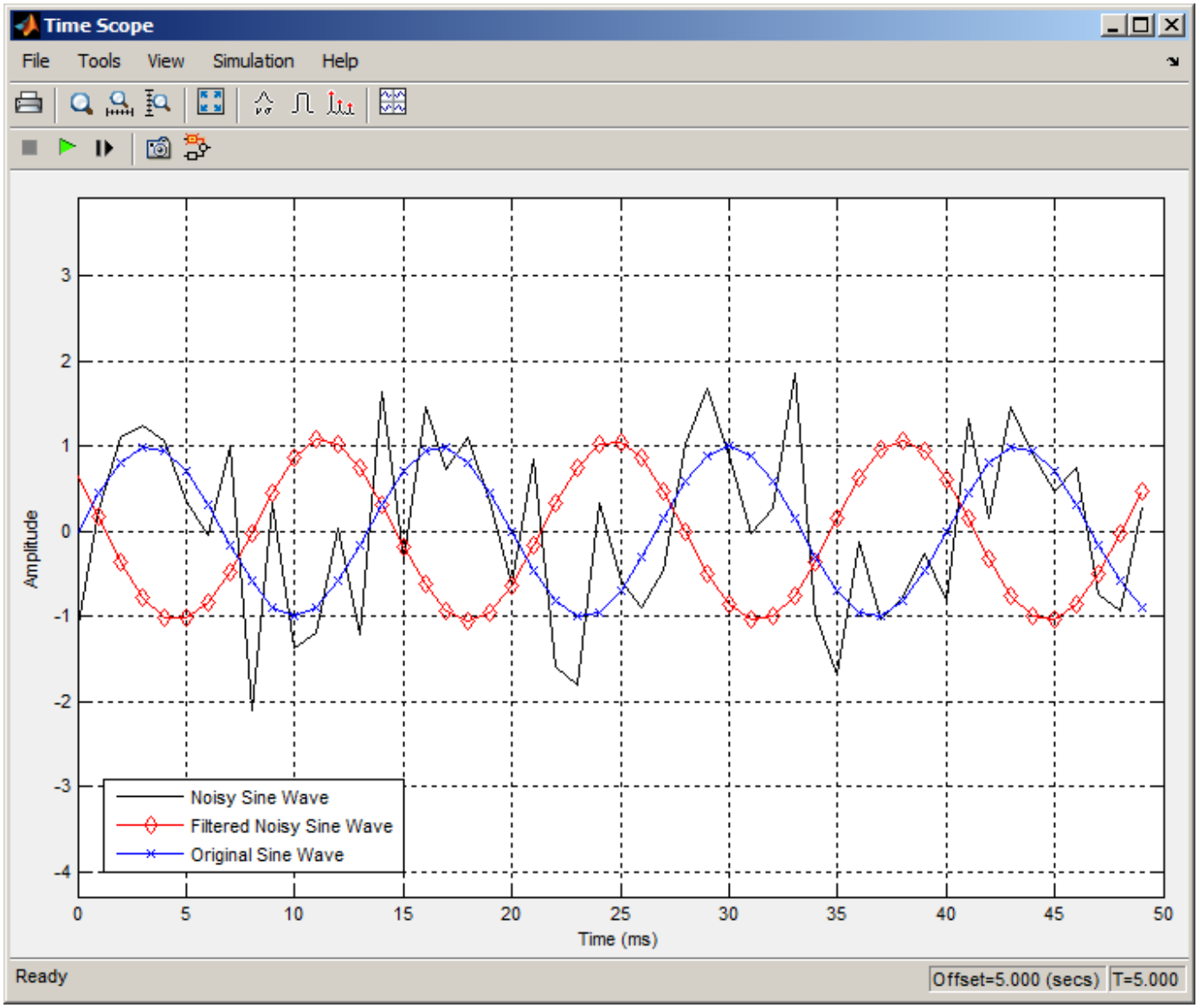
In the next step, you will set the color, style, and marker of each channel.

10 In the Time Scope window, select **View > Line Properties**, and set the following:

Line	Style	Marker	Color
Noisy Sine Wave	**-**	**None**	**Black**
Filtered Noisy Sine Wave	**-**	**diamond**	**Red**
Original Sine Wave	**None**	*	**Blue**

11 The **Time Scope** display should now appear as follows:

You can see that the lowpass filter filters out the high-frequency noise in the noisy sine wave.



You have now used Discrete FIR Filter blocks to build a model that removes high frequency noise from a signal. For more information about designing and implementing a new filter, see “Digital Filter Design Block” on page 4-83.

Specify Static Filters

You can specify a static filter using the Discrete FIR Filter or Biquad Filter block. To do so, set the **Coefficient source** parameter to `Dialog parameters`.

For the Discrete FIR Filter, set the **Coefficients** parameter to a row vector of numerator coefficients. If you set **Filter structure** to `Lattice MA`, the **Coefficients** parameter represents reflection coefficients.

For the Biquad Filter, set the **SOS matrix (Mx6)** to an *M*-by-6 matrix, where *M* is the number of sections in the second-order section filter. Each row of the SOS matrix contains the numerator and

denominator coefficients of the corresponding section in the filter. Set **Scale values** to a scalar or vector of $M+1$ scale values used between SOS stages.

Tuning the Filter Coefficient Values During Simulation

To change the static filter coefficients during simulation, double-click the block, type in the new filter coefficients, and click **OK**. You cannot change the filter order, so you cannot change the number of elements in the matrix of filter coefficients.

Specify Time-Varying Filters

Time-varying filters are filters whose coefficients change with time. You can specify a time-varying filter that changes once per frame. You can filter multiple channels with each filter. However, you cannot apply different filters to each channel; all channels use the same filter.

To specify a time-varying filter using a Biquad Filter block or a Discrete FIR Filter block:

1. Set the **Coefficient source** parameter to `Input port(s)`, which enables extra block input ports for the time-varying filter coefficients.
   - The Discrete FIR Filter block has a `Num` port for the numerator coefficients.
   - The Biquad Filter block has `Num` and `Den` ports rather than a single port for the SOS matrix. Separate ports enable you to use different fraction lengths for numerator and denominator coefficients. The scale values port, `g`, is optional. You can disable the `g` port by setting **Scale values mode** to `Assume all are unity and optimize`.
2. Provide matrices of filter coefficients to the block input ports.
   - For Discrete FIR Filter block, the number of filter taps, N, cannot vary over time. The input coefficients must be in a 1-by-N vector.
   - For Biquad Filter block, the number of filter sections, N, cannot vary over time. The numerator coefficients input, `Num`, must be a 3-by-N matrix. The denominator input coefficients, `Den`, must be a 2-by-N matrix. The scale values input, `g`, must be a 1-by-(N+1) vector.

Specify the SOS Matrix (Biquadratic Filter Coefficients)

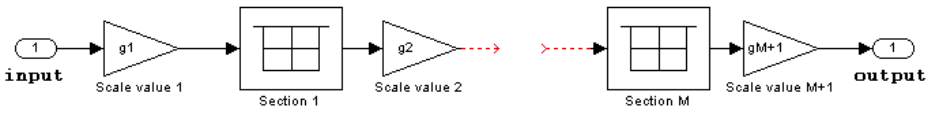
Use the Biquad Filter block to specify a static biquadratic IIR filter (also known as a second-order section or SOS filter). Set the following parameters:

- **Filter structure** — `Direct form I`, or `Direct form I transposed`, or `Direct form II`, or `Direct form II transposed`
- **SOS matrix (Mx6)** M-by-6 SOS matrix

  The SOS matrix is an M-by-6 matrix, where M is the number of sections in the second-order section filter. Each row of the SOS matrix contains the numerator and denominator coefficients (b_{ik} and a_{ik}) of the corresponding section in the filter.
- **Scale values** Scalar or vector of $M+1$ scale values to be used between SOS stages

  If you enter a scalar, the value is used as the gain value before the first section of the second-order filter. The rest of the gain values are set to 1.

If you enter a vector of M+1 values, each value is used for a separate section of the filter. For example, the first element is the first gain value, the second element is the second gain value, and so on.



You can use the `ss2sos` and `tf2sos` functions from Signal Processing Toolbox software to convert a state-space or transfer function description of your filter into the second-order section description used by this block.

$$\begin{bmatrix} b_{01} & b_{11} & b_{21} & a_{01} & a_{11} & a_{21} \\ b_{02} & b_{12} & b_{22} & a_{02} & a_{12} & a_{22} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{0M} & b_{1M} & b_{2M} & a_{0M} & a_{1M} & a_{2M} \end{bmatrix}$$

The block normalizes each row by a_{1i} to ensure a value of 1 for the zero-delay denominator coefficients.

Removing High-Frequency Noise from an ECG Signal

This example shows how to lowpass filter an ECG signal that contains high frequency noise.

Create one period of an ECG signal. The `ecg` function creates an ECG signal of length 500. The `sgolayfilt` function smoothes the ECG signal using a Savitzky-Golay (polynomial) smoothing filter.

```
x = ecg(500).';
y = sgolayfilt(x,0,5);
[M,N] = size(y);
```

Initialize the time scope to view the noisy signal and the filtered signal.

```
Fs = 1000;
TS = dsp.TimeScope('SampleRate',Fs,...
                       'TimeSpan',1.5,...
                       'YLimits',[-1 1],...
                       'ShowGrid',true,...
                       'NumInputPorts',2,...
                       'LayoutDimensions',[2 1],...
                       'Title','Noisy and Filtered Signals');
```

Design a minimum-order lowpass filter with a passband edge frequency of 200 Hz and a stopband edge frequency of 400 Hz. The desired amplitude of the frequency response and the weights are specified in `A` and `D` vectors, respectively. Pass these specification vectors to the `firgr` function to design the filter coefficients. Pass these designed coefficients to the `dsp.FIRFilter` object.

```
Fpass  = 200;
Fstop = 400;
Dpass = 0.05;
Dstop = 0.0001;
F     = [0 Fpass Fstop Fs/2]/(Fs/2);
A     = [1 1 0 0];
D     = [Dpass Dstop];
b = firgr('minorder',F,A,D);
LP = dsp.FIRFilter('Numerator',b);
```

Design a minimum-order highpass filter with a stopband edge frequency of 200 Hz and a passband edge frequency of 400 Hz. Design the filter using the `firgr` function. Pass these designed coefficients to the `dsp.FIRFilter` object.

```
Fstop = 200;
Fpass = 400;
Dstop = 0.0001;
Dpass = 0.05;
F = [0 Fstop Fpass Fs/2]/(Fs/2); % Frequency vector
A = [0 0 1 1]; % Amplitude vector
D = [Dstop Dpass];   % Deviation (ripple) vector
b  = firgr('minord',F,A,D);
HP = dsp.FIRFilter('Numerator',b);
```

The noisy signal contains the smoothed ECG signal along with high frequency noise. The signal is filtered using a lowpass filter. View the noisy signal and the filtered signal using the time scope.

```
tic;
while toc < 30
    x = .1 * randn(M,N);
```

```
    highFreqNoise = HP(x);
    noisySignal = y + highFreqNoise;
    filteredSignal = LP(noisySignal);
    TS(noisySignal,filteredSignal);
end

% Finalize
release(TS)
```

See Also

Functions
`firgr`

Objects
`dsp.FIRFilter` | `dsp.TimeScope`

More About

- “Remove High-Frequency Noise from Gyroscope Data” on page 26-24

Adaptive Filters

Learn how to design and implement adaptive filters.

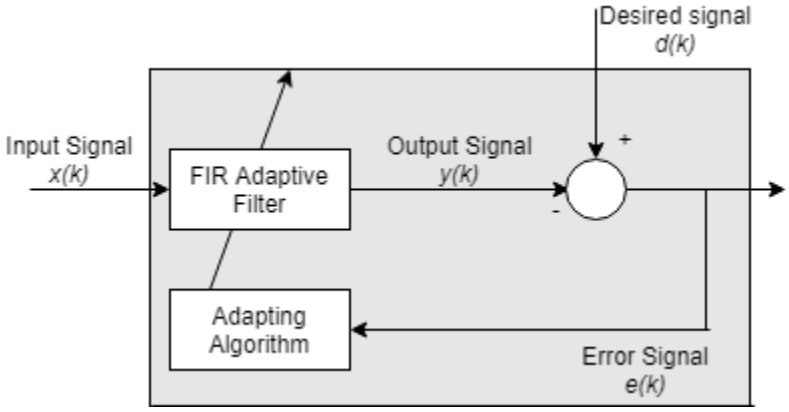
- “Overview of Adaptive Filters and Applications” on page 5-2
- “System Identification of FIR Filter Using LMS Algorithm” on page 5-9
- “System Identification of FIR Filter Using Normalized LMS Algorithm” on page 5-17
- “Compare Convergence Performance Between LMS Algorithm and Normalized LMS Algorithm” on page 5-20
- “Noise Cancellation Using Sign-Data LMS Algorithm” on page 5-22
- “Compare RLS and LMS Adaptive Filter Algorithms” on page 5-26
- “Inverse System Identification Using RLS Algorithm” on page 5-29
- “Signal Enhancement Using LMS and NLMS Algorithms” on page 5-34
- “Noise Cancellation in Simulink Using Normalized LMS Adaptive Filter” on page 5-41

Overview of Adaptive Filters and Applications

In this section...
“Adaptive Filters in DSP System Toolbox” on page 5-2
“Choosing an Adaptive Filter” on page 5-4
“Mean Squared Error Performance” on page 5-5
“Common Applications” on page 5-5

Adaptive filters are digital filters whose coefficients change with an objective to make the filter converge to an optimal state. The optimization criterion is a cost function, which is most commonly the mean square of the error signal between the output of the adaptive filter and the desired signal. As the filter adapts its coefficients, the mean square error (MSE) converges to its minimal value. At this state, the filter is adapted and the coefficients have converged to a solution. The filter output, *y(k)*, is then said to match very closely to the desired signal, *d(k)*. When you change the input data characteristics, sometimes called *filter environment*, the filter adapts to the new environment by generating a new set of coefficients for the new data.

General Adaptive Filter Algorithm



Adaptive Filters in DSP System Toolbox

Least Mean Squares (LMS) Based FIR Adaptive Filters

Adaptive Filter Object	Adapting Algorithm
`dsp.BlockLMSFilter`	Block LMS FIR adaptive filter algorithm
`dsp.FilteredXLMSFilter`	Filtered-x LMS FIR adaptive filter algorithm
`dsp.LMSFilter`	LMS FIR adaptive filter algorithm Normalized LMS FIR adaptive filter algorithm Sign-data LMS FIR adaptive filter algorithm Sign-error LMS FIR adaptive filter algorithm Sign-sign LMS FIR adaptive filter algorithm

Adaptive Filter Block	Adapting Algorithm
Block LMS Filter	Block LMS FIR adaptive filter algorithm
Fast Block LMS Filter	Block LMS FIR adaptive filter algorithm in frequency domain
LMS Filter	LMS FIR adaptive filter algorithm Normalized LMS FIR adaptive filter algorithm Sign-data LMS FIR adaptive filter algorithm Sign-error LMS FIR adaptive filter algorithm Sign-sign LMS FIR adaptive filter algorithm
LMS Update	LMS FIR weight update algorithm Normalized LMS FIR weight update algorithm Sign-data LMS FIR weight update algorithm Sign-error LMS FIR weight update algorithm Sign-sign LMS FIR weight update algorithm

Recursive Least Squares (RLS) Based FIR Adaptive Filters

Adaptive Filter Object	Adapting Algorithm
`dsp.FastTransversalFilter`	Fast transversal least-squares adaptation algorithm Sliding window FTF adaptation algorithm
`dsp.RLSFilter`	QR-decomposition RLS adaptation algorithm Householder RLS adaptation algorithm Householder SWRLS adaptation algorithm Recursive-least squares (RLS) adaptation algorithm Sliding window (SW) RLS adaptation algorithm

Adaptive Filter Block	Adapting Algorithm
RLS Filter	Exponentially weighted recursive least-squares (RLS) algorithm

Affine Projection (AP) FIR Adaptive Filters

Adaptive Filter Object	Adapting Algorithm
`dsp.AffineProjectionFilter`	Affine projection algorithm that uses direct matrix inversion Affine projection algorithm that uses recursive matrix updating Block affine projection adaptation algorithm

FIR Adaptive Filters in the Frequency Domain (FD)

Adaptive Filter Object	Adapting Algorithm
`dsp.FrequencyDomainAdaptiveFilter`	Constrained frequency domain adaptation algorithm Unconstrained frequency domain adaptation algorithm Partitioned and constrained frequency domain adaptation algorithm Partitioned and unconstrained frequency domain adaptation algorithm

Adaptive Filter Block	Adapting Algorithm
Frequency-Domain Adaptive Filter	Constrained frequency domain adaptation algorithm Unconstrained frequency domain adaptation algorithm Partitioned and constrained frequency domain adaptation algorithm Partitioned and unconstrained frequency domain adaptation algorithm

Lattice-Based (L) FIR Adaptive Filters

Adaptive Filter Object	Adapting Algorithm
`dsp.AdaptiveLatticeFilter`	Gradient adaptive lattice filter adaptation algorithm Least squares lattice adaptation algorithm QR decomposition RLS adaptation algorithm

For more information on these algorithms, refer to the algorithm section of the respective reference pages. Full descriptions of the theory appear in the adaptive filter references [1] and [2].

Choosing an Adaptive Filter

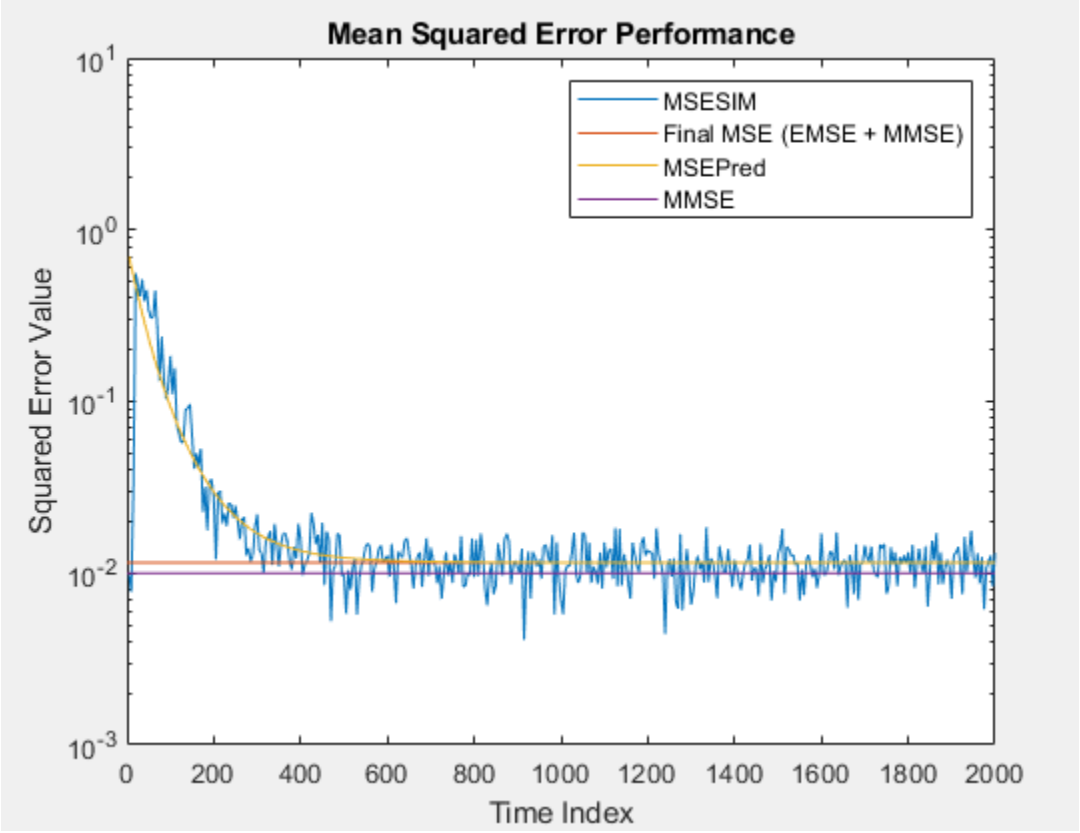
At steady state when the filter has adapted, the error between the filter output and the desired signal is minimal, not zero. This error is known as the steady state error. The speed with which the filter converges to the optimal state, known as the convergence speed, depends on multiple factors such nature of the input signal, choice of the adaptive filter algorithm, and step size of the algorithm. The choice of the filter algorithm usually depends factors such as convergence performance required for the application, computational complexity of the algorithm, filter stability in the environment, and any other constraints.

LMS algorithm is simple to implement, but has stability issues. The normalized version of the LMS algorithm comes with improved convergence speed, more stability, but has increased computational complexity. For an example that compares the two, see “Compare Convergence Performance Between LMS Algorithm and Normalized LMS Algorithm” on page 5-20. RLS algorithms are highly stable, do very well in time-varying environments, but are computationally more complex than the LMS algorithms. For a comparison, see “Compare RLS and LMS Adaptive Filter Algorithms” on page 5-26. Affine projection filters do well when the input is colored and have a very good convergence performance. Adaptive lattice filters provide good convergence but come with increased

computational cost. The choice of the algorithm depends on the environment and the specifics of the application.

Mean Squared Error Performance

Minimizing the mean square of the error signal between the output of the adaptive filter and the desired signal is the most common optimization criterion for adaptive filters. The actual MSE (MSESIM) of the adaptive filter you are implementing can be determined using the `msesim` function. The trajectory of this MSE is expected to follow that of the predicted MSE (MSEPred), which is computed using the `msepred` function. The minimum mean square error (MMSE) is estimated by the `msepred` function using a Wiener filter. The Weiner filter minimizes the mean squared error between the desired signal and the input signal filtered by the Wiener filter. A large value of the mean squared error indicates that the adaptive filter cannot accurately track the desired signal. The minimal value of the mean squared error ensures that the adaptive filter is optimal. The excess mean square error (EMSE), determined by the `msepred` function, is the difference between the MSE introduced by the adaptive filters and the MMSE produced by the corresponding Wiener filter. The final MSE shown below is the sum of EMSE and MMSE, and equals the predicted MSE after convergence.



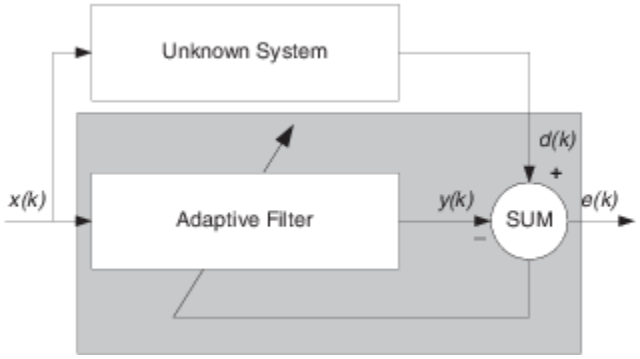
Common Applications

System Identification -- Using an Adaptive Filter to Identify an Unknown System

One common adaptive filter application is to use adaptive filters to identify an unknown system, such as the response of an unknown communications channel or the frequency response of an auditorium,

to pick fairly divergent applications. Other applications include echo cancellation and channel identification.

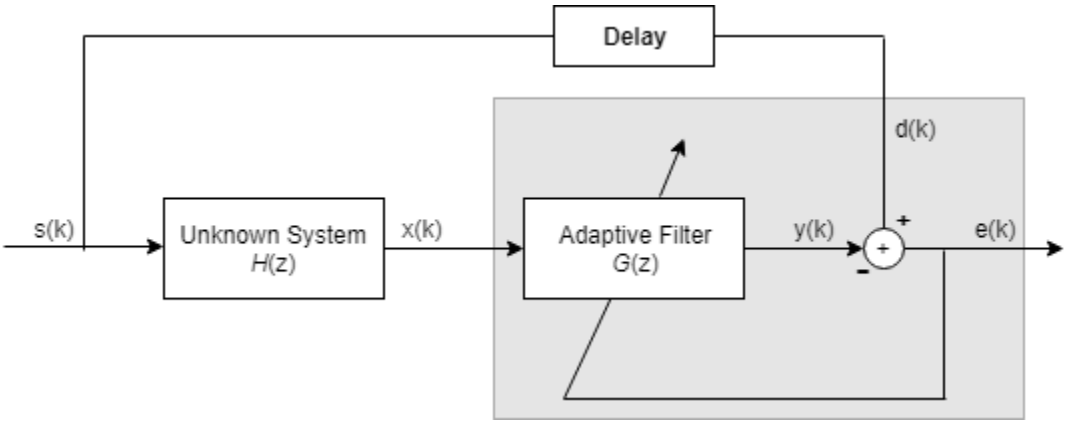
In the figure, the unknown system is placed in parallel with the adaptive filter. This layout represents just one of many possible structures. The shaded area contains the adaptive filter system.



Clearly, when *e(k)* is very small, the adaptive filter response is close to the response of the unknown system. In this case, the same input feeds both the adaptive filter and the unknown. If, for example, the unknown system is a modem, the input often represents white noise, and is a part of the sound you hear from your modem when you log in to your Internet service provider.

Inverse System Identification -- Determining an Inverse Response to an Unknown System

By placing the unknown system in series with your adaptive filter, your filter adapts to become the inverse of the unknown system as *e(k)* becomes very small. As shown in the figure, the process requires a delay inserted in the desired signal *d(k)* path to keep the data at the summation synchronized. Adding the delay keeps the system causal.



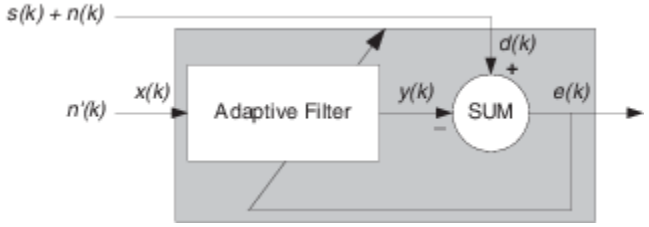
Including the delay to account for the delay caused by the unknown system prevents this condition.

Plain old telephone systems (POTS) commonly use inverse system identification to compensate for the copper transmission medium. When you send data or voice over telephone lines, the copper wires behave like a filter, having a response that rolls off at higher frequencies (or data rates) and having other anomalies as well.

Adding an adaptive filter that has a response that is the inverse of the wire response, and configuring the filter to adapt in real time, lets the filter compensate for the rolloff and anomalies, increasing the available frequency output range and data rate for the telephone system.

Noise or Interference Cancellation –– Using an Adaptive Filter to Remove Noise from an Unknown System

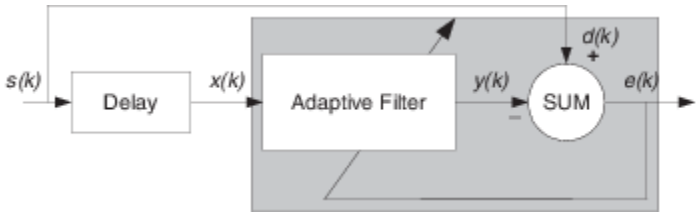
In noise cancellation, adaptive filters let you remove noise from a signal in real time. Here, the desired signal, the one to clean up, combines noise and desired information. To remove the noise, feed a signal *n'(k)* to the adaptive filter that is correlated to the noise to be removed from the desired signal.



So long as the input noise to the filter remains correlated to the unwanted noise accompanying the desired signal, the adaptive filter adjusts its coefficients to reduce the value of the difference between *y(k)* and *d(k),* removing the noise and resulting in a clean signal in *e(k)*. Notice that in this application, the error signal actually converges to the input data signal, rather than converging to zero.

Prediction –– Predicting Future Values of a Periodic Signal

Predicting signals requires that you make some key assumptions. Assume that the signal is either steady or slowly varying over time, and periodic over time as well.



Accepting these assumptions, the adaptive filter must predict the future values of the desired signal based on past values. When *s(k)* is periodic and the filter is long enough to remember previous values, this structure with the delay in the input signal, can perform the prediction. You might use this structure to remove a periodic signal from stochastic noise signals.

Finally, notice that most systems of interest contain elements of more than one of the four adaptive filter structures. Carefully reviewing the real structure may be required to determine what the adaptive filter is adapting to.

Also, for clarity in the figures, the analog-to-digital (A/D) and digital-to-analog (D/A) components do not appear. Since the adaptive filters are assumed to be digital in nature, and many of the problems produce analog data, converting the input signals to and from the analog domain is probably necessary.

References

[1] Hayes, Monson H., *Statistical Digital Signal Processing and Modeling*. Hoboken, NJ: John Wiley & Sons, 1996, pp.493–552.

[2] Haykin, Simon, *Adaptive Filter Theory*. Upper Saddle River, NJ: Prentice-Hall, Inc., 1996.

System Identification of FIR Filter Using LMS Algorithm

System identification is the process of identifying the coefficients of an unknown system using an adaptive filter. The general overview of the process is shown in “System Identification –– Using an Adaptive Filter to Identify an Unknown System” on page 5-5. The main components involved are:

- The adaptive filter algorithm. In this example, set the `Method` property of `dsp.LMSFilter` to `'LMS'` to choose the LMS adaptive filter algorithm.
- An unknown system or process to adapt to. In this example, the filter designed by `fircband` is the unknown system.
- Appropriate input data to exercise the adaptation process. For the generic LMS model, these are the desired signal $d(k)$ and the input signal $x(k)$.

The objective of the adaptive filter is to minimize the error signal between the output of the adaptive filter $y(k)$ and the output of the unknown system (or the system to be identified) $d(k)$. Once the error signal is minimized, the adapted filter resembles the unknown system. The coefficients of both the filters match closely.

Note: If you are using R2016a or an earlier release, replace each call to the object with the equivalent step syntax. For example, `obj(x)` becomes `step(obj,x)`.

Unknown System

Create a `dsp.FIRFilter` object that represents the system to be identified. Use the `fircband` function to design the filter coefficients. The designed filter is a lowpass filter constrained to 0.2 ripple in the stopband.

```
filt = dsp.FIRFilter;
filt.Numerator = fircband(12,[0 0.4 0.5 1],[1 1 0 0],[1 0.2],...
{'w' 'c'});
```

Pass the signal `x` to the FIR filter. The desired signal `d` is the sum of the output of the unknown system (FIR filter) and an additive noise signal `n`.

```
x = 0.1*randn(250,1);
n = 0.01*randn(250,1);
d = filt(x) + n;
```

Adaptive Filter

With the unknown filter designed and the desired signal in place, create and apply the adaptive LMS filter object to identify the unknown filter.

Preparing the adaptive filter object requires starting values for estimates of the filter coefficients and the LMS step size (`mu`). You can start with some set of nonzero values as estimates for the filter coefficients. This example uses zeros for the 13 initial filter weights. Set the `InitialConditions` property of `dsp.LMSFilter` to the desired initial values of the filter weights. For the step size, 0.8 is a good compromise between being large enough to converge well within 250 iterations (250 input sample points) and small enough to create an accurate estimate of the unknown filter.

Create a `dsp.LMSFilter` object to represent an adaptive filter that uses the LMS adaptive algorithm. Set the length of the adaptive filter to 13 taps and the step size to 0.8.

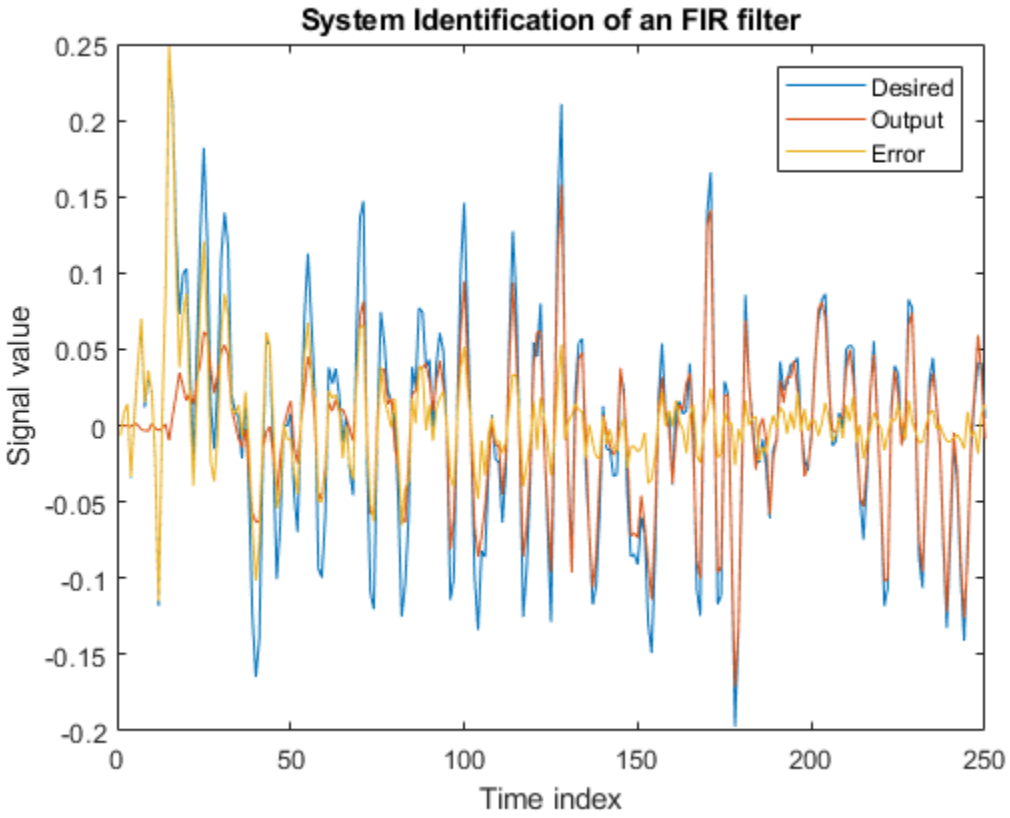
```
mu = 0.8;
lms = dsp.LMSFilter(13,'StepSize',mu)
```

```
lms =
  dsp.LMSFilter with properties:

                  Method: 'LMS'
                  Length: 13
          StepSizeSource: 'Property'
                StepSize: 0.8000
            LeakageFactor: 1
        InitialConditions: 0
           AdaptInputPort: false
    WeightsResetInputPort: false
           WeightsOutput: 'Last'

  Show all properties
```

Pass the primary input signal `x` and the desired signal `d` to the LMS filter. Run the adaptive filter to determine the unknown system. The output `y` of the adaptive filter is the signal converged to the desired signal *d* thereby minimizing the error `e` between the two signals.

Plot the results. The output signal does not match the desired signal as expected, making the error between the two nontrivial.

```
[y,e,w] = lms(x,d);
plot(1:250, [d,y,e])
title('System Identification of an FIR filter')
legend('Desired','Output','Error')
xlabel('Time index')
ylabel('Signal value')
```

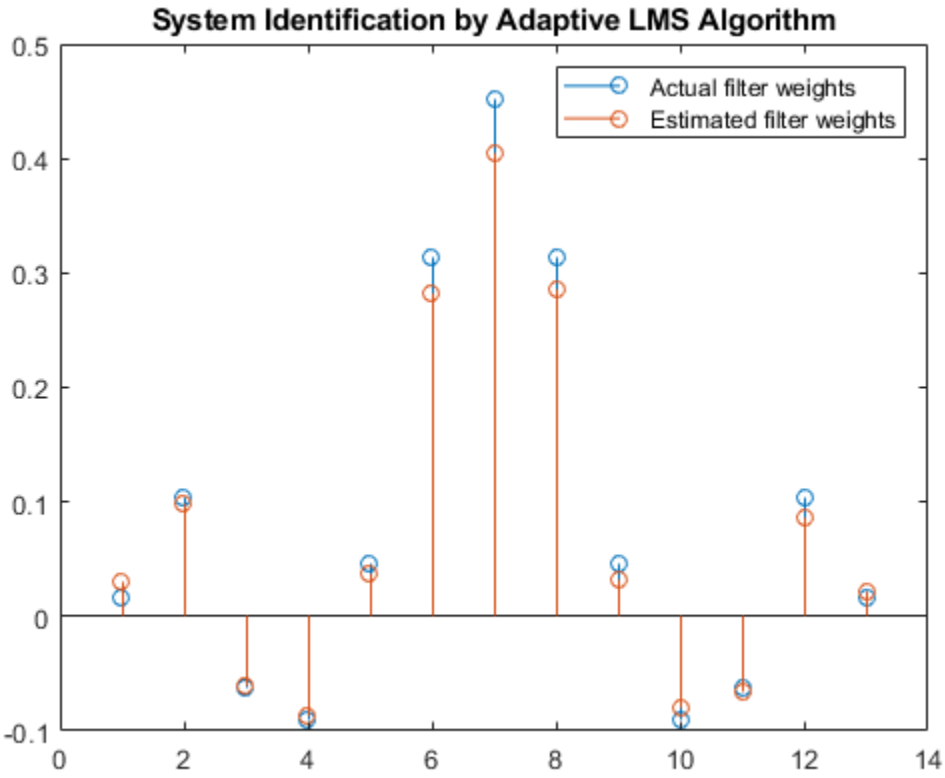


Compare the Weights

The weights vector `w` represents the coefficients of the LMS filter that is adapted to resemble the unknown system (FIR filter). To confirm the convergence, compare the numerator of the FIR filter and the estimated weights of the adaptive filter.

The estimated filter weights do not closely match the actual filter weights, confirming the results seen in the previous signal plot.

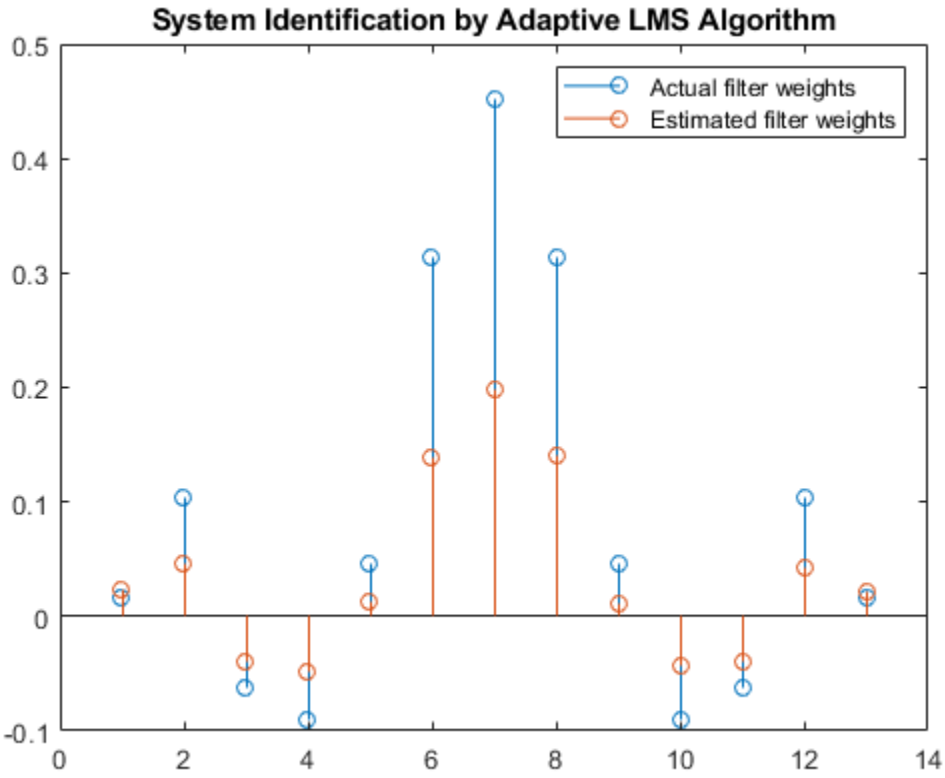
```
stem([(filt.Numerator).' w])
title('System Identification by Adaptive LMS Algorithm')
legend('Actual filter weights','Estimated filter weights',...
       'Location','NorthEast')
```



Changing the Step Size

As an experiment, change the step size to 0.2. Repeating the example with `mu = 0.2` results in the following stem plot. The filters do not converge, and the estimated weights are not good approxmations of the actual weights.

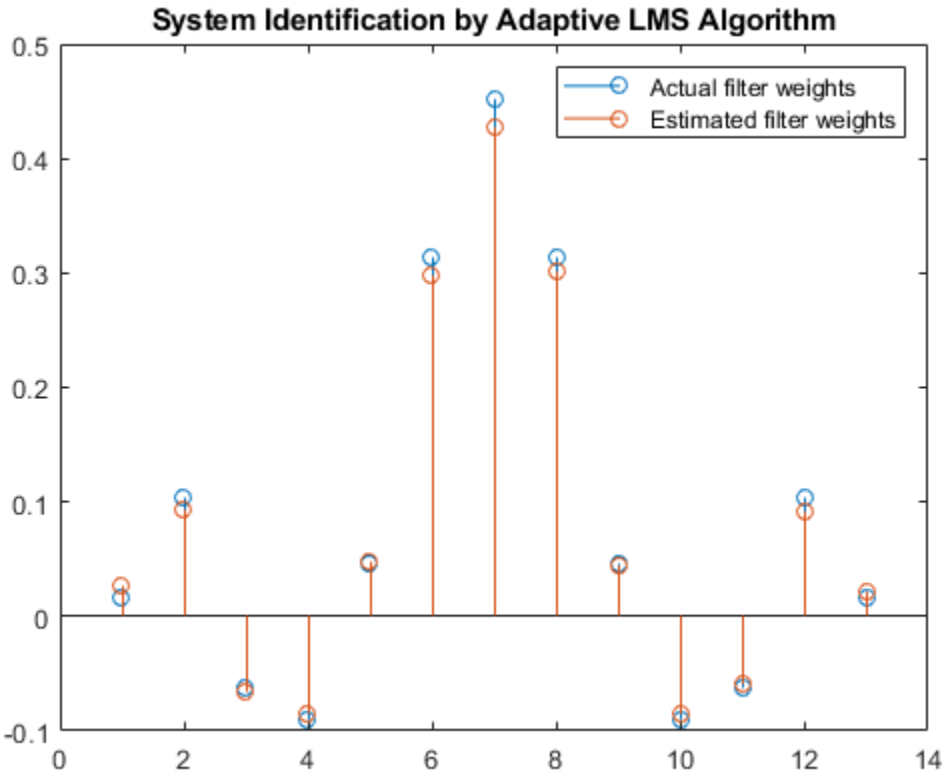
```
mu = 0.2;
lms = dsp.LMSFilter(13,'StepSize',mu);
[~,~,w] = lms(x,d);
stem([(filt.Numerator).' w])
title('System Identification by Adaptive LMS Algorithm')
legend('Actual filter weights','Estimated filter weights',...
       'Location','NorthEast')
```



Increase the Number of Data Samples

Increase the frame size of the desired signal. Even though this increases the computation involved, the LMS algorithm now has more data that can be used for adaptation. With 1000 samples of signal data and a step size of 0.2, the coefficients are aligned closer than before, indicating an improved convergence.

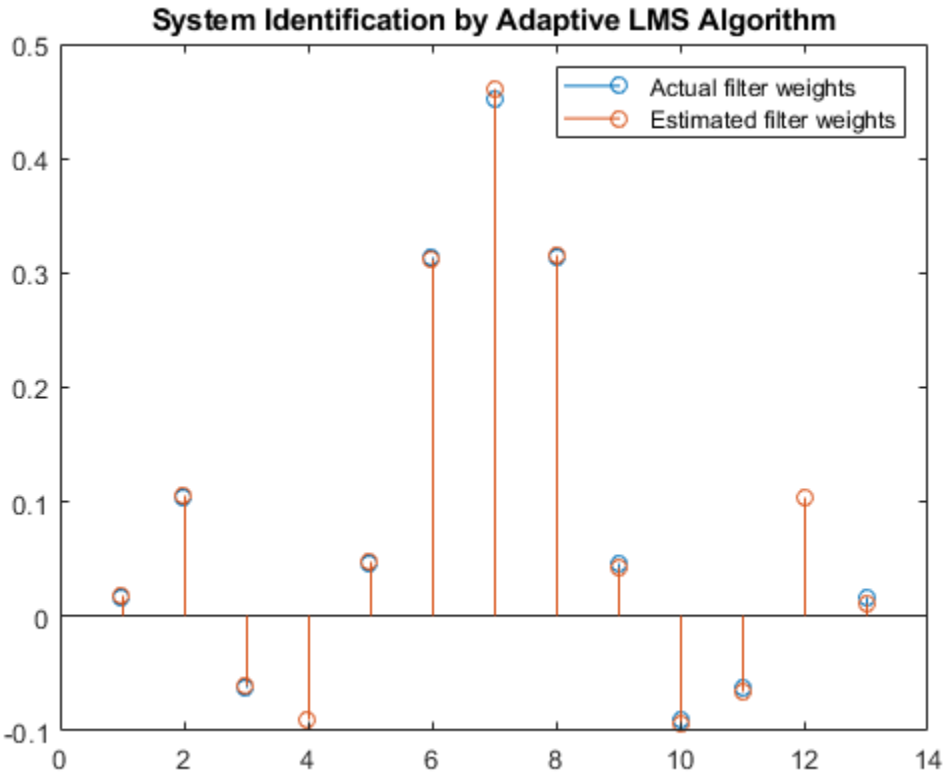
```
release(filt);
x = 0.1*randn(1000,1);
n = 0.01*randn(1000,1);
d = filt(x) + n;
[y,e,w] = lms(x,d);
stem([(filt.Numerator).' w])
title('System Identification by Adaptive LMS Algorithm')
legend('Actual filter weights','Estimated filter weights',...
       'Location','NorthEast')
```



Increase the number of data samples further by inputting the data through iterations. Run the algorithm on 4000 samples of data, passed to the LMS algorithm in batches of 1000 samples over 4 iterations.

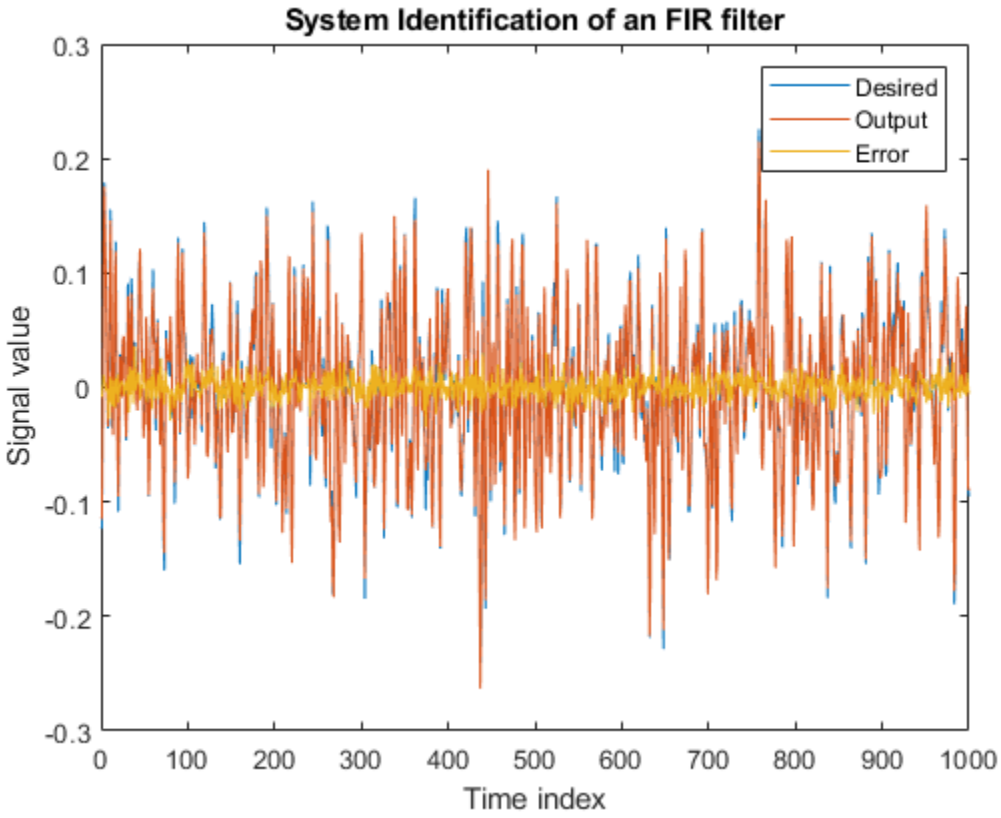
Compare the filter weights. The weights of the LMS filter match the weights of the FIR filter very closely, indicating a good convergence.

```
release(filt);
n = 0.01*randn(1000,1);
for index = 1:4
  x = 0.1*randn(1000,1);
  d = filt(x) + n;
  [y,e,w] = lms(x,d);
end
stem([(filt.Numerator).' w])
title('System Identification by Adaptive LMS Algorithm')
legend('Actual filter weights','Estimated filter weights',...
       'Location','NorthEast')
```



The output signal matches the desired signal very closely, making the error between the two close to zero.

```
plot(1:1000, [d,y,e])
title('System Identification of an FIR filter')
legend('Desired','Output','Error')
xlabel('Time index')
ylabel('Signal value')
```



See Also

Objects
`dsp.LMSFilter`

More About

- “System Identification of FIR Filter Using Normalized LMS Algorithm” on page 5-17
- “Compare Convergence Performance Between LMS Algorithm and Normalized LMS Algorithm” on page 5-20

References

[1] Hayes, Monson H., *Statistical Digital Signal Processing and Modeling*. Hoboken, NJ: John Wiley & Sons, 1996, pp.493–552.

[2] Haykin, Simon, *Adaptive Filter Theory*. Upper Saddle River, NJ: Prentice-Hall, Inc., 1996.

System Identification of FIR Filter Using Normalized LMS Algorithm

To improve the convergence performance of the LMS algorithm, the normalized variant (NLMS) uses an adaptive step size based on the signal power. As the input signal power changes, the algorithm calculates the input power and adjusts the step size to maintain an appropriate value. The step size changes with time, and as a result, the normalized algorithm converges faster with fewer samples in many cases. For input signals that change slowly over time, the normalized LMS algorithm can be a more efficient LMS approach.

For an example using the LMS approach, see “System Identification of FIR Filter Using LMS Algorithm” on page 5-9.

Note: If you are using R2016a or an earlier release, replace each call to the object with the equivalent step syntax. For example, `obj(x)` becomes `step(obj,x)`.

Unknown System

Create a `dsp.FIRFilter` object that represents the system to be identified. Use the `fircband` function to design the filter coefficients. The designed filter is a lowpass filter constrained to 0.2 ripple in the stopband.

```
filt = dsp.FIRFilter;
filt.Numerator = fircband(12,[0 0.4 0.5 1],[1 1 0 0],[1 0.2],...
{'w' 'c'});
```

Pass the signal `x` to the FIR filter. The desired signal `d` is the sum of the output of the unknown system (FIR filter) and an additive noise signal `n`.

```
x = 0.1*randn(1000,1);
n = 0.001*randn(1000,1);
d = filt(x) + n;
```

Adaptive Filter

To use the normalized LMS algorithm variation, set the `Method` property on the `dsp.LMSFilter` to `'Normalized LMS'`. Set the length of the adaptive filter to 13 taps and the step size to 0.2.

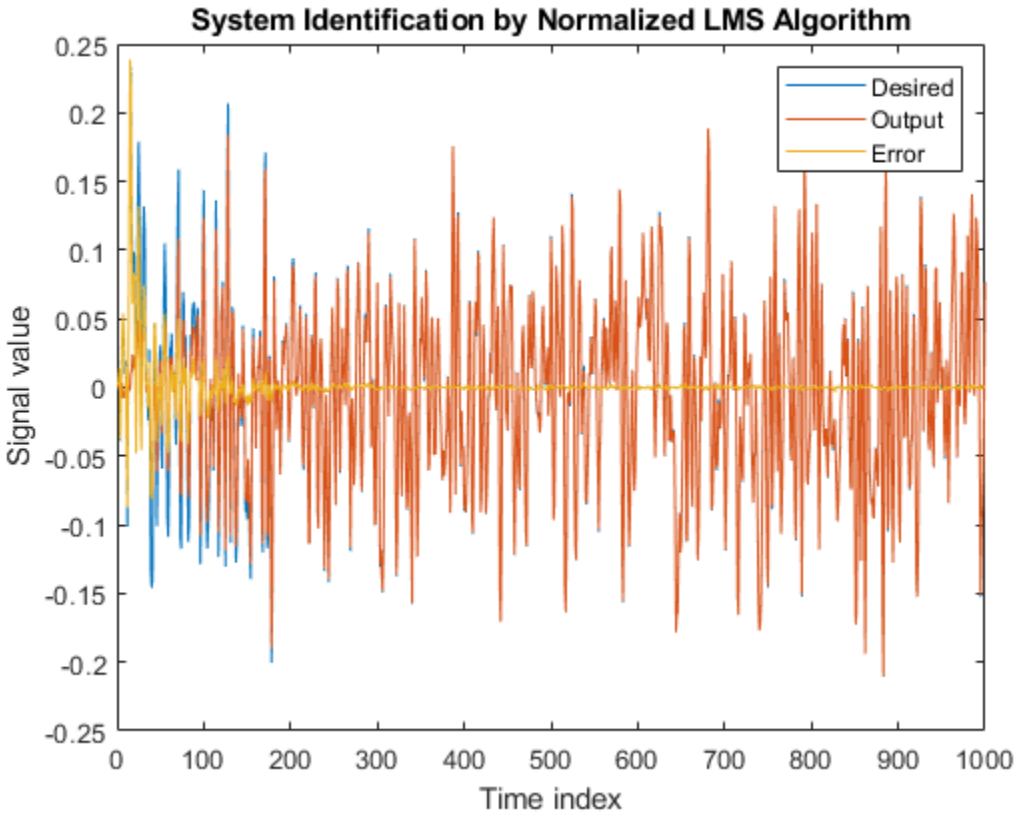
```
mu = 0.2;
lms = dsp.LMSFilter(13,'StepSize',mu,'Method',...
   'Normalized LMS');
```

Pass the primary input signal `x` and the desired signal `d` to the LMS filter.

```
[y,e,w] = lms(x,d);
```

The output `y` of the adaptive filter is the signal converged to the desired signal *d* thereby minimizing the error `e` between the two signals.

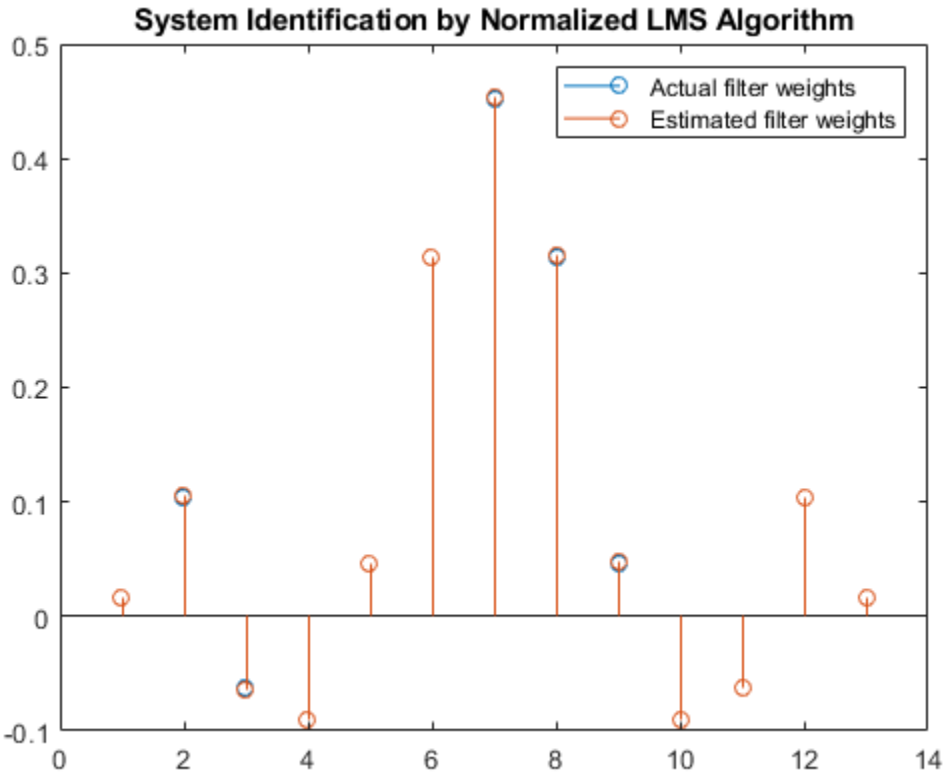
```
plot(1:1000, [d,y,e])
title('System Identification by Normalized LMS Algorithm')
legend('Desired','Output','Error')
xlabel('Time index')
ylabel('Signal value')
```



Compare the Adapted Filter to the Unknown System

The weights vector w represents the coefficients of the LMS filter that is adapted to resemble the unknown system (FIR filter). To confirm the convergence, compare the numerator of the FIR filter and the estimated weights of the adaptive filter.

```
stem([(filt.Numerator).' w])
title('System Identification by Normalized LMS Algorithm')
legend('Actual filter weights','Estimated filter weights',...
       'Location','NorthEast')
```



See Also

Objects
`dsp.LMSFilter`

More About

- “System Identification of FIR Filter Using LMS Algorithm” on page 5-9
- “Compare Convergence Performance Between LMS Algorithm and Normalized LMS Algorithm” on page 5-20

References

[1] Hayes, Monson H., *Statistical Digital Signal Processing and Modeling*. Hoboken, NJ: John Wiley & Sons, 1996, pp.493–552.

[2] Haykin, Simon, *Adaptive Filter Theory*. Upper Saddle River, NJ: Prentice-Hall, Inc., 1996.

Compare Convergence Performance Between LMS Algorithm and Normalized LMS Algorithm

An adaptive filter adapts its filter coefficients to match the coefficients of an unknown system. The objective is to minimize the error signal between the output of the unknown system and the output of the adaptive filter. When these two outputs converge and match closely for the same input, the coefficients are said to match closely. The adaptive filter at this state resembles the unknown system. This example compares the rate at which this convergence happens for the normalized LMS (NLMS) algorithm and the LMS algorithm with no normalization.

Unknown System

Create a `dsp.FIRFilter` that represents the unknown system. Pass the signal `x` as an input to the unknown system. The desired signal `d` is the sum of the output of the unknown system (FIR filter) and an additive noise signal `n`.

```
filt = dsp.FIRFilter;
filt.Numerator = fircband(12,[0 0.4 0.5 1],[1 1 0 0],[1 0.2],...
{'w' 'c'});
x = 0.1*randn(1000,1);
n = 0.001*randn(1000,1);
d = filt(x) + n;
```

Adaptive Filter

Create two `dsp.LMSFilter` objects, with one set to the LMS algorithm, and the other set to the normalized LMS algorithm. Choose an adaptation step size of 0.2 and set the length of the adaptive filter to 13 taps.

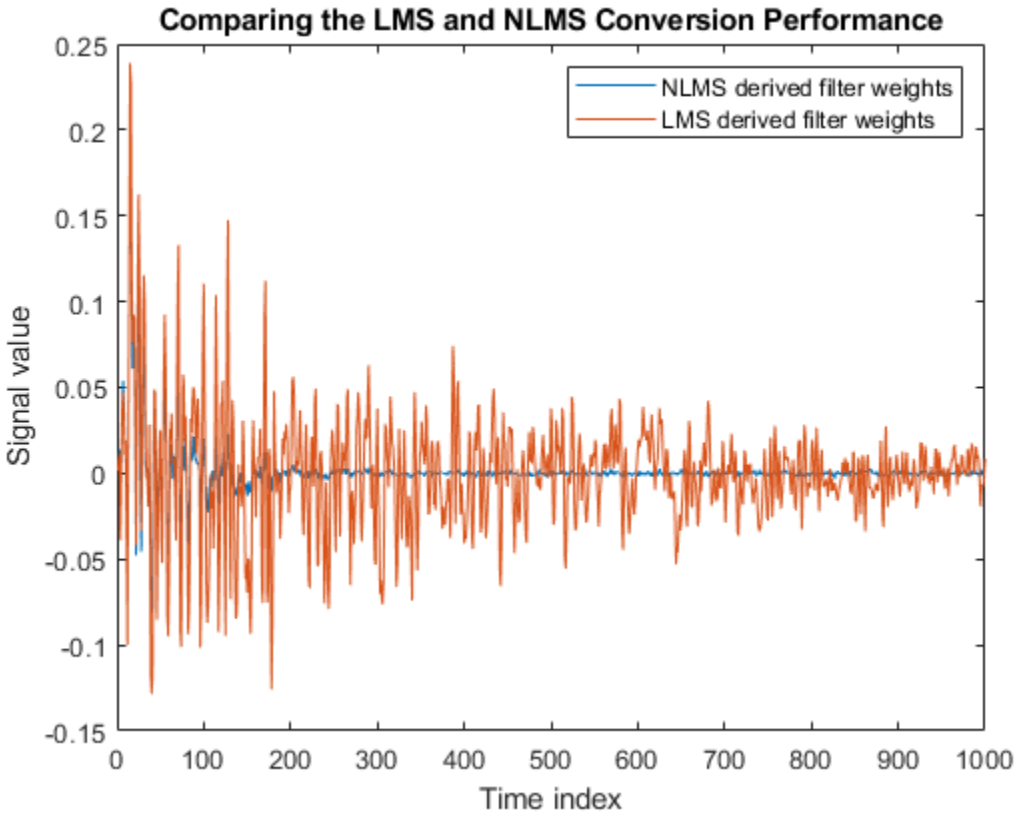
```
mu = 0.2;
lms_nonnormalized = dsp.LMSFilter(13,'StepSize',mu,...
    'Method','LMS');
lms_normalized = dsp.LMSFilter(13,'StepSize',mu,...
    'Method','Normalized LMS');
```

Pass the primary input signal `x` and the desired signal `d` to both the variations of the LMS algorithm. The variables `e1` and `e2` represent the error between the desired signal and the output of the normalized and nonnormalized filters, respecitvely.

```
[~,e1,~] = lms_normalized(x,d);
[~,e2,~] = lms_nonnormalized(x,d);
```

Plot the error signals for both variations. The error signal for the NLMS variant converges to zero much faster than the error signal for the LMS variant. The normalized version adapts in far fewer iterations to a result almost as good as the nonnormalized version.

```
plot([e1,e2]);
title('Comparing the LMS and NLMS Conversion Performance');
legend('NLMS derived filter weights', ...
       'LMS derived filter weights','Location', 'NorthEast');
xlabel('Time index')
ylabel('Signal value')
```



See Also

Objects
`dsp.LMSFilter`

More About

- “System Identification of FIR Filter Using LMS Algorithm” on page 5-9
- “System Identification of FIR Filter Using Normalized LMS Algorithm” on page 5-17
- “Noise Cancellation Using Sign-Data LMS Algorithm” on page 5-22

References

[1] Hayes, Monson H., *Statistical Digital Signal Processing and Modeling*. Hoboken, NJ: John Wiley & Sons, 1996, pp.493–552.

[2] Haykin, Simon, *Adaptive Filter Theory*. Upper Saddle River, NJ: Prentice-Hall, Inc., 1996.

Noise Cancellation Using Sign-Data LMS Algorithm

When the amount of computation required to derive an adaptive filter drives your development process, the sign-data variant of the LMS (SDLMS) algorithm might be a very good choice, as demonstrated in this example.

In the standard and normalized variations of the LMS adaptive filter, coefficients for the adapting filter arise from the mean square error between the desired signal and the output signal from the unknown system. The sign-data algorithm changes the mean square error calculation by using the sign of the input data to change the filter coefficients.

When the error is positive, the new coefficients are the previous coefficients plus the error multiplied by the step size μ. If the error is negative, the new coefficients are again the previous coefficients minus the error multiplied by μ — note the sign change.

When the input is zero, the new coefficients are the same as the previous set.

In vector form, the sign-data LMS algorithm is:

$$w(k+1) = w(k) + \mu e(k)\mathrm{sgn}(x(k)),$$

where

$$\mathrm{sgn}(x(k)) = \begin{cases} 1, & x(k) > 0 \\ 0, & x(k) = 0 \\ -1, & x(k) < 0 \end{cases}$$

with vector w containing the weights applied to the filter coefficients and vector x containing the input data. The vector e is the error between the desired signal and the filtered signal. The objective of the SDLMS algorithm is to minimize this error. Step size is represented by μ.

With a smaller μ, the correction to the filter weights gets smaller for each sample, and the SDLMS error falls more slowly. A larger μ changes the weights more for each step, so the error falls more rapidly, but the resulting error does not approach the ideal solution as closely. To ensure a good convergence rate and stability, select μ within the following practical bounds.

$$0 < \mu < \frac{1}{N\{\mathrm{InputSignalPower}\}},$$

where N is the number of samples in the signal. Also, define μ as a power of two for efficient computing.

Note: How you set the initial conditions of the sign-data algorithm profoundly influences the effectiveness of the adaptation process. Because the algorithm essentially quantizes the input signal, the algorithm can become unstable easily.

A series of large input values, coupled with the quantization process might result in the error growing beyond all bounds. Restrain the tendency of the sign-data algorithm to get out of control by choosing a small step size ($\mu \ll 1$) and setting the initial conditions for the algorithm to nonzero positive and negative values.

In this noise cancellation example, set the `Method` property of `dsp.LMSFilter` to `'Sign-Data LMS'`. This example requires two input data sets:

- Data containing a signal corrupted by noise. In the block diagram under “Noise or Interference Cancellation –– Using an Adaptive Filter to Remove Noise from an Unknown System” on page 5-7, this is the desired signal $d(k)$. The noise cancellation process removes the noise from the signal.
- Data containing random noise. In the block diagram under “Noise or Interference Cancellation –– Using an Adaptive Filter to Remove Noise from an Unknown System” on page 5-7, this is $x(k)$. The signal $x(k)$ is correlated with the noise that corrupts the signal data. Without the correlation between the noise data, the adapting algorithm cannot remove the noise from the signal.

For the signal, use a sine wave. Note that `signal` is a column vector of 1000 elements.

```
signal = sin(2*pi*0.055*(0:1000-1)');
```

Now, add correlated white noise to `signal`. To ensure that the noise is correlated, pass the noise through a lowpass FIR filter and then add the filtered noise to the signal.

```
noise = randn(1000,1);
filt = dsp.FIRFilter;
filt.Numerator = fir1(11,0.4);
fnoise = filt(noise);
d = signal + fnoise;
```

`fnoise` is the correlated noise and `d` is now the desired input to the sign-data algorithm.

To prepare the `dsp.LMSFilter` object for processing, set the initial conditions of the filter weights and `mu` (`StepSize`). As noted earlier in this section, the values you set for `coeffs` and `mu` determine whether the adaptive filter can remove the noise from the signal path.

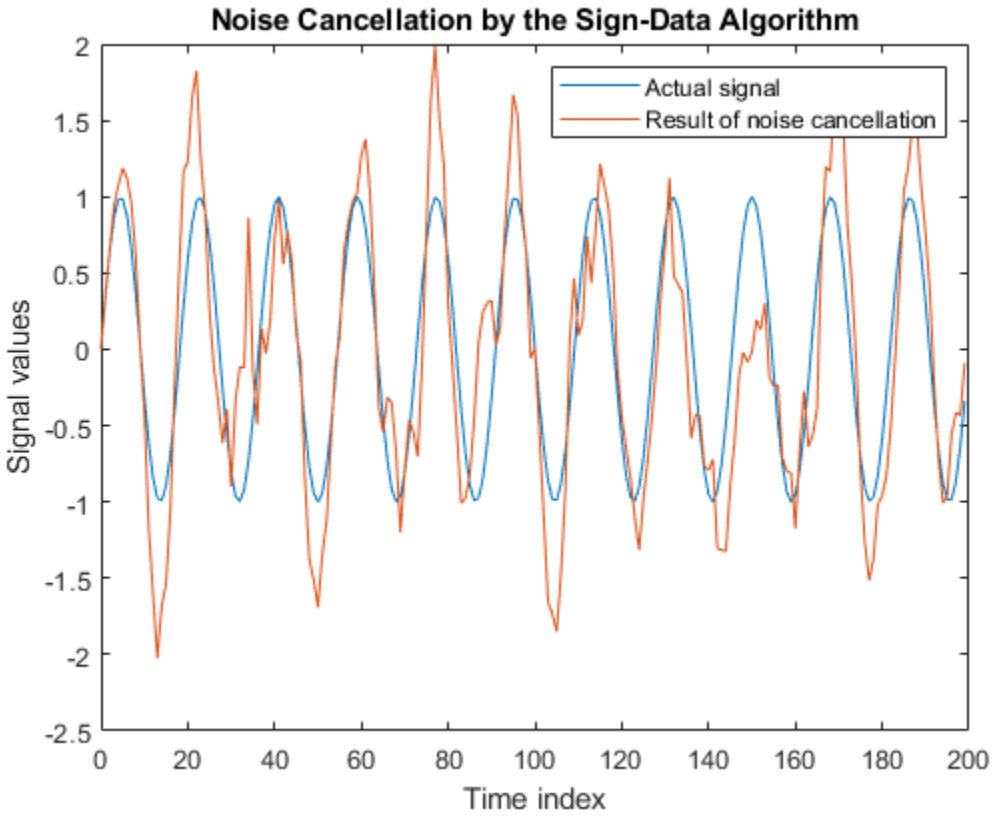
In “System Identification of FIR Filter Using LMS Algorithm”, you constructed a default filter that sets the filter coefficients to zeros. In most cases that approach does not work for the sign-data algorithm. The closer you set your initial filter coefficients to the expected values, the more likely it is that the algorithm remains well behaved and converges to a filter solution that removes the noise effectively.

For this example, start with the coefficients used in the noise filter (`filt.Numerator`), and modify them slightly so the algorithm has to adapt.

```
coeffs = (filt.Numerator).'-0.01; % Set the filter initial conditions.
mu = 0.05; % Set the step size for algorithm updating.
```

With the required input arguments for `dsp.LMSFilter` prepared, construct the LMS filter object, run the adaptation, and view the results.

```
lms = dsp.LMSFilter(12,'Method','Sign-Data LMS',...
   'StepSize',mu,'InitialConditions',coeffs);
[~,e] = lms(noise,d);
L = 200;
plot(0:L-1,signal(1:L),0:L-1,e(1:L));
title('Noise Cancellation by the Sign-Data Algorithm');
legend('Actual signal','Result of noise cancellation',...
       'Location','NorthEast');
xlabel('Time index')
ylabel('Signal values')
```



When `dsp.LMSFilter` runs, it uses far fewer multiplication operations than either of the standard LMS algorithms. Also, performing the sign-data adaptation requires only multiplication by bit shifting when the step size is a power of two.

Although the performance of the sign-data algorithm as shown in this plot is quite good, the sign-data algorithm is much less stable than the standard LMS variations. In this noise cancellation example, the processed signal is a very good match to the input signal, but the algorithm could very easily grow without bound rather than achieve good performance.

Changing the weight initial conditions (`InitialConditions`) and `mu` (`StepSize`), or even the lowpass filter you used to create the correlated noise, can cause noise cancellation to fail.

See Also

Objects
`dsp.LMSFilter`

More About

- “Noise Cancellation Using Sign-Error LMS Algorithm”
- “Noise Cancellation Using Sign-Sign LMS Algorithm”
- “System Identification of FIR Filter Using LMS Algorithm” on page 5-9

References

[1] Hayes, Monson H., *Statistical Digital Signal Processing and Modeling*. Hoboken, NJ: John Wiley & Sons, 1996, pp.493–552.

[2] Haykin, Simon, *Adaptive Filter Theory*. Upper Saddle River, NJ: Prentice-Hall, Inc., 1996.

Compare RLS and LMS Adaptive Filter Algorithms

Least mean squares (LMS) algorithms represent the simplest and most easily applied adaptive algorithms. The recursive least squares (RLS) algorithms, on the other hand, are known for their excellent performance and greater fidelity, but they come with increased complexity and computational cost. In performance, RLS approaches the Kalman filter in adaptive filtering applications with somewhat reduced required throughput in the signal processor. Compared to the LMS algorithm, the RLS approach offers faster convergence and smaller error with respect to the unknown system at the expense of requiring more computations.

Note that the signal paths and identifications are the same whether the filter uses RLS or LMS. The difference lies in the adapting portion.

The LMS filters adapt their coefficients until the difference between the desired signal and the actual signal is minimized (least mean squares of the error signal). This is the state when the filter weights converge to optimal values, that is, they converge close enough to the actual coefficients of the unknown system. This class of algorithms adapt based on the error at the current time. The RLS adaptive filter is an algorithm that recursively finds the filter coefficients that minimize a weighted linear least squares cost function relating to the input signals. These filters adapt based on the total error computed from the beginning.

The LMS filters use a gradient-based approach to perform the adaptation. The initial weights are assumed to be small, in most cases very close to zero. At each step, the filter weights are updated based on the gradient of the mean square error. If the gradient is positive, the filter weights are reduced, so that the error does not increase positively. If the gradient is negative, the filter weights are increased. The step size with which the weights change must be chosen appropriately. If the step size is very small, the algorithm converges very slowly. If the step size is very large, the algorithm converges very fast, and the system might not be stable at the minimum error value. To have a stable system, the step size μ must be within these limits:

$$0 < \mu < \frac{2}{\lambda_{\max}},$$

where $\lambda_{\max}$ is the largest eigenvalue of the input autocorrelation matrix.

The RLS filters minimize the cost function, C by appropriately selecting the filter coefficients w(n) and updating the filter as the new data arrives. The cost function is given by this equation:

$$C(w_n) = \sum_{i=0}^{n} \lambda^{n-i} e^2(i),$$

where

- w_{n} — RLS adaptive filter coefficients.
- e(i) — Error between the desired signal d and the estimate of the desired signal *dest* at the current time index. The signal *dest* is the output of the RLS filter, and so implicitly depends on the current filter coefficients.
- λ — Forgetting factor that gives exponentially less weight to older samples, specified in the range $0 < \lambda \leq 1$. When $\lambda = 1$, all previous errors are considered of equal weight in the total error. As λ approaches zero, the past errors play a smaller role in the total. For example, when $\lambda = 0.1$, the RLS algorithm multiplies an error value from 50 samples in the past by an attenuation factor of $0.1^{50} = 1 \times 10^{-50}$, considerably de-emphasizing the influence of the past errors on the current total error.

In cases where the error value might come from a spurious input data point or points, the forgetting factor lets the RLS algorithm reduce the significance of older error data by multiplying the old data by the forgetting factor.

This table summarizes the key differences between the two types of algorithms:

LMS Algorithm	RLS Algorithm
Simple and can be easily applied.	Increased complexity and computational cost.
Takes longer to converge.	Faster convergence.
Adaptation is based on the gradient-based approach that updates filter weights to converge to the optimum filter weights.	Adaptation is based on the recursive approach that finds the filter coefficients that minimize a weighted linear least squares cost function relating to the input signals.
Larger steady state error with respect to the unknown system.	Smaller steady state error with respect to unknown system.
Does not account for past data.	Accounts for past data from the beginning to the current data point.
Objective is to minimize the current mean square error between the desired signal and the output.	Objective is to minimize the total weighted squared error between the desired signal and the output.
No memory involved. Older error values play no role in the total error considered.	Has infinite memory. All error data is considered in the total error. Using the forgetting factor, the older data can be de-emphasized compared to the newer data. Since $0 \le \lambda < 1$, applying the factor is equivalent to weighting the older error.
LMS based FIR adaptive filters in DSP System Toolbox: • `dsp.LMSFilter` • `dsp.FilteredXLMSFilter` • `dsp.BlockLMSFilter`	RLS based FIR adaptive filters in DSP System Toolbox: • `dsp.RLSFilter` • `dsp.FastTransversalFilter`

Within limits, you can use any of the adaptive filter algorithms to solve an adaptive filter problem by replacing the adaptive portion of the application with a new algorithm.

See Also

Objects
`dsp.LMSFilter` | `dsp.RLSFilter`

More About

- “System Identification of FIR Filter Using LMS Algorithm” on page 5-9
- “System Identification of FIR Filter Using Normalized LMS Algorithm” on page 5-17
- “Noise Cancellation Using Sign-Data LMS Algorithm” on page 5-22

- “Inverse System Identification Using RLS Algorithm” on page 5-29

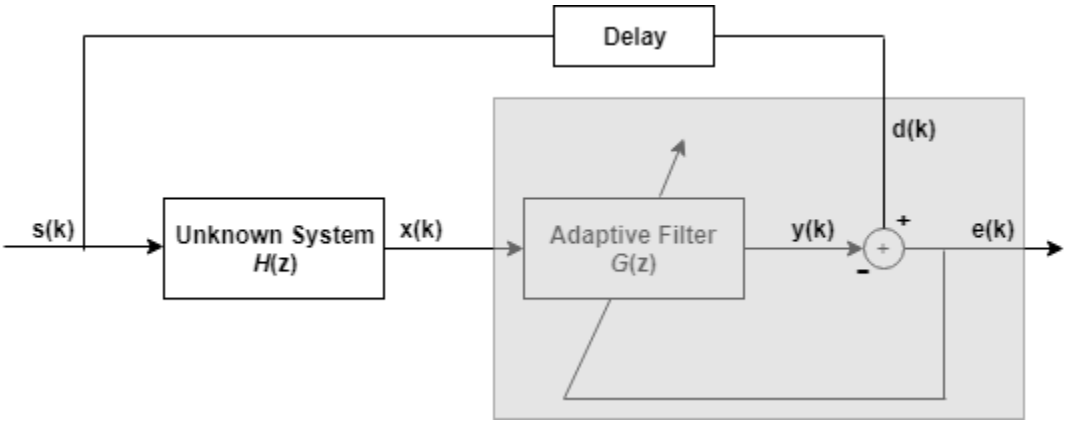
References

[1] Hayes, Monson H., *Statistical Digital Signal Processing and Modeling*. Hoboken, NJ: John Wiley & Sons, 1996, pp.493–552.

[2] Haykin, Simon, *Adaptive Filter Theory*. Upper Saddle River, NJ: Prentice-Hall, Inc., 1996.

Inverse System Identification Using RLS Algorithm

This example demonstrates the RLS adaptive algorithm using the inverse system identification model shown here.



Cascading the adaptive filter with an unknown filter causes the adaptive filter to converge to a solution that is the inverse of the unknown system.

If the transfer function of the unknown system and the adaptive filter are $H(z)$ and $G(z)$, respectively, the error measured between the desired signal and the signal from the cascaded system reaches its minimum when $G(z) \times H(z) = 1$. For this relation to be true, $G(z)$ must equal $1/H(z)$, the inverse of the transfer function of the unknown system.

To demonstrate that this is true, create a signal s to input to the cascaded filter pair.

```
s = randn(3000,1);
```

In the cascaded filters case, the unknown filter results in a delay in the signal arriving at the summation point after both filters. To prevent the adaptive filter from trying to adapt to a signal it has not yet seen (equivalent to predicting the future), delay the desired signal by 12 samples, which is the order of the unknown system.

Generally, you do not know the order of the system you are trying to identify. In that case, delay the desired signal by number of samples equal to half the order of the adaptive filter. Delaying the input requires prepending 12 zero-value samples to the input `s`.

```
delay = zeros(12,1);
d = [delay; s(1:2988)]; % Concatenate the delay and the signal.
```

You have to keep the desired signal vector `d` the same length as `x`, so adjust the signal element count to allow for the delay samples.

Although not generally the case, for this example you know the order of the unknown filter, so add a delay equal to the order of the unknown filter.

For the unknown system, use a lowpass, 12th-order FIR filter.

```
filt = dsp.FIRFilter;
filt.Numerator = fir1(12,0.55,'low');
```

Filtering `s` provides the input data signal for the adaptive algorithm function.

```
x = filt(s);
```

To use the RLS algorithm, create a `dsp.RLSFilter` object and set its `Length`, `ForgettingFactor`, and `InitialInverseCovariance` properties.

For more information about the input conditions to prepare the RLS algorithm object, refer to `dsp.RLSFilter`.

```
p0 = 2 * eye(13);
lambda = 0.99;
rls = dsp.RLSFilter(13,'ForgettingFactor',lambda,...
   'InitialInverseCovariance',p0);
```

This example seeks to develop an inverse solution, you need to be careful about which signal carries the data and which is the desired signal.

Earlier examples of adaptive filters use the filtered noise as the desired signal. In this case, the filtered noise (`x`) carries the unknown system's information. With Gaussian distribution and variance of 1, the unfiltered noise `d` is the desired signal. The code to run this adaptive filter is:

```
[y,e] = rls(x,d);
```

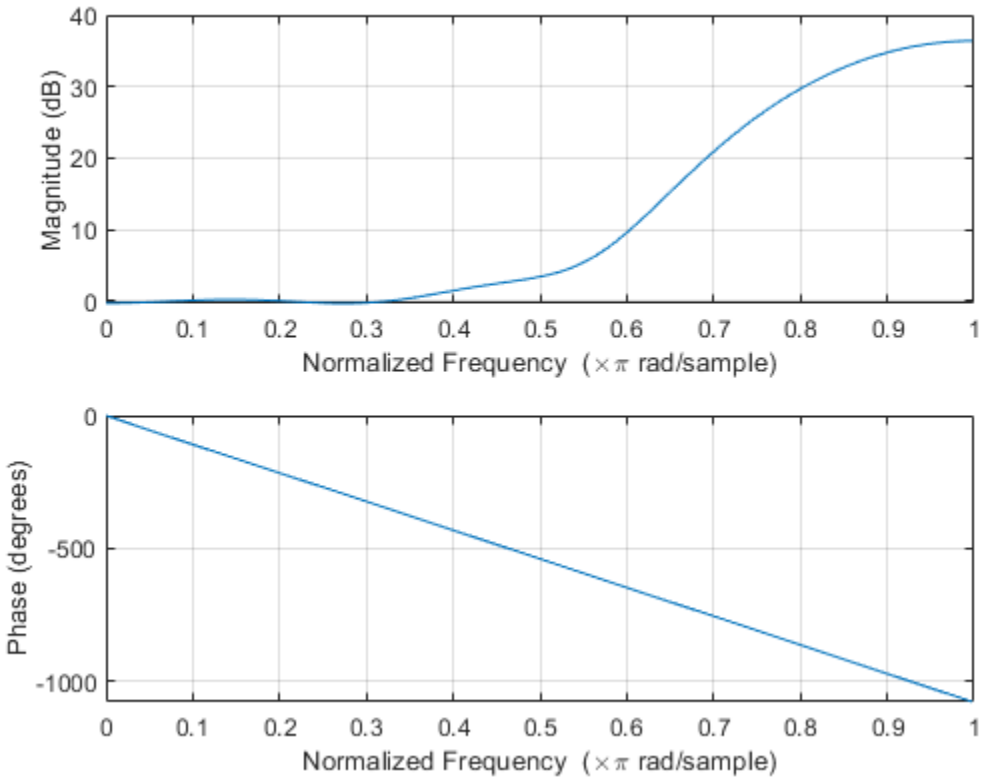
where `y` returns the filtered output and `e` contains the error signal as the filter adapts to find the inverse of the unknown system.

Obtain the estimated coefficients of the RLS filter.

```
b = rls.Coefficients;
```

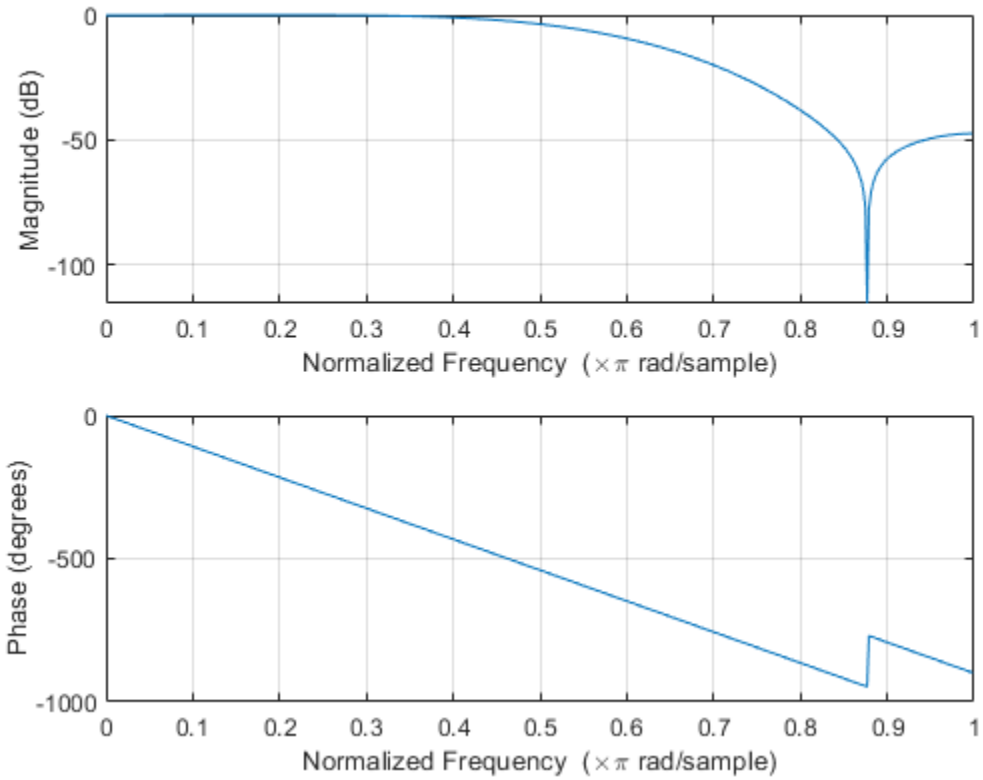
View the frequency response of the adapted RLS filter (inverse system, *G*(z)) using `freqz`. The inverse system looks like a highpass filter with linear phase.

```
freqz(b,1)
```



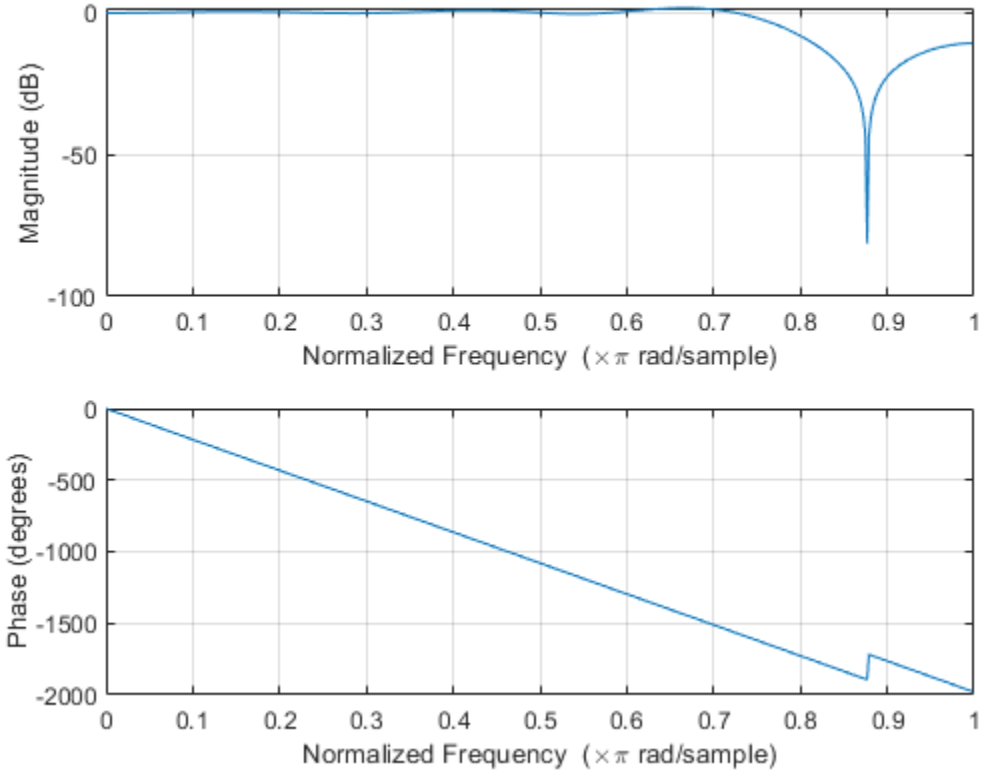
View the frequency response of the unknown system, $H(z)$. The response is that of a lowpass filter with a cutoff frequency of 0.55.

```
freqz(filt.Numerator,1)
```



The result of the cascade of the unknown system and the adapted filter is a compensated system with an extended cutoff frequency of 0.8.

```
overallCoeffs = conv(filt.Numerator,b);
freqz(overallCoeffs,1)
```



See Also

Objects
`dsp.RLSFilter`

More About

- “Compare RLS and LMS Adaptive Filter Algorithms” on page 5-26
- “System Identification of FIR Filter Using LMS Algorithm” on page 5-9

References

[1] Hayes, Monson H., *Statistical Digital Signal Processing and Modeling*. Hoboken, NJ: John Wiley & Sons, 1996, pp.493–552.

[2] Haykin, Simon, *Adaptive Filter Theory*. Upper Saddle River, NJ: Prentice-Hall, Inc., 1996.

Signal Enhancement Using LMS and NLMS Algorithms

Using the least mean square (LMS) and normalized LMS algorithms, extract the desired signal from a noise-corrupted signal by filtering out the noise. Both of these algorithms are available with the `dsp.LMSFilter` System object™.

Create the Signals for Adaptation

The desired signal (the output from the process) is a sinusoid with 1000 samples.

```
n = (1:1000)';
s = sin(0.075*pi*n);
```

To perform adaptation, the filter requires two signals:

- A reference signal
- A noisy signal that contains both the desired signal and an added noise component

Generate the Noise Signal

Create a noise signal with autoregressive noise (defined as `v1`). In autoregressive noise, the noise at time t depends only on the previous values and a random disturbance.

```
v = 0.8*randn(1000,1); % Random noise part.
ar = [1,1/2];          % Autoregression coefficients.
v1 = filter(1,ar,v);   % Noise signal. Applies a 1-D digital
                       % filter.
```

Corrupt the Desired Signal to Create a Noisy Signal

To generate the noisy signal that contains both the desired signal and the noise, add the noise signal `v1` to the desired signal `s`. The noise-corrupted sinusoid `x` is:

```
x = s + v1;
```

Adaptive filter processing seeks to recover `s` from `x` by removing `v1`. To complete the signals needed to perform adaptive filtering, the adaptation process requires a reference signal.

Create a Reference Signal

Define a moving average signal `v2` that is correlated with `v1`. The signal `v2` is the reference signal for this example.

```
ma = [1, -0.8, 0.4 , -0.2];
v2 = filter(ma,1,v);
```

Construct Two Adaptive Filters

Two similar, sixth-order adaptive filters — LMS and NLMS — form the basis of this example. Set the order as a variable in MATLAB™ and create the filters.

```
L = 7;
lms = dsp.LMSFilter(L,'Method','LMS');
nlms = dsp.LMSFilter(L,'Method','Normalized LMS');
```

Choose the Step Size

LMS-like algorithms have a step size that determines the amount of correction applied as the filter adapts from one iteration to the next. A step size that is too small increases the time for the filter to converge on a set of coefficients. A step size that is too large might cause the adapting filter to diverge and never reach convergence. In this case, the resulting filter might not be stable.

As a rule of thumb, smaller step sizes improve the accuracy with which the filter converges to match the characteristics of the unknown system, at the expense of the time it takes to adapt.

The `maxstep` function of `dsp.LMSFilter` object determines the maximum step size suitable for each LMS adaptive filter algorithm that ensures that the filter converges to a solution. Often, the notation for the step size is μ.

```
[mumaxlms,mumaxmselms] = maxstep(lms,x)

mumaxlms = 0.2088

mumaxmselms = 0.1214

[mumaxnlms,mumaxmsenlms] = maxstep(nlms,x)

mumaxnlms = 2

mumaxmsenlms = 2
```

Set the Adapting Filter Step Size

The first output of the `maxstep` function is the value needed for the mean of the coefficients to converge, while the second output is the value needed for the mean squared coefficients to converge. Choosing a large step size often causes large variations from the convergence values, so generally choose smaller step sizes.

```
lms.StepSize  = mumaxmselms/30

lms =
  dsp.LMSFilter with properties:

                   Method: 'LMS'
                   Length: 7
           StepSizeSource: 'Property'
                 StepSize: 0.0040
            LeakageFactor: 1
        InitialConditions: 0
           AdaptInputPort: false
    WeightsResetInputPort: false
            WeightsOutput: 'Last'

  Show all properties


nlms.StepSize = mumaxmsenlms/20

nlms =
  dsp.LMSFilter with properties:

                   Method: 'Normalized LMS'
                   Length: 7
```

```
          StepSizeSource: 'Property'
                StepSize: 0.1000
            LeakageFactor: 1
        InitialConditions: 0
           AdaptInputPort: false
    WeightsResetInputPort: false
           WeightsOutput: 'Last'

  Show all properties
```

Filter with the Adaptive Filters

You have set up the parameters of the adaptive filters and are now ready to filter the noisy signal. The reference signal v2 is the input to the adaptive filters. x is the desired signal in this configuration.

Through adaptation, y, the output of the filters, tries to emulate x as closely as possible.

Since v2 is correlated only with the noise component v1 of x, it can only really emulate v1. The error signal (the desired x), minus the actual output y, constitutes an estimate of the part of x that is not correlated with v2 — s, the signal to extract from x.

```
[ylms,elms,wlms] = lms(v2,x);
[ynlms,enlms,wnlms] = nlms(v2,x);
```

Compute the Optimal Solution

For comparison, compute the optimal FIR Wiener filter.

```
bw = firwiener(L-1,v2,x); % Optimal FIR Wiener filter
yw = filter(bw,1,v2);   % Estimate of x using Wiener filter
ew = x - yw;            % Estimate of actual sinusoid
```

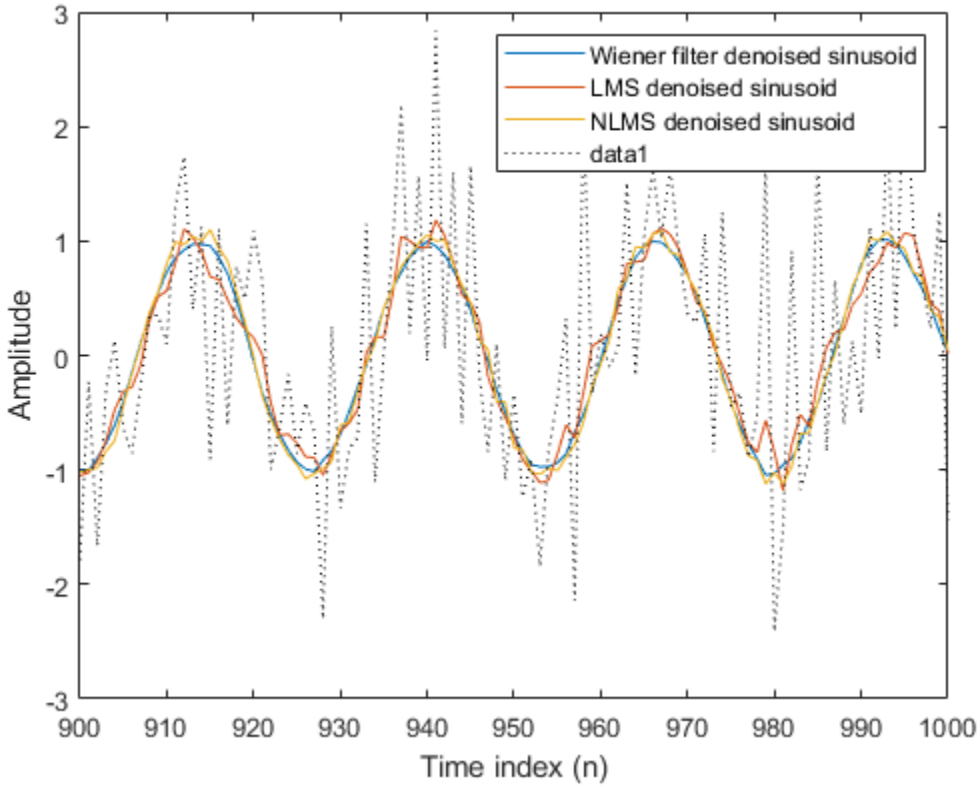
Plot the Results

Plot the resulting denoised sinusoid for each filter — the Wiener filter, the LMS adaptive filter, and the NLMS adaptive filter — to compare the performance of the various techniques.

```
plot(n(900:end),[ew(900:end), elms(900:end),enlms(900:end)])
legend('Wiener filter denoised sinusoid',...
    'LMS denoised sinusoid','NLMS denoised sinusoid')
xlabel('Time index (n)')
ylabel('Amplitude')
```

As a reference point, include the noisy signal as a dotted line in the plot.

```
hold on
plot(n(900:end),x(900:end),'k:')
xlabel('Time index (n)')
ylabel('Amplitude')
hold off
```



Compare the Final Coefficients

Finally, compare the Wiener filter coefficients with the coefficients of the adaptive filters. While adapting, the adaptive filters try to converge to the Wiener coefficients.

```
[bw.' wlms wnlms]
```

```
ans = 7×3

    1.0273    0.8761    1.0202
    0.3390    0.1368    0.3225
    0.1201    0.0095    0.1227
    0.0425   -0.0038    0.0495
    0.1098    0.0503    0.0773
    0.0534   -0.0094    0.0295
    0.0146   -0.0054   -0.0179
```

Reset the Filter Before Filtering

You can reset the internal filter states at any time by calling the `reset` function on the filter object.

For instance, these successive calls produce the same output after resetting the object.

```
[ylms,elms,wlms] = lms(v2,x);
[ynlms,enlms,wnlms] = nlms(v2,x);
```

If you do not reset the filter object, the filter uses the final states and coefficients from the previous run as the initial conditions and data set for the next run.

Investigate Convergence Through Learning Curves

To analyze the convergence of the adaptive filters, use the learning curves. The toolbox provides methods to generate the learning curves, but you need more than one iteration of the experiment to obtain significant results.

This demonstration uses 25 sample realizations of the noisy sinusoids.

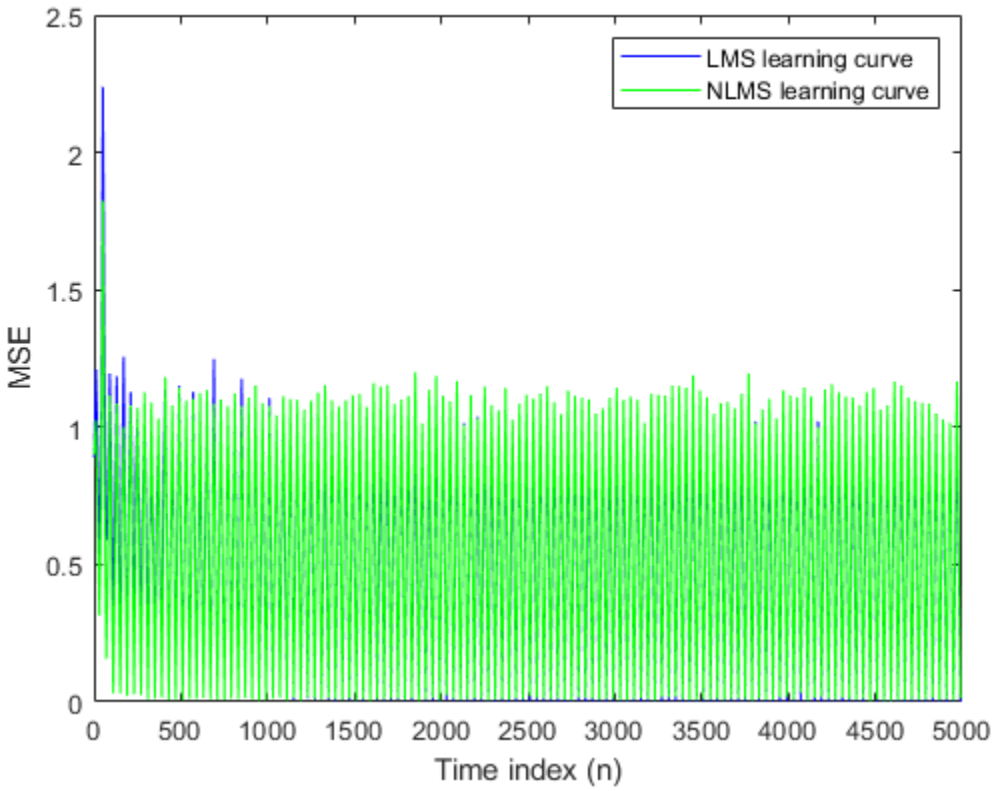
```
n = (1:5000)';
s = sin(0.075*pi*n);
nr = 25;
v = 0.8*randn(5000,nr);
v1 = filter(1,ar,v);
x = repmat(s,1,nr) + v1;
v2 = filter(ma,1,v);
```

Compute the Learning Curves

Now compute mean squared error. To speed things up, compute the error every 10 samples.

First, reset the adaptive filters to avoid using the coefficients it has already computed and the states it has stored. Then plot the learning curves for the LMS and NLMS adaptive filters.

```
reset(lms);
reset(nlms);
M = 10; % Decimation factor
mselms = msesim(lms,v2,x,M);
msenlms = msesim(nlms,v2,x,M);
%plot(1:M:n(end),[mselms,msenlms])
plot(1:M:n(end),mselms,'b',1:M:n(end),msenlms,'g')
legend('LMS learning curve','NLMS learning curve')
xlabel('Time index (n)')
ylabel('MSE')
```



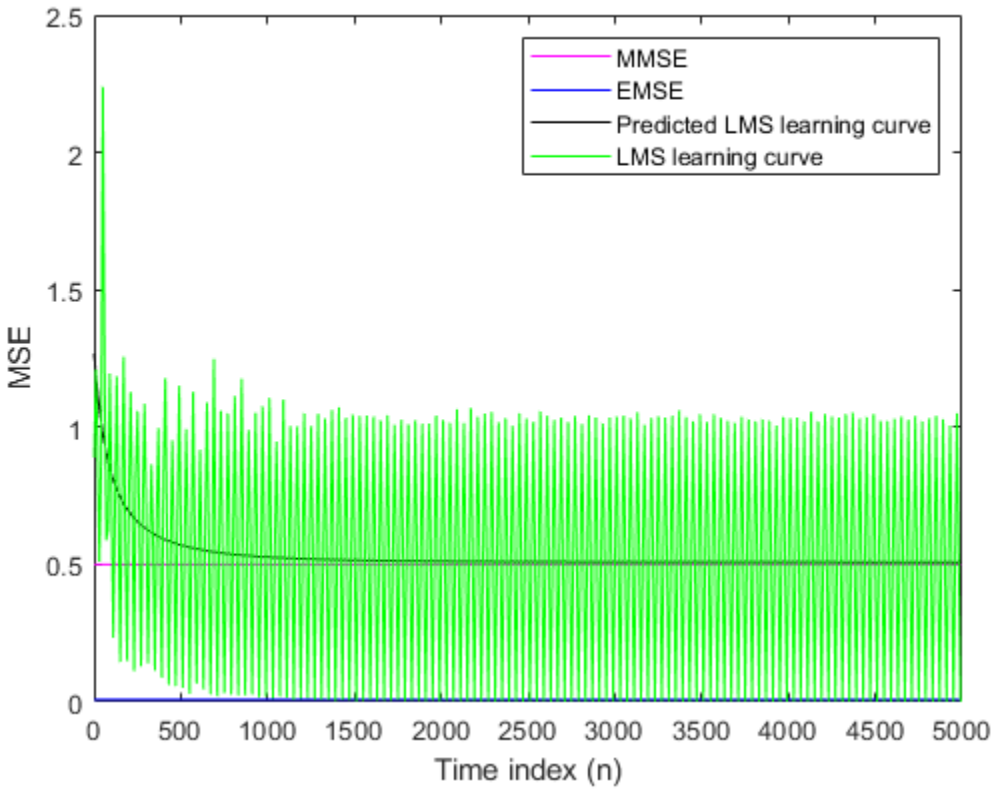
In this plot you see the calculated learning curves for the LMS and NLMS adaptive filters.

Compute the Theoretical Learning Curves

For the LMS and NLMS algorithms, functions in the toolbox help you compute the theoretical learning curves, along with the minimum mean squared error (MMSE), the excess mean squared error (EMSE), and the mean value of the coefficients.

MATLAB might take some time to calculate the curves. The figure shown after the code plots the predicted and actual LMS curves.

```
reset(lms);
[mmselms,emselms,meanwlms,pmselms] = msepred(lms,v2,x,M);
x = 1:M:n(end);
y1 = mmselms*ones(500,1);
y2 = emselms*ones(500,1);
y3 = pmselms;
y4 = mselms;
plot(x,y1,'m',x,y2,'b',x,y3,'k',x,y4,'g')
legend('MMSE','EMSE','Predicted LMS learning curve',...
    'LMS learning curve')
xlabel('Time index (n)')
ylabel('MSE')
```


MMSE
EMSE
Predicted LMS learning curve
LMS learning curve
MSE
Time index (n)
0
0.5
1
1.5
2
2.5
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000

Noise Cancellation in Simulink Using Normalized LMS Adaptive Filter

In this section...
“Create an Acoustic Environment in Simulink” on page 5-41
“LMS Filter Configuration for Adaptive Noise Cancellation” on page 5-42
“Modify Adaptive Filter Parameters During Model Simulation” on page 5-45

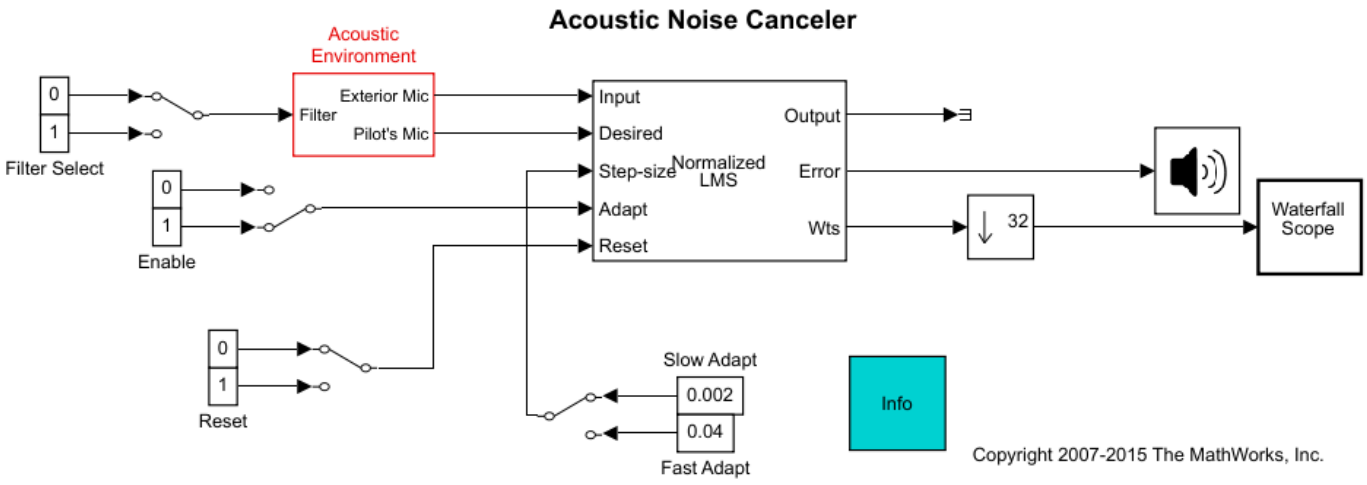
Create an Acoustic Environment in Simulink

Adaptive filters are filters whose coefficients or weights change over time to adapt to the statistics of a signal. They are used in a variety of fields including communications, controls, radar, sonar, seismology, and biomedical engineering.

In this topic, you learn how to create an acoustic environment that simulates both white noise and colored noise added to an input signal. You later use this environment to build a model capable of adaptive noise cancellation using adaptive filtering:

1 At the MATLAB command line, type `dspanc`.

The DSP System Toolbox Acoustic Noise Cancellation example opens.



2 Copy and paste the subsystem called Acoustic Environment into a new model.

3 Double-click the Acoustic Environment subsystem.

Gaussian noise is used to create the signal sent to the Exterior Mic output port. If the input to the Filter port changes from 0 to 1, the Digital Filter block changes from a lowpass filter to a bandpass filter. The filtered noise output from the Digital Filter block is added to signal coming from a `.wav` file to produce the signal sent to the Pilot's Mic output port.

You have now created an acoustic environment. In the following topics, you use this acoustic environment to produce a model capable of adaptive noise cancellation.

LMS Filter Configuration for Adaptive Noise Cancellation

In the previous topic, “Create an Acoustic Environment in Simulink” on page 5-41, you created a system that produced two output signals. The signal output at the Exterior Mic port is composed of white noise. The signal output at the Pilot's Mic port is composed of colored noise added to a signal from a `.wav` file. In this topic, you create an adaptive filter to remove the noise from the Pilot's Mic signal. This topic assumes that you are working on a Windows operating system:

1. If the model you created in “Create an Acoustic Environment in Simulink” on page 5-41 is not open on your desktop, you can open an equivalent model by typing

   ```
   ex_adapt1_audio
   ```

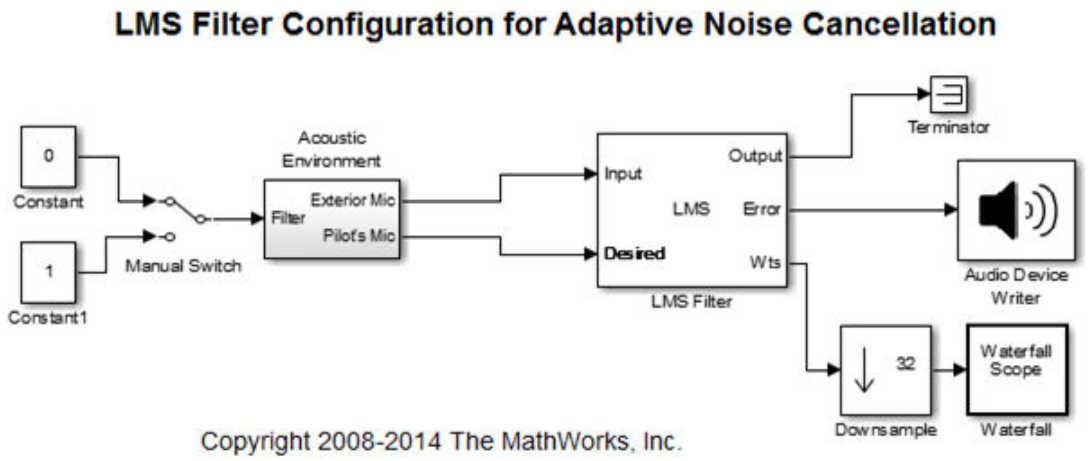
   at the MATLAB command prompt.
2. From the DSP System Toolbox Filtering library, and then from the Adaptive Filters library, click-and-drag an LMS Filter block into the model that contains the Acoustic Environment subsystem.
3. Double-click the LMS Filter block. Set the block parameters as follows, and then click **OK**:

   - **Algorithm** = `Normalized LMS`
   - **Filter length** = `40`
   - **Step size (mu)** = `0.002`
   - **Leakage factor (0 to 1)** = `1`

   The block uses the normalized LMS algorithm to calculate the forty filter coefficients. Setting the **Leakage factor (0 to 1)** parameter to `1` means that the current filter coefficient values depend on the filter's initial conditions and all of the previous input values.
4. Click-and-drag the following blocks into your model.

Block	Library	Quantity
Constant	Simulink/Sources	2
Manual Switch	Simulink/Signal Routing	1
Terminator	Simulink/Sinks	1
Downsample	Signal Operations	1
Audio Device Writer	Sinks	1
Waterfall Scope	Sinks	1

5. Connect the blocks so that your model resembles the following figure.



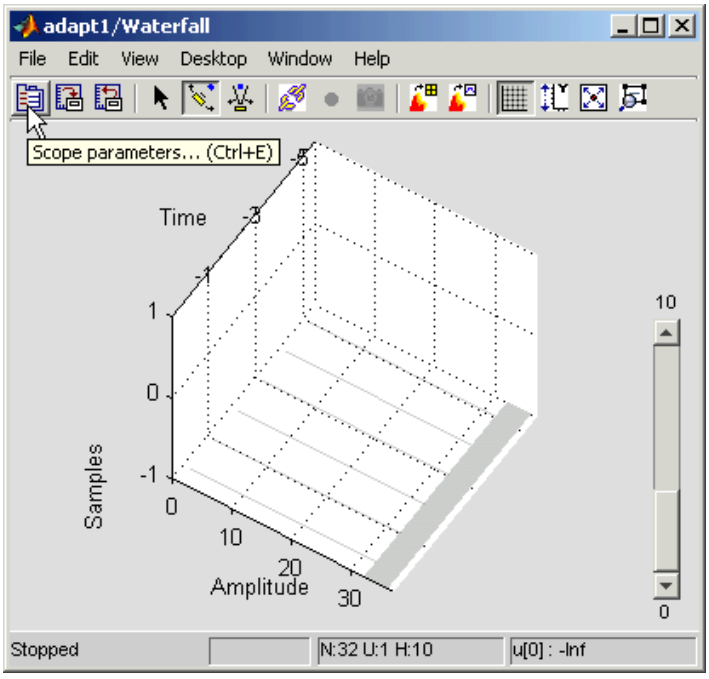
6 Double-click the Constant block. Set the **Constant value** parameter to `0` and then click **OK**.

7 Double-click the Downsample block. Set the **Downsample factor, K** parameter to 32. Click **OK**.

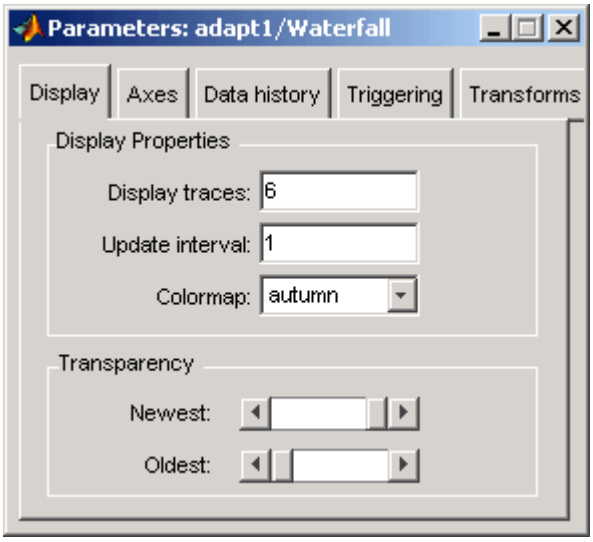
The filter weights are being updated so often that there is very little change from one update to the next. To see a more noticeable change, you need to downsample the output from the Wts port.

8 Double-click the Waterfall Scope block. The **Waterfall** scope window opens.

9 Click the **Scope** parameters button.



The **Parameters** window opens.



10 Click the **Axes** tab. Set the parameters as follows:

- **Y Min** = `-0.188`
- **Y Max** = `0.179`

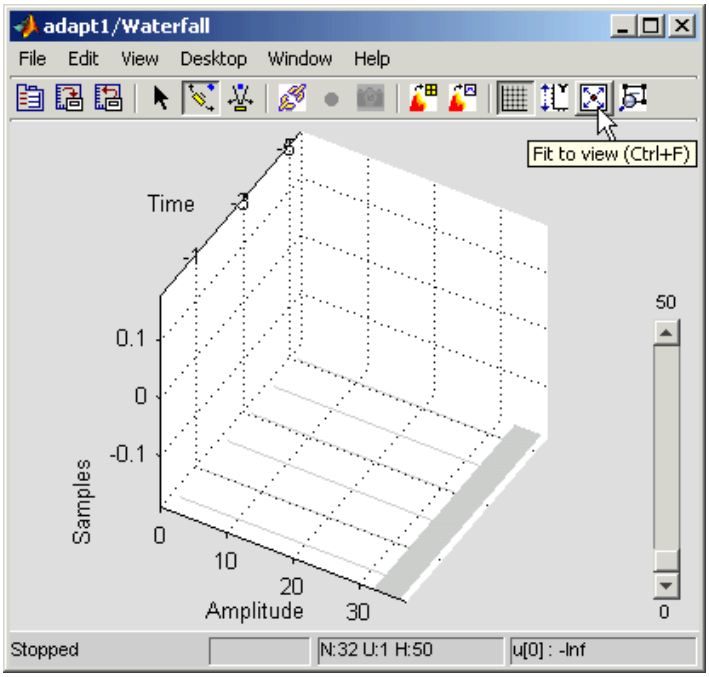
11 Click the **Data history** tab. Set the parameters as follows:

- **History traces** = `50`
- **Data logging** = `All visible`

12 Close the **Parameters** window leaving all other parameters at their default values.

You might need to adjust the axes in the **Waterfall** scope window in order to view the plots.

13 Click the **Fit to view** button in the **Waterfall** scope window. Then, click-and-drag the axes until they resemble the following figure.



14 In the **Modeling** tab, click **Model Settings**. In the **Solver** pane, set the parameters as follows, and then click **OK**:

- **Stop time** = `inf`
- **Type** = `Fixed-step`
- **Solver** = `Discrete (no continuous states)`

15 Run the simulation and view the results in the **Waterfall** scope window. You can also listen to the simulation using the speakers attached to your computer.

16 Experiment with changing the Manual Switch so that the input to the Acoustic Environment subsystem is either 0 or 1.

When the value is 0, the Gaussian noise in the signal is being filtered by a lowpass filter. When the value is 1, the noise is being filtered by a bandpass filter. The adaptive filter can remove the noise in both cases.

You have now created a model capable of adaptive noise cancellation. The adaptive filter in your model is able to filter out both low frequency noise and noise within a frequency range. In the next topic, “Modify Adaptive Filter Parameters During Model Simulation” on page 5-45, you modify the LMS Filter block and change its parameters during simulation.

Modify Adaptive Filter Parameters During Model Simulation

In the previous topic, “LMS Filter Configuration for Adaptive Noise Cancellation” on page 5-42, you created an adaptive filter and used it to remove the noise generated by the Acoustic Environment subsystem. In this topic, you modify the adaptive filter and adjust its parameters during simulation. This topic assumes that you are working on a Windows operating system:

1 If the model you created in “Create an Acoustic Environment in Simulink” on page 5-41 is not open on your desktop, you can open an equivalent model by typing

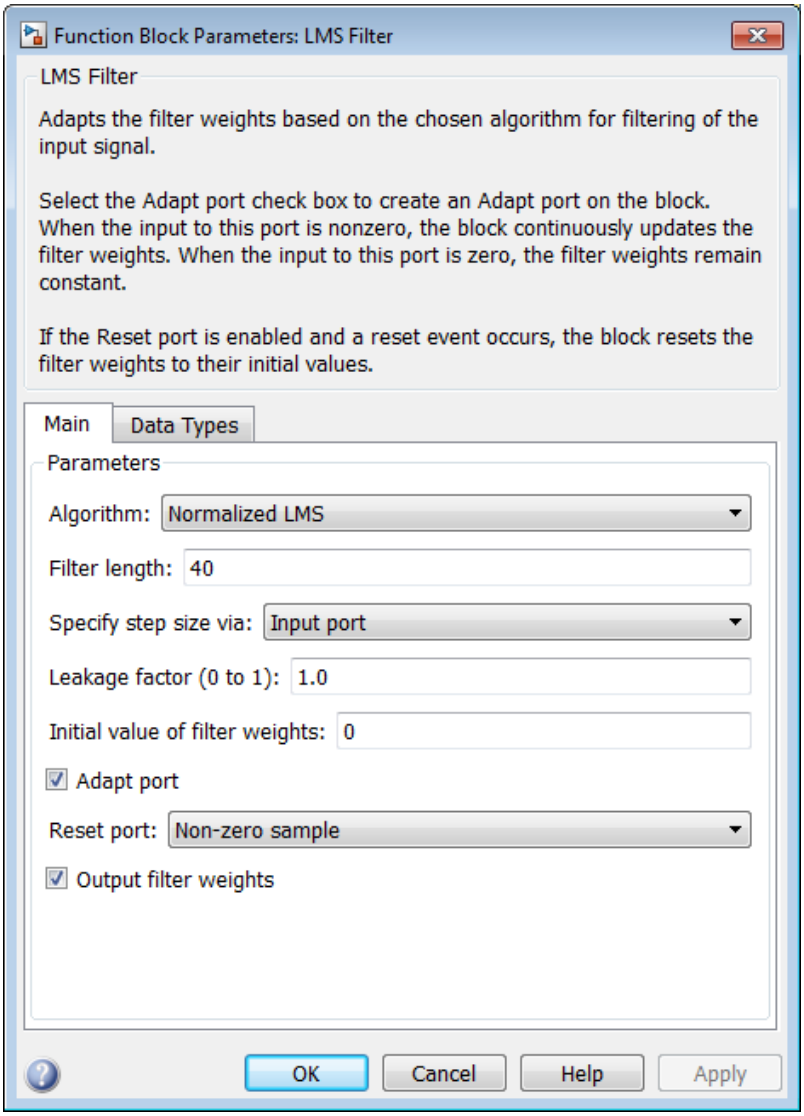
```
ex_adapt2_audio
```

at the MATLAB command prompt.

2 Double-click the LMS filter block. Set the block parameters as follows, and then click **OK**:

- **Specify step size via** = `Input port`
- **Initial value of filter weights** = `0`
- Select the **Adapt port** check box.
- **Reset port** = `Non-zero sample`

The **Block Parameters: LMS Filter** dialog box should now look similar to the following figure.

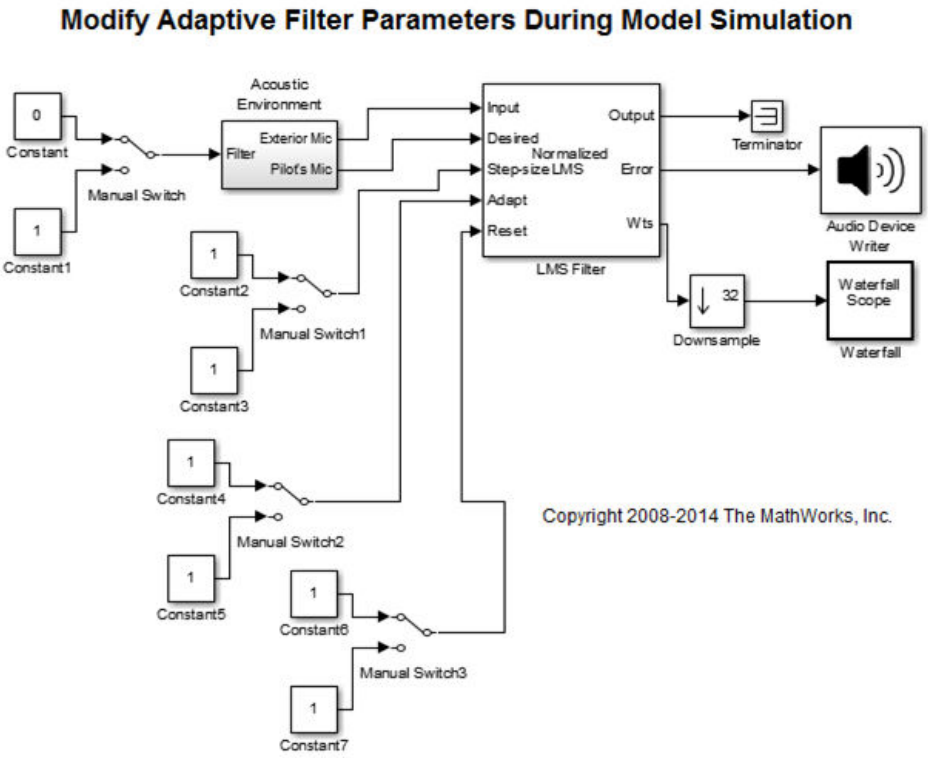


Step-size, Adapt, and Reset ports appear on the LMS Filter block.

3 Click-and-drag the following blocks into your model.

Block	Library	Quantity
Constant	Simulink/Sources	6
Manual Switch	Simulink/Signal Routing	3

4 Connect the blocks as shown in the following figure.



5 Double-click the Constant2 block. Set the block parameters as follows, and then click **OK**:

- **Constant value** = `0.002`
- Select the **Interpret vector parameters as 1-D** check box.
- **Sample time (-1 for inherited)** = `inf`
- **Output data type mode** = `Inherit via back propagation`

6 Double-click the Constant3 block. Set the block parameters as follows, and then click **OK**:

- **Constant value** = `0.04`
- Select the **Interpret vector parameters as 1-D** check box.
- **Sample time (-1 for inherited)** = `inf`
- **Output data type mode** = `Inherit via back propagation`

7 Double-click the Constant4 block. Set the **Constant value** parameter to `0` and then click **OK**.

8 Double-click the Constant6 block. Set the **Constant value** parameter to `0` and then click **OK**.

9 In the **Debug** tab, select **Information Overlays** > **Nonscalar Signals** and **Signal Dimensions**.

10 Double-click Manual Switch2 so that the input to the Adapt port is 1.

11. Run the simulation and view the results in the **Waterfall** scope window. You can also listen to the simulation using the speakers attached to your computer.
12. Double-click the Manual Switch block so that the input to the Acoustic Environment subsystem is 1. Then, double-click Manual Switch2 so that the input to the Adapt port to 0.

    The filter weights displayed in the **Waterfall** scope window remain constant. When the input to the Adapt port is 0, the filter weights are not updated.
13. Double-click Manual Switch2 so that the input to the Adapt port is 1.

    The LMS Filter block updates the coefficients.
14. Connect the Manual Switch1 block to the Constant block that represents 0.002. Then, change the input to the Acoustic Environment subsystem. Repeat this procedure with the Constant block that represents 0.04.

    You can see that the system reaches steady state faster when the step size is larger.
15. Double-click the Manual Switch3 block so that the input to the Reset port is 1.

    The block resets the filter weights to their initial values. In the **Block Parameters: LMS Filter** dialog box, from the **Reset port** list, you chose `Non-zero sample`. This means that any nonzero input to the Reset port triggers a reset operation.

You have now experimented with adaptive noise cancellation using the LMS Filter block. You adjusted the parameters of your adaptive filter and viewed the effects of your changes while the model was running.

For more information about adaptive filters, see the following block reference pages:

- LMS Filter
- RLS Filter
- Block LMS Filter
- Fast Block LMS Filter

References

[1] Hayes, Monson H., *Statistical Digital Signal Processing and Modeling*. Hoboken, NJ: John Wiley & Sons, 1996, pp.493–552.

[2] Haykin, Simon, *Adaptive Filter Theory*. Upper Saddle River, NJ: Prentice-Hall, Inc., 1996.

See Also

Related Examples

- “Time-Delay Estimation”
- “Nonstationary Channel Estimation Using Recursive Least Squares”
- “Adaptive Noise Cancellation Using RLS Adaptive Filtering”

Multirate and Multistage Filters

Learn how to analyze, design, and implement multirate and multistage filters in MATLAB and Simulink.

- “Multirate Filters” on page 6-2
- “Multistage Filters” on page 6-5
- “Compare Single-Rate/Single-Stage Filters with Multirate/Multistage Filters” on page 6-6
- “Design of Decimators/Interpolators” on page 6-9
- “Filter Banks” on page 6-21
- “Multirate Filtering in Simulink” on page 6-27

Multirate Filters

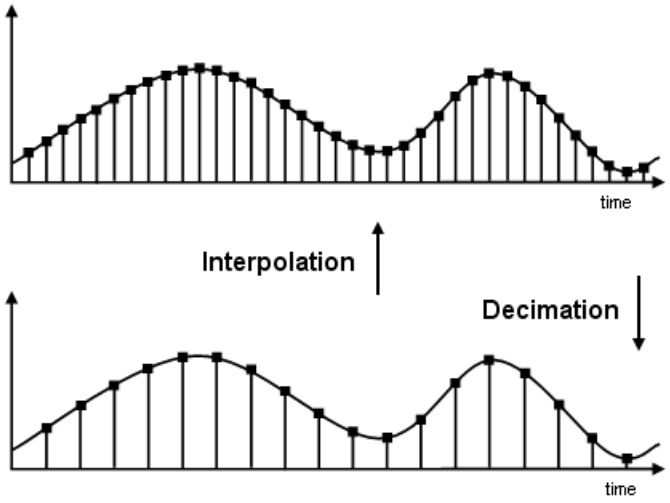
In this section...
“Why Are Multirate Filters Needed?” on page 6-2 “Overview of Multirate Filters” on page 6-2

Why Are Multirate Filters Needed?

Multirate filters can bring efficiency to a particular filter implementation. In general, multirate filters are filters in which different parts of the filter operate at different rates. The most obvious application of such a filter is when the input sample rate and output sample rate need to differ (decimation or interpolation) — however, multirate filters are also often used in designs where this is not the case. For example you may have a system where the input sample rate and output sample rate are the same, but internally there is decimation and interpolation occurring in a series of filters, such that the final output of the system has the same sample rate as the input. Such a design may exhibit lower cost than could be achieved with a single-rate filter for various reasons. For more information about the relative cost benefit of using multirate filters, see Harris, Fredric J., Multirate Signal Processing for Communication Systems, Prentice Hall PTR, 2004.

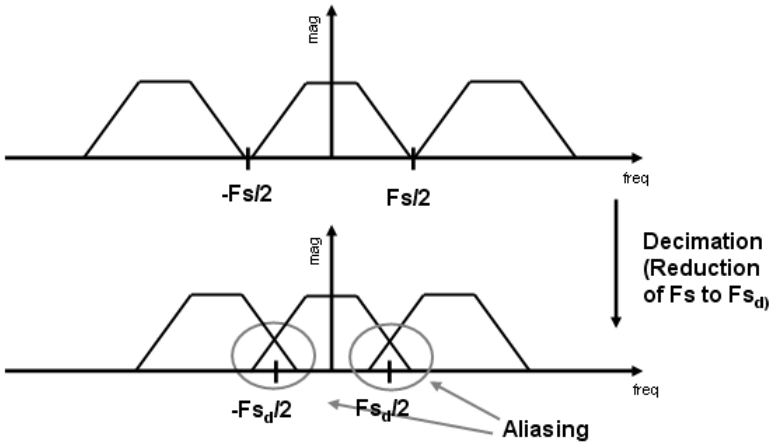
Overview of Multirate Filters

A filter that reduces the input rate is called a *decimator*. A filter that increases the input rate is called an *interpolator*. To visualize this process, examine the following figure, which illustrates the processes of interpolation and decimation in the time domain.

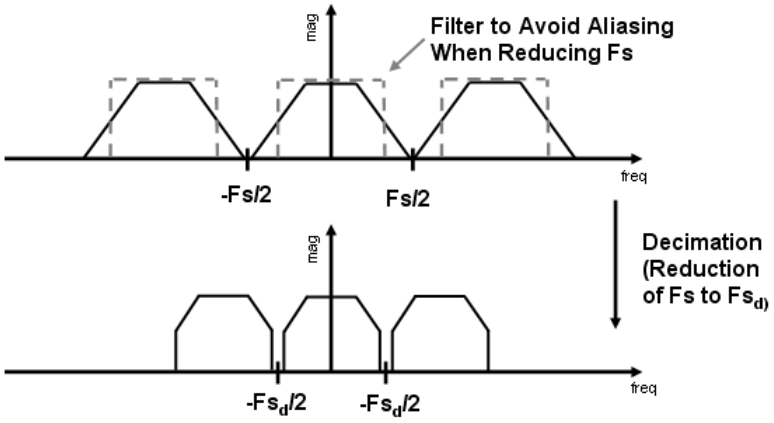


If you start with the top signal, sampled at a frequency `Fs`, then the bottom signal is sampled at `Fs/2` frequency. In this case, the decimation factor, or `M`, is `2`.

The following figure illustrates effect of decimation in the frequency domain.



In the first graphic in the figure you can see a signal that is critically sampled, i.e. the sample rate is equal to two times the highest frequency component of the sampled signal. As such the period of the signal in the frequency domain is no greater than the bandwidth of the sampling frequency. When reduce the sampling frequency (decimation), *aliasing* can occur, where the magnitudes at the frequencies near the edges of the original period become indistinguishable, and the information about these values becomes lost. To work around this problem, the signal can be filtered before the decimation process, avoiding overlap of the signal spectra at Fs/2.

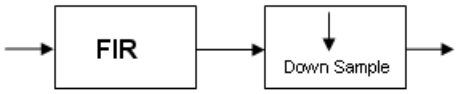


An analogous approach must be taken to avoid *imaging* when performing interpolation on a sampled signal. For more information about the effects of decimation and interpolation on a sampled signal, see “References” on page 6-4.

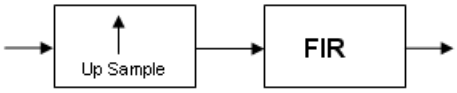
The following list summarizes some guidelines and general requirements regarding decimation and interpolation:

- By the Nyquist Theorem, for band-limited signals, the sampling frequency must be at least twice the bandwidth of the signal. For example, if you have a lowpass filter with the highest frequency of 10 MHz, and a sampling frequency of 60 MHz, the highest frequency that can be handled by the system without aliasing is 60/2=30, which is greater than 10. You could safely set M=2 in this case, since (60/2)/2=15, which is still greater than 10.

- If you wish to decimate a signal which does not meet the frequency criteria, you can either:
  - Interpolate first, and then decimate
  - When decimating, you should apply the filter first, then perform the decimation. When interpolating a signal, you should interpolate first, then filter the signal.
- Typically in decimation of a signal a filter is applied first, thereby allowing decimation without aliasing, as shown in the following figure:



- Conversely, a filter is typically applied after interpolation to avoid imaging:



- M must be an integer. Although, if you wish to obtain an M of 4/5, you could interpolate by 4, and then decimate by 5, provided that frequency restrictions are met. This type of multirate filter will be referred to as a *sample rate converter* in the documentation that follows.

Multirate filters are most often used in stages. This technique is introduced in the following section.

References

[1] Fliege, N.J., *Multirate Digital Signal Processing*, John Wiley and Sons, 1994.

[2] Harris, Fredric J, *Multirate Signal Processing for Communication Systems*, Prentice Hall PTR, 2004.

[3] Hogenauer, E. B., "An Economical Class of Digital Filters for Decimation and Interpolation," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-29, No. 2, April 1981, pp. 155-162.

[4] Lyons, Richard G., *Understanding Digital Signal Processing*, Prentice Hall PTR, 2004

[5] Mitra, S.K., *Digital Signal Processing*, McGraw-Hill, 1998.

[6] Orfanidis, S.J., *Introduction to Signal Processing*, Prentice-Hall, Inc., 1996.

Multistage Filters

In this section...
“Why Are Multistage Filters Needed?” on page 6-5
“Optimal Multistage Filters in DSP System Toolbox” on page 6-5

Why Are Multistage Filters Needed?

Typically used with multirate filters, *multistage filters* can bring efficiency to a particular filter implementation. Multistage filters are composed of several filters. These different parts of the multistage filter, called *stages*, are connected in a cascade or in parallel. However such a design can conserve resources in many cases. There are many different uses for a multistage filter. One of these is a filter requirement that includes a very narrow transition width. For example, you need to design a lowpass filter where the difference between the pass frequency and the stop frequency is .01 (normalized). For such a requirement it is possible to design a single filter, but it will be very long (containing many coefficients) and very costly (having many multiplications and additions per input sample). Thus, this single filter may be so costly and require so much memory, that it may be impractical to implement in certain applications where there are strict hardware requirements. In such cases, a multistage filter is a great solution. Another application of a multistage filter is for a mulitrate system, where there is a decimator or an interpolator with a large factor. In these cases, it is usually wise to break up the filter into several multirate stages, each comprising a multiple of the total decimation/interpolation factor.

Optimal Multistage Filters in DSP System Toolbox

As described in the previous section, within a multirate filter each interconnected filter is called a *stage*. While it is possible to design a multistage filter manually, it is also possible to perform automatic optimization of a multistage filter automatically. When designing a filter manually it can be difficult to guess how many stages would provide an optimal design, optimize each stage, and then optimize all the stages together. DSP System Toolbox software enables you to create a Specifications Object, and then design a filter using multistage as an option. The rest of the work is done automatically. Not only does DSP System Toolbox software determine the optimal number of stages, but it also optimizes the total filter solution.

Compare Single-Rate/Single-Stage Filters with Multirate/ Multistage Filters

This example shows the efficiency gains that are possible when using multirate and multistage filters for certain applications. In this case a distinct advantage is achieved over regular linear-phase equiripple design when a narrow transition-band width is required. A more detailed treatment of the key points made here can be found in the example entitled “Efficient Narrow Transition-Band FIR Filter Design”.

Single-Rate/Single-Stage Equiripple Design

Consider the following design specifications for a lowpass filter (where the ripples are given in linear units):

```
Fpass = 0.13;   % Passband edge
Fstop = 0.14;   % Stopband edge
Rpass = 0.001;  % Passband ripple, 0.0174 dB peak to peak
Rstop = 0.0005; % Stopband ripple, 66.0206 dB minimum attenuation
Hf = fdesign.lowpass(Fpass,Fstop,Rpass,Rstop,'linear');
```

A regular linear-phase equiripple design using these specifications can be designed by evaluating the following:

```
lpFilter = design(Hf,'equiripple','SystemObject',true);
```

When you determine the cost of this design, you can see that 695 multipliers are required.

```
cost(lpFilter)
```

```
ans = struct with fields:
                  NumCoefficients: 695
                        NumStates: 694
    MultiplicationsPerInputSample: 695
          AdditionsPerInputSample: 694
```

Reduce Computational Cost Using Mulitrate/Multistage Design

The number of multipliers required by a filter using a single state, single rate equiripple design is 694. This number can be reduced using multirate/multistage techniques. In any single-rate design, the number of multiplications required by each input sample is equal to the number of non-zero multipliers in the implementation. However, by using a multirate/multistage design, decimation and interpolation can be combined to lessen the computation required. For decimators, the average number of multiplications required per input sample is given by the number of multipliers divided by the decimation factor.

```
lpFilter_multi = design(Hf,'multistage','SystemObject',true);
```

You can then view the cost of the filter created using this design step, and you can see that a significant cost advantage has been achieved.

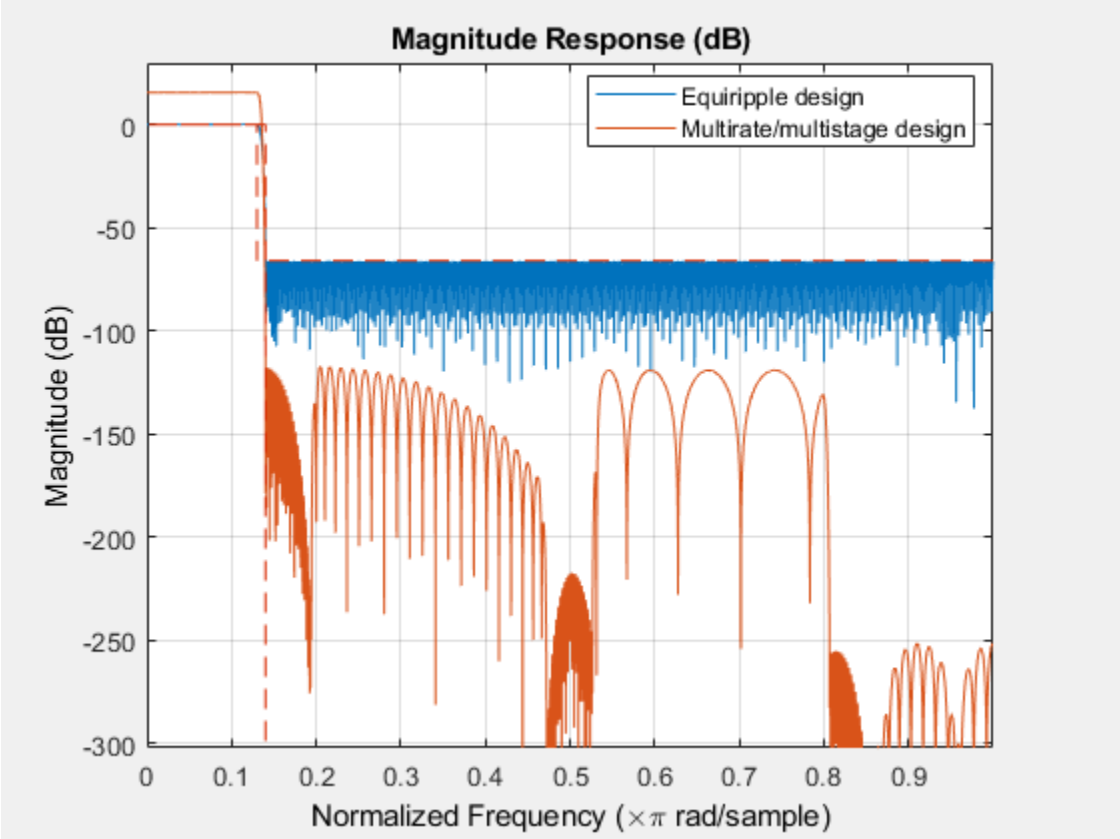
```
cost(lpFilter_multi)
```

```
ans = struct with fields:
                  NumCoefficients: 396
                        NumStates: 352
```

```
    MultiplicationsPerInputSample: 73
          AdditionsPerInputSample: 70.8333
```

Compare the Frequency Responses

```
fvt = fvtool(lpFilter,lpFilter_multi);
legend(fvt,'Equiripple design', 'Multirate/multistage design')
```



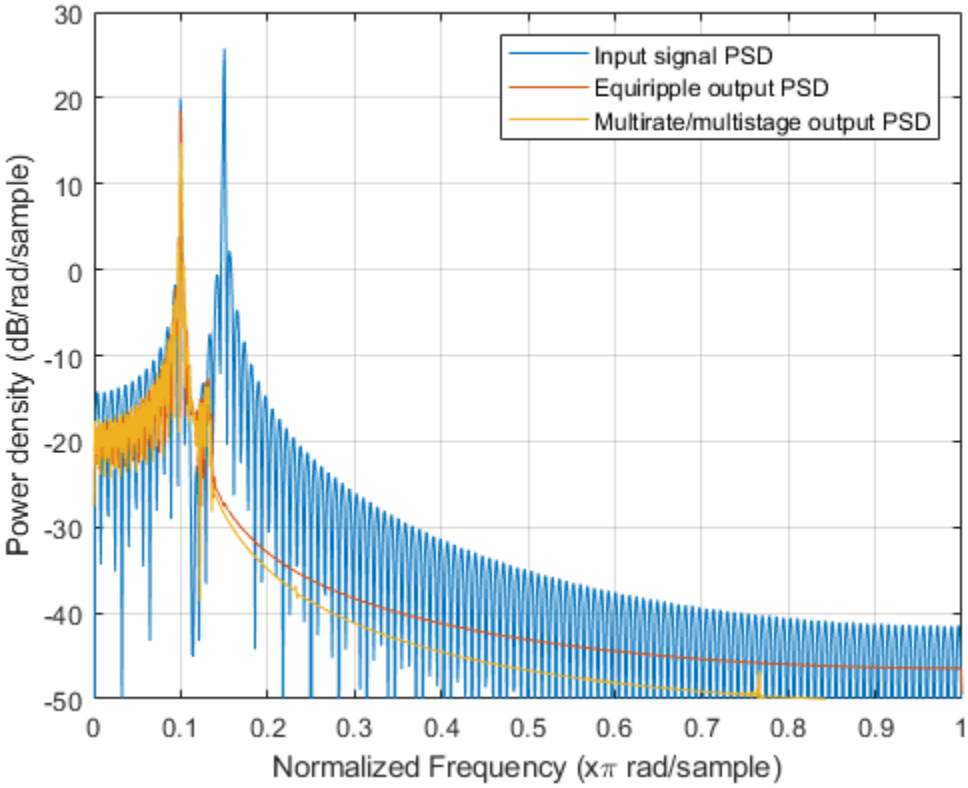
Notice that the stopband attenuation for the multistage design is about twice that of the other designs. This is because the decimators must attenuate out-of-band components by 66 dB in order to avoid aliasing that would violate the specifications. Similarly, the interpolators must attenuate images by 66 dB. You can also see that the passband gain for this design is no longer 0 dB, because each interpolator has a nominal gain (in linear units) equal to its interpolation factor, and the total interpolation factor for the three interpolators is 6.

Compare the Power Spectral Densities

You can check the performance of the multirate/multistage design by plotting the power spectral densities of the input and the various outputs, and you can see that the sinusoid at 0.4π is attenuated comparably by both the equiripple design and the multirate/multistage design.

```
n       = 0:1799;
x       = sin(0.1*pi*n') + 2*sin(0.15*pi*n');
y       = lpFilter(x);
y_multi = lpFilter_multi(x);
[Pxx,w]   = periodogram(x);
```

```
Pyy       = periodogram(y);
Pyy_multi = periodogram(y_multi);
plot(w/pi,10*log10([Pxx,Pyy,Pyy_multi]));
xlabel('Normalized Frequency (x\pi rad/sample)');
ylabel('Power density (dB/rad/sample)');
legend('Input signal PSD','Equiripple output PSD',...
    'Multirate/multistage output PSD')
axis([0 1 -50 30])
grid on
```



Design of Decimators/Interpolators

This example shows how to design filters for decimation and interpolation. Typically lowpass filters are used for decimation and for interpolation. When decimating, lowpass filters are used to reduce the bandwidth of a signal prior to reducing the sampling rate. This is done to minimize aliasing due to the reduction in the sampling rate. When interpolating, lowpass filters are used to remove spectral images from the low-rate signal. For general notes on lowpass filter design see the example on “Designing Low Pass FIR Filters”.

Input signal

The samples of the input signal we will be using are drawn from standard normal distribution to have a flat spectrum.

```
HSource = dsp.SignalSource('SamplesPerFrame', 500);
HSource.Signal = randn(1e6,1);      % Gaussian white noise signal
```

Design of Decimators

When decimating, the bandwidth of a signal is reduced to an appropriate value so that minimal aliasing occurs when reducing the sampling rate. Suppose a signal that occupies the full Nyquist interval (i.e. has been critically sampled) has a sampling rate of 800 Hz. The signal's energy extends up to 400 Hz. If we'd like to reduce the sampling rate by a factor of 4 to 200 Hz, significant aliasing will occur unless the bandwidth of the signal is also reduced by a factor of 4. Ideally, a perfect lowpass filter with a cutoff at 100 Hz would be used. In practice, several things will occur: The signal's components between 0 and 100 Hz will be slightly distorted by the passband ripple of a non-ideal lowpass filter; there will be some aliasing due to the finite stopband attenuation of the filter; the filter will have a transition band which will distort the signal in such band. The amount of distortion introduced by each of these effects can be controlled by designing an appropriate filter. In general, to obtain a better filter, a higher filter order will be required.

Let's start by designing a simple lowpass decimator with a decimation factor of 4.

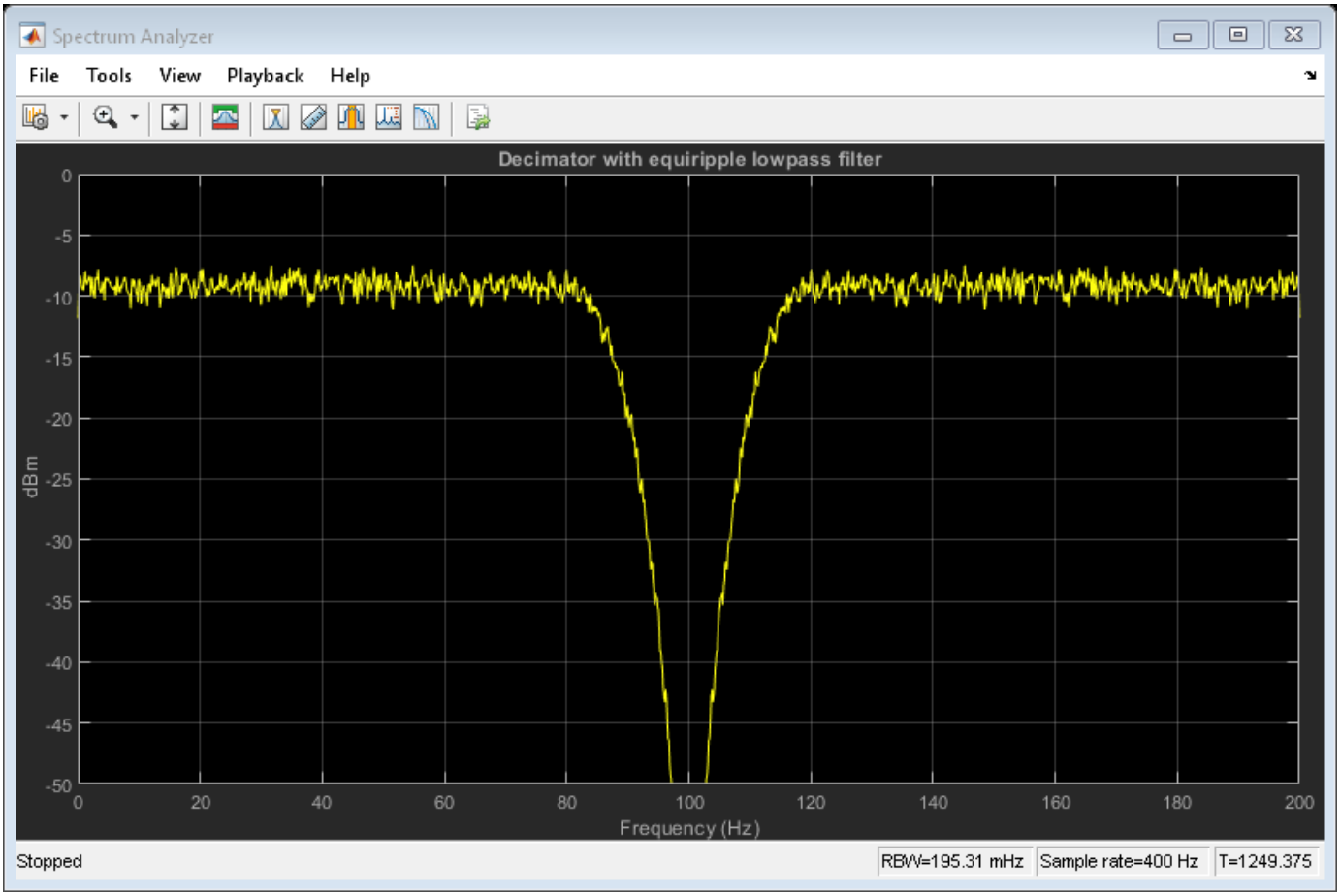
```
M   = 4;   % Decimation factor
Fp  = 80;  % Passband-edge frequency
Fst = 100;  % Stopband-edge frequency
Ap  = 0.1; % Passband peak-to-peak ripple
Ast = 80;  % Minimum stopband attenuation
Fs  = 800; % Sampling frequency
HfdDecim = fdesign.decimator(M,'lowpass',Fp,Fst,Ap,Ast,Fs)
```

```
HfdDecim =

  decimator with properties:

         MultirateType: 'Decimator'
              Response: 'Lowpass'
      DecimationFactor: 4
         Specification: 'Fp,Fst,Ap,Ast'
           Description: {4x1 cell}
    NormalizedFrequency: 0
                     Fs: 800
                  Fs_in: 800
                 Fs_out: 200
                  Fpass: 80
```

```
                Fstop: 100
                Apass: 0.1000
                Astop: 80
```

The specifications for the filter determine that a transition band of 20 Hz is acceptable between 80 and 100 Hz and that the minimum attenuation for out of band components is 80 dB. Also that the maximum distortion for the components of interest is 0.05 dB (half the peak-to-peak passband ripple). An equiripple filter that meets these specs can be easily obtained as follows:

```
HDecim = design(HfdDecim,'equiripple', 'SystemObject', true);
measure(HDecim)

HSpec = dsp.SpectrumAnalyzer(...                    % Spectrum scope
                    'PlotAsTwoSidedSpectrum', false, ...
                    'SpectralAverages', 50, 'OverlapPercent', 50, ...
                    'Title', 'Decimator with equiripple lowpass filter',...
                    'YLimits', [-50, 0], 'SampleRate', Fs/M*2);

while ~isDone(HSource)
    inputSig = HSource();   % Input
    decimatedSig = HDecim(inputSig);  % Decimator
    HSpec(upsample(decimatedSig,2));  % Spectrum
    % The upsampling is done to increase X-limits of SpectrumAnalyzer
    % beyond (1/M)*Fs/2 for better visualization
end
release(HSpec);
reset(HSource);


ans =

Sample Rate      : 800 Hz
Passband Edge    : 80 Hz
3-dB Point       : 85.621 Hz
6-dB Point       : 87.8492 Hz
Stopband Edge    : 100 Hz
Passband Ripple  : 0.092414 dB
Stopband Atten.  : 80.3135 dB
Transition Width : 20 Hz
```



It is clear from the measurements that the design meets the specs.

Using Nyquist Filters

Nyquist filters are attractive for decimation and interpolation due to the fact that a 1/M fraction of the number of coefficients is zero. The band of the Nyquist filter is typically set to be equal to the decimation factor, this centers the cutoff frequency at (1/M)*Fs/2. In this example, the transition band is centered around (1/4)*400 = 100 Hz.

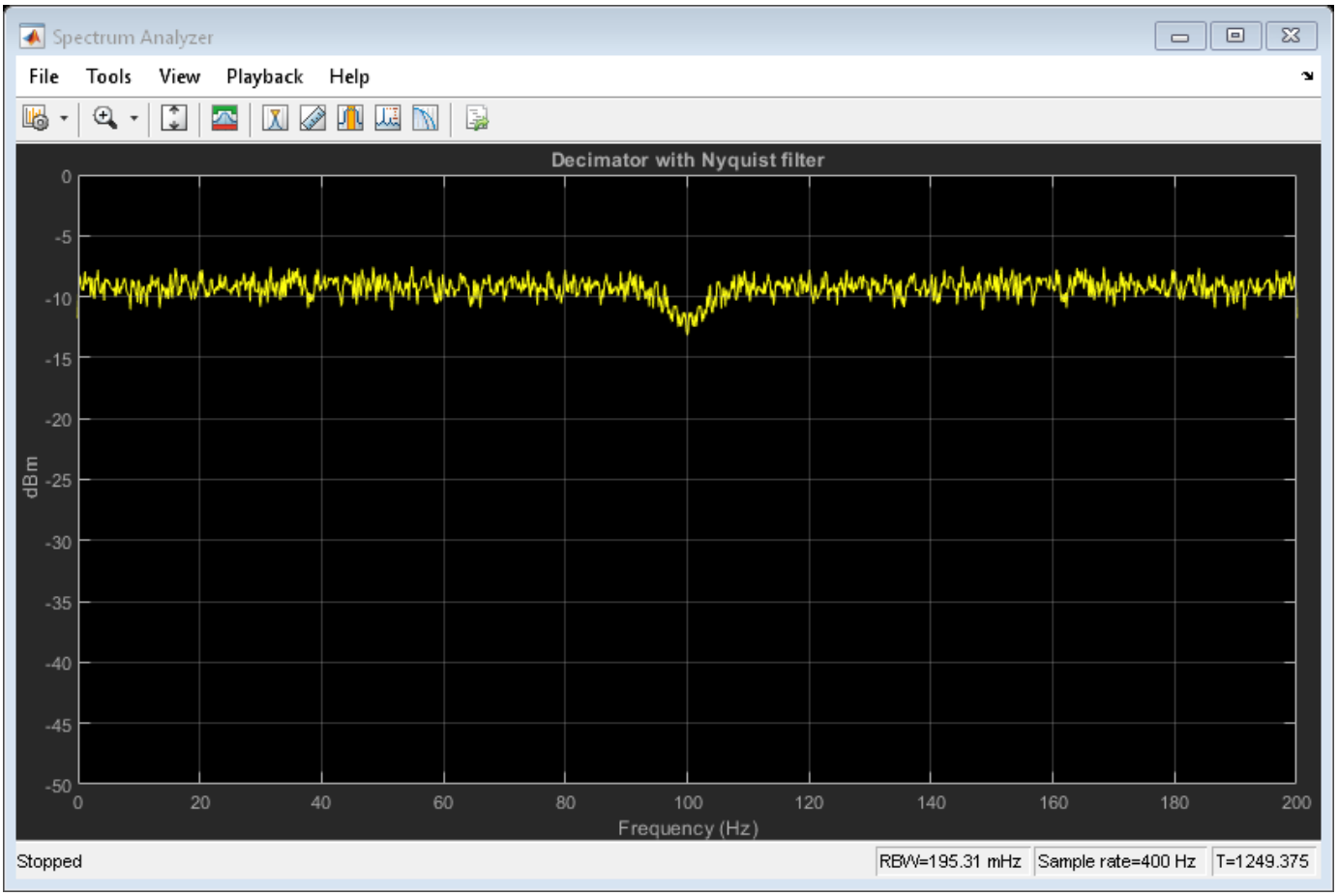
```
TW = 20; % Transition width of 20 Hz
HfdNyqDecim = fdesign.decimator(M,'nyquist',M,TW,Ast,Fs)
```

```
HfdNyqDecim =

  decimator with properties:

         MultirateType: 'Decimator'
              Response: 'Nyquist'
      DecimationFactor: 4
         Specification: 'TW,Ast'
           Description: {2x1 cell}
                  Band: 4
    NormalizedFrequency: 0
                    Fs: 800
```

```
                Fs_in: 800
               Fs_out: 200
      TransitionWidth: 20
                Astop: 80
```

A Kaiser window design can be obtained in a straightforward manner.

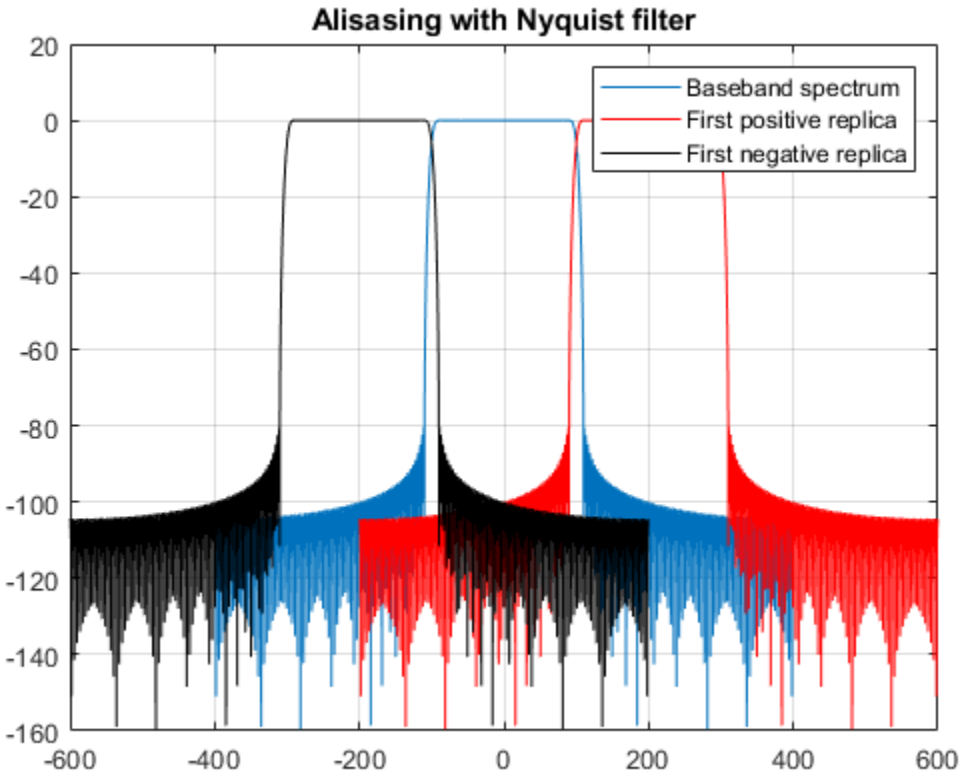
```
HNyqDecim = design(HfdNyqDecim,'kaiserwin','SystemObject', true);

HSpec2 = dsp.SpectrumAnalyzer('PlotAsTwoSidedSpectrum', false, ...
                          'SpectralAverages', 50, 'OverlapPercent', 50, ...
                          'Title', 'Decimator with Nyquist filter',...
                          'YLimits', [-50, 0],...
                          'SampleRate', Fs/M*2);       % Spectrum scope
while ~isDone(HSource)
    inputSig = HSource();   % Input
    decimatedSig = HNyqDecim(inputSig);   % Decimator
    HSpec2(upsample(decimatedSig,2));  % Spectrum
    % The upsampling is done to increase X-limits of SpectrumAnalyzer
    % beyond (1/M)*Fs/2 for better visualization
end
release(HSpec2);
reset(HSource);
```



Aliasing with Nyquist Filters

Suppose the signal to be filtered has a flat spectrum. Once filtered, it acquires the spectral shape of the filter. After reducing the sampling rate, this spectrum is repeated with replicas centered around multiples of the new lower sampling frequency. An illustration of the spectrum of the decimated signal can be found from:

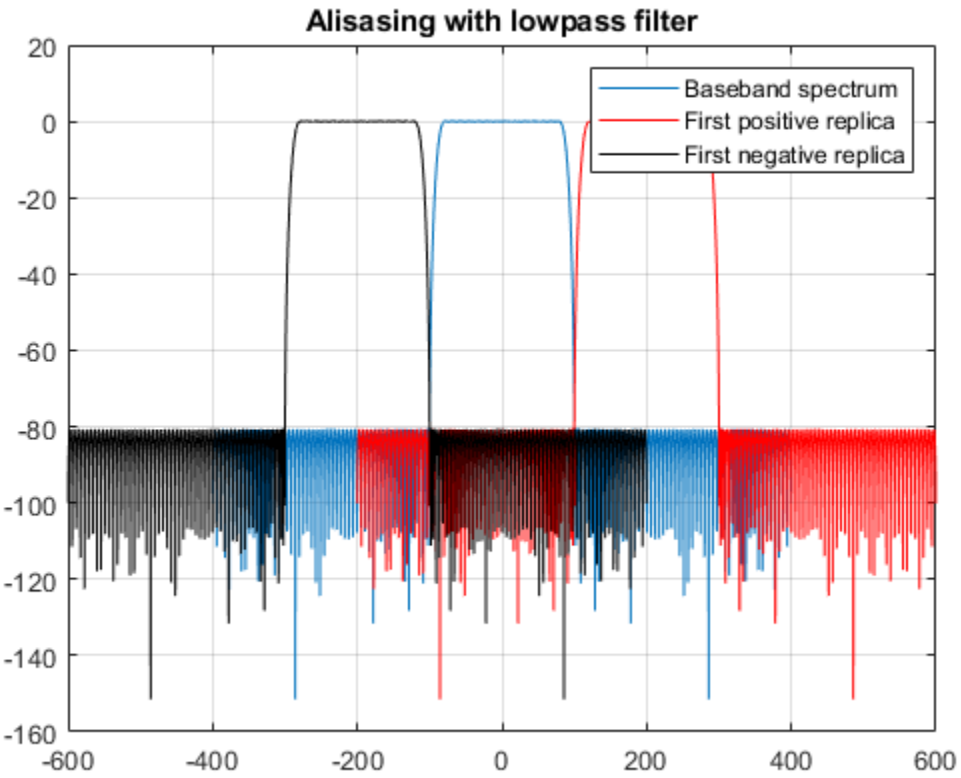
```
NFFT = 4096;
[H,f] = freqz(HNyqDecim,NFFT,'whole',Fs);
figure;
plot(f-Fs/2,20*log10(abs(fftshift(H))))
grid on
hold on
plot(f-Fs/M,20*log10(abs(fftshift(H))),'r-')
plot(f-Fs/2-Fs/M,20*log10(abs(fftshift(H))),'k-')
legend('Baseband spectrum', ...
    'First positive replica', 'First negative replica')
title('Alisasing with Nyquist filter');
fig = gcf;
fig.Color = 'White';
hold off
```



Note that the replicas overlap somewhat, so aliasing is introduced. However, the aliasing only occurs in the transition band. That is, significant energy (above the prescribed 80 dB) from the first replica only aliases into the baseband between 90 and 100 Hz. Since the filter was transitioning in this region anyway, the signal has been distorted in that band and aliasing there is not important.

On the other hand, notice that although we have used the same transition width as with the lowpass design from above, we actually retain a wider usable band (90 Hz rather than 80) when comparing this Nyquist design with the original lowpass design. To illustrate this, let's follow the same procedure to plot the spectrum of the decimated signal when the lowpass design from above is used

```
[H,f] = freqz(HDecim,NFFT,'whole',Fs);
figure;
plot(f-Fs/2,20*log10(abs(fftshift(H))))
grid on
hold on
plot(f-Fs/M,20*log10(abs(fftshift(H))),'r-')
plot(f-Fs/2-Fs/M,20*log10(abs(fftshift(H))),'k-')
legend('Baseband spectrum', ...
    'First positive replica', 'First negative replica')
title('Alisasing with lowpass filter');
fig = gcf;
fig.Color = 'White';
hold off
```



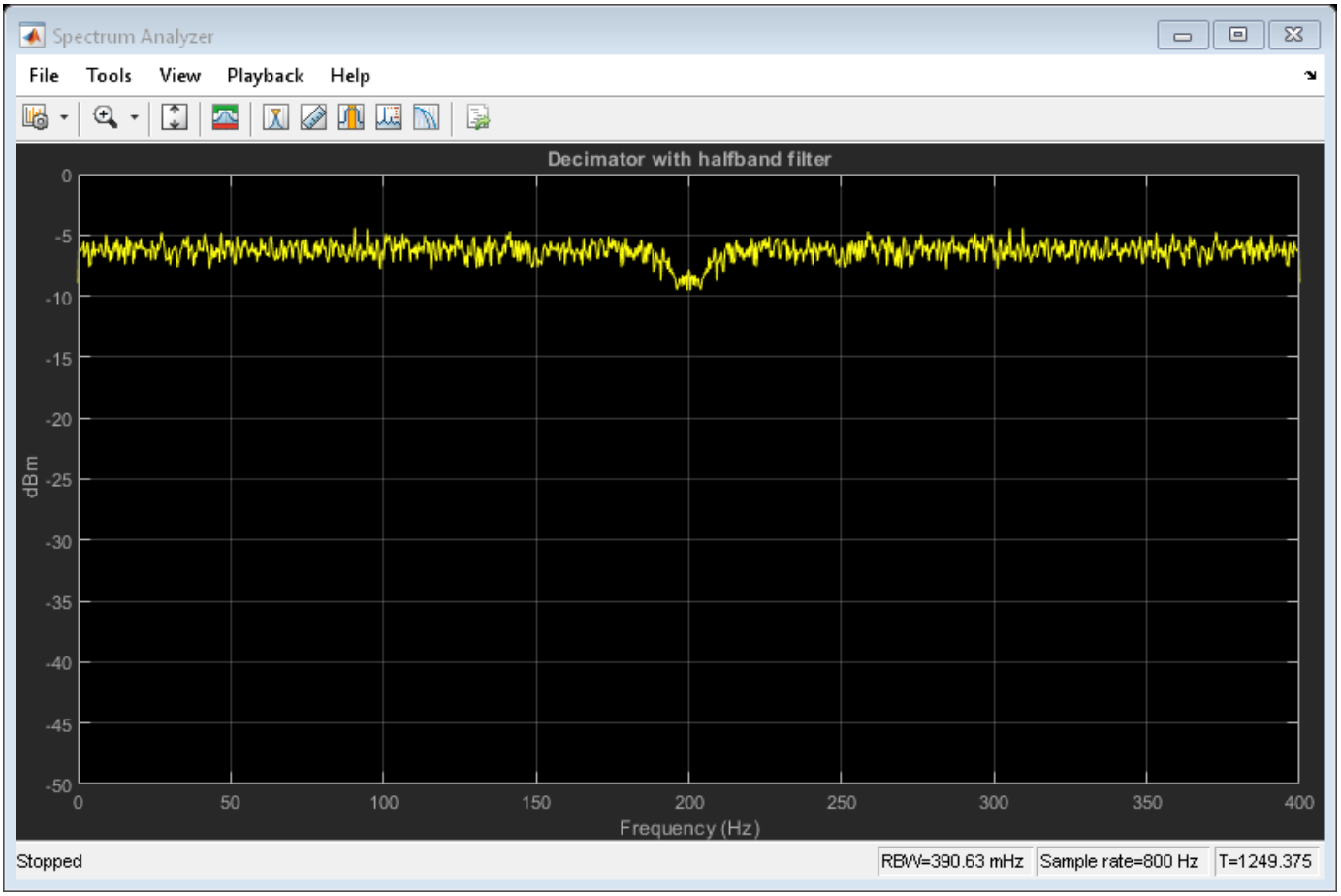
In this case, there is no significant overlap (above 80 dB) between replicas, however because the transition region started at 80 Hz, the resulting decimated signal has a smaller usable bandwidth.

Decimating by 2: Halfband Filters

When the decimation factor is 2, the Nyquist filter becomes a halfband filter. These filters are very attractive because just about half of their coefficients are equal to zero. Often, to design Nyquist

filters when the band is an even number, it is desirable to perform a multistage design that uses halfband filters in some/all of the stages.

```
HfdHBDecim = fdesign.decimator(2,'halfband');
HHBDecim = design(HfdHBDecim,'equiripple','SystemObject', true);
HSpec3 = dsp.SpectrumAnalyzer('PlotAsTwoSidedSpectrum', false, ...
                          'SpectralAverages', 50, 'OverlapPercent', 50, ...
                          'Title', 'Decimator with halfband filter',...
                          'YLimits', [-50, 0],...
                          'SampleRate', Fs);         % Spectrum scope
while ~isDone(HSource)
    inputSig = HSource();   % Input
    decimatedSig = HHBDecim(inputSig);   % Decimator
    HSpec3(upsample(decimatedSig,2));  % Spectrum
end
release(HSpec3);
reset(HSource);
```



As with other Nyquist filters, when halfbands are used for decimation, aliasing will occur only in the transition region.

Interpolation

When interpolating a signal, the baseband response of the signal should be left as unaltered as possible. Interpolation is obtained by removing spectral replicas when the sampling rate is increased.

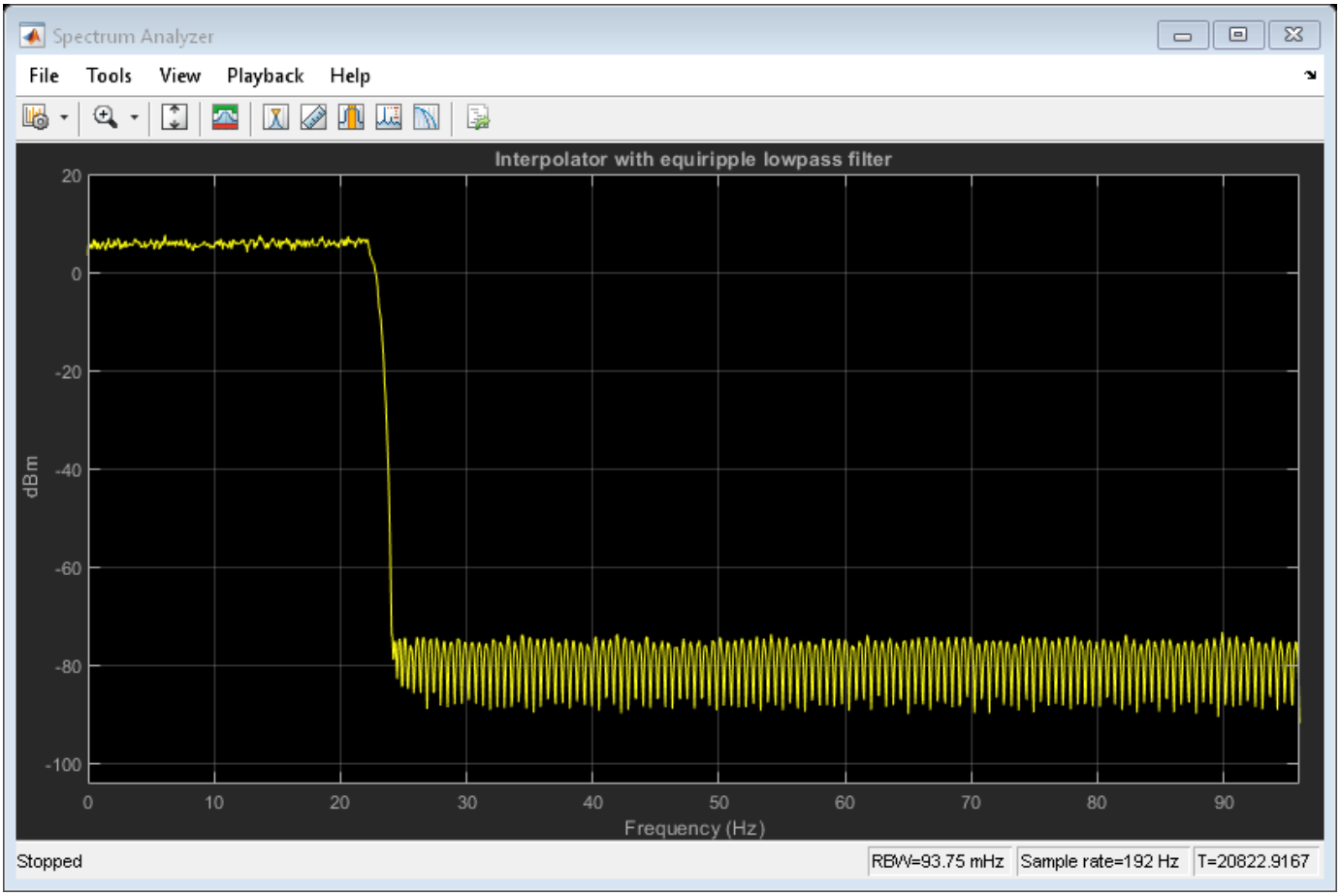
Suppose we have a signal sampled at 48 Hz. If it is critically sampled, there is significant energy in the signal up to 24 Hz. If we want to interpolate by a factor of 4, we would ideally design a lowpass filter running at 192 Hz with a cutoff at 24 Hz. As with decimation, in practice an acceptable transition width needs to be incorporated into the design of the lowpass filter used for interpolation along with passband ripple and a finite stopband attenuation. For example, consider the following specs:

```
L   = 4;   % Interpolation factor
Fp  = 22;  % Passband-edge frequency
Fst = 24;  % Stopband-edge frequency
Ap  = 0.1; % Passband peak-to-peak ripple
Ast = 80;  % Minimum stopband attenuation
Fs  = 48;  % Sampling frequency
HfdInterp = fdesign.interpolator(L,'lowpass',Fp,Fst,Ap,Ast,Fs*L)
```

```
HfdInterp =

  interpolator with properties:

         MultirateType: 'Interpolator'
              Response: 'Lowpass'
    InterpolationFactor: 4
         Specification: 'Fp,Fst,Ap,Ast'
           Description: {4x1 cell}
    NormalizedFrequency: 0
                    Fs: 192
                 Fs_in: 48
                Fs_out: 192
                 Fpass: 22
                 Fstop: 24
                 Apass: 0.1000
                 Astop: 80
```

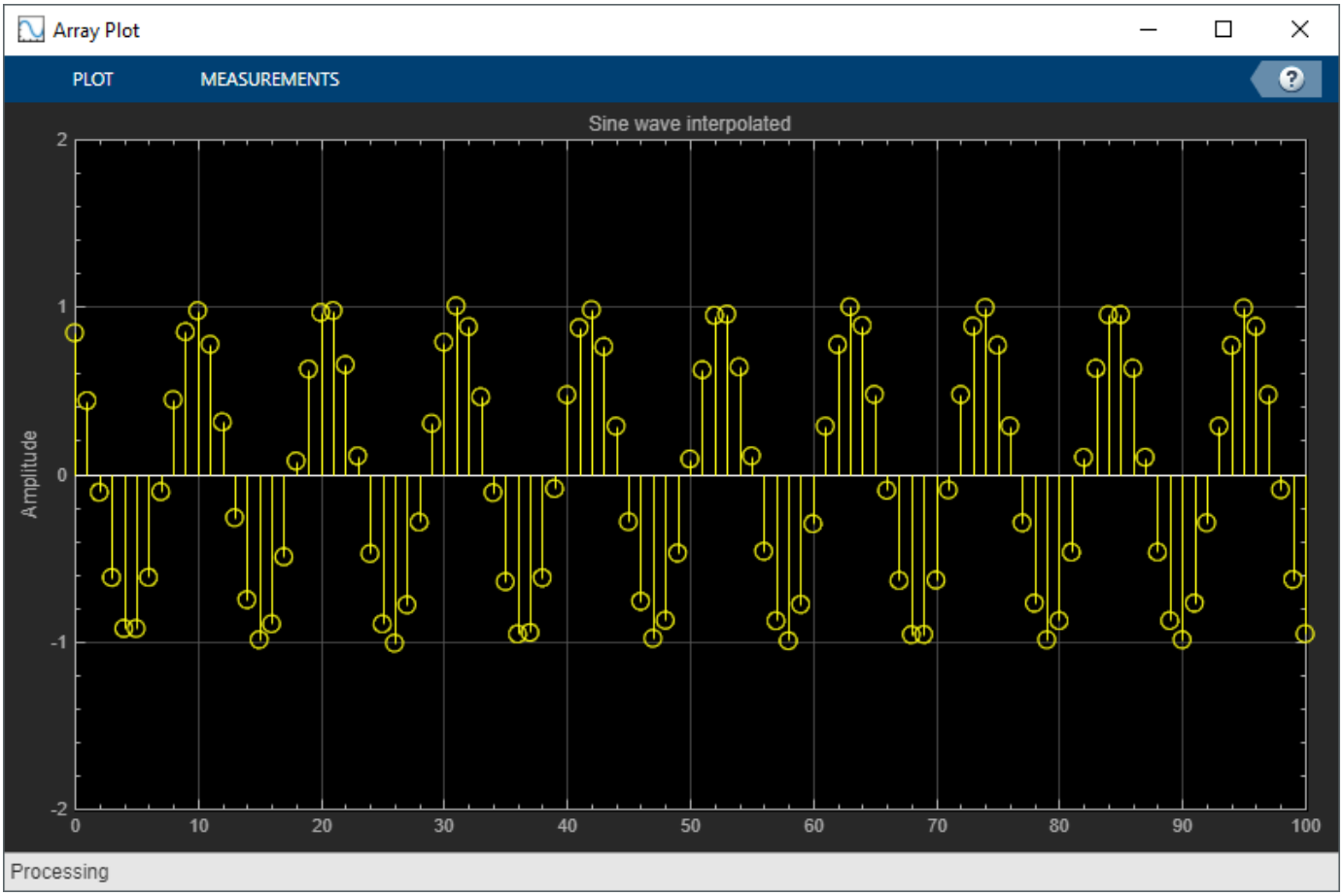
An equiripple design that meets the specs can be found in the same manner as with decimators

```
HInterp = design(HfdInterp,'equiripple','SystemObject', true);

HSpec4 = dsp.SpectrumAnalyzer('PlotAsTwoSidedSpectrum', false, ...
                 'SpectralAverages', 50, 'OverlapPercent', 50, ...
                 'Title', 'Interpolator with equiripple lowpass filter',...
                 'SampleRate', Fs*L);         % Spectrum scope
while ~isDone(HSource)
    inputSig = HSource();   % Input
    interpSig = HInterp(inputSig);   % Interpolator
    HSpec4(interpSig);  % Spectrum
end
release(HSpec4);
reset(HSource);
```



Notice that the filter has a gain of 6 dBm. In general interpolators will have a gain equal to the interpolation factor. This is needed for the signal being interpolated to maintain the same range after interpolation. For example,

```
release(HInterp);
HSin = dsp.SineWave('Frequency', 18, 'SampleRate', Fs, ...
                    'SamplesPerFrame', 100);
interpSig = HInterp(HSin());
HPlot = dsp.ArrayPlot('YLimits', [-2, 2], ...
                      'Title', 'Sine wave interpolated');
HPlot(interpSig(200:300)) % Plot the output
```



Note that although the filter has a gain of 4, the interpolated signal has the same amplitude as the original signal.

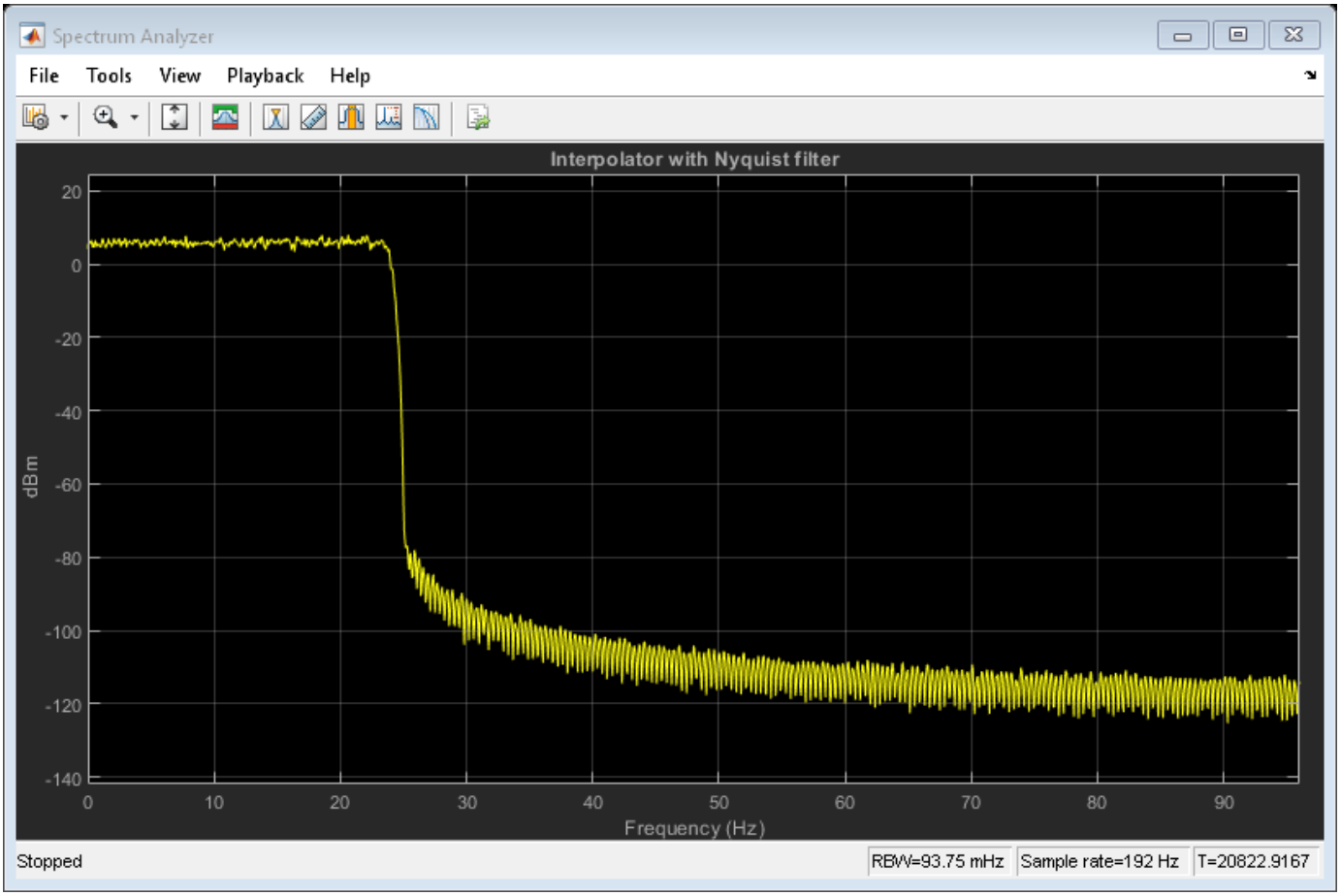
Use of Nyquist Filters for Interpolation

Similar to the decimation case, Nyquist filters are attractive for interpolation purposes. Moreover, given that there is a coefficient equal to zero every L samples, the use of Nyquist filters ensures that the samples from the input signal are retained unaltered at the output. This is not the case for other lowpass filters when used for interpolation (on the other hand, distortion may be minimal in other filters, so this is not necessarily a huge deal).

```
TW = 2;
HfdNyqInterp = fdesign.interpolator(L,'nyquist',L,TW,Ast,Fs*L)
HNyqInterp = design(HfdNyqInterp,'kaiserwin', 'SystemObject', true);

HSpec5 = dsp.SpectrumAnalyzer('PlotAsTwoSidedSpectrum', false, ...
                          'SpectralAverages', 30, 'OverlapPercent', 50, ...
                          'Title', 'Interpolator with Nyquist filter',...
                          'SampleRate', Fs*L);         % Spectrum scope
while ~isDone(HSource)
    inputSig = HSource();   % Input
    interpSig = HNyqInterp(inputSig);   % Decimator
    HSpec5(interpSig);  % Spectrum
end
```

```
release(HSpec5);
reset(HSource);
```

```
HfdNyqInterp =

  interpolator with properties:

         MultirateType: 'Interpolator'
              Response: 'Nyquist'
    InterpolationFactor: 4
         Specification: 'TW,Ast'
           Description: {2x1 cell}
                  Band: 4
    NormalizedFrequency: 0
                    Fs: 192
                 Fs_in: 48
                Fs_out: 192
       TransitionWidth: 2
                 Astop: 80
```



In an analogous manner to decimation, when used for interpolation, Nyquist filters allow some degree of imaging. That is, some frequencies above the cutoff frequency are not attenuated by the value of Ast. However, this occurs only in the transition band of the filter. On the other hand, once again a

wider portion of the baseband of the original signal is maintained intact when compared to a lowpass filter with stopband-edge at the ideal cutoff frequency when both filters have the same transition width.

Filter Banks

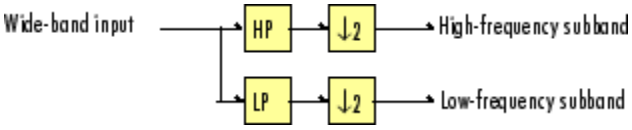
Multirate filters alter the sample rate of the input signal during the filtering process. Such filters are useful in both rate conversion and filter bank applications.

The Dyadic Analysis Filter Bank block decomposes a broadband signal into a collection of subbands with smaller bandwidths and slower sample rates. The Dyadic Synthesis Filter Bank block reconstructs a signal decomposed by the Dyadic Analysis Filter Bank block.

To use a dyadic synthesis filter bank to perfectly reconstruct the output of a dyadic analysis filter bank, the number of levels and tree structures of both filter banks *must* be the same. In addition, the filters in the synthesis filter bank *must* be designed to perfectly reconstruct the outputs of the analysis filter bank. Otherwise, the reconstruction will not be perfect.

Dyadic Analysis Filter Banks

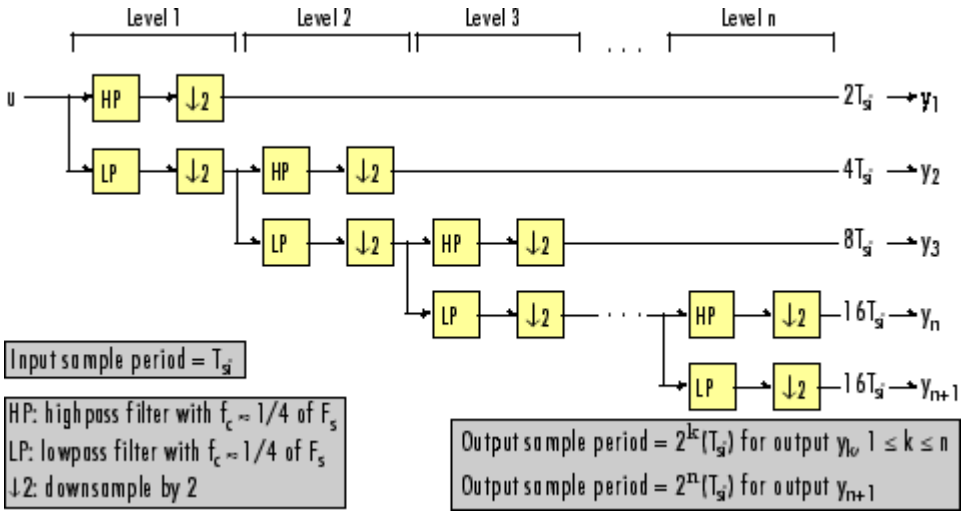
Dyadic analysis filter banks are constructed from the following basic unit. The unit can be cascaded to construct dyadic analysis filter banks with either a symmetric or asymmetric tree structure.



Each unit consists of a lowpass (LP) and highpass (HP) FIR filter pair, followed by a decimation by a factor of 2. The filters are halfband filters with a cutoff frequency of F_s / 4, a quarter of the input sampling frequency. Each filter passes the frequency band that the other filter stops.

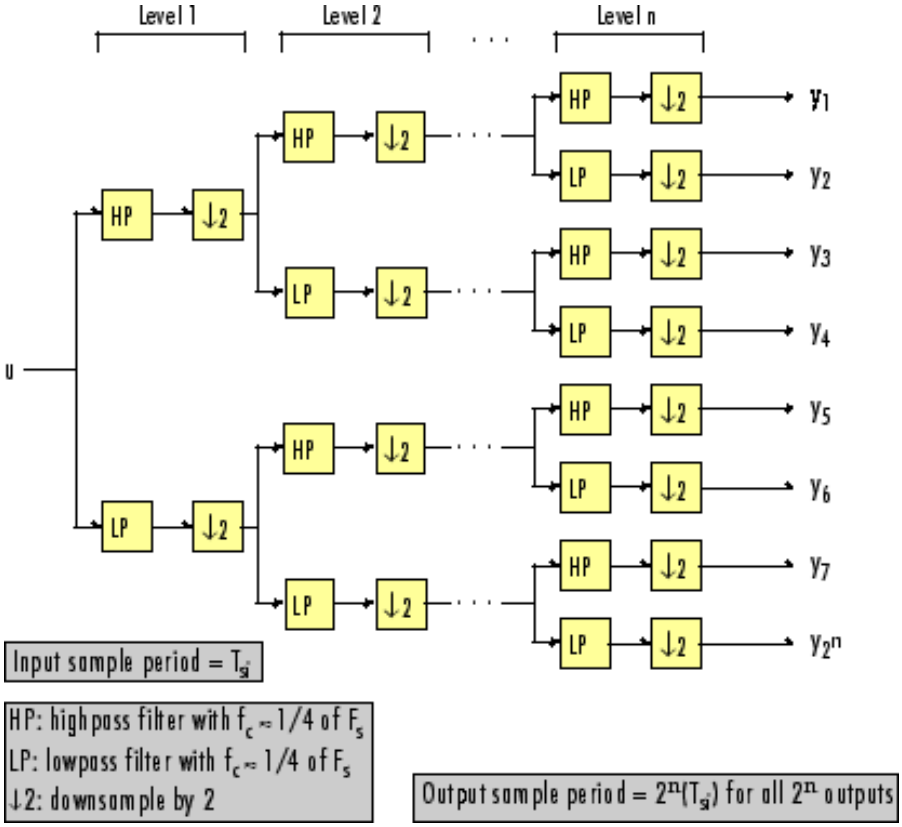
The unit decomposes its input into adjacent high-frequency and low-frequency subbands. Compared to the input, each subband has half the bandwidth (due to the half-band filters) and half the sample rate (due to the decimation by 2).

Note The following figures illustrate the *concept* of a filter bank, but *not* how the block implements a filter bank; the block uses a more efficient polyphase implementation.



n-Level Asymmetric Dyadic Analysis Filter Bank

Use the above figure and the following figure to compare the two tree structures of the dyadic analysis filter bank. Note that the asymmetric structure decomposes only the low-frequency output from each level, while the symmetric structure decomposes the high- and low-frequency subbands output from each level.



n-Level Symmetric Dyadic Analysis Filter Bank

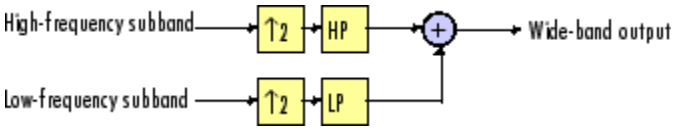
The following table summarizes the key characteristics of the symmetric and asymmetric dyadic analysis filter bank.

Notable Characteristics of Asymmetric and Symmetric Dyadic Analysis Filter Banks

Characteristic	N-Level Symmetric	N-Level Asymmetric
Low- and High-Frequency Subband Decomposition	All the low-frequency and high-frequency subbands in a level are decomposed in the next level.	Each level's low-frequency subband is decomposed in the next level, and each level's high-frequency band is an output of the filter bank.
Number of Output Subbands	2^n	n+1
Bandwidth and Number of Samples in Output Subbands	For an input with bandwidth BW and N samples, all outputs have bandwidth BW / 2^n and N / 2^n samples.	For an input with bandwidth BW and N samples, y_k has the bandwidth BW_k, and N_k samples, where $BW_k = \begin{cases} BW/2^k & (1 \le k \le n) \\ BW/2^n & (k = n+1) \end{cases}$ $N_k = \begin{cases} N/2^k & (1 \le k \le n) \\ N/2^n & (k = n+1) \end{cases}$ The bandwidth of, and number of samples in each subband (except the last) is half those of the previous subband. The last two subbands have the same bandwidth and number of samples since they originate from the same level in the filter bank.
Output Sample Period	All output subbands have a sample period of $2^n(T_{si})$	Sample period of kth output $= \begin{cases} 2^k(T_{si}) & (1 \le k \le n) \\ 2^n(T_{si}) & (k = n+1) \end{cases}$ Due to the decimations by 2, the sample period of each subband (except the last) is twice that of the previous subband. The last two subbands have the same sample period since they originate from the same level in the filter bank.
Total Number of Output Samples	The total number of samples in all of the output subbands is equal to the number of samples in the input (due to the decimations by 2 at each level).	
Wavelet Applications	In wavelet applications, the highpass and lowpass wavelet-based filters are designed so that the aliasing introduced by the decimations are exactly canceled in reconstruction.	

Dyadic Synthesis Filter Banks

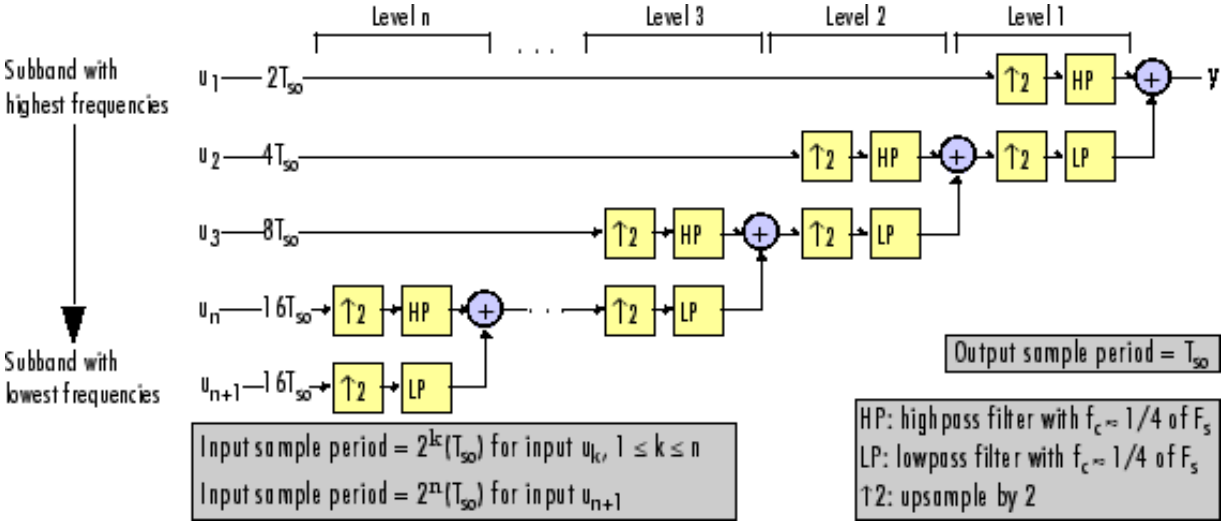
Dyadic synthesis filter banks are constructed from the following basic unit. The unit can be cascaded to construct dyadic synthesis filter banks with either a asymmetric or symmetric tree structure as illustrated in the figures entitled n-Level Asymmetric Dyadic Synthesis Filter Bank and n-Level Symmetric Dyadic Synthesis Filter Bank.



Each unit consists of a lowpass (LP) and highpass (HP) FIR filter pair, preceded by an interpolation by a factor of 2. The filters are halfband filters with a cutoff frequency of F_s / 4, a quarter of the input sampling frequency. Each filter passes the frequency band that the other filter stops.

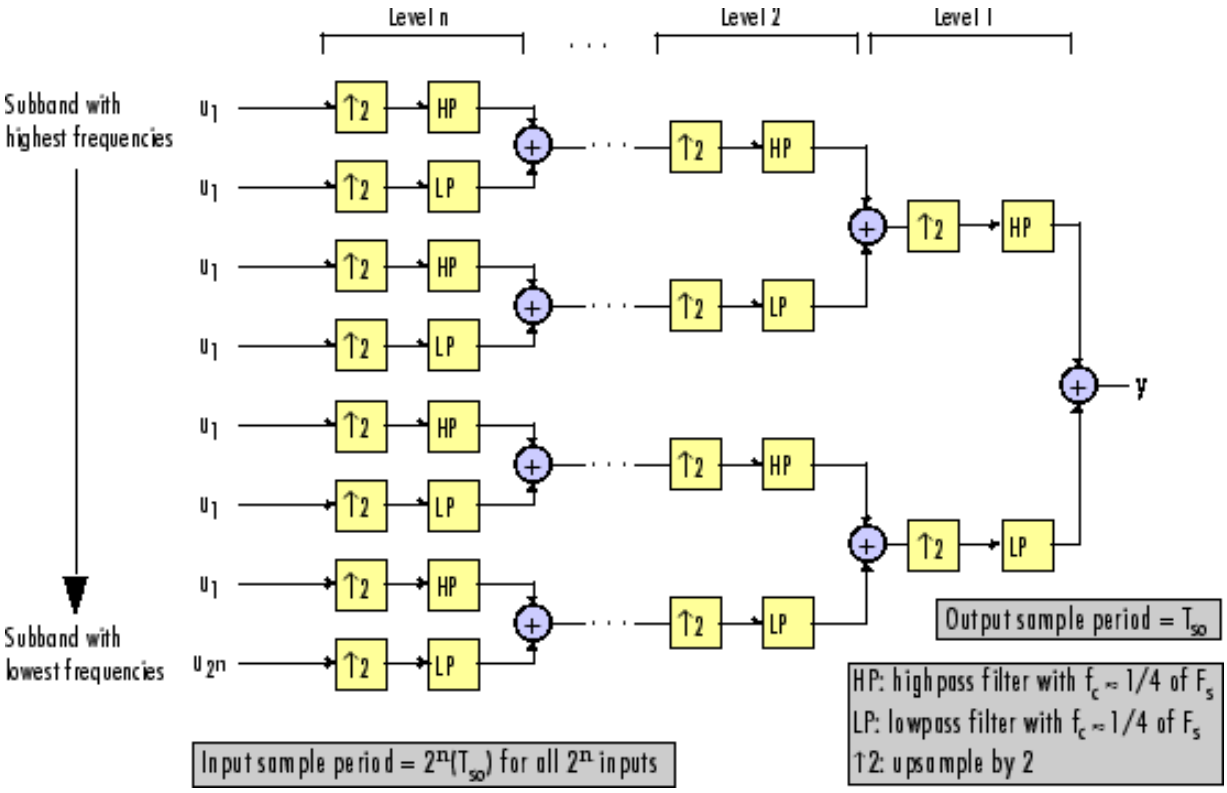
The unit takes in adjacent high-frequency and low-frequency subbands, and reconstructs them into a wide-band signal. Compared to each subband input, the output has twice the bandwidth and twice the sample rate.

Note The following figures illustrate the *concept* of a filter bank, but *not* how the block implements a filter bank; the block uses a more efficient polyphase implementation.



n-Level Asymmetric Dyadic Synthesis Filter Bank

Use the above figure and the following figure to compare the two tree structures of the dyadic synthesis filter bank. Note that in the asymmetric structure, the low-frequency subband input to each level is the output of the previous level, while the high-frequency subband input to each level is an input to the filter bank. In the symmetric structure, both the low- and high-frequency subband inputs to each level are outputs from the previous level.



n-Level Symmetric Dyadic Synthesis Filter Bank

The following table summarizes the key characteristics of symmetric and asymmetric dyadic synthesis filter banks.

Notable Characteristics of Asymmetric and Symmetric Dyadic Synthesis Filter Banks

Characteristic	N-Level Symmetric	N-Level Asymmetric
Input Paths Through the Filter Bank	Both the high-frequency and low-frequency input subbands to each level (except the first) are the outputs of the previous level. The inputs to the first level are the inputs to the filter bank.	The low-frequency subband input to each level (except the first) is the output of the previous level. The low-frequency subband input to the first level, and the high-frequency subband input to each level, are inputs to the filter bank.
Number of Input Subbands	2^n	n+1
Bandwidth and Number of Samples in Input Subbands	All inputs subbands have bandwidth BW / 2^n and N / 2^n samples, where the output has bandwidth BW and N samples.	For an output with bandwidth BW and N samples, the kth input subband has the following bandwidth and number of samples. $BW_k = \begin{cases} BW/2^k & (1 \le k \le n) \\ BW/2^n & (k = n+1) \end{cases}$ $N_k = \begin{cases} N/2^k & (1 \le k \le n) \\ N/2^n & (k = n+1) \end{cases}$
Input Sample Periods	All input subbands have a sample period of $2^n(T_{so})$, where the output sample period is T_{so}.	Sample period of kth input subband $= \begin{cases} 2^k(T_{so}) & (1 \le k \le n) \\ 2^n(T_{so}) & (k = n+1) \end{cases}$ where the output sample period is T_{so}.
Total Number of Input Samples	The number of samples in the output is always equal to the total number of samples in all of the input subbands.	
Wavelet Applications	In wavelet applications, the highpass and lowpass wavelet-based filters are carefully selected so that the aliasing introduced by the decimation in the dyadic *analysis* filter bank is exactly canceled in the reconstruction of the signal in the dyadic *synthesis* filter bank.	

For more information, see Dyadic Synthesis Filter Bank.

Multirate Filtering in Simulink

DSP System Toolbox software provides a collection of multirate filtering examples that illustrate typical applications of the multirate filtering blocks.

Multirate Filtering Examples	Description	Command for Opening Examples in MATLAB
Sigma-Delta A/D Converter	Illustrates analog-to-digital conversion using a sigma-delta algorithm implementation	`dspsdadc`
Wavelet Reconstruction and Noise Reduction	Uses the Dyadic Analysis Filter Bank and Dyadic Synthesis Filter Bank blocks to show both the perfect reconstruction property of wavelets and an application for noise reduction	`dspwavelet`

Dataflow

- “Dataflow Domain” on page 7-2
- “Model Multirate Signal Processing Systems Using Dataflow” on page 7-10
- “Multicore Simulation and Code Generation of Dataflow Domains” on page 7-12
- “Multicore Execution using Dataflow Domain” on page 7-19
- “Multicore Code Generation for Dataflow Domain” on page 7-27

Dataflow Domain

Using a dataflow domain, you can model and simulate a computationally intensive signal processing or multirate signal processing system. Dataflow domains simulate using a model of computation synchronous dataflow, which is data-driven and statically scheduled.

There are two primary reasons to use a dataflow domain in your model.

- Improve simulation throughput with multithreaded execution.

  A dataflow domain leverages the multicore CPU architecture of the host computer and can improve simulation speed significantly. The domain automatically partitions your model and simulates the system using multiple threads. By adding latency to your system, you can further increase concurrency and improve the simulation throughput of your model.
- Automatically infer signal sizes for frame-based multirate models.

  When the **Automatic frame-size calculation** parameter is enabled, dataflow domains automatically calculate frame sizes and insert buffers into your model, avoiding signal size propagation errors in multirate signal processing systems.

Specifying Dataflow Domains

To create a dataflow domain, use the Dataflow Subsystem block. The domain of the Dataflow Subsystem block is preconfigured.

To convert an existing subsystem into a Dataflow Subsystem:

1. In the **Execution** tab of the **Property Inspector**, select the **Set execution domain** check box.

   If the Property Inspector is not visible, in the **Modeling** tab, under **Design**, select **Property Inspector**. For more information on the Property Inspector, see “Setting Model and Block Properties with Property Inspector” (Simulink).
2. With the subsystem selected, set the **Domain** to `Dataflow`.

Inside the subsystem, in the lower left corner of the model canvas, there is now a icon, which indicates that the subsystem is a Dataflow subsystem.

Note Dataflow domains are supported only at the subsystem level. You cannot set the **Domain** of a top-level model to `Dataflow`.

Simulation of Dataflow Domains

Simulation of dataflow domains leverages the multicore CPU architecture of the host computer. It automatically partitions your model and simulates the subsystem using multiple threads.

The first time you simulate a dataflow domain, the simulation is single threaded. During this simulation, the software performs a cost analysis. The next time the model compiles, the software automatically partitions the system for multithreaded execution.

Each time you make a change inside the dataflow domain, the next simulation may be single threaded to allow the software to perform a new cost analysis. Following a cost analysis simulation, the next

time the model compiles, the software repartitions the domain and subsequent simulations are multithreaded.

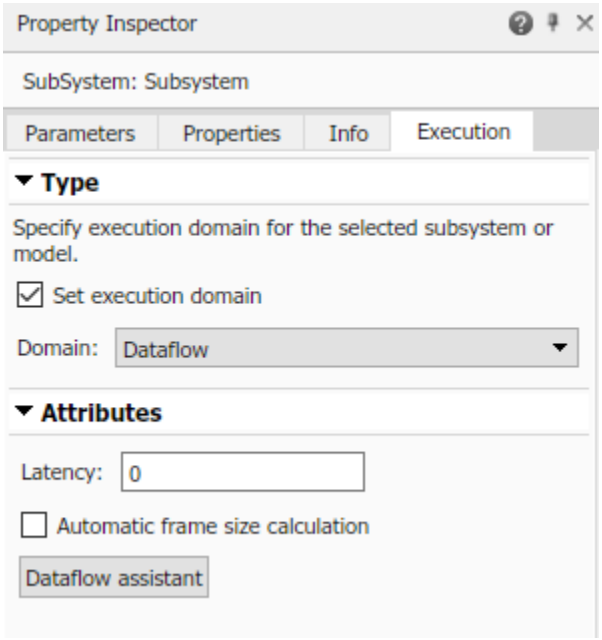
Some blocks and language features are not supported for multithreaded simulation. If a dataflow subsystem contains blocks or language features that do not support multithreaded simulation, Simulink issues a warning and the subsystem always simulates in a single thread.

If a dataflow subsystem contains blocks or language features that are not supported inside a dataflow subsystem, Simulink generates an error. For more information, see “Unsupported Simulink Software Features in Dataflow Domains” on page 7-8.

Dataflow Parameters

Latency

To increase the throughput of a system, it can be advantageous to increase the latency of a system. Specify the **Latency** value in the **Execution** tab of the Property Inspector.



To further improve the simulation performance, the Dataflow Simulation Assistant can recommend a latency value for simulation, as well as other model properties. Click the **Dataflow assistant** button to open the Dataflow Simulation Assistant.

The Dataflow Simulation Assistant suggests that you use the following settings for optimal simulation performance.

- Set **Compiler optimization level** to `Optimizations on (faster runs)`.

  ```
  set_param(gcs, 'SimCompilerOptimization', 'on')
  ```

- Disable the **Ensure responsiveness** parameter.

  ```
  set_param(gcs, 'SimCtrlC', 'off')
  ```

- Set **Wrap on overflow** to none.

  ```
  set_param(gcs, 'IntegerOverflowMsg', 'none')
  ```

- Set **Saturate on overflow** to none.

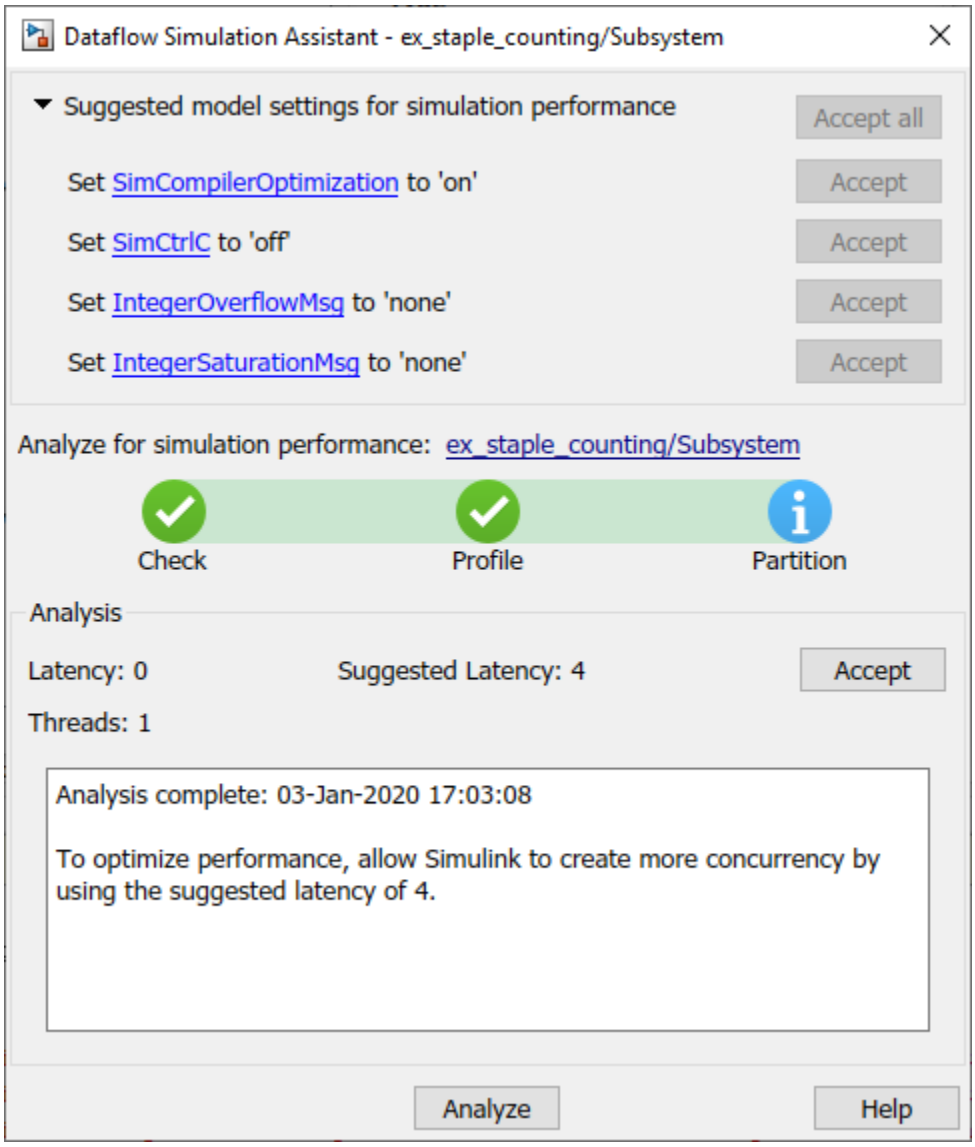
  ```
  set_param(gcs, 'IntegerSaturationMsg', 'none')
  ```

To accept the proposed model settings, next to **Suggested model settings for simulation performance**, click **Accept all**. Alternatively, you can expand the section to change the settings individually.

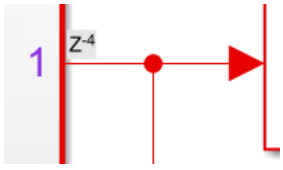
Click the **Analyze** button to analyze a dataflow domain for simulation performance and have the Dataflow Simulation Assistant suggest the optimal latency for your dataflow subsystem.

The dataflow analysis is a three-step process. During the first step, the analysis determines if it needs to repartition the model into threads by checking the dataflow subsystem for modeling changes since the last simulation. If partitioning is needed, the dataflow subsystem simulates with run-time profiling enabled for each block inside the subsystem in a single thread. In the last step, the assistant recompiles the model, which automatically partitions the subsystem into one or more threads to take advantage of concurrency in the model.

After the analysis completes, the assistant suggests a latency value that optimizes the throughput of the system for the multicore CPU architecture of the host computer.



The Dataflow Simulation Assistant indicates the number of threads the model will use if you apply the suggested latency. Click the **Accept** button to apply the suggested latency to the subsystem. When latency is introduced into a dataflow domain, the output of the dataflow subsystem is marked with a delay icon on the model canvas. Changes to the model within the dataflow subsystem may require a cost analysis and repartition.

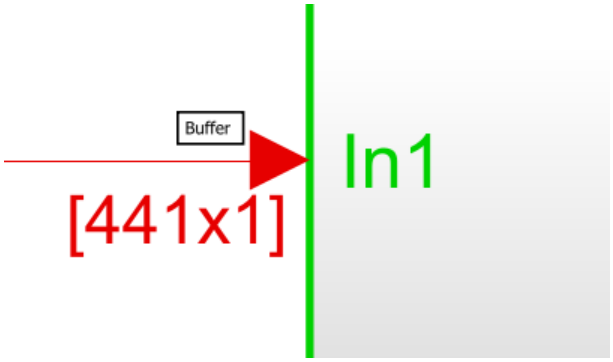


If you specify a latency greater than the latency suggested by the Dataflow Simulation Assistant, the additional delays are all inserted at the output of the subsystem. If you specify a latency value smaller than the value suggested by the Dataflow Simulation Assistant, the assistant warns that you are missing out on available concurrency.

For more information on types of parallelism in dataflow domains, see “Multicore Simulation and Code Generation of Dataflow Domains” on page 7-12.

Automatic Frame Size Calculation

Simulink can automatically calculate the frame sizes needed for each block in a frame-based signal processing system, and insert buffers where needed. To enable automatic frame size calculation in a Dataflow subsystem, select **Automatic frame size calculation** in the **Execution** tab of the Property Inspector.



Features Supported for Automatic Frame Size Calculation

Automatic frame size calculation is supported only for signals whose data types are one of the numeric types (built-in integer, double, single, or fixed-point). Signals using an enumerated type or whose data type is a bus are not supported.

Frame size calculation supports only two-dimensional signals.

Blocks Supported for Automatic Frame Size Calculation

The following blocks support automatic frame size calculation.

Note Not every configuration of these blocks supports automatic frame size calculation.

Simulink Blocks

- MATLAB System
- MATLAB Function
- Selector
- Add
- Delay
- Product
- From
- Goto
- Gain
- Vector Concatenate, Matrix Concatenate

- Terminator
- Mux
- Demux
- Math Function
- Data Type Conversion
- Abs
- Relational Operator
- Logical Operator
- Unit Delay
- Discrete FIR Filter
- Bitwise Operator
- Bias
- Signal Specification
- Squeeze
- Ground

DSP System Toolbox Blocks

- Submatrix
- Window Function
- Pad
- Buffer
- Minimum
- Mean
- Maximum
- Standard Deviation
- RMS
- Multiport Selector
- FFT
- IFFT
- Downsample
- Upsample
- Array-Vector Add
- Array-Vector Subtract
- Array-Vector Multiply
- Array-Vector Divide
- Flip
- FIR Decimation
- FIR Interpolation
- Biquad Filter

- Two-Channel Analysis Subband Filter
- Repeat
- LMS Filter
- Variable Selector
- FIR Rate Conversion

Unsupported Simulink Software Features in Dataflow Domains

Dataflow subsystems do not support the following Simulink software features.

Not Supported	Description
Variable-size signals	The software does not support variable-size signals. A variable-size signal is a signal whose size (number of elements in a dimension), in addition to its values, can change during model execution.
Referenced models	Model blocks are not supported in dataflow domains.
Nonvirtual Simulink subsystems, including Triggered Subsystem, Enabled Subsystem, and atomic subsystems	Only virtual subsystems are supported in dataflow domains.
Blocks with non-constant or non-inherited sample times	All sample times inside dataflow subsystems must be inherited (`-1`), or constant (`inf`).
Continuous blocks	Blocks in the “Continuous” (Simulink) library are not supported in dataflow domains. Simulink indicates in the model canvas at edit-time that these blocks are not supported by highlighting the block in orange.
Data Store blocks	Data Store Memory, Data Store Read, and Data Store Write blocks are not supported inside dataflow subsystems.
Subset of Simulink blocks	If a dataflow subsystem contains blocks or language features that are not supported, Simulink generates an error when the model compiles. For some blocks, such as Scope blocks, Simulink indicates in the model canvas at edit-time that they are not supported by highlighting the block in orange.
Stateflow® charts	Stateflow charts are not supported inside dataflow subsystems.
SimEvents® blocks	SimEvents blocks are not supported inside dataflow subsystems.

Not Supported	Description
HDL code generation	Only C/C++ code generation is supported for models with dataflow subsystems.

See Also

Dataflow Subsystem

More About

- “Multicore Simulation and Code Generation of Dataflow Domains” on page 7-12
- “Model Multirate Signal Processing Systems Using Dataflow” on page 7-10

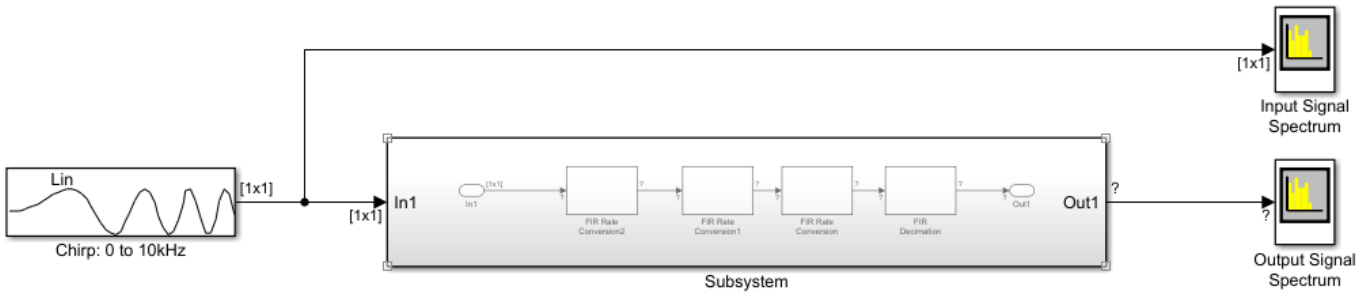
Model Multirate Signal Processing Systems Using Dataflow

This example shows how to model multirate signal processing systems using the dataflow subsystem. When you set the domain of a subsystem to dataflow and enable the **Automatic frame size calculation** parameter, the software calculates the signal sizes of frame-based multirate models and inserts buffers so that the model compiles with no frame size propagation errors.

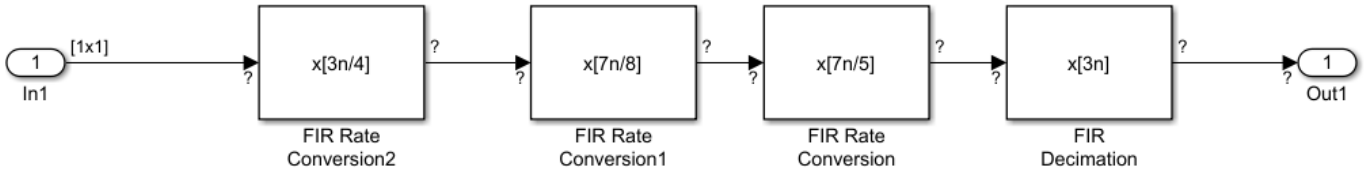
For more information on dataflow domains, see “Dataflow Domain” on page 7-2.

1 To begin, open the model.

```
addpath (fullfile(docroot, 'toolbox', 'dsp', 'examples'));
ex_multistage_filter
```



The subsystem of this model contains several rate conversion blocks.



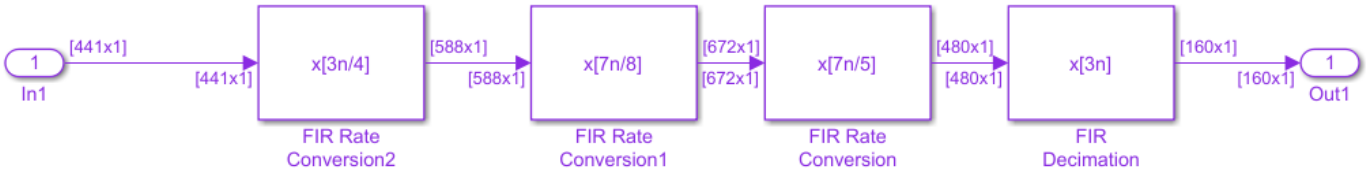
2 When you update the model diagram, Simulink generates an error due to a port dimension mismatch in the model. To fix this error, set the domain of the subsystem to dataflow.

3 If the Property Inspector is not visible, in the **Modeling** tab, under **Design**, select **Property Inspector**.

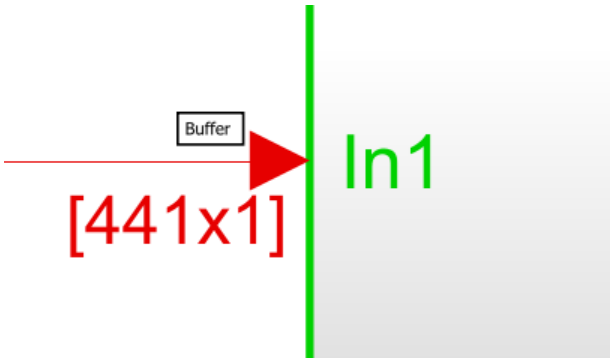
With the subsystem selected, in the **Execution** tab of the Property Inspector, select **Set execution domain**. Set the **Domain** to `Dataflow`.

4 Select **Automatic frame size calculation** to have the software automatically calculate frame sizes and insert buffers where needed.

5 Update the diagram again. The model now updates successfully.



Badges on the model canvas at the input of the subsystem indicate the places where buffers are inserted.



See Also

Dataflow Subsystem

More About

- “Dataflow Domain” on page 7-2

Multicore Simulation and Code Generation of Dataflow Domains

Simulation of Dataflow Domains

Simulation of dataflow domains leverages the multicore CPU architecture of the host computer. It automatically partitions your model and simulates the subsystem using multiple threads.

The first time you simulate a dataflow domain, the simulation is single threaded. During this simulation, the software performs a cost-analysis. The next time the model compiles, the software automatically partitions the system for multithreaded execution. Subsequent simulations are multithreaded.

Code Generation of Dataflow Domains

Dataflow domains support code generation for both single-core and multi-core targets. When all blocks inside a dataflow subsystem support multithreading, and the model is configured for multicore code generation, the generated code is multithreaded. During code generation, the dataflow subsystem is automatically partitioned according to the specified target hardware.

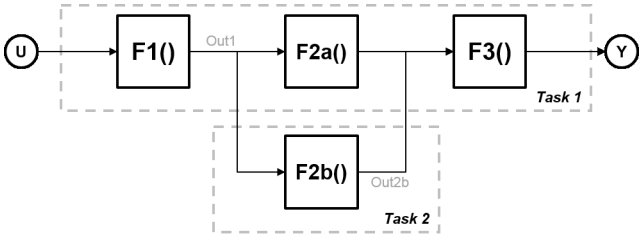
Types of Parallelism

In both simulation and code generation of models with dataflow domains, the software identifies possible concurrencies in your system, and partitions the dataflow domain using the following types of parallelism.

Task Parallelism

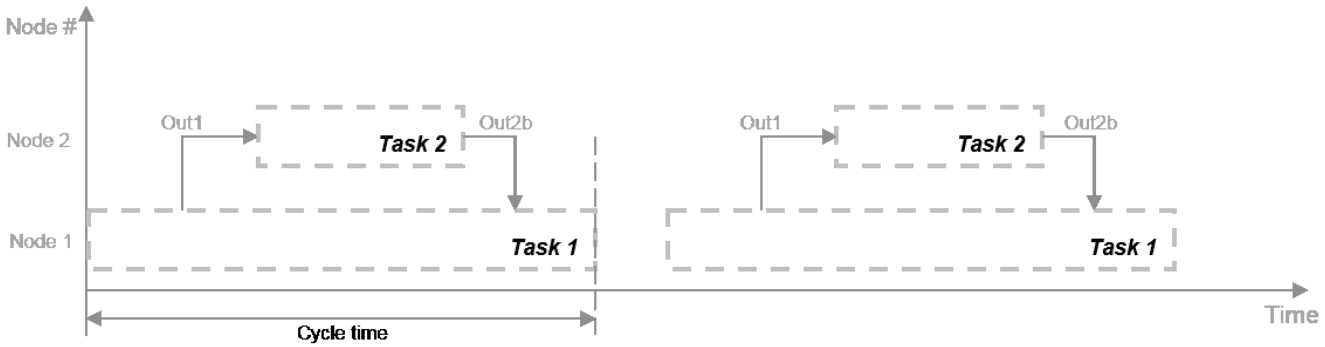
Task parallelism achieves parallelism by splitting up an application into multiple tasks. Task parallelism involves distributing tasks within an application across multiple processing nodes. Some tasks can have data dependency on others, so all tasks do not run at exactly the same time.

Consider a system that involves four functions. Functions F2a() and F2b() are in parallel, that is, they can run simultaneously. In task parallelism, you can divide your computation into two tasks. Function F2b() runs on a separate processing node after it gets data Out1 from Task 1, and it outputs back to F3() in Task 1.



The figure shows the timing diagram for this parallelism. Task 2 does not run until it gets data Out1 from Task 1. Hence, these tasks do not run completely in parallel. The time taken per processor cycle, known as cycle time, is

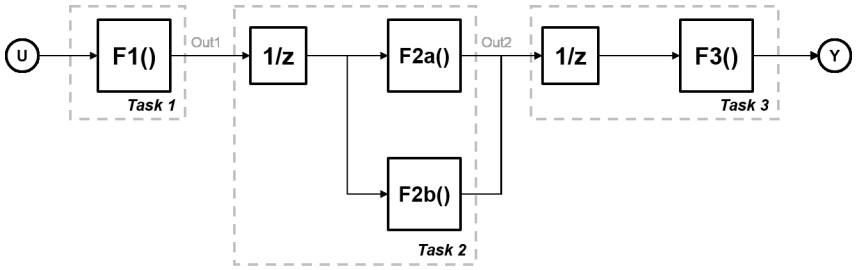
t = tF1 + max(tF2a, tF2b) + tF3.



Model Pipeline Execution (Pipelining)

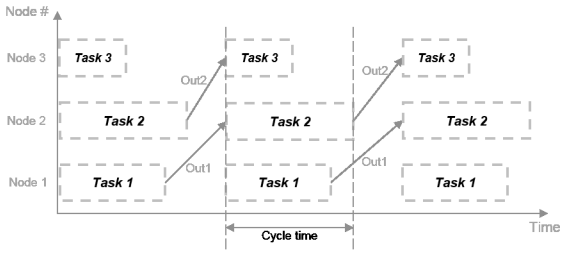
The software uses model pipeline execution, or pipelining, to work around the problem of task parallelism where threads do not run completely in parallel. This approach involves modifying the system to introduce delays between tasks where there is a data dependency.

In this figure, the system is divided into three tasks to run on three different processing nodes, with delays introduced between functions. At each time step, each task takes in the value from the previous time step by way of the delay.



Each task can start processing at the same time, as this timing diagram shows. These tasks are truly parallel and they are no longer serially dependent on each other in one processor cycle. The cycle time does not have any additions but is the maximum processing time of all the tasks.

t = max(Task1, Task2, Task3) = max(tF1, tF2a, tF2b, tF3).



Pipelining can be used when you can introduce delays artificially in your concurrently executing system. The resulting overhead due to this introduction must not exceed the time saved by pipelining.

Unfolding

When the cost analysis identifies a single block in a system that is computationally dominant, the system uses unfolding technology. Unfolding is a technique to improve throughput through

parallelization. The software duplicates the functionality of the computationally intensive block, divides the input data into multiple pieces, and the processor performs the same operation on each piece of data.

Unfolding is used in scenarios where it is possible to process each piece of input data independently without affecting the output, and the block is stateless or contains a finite number of states.

Improve Simulation Throughput with Multicore Simulation

This example shows how to improve simulation throughput of a system by simulating a subsystem with multiple threads. To enable automatic partitioning of a system and multithreaded simulation, set the **Domain** of the subsystem to `Dataflow`. For more information on dataflow domains, see “Dataflow Domain” on page 7-2

1. To begin, open the model.

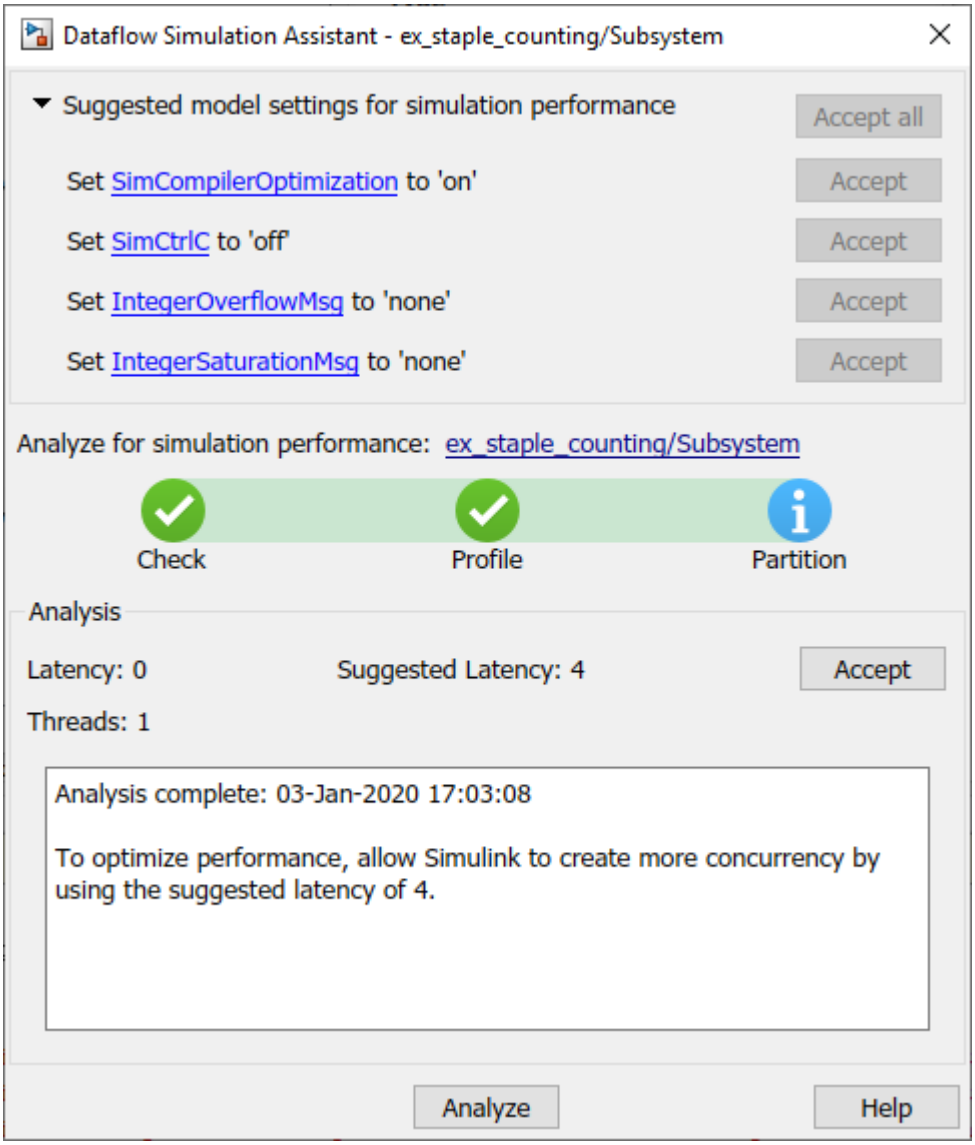
   ```
   addpath (fullfile(docroot, 'toolbox', 'dsp', 'examples'));
   ex_staple_counting
   ```
2. Simulate the model and observe the frame rate of the system in the Frame Rate Display block. This number indicates the number of frames per second that Simulink is able to process in a standard simulation.
3. To enable multithreaded simulation and improve the simulation throughput, set the domain of the subsystem to dataflow.

   If the Property Inspector is not visible, in the **Modeling** tab, under **Design**, select **Property Inspector**.

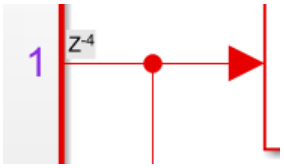
   With the subsystem selected, in the **Execution** tab of the Property Inspector, select **Set execution domain**. Set the **Domain** to `Dataflow`.
4. Sometimes, you can increase the available concurrency in your system by adding **Latency** to the system. To select an optimal latency value, use the Dataflow Simulation Assistant. Click the **Dataflow assistant** button to open the Dataflow Simulation Assistant.
5. In addition to suggesting a latency value, the Dataflow Simulation Assistant also suggests model settings for optimal simulation performance. In this example, to improve the simulation performance, the Dataflow Simulation Assistant suggests disabling the **Ensure responsiveness** parameter.

   To accept the proposed model settings, next to **Suggested model settings for simulation performance**, click **Accept all**.
6. Next, click the **Analyze** button. The Dataflow Simulation Assistant analyzes the subsystem for simulation performance and suggests an optimal latency for your Dataflow Subsystem.

   The dataflow analysis is a two-step process. During the first step, the dataflow subsystem simulates using a single-thread. During this simulation, the software performs a cost analysis. The next time the model compiles, the software automatically partitions the subsystem into one or more threads to take advantage of concurrency in the model. Subsequent simulations are multithreaded. The assistant suggests a latency value that optimizes the throughput of the system.



7 Click the **Accept** button to apply the suggested latency to the system. The Dataflow Simulation Assistant applies the latency to the model and indicates the number of threads the model will use during subsequent simulations. The latency of the system is indicated with a delay icon on the model canvas at the output of the subsystem.



8 Simulate the model again. Observe the improved simulation throughput from the multithreaded simulation in the Frame Rate Display block.

Generate Multicore Code from a Dataflow Subsystem

Configure Your Model for Multicore Code Generation

Code generation requires a Simulink Coder or an Embedded Coder® license. Single-core and multicore targets are supported.

Code generated for single-core targets generates nonvirtual subsystem code.

To generate multicore code, you must configure your model for concurrent execution. If you do not configure your model for concurrent execution, the generated code will be single threaded.

1. In **Configuration Parameters** > **Solver** > **Solver selection**, choose `Fixed-step` for the **Type** and `auto (Automatic solver selection)` for the **Solver**.
2. Select the **Allow tasks to execute concurrently on target** check box in the **Solver** pane under **Solver details**. Selecting this check box is optional for models referenced in the model hierarchy. When you select this option for a referenced model, Simulink allows each rate in the referenced model to execute as an independent concurrent task on the target processor.
3. In **Configuration Parameters** > **Code Generation** > **Interface** > **Advanced parameters**, clear the **MAT-file logging** check box.
4. Click **Apply** to apply the settings to the model.

Generate Multicore Code

To generate multicore code, the software performs cost analysis and partitions the dataflow domain based on your specified target. The partitioning of the dataflow domain may or may not match the partitioning during simulation.

The generated C code contains one `void(void)` function for each task or thread created by the dataflow subsystem. Each of these functions consists of:

- The C code corresponding to the blocks that were partitioned into that thread
- The code that is generated to handle how data is transferred between the threads.

  This can be in the form of pipeline delays or target-specific implementation of data synchronization semaphores.

The following multicore targets are supported for code generation.

- Linux®, Windows, and Mac OS desktop targets using `ert.tlc` and `grt.tlc`.
- Simulink Real-Time™ using `slrt.tlc`.
- Embedded Coder targets using Linux and VxWorks® operating systems.

Code generated for `grt.tlc` and `ert.tlc` desktop targets is multithreaded using OpenMP within the dataflow subsystem. Code generated for Embedded Coder targets is multithreaded using POSIX threads.

If your system contains blocks that do not support multithreaded execution, the generated code is single-threaded.

To build the model and generate code, press **Ctrl+B**.

In the generated code, you can observe calls to the threading API and the pipeline delays that were inserted into the model to create more concurrency.

The following example shows that there are two thread functions generated by dataflow subsystem, `ex_staple_counting_ThreadFcn0` and `ex_staple_counting_ThreadFcn1`, which are executed using OpenMP sections. These functions are part of the `dataflow_subsystem_output/step()` function.

```
static void ex_staple_counting_ThreadFcn0(void)
     {
       ...

      if (pipeStage_Concurrent0 >= 2) {
        /* Delay: '<S3>/TmpDelayBufferAtDraw Markers1Inport1' */
        memcpy(&ex_staple_counting_B.TmpDelayBufferAtDrawMarkers1I_i[0],
               &ex_staple_counting_DW.TmpDelayBufferAtDrawMarkers1I_i[0], 202176U *
               sizeof(real32_T));

        /* Delay: '<S3>/TmpDelayBufferAtDraw Markers1Inport2' */
        line_idx_1 = (int32_T)ex_staple_counting_DW.CircBufIdx * 100;
        memcpy(&ex_staple_counting_B.TmpDelayBufferAtDrawMarkers1Inp[0],
               &ex_staple_counting_DW.TmpDelayBufferAtDrawMarkers1Inp[line_idx_1],
               100U * sizeof(real_T));
...
       }
```

```
void ex_staple_counting_Concurrent0(void)
     {
...

     #pragma omp parallel num_threads(3)

       {
     #pragma omp sections

         {

     #pragma omp section

           {
             ex_staple_counting_ThreadFcn0();
           }

     #pragma omp section

           {
             ex_staple_counting_ThreadFcn1();
           }

     #pragma omp section

           {
             ex_staple_counting_ThreadFcn2();
           }
         }
       }
```

See Also

Dataflow Subsystem

More About

- “Dataflow Domain” on page 7-2
- “Multicore Programming with Simulink” (Simulink)

- “Optimize and Deploy on a Multicore Target” (Simulink)

Multicore Execution using Dataflow Domain

This example shows how to speed up execution of models using dataflow domain in Simulink. We use the digital up converter and digital down converter blocks to create a family radio service transmitter and receiver.

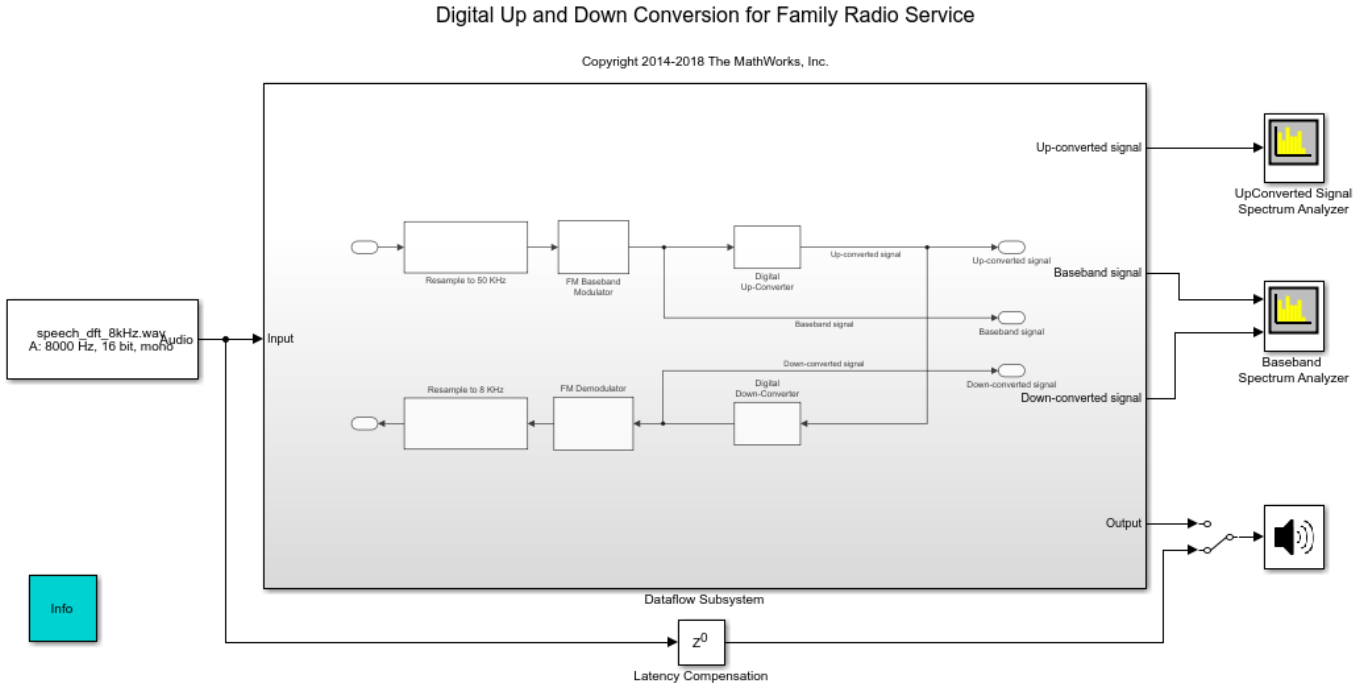
Introduction

Dataflow execution domain allows you to make use of multiple cores in the simulation of computationally intensive signal processing systems.

This example shows how to specify dataflow as the execution domain of a subsystem, improve simulation performance of the model, and generate multicore code.

Family Radio Service System

This example uses the Digital Up-Converter (DUC) and Digital Down-Converter (DDC) blocks to create a Family Radio Service (FRS) transmitter and receiver. The Digital Up-Converter (DUC) block converts a complex digital baseband signal to real passband signal. The Digital Down-Converter (DDC) block converts the digitized real signal back to a baseband complex signal. Open familyRadioServiceExample model.



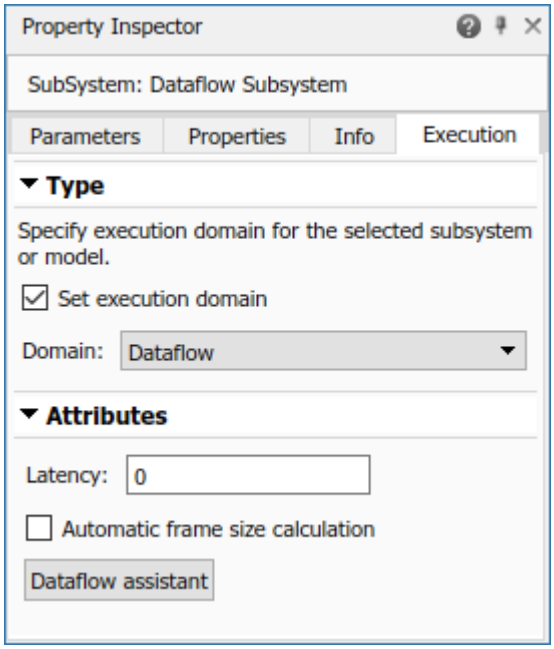
Simulate the model and measure the execution time. Execution time is measured using the output of the sim command which returns the simulation execution time of the model. To measure the time taken primarily for the Dataflow subsystem, comment out the Spectrum Analyzer blocks and Audio Device Writer block.

```
Simulation execution time for single-threaded model = 3.48s
```

Specify Dataflow Execution Domain

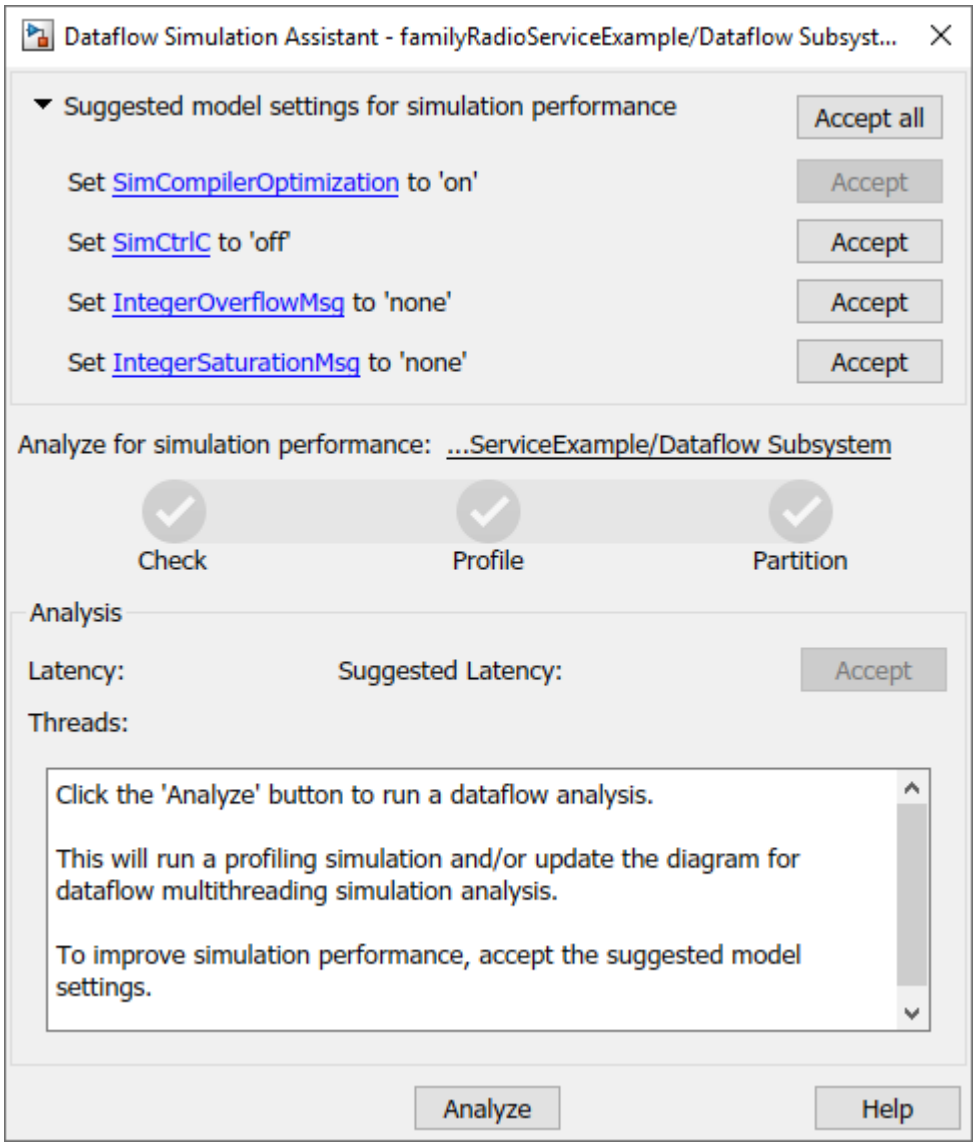
In Simulink, you specify dataflow as the execution domain for a subsystem by setting the Domain parameter to Dataflow using Property Inspector. You can view the Property Inspector for a subsystem, first by selecting the subsystem and then selecting View>Property Inspector. In the Property Inspector, you can set the domain to dataflow by selecting **Set domain specification** and then choosing "Dataflow" for "Domain" setting. You can also use Dataflow Subsystem block from the Dataflow library of DSP System toolbox to get a subsystem that is preconfigured with the dataflow execution domain.

The Dataflow subsystem automatically partitions the blocks for multicore execution.

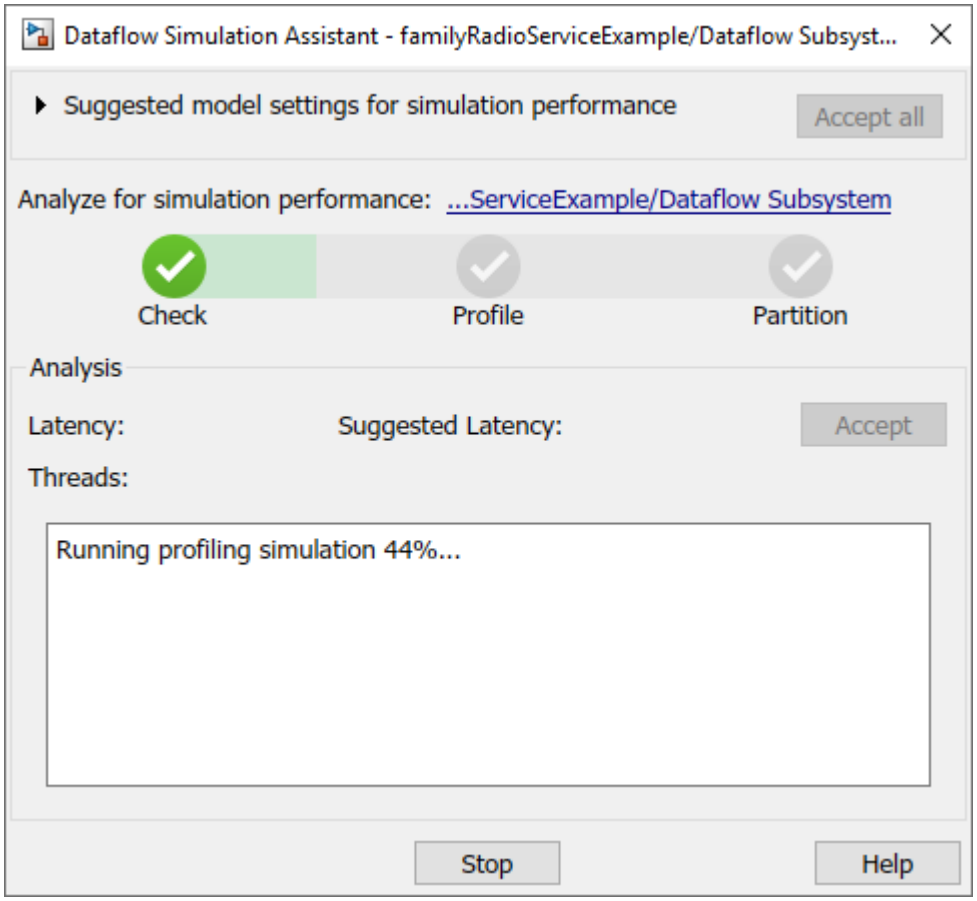


Multicore Simulation of Dataflow Domain

Dataflow domains automatically partition your model and simulate the system using multiple threads for better simulation performance. You can use the Dataflow Simulation Assistant to analyze the Dataflow Subsystem and further improve the simulation performance. You can open the assistant by clicking on the **Dataflow assistant** button below the **Automatic frame size calculation** parameter in Property Inspector.

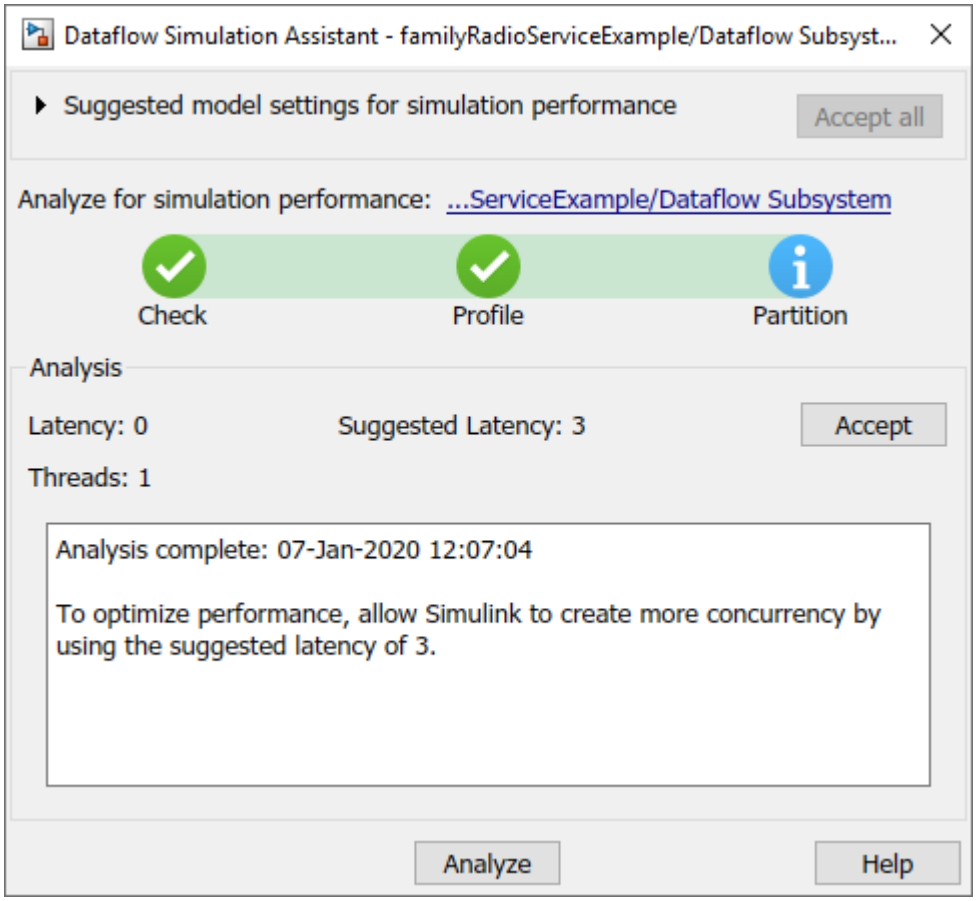


The Dataflow Simulation Assistant suggests changing model settings for optimal simulation performance. To accept the proposed model settings, next to **Suggested model settings for simulation performance**, click **Accept all**. Alternatively, you can expand the section to change the settings individually. In the Dataflow Simulation Assistant, click the **Analyze** button to start the analysis of the dataflow domain for simulation performance. Once the analysis is finished, the Dataflow Simulation Assistant shows how many threads the dataflow subsystem will use during simulation.



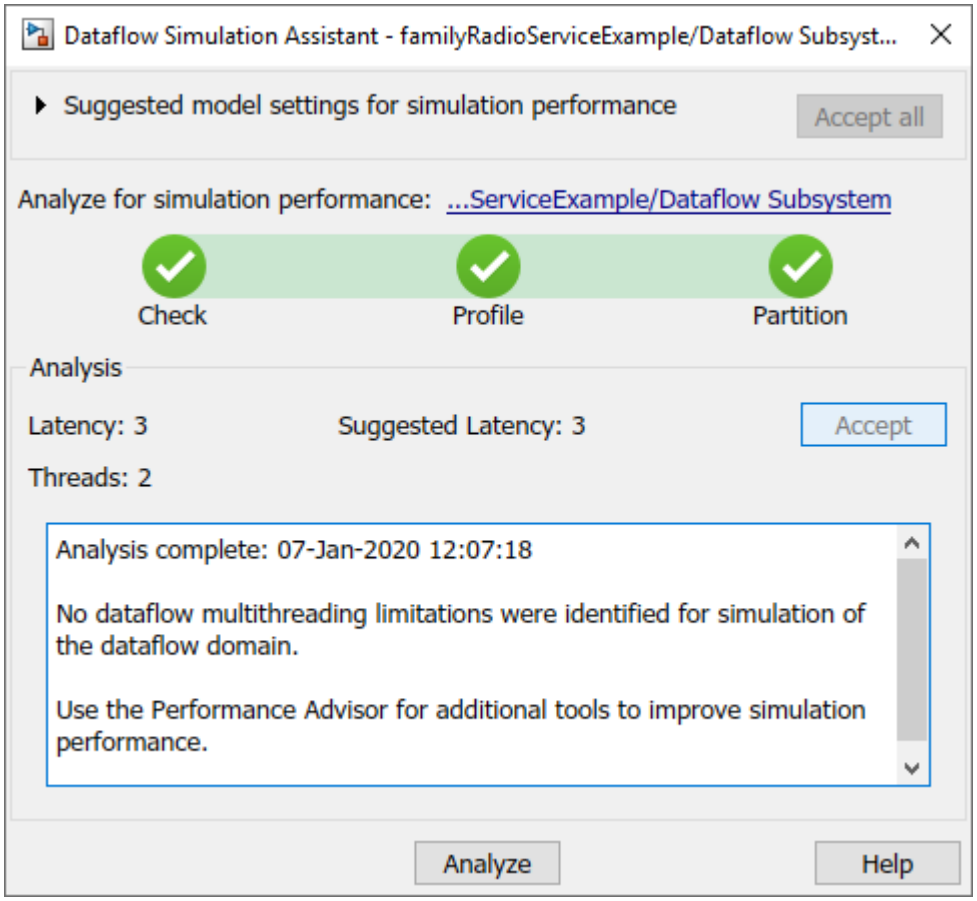
After analyzing the model, the assistant shows one thread because the data dependency between the blocks in the model prevents blocks from being executed concurrently. By pipelining the data dependent blocks, the Dataflow Subsystem can increase concurrency for higher data throughput. Dataflow Simulation Assistant shows the recommended number of pipeline delays as **Suggested Latency**. The suggested latency value is computed to give the best performance.

The following diagram shows the Dataflow Simulation Assistant where the Dataflow Subsystem currently specifies a latency value of 0, and the suggested latency for the system is 3.



Click the **Accept** button next to **Suggested Latency** in the Dataflow Simulation Assistant to use the recommended latency for the Dataflow Subsystem. This value can also be entered directly in the Property Inspector for "Latency" parameter. Simulink shows the latency parameter value using Z^{-1} tags at the output ports of the dataflow subsystem.

Dataflow Simulation Assistant now shows the number of threads as 2 meaning that the blocks inside the dataflow subsystem simulate in parallel using 2 threads.



Compensating for Latency

When latency is increased in the dataflow execution domain to break data dependencies between blocks and create concurrency, that delay needs to be accounted for in other parts of the model. For example, signals that are compared or combined with the signals at the output ports of the Dataflow Subsystem must be delayed to align in time with the signals at the output ports of the Dataflow Subsystem. In this example, the audio signal from the source block that goes to the Audio Device Writer must be delayed to align with other signals. To compensate for the latency specified on the dataflow subsystem, use a delay block to delay this signal by 2 frames. For this signal, the frame length is 1000. A delay value of 2000 is set in the delay block to align the signal from source and the signal processed through Dataflow Subsystem.

Dataflow Simulation Performance

Simulate the model and measure model execution time. When measuring the time taken for simulating the model, comment out the Spectrum Analyzer blocks and Audio Device Writer blocks to measure the time taken primarily for the Dataflow Subsystem. Execution time is measured using the sim command, which returns the simulation execution time of the model. We can measure the amount of speedup obtained by dividing the execution time taken by the model using multiple threads with the execution time taken by the original model. This number is computed and shown below.

These numbers and analysis were published on a Windows desktop computer with Intel® Xeon® CPU W-2133 @ 3.6 GHz 6 Cores 12 Threads processor.

```
Simulation execution time for multithreaded model = 2.21s
Actual speedup with dataflow: 1.6x
```

Code Generation

Code generation requires a Simulink Coder™ or an Embedded Coder® license. Press Ctrl+B to build the model and generate single-core code for your desktop target. If your desktop machine is Windows or Linux, you can generate multicore code for the model. To enable multicore code generation for the model, you must select the **Allow tasks to execute concurrently on target** parameter in the **Solver** pane under **Solver details**. Selecting this parameter allows:

- Each rate in the model to execute as an independent concurrent task on the target processor
- The dataflow subsystem to generate additional concurrent tasks by automatically partitioning the blocks

In the generated code you can observe the generated functions for each concurrent task created by the dataflow domain and realized as an OpenMP section.

```
#pragma omp parallel num_threads(2)

  {
#pragma omp sections

    {

#pragma omp section

      {
        familyRadioServiceEx_ThreadFcn0();
      }

#pragma omp section

      {
        familyRadioServiceEx_ThreadFcn1();
      }
    }
  }
```

Summary

This example shows how to specify dataflow as the execution domain in a model to design computationally-intensive signal processing systems, improve simulation performance of the model and generate multicore code.

Appendix

The following helper functions are used in this example.

- dspSimulateDataflowExample

Multicore Code Generation for Dataflow Domain

This example shows how to deploy a noise reduction application on a multicore target hardware using dataflow.

Before You Begin

To run this example, you must have the following software and hardware installed and set up:

- Embedded Coder Support Package for Xilinx Zynq Platform
- Zynq board

For details on installing the support package and setting up the Zynq hardware, refer to the Setup and Configuration section of Embedded Coder Support Package for Xilinx Zynq Platform.

Introduction

The dataflow execution domain allows you to make use of the multiple cores on the target hardware for computationally intensive signal processing systems.

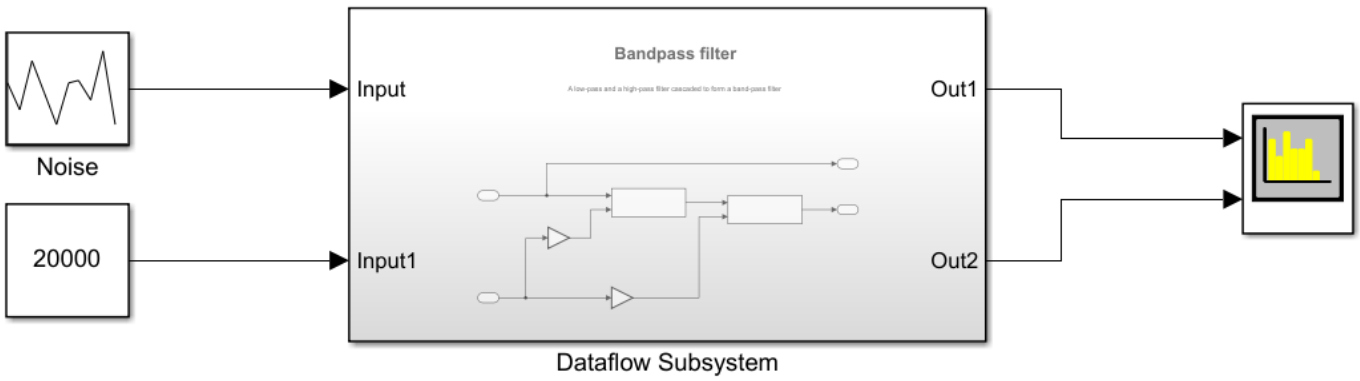
This example shows how to specify dataflow as the execution domain of a subsystem and improve performance by generating multicore code. The example uses processor-in-the-loop (PIL) simulation for deploying the application on the ARM CPU within a Zynq hardware and execution-time profiling for measuring the performance.

Noise Reduction System

The model in this example uses two Variable Bandwidth IIR filter blocks configured as a low-pass and a high-pass filter respectively. The filters are connected in series within the Dataflow Subsystem to collectively form a bandpass noise filtering system. The source signal is a random noise. Open dspdataflowzynq model.

Efficient Noise Reduction with Dataflow on Zynq

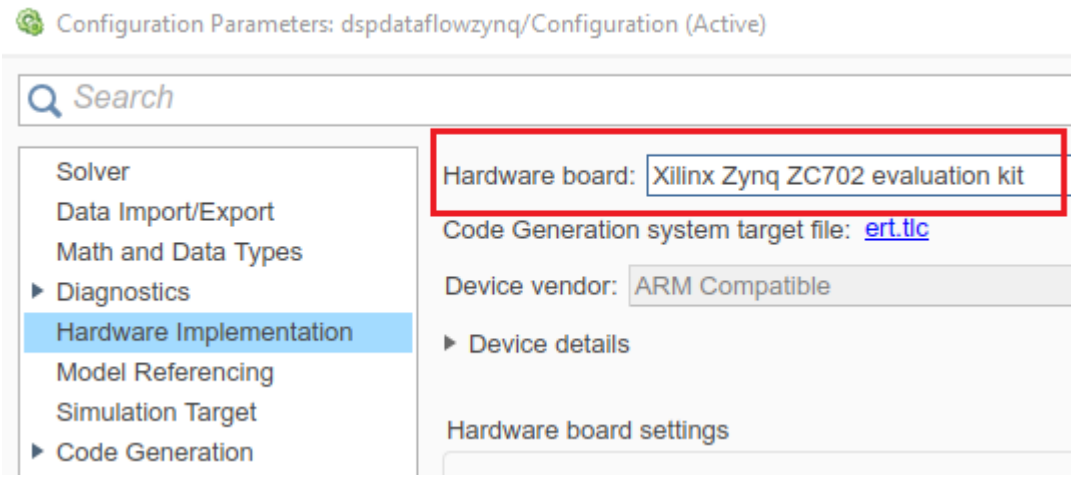
Copyright 2019 The MathWorks, Inc.



Info

Configure hardware settings

Configure the model to generate code for the Zynq-based hardware. This example uses a Zynq-7000 SoC ZC702 Evaluation Kit and performs processor-in-the-loop (PIL) simulation on the target hardware.



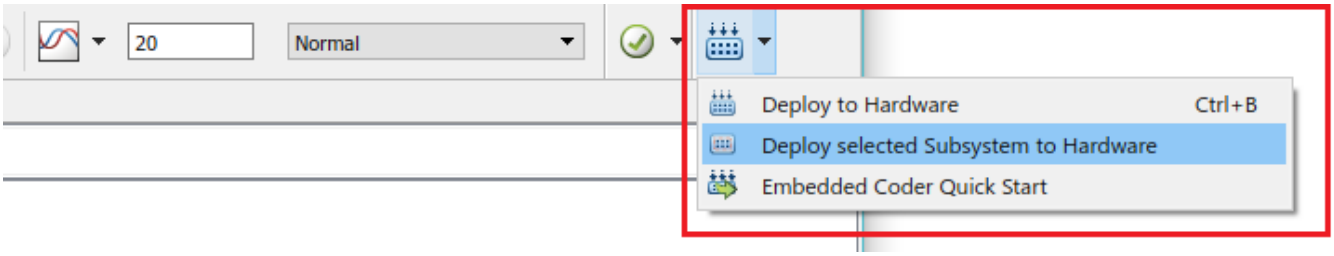
Configure PIL simulation and execution-time profiling

Configure the model to generate a PIL block when code is generated for a subsystem. This allows you to measure the time taken for the Dataflow Subsystem block on the target hardware. For details on configuring PIL simulation with PIL blocks, see Simulation with Blocks From Subsystems.

Enable profiling of function execution times for subsystems. For details on profiling with PIL, see Code Execution Profiling with SIL and PIL.

Generate code and simulate on target

Build the Dataflow Subsystem block. This step generates the PIL block from Dataflow Subsystem.



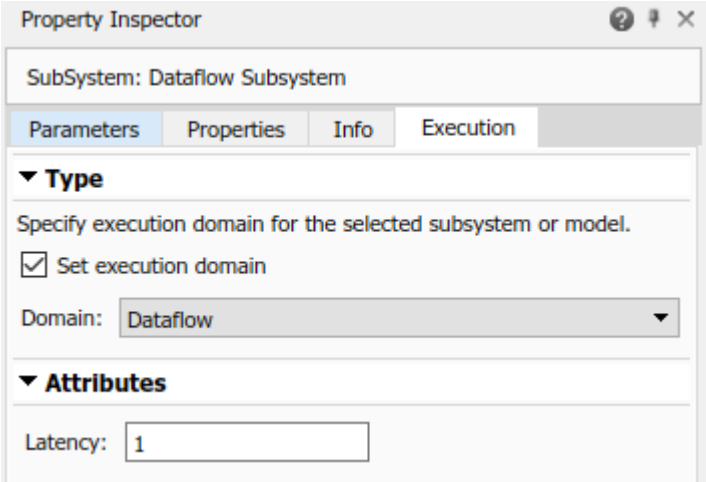
Replace Dataflow Subsystem block in the original model with the generated PIL block.

Simulate the model and measure average execution time of the subsystem using the profiling results retrieved at the end of PIL simulation. Average execution time can be obtained by dividing the total execution time taken by the subsystem by the number of calls to the subsystem. This number is computed and shown below.

```
Average execution time of generated code for single-core = 5.6 ms
```

Specify Dataflow Execution Domain for Subsystem

Dataflow domains automatically partition your model and generates code with multiple threads for multicore targets. In Simulink, you specify dataflow as the execution domain for a subsystem by setting the Domain parameter to Dataflow using Property Inspector. You can view the Property Inspector for a subsystem, first by selecting the subsystem and then selecting **View>Property** Inspector. In the Property Inspector, you can set the domain to dataflow by selecting **Set domain specification** and then selecting "Dataflow" for **Domain** setting. You can also use the Dataflow Subsystem block from the Dataflow library of DSP System toolbox to get a subsystem that is preconfigured with the dataflow execution domain.



To increase the throughput of a system, it can be advantageous to increase the latency of a system. Specify the **Latency** value in the Execution tab of the Property Inspector. Setting a **Latency** value of 1 will add a pipeline delay to break dependency between the filter blocks and enable the dataflow domain to achieve concurrency.

Multicore Code Generation of Dataflow Subsystem

To enable multicore code generation, you must select the **Allow tasks to execute concurrently on target** parameter in the **Solver** pane of the Configuration Parameters under **Solver details**.

Rebuild the Dataflow Subsystem block to generate the multicore version of the PIL block.

After code generation is completed for the subsystem, you can observe the generated functions for each concurrent thread created by the dataflow domain and how they are triggered during execution of the model step function.

The Dataflow Subsystem block generates two thread functions, Dataflow_ThreadFcn0 and Dataflow_ThreadFcn1.

```
static void Dataflow_ThreadFcn0(void)
{
  dsp_simulink_VariableBandwidt_T *obj;
  boolean_T p;
  if (pipeStage_Concurrent0_Dataflow >= 1) {
    /* Delay generated from: '<S1>/Variable Bandwidth IIR Filter1' */
    memcpy(&Dataflow_B.TmpDelayBufferAtVariableBandwid[0],
           &Dataflow_DW.TmpDelayBufferAtVariableBandwid[0], sizeof(real_T) <<
           12U);
      .
      .
  }
}
```

```
static void Dataflow_ThreadFcn1(void)
{
 dsp_simulink_VariableBandwidt_T *obj;
 boolean_T p;

 /* Gain: '<S1>/Gain' */
 Dataflow_B.Gain = Dataflow_P.Gain_Gain * Dataflow_B.TmpDataInAtInput1Outport1;

 /* MATLABSystem: '<S1>/Variable Bandwidth IIR Filter' */
 obj = &Dataflow_DW.obj;
 Dataflow_B.varargin_1 = Dataflow_DW.obj.PassbandFrequency;
 p = false;

      .
      .

}
```

The thread functions are registered as POSIX threads at model initialization and triggered during each model step. The consecutive trigger and wait function calls implement the fork-join pattern for the dataflow threads.

```
/* Model initialize function */
void Dataflow_initialize(void)
{
      .
      .
  {
    /* SystemInitialize for Atomic SubSystem: '<Root>/Dataflow Subsystem' */
    pipeStage_Concurrent0_Dataflow = 0;
    threadHandle_Concurrent0[0] = rtw_register_task(Dataflow_ThreadFcn0);
    threadHandle_Concurrent0[1] = rtw_register_task(Dataflow_ThreadFcn1);

      .
      .

    /* End of SystemInitialize for SubSystem: '<Root>/Dataflow Subsystem' */
  }
}
```

```
/* Model step function */
void Dataflow_step(void)
{
      .
      .
  rtw_trigger_task(threadHandle_Concurrent0[0]);
  rtw_trigger_task(threadHandle_Concurrent0[1]);
  rtw_waitfor_task(threadHandle_Concurrent0[0]);
  rtw_waitfor_task(threadHandle_Concurrent0[1]);
      .
      .
  if (pipeStage_Concurrent0_Dataflow < 1) {
    pipeStage_Concurrent0_Dataflow++;
  }

  /* End of Outputs for SubSystem: '<Root>/Dataflow Subsystem' */
}

/* Model terminate function */
void Dataflow_terminate(void)
{
  /* Terminate for Atomic SubSystem: '<Root>/Dataflow Subsystem' */
  rtw_deregister_task(threadHandle_Concurrent0[0]);
  rtw_deregister_task(threadHandle_Concurrent0[1]);

  /* End of Terminate for SubSystem: '<Root>/Dataflow Subsystem' */
}
```

Multicore Execution Performance

Simulate the model with the multicore version of the PIL block and repeat the measurement for the execution time of the subsystem.

```
Average execution time of generated code for multicore = 3.9 ms

Actual speedup with dataflow: 1.44x
```

Copyright 2018 The MathWorks, Inc.

Simulink HDL Optimized Block Examples in DSP System Toolbox

- “Implement CIC Decimation Filter for HDL” on page 8-2
- “Fully Parallel Systolic FIR Filter Implementation” on page 8-5
- “Partly Serial Systolic FIR Filter Implementation” on page 8-9
- “Automatic Delay Matching for the Latency of FFT HDL Optimized Block” on page 8-13

Implement CIC Decimation Filter for HDL

This example shows how to use the CIC Decimation HDL Optimized block to filter and downsample data. You can generate HDL code from the subsystem in the Simulink(R)model.

1. Generate frames of random input samples.
2. Generate reference output data from the `dsp.CICDecimation` System object™.
3. Run the Simulink model that contains the CIC Decimation HDL Optimized block.
4. Compare the Simulink block output data with the reference data.

Set up input data parameters. The model uses these workspace variables to configure the CIC Decimation HDL Optimized block. The block supports variable decimation factor specified at an input port. Choose a range for the input decimFactor and set the Decimation factor parameter of the block to the maximum expected decimation factor.

```
R = 8; % Decimation factor
M = 1; % Differential delay
N = 3; % Number of sections

varRValue = [4,R];

numFrames = length(varRValue);
dataSamples = cell(1,numFrames);
varRtemp = cell(1,numFrames);
cicFcnOutput = [];
WL = 0; % Word length
FL = 0; % Fraction length
```

Generate frames of random input samples. To generate reference output data for comparison, apply the samples to the `dsp.CICDecimation` System object. This object does not support variable decimation rate, so you must create and release the object for each change in the decimation factor.

```
for i = 1:numFrames
    framesize = varRValue(i)*randi([5 20],1,1);
    dataSamples{i} = fi(randn(framesize,1),1,16,8);
    varRtemp{i} = fi(varRValue(i)*ones(framesize,1),0,12,0);
    obj = dsp.CICDecimator('DifferentialDelay',M,'NumSections',N,'DecimationFactor',varRValue(i))
    cicOut = step(obj,dataSamples{i}).';
    WL = max([WL,cicOut.WordLength]);
    FL = max([FL,cicOut.FractionLength]);
    cicFcnOutput = [fi(cicFcnOutput,1,WL,FL),cicOut];
    release(obj);
end
```

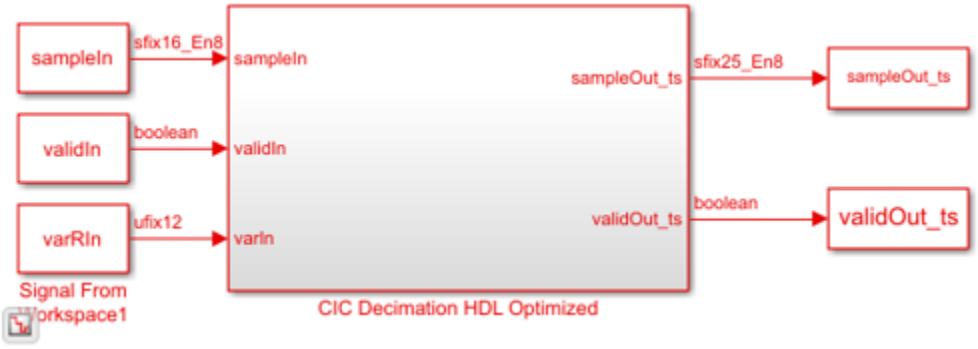
Generate a stream of samples by converting frames to samples, and provide those samples (`sampleIn`) as input to the Simulink model along with the corresponding valid signal (`validIn`) and decimation rate (`varRIn`).

```
idlecyclesbetweensamples = 0;
idlecyclesbetweenframes = 4+N;

% The block has a latency of 3+N cycles for fixed decimation rate and 4+N
% for variable decimation rates when Gain correction is Off. To flush
% remaining data before changing the decimation rate, run the model by
% inserting required number of idle cycles after each frame.
```

```
sampleIn = []; validIn = []; varRIn = [];

for ij = 1:numFrames

    dataInFrame =  dataSamples{ij};

    data = []; valid=[]; varR = [];
    for ii = 1:length(dataInFrame)
        data = [data dataInFrame(ii) ...
            zeros(1,idlecyclesbetweensamples)];
        valid = [valid true(1,1) ...
            false(1,idlecyclesbetweensamples)];
        varR = [varR varRtemp{ij}(ii) ...
            zeros(1,idlecyclesbetweensamples)];
    end

    sampleIn = cast([sampleIn,data,zeros(1,idlecyclesbetweenframes)],'like',dataInFrame);
    validIn = logical([validIn,valid,zeros(1,idlecyclesbetweenframes)]);
    varRIn = fi([varRIn,varR,zeros(1,idlecyclesbetweenframes)],0,12,0);

end

sampletime = 1;
simTime = length(validIn);
```

Run the Simulink model.

```
modelname = 'HDLCICDecimationModel';
open_system(modelname);
sim(modelname);
```



Capture the output data from the Simulink model, and compare it against the output from the `dsp.CICDecimation` System object.

```
sampleOut = squeeze(sampleOut_ts.Data).';
validOut  = squeeze(validOut_ts.Data);
cicOutput = sampleOut(validOut);

fprintf('\nHDL CIC Decimation\n');
difference = (abs(cicOutput-cicFcnOutput(1:length(cicOutput)))>0);
fprintf('\nTotal number of samples differed between Simulink block output and MATLAB function ou
```

```
HDL CIC Decimation
```

```
Total number of samples differed between Simulink block output and MATLAB function output is: 0
```

See Also

Blocks
CIC Decimation HDL Optimized

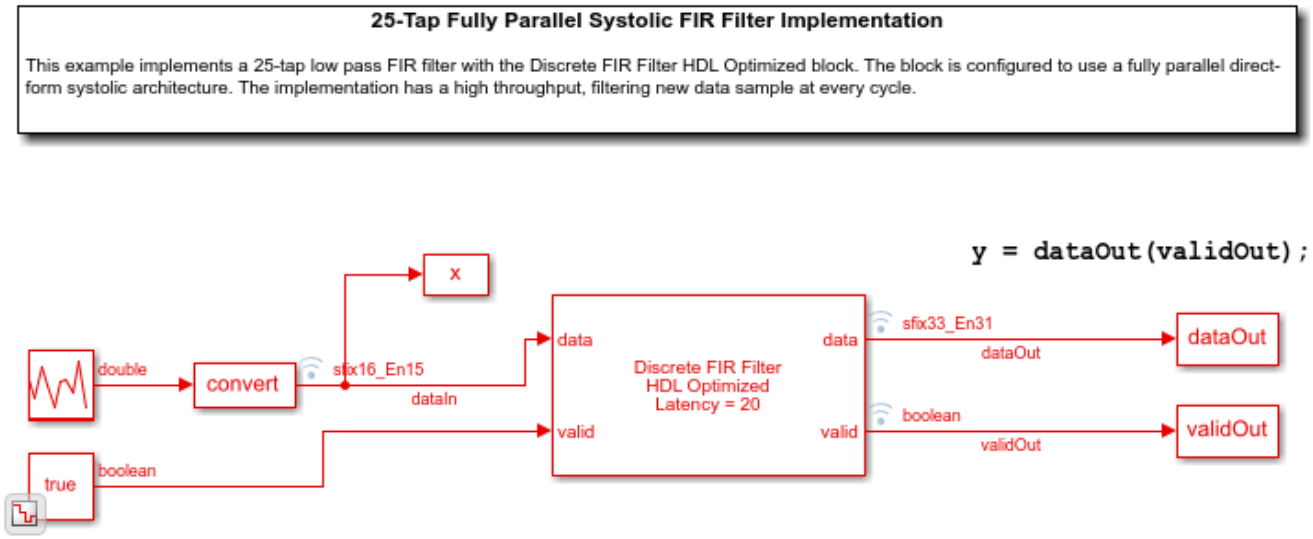
Fully Parallel Systolic FIR Filter Implementation

Implement a 25-tap lowpass FIR filter by using the Discrete FIR Filter HDL Optimized block.

The model filters new data samples at every cycle.

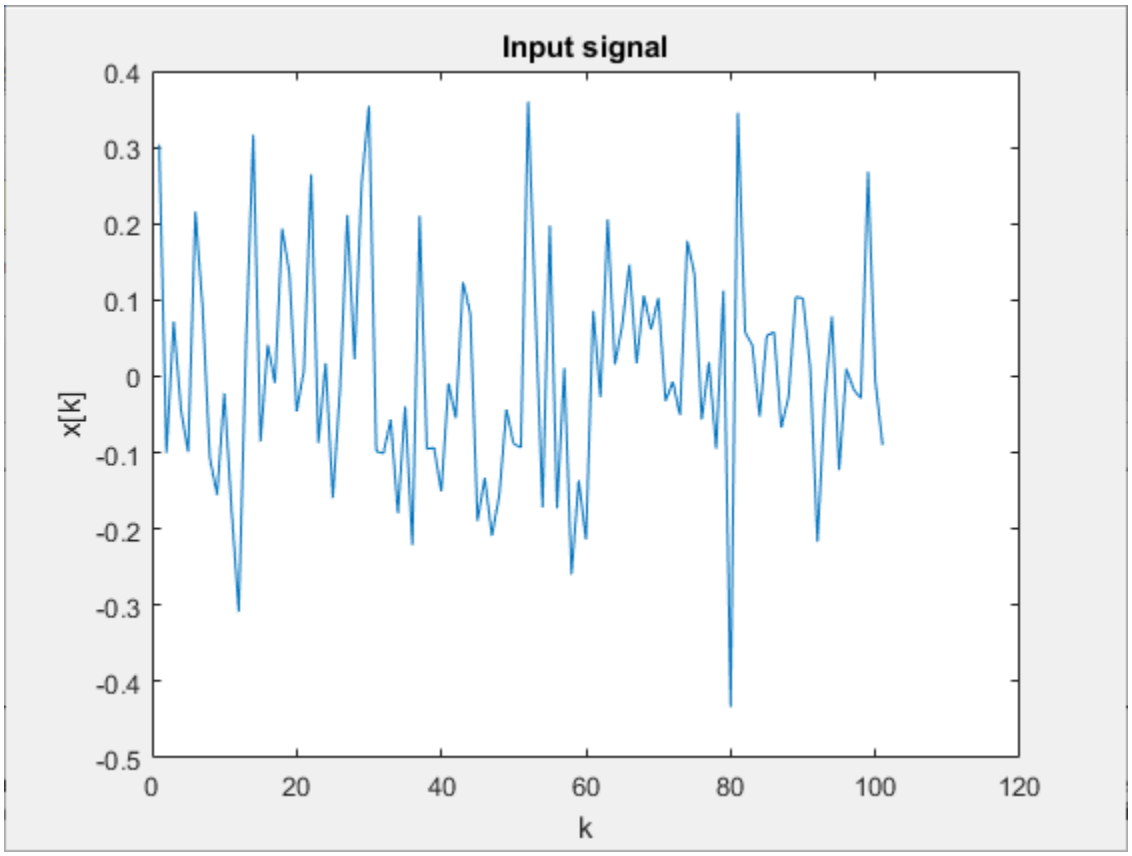
Open the Model

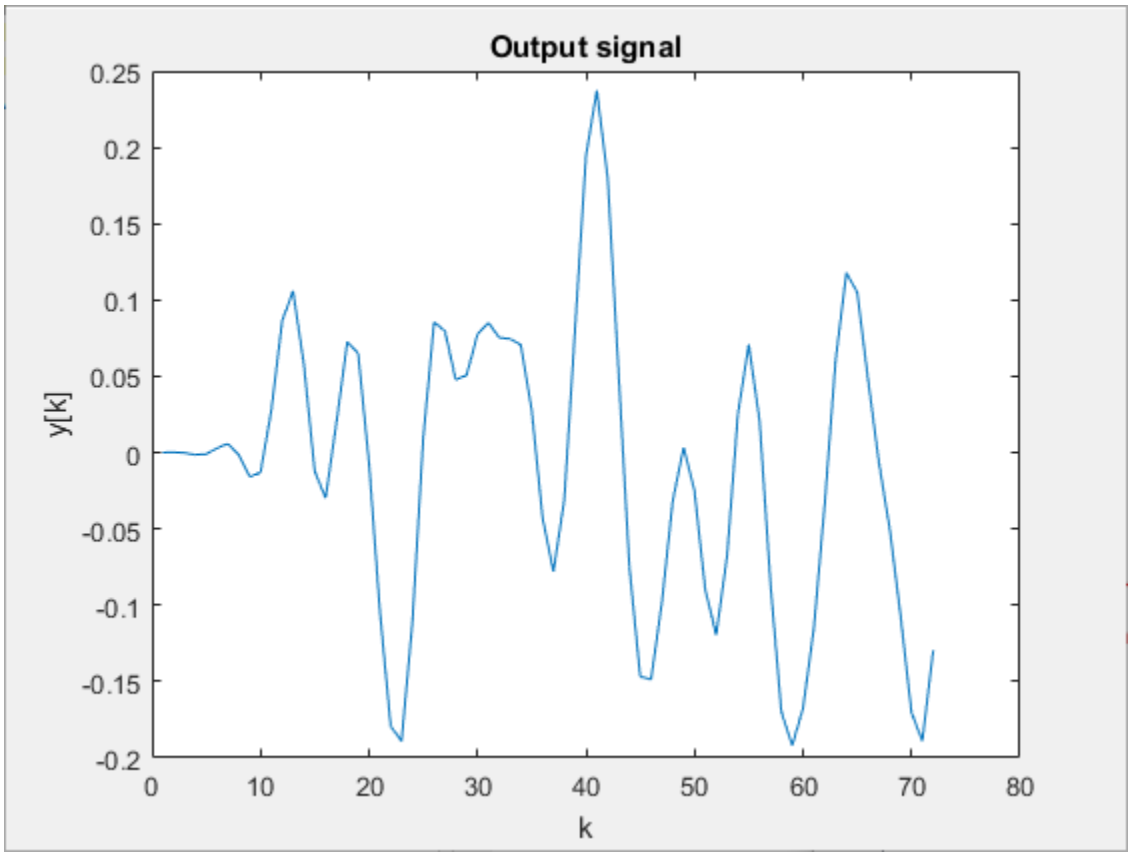
Open the model. Inspect the block parameters. The **Filter structure** is set to `Direct form systolic`.



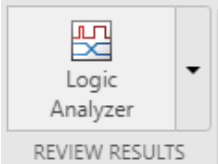
Run the Model and Inspect Results

Run the model. Observe the input and output signals in the generated plots.

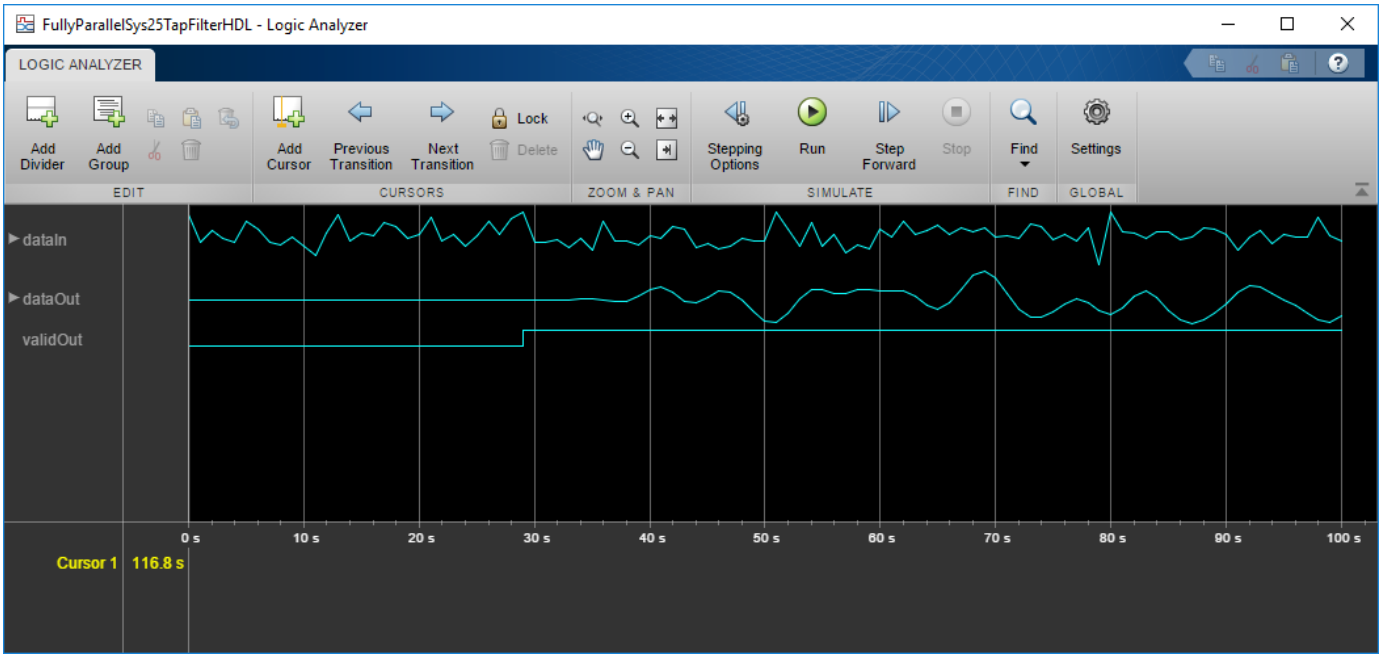

Input signal
x[k]
k
0.4
0.3
0.2
0.1
0
-0.1
-0.2
-0.3
-0.4
-0.5
0
20
40
60
80
100
120



From the model toolbar, open the **Logic Analyzer**. If the button is not displayed, expand the **Review Results** app gallery.



Note the pattern of the `validOut` signal.



Generate HDL Code

To generate HDL code from the Discrete FIR Filter HDL Optimized block, right-click the block and select **Create Subsystem from Selection**. Then right-click the subsystem and select **HDL Code** > **Generate HDL Code for Subsystem**.

See Also

Blocks

Discrete FIR Filter HDL Optimized

Partly Serial Systolic FIR Filter Implementation

This example shows how to implement a 32-tap lowpass FIR filter using the Discrete FIR Filter HDL Optimized block.

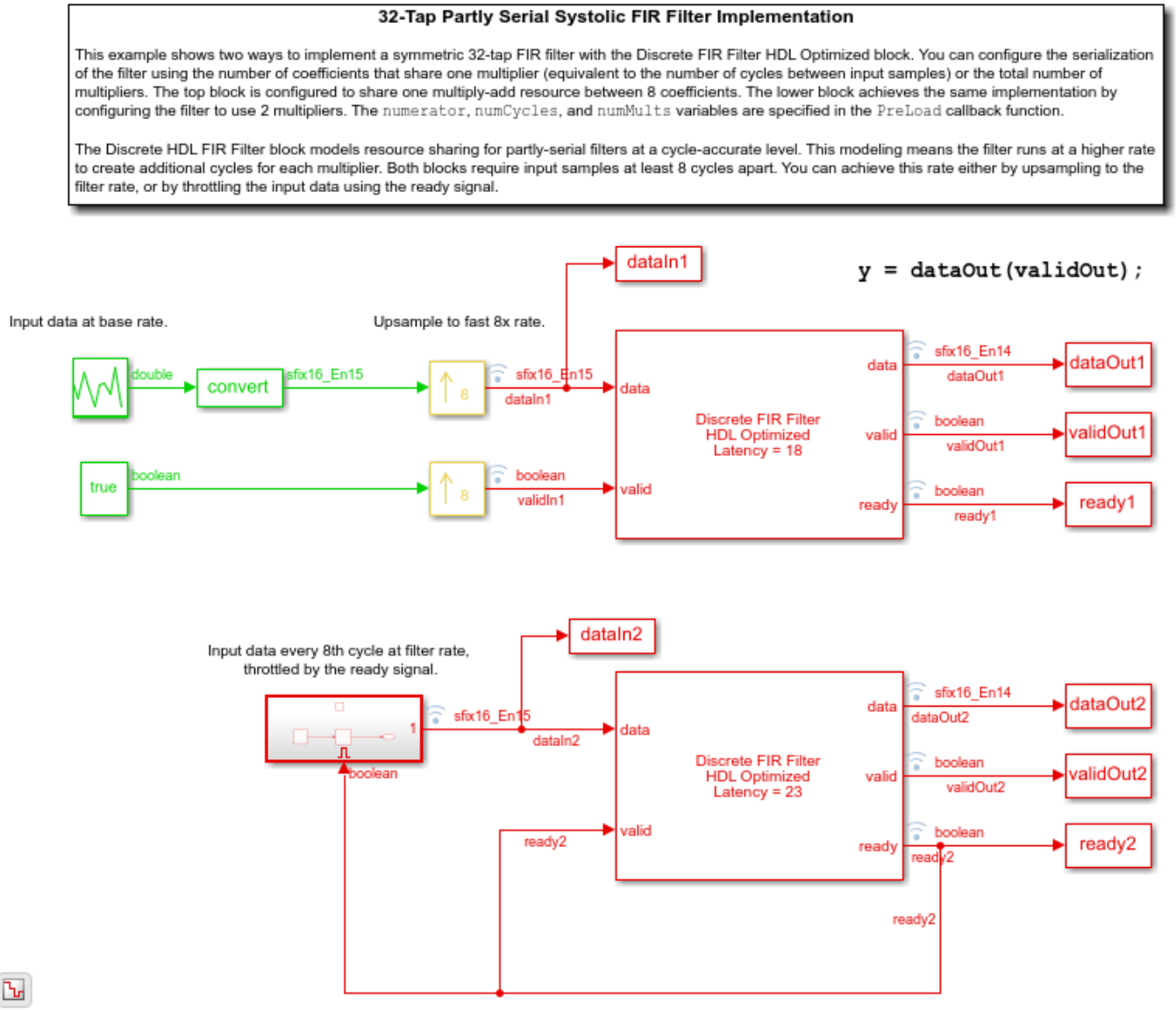
Both filter blocks in the example model implement an identical partly- serial 32-tap filter. The top block configures the serial filter by specifying the number of cycles, N , between input samples. This spacing allows each multiplier to be shared by N coefficients. The second block is configured to use a certain number of multipliers, M. These two configurations result in the same filter implementation. For 32 symmetric coefficients, there are 16 unique coefficients. Therefore the filter shares each of 2 multipliers between 8 coefficients.

The model shows two ways of applying input samples, depending on the rate of the rest of your design.

Open the Model

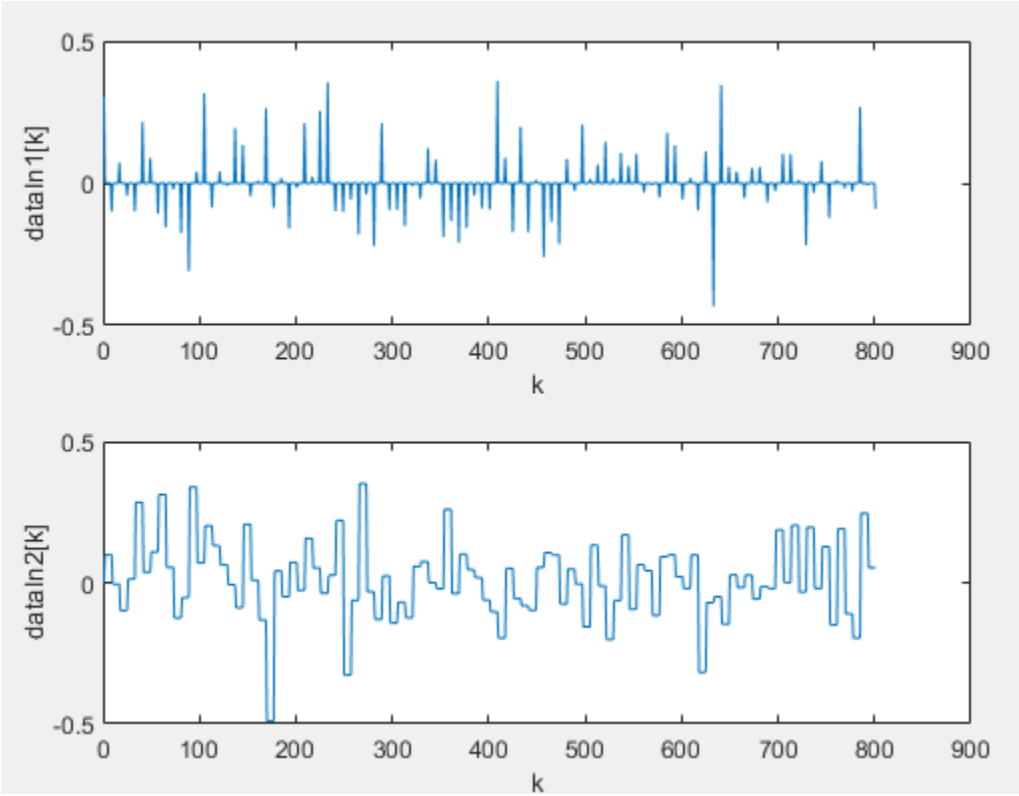
Open the model. Inspect the top block parameters. The **Filter structure** is set to `Partly serial systolic` and **Specify serialization factor as** is set to `Minimum number of cycles between valid input samples`. The **Number of cycles** is set (to 8) using a variable, *numCycles*. In the lower block, **Specify serialization factor as** is set to `Maximum number of multipliers`. The **Number of multipliers** is set (to 2) using a variable. The variables are defined in the `PostLoadFcn` callback function.

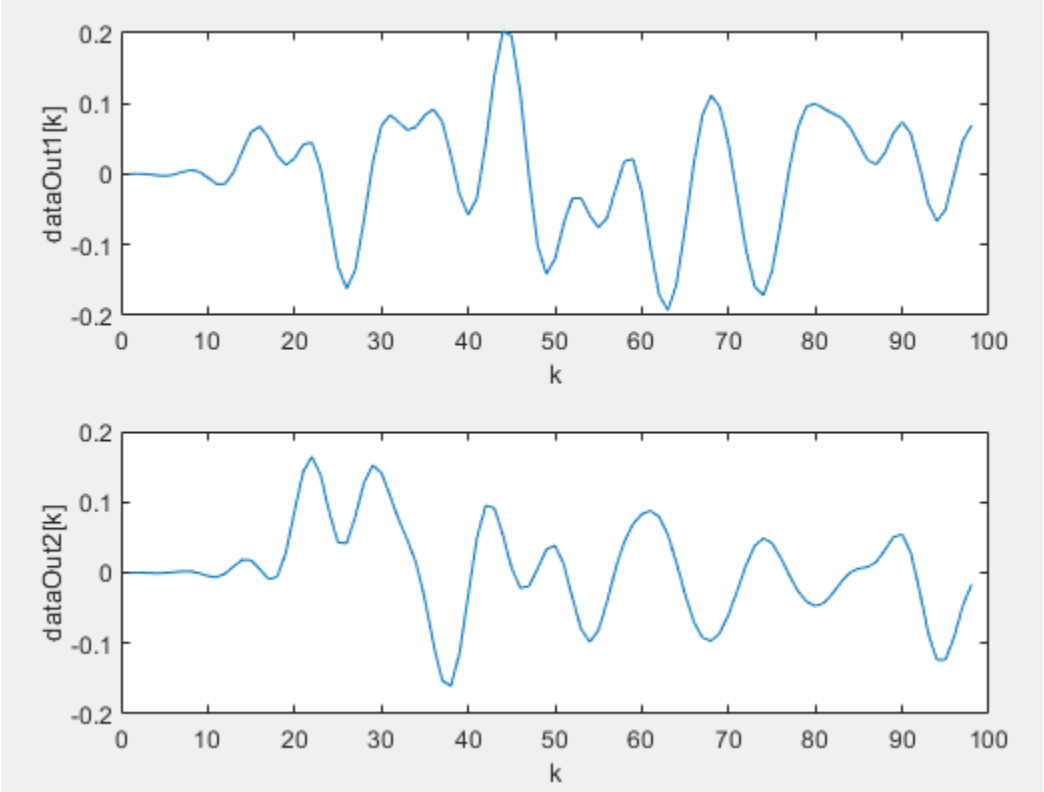
From the color coding, you can see the rate of both filter blocks is the same, while the rate of the generated input samples is different.



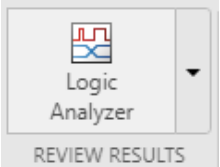
Run the Model and Inspect Results

Run the model. Observe the input and output signals in the generated plots. The code to generate the plots is in the `PostSimFcn` callback.

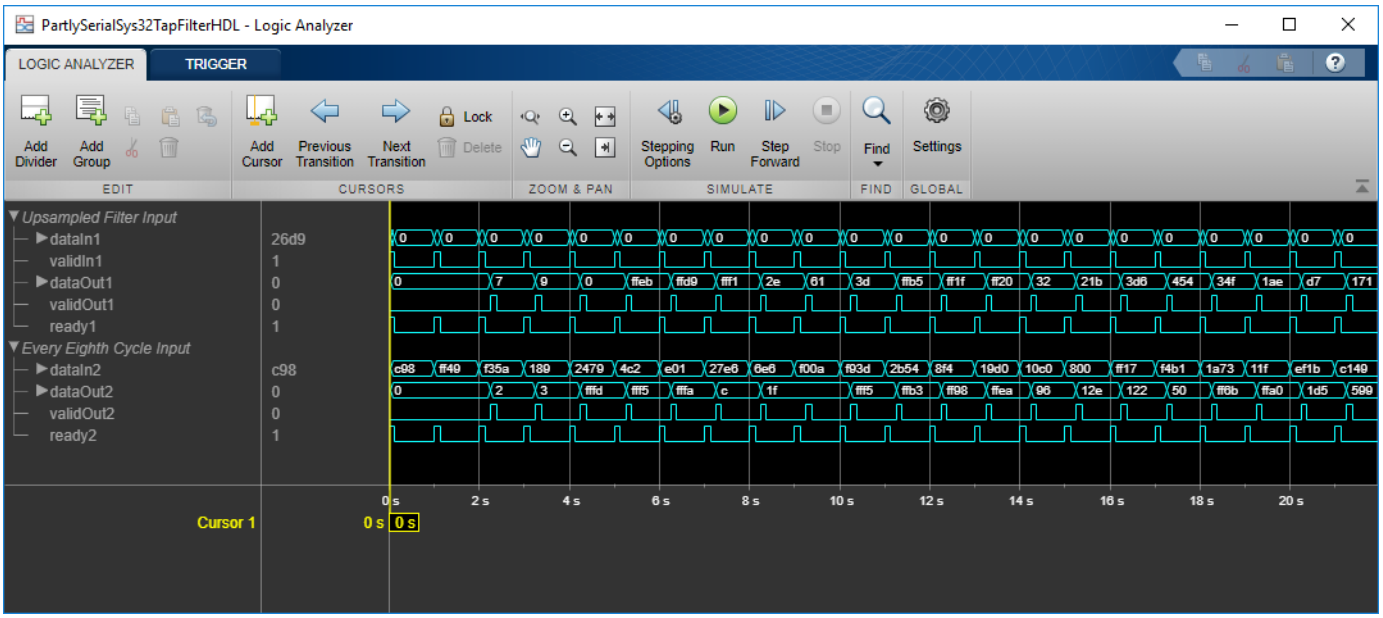

dataIn1[k]
0.5
0
-0.5
0
100
200
300
400
500
600
700
800
900
k
dataIn2[k]
0.5
0
-0.5
0
100
200
300
400
500
600
700
800
900
k


dataOut1[k]
0.2
0.1
0
-0.1
-0.2
0
10
20
30
40
50
60
70
80
90
100
k
dataOut2[k]
0.2
0.1
0
-0.1
-0.2
0
10
20
30
40
50
60
70
80
90
100
k

From the model toolbar, open the **Logic Analyzer**. If the button is not displayed, expand the **Review Results** app gallery.



Inspect the rising edges of `ready`, `validIn`, and `validOut`.



Generate HDL Code

To generate HDL code from either Discrete FIR Filter HDL Optimized block, right-click the block and select **Create Subsystem from Selection**. Then right-click the subsystem and select **HDL Code > Generate HDL Code for Subsystem**. Both blocks generate the same HDL code.

See Also

Blocks

Discrete FIR Filter HDL Optimized

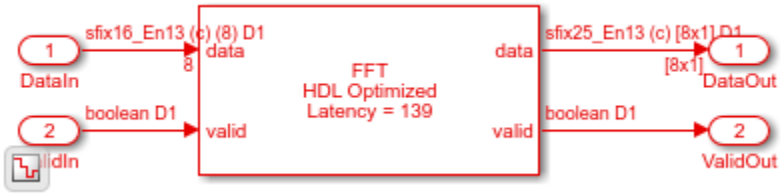
Automatic Delay Matching for the Latency of FFT HDL Optimized Block

This example shows how to programmatically obtain the latency of an FFT HDL Optimized block in a model. You can use the latency value for delay matching of parallel data paths.

Be cautious with delay matching for large signals or long latencies, since it adds memory to your hardware implementation. Alternatively, if the signal does not change within a frame, use the valid or frame control signals to align the signal with the output of the FFT block.

Open a model that contains an FFT HDL Optimized or an IFFT HDL Optimized block, such as the model from the “Implement FFT for FPGA Using FFT HDL Optimized Block” example.

```
modelname = 'FFTHDLOptimizedExample_Streaming';
load_system(modelname);
set_param(modelname,'SimulationCommand','Update');
open_system([modelname '/FFT HDL Optimized Streaming']);
```



This example includes a `ffthdlLatency` function that calculates the latency of the block for its current parameters. Call the function with the block pointer and input vector size. You can use the Simulink path to the block, or select the block in the model to obtain a pointer, `gcb`. In this model, the input signal is a vector of 8 samples.

```
latency = ffthdlLatency([modelname '/FFT HDL Optimized Streaming/FFT HDL Optimized'],8)
```

```
latency =

   139
```

The function copies the parameters from the block pointer and creates a System object™ with the same settings as the block. Then it calls the `getLatency` function on the object. See `getLatency`.

```
function lat = ffthdlLatency(block, vectorsize)

% default vector size = 1
if nargin == 1
    vectorsize = 1;
end

fftlen = evalin('base',get_param(block, 'FFTLength'));
arch = get_param(block, 'Architecture');
bri = strcmpi(get_param(block, 'BitReversedInput'), 'on');
bro = strcmpi(get_param(block, 'BitReversedOutput'), 'on');
```

```
fftobj = dsp.HDLFFT('FFTLength',fftlen, ...
    'Architecture', arch, ...
    'BitReversedInput', bri, ...
    'BitReversedOutput', bro);

lat = getLatency(fftobj, fftlen, vectorsize);

end
```

See Also

Blocks
FFT HDL Optimized | IFFT HDL Optimized

Simulink Block Examples in Scopes and Data Logging Category

- “Obtain Measurement Data Programmatically for dsp.SpectrumAnalyzer System object” on page 9-2
- “Obtain Measurements Data Programmatically for Spectrum Analyzer Block” on page 9-5

Obtain Measurement Data Programmatically for dsp.SpectrumAnalyzer System object

Compute and display the power spectrum of a noisy sinusoidal input signal using the `dsp.SpectrumAnalyzer` System object. Measure the peaks, cursor placements, adjacent channel power ratio, distortion, and CCDF values in the spectrum by enabling the following properties:

- `PeakFinder`
- `CursorMeasurements`
- `ChannelMeasurements`
- `DistortionMeasurements`
- `CCDFMeasurements`

Initialization

The input sine wave has two frequencies: 1000 Hz and 5000 Hz. Create two `dsp.SineWave` System objects to generate these two frequencies. Create a `dsp.SpectrumAnalyzer` System object to compute and display the power spectrum.

```
Fs = 44100;
Sineobject1 = dsp.SineWave('SamplesPerFrame',1024,'PhaseOffset',10,...
    'SampleRate',Fs,'Frequency',1000);
Sineobject2 = dsp.SineWave('SamplesPerFrame',1024,...
    'SampleRate',Fs,'Frequency',5000);
SA = dsp.SpectrumAnalyzer('SampleRate',Fs,'Method','Filter bank',...
    'SpectrumType','Power','PlotAsTwoSidedSpectrum',false,...
    'ChannelNames',{'Power spectrum of the input'},'YLimits',[-120 40],'ShowLegend',true);
```

Enable Measurements Data

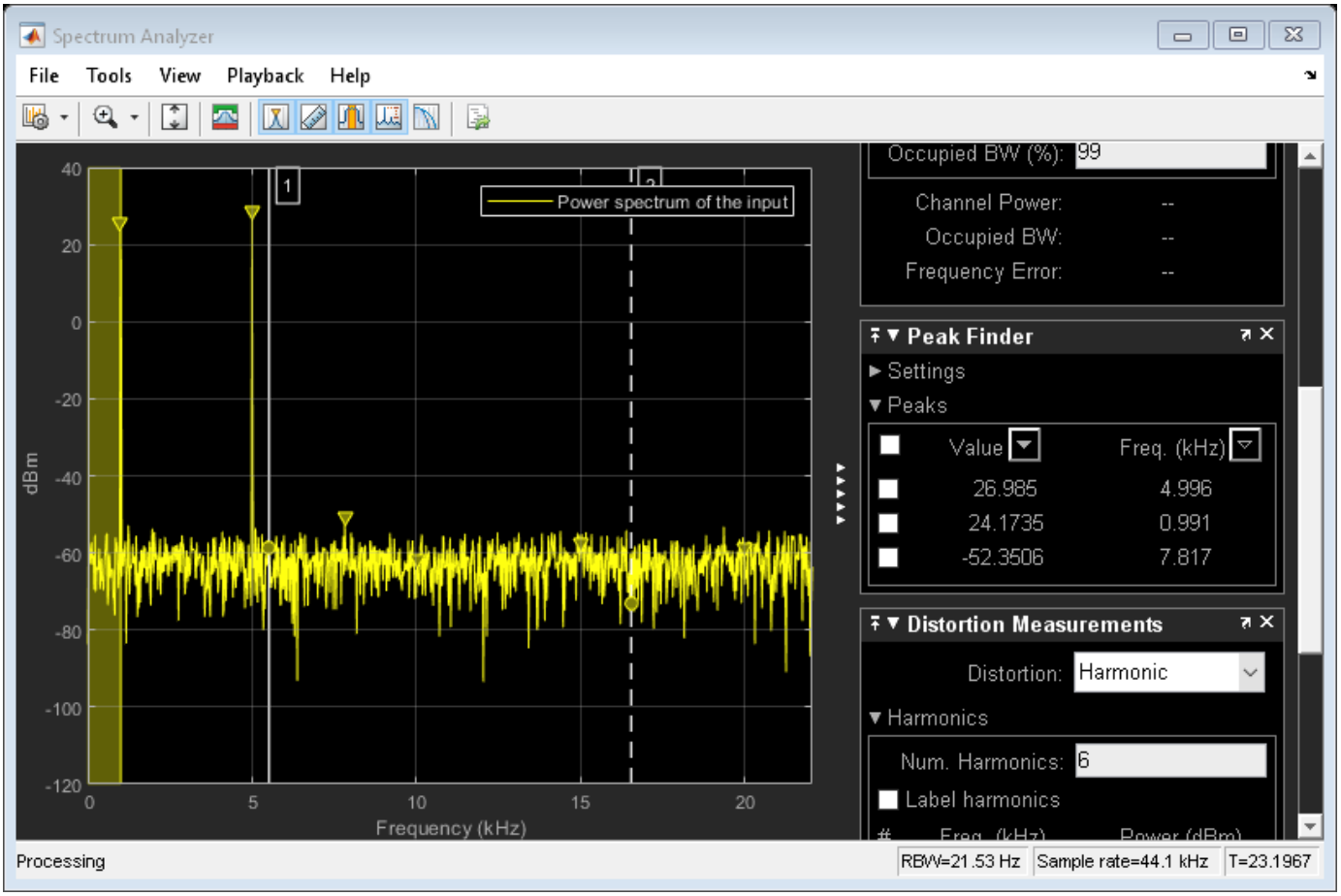
To obtain the measurements, set the `Enable` property of the measurements to `true`.

```
SA.CursorMeasurements.Enable = true;
SA.ChannelMeasurements.Enable = true;
SA.PeakFinder.Enable = true;
SA.DistortionMeasurements.Enable = true;
```

Use `getMeasurementsData`

Stream in the noisy sine wave input signal and estimate the power spectrum of the signal using the spectrum analyzer. Measure the characteristics of the spectrum. Use the `getMeasurementsData` function to obtain these measurements programmatically. The `isNewDataReady` function indicates when there is new spectrum data. The measured data is stored in the variable `data`.

```
data = [];
for Iter = 1:1000
    Sinewave1 = Sineobject1();
    Sinewave2 = Sineobject2();
    Input = Sinewave1 + Sinewave2;
    NoisyInput = Input + 0.001*randn(1024,1);
    SA(NoisyInput);
     if SA.isNewDataReady
        data = [data;getMeasurementsData(SA)];
     end
end
```



The right side of the spectrum analyzer shows the enabled measurement panes. The values shown in these panes match with the values shown in the last time step of the `data` variable. You can access the individual fields of `data` to obtain the various measurements programmatically.

Compare Peak Values

Peak values are obtained by the `PeakFinder` property. Verify that the peak values obtained in the last time step of `data` match the values shown on the spectrum analyzer plot.

```
peakvalues = data.PeakFinder(end).Value
```

```
peakvalues = 3×1

   26.9850
   24.1735
  -52.3506
```

```
frequencieskHz = data.PeakFinder(end).Frequency/1000
```

```
frequencieskHz = 3×1

    4.9957
    0.9905
```

```
    7.8166
```

See Also

Functions
`getMeasurementsData`

Objects
`dsp.SineWave` | `dsp.SpectrumAnalyzer`

Obtain Measurements Data Programmatically for Spectrum Analyzer Block

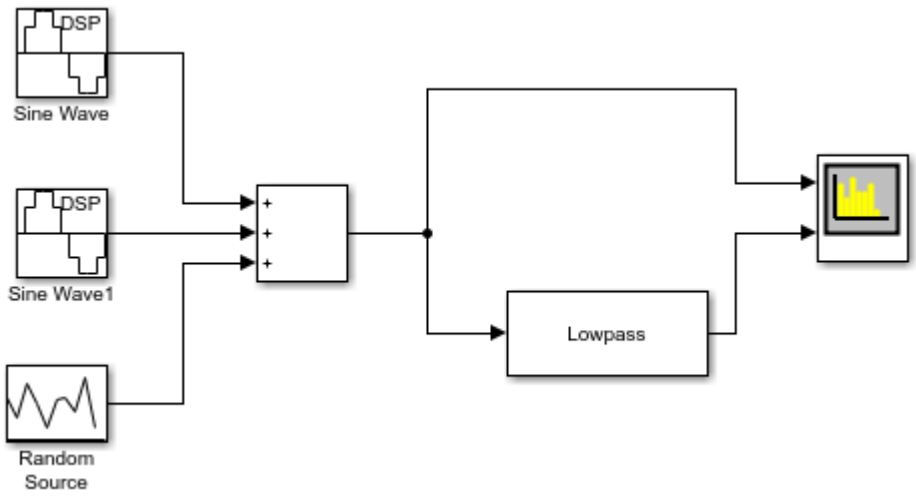
Compute and display the power spectrum of a noisy sinusoidal input signal using the Spectrum Analyzer block. Measure the peaks, cursor placements, adjacent channel power ratio, distortion, and CCDF values in the spectrum by enabling these block configuration properties:

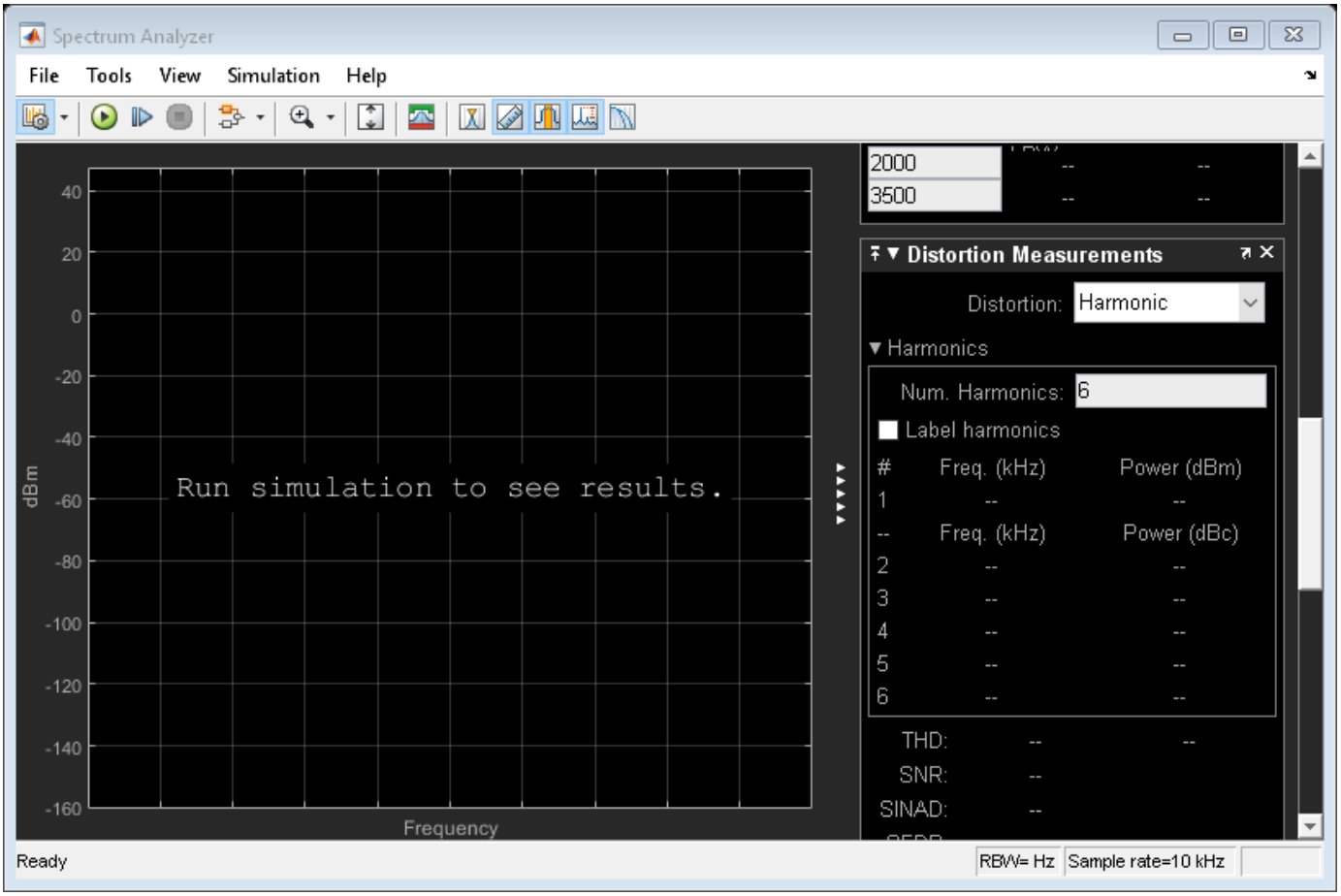
- PeakFinder
- CursorMeasurements
- ChannelMeasurements
- DistortionMeasurements
- CCDFMeasurements

Open and Inspect the Model

Filter a streaming noisy sinusoidal input signal using a Lowpass Filter block. The input signal consists of two sinusoidal tones: 1 kHz and 15 kHz. The noise is white Gaussian noise with zero mean and a variance of 0.05. The sampling frequency is 44.1 kHz. Open the model and inspect the various block settings.

```
model = 'spectrumanalyzer_measurements.slx';
open_system(model)
```





Access the configuration properties of the Spectrum Analyzer block using the `get_param` function.

```
sablock = 'spectrumanalyzer_measurements/Spectrum Analyzer';
cfg = get_param(sablock,'ScopeConfiguration');
```

Enable Measurements Data

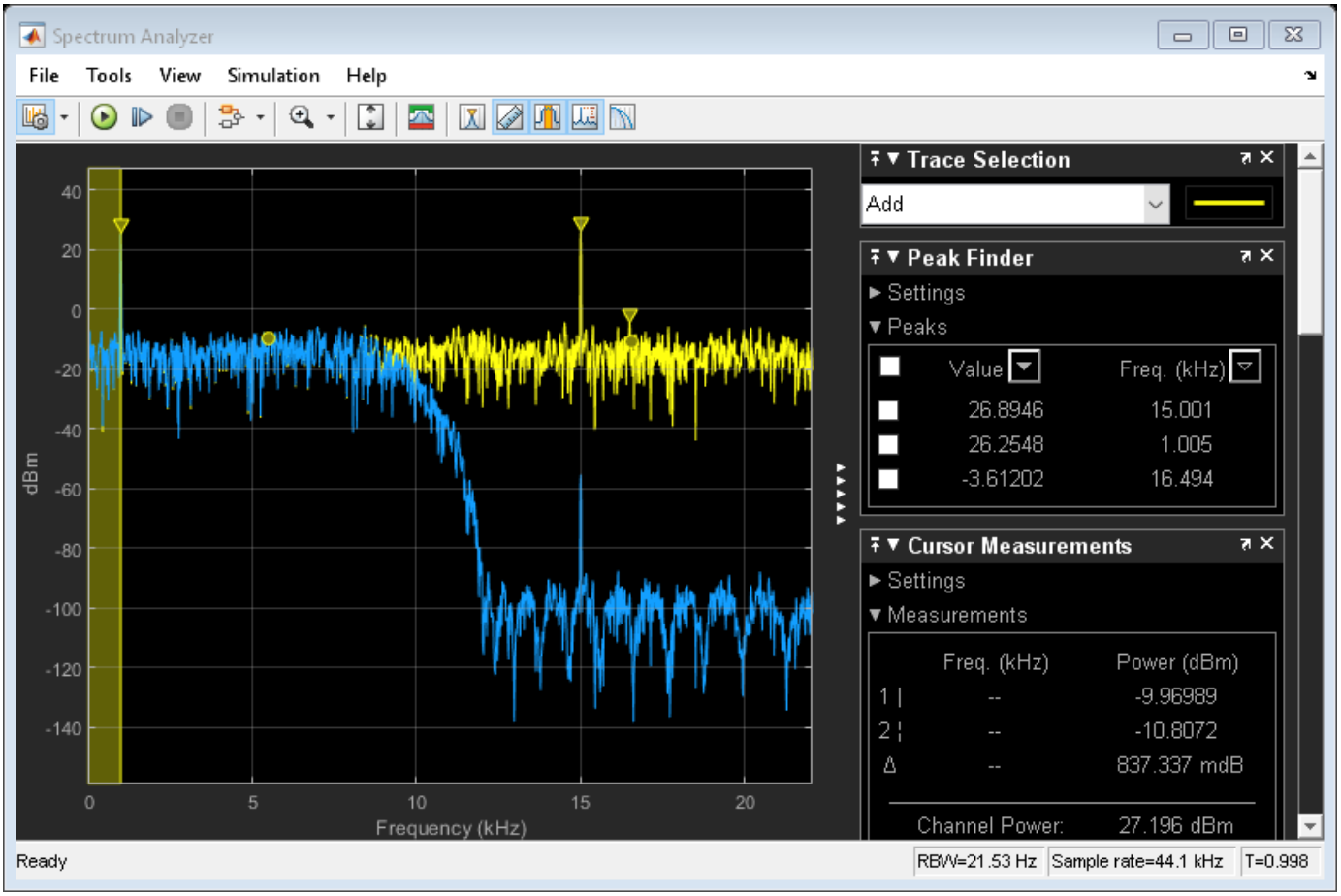
To obtain the measurements, set the `Enable` property of the measurements to `true`.

```
cfg.CursorMeasurements.Enable = true;
cfg.ChannelMeasurements.Enable = true;
cfg.PeakFinder.Enable = true;
cfg.DistortionMeasurements.Enable = true;
```

Simulate the Model

Run the model. The Spectrum Analyzer block compares the original spectrum with the filtered spectrum.

```
sim(model)
```



The right side of the spectrum analyzer shows the enabled measurement panes.

Using `getMeasurementsData`

Use the `getMeasurementsData` function to obtain these measurements programmatically.

```
data = getMeasurementsData(cfg)
```

```
data =

  1x5 table

    SimulationTime      PeakFinder      CursorMeasurements      ChannelMeasurements      DistortionMeas
    ______________      __________      __________________      ___________________      ______________

      {[0.9985]}      [1x1 struct]        [1x1 struct]            [1x1 struct]              [1x1 stru
```

The values shown in measurement panes match the values shown in `data`. You can access the individual fields of `data` to obtain the various measurements programmatically.

Compare Peak Values

As an example, compare the peak values. Verify that the peak values obtained by `data.PeakFinder` match with the values seen in the `Spectrum Analyzer` window.

```
peakvalues = data.PeakFinder.Value
frequencieskHz = data.PeakFinder.Frequency/1000
```

```
peakvalues =

   26.8946
   26.2548
   -3.6120


frequencieskHz =

   15.0015
    1.0049
   16.4944
```

Save and Close the Model

```
save_system(model);
close_system(model);
```

See Also

Functions
`getMeasurementsData`

Objects
`SpectrumAnalyzerConfiguration`

Blocks
Lowpass Filter | Sine Wave | Spectrum Analyzer

DSP System Toolbox Simulink block Examples in Signal Input and Output Category

- “Write and Read Binary Files in Simulink” on page 10-2
- “Write and Read Matrix Data from Binary Files in Simulink” on page 10-6
- “Write and Read Fixed-Point Data from Binary Files in Simulink” on page 10-8
- “Write and Read Character Data from Binary Files in Simulink” on page 10-10
- “Change the Endianness of the Data in Simulink” on page 10-11

Write and Read Binary Files in Simulink

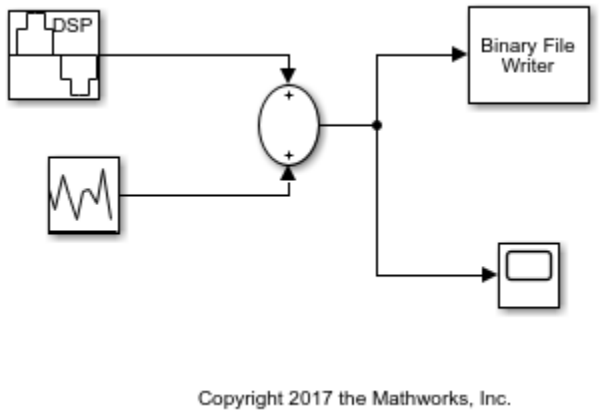
Create a binary file with a custom header using the Binary File Writer block. Write data to this file. Read the header and data using the Binary File Reader block.

Write the Data

Specify the file header in the **File header** parameter of the `Binary File Writer` block as `struct('A',[1 2 3 4],'B','x7')`. The block writes the header first, followed by the data to the `ex_file.bin` file. The data is a noisy sine wave signal with a frequency of 100 Hz containing 1000 samples per frame. The sample rate of the signal is 1000 Hz. Set the **Time span** of the **Time Scope** block to 1 second.

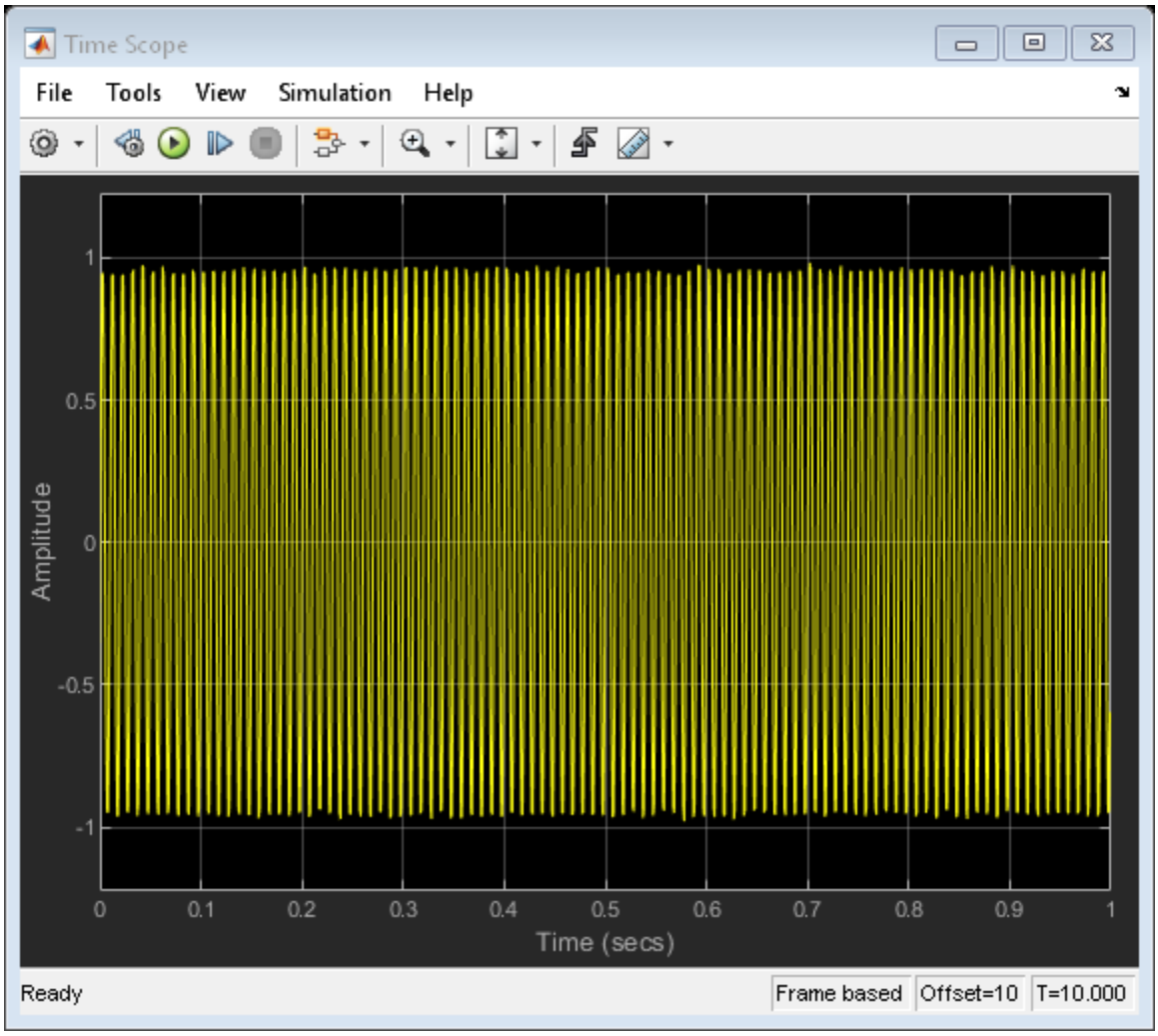
Open the model.

```
writeModel = 'writeData';
open_system(writeModel)
```



Run the model to write the data to `ex_file.bin`. Alternatively, view the data in a time scope.

```
sim(writeModel)
```



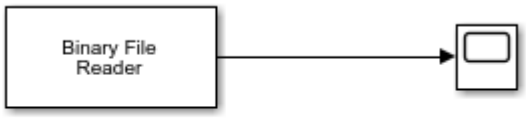
Read the Data

Read the data in `ex_file.bin` file using the `Binary File Reader` block. The data is read as a single channel (column) containing multiple frames, where each frame has 1000 samples. View the data in a time scope.

Specify the header using the **File header** parameter in the reader. If the exact header is not known, you must at least specify the prototype of the header, that is, its size and data type. In this example, the header prototype is `struct('A',[0 0 0 0],'B','-0')` which has the same format as the header structure.

Open the model.

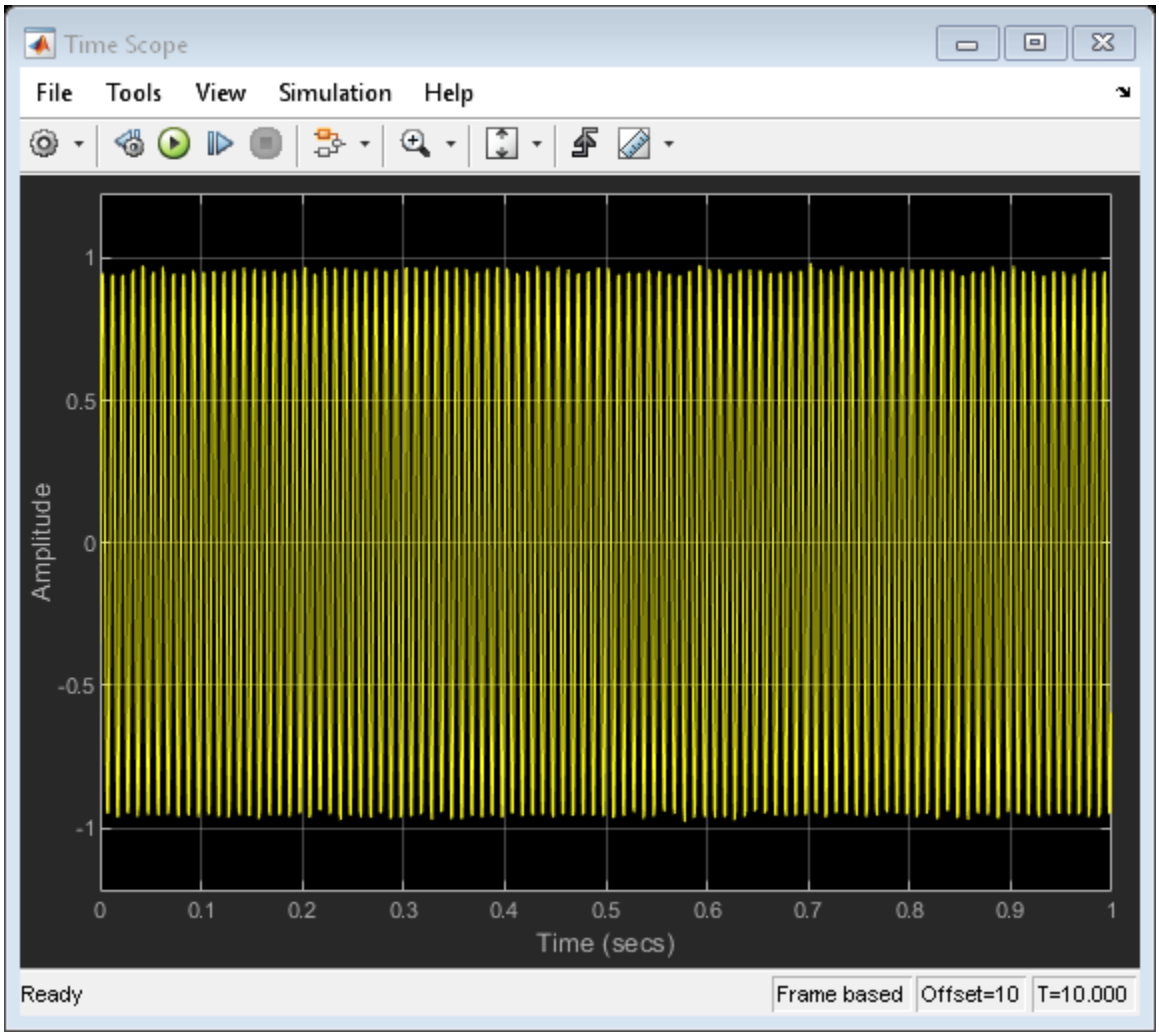
```
readModel = 'readData';
open_system(readModel)
```



Copyright 2017 the Mathworks, Inc.

Run the model to read the data. Alternatively, view the data in a time scope.

```
sim(readModel)
```



The output data in both the timescopes matches exactly. Once the processing is complete, close the models.

```
close_system(readModel);
close_system(writeModel);
```

See Also

Binary File Reader | Binary File Writer | Random Source | Sine Wave | Time Scope,

Write and Read Matrix Data from Binary Files in Simulink

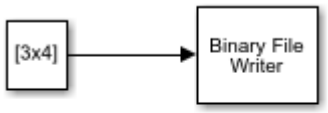
Use a Binary File Reader block to read real and complex matrix data from a binary file in a row-major format.

Write the Data

Write the matrix A = [1 2 3 8; 4 5 6 10; 7 8 9 11] to a binary file, `Matdata.bin` using the Binary File Writer block. The block writes the specified header, `struct('A',[1 2],'B','x7')` followed by the data.

Open the model.

```
open_system('writeMatrixData')
```



Copyright 2017 the Mathworks, Inc.

Run the model to write the data to `Matdata.bin`.

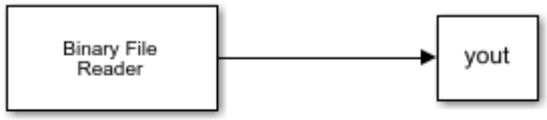
```
sim('writeMatrixData')
```

Read the Data

The `Binary File Reader` block reads the data in binary file `Matdata.bin` into 4 channels, with each channel containing 5 samples. The **File header** parameter of the reader specifies the header of the data. If the exact header is not known, you must at least specify the prototype of the header, that is, its size and data type.

Open the model.

```
open_system('readMatrixData')
```



Copyright 2017 the Mathworks, Inc.

Run the model to read the data. Display the output data variable, `yout`.

```
sim('readMatrixData')
display(yout)
```

```
yout =

     1     2     3     8
     4     5     6    10
     7     8     9    11
     0     0     0     0
     0     0     0     0
```

Each frame of `yout` contains frames of the matrix A, followed by zeros to complete the frame. The original matrix A contains 4 channels with 3 samples in each channel. The reader is specified to read data into 4 channels, with each channel containing 5 samples. Because there are not enough samples to complete the frame, the reader appends zeros at the end of each frame.

If you select the **Data is complex** parameter, the reader reads the data as an *M*-by- *N* matrix of complex values, where *M* and *N* are specified by the **Samples per frame** and **Number of channels** parameters, respectively. Select the **Data is complex** parameter and run the model.

```
set_param('readMatrixData/Binary File Reader','IsDataComplex','on')
sim('readMatrixData')
display(yout)
```

```
yout =

   1.0000 + 2.0000i   3.0000 + 8.0000i   4.0000 + 5.0000i   6.0000 +10.0000i
   7.0000 + 8.0000i   9.0000 +11.0000i   0.0000 + 0.0000i   0.0000 + 0.0000i
   0.0000 + 0.0000i   0.0000 + 0.0000i   0.0000 + 0.0000i   0.0000 + 0.0000i
   0.0000 + 0.0000i   0.0000 + 0.0000i   0.0000 + 0.0000i   0.0000 + 0.0000i
   0.0000 + 0.0000i   0.0000 + 0.0000i   0.0000 + 0.0000i   0.0000 + 0.0000i
```

The block reads the data as interleaved real and imaginary components. If there are not enough samples in the binary file to complete the matrix, the reader fills those samples with zeros.

If you make any changes to the model, save the model before closing.

```
save_system('readMatrixData')
close_system('readMatrixData')
close_system('writeMatrixData')
```

See Also

Binary File Reader | Binary File Writer | To Workspace

Write and Read Fixed-Point Data from Binary Files in Simulink

The Binary File Writer and Binary File Reader blocks do not support writing and reading fixed-point data. As a workaround, you can write the stored integer portion of the `fi` data, read the data, and use this value to reconstruct the `fi` data.

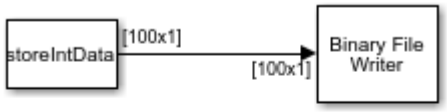
Write the Fixed-Point Data

Create a `fi` object to represent 100 signed random numbers with a word length of 14 and a fraction length of 12.

```
data = randn(100,1);
fiDataWriter = fi(data,1,14,12);
storeIntData = storedInteger(fiDataWriter);
```

Write the stored integer portion of the `fi` object to the data file `myFile.dat`. The built-in data type is `int16`, which can be computed using `class(storeIntData)`.

```
writeModel = 'writeFixedData';
open_system(writeModel)
sim(writeModel)
```

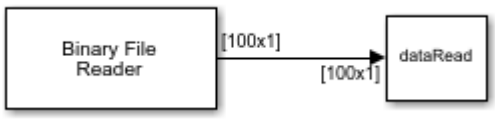


Copyright 2017 the Mathworks, Inc.

Read the Fixed-Point Data

Specify the reader to read the stored integer data as `int16` data with 100 samples per data frame.

```
readModel = 'readFixedData';
open_system(readModel)
sim(readModel)
```



The real-world value of the fixed-point number can be represented using `2^[-fractionLength*storedInteger]`. If you know the signedness, word length, and fraction length of the fixed-point data, you can reconstruct the `fi` data using `fi(realValue,signedness,wordLength,fractionLength)`. In this example, the data is signed with a word length of 14 and a fraction length of 12.

```
fractionLength = 12;
wordLength = 14;
```

```
realValue = 2^(-fractionLength)*double(dataRead);
fiDataReader = fi(realValue,1,wordLength,fractionLength);
```

Verify that the writer data is same as the reader data.

```
isequal(fiDataWriter,fiDataReader)
```

```
ans =

  logical

   1
```

See Also

Binary File Reader | Binary File Writer | Signal From Workspace | To Workspace

Write and Read Character Data from Binary Files in Simulink

The Binary File Writer and Binary File Reader blocks do not support writing and reading characters. As a workaround, cast character data to one of the built-in data types and write the integer data. After the reader reads the data, convert the data to a character using the `char` function.

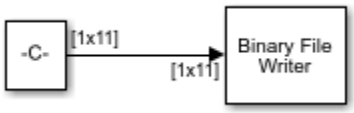
Write the Character Data

Cast the character data, `'binary_file'` into `uint8` using the `cast` function.

```
data = 'binary_file';
```

Write the cast data to the data file `myCharFile.dat`.

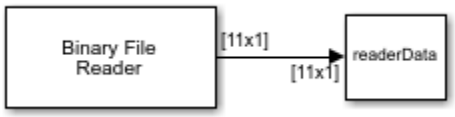
```
writeModel = 'writeCharData';
open_system(writeModel)
sim(writeModel)
```



Read the `uint8` Data

Specify the reader to read the cast data as `uint8` data.

```
readModel = 'readCharData';
open_system(readModel)
sim(readModel);
```




```
charData = char(readerData);
```

Verify that the writer data is same as the reader data. By default, the reader returns the data in a column-major format.

```
strcmp(data,charData.')
```

```
ans =

  logical

   1
```

See Also

Binary File Reader | Binary File Writer | Constant | To Workspace

Change the Endianness of the Data in Simulink

By default, the Binary File Reader block uses the endianness of the host machine. To change the endianness, such as when the host machine that writes the data does not have the same endianness as the host machine that reads the data, use the `swapbytes` function.

Write a numeric array in big endian format into the file `bigEndian.dat`. Read the data using the `Binary File Reader` block. The reader reads the data in little endian format.

```
fid = fopen('bigEndian.dat','w','b');
fwrite(fid,[1 2 3 4 5 6 7 8],'double');
fclose(fid);
```

Open and simulate the model.

```
model = 'changeEndian';
open_system(model)
sim(model)
```

Display the data variable, *x*.

```
display(x)
```

```
x =

  1.0e-318 *

    0.3039
    0.0003
    0.0104
    0.0206
    0.0256
    0.0307
    0.0357
    0.0408
```

The array `x` does not match the original data. Change the endianness of `x` using the `swapbytes` function.

```
y = swapbytes(x);
display(y)
```

```
y =

     1
     2
     3
     4
```

```
5
6
7
8
```

That array `y` matches the original data.

See Also

Functions
`fclose` | `fopen` | `fwrite` | `swapbytes`

Blocks
Binary File Reader | Constant

Simulink Block Examples in Signal Generation and Operations Category

- “Delay Signal Using Multitap Fractional Delay” on page 11-2
- “Bidirectional Linear Sweep” on page 11-7
- “Unidirectional Linear Sweep” on page 11-10
- “When Sweep Time Is Greater than Target Time” on page 11-12
- “Sweep with Negative Frequencies” on page 11-14
- “Aliased Sweep” on page 11-17
- “Generate Discrete Impulse with Three Channels” on page 11-19
- “Generate Unit-Diagonal and Identity Matrices” on page 11-20
- “Generate Five-Phase Output from the Multiphase Clock Block” on page 11-21
- “Count Down Through Range of Numbers” on page 11-23
- “Import Frame-Based Signal From Workspace” on page 11-25
- “Import 3-D Array From Workspace” on page 11-26
- “Generate Sample-Based Sine Waves” on page 11-27
- “Generate Frame-Based Sine Waves” on page 11-28
- “Design an NCO Source Block” on page 11-29
- “Generate Constant Ramp Signal” on page 11-32
- “Averaged Power Spectrum of Pink Noise” on page 11-33
- “Downsample a Signal” on page 11-35
- “Sample and Hold a Signal” on page 11-38
- “Generate and Apply Hamming Window” on page 11-41
- “Convert Sample Rate of Speech Signal” on page 11-44
- “Unwrap Signal” on page 11-46
- “Convolution of Two Inputs” on page 11-48
- “Select Rows or Columns from Matrices” on page 11-50
- “Convert 2-D Matrix to 1-D Array” on page 11-51

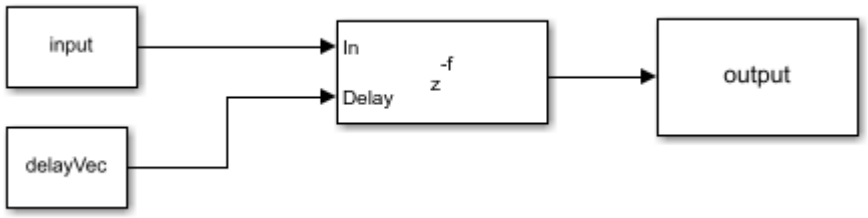
Delay Signal Using Multitap Fractional Delay

Delay the input signal using the Variable Fractional Delay block. Each delay value is unique and can vary from sample to sample within a frame, and can vary across channels. You can compute multiple delayed versions of the same input signal concurrently by passing a delay input with the appropriate dimension.

Consider the input to be a random signal with one channel and a frame size of 10. Apply a delay of 4.8 and 8.2 samples concurrently.

Open the model.

```
model = 'MultitapFractionalDelay';
open_system(model)
```



Run the model.

```
input = randn(10,1) %#ok
```

```
input =

    0.5377
    1.8339
   -2.2588
    0.8622
    0.3188
   -1.3077
   -0.4336
    0.3426
    3.5784
    2.7694
```

```
delayVec = [4.8 8.2]; %#ok
sim(model)
display(output)
```

```
output =

         0         0
         0         0
         0         0
         0         0
    0.1075         0
    0.7969         0
    1.0153         0
```

```
   -1.6346         0
    0.7535    0.4301
   -0.0065    1.5746
```

Each channel in the output is delayed by 4.8 and 8.2 samples, respectively. The block uses the 'Linear' interpolation method to compute the delayed value. For more details, see 'Algorithms' in the Variable Fractional Delay block page.

For the same delay vector, if the input has 2 channels, each element of the delay vector is applied on the corresponding channel in the input.

```
input = randn(10,2);

sim(model);
display(output);


output =

         0         0
         0         0
         0         0
         0         0
   -0.2700         0
   -0.4729         0
    2.5730         0
    0.5677         0
    0.0925    0.5372
    0.5308   -0.8317
```

To compute multiple delayed versions of the two-dimensional input signal, pass the delay vector as a three-dimensional array. The third dimension contains the taps or delays to apply on the signal. If you pass a non-singleton third dimension (1-by-1-by-P), where P represents the number of taps, the same tap is applied across all the channels. Pass the delays [4.8 8.2] in the third dimension.

```
clear delayVec;
delayVec(1,1,1) = 4.8;
delayVec(1,1,2) = 8.2; %#ok
whos delayVec
```

```
  Name          Size             Bytes  Class     Attributes

  delayVec      1x1x2               16  double
```

`delayVec` is a 1-by-1-by-2 array. Pass the two-dimensional input to the `Variable Fractional Delay` block with this delay vector.

```
sim(model)
display(output)


output(:,:,1) =

         0         0
         0         0
```

```
         0         0
         0         0
   -0.2700    0.1343
   -0.4729    0.2957
    2.5730   -0.8225
    0.5677    0.8998
    0.0925    1.4020
    0.5308    0.5981


output(:,:,2) =

         0         0
         0         0
         0         0
         0         0
         0         0
         0         0
         0         0
         0         0
   -1.0799    0.5372
    2.1580   -0.8317
```

```
whos output
```

```
  Name         Size              Bytes  Class     Attributes

  output      10x2x2               320  double
```

`output(:,:,1)` represents the input signal delayed by 4.8 samples. `output(:,:,2)` represents the input signal delayed by 8.2 samples. The same delay is applied across all the channels.

In addition, if you pass a non-singleton second dimension (1-by-L-by-P), where *L* is the number of input channels, taps vary across channels. Apply the delay vectors [2.3 3.5] and [4.4 5.6] to compute the two delayed versions of the input signal.

```
clear delayVec;
delayVec(1,1,1) = 2.3;
delayVec(1,2,1) = 3.5;
delayVec(1,1,2) = 4.4;
delayVec(1,2,2) = 5.6; %#ok
whos delayVec
```

```
  Name          Size              Bytes  Class     Attributes

  delayVec      1x2x2                32  double
```

```
sim(model)
display(output)
```

```
output(:,:,1) =

         0         0
         0         0
   -0.9449         0
```

```
    1.7195    0.3357
    1.4183   -0.2680
    0.1735   -0.2451
    0.4814    1.1737
    0.0709    1.0596
   -0.1484    0.7618
    1.0055    0.8808


output(:,:,2) =

         0         0
         0         0
         0         0
         0         0
   -0.8099         0
    1.2810    0.2686
    1.6492   -0.0801
    0.2523   -0.4376
    0.4036    1.0824
    0.1629    1.1737
```

```
whos output
```

```
  Name         Size              Bytes  Class     Attributes

  output      10x2x2               320  double
```

output(:,:,1) contains the input signal delayed by the vector [2.3 3.5]. output(:,:,2) contains the input signal delayed by the vector [4.4 5.6].

To vary the delay within a frame from sample to sample, the first dimension of the delay vector (N-by-1-by-P or N-by-L-by-P) must equal the frame size of the input (N-by-L). Pass a delay vector of size 10-by-1-by-2.

```
clear delayVec;
delayVec(:,1,1) = 3.1:0.1:4;
delayVec(:,1,2) = 0.1:0.1:1;
whos delayVec
```

```
  Name           Size              Bytes  Class     Attributes

  delayVec      10x1x2               160  double
```

```
sim(model)
display(output)
```

```
output(:,:,1) =

         0         0
         0         0
         0         0
   -0.8099    0.4029
    0.8425   -0.2680
    2.1111   -0.4376
```

```
    0.4889    0.9911
    0.0925    1.4020
    0.6228    0.5435
   -0.2050    1.0347


output(:,:,2) =

   -1.2149    0.6043
    2.1580   -0.8317
    1.4183    0.1398
    0.2523    1.2650
    0.3258    1.0596
    0.3469    0.7072
   -0.1807    0.9424
    0.1986    0.5208
    1.4816   -0.2437
    1.4090    0.2939
```

Delay varies across each element in a channel. Same set of delay values apply across all channels. `delayVec(:,1,1)` applies to the first delayed signal and `delayVec(:,1,2)` applies to the second delayed signal.

See Also

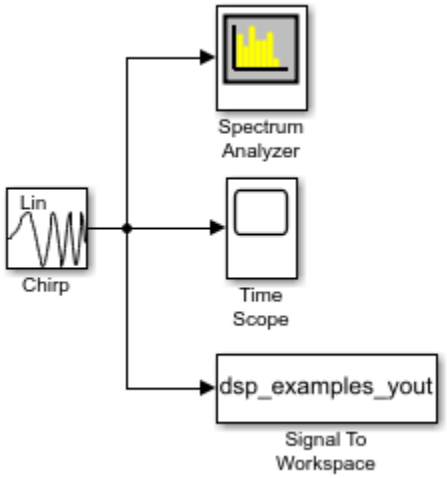
Blocks
Delay | Unit Delay | Variable Integer Delay

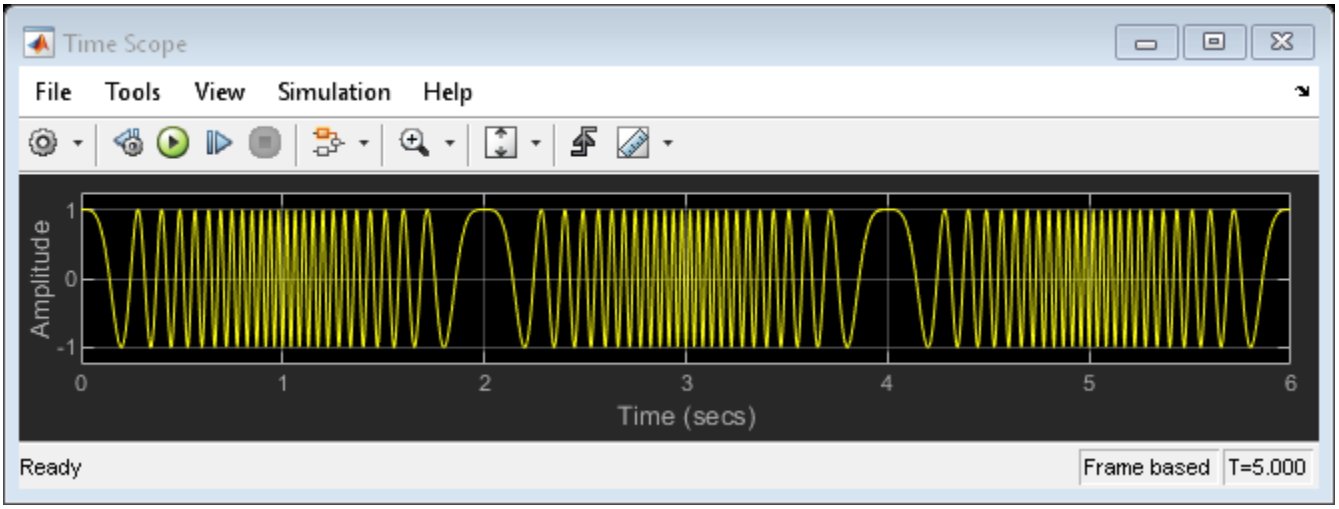
Bidirectional Linear Sweep

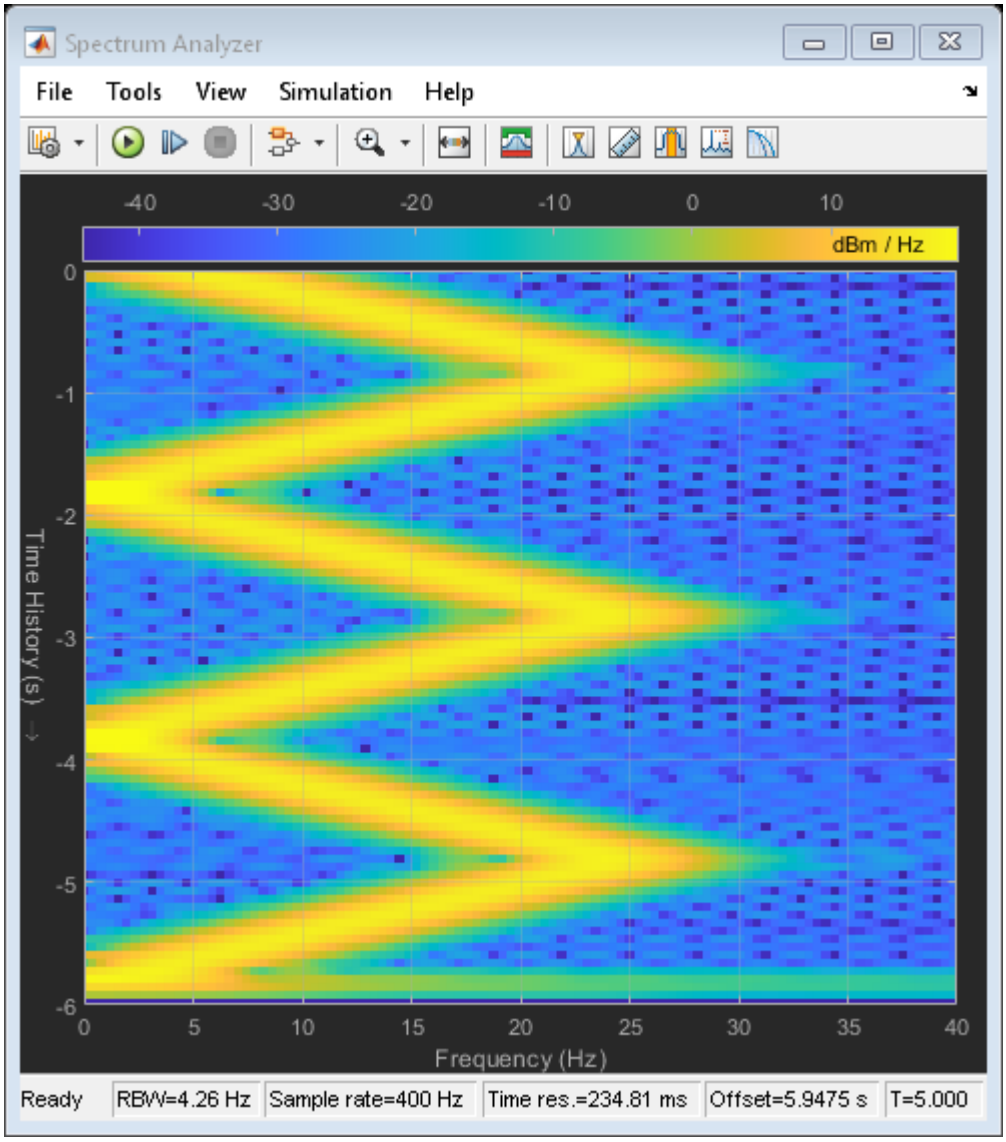
In this example, the Chirp block outputs a bidirectional, linearly swept chirp signal, which is displayed by the Time Scope and the Spectrum Analyzer. You can also export the signal to the MATLAB workspace by the Signal To Workspace block.

To create a bidirectional sweep, set the **Sweep mode** parameter to `Bidirectional`. Specify the final frequency of a bidirectional sweep by setting **Target time** equal to **Sweep time**, in which case the **Target frequency** becomes the final frequency in the sweep. Note that in the bidirectional sweep, the period of the sweep is twice the **Sweep time** of the unidirectional sweep.

Run the model to see the output in the Time Scope and the Spectrum Analyzer.

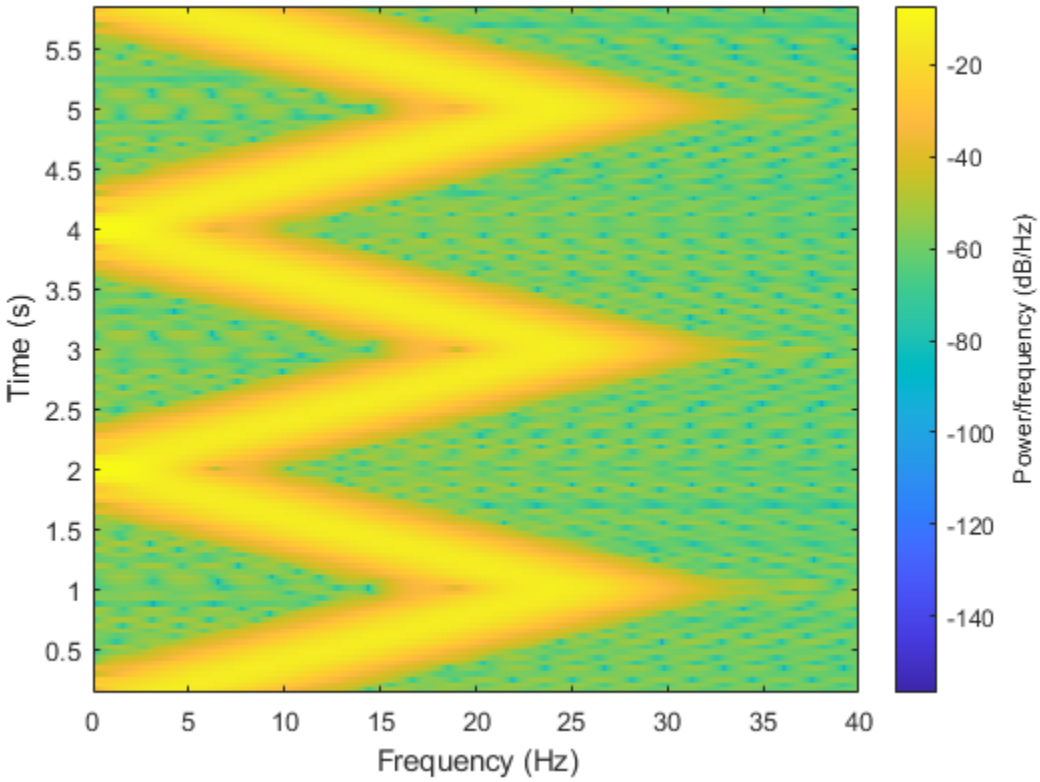






You can also view the spectrogram by saving the output to the workspace and using this command:

```
spectrogram(dsp_examples_yout,hamming(128),110,0:.01:40,400)
```



See Also

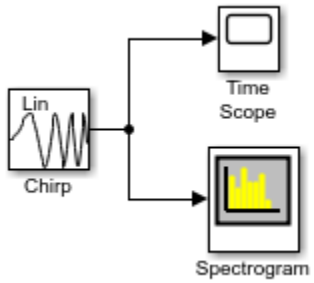
Blocks
Chirp | Signal To Workspace | Spectrum Analyzer | Time Scope

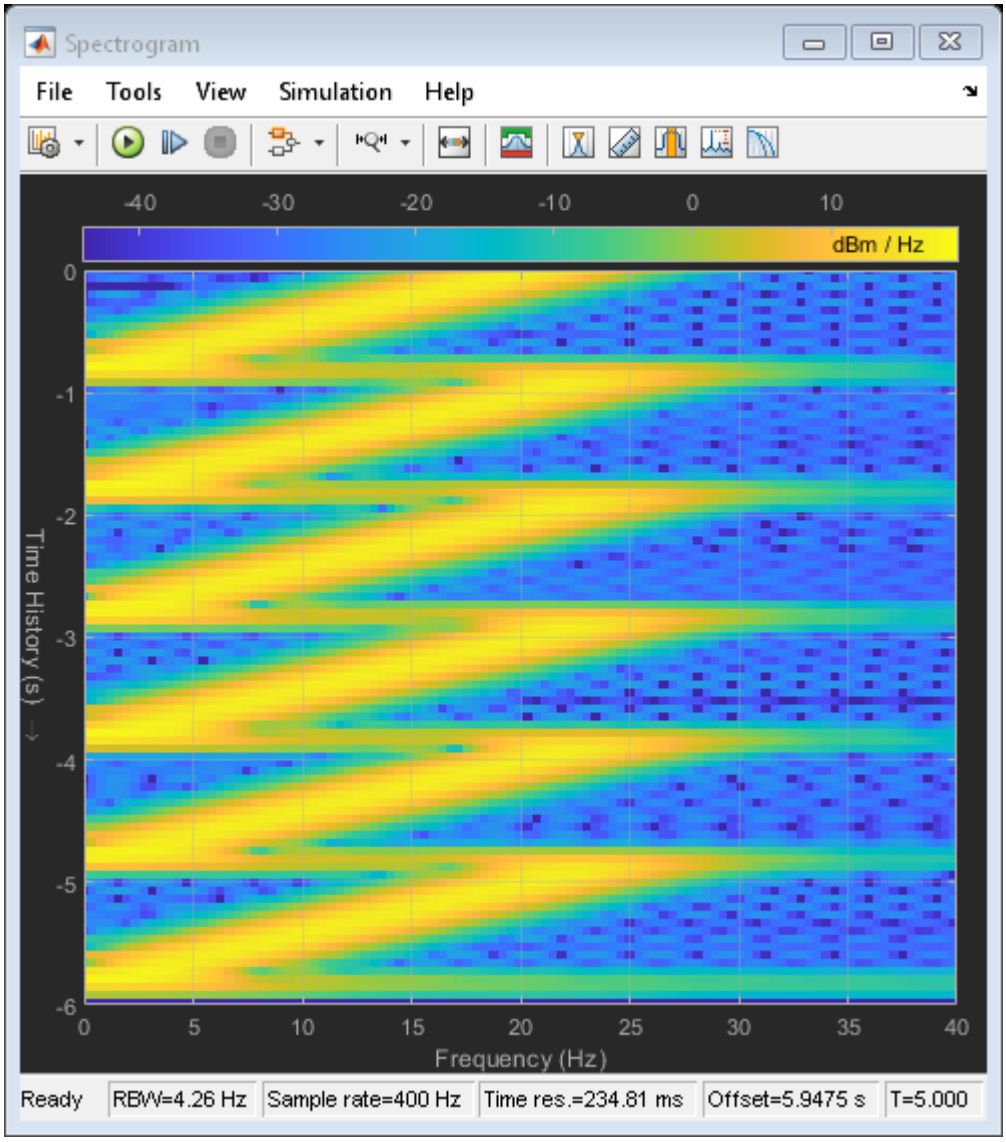
Unidirectional Linear Sweep

In this example, the Chirp block outputs a unidirectional, linearly swept chirp signal. The Time Scope displays the signal output in the time domain and the Spectrum Analyzer displays the spectrogram in the frequency domain.

To obtain a unidirectional sweep with known initial and final frequency values, in the Chirp block set the **Target time** equal to **Sweep time**. In which case, the **Target frequency** becomes the final frequency in the sweep. Since the **Target time** is set to equal **Sweep time** (1 second), the **Target frequency** (25 Hz) is the final frequency of the unidirectional sweep. This technique might not work for swept cosine sweeps. For details, see the “Frequency sweep” described for the **Frequency sweep** parameter.

Run the model to see the time domain output:





See Also

Blocks
Chirp | Spectrum Analyzer | Time Scope

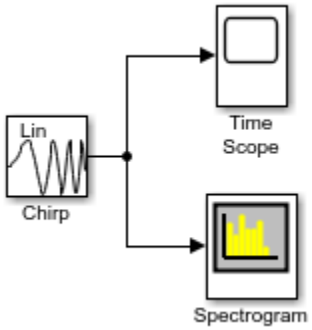
When Sweep Time Is Greater than Target Time

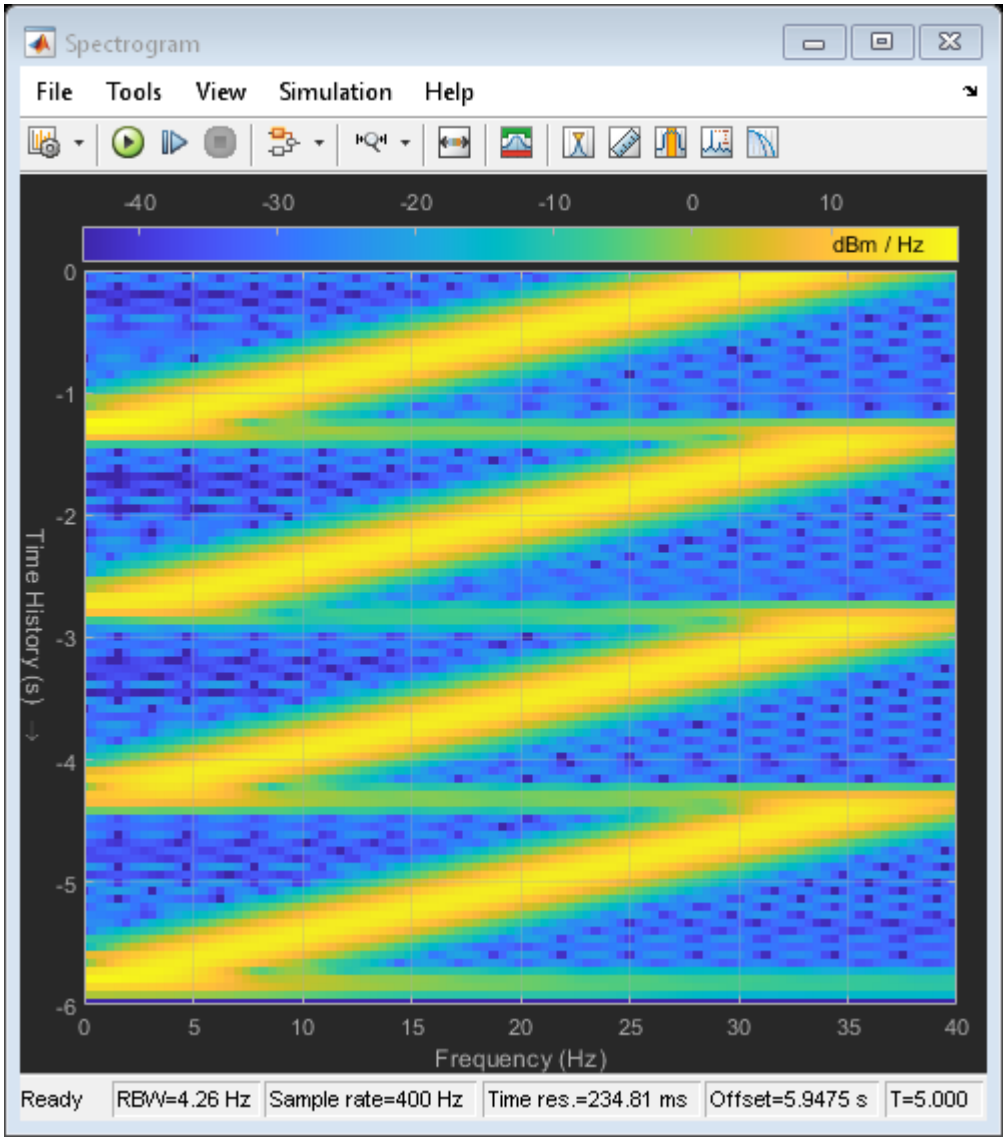
This example shows the unexpected behavior that might arise in the Chirp block when the **Sweep time** is greater than **Target time**. The Time Scope displays the signal output, and the Spectrum Analyzer displays the spectrogram in the frequency domain.

Set the **Sweep time** parameter to `1.5` and specify the final frequency of a bidirectional sweep by setting **Target time** equal to **Sweep time**. The sweep reaches the **Target frequency** (25 Hz) at the **Target time** (1 second), but since **Sweep time** is greater than **Target time**, the sweep continues on its linear path until one **Sweep time** (1.5 seconds) is traversed.

Unexpected behavior might arise when you set **Sweep time** greater than **Target time**.

Run the model to see the chirp signal output and a spectrogram of the frequency domain.





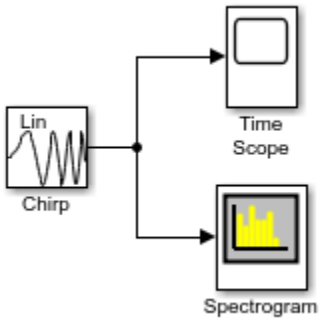
See Also

Blocks
Chirp | Spectrum Analyzer | Time Scope

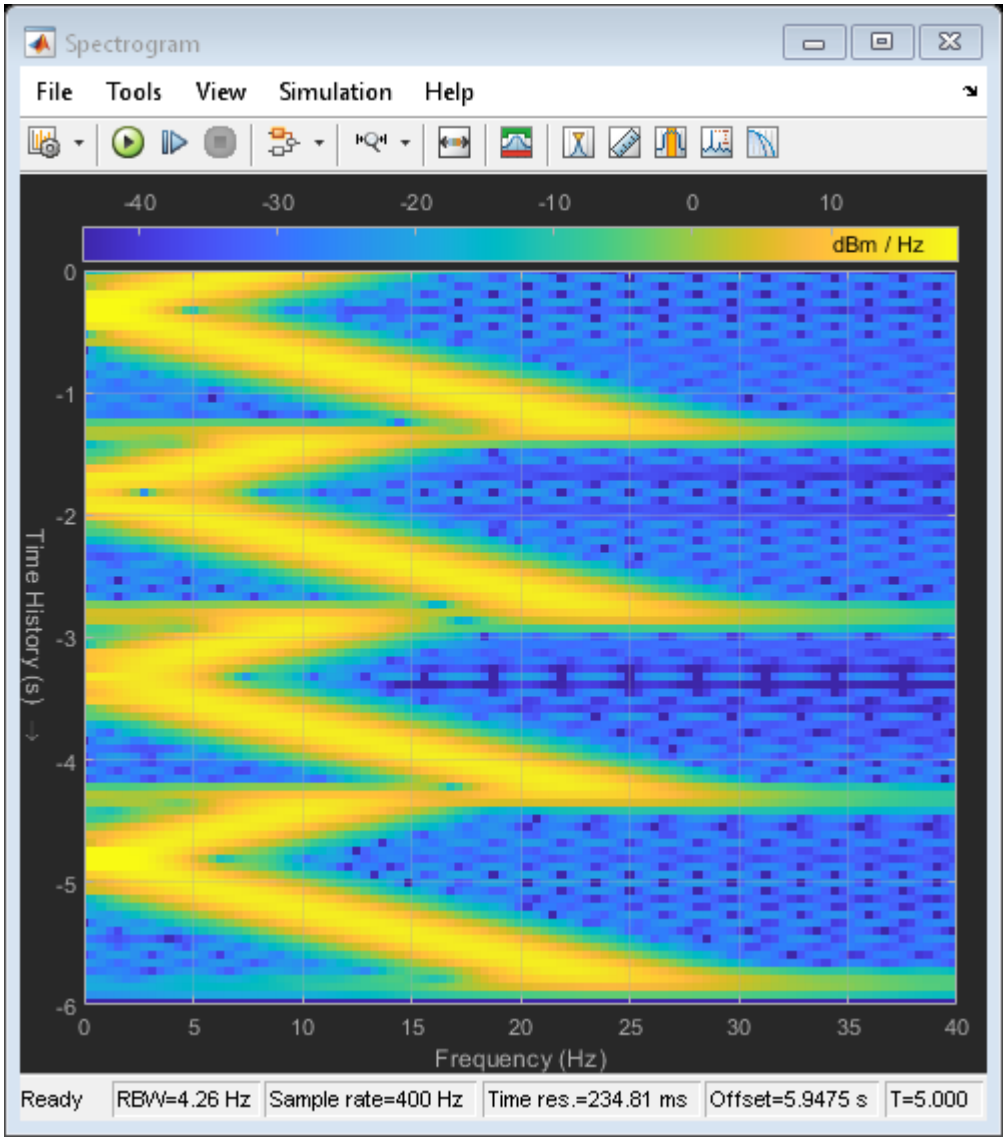
Sweep with Negative Frequencies

In this example, the Chirp block outputs a chirp signal containing negative frequencies. The Time Scope displays the signal output in the time domain, and the Spectrum Analyzer displays the spectrogram in the frequency domain.

Set the **Sweep time** to `1.5`, **Initial frequency** to `25`, **Target frequency** to `0`, and **Target time** equal to **Sweep time**. *The output chirp of this example might not behave as you expect* because the sweep contains negative frequencies between 1 and 1.5 seconds. The sweep reaches the **Target frequency** (0 Hz) at 1 second, then continues on its negative slope, taking on negative frequency values until it traverses one **Sweep time** (1.5 seconds).



Run the model to see the time domain output.


Spectrogram
File
Tools
View
Simulation
Help
-40
-30
-20
-10
0
10
dBm / Hz
0
-1
-2
-3
-4
-5
-6
Time History (s)
0
5
10
15
20
25
30
35
40
Frequency (Hz)
Ready
RBW=4.26 Hz
Sample rate=400 Hz
Time res.=234.81 ms
Offset=5.9475 s
T=5.000

See Also

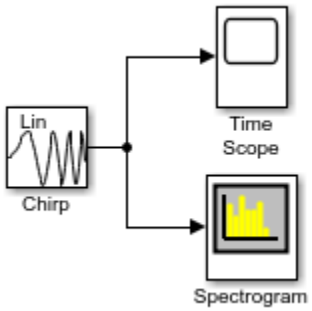
Blocks
Chirp | Spectrum Analyzer | Time Scope

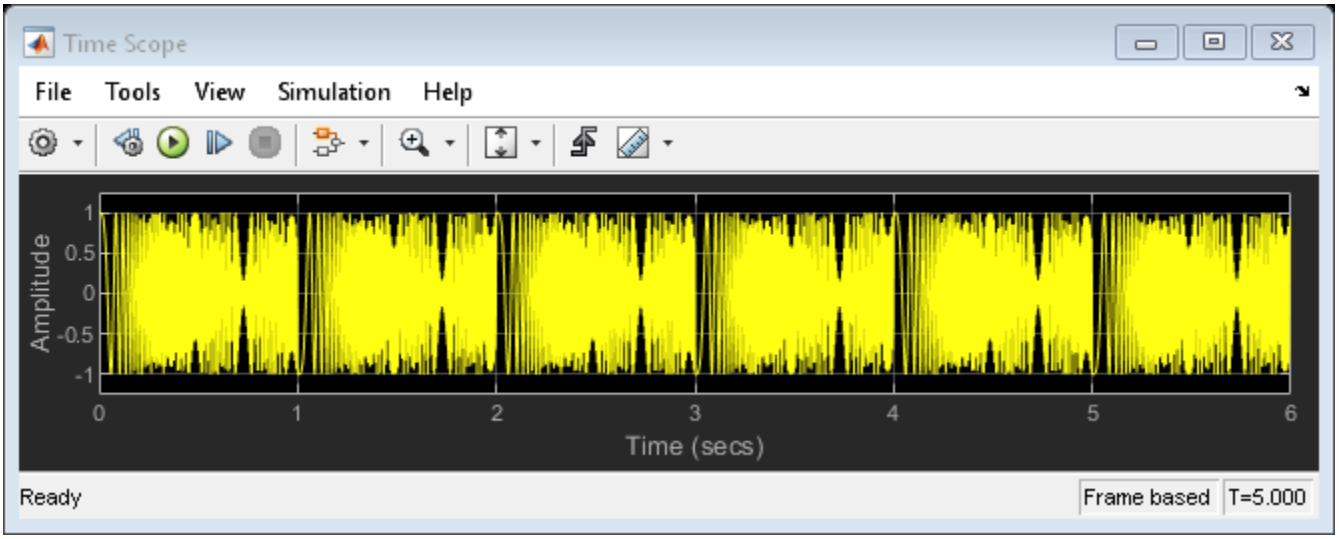
Aliased Sweep

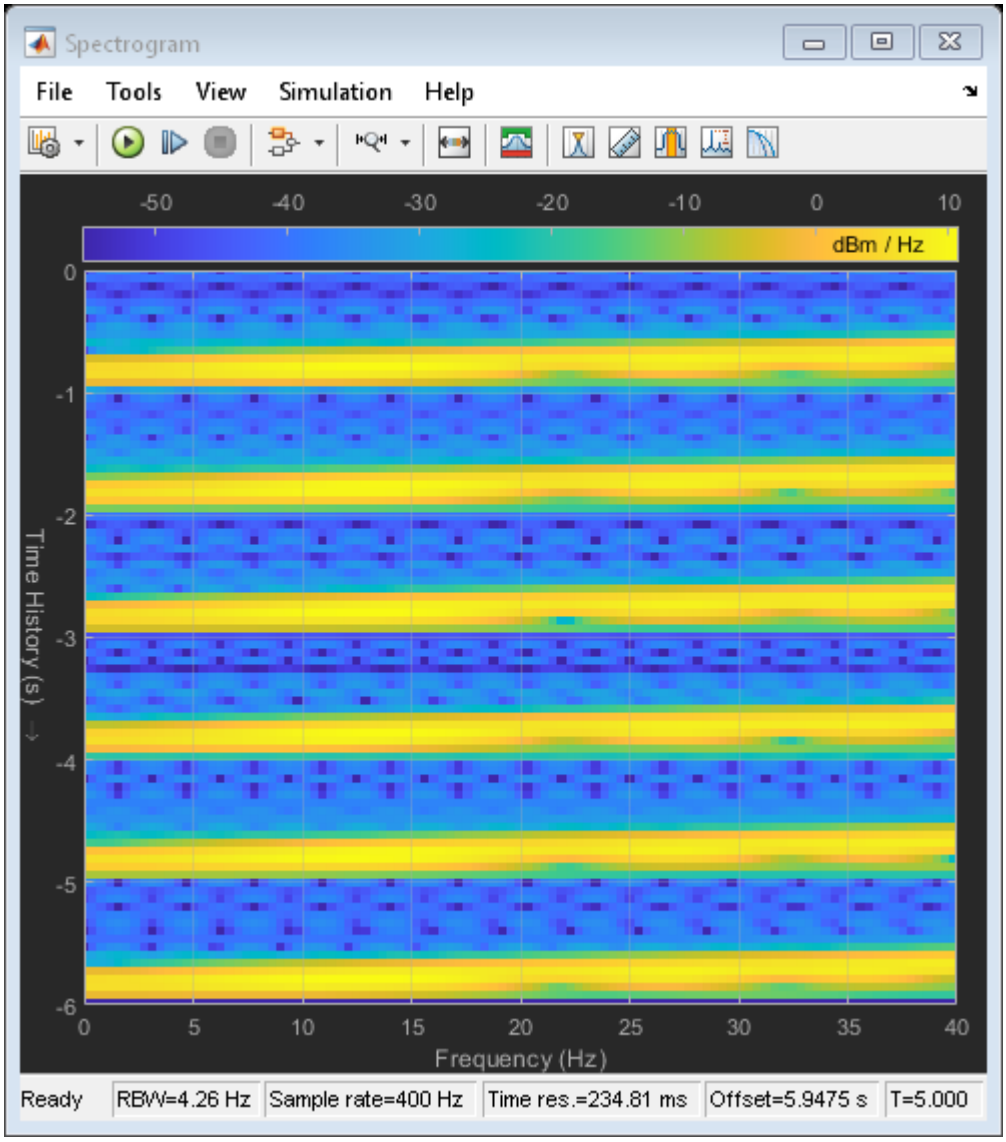
In this example, the Chirp block outputs a chirp signal containing negative frequencies. The Time Scope displays the signal output in the time domain, and the Spectrum Analyzer displays the spectrogram in the frequency domain.

Set the **Target frequency** to 275 and specify **Target time** equal to **Sweep time**. *The output chirp of this example might not behave as you expect* because the sweep contains frequencies greater than half the sampling frequency (200 Hz). If you unexpectedly get a chirp output with a spectrogram resembling the one following, your chirp's sweep might contain frequencies greater than half the sampling frequency.

Run the model to see the signal output and the spectrogram.







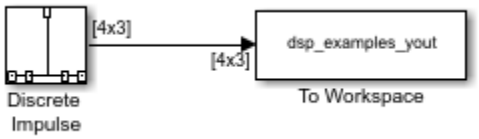
See Also

Blocks
Chirp | Spectrum Analyzer | Time Scope

Generate Discrete Impulse with Three Channels

This example shows how to generate a discrete impulse signal with three channels and a data type of `double`. The Discrete Impulse block has the following settings:

- **Delay** = `[0 3 5]`
- **Sample time** = `0.25`
- **Samples per frame** = `4`
- **Output data type** = `double`



Run the model and look at the output, `dsp_examples_yout`. The first few samples of each channel are shown below:

```
dsp_examples_yout(1:10,:)
ans =
     1     0     0
     0     0     0
     0     0     0
     0     1     0
     0     0     0
     0     0     1
     0     0     0
     0     0     0
     0     0     0
     0     0     0
```

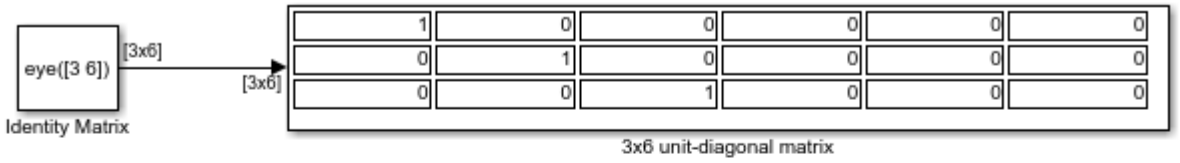
The block generates an impulse at sample 1 of channel 1 (first column), at sample 4 of channel 2 (second column), and at sample 6 of channel 3 (third column).

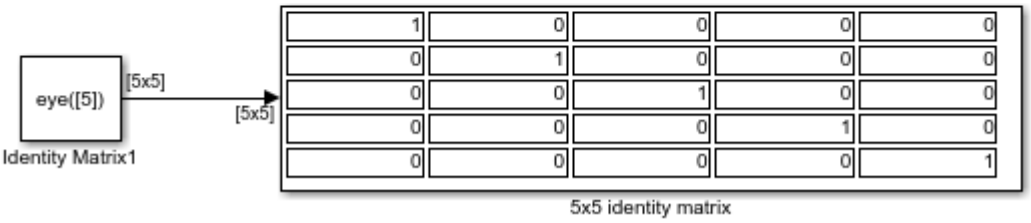
See Also

Blocks
Discrete Impulse | To Workspace

Generate Unit-Diagonal and Identity Matrices

This example shows how to generate a 3-by-6 unit-diagonal matrix and a 5-by-5 identity matrix using the Identity Matrix block.



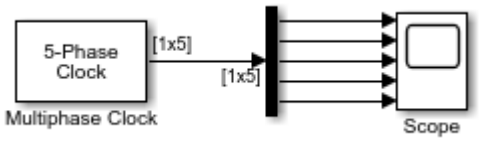


See Also

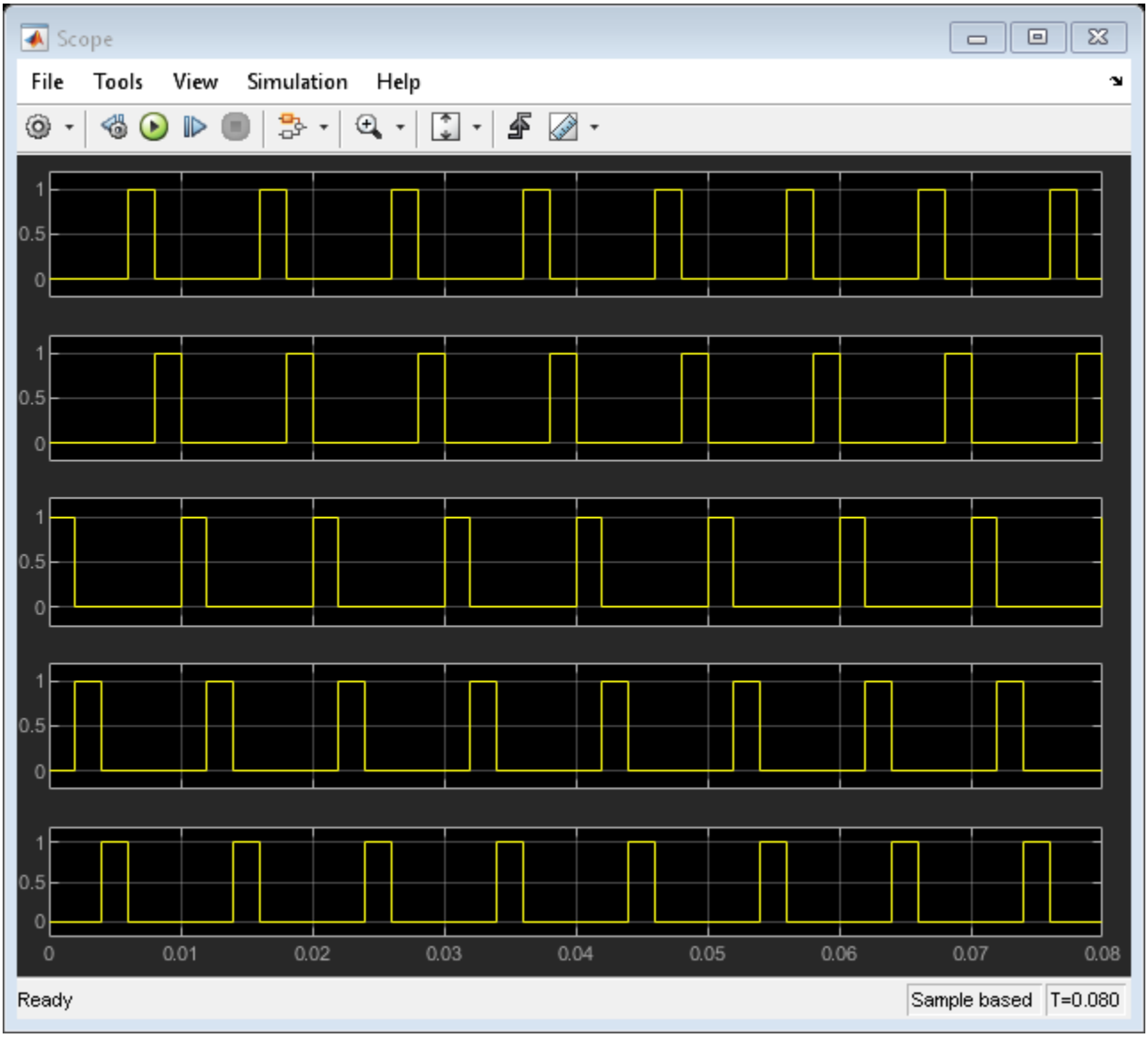
Blocks
Display | Identity Matrix

Generate Five-Phase Output from the Multiphase Clock Block

This example shows how to use the Multiphase Clock block to generate a 100 Hz five-phase output in which the third signal is first to become active. The block uses a high active level with a duration of one interval.



The Scope window below shows the output of the Multiphase Clock block. Note that the first active level appears at `t=0` on `y(3)`, the second active level appears at `t=0.002` on `y(4)`, the third active level appears at `t=0.004` on `y(5)`, the fourth active level appears at `t=0.006` on `y(1)`, and the fifth active level appears at `t=0.008` on `y(2)`. Each signal becomes active `1/(5*100)` seconds after the previous signal.

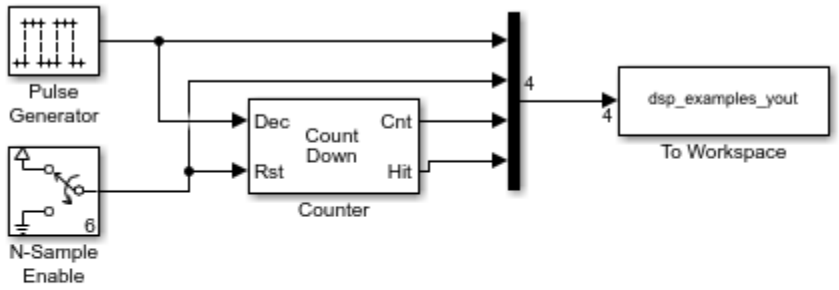


See Also

Blocks
Multiphase Clock | Scope

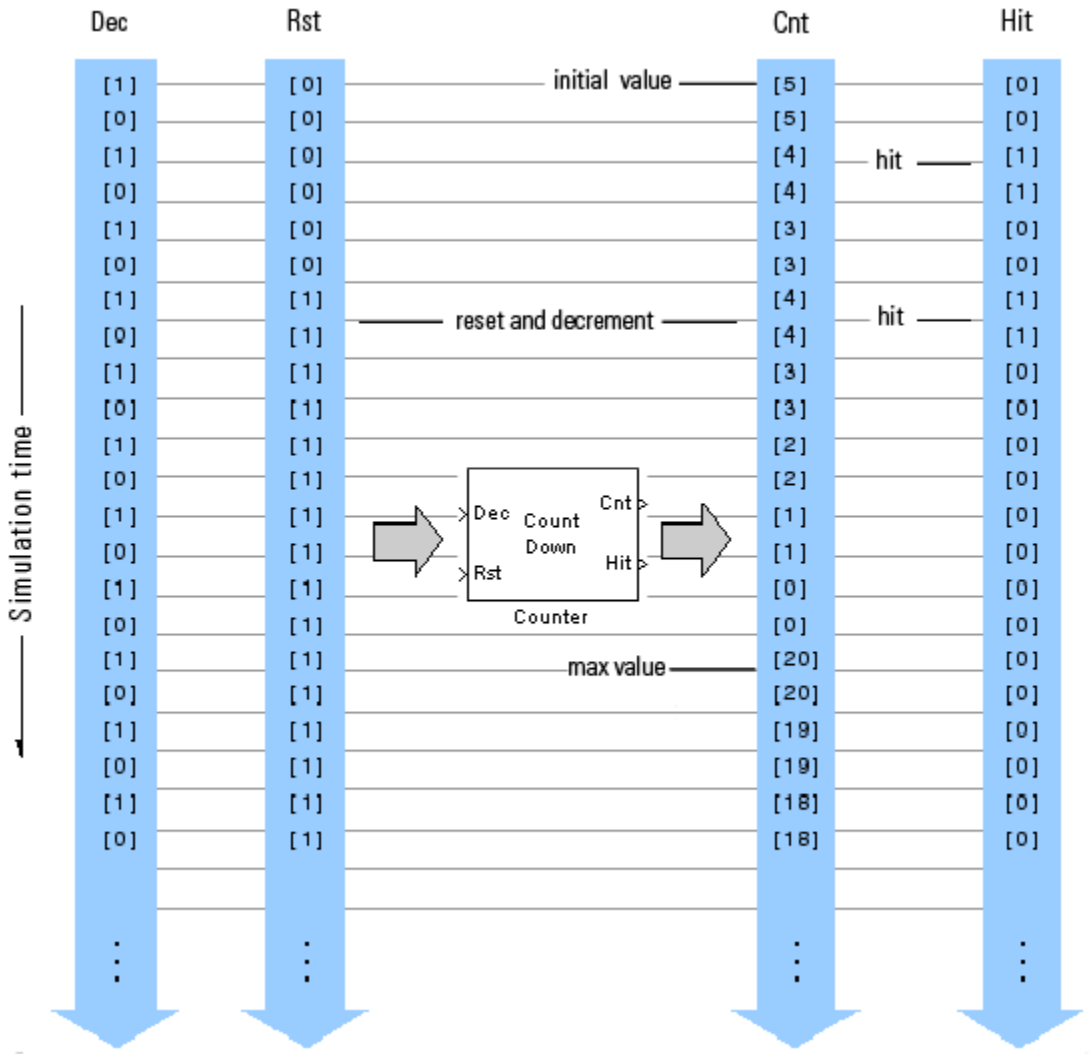
Count Down Through Range of Numbers

This example shows how to use the Counter block to count down through a range of numbers. The Pulse Generator block drives the **Dec** port of the Counter block, and the N-Sample Enable block triggers the **Rst** port. All inputs to and outputs from the Counter block are multiplexed into a single To Workspace block using a 4-port Mux block.



Copyright 2004-2010 The MathWorks, Inc.

The following figure shows the first 22 samples of the four-column output, `dsp_examples_yout`.



You can see that the seventh input sample to both the **Dec** and **Rst** ports of the Counter block represent trigger events (rising edges). When this occurs, the block first resets the counter to its initial value of `5`, and then immediately decrements the counter to `4`. When the counter reaches its minimum value of `0`, the block restarts the counter at its maximum value of `20` the next time a trigger event occurs at the **Dec** port.

See Also

Blocks

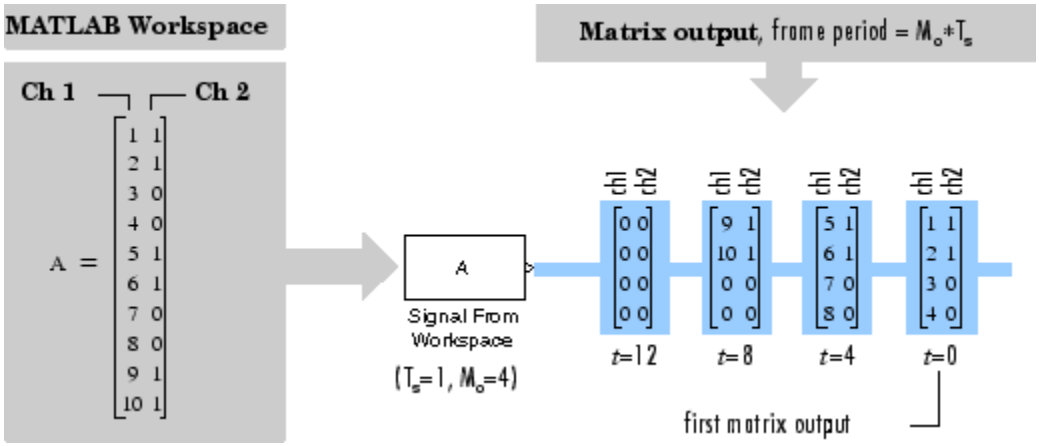
Counter | N-Sample Enable | Pulse Generator | To Workspace

Import Frame-Based Signal From Workspace

This example shows how to import a two-channel signal from the workspace matrix, `A`. The Signal From Workspace block parameters are configured as follows:

- **Sample time**: 1
- **Samples per frame**: 4
- **Form output after final value by**: `Setting to zero`

As shown in the following figure, the output has a frame size of 4 and a frame period of 4 seconds. All outputs after the third frame (at `t=8`) are zero.



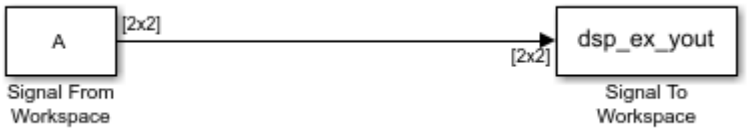
See Also

Blocks
Signal From Workspace | Signal To Workspace

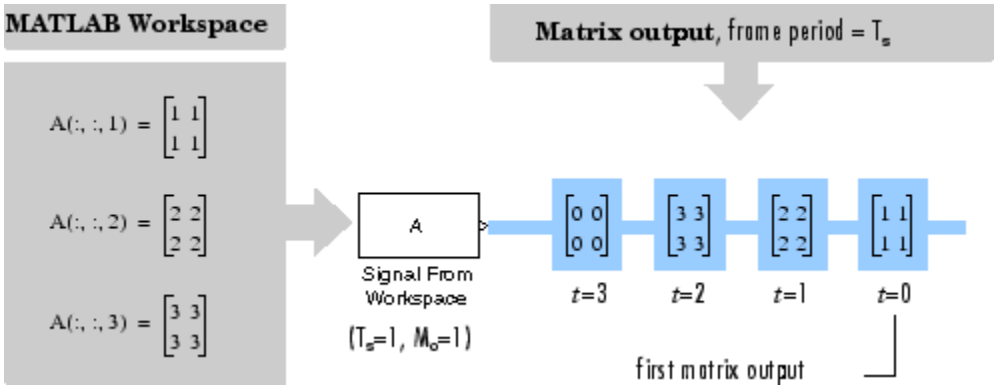
Import 3-D Array From Workspace

This example shows how to import a 3-D workspace array, `A`. The Signal From Workspace block parameters are configured as follows:

- **Sample time**: 1
- **Samples per frame**: 1
- **Form output after final data value by**: `Setting to zero`



The following figure illustrates how the block imports the 3-D array, `A`:

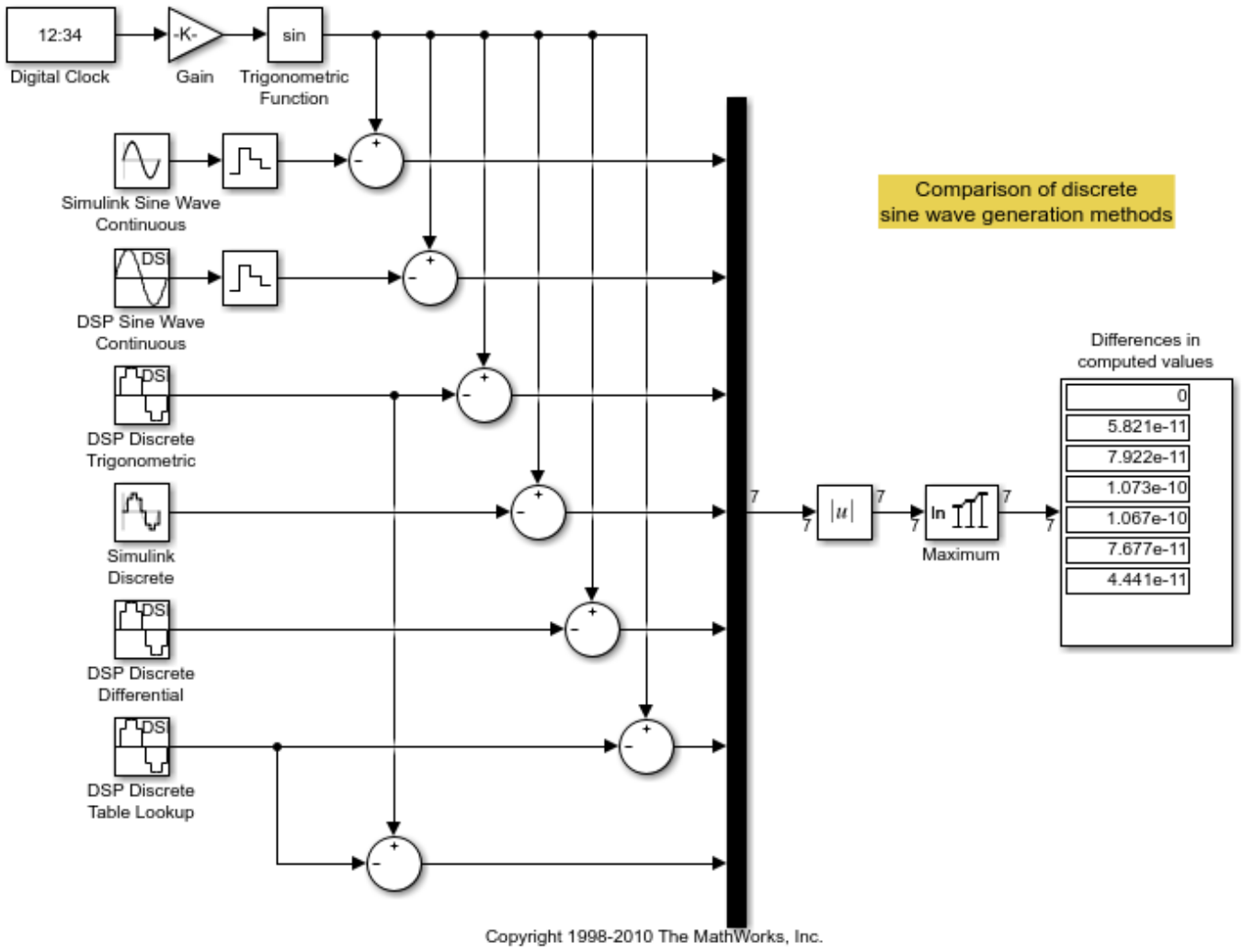


See Also

Blocks
Signal From Workspace | Signal To Workspace

Generate Sample-Based Sine Waves

This example compares the different methods of generating sample-based sine waves from the Sine Wave block in DSP System Toolbox.



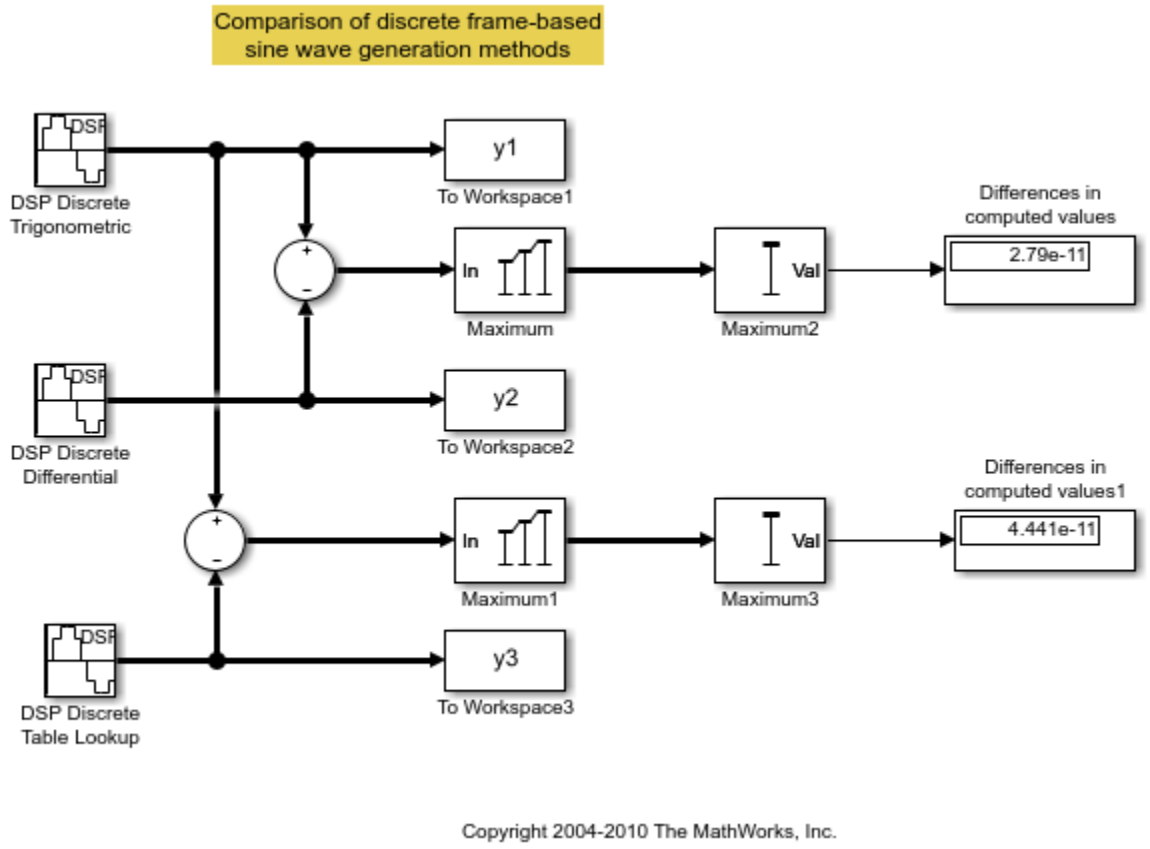
See Also

Blocks

Abs | Add | Digital Clock | Display | Gain | Maximum | Sine Wave | To Workspace | Trigonometric Function | Zero-Order Hold

Generate Frame-Based Sine Waves

This example compares the different methods of generating frame-based sine waves from the Sine Wave block in DSP System Toolbox.



See Also

Blocks

Add | Display | Maximum | Sine Wave | To Workspace

Design an NCO Source Block

This example shows how to design an NCO source block with the following specifications:

- Desired output frequency: $F_0 = 510\text{Hz}$
- Frequency resolution: $\Delta f = 0.05\text{Hz}$
- Spurious free dynamic range: $SFDR \geq 90\text{dB}$
- Sample period: $T_s = 1/8000\text{s}$
- Desired phase offset: $\pi/2$

1. Calculate the number of required accumulator bits from the equation for frequency resolution:

$$\Delta f = \frac{1}{T_s \cdot 2^N}\text{Hz}$$

$$0.05 = \frac{1}{\frac{1}{8000} \cdot 2^N}\text{Hz}$$

$$N = 18$$

Note that `N` must be an integer value. The value of `N` is rounded up to the nearest integer; 18 accumulator bits are needed to accommodate the value of the frequency resolution.

2. Using this best value of *N*, calculate the frequency resolution that will be achieved by the NCO block:

$$\Delta f = \frac{1}{T_s \cdot 2^N}\text{Hz}$$

$$\Delta f = \frac{1}{\frac{1}{8000} \cdot 2^{18}}\text{Hz}$$

$$\Delta f = 0.0305$$

3. Calculate the number of quantized accumulator bits from the equation for spurious free dynamic range and the fact that for a lookup table with 2^P entries, *P* is the number of quantized accumulator bits:

$$SFDR = (6P + 12)\,\text{dB}$$

$$96\text{dB} = (6P + 12)\text{dB}$$

$$P = 14$$

4. Select the number of dither bits. In general, a good choice for the number of dither bits is the accumulator word length minus the number of quantized accumulator bits; in this case 4.

5. Calculate the phase increment:

phase increment = $\text{round}(\frac{F_0 \cdot 2^N}{F_s})$

phase increment = $\text{round}(\frac{510 \cdot 2^{18}}{8000})$

phase increment = 16712

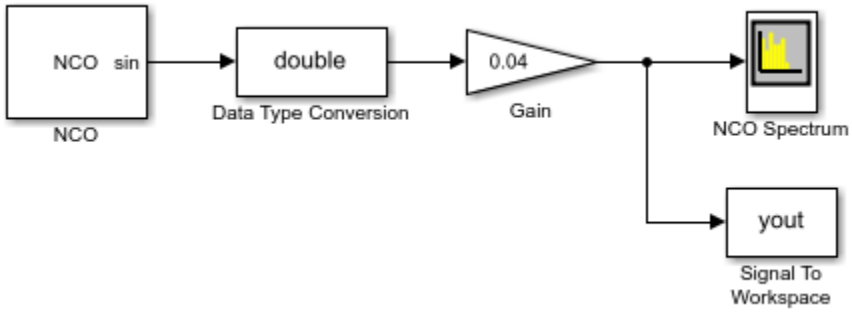
6. Calculate the phase offset, p_o, using the desired phase offset, $p_o desired$:

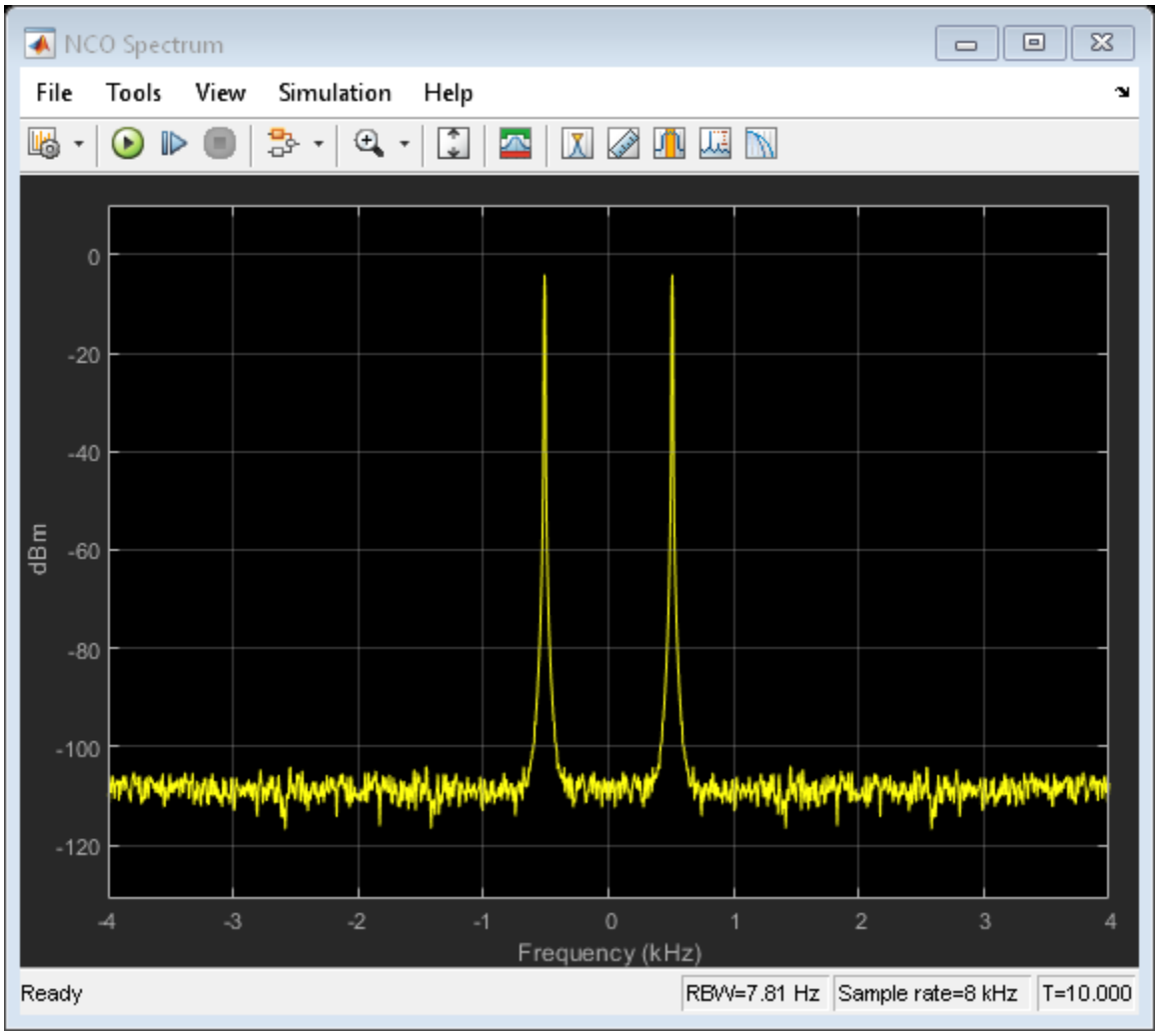
$$p_o = \frac{2^{accumulatorwordlength} \cdot p_o desired}{2\pi}$$

$$p_o = \frac{2^{18} \cdot \frac{\pi}{2}}{2\pi}$$

$$p_o = 65536$$

7. Open and simulate the model:





8. Experiment with the model to observe the effects on the output shown on the Spectrum Analyzer. For example, try turning dithering on and off, and try changing the number of dither bits.

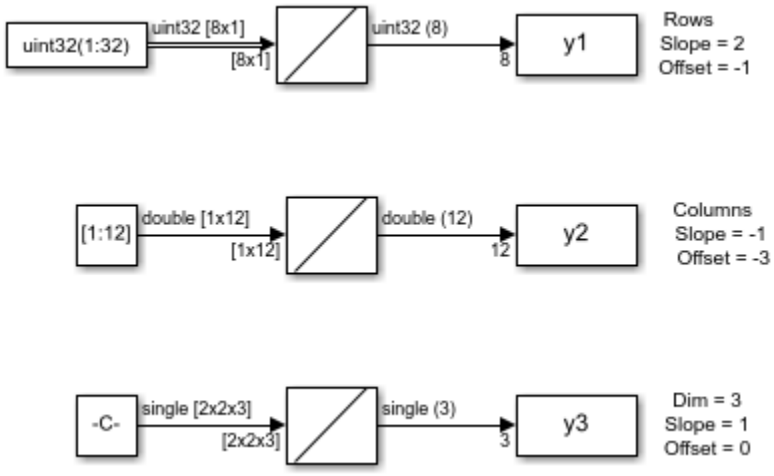
See Also

Blocks

Data Type Conversion | Gain | NCO | Signal To Workspace | Spectrum Analyzer

Generate Constant Ramp Signal

This example shows how to generate a ramp signal from the Constant Ramp block. The **Ramp length equals number of** parameter allows you to specify which dimension of the input signal determines the length of the generated constant ramp signal.



For N-D input signals, set the **Ramp length equals number of** parameter to `Elements in specified dimension`. Then specify the desired dimension using the **Dimension** parameter.

See Also

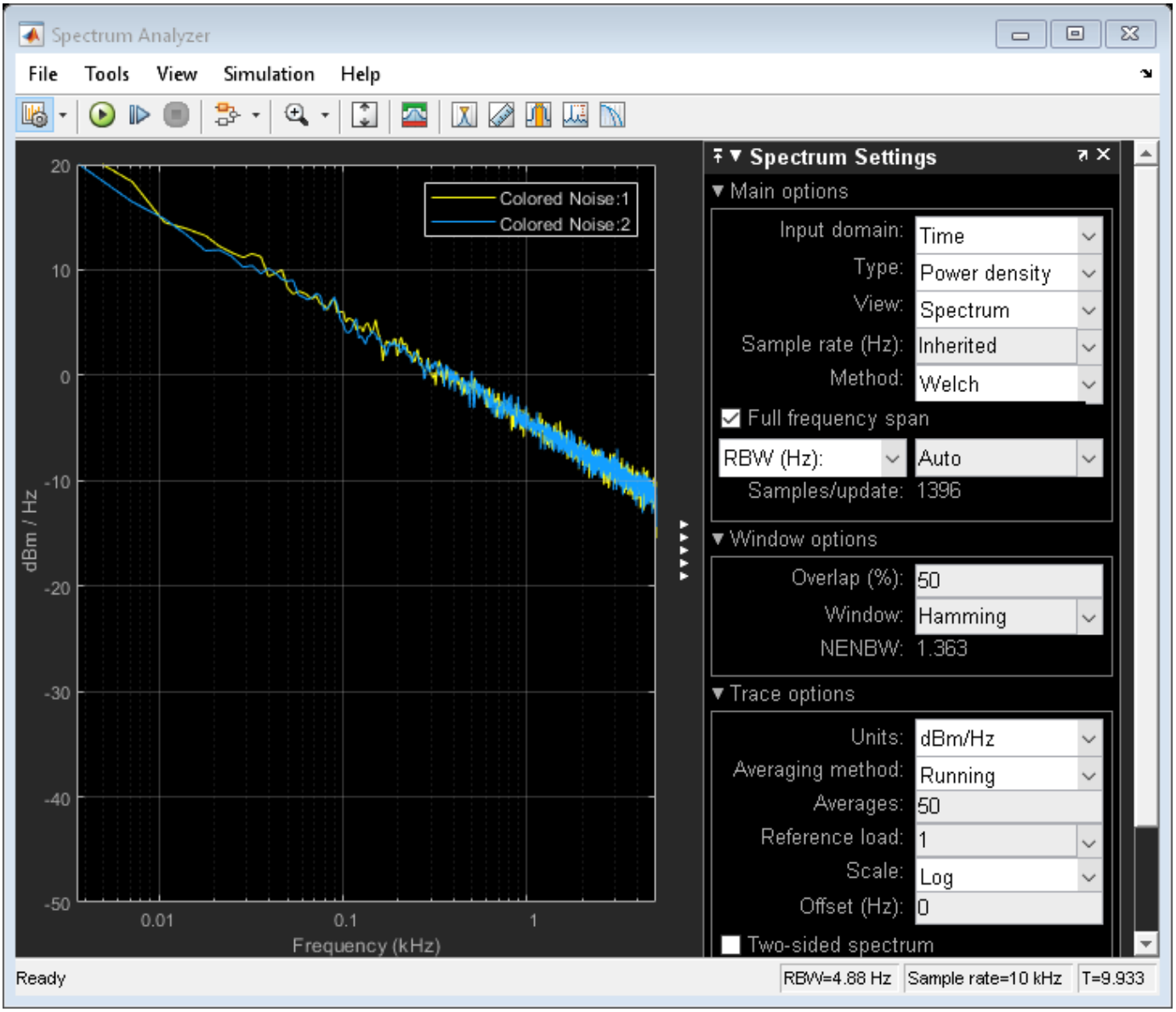
Blocks

Constant Ramp | DSP Constant (Obsolete) | Signal From Workspace | To Workspace

Averaged Power Spectrum of Pink Noise

This example shows how to generate two-channels of pink noise from the Colored Noise block and compute the power spectrum based on a running average of 50 PSD estimates.

The Colored Noise block generates two-channels of pink noise with 1024 samples. The Spectrum Analyzer computes modified periodograms using a Hamming window and 50% overlap. The running average of the PSD uses 50 spectral averages.



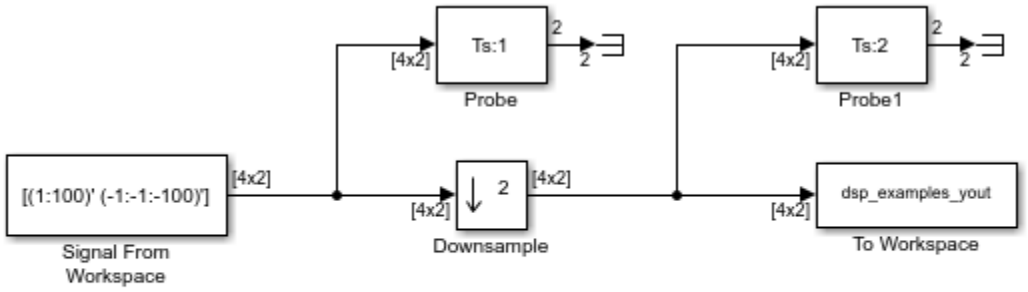
See Also

Blocks
Colored Noise | Spectrum Analyzer

Downsample a Signal

Downsample a signal by a factor of 2 using the Downsample block.

Open the System and Run the Model



Copyright 2008-2019 The MathWorks, Inc.

Note: This model creates a workspace variable called "dsp_examples_yout".

The Signal From Workspace block generates a two-channel signal with a frame size of 4. The two Probe blocks are specified to show the sample time of the signal before and after the downsampling operation.

Run the model. The sample time of the signal before the downsampling operation is half the sample time of the signal after the operation. You can see this from the `Ts` parameter visible on the two Probe blocks.

One-Frame Latency

The **Input processing** parameter in the Downsample block is set to `Columns as channels (frame based)` and the input frame size (number of rows in the input), *Mi*, is greater than 1. Hence, the latency of the signal is one frame. The **Initial conditions** parameter is set to `[11 -11;12 -12;13 -13;14 -14]`. In all cases of one-frame latency, the *Mi* rows of the initial condition matrix appear in sequence as the first four output rows. Input sample *D*+1 (i.e, row *D*+1 of the input matrix) appears in the output as sample *Mi*+1, followed by input sample *D*+1+K, input sample *D*+1+2K, and so on.

- *Mi* - Number of input rows. In this example, *Mi* equals 4.
- *D* - Sample offset parameter. In this example, *D* equals 1.
- *K* - Downsample factor. In this example, *K* equals 2.

The Initial conditions value can be an *Mi*-by- *N* matrix containing one value for each channel, or a scalar to be repeated across all elements of the *Mi*-by- *N* matrix.

Here is the downsampled output signal written to the `dsp_examples_yout` variable in the base workspace.

```
dsp_examples_yout =

    11   -11
```

```
 12   -12
 13   -13
 14   -14
  2    -2
  4    -4
  6    -6
  8    -8
 10   -10
 12   -12
 14   -14
 16   -16
 18   -18
 20   -20
 22   -22
 24   -24
 26   -26
 28   -28
 30   -30
 32   -32
 34   -34
 36   -36
 38   -38
 40   -40
 42   -42
 44   -44
 46   -46
 48   -48
 50   -50
 52   -52
 54   -54
 56   -56
 58   -58
 60   -60
 62   -62
 64   -64
 66   -66
 68   -68
 70   -70
 72   -72
 74   -74
 76   -76
 78   -78
 80   -80
 82   -82
 84   -84
 86   -86
 88   -88
 90   -90
 92   -92
 94   -94
 96   -96
 98   -98
100  -100
  0     0
  0     0
  0     0
  0     0
  0     0
```

```
     0     0
     0     0
     0     0
     0     0
     0     0
```

See Also

Blocks
Downsample | Probe | Signal From Workspace | To Workspace

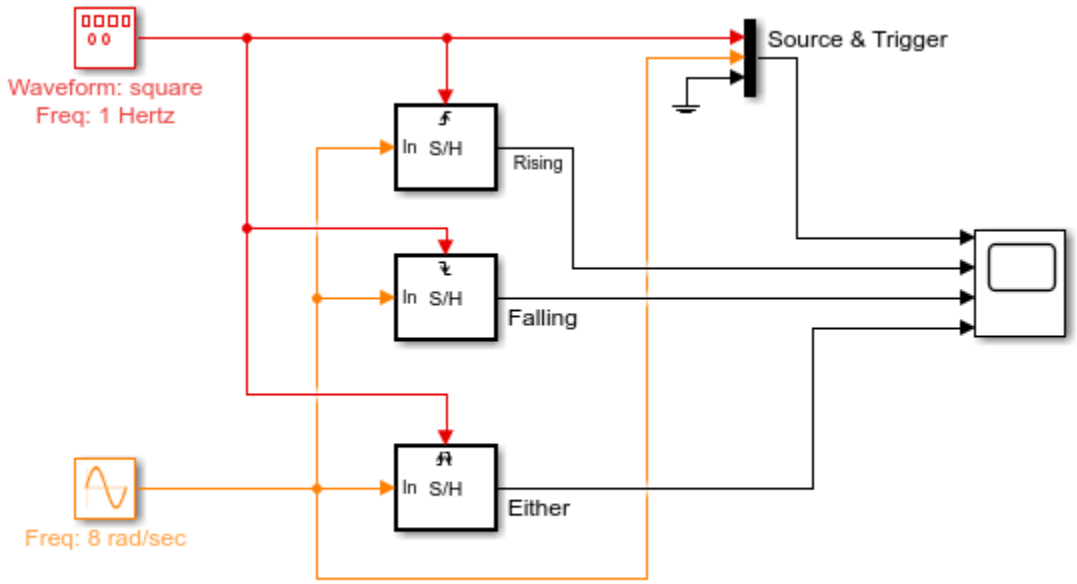
Sample and Hold a Signal

Sample an input signal when a trigger event occurs and hold the value until the next trigger event using the Sample and Hold block. The trigger event can be one of the following:

- Rising edge - Negative value or zero to a positive value.
- Falling edge - Positive value or zero to a negative value.
- Either edge - Negative value or zero to a positive value and positive value or zero to a negative value.

Open the Model

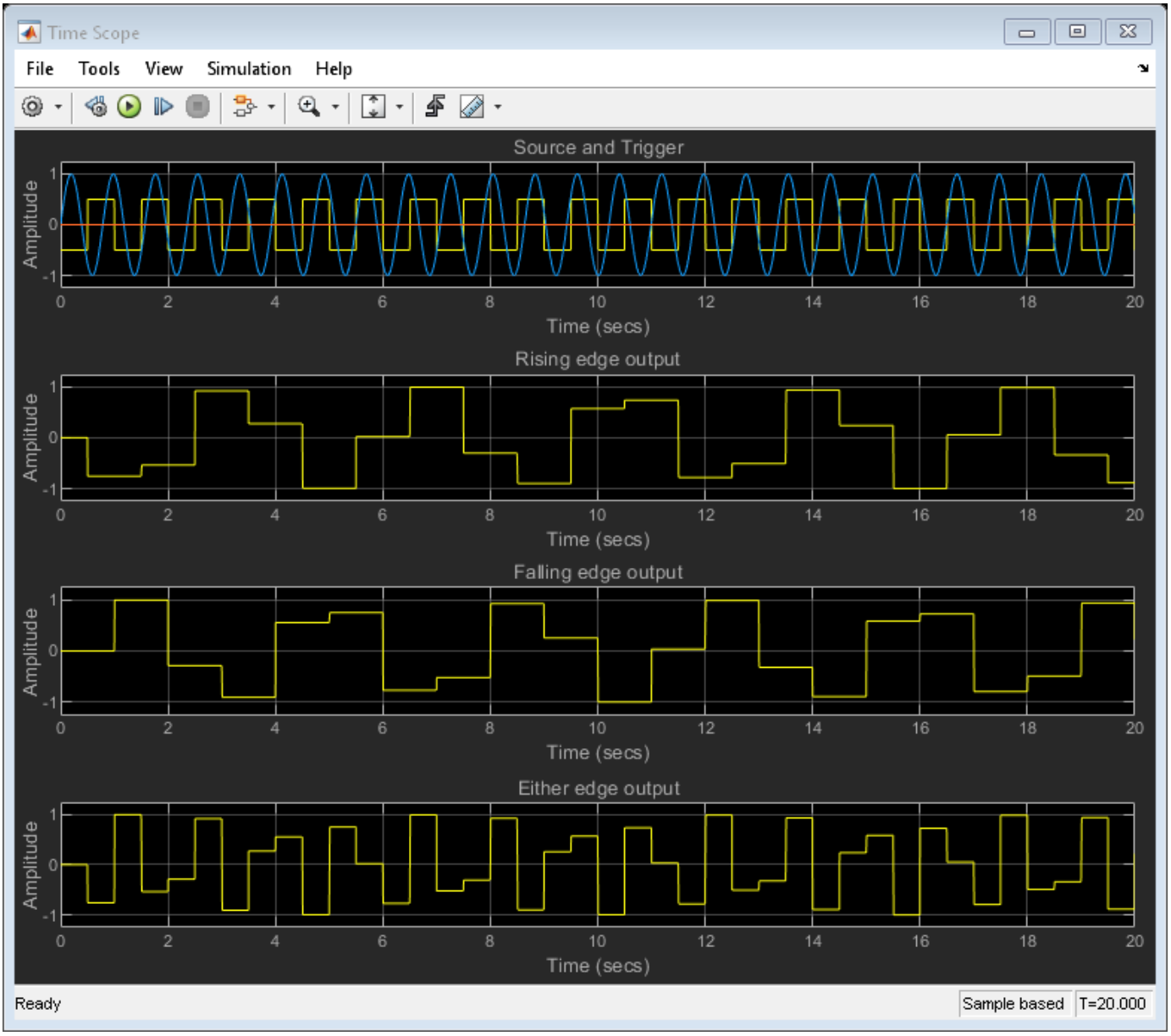
The model contains three **Sample and Hold** blocks which accept the three type of trigger events. The input signal is a continuous-time sine wave with an amplitude of 1 and a frequency of 8 rad/sec. The trigger signal is a square wave with an amplitude of 0.5 and a frequency of 1 Hz.



Copyright 2019 The MathWorks, Inc.

Run the Model

When you run the model, the Time Scope block shows the source and trigger signals on the first plot. You can see the three sample and hold outputs on the three remaining plots.



The **Initial condition** parameter in all the three **Sample and Hold** blocks is set to 0. Hence, the three output plots start at 0 value. The first trigger is a rising edge that happens at 0.5 seconds. The first and third outputs respond to this trigger by dropping to the value of the input sine wave at that point in time. This input value is held until the next respective trigger event occurs. The second output plot responds to the first falling edge that occurs at 1 second. At 1 second, the second output plot jumps to 1, which is the value of the sine wave at that point in time. This value is held until 2

seconds when the next falling edge event occurs. The output of the second plot then drops down to the value of the sine wave at that point in time. This value is held until the next trigger event occurs.

See Also

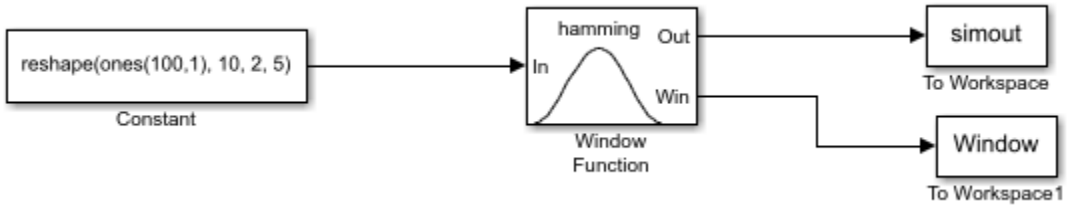
Blocks
Ground | Mux | Sample and Hold | Signal Generator | Sine Wave | Time Scope

Generate and Apply Hamming Window

The following model uses the Window Function block to generate and apply a Hamming window to a 3-dimensional input array.

In this example, set the **Operation** parameter of the Window Function block to `Generate and apply window`. The block provides two outputs: the window vector, *Window* at the **Win** port, and the result of the multiplication, *simout* at the **Out** port.

Open the model `ex_windowfunction_ref`.



Run the model.

The length of the first dimension of the input array is 10, so the **Window Function** block generates and outputs a Hamming window vector of length 10. To see the window vector generated by the **Window Function** block, type `Window` at the MATLAB® command line.

```
Window =

    0.0800
    0.1876
    0.4601
    0.7700
    0.9723
    0.9723
    0.7700
    0.4601
    0.1876
    0.0800

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 15
```

To see the result of the multiplcation, type `simout` at the MATLAB command line.

```
simout =

(:,:,1) =
    0.0791    0.0791
    0.1875    0.1875
    0.4600    0.4600
    0.7695    0.7695
    0.9717    0.9717
```

```
    0.9717    0.9717
    0.7695    0.7695
    0.4600    0.4600
    0.1875    0.1875
    0.0791    0.0791
(:,:,2) =
    0.0791    0.0791
    0.1875    0.1875
    0.4600    0.4600
    0.7695    0.7695
    0.9717    0.9717
    0.9717    0.9717
    0.7695    0.7695
    0.4600    0.4600
    0.1875    0.1875
    0.0791    0.0791
(:,:,3) =
    0.0791    0.0791
    0.1875    0.1875
    0.4600    0.4600
    0.7695    0.7695
    0.9717    0.9717
    0.9717    0.9717
    0.7695    0.7695
    0.4600    0.4600
    0.1875    0.1875
    0.0791    0.0791
(:,:,4) =
    0.0791    0.0791
    0.1875    0.1875
    0.4600    0.4600
    0.7695    0.7695
    0.9717    0.9717
    0.9717    0.9717
    0.7695    0.7695
    0.4600    0.4600
    0.1875    0.1875
    0.0791    0.0791
(:,:,5) =
    0.0791    0.0791
    0.1875    0.1875
    0.4600    0.4600
    0.7695    0.7695
    0.9717    0.9717
    0.9717    0.9717
    0.7695    0.7695
    0.4600    0.4600
    0.1875    0.1875
    0.0791    0.0791

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
```

```
    WordLength: 16
FractionLength: 10
```

See Also

Blocks
Window Function

Convert Sample Rate of Speech Signal

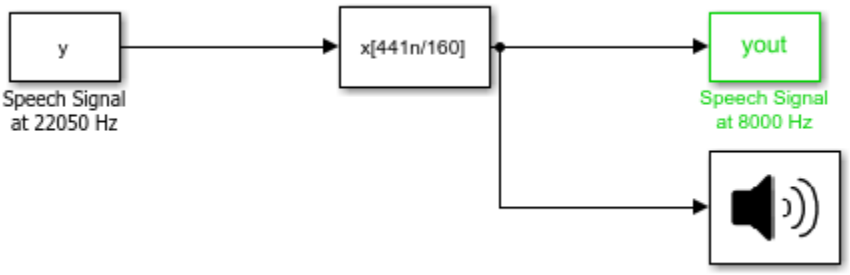
This example shows how to use the **FIR Rate Conversion** block to convert the sample rate of a speech signal. It compares the output/signal behavior in two scenarios -single rate processing and multirate processing. Convert the sample rate of a speech signal from 22.5 kHz to 8 kHz.

Enforce single-rate processing

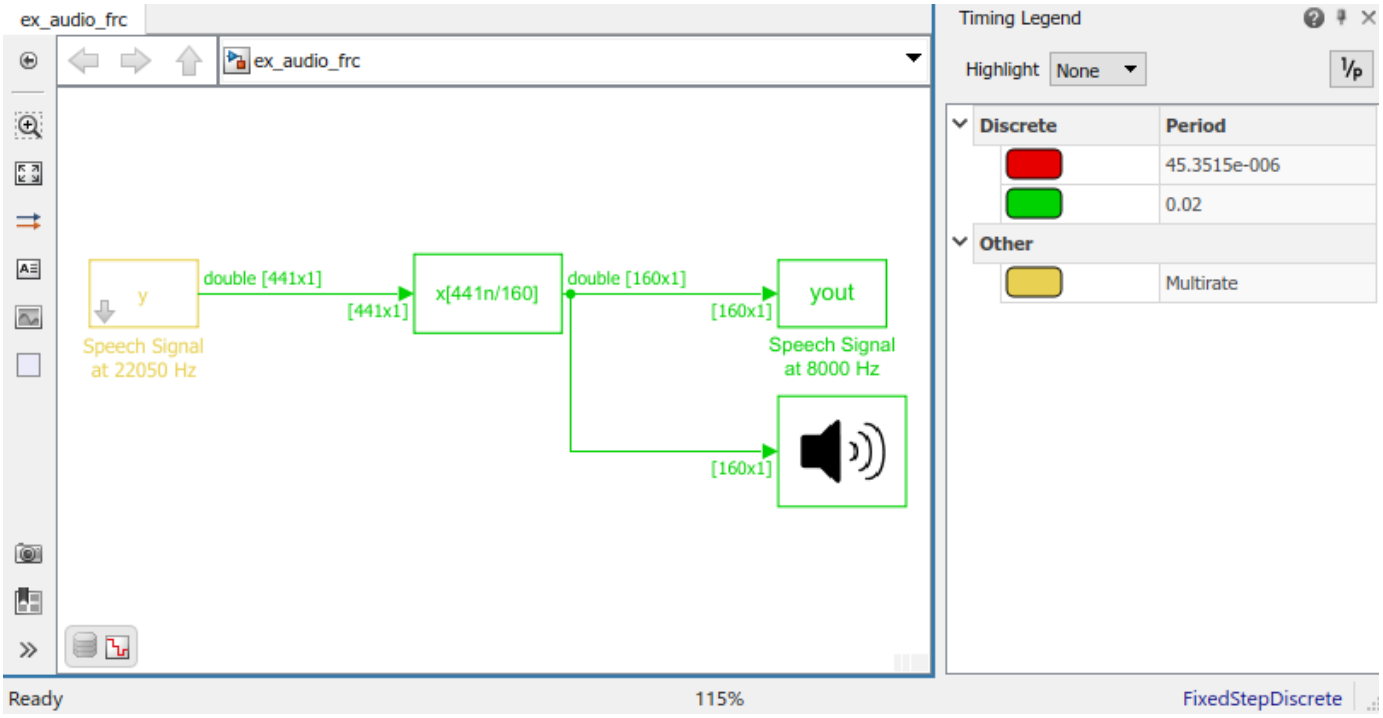
The output signal rate and the input signal rate in Simulink® are the same.

The interpolation factor *L* is set to 160 and the decimation factor *K* is set to 441. The output frame size is L/K times the input frame size.

Open and run the model to listen to the output. High frequency content has been removed from the signal, although the speech sounds basically the same.



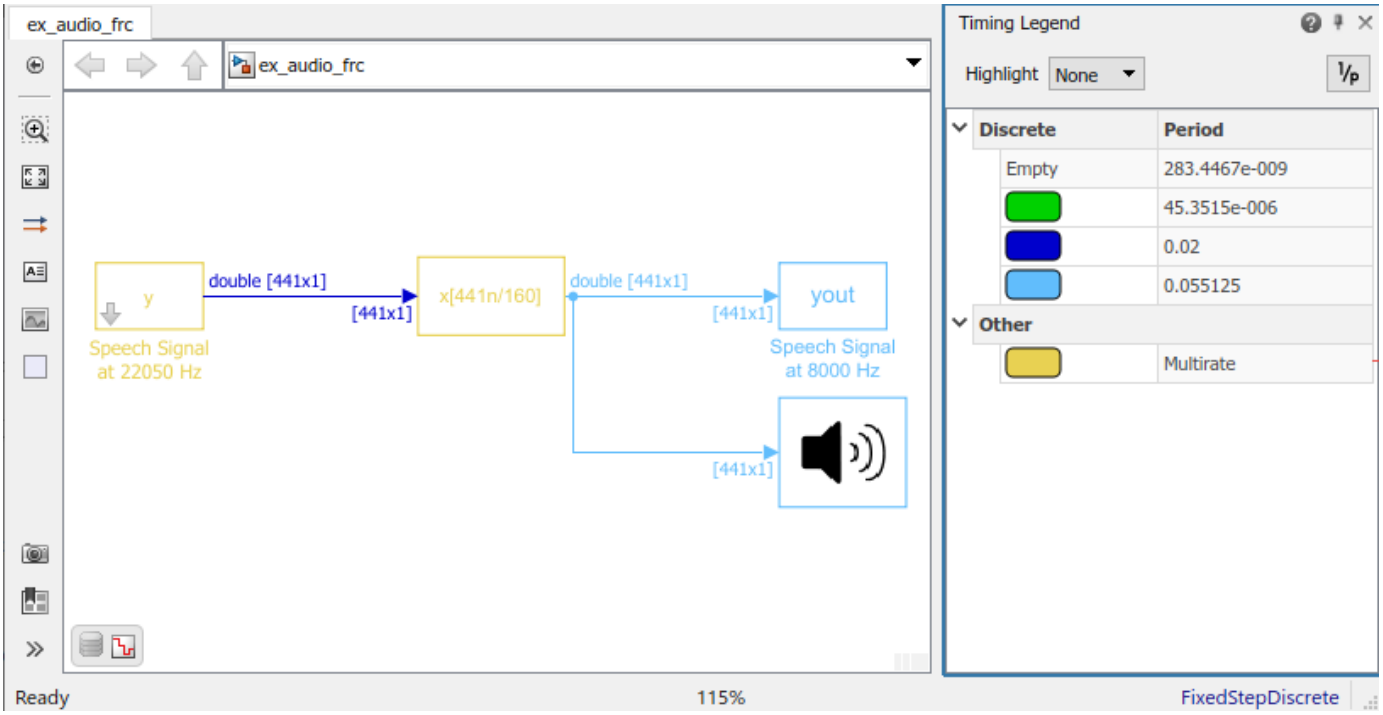
The output and the input signal rate are the same in Simulink. This is shown by the green color-coded signal lines at the input and output of the block.



Allow multirate processing

Change the **Rate options** parameter to `Allow multirate processing`.

Run the model. You can see that the output signal and the blocks connected to the output signal operate at a rate that is L/K times the rate at which the input signal operates in Simulink.



See Also

Blocks

Audio Device Writer | FIR Rate Conversion | Signal From Workspace | Signal To Workspace

Unwrap Signal

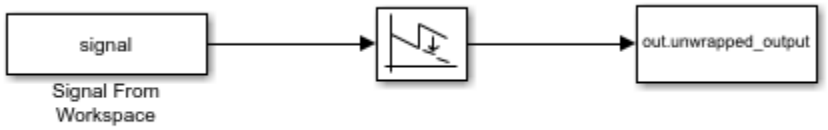
This example shows how to use the Unwrap block to unwrap a 3-by-2-by-3 array that has discontinuity.

Each 3-by-2 frame of the signal has a discontinuity. In the first frame, there is a discontinuity between the second and the third element greater than the tolerance of π set in the block parameters. In the second frame, the discontinuity between the second and third element is less than π, and in the third frame, the discontinuity is equal to π. Since we expect the block to wrap signals with discontinuities greater than the tolerance, unwrapping the signal does not affect the second or third frame.

Open the Simulink model.

```
model = "ex_unwrap";
open_system(model);
frame1 = [0 0; 2*(pi)/3 0; -2*(pi)/3 0];
frame2 = [2*pi 0; 8*pi/3 0; 9.5*pi/3 0];
frame3 = [4*pi 0; 13*pi/3 0; 16*pi/3 0];
signal = cat(3,frame1,frame2,frame3)
```

```
signal(:,:,1) =

         0         0
    2.0944         0
   -2.0944         0


signal(:,:,2) =

    6.2832         0
    8.3776         0
    9.9484         0


signal(:,:,3) =

   12.5664         0
   13.6136         0
   16.7552         0
```



Copyright 2019 The MathWorks, Inc.

The `Tolerance` parameter of the Unwrap block is set to π. The block therefore unwraps discontinuities which are larger than π.

Run the model.

```
output = sim(model);
```

See that the output, `output.unwrapped_signal` does not have the first discontinuity, but the other two remain.

```
output.unwrapped_output
```

```
ans =

         0         0
    2.0944         0
    4.1888         0
    6.2832         0
    8.3776         0
    9.9484         0
   12.5664         0
   13.6136         0
   16.7552         0
```

See Also

Blocks
From Workspace | To Workspace | Unwrap

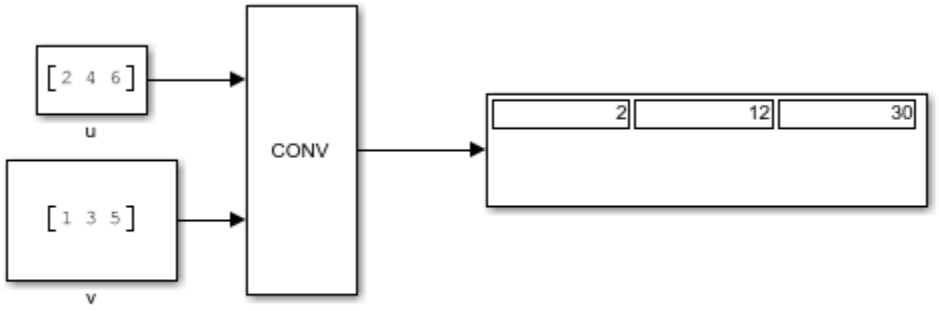
Convolution of Two Inputs

Using the Convolution block, convolve two input signals.

Open the `ex_convolution1.slx` model, which convolves two vectors.

For this model, the **Convolution** block returns a 1-by-3 vector. This is because both *u* and *v* are of the same shape and size.

```
model1 = "ex_convolution1";
open_system(model1);
sim(model1);
```

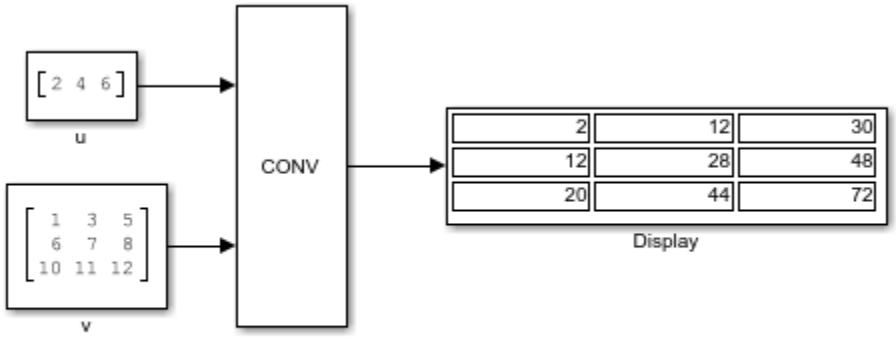


Copyright 2019 The MathWorks, Inc.

Open the `ex_convolution2.slx` model, which convolves a vector with a matrix.

In this model, the **Convolution** block returns a 3-by-3 matrix. The two inputs can be convolved because they share the same last dimension, which becomes the size of the output's last dimension. The number of rows of the output is equal to the sum of the first dimension of the two inputs minus one. In this model that results in three rows, so the output is a 3-by-3 matrix.

```
model2 = "ex_convolution2";
open_system(model2);
sim(model2);
```



Copyright 2019 The MathWorks, Inc.

When creating models that convolve *N*-D arrays, keep in mind that except for the first dimension, all other dimensions must be the same.

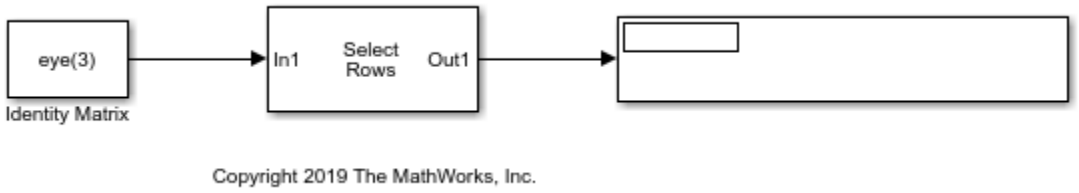
See Also

Blocks
Constant | Convolution | Display

Select Rows or Columns from Matrices

This example shows how to use the Variable Selector block.

Open the Simulink model.



Copyright 2019 The MathWorks, Inc.

The **Variable Selector** block returns a matrix with only the selected rows or columns of the input matrix. In this example, the **Select** parameter of the block is set to `Rows` and the **Elements** parameter is set to [1 3]. All the other parameters are set to their default values. Because the input is an identity matrix, the output is:

```
ans =

     1     0     0
     0     0     1
```

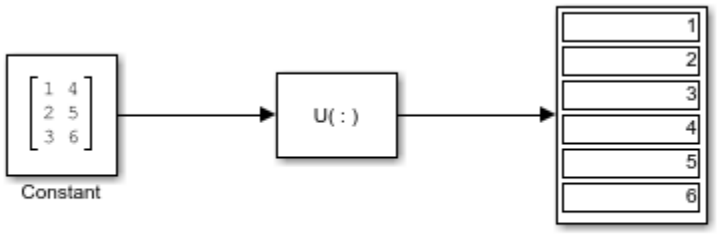
Run the model to verify this output.

See Also

Blocks
Display | Identity Matrix | Variable Selector

Convert 2-D Matrix to 1-D Array

In this model, the Convert 2-D to 1-D block takes the 2-D input matrix and outputs an *N*-element vector, where *N* is the total number of elements in the input matrix. The block performs a column-wise conversion. The first set of elements of the output array is composed of the first column of the input matrix, the second set of elements is composed of the second column, and so on.



Copyright 2019 The MathWorks, Inc.

See Also

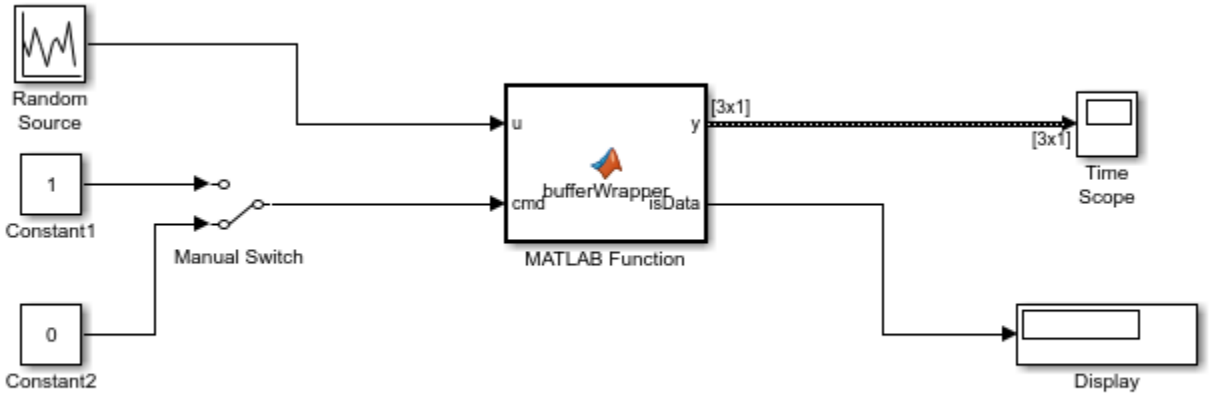
Blocks
Constant | Convert 2-D to 1-D | Display

Simulink Block Examples in DSP System Toolbox

- “Why Does Reading Data from the dsp.AsyncBuffer Object Give a Dimension Mismatch Error in the MATLAB Function Block?” on page 12-2
- “Why Does the dsp.AsyncBuffer Object Error When You Call read Before write?” on page 12-7
- “Buffering Input with Overlap” on page 12-9

Why Does Reading Data from the dsp.AsyncBuffer Object Give a Dimension Mismatch Error in the MATLAB Function Block?

If you are reading data from an asynchronous buffer inside a MATLAB Function block, the block throws a dimension mismatch error if the output of the `read` method is not specified to be a variable-size signal.



Copyright 2016 The MathWorks, Inc.

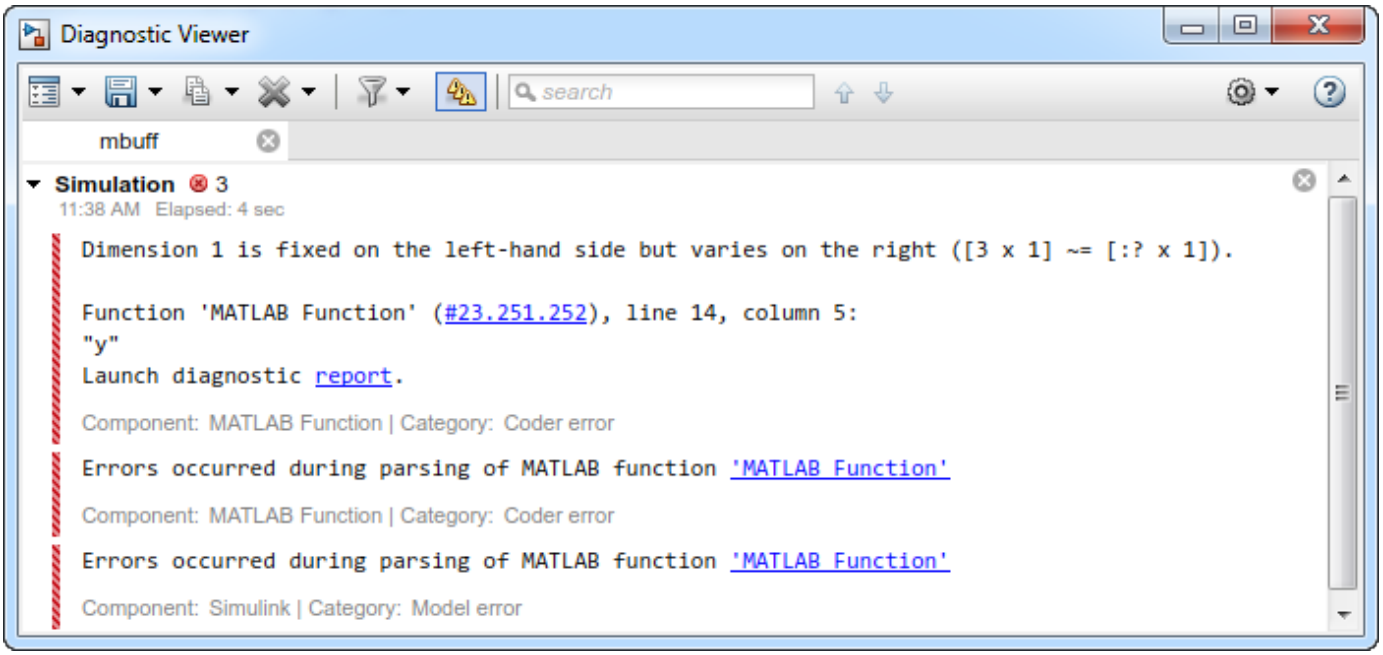
Here is the `bufferWrapper` function that contains the algorithm inside the MATLAB Function block. When input on the `cmd` port is `1`, the `dsp.AsyncBuffer` object writes the data input, `u`, to the buffer. When input on the `cmd` port is `0`, the object reads data from the buffer.

```
function [y,isData] = bufferWrapper(u,cmd)

persistent asyncBuff
if isempty(asyncBuff)
   asyncBuff = dsp.AsyncBuffer;
   setup(asyncBuff,u);
end

if cmd % write
    write(asyncBuff,u);
    y = zeros(3,1);
    isData = false;
else % read
    y = read(asyncBuff,3);
    isData = true;
end
```

You must initialize the buffer by calling either `write` or `setup` before the first call to `read`.

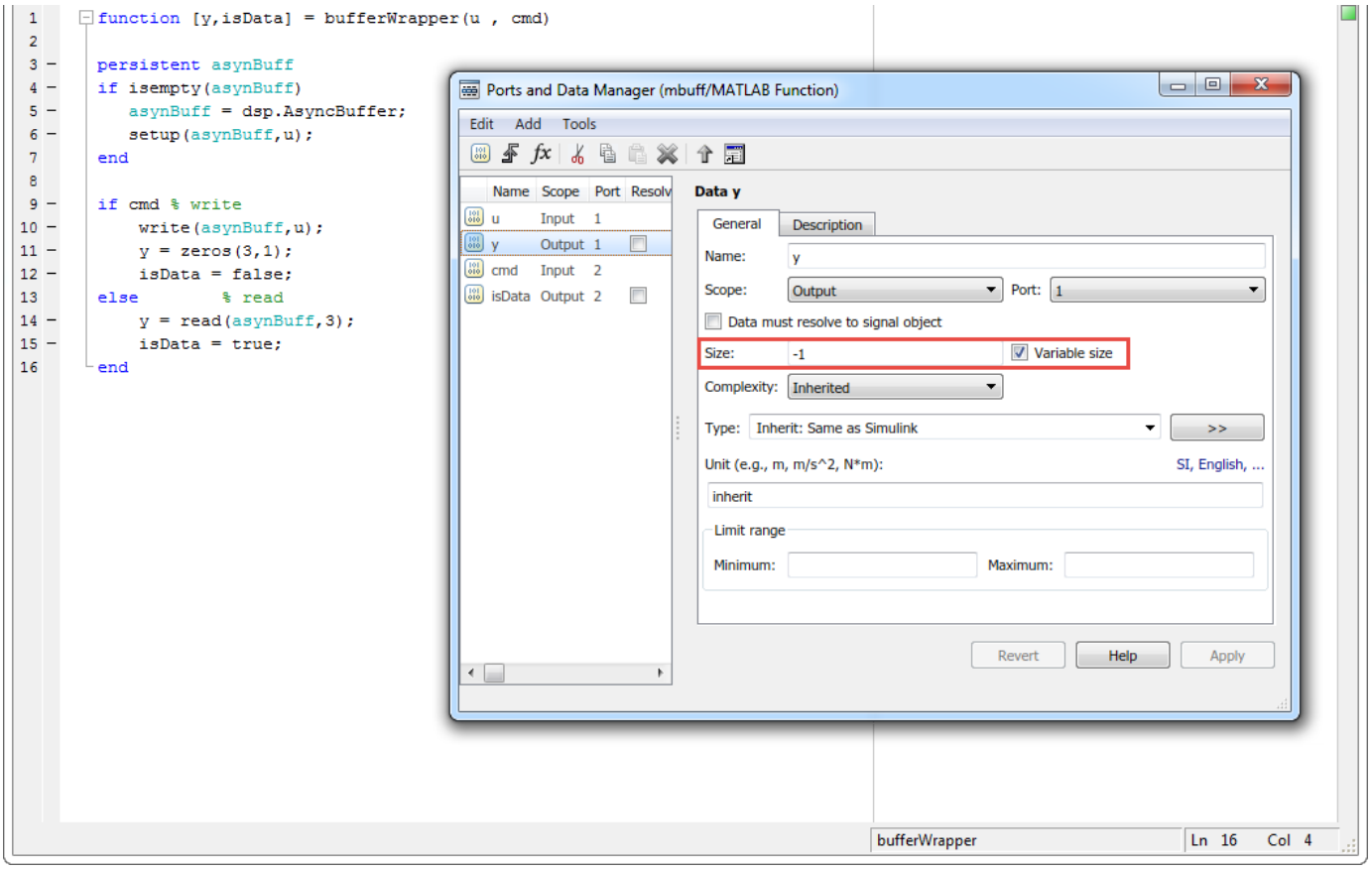
During the write operation, the first output, `y`, is `zeros(3,1)` and the second output, `isData`, is 0. During the read operation, `y` is the data in the buffer and `isData` is 1.

Run the model and the following error occurs.

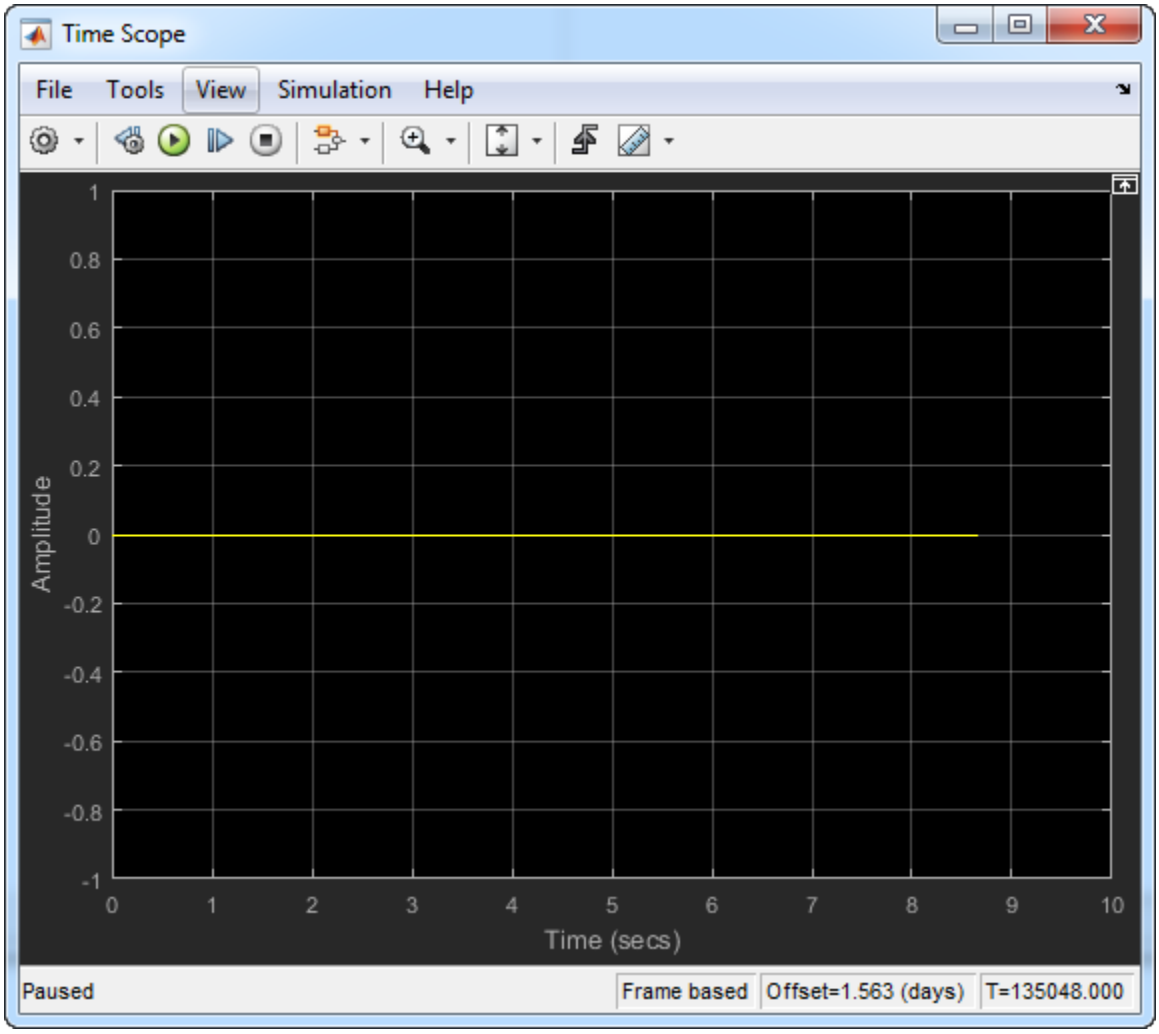


The output of `read(asyncBuff,3)` on line 14 is variable sized. The output is variable sized because the size of the signal output by the `read` function depends on the input arguments to `read`. To resolve this error, specify `y` as a variable-size signal.

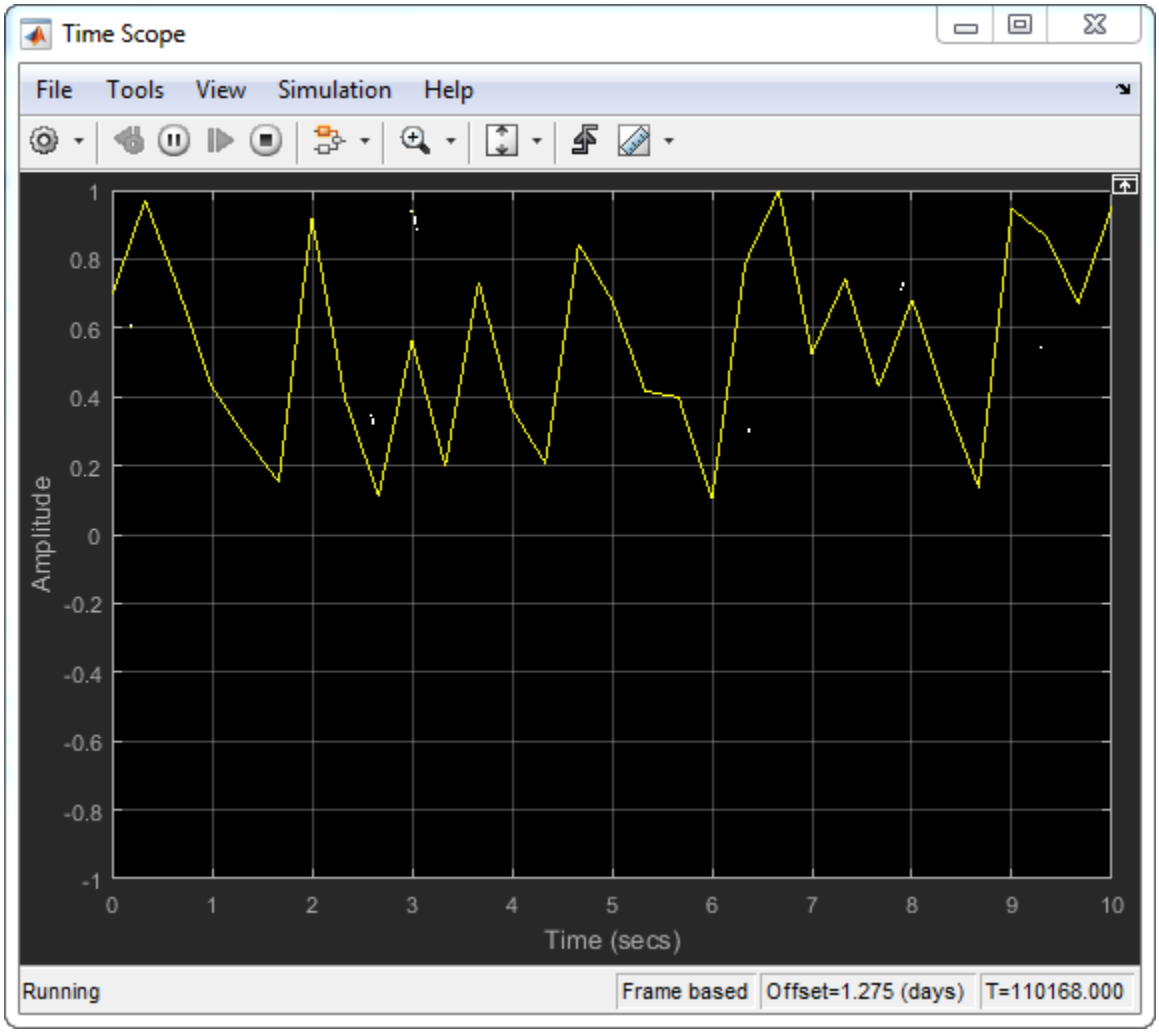
1. In the MATLAB Function block Editor, click **Edit Data** to open the Ports and Data Manager.
2. For the output `y`, select the **Variable size** check box.
3. Click **Apply**.



Run the model and view the output y in the Time Scope.



With `cmd = 0`, no data is written into the buffer. Therefore, the output is 0. To write the input data `u` to the buffer, set `cmd = 1`. After you write some data, if you change `cmd` back to 0, the Time Scope output changes to the following.



See Also

Objects
`dsp.AsyncBuffer`

Related Examples

- “Why Does the dsp.AsyncBuffer Object Error When You Call read Before write?” on page 12-7
- “High Resolution Spectral Analysis”

Why Does the dsp.AsyncBuffer Object Error When You Call read Before write?

In the `dsp.AsyncBuffer` System object, you must initialize the buffer before the first call to the `read` method. To initialize the buffer, call either the `write` or `setup` method.

Consider the `bufferWrapper` function, which writes and reads data from an asynchronous buffer. When the input `cmd` is set to `true`, the object writes data to the buffer. When `cmd` is `false`, the object reads data from the buffer.

```
function [y,isData] = bufferWrapper(u,cmd)

persistent asyncBuff
if isempty(asyncBuff)
   asyncBuff  = dsp.AsyncBuffer;
end

if cmd % write
   write(asyncBuff,u);
   y = zeros(128,1);
   isData = false;
else % read
    isData = true;
    y = read(asyncBuff,128,64);
end
```

Call the buffer with `cmd` set to `false`.

```
bufferWrapper(1,false);
```

The function errors with the following message:

```
Buffer not initialized. You must call write before read.
```

When you generate code from this function, the object throws an error that the buffer `'Cache'` is undefined.

```
codegen bufferWrapper -args {1,false}
```

```
??? Property 'Cache' is undefined on some execution paths but is used inside the called function.
```

Note that the `codegen` function is not supported in MATLAB Online.

Both these error messages indicate that the buffer is not initialized at the first call to the `read` method in one of the execution paths.

To resolve these errors, call `write` or `setup` before the first call to `read`. If you are calling `setup`, call it only once at the beginning, during the buffer construction.

In this function, `setup` is called before `read`.

```
function [y,isData] = bufferWrapper_setup(u,cmd)

persistent asyncBuff
if isempty(asyncBuff)
   asyncBuff = dsp.AsyncBuffer;
   setup(asyncBuff,u);
```

```
end

if cmd % write
   write(asyncBuff,u);
   y = zeros(128,1);
   isData = false;
else % read
    isData = true;
    y = read(asyncBuff,128,64);
end
```

You can now read the buffer without any errors.

```
bufferWrapper_setup(1,false);
```

Generating code from this function now successfully generates the MEX file, because the cache is defined on all execution paths.

```
codegen bufferWrapper_setup -args {1,false}
```

See Also

Objects
`dsp.AsyncBuffer`

Related Examples

- “High Resolution Spectral Analysis”

More About

- “Why Does Reading Data from the dsp.AsyncBuffer Object Give a Dimension Mismatch Error in the MATLAB Function Block?” on page 12-2

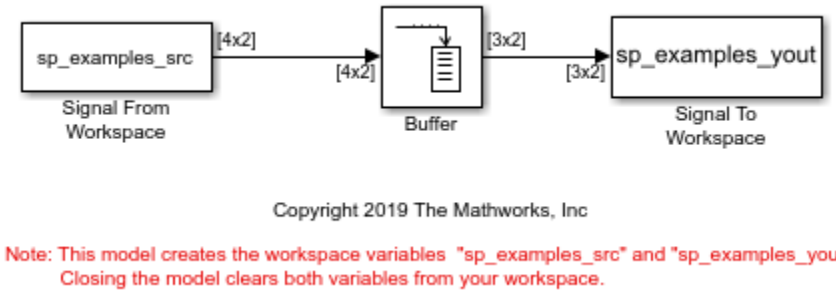
Buffering Input with Overlap

Buffering Two-Channel Input with Overlap

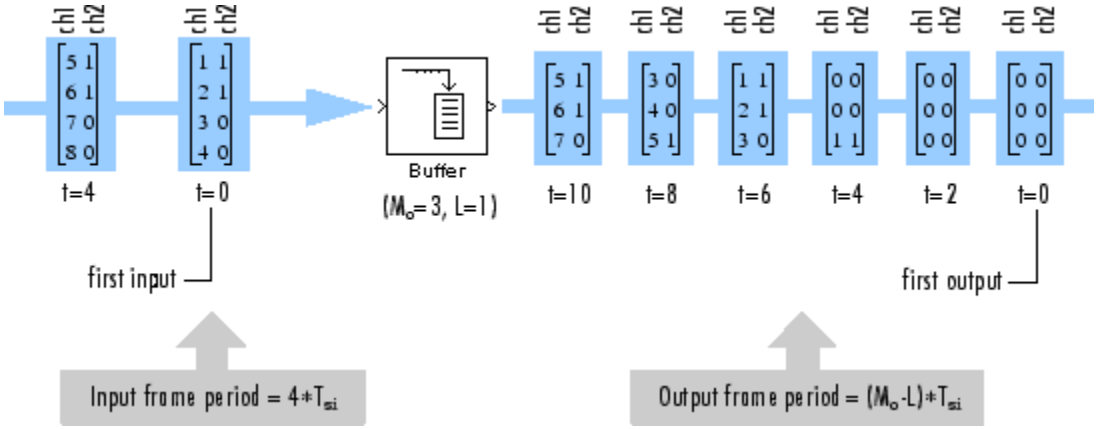
In the `ex_buffer_tut4` model, the Buffer block uses a one-sample overlap and rebuffers a signal with the frame size of 4 into a signal with the frame size of 3.

Open and run the model.

```
two_channel_model = "ex_buffer_tut4";
open_system(two_channel_model);
sim(two_channel_model);
```



The following diagram illustrates the inputs and outputs of the Buffer block.



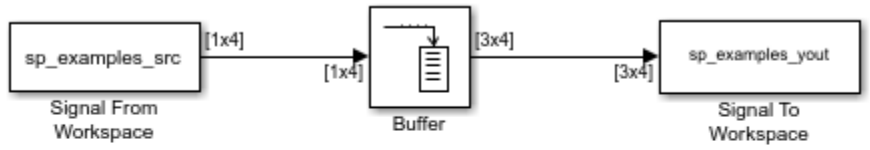
The output is delayed by eight samples. This latency occurs because of the parameter settings chosen in this model, and because the model is running in Simulink in the multitasking mode. The first eight output samples therefore adopt the value specified in the **Initial conditions** parameter, which in this case is zero. You can use the rebuffer_delay function to determine the latency of the Buffer block for any combination of frame size and overlap values.

Buffering Four-Channel Input with Overlap

The ex_buffer_tut3 model buffers a 1-by-4 input signal using an output buffer size of 3 and a buffer overlap of 1. The buffered output is a 3-by-4 signal.

Open and run the model.

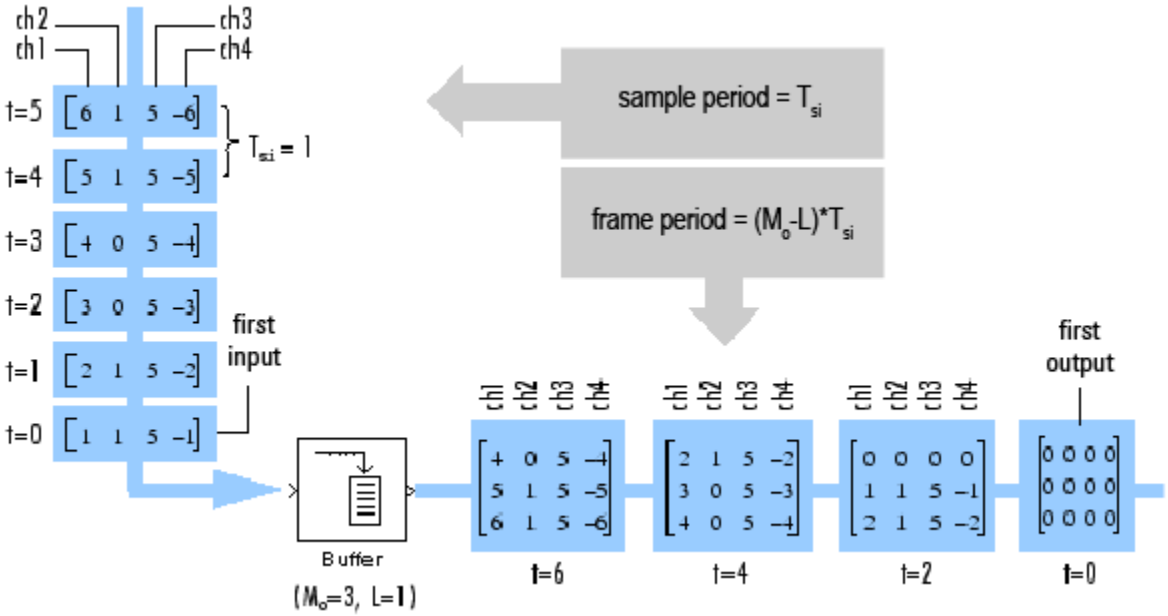
```
four_channel_model = "ex_buffer_tut3";
open_system(four_channel_model);
sim(four_channel_model);
```



Copyright 2019 The Mathworks, Inc

Note: This model creates the workspace variables "sp_examples_src" and "sp_examples_yout". Closing the model clears both variables from your workspace.

This diagram illustrates the inputs and outputs of the Buffer block.



The input vectors do not begin appearing at the output until the second row of the second matrix. This is due to latency in the Buffer block. The first output matrix (all zeros in this example) reflects the value of the **Initial conditions** parameter, while the first row of zeros in the second output is a result of the one-sample overlap between consecutive output frames.

You can use the rebuffer_delay function with a frame size of 1 to precisely compute the delay (in samples).

```
d = rebuffer_delay(1,3,1)
```

```
d =

     4
```

This number agrees with the four samples of delay (zeros) per channel shown in the previous figure.

See Also

Blocks
Buffer | Signal From Workspace | To Workspace

Functions
`rebuffer_delay`

Simulink Block Examples in DSP System Toolbox

- “Synthesize and Channelize Audio” on page 13-2
- “Filter input with Butterworth Filter in Simulink” on page 13-9

Synthesize and Channelize Audio

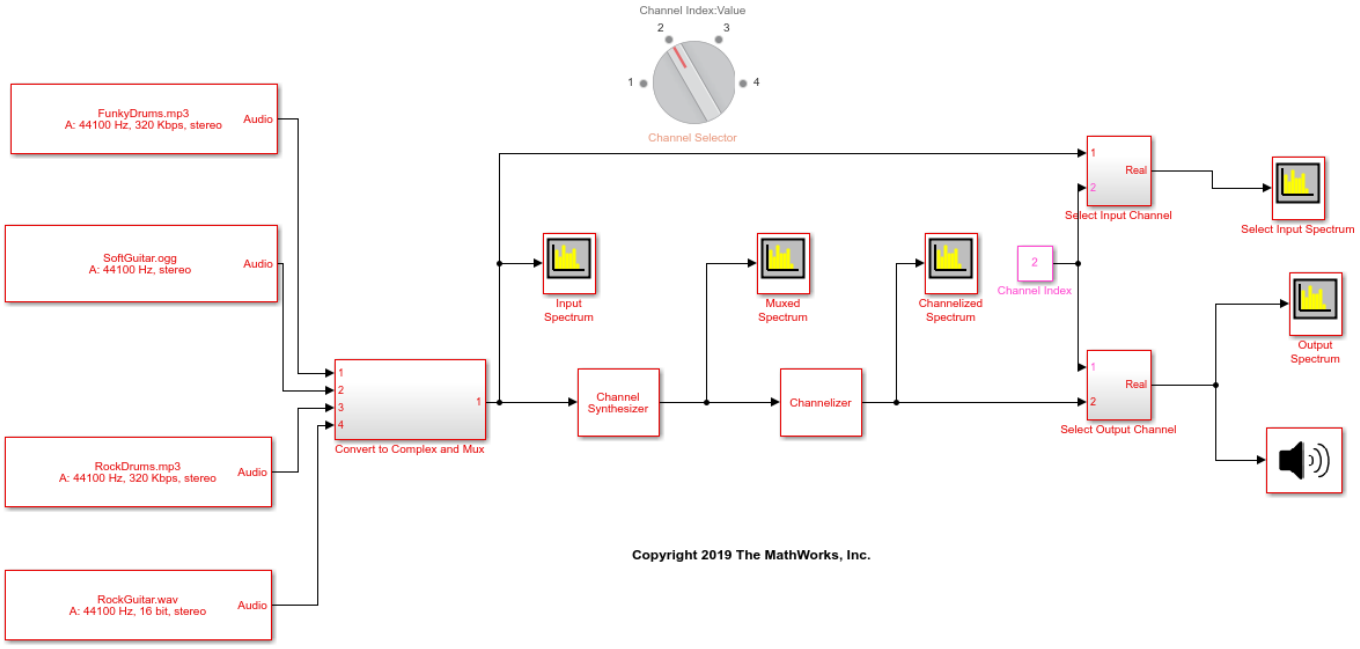
Synthesize a series of four stereo signals into a broadband signal by using the Channel Synthesizer block. At the receiving end of the model, split this broadband signal back into the individual narrowband signals by using the Channelizer block.

The inputs to the model are four stereo signals.

- `FunkyDrums.mp3`
- `SoftGuitar.ogg`
- `RockDrums.mp3`
- `RockGuitar.wav`

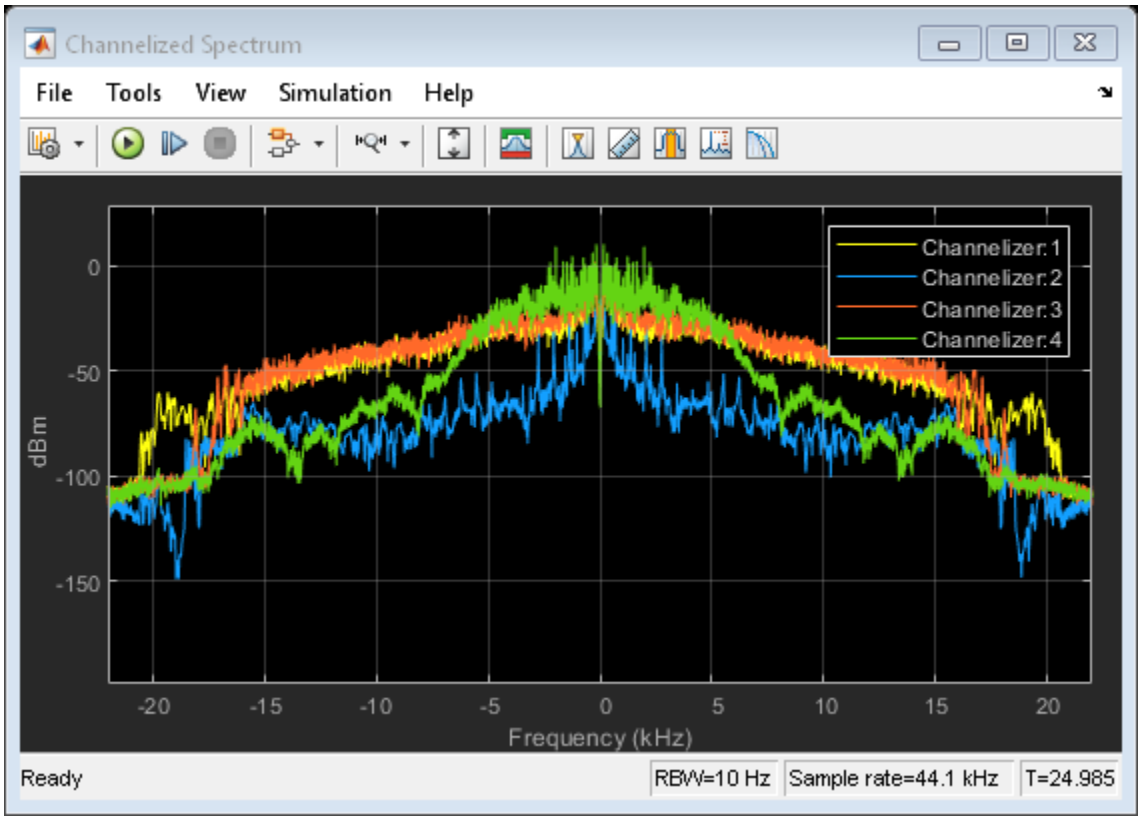
Each signal has a size of 1024-by-2 samples. The two channels represent the left channel and the right channel of the stereo signal. To store the stereo channels, each signal is converted into complex, and multiplexed by a Matrix Concatenate block to form a 1024-by-4 matrix. The Channel Synthesizer block synthesizes these four signals into a single broadband signal of size 4096-by-1. The Channelizer block that follows splits this broadband signal back into narrow subbands. The output of the Channelizer block is a 1024-by-4 matrix, with each channel representing a narrow band.

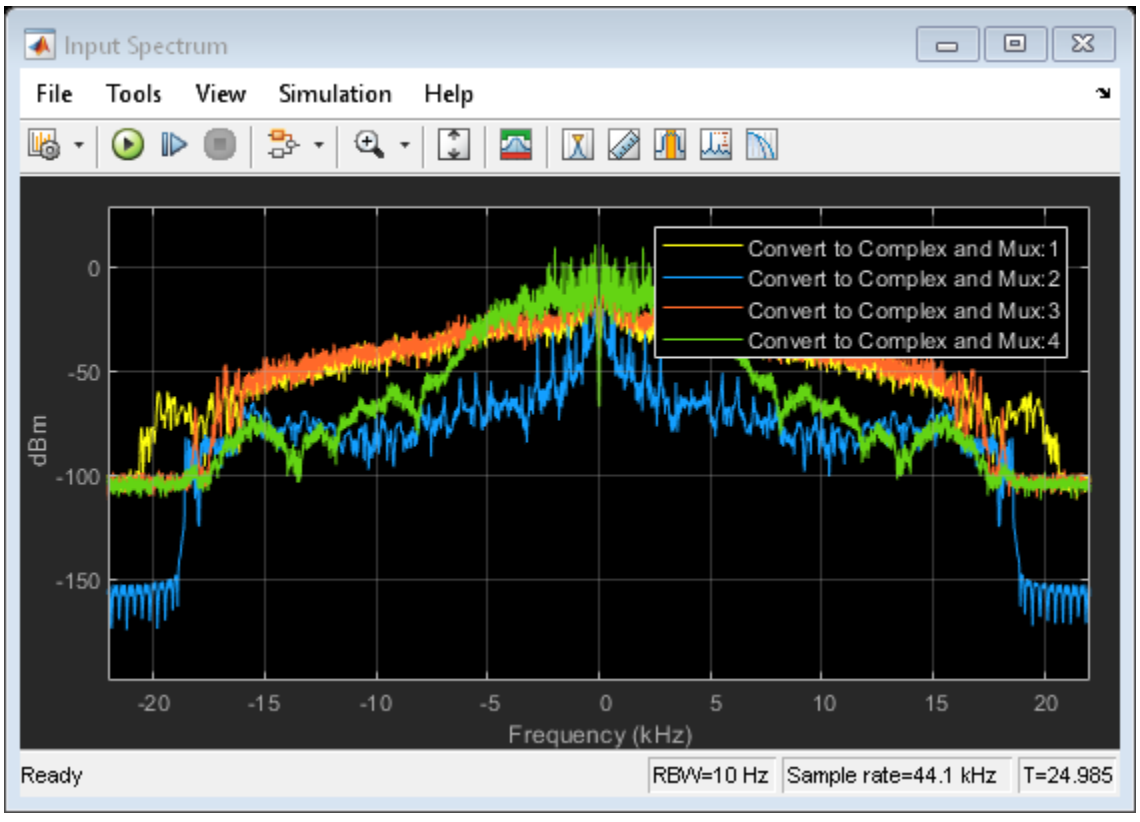
Open the model.

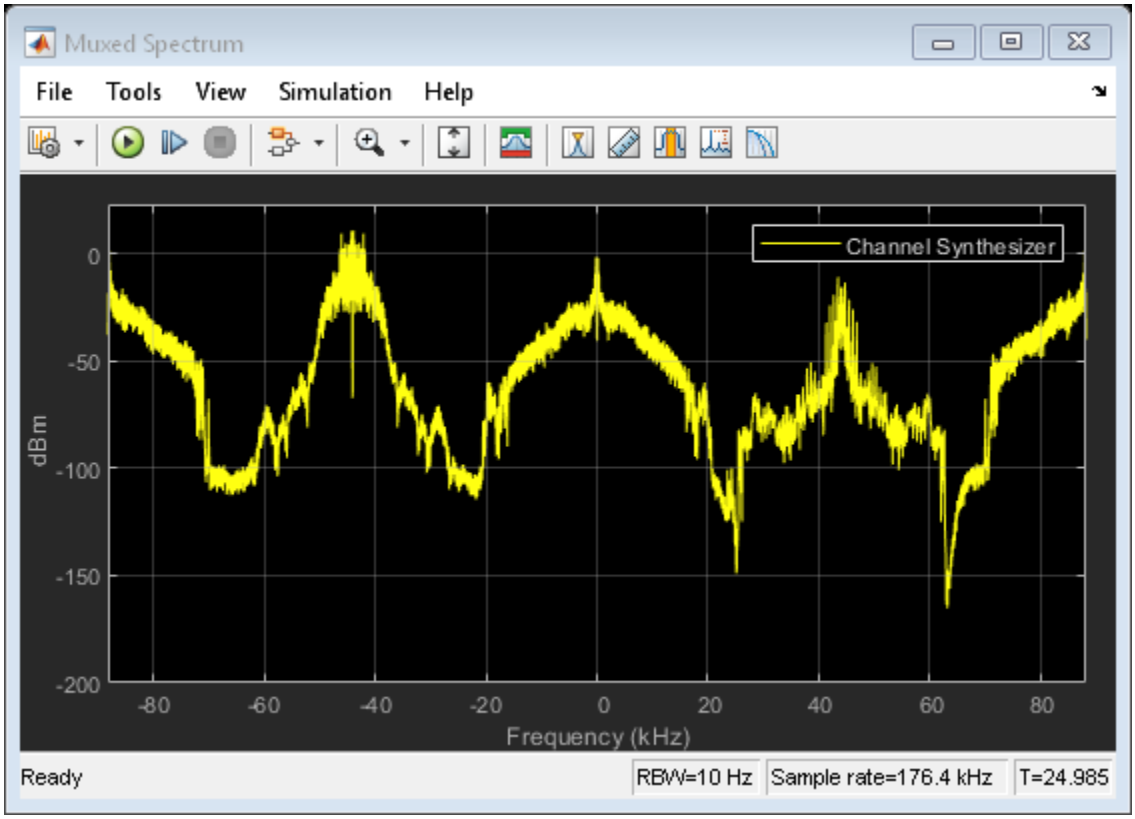


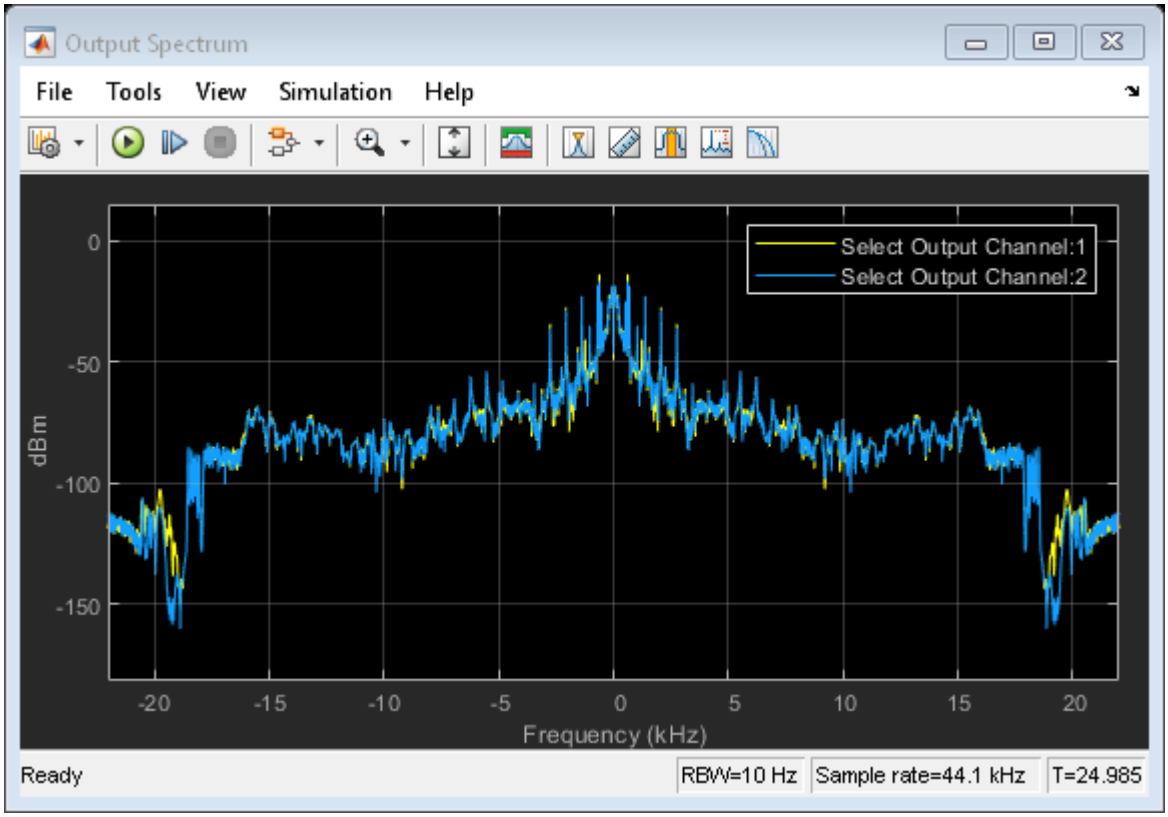
Select the audio signal you want to listen to and play this signal using the Audio Device Writer block.

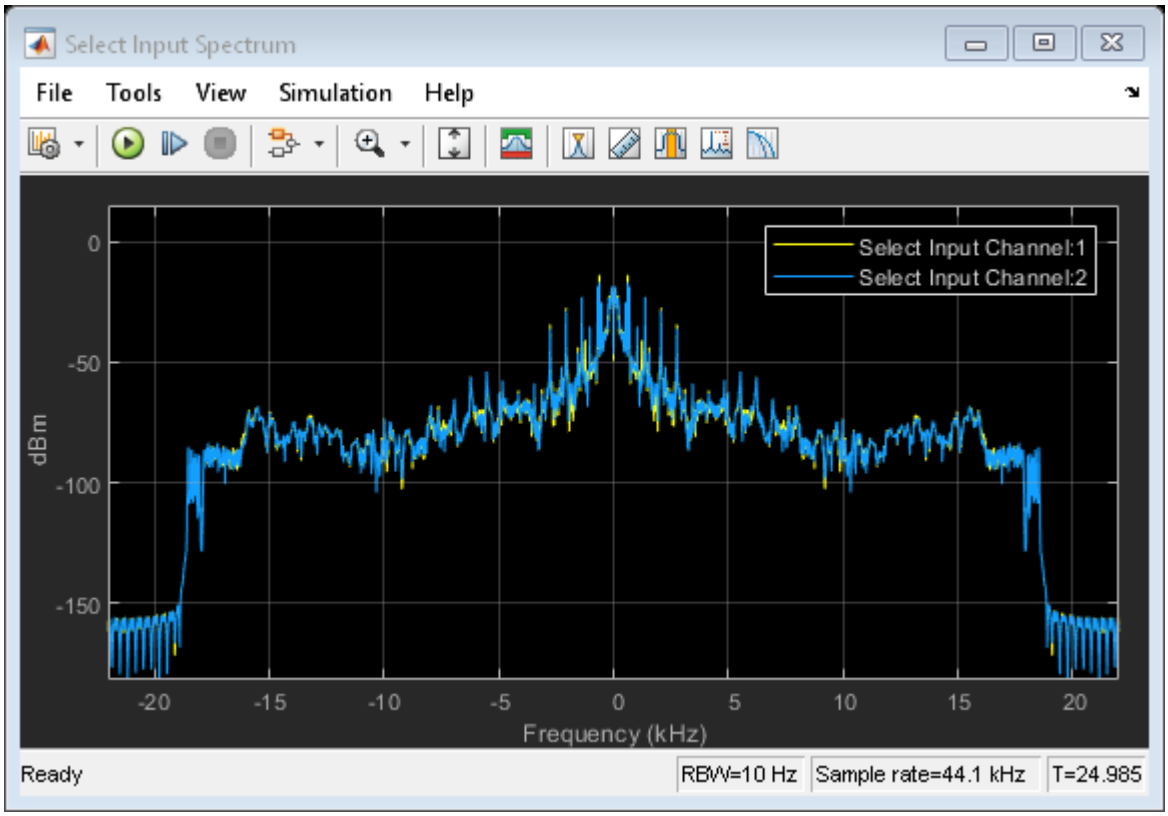
Run the model. View the spectra of the input, muxed, and output signals.


Channelized Spectrum
File Tools View Simulation Help
Channelizer: 1
Channelizer: 2
Channelizer: 3
Channelizer: 4
dBm
0
-50
-100
-150
-20 -15 -10 -5 0 5 10 15 20
Frequency (kHz)
Ready
RBW=10 Hz
Sample rate=44.1 kHz
T=24.985


Input Spectrum
File Tools View Simulation Help
Convert to Complex and Mux: 1
Convert to Complex and Mux: 2
Convert to Complex and Mux: 3
Convert to Complex and Mux: 4
dBm
0
-50
-100
-150
-20 -15 -10 -5 0 5 10 15 20
Frequency (kHz)
Ready
RBW=10 Hz
Sample rate=44.1 kHz
T=24.985

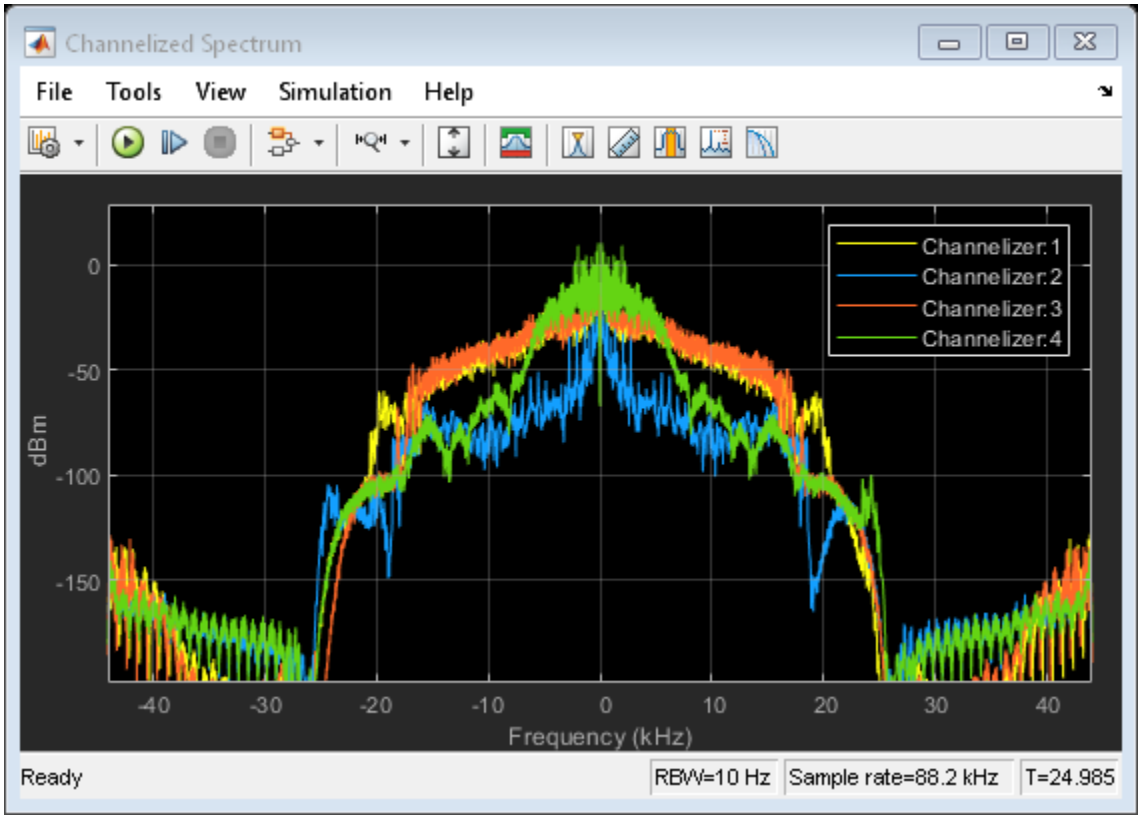

Muxed Spectrum
File Tools View Simulation Help
Channel Synthesizer
dBm
0
-50
-100
-150
-200
-80 -60 -40 -20 0 20 40 60 80
Frequency (kHz)
Ready
RBW=10 Hz
Sample rate=176.4 kHz
T=24.985

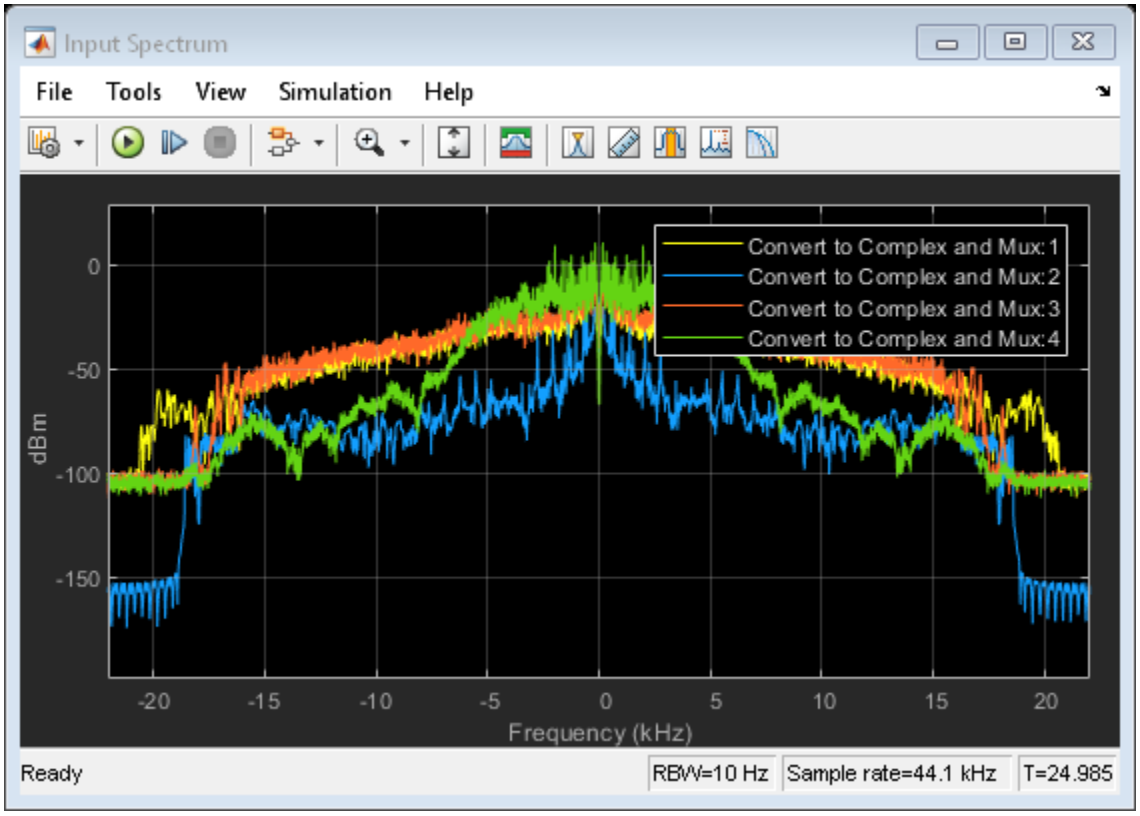

Output Spectrum
File Tools View Simulation Help
Select Output Channel:1
Select Output Channel:2
dBm
0
-50
-100
-150
-20 -15 -10 -5 0 5 10 15 20
Frequency (kHz)
Ready
RBW=10 Hz
Sample rate=44.1 kHz
T=24.985

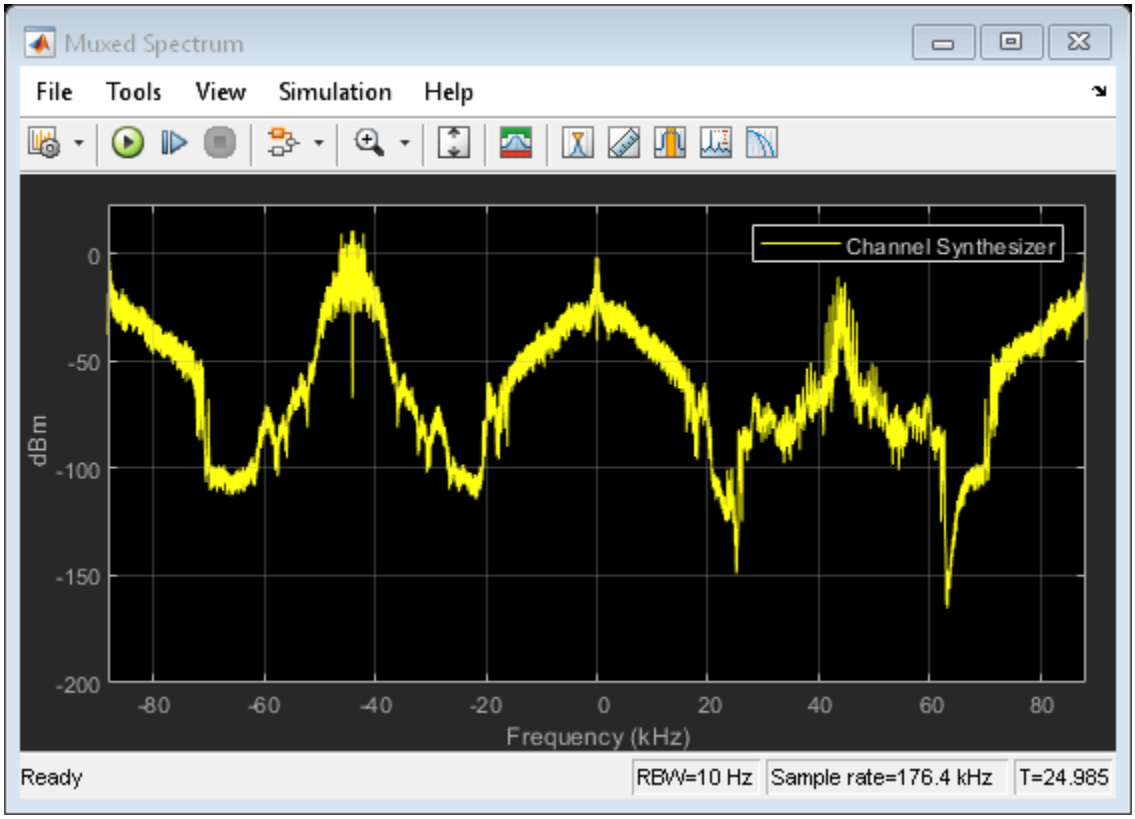


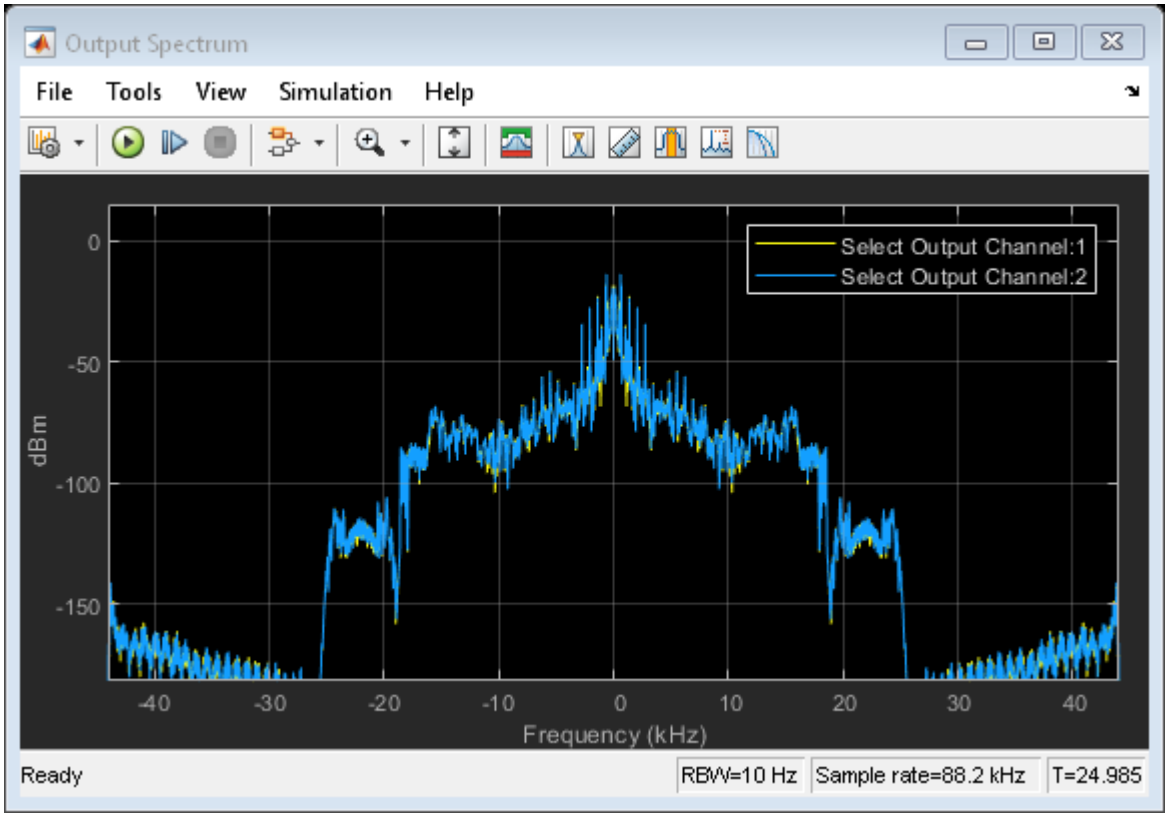
The Muxed Spectrum window shows the spectrum of the broadband signal. The Channelized Spectrum window shows the spectra of the four narrowband signals. The input and output spectra match for any selected signal.

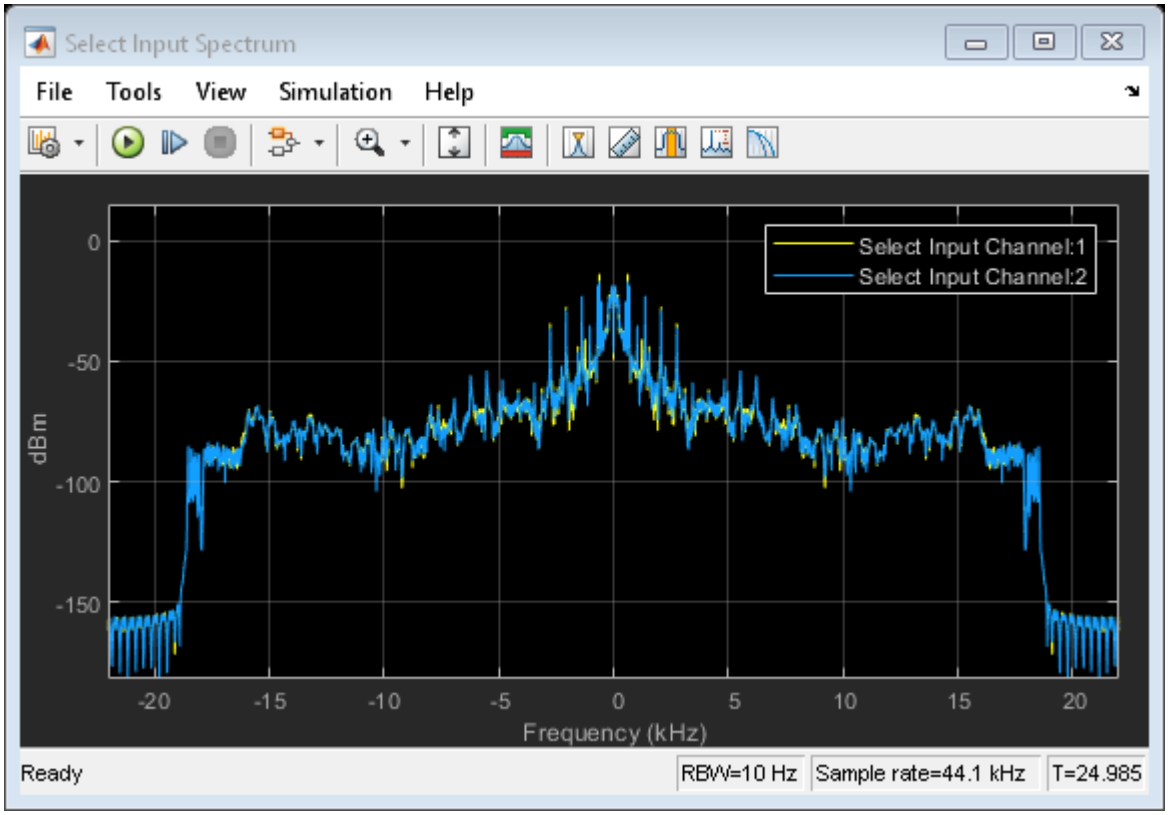
Increase the oversampling ratio of the Channelizer block to 2, which means the M/D ratio described in “Algorithm” is now 2.


Channelized Spectrum
File Tools View Simulation Help
Channelizer: 1
Channelizer: 2
Channelizer: 3
Channelizer: 4
dBm
0
-50
-100
-150
-40 -30 -20 -10 0 10 20 30 40
Frequency (kHz)
Ready
RBW=10 Hz
Sample rate=88.2 kHz
T=24.985


Input Spectrum
File Tools View Simulation Help
Convert to Complex and Mux: 1
Convert to Complex and Mux: 2
Convert to Complex and Mux: 3
Convert to Complex and Mux: 4
dBm
0
-50
-100
-150
-20 -15 -10 -5 0 5 10 15 20
Frequency (kHz)
Ready
RBW=10 Hz
Sample rate=44.1 kHz
T=24.985


Muxed Spectrum
File Tools View Simulation Help
Channel Synthesizer
dBm
0
-50
-100
-150
-200
-80 -60 -40 -20 0 20 40 60 80
Frequency (kHz)
Ready
RBW=10 Hz
Sample rate=176.4 kHz
T=24.985


Output Spectrum
File Tools View Simulation Help
Select Output Channel:1
Select Output Channel:2
dBm
0
-50
-100
-150
-40 -30 -20 -10 0 10 20 30 40
Frequency (kHz)
Ready
RBW=10 Hz
Sample rate=88.2 kHz
T=24.985



As you can see in the **Output Spectrum** plot, the output sample rate of the polyphase filter bank has increased by a factor of 2.

Save and close the model.

See Also

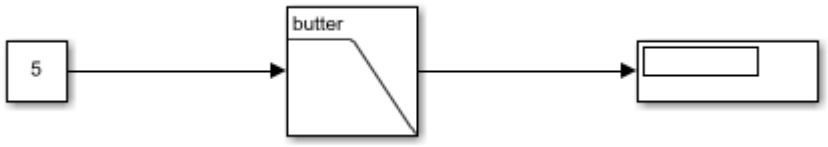
Blocks
Channel Synthesizer | Channelizer | From Multimedia File | To Audio Device

Filter input with Butterworth Filter in Simulink

This example shows how to use the Analog Filter Design block.

Open the Simulink model.

```
model = "ex_analog_filter_design";
open_system(model);
```



The **Analog Filter Design** block returns the filtered input as a scalar. Run the model.

```
sim(model);
```

Simulink Block Examples in DSP System Toolbox

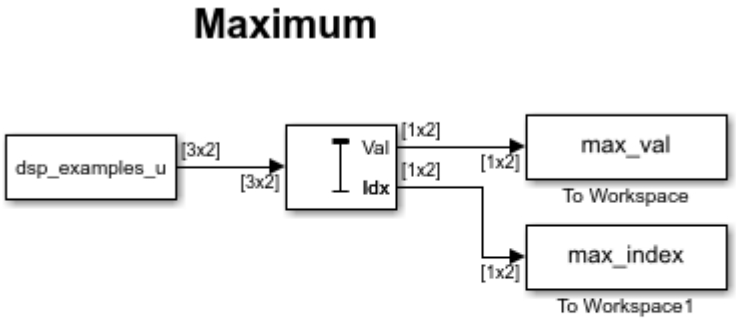
- “Compute the Maximum” on page 14-2
- “Compute the Running Maximum” on page 14-4
- “Compute the Minimum” on page 14-6
- “Compute the Running Minimum” on page 14-8
- “Compute the Mean” on page 14-10
- “Compute the Running Mean” on page 14-12
- “Compute the Histogram of Real and Complex Data” on page 14-14
- “Extract Submatrix from Input Signal” on page 14-19
- “Compute Difference of a Matrix” on page 14-21
- “Compute Maximum Column Sum of Matrix” on page 14-22
- “Convert Amplitude Signal to dB” on page 14-23
- “Extract Diagonal of Matrix” on page 14-24
- “Generate Diagonal Matrix from Vector Input” on page 14-25
- “Permute Matrix by Row or Column” on page 14-26
- “LDL Factorization of 3-by-3 Hermitian Positive Definite Matrix” on page 14-27

Compute the Maximum

Compute the maximum of a 3-by-2 matrix input, `dsp_examples_u`, using the Maximum block.

Open the model.

```
model = 'ex_maximum_ref';
open_system(model)
```



Note: This model created workspace variables called "dsp_examples_u", "max_val", and "max_index".

The **Mode** parameter of the Maximum block is set to `Value and Index`. The block processes the input as a two-channel signal with a frame size of three.

Run the model.

```
sim(model)
```

Display the input and output values.

```
disp('Data Input')
disp(dsp_examples_u)
disp('Maximum Values')
disp(max_val)
disp('Max Index Array')
disp(max_index)
```

```
Data Input
     6     1
     1     3
     3     9
    -7     2
     2     4
     5     1
     8     6
     0     2
    -1     5
    -3     0
     2     4
     1    17
```

```
Maximum Values
     6     9
     5     4
     8     6
     2    17
     0     0
     0     0

Max Index Array
     1     3
     3     2
     1     1
     2     3
     1     1
     1     1
```

In the `Value and Index` mode, the block outputs:

- The maximum value over each frame of data along the channel.
- The index of the maximum value in the respective frame.

Close the model.

```
close_system(model)
```

See Also

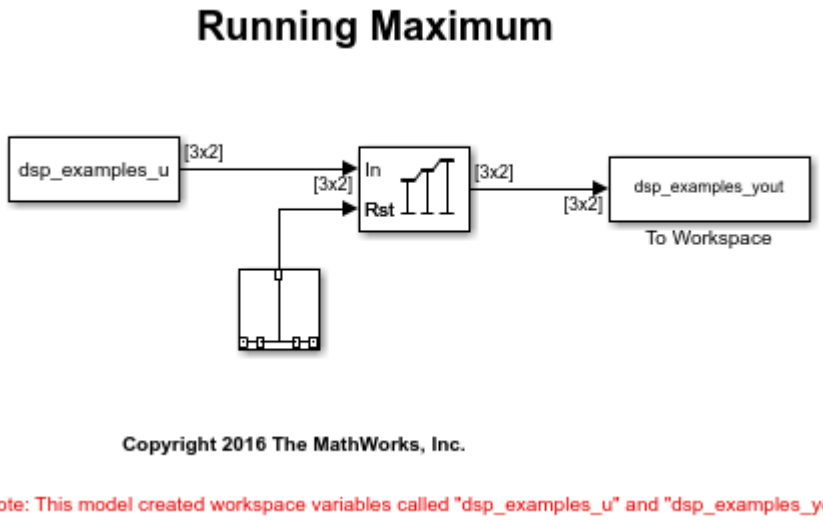
Blocks
Maximum | Signal From Workspace

Compute the Running Maximum

Compute the running maximum of a 3-by-2 matrix input, `dsp_examples_u`, using the Maximum block.

Open the model.

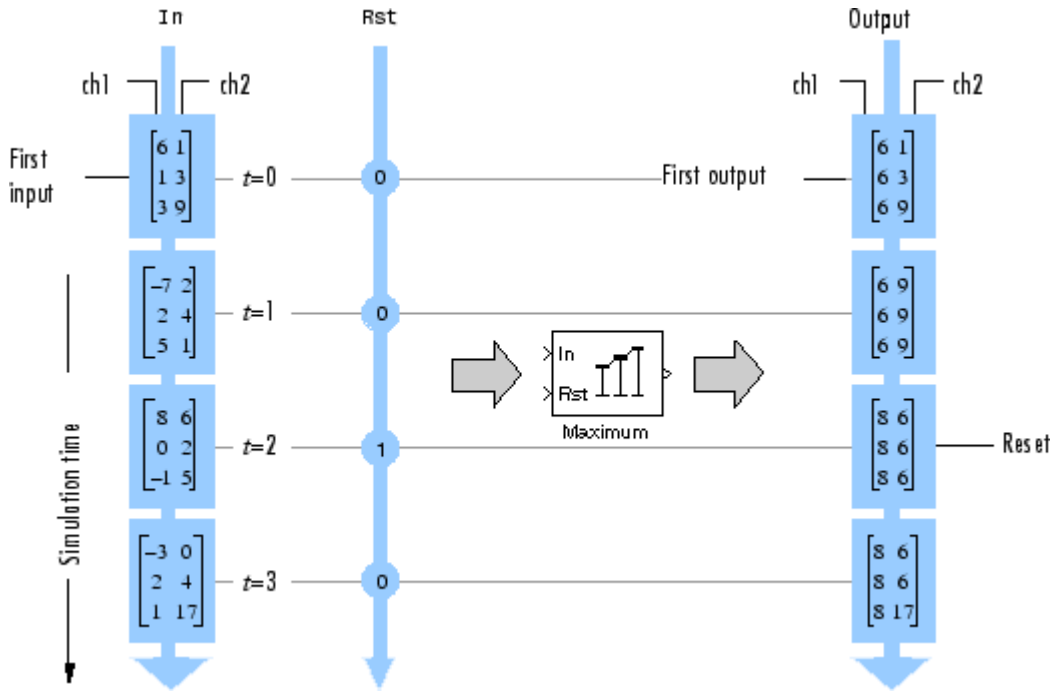
```
model = 'ex_runningmaximum_ref';
open_system(model)
```



The **Input processing** parameter is set to `Columns as channels (frame based)`. The block processes the input as a two-channel signal with a frame size of three. The running maximum is reset at $t = 2$ by an impulse to the block's **Rst** port.

Run the model.

```
sim(model)
```



In the `Running` mode, the block outputs the maximum value over each channel since the last reset. At $t = 2$, the reset event occurs. The maximum value in the second column changes to 6, even though 6 is less than 9, which was the maximum value since the previous reset event.

Close the model.

```
close_system(model)
```

See Also

Blocks

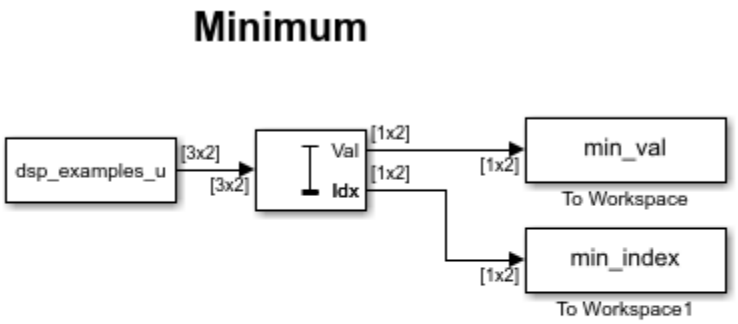
Discrete Impulse | Maximum | Signal From Workspace

Compute the Minimum

Compute the minimum of a 3-by-2 matrix input, `dsp_examples_u`, using the Minimum block.

Open the model.

```
model = 'ex_minimum_ref';
open_system(model)
```



The **Mode** parameter of the Minimum block is set to `Value and Index`. The block processes the input as a two-channel signal with a frame size of three.

Run the model. Display the input and output values.

```
sim(model)
disp('Data Input')
disp(dsp_examples_u)
disp('Minimum Values')
disp(min_val)
disp('Min Index Array')
disp(min_index)
```

```
Data Input
     6     1
     1     3
     3     9
    -7     2
     2     4
     5     1
     8     6
     0     2
    -1     5
    -3     0
     2     4
     1    17

Minimum Values
```

```
     1     1
    -7     1
    -1     2
    -3     0
     0     0
     0     0

Min Index Array
     2     1
     1     3
     3     2
     1     1
     1     1
     1     1
```

In the `Value and Index` mode, the block outputs:

- The minimum value over each frame of data along the channel.
- The index of the minimum value in the respective frame.

Close the model.

```
close_system(model)
```

See Also

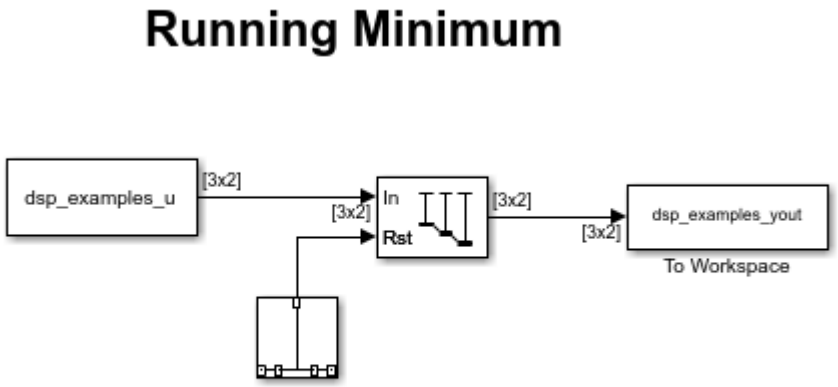
Blocks
Minimum | Signal From Workspace

Compute the Running Minimum

Compute the running minimum of a 3-by-2 matrix input, `dsp_examples_u`, using the Minimum block.

Open the model.

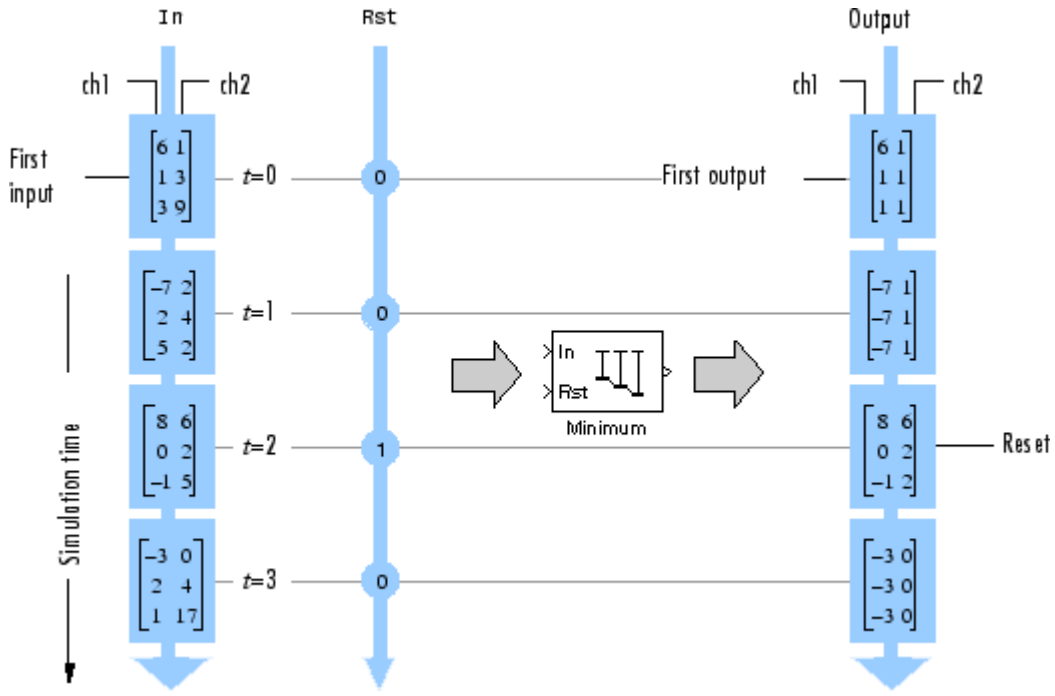
```
model = 'ex_runningminimum_ref';
open_system(model)
```



The **Input processing** parameter is set to `Columns as channels (frame based)`. The block processes the input as a two-channel signal with a frame size of three. The running minimum is reset at $t = 2$ by an impulse to the block's **Rst** port.

Run the model.

```
sim(model)
```



In the `Running` mode, the block outputs the minimum value over each channel since the last reset. At $t = 2$, the reset event occurs. The minimum value in the second column changes to 6, and then 2, even though these values are greater than 1, which was the minimum value since the previous reset event.

Close the model.

```
close_system(model)
```

See Also

Blocks

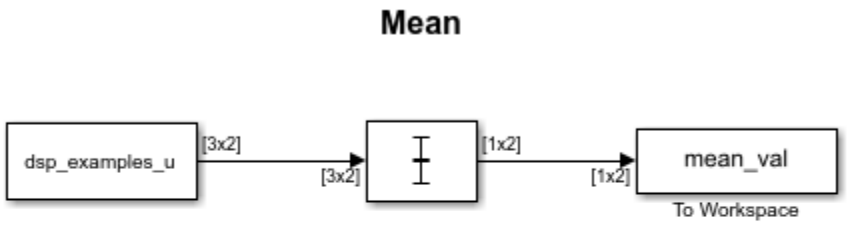
Discrete Impulse | Minimum | Signal From Workspace

Compute the Mean

Compute the mean of a 3-by-2 matrix input, `dsp_examples_u`, using the Mean block.

Open the model.

```
model = 'ex_mean_ref';
open_system(model)
```



In the Mean block, clear the **Running mean** check box and set the **Find the mean value over** parameter to `Each column`. The block processes the input as a two-channel signal with a frame size of three.

Run the model. Display the input and output values.

```
sim(model)
disp('Data Input')
disp(dsp_examples_u)
disp('Mean Values')
disp(mean_val)
```

```
Data Input
     6     1
     1     3
     3     9
    -7     2
     2     4
     5     1
     8     6
     0     2
    -1     5
    -3     0
     2     4
     1    17

Mean Values
    3.3333    4.3333
         0    2.3333
    2.3333    4.3333
```

```
         0    7.0000
```

Under these settings, the block outputs the mean value over each frame of data along both the channels.

Close the model.

```
close_system(model)
```

See Also

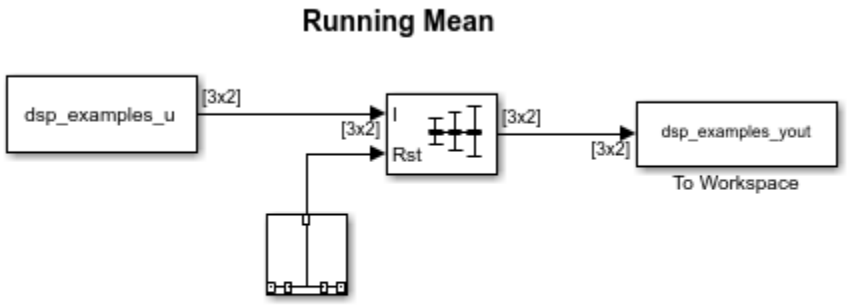
Blocks
Mean | Signal From Workspace | To Workspace

Compute the Running Mean

Compute the running mean of a 3-by-2 matrix input, `dsp_examples_u`, using the Mean block.

Open the model.

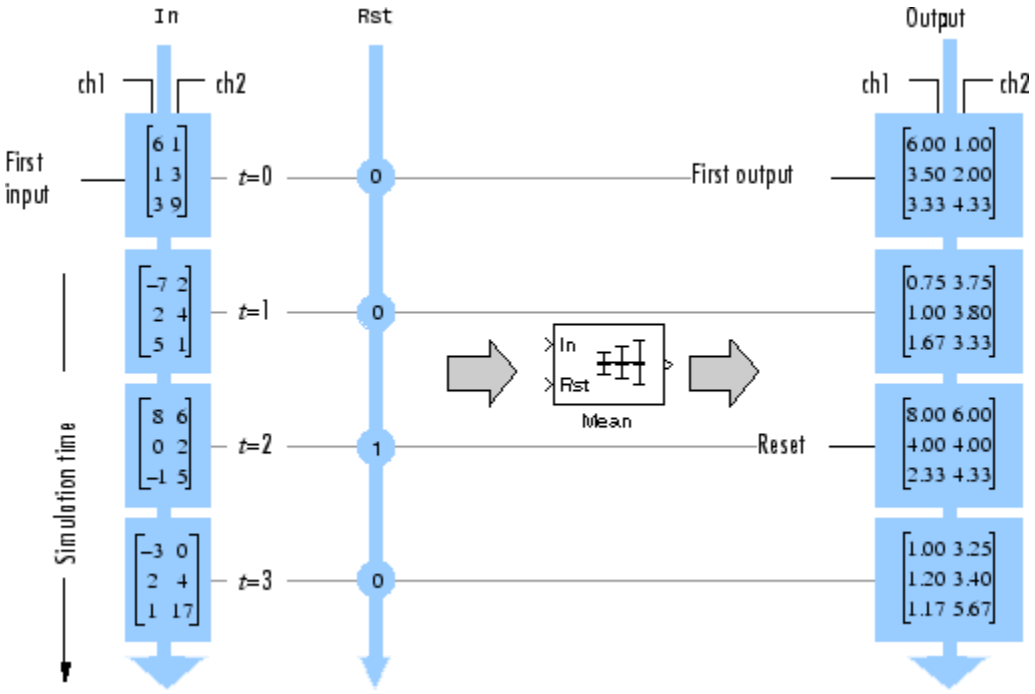
```
model = 'ex_runningmean_ref';
open_system(model)
```



The **Input processing** parameter is set to `Columns as channels (frame based)`. The block processes the input as a two-channel signal with a frame size of three. The running mean is reset at $t = 2$ by an impulse to the block's **Rst** port.

Run the model.

```
sim(model)
```



In the `Running` mode, the block outputs the mean value over each channel since the last reset. At t = 2, the reset event occurs. The window of data in the second column now contains only 6.

Close the model.

```
close_system(model)
```

See Also

Blocks
Mean | Signal From Workspace | To Workspace

Compute the Histogram of Real and Complex Data

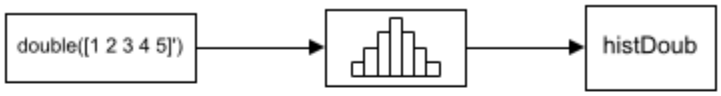
The bin boundaries created by the Histogram block are determined by the data type of the input. The following two models show the differences in the output of the Histogram block based on the data type of the input.

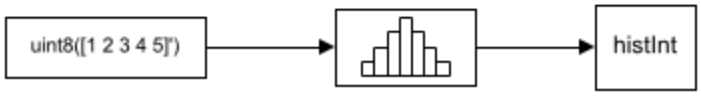
Real Input Data

When the input data is real, the bin boundaries are cast into the data type of the input.

Open the model.

```
modelRealData = 'ex_realData_hist';
open_system(modelRealData)
```



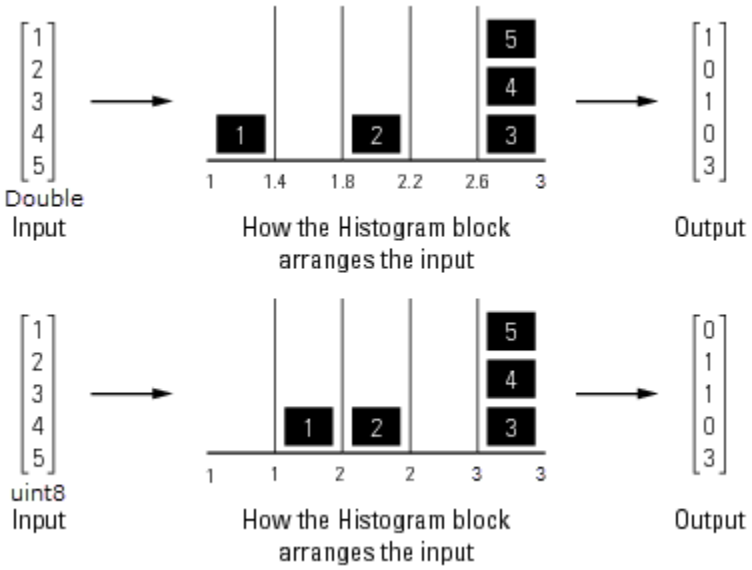


Run the model.

```
sim(modelRealData)
```

```
Warning: Reported in '<a href="matlab:open_and_hilite_hyperlink
('ex_realData_hist/Histogram1','error')">ex_realData_hist/Histogram1</a>': The
bin width resulting from the specified parameters is less than the precision of
the input data type. This might cause unexpected results. Since bin width is
calculated by ((upper limit - lower limit)/number of bins), you could increase
upper limit or decrease lower limit or number of bins.
```

The block produces two histogram outputs.



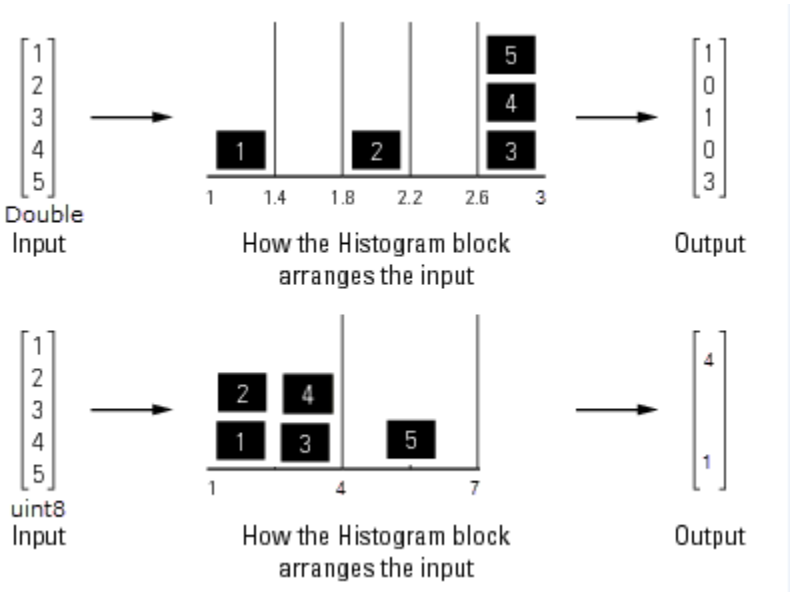
The output of the Histogram block differs based on the data type of the input. A warning occurs in the second histogram block, where the bin boundaries are uint8([1 1.4 1.8 2.2 2.6 3.0]) = [1 1 2 2 3 3]. The width of the first and third bins are 0, and the precision of the data is 1. The block expects the width of each bin to be at least equal to 1.

To resolve this warning, increase the upper limit of second Histogram block to 7 and decrease the number of bins to 2. The bin width becomes ((7-1)/2) = 3. With the integer input, the new bin boundaries are uint8[1 4 7] = [1 4 7]. The bins are spread out more evenly.

```
set_param('ex_realData_hist/Histogram1','umax','7','nbins','2');
```

Simulate the model. The warning no longer appears and the bins spread out more evenly.

```
sim(modelRealData)
```



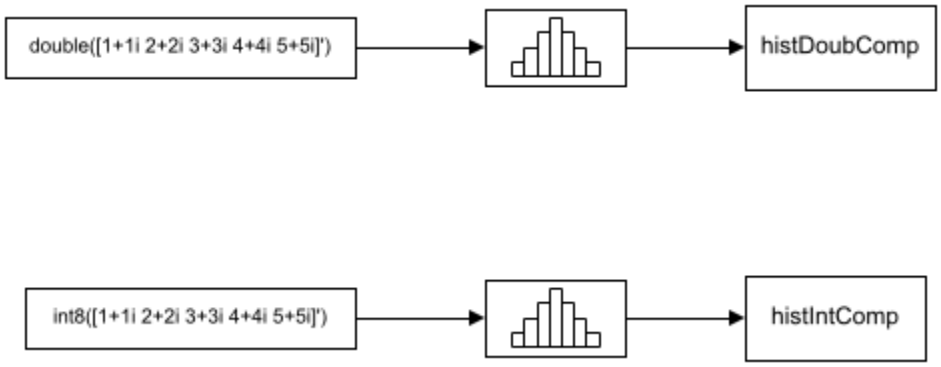
Complex Input Data

When the input data is complex:

- Bin boundaries for double-precision inputs are cast into the double data type. All complex, double-precision input values are placed in the bins according to their magnitude, which is the square root of the sum of the squares of the real and imaginary parts.
- Bin boundaries for integer inputs are cast into the data type double and squared. All complex, integer input values are placed in bins according to their magnitude-squared value.

Open the model.

```
modelComplexData = 'ex_complexData_hist';
open_system(modelComplexData)
```

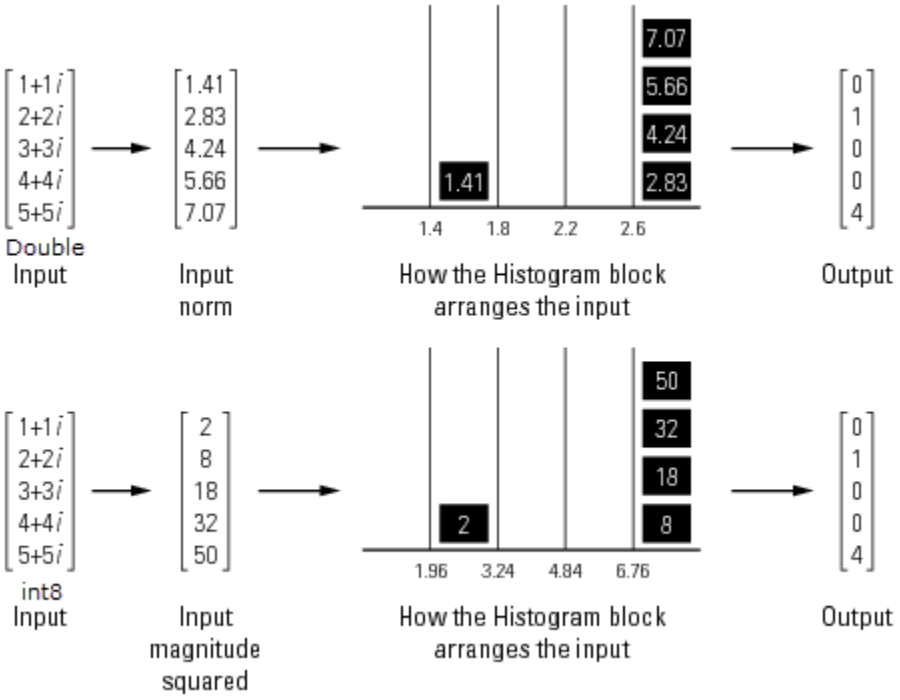


Run the model.

```
sim(modelComplexData)
```

```
Warning: Reported in '<a href="matlab:open_and_hilite_hyperlink
('ex_complexData_hist/Histogram1','error')">ex_complexData_hist/Histogram1</a>':
The bin width resulting from the specified parameters is less than the precision
of the input data type. This might cause unexpected results. Since bin width is
calculated by ((upper limit - lower limit)/number of bins), you could increase
upper limit or decrease lower limit or number of bins.
```

The model produces two histogram outputs.



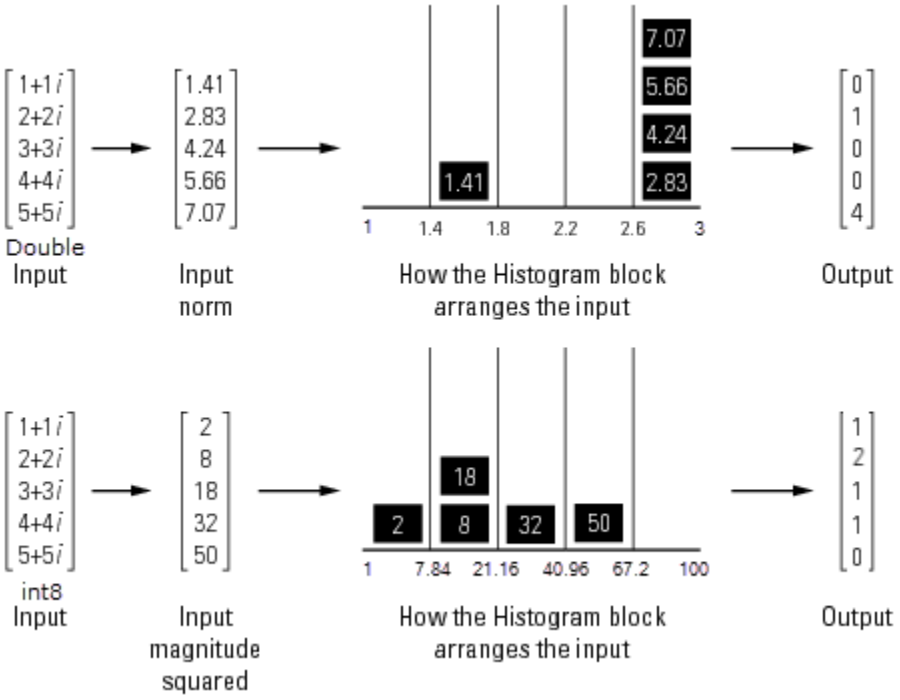
The top figure shows the histogram for the double-precision input, and the bottom figure shows the histogram for the integer input. The double-precision inputs are normalized for comparison, whereas the integer inputs are placed using their magnitude-squared value. A warning occurs in the second histogram block, where the bin boundaries are [1 $(1.4)^2$ $(1.8)^2$ $(2.2)^2$ $(2.6)^2$ $(3.0)^2$]. The precision of the data is at least 6, and the width of the bins is less than 2.

To resolve this warning, increase the upper limit of second Histogram block to 10. With the new upper limit, the bin boundaries are [1 $(2.8)^2$ $(4.6)^2$ $(6.4)^2$ $(8.2)^2$ 10^2] = [1 7.84 21.16 40.96 67.24 100].

```
set_param('ex_complexData_hist/Histogram1','umax','10');
```

Simulate the model. The warning no longer appears and the bins in the second Histogram block are spread out more evenly.

```
sim(modelComplexData)
```



Save and close the models.

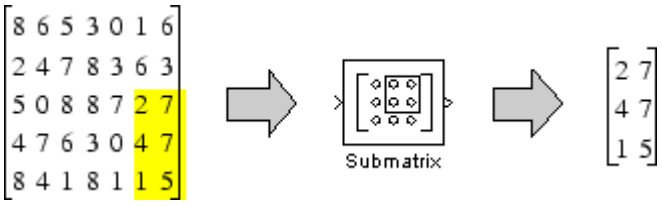
```
save_system(modelRealData);
save_system(modelComplexData)
close_system(modelRealData);
close_system(modelComplexData);
```

See Also

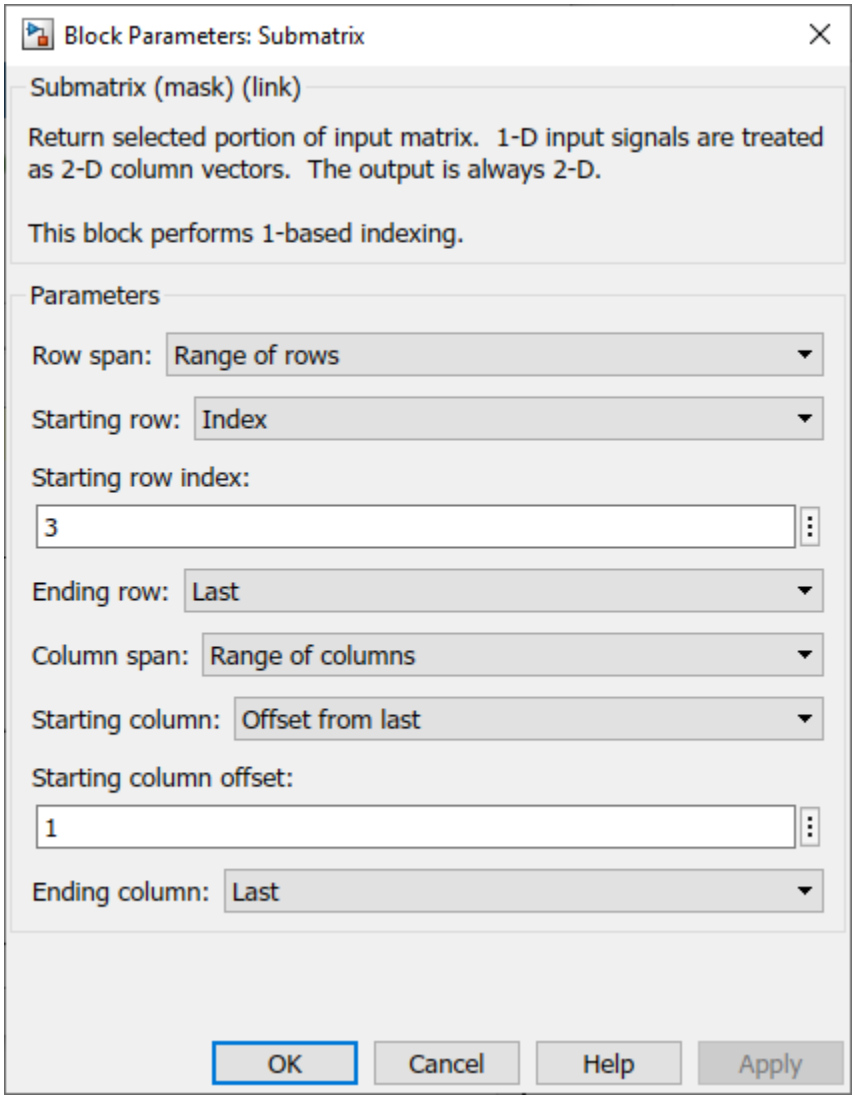
Blocks
Histogram

Extract Submatrix from Input Signal

This example shows how to use the Submatrix block to extract a 3-by-2 submatrix from the lower-right corner of a 5-by-7 input matrix. The following figure illustrates the operation of the Submatrix block with a 5-by-7 input matrix of random integer elements, `randi([0 9],5,7)`.



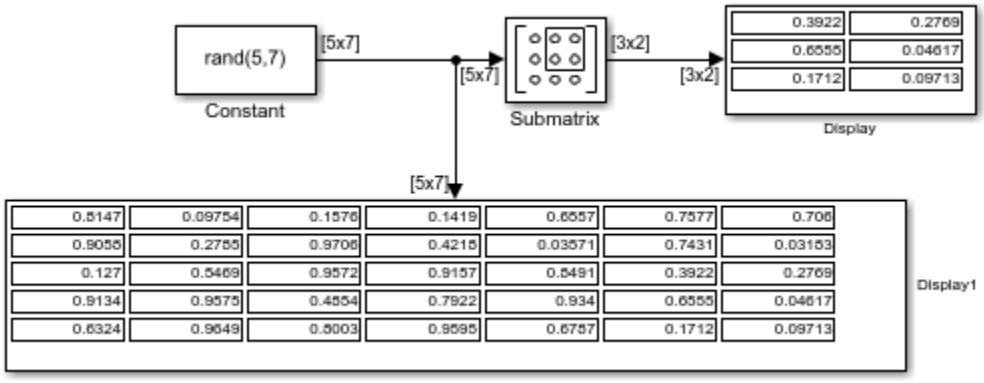
Here are the settings used for the `Submatrix` block in this example.



There are often several possible parameter combinations that you can use to select the same submatrix from the input. For example, in the case of a 5-by-7 input matrix, instead of specifying `Last` for **Ending column**, you could select the same submatrix by specifying:

- **Ending column** = `Index`
- **Ending column index** = 7

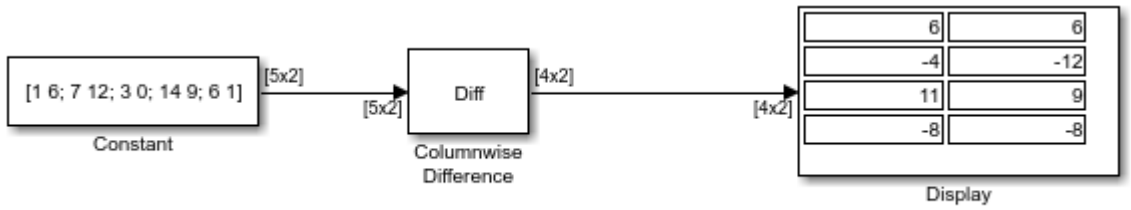
Open and simulate the model. You can see that the 3-by-2 submatrix from the lower-right corner of a 5-by-7 input matrix has been extracted.



Compute Difference of a Matrix

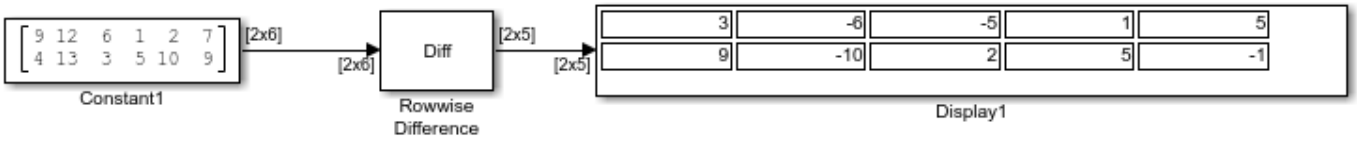
Open and run the model.

The first part of the model shows the output of the **Difference** block when the **Running difference** parameter is set to `No` and the **Difference along** parameter is set to `Columns`.



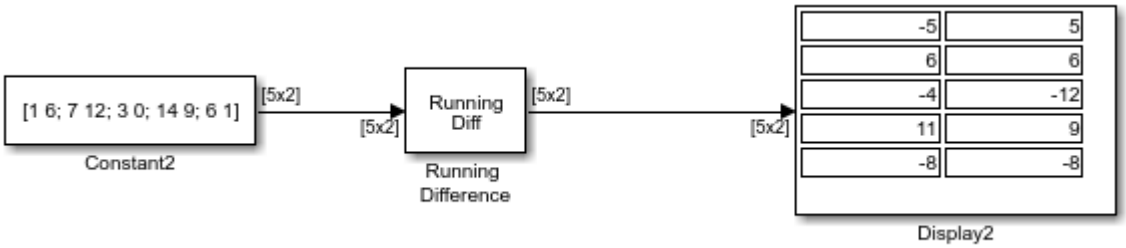
Copyright 2019 The MathWorks, Inc.

The second part of model shows the output of the block in a non-running mode when **Difference along** is set to `Rows`.



Copyright 2019 The MathWorks, Inc.

The last part of the model shows the output when the block computes the running difference.



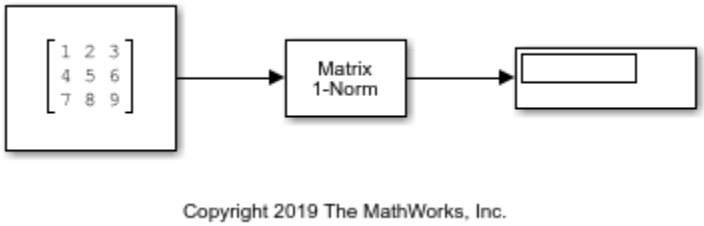
Copyright 2019 The MathWorks, Inc.

See Also

Blocks
Constant | Difference | Display

Compute Maximum Column Sum of Matrix

Open the Simulink model.



Copyright 2019 The MathWorks, Inc.

The `Matrix 1-Norm` block returns the highest column-sum among all the columns in the matrix. In this case, the highest column-sum is that of the third column. It has a sum of 18, so the output is expected to be 18.

Run the model to verify.

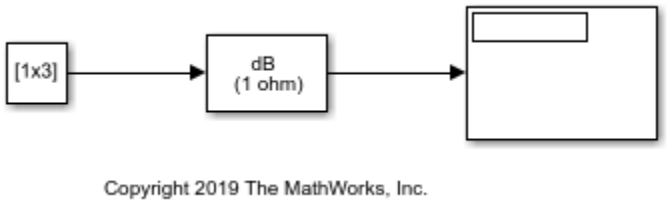
See Also

Blocks
Constant | Display | Matrix 1-Norm

Convert Amplitude Signal to dB

Example showing the modes of the dB Conversion block.

Open the model, `ex_dbconversion`.



Copyright 2019 The MathWorks, Inc.

Run the model.

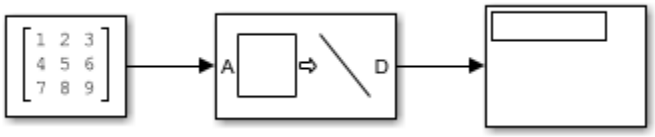
See Also

Blocks
Constant | Display | dB Conversion

Extract Diagonal of Matrix

This example shows how to use the Extract Diagonal block.

Open the Simulink model.



The **Extract Diagonal** block returns the main diagonal of the input matrix. The main diagonal of the input matrix is [1,5,9].

Run the model to verify.

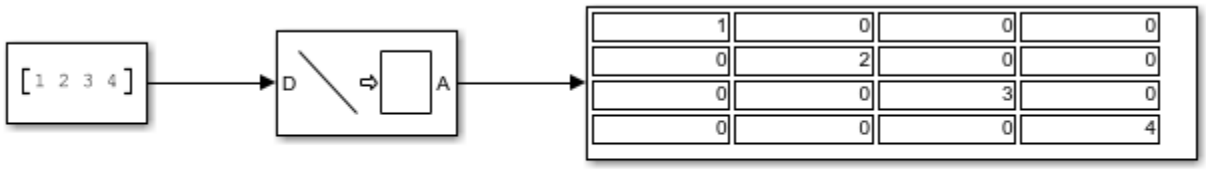
See Also

Blocks
Constant | Display | Extract Diagonal

Generate Diagonal Matrix from Vector Input

This example shows how to use the Create Diagonal Matrix block.

Open the Simulink model.



Copyright 2019 The MathWorks, Inc.

The **Create Diagonal Matrix** block creates a diagonal matrix from the input values. The output matrix in the model has the input vector as its diagonal.

Run the model to verify the output.

See Also

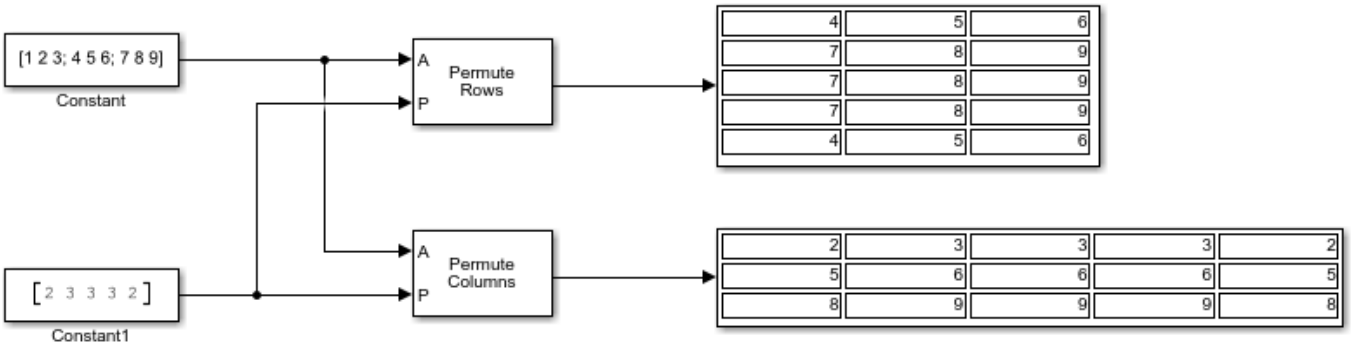
Blocks
Constant | Create Diagonal Matrix | Display

Permute Matrix by Row or Column

This example shows how to use the Permute Matrix block to permute blocks by row or column.

In the model, the top Permute Matrix block places the second row of the input matrix in the first and fifth rows of the output matrix. The block places the third row of the input matrix in the three middle rows of the output matrix. The bottom Permute Matrix block places the second column of the input matrix in the first and fifth columns of the output matrix. It places the third column of the input matrix in the three middle columns of the output matrix.

Rows and columns of A can appear any number of times in the output, or not at all depending on the index vector.



Copyright 2019 The MathWorks, Inc.

See Also

Blocks

Constant | Display | Permute Matrix

LDL Factorization of 3-by-3 Hermitian Positive Definite Matrix

This example shows how to use LDL Factorization to LDL-factor a 3-by-3 Hermitian positive definite matrix.

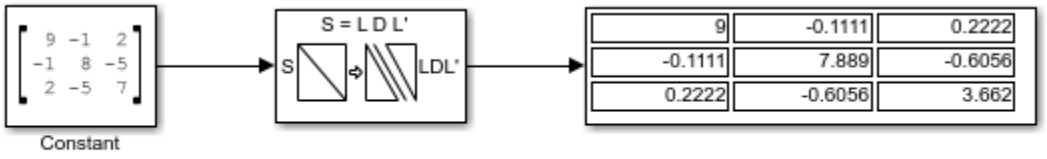
For the input in the model, the corresponding L and D values become:

```
L =

    1.0000         0         0
   -0.1100    1.0000         0
    0.2200   -0.6100    1.0000


D =

    9.0000         0         0
         0    7.8900         0
         0         0    3.6600


Ltranspose =

    1.0000   -0.1100    0.2200
         0    1.0000   -0.6100
         0         0    1.0000
```

The output matrix is composed of the bottom left triangle of the `L` matrix, the diagonal of the `D` matrix, and the top right triangle of `Ltranspose`. When they are combined, the output is:

```
S =

    9.0000   -0.1100    0.2200
   -0.1100    7.8900   -0.6100
    0.2200   -0.6100    3.6600
```

In the model, the output is the same.



Copyright 2019 The MathWorks, Inc.

See Also

Blocks
Constant | Display | LDL Factorization

Simulink Block Examples in Transforms and Spectral Analysis Category

Analyze a Subband of Input Frequencies Using Zoom FFT

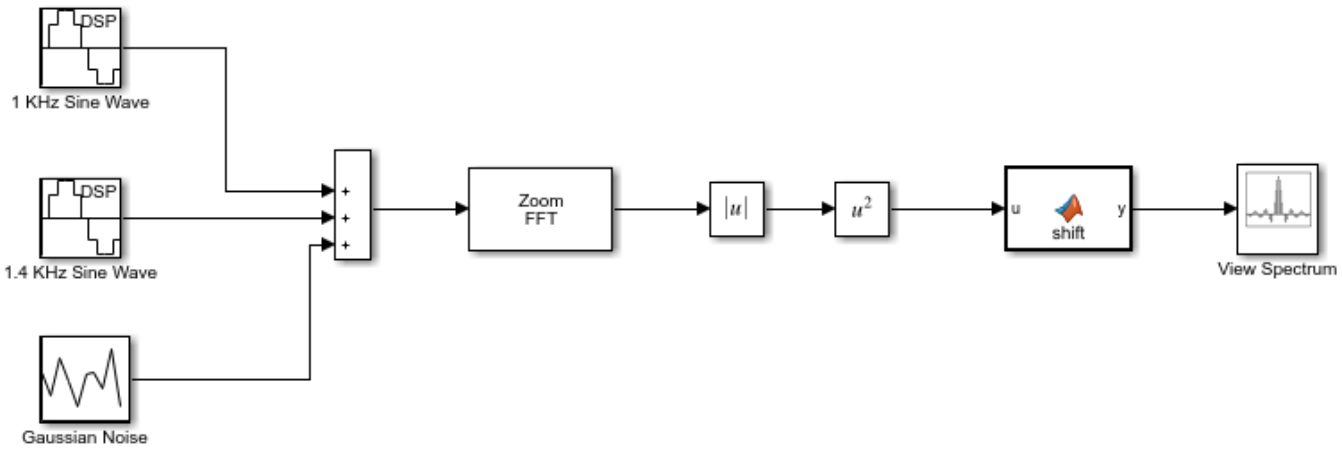
The `Zoom FFT` block implements zoom FFT based on the multirate multistage bandpass filter designed in “Complex Bandpass Filter Design”. If you specify the center frequency and the decimation factor, the `Zoom FFT` block designs and applies the filter to the input signal. Using zoom FFT, you can zoom into the tones of the input sine wave.

The input is a noisy sine wave signal with frequencies at 1 kHz and 1.4 kHz. The noise is an additive white Gaussian noise with zero mean and a variance of 1e-2. The input sample rate, Fs, is 44.1 kHz and the input frame size, L, is 440 samples.

Configure the `Zoom FFT` block to analyze a bandwidth of 800 Hz with the center frequency at 1200 Hz. The decimation factor, D, is the ratio of the input sample rate, 44.1 kHz, and the bandwidth of interest, 800 Hz. The FFT length is the ratio of input frame size, 440, and the decimation factor. The FFT is computed over frequencies starting at 800 Hz and spaced by Fs/L Hz apart, which is the resolution or the minimum frequency that can be discriminated. With the above values, the resolution is $44100/440$, or approximately 100 Hz.

Open the model.

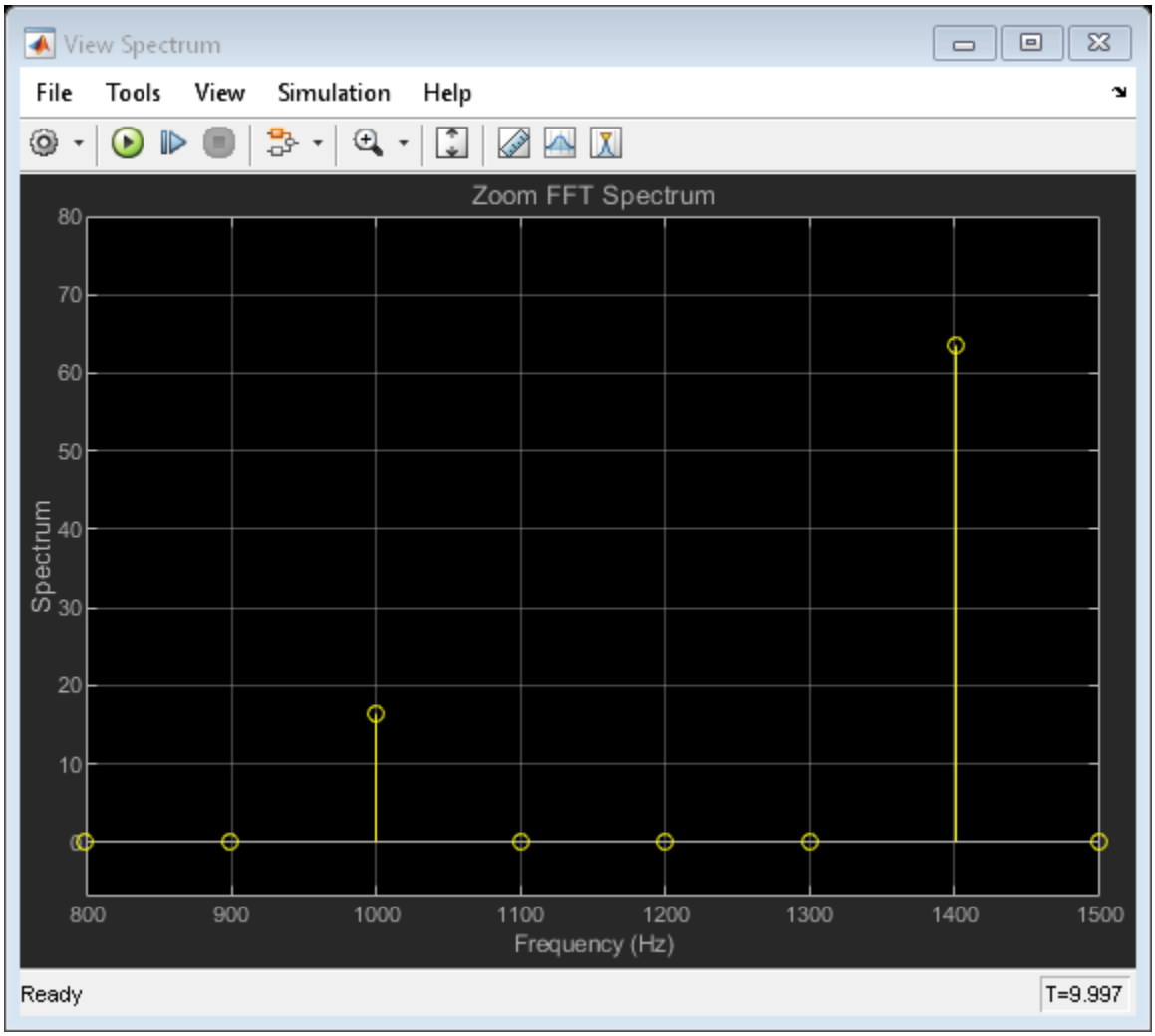
```
model = 'zoomfftEx';
open_system(model)
```



Copyright 2016-2017 The Mathworks, Inc.

Run the model. Compute the square of the magnitude of the zoom FFT output, perform FFT shift, and view the resulting spectrum in array plot.

```
sim(model)
```



The spectrum shows the frequencies in the range [800 1600] Hz, with tones at 1 kHz and 1.4 kHz. The FFT length reduced to length L/D. This is the basic concept of zoom FFT. By decimating the original signal, you can retain the same resolution you would achieve with a full size FFT on your original signal by computing a small FFT on a shorter signal. You can alternatively achieve a better resolution by using the same FFT length.

If you make any changes to the model, save the model before closing the model.

```
close_system(model)
bdclose('all');
```

See Also

Blocks

Abs | Array Plot | Math Function | Random Source | Sine Wave | Zoom FFT

Transforms, Estimation, and Spectral Analysis

Learn about transforms, estimation and spectral analysis.

- “Transform Time-Domain Data into Frequency Domain” on page 16-2
- “Transform Frequency-Domain Data into Time Domain” on page 16-4
- “Linear and Bit-Reversed Output Order” on page 16-6
- “Calculate Channel Latencies Required for Wavelet Reconstruction” on page 16-8
- “Estimate the Power Spectrum in MATLAB” on page 16-14
- “Estimate the Power Spectrum in Simulink” on page 16-27
- “Estimate the Transfer Function of an Unknown System” on page 16-43
- “View the Spectrogram Using Spectrum Analyzer” on page 16-50
- “Spectral Analysis” on page 16-59

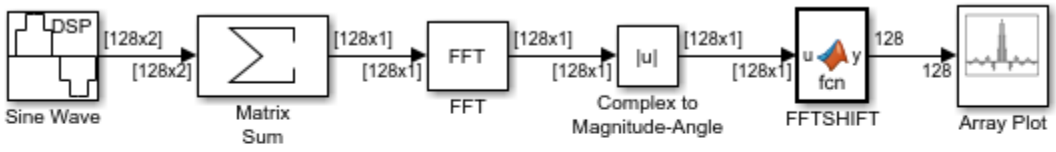
Transform Time-Domain Data into Frequency Domain

This example shows how to transform time-domain data into the frequency domain using the FFT block.

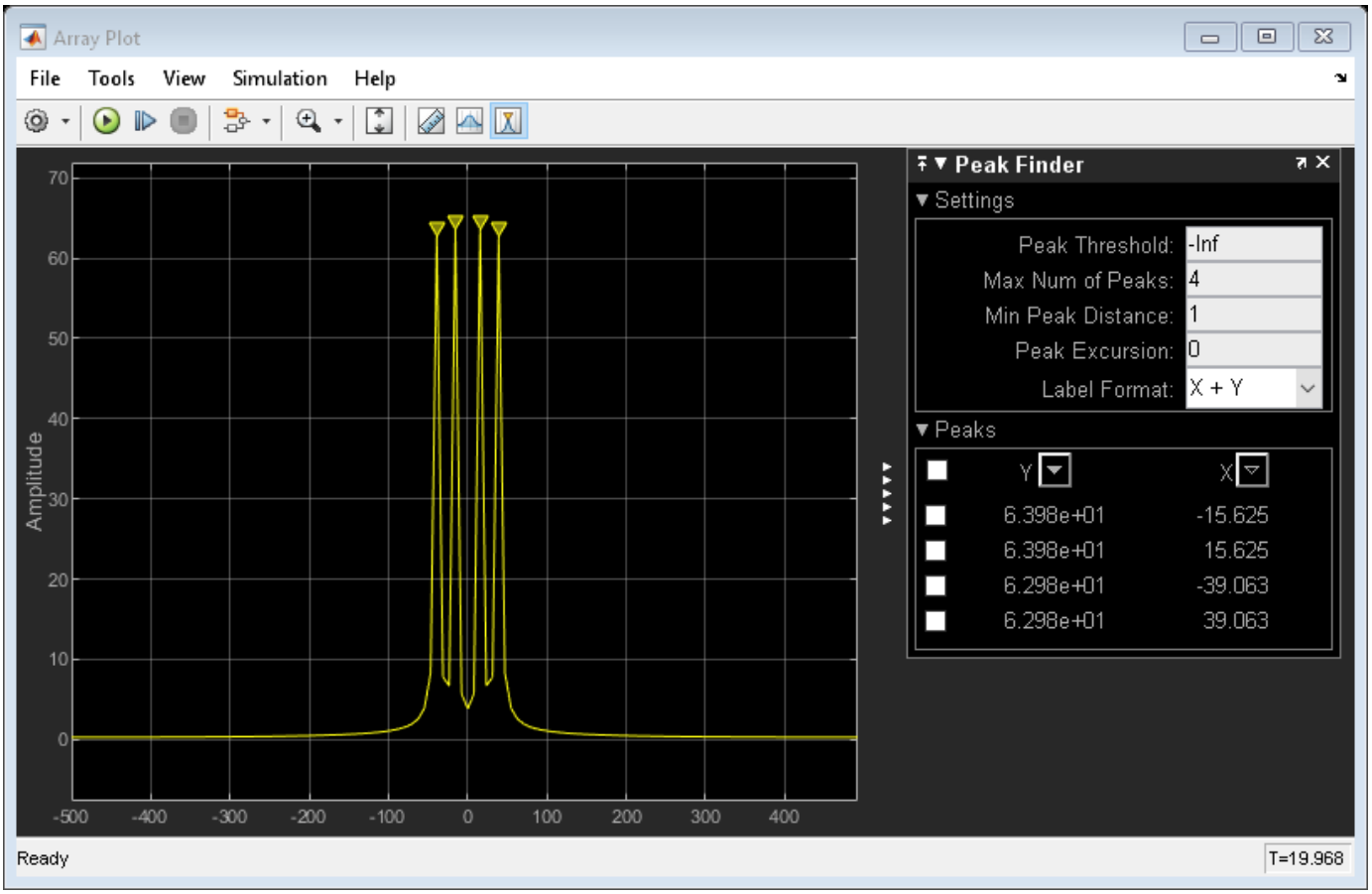
Use the Sine Wave block to generate two sinusoids, one at 15 Hz and the other at 40 Hz. Use the Matrix Sum block to add the sinusoids point-by-point to generate the compound sinusoid:

$$u = sin(2 * 15\pi t) + sin(2 * 40\pi t)$$

Transform this sinusoid into the frequency domain using an FFT block. See the `ex_fft_tut` model:

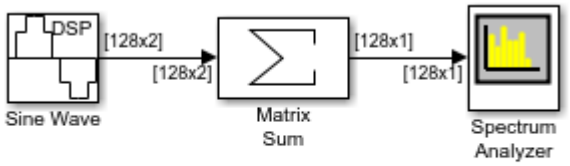


Copyright 2018 The MathWorks, Inc.

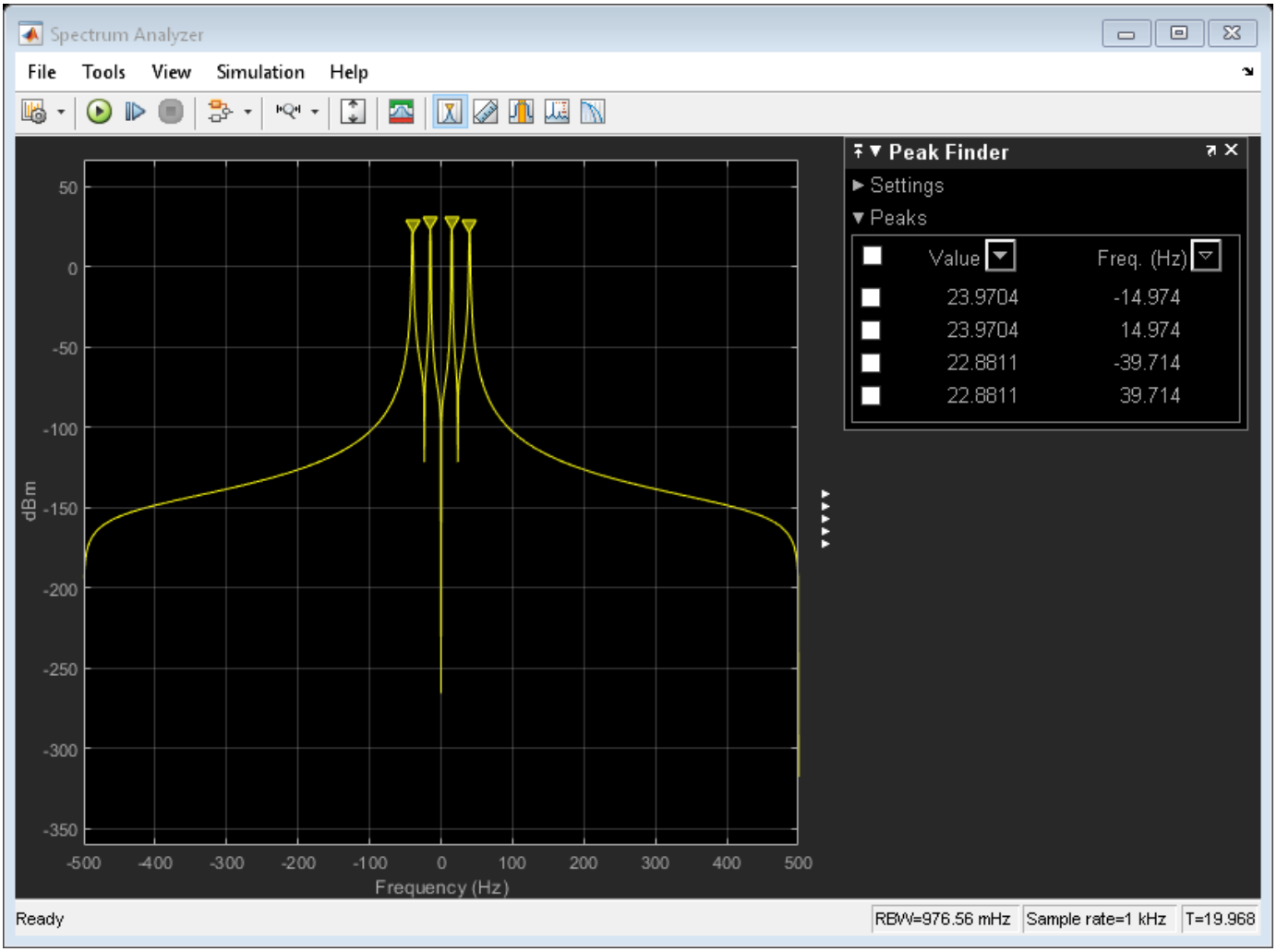


The scope shows peaks at 15 and 40 Hz, as expected. You have now transformed two sinusoidal signals from the time domain to the frequency domain.

You can use a Spectrum Analyzer block in place of the sequence of FFT, Complex to Magnitude-Angle, MATLAB Function, and Array Plot blocks. The Spectrum Analyzer computes the magnitude FFT and shifts the FFT internally. See the `ex_time_freq_sa` model:



Copyright 2018 The MathWorks, Inc.



The blocks in the Power Spectrum Estimation library compute the FFT internally.

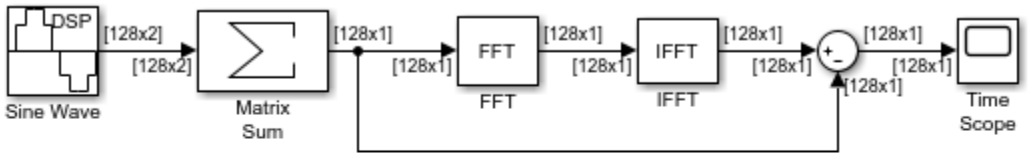
Transform Frequency-Domain Data into Time Domain

When you want to transform frequency-domain data into the time domain, use the IFFT block.

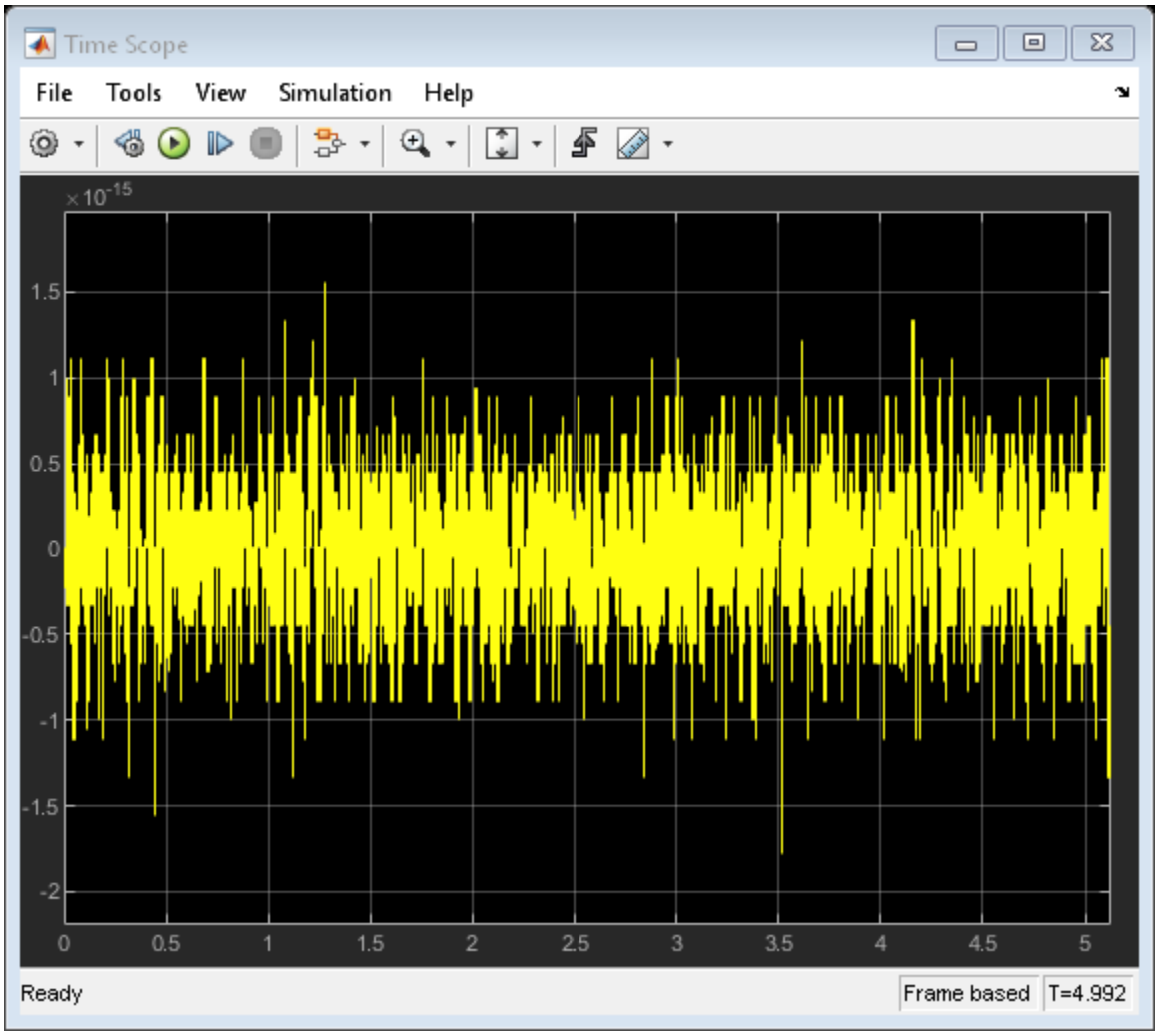
Use the Sine Wave block to generate two sinusoids, one at 15 Hz and the other at 40 Hz. Use a Matrix Sum block to add the sinusoids point-by-point to generate the compound sinusoid:

$$u = sin(30\pi t) + sin(80\pi t)$$

Transform this sinusoid into the frequency domain using an FFT block, and then immediately transform the frequency-domain signal back to the time domain using the IFFT block. Plot the difference between the original time-domain signal and transformed time-domain signal using a scope:



Copyright 2004-2018 The MathWorks, Inc.



The two signals are identical to within round-off error. The scope shows that the difference between the two signals is on the order of 10^{-15}.

Linear and Bit-Reversed Output Order

In this section...
“FFT and IFFT Blocks Data Order” on page 16-6 “Find the Bit-Reversed Order of Your Frequency Indices” on page 16-6

FFT and IFFT Blocks Data Order

The FFT block enables you to output the frequency indices in linear or bit-reversed order. Because linear ordering of the frequency indices requires a bit-reversal operation, the FFT block may run more quickly when the output frequencies are in bit-reversed order.

The input to the IFFT block can be in linear or bit-reversed order. Therefore, you do not have to alter the ordering of your data before transforming it back into the time domain. However, the IFFT block may run more quickly when the input is provided in bit-reversed order.

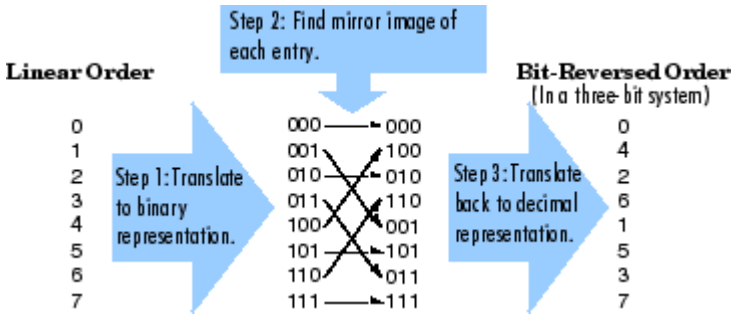
Note that FFT and IFFT blocks are not supported in MATLAB Online.

Find the Bit-Reversed Order of Your Frequency Indices

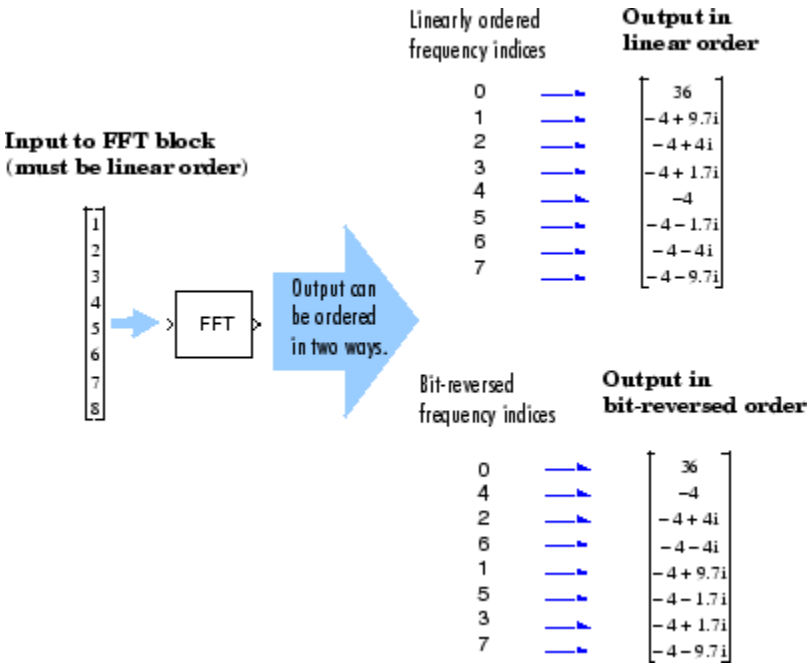
Two numbers are bit-reversed values of each other when the binary representation of one is the mirror image of the binary representation of the other. For example, in a three-bit system, one and four are bit-reversed values of each other, since the three-bit binary representation of one, 001, is the mirror image of the three-bit binary representation of four, 100. In the diagram below, the frequency indices are in linear order. To put them in bit-reversed order

1. Translate the indices into their binary representation with the minimum number of bits. In this example, the minimum number of bits is three because the binary representation of 7 is 111.
2. Find the mirror image of each binary entry, and write it beside the original binary representation.
3. Translate the indices back to their decimal representation.

   The frequency indices are now in bit-reversed order.



The next diagram illustrates the linear and bit-reversed outputs of the FFT block. The output values are the same, but they appear in different order.


Input to FFT block
(must be linear order)
1
2
3
4
5
6
7
8
FFT
Output can be ordered in two ways.
Linearly ordered frequency indices
0
1
2
3
4
5
6
7
Output in linear order
36
−4 + 9.7i
−4 + 4i
−4 + 1.7i
−4
−4 − 1.7i
−4 − 4i
−4 − 9.7i
Bit-reversed frequency indices
0
4
2
6
1
5
3
7
Output in bit-reversed order
36
−4
−4 + 4i
−4 − 4i
−4 + 9.7i
−4 − 1.7i
−4 + 1.7i
−4 − 9.7i

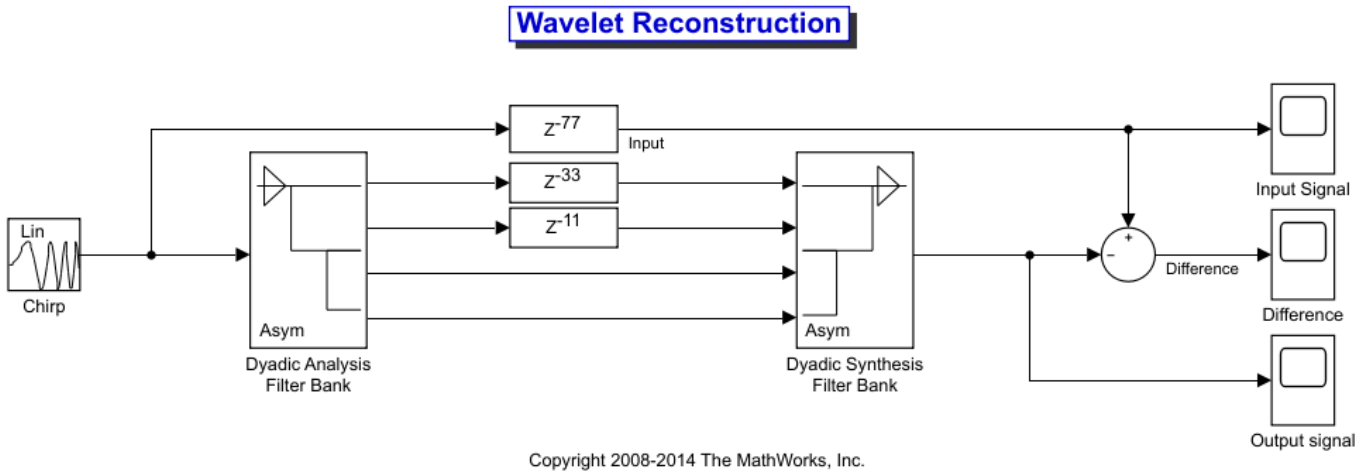
Calculate Channel Latencies Required for Wavelet Reconstruction

In this section...
“Analyze Your Model” on page 16-8
“Calculate the Group Delay of Your Filters” on page 16-9
“Reconstruct the Filter Bank System” on page 16-10
“Equalize the Delay on Each Filter Path” on page 16-11
“Update and Run the Model” on page 16-12
“References” on page 16-13

Analyze Your Model

The following sections guide you through the process of calculating the channel latencies required for perfect wavelet reconstruction. This example uses the `ex_wavelets` model, but you can apply the process to perform perfect wavelet reconstruction in any model. To open the example model, type `ex_wavelets` at the MATLAB command line.

Note You must have a Wavelet Toolbox™ product license to run the `ex_wavelets` model.



Before you can begin calculating the latencies required for perfect wavelet reconstruction, you must know the types of filters being used in your model.

The Dyadic Analysis Filter Bank and the Dyadic Synthesis Filter Bank blocks in the `ex_wavelets` model have the following settings:

- **Filter** = `Biorthogonal`
- **Filter order [synthesis/analysis]** = `[3/5]`
- **Number of levels** = 3

- **Tree structure** = `Asymmetric`
- **Input** = `Multiple ports`

Based on these settings, the Dyadic Analysis Filter Bank and the Dyadic Synthesis Filter Bank blocks construct biorthogonal filters using the Wavelet Toolbox `wfilters` function.

Calculate the Group Delay of Your Filters

Once you know the types of filters being used by the Dyadic Analysis and Dyadic Synthesis Filter Bank blocks, you need to calculate the group delay of those filters. To do so, you can use the Signal Processing Toolbox `fvtool`.

Before you can use `fvtool`, you must first reconstruct the filters in the MATLAB workspace. To do so, type the following code at the MATLAB command line:

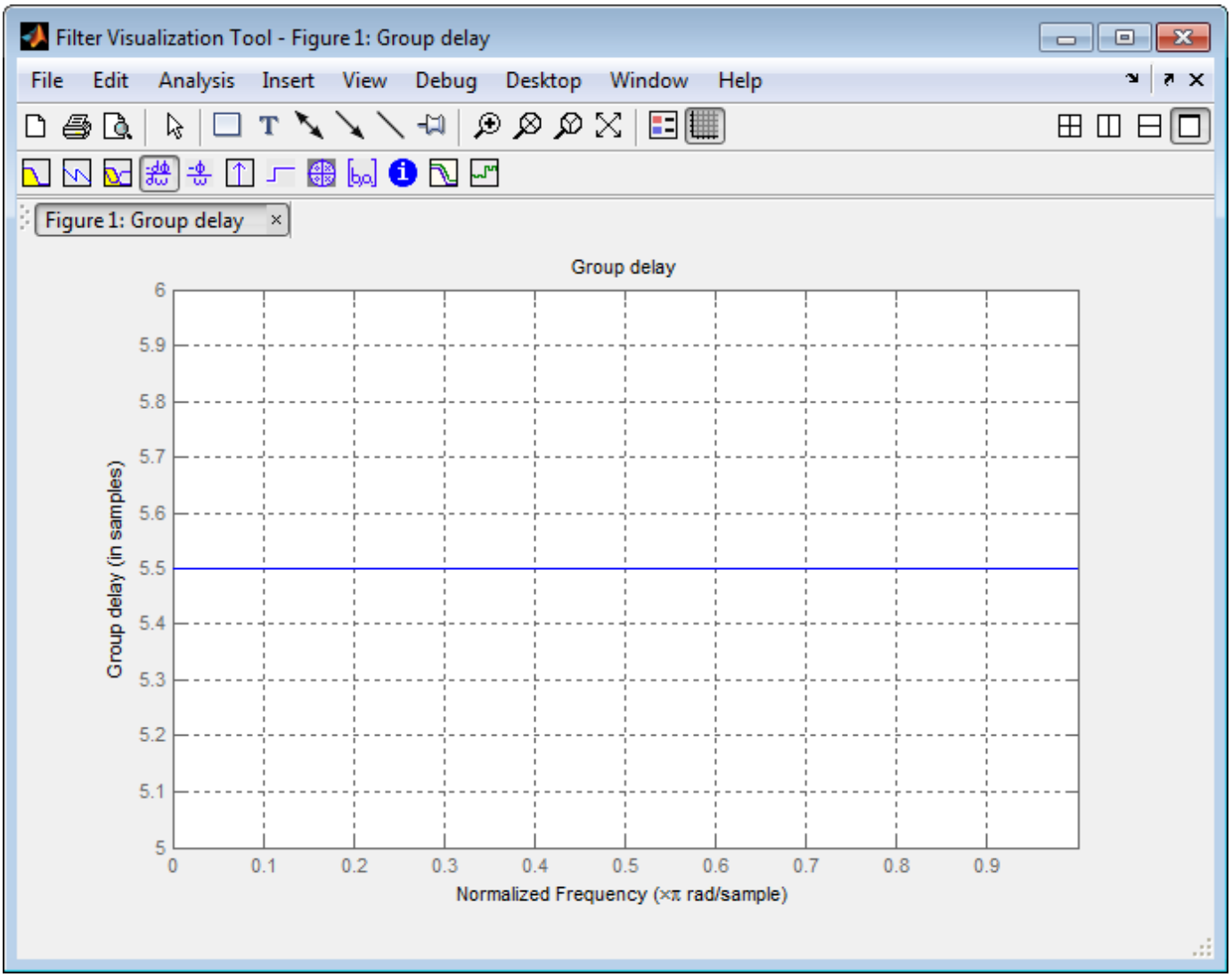
```
[Lo_D, Hi_D, Lo_R, Hi_R] = wfilters('bior3.5')
```

Where `Lo_D` and `Hi_D` represent the low- and high-pass filters used by the Dyadic Analysis Filter Bank block, and `Lo_R` and `Hi_R` represent the low- and high-pass filters used by the Dyadic Synthesis Filter Bank block.

After you construct the filters in the MATLAB workspace, you can use `fvtool` to determine the group delay of the filters. To analyze the low-pass biorthogonal filter used by the Dyadic Analysis Filter Bank block, you must do the following:

- Type `fvtool(Lo_D)` at the MATLAB command line to launch the Filter Visualization Tool.
- When the Filter Visualization Tool opens, click the Group delay response button ( ) on the toolbar, or select **Group Delay Response** from the **Analysis** menu.

Based on the Filter Visualization Tool's analysis, you can see that the group delay of the Dyadic Analysis Filter Bank block's low-pass biorthogonal filter is 5.5.



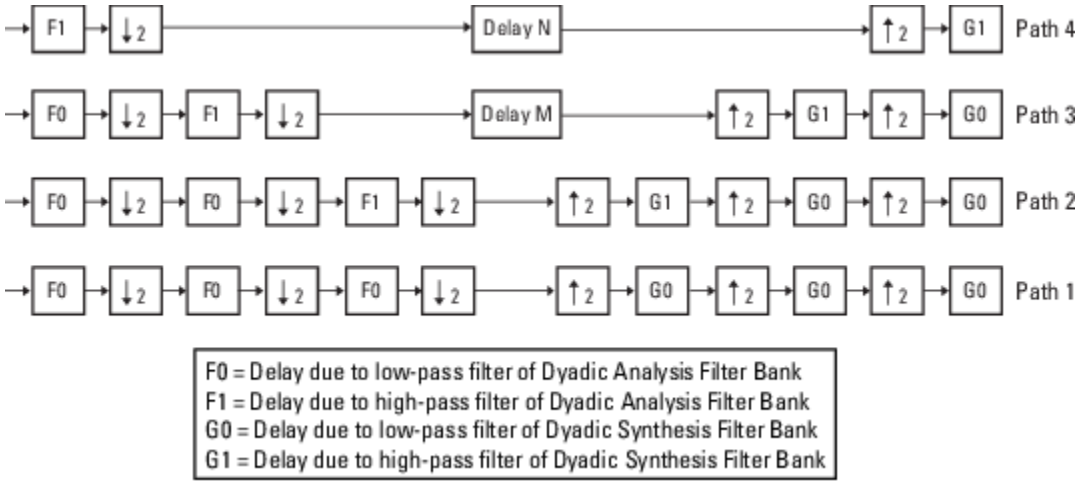
Note Repeat this procedure to analyze the group delay of each of the filters in your model. This section does not show the results for each filter in the `ex_wavelets` model because all wavelet filters in this particular example have the same group delay.

Reconstruct the Filter Bank System

To determine the delay introduced by the analysis and synthesis filter bank system, you must reconstruct the tree structures of the Dyadic Analysis Filter Bank and the Dyadic Synthesis Filter Bank blocks. To learn more about constructing tree structures for the Dyadic Analysis Filter Bank and Dyadic Synthesis Filter Bank blocks, see the following sections of the DSP System Toolbox User's Guide:

- “Dyadic Analysis Filter Banks” on page 6-21
- “Dyadic Synthesis Filter Banks” on page 6-23

Because the filter blocks in the `ex_wavelets` model use biorthogonal filters with three levels and an asymmetric tree structure, the filter bank system appears as shown in the following figure.



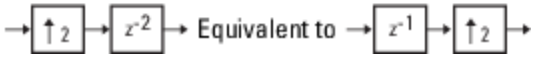
The extra delay values of M and N on paths 3 and 4 in the previous figure ensure that the total delay on each of the four filter paths is identical.

Equalize the Delay on Each Filter Path

Now that you have reconstructed the filter bank system, you can calculate the delay on each filter path. To do so, use the following Noble identities:

First Noble Identity

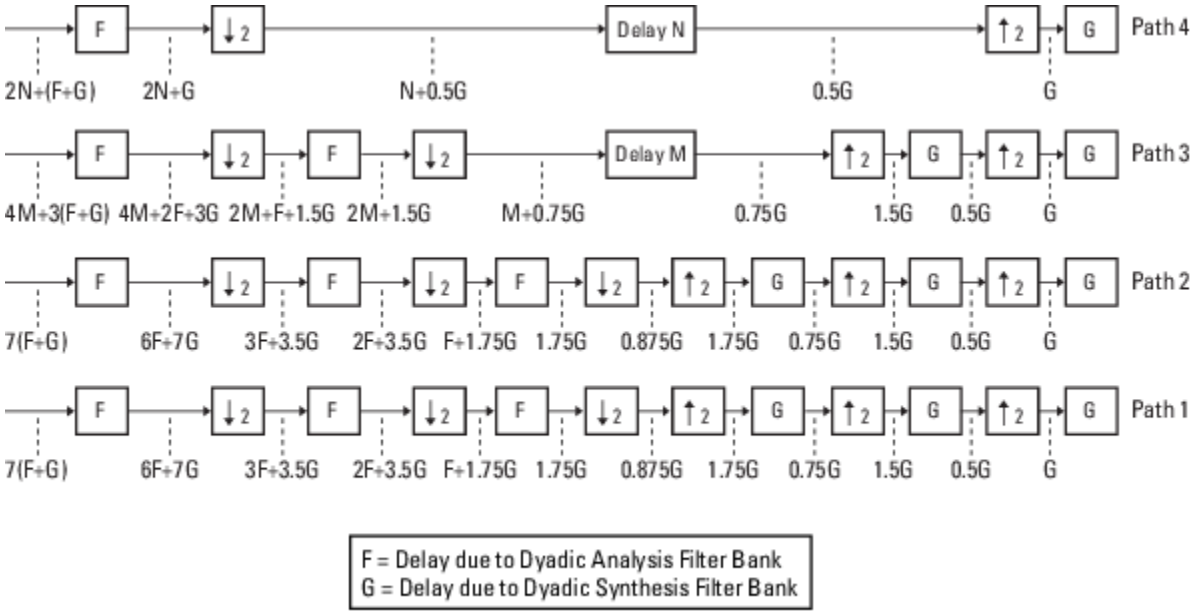
Second Noble Identity



You can apply the Noble identities by summing the delay on each signal path from right to left. The first Noble identity indicates that moving a delay of 1 before a downsample of 2 is equivalent to multiplying that delay value by 2. Similarly, the second Noble identity indicates that moving a delay of 2 before an upsample of 2 is equivalent to dividing that delay value by 2.

The `fvtool` analysis in step 1 found that both the low- and high-pass filters of the analysis filter bank have the same group delay ($F_0 = F_1 = 5.5$). Thus, you can use F to represent the group delay of the analysis filter bank. Similarly, the group delay of the low- and high-pass filters of the synthesis filter bank is the same (G_0=G_1=5.5), so you can use G to represent the group delay of the synthesis filter bank.

The following figure shows the filter bank system with the intermediate delay sums displayed below each path.



You can see from the previous figure that the signal delays on paths 1 and 2 are identical: $7(F+G)$. Because each path of the filter bank system has identical delay, you can equate the delay equations for paths 3 and 4 with the delay equation for paths 1 and 2. After constructing these equations, you can solve for M and N, respectively:

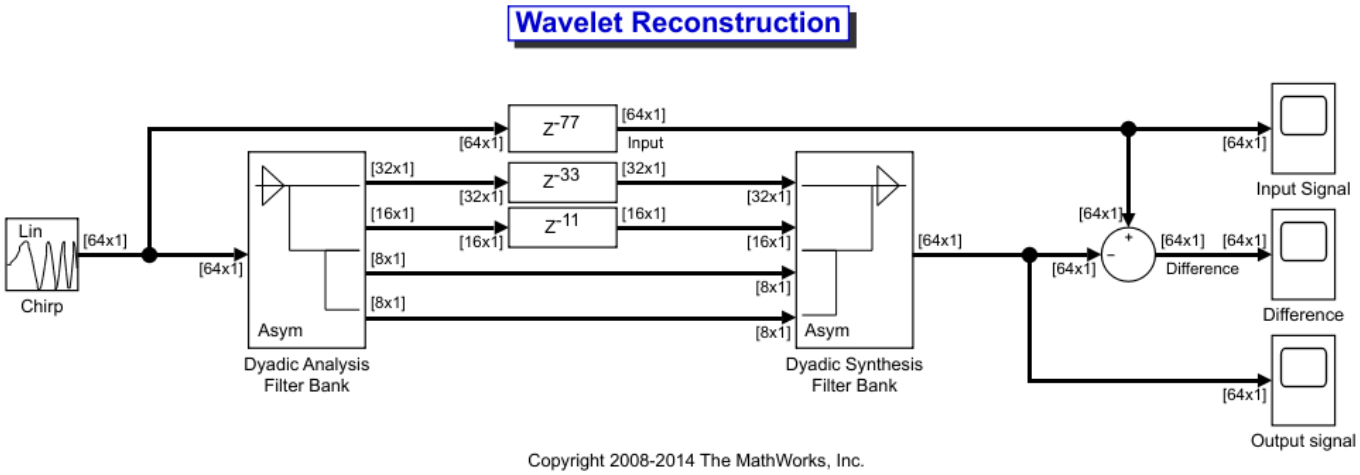
$$\text{Path 3} = \text{Path 1} \Rightarrow 4M + 3(F+G) = 7(F+G)$$
$$\Rightarrow M = F + G$$

$$\text{Path 4} = \text{Path 1} \Rightarrow 2N + (F+G) = 7(F+G)$$
$$\Rightarrow N = 3(F+G)$$

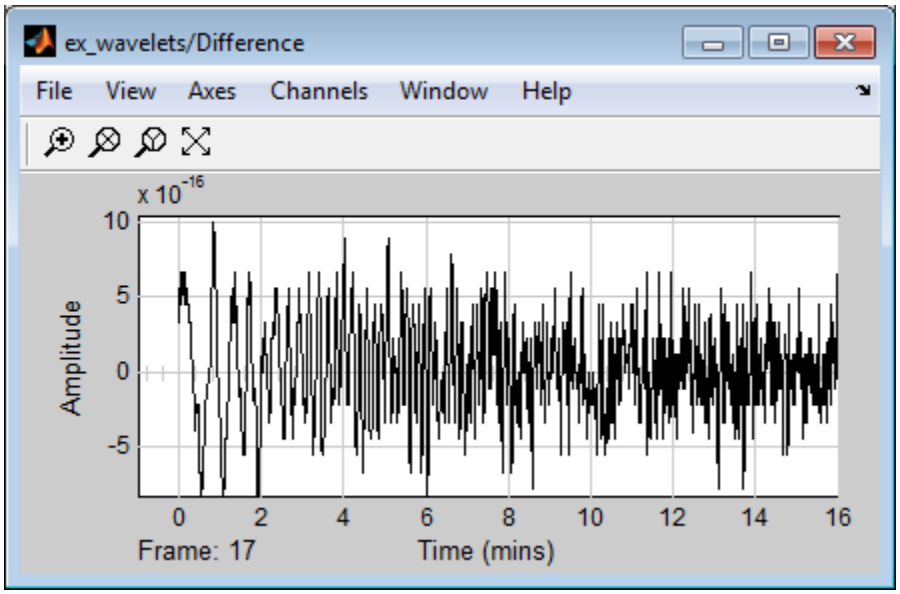
The `fvtool` analysis in step 1 found the group delay of each biorthogonal wavelet filter in this model to be 5.5 samples. Therefore, $F = 5.5$ and $G = 5.5$. By inserting these values into the two previous equations, you get $M = 11$ and $N = 33$. Because the total delay on each filter path must be the same, you can find the overall delay of the filter bank system by inserting $F = 5.5$ and $G = 5.5$ into the delay equation for any of the four filter paths. Inserting the values of F and G into $7(F+G)$ yields an overall delay of 77 samples for the filter bank system of the `ex_wavelets` model.

Update and Run the Model

Now that you know the latencies required for perfect wavelet reconstruction, you can incorporate those delay values into the model. The `ex_wavelets` model has already been updated with the correct delay values ($M = 11$, $N = 33$, *Overall* = 77), so it is ready to run.



After you run the model, examine the reconstruction error in the Difference scope. To further examine any particular areas of interest, use the zoom tools available on the toolbar of the scope window or from the **View** menu.



References

[1] Strang, G. and Nguyen, T. *Wavelets and Filter Banks*. Wellesley, MA: Wellesley-Cambridge Press, 1996.

Estimate the Power Spectrum in MATLAB

In this section...
“Estimate the Power Spectrum Using dsp.SpectrumAnalyzer” on page 16-14
“Convert the Power Between Units” on page 16-21
“Estimate the Power Spectrum Using dsp.SpectrumEstimator” on page 16-23

The power spectrum (PS) of a time-domain signal is the distribution of power contained within the signal over frequency, based on a finite set of data. The frequency-domain representation of the signal is often easier to analyze than the time-domain representation. Many signal processing applications, such as noise cancellation and system identification, are based on the frequency-specific modifications of signals. The goal of the power spectral estimation is to estimate the power spectrum of a signal from a sequence of time samples. Depending on what is known about the signal, estimation techniques can involve parametric or nonparametric approaches and can be based on time-domain or frequency-domain analysis. For example, a common parametric technique involves fitting the observations to an autoregressive model. A common nonparametric technique is the periodogram. The power spectrum is estimated using Fourier transform methods such as the Welch method and the filter bank method. For signals with relatively small length, the filter bank approach produces a spectral estimate with a higher resolution, a more accurate noise floor, and peaks more precise than the Welch method, with low or no spectral leakage. These advantages come at the expense of increased computation and slower tracking. For more details on these methods, see “Spectral Analysis” on page 16-59. You can also use other techniques such as the maximum entropy method.

In MATLAB, you can perform real-time spectral analysis of a dynamic signal using the `dsp.SpectrumAnalyzer` System object. You can view the spectral data in the spectrum analyzer and store the data in a workspace variable using the `isNewDataReady` and `getSpectrumData` object functions. Alternately, you can use the `dsp.SpectrumEstimator` System object followed by `dsp.ArrayPlot` object to view the spectral data. The output of the `dsp.SpectrumEstimator` object is the spectral data. This data can be acquired for further processing.

Estimate the Power Spectrum Using dsp.SpectrumAnalyzer

To view the power spectrum of a signal, you can use the `dsp.SpectrumAnalyzer` System object™. You can change the dynamics of the input signal and see the effect those changes have on the power spectrum of the signal in real time.

Initialization

Initialize the sine wave source to generate the sine wave and the spectrum analyzer to show the power spectrum of the signal. The input sine wave has two frequencies: one at 1000 Hz and the other at 5000 Hz. Create two `dsp.SineWave` objects, one to generate the 1000 Hz sine wave and the other to generate the 5000 Hz sine wave.

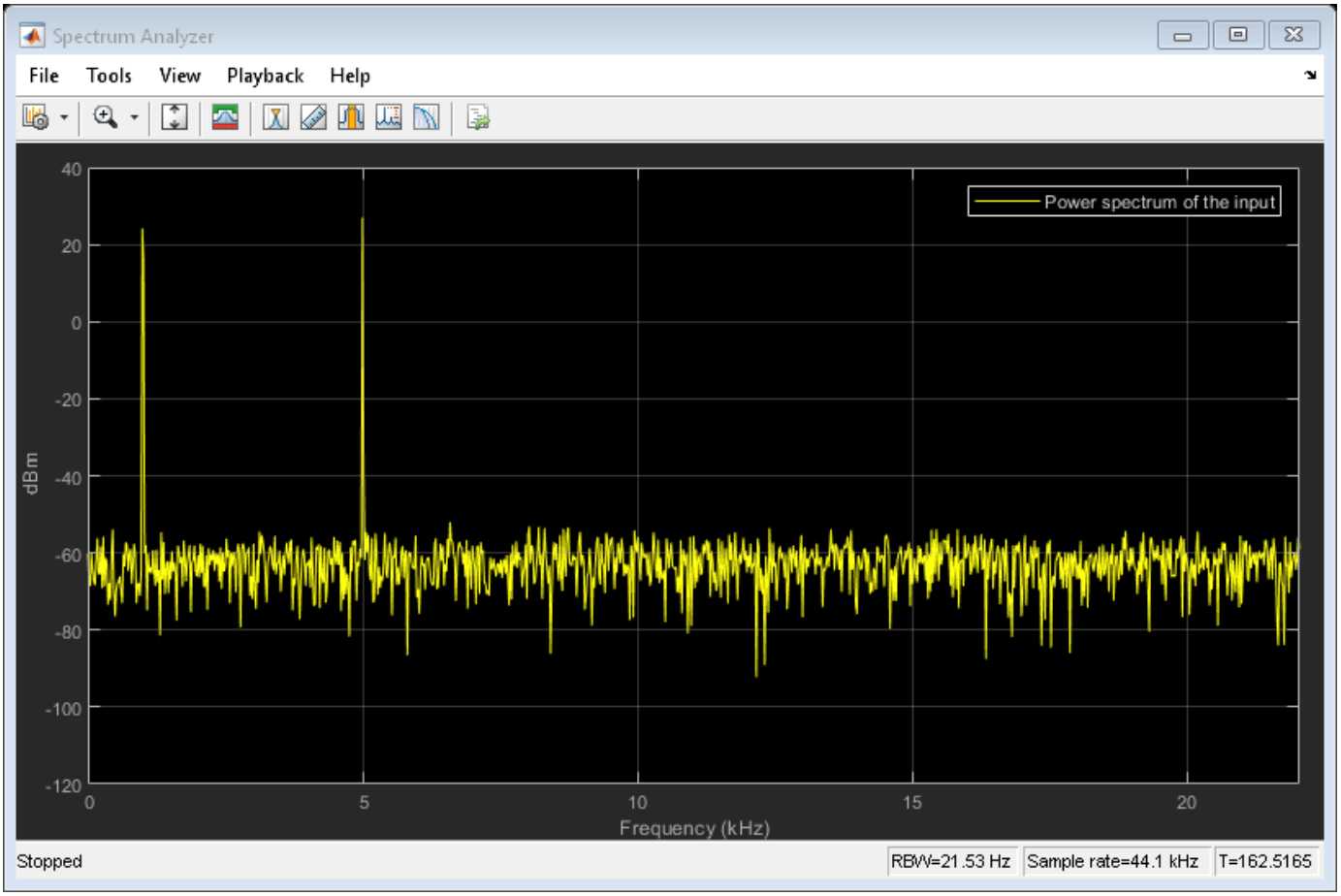
```
Fs = 44100;
Sineobject1 = dsp.SineWave('SamplesPerFrame',1024,'PhaseOffset',10,...
    'SampleRate',Fs,'Frequency',1000);
Sineobject2 = dsp.SineWave('SamplesPerFrame',1024,...
    'SampleRate',Fs,'Frequency',5000);
SA = dsp.SpectrumAnalyzer('SampleRate',Fs,'Method','Filter bank',...
    'SpectrumType','Power','PlotAsTwoSidedSpectrum',false,...
    'ChannelNames',{'Power spectrum of the input'},'YLimits',[-120 40],'ShowLegend',true);
```

The spectrum analyzer uses the filter bank approach to compute the power spectrum of the signal.

Estimation

Stream in and estimate the power spectrum of the signal. Construct a `for`-loop to run for 5000 iterations. In each iteration, stream in 1024 samples (one frame) of each sine wave and compute the power spectrum of each frame. To generate the input signal, add the two sine waves. The resultant signal is a sine wave with two frequencies: one at 1000 Hz and the other at 5000 Hz. Add Gaussian noise with zero mean and a standard deviation of 0.001. To acquire the spectral data for further processing, use the `isNewDataReady` and the `getSpectrumData` object functions. The variable `data` contains the spectral data that is displayed on the spectrum analyzer along with additional statistics about the spectrum.

```
data = [];
for Iter = 1:7000
    Sinewave1 = Sineobject1();
    Sinewave2 = Sineobject2();
    Input = Sinewave1 + Sinewave2;
    NoisyInput = Input + 0.001*randn(1024,1);
    SA(NoisyInput);
     if SA.isNewDataReady
        data = [data;getSpectrumData(SA)];
     end
end
release(SA);
```



In the spectrum analyzer output, you can see two distinct peaks: one at 1000 Hz and the other at 5000 Hz.

Resolution Bandwidth (RBW) is the minimum frequency bandwidth that can be resolved by the spectrum analyzer. By default, the `RBWSource` property of the `dsp.SpectrumAnalyzer` object is set to `Auto`. In this mode, RBW is the ratio of the frequency span to 1024. In a two-sided spectrum, this value is $\frac{F_s}{1024}$, while in a one-sided spectrum, it is $\frac{\frac{F_s}{2}}{1024}$. The spectrum analyzer in this example shows a one-sided spectrum. Hence, RBW is (44100/2)/1024 or 21.53Hz

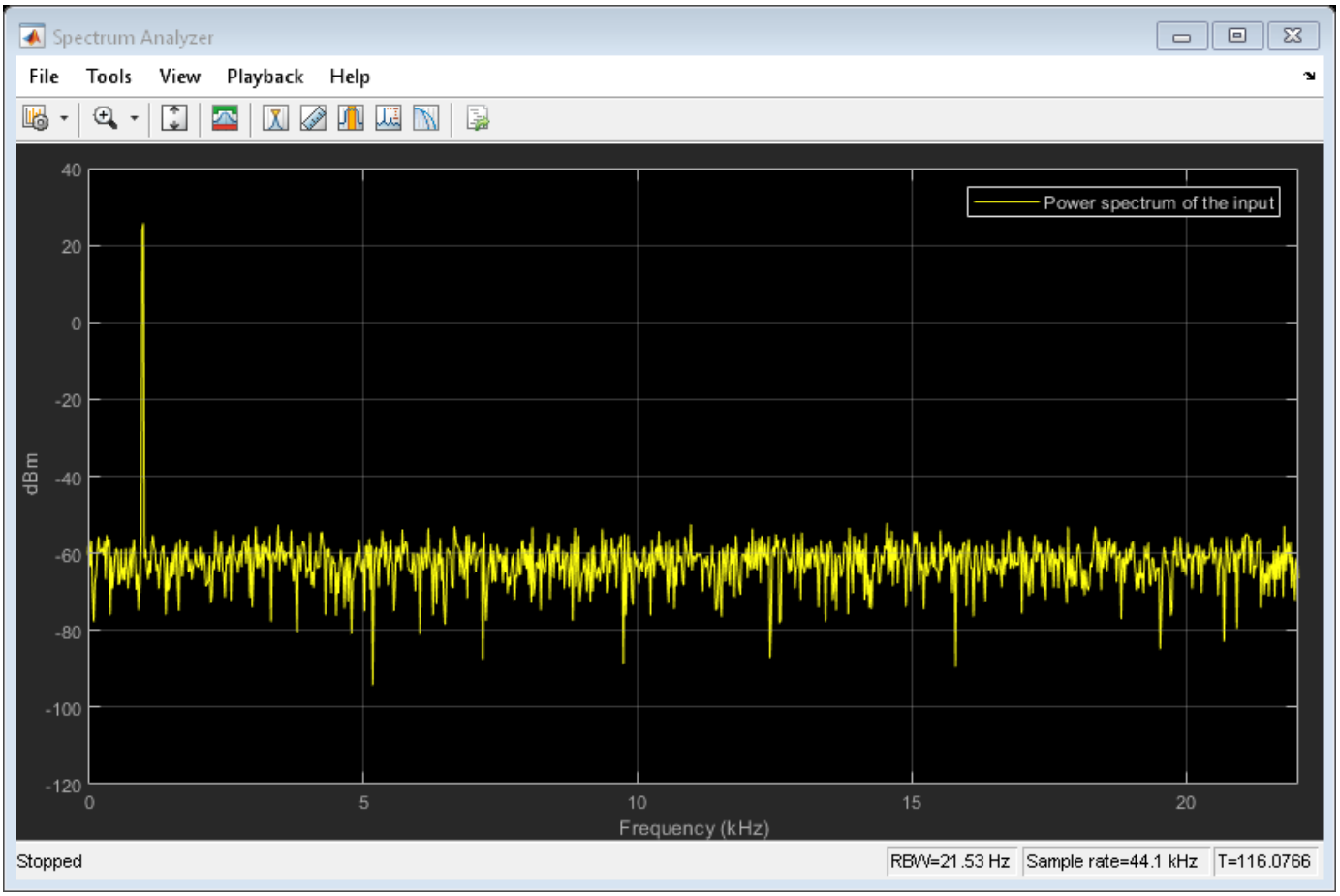
Using this value of RBW, the number of input samples required to compute one spectral update, $N_{samples}$ is given by the following equation: $N_{samples} = \frac{F_s}{RBW}$.

In this example, $N_{samples}$ is 44100/21.53 or 2048 samples.

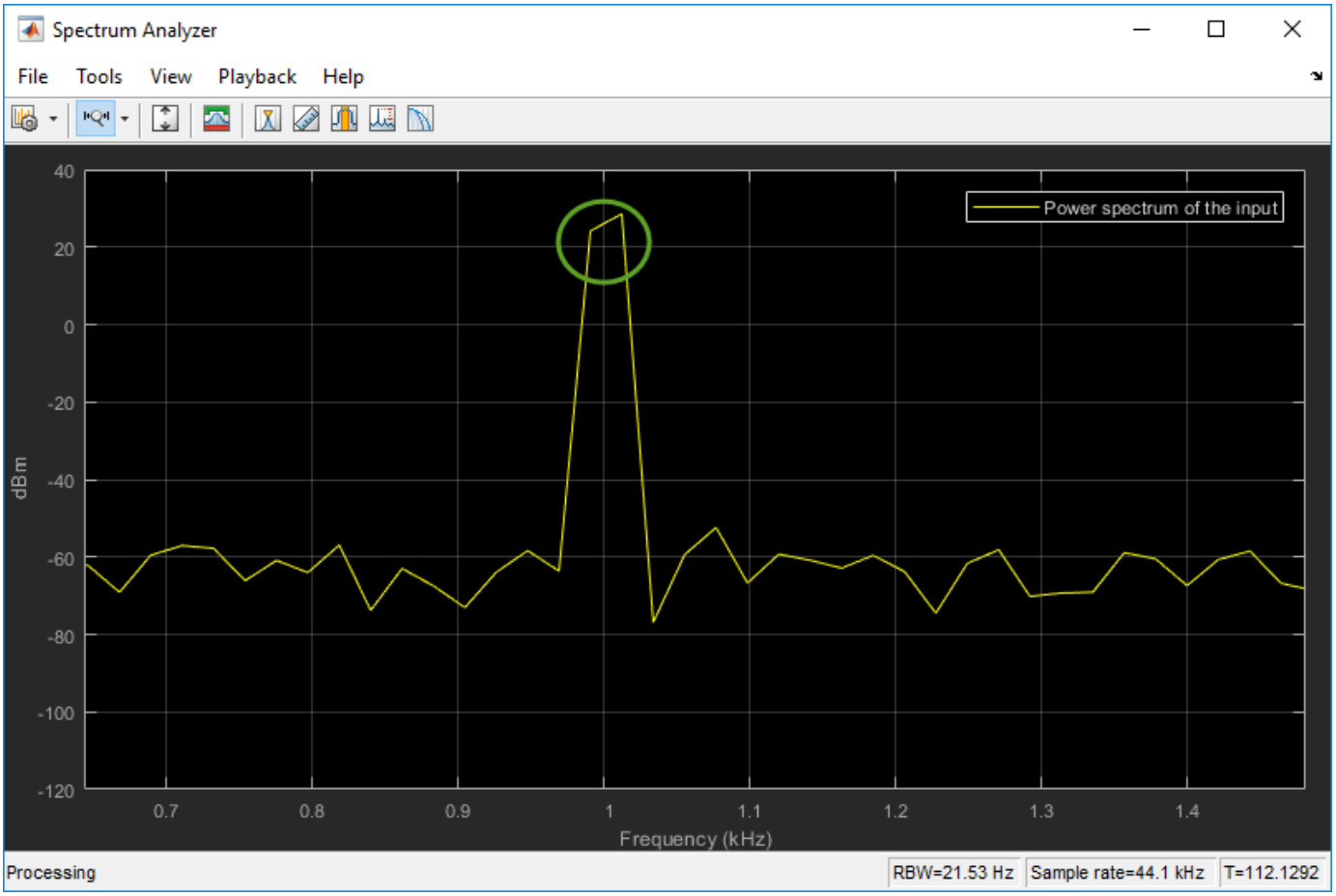
RBW calculated in the 'Auto' mode gives a good frequency resolution.

To distinguish between two frequencies in the display, the distance between the two frequencies must be at least RBW. In this example, the distance between the two peaks is 4000 Hz, which is greater than RBW. Hence, you can see the peaks distinctly. Change the frequency of the second sine wave to 1015 Hz. The difference between the two frequencies is less than RBW.

```
release(Sineobject2);
Sineobject2.Frequency = 1015;
for Iter = 1:5000
    Sinewave1 = Sineobject1();
    Sinewave2 = Sineobject2();
    Input = Sinewave1 + Sinewave2;
    NoisyInput = Input + 0.001*randn(1024,1);
    SA(NoisyInput);
end
release(SA);
```

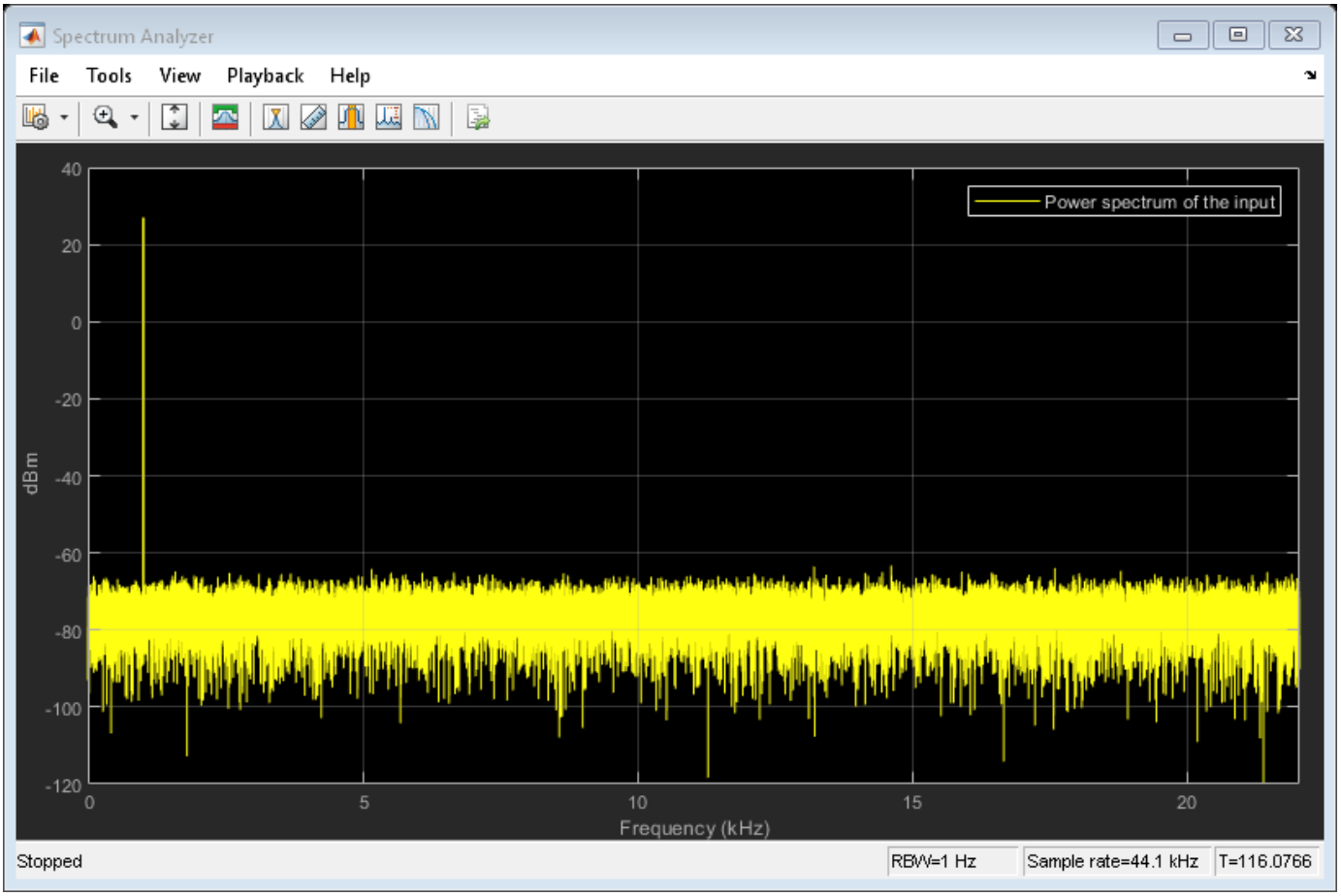


The peaks are not distinguishable.

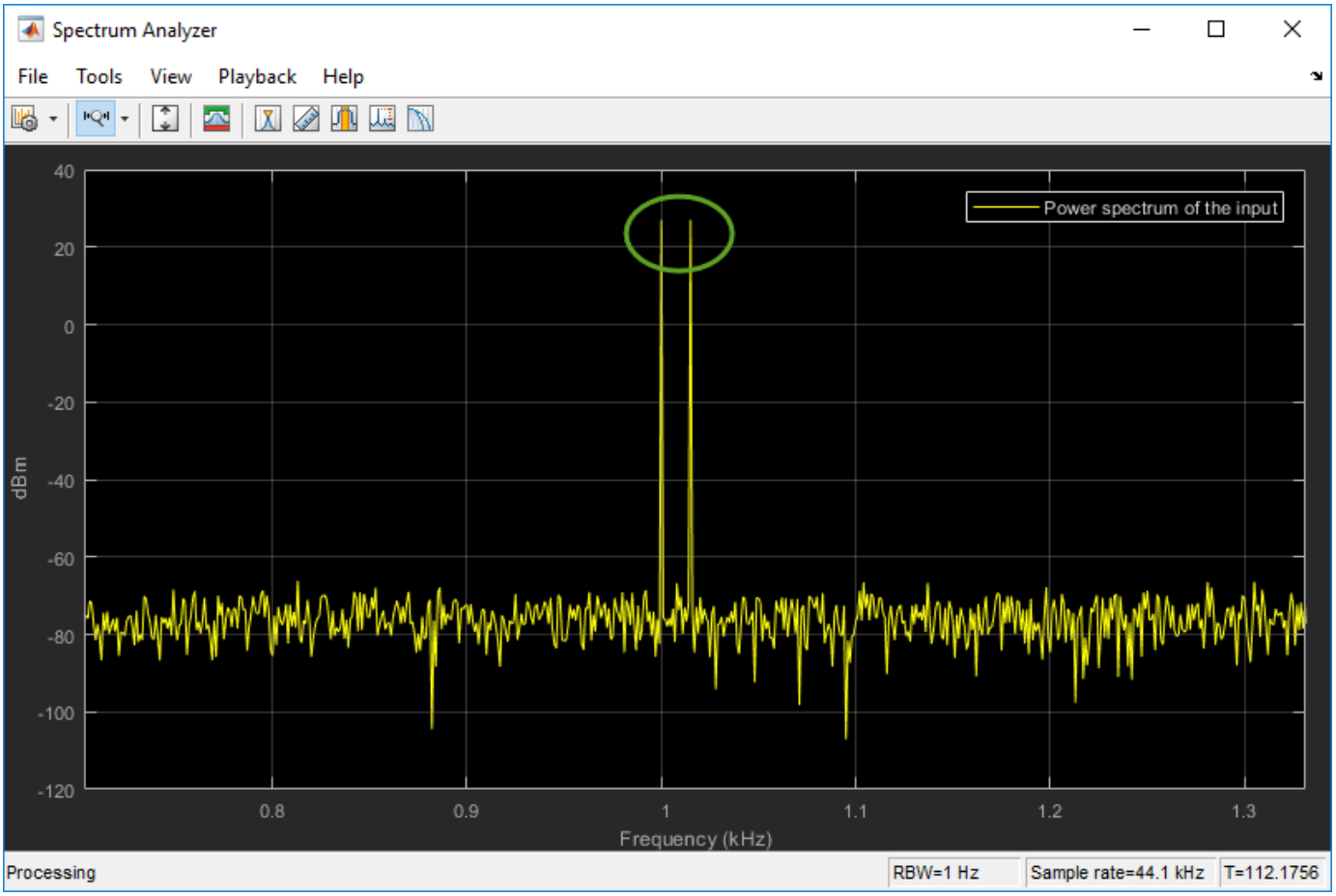


To increase the frequency resolution, decrease RBW to 1 Hz.

```
SA.RBWSource = 'property';
SA.RBW = 1;
for Iter = 1:5000
    Sinewave1 = Sineobject1();
    Sinewave2 = Sineobject2();
    Input = Sinewave1 + Sinewave2;
    NoisyInput = Input + 0.001*randn(1024,1);
    SA(NoisyInput);
end
release(SA);
```



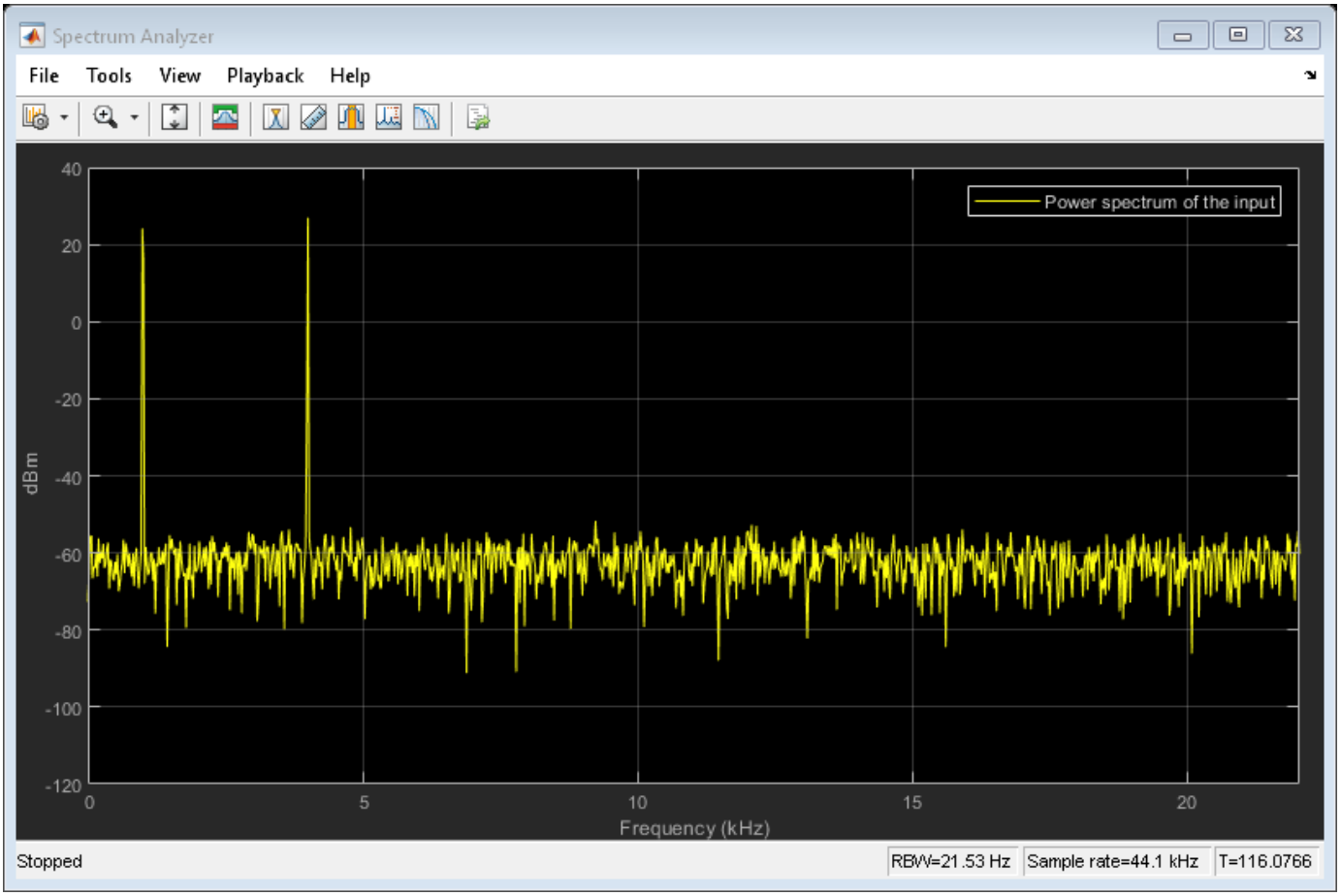
On zooming, the two peaks, which are 15 Hz apart, are now distinguishable.



When you increase the frequency resolution, the time resolution decreases. To maintain a good balance between the frequency resolution and time resolution, change the `RBWSource` property to `Auto`.

During streaming, you can change the input properties or the spectrum analyzer properties and see the effect on the spectrum analyzer output immediately. For example, change the frequency of the second sine wave when the index of the loop is a multiple of 1000.

```
release(Sineobject2);
SA.RBWSource = 'Auto';
for Iter = 1:5000
    Sinewave1 = Sineobject1();
    if (mod(Iter,1000) == 0)
        release(Sineobject2);
        Sineobject2.Frequency = Iter;
        Sinewave2 = Sineobject2();
    else
        Sinewave2 = Sineobject2();
    end
    Input = Sinewave1 + Sinewave2;
    NoisyInput = Input + 0.001*randn(1024,1);
    SA(NoisyInput);
end
release(SA);
```



While running the streaming loop, you can see that the peak of the second sine wave changes according to the iteration value. Similarly, you can change any of the spectrum analyzer properties while the simulation is running and see a corresponding change in the output.

Convert the Power Between Units

The spectrum analyzer provides three units to specify the power spectral density: `Watts/Hz`, `dBm/Hz`, and `dBW/Hz`. Corresponding units of power are `Watts`, `dBm`, and `dBW`. For electrical engineering applications, you can also view the RMS of your signal in `Vrms` or `dBV`. The default spectrum type is **Power** in `dBm`.

Convert the Power in Watts to dBW and dBm

Power in `dBW` is given by:

$$P_{dBW} = 10\log10(power\ in\ watt/1\ watt)$$

Power in `dBm` is given by:

$$P_{dBm} = 10\log10(power\ in\ watt/1\ milliwatt)$$

For a sine wave signal with an amplitude of 1 V, the power of a one-sided spectrum in `Watts` is given by:

$$P_{Watts} = A^2/2$$
$$P_{Watts} = 1/2$$

In this example, this power equals 0.5 W. Corresponding power in dBm is given by:

$$P_{dBm} = 10\log10(power\ in\ watt/1\ milliwatt)$$
$$P_{dBm} = 10\log10(0.5/10^{-3})$$

Here, the power equals 26.9897 dBm. To confirm this value with a peak finder, click **Tools** > **Measurements** > **Peak Finder**.

For a white noise signal, the spectrum is flat for all frequencies. The spectrum analyzer in this example shows a one-sided spectrum in the range [0 Fs/2]. For a white noise signal with a variance of 1e-4, the power per unit bandwidth ($P_{unitbandwidth}$) is 1e-4. The total power of white noise in **watts** over the entire frequency range is given by:

$$P_{whitenoise} = P_{unitbandwidth} * number\ of\ frequency\ bins,$$
$$P_{whitenoise} = (10^{-4}) * \left(\frac{Fs/2}{RBW}\right),$$
$$P_{whitenoise} = (10^{-4}) * \left(\frac{22050}{21.53}\right)$$

The number of frequency bins is the ratio of total bandwidth to RBW. For a one-sided spectrum, the total bandwidth is half the sampling rate. RBW in this example is 21.53 Hz. With these values, the total power of white noise in **watts** is 0.1024 W. In dBm, the power of white noise can be calculated using 10*log10(0.1024/10^-3), which equals 20.103 dBm.

Convert Power in Watts to dBFS

If you set the spectral units to `dBFS` and set the full scale (`FullScaleSource`) to `Auto`, power in `dBFS` is computed as:

$$P_{dBFS} = 20 \cdot \log_{10}(\sqrt{P_{watts}}/Full_Scale)$$

where:

- P_{watts} is the power in watts
- For double and float signals, *Full_Scale* is the maximum value of the input signal.
- For fixed point or integer signals, *Full_Scale* is the maximum value that can be represented.

If you specify a manual full scale (set `FullScaleSource` to `Property`), power in `dBFS` is given by:

$$P_{FS} = 20 \cdot \log_{10}(\sqrt{P_{watts}}/FS)$$

Where `FS` is the full scaling factor specified in the `FullScale` property.

For a sine wave signal with an amplitude of 1 V, the power of a one-sided spectrum in `Watts` is given by:

$$P_{Watts} = A^2/2$$
$$P_{Watts} = 1/2$$

In this example, this power equals 0.5 W and the maximum input signal for a sine wave is 1 V. The corresponding power in dBFS is given by:

$$P_{FS} = 20 \cdot \log_{10}(\sqrt{1/2}/1)$$

Here, the power equals -3.0103. To confirm this value in the spectrum analyzer, run these commands:

```
Fs = 1000;  % Sampling frequency
sinef = dsp.SineWave('SampleRate',Fs,'SamplesPerFrame',100);
scope = dsp.SpectrumAnalyzer('SampleRate',Fs,...
   'SpectrumUnits','dBFS','PlotAsTwoSidedSpectrum',false)
%%
for ii = 1:100000
xsine = sinef();
scope(xsine)
end
```

Then, click **Tools** > **Measurements** > **Peak Finder**.

Convert the Power in dBm to RMS in Vrms

Power in `dBm` is given by:

$$P_{dBm} = 10\log10(power\ in\ watt/1\ milliwatt)$$

Voltage in RMS is given by:

$$V_{rms} = 10^{P_{dBm}/20}\sqrt{10^{-3}}$$

From the previous example, P_{dBm} equals 26.9897 dBm. The V_{rms} is calculated as

$$V_{rms} = 10^{26.9897/20}\sqrt{0.001}$$

which equals 0.7071.

To confirm this value:

1. Change **Type** to `RMS`.
2. Open the peak finder by clicking **Tools** > **Measurements** > **Peak Finder**.

Estimate the Power Spectrum Using dsp.SpectrumEstimator

Alternately, you can compute the power spectrum of the signal using the `dsp.SpectrumEstimator` System object. You can acquire the output of the spectrum estimator and store the data for further processing. To view other objects in the `Estimation` library, type `help dsp` in the MATLAB® command prompt, and click `Estimation`.

Initialization

Use the same source as in the previous section on using the `dsp.SpectrumAnalyzer` to estimate the power spectrum. The input sine wave has two frequencies: one at 1000 Hz and the other at 5000 Hz. Initialize `dsp.SpectrumEstimator` to compute the power spectrum of the signal using the filter bank approach. View the power spectrum of the signal using the `dsp.ArrayPlot` object.

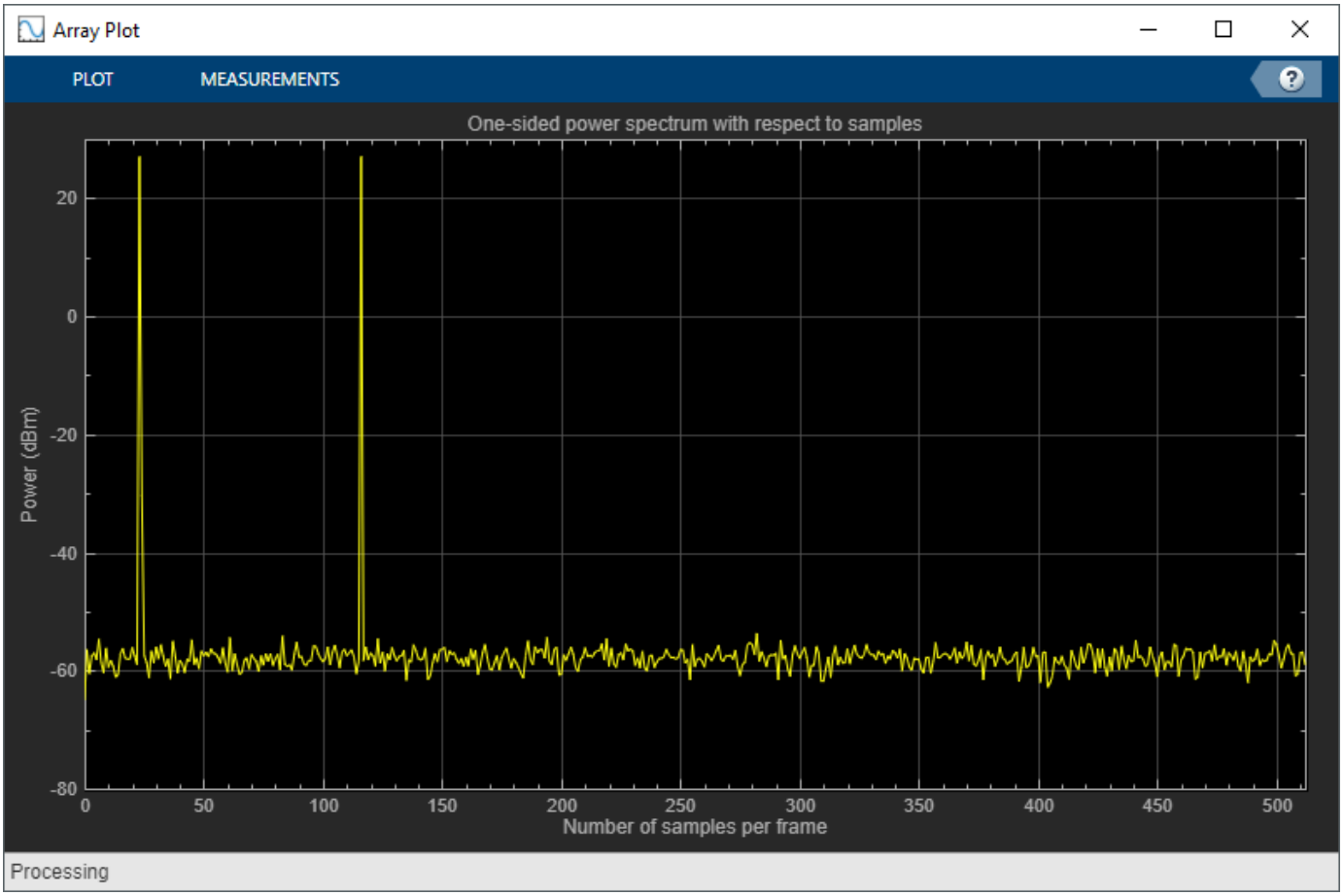
```
Fs = 44100;
Sineobject1 = dsp.SineWave('SamplesPerFrame',1024,'PhaseOffset',10,...
```

```
    'SampleRate',Fs,'Frequency',1000);
Sineobject2 = dsp.SineWave('SamplesPerFrame',1024,...
    'SampleRate',Fs,'Frequency',5000);

SpecEst = dsp.SpectrumEstimator('Method','Filter bank',...
    'PowerUnits','dBm','SampleRate',Fs,'FrequencyRange','onesided');
ArrPlot = dsp.ArrayPlot('PlotType','Line','ChannelNames',{'Power spectrum of the input'},...
    'YLimits',[-80 30],'XLabel','Number of samples per frame','YLabel',...
    'Power (dBm)','Title','One-sided power spectrum with respect to samples');
```

Estimation

Stream in and estimate the power spectrum of the signal. Construct a `for`-loop to run for 5000 iterations. In each iteration, stream in 1024 samples (one frame) of each sine wave and compute the power spectrum of each frame. Add Gaussian noise with mean at 0 and a standard deviation of 0.001 to the input signal.

```
for Iter = 1:5000
    Sinewave1 = Sineobject1();
    Sinewave2 = Sineobject2();
    Input = Sinewave1 + Sinewave2;
    NoisyInput = Input + 0.001*randn(1024,1);
    PSoutput = SpecEst(NoisyInput);
    ArrPlot(PSoutput);
end
```



Using the filter bank approach, the spectral estimate has a high resolution and the peaks are precise with no spectral leakage.

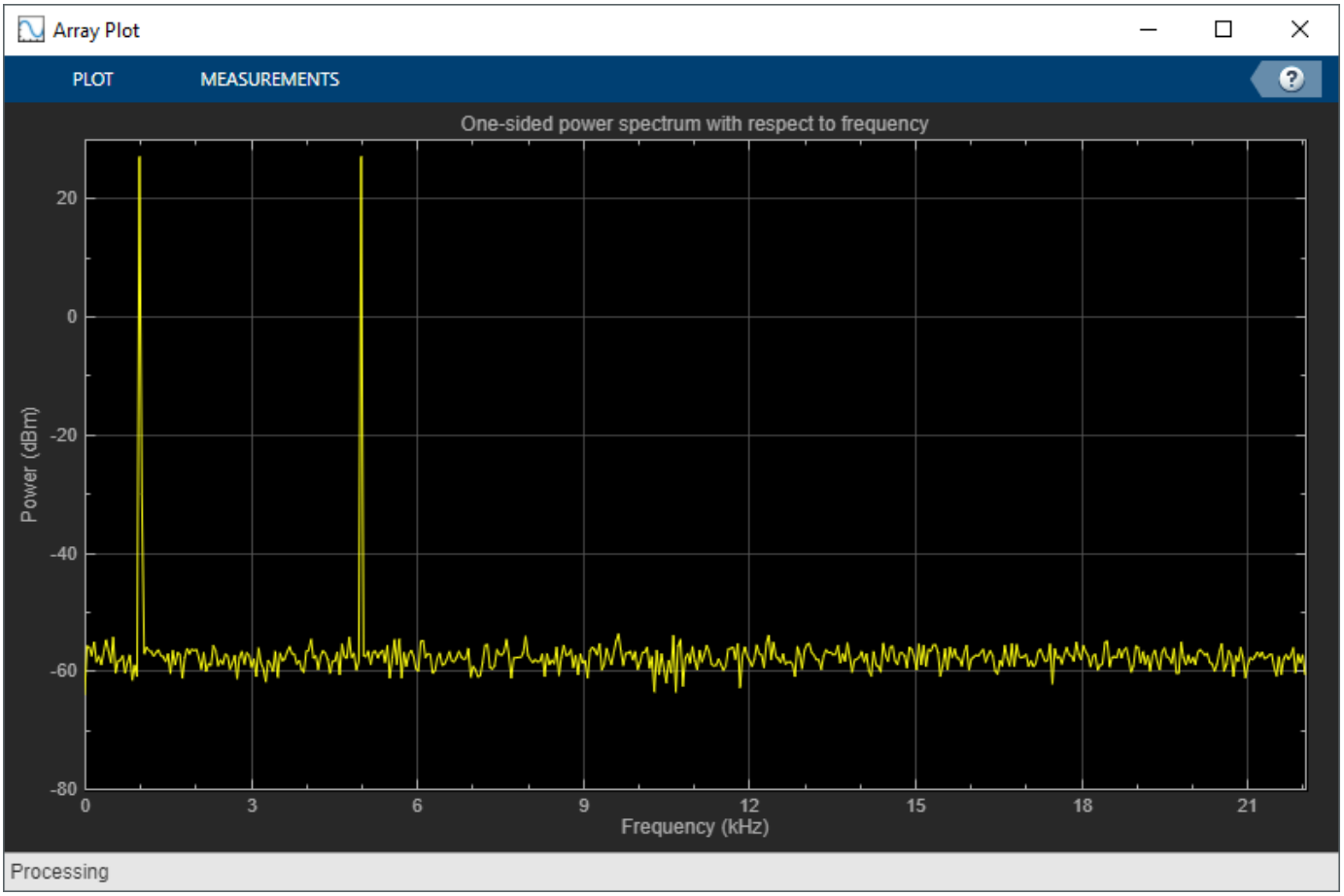
Convert *x*-axis to Represent Frequency

By default, the array plot shows the power spectral data with respect to the number of samples per frame. The number of points on the x-axis equals the length of the input frame. The spectrum analyzer plots the power spectral data with respect to frequency. For a one-sided spectrum, the frequency varies in the range [0 Fs/2]. For a two-sided spectrum, the frequency varies in the range [-Fs/2 Fs/2]. To convert the x-axis of the array plot from sample-based to frequency-based, do the following:

- Click on the **Configuration Properties** icon.
- For a one-sided spectrum - On **Main** tab, set **Sample increment** to $Fs/FrameLength$ and **X-offset** to 0.
- For a two-sided spectrum - On **Main** tab, set **Sample increment** to $Fs/FrameLength$ and **X-offset** to $-Fs/2$.

In this example, the spectrum is one-sided and hence, the **Sample increment** and **X-offset** are set to 44100/1024 and 0, respectively. To specify the frequency in kHz, set the **Sample increment** to 44.1/1024.

```
ArrPlot.SampleIncrement = (Fs/1000)/1024;
ArrPlot.XLabel = 'Frequency (kHz)';
ArrPlot.Title = 'One-sided power spectrum with respect to frequency';

for Iter = 1:5000
    Sinewave1 = Sineobject1();
    Sinewave2 = Sineobject2();
    Input = Sinewave1 + Sinewave2;
    NoisyInput = Input + 0.001*randn(1024,1);
    PSoutput = SpecEst(NoisyInput);
    ArrPlot(PSoutput);
end
```



Live Processing

The output of the `dsp.SpectrumEstimator` object contains the spectral data and is available for further processing. The data can be processed in real-time or it can be stored in the workspace.

See Also

More About

- “Estimate the Power Spectrum in Simulink” on page 16-27
- “Estimate the Transfer Function of an Unknown System” on page 16-43
- “View the Spectrogram Using Spectrum Analyzer” on page 16-50
- “Spectral Analysis” on page 16-59

Estimate the Power Spectrum in Simulink

In this section...
“Estimate the Power Spectrum Using the Spectrum Analyzer” on page 16-27
“Convert the Power Between Units” on page 16-36
“Estimate Power Spectrum Using the Spectrum Estimator Block” on page 16-38

The power spectrum (PS) of a time-domain signal is the distribution of power contained within the signal over frequency, based on a finite set of data. The frequency-domain representation of the signal is often easier to analyze than the time-domain representation. Many signal processing applications, such as noise cancellation and system identification, are based on the frequency-specific modifications of signals. The goal of the power spectral estimation is to estimate the power spectrum of a signal from a sequence of time samples. Depending on what is known about the signal, estimation techniques can involve parametric or nonparametric approaches and can be based on time-domain or frequency-domain analysis. For example, a common parametric technique involves fitting the observations to an autoregressive model. A common nonparametric technique is the periodogram. The power spectrum is estimated using Fourier transform methods such as the Welch method and the filter bank method. For signals with relatively small length, the filter bank approach produces a spectral estimate with a higher resolution, a more accurate noise floor, and peaks more precise than the Welch method, with low or no spectral leakage. These advantages come at the expense of increased computation and slower tracking. For more details on these methods, see “Spectral Analysis” on page 16-59. You can also use other techniques such as the maximum entropy method.

In Simulink, you can perform real-time spectral analysis of a dynamic signal using the Spectrum Analyzer block. You can view the spectral data in the spectrum analyzer. To acquire the last spectral data for further processing, create a `Spectrum Analyzer Configuration` object and run the `getSpectrumData` function on this object. Alternately, you can use the Spectrum Estimator block from the `dspspect3` library to compute the power spectrum, and Array Plot block to view the spectrum.

Estimate the Power Spectrum Using the Spectrum Analyzer

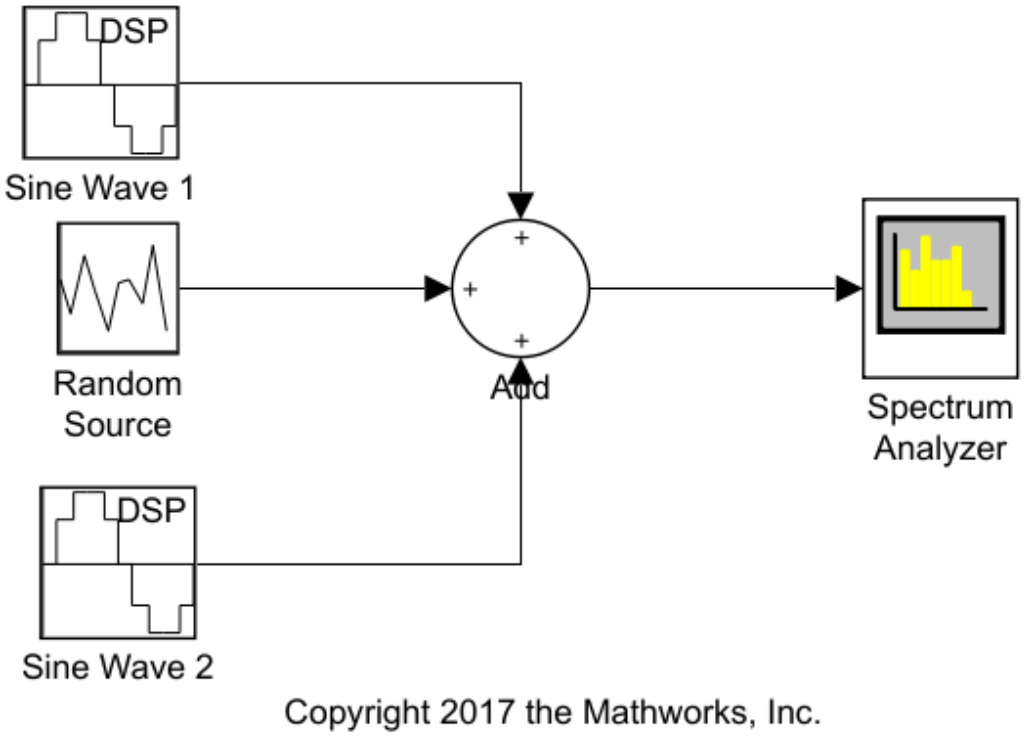
You can view the power spectrum (PS) of a signal using the Spectrum Analyzer block. The PS is computed in real time and varies with the input signal, and with changes in the properties of the Spectrum Analyzer block. You can change the dynamics of the input signal and see what effect those changes have on the spectrum of the signal in real time.

The model `ex_psd_sa` feeds a noisy sine wave signal to the Spectrum Analyzer block. The sine wave signal is a sum of two sinusoids: one at a frequency of 5000 Hz and the other at a frequency of 10,000 Hz. The noise at the input is Gaussian, with zero mean and a standard deviation of 0.01.

Open and Inspect the Model

To open the model, enter `ex_psd_sa` in the MATLAB command prompt.

Power Spectrum Estimation Using the Spectrum Analyzer

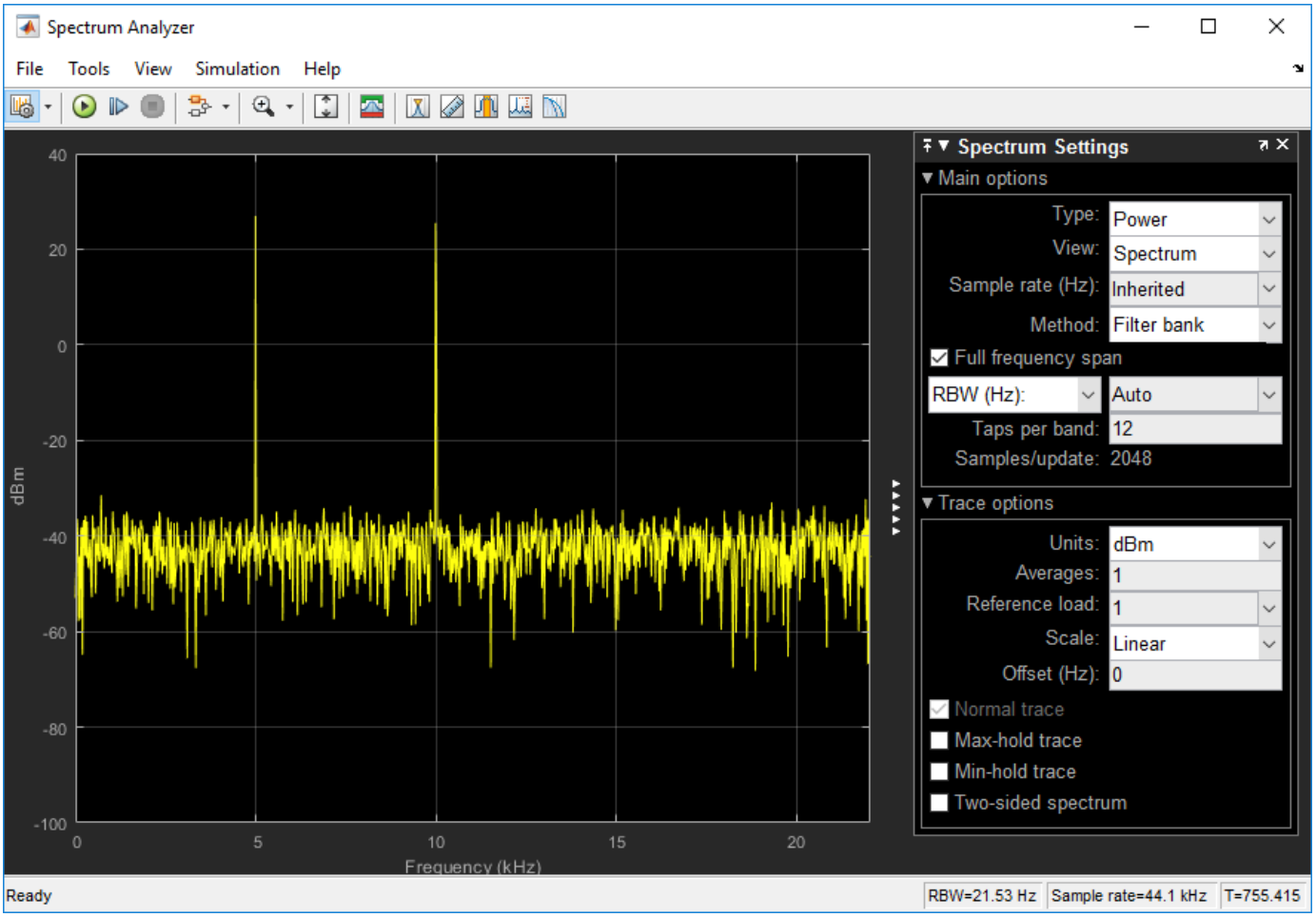


Here are the settings of the blocks in the model.

Block	Parameter Changes	Purpose of the block
Sine Wave 1	• **Frequency** to 5000 • **Sample time** to 1/44100 • **Samples per frame** to 1024	Sinusoid signal with frequency at 5000 Hz
Sine Wave 2	• **Frequency** to 10000 • **Phase offset (rad)** to 10 • **Sample time** to 1/44100 • **Samples per frame** to 1024	Sinusoid signal with frequency at 10000 Hz
Random Source	• **Source type** to `Gaussian` • **Variance** to 1e-4 • **Sample time** to 1/44100 • **Samples per frame** to 1024	Random Source block generates a random noise signal with properties specified through the block dialog box
Add	**List of signs** to +++.	Add block adds random noise to the input signal

Block	Parameter Changes	Purpose of the block
Spectrum Analyzer	Click the **Spectrum Settings** icon . A pane appears on the right. • In the **Main options** pane, under **Type**, select `Power`. Under **Method**, select `Filter bank`. • In the **Trace options** pane, clear the **Two-sided spectrum** check box. This shows only the real-half of the spectrum. • If needed, select the **Max-hold trace** and **Min-hold trace** check boxes. Click the **Configuration Properties** icon and set **Y-limits (Minimum)** as `-100` and **Y-limits (Maximum)** as `40`.	Spectrum Analyzer block shows the Power Spectrum Density of the signal

Play the model. Open the Spectrum Analyzer block to view the power spectrum of the sine wave signal. There are two tones at frequencies 5000 Hz and 10,000 Hz, which correspond to the two frequencies at the input.

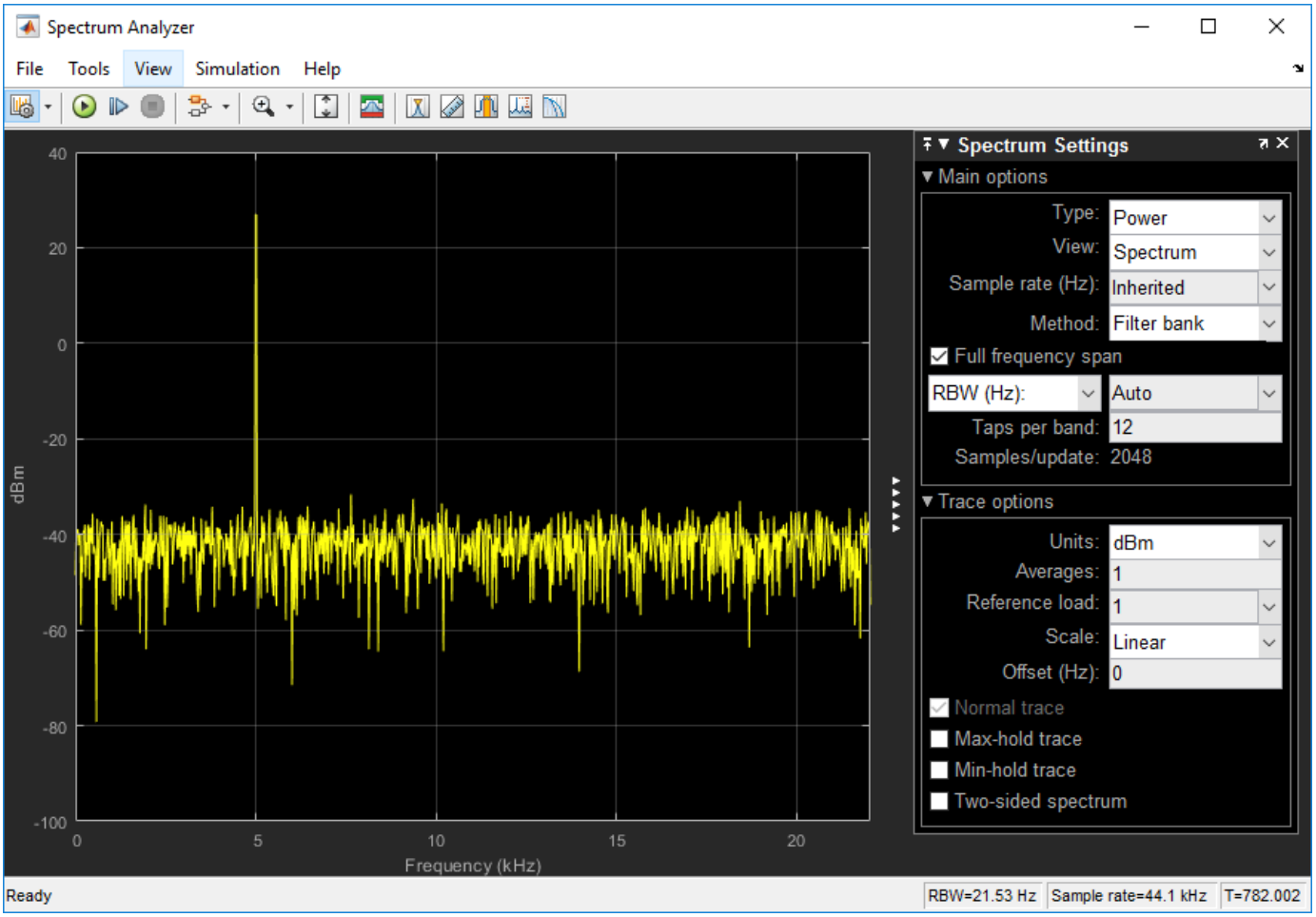


RBW, the resolution bandwidth is the minimum frequency bandwidth that can be resolved by the spectrum analyzer. By default, **RBW (Hz)** is set to `Auto`. In the `Auto` mode, RBW is the ratio of the frequency span to 1024. In a two-sided spectrum, this value is $F_s/1024$, while in a one-sided spectrum, it is $(F_s/2)/1024$. The spectrum analyzer in `ex_psd_sa` is configured to show one-sided spectrum. Hence, the RBW is (44100/2)/1024 or 21.53 Hz.

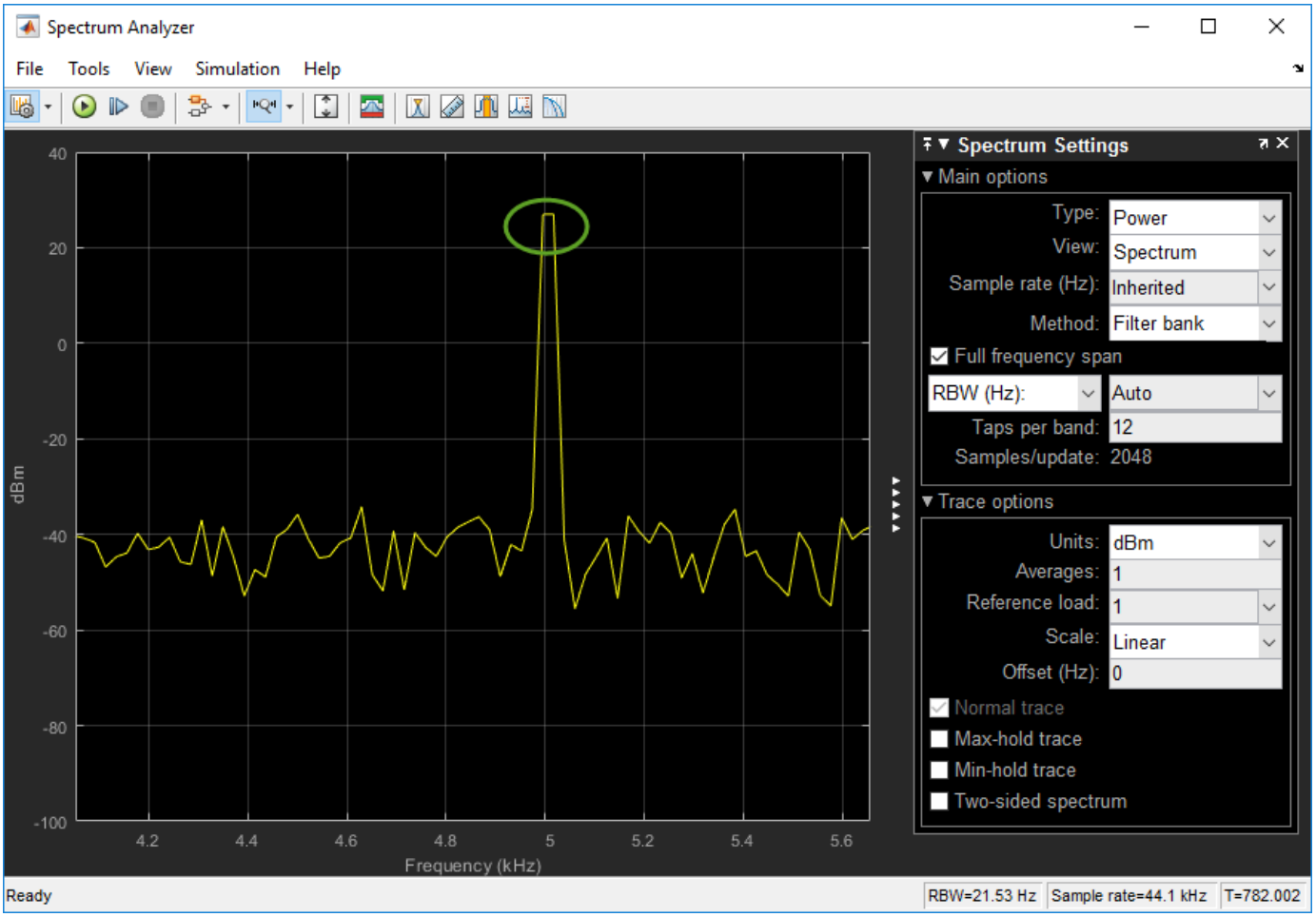
Using this value of RBW, the number of input samples used to compute one spectral update is given by $N_{samples}$ = Fs/RBW, which is 44100/21.53 or 2048.

RBW calculated in this mode gives a good frequency resolution.

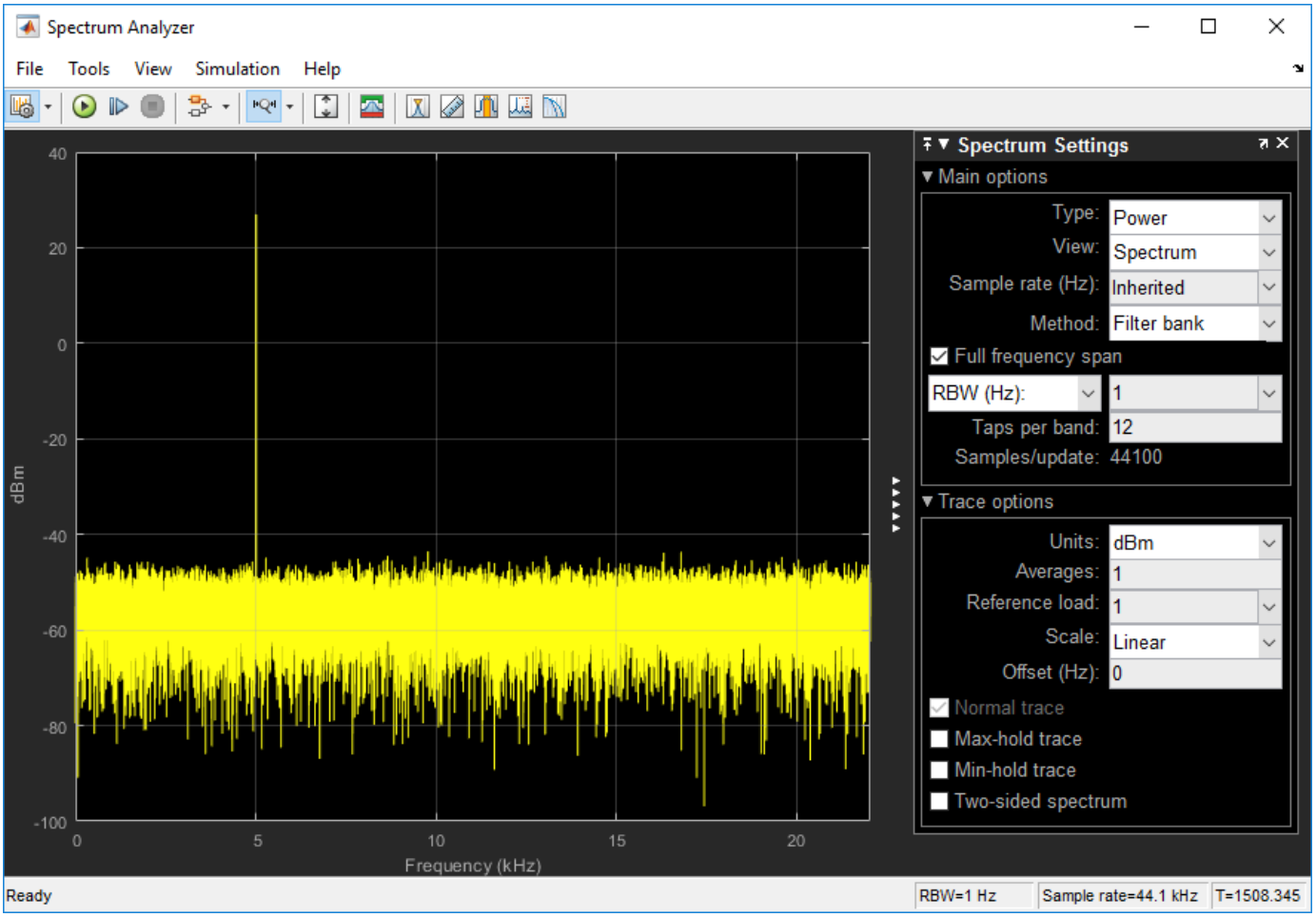
To distinguish between two frequencies in the display, the distance between the two frequencies must be at least RBW. In this example, the distance between the two peaks is 5000 Hz, which is greater than RBW. Hence, you can see the peaks distinctly. Change the frequency of the second sine wave from 10000 Hz to 5015 Hz. The difference between the two frequencies is less than *RBW*.



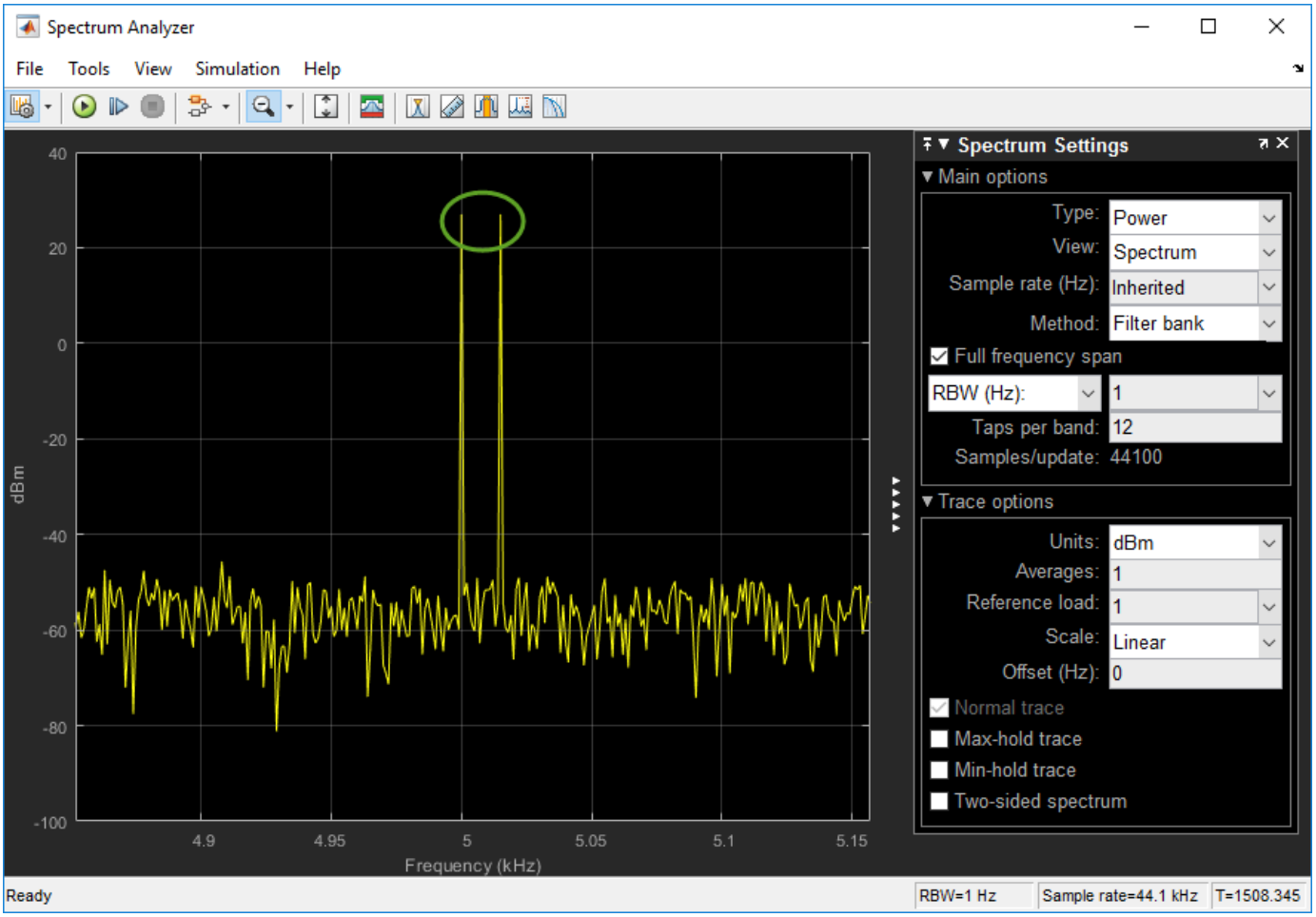
On zooming, you can see that the peaks are not distinguishable.



To increase the frequency resolution, decrease *RBW* to 1 Hz and run the simulation.



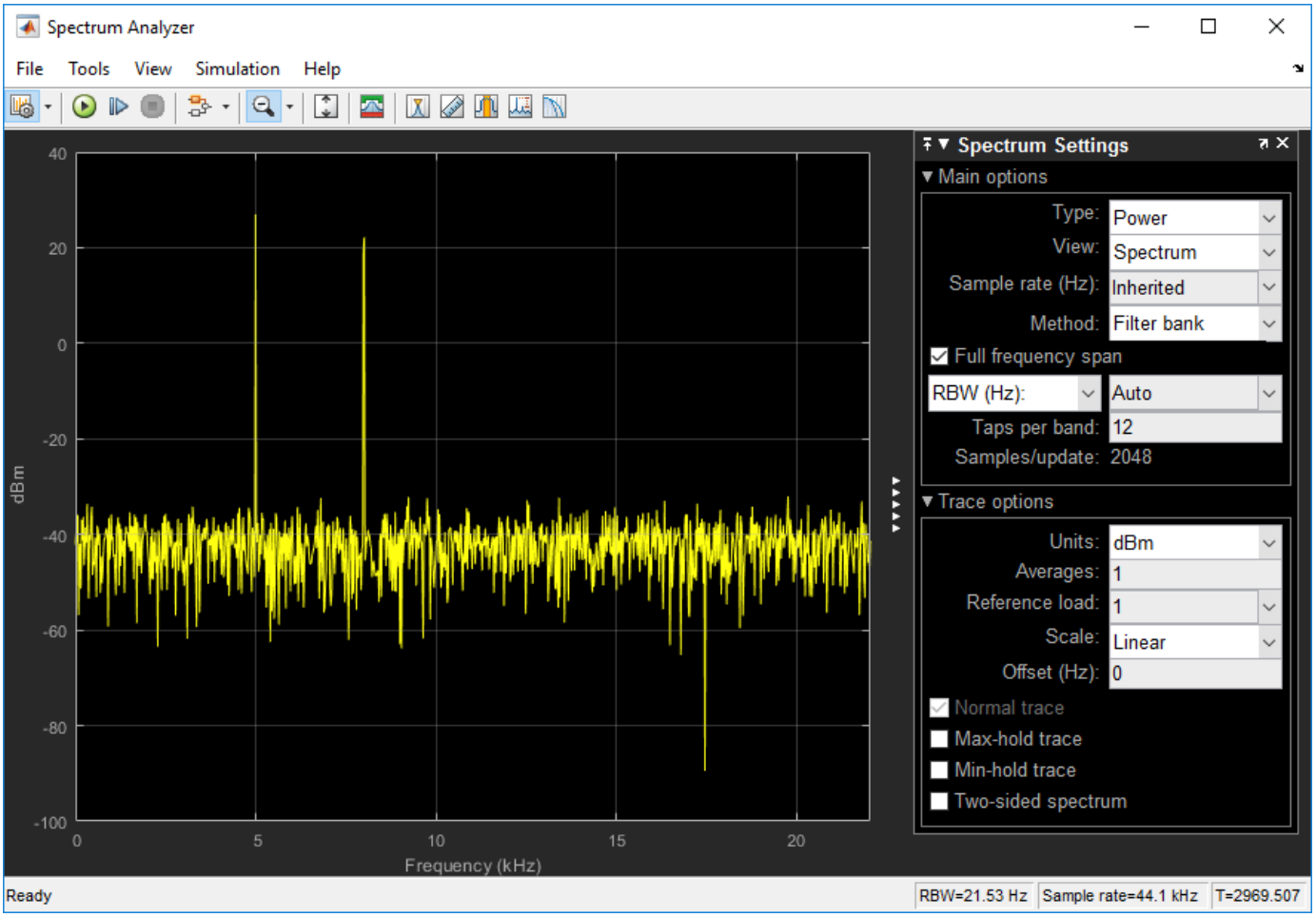
On zooming, the two peaks, which are 15 Hz apart, are now distinguishable



When you increase the frequency resolution, the time resolution decreases. To maintain a good balance between the frequency resolution and time resolution, change the **RBW (Hz)** to `Auto`.

Change the Input Signal

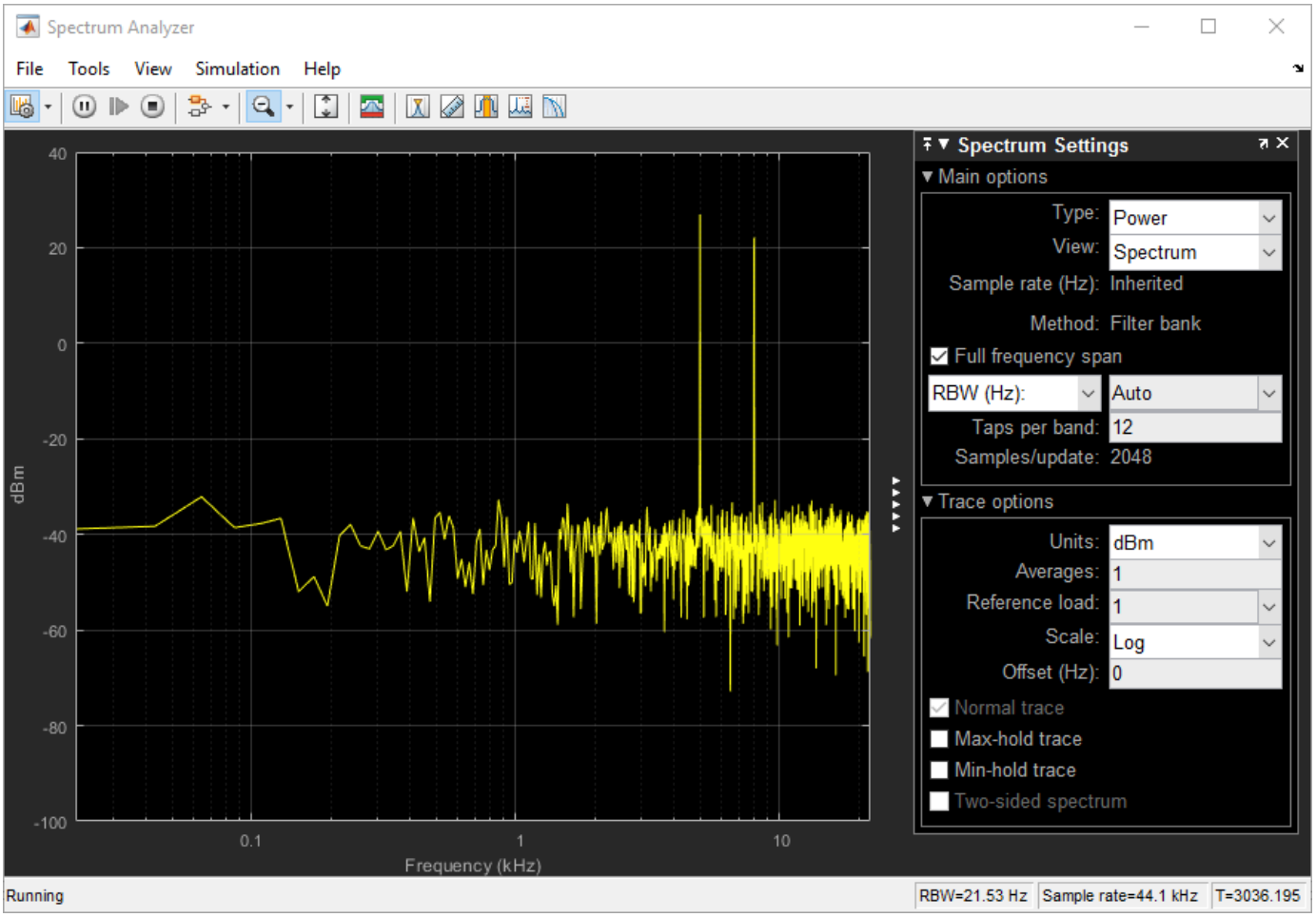
When you change the dynamics of the input signal during simulation, the power spectrum of the signal also changes in real time. While the simulation is running, change the **Frequency** of the Sine Wave 1 block to `8000` and click **Apply**. The second tone in the spectral analyzer output shifts to 8000 Hz and you can see the change in real time.



Change the Spectrum Analyzer Settings

When you change the settings in the Spectrum Analyzer block, the effect can be seen on the spectral data in real time.

When the model is running, in the **Trace** options pane of the Spectrum Analyzer block, change the **Scale** to `Log`. The PS is now displayed on a log scale.



For more information on how the Spectrum Analyzer settings affect the power spectrum data, see the 'Algorithms' section of the Spectrum Analyzer block reference page.

Convert the Power Between Units

The spectrum analyzer provides three units to specify the power spectral density: `Watts/Hz`, `dBm/Hz`, and `dBW/Hz`. Corresponding units of power are `Watts`, `dBm`, and `dBW`. For electrical engineering applications, you can also view the RMS of your signal in `Vrms` or `dBV`. The default spectrum type is **Power** in `dBm`.

Convert the Power in Watts to dBW and dBm

Power in `dBW` is given by:

$$P_{dBW} = 10\log10(power\ in\ watt/1\ watt)$$

Power in `dBm` is given by:

$$P_{dBm} = 10\log10(power\ in\ watt/1\ milliwatt)$$

For a sine wave signal with an amplitude of 1 V, the power of a one-sided spectrum in `Watts` is given by:

$$P_{Watts} = A^2/2$$
$$P_{Watts} = 1/2$$

In this example, this power equals 0.5 W. Corresponding power in dBm is given by:

$$P_{dBm} = 10\log10(power\ in\ watt/1\ milliwatt)$$
$$P_{dBm} = 10\log10(0.5/10^{-3})$$

Here, the power equals 26.9897 dBm. To confirm this value with a peak finder, click **Tools** > **Measurements** > **Peak Finder**.

For a white noise signal, the spectrum is flat for all frequencies. The spectrum analyzer in this example shows a one-sided spectrum in the range [0 Fs/2]. For a white noise signal with a variance of 1e-4, the power per unit bandwidth ($P_{unitbandwidth}$) is 1e-4. The total power of white noise in **watts** over the entire frequency range is given by:

$$P_{whitenoise} = P_{unitbandwidth} * number\ of\ frequency\ bins,$$
$$P_{whitenoise} = (10^{-4}) * \left(\frac{Fs/2}{RBW}\right),$$
$$P_{whitenoise} = (10^{-4}) * \left(\frac{22050}{21.53}\right)$$

The number of frequency bins is the ratio of total bandwidth to RBW. For a one-sided spectrum, the total bandwidth is half the sampling rate. RBW in this example is 21.53 Hz. With these values, the total power of white noise in **watts** is 0.1024 W. In dBm, the power of white noise can be calculated using 10*log10(0.1024/10^-3), which equals 20.103 dBm.

Convert Power in Watts to dBFS

If you set the spectral units to `dBFS` and set the full scale (`FullScaleSource`) to `Auto`, power in `dBFS` is computed as:

$$P_{dBFS} = 20 \cdot \log_{10}(\sqrt{P_{watts}}/Full_Scale)$$

where:

- P_{watts} is the power in watts
- For double and float signals, *Full_Scale* is the maximum value of the input signal.
- For fixed point or integer signals, *Full_Scale* is the maximum value that can be represented.

If you specify a manual full scale (set `FullScaleSource` to `Property`), power in `dBFS` is given by:

$$P_{FS} = 20 \cdot \log_{10}(\sqrt{P_{watts}}/FS)$$

Where `FS` is the full scaling factor specified in the `FullScale` property.

For a sine wave signal with an amplitude of 1 V, the power of a one-sided spectrum in `Watts` is given by:

$$P_{Watts} = A^2/2$$
$$P_{Watts} = 1/2$$

In this example, this power equals 0.5 W and the maximum input signal for a sine wave is 1 V. The corresponding power in dBFS is given by:

$$P_{FS} = 20 \cdot \log_{10}(\sqrt{1/2}/1)$$

Here, the power equals -3.0103. To confirm this value in the spectrum analyzer, run these commands:

```
Fs = 1000;  % Sampling frequency
sinef = dsp.SineWave('SampleRate',Fs,'SamplesPerFrame',100);
scope = dsp.SpectrumAnalyzer('SampleRate',Fs,...
   'SpectrumUnits','dBFS','PlotAsTwoSidedSpectrum',false)
%%
for ii = 1:100000
xsine = sinef();
scope(xsine)
end
```

Then, click **Tools** > **Measurements** > **Peak Finder**.

Convert the Power in dBm to RMS in Vrms

Power in `dBm` is given by:

$$P_{dBm} = 10\log10(power\ in\ watt/1\ milliwatt)$$

Voltage in RMS is given by:

$$V_{rms} = 10^{P_{dBm}/20}\sqrt{10^{-3}}$$

From the previous example, P_{dBm} equals 26.9897 dBm. The V_{rms} is calculated as

$$V_{rms} = 10^{26.9897/20}\sqrt{0.001}$$

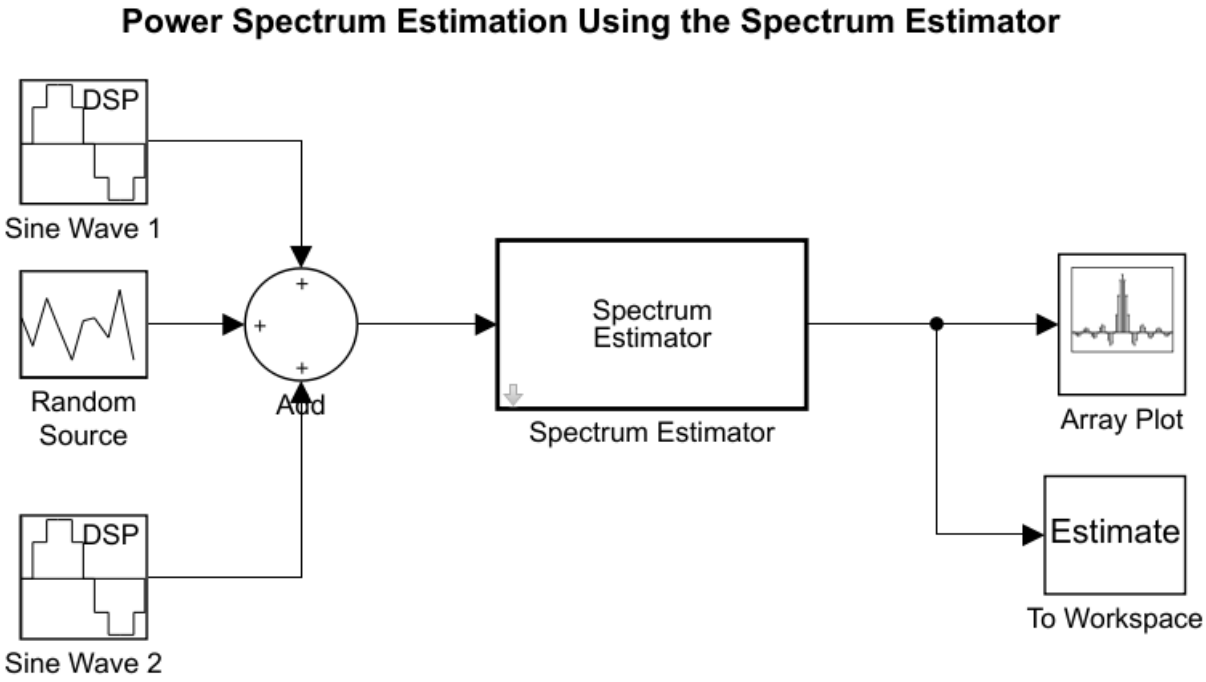
which equals 0.7071.

To confirm this value:

1. Change **Type** to `RMS`.
2. Open the peak finder by clicking **Tools** > **Measurements** > **Peak Finder**.

Estimate Power Spectrum Using the Spectrum Estimator Block

Alternately, you can compute the power spectrum of the signal using the Spectrum Estimator block in the `dspspect3` library. You can acquire the output of the spectrum estimator and store the data for further processing.

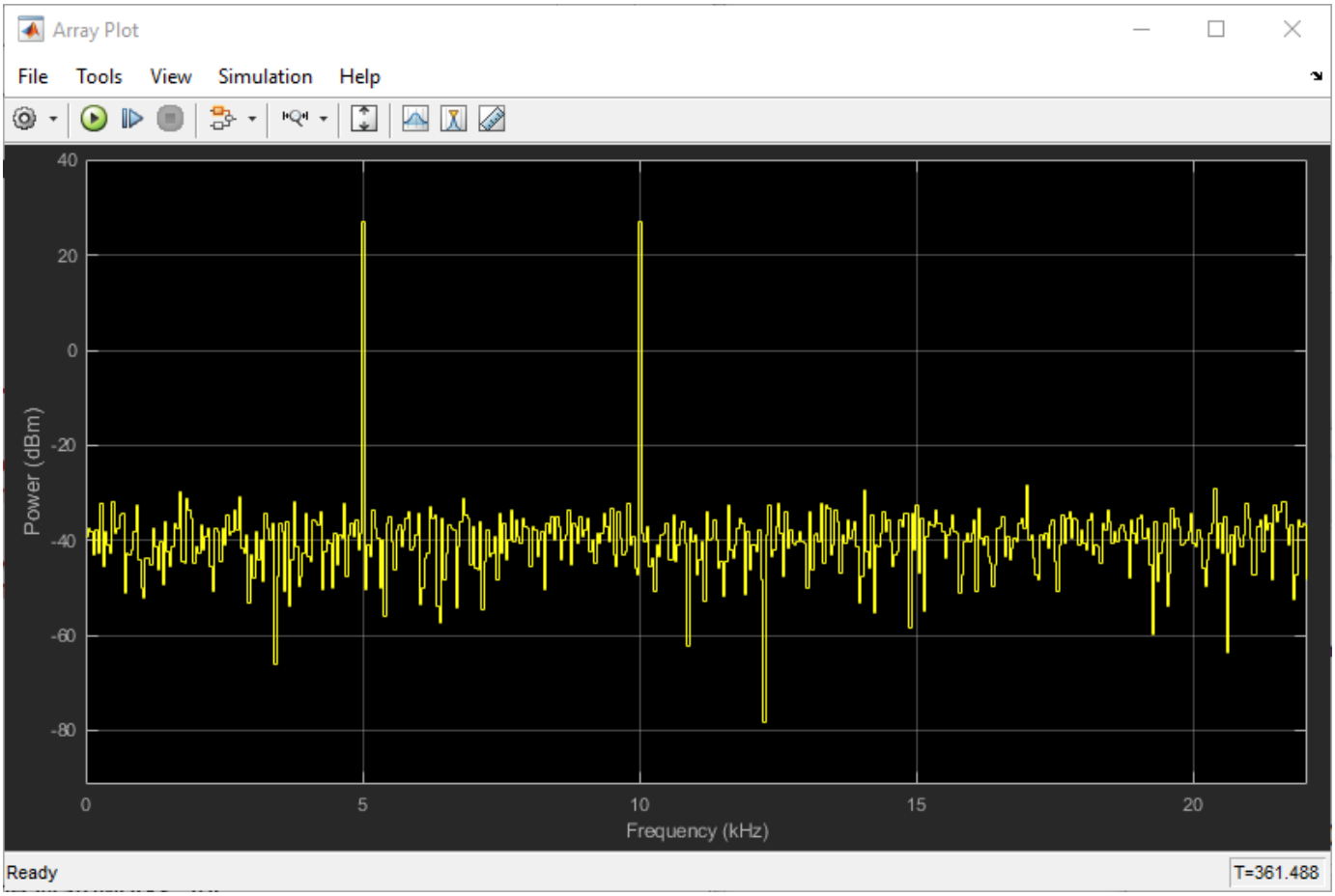


Replace the Spectrum Analyzer block in `ex_psd_sa` with the Spectrum Estimator block followed by an Array Plot block. To view the model, enter `ex_psd_estimatorblock` in the MATLAB command prompt. In addition, to access the spectral estimation data in MATLAB, connect the To Workspace block to the output of the Spectrum Estimator block. Here are the changes to the settings of the Spectrum Estimator block and the Array Plot block.

Block	Parameter Changes	Purpose of the block
Spectrum Estimator	• **Frequency resolution method** to `Number of frequency bands`. • **Frequency range** to `One-sided`.	Computes the power spectrum of the input signal using the filter bank approach.

Block	Parameter Changes	Purpose of the block
Array Plot	Click **View** and • select `Style`. In the Style window, select the **Plot type** as `Stairs`. • select `Configuration Properties`. In the Configuration Properties window, on the **Main** tab, set the **Sample increment** as `44.1/1024`. On the **Display** tab, change **X-label** to `Frequency (kHz)`, **Y-label** to `Power (dBm)`. For details, see the section 'Convert x-axis to Represent Frequency'. In addition, set **Y-limits (Minimum)** to `-100` and **Y-limits (Maximum)** to `40`.	Displays the power spectrum data.

The spectrum displayed in the Array Plot block is similar to the spectrum seen in the Spectrum Analyzer block in `ex_psd_sa`.



The filter bank approach produces peaks that have very minimal spectral leakage.

Convert x-axis to Represent Frequency

By default, the Array Plot block plots the PS data with respect to the number of samples per frame. The number of points on the x-axis equals the length of the input frame. The spectrum analyzer plots the PS data with respect to frequency. For a one-sided spectrum, the frequency varies in the range [0 Fs/2]. For a two-sided spectrum, the frequency varies in the range [-Fs/2 Fs/2]. To convert the x-axis of the array plot from sample-based to frequency-based, do the following:

- Click on the **Configuration Properties** icon . On **Main** tab, set **Sample increment** to `Fs/FrameLength`.
- For a one-sided spectrum, set **X-offset** to `0`.
- For a two-sided spectrum, set **X-offset** to `-Fs/2`.

In this example, the spectrum is one-sided and hence, the **Sample increment** and **X-offset** are set to `44100/1024` and `0`, respectively. To specify the frequency in kHz, set the **Sample increment** to `44.1/1024`.

Live Processing

The output of the Spectrum Estimator block contains the spectral data and is available for further processing. The data can be processed in real-time or it can be stored in the workspace using the To Workspace block. This example writes the spectral data to the workspace variable `Estimate`.

See Also

More About

- “Estimate the Power Spectrum in MATLAB” on page 16-14
- “Estimate the Transfer Function of an Unknown System” on page 16-43
- “View the Spectrogram Using Spectrum Analyzer” on page 16-50
- “Spectral Analysis” on page 16-59

Estimate the Transfer Function of an Unknown System

In this section...
“Estimate the Transfer Function in MATLAB” on page 16-43
“Estimate the Transfer Function in Simulink” on page 16-46

You can estimate the transfer function of an unknown system based on the system's measured input and output data.

In DSP System Toolbox, you can estimate the transfer function of a system using the `dsp.TransferFunctionEstimator` System object in MATLAB and the Discrete Transfer Function Estimator block in Simulink. The relationship between the input x and output y is modeled by the linear, time-invariant transfer function T_{xy}. The transfer function is the ratio of the cross power spectral density of x and y, P_{yx}, to the power spectral density of x, P_{xx}:

$$T_{xy}(f) = \frac{P_{yx}(f)}{P_{xx}(f)}$$

The `dsp.TransferFunctionEstimator` object and Discrete Transfer Function Estimator block use the Welch’s averaged periodogram method to compute the P_{xx} and P_{xy}. For more details on this method, see “Spectral Analysis” on page 16-59.

Coherence

The coherence, or magnitude-squared coherence, between x and y is defined as:

$$C_{xy}(f) = \frac{\left|P_{xy}\right|^2}{P_{xx} * P_{yy}}$$

The coherence function estimates the extent to which you can predict y from x. The value of the coherence is in the range $0 \le C_{xy}(f) \le 1$. If $C_{xy} = 0$, the input x and output y are unrelated. A C_{xy} value greater than 0 and less than 1 indicates one of the following:

- Measurements are noisy.
- The system is nonlinear.
- Output y is a function of x and other inputs.

The coherence of a linear system represents the fractional part of the output signal power that is produced by the input at that frequency. For a particular frequency, $1 - C_{xy}$ is an estimate of the fractional power of the output that the input does not contribute to.

When you set the `OutputCoherence` property of `dsp.TransferFunctionEstimator` to `true`, the object computes the output coherence. In the Discrete Transfer Function Estimator block, to compute the coherence spectrum, select the **Output magnitude squared coherence estimate** check box.

Estimate the Transfer Function in MATLAB

To estimate the transfer function of a system in MATLAB™, use the `dsp.TransferFunctionEstimator` System object™. The object implements the Welch's average modified periodogram method and uses the measured input and output data for estimation.

Initialize the System

The system is a cascade of two filter stages: dsp.LowpassFilter and a parallel connection of dsp.AllpassFilter and dsp.AllpoleFilter.

```
allpole = dsp.AllpoleFilter;
allpass = dsp.AllpassFilter;
lpfilter = dsp.LowpassFilter;
```

Specify Signal Source

The input to the system is a sine wave with a frequency of 100 Hz. The sampling frequency is 44.1 kHz.

```
sine = dsp.SineWave('Frequency',100,'SampleRate',44100,...
    'SamplesPerFrame',1024);
```

Create Transfer Function Estimator

To estimate the transfer function of the system, create the `dsp.TransferFunctionEstimator` System object.

```
tfe  = dsp.TransferFunctionEstimator('FrequencyRange','onesided',...
    'OutputCoherence', true);
```

Create Array Plot

Initialize two `dsp.ArrayPlot` objects: one to display the magnitude response of the system and the other to display the coherence estimate between the input and the output.

```
tfeplotter = dsp.ArrayPlot('PlotType','Line',...
    'XLabel','Frequency (Hz)',...
    'YLabel','Magnitude Response (dB)',...
    'YLimits',[-120 20],...
    'XOffset',0,...
    'XLabel','Frequency (Hz)',...
    'Title','System Transfer Function',...
    'SampleIncrement',44100/1024);
coherenceplotter = dsp.ArrayPlot('PlotType','Line',...
    'YLimits',[0 1.2],...
    'YLabel','Coherence',...
    'XOffset',0,...
    'XLabel','Frequency (Hz)',...
    'Title','Coherence Estimate',...
    'SampleIncrement',44100/1024);
```

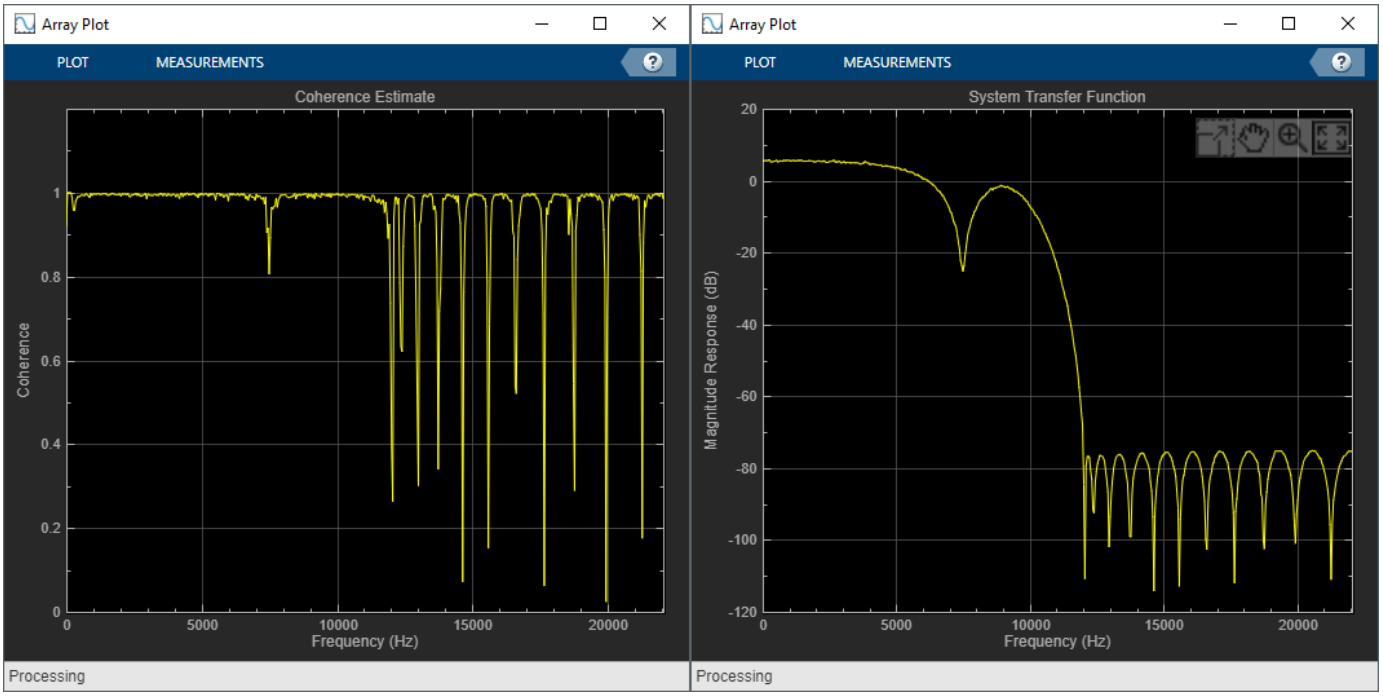
By default, the x-axis of the array plot is in samples. To convert this axis into frequency, set the 'SampleIncrement' property of the `dsp.ArrayPlot` object to Fs/1024. In this example, this value is 44100/1024, or 43.0664. For a two-sided spectrum, the `XOffset` property of the `dsp.ArrayPlot` object must be [-Fs/2]. The frequency varies in the range [-Fs/2 Fs/2]. In this example, the array plot shows a one-sided spectrum. Hence, set the `XOffset` to 0. The frequency varies in the range [0 Fs/2].

Estimate the Transfer Function

The transfer function estimator accepts two signals: input to the two-stage filter and output of the two-stage filter. The input to the filter is a sine wave containing additive white Gaussian noise. The noise has a mean of zero and a standard deviation of 0.1. The estimator estimates the transfer

function of the two-stage filter. The output of the estimator is the frequency response of the filter, which is complex. To extract the magnitude portion of this complex estimate, use the abs function. To convert the result into dB, apply a conversion factor of 20*log10(magnitude).

```
for Iter = 1:1000
    input = sine() + .1*randn(1024,1);
    lpfout = lpfilter(input);
    allpoleout = allpole(lpfout);
    allpassout = allpass(lpfout);
    output = allpoleout + allpassout;
    [tfeoutput,outputcoh] = tfe(input,output);
    tfeplotter(20*log10(abs(tfeoutput)));
    coherenceplotter(outputcoh);
end
```



The first plot shows the magnitude response of the system. The second plot shows the coherence estimate between the input and output of the system. Coherence in the plot varies in the range [0 1] as expected.

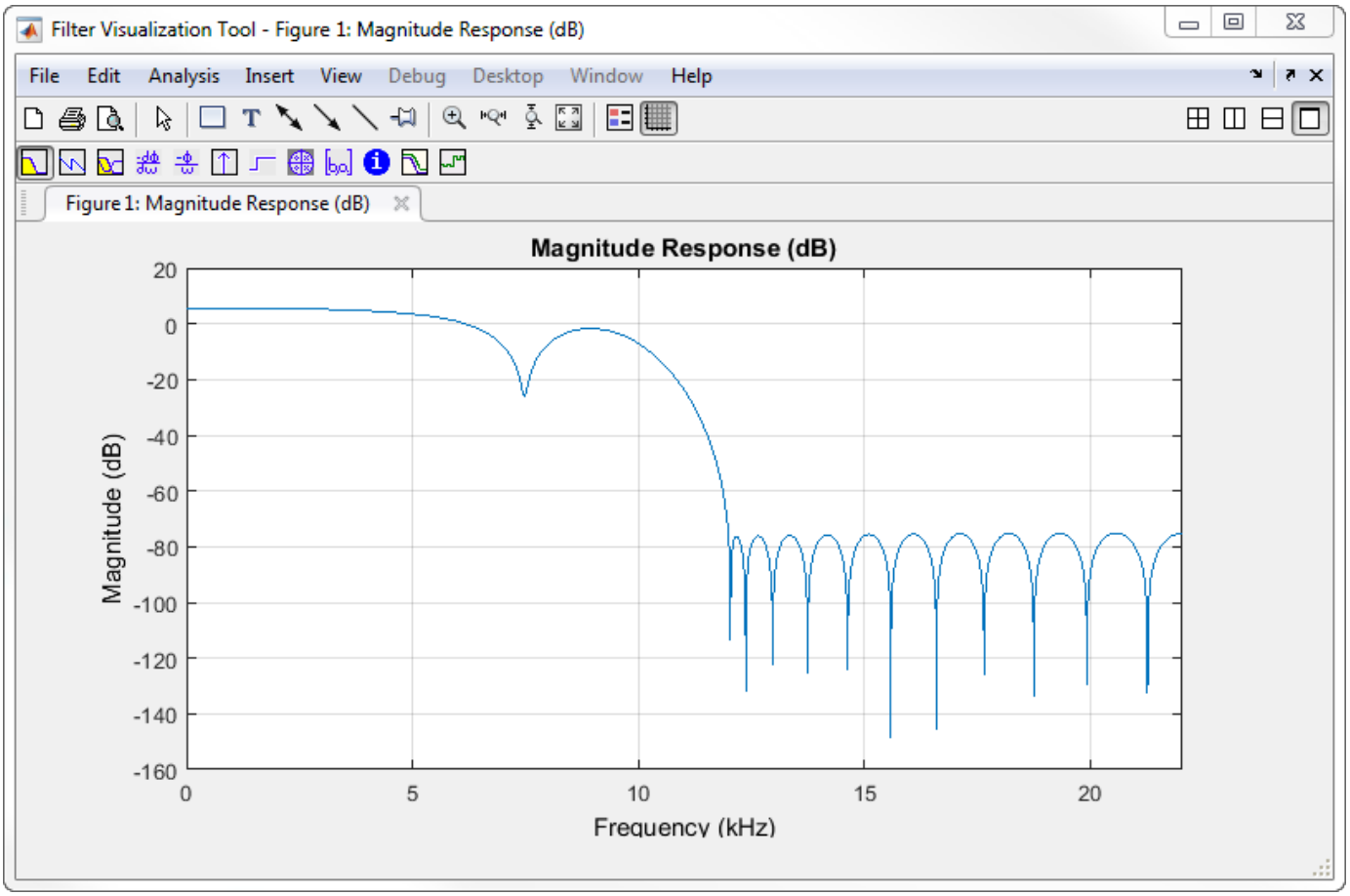
Magnitude Response of the Filter Using fvtool

The filter is a cascade of two filter stages - dsp.LowpassFilter and a parallel connection of dsp.AllpassFilter and dsp.AllpoleFilter. All the filter objects are used in their default state. Using the filter coefficients, derive the system transfer function and plot the frequency response using freqz. Below are the coefficients in the [Num] [Den] format:

- All pole filter - [1 0] [1 0.1]
- All pass filter - [0.5 -1/sqrt(2) 1] [1 -1/sqrt(2) 0.5]
- Lowpass filter - Determine the coefficients using the following commands:

```
lpf = dsp.LowpassFilter;
Coefficients = coeffs(lpf);
```

Coefficients.Numerator gives the coefficients in an array format. The mathematical derivation of the overall system transfer function is not shown here. Once you derive the transfer function, run fvtool and you can see the frequency response below:



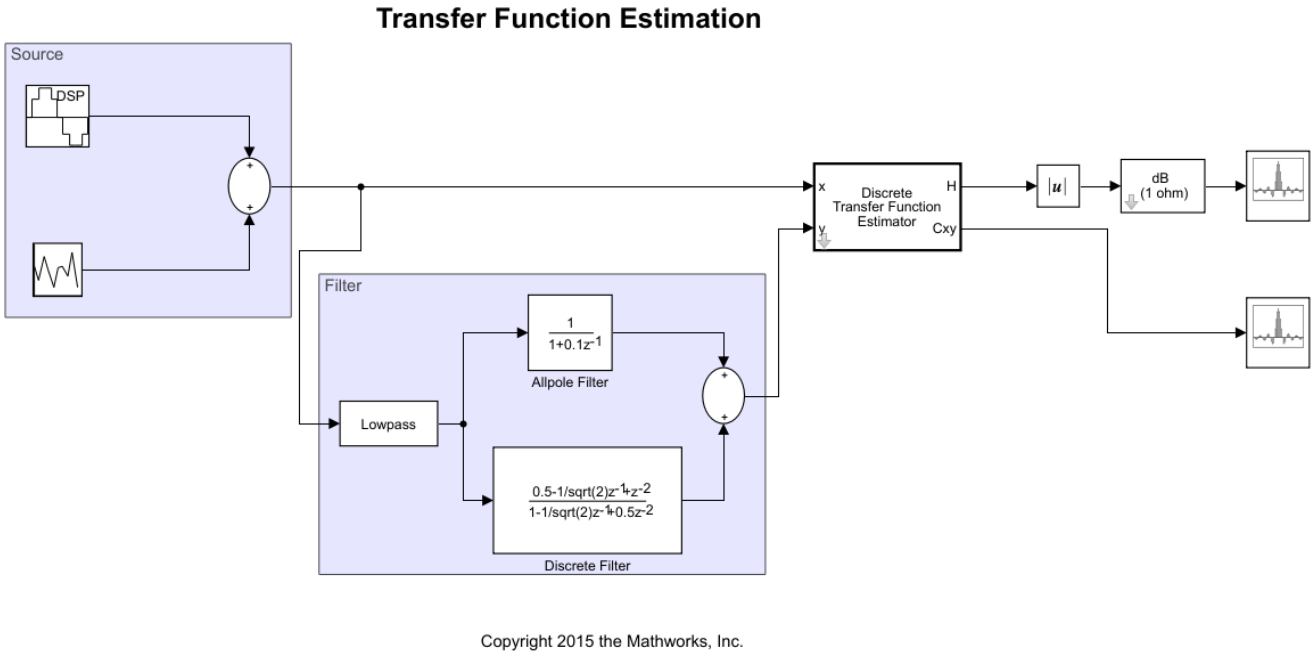
The magnitude response that fvtool shows matches the magnitude response that the `dsp.TransferFunctionEstimator` object estimates.

Estimate the Transfer Function in Simulink

Not supported in MATLAB Online.

To estimate the transfer function of a system in Simulink, use the Discrete Transfer Function Estimator block. The block implements the Welch's average modified periodogram method and uses the measured input and output data for estimation.

The system is a cascade of two filter stages: a lowpass filter and a parallel connection of an allpole filter and allpass filter. The input to the system is a sine wave containing additive white Gaussian noise. The noise has a mean of zero and a standard deviation of 0.1. The input to the estimator is the system input and the system output. The output of the estimator is the frequency response of the system, which is complex. To extract the magnitude portion of this complex estimate, use the Abs block. To convert the result into dB, the system uses a dB (1 ohm) block.



Open and Inspect the Model

To open the model, enter `ex_transfer_function_estimator` in the MATLAB command prompt.

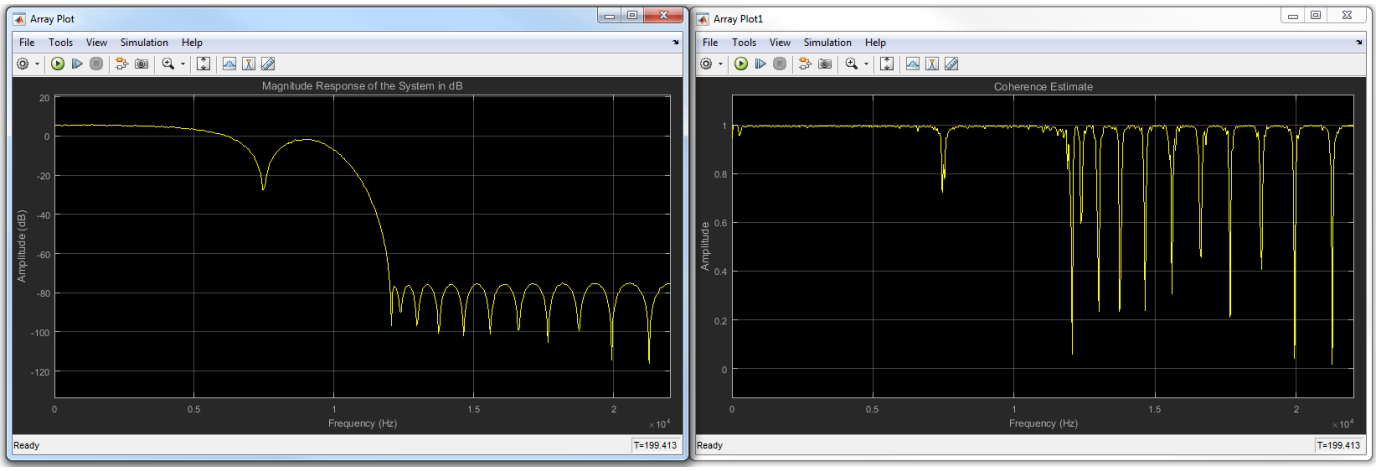
Here are the settings of the blocks in the model.

Block	Parameter Changes	Purpose of the block
Sine Wave	• **Sample time** to `1/44100` • **Samples per frame** to `1024`	Sinusoid signal with frequency at 100 Hz
Random Source	• **Source type** to `Gaussian` • **Variance** to 0.01 • **Sample time** to 1/44100 • **Samples per frame** to 1024	Random Source block generates a random noise signal with properties specified through the block dialog box
Lowpass Filter	No change	Lowpass filter
Allpole Filter	No change	Allpole filter with coefficients `[1 0.1]`
Discrete Filter	• **Numerator** to `[0.5 -1/sqrt(2) 1]` • **Denominator** to `[1 -1/sqrt(2) 0.5]`	Allpass filter with coefficients `[-1/sqrt(2) 0.5]`
Discrete Transfer Function Estimator	• **Frequency range** to `One-sided` • **Number of spectral averages** to 8	Transfer function estimator

Block	Parameter Changes	Purpose of the block
Abs	No change	Extracts the magnitude information from the output of the transfer function estimator
First Array Plot block	Click **View**: • Select **Style** and set **Plot type** to `Line`. • Select **Configuration Properties**: From the **Main** tab, set **Sample increment** to `44100/1024` and **X-offset** to `0`. In the **Display** tab, specify the **Title** as `Magnitude Response of the System in dB`, **X-label** as `Frequency (Hz)`, and **Y-label** as `Amplitude (dB)`	Shows the magnitude response of the system
Second Array Plot block	Click **View**: • Select **Style** and set **Plot type** to `Line`. • Select **Configuration Properties**: From the **Main** tab, set **Sample increment** to `44100/1024` and **X-offset** to `0`. In the **Display** tab, specify the **Title** as `Coherence Estimate`, **X-label** as `Frequency (Hz)`, and **Y-label** as `Amplitude`	Shows the coherence estimate

By default, the *x*-axis of the array plot is in samples. To convert this axis into frequency, the **Sample increment** parameter is set to `Fs/1024`. In this example, this value is `44100/1024`, or `43.0664`. For a two-sided spectrum, the **X-offset** parameter must be `–Fs/2`. The frequency varies in the range `[-Fs/2 Fs/2]`. In this example, the array plot shows a one-sided spectrum. Hence, the **X-offset** is set to 0. The frequency varies in the range `[0 Fs/2]`.

Run the Model



The first plot shows the magnitude response of the system. The second plot shows the coherence estimate between the input and output of the system. Coherence in the plot varies in the range `[0 1]` as expected.

See Also

More About

- “Spectral Analysis” on page 16-59
- “Estimate the Power Spectrum in MATLAB” on page 16-14
- “Estimate the Power Spectrum in Simulink” on page 16-27

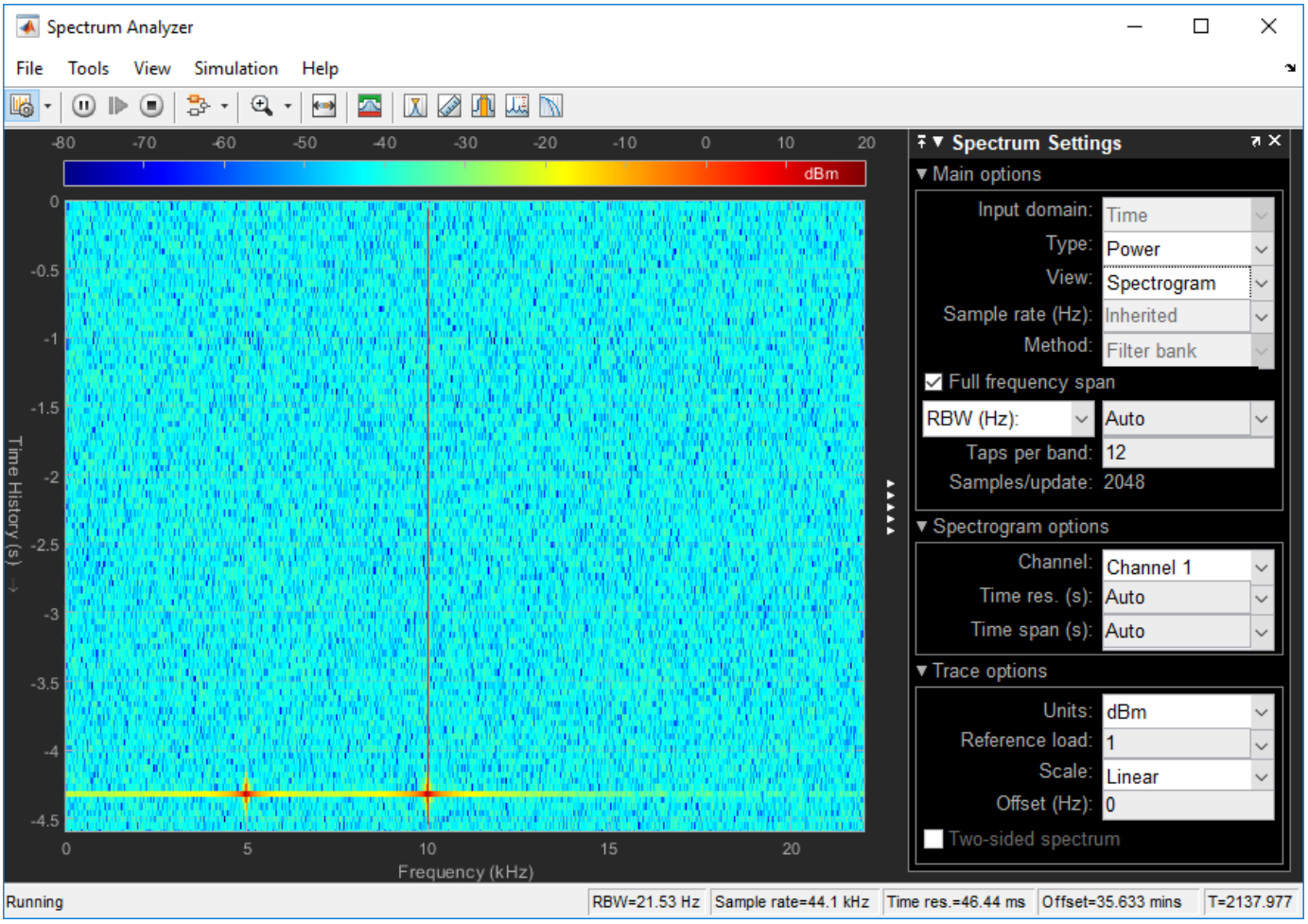
View the Spectrogram Using Spectrum Analyzer

In this section...
“Colormap” on page 16-51
“Display” on page 16-52
“Resolution Bandwidth (RBW)” on page 16-52
“Time Resolution” on page 16-55
“Convert the Power Between Units” on page 16-55
“Scale Color Limits” on page 16-57

Spectrograms are a two-dimensional representation of the power spectrum of a signal as this signal sweeps through time. They give a visual understanding of the frequency content of your signal. Each line of the spectrogram is one periodogram computed using either the filter bank approach or the Welch’s algorithm of averaging modified periodogram.

To show the concepts of the spectrogram, this example uses the model `ex_psd_sa` as the starting point. Note that Simulink models are not supported in MATLAB Online.

Open the model and double-click the Spectrum Analyzer block. In the **Spectrum Settings** pane, change **View** to `Spectrogram`. The **Method** is set to `Filter bank`. Run the model. You can see the spectrogram output in the spectrum analyzer window. To acquire and store the data for further processing, create a `Spectrum Analyzer Configuration` object and run the `getSpectrumData` function on this object.

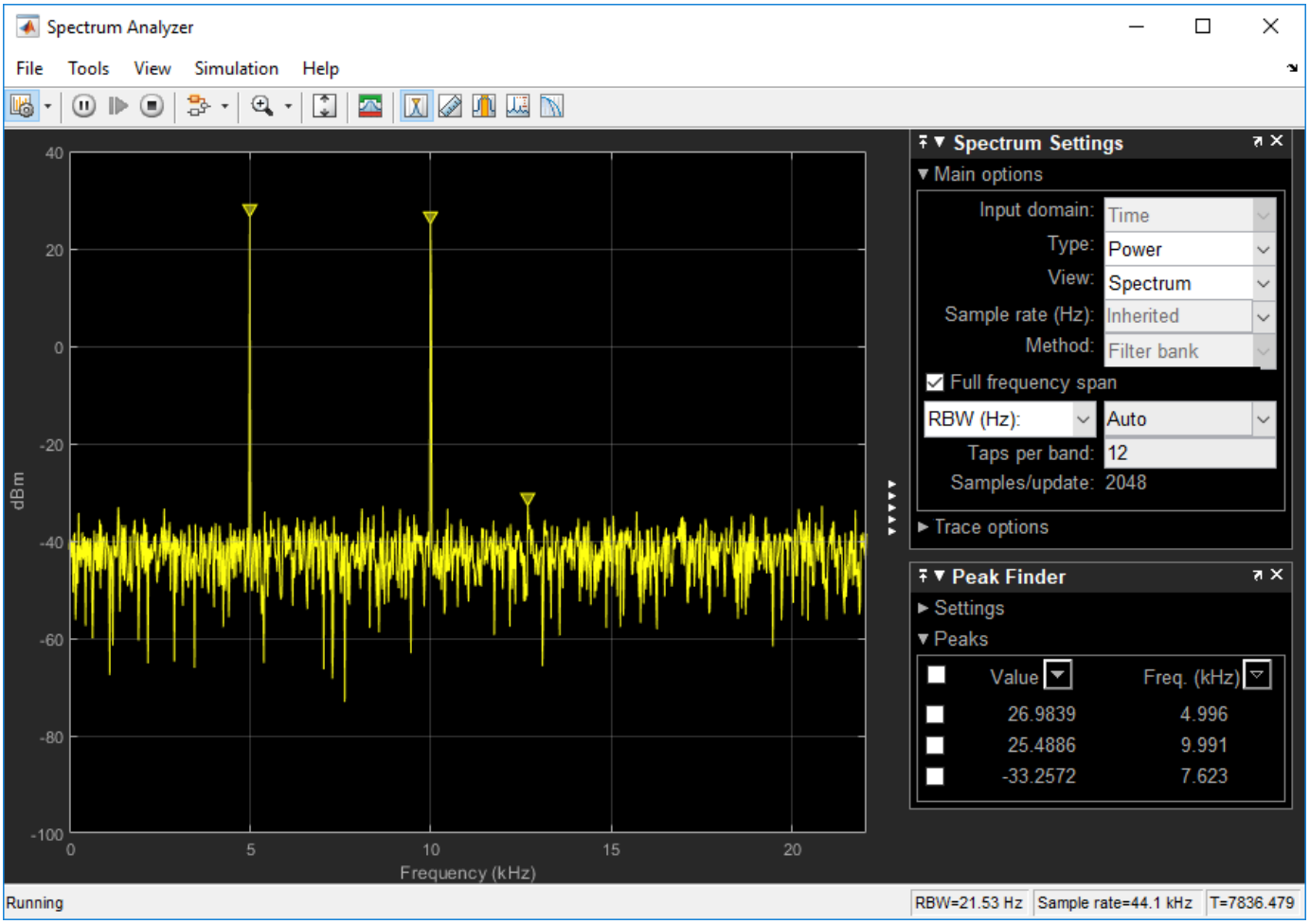


Colormap

Power spectrum is computed as a function of frequency `f` and is plotted as a horizontal line. Each point on this line is given a specific color based on the value of the power at that particular frequency. The color is chosen based on the colormap seen at the top of the display. To change the colormap, click **View > Configuration Properties**, and choose one of the options in **color map**. Make sure **View** is set to `Spectrogram`. By default, **color map** is set to `jet(256)`.

The two frequencies of the sine wave are distinctly visible at 5 kHz and 10 kHz. Since the spectrum analyzer uses the filter bank approach, there is no spectral leakage at the peaks. The sine wave is embedded in Gaussian noise, which has a variance of 0.0001. This value corresponds to a power of -40 dBm. The color that maps to -40 dBm is assigned to the noise spectrum. The power of the sine wave is 26.9 dBm at 5 kHz and 10 kHz. The color used in the display at these two frequencies corresponds to 26.9 dBm on the colormap. For more information on how the power is computed in dBm, see 'Conversion of power in watts to dBW and dBm'.

To confirm the dBm values, change **View** to `Spectrum`. This view shows the power of the signal at various frequencies.



You can see that the two peaks in the power display have an amplitude of about 26 dBm and the white noise is averaging around -40 dBm.

Display

In the spectrogram display, time scrolls from top to bottom, so the most recent data is shown at the top of the display. As the simulation time increases, the offset time also increases to keep the vertical axis limits constant while accounting for the incoming data. The `Offset` value, along with the simulation time, is displayed at the bottom-right corner of the spectrogram scope.

Resolution Bandwidth (RBW)

Resolution Bandwidth (RBW) is the minimum frequency bandwidth that can be resolved by the spectrum analyzer. By default, **RBW (Hz)** is set to `Auto`. In the auto mode, *RBW* is the ratio of the frequency span to 1024. In a two-sided spectrum, this value is $F_s/1024$, while in a one-sided spectrum, it is $(F_s/2)/1024$. In this example, RBW is (44100/2)/1024 or 21.53 Hz.

If the **Method** is set to `Filter bank`, using this value of *RBW*, the number of input samples used to compute one spectral update is given by $N_{samples} = F_s/RBW$, which is 44100/21.53 or 2048 in this example.

If the **Method** is set to `Welch`, using this value of *RBW*, the window length ($N_{samples}$) is computed iteratively using this relationship:

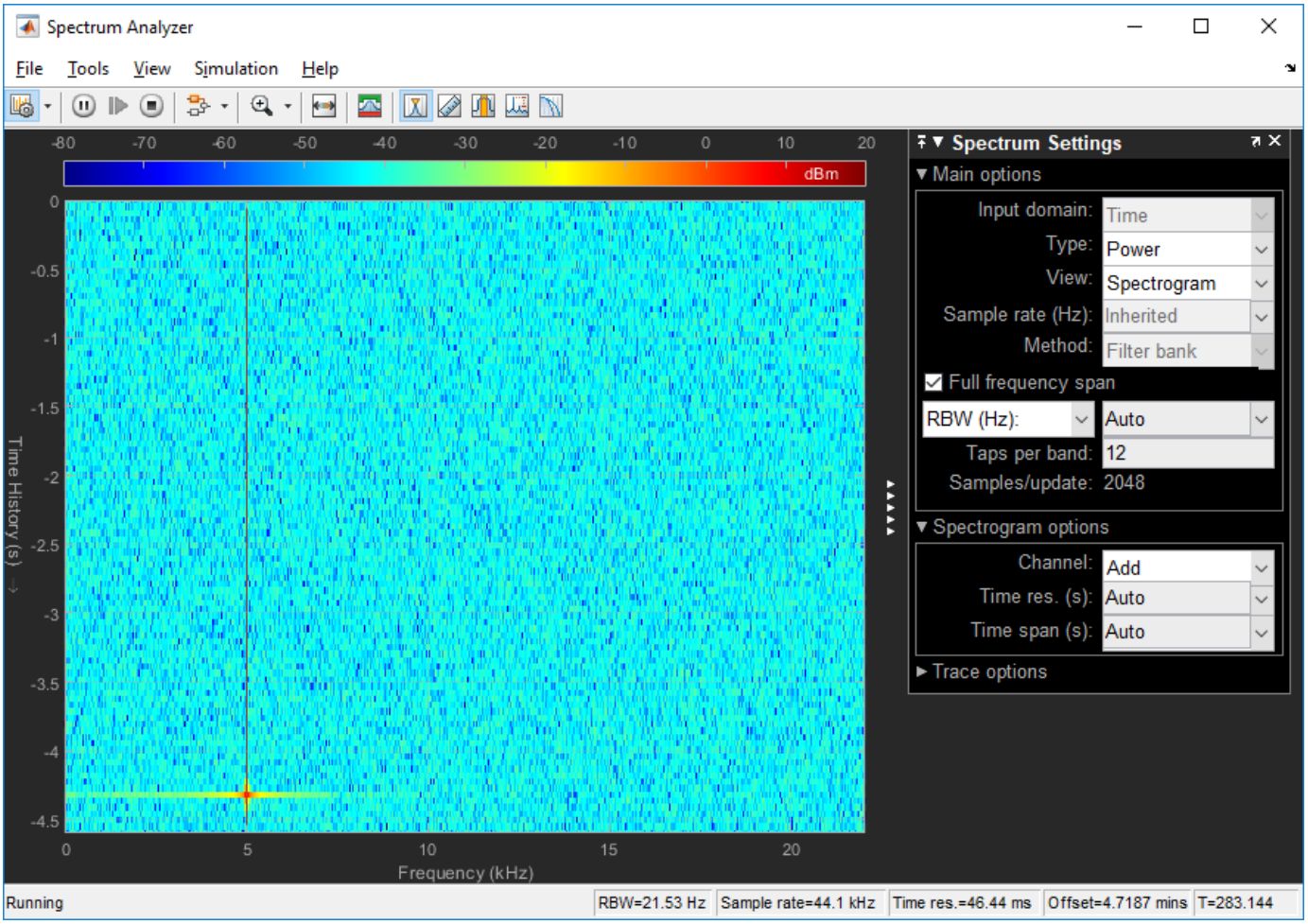
$$N_{samples} = \frac{\left(1 - \frac{O_p}{100}\right) \times NENBW \times F_s}{RBW}$$

O_p is the amount of overlap between the previous and current buffered data segments. *NENBW* is the equivalent noise bandwidth of the window.

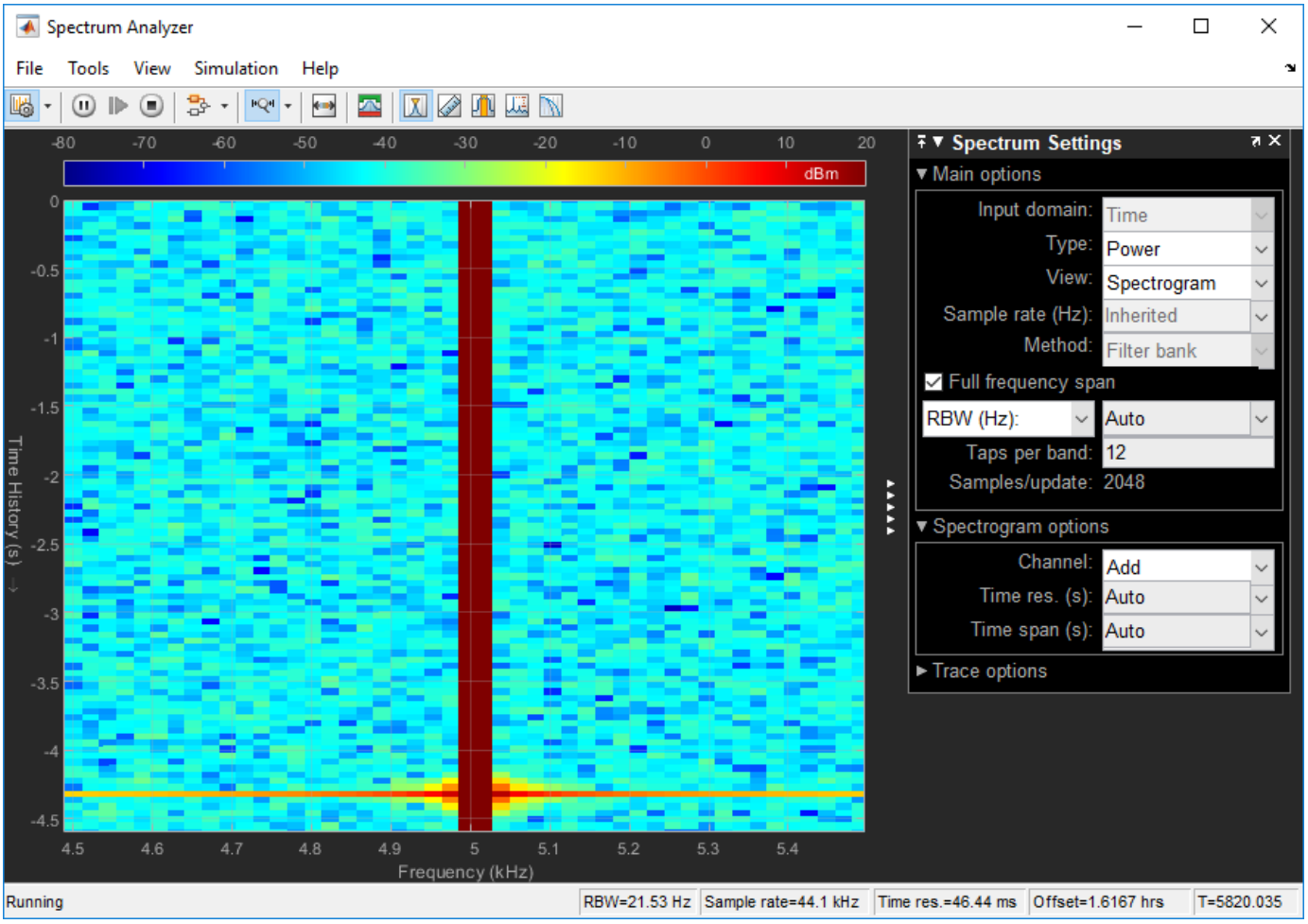
For more information on the details of the spectral estimation algorithm, see “Spectral Analysis” on page 16-59.

To distinguish between two frequencies in the display, the distance between the two frequencies must be at least RBW. In this example, the distance between the two peaks is 5000 Hz, which is greater than *RBW*. Hence, you can see the peaks distinctly.

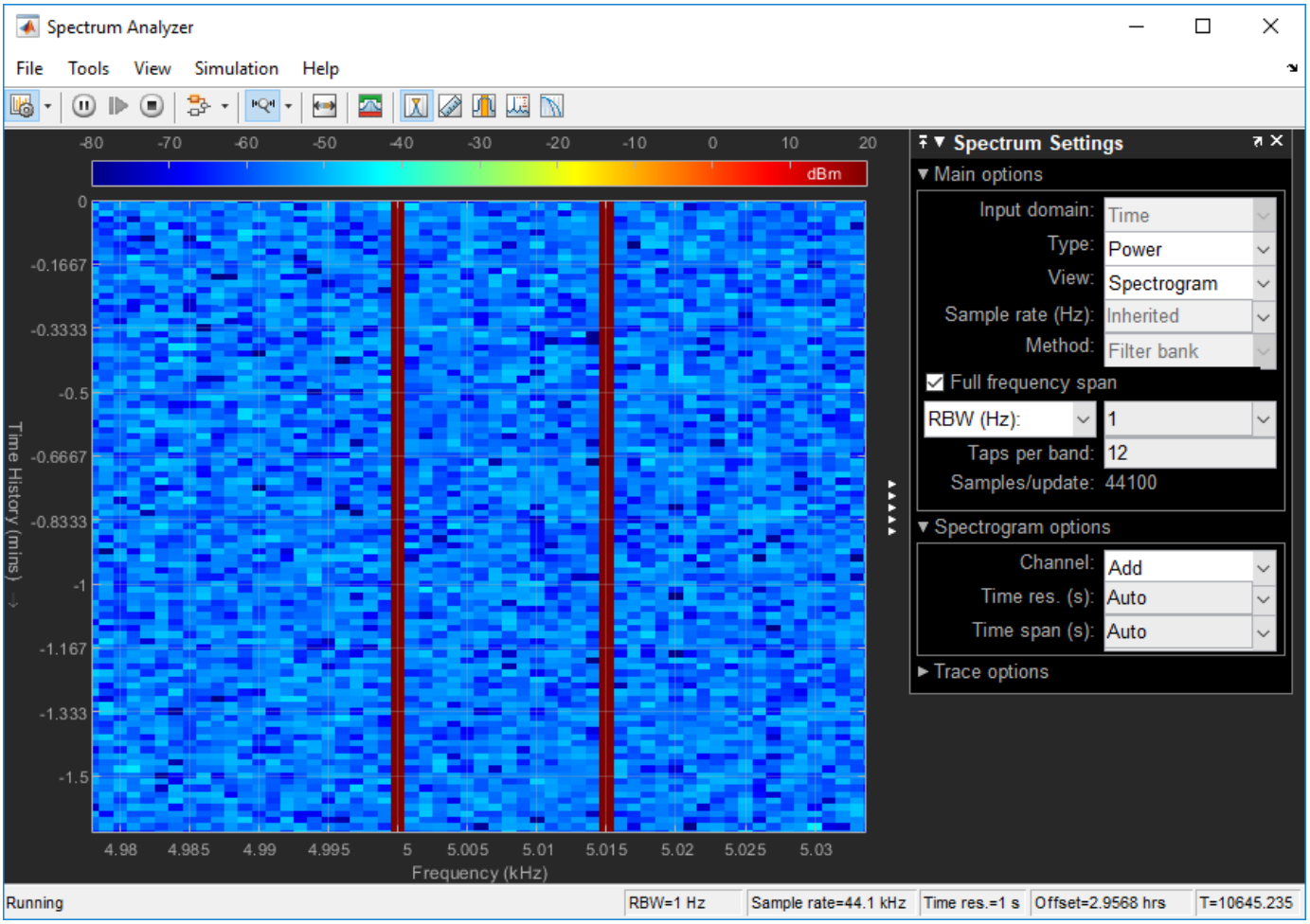
Change the frequency of the second sine wave from 10000 Hz to 5015 Hz. The difference between the two frequencies is 15 Hz, which is less than *RBW*.



On zooming, you can see that the peaks are not distinguishable.



To increase the frequency resolution, decrease *RBW* to 1 Hz and run the simulation. On zooming, the two peaks, which are 15 Hz apart, are now distinguishable



Time Resolution

Time resolution is the distance between two spectral lines in the vertical axis. By default, **Time res (s)** is set to `Auto`. In this mode, the value of time resolution is `1/RBW` s, which is the minimum attainable resolution. When you increase the frequency resolution, the time resolution decreases. To maintain a good balance between the frequency resolution and time resolution, change the **RBW (Hz)** to `Auto`. You can also specify the **Time res (s)** as a numeric value.

Convert the Power Between Units

The spectrum analyzer provides three units to specify the power spectral density: `Watts/Hz`, `dBm/Hz`, and `dBW/Hz`. Corresponding units of power are `Watts`, `dBm`, and `dBW`. For electrical engineering applications, you can also view the RMS of your signal in `Vrms` or `dBV`. The default spectrum type is **Power** in `dBm`.

Convert the Power in Watts to dBW and dBm

Power in `dBW` is given by:

$$P_{dBW} = 10\log10(power\ in\ watt/1\ watt)$$

Power in `dBm` is given by:

$$P_{dBm} = 10\log10(power\ in\ watt/1\ milliwatt)$$

For a sine wave signal with an amplitude of 1 V, the power of a one-sided spectrum in `Watts` is given by:

$$P_{Watts} = A^2/2$$
$$P_{Watts} = 1/2$$

In this example, this power equals 0.5 W. Corresponding power in dBm is given by:

$$P_{dBm} = 10\log10(power\ in\ watt/1\ milliwatt)$$
$$P_{dBm} = 10\log10(0.5/10^{-3})$$

Here, the power equals 26.9897 dBm. To confirm this value with a peak finder, click **Tools** > **Measurements** > **Peak Finder**.

For a white noise signal, the spectrum is flat for all frequencies. The spectrum analyzer in this example shows a one-sided spectrum in the range [0 Fs/2]. For a white noise signal with a variance of 1e-4, the power per unit bandwidth ($P_{unitbandwidth}$) is 1e-4. The total power of white noise in **watts** over the entire frequency range is given by:

$$P_{whitenoise} = P_{unitbandwidth} * number\ of\ frequency\ bins,$$
$$P_{whitenoise} = (10^{-4}) * \left(\frac{Fs/2}{RBW}\right),$$
$$P_{whitenoise} = (10^{-4}) * \left(\frac{22050}{21.53}\right)$$

The number of frequency bins is the ratio of total bandwidth to RBW. For a one-sided spectrum, the total bandwidth is half the sampling rate. RBW in this example is 21.53 Hz. With these values, the total power of white noise in **watts** is 0.1024 W. In dBm, the power of white noise can be calculated using 10*log10(0.1024/10^-3), which equals 20.103 dBm.

Convert Power in Watts to dBFS

If you set the spectral units to `dBFS` and set the full scale (`FullScaleSource`) to `Auto`, power in `dBFS` is computed as:

$$P_{dBFS} = 20 \cdot \log_{10}(\sqrt{P_{watts}}/Full_Scale)$$

where:

- P_{watts} is the power in watts
- For double and float signals, *Full_Scale* is the maximum value of the input signal.
- For fixed point or integer signals, *Full_Scale* is the maximum value that can be represented.

If you specify a manual full scale (set `FullScaleSource` to `Property`), power in `dBFS` is given by:

$$P_{FS} = 20 \cdot \log_{10}(\sqrt{P_{watts}}/FS)$$

Where `FS` is the full scaling factor specified in the `FullScale` property.

For a sine wave signal with an amplitude of 1 V, the power of a one-sided spectrum in `Watts` is given by:

$$P_{Watts} = A^2/2$$
$$P_{Watts} = 1/2$$

In this example, this power equals 0.5 W and the maximum input signal for a sine wave is 1 V. The corresponding power in dBFS is given by:

$$P_{FS} = 20 \cdot \log_{10}(\sqrt{1/2}/1)$$

Here, the power equals -3.0103. To confirm this value in the spectrum analyzer, run these commands:

```
Fs = 1000;  % Sampling frequency
sinef = dsp.SineWave('SampleRate',Fs,'SamplesPerFrame',100);
scope = dsp.SpectrumAnalyzer('SampleRate',Fs,...
   'SpectrumUnits','dBFS','PlotAsTwoSidedSpectrum',false)
%%
for ii = 1:100000
xsine = sinef();
scope(xsine)
end
```

Then, click **Tools** > **Measurements** > **Peak Finder**.

Convert the Power in dBm to RMS in Vrms

Power in `dBm` is given by:

$$P_{dBm} = 10\log10(power\ in\ watt/1\ milliwatt)$$

Voltage in RMS is given by:

$$V_{rms} = 10^{P_{dBm}/20}\sqrt{10^{-3}}$$

From the previous example, P_{dBm} equals 26.9897 dBm. The V_{rms} is calculated as

$$V_{rms} = 10^{26.9897/20}\sqrt{0.001}$$

which equals 0.7071.

To confirm this value:

1. Change **Type** to `RMS`.
2. Open the peak finder by clicking **Tools** > **Measurements** > **Peak Finder**.

Scale Color Limits

When you run the model and do not see the spectrogram colors, click the **Scale Color Limits** button. This option autoscales the colors.

The spectrogram updates in real time. During simulation, if you change any of the tunable parameters in the model, the changes are effective immediately in the spectrogram.

See Also

More About

- “Estimate the Power Spectrum in MATLAB” on page 16-14
- “Estimate the Power Spectrum in Simulink” on page 16-27

Spectral Analysis

In this section...
“Welch’s Algorithm of Averaging Modified Periodograms” on page 16-59
“Filter Bank” on page 16-62

Spectral analysis is the process of estimating the power spectrum (PS) of a signal from its time-domain representation. Spectral density characterizes the frequency content of a signal or a stochastic process. Intuitively, the spectrum decomposes the signal or the stochastic process into the different frequencies, and identifies periodicities. The most commonly used instrument for performing spectral analysis is the spectrum analyzer.

Spectral analysis is done based on the nonparametric methods and the parametric methods. Nonparametric methods are based on dividing the time-domain data into segments, applying Fourier transform on each segment, computing the squared-magnitude of the transform, and summing and averaging the transform. Nonparametric methods such as modified periodogram, Bartlett, Welch, and the Blackman-Tukey methods, are a variation of this approach. These methods are based on measured data and do not require prior knowledge about the data or the model. Parametric methods are model-based approaches. The model for generating the signal can be constructed with a number of parameters that can estimated from the observed data. From the model and estimated parameters, the algorithm computes the power spectrum implied by the model.

The spectrum analyzer in DSP System Toolbox uses the Welch’s nonparametric method of averaging modified periodogram and the filter bank method to estimate the power spectrum of a streaming signal in real time. You can launch the spectrum analyzer using the `dsp.SpectrumAnalyzer` System object in MATLAB and the Spectrum Analyzer block in Simulink.

Welch’s Algorithm of Averaging Modified Periodograms

To use the Welch method in the spectrum analyzer, set the **Method** parameter to `Welch`. The Welch's technique to reduce the variance of the periodogram breaks the time series into overlapping segments. This method computes a modified periodogram for each segment and then averages these estimates to produce the estimate of the power spectrum. Because the process is wide-sense stationary and Welch's method uses PS estimates of different segments of the time series, the modified periodograms represent approximately uncorrelated estimates of the true PS. The averaging reduces the variability.

The segments are multiplied by a window function, such as a Hann window, so that Welch's method amounts to averaging modified periodograms. Because the segments usually overlap, data values at the beginning and end of the segment tapered by the window in one segment, occur away from the ends of adjacent segments. The overlap guards against the loss of information caused by windowing. In the Spectrum Analyzer block, you can specify the window using the **Window** parameter.

The algorithm in the Spectrum Analyzer block consists of these steps:

1 The block buffers the input into N point data segments. Each data segment is split up into L overlapping data segments, each of length M, overlapping by D points. The data segments can be represented as:

$$x_i(n) = x(n + iD), \quad n = 0, 1, ..., M - 1$$
$$i = 0, 1, ..., L - 1$$

- If $D = M/2$, the overlap is 50%.
- If $D = 0$, the overlap is 0%.

The block uses the `RBW` or the `Window Length` setting in the **Spectrum Settings** pane to determine the data window length. Then, it partitions the input signal into a number of windowed data segments.

The spectrum analyzer requires a minimum number of samples ($N_{samples}$) to compute a spectral estimate. This number of input samples required to compute one spectral update is shown as **Samples/update** in the **Main options** pane. This value is directly related to the resolution bandwidth, *RBW*, by the following equation:

$$N_{samples} = \frac{\left(1 - \frac{O_p}{100}\right) \times NENBW \times F_s}{RBW}.$$

- O_p, the amount of overlap (%) between the previous and current buffered data segments, is specified through the **Overlap (%)** parameter in the **Window options** pane.
- *NENBW*, the normalized effective noise bandwidth of the window depends on the windowing method. This parameter is shown in the **Window options** pane.
- F_s is the sample rate of the input signal.

When in `RBW` mode, the window length required to compute one spectral update, N_{window}, is directly related to the resolution bandwidth and normalized effective noise bandwidth:

$$N_{window} = \frac{NENBW \times F_s}{RBW}$$

When in `Window length` mode, the window length is used as specified.

The number of input samples required to compute one spectral update, $N_{samples}$, is directly related to the window length and the amount of overlap:

$$N_{samples} = \left(1 - \frac{O_p}{100}\right) N_{window}$$

When you increase the overlap percentage, fewer new input samples are needed to compute a new spectral update. For example, the table shows the number of input samples required to compute one spectral update when the window length is 100.

Overlap	***$N_{samples}$***
0%	100
50%	50
80%	20

The normalized effective noise bandwidth, *NENBW*, is a window parameter determined by the window length, N_{window}, and the type of window used. If $w(n)$ denotes the vector of N_{window} window coefficients, then *NENBW* is:

$$NENBW = N_{window} \times \frac{\sum_{n=1}^{N_{window}} w^2(n)}{\left[\sum_{n=1}^{N_{window}} w(n)\right]^2}$$

When in RBW mode, you can set the resolution bandwidth using the value of the **RBW** parameter on the **Main options** pane. You must specify a value so that there are at least two RBW intervals over the specified frequency span. The ratio of the overall span to RBW must be greater than two:

$$\frac{span}{RBW} > 2$$

By default, the **RBW** parameter on the **Main options** pane is set to `Auto`. In this case, the Spectrum Analyzer determines the appropriate value so that there are 1024 RBW intervals over the specified frequency span. Thus, when you set **RBW** to `Auto`, RBW is calculated by:

$$RBW_{auto} = \frac{span}{1024}$$

When in window length mode, you specify N_{window} and the resulting RBW is

$$\frac{NENBW \times F_s}{N_{window}}.$$

2 Apply a window to each of the L overlapping data segments in the time domain. Most window functions afford more influence to the data at the center of the set than to the data at the edges, which represents a loss of information. To mitigate that loss, the individual data sets are commonly overlapped in time. For each windowed segment, compute the periodogram by computing the discrete Fourier transform. Then compute the squared magnitude of the result, and divide the result by M.

$$P_{xx}^{i}(f) = \frac{1}{MU}\left|\sum_{n=0}^{M-1} x_i(n)w(n)e^{-j2\pi fn}\right|^2, \qquad i = 0, 1, ..., L-1$$

where U is a normalization factor for the power in the window function and is given by

$$U = \frac{1}{M}\sum_{n=0}^{M-1} w^2(n)$$

.

You can specify the window using the **Window** parameter.

3 To determine the Welch power spectrum estimate, the Spectrum Analyzer block averages the result of the periodograms for the last L data segments. The averaging reduces the variance, compared to the original N point data segment.

$$P_{xx}^{W}(f) = \frac{1}{L}\sum_{i=0}^{L-1} P_{xx}^{i}(f)$$

L is specified through the **Averages** parameter in the **Trace options** pane.

4 The Spectrum Analyzer block computes the power spectral density using:

$$P_{xx}^{W}(f) = \frac{1}{L * F_s} \sum_{i=0}^{L-1} P_{xx}^{i}(f)$$

.

Filter Bank

To use the filter bank approach in the spectrum analyzer, set the **Method** parameter to `Filter bank`. In the filter bank approach, the analysis filter bank splits the broadband input signal into multiple narrow subbands. The spectrum analyzer computes the power in each narrow frequency band and the computed value is the spectral estimate over the respective frequency band. For signals with relatively small length, the filter bank approach produces a spectral estimate with a higher resolution, a more accurate noise floor, and peaks more precise than the Welch method, with low or no spectral leakage. These advantages come at the expense of increased computation and slower tracking.

For information on how the filter bank computes the power, see the “Algorithms” section in `dsp.SpectrumEstimator`. For more information on the analysis filter bank and how it is implemented, see the “More About” and the “Algorithm” sections in `dsp.Channelizer`.

References

[1] Proakis, John G., and Dimitris G. Manolakis. *Digital Signal Processing*. 3rd ed. Upper Saddle River, NJ: Prentice Hall, 1996.

[2] Hayes, Monson H. *Statistical Digital Signal Processing and Modeling* Hoboken, NJ: John Wiley & Sons, 1996.

See Also

Objects
`dsp.SpectrumAnalyzer`

Blocks
Spectrum Analyzer

More About

- “Estimate the Power Spectrum in MATLAB” on page 16-14
- “Estimate the Power Spectrum in Simulink” on page 16-27
- “Estimate the Transfer Function of an Unknown System” on page 16-43
- “View the Spectrogram Using Spectrum Analyzer” on page 16-50

Fixed-Point Design

Learn about fixed-point data types and how to convert floating-point models to fixed point.

- “Fixed-Point Signal Processing” on page 17-2
- “Fixed-Point Concepts and Terminology” on page 17-4
- “Arithmetic Operations” on page 17-8
- “System Objects in DSP System Toolbox that Support Fixed-Point” on page 17-15
- “Simulink Blocks in DSP System Toolbox that Support Fixed-Point” on page 17-20
- “System Objects Supported by Fixed-Point Converter App” on page 17-26
- “Convert dsp.FIRFilter Object to Fixed-Point Using the Fixed-Point Converter App” on page 17-27
- “Specify Fixed-Point Attributes for Blocks” on page 17-33
- “Quantizers” on page 17-49
- “Create an FIR Filter Using Integer Coefficients” on page 17-58
- “Fixed-Point Precision Rules for Avoiding Overflow in FIR Filters” on page 17-70

Fixed-Point Signal Processing

In this section...
“Fixed-Point Features” on page 17-2
“Benefits of Fixed-Point Hardware” on page 17-2
“Benefits of Fixed-Point Design with System Toolboxes Software” on page 17-2

Note To take full advantage of fixed-point support in System Toolbox software, you must install Fixed-Point Designer software.

Fixed-Point Features

Many of the blocks in this product have fixed-point support, so you can design signal processing systems that use fixed-point arithmetic. Fixed-point support in DSP System Toolbox software includes

- Signed two's complement and unsigned fixed-point data types
- Word lengths from 2 to 128 bits in simulation
- Word lengths from 2 to the size of a `long` on the Simulink Coder C code-generation target
- Overflow handling and rounding methods
- C code generation for deployment on a fixed-point embedded processor, with Simulink Coder code generation software. The generated code uses all allowed data types supported by the embedded target, and automatically includes all necessary shift and scaling operations

Benefits of Fixed-Point Hardware

There are both benefits and trade-offs to using fixed-point hardware rather than floating-point hardware for signal processing development. Many signal processing applications require low-power and cost-effective circuitry, which makes fixed-point hardware a natural choice. Fixed-point hardware tends to be simpler and smaller. As a result, these units require less power and cost less to produce than floating-point circuitry.

Floating-point hardware is usually larger because it demands functionality and ease of development. Floating-point hardware can accurately represent real-world numbers, and its large dynamic range reduces the risk of overflow, quantization errors, and the need for scaling. In contrast, the smaller dynamic range of fixed-point hardware that allows for low-power, inexpensive units brings the possibility of these problems. Therefore, fixed-point development must minimize the negative effects of these factors, while exploiting the benefits of fixed-point hardware; cost- and size-effective units, less power and memory usage, and fast real-time processing.

Benefits of Fixed-Point Design with System Toolboxes Software

Simulating your fixed-point development choices before implementing them in hardware saves time and money. The built-in fixed-point operations provided by the System Toolboxes software save time in simulation and allow you to generate code automatically.

This software allows you to easily run multiple simulations with different word length, scaling, overflow handling, and rounding method choices to see the consequences of various fixed-point

designs before committing to hardware. The traditional risks of fixed-point development, such as quantization errors and overflow, can be simulated and mitigated in software before going to hardware.

Fixed-point C code generation with System Toolbox software and Simulink Coder code generation software produces code ready for execution on a fixed-point processor. All the choices you make in simulation in terms of scaling, overflow handling, and rounding methods are automatically optimized in the generated code, without necessitating time-consuming and costly hand-optimized code.

Fixed-Point Concepts and Terminology

In this section...
“Fixed-Point Data Types” on page 17-4 “Scaling” on page 17-5 “Precision and Range” on page 17-6

Fixed-Point Data Types

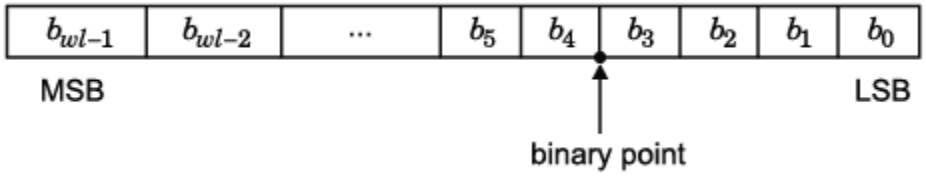
In digital hardware, numbers are stored in binary words. A binary word is a fixed-length sequence of bits (1's and 0's). The way hardware components or software functions interpret this sequence of 1's and 0's is defined by the data type.

Binary numbers are represented as either floating-point or fixed-point data types. In this section, we discuss many terms and concepts relating to fixed-point numbers, data types, and mathematics.

A fixed-point data type is characterized by the word length in bits, the position of the binary point, and the signedness of a number which can be signed or unsigned. Signed numbers and data types can represent both positive and negative values, whereas unsigned numbers and data types can only represent values that are greater than or equal to zero.

The position of the binary point is the means by which fixed-point values are scaled and interpreted.

For example, a binary representation of a generalized fixed-point number (either signed or unsigned) is shown below:



where

- b_i is the i^{th} binary digit.
- *wl* is the number of bits in a binary word, also known as word length.
- b_{wl-1} is the location of the most significant, or highest, bit (MSB). In signed binary numbers, this bit is the sign bit which indicates whether the number is positive or negative.
- b_0 is the location of the least significant, or lowest, bit (LSB). This bit in the binary word can represent the smallest value. The weight of the LSB is given by:

  $$weight_{LSB} = 2^{-fractionlength}$$

  where, *fractionlength* is the number of bits to the right of the binary point.
- Bits to the left of the binary point are integer bits and/or sign bits, and bits to the right of the binary point are fractional bits. Number of bits to the left of the binary point is known as the integer length. The binary point in this example is shown four places to the left of the LSB. Therefore, the number is said to have four fractional bits, or a fraction length of four.

Fixed-point data types can be either signed or unsigned.

Signed binary fixed-point numbers are typically represented in one of three ways:

- Sign/magnitude –– Representation of signed fixed-point or floating-point numbers. In the sign/magnitude representation, one bit of a binary word is always the dedicated sign bit, while the remaining bits of the word encode the magnitude of the number. Negation using sign/magnitude representation consists of flipping the sign bit from 0 (positive) to 1 (negative), or from 1 to 0.
- One's complement
- Two's complement –– Two's complement is the most common representation of signed fixed-point numbers. See “Two's Complement” on page 17-8 for more information.

Unsigned fixed-point numbers can only represent numbers greater than or equal to zero.

Scaling

In [Slope Bias] representation, fixed-point numbers can be encoded according to the scheme

$$real\text{-}worldvalue = (slope \times integer) + bias$$

where the slope can be expressed as

$$slope = slope\ adjustment \times 2^{exponent}$$

The term *slope adjustment* is sometimes used as a synonym for fractional slope.

In the trivial case, slope = 1 and bias = 0. Scaling is always trivial for pure integers, such as int8, and also for the true floating-point types single and double.

The integer is sometimes called the *stored integer*. This is the raw binary number, in which the binary point assumed to be at the far right of the word. In System Toolboxes, the negative of the exponent is often referred to as the *fraction length*.

The slope and bias together represent the scaling of the fixed-point number. In a number with zero bias, only the slope affects the scaling. A fixed-point number that is only scaled by binary point position is equivalent to a number in the Fixed-Point Designer [Slope Bias] representation that has a bias equal to zero and a slope adjustment equal to one. This is referred to as binary point-only scaling or power-of-two scaling:

$$real\text{-}world\ value = 2^{exponent} \times integer$$

or

$$real\text{-}world\ value = 2^{-fractionlength} \times integer$$

In System Toolbox software, you can define a fixed-point data type and scaling for the output or the parameters of many blocks by specifying the word length and fraction length of the quantity. The word length and fraction length define the whole of the data type and scaling information for binary-point only signals.

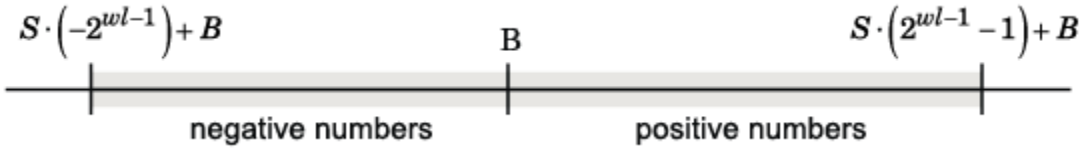
All System Toolbox blocks that support fixed-point data types support signals with binary-point only scaling. Many fixed-point blocks that do not perform arithmetic operations but merely rearrange data, such as Delay and Matrix Transpose, also support signals with [Slope Bias] scaling.

Precision and Range

You must pay attention to the precision and range of the fixed-point data types and scalings you choose for the blocks in your simulations, in order to know whether rounding methods will be invoked or if overflows will occur.

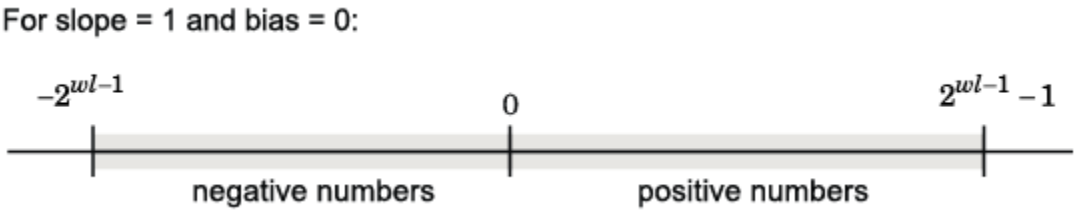
Range

The range is the span of numbers that a fixed-point data type and scaling can represent. The range of representable numbers for a two's complement fixed-point number of word length *wl*, scaling *S*, and bias *B* is illustrated below:



For both signed and unsigned fixed-point numbers of any data type, the number of different bit patterns is 2^{wl}.

For example, in two's complement, negative numbers must be represented as well as zero, so the maximum value is 2^{wl-1}. Because there is only one representation for zero, there are an unequal number of positive and negative numbers. This means there is a representation for -2^{wl-1} but not for 2^{wl-1}:



The full range is the broadest range for a data type. For floating-point types, the full range is –∞ to ∞. For integer types, the full range is the range from the smallest to largest integer value (finite) the type can represent. For example, from -128 to 127 for a signed 8-bit integer.

Overflow Handling

Because a fixed-point data type represents numbers within a finite range, overflows can occur if the result of an operation is larger or smaller than the numbers in that range.

System Toolbox software does not allow you to add guard bits to a data type on-the-fly in order to avoid overflows. Guard bits are extra bits in either a hardware register or software simulation that are added to the high end of a binary word to ensure that no information is lost in case of overflow. Any guard bits must be allocated upon model initialization. However, the software does allow you to either *saturate* or *wrap* overflows. Saturation represents positive overflows as the largest positive number in the range being used, and negative overflows as the largest negative number in the range being used. Wrapping uses modulo arithmetic to cast an overflow back into the representable range of the data type. See “Modulo Arithmetic” on page 17-8 for more information.

Precision

The precision of a fixed-point number is the difference between successive values representable by its data type and scaling, which is equal to the value of its least significant bit. The value of the least significant bit, and therefore the precision of the number, is determined by the number of fractional bits. A fixed-point value can be represented to within half of the precision of its data type and scaling. The term resolution is sometimes used as a synonym for this definition.

For example, a fixed-point representation with four bits to the right of the binary point has a precision of 2^{-4} or 0.0625, which is the value of its least significant bit. Any number within the range of this data type and scaling can be represented to within $(2^{-4})/2$ or 0.03125, which is half the precision. This is an example of representing a number with finite precision.

Rounding Modes

When you represent numbers with finite precision, not every number in the available range can be represented exactly. If a number cannot be represented exactly by the specified data type and scaling, it is *rounded* to a representable number. Although precision is always lost in the rounding operation, the cost of the operation and the amount of bias that is introduced depends on the rounding mode itself. To provide you with greater flexibility in the trade-off between cost and bias, DSP System Toolbox software currently supports the following rounding modes:

- `Ceiling` rounds the result of a calculation to the closest representable number in the direction of positive infinity.
- `Convergent` rounds the result of a calculation to the closest representable number. In the case of a tie, `Convergent` rounds to the nearest even number. This is the least biased rounding mode provided by the toolbox.
- `Floor`, which is equivalent to truncation, rounds the result of a calculation to the closest representable number in the direction of negative infinity. The truncation operation results in dropping of one or more least significant bits from a number.
- `Nearest` rounds the result of a calculation to the closest representable number. In the case of a tie, `Nearest` rounds to the closest representable number in the direction of positive infinity.
- `Round` rounds the result of a calculation to the closest representable number. In the case of a tie, `Round` rounds positive numbers to the closest representable number in the direction of positive infinity, and rounds negative numbers to the closest representable number in the direction of negative infinity.
- `Simplest` rounds the result of a calculation using the rounding mode (`Floor` or `Zero`) that adds the least amount of extra rounding code to your generated code. For more information, see “Rounding Mode: Simplest” (Fixed-Point Designer).
- `Zero` rounds the result of a calculation to the closest representable number in the direction of zero.

To learn more about each of these rounding modes, see “Rounding” (Fixed-Point Designer).

For a direct comparison of the rounding modes, see “Choosing a Rounding Method” (Fixed-Point Designer).

Arithmetic Operations

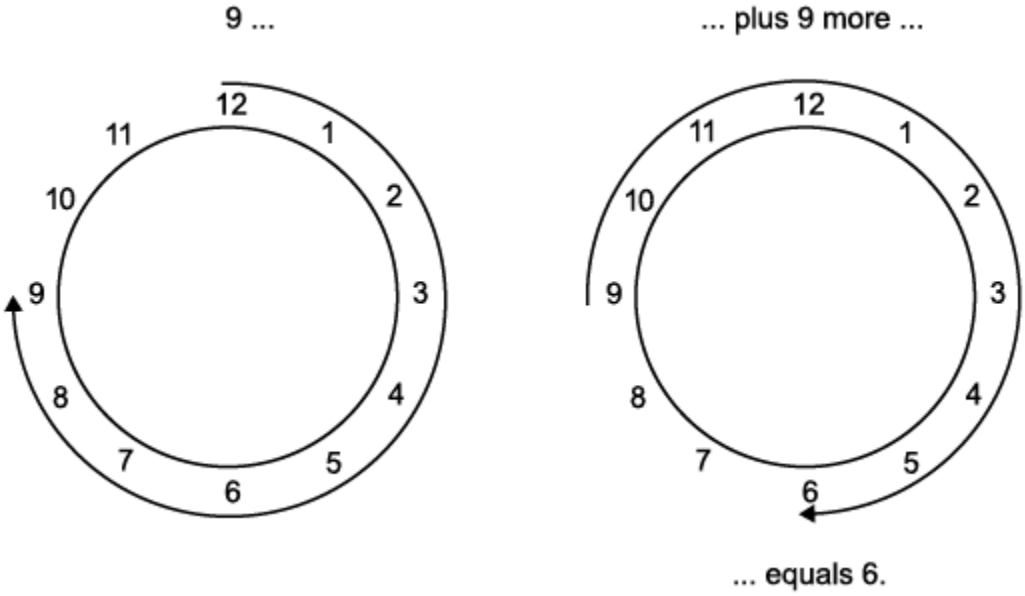
In this section...
“Modulo Arithmetic” on page 17-8
“Two's Complement” on page 17-8
“Addition and Subtraction” on page 17-9
“Multiplication” on page 17-10
“Casts” on page 17-12

Note These sections will help you understand what data type and scaling choices result in overflows or a loss of precision.

Modulo Arithmetic

Binary math is based on modulo arithmetic. Modulo arithmetic uses only a finite set of numbers, wrapping the results of any calculations that fall outside the given set back into the set.

For example, the common everyday clock uses modulo 12 arithmetic. Numbers in this system can only be 1 through 12. Therefore, in the “clock” system, 9 plus 9 equals 6. This can be more easily visualized as a number circle:



Similarly, binary math can only use the numbers 0 and 1, and any arithmetic results that fall outside this range are wrapped “around the circle” to either 0 or 1.

Two's Complement

Two's complement is a common representation of signed fixed-point numbers. In two's complement, positive numbers always start with a 0 and negative numbers always start with a 1. If the leading bit

of a two's complement number is 0, the value is obtained by calculating the standard binary value of the number. If the leading bit of a two's complement number is 1, the value is obtained by assuming that the leftmost bit is negative, and then calculating the binary value of the number. For example,

$$01 = (0 + 2^0) = 1$$
$$11 = ((-2^1) + (2^0)) = (-2 + 1) = -1$$

To compute the negative of a binary number using two's complement,

1. Take the one's complement. That is, all 0's are flipped to 1's and all 1's are flipped to 0's.
2. Add a 1 using binary math.
3. Discard any bits carried beyond the original word length.

For example, consider taking the negative of 11010 (-6). First, take the one's complement of the number, or flip the bits:

$$11010 \rightarrow 00101$$

Next, add a 1, wrapping all numbers to 0 or 1:

```
00101
  +1
00110  (6)
```

Addition and Subtraction

The addition of fixed-point numbers requires that the binary points of the addends be aligned. The addition is then performed using binary arithmetic so that no number other than 0 or 1 is used.

For example, consider the addition of 010010.1 (18.5) with 0110.110 (6.75):

```
 010010.1     (18.5)
+0110.110     (6.75)
011001.010    (25.25)
```

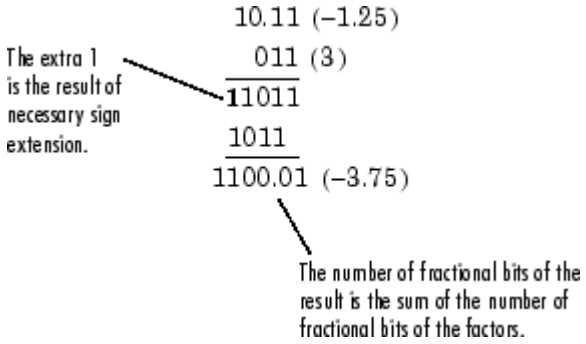
Fixed-point subtraction is equivalent to adding while using the two's complement value for any negative values. In subtraction, the addends must be sign extended to match each other's length. For example, consider subtracting 0110.110 (6.75) from 010010.1 (18.5):

Most fixed-point DSP System Toolbox blocks that perform addition cast the adder inputs to an accumulator data type before performing the addition. Therefore, no further shifting is necessary during the addition to line up the binary points. See “Casts” on page 17-12 for more information.

Multiplication

The multiplication of two's complement fixed-point numbers is directly analogous to regular decimal multiplication, with the exception that the intermediate results must be sign extended so that their left sides align before you add them together.

For example, consider the multiplication of 10.11 (-1.25) with 011 (3):



Multiplication Data Types

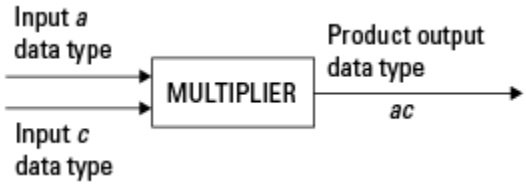
The following diagrams show the data types used for fixed-point multiplication in the System Toolbox software. The diagrams illustrate the differences between the data types used for real-real, complex-real, and complex-complex multiplication. See individual reference pages to determine whether a particular block accepts complex fixed-point inputs.

In most cases, you can set the data types used during multiplication in the block mask. For details, see “Casts” on page 17-12.

Note The following diagrams show the use of fixed-point data types in multiplication in System Toolbox software. They do not represent actual subsystems used by the software to perform multiplication.

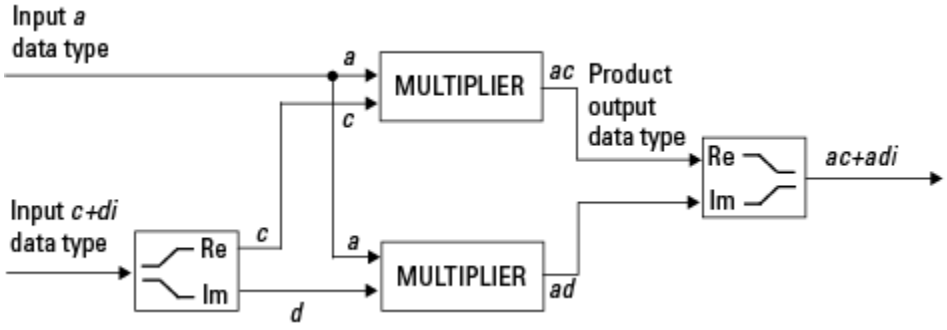
Real-Real Multiplication

The following diagram shows the data types used in the multiplication of two real numbers in System Toolbox software. The software returns the output of this operation in the product output data type, as the next figure shows.



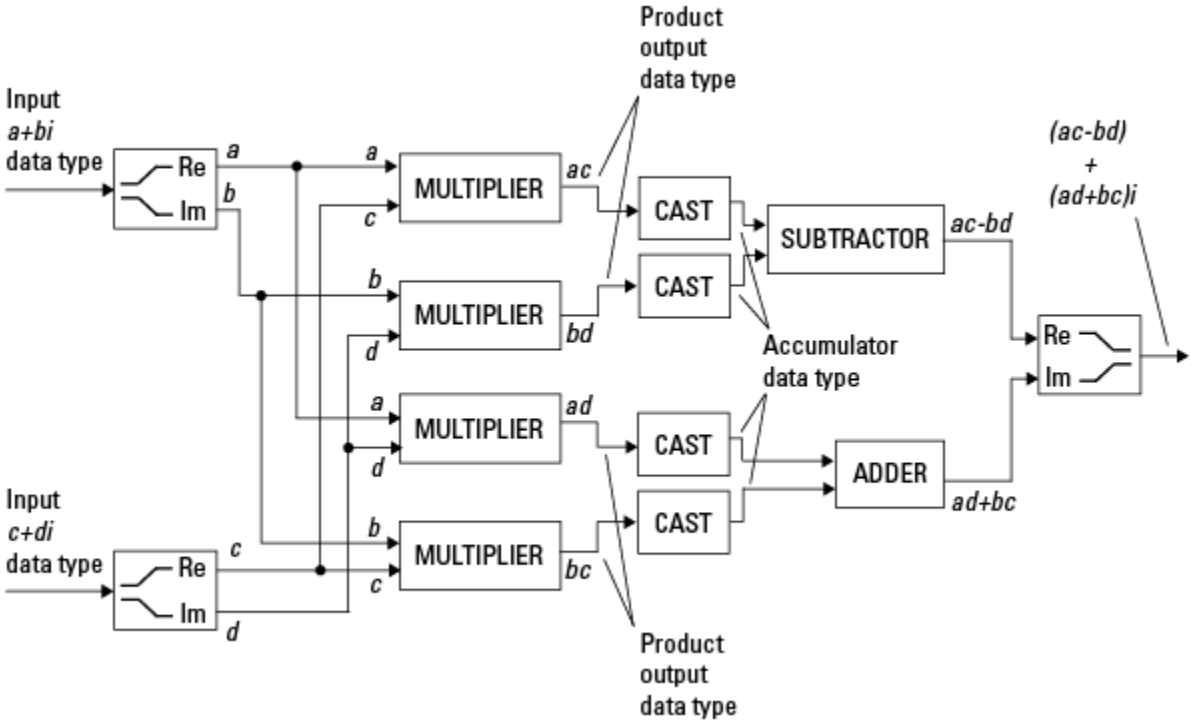
Real-Complex Multiplication

The following diagram shows the data types used in the multiplication of a real and a complex fixed-point number in System Toolbox software. Real-complex and complex-real multiplication are equivalent. The software returns the output of this operation in the product output data type, as the next figure shows.



Complex-Complex Multiplication

The following diagram shows the multiplication of two complex fixed-point numbers in System Toolbox software. Note that the software returns the output of this operation in the accumulator output data type, as the next figure shows.



System Toolbox blocks cast to the accumulator data type before performing addition or subtraction operations. In the preceding diagram, this is equivalent to the C code

```
acc=ac;
acc-=bd;
```

for the subtractor, and

```
acc=ad;
acc+=bc;
```

for the adder, where *acc* is the accumulator.

Casts

Many fixed-point System Toolbox blocks that perform arithmetic operations allow you to specify the accumulator, intermediate product, and product output data types, as applicable, as well as the output data type of the block. This section gives an overview of the casts to these data types, so that you can tell if the data types you select will invoke sign extension, padding with zeros, rounding, and/or overflow. Sign extension is the addition of bits that have the value of the most significant bit to the high end of a two's complement number. Sign extension does not change the value of the binary number. Padding is extending the least significant bit of a binary word with one or more zeros.

Casts to the Accumulator Data Type

For most fixed-point System Toolbox blocks that perform addition or subtraction, the operands are first cast to an accumulator data type. Most of the time, you can specify the accumulator data type on the block mask. For details, see the description for **Accumulator** data type parameter in “Specify Fixed-Point Attributes for Blocks” on page 17-33. Since the addends are both cast to the same accumulator data type before they are added together, no extra shift is necessary to insure that their binary points align. The result of the addition remains in the accumulator data type, with the possibility of overflow.

Casts to the Intermediate Product or Product Output Data Type

For System Toolbox blocks that perform multiplication, the output of the multiplier is placed into a product output data type. Blocks that then feed the product output back into the multiplier might first cast it to an intermediate product data type. Most of the time, you can specify these data types on the block mask. For details, see the description for **Intermediate Product** and **Product Output** data type parameters in “Specify Fixed-Point Attributes for Blocks” on page 17-33.

Casts to the Output Data Type

Many fixed-point System Toolbox blocks allow you to specify the data type and scaling of the block output on the mask. Remember that the software does not allow mixed types on the input and output ports of its blocks. Therefore, if you would like to specify a fixed-point output data type and scaling for a System Toolbox block that supports fixed-point data types, you must feed the input port of that block with a fixed-point signal. The final cast made by a fixed-point System Toolbox block is to the output data type of the block.

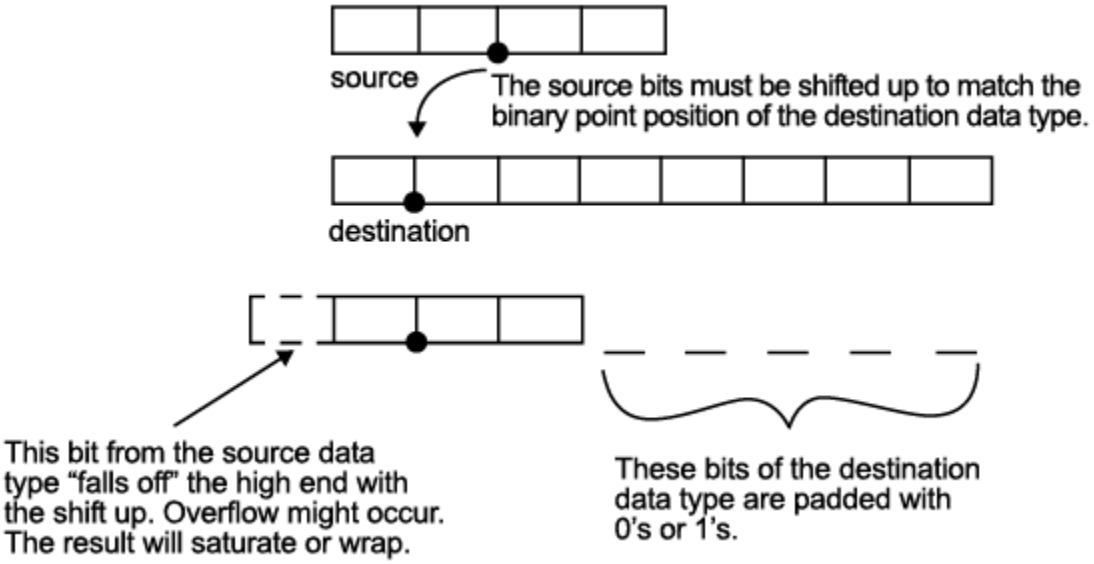
Note that although you cannot mix fixed-point and floating-point signals on the input and output ports of blocks, you can have fixed-point signals with different word and fraction lengths on the ports of blocks that support fixed-point signals.

Casting Examples

It is important to keep in mind the ramifications of each cast when selecting these intermediate data types, as well as any other intermediate fixed-point data types that are allowed by a particular block. Depending upon the data types you select, overflow and/or rounding might occur. The following two examples demonstrate cases where overflow and rounding can occur.

Cast from a Shorter Data Type to a Longer Data Type

Consider the cast of a nonzero number, represented by a four-bit data type with two fractional bits, to an eight-bit data type with seven fractional bits:



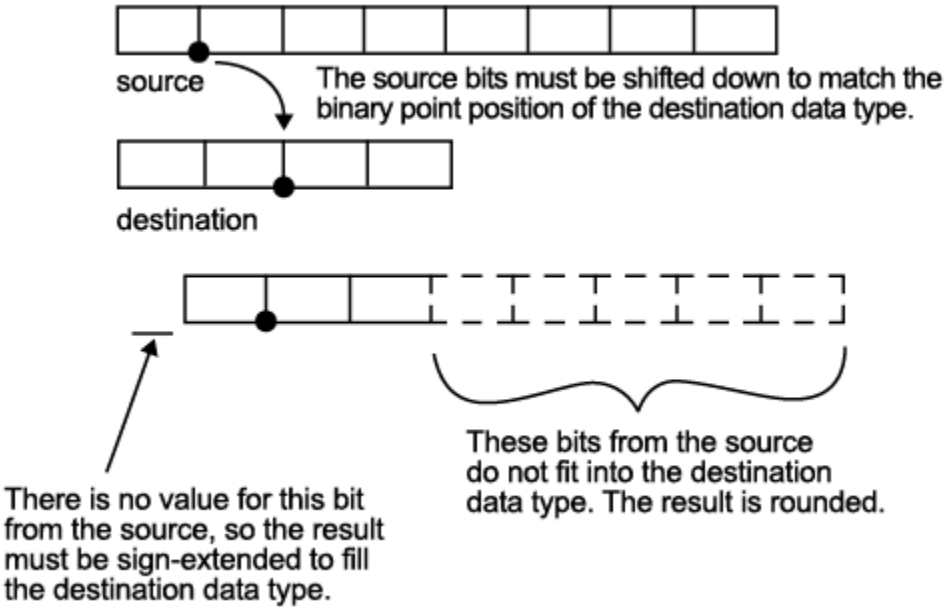
As the diagram shows, the source bits are shifted up so that the binary point matches the destination binary point position. The highest source bit does not fit, so overflow might occur and the result can saturate or wrap. The empty bits at the low end of the destination data type are padded with either 0's or 1's:

- If overflow does not occur, the empty bits are padded with 0's.
- If wrapping occurs, the empty bits are padded with 0's.
- If saturation occurs,
  - The empty bits of a positive number are padded with 1's.
  - The empty bits of a negative number are padded with 0's.

You can see that even with a cast from a shorter data type to a longer data type, overflow might still occur. This can happen when the integer length of the source data type (in this case two) is longer than the integer length of the destination data type (in this case one). Similarly, rounding might be necessary even when casting from a shorter data type to a longer data type, if the destination data type and scaling has fewer fractional bits than the source.

Cast from a Longer Data Type to a Shorter Data Type

Consider the cast of a nonzero number, represented by an eight-bit data type with seven fractional bits, to a four-bit data type with two fractional bits:



As the diagram shows, the source bits are shifted down so that the binary point matches the destination binary point position. There is no value for the highest bit from the source, so the result is sign extended to fill the integer portion of the destination data type. The bottom five bits of the source do not fit into the fraction length of the destination. Therefore, precision can be lost as the result is rounded.

In this case, even though the cast is from a longer data type to a shorter data type, all the integer bits are maintained. Conversely, full precision can be maintained even if you cast to a shorter data type, as long as the fraction length of the destination data type is the same length or longer than the fraction length of the source data type. In that case, however, bits are lost from the high end of the result and overflow might occur.

The worst case occurs when both the integer length and the fraction length of the destination data type are shorter than those of the source data type and scaling. In that case, both overflow and a loss of precision can occur.

System Objects in DSP System Toolbox that Support Fixed-Point

In this section...
“Get Information About Fixed-Point System Objects” on page 17-15
“Set System Object Fixed-Point Properties” on page 17-18
“Full Precision for Fixed-Point System Objects” on page 17-19

Get Information About Fixed-Point System Objects

System objects that support fixed-point data processing have fixed-point properties. When you display the properties of a System object, click `show all properties` at the end of the property list to display the fixed-point properties for that object. You can also display the fixed-point properties for a particular object by typing `dsp.<ObjectName>.helpFixedPoint` at the MATLAB command line.

DSP System Toolbox System Objects That Support Fixed Point

Object	Description
Sources	
`dsp.SignalSource`	Import a variable from the MATLAB workspace
`dsp.SineWave`	Generate discrete sine wave
Sinks	
`dsp.ArrayPlot`	Display vectors or arrays
`dsp.AudioFileWriter`	Write audio samples to audio file
`dsp.SignalSink`	Log MATLAB simulation data
`dsp.SpectrumAnalyzer`	Display frequency spectrum of time-domain signals
`dsp.TimeScope`	Display time-domain signals
Adaptive Filters	
`dsp.LMSFilter`	Compute output, error, and weights using LMS adaptive algorithm
Filter Designs	
`dsp.CICCompensationDecimator`	Compensate for CIC filter using a FIR decimator
`dsp.CICCompensationInterpolator`	Compensate for CIC filter using a FIR interpolator
`dsp.Differentiator`	Direct form FIR full band differentiator filter
`dsp.FIRHalfbandDecimator`	Halfband decimator
`dsp.FIRHalfbandInterpolator`	Halfband interpolator
`dsp.HighpassFilter`	FIR or IIR highpass filter
`dsp.LowpassFilter`	FIR or IIR lowpass filter
Filter Implementations	
`dsp.AllpoleFilter`	IIR Filter with no zeros
`dsp.BiquadFilter`	Model biquadratic IIR (SOS) filters
`dsp.FIRFilter`	Static or time-varying FIR filter
`dsp.IIRFilter`	Infinite Impulse Response (IIR) filter
Multirate Filters	
`dsp.CICDecimator`	Decimate inputs using a Cascaded Integrator-Comb (CIC) filter
`dsp.CICInterpolator`	Interpolate inputs using a Cascaded Integrator-Comb (CIC) filter
`dsp.FIRDecimator`	Filter and downsample input signals
`dsp.FIRInterpolator`	Upsample and filter input signals
`dsp.FIRRateConverter`	Upsample, filter, and downsample input signals
`dsp.HDLFIRRateConverter`	Upsample, filter, and downsample—optimized for HDL code generation
`dsp.SubbandAnalysisFilter`	Decompose signal into high-frequency and low-frequency subbands

Object	Description
`dsp.SubbandSynthesisFilter`	Reconstruct a signal from high-frequency and low-frequency subbands
Linear Prediction	
`dsp.LevinsonSolver`	Solve linear system of equations using Levinson-Durbin recursion
Transforms	
`dsp.FFT`	Compute fast Fourier transform (FFT) of input
`dsp.HDLFFT`	Compute fast Fourier transform (FFT) of input — optimized for HDL Code generation
`dsp.HDLIFFT`	Compute inverse fast Fourier transform (IFFT) of input — optimized for HDL Code generation
`dsp.IFFT`	Compute inverse fast Fourier transform (IFFT) of input
Statistics	
`dsp.Autocorrelator`	Compute autocorrelation of vector inputs
`dsp.Crosscorrelator`	Compute cross-correlation of two inputs
Quantizers	
`dsp.ScalarQuantizerDecoder`	Convert each index value into quantized output value
`dsp.ScalarQuantizerEncoder`	Perform scalar quantization encoding
`dsp.VectorQuantizerDecoder`	Find vector quantizer codeword for given index value
`dsp.VectorQuantizerEncoder`	Perform vector quantization encoding
Signal Operations	
`dsp.Convolver`	Compute convolution of two inputs
`dsp.DCBlocker`	Remove DC component
`dsp.Delay`	Delay input by specified number of samples or frames
`dsp.DigitalDownConverter`	Translate digital signal from Intermediate Frequency (IF) band to baseband and decimate it
`dsp.DigitalUpConverter`	Interpolate digital signal and translate it from baseband to Intermediate Frequency (IF) band
`dsp.FarrowRateConverter`	Polynomial sample rate converter with arbitrary conversion factor
`dsp.HDLNCO`	Generate real or complex sinusoidal signals — optimized for HDL code generation
`dsp.NCO`	Generate real or complex sinusoidal signals
`dsp.VariableFractionalDelay`	Delay input by time-varying fractional number of sample periods
`dsp.VariableIntegerDelay`	Delay input by time-varying integer number of sample periods
`dsp.Window`	Window object
`dsp.ZeroCrossingDetector`	Zero crossing detector

Object	Description
Math Operations	
`dsp.CumulativeProduct`	Compute cumulative product of channel, column, or row elements
`dsp.CumulativeSum`	Compute cumulative sum of channel, column, or row elements
`dsp.HDLComplexToMagnitudeAngle`	Compute magnitude and phase angle of complex signal—optimized for HDL code generation
Matrix Operations	
`dsp.ArrayVectorAdder`	Add vector to array along specified dimension
`dsp.ArrayVectorDivider`	Divide array by vector along specified dimension
`dsp.ArrayVectorMultiplier`	Multiply array by vector along specified dimension
`dsp.ArrayVectorSubtractor`	Subtract vector from array along specified dimension
Matrix Factorizations	
`dsp.LDLFactor`	Factor square Hermitian positive definite matrices into lower, upper, and diagonal components
`dsp.LUFactor`	Factor square matrix into lower and upper triangular matrices
Linear System Solvers	
`dsp.LowerTriangularSolver`	Solve LX = B for X when L is lower triangular matrix
`dsp.UpperTriangularSolver`	Solve UX = B for X when U is upper triangular matrix
Switches and Counters	
`dsp.Counter`	Count up or down through specified range of numbers

Set System Object Fixed-Point Properties

Several properties affect the fixed-point data processing used by a System object. Objects perform fixed-point processing and use the current fixed-point property settings when they receive fixed-point input.

You change the values of fixed-point properties in the same way as you change any System object property value. See “Configure Components” (MATLAB). You also use the Fixed-Point Designer `numerictype` object to specify the desired data type as fixed-point, the signedness, and the word- and fraction-lengths. System objects support these values of `DataTypeMode`: `Boolean`, `Double`, `Single`, and `Fixed-point: binary point scaling`.

In the same way as for blocks, the data type properties of many System objects can set the appropriate word lengths and scalings automatically by using full precision. System objects assume that the target specified on the Configuration Parameters Hardware Implementation target is `ASIC/FPGA`.

If you have not set the property that activates a dependent property and you attempt to change that dependent property, a warning message displays. For example, for the `dsp.FFT` object, before you set `CustomOutputDataType` to `numerictype(1,32,30)`, set `OutputDataType` to `'Custom'`.

Note System objects do not support fixed-point word lengths greater than 128 bits.

For any System object provided in the Toolbox, the fimath settings for any fimath attached to a fi input or a fi property are ignored. Outputs from a System object never have an attached fimath.

Full Precision for Fixed-Point System Objects

`FullPrecisionOverride` is a convenience property that, when you set to `true`, automatically sets the appropriate properties for an object to use full-precision to process fixed-point input. For System objects, full precision, fixed-point operation refers to growing just enough additional bits to compute the ideal full precision result. This operation has no minimum or maximum range overflow nor any precision loss due to rounding or underflow. It is also independent of any hardware-specific settings. The data types chosen are based only on known data type ranges and not on actual numeric values. Full precision for System objects does not optimize coefficient values.

When you set the `FullPrecisionOverride` property to `true`, the other fixed-point properties it controls no longer apply and any of their non-default values are ignored. These properties are also hidden. To specify individual fixed-point properties, first set `FullPrecisionOverride` to `false`.

See Also

More About

- “Simulink Blocks in DSP System Toolbox that Support Fixed-Point” on page 17-20

Simulink Blocks in DSP System Toolbox that Support Fixed-Point

This section lists the Simulink blocks in DSP System Toolbox which support fixed-point operations. This information is also available in the Simulink block data type support table for DSP System Toolbox. To access this table, type the command below in the MATLAB command prompt.

```
showsignalblockdatatypetable
```

Block	Description
Sources	
Constant	Generate constant value
Discrete Impulse	Generate discrete impulse
Identity Matrix	Generate matrix with ones on main diagonal and zeros elsewhere
NCO	Generate real or complex sinusoidal signals
NCO HDL Optimized	Generate real or complex sinusoidal signals—optimized for HDL code generation
Signal From Workspace	Import signal from MATLAB workspace
Sine Wave	Generate continuous or discrete sine wave
Sinks	
Array Plot	Display vectors or arrays
Display	Show value of input
Matrix Viewer	Display matrices as color images
Spectrum Analyzer	Display frequency spectrum of time-domain signals
Time Scope	Display time-domain signals
To Workspace	Write data to MATLAB workspace
Triggered To Workspace	Write input sample to MATLAB workspace when triggered
Waterfall	View vectors of data over time
Adaptive Filters	
LMS Filter	Compute output, error, and weights using LMS adaptive algorithm
Filter Designs	
CIC Compensation Decimator	Compensate for CIC filter using FIR decimator
CIC Compensation Interpolator	Compensate for CIC filter using FIR interpolator
Differentiator Filter	Direct form FIR full band differentiator filter
FIR Halfband Decimator	Decimate signal using polyphase FIR halfband filter
FIR Halfband Interpolator	Interpolate signal using polyphase FIR half band filter

Block	Description
Highpass Filter	Design FIR or IIR highpass filter
Lowpass Filter	Design FIR or IIR lowpass filter
Filter Implementations	
Allpole Filter	Model allpole filters
Biquad Filter	Model biquadratic IIR (SOS) filters
Discrete FIR Filter	Model FIR filters
Discrete Filter	Model Infinite Impulse Response (IIR) filters
Filter Realization Wizard	Construct filter realizations using digital filter blocks or Sum, Gain, and Delay blocks
Multirate Filters	
CIC Decimation	Decimate signal using Cascaded Integrator-Comb filter
CIC Interpolation	Interpolate signal using Cascaded Integrator-Comb filter
FIR Decimation	Filter and downsample input signals
FIR Interpolation	Upsample and filter input signals
FIR Rate Conversion	Upsample, filter, and downsample input signals
FIR Rate Conversion HDL Optimized	Upsample, filter, and downsample input signals—optimized for HDL code generation
Two-Channel Analysis Subband Filter	Decompose signal into high-frequency and low-frequency subbands
Two-Channel Synthesis Subband Filter	Reconstruct signal from high-frequency and low-frequency subbands
Linear Prediction	
Levinson-Durbin	Solve linear system of equations using Levinson-Durbin recursion
Transforms	
DCT	Discrete cosine transform (DCT) of input
FFT	Fast Fourier transform (FFT) of input
FFT HDL Optimized	Fast Fourier transform—optimized for HDL code generation
IDCT	Inverse discrete cosine transform (IDCT) of input
IFFT	Inverse fast Fourier transform (IFFT) of input
IFFT HDL Optimized	Inverse fast Fourier transform—optimized for HDL code generation
Magnitude FFT	Compute nonparametric estimate of spectrum using periodogram method
Short-Time FFT	Nonparametric estimate of spectrum using short-time, fast Fourier transform (FFT) method

Block	Description
Statistics	
Autocorrelation	Autocorrelation of vector or matrix input
Correlation	Cross-correlation of two inputs
Histogram	Generate histogram of input or sequence of inputs
Maximum	Find maximum values in input or sequence of inputs
Mean	Find mean value of input or sequence of inputs
Median	Find median value of input
Minimum	Find minimum values in input or sequence of inputs
Sort	Sort input elements by value
Variance	Compute variance of input or sequence of inputs
Quantizers	
Scalar Quantizer Decoder	Convert each index value into quantized output value
Scalar Quantizer Encoder	Encode each input value by associating it with index value of quantization region
Vector Quantizer Decoder	Find vector quantizer codeword that corresponds to given, zero-based index value
Vector Quantizer Encoder	For given input, find index of nearest codeword based on Euclidean or weighted Euclidean distance measure
Signal Operations	
Constant Ramp	Generate ramp signal with length based on input dimensions
Convolution	Convolution of two inputs
DC Blocker	lock DC component
Digital Down-Converter	Translate digital signal from Intermediate Frequency (IF) band to baseband and decimate it
Digital Up-Converter	Interpolate digital signal and translate it from baseband to Intermediate Frequency (IF) band
Downsample	Resample input at lower rate by deleting samples
Farrow Rate Converter	Polynomial sample-rate converter with arbitrary conversion factor
NCO	Generate real or complex sinusoidal signals
NCO HDL Optimized	Generate real or complex sinusoidal signals—optimized for HDL code generation
Offset	Truncate vectors by removing or keeping beginning or ending values

Block	Description
Pad	Pad or truncate specified dimension(s)
Peak Finder	Determine whether each value of input signal is local minimum or maximum
Repeat	Resample input at higher rate by repeating values
Sample and Hold	Sample and hold input signal
Triggered Signal From Workspace	Import signal samples from MATLAB workspace when triggered
Upsample	Resample input at higher rate by inserting zeros
Variable Fractional Delay	Delay input by time-varying fractional number of sample periods
Variable Integer Delay	Delay input by time-varying integer number of sample periods
Window Function	Compute and/or apply window to input signal
Zero Crossing	Count number of times signal crosses zero in single time step
Math Operations	
Complex to Magnitude-Angle HDL Optimized	Compute magnitude and/or phase angle of complex signal—optimized for HDL code generation using the CORDIC algorithm
Cumulative Product	Cumulative product of channel, column, or row elements
Cumulative Sum	Cumulative sum of channel, column, or row elements
Difference	Compute element-to-element difference along specified dimension of input
Normalization	Perform vector normalization along rows, columns, or specified dimension
dB Gain	Apply decibel gain
Matrix Operations	
Array-Vector Add	Add vector to array along specified dimension
Array-Vector Divide	Divide array by vector along specified dimension
Array-Vector Multiply	Multiply array by vector along specified dimension
Array-Vector Subtract	Subtract vector from array along specified dimension
Create Diagonal Matrix	Create square diagonal matrix from diagonal elements
Extract Diagonal	Extract main diagonal of input matrix
Extract Triangular Matrix	Extract lower or upper triangle from input matrices

Block	Description
Identity Matrix	Generate matrix with ones on main diagonal and zeros elsewhere
Matrix Concatenate	Concatenate input signals of same data type to create contiguous output signal
Matrix Product	Multiply matrix elements along rows, columns, or entire input
Matrix Square	Compute square of input matrix
Matrix Sum	Sum matrix elements along rows, columns, or entire input
Matrix 1-Norm	Compute 1-norm of matrix
Matrix Multiply	Multiply or divide inputs
Overwrite Values	Overwrite submatrix or subdiagonal of input
Permute Matrix	Reorder matrix rows or columns
Submatrix	Select subset of elements (submatrix) from matrix input
Toeplitz	Generate matrix with Toeplitz symmetry
Matrix Factorizations	
LDL Factorization	Factor square Hermitian positive definite matrices into lower, upper, and diagonal components
LU Factorization	Factor square matrix into lower and upper triangular components
Linear System Solvers	
Backward Substitution	Solve UX=B for X when U is upper triangular matrix
Forward Substitution	Solve LX=B for X when L is lower triangular matrix
Levinson-Durbin	Solve linear system of equations using Levinson-Durbin recursion
Switches and Counters	
Edge Detector	Detect transition from zero to nonzero value
Event-Count Comparator	Detect threshold crossing of accumulated nonzero inputs
N-Sample Switch	Switch between two inputs after specified number of sample periods
Buffers	
Buffer	Buffer input sequence to smaller or larger frame size
Delay Line	Rebuffer sequence of inputs
Queue	Store inputs in FIFO register

Block	Description
Stack	Store inputs into LIFO register
Unbuffer	Unbuffer input frame into sequence of scalar outputs
Indexing	
Flip	Flip input vertically or horizontally
Multiport Selector	Distribute arbitrary subsets of input rows or columns to multiple output ports
Overwrite Values	Overwrite submatrix or subdiagonal of input
Selector	Select input elements from vector, matrix, or multidimensional signal
Submatrix	Select subset of elements (submatrix) from matrix input
Variable Selector	Select subset of rows or columns from input
Signal Attributes	
Check Signal Attributes	Error when input signal does or does not match selected attributes exactly
Convert 1-D to 2-D	Reshape 1-D or 2-D input to 2-D matrix with specified dimensions
Convert 2-D to 1-D	Convert 2-D matrix input to 1-D vector
Data Type Conversion	Convert input signal to specified data type
Frame Conversion	Specify sampling mode of output signal
Inherit Complexity	Change complexity of input to match reference signal

See Also

More About

- “System Objects in DSP System Toolbox that Support Fixed-Point” on page 17-15

System Objects Supported by Fixed-Point Converter App

You can use the Fixed-Point Converter app to automatically propose and apply data types for commonly used system objects. The proposed data types are based on simulation data from the System object.

Automated conversion is available for these DSP System Toolbox System Objects:

- `dsp.ArrayVectorAdder`
- `dsp.BiquadFilter`
- `dsp.FIRDecimator`
- `dsp.FIRInterpolator`
- `dsp.FIRFilter` (Direct Form and Direct Form Transposed only)
- `dsp.FIRRateConverter`
- `dsp.LowerTriangularSolver`
- `dsp.LUFactor`
- `dsp.UpperTriangularSolver`
- `dsp.VariableFractionalDelay`
- `dsp.Window`

The Fixed-Point Converter app can display simulation minimum and maximum values, whole number information, and histogram data.

- You cannot propose data types for these System objects based on static range data.
- You must configure the System object to use `'Custom'` fixed-point settings.
- The app applies the proposed data types only if the input signal is floating point, not fixed-point.

  The app treats scaled doubles as fixed-point. The scaled doubles workflow for System objects is the same as that for regular variables.
- The app ignores the **Default word length** setting in the **Settings** menu. The app also ignores specified rounding and overflow modes. Data-type proposals are based on the settings of the System object.

See Also

Related Examples

- “Convert dsp.FIRFilter Object to Fixed-Point Using the Fixed-Point Converter App” on page 17-27
- “Floating-Point to Fixed-Point Conversion of IIR Filters”

Convert dsp.FIRFilter Object to Fixed-Point Using the Fixed-Point Converter App

This example converts a `dsp.FIRFilter` System object, which filters a high-frequency sinusoid signal, to fixed-point using the Fixed-Point Converter app. This example requires Fixed-Point Designer and DSP System Toolbox licenses.

Create DSP Filter Function and Test Bench

Create a `myFIRFilter` function from a `dsp.FIRFilter` System object.

By default, System objects are configured to use full-precision fixed-point arithmetic. To gather range data and get data type proposals from the Fixed-Point Converter app, configure the System object to use 'Custom' settings.

Save the function to a local writable folder.

```
function output = myFIRFilter(input, num)

    persistent lowpassFIR;
    if isempty(lowpassFIR)
        lowpassFIR  = dsp.FIRFilter('NumeratorSource', 'Input port', ...
            'FullPrecisionOverride', false, ...
            'ProductDataType', 'Full precision', ... % default
            'AccumulatorDataType', 'Custom', ...
            'CustomAccumulatorDataType', numerictype(1,16,4), ...
            'OutputDataType', 'Custom', ...
            'CustomOutputDataType', numerictype(1,8,2));
    end
    output = lowpassFIR(input, num);

end
```

Create a test bench, `myFIRFilter_tb`, for the filter. The test bench generates a signal that gathers range information for conversion. Save the test bench.

```
% Test bench for myFIRFilter
% Remove high-frequency sinusoid using an FIR filter.

% Initialize
f1 = 1000;
f2 = 3000;
Fs = 8000;
Fcutoff = 2000;

% Generate input
SR = dsp.SineWave('Frequency',[f1,f2],'SampleRate',Fs,...
    'SamplesPerFrame',1024);

% Filter coefficients
num = fir1(130,Fcutoff/(Fs/2));

% Visualize input and output spectra
plot = dsp.SpectrumAnalyzer('SampleRate',Fs,'PlotAsTwoSidedSpectrum',...
    false,'ShowLegend',true,'YLimits',[-120 30],...
```

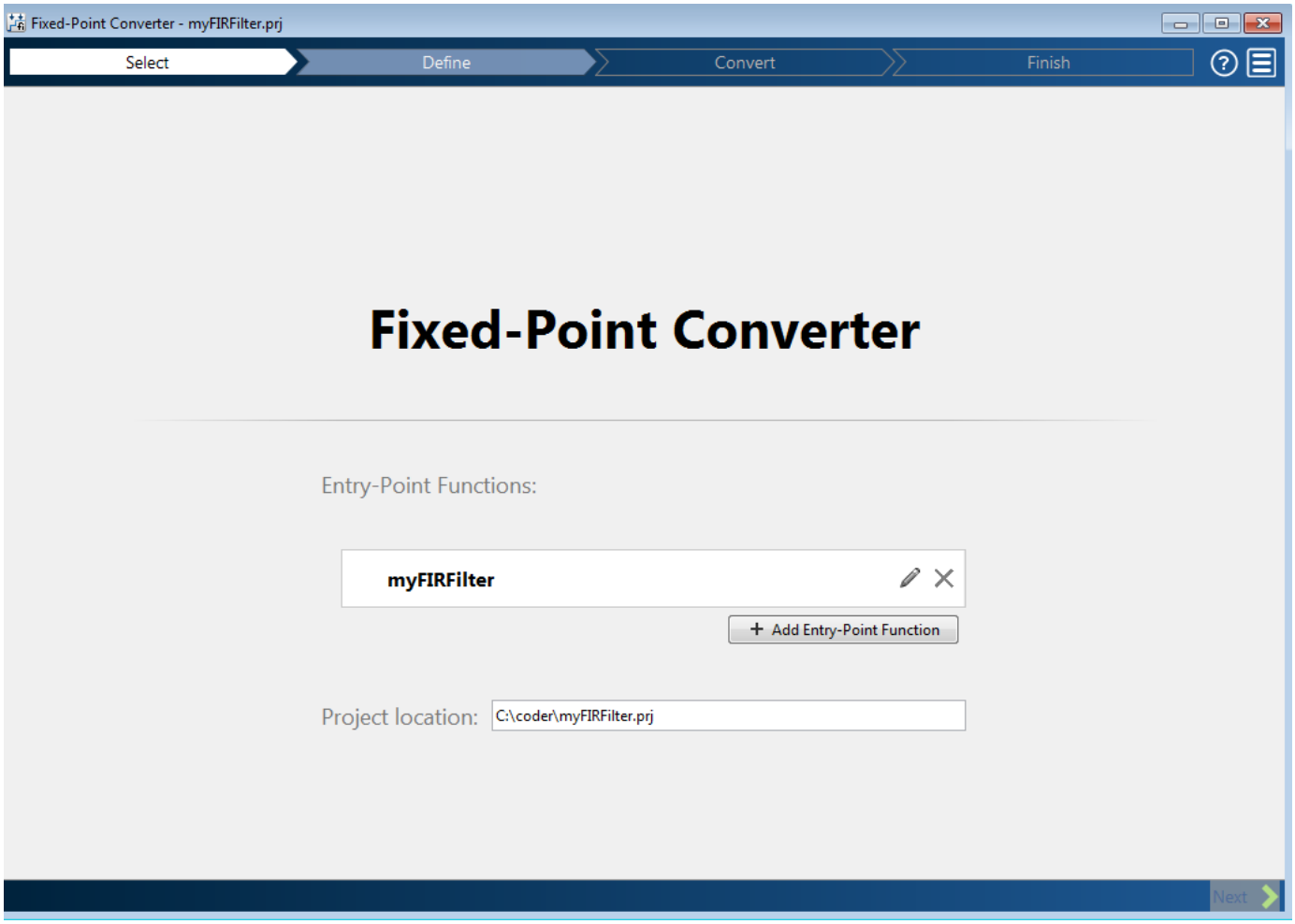
```
    'Title','Input Signal (Channel 1) Output Signal (Channel 2)');

% Stream
for k = 1:100
    input = sum(SR(),2); % Add the two sinusoids together
    filteredOutput = myFIRFilter(input, num); % Filter
    plot([input,filteredOutput]); % Visualize
end
```

Convert the Function to Fixed-Point

1. Open the Fixed-Point Converter app.
   - MATLAB Toolstrip: On the **Apps** tab, under **Code Generation**, click the app icon.
   - MATLAB command prompt: Enter

     ```
     fixedPointConverter
     ```
2. To add the entry-point function `myFIRFilter` to the project, browse to the file `myFIRFilter.m`, and then click **Open**.

   By default, the app saves information and settings for this project in the current folder in a file named `myFirFilter.prj`.

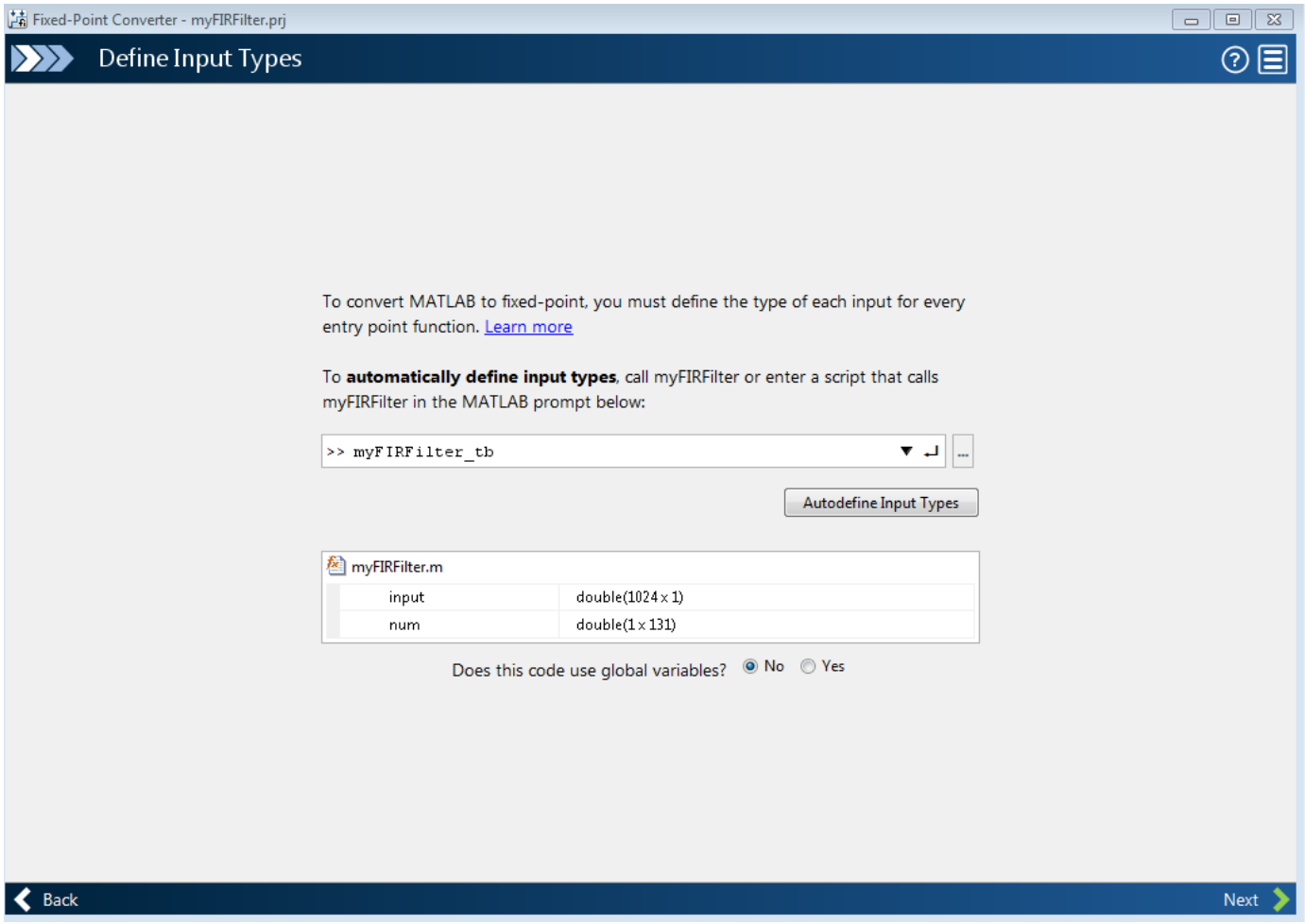


3 Click **Next** to go to the **Define Input Types** step.

The app screens `myFIRFilter.m` for code violations and readiness issues. The app does not find issues in `myFIRFilter.m`.

4 On the **Define Input Types** page, to add `myFIRFilter_tb` as a test file, browse to `myFIRFilter_tb.m`, and then click **Autodefine Input Types**.

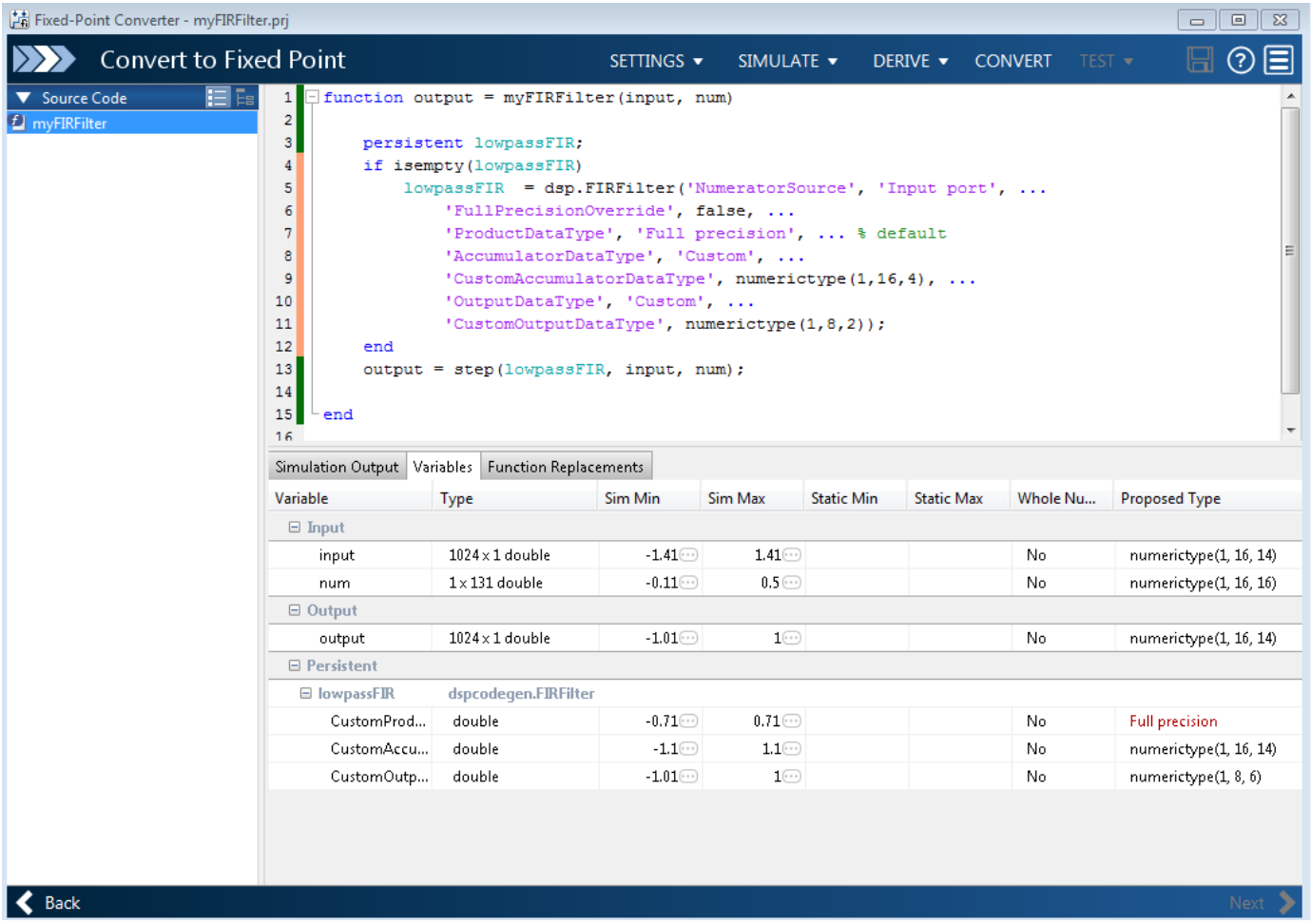
The app determines from the test file that the type of `input` is `double(1024 x 1)` and the type of `num` is `double(1 x 131)`.



5 Click **Next** to go to the **Convert to Fixed Point** step.

6 On the **Convert to Fixed Point** page, click **Simulate** to collect range information.

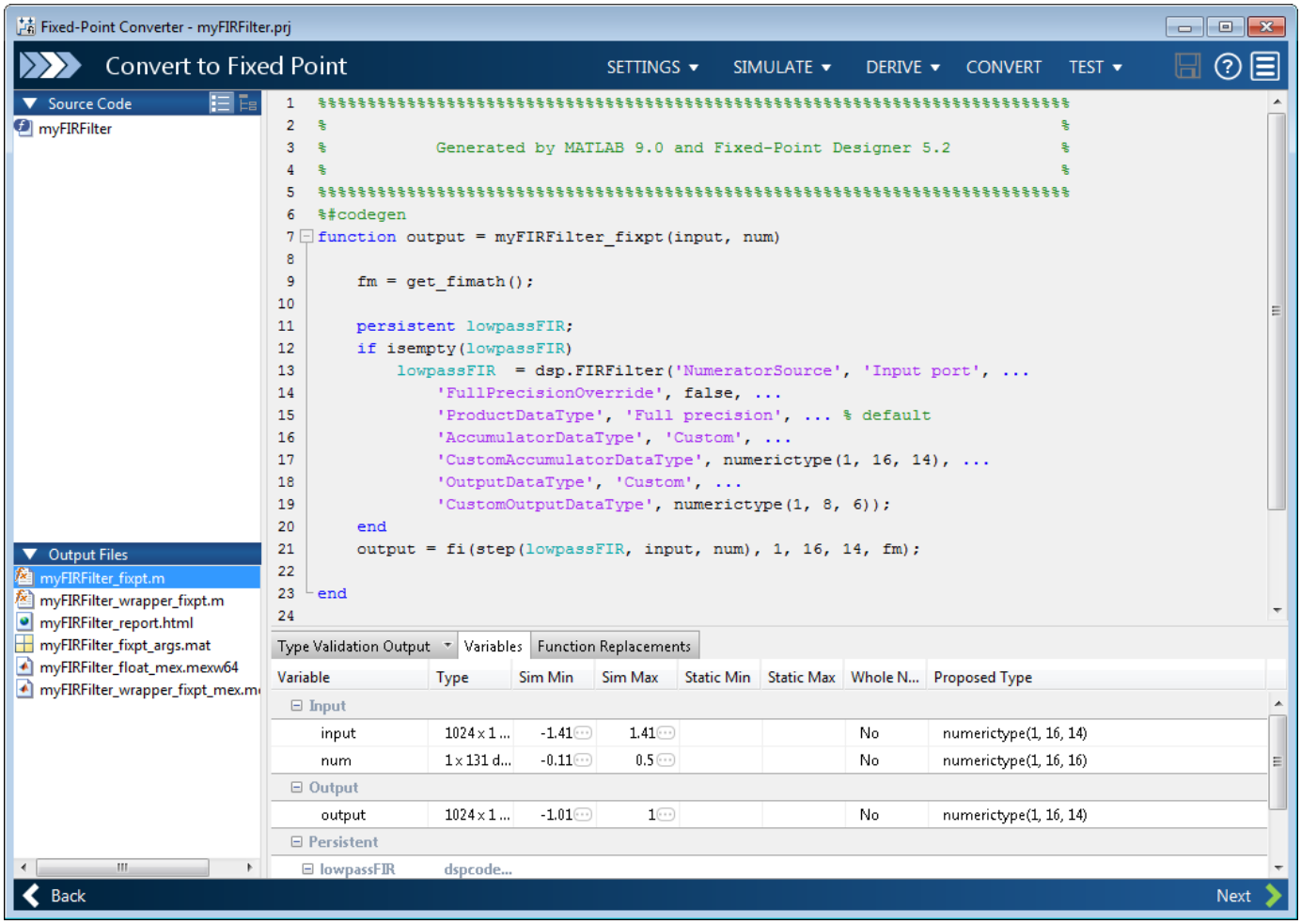
The **Variables** tab displays the collected range information and type proposals. Manually edit the data type proposals as needed.



In the **Variables** tab, the **Proposed Type** field for `lowpassFIR.CustomProductDataType` is `Full Precision`. The Fixed-Point Converter app did not propose a data type for this field because its `'ProductDataType'` setting is not set to `'Custom'`.

7 Click **Convert** to apply the proposed data types to the function.

The Fixed-Point Converter app applies the proposed data types and generates a fixed-point function, `myFIRFilter_fixpt`.




```
function output = myFIRFilter_fixpt(input, num)

    fm = get_fimath();

    persistent lowpassFIR;
    if isempty(lowpassFIR)
        lowpassFIR  = dsp.FIRFilter('NumeratorSource', 'Input port', ...
            'FullPrecisionOverride', false, ...
            'ProductDataType', 'Full precision', ... % default
            'AccumulatorDataType', 'Custom', ...
            'CustomAccumulatorDataType', numerictype(1, 16, 14), ...
            'OutputDataType', 'Custom', ...
            'CustomOutputDataType', numerictype(1, 8, 6));
    end
    output = fi(lowpassFIR(input, num), 1, 16, 14, fm);

end


function fm = get_fimath()
    fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap',...
 'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128, 'SumMode',...
 'FullPrecision', 'MaxSumWordLength', 128);
end
```

See Also

More About

- “System Objects Supported by Fixed-Point Converter App” on page 17-26
- “Floating-Point to Fixed-Point Conversion of IIR Filters”

Specify Fixed-Point Attributes for Blocks

In this section...

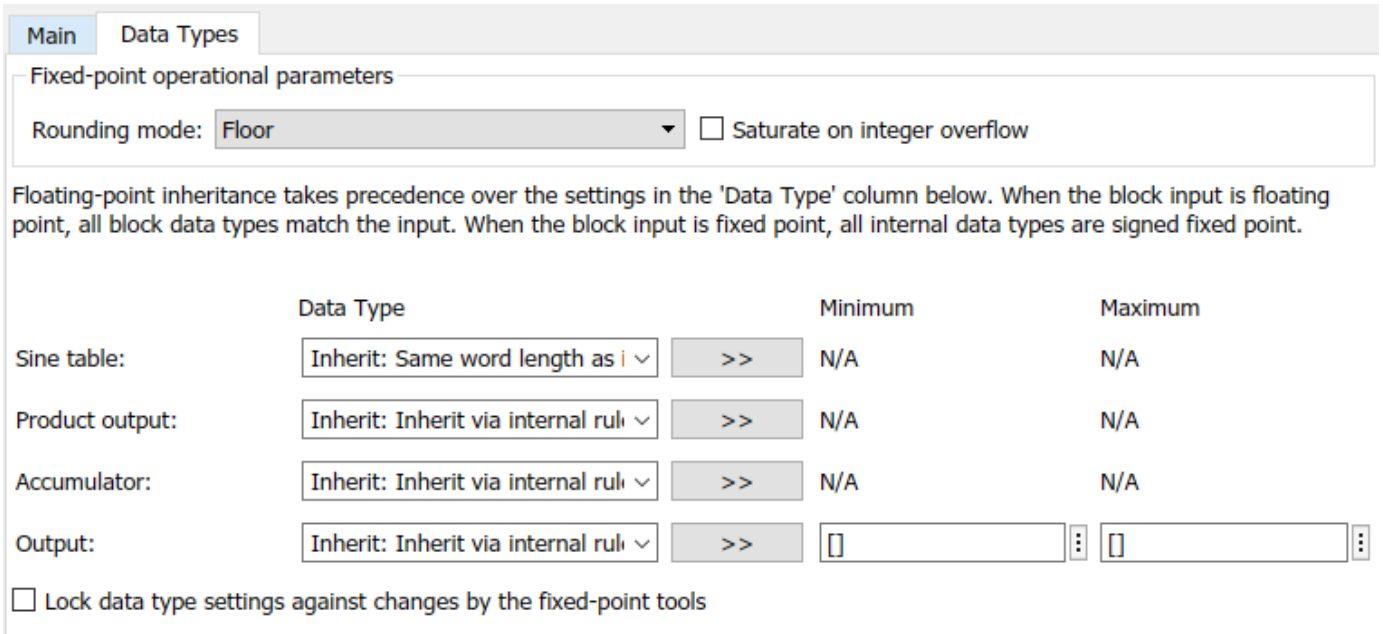
"Fixed-Point Block Parameters" on page 17-33
"Specify System-Level Settings" on page 17-35
"Inherit via Internal Rule" on page 17-35
"Specify Data Types for Fixed-Point Blocks" on page 17-42

Fixed-Point Block Parameters

Toolbox blocks that have fixed-point support usually allow you to specify fixed-point characteristics through block parameters. By specifying data type and scaling information for these fixed-point parameters, you can simulate your target hardware more closely.

Note Floating-point inheritance takes precedence over the settings discussed in this section. When the block has floating-point input, all block data types match the input.

You can find most fixed-point parameters on the **Data Types** pane of toolbox blocks. The following figure shows a typical **Data Types** pane.



All toolbox blocks with fixed-point capabilities share a set of common parameters, but each block can have a different subset of these fixed-point parameters. The following table provides an overview of the most common fixed-point block parameters.

Fixed-Point Data Type Parameter	Description
Rounding Mode	Specifies the rounding mode for the block to use when the specified data type and scaling cannot exactly represent the result of a fixed-point calculation. See “Rounding Modes” on page 17-7 for more information on the available options.
Saturate on integer overflow	When you select this parameter, the block saturates the result of its fixed-point operation. When you clear this parameter, the block wraps the result of its fixed-point operation. For details on saturate and wrap, see “Overflow Handling” on page 17-6 for fixed-point operations.
Intermediate Product	Specifies the data type and scaling of the intermediate product for fixed-point blocks. Blocks that feed multiplication results back to the input of the multiplier use the intermediate product data type. See the reference page of a specific block to learn about the intermediate product data type for that block.
Product Output	Specifies the data type and scaling of the product output for fixed-point blocks that must compute multiplication results. See the reference page of a specific block to learn about the product output data type for that block. For or complex-complex multiplication, the multiplication result is in the accumulator data type. See “Multiplication Data Types” on page 17-10 for more information on complex fixed-point multiplication in toolbox software.
Accumulator	Specifies the data type and scaling of the accumulator (sum) for fixed-point blocks that must hold summation results for further calculation. Most such blocks cast to the accumulator data type before performing the add operations (summation). See the reference page of a specific block for details on the accumulator data type of that block.
Output	Specifies the output data type and scaling for blocks.

Using the Data Type Assistant

The **Data Type Assistant** is an interactive graphical tool available on the **Data Types** pane of some fixed-point toolbox blocks.

To learn more about using the **Data Type Assistant** to help you specify block data type parameters, see “Specify Data Types Using Data Type Assistant” (Simulink).

Checking Signal Ranges

Some fixed-point toolbox blocks have **Minimum** and **Maximum** parameters on the **Data Types** pane. When a fixed-point data type has these parameters, you can use them to specify appropriate minimum and maximum values for range checking purposes.

To learn how to specify signal ranges and enable signal range checking, see “Specify Signal Ranges” (Simulink).

Specify System-Level Settings

You can monitor and control fixed-point settings for toolbox blocks at a system or subsystem level with the Fixed-Point Tool. For more information, see **Fixed-Point Tool**.

Logging

The Fixed-Point Tool logs overflows, saturations, and simulation minimums and maximums for fixed-point toolbox blocks. The Fixed-Point Tool does not log overflows and saturations when the `Data overflow` line in the **Diagnostics > Data Integrity** pane of the Configuration Parameters dialog box is set to `None`.

Autoscaling

You can use the Fixed-Point Tool autoscaling feature to set the scaling for toolbox fixed-point data types.

Data type override

toolbox blocks obey the `Use local settings`, `Double`, `Single`, and `Off` modes of the **Data type override** parameter in the Fixed-Point Tool. The `Scaled double` mode is also supported for toolboxes source and byte-shuffling blocks, and for some arithmetic blocks such as Difference and Normalization.

Scaled double is a double data type that retains fixed-point scaling information. Using the data type override, you can convert your fixed-point data types to scaled doubles. You can then simulate to determine the ideal floating-point behavior of your system. After you gather that information, you can turn data type override off to return to fixed-point data types, and your quantities still have their original scaling information because it was held in the scaled double data types.

Inherit via Internal Rule

Selecting appropriate word lengths and scalings for the fixed-point parameters in your model can be challenging. To aid you, an `Inherit via internal rule` choice is often available for fixed-point block data type parameters, such as the **Accumulator** and **Product output** signals. The following sections describe how the word and fraction lengths are selected for you when you choose `Inherit via internal rule` for a fixed-point block data type parameter in toolbox software:

- “Internal Rule for Accumulator Data Types” on page 17-36
- “Internal Rule for Product Data Types” on page 17-36
- “Internal Rule for Output Data Types” on page 17-36
- “The Effect of the Hardware Implementation Pane on the Internal Rule” on page 17-36
- “Internal Rule Examples” on page 17-37

Note In the equations in the following sections, *WL* = word length and *FL* = fraction length.

Internal Rule for Accumulator Data Types

The internal rule for accumulator data types first calculates the ideal, full-precision result. Where N is the number of addends:

$$WL_{idealaccumulator} = WL_{inputtoaccumulator} + \text{floor}(\log_2(N-1)) + 1$$

$$FL_{idealaccumulator} = FL_{inputtoaccumulator}$$

For example, consider summing all the elements of a vector of length 6 and data type sfix10_En8. The ideal, full-precision result has a word length of 13 and a fraction length of 8.

The accumulator can be real or complex. The preceding equations are used for both the real and imaginary parts of the accumulator. For any calculation, after the full-precision result is calculated, the final word and fraction lengths set by the internal rule are affected by your particular hardware. See “The Effect of the Hardware Implementation Pane on the Internal Rule” on page 17-36 for more information.

Internal Rule for Product Data Types

The internal rule for product data types first calculates the ideal, full-precision result:

$$WL_{idealproduct} = WL_{input1} + WL_{input2}$$

$$FL_{idealproduct} = FL_{input1} + FL_{input2}$$

For example, multiplying together the elements of a real vector of length 2 and data type sfix10_En8. The ideal, full-precision result has a word length of 20 and a fraction length of 16.

For real-complex multiplication, the ideal word length and fraction length is used for both the complex and real portion of the result. For complex-complex multiplication, the ideal word length and fraction length is used for the partial products, and the internal rule for accumulator data types described above is used for the final sums. For any calculation, after the full-precision result is calculated, the final word and fraction lengths set by the internal rule are affected by your particular hardware. See “The Effect of the Hardware Implementation Pane on the Internal Rule” on page 17-36 for more information.

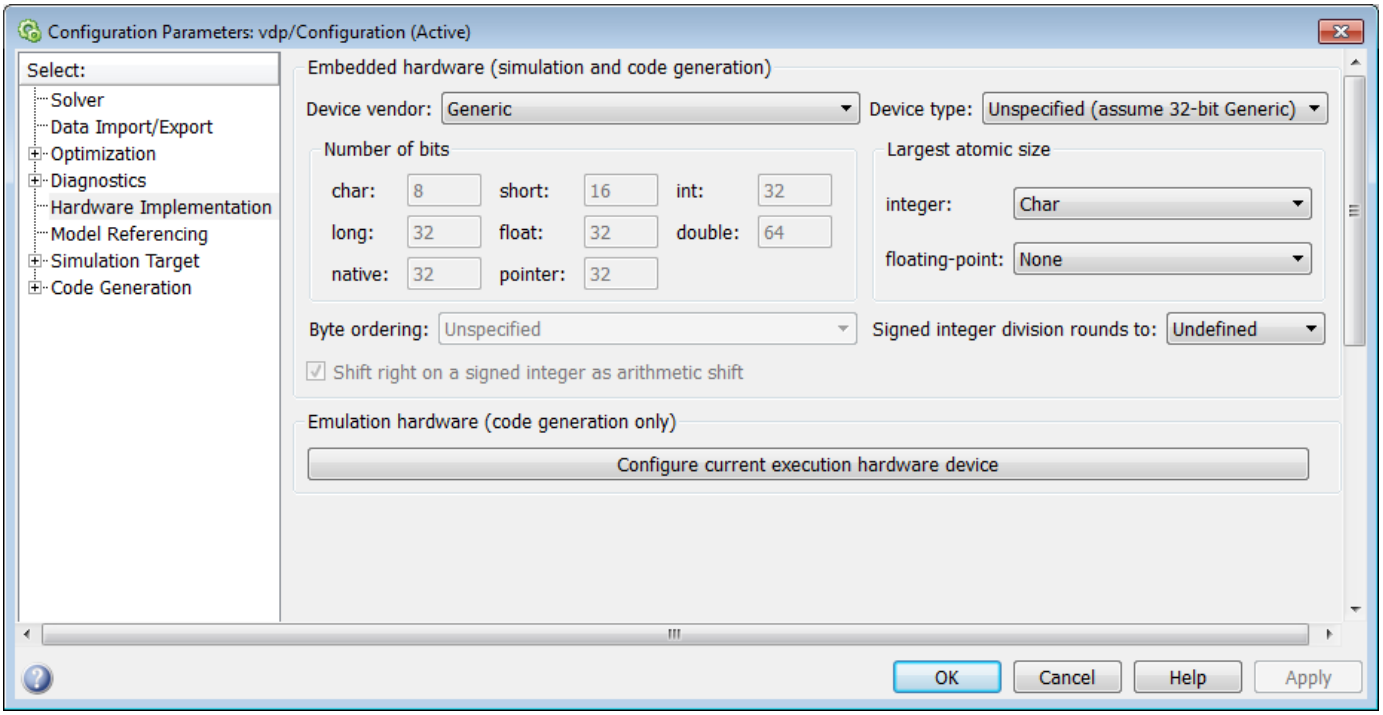
Internal Rule for Output Data Types

A few toolbox blocks have an `Inherit via internal rule` choice available for the block output. The internal rule used in these cases is block-specific, and the equations are listed in the block reference page.

As with accumulator and product data types, the final output word and fraction lengths set by the internal rule are affected by your particular hardware, as described in “The Effect of the Hardware Implementation Pane on the Internal Rule” on page 17-36.

The Effect of the Hardware Implementation Pane on the Internal Rule

The internal rule selects word lengths and fraction lengths that are appropriate for your hardware. To get the best results using the internal rule, you must specify the type of hardware you are using on the **Hardware Implementation** pane of the Configuration Parameters dialog box. You can open this dialog box from the **Simulation** menu in your model.



ASIC/FPGA

On an ASIC/FPGA target, the ideal, full-precision word length and fraction length calculated by the internal rule are used. If the calculated ideal word length is larger than the largest allowed word length, you receive an error.

Other targets

For all targets other than ASIC/FPGA, the ideal, full-precision word length calculated by the internal rule is rounded up to the next available word length of the target. The calculated ideal fraction length is used, keeping the least-significant bits.

If the calculated ideal word length for a product data type is larger than the largest word length on the target, you receive an error. If the calculated ideal word length for an accumulator or output data type is larger than the largest word length on the target, the largest target word length is used.

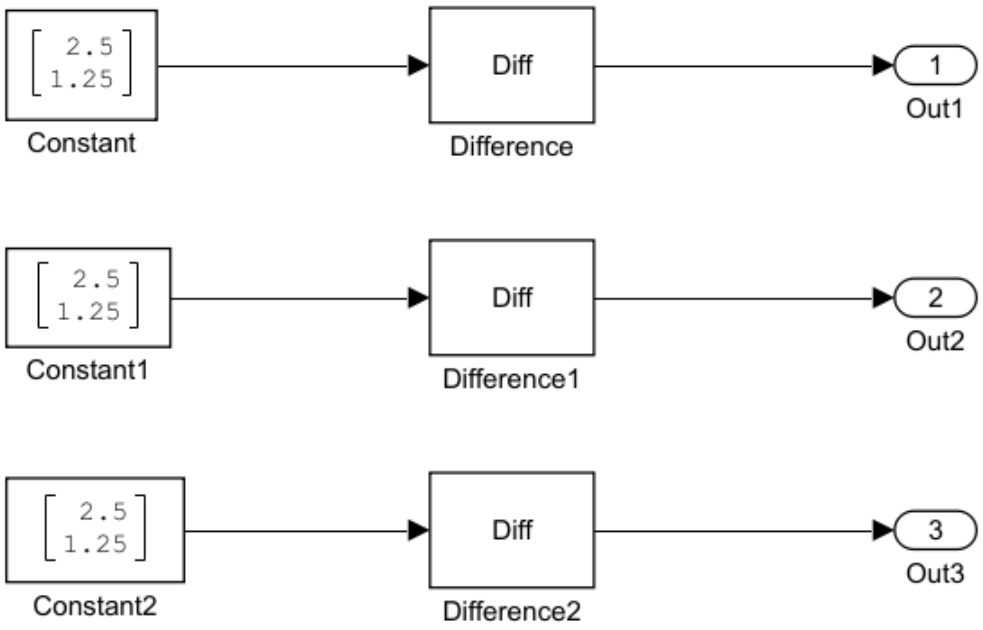
The largest word length allowed for Simulink and toolbox software on any target is 128 bits.

Internal Rule Examples

The following sections show examples of how the internal rule interacts with the **Hardware Implementation** pane to calculate accumulator data types on page 17-37 and product data types on page 17-40.

Accumulator Data Types

Consider the following model ex_internalRule_accumExp.



In the Difference blocks, the **Accumulator** parameter is set to `Inherit: Inherit via internal rule`, and the **Output** parameter is set to `Inherit: Same as accumulator`. Therefore, you can see the accumulator data type calculated by the internal rule on the output signal in the model.

In the preceding model, the **Device type** parameter in the **Hardware Implementation** pane of the Configuration Parameters dialog box is set to `ASIC/FPGA`. Therefore, the accumulator data type used by the internal rule is the ideal, full-precision result.

Calculate the full-precision word length for each of the Difference blocks in the model:

$$WL_{idealaccumulator} = WL_{inputtoaccumulator} + \text{floor}(\log_2(numberofaccumulations)) + 1$$
$$WL_{idealaccumulator} = 9 + \text{floor}(\log_2(1)) + 1$$
$$WL_{idealaccumulator} = 9 + 0 + 1 = 10$$

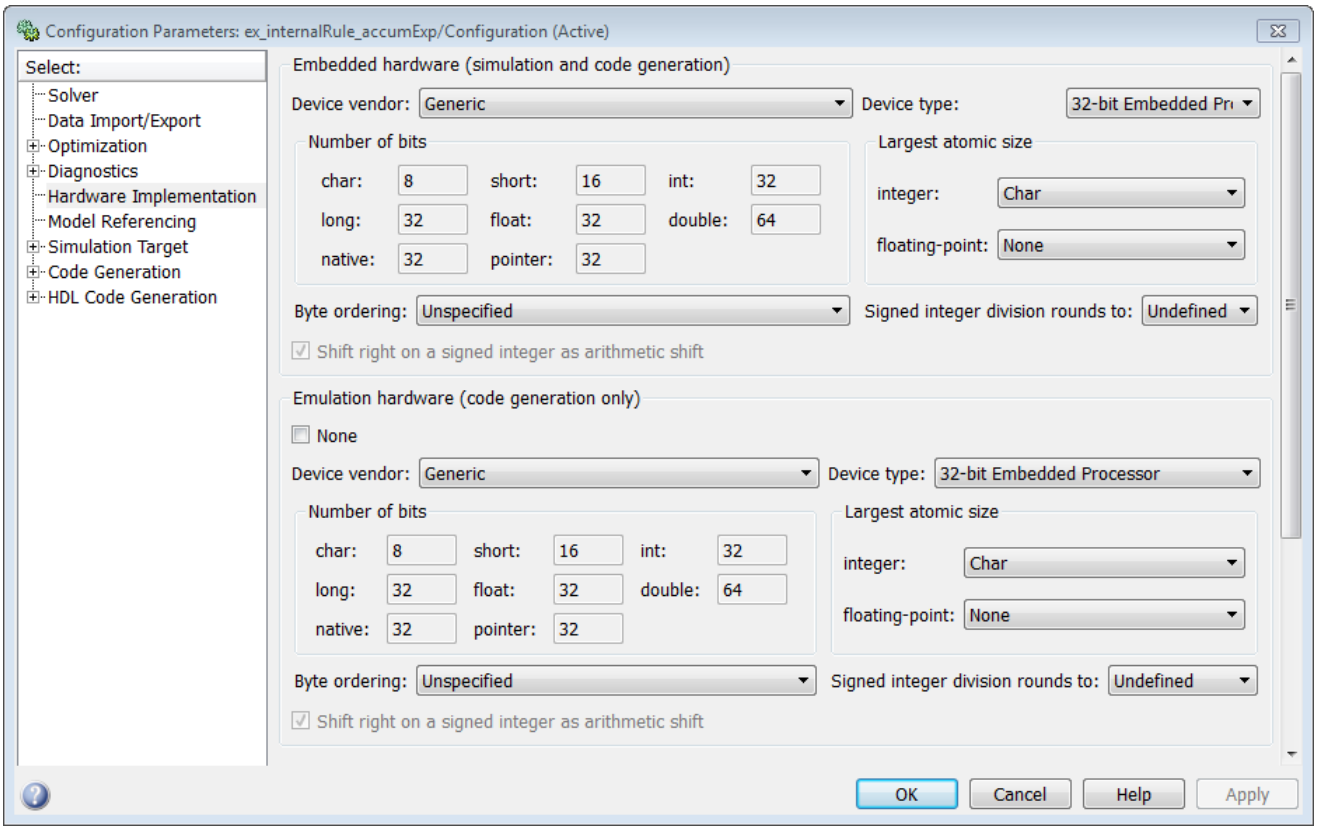
$$WL_{idealaccumulator1} = WL_{inputtoaccumulator1} + \text{floor}(\log_2(numberofaccumulations)) + 1$$
$$WL_{idealaccumulator1} = 16 + \text{floor}(\log_2(1)) + 1$$
$$WL_{idealaccumulator1} = 16 + 0 + 1 = 17$$

$$WL_{idealaccumulator2} = WL_{inputtoaccumulator2} + \text{floor}(\log_2(numberofaccumulations)) + 1$$
$$WL_{idealaccumulator2} = 127 + \text{floor}(\log_2(1)) + 1$$
$$WL_{idealaccumulator2} = 127 + 0 + 1 = 128$$

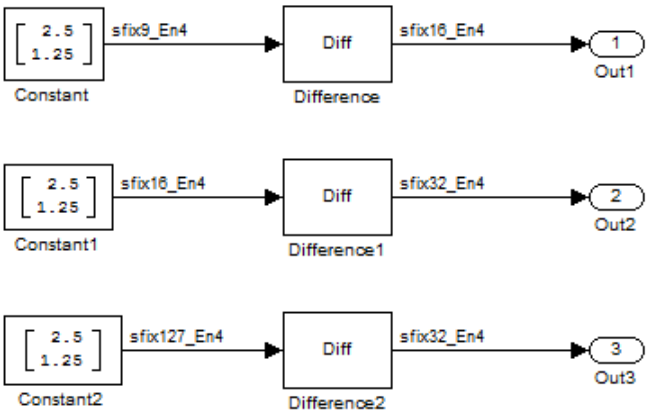
Calculate the full-precision fraction length, which is the same for each Matrix Sum block in this example:

$$FL_{idealaccumulator} = FL_{inputtoaccumulator}$$
$$FL_{idealaccumulator} = 4$$

Now change the **Device type** parameter in the **Hardware Implementation** pane of the Configuration Parameters dialog box to `32—bit Embedded Processor`, by changing the parameters as shown in the following figure.

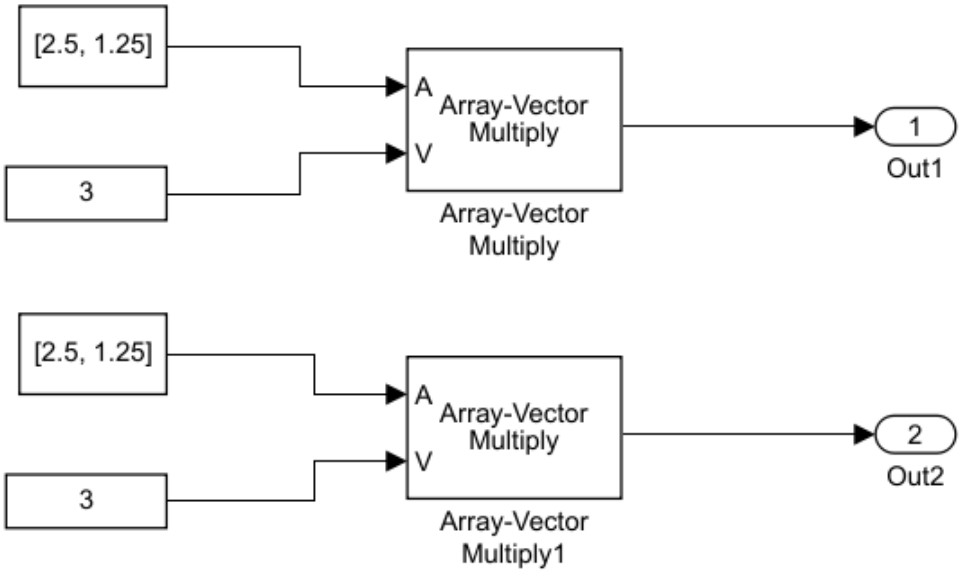


As you can see in the dialog box, this device has 8-, 16-, and 32-bit word lengths available. Therefore, the ideal word lengths of 10, 17, and 128 bits calculated by the internal rule cannot be used. Instead, the internal rule uses the next largest available word length in each case You can see this if you rerun the model, as shown in the following figure.



Product Data Types

Consider the following model ex_internalRule_prodExp.



In the Array-Vector Multiply blocks, the **Product Output** parameter is set to `Inherit: Inherit via internal rule`, and the **Output** parameter is set to `Inherit: Same as product output`. Therefore, you can see the product output data type calculated by the internal rule on the output signal in the model. The setting of the **Accumulator** parameter does not matter because this example uses real values.

For the preceding model, the **Device type** parameter in the **Hardware Implementation** pane of the Configuration Parameters dialog box is set to `ASIC/FPGA`. Therefore, the product data type used by the internal rule is the ideal, full-precision result.

Calculate the full-precision word length for each of the Array-Vector Multiply blocks in the model:

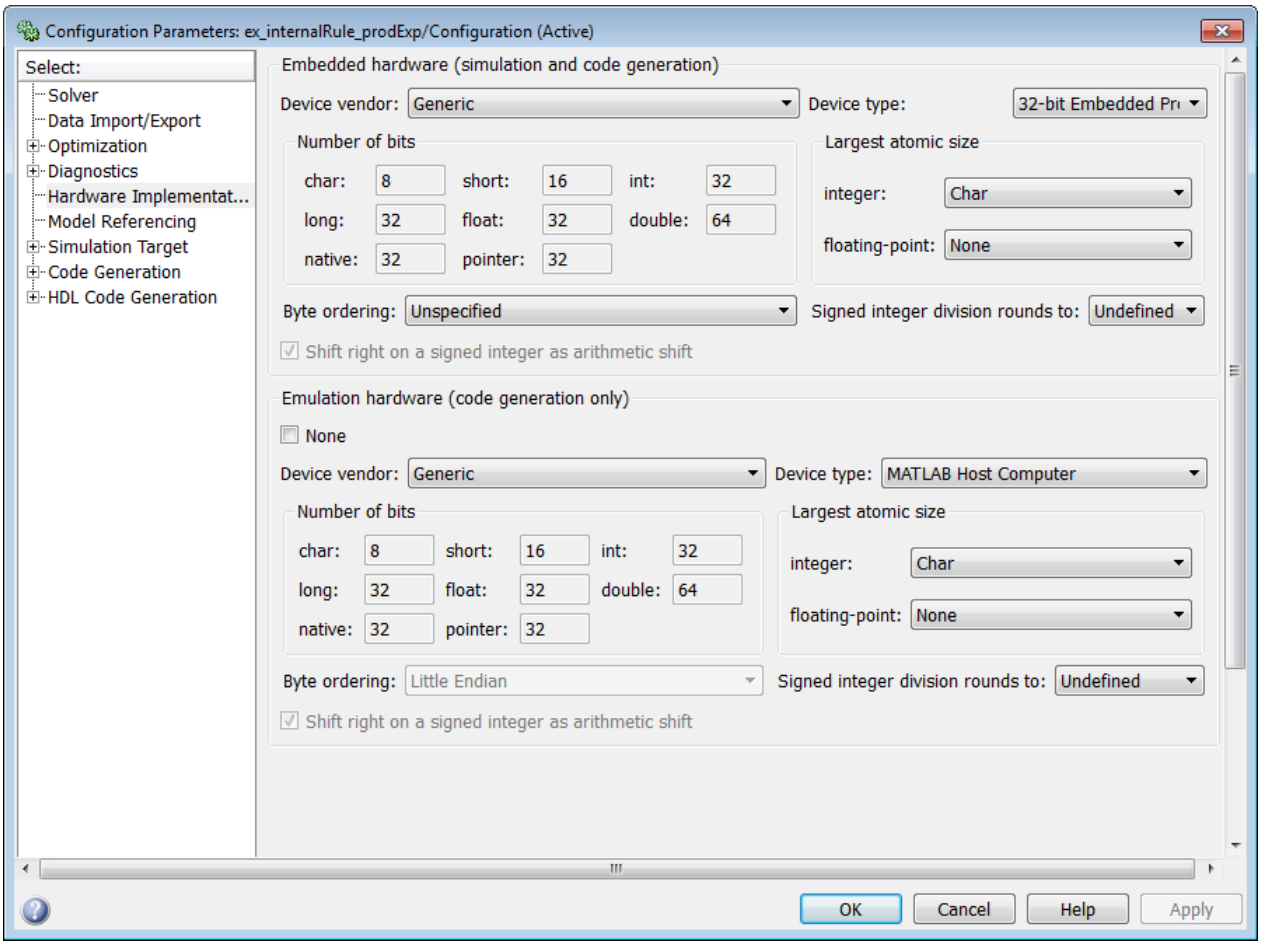
$$WL_{idealproduct} = WL_{inputa} + WL_{inputb}$$
$$WL_{idealproduct} = 7 + 5 = 12$$

$$WL_{idealproduct1} = WL_{inputa} + WL_{inputb}$$
$$WL_{idealproduct1} = 16 + 15 = 31$$

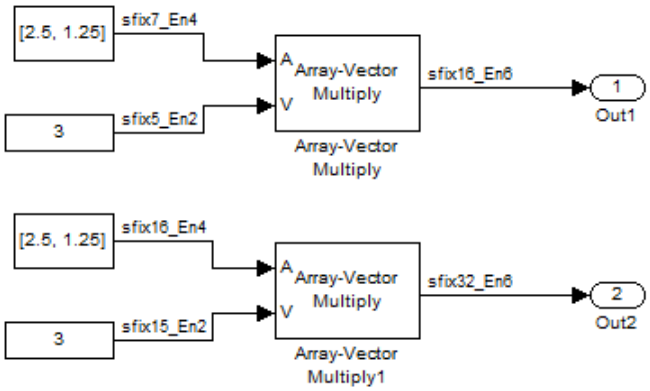
Calculate the full-precision fraction length, which is the same for each Array-Vector Multiply block in this example:

$$FL_{idealaccumulator} = FL_{inputtoaccumulator}$$
$$FL_{idealaccumulator} = 4$$

Now change the **Device type** parameter in the **Hardware Implementation** pane of the Configuration Parameters dialog box to `32-bit Embedded Processor`, as shown in the following figure.



As you can see in the dialog box, this device has 8-, 16-, and 32-bit word lengths available. Therefore, the ideal word lengths of 12 and 31 bits calculated by the internal rule cannot be used. Instead, the internal rule uses the next largest available word length in each case. You can see this if you rerun the model, as shown in the following figure.



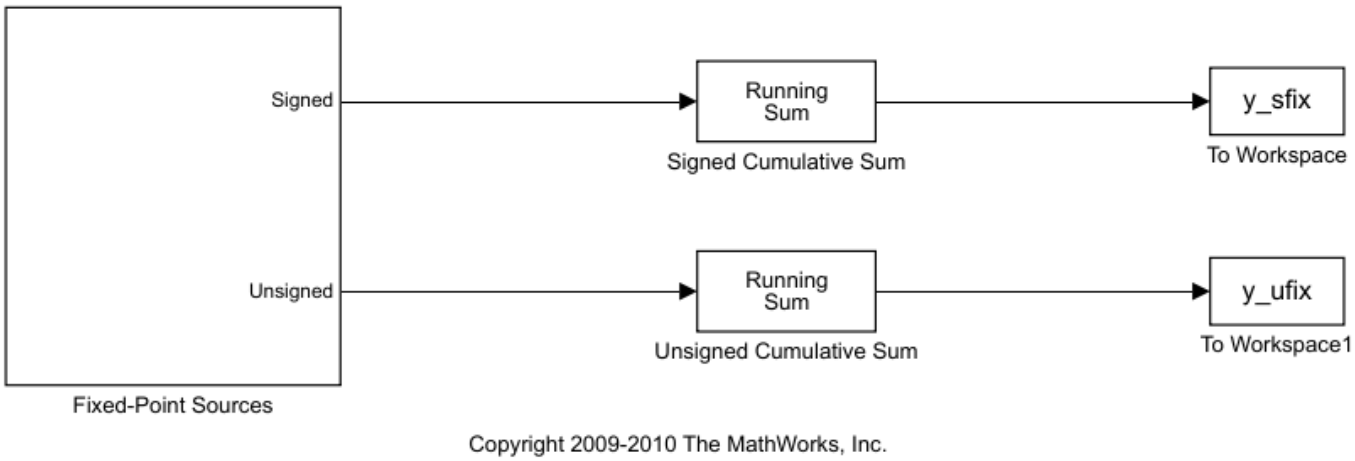
Specify Data Types for Fixed-Point Blocks

The following sections show you how to use the Fixed-Point Tool to select appropriate data types for fixed-point blocks in the `ex_fixedpoint_tut` model:

- “Prepare the Model” on page 17-42
- “Use Data Type Override to Find a Floating-Point Benchmark” on page 17-47
- “Use the Fixed-Point Tool to Propose Fraction Lengths” on page 17-47
- “Examine the Results and Accept the Proposed Scaling” on page 17-47

Prepare the Model

1 Open the model by typing ex_fixedpoint_tut at the MATLAB command line.



Copyright 2009-2010 The MathWorks, Inc.

This model uses the Cumulative Sum block to sum the input coming from the Fixed-Point Sources subsystem. The Fixed-Point Sources subsystem outputs two signals with different data types:

- The Signed source has a word length of 16 bits and a fraction length of 15 bits.
- The Unsigned source has a word length of 16 bits and a fraction length of 16 bits.

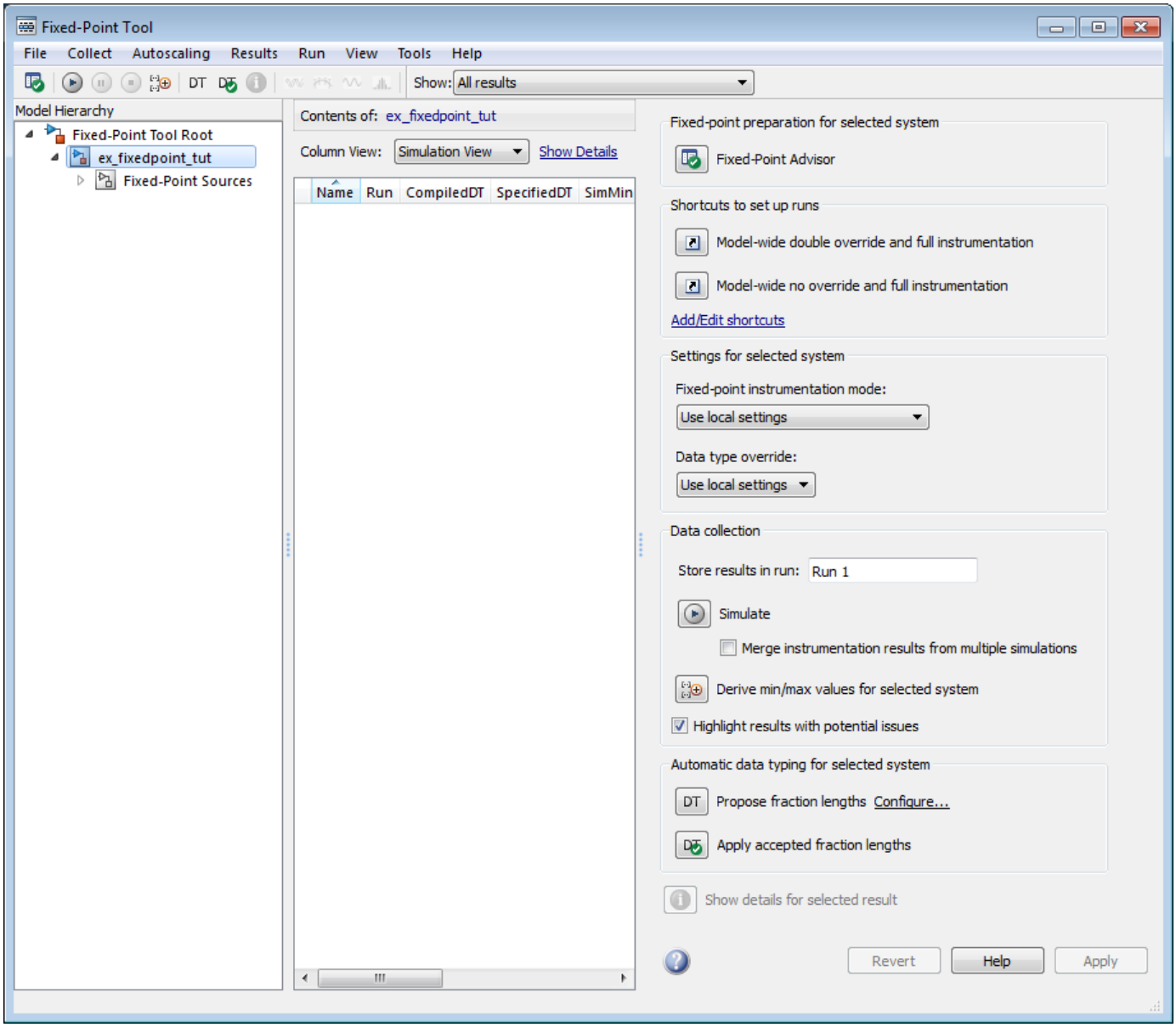
2 Run the model to check for overflow. MATLAB displays the following warnings at the command line:

```
Warning: Overflow occurred. This originated from
'ex_fixedpoint_tut/Signed Cumulative Sum'.
Warning: Overflow occurred. This originated from
'ex_fixedpoint_tut/Unsigned Cumulative Sum'.
```

According to these warnings, overflow occurs in both Cumulative Sum blocks.

3 To investigate the overflows in this model, use the Fixed-Point Tool. You can open the Fixed-Point Tool by selecting **Tools > Fixed-Point > Fixed-Point Tool** from the model menu. Turn on logging for all blocks in your model by setting the **Fixed-point instrumentation mode** parameter to `Minimums, maximums and overflows`.

4 Now that you have turned on logging, rerun the model by clicking the Simulation button.



5 The results of the simulation appear in a table in the central **Contents** pane of the Fixed-Point Tool. Review the following columns:

- **Name** — Provides the name of each signal in the following format: `Subsystem Name/Block Name: Signal Name`.
- **SimDT** — The simulation data type of each logged signal.
- **SpecifiedDT** — The data type specified on the block dialog for each signal.
- **SimMin** — The smallest representable value achieved during simulation for each logged signal.
- **SimMax** — The largest representable value achieved during simulation for each logged signal.
- **OverflowWraps** — The number of overflows that wrap during simulation.

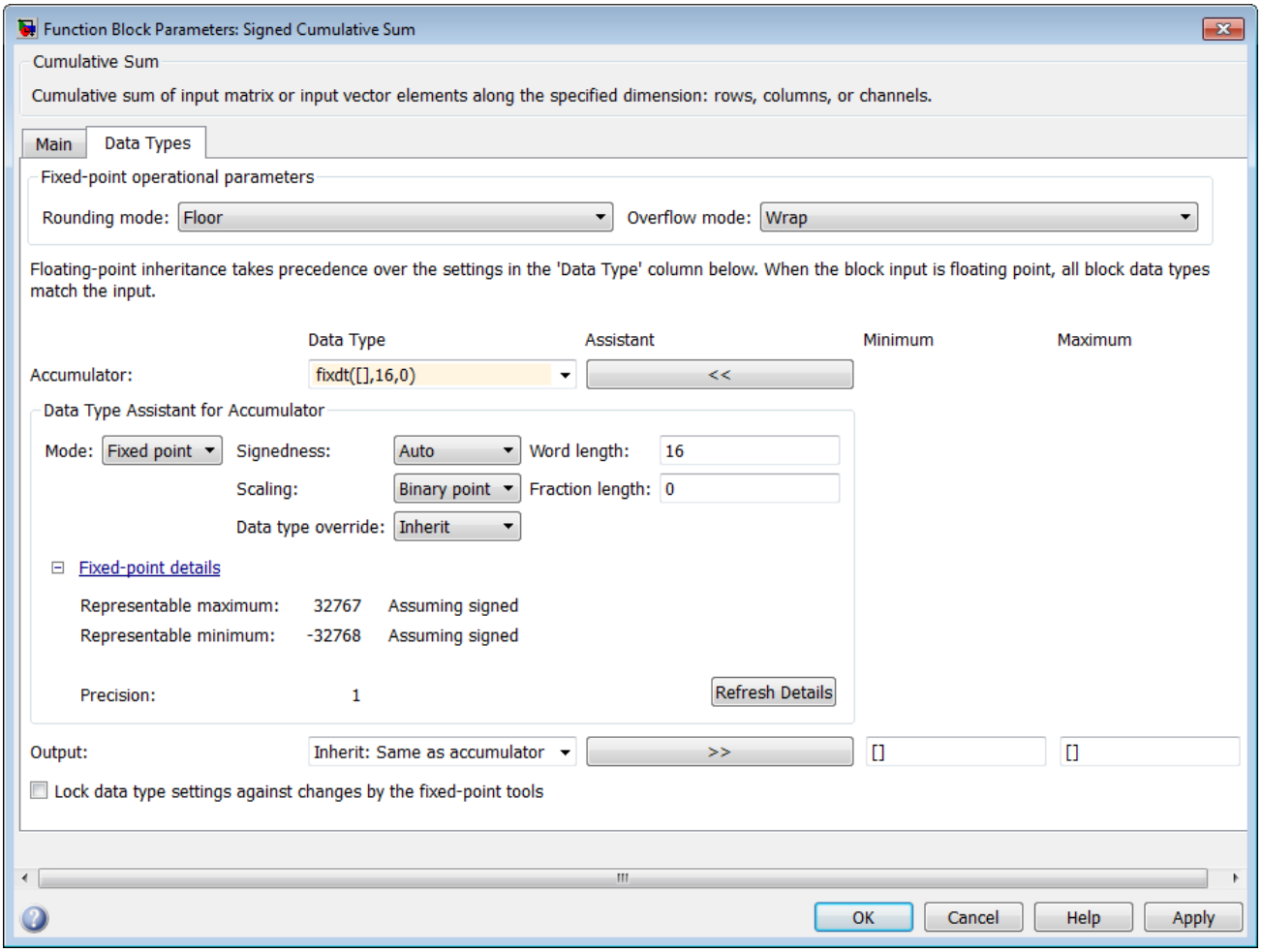
For more information on each of the columns in this table, see the section of the Simulink `fxptdlg` function reference page.

You can also see that the **SimMin** and **SimMax** values for the Accumulator data types range from `0` to `.9997`. The logged results indicate that 8,192 overflows wrapped during simulation in the Accumulator data type of the Signed Cumulative Sum block. Similarly, the Accumulator data type of the Unsigned Cumulative Sum block had 16,383 overflows wrap during simulation.

To get more information about each of these data types, highlight them in the **Contents** pane, and click the **Show details for selected result** button ( )

6 Assume a target hardware that supports 32-bit integers, and set the Accumulator word length in both Cumulative Sum blocks to `32`. To do so, perform the following steps:

1. Right-click the `Signed Cumulative Sum: Accumulator` row in the Fixed-Point Tool pane, and select **Highlight Block In Model**.
2. Double-click the block in the model, and select the **Data Types** pane of the dialog box.
3. Open the **Data Type Assistant** for Accumulator by clicking the Assistant button ( ) in the Accumulator data type row.
4. Set the **Mode** to `Fixed Point`. To see the representable range of the current specified data type, click the **Fixed-point details** link. The tool displays the representable maximum and representable minimum values for the current data type.



5. Change the **Word length** to 32, and click the **Refresh details** button in the **Fixed-point details** section to see the updated representable range. When you change the value of the **Word length** parameter, the **Data Type** edit box automatically updates.
6. Click **OK** on the block dialog box to save your changes and close the window.
7. Set the word length of the Accumulator data type of the Unsigned Cumulative Sum block to 32 bits. You can do so in one of two ways:

   - Type the data type `fixdt([],32,0)` directly into **Data Type** edit box for the Accumulator data type parameter.
   - Perform the same steps you used to set the word length of the Accumulator data type of the Signed Cumulative Sum block to 32 bits.

7. To verify your changes in word length and check for overflow, rerun your model. To do so, click the **Simulate** button in the Fixed-Point Tool.

   The **Contents** pane of the Fixed-Point Tool updates, and you can see that no overflows occurred in the most recent simulation. However, you can also see that the **SimMin** and **SimMax** values range from `0` to `0`. This underflow happens because the fraction length of the Accumulator data type is too small. The **SpecifiedDT** cannot represent the precision of the data values. The following sections discuss how to find a floating-point benchmark and use the Fixed-Point Tool to propose fraction lengths.

Use Data Type Override to Find a Floating-Point Benchmark

The **Data type override** feature of the Fixed-Point tool allows you to override the data types specified in your model with floating-point types. Running your model in `Double` override mode gives you a reference range to help you select appropriate fraction lengths for your fixed-point data types. To do so, perform the following steps:

1. Open the Fixed-Point Tool and set **Data type override** to `Double`.
2. Run your model by clicking the **Run simulation and store active results** button.
3. Examine the results in the **Contents** pane of the Fixed-Point Tool. Because you ran the model in `Double` override mode, you get an accurate, idealized representation of the simulation minimums and maximums. These values appear in the **SimMin** and **SimMax** parameters.
4. Now that you have an accurate reference representation of the simulation minimum and maximum values, you can more easily choose appropriate fraction lengths. Before making these choices, save your active results to reference so you can use them as your floating-point benchmark. To do so, select **Results > Move Active Results To Reference** from the Fixed-Point Tool menu. The status displayed in the **Run** column changes from `Active` to `Reference` for all signals in your model.

Use the Fixed-Point Tool to Propose Fraction Lengths

Now that you have your `Double` override results saved as a floating-point reference, you are ready to propose fraction lengths.

1. To propose fraction lengths for your data types, you must have a set of `Active` results available in the Fixed-Point Tool. To produce an active set of results, simply rerun your model. The tool now displays both the `Active` results and the `Reference` results for each signal.
2. Select the **Use simulation min/max if design min/max is not available** check box. You did not specify any design minimums or maximums for the data types in this model. Thus, the tool uses the logged information to compute and propose fraction lengths. For information on specifying design minimums and maximums, see “Specify Signal Ranges” (Simulink).
3. Click the **Propose fraction lengths** button ( ). The tool populates the proposed data types in the **ProposedDT** column of the **Contents** pane. The corresponding proposed minimums and maximums are displayed in the **ProposedMin** and **ProposedMax** columns.

Examine the Results and Accept the Proposed Scaling

Before accepting the fraction lengths proposed by the Fixed-Point Tool, it is important to look at the details of that data type. Doing so allows you to see how much of your data the suggested data type can represent. To examine the suggested data types and accept the proposed scaling, perform the following steps:

1. In the **Contents** pane of the Fixed-Point Tool, you can see the proposed fraction lengths for the data types in your model.

   - The proposed fraction length for the Accumulator data type of both the Signed and Unsigned Cumulative Sum blocks is 17 bits.
   - To get more details about the proposed scaling for a particular data type, highlight the data type in the **Contents** pane of the Fixed-Point Tool.
   - Open the Autoscale Information window for the highlighted data type by clicking the **Show autoscale information for the selected result** button ( ).

2. When the Autoscale Information window opens, check the **Value** and **Percent Proposed Representable** columns for the **Simulation Minimum** and **Simulation Maximum** parameters. You can see that the proposed data type can represent 100% of the range of simulation data.
3. To accept the proposed data types, select the check box in the **Accept** column for each data type whose proposed scaling you want to keep. Then, click the **Apply accepted fraction lengths** button ( ). The tool updates the specified data types on the block dialog boxes and the **SpecifiedDT** column in the **Contents** pane.
4. To verify the newly accepted scaling, set the **Data type override** parameter back to **Use local settings**, and run the model. Looking at **Contents** pane of the Fixed-Point Tool, you can see the following details:

   - The **SimMin** and **SimMax** values of the `Active` run match the **SimMin** and **SimMax** values from the floating-point `Reference` run.
   - There are no longer any overflows.
   - The **SimDT** does not match the **SpecifiedDT** for the Accumulator data type of either Cumulative Sum block. This difference occurs because the Cumulative Sum block always inherits its **Signedness** from the input signal and only allows you to specify a **Signedness** of `Auto`. Therefore, the **SpecifiedDT** for both Accumulator data types is `fixdt([],32,17)`. However, because the Signed Cumulative Sum block has a signed input signal, the **SimDT** for the Accumulator parameter of that block is also signed (`fixdt(1,32,17)`). Similarly, the **SimDT** for the Accumulator parameter of the Unsigned Cumulative Sum block inherits its **Signedness** from its input signal and thus is unsigned (`fixdt(0,32,17)`).

Quantizers

In this section...
“Scalar Quantizers” on page 17-49
“Vector Quantizers” on page 17-53

Scalar Quantizers

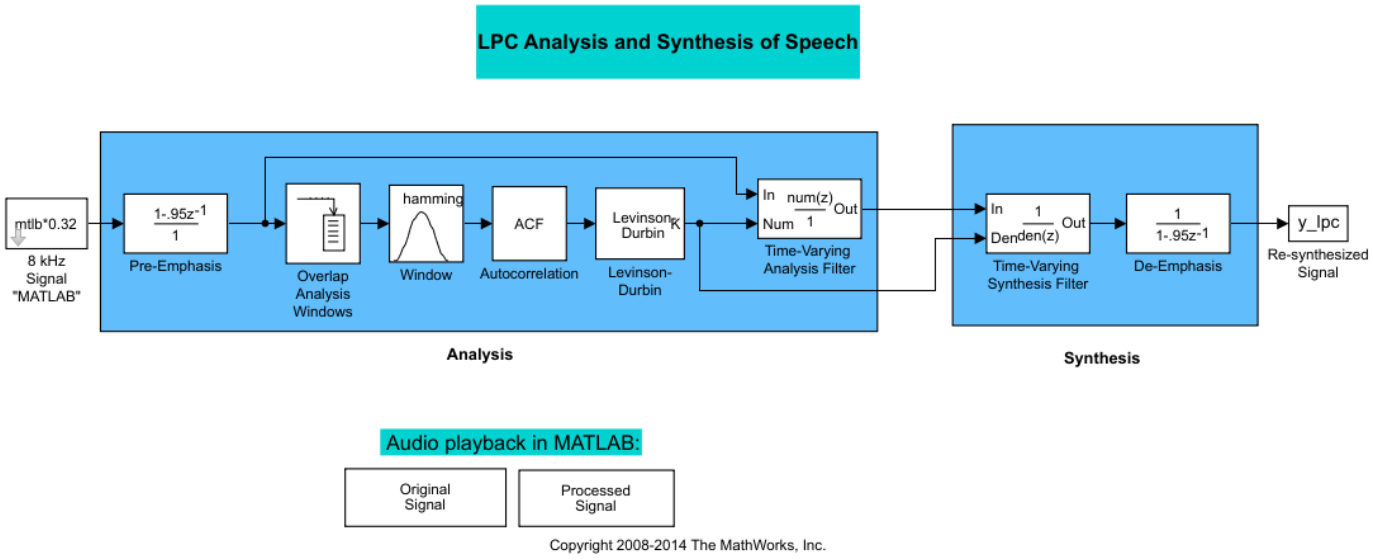
- “Analysis and Synthesis of Speech” on page 17-49
- “Identify Your Residual Signal and Reflection Coefficients” on page 17-50
- “Create a Scalar Quantizer” on page 17-51

Analysis and Synthesis of Speech

You can use blocks from the DSP System Toolbox Quantizers library to design scalar quantizer encoders and decoders. A speech signal is usually represented in digital format, which is a sequence of binary bits. For storage and transmission applications, it is desirable to compress a signal by representing it with as few bits as possible, while maintaining its perceptual quality. Quantization is the process of representing a signal with a reduced level of precision. If you decrease the number of bits allocated for the quantization of your speech signal, the signal is distorted and the speech quality degrades.

In narrowband digital speech compression, speech signals are sampled at a rate of 8000 samples per second. Each sample is typically represented by 8 bits. This 8-bit representation corresponds to a bit rate of 64 kbits per second. Further compression is possible at the cost of quality. Most of the current low bit rate speech coders are based on the principle of linear predictive speech coding. This topic shows you how to use the Scalar Quantizer Encoder and Scalar Quantizer Decoder blocks to implement a simple speech coder.

1 Type `ex_sq_example1` at the MATLAB command line to open the example model.



This model pre-emphasizes the input speech signal by applying an FIR filter. Then, it calculates the reflection coefficients of each frame using the Levinson-Durbin algorithm. The model uses these reflection coefficients to create the linear prediction analysis filter (lattice-structure). Next, the model calculates the residual signal by filtering each frame of the pre-emphasized speech samples using the reflection coefficients. The residual signal, which is the output of the analysis stage, usually has a lower energy than the input signal. The blocks in the synthesis stage of the model filter the residual signal using the reflection coefficients and apply an all-pole de-emphasis filter. The de-emphasis filter is the inverse of the pre-emphasis filter. The result is the full recovery of the original signal.

2. Run this model.
3. Double-click the Original Signal and Processed Signal blocks and listen to both the original and the processed signal.

   There is no significant difference between the two because no quantization was performed.

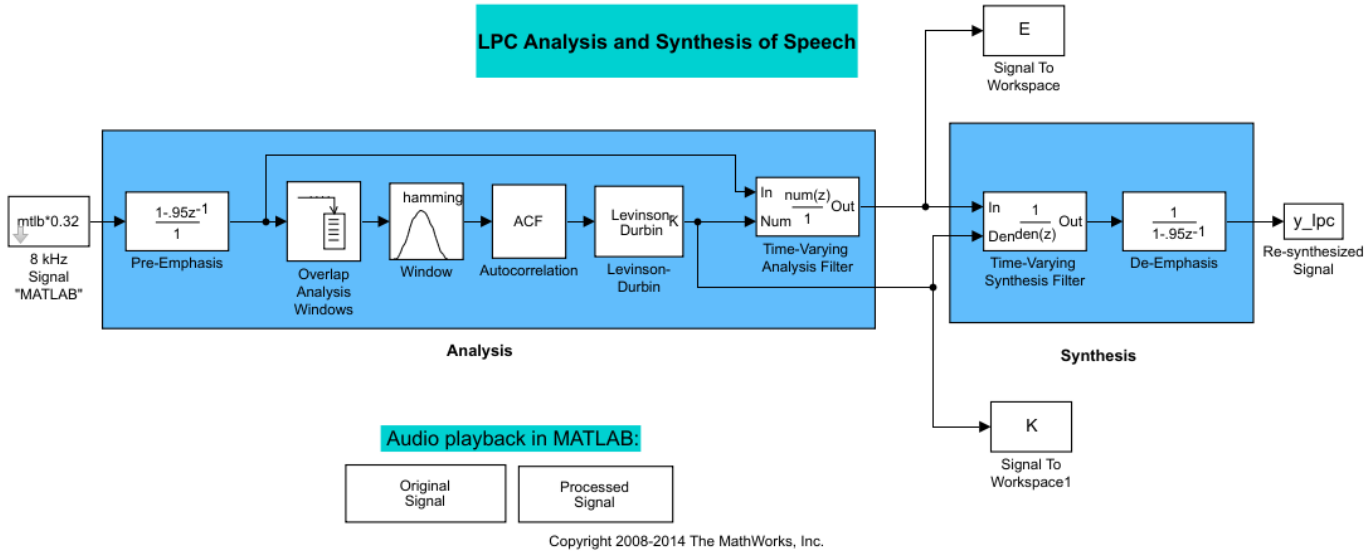
To better approximate a real-world speech analysis and synthesis system, quantize the residual signal and reflection coefficients before they are transmitted. The following topics show you how to design scalar quantizers to accomplish this task.

Identify Your Residual Signal and Reflection Coefficients

In the previous topic, “Analysis and Synthesis of Speech” on page 17-49, you learned the theory behind the LPC Analysis and Synthesis of Speech example model. In this topic, you define the residual signal and the reflection coefficients in your MATLAB workspace as the variables *E* and *K*, respectively. Later, you use these values to create your scalar quantizers:

1. Open the example model by typing `ex_sq_example2` at the MATLAB command line.
2. Save the model file as `ex_sq_example2` in your working folder.
3. From the Simulink Sinks library, click-and-drag two To Workspace blocks into your model.
4. Connect the output of the Levinson-Durbin block to one of the To Workspace blocks.
5. Double-click this To Workspace block and set the **Variable name** parameter to `K`. Click **OK**.
6. Connect the output of the Time-Varying Analysis Filter block to the other To Workspace block.
7. Double-click this To Workspace block and set the **Variable name** parameter to `E`. Click **OK**.

   Your model should now look similar to this figure.



8 Run your model.

The residual signal, *E*, and your reflection coefficients, *K*, are defined in the MATLAB workspace. In the next topic, you use these variables to design your scalar quantizers.

Create a Scalar Quantizer

In this topic, you create scalar quantizer encoders and decoders to quantize the residual signal, *E*, and the reflection coefficients, *K*:

1 If the model you created in “Identify Your Residual Signal and Reflection Coefficients” on page 17-50 is not open on your desktop, you can open an equivalent model by typing ex_sq_example2 at the MATLAB command prompt.

2 Run this model to define the variables *E* and *K* in the MATLAB workspace.

3 From the Quantizers library, click-and-drag a Scalar Quantizer Design block into your model. Double-click this block to open the SQ Design Tool GUI.

4 For the **Training Set** parameter, enter `K`.

The variable *K* represents the reflection coefficients you want to quantize. By definition, they range from -1 to 1.

Note Theoretically, the signal that is used as the **Training Set** parameter should contain a representative set of values for the parameter to be quantized. However, this example provides an approximation to this global training process.

---

5 For the **Number of levels** parameter, enter `128`.

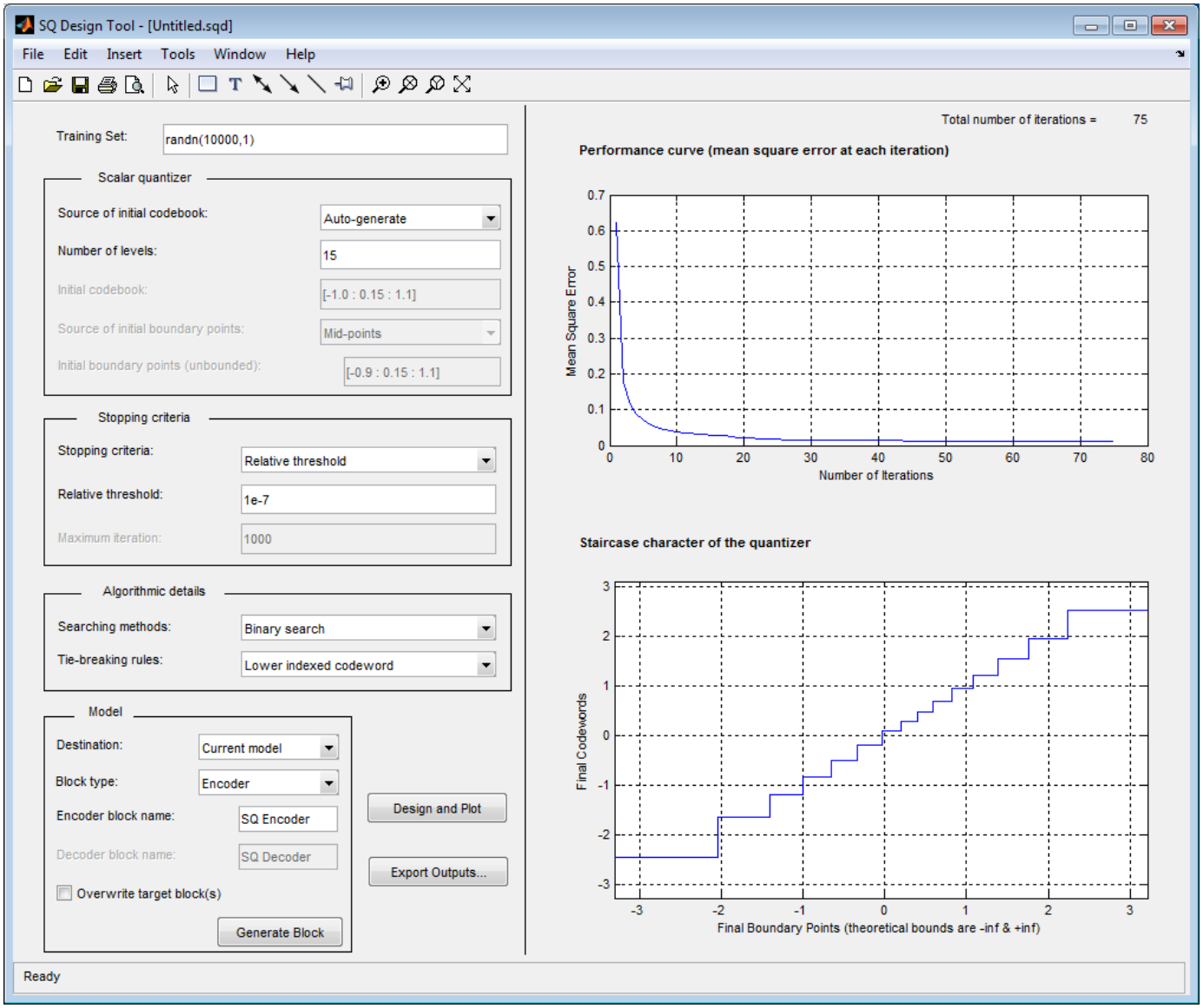
Assume that your compression system has 7 bits to represent each reflection coefficient. This means it is capable of representing 2^7 or 128 values. The **Number of levels** parameter is equal to the total number of codewords in the codebook.

6 Set the **Block type** parameter to `Both`.

7 For the **Encoder block name** parameter, enter `SQ Encoder - Reflection Coefficients`.

8. For the **Decoder block name** parameter, enter `SQ Decoder - Reflection Coefficients`.
9. Make sure that your desired destination model, `ex_sq_example2`, is the current model. You can type `gcs` in the MATLAB Command Window to display the name of your current model.
10. In the SQ Design Tool GUI, click the **Design and Plot** button to apply the changes you made to the parameters.

    The GUI should look similar to the following figure.



11. Click the **Generate Model** button.

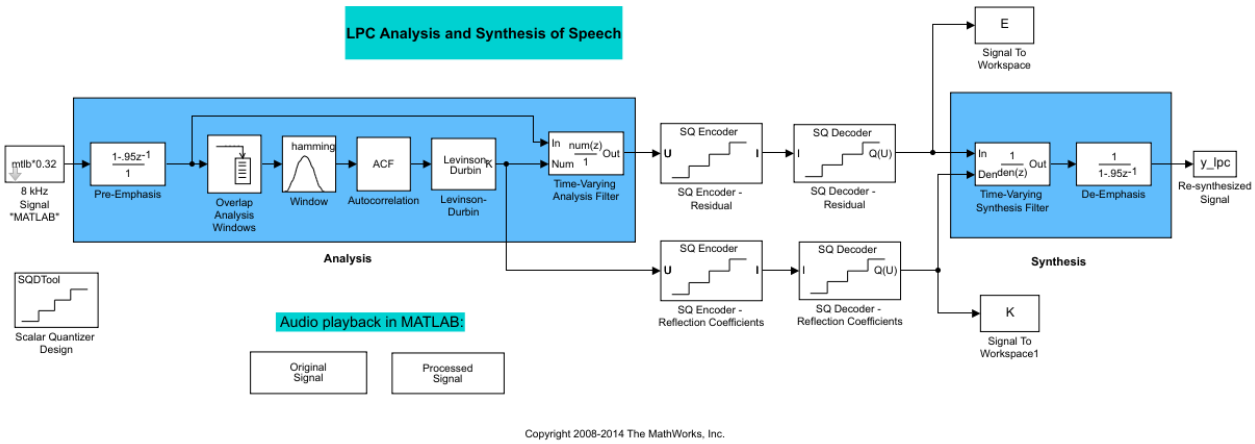
    Two new blocks, SQ Encoder - Reflection Coefficients and SQ Decoder - Reflection Coefficients, appear in your model file.
12. Click the SQ Design Tool GUI and, for the **Training Set** parameter, enter `E`.
13. Repeat steps 5 to 11 for the variable *E*, which represents the residual signal you want to quantize. In steps 6 and 7, name your blocks SQ Encoder - Residual and SQ Decoder - Residual.

Once you have completed these steps, two new blocks, SQ Encoder - Residual and SQ Decoder - Residual, appear in your model file.

14 Close the SQ Design Tool GUI. You do not need to save the SQ Design Tool session.

You have now created a scalar quantizer encoder and a scalar quantizer decoder for each signal you want to quantize. You are ready to quantize the residual signal, E, and the reflection coefficients, K.

15 Save the model as `ex_sq_example3`. Your model should look similar to the following figure.



16 Run your model.

17 Double-click the Original Signal and Processed Signal blocks, and listen to both signals.

Again, there is no perceptible difference between the two. You can therefore conclude that quantizing your residual and reflection coefficients did not affect the ability of your system to accurately reproduce the input signal.

You have now quantized the residual and reflection coefficients. The bit rate of a quantization system is calculated as (bits per frame)*(frame rate).

In this example, the bit rate is [(80 residual samples/frame)*(7 bits/sample) + (12 reflection coefficient samples/frame)*(7 bits/sample)]*(100 frames/second), or 64.4 kbits per second. This is higher than most modern speech coders, which typically have a bit rate of 8 to 24 kbits per second. If you decrease the number of bits allocated for the quantization of the reflection coefficients or the residual signal, the overall bit rate would decrease. However, the speech quality would also degrade.

For information about decreasing the bit rate without affecting speech quality, see “Vector Quantizers” on page 17-53.

Vector Quantizers

- “Build Your Vector Quantizer Model” on page 17-53
- “Configure and Run Your Model” on page 17-54

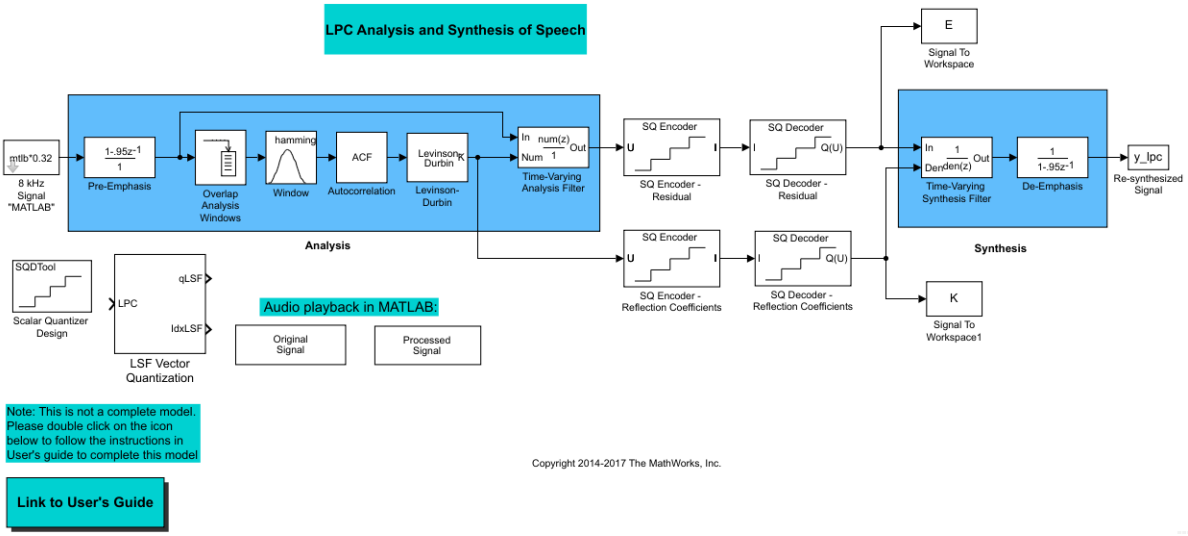
Build Your Vector Quantizer Model

In the previous section, you created scalar quantizer encoders and decoders and used them to quantize your residual signal and reflection coefficients. The bit rate of your scalar quantization

system was 64.4 kbits per second. This bit rate is higher than most modern speech coders. To accommodate a greater number of users in each channel, you need to lower this bit rate while maintaining the quality of your speech signal. You can use vector quantizers, which exploit the correlations between each sample of a signal, to accomplish this task.

In this topic, you modify your scalar quantization model so that you are using a split vector quantizer to quantize your reflection coefficients:

1. Open a model similar to the one you created in “Create a Scalar Quantizer” on page 17-51 by typing ex_vq_example1 at the MATLAB command prompt. The example model `ex_vq_example1` adds a new LSF Vector Quantization subsystem to the `ex_sq_example3` model. This subsystem is preconfigured to work as a vector quantizer. You can use this subsystem to encode and decode your reflection coefficients using the split vector quantization method.
2. Delete the SQ Encoder – Reflection Coefficients and SQ Decoder – Reflection Coefficients blocks.
3. From the Simulink Sinks library, click-and-drag a Terminator block into your model.
4. From the DSP System Toolbox Estimation > Linear Prediction library, click-and-drag a LSF/LSP to LPC Conversion block and two LPC to/from RC blocks into your model.
5. Connect the blocks as shown in the following figure. You do not need to connect Terminator blocks to the P ports of the LPC to/from RC blocks. These ports disappear once you set block parameters.



You have modified your model to include a subsystem capable of vector quantization. In the next topic, you reset your model parameters to quantize your reflection coefficients using the split vector quantization method.

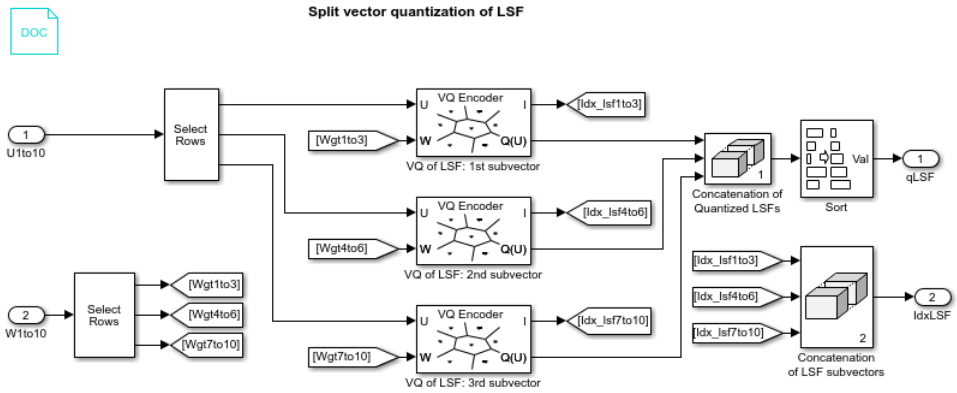
Configure and Run Your Model

In the previous topic, you configured your scalar quantization model for vector quantization by adding the LSF Vector Quantization subsystem. In this topic, you set your block parameters and quantize your reflection coefficients using the split vector quantization method.

1. If the model you created in “Build Your Vector Quantizer Model” on page 17-53 is not open on your desktop, you can open an equivalent model by typing ex_vq_example2 at the MATLAB command prompt.

2 Double-click the LSF Vector Quantization subsystem, and then double-click the LSF Split VQ subsystem.

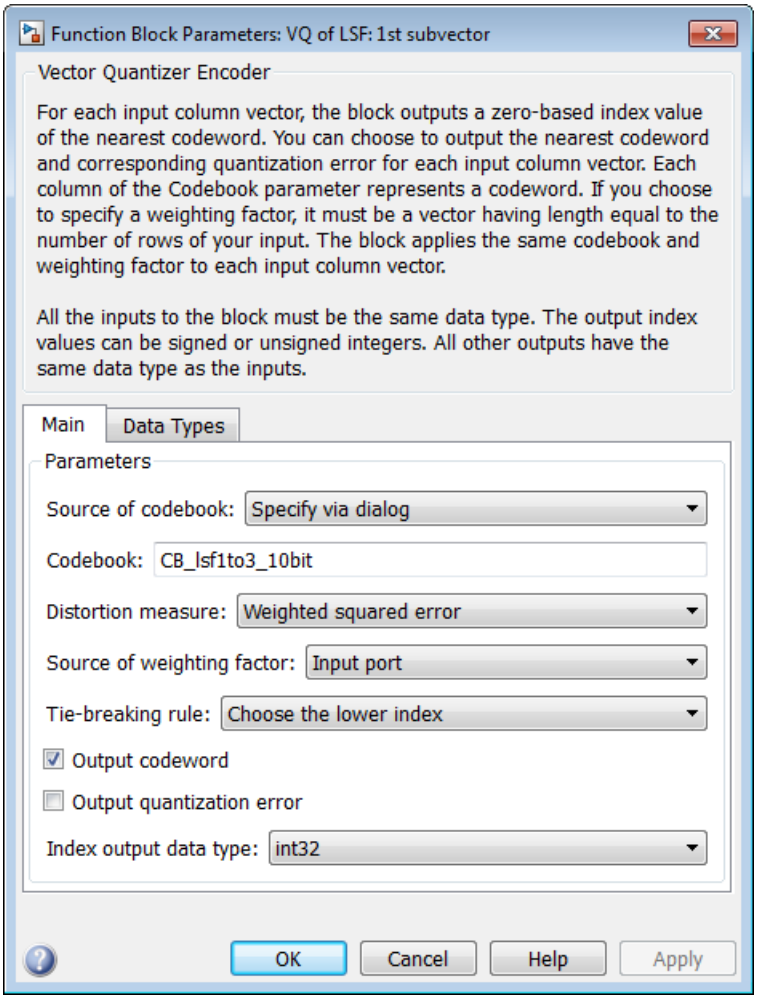
The subsystem opens, and you see the three Vector Quantizer Encoder blocks used to implement the split vector quantization method.



This subsystem divides each vector of 10 line spectral frequencies (LSFs), which represent your reflection coefficients, into three LSF subvectors. Each of these subvectors is sent to a separate vector quantizer. This method is called split vector quantization.

3 Double-click the VQ of LSF: 1st subvector block.

The Block Parameters: VQ of LSF: 1st subvector dialog box opens.



The variable `CB_lsf1to3_10bit` is the codebook for the subvector that contains the first three elements of the LSF vector. It is a 3-by-1024 matrix, where 3 is the number of elements in each codeword and 1024 is the number of codewords in the codebook. Because $2^{10} = 1024$, it takes 10 bits to quantize this first subvector. Similarly, a 10-bit vector quantizer is applied to the second and third subvectors, which contain elements 4 to 6 and 7 to 10 of the LSF vector, respectively. Therefore, it takes 30 bits to quantize all three subvectors.

Note If you used the vector quantization method to quantize your reflection coefficients, you would need 2_{30} or 1.0737e9 codebook values to achieve the same degree of accuracy as the split vector quantization method.

---

4 In your model file, double-click the Autocorrelation block and set the **Maximum non-negative lag (less than input length)** parameter to `10`. Click **OK**.

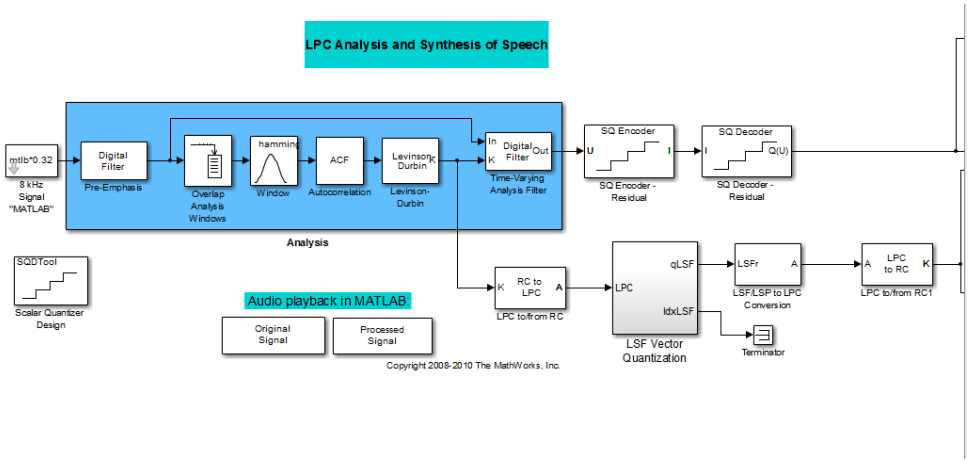
This parameter controls the number of linear polynomial coefficients (LPCs) that are input to the split vector quantization method.

5 Double-click the LPC to/from RC block that is connected to the input of the LSF Vector Quantization subsystem. Clear the **Output normalized prediction error power** check box. Click **OK**.

6 Double-click the LSF/LSP to LPC Conversion block and set the **Input** parameter to `LSF in range (0 to pi)`. Click **OK**.

7 Double-click the LPC to/from RC block that is connected to the output of the LSF/LSP to LPC Conversion block. Set the **Type of conversion** parameter to `LPC to RC`, and clear the **Output normalized prediction error power** check box. Click **OK**.

8 Run your model.



9 Double-click the Original Signal and Processed Signal blocks to listen to both the original and the processed signal.

There is no perceptible difference between the two. Quantizing your reflection coefficients using a split vector quantization method produced good quality speech without much distortion.

You have now used the split vector quantization method to quantize your reflection coefficients. The vector quantizers in the LSF Vector Quantization subsystem use 30 bits to quantize a frame containing 80 reflection coefficients. The bit rate of a quantization system is calculated as (bits per frame)*(frame rate).

In this example, the bit rate is [(80 residual samples/frame)*(7 bits/sample) + (30 bits/frame)]*(100 frames/second), or 59 kbits per second. This is less than 64.4 kbits per second, the bit rate of the scalar quantization system. However, the quality of the speech signal did not degrade. If you want to further reduce the bit rate of your system, you can use the vector quantization method to quantize the residual signal.

Create an FIR Filter Using Integer Coefficients

In this section...

“Define the Filter Coefficients” on page 17-58
“Build the FIR Filter” on page 17-58
“Set the Filter Parameters to Work with Integers” on page 17-59
“Create a Test Signal for the Filter” on page 17-60
“Filter the Test Signal” on page 17-60
“Truncate the Output WordLength” on page 17-62
“Scale the Output” on page 17-64
“Configure Filter Parameters to Work with Integers Using the set2int Method” on page 17-67

This section provides an example of how you can create a filter with integer coefficients. In this example, a raised-cosine filter with floating-point coefficients is created, and the filter coefficients are then converted to integers.

Define the Filter Coefficients

To illustrate the concepts of using integers with fixed-point filters, this example will use a raised-cosine filter:

```
b = rcosdesign(.25, 12.5, 8, 'sqrt');
```

The coefficients of b are normalized so that the passband gain is equal to 1, and are all smaller than 1. In order to make them integers, they will need to be scaled. If you wanted to scale them to use 18 bits for each coefficient, the range of possible values for the coefficients becomes:

$$[-2^{-17}, 2^{17}-1] = \ = [-131072, 131071]$$

Because the largest coefficient of b is positive, it will need to be scaled as close as possible to 131071 (without overflowing) in order to minimize quantization error. You can determine the exponent of the scale factor by executing:

```
B = 18; % Number of bits
L = floor(log2((2^(B-1)-1)/max(b)));  % Round towards zero to avoid overflow
bsc = b*2^L;
```

Alternatively, you can use the fixed-point numbers autoscaling tool as follows:

```
bq = fi(b, true, B);  % signed = true, B = 18 bits
L = bq.FractionLength;
```

It is a coincidence that B and L are both 18 in this case, because of the value of the largest coefficient of b. If, for example, the maximum value of b were 0.124, L would be 20 while B (the number of bits) would remain 18.

Build the FIR Filter

First create the filter using the direct form, tapped delay line structure:

```
h = dfilt.dffir(bsc);
```

In order to set the required parameters, the arithmetic must be set to fixed-point:

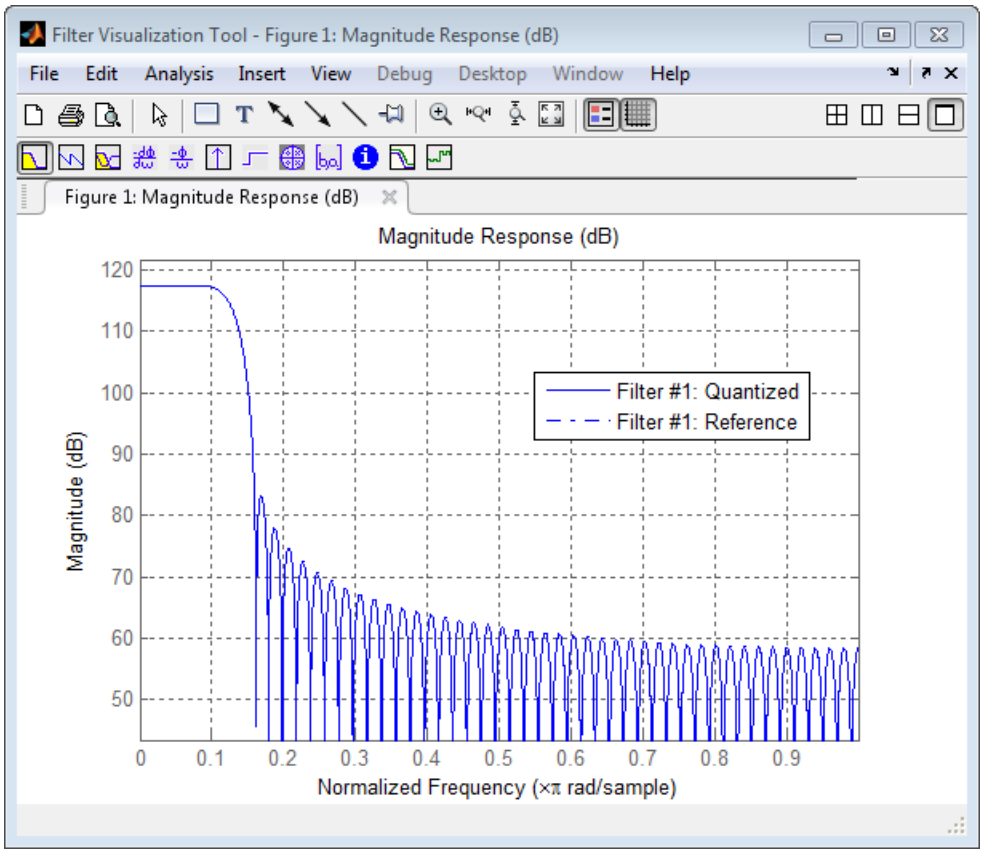
```
h.Arithmetic = 'fixed';
h.CoeffWordLength = 18;
```

You can check that the coefficients of `h` are all integers:

```
all(h.Numerator == round(h.Numerator))

ans =

    1
```

Now you can examine the magnitude response of the filter using `fvtool`:

```
fvtool(h, 'Color', 'white')
```



This shows a large gain of 117 dB in the passband, which is due to the large values of the coefficients — this will cause the output of the filter to be much larger than the input. A method of addressing this will be discussed in the following sections.

Set the Filter Parameters to Work with Integers

You will need to set the input parameters of your filter to appropriate values for working with integers. For example, if the input to the filter is from a A/D converter with 12 bit resolution, you should set the input as follows:

```
h.InputWordLength = 12;
h.InputFracLength = 0;
```

The `info` method returns a summary of the filter settings.

```
info(h)
```

```
Discrete-Time FIR Filter (real)
-------------------------------
Filter Structure  : Direct-Form FIR
Filter Length     : 101
Stable            : Yes
Linear Phase      : Yes (Type 1)
Arithmetic        : fixed
Numerator         : s18,0 -> [-131072 131072)
Input             : s12,0 -> [-2048 2048)
Filter Internals  : Full Precision
  Output          : s31,0 -> [-1073741824 1073741824)  (auto determined)
  Product         : s29,0 -> [-268435456 268435456)  (auto determined)
  Accumulator     : s31,0 -> [-1073741824 1073741824)  (auto determined)
  Round Mode      : No rounding
  Overflow Mode   : No overflow
```

In this case, all the fractional lengths are now set to zero, meaning that the filter `h` is set up to handle integers.

Create a Test Signal for the Filter

You can generate an input signal for the filter by quantizing to 12 bits using the autoscaling feature, or you can follow the same procedure that was used for the coefficients, discussed previously. In this example, create a signal with two sinusoids:

```
n = 0:999;
f1 = 0.1*pi;  % Normalized frequency of first sinusoid
f2 = 0.8*pi;  % Normalized frequency of second sinusoid
x = 0.9*sin(0.1*pi*n) + 0.9*sin(0.8*pi*n);
xq = fi(x, true, 12);  % signed = true, B = 12
xsc = fi(xq.int, true, 12, 0);
```

Filter the Test Signal

To filter the input signal generated above, enter the following:

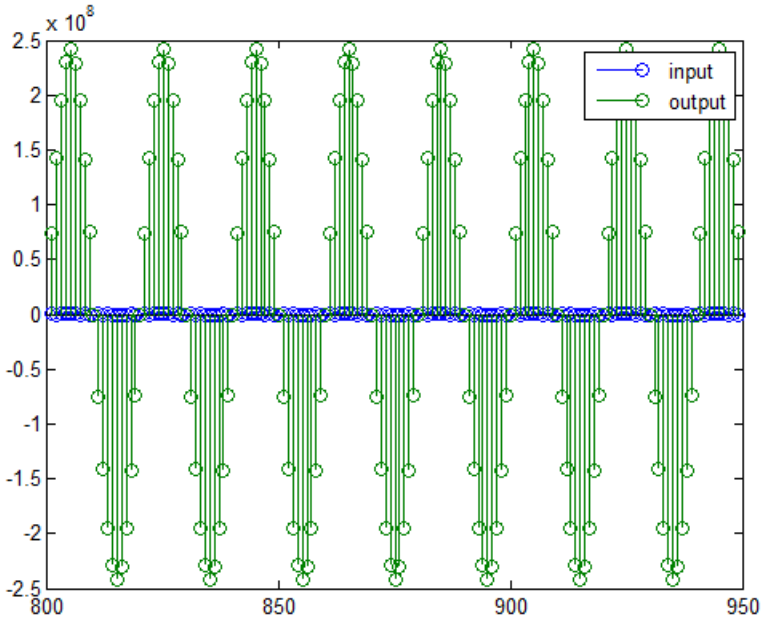
```
ysc = filter(h, xsc);
```

Here `ysc` is a full precision output, meaning that no bits have been discarded in the computation. This makes `ysc` the best possible output you can achieve given the 12–bit input and the 18–bit coefficients. This can be verified by filtering using double-precision floating-point and comparing the results of the two filtering operations:

```
hd = double(h);
xd = double(xsc);
yd = filter(hd, xd);
norm(yd-double(ysc))

ans =

     0
```

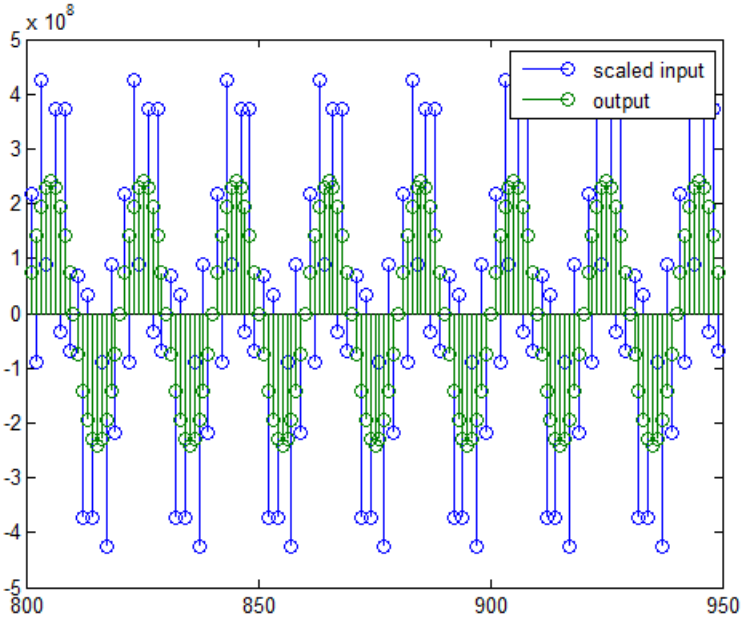
Now you can examine the output compared to the input. This example is plotting only the last few samples to minimize the effect of transients:

```
idx = 800:950;
xscext = double(xsc(idx)');
gd = grpdelay(h, [f1 f2]);
yidx = idx + gd(1);
yscext = double(ysc(yidx)');
stem(n(idx)', [xscext, yscext]);
axis([800 950 -2.5e8 2.5e8]);
legend('input', 'output');
set(gcf, 'color', 'white');
```



It is difficult to compare the two signals in this figure because of the large difference in scales. This is due to the large gain of the filter, so you will need to compensate for the filter gain:

```
stem(n(idx)', [2^18*xscext, yscext]);
axis([800 950 -5e8 5e8]);
legend('scaled input', 'output');
```



You can see how the signals compare much more easily once the scaling has been done, as seen in the above figure.

Truncate the Output WordLength

If you examine the output wordlength,

```
ysc.WordLength

ans =

    31
```

you will notice that the number of bits in the output is considerably greater than in the input. Because such growth in the number of bits representing the data may not be desirable, you may need to truncate the wordlength of the output. The best way to do this is to discard the least significant bits, in order to minimize error. However, if you know there are *unused* high order bits, you should discard those bits as well.

To determine if there are unused most significant bits (MSBs), you can look at where the growth in WordLength arises in the computation. In this case, the bit growth occurs to accommodate the results of adding products of the input (12 bits) and the coefficients (18 bits). Each of these products is 29 bits long (you can verify this using `info(h)`). The bit growth due to the accumulation of the product depends on the filter length and the coefficient values- however, this is a worst-case determination in the sense that no assumption on the input signal is made besides, and as a result there may be unused MSBs. You will have to be careful though, as MSBs that are deemed unused incorrectly will cause overflows.

Suppose you want to keep 16 bits for the output. In this case, there is no bit-growth due to the additions, so the output bit setting will be 16 for the wordlength and –14 for the fraction length.

Since the filtering has already been done, you can discard some bits from `ysc`:

```
yout = fi(ysc, true, 16, -14);
```

Alternatively, you can set the filter output bit lengths directly (this is useful if you plan on filtering many signals):

```
specifyall(h);
h.OutputWordLength = 16;
h.OutputFracLength = -14;
yout2 = filter(h, xsc);
```

You can verify that the results are the same either way:

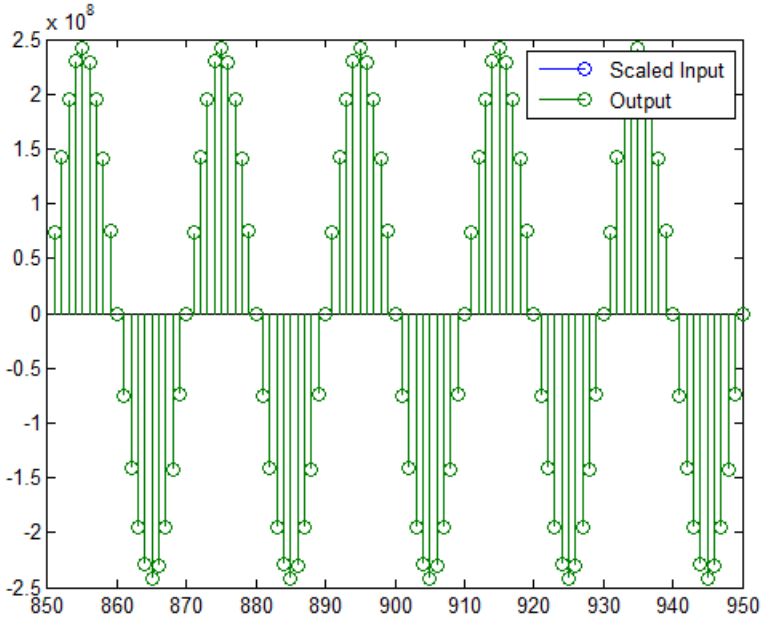
```
norm(double(yout) - double(yout2))

ans =

     0
```

However, if you compare this to the full precision output, you will notice that there is rounding error due to the discarded bits:

```
norm(double(yout)-double(ysc))

ans =

    1.446323386867543e+005
```

In this case the differences are hard to spot when plotting the data, as seen below:

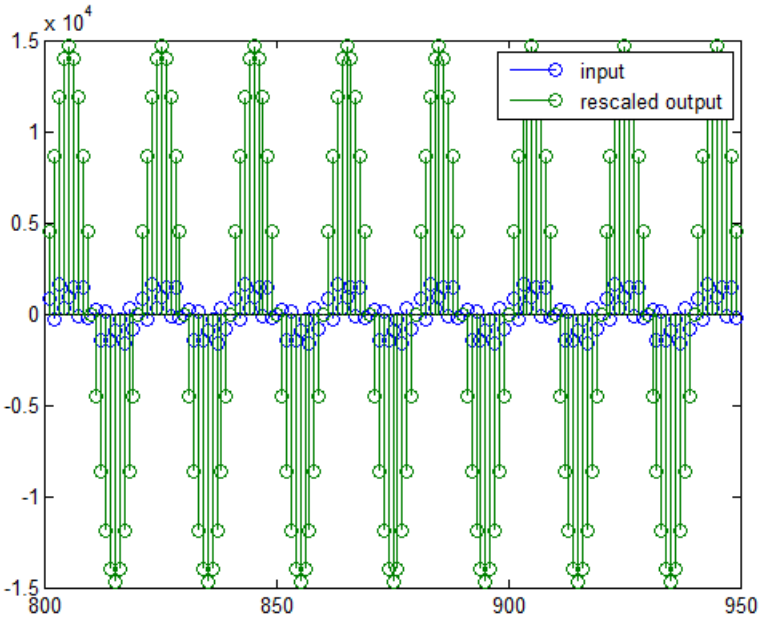
```
stem(n(yidx), [double(yout(yidx)'), double(ysc(yidx)')]);
axis([850 950 -2.5e8 2.5e8]);
legend('Scaled Input', 'Output');
set(gcf, 'color', 'white');
```



Scale the Output

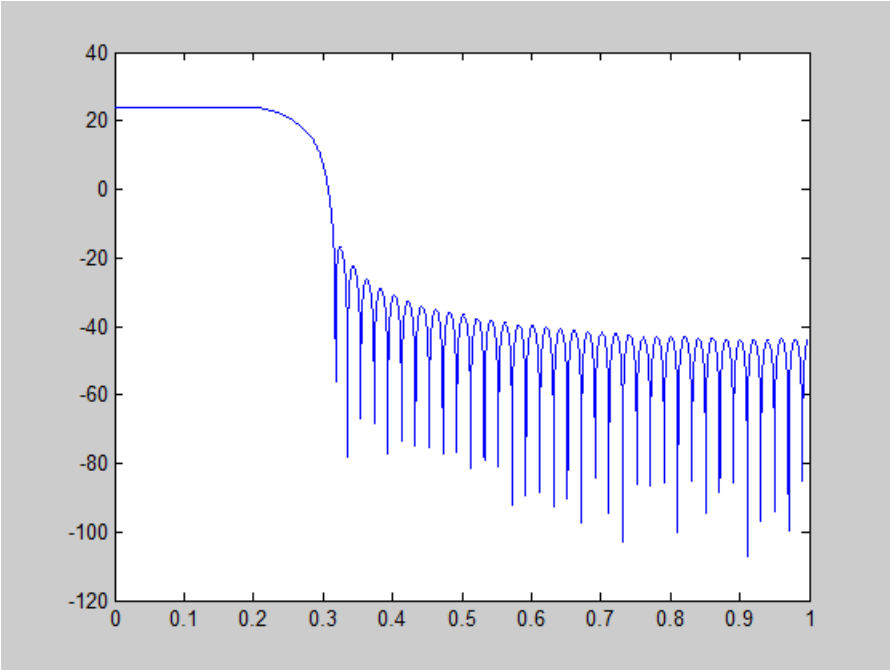
Because the filter in this example has such a large gain, the output is at a different scale than the input. This scaling is purely theoretical however, and you can scale the data however you like. In this case, you have 16 bits for the output, but you can attach whatever scaling you choose. It would be natural to reinterpret the output to have a weight of 2^0 (or L = 0) for the LSB. This is equivalent to scaling the output signal down by a factor of 2^(-14). However, there is no computation or rounding error involved. You can do this by executing the following:

```
yri = fi(yout.int, true, 16, 0);
stem(n(idx)', [xscext, double(yri(yidx)')]);
axis([800 950 -1.5e4 1.5e4]);
legend('input', 'rescaled output');
```



This plot shows that the output is still larger than the input. If you had done the filtering in double-precision floating-point, this would not be the case— because here more bits are being used for the output than for the input, so the MSBs are weighted differently. You can see this another way by looking at the magnitude response of the scaled filter:

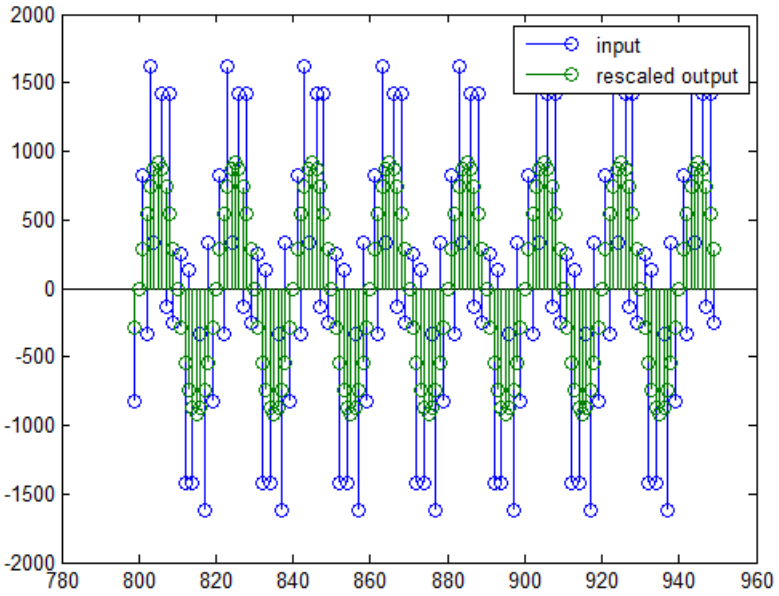
```
[H,w] = freqz(h);
plot(w/pi, 20*log10(2^(-14)*abs(H)));
```



This plot shows that the passband gain is still above 0 dB.

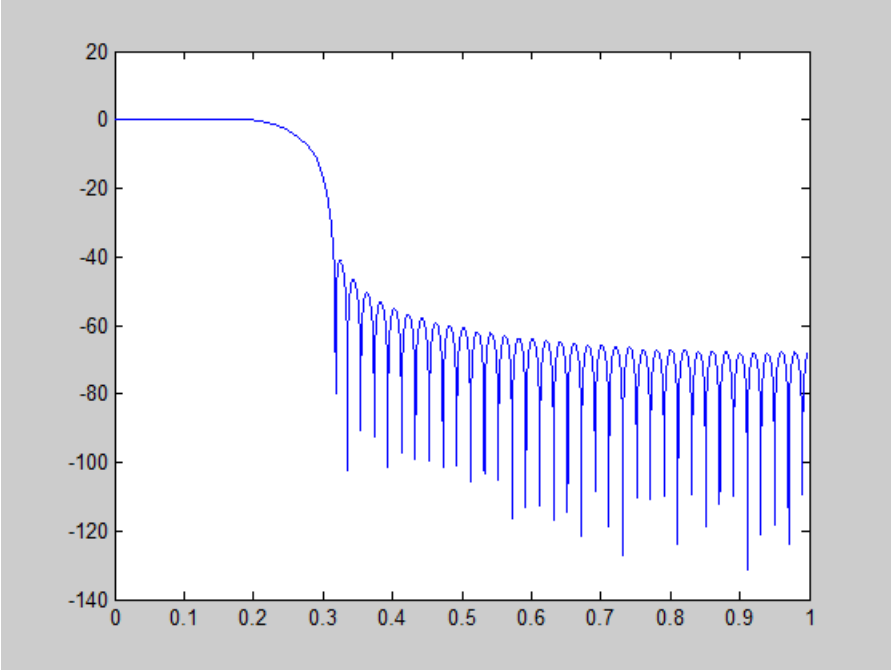
To put the input and output on the same scale, the MSBs must be weighted equally. The input MSB has a weight of 2^11, whereas the scaled output MSB has a weight of 2^(29–14) = 2^15. You need to give the output MSB a weight of 2^11 as follows:

```
yf = fi(zeros(size(yri)), true, 16, 4);
yf.bin = yri.bin;
stem(n(idx)', [xscext, double(yf(yidx)')]);
legend('input', 'rescaled output');
```



This operation is equivalent to scaling the filter gain down by 2^(-18).

```
[H,w] = freqz(h);
plot(w/pi, 20*log10(2^(-18)*abs(H)));
```



The above plot shows a 0 dB gain in the passband, as desired.

With this final version of the output, `yf` is no longer an integer. However this is only due to the interpretation- the integers represented by the bits in `yf` are identical to the ones represented by the bits in `yri`. You can verify this by comparing them:

```
max(abs(yf.int - yri.int))

ans =

      0
```

Configure Filter Parameters to Work with Integers Using the set2int Method

Set the Filter Parameters to Work with Integers

The `set2int` method provides a convenient way of setting filter parameters to work with integers. The method works by scaling the coefficients to integer numbers, and setting the coefficients and input fraction length to zero. This makes it possible for you to use floating-point coefficients directly.

```
h = dfilt.dffir(b);
h.Arithmetic = 'fixed';
```

The coefficients are represented with 18 bits and the input signal is represented with 12 bits:

```
g = set2int(h, 18, 12);
g_dB = 20*log10(g)

g_dB =

    1.083707984390332e+002
```

The `set2int` method returns the gain of the filter by scaling the coefficients to integers, so the gain is always a power of 2. You can verify that the gain we get here is consistent with the gain of the filter previously. Now you can also check that the filter `h` is set up properly to work with integers:

```
info(h)
Discrete-Time FIR Filter (real)
-------------------------------
Filter Structure  : Direct-Form FIR
Filter Length     : 101
Stable            : Yes
Linear Phase      : Yes (Type 1)
Arithmetic        : fixed
Numerator         : s18,0 -> [-131072 131072)
Input             : s12,0 -> [-2048 2048)
Filter Internals  : Full Precision
  Output     : s31,0 -> [-1073741824 1073741824) (auto determined)
  Product    : s29,0 -> [-268435456 268435456) (auto determined)
  Accumulator: s31,0 -> [-1073741824 1073741824) (auto determined)
  Round Mode      : No rounding
  Overflow Mode   : No overflow
```

Here you can see that all fractional lengths are now set to zero, so this filter is set up properly for working with integers.

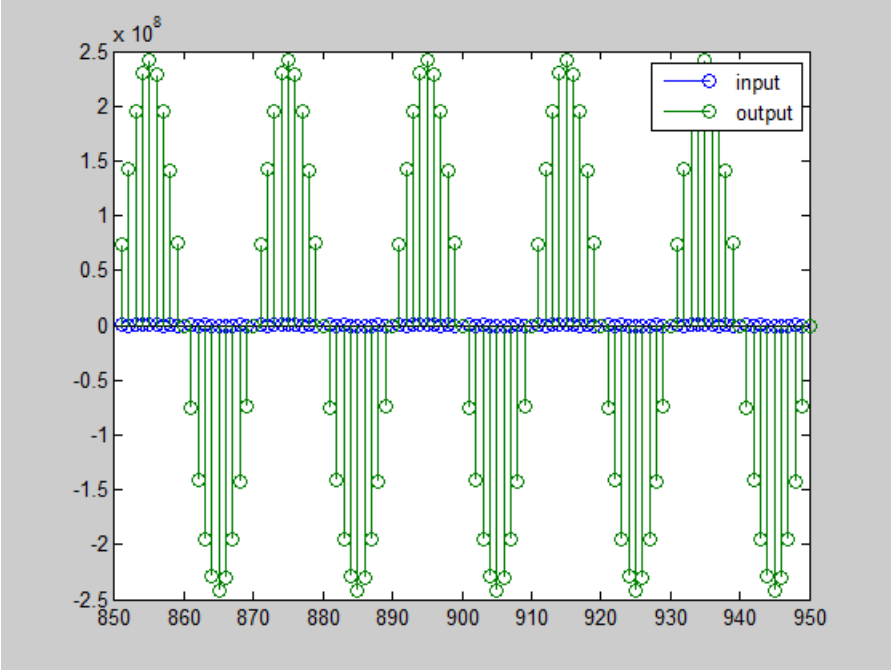
Reinterpret the Output

You can compare the output to the double-precision floating-point reference output, and verify that the computation done by the filter `h` is done in full precision.

```
yint = filter(h, xsc);
norm(yd - double(yint))

ans =

     0
```

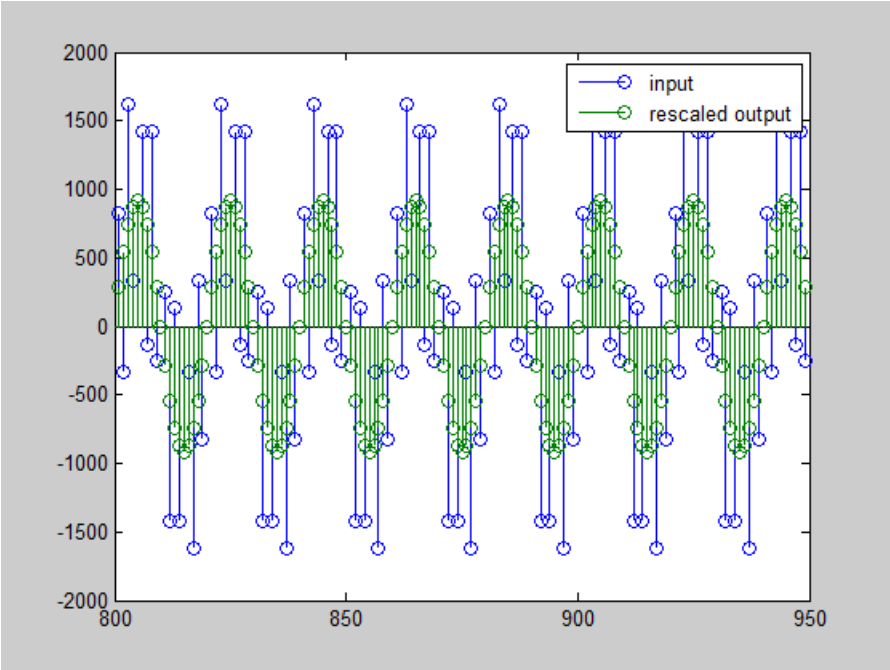
You can then truncate the output to only 16 bits:

```
yout = fi(yint, true, 16);
stem(n(yidx), [xscext, double(yout(yidx)')]);
axis([850 950 -2.5e8 2.5e8]);
legend('input', 'output');
```



Once again, the plot shows that the input and output are at different scales. In order to scale the output so that the signals can be compared more easily in a plot, you will need to weigh the MSBs appropriately. You can compute the new fraction length using the gain of the filter when the coefficients were integer numbers:

```
WL = yout.WordLength;
FL = yout.FractionLength + log2(g);
yf2 = fi(zeros(size(yout)), true, WL, FL);
yf2.bin = yout.bin;

stem(n(idx)', [xscext, double(yf2(yidx)')]);
axis([800 950 -2e3 2e3]);
legend('input', 'rescaled output');
```



This final plot shows the filtered data re-scaled to match the input scale.

Fixed-Point Precision Rules for Avoiding Overflow in FIR Filters

In this section...
“Output Limits for FIR Filters” on page 17-70
“Fixed-Point Precision Rules” on page 17-72
“Polyphase Interpolators and Decimators” on page 17-73

Fixed-point FIR filters are commonly implemented on digital signal processors, FPGAs, and ASICs. A fixed-point filter uses fixed-point arithmetic and is represented by an equation with fixed-point coefficients. If the accumulator and output of the FIR filter do not have sufficient bits to represent their data, overflow occurs and distorts the signal. Use these two rules to determine FIR filter precision settings automatically. The aim is to minimize resource utilization (memory/storage and processing elements) while avoiding overflow. Because the rules are optimized based on the input precision, coefficient precision, and the coefficient values, the FIR filter must have nontunable coefficients.

The precision rules define the minimum and the maximum values of the FIR filter output. To determine these values, perform min/max analysis on the FIR filter coefficients.

Output Limits for FIR Filters

FIR filter is defined by:

$$y[n] = \sum_{k=0}^{N-1} h_k x[n-k]$$

- *x[n]* is the input signal.
- *y[n]* is the output signal.
- h_k is the k^{th} filter coefficient.
- N is the length of the filter.

Output Limits for FIR Filters with Real Input and Real Coefficients

Let the minimum value of the input signal be X_{min}, where $X_{min} \le 0$, and the maximum value be X_{max}, where $X_{max} \ge 0$. The minimum output occurs when you multiply the positive coefficients by X_{min} and the negative coefficients by X_{max}. Similarly, the maximum output occurs when you multiply the positive coefficients by X_{max} and the negative coefficients by X_{min}.

If the sum of all the positive coefficients is

$$G^+ = \sum_{k=0, h_k > 0}^{N-1} h_k$$

and the sum of all the negative coefficients is denoted as

$$G^- = \sum_{k=0, h_k < 0}^{N-1} h_k$$

then you can express the minimum output of the filter as

$$Y_{\min} = G^{+}X_{\min} + G^{-}X_{\max}$$

and the maximum output of the filter as

$$Y_{\max} = G^{+}X_{\max} + G^{-}X_{\min}$$

Therefore, the output of the filter lies in the interval $[Y_{min}, Y_{max}]$.

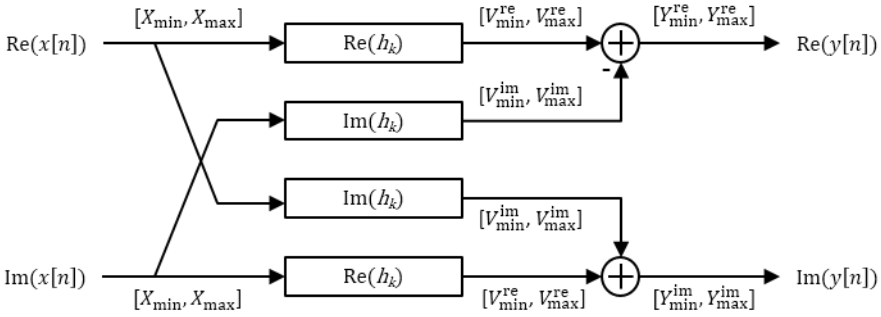
Complex Filter Convolution Equations

You can define a complex filter (complex inputs and complex coefficients) in terms of the real and imaginary parts of its signals and coefficients:

$$\mathrm{Re}(y[n]) = \sum_{k=0}^{N-1} \mathrm{Re}(h_k)\mathrm{Re}(x[n-k]) - \sum_{k=0}^{N-1} \mathrm{Im}(h_k)\mathrm{Im}(x[n-k])$$

$$\mathrm{Im}(y[n]) = \sum_{k=0}^{N-1} \mathrm{Re}(h_k)\mathrm{Im}(x[n-k]) + \sum_{k=0}^{N-1} \mathrm{Im}(h_k)\mathrm{Re}(x[n-k])$$

The complex filter is decomposed into four real filters as depicted in the signal flow diagram. Each signal is annotated with an interval denoting its range.



Output Limits for FIR Filters with Complex Input and Complex Coefficients

You can extend the real filter min/max analysis to complex filters. Assume that both the real and imaginary parts of the input signal lie in the interval $[X_{min}, X_{max}]$.

The complex filter contains two instances of the filter $\mathrm{Re}(h_k)$. Both filters have the same input range and therefore the same output range in the interval $[V^{re}_{min}, V^{re}_{max}]$. Similarly, the complex filter contains two instances of the filter $\mathrm{Im}(h_k)$. Both filters have the same output range in the interval $[V^{im}_{min}, V^{im}_{max}]$.

Based on the min/max analysis of real filters, you can express V^{re}_{min}, V^{re}_{max}, V^{im}_{min}, and V^{im}_{max} as:

$$V^{re}_{\min} = G^{+}_{re}X_{\min} + G^{-}_{re}X_{\max}$$

$$V^{re}_{\max} = G^{+}_{re}X_{\max} + G^{-}_{re}X_{\min}$$

$$V^{im}_{\min} = G^{+}_{im}X_{\min} + G^{-}_{im}X_{\max}$$

$$V^{im}_{\max} = G^{+}_{im}X_{\max} + G^{-}_{im}X_{\min}$$

- G^{+}_{re} is the sum of the positive real parts of h_k, given by

$$G_{re}^{+} = \sum_{k\,=\,0,\,\mathrm{Re}(h_k)\,>\,0}^{N\,-\,1} \mathrm{Re}(h_k)$$

- G^{-}_{re} is the sum of the negative real parts of h_k, given by

$$G_{re}^{-} = \sum_{k\,=\,0,\,\mathrm{Re}(h_k)\,<\,0}^{N\,-\,1} \mathrm{Re}(h_k)$$

- G^{+}_{im} is the sum of the positive imaginary parts of h_k, given by

$$G_{im}^{+} = \sum_{k\,=\,0,\,\mathrm{Im}(h_k)\,>\,0}^{N\,-\,1} \mathrm{Im}(h_k)$$

- G^{-}_{im} is the sum of the negative imaginary parts of h_k, given by

$$G_{im}^{-} = \sum_{k\,=\,0,\,\mathrm{Im}(h_k)\,<\,0}^{N\,-\,1} \mathrm{Im}(h_k)$$

The minimum and maximum values of the real and imaginary parts of the output are:

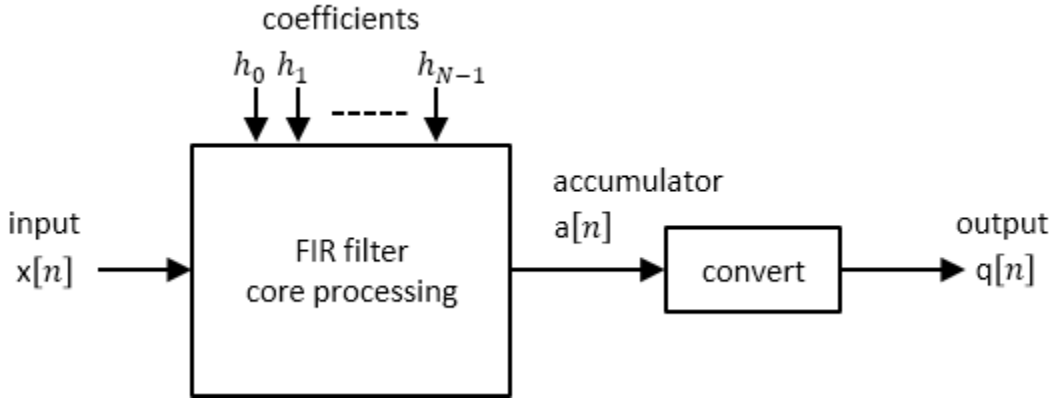
$$Y_{\min}^{re} = V_{\min}^{re} - V_{\max}^{im}$$
$$Y_{\max}^{re} = V_{\max}^{re} - V_{\min}^{im}$$
$$Y_{\min}^{im} = V_{\min}^{re} + V_{\min}^{im}$$
$$Y_{\max}^{im} = V_{\max}^{re} + V_{\max}^{im}$$

The worst-case minimum and maximum on either the real or imaginary part of the output is given by

$$Y_{\min} = \min(Y_{\min}^{re}, Y_{\min}^{im})$$
$$Y_{\max} = \max(Y_{\max}^{re}, Y_{\max}^{im})$$

Fixed-Point Precision Rules

The fixed-point precision rules define the output word length and fraction length of the filter in terms of the accumulator word length and fraction length.



Full-Precision Accumulator Rule

Assume that the input is a signed or unsigned fixed-point signal with word length W_x and fraction length F_x. Also assume that the coefficients are signed or unsigned fixed-point values with fraction length F_h. You can now define full precision as the fixed-point settings that minimize the word length of the accumulator while avoiding overflow or any loss of precision.

- The accumulator fraction length is equal to the product fraction length, which is the sum of the input and coefficient fraction lengths.

  $$F_a = F_x + F_h$$

- If $Y_{min} = 0$, then the accumulator is unsigned with word length

  $$W_a = \left\lceil \log_2(Y_{\max} 2^{F_a} + 1) \right\rceil$$

If $Y_{min} > 0$, then the accumulator is signed with word length

$$W_a = \left\lceil \log_2(\max(-Y_{\min} 2^{F_a}, Y_{\max} 2^{F_a} + 1)) \right\rceil + 1$$

The ceil operator rounds to the nearest integer towards $+\infty$.

Output Same Word Length as Input Rule

This rule sets the output word length to be the same as the input word length. Then, it adjusts the fraction length to avoid overflow. W_q is the output word length and F_q is the output fraction length.

Truncate the accumulator to make the output word length same as the input word length.

$$W_q = W_x$$

.

Set the output fraction length F_q to

$$F_q = F_a - (W_a - W_x)$$

.

Polyphase Interpolators and Decimators

You can extend these rules to polyphase FIR interpolators and decimators.

FIR Interpolators

Treat each polyphase branch of the FIR interpolator as a separate FIR filter. The output data type of the FIR interpolator is the worst-case data type of all the polyphase branches.

FIR Decimators

For decimators, the polyphase branches add up at the output. Hence, the output data type is computed as if it were a single FIR filter with all the coefficients of all the polyphase branches.

See Also

More About

- “Fixed-Point Concepts and Terminology” on page 17-4
- “System Objects Supported by Fixed-Point Converter App” on page 17-26
- “Floating-Point to Fixed-Point Conversion of IIR Filters”

C Code Generation

Learn how to generate code for signal processing applications.

- “Functions and System Objects Supported for C Code Generation” on page 18-2
- “Understanding C Code Generation in DSP System Toolbox” on page 18-8
- “Generate C Code from MATLAB Code” on page 18-12
- “Relocate Code Generated from MATLAB Code to Another Development Environment” on page 18-19
- “Generate C Code from Simulink Model” on page 18-21
- “Relocate Code Generated from a Simulink Model to Another Development Environment” on page 18-26
- “How To Run a Generated Executable Outside MATLAB” on page 18-29
- “Use Generated Code to Accelerate an Application Deployed with MATLAB Compiler” on page 18-32
- “How Is dspunfold Different from parfor?” on page 18-43
- “Workflow for Generating a Multithreaded MEX File using dspunfold” on page 18-45
- “Why Does the Analyzer Choose the Wrong State Length?” on page 18-49
- “Why Does the Analyzer Choose a Zero State Length?” on page 18-51
- “Array Plot with Android Devices” on page 18-52

Functions and System Objects Supported for C Code Generation

If you have a MATLAB Coder license, you can generate C and C++ code from MATLAB code that contains DSP System Toolbox functions and System objects. For more information about C and C++ code generation from MATLAB code, see the MATLAB Coder documentation. For more information about generating code from System objects, see “System Objects in MATLAB Code Generation” (MATLAB Coder).

The following DSP System Toolbox functions and System objects are supported for C and C++ code generation from MATLAB code.

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

`audioDeviceWriter*`	Play to sound card
`ca2tf*`	Convert coupled allpass filter to transfer function form
`cl2tf*`	Convert coupled allpass lattice to transfer function form
`designMultirateFIR*`	Multirate FIR filter design
`dsp.AdaptiveLatticeFilter*`	Adaptive lattice filter
`dsp.AffineProjectionFilter*`	Compute output, error and coefficients using affine projection (AP) Algorithm
`dsp.AllpassFilter*`	Single section or cascaded allpass filter
`dsp.AllpoleFilter*`	IIR Filter with no zeros
`dsp.AnalyticSignal*`	Analytic signals of discrete-time inputs
`dsp.ArrayPlot*`	Display vectors or arrays
`dsp.ArrayVectorAdder*`	Add array to vector along specified dimension
`dsp.ArrayVectorDivider*`	Divide array by vector along specified dimension
`dsp.ArrayVectorMultiplier*`	Multiply array by vector along specified dimension
`dsp.ArrayVectorSubtractor*`	Subtract vector from array along specified dimension
`dsp.AsyncBuffer*`	FIFO buffer
`dsp.AudioFileReader*`	Stream from audio file
`dsp.AudioFileWriter*`	Stream to audio file
`dsp.Autocorrelator*`	Autocorrelation sequence
`dsp.BinaryFileReader*`	Read data from binary file
`dsp.BinaryFileWriter*`	Write data to binary files

`dsp.BiquadFilter*`	IIR filter using biquadratic structures
`dsp.BlockLMSFilter*`	Compute output, error, and weights using block LMS adaptive algorithm
`dsp.BurgAREstimator*`	(To be removed) Estimate of autoregressive (AR) model parameters using Burg method
`dsp.BurgSpectrumEstimator*`	(To be removed) Parametric spectral estimate using Burg method
`dsp.CepstralToLPC*`	(To be removed) Convert cepstral coefficients to linear prediction coefficients
`dsp.Channelizer*`	Polyphase FFT analysis filter bank
`dsp.ChannelSynthesizer*`	Polyphase FFT synthesis filter bank
`dsp.CICCompensationDecimator*`	Compensate for CIC decimation filter using FIR decimator
`dsp.CICCompensationInterpolator*`	Compensate for CIC interpolation filter using FIR interpolator
`dsp.CICDecimator*`	Decimate signal using cascaded integrator-comb (CIC) filter
`dsp.CICInterpolator*`	Interpolate signal using cascaded integrator-comb filter
`dsp.ColoredNoise*`	Generate colored noise signal
`dsp.ComplexBandpassDecimator*`	Extract a frequency subband using a one-sided (complex) bandpass decimator
`dsp.Convolver*`	Convolution of two signals
`dsp.Counter*`	Count up or down through specified range of numbers
`dsp.Crosscorrelator*`	Cross-correlation of two inputs
`dsp.CrossSpectrumEstimator*`	Estimate cross-spectral density
`dsp.CumulativeProduct*`	Cumulative product of channel, column, or row elements
`dsp.CumulativeSum*`	Cumulative sum of channel, column, or row elements
`dsp.DCBlocker*`	Block DC component (offset) from input signal
`dsp.DCT*`	(To be removed) Discrete cosine transform (DCT)
`dsp.Delay*`	Delay input signal by fixed samples
`dsp.DelayLine*`	Rebuffer sequence of inputs with one-sample shift
`dsp.Differentiator*`	Direct form FIR fullband differentiator filter
`dsp.DigitalDownConverter*`	Translate digital signal from intermediate frequency (IF) band to baseband and decimate it
`dsp.DigitalUpConverter*`	Interpolate digital signal and translate it from baseband to IF band
`dsp.FarrowRateConverter*`	Polynomial sample rate converter with arbitrary conversion factor
`dsp.FastTransversalFilter*`	Fast transversal least-squares FIR adaptive filter
`dsp.FFT*`	Discrete Fourier transform

`dsp.FilterCascade*`	Create cascade of filter System objects
`dsp.FilteredXLMSFilter*`	Filtered XLMS filter
`dsp.FIRDecimator*`	Polyphase FIR decimator
`dsp.FIRFilter*`	Static or time-varying FIR filter
`dsp.FIRHalfbandDecimator*`	Halfband decimator
`dsp.FIRHalfbandInterpolator*`	Halfband interpolator
`dsp.FIRInterpolator*`	Polyphase FIR interpolator
`dsp.FIRRateConverter*`	Sample rate converter
`dsp.FrequencyDomainAdaptiveFilter*`	Compute output, error, and coefficients using frequency-domain FIR adaptive filter
`dsp.FrequencyDomainFIRFilter*`	Filter input signal in frequency domain
`dsp.HampelFilter*`	Filter outliers using Hampel identifier
`dsp.HighpassFilter*`	FIR or IIR highpass filter
`dsp.Histogram*`	(To be removed) Histogram of input or sequence of inputs
`dsp.IDCT*`	(To be removed) Inverse discrete cosine transform (IDCT)
`dsp.IFFT*`	Inverse discrete Fourier transform (IDFT)
`dsp.IIRFilter*`	Infinite impulse response (IIR) filter
`dsp.IIRHalfbandDecimator*`	Decimate by factor of two using polyphase IIR
`dsp.IIRHalfbandInterpolator*`	Interpolate by a factor of two using polyphase IIR
`dsp.Interpolator*`	Linear or polyphase FIR interpolation
`dsp.ISTFT*`	Inverse short-time FFT
`dsp.KalmanFilter*`	Estimate system measurements and states using Kalman filter
`dsp.LDLFactor*`	Factor square Hermitian positive definite matrices into components
`dsp.LevinsonSolver*`	Solve linear system of equations using Levinson-Durbin recursion
`dsp.LMSFilter*`	Compute output, error, and weights of LMS adaptive filter
`dsp.LowerTriangularSolver*`	Solve lower-triangular matrix equation
`dsp.LowpassFilter*`	FIR or IIR lowpass filter
`dsp.LPCToAutocorrelation*`	(To be removed) Convert linear prediction coefficients to autocorrelation coefficients
`dsp.LPCToCepstral*`	(To be removed) Convert linear prediction coefficients to cepstral coefficients
`dsp.LPCToLSF*`	(To be removed) Convert linear prediction coefficients to line spectral frequencies
`dsp.LPCToLSP*`	(To be removed) Convert linear prediction coefficients to line spectral pairs

dsp.LPCToRC*	(To be removed) Convert linear prediction coefficients to reflection coefficients
dsp.LSFToLPC*	(To be removed) Convert line spectral frequencies to linear prediction coefficients
dsp.LSPToLPC*	(To be removed) Convert line spectral pairs to linear prediction coefficients
dsp.LUFactor*	Factor square matrix into lower and upper triangular matrices
dsp.Maximum*	(To be removed) Find maximum value of input or sequence of inputs
dsp.Mean*	(To be removed) Find mean value of input or sequence of inputs
dsp.Median*	(To be removed) Median value of input
dsp.MedianFilter*	Median filter
dsp.Minimum*	(To be removed) Find minimum values of input or sequence of inputs
dsp.MovingAverage*	Moving average
dsp.MovingMaximum*	Moving maximum
dsp.MovingMinimum*	Moving minimum
dsp.MovingRMS*	Moving root mean square
dsp.MovingStandardDeviation*	Moving standard deviation
dsp.MovingVariance*	Moving variance
dsp.NCO*	Generate real or complex sinusoidal signals
dsp.Normalizer*	(To be removed) Vector normalization along specified dimension
dsp.PeakFinder*	(To be removed) Identify peak values in input signal
dsp.PeakToPeak*	Peak-to-peak value
dsp.PeakToRMS*	Peak-to-root-mean-square value of vector
dsp.PhaseExtractor*	Extract the unwrapped phase of a complex input
dsp.PhaseUnwrapper*	Unwrap signal phase
dsp.RCToAutocorrelation*	(To be removed) Convert reflection coefficients to autocorrelation coefficients
dsp.RCToLPC*	(To be removed) Convert reflection coefficients to linear prediction coefficients
dsp.RLSFilter*	Compute output, error and coefficients using recursive least squares (RLS) algorithm
dsp.RMS*	(To be removed) Root mean square of vector elements
dsp.SampleRateConverter*	Multistage sample rate converter
dsp.ScalarQuantizerDecoder*	Convert each index value into quantized output value
dsp.ScalarQuantizerEncoder*	Associate input value with index value of quantization region
dsp.SignalSource*	Import variable from workspace
dsp.SineWave*	Generate discrete sine wave
dsp.SpectrumAnalyzer*	Display frequency spectrum of time-domain signals

`dsp.SpectrumEstimator*`	Estimate power spectrum or power density spectrum
`dsp.SOSFilter*`	Second-order section biquadratic IIR filter structures
`dsp.StandardDeviation*`	(To be removed) Standard deviation of input or sequence of inputs
`dsp.StateLevels*`	State-level estimation for bilevel rectangular waveform
`dsp.STFT`	Short-time FFT
`dsp.SubbandAnalysisFilter*`	Decompose signal into high-frequency and low-frequency subbands
`dsp.SubbandSynthesisFilter*`	Reconstruct signal from high-frequency and low-frequency subbands
`dsp.TimeScope*`	Time domain signal display and measurement
`dsp.TransferFunctionEstimator*`	Estimate transfer function
`dsp.UDPReceiver*`	Receive UDP packets from the network
`dsp.UDPSender*`	Send UDP packets to network
`dsp.UpperTriangularSolver*`	Solve upper-triangular matrix equation
`dsp.VariableBandwidthFIRFilter*`	Variable bandwidth FIR filter
`dsp.VariableBandwidthIIRFilter*`	Variable bandwidth IIR filter
`dsp.VariableFractionalDelay*`	Delay input by time-varying fractional number of sample periods
`dsp.VariableIntegerDelay*`	Delay input by time-varying integer number of sample periods
`dsp.Variance*`	(To be removed) Variance of input or sequence of inputs
`dsp.VectorQuantizerDecoder*`	Vector quantizer codeword for given index value
`dsp.VectorQuantizerEncoder*`	Vector quantization encoding
`dsp.Window*`	Apply window to input signal
`dsp.ZeroCrossingDetector*`	Detect zero crossings
`dsp.ZoomFFT *`	High-resolution FFT of a portion of a spectrum
`firceqrip*`	Constrained equiripple FIR filter
`fireqint*`	Equiripple FIR interpolators
`firgr*`	Parks-McClellan FIR filter
`firhalfband*`	Halfband FIR filter design
`firlpnorm*`	Least P-norm optimal FIR filter
`firminphase*`	Minimum-phase FIR spectral factor

`firnyquist*`	Lowpass Nyquist (Lth-band) FIR filter
`firpr2chfb*`	Two-channel FIR filter bank for perfect reconstruction
`ifir*`	Interpolated FIR filter design
`iircomb*`	IIR comb notch or peak filter
`iirgrpdelay*`	Optimal IIR filter with prescribed group-delay
`iirlpnorm*`	Least P-norm optimal IIR filter
`iirlpnormc*`	Constrained least Pth-norm optimal IIR filter
`iirnotch*`	Second-order IIR notch filter
`iirpeak*`	Second-order IIR peak or resonator filter
`tf2ca*`	Transfer function to coupled allpass
`tf2cl*`	Transfer function to coupled allpass lattice

Understanding C Code Generation in DSP System Toolbox

In this section...
“Generate C and C++ code from MATLAB code” on page 18-8
“Generate C and C++ Code from a Simulink Model” on page 18-8
“Shared Library Dependencies” on page 18-9
“Generate C Code for ARM Cortex-M and ARM Cortex-A Processors” on page 18-10
“Generate Code for Mobile Devices” on page 18-10

Generate C and C++ code from signal processing algorithms in DSP System Toolbox using the MATLAB Coder and Simulink Coder products. You can integrate the generated code into your projects as source code, static libraries, dynamic libraries, or even as standalone executables. You can also generate code optimized for ARM® Cortex®-M and ARM Cortex-A processors using the Embedded Coder product.

Generate C and C++ code from MATLAB code

Using the MATLAB Coder, you can generate highly optimized ANSI C and C++ code from functions and System objects in DSP System Toolbox. For a list of functions and System objects that support code generation, see “Functions and System Objects Supported for C Code Generation” on page 18-2. You can use either the MATLAB Coder app or the `codegen` function to generate code according to the build type you choose. When the build type is one of the following:

- Source Code –– Generate C source code to integrate with an external project.
- MEX Code –– Generate a MEX function to run inside MATLAB using the default configuration parameters.
- Static library (.lib) –– Generate a binary library for static linking with another project.
- Dynamic library (.dll) –– Generate a binary library for dynamic linking with an external project.
- Executable –– Generate a standalone program (requires a separate main file written in C or C++).

If you use build scripts to specify input parameter types and code generation options, use the `codegen` function.

For an example that illustrates the code generation workflow using the `codegen` function, see “Generate C Code from MATLAB Code” on page 18-12. For detailed information on each of the code generation steps, see “Generate C Code by Using the MATLAB Coder App” (MATLAB Coder) and “Generate C Code at the Command Line” (MATLAB Coder).

In order to improve the execution speed and memory usage of generated code, MATLAB Coder has several optimization options. For more details, see “MATLAB Coder Optimizations in Generated Code” (MATLAB Coder).

Generate C and C++ Code from a Simulink Model

Using the Simulink Coder, you can generate highly optimized ANSI C and C++ code from Simulink blocks in DSP System Toolbox. For a list of blocks that support code generation, open the Simulink block data type support table for DSP System Toolbox. To access this table, type the following command in the MATLAB command prompt:

```
showsignalblockdatatypetable
```

The blocks with 'X' under 'Code Generation Support' column support code generation.

You can generate code from your Simulink model, build an executable, and even run the executable within MATLAB. For an example, see “Generate C Code from Simulink Model” on page 18-21.

For detailed information on each of the code generation steps, see “Generate C Code for a Model” (Simulink Coder).

Generated ANSI C Code Optimizations

The generated C code is often suitable for embedded applications and includes the following optimizations:

- **Function reuse (run-time libraries)** — Reuse of common algorithmic functions via calls to shared utility functions. Shared utility functions are highly optimized ANSI/ISO C functions that implement core algorithms such as FFT and convolution.
- **Parameter reuse (Simulink Coder run-time parameters)** — Multiple instances of a block that have the same value for a specific parameter point to the same variable in the generated code. This process reduces memory requirements.
- **Blocks have parameters that affect code optimization** — Some blocks, such as the Sine Wave block, have parameters that enable you to optimize the simulation for memory or for speed. These optimizations also apply to code generation.
- **Other optimizations** — Use of contiguous input and output arrays, reusable inputs, overwritable arrays, and inlined algorithms provide smaller generated C code that is more efficient at run time.

Shared Library Dependencies

In most cases, the C/C++ code you generate from DSP System Toolbox objects and blocks is portable. After you generate the code, using the pack-and-go utility, you can package and relocate the code to another development environment that does not have MATLAB and Simulink installed. For examples, see “Relocate Code Generated from MATLAB Code to Another Development Environment” on page 18-19 and “Relocate Code Generated from a Simulink Model to Another Development Environment” on page 18-26.

There are a few DSP System Toolbox features that generate code with limited portability. The executables generated from these features rely on prebuilt dynamic library files (.dll files) included with MATLAB. You must include these .dll files when you run the corresponding executables on the external environment. For a list of such objects and blocks and for information on how to run those executables outside MATLAB, see “How To Run a Generated Executable Outside MATLAB” on page 18-29.

Both Simulink Coder and MATLAB Coder provide functions to help you set up and manage the build information for your models. For example, one of the functions that Simulink Coder provides, `getNonBuildFiles`, allows you to identify the shared libraries required by the blocks in your model. If your model contains any blocks that use precompiled shared libraries, you can install those libraries on the target system. The folder that you install the shared libraries in must be on the system path. The target system does not need to have MATLAB installed, but it does need to be supported by MATLAB. For additional information, see “Build Process Customization” (Simulink Coder). The function `getNonBuildFiles` can also apply to MATLAB algorithms. For more information, see “Build Process Customization” (MATLAB Coder).

Generate C Code for ARM Cortex-M and ARM Cortex-A Processors

The DSP System Toolbox supports optimized C code generation for popular algorithms like FIR filtering and FFT on ARM Cortex-M and ARM Cortex-A processors. To generate this optimized code, you must install the Embedded Coder Support Package for ARM Cortex-M Processors or Embedded Coder Support Package for ARM Cortex-A Processors. In addition, you must have the following products: DSP System Toolbox, MATLAB Coder, Embedded Coder, Simulink and Simulink Coder for Simulink based workflows.

Using these Embedded Coder support packages, you can generate C code that can link with the CMSIS library or calls the Ne10 library functions. This generated code can be compiled to provide optimized executables that run on ARM Cortex-M or ARM Cortex-A processors.

You can also port the generated ARM Cortex-M CRL code from MATLAB to KEIL µVision IDE and IAR Embedded Workbench. For details, see Port the Generated ARM Cortex-M CRL Code from MATLAB to KEIL µVision IDE and Port the Generated ARM Cortex-M CRL Code from MATLAB to IAR Embedded Workbench.

To download the Embedded Coder support packages for the ARM Cortex processors, see https://www.mathworks.com/hardware-support.html.

For more information on the support packages and instructions for downloading them, see "Embedded Coder Support Package for ARM Cortex-M Processors" and "Embedded Coder Support Package for ARM Cortex-A Processors".

Generate Code for Mobile Devices

Using Simulink Support Package for Apple iOS Devices, you can create and run Simulink models on the iPhone, iPod Touch, and iPad. You can also monitor and tune the algorithms running on the Apple devices. For an example, see Array Plot with Apple iOS Devices (Simulink Support Package for Apple iOS Devices).

Using Simulink Support Package for Android™ Devices, you can create and run Simulink models on supported Android devices. For an example, see "Array Plot with Android Devices" on page 18-52.

See Also

Functions

`codegen` | `getNonBuildFiles`

More About

- "Generate C Code from MATLAB Code" on page 18-12
- "Generate C Code by Using the MATLAB Coder App" (MATLAB Coder)
- "Generate C Code at the Command Line" (MATLAB Coder)
- "Generate C Code from Simulink Model" on page 18-21
- "Generate C Code for a Model" (Simulink Coder)
- "Relocate Code Generated from MATLAB Code to Another Development Environment" on page 18-19
- "Relocate Code Generated from a Simulink Model to Another Development Environment" on page 18-26

- “Relocate Code to Another Development Environment” (Simulink Coder)
- “Build Process Customization” (Simulink Coder)
- “Build Process Customization” (MATLAB Coder)
- Array Plot with Apple iOS Devices (Simulink Support Package for Apple iOS Devices)
- “Array Plot with Android Devices” on page 18-52

External Websites

- Supported and Compatible Compilers
- Port the Generated ARM Cortex-M CRL Code from MATLAB to KEIL µVision IDE
- Port the Generated ARM Cortex-M CRL Code from MATLAB to IAR Embedded Workbench

Generate C Code from MATLAB Code

MATLAB Coder generates highly optimized ANSI C and C++ code from functions and System objects in DSP System Toolbox . You can deploy this code in a wide variety of applications.

This example generates C code from the “Construct a Sinusoidal Signal Using High Energy FFT Coefficients” example and builds an executable from the generated code.

Here is the MATLAB code for this example:

```
L = 1020;
Sineobject = dsp.SineWave('SamplesPerFrame',L,...
'PhaseOffset',10,'SampleRate',44100,'Frequency',1000);
ft = dsp.FFT('FFTImplementation','FFTW');
ift = dsp.IFFT('FFTImplementation','FFTW','ConjugateSymmetricInput',true);
rng(1);
numIter = 1000;
for Iter = 1:numIter
    Sinewave1 = Sineobject();
    Input = Sinewave1 + 0.01*randn(size(Sinewave1));
    FFTCoeff = ft(Input);
    FFTCoeffMagSq = abs(FFTCoeff).^2;

    EnergyFreqDomain = (1/L)*sum(FFTCoeffMagSq);
    [FFTCoeffSorted, ind] = sort(((1/L)*FFTCoeffMagSq),1,'descend');

    CumFFTCoeffs = cumsum(FFTCoeffSorted);
    EnergyPercent = (CumFFTCoeffs/EnergyFreqDomain)*100;
    Vec = find(EnergyPercent > 99.99);
    FFTCoeffsModified = zeros(L,1);
    FFTCoeffsModified(ind(1:Vec(1))) = FFTCoeff(ind(1:Vec(1)));
    ReconstrSignal = ift(FFTCoeffsModified);
end
max(abs(Input-ReconstrSignal))
plot(Input,'*');
hold on;
plot(ReconstrSignal,'o');
hold off;
```

You can run the generated executable inside the MATLAB environment. In addition, you can package and relocate the code to another development environment that does not have MATLAB installed. You can generate code using the MATLAB Coder app or the `codegen` function. This example shows you the workflow using the `codegen` function. For more information on the app workflow, see “Generate C Code by Using the MATLAB Coder App” (MATLAB Coder).

Set Up the Compiler

The first step is to set up a supported C compiler. MATLAB Coder automatically locates and uses a supported installed compiler. You can change the default compiler using `mex -setup`. For more details, see “Change Default Compiler” (MATLAB). For a current list of supported compilers, see Supported and Compatible Compilers.

Break Out the Computational Part of the Algorithm into a MATLAB Function

To generate C code, the entry point must be a function. You do not have to generate code for the entire MATLAB application. If you have specific portions that are computationally intensive, generate code from these portions in order to speed up your algorithm. The harness or the driver that calls this MATLAB function does not need to generate code. The harness runs in MATLAB and can contain visualization and other verification tools that are not actually part of the system under test. For example, in the “Construct a Sinusoidal Signal Using High Energy FFT Coefficients” example, the `plot` functions plot the input signal and the reconstructed signal. `plot` is not supported for code

generation and must stay in the harness. To generate code from the harness that contains the visualization tools, rewrite the harness as a function and declare the visualization functions as extrinsic functions using `coder.extrinsic`. To run the generated code that contains the extrinsic functions, you must have MATLAB installed on your machine.

The MATLAB code in the `for` loop that reconstructs the original signal using high-energy FFT coefficients is the computationally intensive portion of this algorithm. Speed up the `for` loop by moving this computational part into a function of its own, `GenerateSignalWithHighEnergyFFTCoeffs.m`.

```
L = 1020;
Sineobject = dsp.SineWave('SamplesPerFrame',L,...
    'SampleRate',44100,'Frequency',1000);
rng(1);
numIter = 1000;
for Iter = 1:numIter
    Sinewave1 = Sineobject();
    Input = Sinewave1 + 0.01*randn(size(Sinewave1));
    [ReconstrSignal,numCoeff] = GenerateSignalWithHighEnergyFFTCoeffs(Input);
end
max(abs(Input-ReconstrSignal))
figure(1);
plot(Input)
hold on;
plot(ReconstrSignal,'*')
hold off

function [ReconstrSignal,numCoeff] = GenerateSignalWithHighEnergyFFTCoeffs(Input)

ft = dsp.FFT('FFTImplementation','FFTW');
ift = dsp.IFFT('FFTImplementation','FFTW','ConjugateSymmetricInput',true);

FFTCoeff = ft(Input);
FFTCoeffMagSq = abs(FFTCoeff).^2;
L = size(Input,1);
EnergyF = (1/L)*sum(FFTCoeffMagSq);
[FFTCoeffSorted, ind] = sort(((1/L)*FFTCoeffMagSq),1,'descend');

CumFFTCoeffs = cumsum(FFTCoeffSorted);
EnergyPercent = (CumFFTCoeffs/EnergyF)*100;
Vec = find(EnergyPercent > 99.99);
FFTCoeffsModified = zeros(L,1);
FFTCoeffsModified(ind(1:Vec(1))) = FFTCoeff(ind(1:Vec(1)));
numCoeff = Vec(1);
ReconstrSignal = ift(FFTCoeffsModified);
end
```

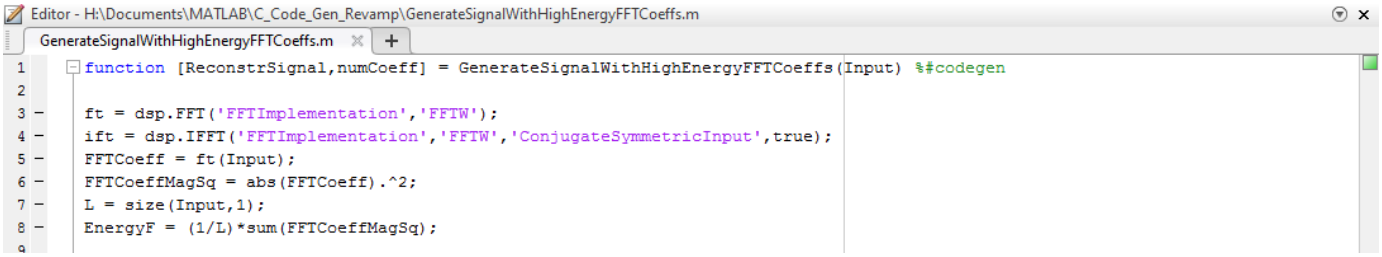
Make Code Suitable for Code Generation

Before you generate code, you must prepare your MATLAB code for code generation.

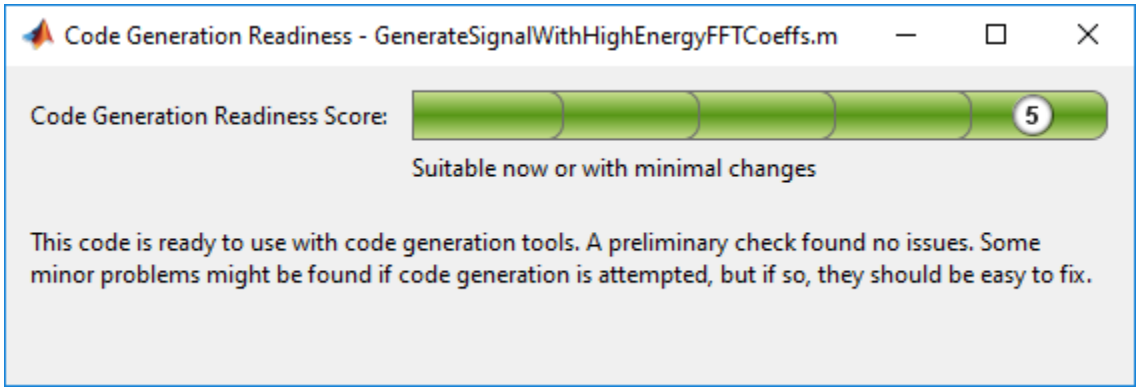
Check Issues at Design Time

The first step is to eliminate unsupported constructs and check for any code generation issues. For a list of DSP System Toolbox features supported by MATLAB Coder, see Functions and System Objects Supported for C Code Generation. For a list of supported language constructs, see “MATLAB Language Features Supported for C/C++ Code Generation” (MATLAB Coder).

The code analyzer detects coding issues at design time as you enter the code. To enable the code analyzer, you must add the `%#codegen` pragma to your MATLAB file.



The code generation readiness tool screens MATLAB code for features that are not supported for code generation. One of the ways to access this tool is by right-clicking on the MATLAB file in its current folder. Running the code generation tool on `GenerateSignalWithHighEnergyFFTCoeffs.m` finds no issues.



Check Issues at Code Generation Time

Before you generate C code, ensure that the MATLAB code successfully generates a MEX function. The `codegen` command used to generate the MEX function detects any errors that prevent the code for being suitable for code generation.

Run `codegen` on `GenerateSignalWithHighEnergyFFTCoeffs.m` function.

```
codegen -args {Input} GenerateSignalWithHighEnergyFFTCoeffs
```

The following message appears in the MATLAB command prompt:

```
??? The left-hand side has been constrained to be non-complex, but the right-hand side
is complex. To correct this problem, make the right-hand side real using the function
REAL, or change the initial assignment to the left-hand side variable to be a complex
value using the COMPLEX function.

Error in ==> GenerateSignalWithHighEnergy Line: 24 Column: 1
Code generation failed: View Error Report
Error using codegen
```

This message is referring to the variable `FFTCoeffsModified`. The coder is expecting this variable to be initialized as a complex variable. To resolve this issue, initialize the `FFTCoeffsModified` variable as complex.

```
FFTCoeffsModified = zeros(L,1)+0i;
```

Rerun the `codegen` function and you can see that a MEX file is generated successfully in the current folder with a `.mex` extension.

```
codegen -args {Input} GenerateSignalWithHighEnergyFFTCoeffs
```

Check Issues at Run Time

Run the generated MEX function to see if there are any run-time issues reported. To do so, replace

```
[ReconstrSignal,numCoeff] = GenerateSignalWithHighEnergyFFTCoeffs(Input);
```

with

```
[ReconstrSignalMex,numCoeffMex] = GenerateSignalWithHighEnergyFFTCoeffs_mex(Input);
```

inside the harness.

The harness now looks like:

```
L = 1020;
Sineobject = dsp.SineWave('SamplesPerFrame',L,...
    'SampleRate',44100,'Frequency',1000);
rng(1);
numIter = 1000;
for Iter = 1:numIter
    Sinewave1 = Sineobject();
    Input = Sinewave1 + 0.01*randn(size(Sinewave1));
    [ReconstrSignalMex,numCoeffMex] = GenerateSignalWithHighEnergyFFTCoeffs_mex(Input,L);
end
max(abs(Input-ReconstrSignalMex))
figure(1);
plot(Input)
hold on;
plot(ReconstrSignalMex,'*')
hold off
```

The code runs successfully, indicating that there are no run-time errors.

Compare the MEX Function with the Simulation

Notice that the harness runs much faster with the MEX function compared to the regular function. The reason for generating the MEX function is not only to detect code generation and run-time issues, but also to speed up specific parts of your algorithm. For an example, see “Signal Processing Algorithm Acceleration in MATLAB” on page 1-51.

You must also check that the numeric output results from the MEX and the regular function match. Compare the reconstructed signal generated by the `GenerateSignalWithHighEnergyFFTCoeffs.m` function and its MEX counterpart `GenerateSignalWithHighEnergyFFTCoeffs_mex`.

```
max(abs(ReconstrSignal-ReconstrSignalMex))

ans =

     2.2204e-16
```

The results match very closely, confirming that the code generation is successful.

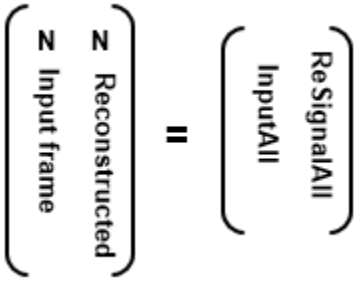
Generate a Standalone Executable

If your goal is to run the generated code inside the MATLAB environment, your build target can just be a MEX function. If deployment of code to another application is the goal, then generate a standalone executable from the entire application. To do so, the harness must be a function that calls the subfunction `GenerateSignalWithHighEnergyFFTCoeffs`. Rewrite the harness as a function.

```
function reconstructSignalTestbench()
L = 1020;
```

```
Sineobject = dsp.SineWave('SamplesPerFrame',L,...
    'SampleRate',44100,'Frequency',1000);
rng(1);
numIter = 1000;
for Iter = 1:numIter
    Sinewave1 = Sineobject();
    Input = Sinewave1 + 0.01*randn(size(Sinewave1));
    [ReconstrSignal,numCoeff] = GenerateSignalWithHighEnergyFFTCoeffs(Input,L);
end
```

Log all 1000 frames of the input and reconstructed signal and the number of FFT coefficients used to reconstruct each frame of the signal. Write all this data to a binary file named `data.bin` using the `dsp.BinaryFileWriter` System object. This example logs the number of coefficients, which are scalar values, as the first element of each frame of the input signal and the reconstructed signal. The data to be written has a frame size of $M = L + 1$ and has a format that looks like this figure.



N is the number of FFT coefficients that represent 99.99% of the signal energy of the current input frame. The meta data of the binary file specifies this information. Release the binary file writer and close the binary file at the end.

The updated harness function, `reconstructSignalTestbench`, is shown here:

```
function reconstructSignalTestbench()
L = 1020;
Sineobject = dsp.SineWave('SamplesPerFrame',L,...
    'SampleRate',44100,'Frequency',1000);
header = struct('FirstElemInBothCols','Number of Coefficients',...
    'FirstColumn','Input','SecondColumn','ReconstructedSignal');
bfw = dsp.BinaryFileWriter('data.bin','HeaderStructure',header);
numIter = 1000;

M = L+1;
ReSignalAll = zeros(M*numIter,1);
InputAll = zeros(M*numIter,1);
rng(1);

for Iter = 1 : numIter
    Sinewave1 = Sineobject();
    Input = Sinewave1 + 0.01*randn(size(Sinewave1));
    [ReconstrSignal,numCoeffs] = GenerateSignalWithHighEnergyFFTCoeffs(Input);
    InputAll(((Iter-1)*M)+1:Iter*M) = [numCoeffs;Input];
    ReSignalAll(((Iter-1)*M)+1:Iter*M) = [numCoeffs;ReconstrSignal];
end

bfw([InputAll ReSignalAll]);
release(bfw);
```

The next step in generating a C executable is to create a `coder.config` object for an executable and provide a `main.c` function to this object.

```
cfg =  coder.config('exe');
cfg.CustomSource = 'reconstructSignalTestbench_Main.c';
```

Here is how the `reconstructSignalTestbench_Main.c` function looks for this example.

```
/*
** reconstructSignalTestbench_main.c
*
* Copyright 2017 The MathWorks, Inc.
*/
#include <stdio.h>
#include <stdlib.h>

#include "reconstructSignalTestbench_initialize.h"
#include "reconstructSignalTestbench.h"
#include "reconstructSignalTestbench_terminate.h"

int main()
{
    reconstructSignalTestbench_initialize();
    reconstructSignalTestbench();
    reconstructSignalTestbench_terminate();

    return 0;
}
```

For additional details on creating the main function, see “Generating Standalone C/C++ Executables from MATLAB Code” (MATLAB Coder).

Set the `CustomInclude` property of the configuration object to specify the location of the main file. In this example, the location is the current folder.

```
cfg.CustomInclude = ['"',pwd,'"'];
```

Generate the C executable by running the following command in the MATLAB command prompt:

```
codegen -config cfg -report reconstructSignalTestbench
```

MATLAB Coder compiles and links the main function with the C code that it generates from the `reconstructSignalTestbench.m`.

If you are using Windows, you can see that `reconstructSignalTestbench.exe` is generated in the current folder. If you are using Linux, the generated executable does not have the `.exe` extension.

Read and Verify the Binary File Data

Running the executable creates a binary file, `data.bin`, in the current directory and writes the input, reconstructed signal, and the number of FFT coefficients used to reconstruct the signal.

```
!reconstructSignalTestbench
```

You can read this data from the binary file using the `dsp.BinaryFileReader` object. To verify that the data is written correctly, read data from the binary file in MATLAB and compare the output with variables `InputAll` and `ReSignalAll`.

The header prototype must have a structure similar to the header structure written to the file. Read the data as two channels.

```
M = 1021;
numIter = 1000;
headerPro = struct('FirstElemInBothCols','Number of Coefficients',...
    'FirstColumn','Input','SecondColumn','ReconstructedSignal');
bfr = dsp.BinaryFileReader('data.bin','HeaderStructure',...
```

```
headerPro,'SamplesPerFrame',M*numIter,'NumChannels',2);
Data = bfr();
```

Compare the first channel with `InputAll` and the second channel with `ReSignalAll`.

```
isequal(InputAll,Data(:,1))

ans =

  logical

   1

isequal(ReSignalAll,Data(:,2))

ans =

  logical

   1
```

The results match exactly, indicating a successful write operation.

Relocate Code to Another Development Environment

Once you generate code from your MATLAB algorithm, you can relocate the code to another development environment, such as a system or an integrated development environment (IDE) that does not include MATLAB. You can package the files into a compressed file using the `packNGo` function at the command line or the **Package** option in the MATLAB Coder app. For an example that illustrates both the workflows, see “Package Code for Other Development Environments” (MATLAB Coder). For more information on the `packNGo` option, see `packNGo` in “Build Information Methods” (MATLAB Coder). You can relocate and unpack the compressed zip file using a standard zip utility. For an example on how to package the executable generated in this example, see “Relocate Code Generated from MATLAB Code to Another Development Environment” on page 18-19.

See Also

Functions

`codegen`

More About

- “Relocate Code Generated from MATLAB Code to Another Development Environment” on page 18-19
- “Generate C Code from Simulink Model” on page 18-21
- “Generate C Code by Using the MATLAB Coder App” (MATLAB Coder)
- “Generate C Code at the Command Line” (MATLAB Coder)
- “Code Generation Workflow” (MATLAB Coder)

External Websites

- Supported and Compatible Compilers

Relocate Code Generated from MATLAB Code to Another Development Environment

Once you generate code from your MATLAB algorithm, you can relocate the code to another development environment, such as a system or an integrated development environment (IDE) that does not include MATLAB. You can package the files into a compressed file using the `packNGo` function at the command line or the **Package** option in the MATLAB Coder app. Once you create the zip file, you can relocate and unpack the compressed zip file using a standard zip utility.

Package the Code

This example shows how to package the executable generated from the "Generate C Code from MATLAB Code" on page 18-12 example using the `packNGo` function. You can also generate and package a static library file or a dynamic library file. You cannot package a C-MEX file since a MEX file requires MATLAB to run. For more information on `packNGo`, see `packNGo` in "Build Information Methods" (MATLAB Coder).

The files needed to generate the executable are `reconstructSignalTestbench.m`, `GenerateSignalWithHighEnergyFFTCoeffs.m`, and the `reconstructSignalTestbench_Main.c` files from the "Generate C Code from MATLAB Code" on page 18-12 example. Copy all these files into the current working folder. To generate the executable, run the following commands in the MATLAB command prompt:

```
cfg =  coder.config('exe');
cfg.CustomSource = 'reconstructSignalTestbench_Main.c';
cfg.CustomInclude = ['"',pwd,'"'];
codegen -config cfg -report reconstructSignalTestbench
```

If you are using Windows, you can see that `reconstructSignalTestbench.exe` is generated in the current folder. If you are using a Linux machine, the generated executable is `reconstructSignalTestbench`. The `codegen` function generates the dependency source code and the `buildinfo.mat` file in the `codegen\exe\reconstructSignalTestbench` folder.

Load the `buildInfo` object.

```
load('codegen\exe\reconstructSignalTestbench\buildinfo.mat')
```

Package the code in a `.zip` file using the `packNGo` function.

```
packNGo(buildInfo,'fileName','reconstructSignalWithHighEnergyFFTCoeffs.zip');
```

The `packNGo` function creates a zip file, `reconstructSignalWithHighEnergyFFTCoeffs.zip` in the current working folder. In this example, you specify only the file name. Optionally, you can specify additional packaging options. See "Specify packNGo Options" (MATLAB Coder).

This `.zip` file contains the C code, header files, `.dll` files, and the executable that needs to run on the external environment. Relocate the `.zip` file to the destination development environment and unpack the file to run the executable.

Prebuilt Dynamic Library Files (.dll)

If you compare the contents of the `codegen\exe\reconstructSignalTestbench` folder and the `reconstructSignalWithHighEnergyFFTCoeffs.zip` folder, you can see that there are

additional `.dll` files that appear in the zip folder. These `.dll` files are prebuilt dynamic library files that are shipped with MATLAB. Executables generated from certain System objects require these prebuilt `.dll` files. The “Generate C Code from MATLAB Code” on page 18-12 example uses `dsp.FFT` and `dsp.IFFT` System objects whose `'FFTImplementation'` is set to `'FFTW'`. In the FFTW mode, the executables generated from these objects depend on the prebuilt `.dll` files. To package code that runs on an environment with no MATLAB installed, MATLAB Coder packages these `.dll` files in the zip folder. For a list of all the System objects in DSP System Toolbox that require prebuilt `.dll` files, see “How To Run a Generated Executable Outside MATLAB” on page 18-29.

To identify the prebuilt `.dll` files your executable requires, run the following command in the MATLAB command prompt.

```
files = getNonBuildFiles(buildInfo,'true','true');
```

For more details, see `getNonBuildFiles` in “Build Process Customization” (MATLAB Coder).

For an example showing the **Package** option workflow to relocate code using the MATLAB Coder app, see “Package Code for Other Development Environments” (MATLAB Coder).

See Also

More About

- “Generate C Code from MATLAB Code” on page 18-12
- “Relocate Code Generated from a Simulink Model to Another Development Environment” on page 18-26
- “Build Information Methods” (MATLAB Coder)
- “Package Code for Other Development Environments” (MATLAB Coder)

Generate C Code from Simulink Model

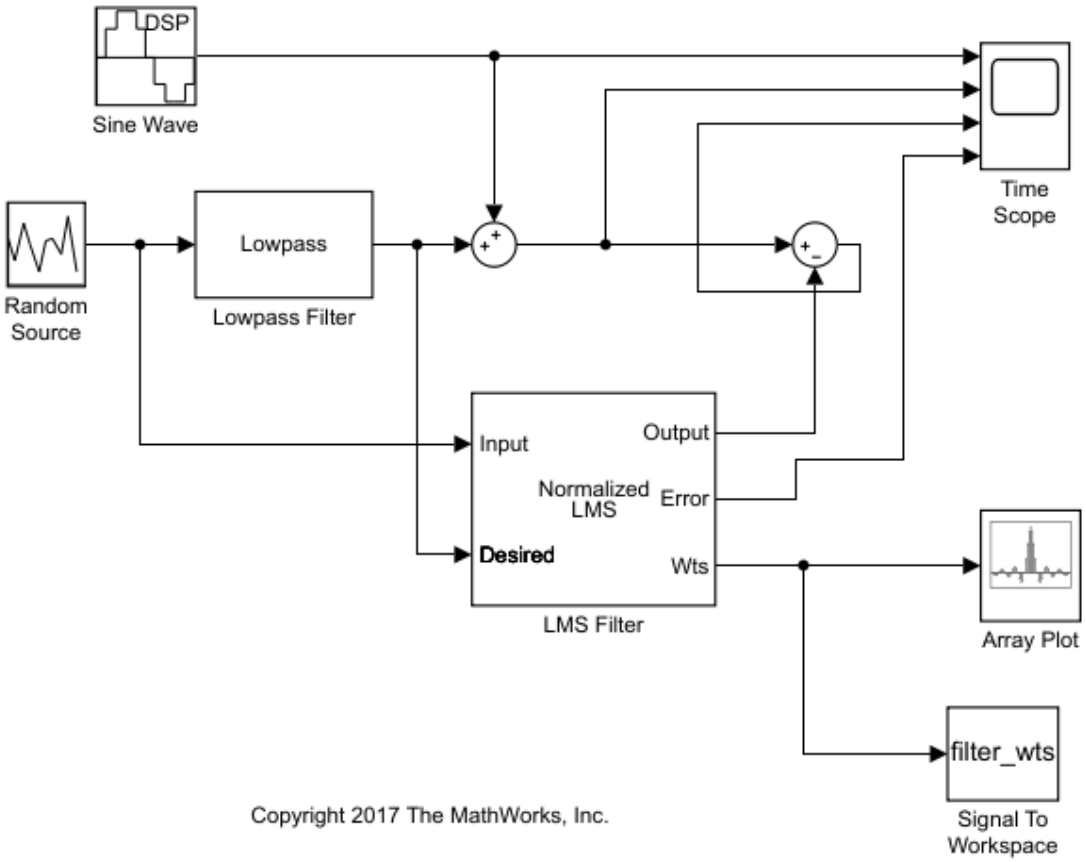
Simulink Coder generates standalone C and C++ code from Simulink models for deployment in a wide variety of applications. For a list of DSP System Toolbox features supported by Simulink Coder, see Blocks Supported for C Code Generation.

This example generates C code from the `ex_codegen_dsp` model and builds an executable from the generated code. You can run the executable inside the MATLAB environment. In addition, you can package and relocate the code to another development environment that does not have the MATLAB and Simulink products installed.

Open the Model

The `ex_codegen_dsp` model implements a simple adaptive filter to remove noise from a signal while simultaneously identifying a filter that characterizes the noise frequency content. To open this model, enter the following command in MATLAB command prompt:

```
open_system('ex_codegen_dsp')
```



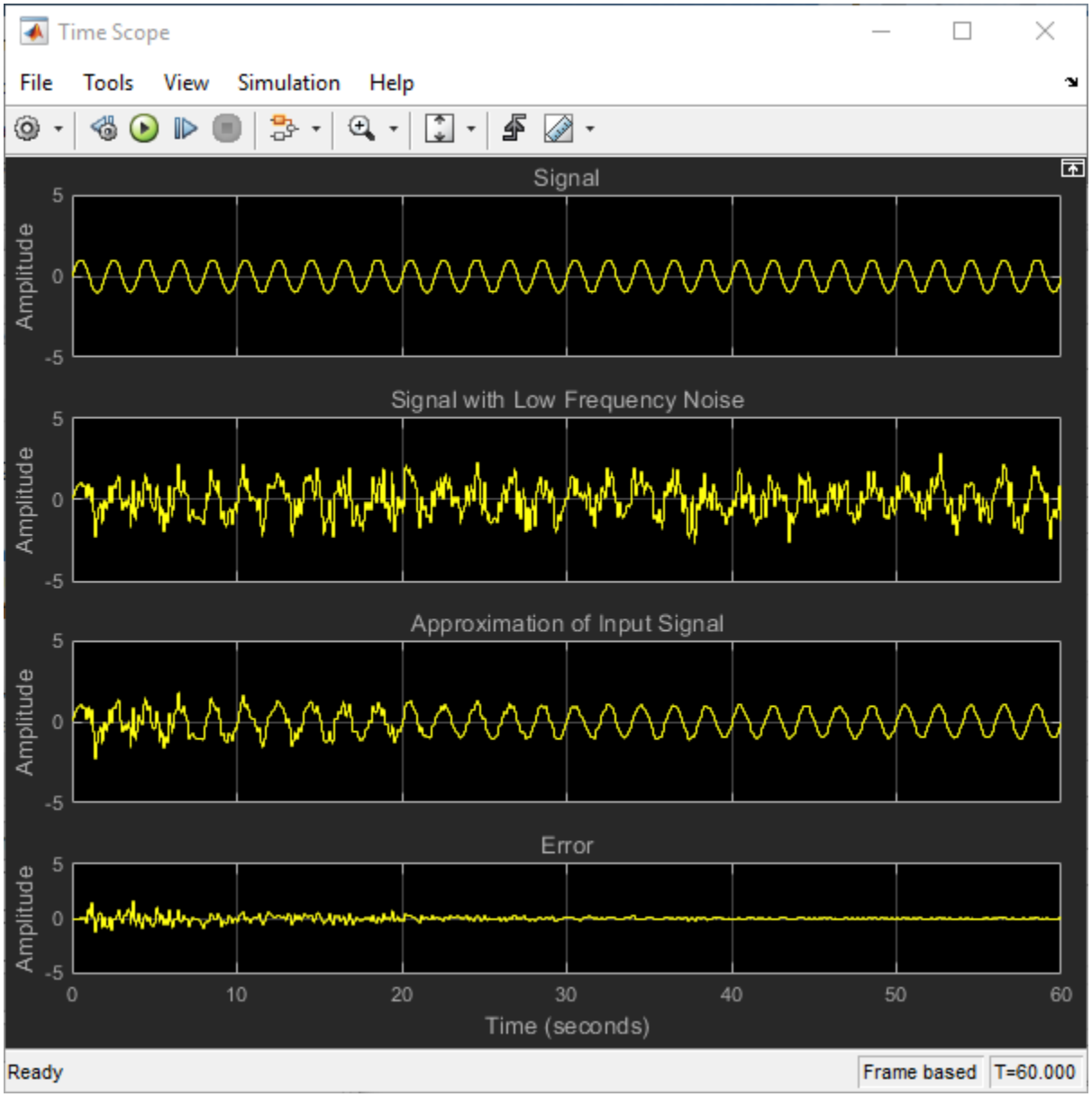
You can alternatively create the model using the **DSP System** template. For more information, see “Configure the Simulink Environment for Signal Processing Models”.

Configure Model for Code Generation

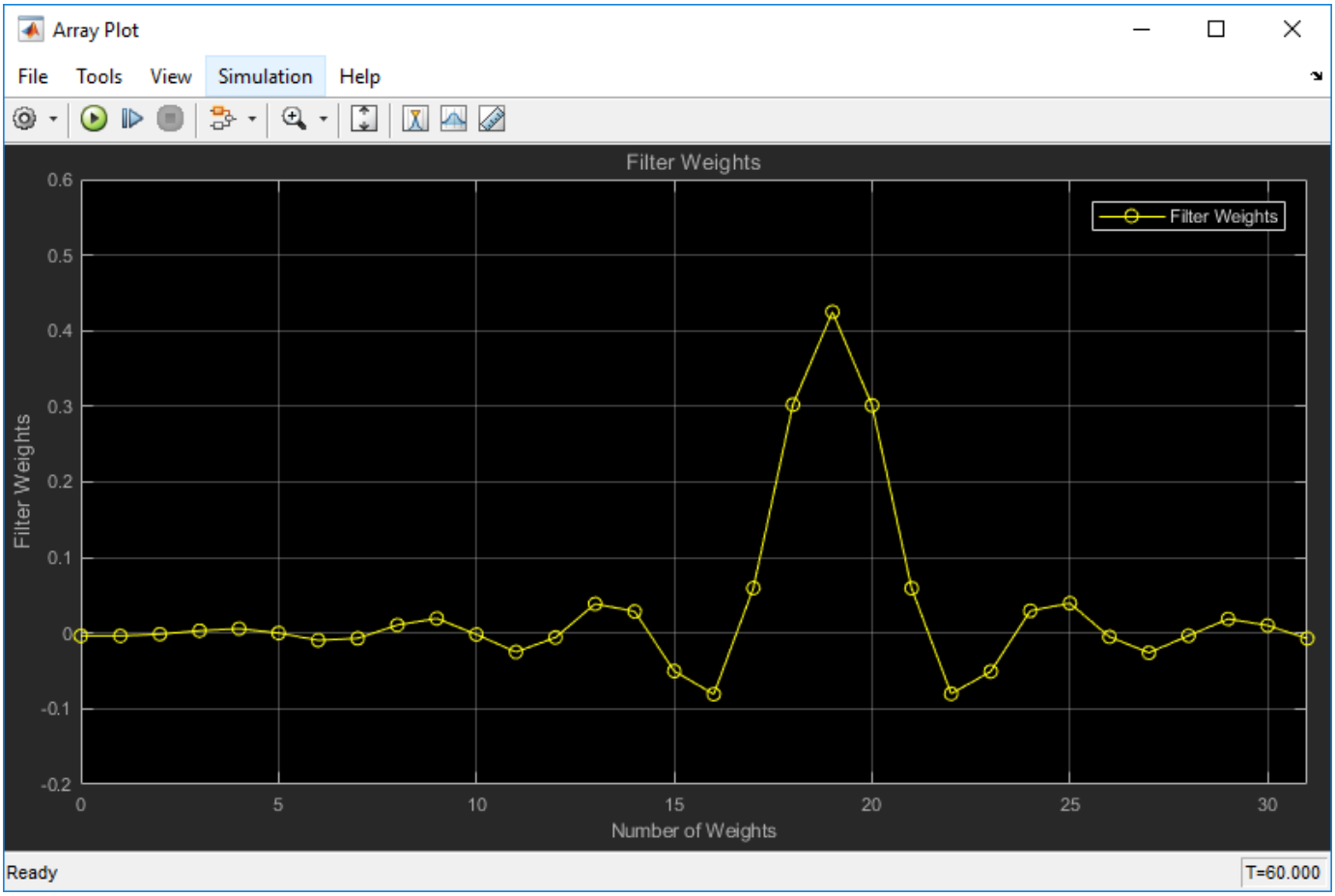
Prepare the model for code generation by specifying code generation settings in the **Configuration Parameters** dialog box. Choose the appropriate solver and code generation target, and check the model configuration for execution efficiency. For more details on each of these steps, see “Generate C Code for a Model” (Simulink Coder).

Simulate the Model

Simulate the model. The Time Scope shows the input and filtered signal characteristics.



The Array Plot shows the last 32 filter weights for which the LMS filter has effectively adapted and filtered out the noise from the signal.



These coefficients can also be accessed using the following command:

```
filter_wts(:,:,1201)
```

Generate Code from the Model

Before you generate code from the model, you must first ensure that you have write permission in your current folder.

To generate code, you must make the following changes:

1. In the **Modeling** tab of the model toolstrip, click **Model Settings**. The **Configuration Parameters** dialog opens. Navigate to the **Code Generation** tab, select the **Generate code only** parameter, and click **Apply**.
2. In the Apps gallery, click **Simulink Coder**. The **C Code** tab appears. Click the **Generate Code** icon ( ).

After the model finishes generating code, the **Code Generation Report** appears, allowing you to inspect the generated code. Note that the build process creates a new subfolder called `ex_codegen_dsp_grt_rtw` in your current MATLAB working folder. This subfolder contains all the files created by the code generation process, including those that contain the generated C source

code. For more information on viewing the generated code, see “Generate C Code for a Model” (Simulink Coder).

Build and Run the Generated Code

Set Up the C/C++ Compiler

To build an executable, you must set up a supported C compiler. For a list of compilers supported in the current release, see Supported and Compatible Compilers.

To set up your compiler, run the following command in the MATLAB command prompt:

```
mex –setup
```

Build the Generated Code

After your compiler is setup, you can build and run the compiled code. The ex_codegen_dsp model is currently configured to generate code only. To build the generated code, you must first make the following changes:

1. In the **Modeling** tab of the model toolstrip, click **Model Settings**. The **Configuration Parameters** dialog opens. Navigate to the **Code Generation** tab, clear the **Generate code only** parameter, and click **Apply**.
2. In the **C Code** tab of the model toolstrip, click the **Build** icon ( ).

The code generator builds the executable and generates the **Code Generation Report**. The code generator places the executable in the working folder. On Windows, the executable is `ex_codegen_dsp.exe`. On Linux, the executable is `ex_codegen_dsp`.

Run the Generated Code

To run the generated code, enter the following command in the MATLAB command prompt:

```
!ex_codegen_dsp
```

Running the generated code creates a MAT-file that contains the same variables as those generated by simulating the model. The variables in the MAT-file are named with a prefix of `rt_`. After you run the generated code, you can load the variables from the MAT-file by typing the following command at the MATLAB prompt:

```
load ex_codegen_dsp.mat
```

You can now compare the variables from the generated code with the variables from the model simulation. To access the last set of coefficients from the generated code, enter the following in the MATLAB prompt:

```
rt_filter_wts(:,:,1201)
```

Note that the coefficients in `filter_wts(:,:,1201)` and `rt_filter_wts(:,:,1201)` match.

For more details on building and running the executable, see “Generate C Code for a Model” (Simulink Coder).

Relocate Code to Another Development Environment

Once you generate code from your Simulink model, you can relocate the code to another development environment using the pack-and-go utility. Use this utility when the development environment does not have the MATLAB and Simulink products.

The pack-and-go utility uses the tools for customizing the build process after code generation and a `packNGo` function to find and package files for building an executable image. The files are packaged in a compressed file that you can relocate and unpack using a standard zip utility.

You can package the code by either using the user interface or by using the command-line interface. The command-line interface provides more control over the details of code packaging. For more information on each of these methods, see “Relocate Code to Another Development Environment” (Simulink Coder).

For an example on how to package the C code and executable generated from this example, see “Relocate Code Generated from a Simulink Model to Another Development Environment” on page 18-26.

See Also

More About

- “Generate C Code for a Model” (Simulink Coder)
- “Relocate Code Generated from a Simulink Model to Another Development Environment” on page 18-26
- “Relocate Code to Another Development Environment” (Simulink Coder)
- “Generate C Code from MATLAB Code” on page 18-12
- “How To Run a Generated Executable Outside MATLAB” on page 18-29

External Websites

- Supported and Compatible Compilers

Relocate Code Generated from a Simulink Model to Another Development Environment

Once you generate code from your Simulink model, you can relocate the code to another development environment using the pack-and-go utility. Use this utility when the development environment does not have the MATLAB and Simulink products.

The pack-and-go utility uses the tools for customizing the build process after code generaton and a `packNGo` function to find and package files for building an executable image. The files are packaged in a compressed file that you can relocate and unpack using a standard zip utility.

You can package the code using either the user interface or the command-line interface. The command-line interface provides more control over the details of code packaging. For more information on each of these methods, see “Relocate Code to Another Development Environment” (Simulink Coder).

Package the Code

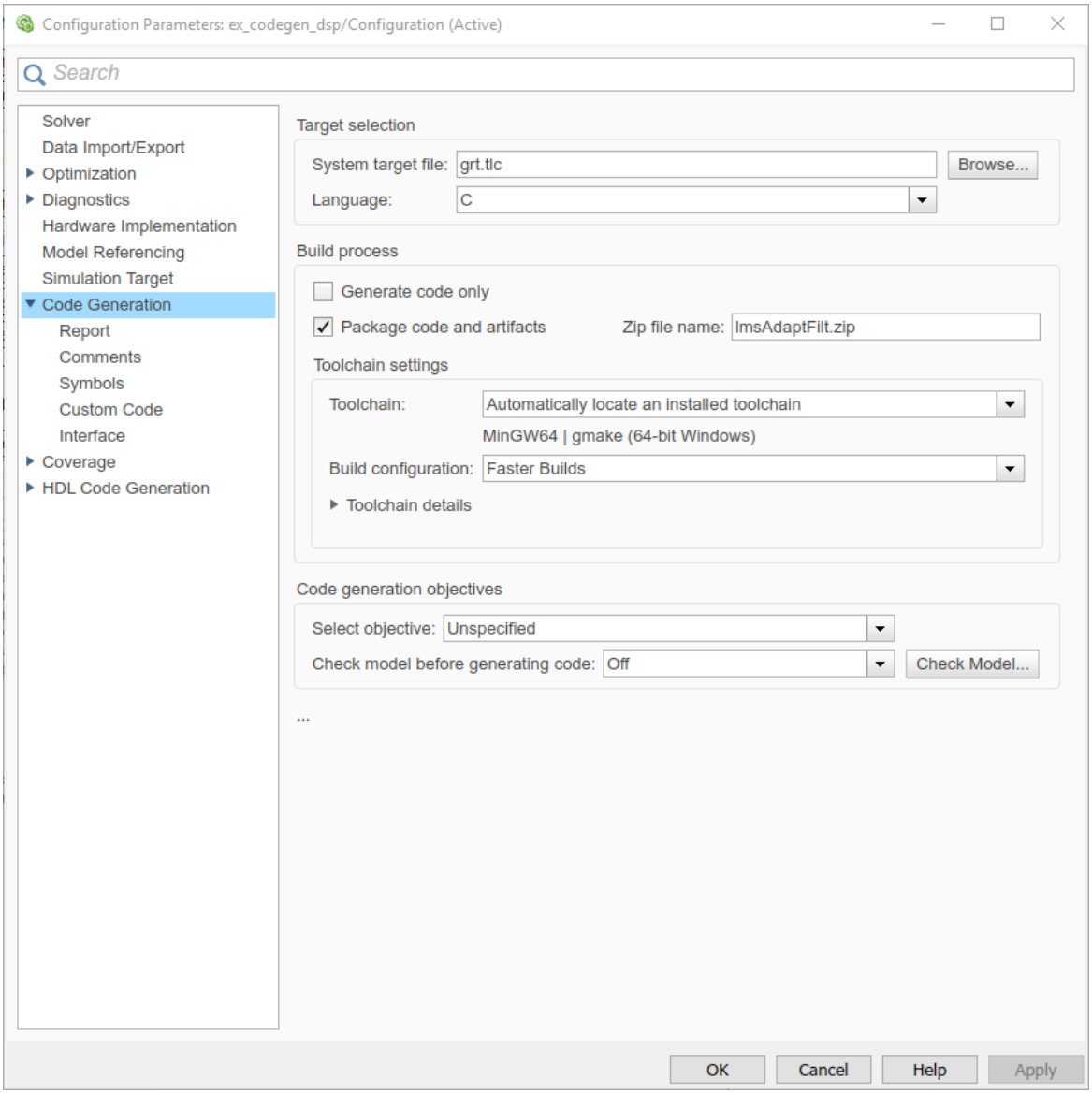
This example shows how to package the executable generated from the `ex_codegen_dsp` model in the “Generate C Code from Simulink Model” on page 18-21 example using the user interface. You can also generate and package a static library file or a dynamic library file.

Open the model by running the following command in the MATLAB command prompt.

```
open_system('ex_codegen_dsp')
```

To package and relocate code for your model using the user interface:

1. In the **Modeling** tab, click **Model Settings**. The **Configuration Parameters** dialog opens. Navigate to the **Code Generation** tab.
2. To package the executable along with the source code, clear **Generate code only** check box and select the option **Package code and artifacts** (Simulink Coder). This option configures the build process to run the `packNGo` function after code generation to package generated code and artifacts for relocation.
3. In the **Zip file name** (Simulink Coder) field, enter the name of the zip file in which to package generated code and artifacts for relocation. In this example, the name of the zip file is `lmsAdaptFilt.zip`. You can specify the file name with or without the `.zip` extension. If you specify no extension or an extension other than `.zip`, the zip utility adds the `.zip` extension. If you do not specify a value, the build process uses the name *`model`*`.zip`, where *`model`* is the name of the top model for which code is being generated.



4 Click **Apply**. In the **C Code** tab of the model toolstrip, click the **Build Model** icon ( ). If the **C Code** tab is not open, in the **Apps** gallery of the model toolstrip, click **Simulink Coder**. The **C Code** tab appears. When you click on the **Build Model** icon ( ), the code generator builds the executable, generates the **Code Generation Report** and places the executable in the current working folder. Note that the build process creates a new subfolder called `ex_codegen_dsp_grt_rtw` in your current MATLAB working folder. This subfolder contains the generated source code files. In addition, you can also see `lmsAdaptFilt.zip` file in the current directory. The zip files contains the `ex_codegen_dsp_grt_rtw` folder, the executable, and other additional dependency source files required to run the executable without Simulink and MATLAB installed.

5 Relocate the zip file to the destination development environment and unpack the file to run the executable.

Prebuilt Dynamic Library Files (.dll)

If your model contains any blocks mentioned in “How To Run a Generated Executable Outside MATLAB” on page 18-29, the executable generated from the model requires certain prebuilt dynamic library (`.dll`) files. These `.dll` files are shipped with MATLAB. To package code that runs on an environment without MATLAB and Simulink installed, the Simulink Coder packages these `.dll` files into the `zip` folder.

See Also

More About

- “Generate C Code from Simulink Model” on page 18-21
- “Relocate Code to Another Development Environment” (Simulink Coder)
- “Relocate Code Generated from MATLAB Code to Another Development Environment” on page 18-19

How To Run a Generated Executable Outside MATLAB

You can generate a standalone executable from the System objects and blocks in DSP System Toolbox which support code generation. This executable can run outside the MATLAB and Simulink environments.

To generate an executable from the System objects, you must have the MATLAB Coder installed. To generate an executable from the Simulink blocks, you must have the Simulink Coder installed in addition to the MATLAB Coder.

The executables generated from the following objects and blocks rely on prebuilt dynamic library files (`.dll` files) included with MATLAB.

System Objects

- `audioDeviceWriter`
- `dsp.AudioFileReader`
- `dsp.AudioFileWriter`
- `dsp.FFT`
  - When `FFTImplementation` is set to `'FFTW'`.
  - When `FFTImplementation` is set to `'Auto'`, `FFTLengthSource` is set to `'Property'`, and `FFTLength` is not a power of two.
- `dsp.IFFT`
  - When `FFTImplementation` is set to `'FFTW'`.
  - When `FFTImplementation` is set to `'Auto'`, `FFTLengthSource` is set to `'Property'`, and `FFTLength` is not a power of two.
- `dsp.UDPReceiver`
- `dsp.UDPSender`

Objects

- `dsp.ISTFT` (when the FFT length determined by the number of input rows is not a power of 2)

Blocks

- Audio Device Writer
- Burg Method (when the FFT length is not a power of two)
- From Multimedia File
- To Multimedia File
- FFT
  - When **FFT implementation** is set to `FFTW`.
  - When you clear the **Inherit FFT length from input dimensions** check box, and set **FFT length** to a value that is not a power of two.
- IFFT
  - When **FFT implementation** is set to `FFTW`.

    - When you clear the **Inherit FFT length from input dimensions** check box, and set **FFT length** to a value that is not a power of two.
- Inverse Short-Time FFT (when the input length is not a power of two)
- Magnitude FFT
    - When **FFT implementation** is set to `FFTW`.
    - When you clear the **Inherit FFT length from input dimensions** check box, and set **FFT length** to a value that is not a power of two.
- Periodogram
    - When **FFT implementation** is set to `FFTW`.
    - When you clear the **Inherit FFT length from input dimensions** check box, and set **FFT length** to a value that is not a power of two.
- Short-Time FFT (when the FFT length is not a power of two)
- UDP Receive
- UDP Send

Running the Executable

To run the corresponding executable outside the MATLAB and Simulink environments, for example Windows command prompt on a Windows machine, you must include these prebuilt `.dll` files. The method of including the `.dll` files depends on whether MATLAB or MATLAB compiler runtime (MCR) is installed on the external machine. The MATLAB compiler runtime (MCR), also known as MATLAB Runtime, is a standalone set of shared libraries, MATLAB code, and other files that enables the execution of MATLAB files on computers without an installed version of MATLAB. For more details on MCR, see “About the MATLAB Runtime” (MATLAB Compiler).

MATLAB or MCR is not installed on the machine you are running the executable

To run the executable generated from the above System objects and blocks on a machine that does not have MATLAB or MCR installed, use the `packNGo` function. The `packNGo` function packages all the relevant files including the prebuilt `.dll` files in a compressed zip file so that you can relocate, unpack, and rebuild your project in another development environment where MATLAB or MCR is not installed.

You can use the `packNGo` function at the command line or the **Package** option in the MATLAB Coder app. The files are packaged in a compressed file that you can relocate and unpack using a standard zip utility. For more details on how to pack the code generated from MATLAB code, see “Relocate Code Generated from MATLAB Code to Another Development Environment” on page 18-19. For more details on how to pack the code generated from Simulink blocks, see “Relocate Code Generated from a Simulink Model to Another Development Environment” on page 18-26.

MATLAB or MCR is installed on the machine you are running the executable

To include the prebuilt `.dll` files on a machine with MATLAB or MCR installed, set your system environment by running the commands below. These commands assume that the computer has MATLAB installed. If you run the standalone executable on a machine with only MCR, and no MATLAB installed, replace `$MATLABROOT/bin/...` with the path to the MCR.

Platform	Command

Mac	`setenv DYLD_LIBRARY_PATH "$ {DYLD_LIBRARY_PATH}:$MATLABROOT/bin/ maci64" (csh/tcsh)`
	
	`export DYLD_LIBRARY_PATH= $DYLD_LIBRARY_PATH:$MATLABROOT/bin/ maci64 (Bash)`
	
	For more information, see Append library path to "DYLD_LIBRARY_PATH" in MAC.

Linux	`setenv LD_LIBRARY_PATH $ {LD_LIBRARY_PATH}:$MATLABROOT/bin/ glnxa64 (csh/tcsh)`
	
	`export LD_LIBRARY_PATH= $LD_LIBRARY_PATH:$MATLABROOT/bin/ glnxa64 (Bash)`

Windows	`set PATH=%PATH%;%MATLABROOT%\bin\win64`

See Also

More About

- “Understanding C Code Generation in DSP System Toolbox” on page 18-8
- “MATLAB Programming for Code Generation” (MATLAB Coder)
- “Relocate Code Generated from MATLAB Code to Another Development Environment” on page 18-19
- “Relocate Code Generated from a Simulink Model to Another Development Environment” on page 18-26

Use Generated Code to Accelerate an Application Deployed with MATLAB Compiler

This example shows how to use generated code to accelerate an application that you deploy with MATLAB® Compiler. The example accelerates an algorithm by using MATLAB® Coder™ to generate a MEX version of the algorithm. It uses MATLAB Compiler to deploy a standalone application that calls the MEX function. The deployed application uses the MATLAB® Runtime which enables royalty-free deployment to someone who does not have MATLAB.

This workflow is useful when:

- You want to deploy an application to a platform that the MATLAB Runtime supports.
- The application includes a computationally intensive algorithm that is suitable for code generation.
- The generated MEX for the algorithm is faster than the original MATLAB algorithm.
- You do not need to deploy readable C/C++ source code for the application.

The example application uses a DSP algorithm that requires the DSP System Toolbox™.

Create the MATLAB Application

For acceleration, it is a best practice to separate the computationally intensive algorithm from the code that calls it.

In this example, `myRLSFilterSystemIDSim` implements the algorithm. `myRLSFilterSystemIDApp` provides a user interface that calls `myRLSFilterSystemIDSim`.

`myRLSFilterSystemIDSim` simulates system identification by using recursive least-squares (RLS) adaptive filtering. It uses `dsp.VariableBandwidthFIRFilter` to model the unidentified system and `dsp.RLSFilter` to identify the FIR filter.

`myRLSFilterSystemIDApp` provides a user interface that you use to dynamically tune simulation parameters. It runs the simulation for a specified number of time steps or until you stop the simulation. It plots the results on scopes.

For details about this application, see “System Identification Using RLS Adaptive Filtering” in the DSP System Toolbox documentation.

In a writable folder, create `myRLSFilterSystemIDSim` and `myRLSFilterSystemIDApp`. Alternatively, to access these files, click **Open Script**.

myRLSFilterSystemIDSim

```
function [tfe,err,cutoffFreq,ff] = ...
    myRLSFilterSystemIDSim(tuningUIStruct)
% myRLSFilterSystemIDSim implements the algorithm used in
% myRLSFilterSystemIDApp.
% This function instantiates, initializes, and steps through the System
% objects used in the algorithm.
%
% You can tune the cutoff frequency of the desired system and the
% forgetting factor of the RLS filter through the GUI that appears when
```

```
% myRLSFilterSystemIDApp is executed.

%   Copyright 2013-2017 The MathWorks, Inc.

%#codegen

% Instantiate and initialize System objects. The objects are declared
% persistent so that they are not recreated every time the function is
% called inside the simulation loop.
persistent rlsFilt sine unknownSys transferFunctionEstimator
if isempty(rlsFilt)
    % FIR filter models the unidentified system
    unknownSys = dsp.VariableBandwidthFIRFilter('SampleRate',1e4,...
        'FilterOrder',30,...
        'CutoffFrequency',.48 * 1e4/2);
    % RLS filter is used to identify the FIR filter
    rlsFilt = dsp.RLSFilter('ForgettingFactor',.99,...
        'Length',28);
    % Sine wave used to generate input signal
    sine = dsp.SineWave('SamplesPerFrame',1024,...
        'SampleRate',1e4,...
        'Frequency',50);
    % Transfer function estimator used to estimate frequency responses of
    % FIR and RLS filters.
    transferFunctionEstimator = dsp.TransferFunctionEstimator(...
        'FrequencyRange','centered',...
        'SpectralAverages',10,...
        'FFTLengthSource','Property',...
        'FFTLength',1024,...
        'Window','Kaiser');
end

if tuningUIStruct.Reset
    % reset System objects
    reset(rlsFilt);
    reset(unknownSys);
    reset(transferFunctionEstimator);
    reset(sine);
end

% Tune FIR cutoff frequency and RLS forgetting factor
if  tuningUIStruct.ValuesChanged
    param = tuningUIStruct.TuningValues;
    unknownSys.CutoffFrequency  = param(1);
    rlsFilt.ForgettingFactor = param(2);
end

% Generate input signal - sine wave plus Gaussian noise
inputSignal = sine() +  .1 * randn(1024,1);

% Filter input though FIR filter
desiredOutput = unknownSys(inputSignal);

% Pass original and desired signals through the RLS Filter
[rlsOutput , err] = rlsFilt(inputSignal,desiredOutput);

% Prepare system input and output for transfer function estimator
inChans = repmat(inputSignal,1,2);
```

```
outChans = [desiredOutput,rlsOutput];

% Estimate transfer function
tfe = transferFunctionEstimator(inChans,outChans);

% Save the cutoff frequency and forgetting factor
cutoffFreq = unknownSys.CutoffFrequency;
ff = rlsFilt.ForgettingFactor;

end
```

myRLSFilterSystemIDApp

```
function scopeHandles = myRLSFilterSystemIDApp(numTSteps)
% myRLSFilterSystemIDApp initialize and execute RLS Filter
% system identification example. Then, display results using
% scopes. The function returns the handles to the scope and UI objects.
%
% Input:
%   numTSteps - number of time steps
% Outputs:
%   scopeHandles    - Handle to the visualization scopes

% Copyright 2013-2017 The MathWorks, Inc.


if nargin == 0
    numTSteps = Inf; % Run until user stops simulation.
end

% Create scopes
tfescope = dsp.ArrayPlot('PlotType','Line',...
    'Position',[8 696 520 420],...
    'YLimits',[-80 30],...
    'SampleIncrement',1e4/1024,...
    'YLabel','Amplitude (dB)',...
    'XLabel','Frequency (Hz)',...
    'Title','Desired and Estimated Transfer Functions',...
    'ShowLegend',true,...
    'XOffset',-5000);

msescope = dsp.TimeScope('SampleRate',1e4,'TimeSpan',.01,...
    'Position',[8 184 520 420],...
    'YLimits',[-300 10],'ShowGrid',true,...
    'YLabel','Mean-Square Error (dB)',...
    'Title','RLSFilter Learning Curve');

screen = get(0,'ScreenSize');
outerSize = min((screen(4)-40)/2, 512);
tfescope.Position = [8, screen(4)-outerSize+8, outerSize+8,...
    outerSize-92];
msescope.Position = [8, screen(4)-2*outerSize+8, outerSize+8, ...
    outerSize-92];

% Create UI to tune FIR filter cutoff frequency and RLS filter
%  forgetting factor
```

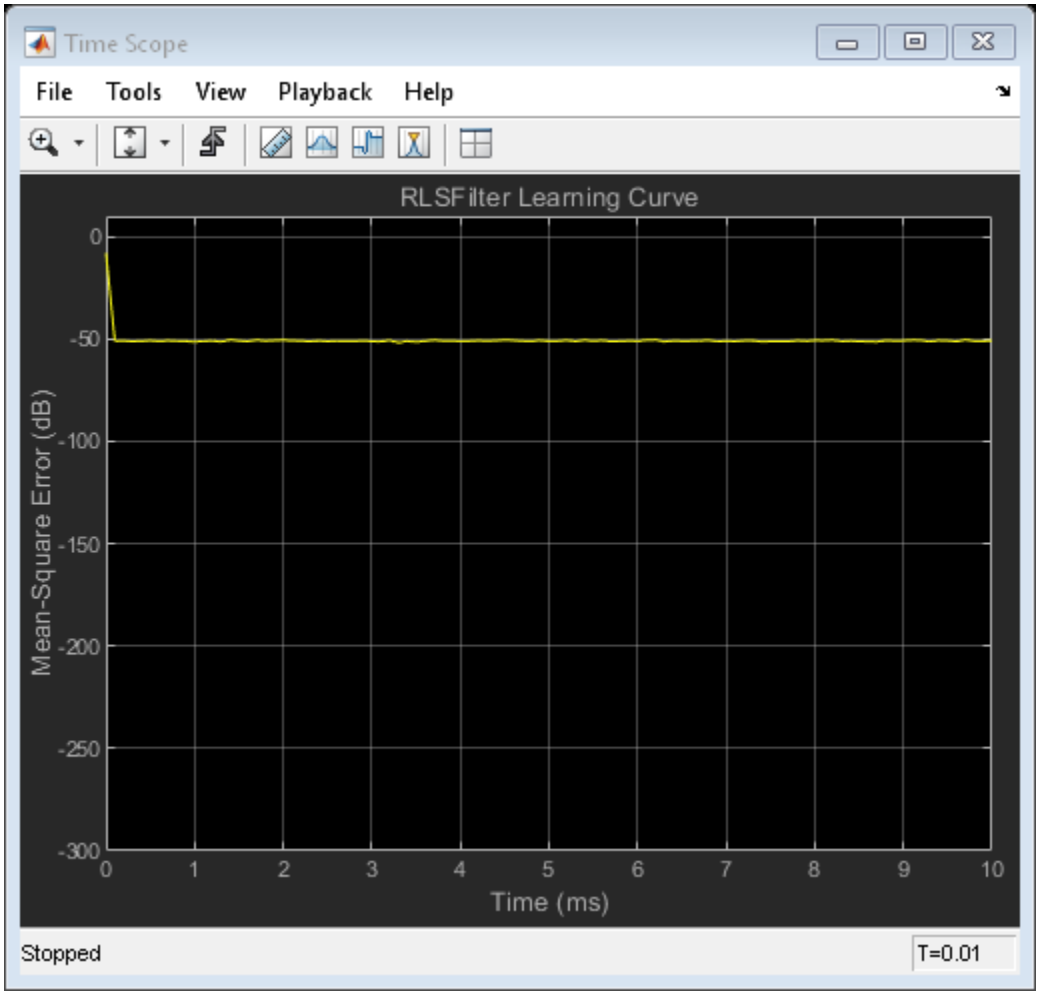
```
Fs = 1e4;
param = struct([]);
param(1).Name = 'Cutoff Frequency (Hz)';
param(1).InitialValue = 0.48 * Fs/2;
param(1).Limits = Fs/2 * [1e-5, .9999];
param(2).Name = 'RLS Forgetting Factor';
param(2).InitialValue = 0.99;
param(2).Limits = [.3, 1];
hUI = HelperCreateParamTuningUI(param, 'RLS FIR Demo');
set(hUI,'Position',[outerSize+32, screen(4)-2*outerSize+8, ...
    outerSize+8, outerSize-92]);

% Execute algorithm
while(numTSteps>=0)

    S = HelperUnpackUIData(hUI);

    drawnow limitrate;   % needed to process UI callbacks

    [tfe,err] = myRLSFilterSystemIDSim(S);

    if S.Stop     % If "Stop Simulation" button is pressed
        break;
    end
    if S.Pause
        continue;
    end

    % Plot transfer functions
    tfescope(20*log10(abs(tfe)));
    % Plot learning curve
    msescope(10*log10(sum(err.^2)));
    numTSteps = numTSteps - 1;
end

if ishghandle(hUI)  % If parameter tuning UI is open, then close it.
    delete(hUI);
    drawnow;
    clear hUI
end

scopeHandles.tfescope = tfescope;
scopeHandles.msescope = msescope;
end
```

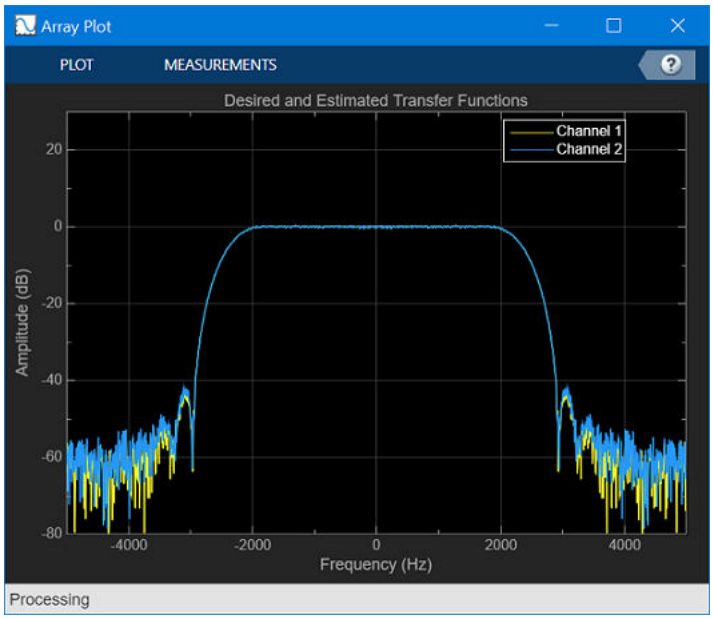
Test the MATLAB Application

Run the system identification application for 100 time steps. The application runs the simulation for 100 time steps or until you click **Stop Simulation**. It plots the results on scopes.

```
scope1 = myRLSFilterSystemIDApp(100);
release(scope1.tfescope);
release(scope1.msescope);
```

```
Warning: MATLAB Connector is not running.
Warning: MATLAB Connector is not running.
```


Time Scope
File
Tools
View
Playback
Help
RLSFilter Learning Curve
Mean-Square Error (dB)
0
-50
-100
-150
-200
-250
-300
0
1
2
3
4
5
6
7
8
9
10
Time (ms)
Stopped
T=0.01


Array Plot
PLOT
MEASUREMENTS
Desired and Estimated Transfer Functions
Channel 1
Channel 2
Amplitude (dB)
20
0
-20
-40
-60
-80
-4000
-2000
0
2000
4000
Frequency (Hz)
Processing

Prepare Algorithm for Acceleration

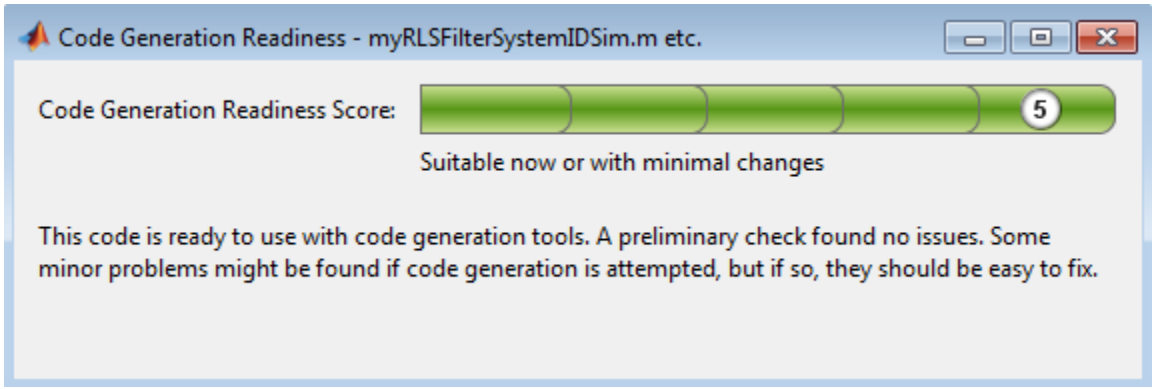
When you use MATLAB Coder to accelerate a MATLAB algorithm, the code must be suitable for code generation.

1. Make sure that `myRLSFilterSystemIDSim.m` includes the `%#codegen` directive after the function signature.

This directive indicates that you intend to generate code for the function. In the MATLAB Editor, it enables the code analyzer to detect code generation issues.

2. Screen the algorithm for unsupported functions or constructs.

```
coder.screener('myRLSFilterSystemIDSim');
```



The code generation readiness tool does not find code generation issues in this algorithm.

Accelerate the Algorithm

To accelerate the algorithm, this example use the MATLAB Coder `codegen` command. Alternatively, you can use the MATLAB Coder app. For code generation, you must specify the type, size, and complexity of the input arguments. The function `myRLSFilterSystemIDSim` takes a structure that stores tuning information. Define an example tuning structure and pass it to `codegen` by using the `-args` option.

```
ParamStruct.TuningValues = [2400 0.99];
ParamStruct.ValuesChanged = false;
ParamStruct.Reset = false;
ParamStruct.Pause = false;
ParamStruct.Stop  = false;
codegen myRLSFilterSystemIDSim -args {ParamStruct};
```

`codegen` creates the MEX function `myRLSFilterSystemIDSim_mex` in the current folder.

Compare MEX Function and MATLAB Function Performance

1. Time 100 executions of `myRLSFilterSystemIDSim`.

```
clear myRLSFilterSystemIDSim
disp('Running the MATLAB function ...')
tic
nTimeSteps = 100;
for ind = 1:nTimeSteps
```

```
     myRLSFilterSystemIDSim(ParamStruct);
end
tMATLAB = toc;

Running the MATLAB function ...
```

2. Time 100 executions of `myRLSFilterSystemIDSim_mex`.

```
clear myRLSFilterSystemIDSim
disp('Running the MEX function ...')
tic
for ind = 1:nTimeSteps
    myRLSFilterSystemIDSim_mex(ParamStruct);
end
tMEX = toc;

disp('RESULTS:')
disp(['Time for original MATLAB function: ', num2str(tMATLAB),...
     ' seconds']);
disp(['Time for MEX function: ', num2str(tMEX), ' seconds']);
disp(['The MEX function is ', num2str(tMATLAB/tMEX),...
    ' times faster than the original MATLAB function.']);

Running the MEX function ...
RESULTS:
Time for original MATLAB function: 4.4535 seconds
Time for MEX function: 0.57979 seconds
The MEX function is 7.6813 times faster than the original MATLAB function.
```

Optimize the MEX code

You can sometimes generate faster MEX by using a different C/C++ compiler or by using certain options or optimizations. See “Accelerate MATLAB Algorithms” (MATLAB Coder).

For this example, the MEX is sufficiently fast without further optimization.

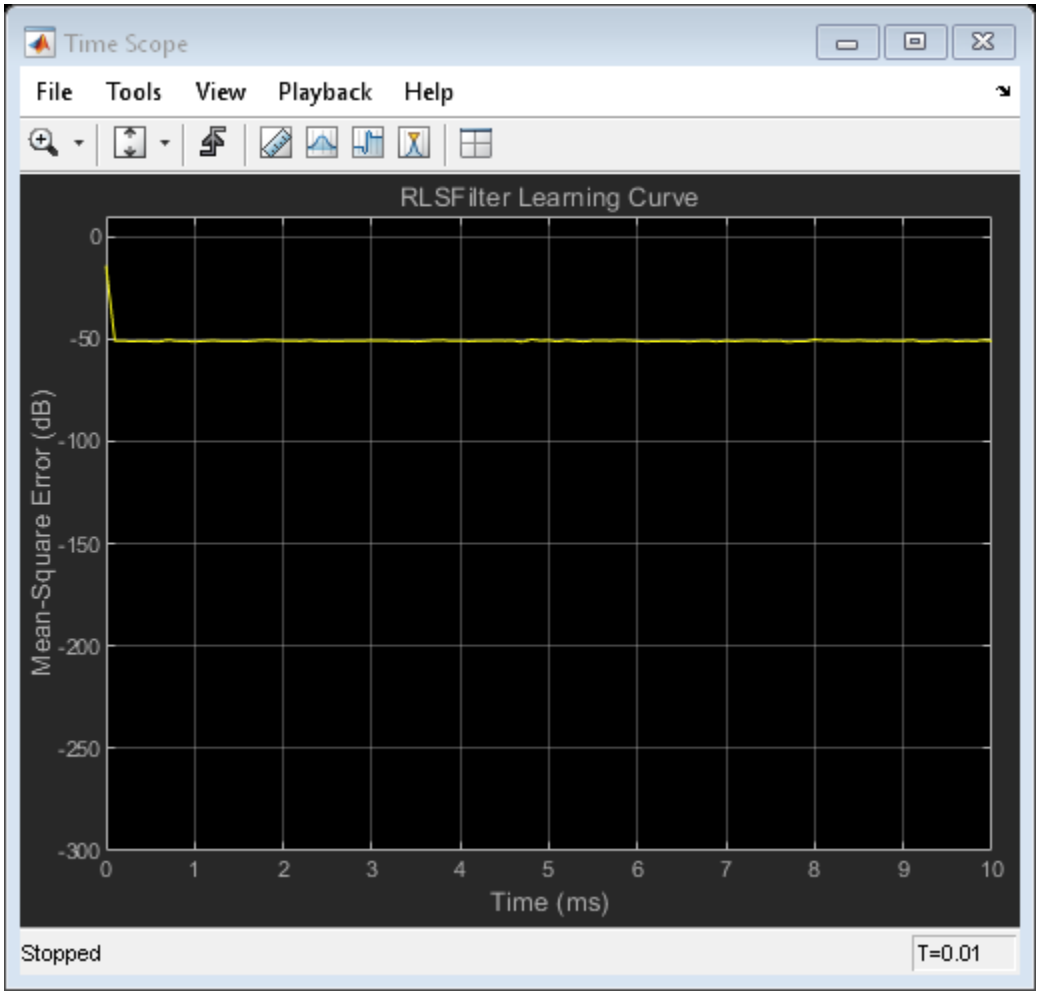
Modify the Application to Call the MEX Function

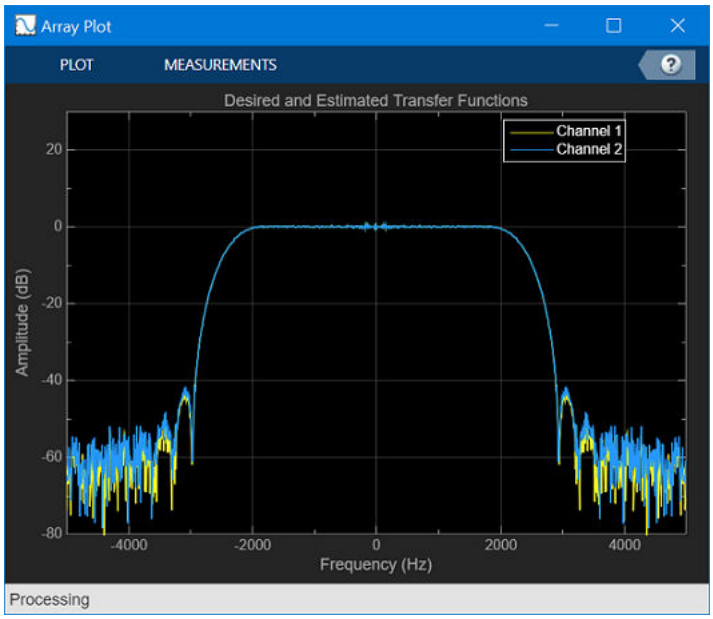
Modify `myRLSFilterSystemIDApp` so that it calls `myRLSFilterSystemIDSim_mex` instead of `myRLSFilterSystemIDSim`.

Save the modified function in `myRLSFilterSystemIDApp_acc.m`.

Test the Application with the Accelerated Algorithm

```
clear myRLSFilterSystemIDSim_mex;
scope2 = myRLSFilterSystemIDApp_acc(100);
release(scope2.tfescope);
release(scope2.msescope);
```


Time Scope
File
Tools
View
Playback
Help
RLSFilter Learning Curve
Mean-Square Error (dB)
0
-50
-100
-150
-200
-250
-300
0
1
2
3
4
5
6
7
8
9
10
Time (ms)
Stopped
T=0.01


Array Plot
PLOT
MEASUREMENTS
Desired and Estimated Transfer Functions
Channel 1
Channel 2
Amplitude (dB)
20
0
-20
-40
-60
-80
-4000
-2000
0
2000
4000
Frequency (Hz)
Processing

The behavior of the application that calls the MEX function is the same as the behavior of the application that calls the original MATLAB function. However, the plots update more quickly because the simulation is faster.

Create the Standalone Application

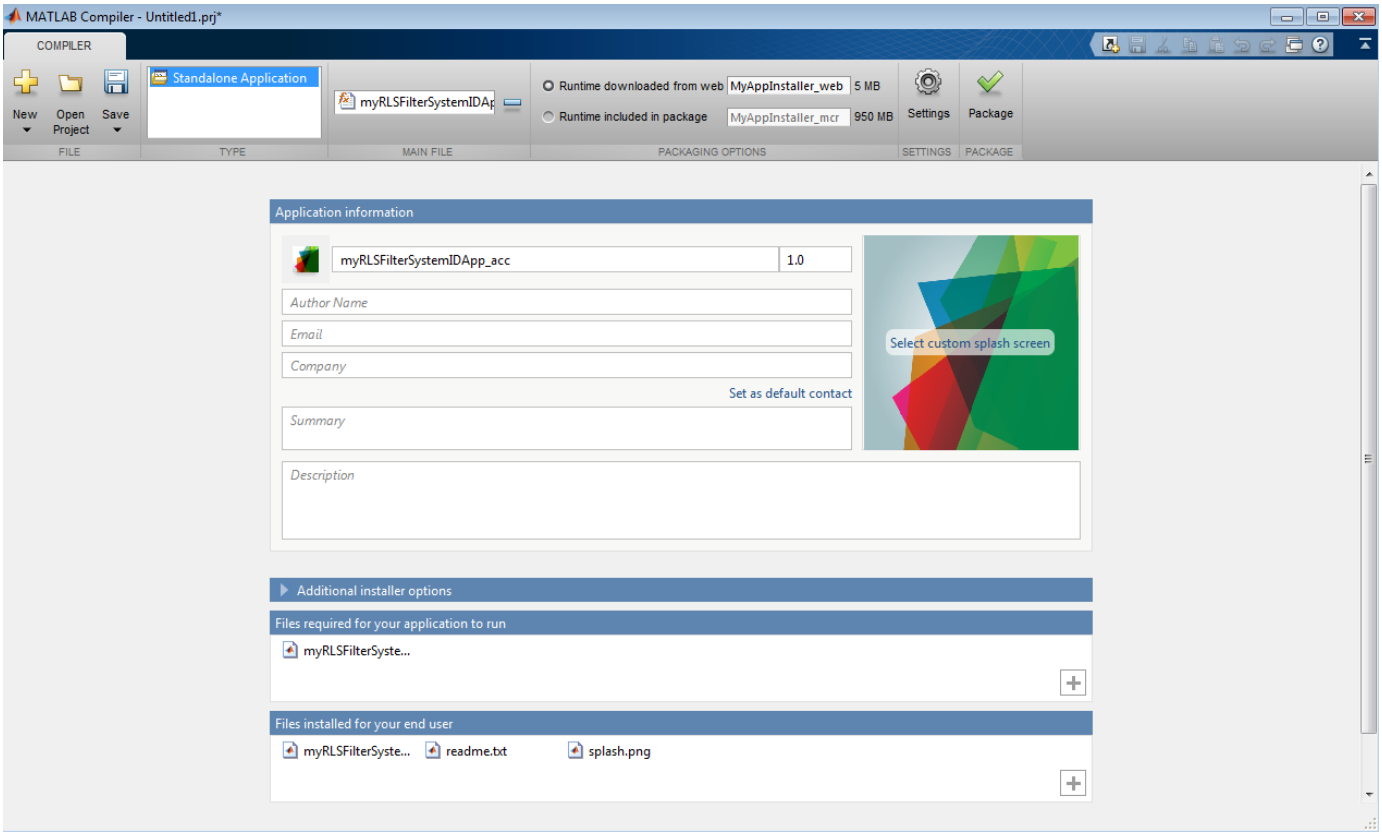
1. To open the Application Compiler App, on the **Apps** tab, under **Application Deployment**, click the app icon.

2. Specify that the main file is `myRLSFilterSystemIDApp_acc`.

The app determines the required files for this application. The app can find the MATLAB files and MEX-files that an application uses. You must add other types of files, such as MAT-files or images, as required files.

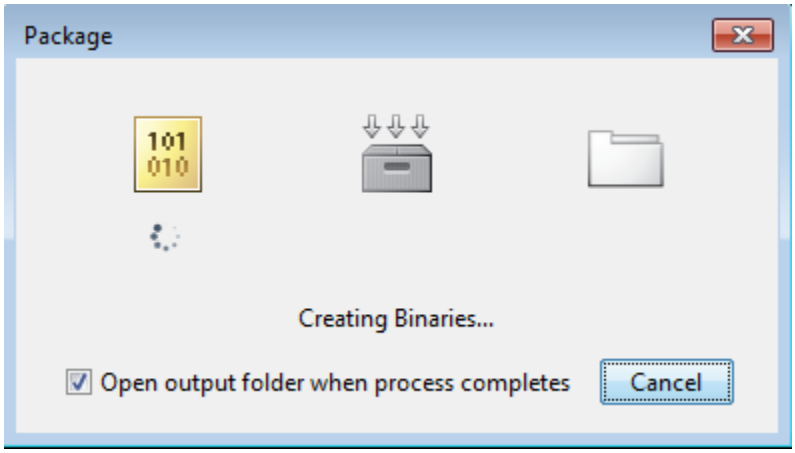
3. In the **Packaging Options** section of the toolstrip, make sure that the **Runtime downloaded from web** check box is selected.

This option creates an application installer that downloads and installs the MATLAB Runtime with the deployed MATLAB application.



4. Click **Package** and save the project.

5. In the Package window, make sure that the **Open output folder when process completes** check box is selected.



When the packaging is complete, the output folder opens.

Install the Application

1. Open the `for_redistribution` folder.

2. Run `MyAppInstaller_web`.

3. If you connect to the internet by using a proxy server, enter the server settings.

4. Advance through the pages of the installation wizard.

- On the Installation Options page, use the default installation folder.
- On the Required Software page, use the default installation folder.
- On the License agreement page, read the license agreement and accept the license.
- On the Confirmation page, click **Install**.

If the MATLAB Runtime is not already installed, the installer installs it.

5. Click **Finish**.

Run the Application

1. Open a terminal window.

2. Navigate to the folder where the application is installed.

- For Windows®, navigate to `C:\Program Files\myRLSFilterSystemIDApp_acc`.
- For macOS, navigate to `/Applications/myRLSFilterSystemIDApp_acc`.
- For Linux, navigate to `/usr/myRLSFilterSystemIDApp_acc`.

3. Run the application by using the appropriate command for your platform.

- For Windows, use `application\myRLSFilterSystemIDApp_acc`.
- For macOS, use `myRLSFilterSystemIDApp_acc.app/Contents/MacOS/myRLSFilterSystemIDApp_acc`.
- For Linux, use `/myRLSFilterSystemIDApp_acc`.

Starting the application takes approximately the same amount of time as starting MATLAB.

See Also

More About

- “System Identification Using RLS Adaptive Filtering”
- “Workflow for Accelerating MATLAB Algorithms” (MATLAB Coder)
- “Accelerate MATLAB Algorithms” (MATLAB Coder)
- “Create Standalone Application from MATLAB” (MATLAB Compiler)
- “About the MATLAB Runtime” (MATLAB Compiler)

External Websites

- MATLAB Compiler Support for MATLAB and toolboxes.

How Is dspunfold Different from parfor?

In this section...
“DSP Algorithms Involve States” on page 18-43
“dspunfold Introduces Latency” on page 18-43
“parfor Requires Significant Restructuring in Code” on page 18-43
“parfor Used with dspunfold” on page 18-43

The `dspunfold` and `parfor` functions accelerate MATLAB algorithms through parallelization. Each function has its own advantages and disadvantages.

When you use `parfor` inside the entry-point MATLAB function, and call `codegen` on this function, the generated MEX file is multithreaded. For more information, see “Algorithm Acceleration Using Parallel for-Loops (parfor)” (MATLAB Coder). However, `parfor` is not ideal for DSP algorithms. The reason being that DSP algorithms involve states.

DSP Algorithms Involve States

Most algorithms in DSP System Toolbox contain states and stream data. States in MATLAB are modeled using persistent variables. Because `parfor` does not support persistent variables, you cannot model states using `parfor` loops. See “Global or Persistent Declarations in parfor-Loop” (MATLAB Coder). In addition, you cannot have any data dependency across `parfor` loops. Hence, you cannot maintain state information across these loops. See “When Not to Use parfor-Loops” (MATLAB Coder). `dspunfold` overcomes these limitations by supporting persistent variables.

dspunfold Introduces Latency

If your application does not tolerate latency, use `parfor` instead. `parfor` does not introduce latency. Latency is the number of input frames processed before generating the first output frame.

parfor Requires Significant Restructuring in Code

`parfor` requires you to restructure your algorithm to have a loop-like structure that is iteration independent. Due to the semantic limitations of `parfor`, replacing a `for`-loop with a `parfor`-loop often requires significant code refactoring. `dspunfold` does not require you to restructure your code.

parfor Used with dspunfold

When you call `dspunfold` on an entry-point MATLAB function that contains `parfor`, `parfor` multi-threading is disabled. `dspunfold` calls `codegen` with the `–O` option set to `disable:openmp`. With this option set, `parfor` loops are treated as `for`- loops. The multi-threading behavior of the generated MEX file is due entirely to `dspunfold`.

See Also

Functions
`dspunfold` | `parfor`

More About

- “Generate Code with Parallel for-Loops (parfor)” (MATLAB Coder)
- “Algorithm Acceleration Using Parallel for-Loops (parfor)” (MATLAB Coder)
- “MATLAB Algorithm Acceleration” (MATLAB Coder)

Workflow for Generating a Multithreaded MEX File using dspunfold

1. Run the entry-point MATLAB function with the inputs that you want to test. Make sure that the function has no runtime errors. Call `codegen` on the function and make sure that it generates a MEX file successfully.
2. Generate the multithreaded MEX file using `dspunfold`. Specify a state length using the `-s` option. The state length must be at least the same length as the algorithm in the MATLAB function. By default, `-s` is set to `0`, indicating that the algorithm is stateless.
3. Run the generated analyzer function. Use the `pass` flag to verify that the output results of the multithreaded MEX file and the single-threaded MEX file match. Also, check if the speedup and latency displayed by the analyzer function are satisfactory.
4. If the output results do not match, increase the state length and generate the multithreaded MEX file again. Alternatively, use the automatic state length detection (specified using `-s auto`) to determine the minimum state length that matches the outputs.
5. If the output results match but the speedup and latency are not satisfactory, increase the repetition factor using `-r` or increase the number of threads using `-t`. In addition, you can adjust the state length. Adjust the `dspunfold` options and generate new multithreaded MEX files until you are satisfied with the results..

For best practices for generating the multithreaded MEX file using `dspunfold`, see the 'Tips' section of `dspunfold`.

Workflow Example

Run the Entry Point MATLAB Function

Create the entry-point MATLAB function.

```
function [y,mse] = AdaptiveFilter(x,noise)

persistent rlsf1 ffilt noise_var
if isempty (rlsf1)
    rlsf1 = dsp.RLSFilter(32, 'ForgettingFactor', 0.98);
    ffilt = dsp.FIRFilter('Numerator',fir1(32, .25)); % Unknown System
    noise_var = 1e-4;
end

d = ffilt(x) + noise_var * noise; % desired signal
[y,e] = rlsf1(x, d);

mse = 10*log10(sum(e.^2));
end
```

The function models an RLS filter that filters the input signal `x`, using `d` as the desired signal. The function returns the filtered output in `y` and the filter error in `e`.

Run `AdaptiveFilter` with the inputs that you want to test. Verify that the function runs without errors.

```
AdaptiveFilter(randn(1000,1), randn(1000,1));
```

Call `codegen` on `AdaptiveFilter` and generate a MEX file.

```
codegen AdaptiveFilter -args {randn(1000,1), randn(1000,1)}
```

Generate a Multithreaded MEX File Using `dspunfold`

Set the state length to 32 samples and the repetition factor to 1. Provide a state length that is greater than or equal to the algorithm in the MATLAB function. When at least one entry of `frameinputs` is set to `true`, state length is considered in samples.

```
dspunfold AdaptiveFilter -args {randn(1000,1), randn(1000,1)} -s 32 -f true
```

```
Analyzing input MATLAB function AdaptiveFilter
Creating single-threaded MEX file AdaptiveFilter_st.mexw64
Creating multi-threaded MEX file AdaptiveFilter_mt.mexw64
Creating analyzer file AdaptiveFilter_analyzer
```

Run the Generated Analyzer Function

The analyzer considers the actual values of the input. To increase the analyzer effectiveness, provide at least two different frames along the first dimension of the inputs.

```
AdaptiveFilter_analyzer(randn(1000*4,1),randn(1000*4,1))
```

```
Analyzing multi-threaded MEX file AdaptiveFilter_mt.mexw64  ...
Latency = 8 frames
Speedup = 3.5x
Warning: The output results of the multi-threaded MEX file AdaptiveFilter_mt.mexw64 do not match
the output results of the single-threaded MEX file AdaptiveFilter_st.mexw64. Check that you
provided the correct state length value to the dspunfold function when you generated the
multi-threaded MEX file AdaptiveFilter_mt.mexw64. For best practices and possible solutions to
this problem, see the 'Tips' section in the dspunfold function reference page.
> In coder.internal.warning (line 8)
  In AdaptiveFilter_analyzer

ans =

    Latency: 8
    Speedup: 3.4686
       Pass: 0
```

Increase the State Length

The analyzer did not pass the verification. The warning message displayed indicates that a wrong state length value is provided to the `dspunfold` function. Increase the state length to `1000` samples and repeat the process from the previous section.

```
dspunfold AdaptiveFilter -args {randn(1000,1),randn(1000,1)} -s 1000 -f true
```

```
Analyzing input MATLAB function AdaptiveFilter
Creating single-threaded MEX file AdaptiveFilter_st.mexw64
Creating multi-threaded MEX file AdaptiveFilter_mt.mexw64
Creating analyzer file AdaptiveFilter_analyzer
```

Run the generated analyzer.

```
AdaptiveFilter_analyzer(randn(1000*4,1),randn(1000*4,1))
```

```
Analyzing multi-threaded MEX file AdaptiveFilter_mt.mexw64  ...
Latency = 8 frames
Speedup = 1.8x

ans =

    Latency: 8
    Speedup: 1.7778
       Pass: 1
```

The analyzer passed verification. It is recommended that you provide different numerics to the analyzer function and make sure that the analyzer function passes.

Improve Speedup and Adjust Latency

If you want to increase speedup and your system can afford a larger latency, increase the repetition factor to 2.

```
dspunfold AdaptiveFilter -args {randn(1000,1),randn(1000,1)} -s 1000 -r 2 -f true
```

```
Analyzing input MATLAB function AdaptiveFilter
Creating single-threaded MEX file AdaptiveFilter_st.mexw64
Creating multi-threaded MEX file AdaptiveFilter_mt.mexw64
Creating analyzer file AdaptiveFilter_analyzer
```

Run the analyzer.

```
 AdaptiveFilter_analyzer(randn(1000*4,1), randn(1000*4,1))
```

```
Analyzing multi-threaded MEX file AdaptiveFilter_mt.mexw64  ...
Latency = 16 frames
Speedup = 2.4x

ans =

    Latency: 16
    Speedup: 2.3674
       Pass: 1
```

Repeat the process until you achieve satisfactory speedup and latency.

Use Automatic State Length Detection

Choose a state length that is greater than or equal to the state length of your algorithm. If it is not easy to determine the state length for your algorithm analytically, use the automatic state length detection tool. Invoke automatic state length detection by setting `-s` to `auto`. The tool detects the minimum state length with which the analyzer passes the verification.

```
dspunfold AdaptiveFilter -args {randn(1000,1),randn(1000,1)} -s auto -f true
```

```
Analyzing input MATLAB function AdaptiveFilter
Creating single-threaded MEX file AdaptiveFilter_st.mexw64
Searching for minimal state length (this might take a while)
Checking stateless ... Insufficient
Checking 1000 ... Sufficient
Checking 500 ... Insufficient
Checking 750 ... Insufficient
Checking 875 ... Sufficient
Checking 812 ... Insufficient
Checking 843 ... Sufficient
Checking 827 ... Insufficient
Checking 835 ... Insufficient
Checking 839 ... Sufficient
Checking 837 ... Sufficient
Checking 836 ... Sufficient
Minimal state length is 836
Creating multi-threaded MEX file AdaptiveFilter_mt.mexw64
Creating analyzer file AdaptiveFilter_analyzer
```

Minimal state length is 836 samples.

Run the generated analyzer.

```
AdaptiveFilter_analyzer(randn(1000*4,1), randn(1000*4,1))
```

```
Analyzing multi-threaded MEX file AdaptiveFilter_mt.mexw64  ...
Latency = 8 frames
Speedup = 1.9x

ans =

    Latency: 8
    Speedup: 1.9137
       Pass: 1
```

The analyzer passed the verification.

See Also

Functions
`dspunfold`

More About

- “Why Does the Analyzer Choose the Wrong State Length?” on page 18-49
- “Why Does the Analyzer Choose a Zero State Length?” on page 18-51

Why Does the Analyzer Choose the Wrong State Length?

In this section...
“Reason for Verification Failure” on page 18-50 “Recommendation” on page 18-50

If the state length of the algorithm depends on the inputs to the algorithm, make sure that you use inputs that choose the same state length when generating the MEX file and running the analyzer. Otherwise, the analyzer fails the verification.

The algorithm in the function `FIR_Mean` has no states when `mean(input) > 0`, and has states otherwise.

```
function [ Output ] = FIR_Mean( input )

persistent Filter
if isempty(Filter)
   Filter = dsp.FIRFilter('Numerator', fir1(12,0.4));
end


 if (mean(input) > 0)
     % stateless
     Output = mean(input);
 else
     % this path contains states
    yFilt = Filter(input);
   Output = mean(yFilt);
 end
 end
```

When you invoke the automatic state length detection on this function, the analyzer detects a state length of 14 samples.

```
dspunfold FIR_Mean -args {randn(10,1)} -s auto -f true
```

```
Analyzing input MATLAB function FIR_Mean
Creating single-threaded MEX file FIR_Mean_st.mexw64
Searching for minimal state length (this might take a while)
Checking stateless ... Insufficient
Checking 10 ... Insufficient
Checking Infinite ... Sufficient
Checking 20 ... Sufficient
Checking 15 ... Sufficient
Checking 12 ... Insufficient
Checking 13 ... Insufficient
Checking 14 ... Sufficient
Minimal state length is 14
Creating multi-threaded MEX file FIR_Mean_mt.mexw64
Creating analyzer file FIR_Mean_analyzer
```

Run the analyzer function. Use an input with four different frames. Check if the output results match.

```
FIR_Mean_analyzer(randn(10*4,1))
```

```
Analyzing multi-threaded MEX file FIR_Mean_mt.mexw64  ...
Latency = 8 frames
```

```
Speedup = 0.5x
Warning: The output results of the multi-threaded MEX file FIR_Mean_mt.mexw64 do not match
the output results of the single-threaded MEX file FIR_Mean_st.mexw64. Check that you
provided the correct state length value to the dspunfold function when you generated the
multi-threaded MEX file FIR_Mean_mt.mexw64. For best practices and possible solutions to
this problem, see the 'Tips' section in the dspunfold function reference page.
> In coder.internal.warning (line 8)
  In FIR_Mean_analyzer

ans =

    Latency: 8
    Speedup: 0.5040
       Pass: 0
```

`Pass = 0`, and the function throws a warning message indicating a possible reason for the verification failure.

Reason for Verification Failure

The state length of the algorithm depends on the input. When `mean(input) > 0`, the algorithm is stateless. Otherwise, the algorithm contains states. When generating the MEX file, the input arguments choose the code path with states. When the analyzer is called, the multi-frame input chooses the code path without states. Hence, the state length is different in both the cases leading to the verification failure.

Recommendation

The recommendation is to use inputs which choose the same state length when generating the MEX file and running the analyzer.

For best practices, see the 'Tips' section of `dspunfold`.

See Also

More About

- “Workflow for Generating a Multithreaded MEX File using dspunfold” on page 18-45
- “Why Does the Analyzer Choose a Zero State Length?” on page 18-51

Why Does the Analyzer Choose a Zero State Length?

When the output of the algorithm does not change for any input given to the algorithm, the analyzer considers the algorithm stateless, even if it contains states. Make sure the inputs to the algorithm have an immediate effect on the output of the algorithm.

The function `Input_Output` uses an FIR filter that contains states.

```
function [output] = Input_Output(input)

persistent Filter
if isempty(Filter)
   Filter = dsp.FIRFilter('Numerator', (1:12));
end

y = Filter(input);

output = any(y(:)>0);

end
```

When you call automatic state length detection on this function, the analyzer detects a minimal state length of `0`.

```
dspunfold Input_Output -args {randn(10,1)} -s auto -f true

Analyzing input MATLAB function Input_Output
Creating single-threaded MEX file Input_Output_st.mexw64
Searching for minimal state length (this might take a while)
Checking stateless ... Sufficient
Minimal state length is 0
Creating multi-threaded MEX file Input_Output_mt.mexw64
Creating analyzer file Input_Output_analyzer
```

The analyzer detects a zero state length because the output of the function is the same irrespective of the value of the input. When the analyzer tests the algorithm with zero state length, the outputs of the multithreaded MEX and single-threaded MEX match. Therefore, the analyzer considers the algorithm stateless and sets the minimal state length to zero.

Recommendation

To prevent the analyzer from choosing the wrong state length, rewrite your algorithm so that inputs have an immediate effect on the output. Also, choose inputs which stress the code path with maximal state length.

For best practices, see the 'Tips' section of `dspunfold`.

See Also

More About

- “Workflow for Generating a Multithreaded MEX File using dspunfold” on page 18-45
- “Why Does the Analyzer Choose the Wrong State Length?” on page 18-49

Array Plot with Android Devices

This example shows how to create an Android™ app to plot vector or array data on an Android device using the Array Plot block of DSP System Toolbox™ through a Simulink® model. To implement this workflow, you must install the Simulink Support Package for Android Devices.

Introduction

Android devices provide a user interface to visualize signals or display data on device screen. By using Array Plot block, you can display signals generated during simulation in real-time.

By displaying the Array Plot on an Android device screen, you can:

- Visualize vector or array data in real-time directly on your Android device screen.
- View signals without a connection to your development computer.
- Customize the Array Plot style to suit your app.

This example provides two Simulink models:

- **dspstreamingwelch**: This model displays the power spectrum estimate of a streaming time-domain input via Welch's method of averaged modified periodograms. This model runs on the development computer. For more information on this model, see “Streaming Power Spectrum Estimation Using Welch's Method”.
- **androidarrayplot**: Showcases how the power spectrum estimate is displayed on an Android device using an Array Plot.

Prerequisites

- Download and Install Simulink Support Package for Android Devices
- “Getting Started with Android™ Devices” (Simulink Support Package for Android Devices) example

Required Products

- DSP System Toolbox
- Simulink Support Package for Android Devices
- Simulink

Required Hardware

- Android device
- USB cable to connect the device to your development computer

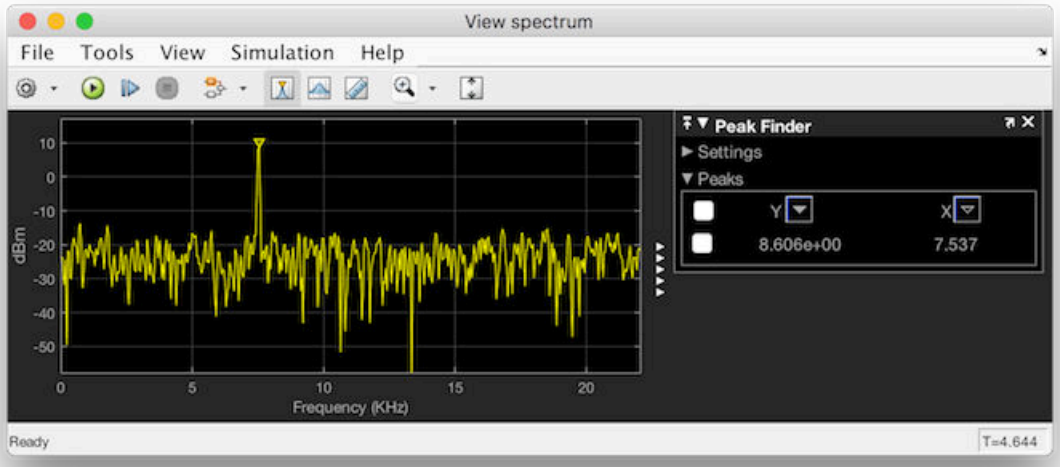
Task 1 - Display Signals on the Development Computer

The Spectrum Estimator block in dspstreamingwelch model estimates the spectrum of a noisy chirp signal, sampled at 44100 Hz. The Array Plot block displays the power spectrum estimate.

1. Open the dspstreamingwelch model on your development computer.

2. Double-click the **Array Plot** block to open the Array Plot window.

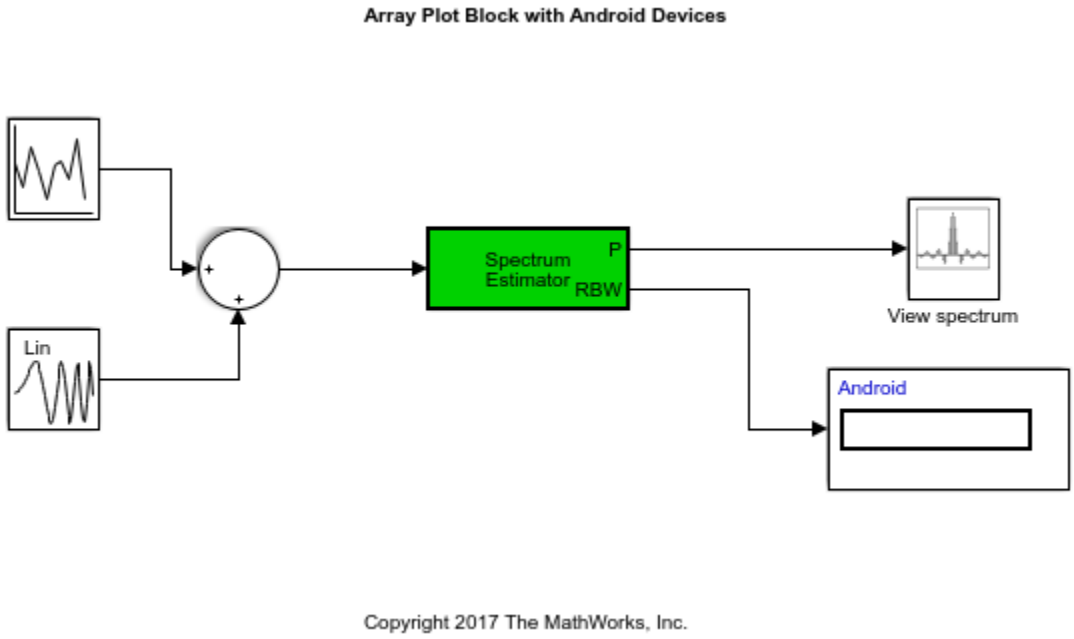
3. On the model editor, click the **Run** button to see the output of the streaming power spectrum estimate.



Task 2 - Display Signals on Your Android Device

Display the power spectrum estimate on your Android device

1. Open the androidarrayplot model.



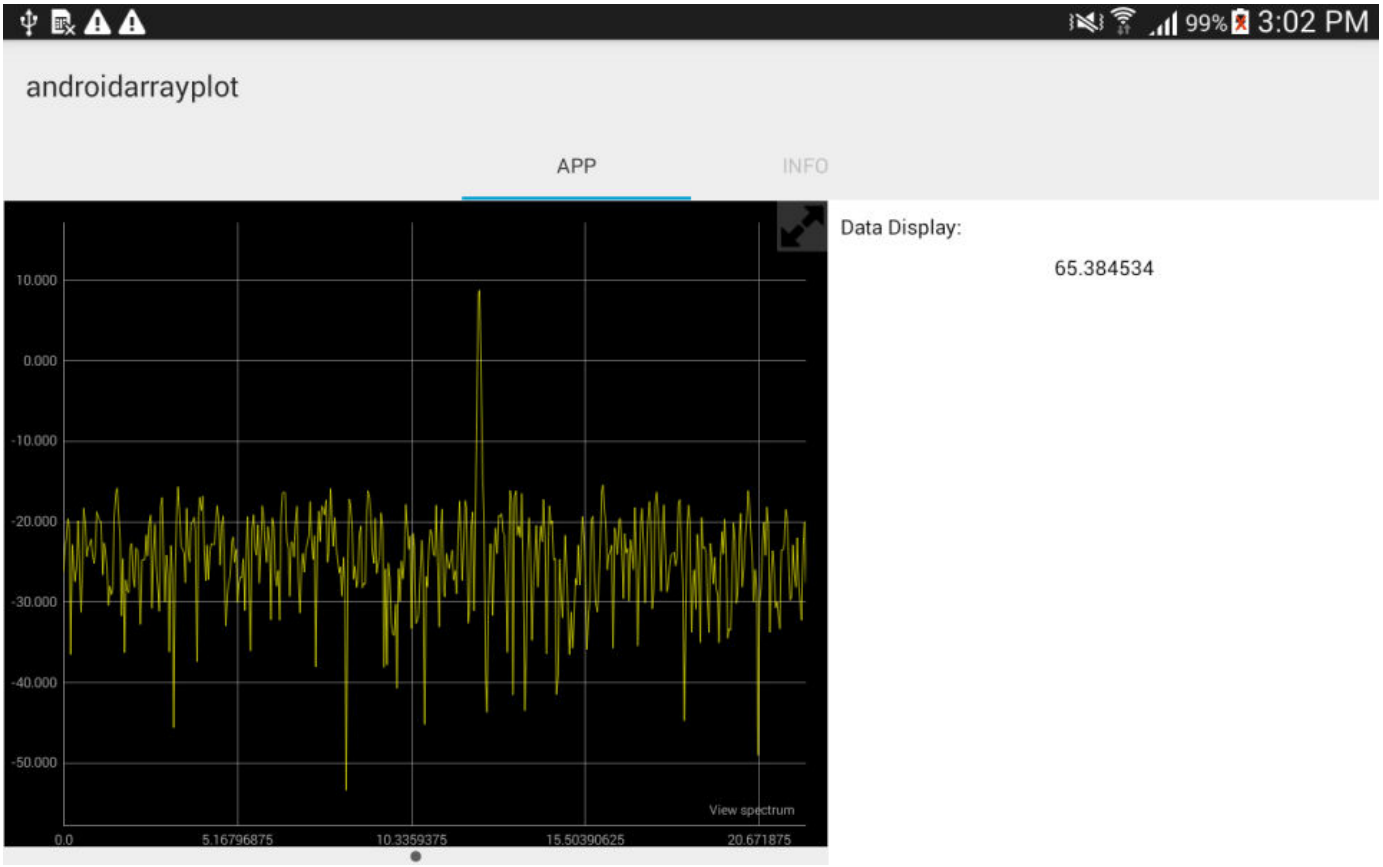
2. Note how the Simulink **Display** block has been replaced with a **Data Display** block from the Android Support Package. This allows you to view the resolution bandwidth (RBW) on your Android Device.

3. In the **Modeling** tab of the toolstrip, select **Model Settings**.

4. Select the **Hardware Implementation** pane and from the **Hardware board** parameter list, and confirm it is set to **Android Device**.

5. Click **Device options** and ensure that the device matches your device setting. Click **OK**.

6. In the **Hardware** tab of the toolstrip, click **Build, Deploy & Start** to build, download, and run the model on your device. In the current working directory, a folder named "androidarrayplot_ert_rtw" contains all the model's generated project files.



The app displays the power spectrum estimate on your device.

Task 3 - Customize the Array Plot Style on Your Android Device

Using the model from Task 2, configure the appearance and style of the **Array Plot** displayed on your Android device.

1. Open the androidarrayplot model.

2. Double-click the Array Plot block to open the Scope window.

3. In the Scope menu, click **View > Style** to open the Style dialog.

3. Set the **Figure color** to gray.

4. Modify the **Axes colors**. Set the **Axes background color** to white. Set the **Ticks, labels, and grid colors** to gray.

5. Set **Line width** to 6 and set the **Line color** to blue.

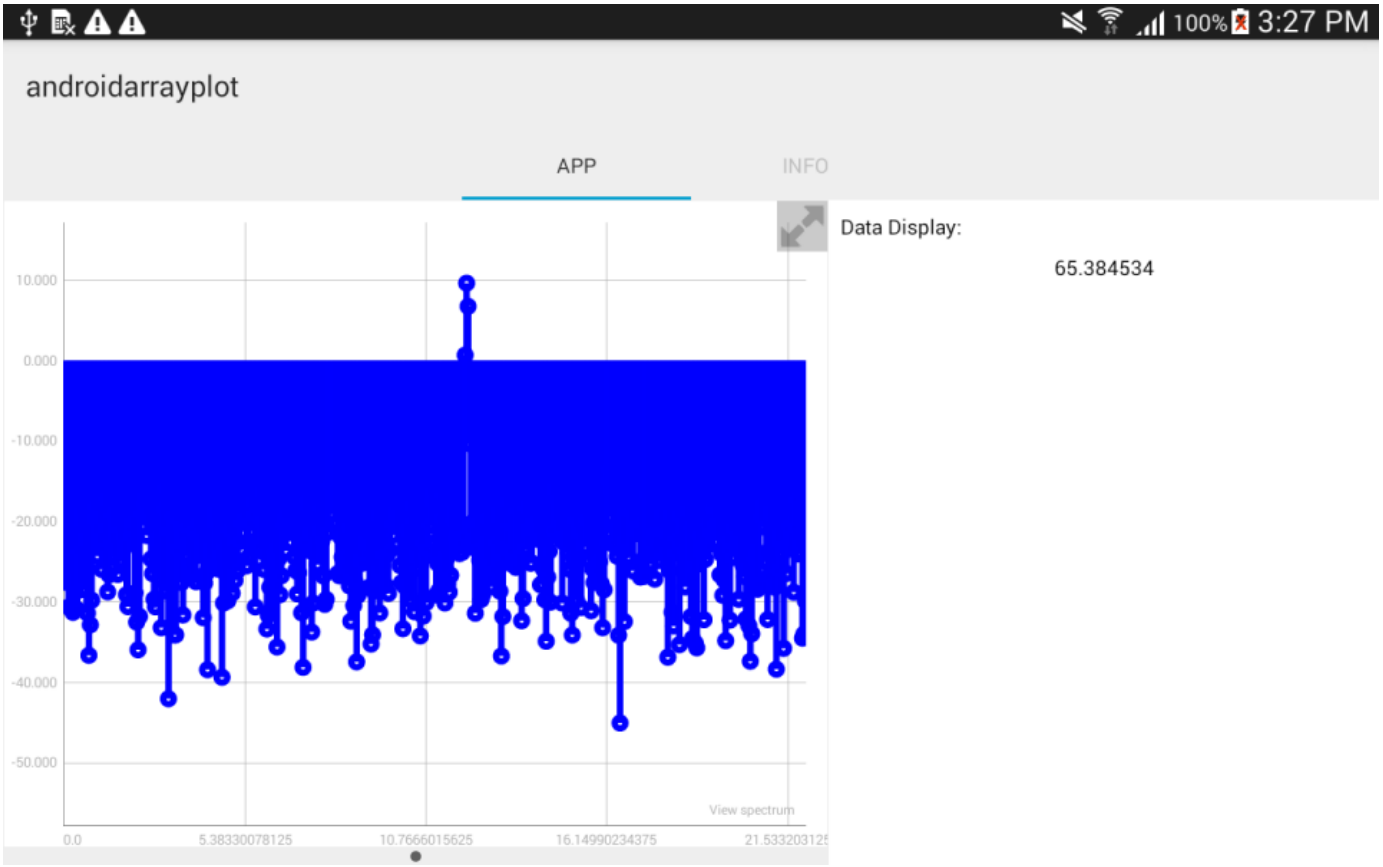
6. Set **Plot type** to **Stem**.

7. Click **OK**.

8. To see the style changes reflected in the app, you must remove the previous project. In the MATLAB Command Window, run:

```
rmdir('androidarrayplot_ert_rtw', 's');
```

9. To update these style changes on the Android device, you must re-build and download the changes by clicking on the Deploy to Hardware button on the model editor.



The Array Plot in the app reflects the new line and axes properties from the Style dialog box of the Array Plot block.

Other Things to Try

- Modify the model to display signals from Android Device sensors.
- Change the scope style to suit your app.

See Also

Blocks

Array Plot | Chirp | Random Source | Spectrum Analyzer | Spectrum Estimator

Related Examples

- “Streaming Power Spectrum Estimation Using Welch's Method”

HDL Code Generation

- “HDL Code Generation Support for DSP System Toolbox” on page 19-2
- “Find Blocks and System Objects Supporting HDL Code Generation” on page 19-5
- “High Throughput HDL Algorithms” on page 19-7
- “HDL Filter Architectures” on page 19-9
- “Subsystem Optimizations for Filters” on page 19-14

HDL Code Generation Support for DSP System Toolbox

In this section...
“Blocks” on page 19-2 “System Objects” on page 19-3

Blocks

You can find DSP System Toolbox blocks that support HDL code generation, in the ‘DSP System Toolbox HDL Support' library, in the Simulink library browser. Alternately, you can type `dsphdllib` in the MATLAB command prompt to open this library. The blocks in `dsphdllib` have their parameters set for HDL code generation.

You must also have HDL Coder to generate HDL code for these blocks.

Transforms

- FFT HDL Optimized
- IFFT HDL Optimized

Filtering

- Biquad Filter
- CIC Decimation
- CIC Interpolation
- CIC Decimation HDL Optimized
- Channelizer HDL Optimized
- DC Blocker
- Discrete FIR Filter
- Discrete FIR Filter HDL Optimized
- FIR Decimation
- FIR Interpolation
- FIR Rate Conversion HDL Optimized
- LMS Filter

Math Functions

- Complex to Magnitude-Angle HDL Optimized

Signal Operations

- Downsample
- NCO HDL Optimized
- Repeat
- Sample and Hold
- Upsample

Signal Management

- Convert 1-D to 2-D
- Data Type Conversion
- Frame Conversion
- Multiport Selector
- Selector
- Variable Selector

Sinks

These blocks can be used for simulation visibility in models that generate HDL code, but are not included in the hardware implementation.

- Display
- Matrix Viewer
- Spectrum Analyzer
- Time Scope
- To Workspace
- Triggered To Workspace
- Waterfall

Sources

- Constant
- Sine Wave
- NCO HDL Optimized

Statistics

- Maximum
- Minimum

System Objects

HDL Coder supports the following DSP System Toolbox System objects for HDL code generation:

Transforms

- `dsp.HDLFFT`
- `dsp.HDLIFFT`

Filtering

- `dsp.BiquadFilter`
- `dsp.DCBlocker`
- `dsp.FIRFilter`
- `dsp.HDLFIRFilter`

- `dsp.HDLChannelizer`
- `dsp.HDLFIRRateConverter`
- `dsp.HDLFIRFilter`
- `dsp.HDLCICDecimation`

Math Functions

- `dsp.HDLComplexToMagnitudeAngle`

Signal Operations

- `dsp.Delay`
- `dsp.HDLNCO`

Find Blocks and System Objects Supporting HDL Code Generation

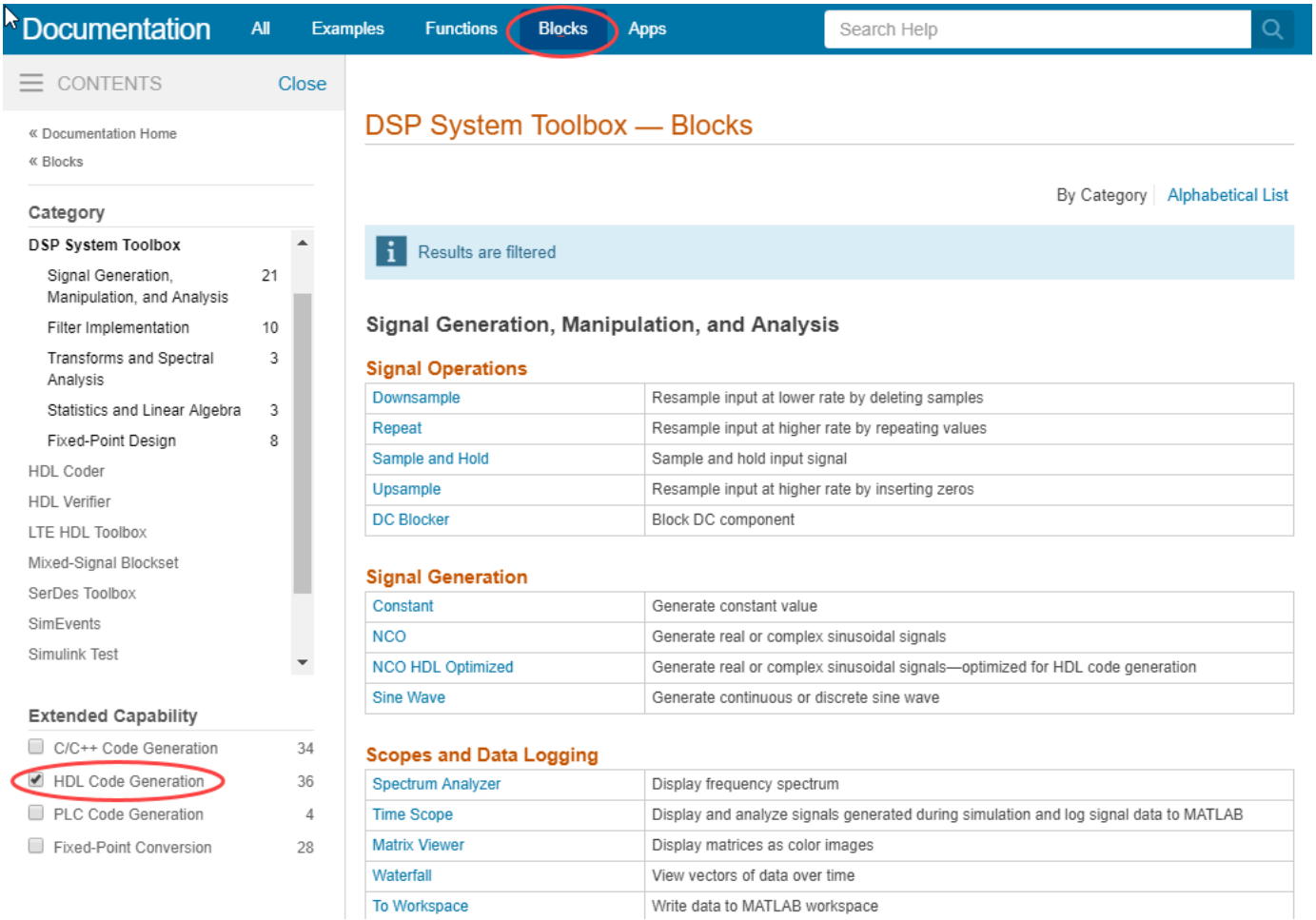
Blocks

In the Simulink library browser, you can find libraries of blocks supported for HDL code generation in the **HDL Coder**, **Communications Toolbox HDL Support**, and **DSP System Toolbox HDL Support** block libraries.

To create a library of HDL-supported blocks from all your installed products, enter `hdllib` at the MATLAB command line. This command requires an HDL Coder license.

Refer to the "Extended Capabilities > HDL Code Generation" section of each block page for block implementations, properties, and restrictions for HDL code generation.

You can also view blocks that are supported for HDL code generation in documentation by filtering the block reference list. Click **Blocks** in the blue bar at the top of the Help window, then select the **HDL code generation** check box at the bottom of the left column. The blocks are listed in their respective products. You can use the table of contents in the left column to navigate between products and categories.



System Objects

To find System objects supported for HDL code generation, see Predefined System Objects (HDL Coder).

High Throughput HDL Algorithms

You can increase the throughput of HDL designs by using frame-based processing. The ports of these blocks accept column vector input and output signals. Each element of the vector represents a sample in time. The generated HDL code implements the algorithm in parallel on each sample in the input vector. These implementations increase data throughput while using more hardware resources. Use vector input to achieve giga-sample-per-second (GSPS) throughput.

For more information on frame-based design, see “Sample- and Frame-Based Concepts” on page 3-2.

Blocks with HDL Support for Frame Input

Supported Block	Parameters to Enable Frame Input	Limitations
Discrete FIR Filter	1 Connect a column vector to the input port. The input vector size can be up to 512 samples. 2 Set **Input processing** to `Columns as channels (frame based)`. 3 Right-click the block, open **HDL Code** > **HDL Block Properties**, and set the **Architecture** to `Frame Based`. For more information on HDL architectures and parameters, see the “HDL Code Generation” (Simulink) section of the block page.	Frame-based input is not supported with: • Optional block-level reset and enable control signals • Resettable and enabled subsystems • Complex input signals with complex coefficients. You can use either complex input signals and real coefficients, or complex coefficients and real input signals. • Multichannel input • Sharing and streaming optimizations • **Filter structure** set to anything other than `Direct form`.
FFT HDL Optimized and IFFT HDL Optimized	Connect a column vector to the `dataIn` port. The vector size must be a power of 2 between 1 and 64, that is not greater than the FFT length.	Vector input is supported with `Streaming Radix 2^2` architecture only.
Channelizer HDL Optimized	Connect a column vector to the `dataIn` port. The vector size must be a power of 2 between 1 and 64, that is not greater than the FFT length.	

Supported Block	Parameters to Enable Frame Input	Limitations
FIR Decimation	1 Connect a column vector to the input port. The input vector size can be up to 512 samples. 2 Set **Input processing** to `Columns as channels (frame based)`. 3 Set **Rate options** to `Enforce single-rate processing`. 4 Right-click the block, open **HDL Code > HDL Block Properties**, and set the **Architecture** to `Frame Based`.	Frame-based input is not supported with: • Resettable and enabled subsystems • Complex input signals with complex coefficients. You can use either complex input signals and real coefficients, or complex coefficients and real input signals. • Sharing and streaming optimizations
NCO HDL Optimized	Set the **Samples per frame** parameter to the desired output vector size.	
Delay	1 Connect a column vector to the input port. The input vector size can be up to 512 samples. 2 Set **Input processing** to `Columns as channels (frame based)`.	

See Also

Related Examples

- “High Throughput Channelizer for FPGA”

HDL Filter Architectures

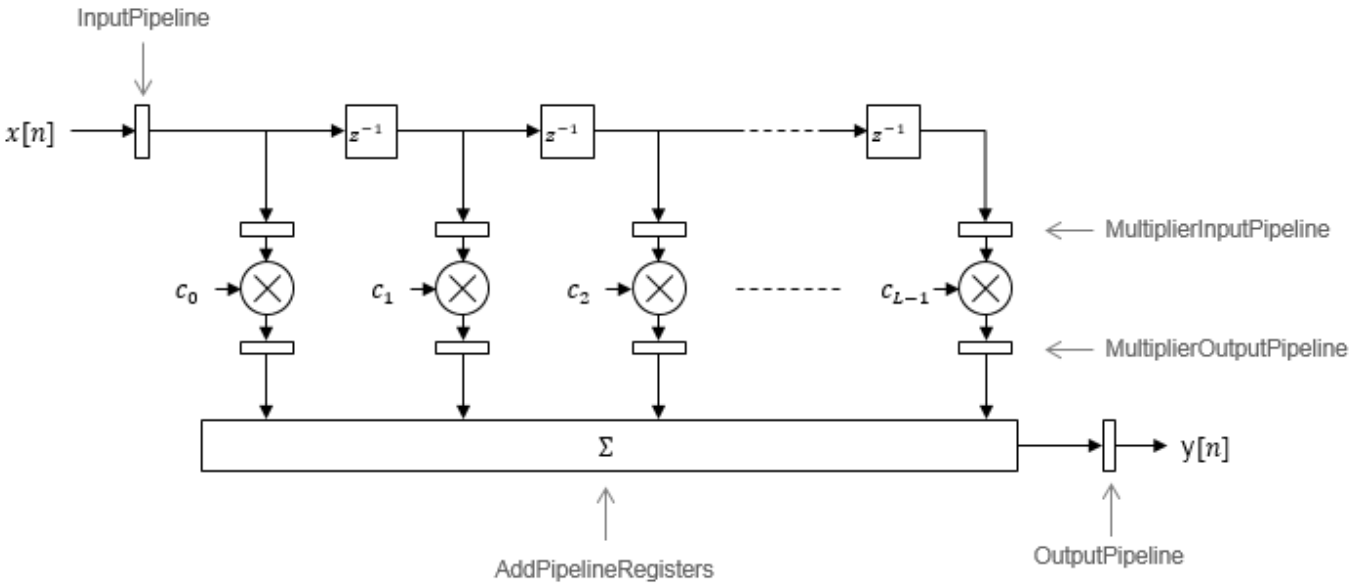
The HDL Coder software provides architecture options that extend your control over speed vs. area tradeoffs in the realization of filter designs. To achieve the desired tradeoff for generated HDL code, you can either specify a fully parallel architecture, or choose one of several serial architectures. Configure a serial architecture using the “SerialPartition” (HDL Coder) and “ReuseAccum” (HDL Coder) parameters. You can also choose a frame-based filter for increased throughput.

Use pipelining parameters to improve speed performance of your filter designs. Add pipelines to the adder logic of your filter using AddPipelineRegisters (HDL Coder) for scalar input filters, and “AdderTreePipeline” (HDL Coder) for frame-based filters. Specify pipeline stages before and after each multiplier with MultiplierInputPipeline (HDL Coder) and MultiplierOutputPipeline (HDL Coder). Set the number of pipeline stages before and after the filter using “InputPipeline” (HDL Coder) and “OutputPipeline” (HDL Coder). The architecture diagrams show the locations of the various configurable pipeline stages.

Fully Parallel Architecture

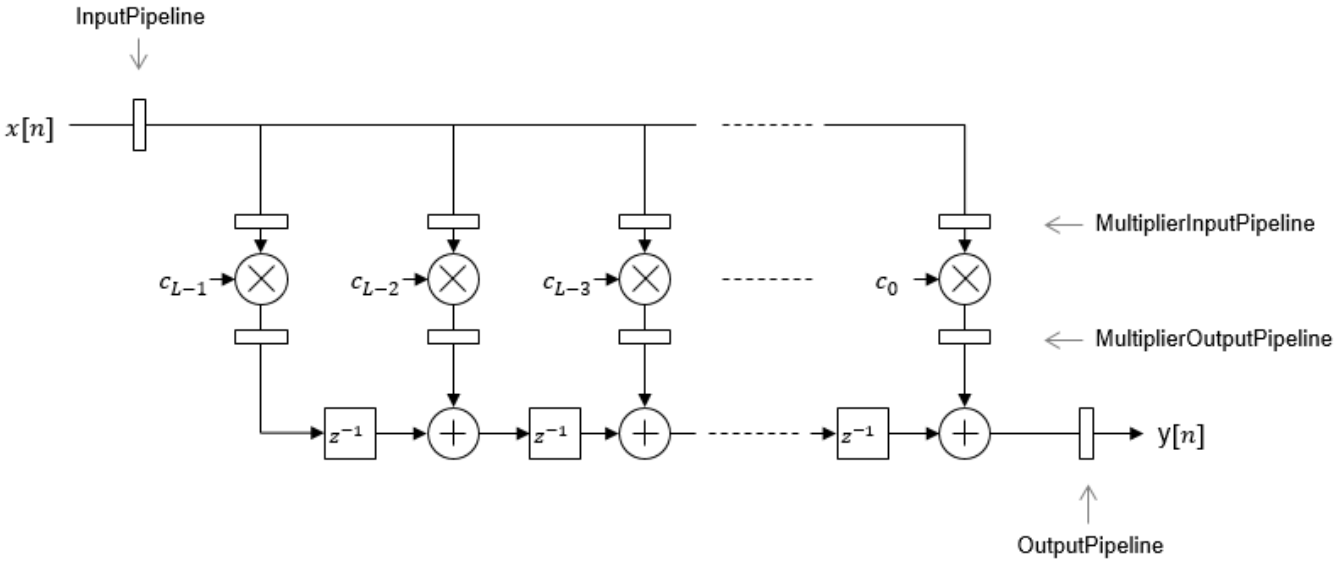
This option is the default architecture. A fully parallel architecture uses a dedicated multiplier and adder for each filter tap. The taps execute in parallel. A fully parallel architecture is optimal for speed. However, it requires more multipliers and adders than a serial architecture, and therefore consumes more chip area. The diagrams show the architectures for direct form and for transposed filter structures with fully parallel implementations, and the location of configurable pipeline stages.

Direct Form



By default, the block implements linear adder logic. When you enable `AddPipelineRegisters`, the adder logic is implemented as a pipelined adder tree. The adder tree uses full-precision data types. If you generate a validation model, you must use full precision in the original model to avoid validation mismatches.

Transposed



The `AddPipelineRegisters` parameter has no effect on a transposed filter implementation.

Serial Architectures

Serial architectures reuse hardware resources in time, saving chip area. Configure a serial architecture using the “SerialPartition” (HDL Coder) and “ReuseAccum” (HDL Coder) parameters. The available serial architecture options are *fully serial*, *partly serial*, and *cascade serial*.

Fully Serial

A fully serial architecture conserves area by reusing multiplier and adder resources sequentially. For example, a four-tap filter design uses a single multiplier and adder, executing a multiply-accumulate operation once for each tap. The multiply-accumulate section of the design runs at four times the filter's input/output sample rate. This design saves area at the cost of some speed loss and higher power consumption.

In a fully serial architecture, the system clock runs at a much higher rate than the sample rate of the filter. Thus, for a given filter design, the maximum speed achievable by a fully serial architecture is less than that of a parallel architecture.

Partly Serial

Partly serial architectures cover the full range of speed vs. area tradeoffs that lie between fully parallel and fully serial architectures.

In a partly serial architecture, the filter taps are grouped into a number of serial partitions. The taps within each partition execute serially, but the partitions execute in parallel with respect to one another. The outputs of the partitions are summed at the final output.

When you select a partly serial architecture, you specify the number of partitions and the length (number of taps) of each partition. Suppose you specify a four-tap filter with two partitions, each having two taps. The system clock runs at twice the filter's sample rate.

Cascade Serial

A cascade-serial architecture closely resembles a partly serial architecture. As in a partly serial architecture, the filter taps are grouped into a number of serial partitions that execute in parallel with respect to one another. However, the accumulated output of each partition is cascaded to the accumulator of the previous partition. The output of all partitions is therefore computed at the accumulator of the first partition. This technique is termed *accumulator reuse*. A final adder is not required, which saves area.

The cascade-serial architecture requires an extra cycle of the system clock to complete the final summation to the output. Therefore, the frequency of the system clock must be increased slightly with respect to the clock used in a noncascade partly serial architecture.

To generate a cascade-serial architecture, specify a partly serial architecture with accumulator reuse enabled. If you do not specify the serial partitions, HDL Coder automatically selects an optimal partitioning.

Latency in Serial Architectures

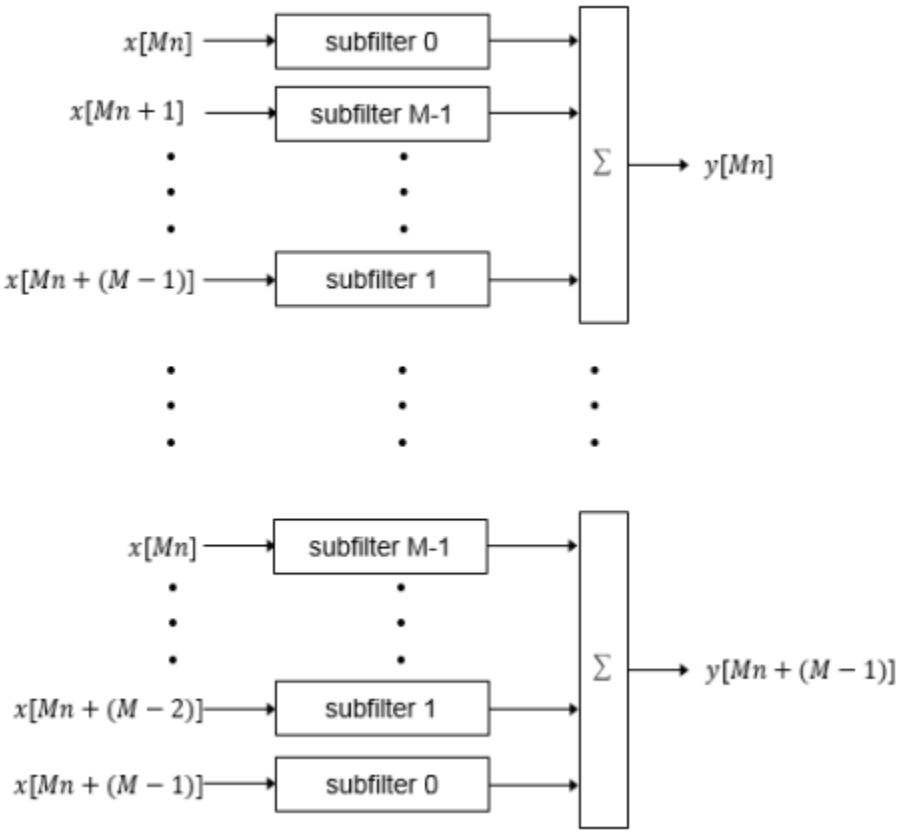
Serialization of a filter increases the total latency of the design by one clock cycle. The serial architectures use an accumulator (an adder with a register) to add the products sequentially. An additional final register is used to store the summed result of all the serial partitions, requiring an extra clock cycle for the operation. To model this latency, HDL Coder inserts a Delay block into the generated model after the filter block.

Full-Precision for Serial Architectures

When you choose a serial architecture, the code generator uses full precision in the HDL code. HDL Coder therefore forces full precision in the generated model. If you generate a validation model, you must use full precision in the original model to avoid validation mismatches.

Frame-Based Architecture

When you select a frame-based architecture and provide an M-sample input frame, the coder implements a fully parallel filter architecture. The filter includes M parallel subfilters for each input sample.

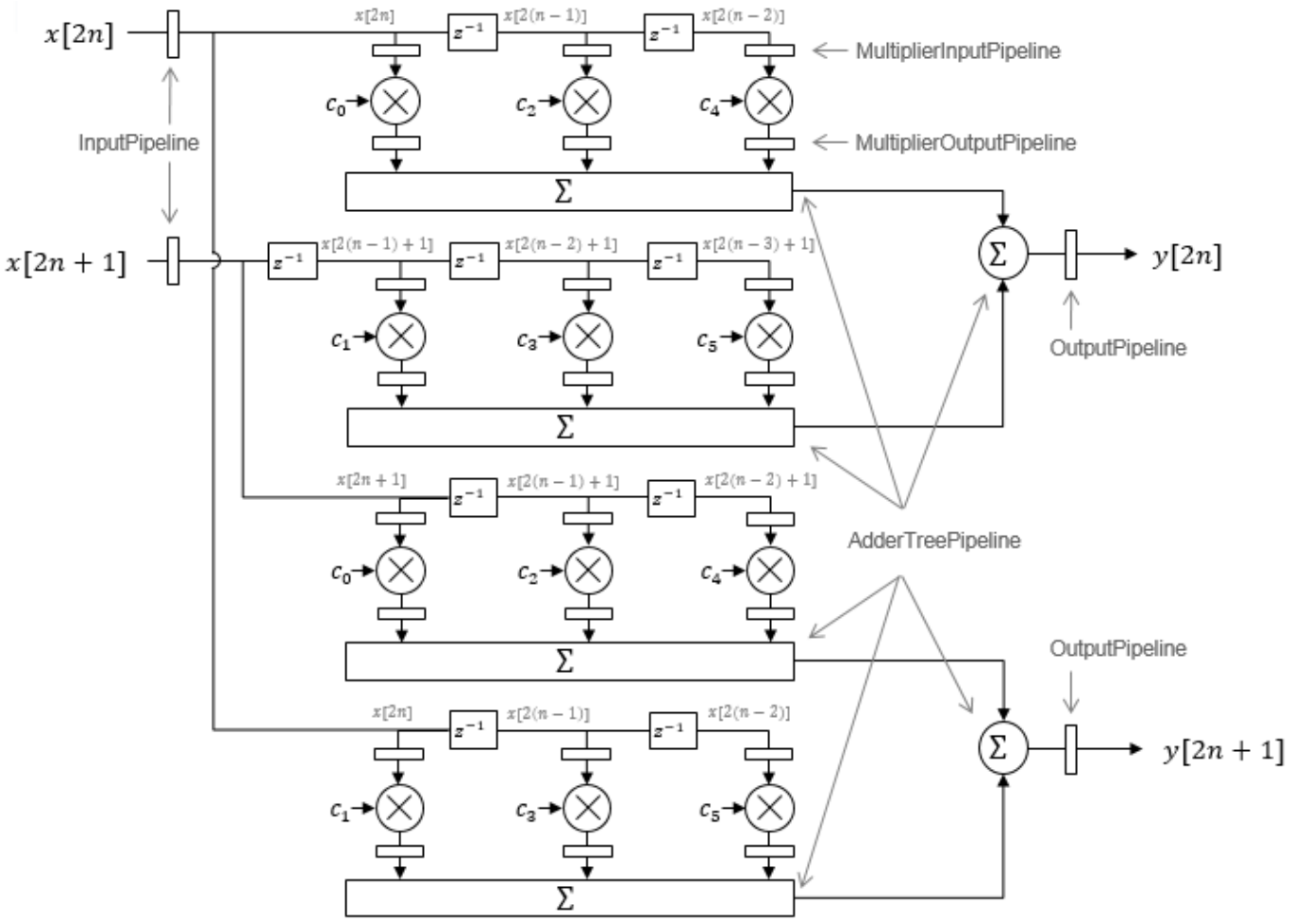


Each of the subfilters includes every *M*th coefficient. The subfilter results are added so that each output sample is the sum of each of the coefficients multiplied with one input sample.

subfilter 0 = $c_0, c_M, \ldots$
subfilter 1 = $c_1, c_{M+1}, \ldots$
...
subfilter *M*–1 = $c_{M-1}, c_{2M-1}, \ldots$

The diagram shows the filter architecture for a frame size of two samples ($M = 2$), and a filter length of six coefficients. The input is a vector with two values representing samples in time. The input samples, *x[2n]* and *x[2n+1]*, represent the *n*th input pair. Every second sample from each stream is fed to two parallel subfilters. The four subfilter results are added together to create two output samples. In this way, each output sample is the sum of each of the coefficients multiplied with one of the input samples.

The sums are implemented as a pipelined adder tree. Set “AdderTreePipeline” (HDL Coder) to specify the number of pipeline stages between levels of the adder tree. To improve clock speed, it is recommended that you set this parameter to 2. To fit the multipliers into DSP blocks on your FPGA, add pipeline stages before and after the multipliers using MultiplierInputPipeline (HDL Coder) and MultiplierOutputPipeline (HDL Coder).



For symmetric or antisymmetric coefficients, the filter architecture reuses the coefficient multipliers and adds design delay between the multiplier and summation stages as required.

See Also

More About

- “HDL Filter Block Properties” (HDL Coder)
- “Distributed Arithmetic for HDL Filters” (HDL Coder)

Subsystem Optimizations for Filters

The Discrete FIR Filter (when used with scalar or multichannel input data) and Biquad Filter blocks participate in subsystem-level optimizations. To set optimization properties, right-click on the subsystem and open the **HDL Properties** dialog box.

For these blocks to participate in subsystem-level optimizations, you must leave the block-level **Architecture** set to the default, `Fully parallel`.

You cannot use these subsystem optimizations when using the Discrete FIR Filter in frame-based input mode.

Sharing

These filter blocks support sharing resources within the filter and across multiple blocks in the subsystem. When you specify a **SharingFactor**, the optimization tools generate a filter implementation in HDL that shares resources using time-multiplexing. To generate an HDL implementation that uses the minimum number of multipliers, set the **SharingFactor** to a number greater than or equal to the total number of multipliers. The sharing algorithm shares multipliers that have the same input and output data types. To enable sharing between blocks, you may need to customize the internal data types of the filters. Alternatively, you can target a particular system clock rate with your choice of **SharingFactor**.

Resource sharing applies to multipliers by default. To share adders, select the check box under **Resource sharing** on the **Configuration Parameters > HDL Code Generation > Global Settings > Optimizations** dialog box.

For more information, see “Resource Sharing” (HDL Coder) and the “Area Reduction of Filter Subsystem” on page 19-15 example.

You can also use a **SharingFactor** with multichannel filters. See “Area Reduction of Multichannel Filter Subsystem” on page 19-17.

Streaming

Streaming refers to sharing an atomic part of the design across multiple channels. To generate a streaming HDL implementation of a multichannel subsystem, set **StreamingFactor** to the number of channels in your design.

If the subsystem contains a single filter block, the block-level **ChannelSharing** option and the subsystem-level **StreamingFactor** option result in similar HDL implementations. Use **StreamingFactor** when your subsystem contains either more than one filter block or additional multichannel logic that can participate in the optimization. You must set block-level **ChannelSharing** to `off` to use **StreamingFactor** at the subsystem level.

See “Streaming” (HDL Coder) and the “Area Reduction of Multichannel Filter Subsystem” on page 19-17 example.

Pipelining

You can enable **DistributedPipelining** at the subsystem level to allow the filter to participate in pipeline optimizations. The optimization tools operate on the **InputPipeline** and **OutputPipeline**

pipeline stages specified at subsystem level. The optimization tools also operate on these block-level pipeline stages:

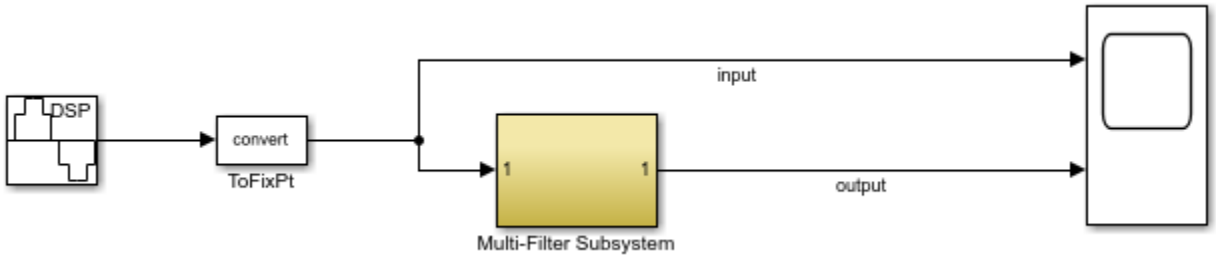
- **InputPipeline** and **OutputPipeline**
- **MultiplierInputPipeline** and **MultiplierOutputPipeline**
- **AddPipelineRegisters**

The optimization tools do not move design delays within the filter architecture. See “Distributed Pipelining” (HDL Coder).

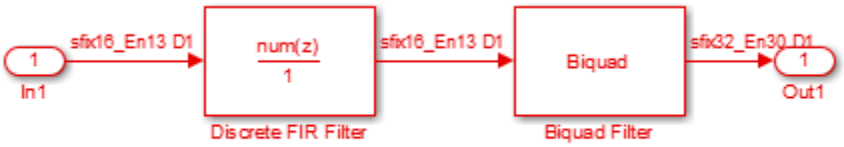
The filter block also participates in clock-rate pipelining, if enabled in **Configuration Parameters**. This feature is enabled by default. See “Clock-Rate Pipelining” (HDL Coder).

Area Reduction of Filter Subsystem

To reduce the number of multipliers in the HDL implementation of a multifilter design, use the **SharingFactor** HDL Coder™ optimization.



The model includes a sinusoidal signal source feeding a filter subsystem targeted for HDL code generation.

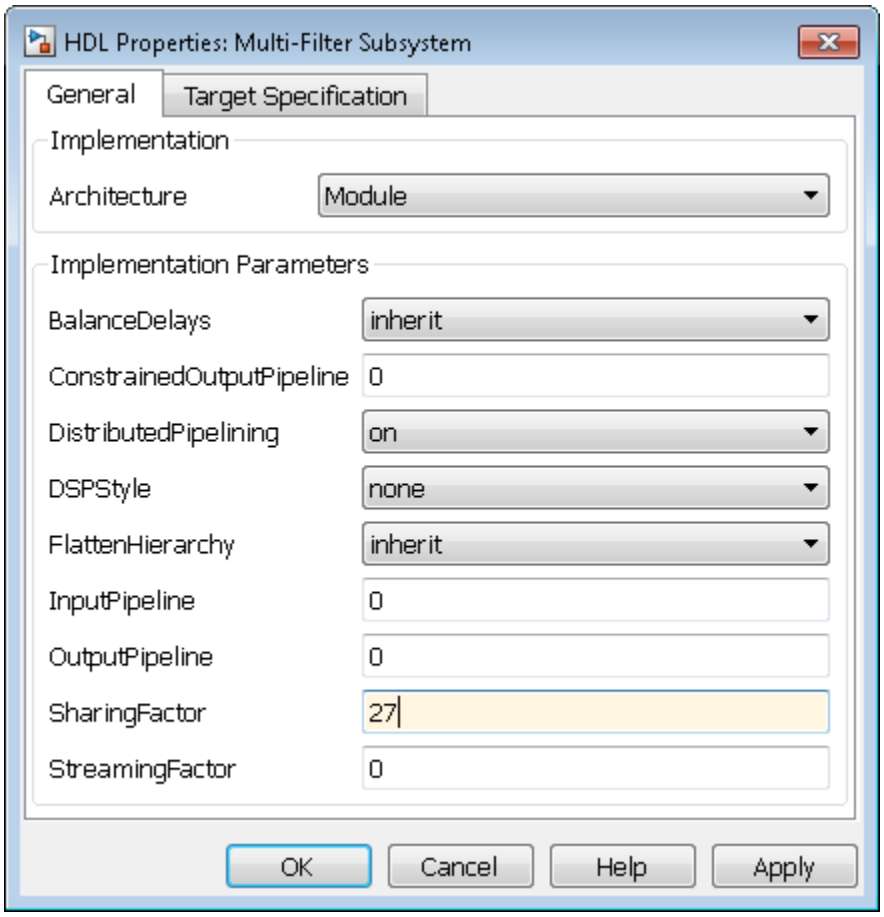


The subsystem contains a Discrete FIR Filter block and a Biquad Filter block. This design demonstrates how the optimization tools share resources between multiple filter blocks.

The Discrete FIR Filter block has 43 symmetric coefficients. The Biquad Filter block has 6 coefficients, two of which are unity. With no optimizations enabled, the generated HDL code takes advantage of symmetry and unity coefficients. The nonoptimized HDL implementation of the subsystem uses 27 multipliers.

Multipliers	27
Adders/Subtractors	46
Registers	49
Total 1-Bit Registers	800
RAMs	0
Multiplexers	0
I/O Bits	52

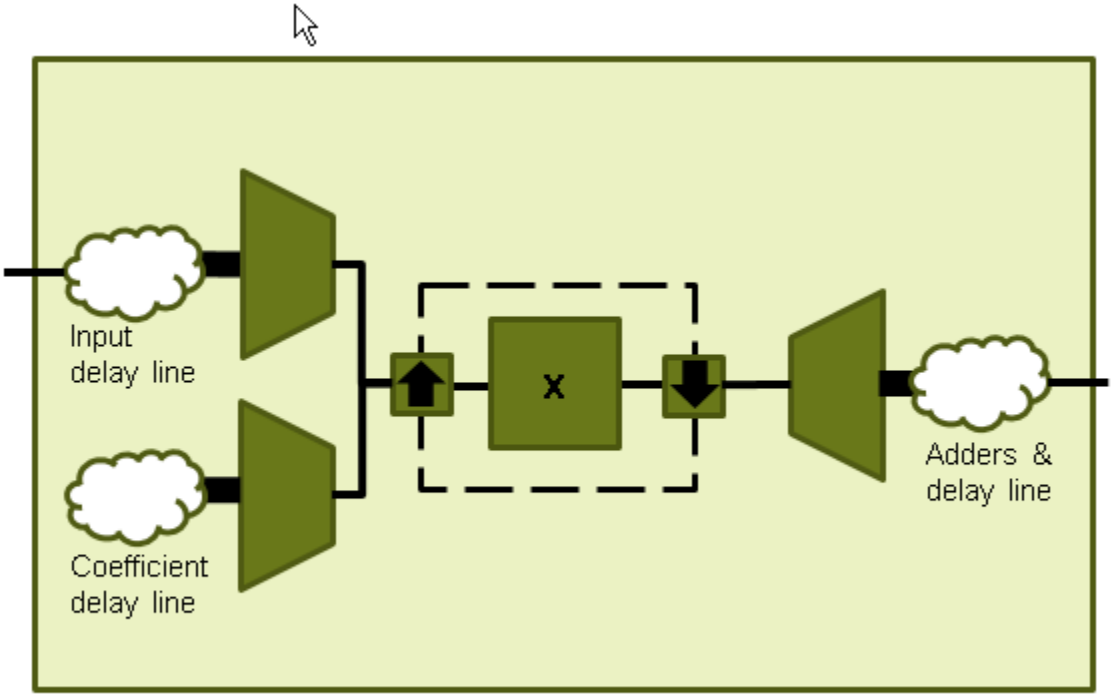
To enable streaming optimization for the **Multi-Filter Subsystem**, right-click the subsystem and select **HDL Code > HDL Block Properties**.



Set the **SharingFactor** to 27 to reduce the design to a single multiplier. The optimization tools attempt to share multipliers with matching data types. To reduce to a single multiplier, you must set the internal data types of the filter blocks to match each other.

To observe the effect of the optimization, under **Configuration Parameters > HDL Code Generation**, select **Generate resource utilization report** and **Generate optimization report**. Then, to generate HDL code, right-click the Multi-Filter Subsystem and select **HDL Code > Generate HDL for Subsystem**.

With the **SharingFactor** applied, the subsystem upsamples the rate by 27 to share a single multiplier for all the coefficients.

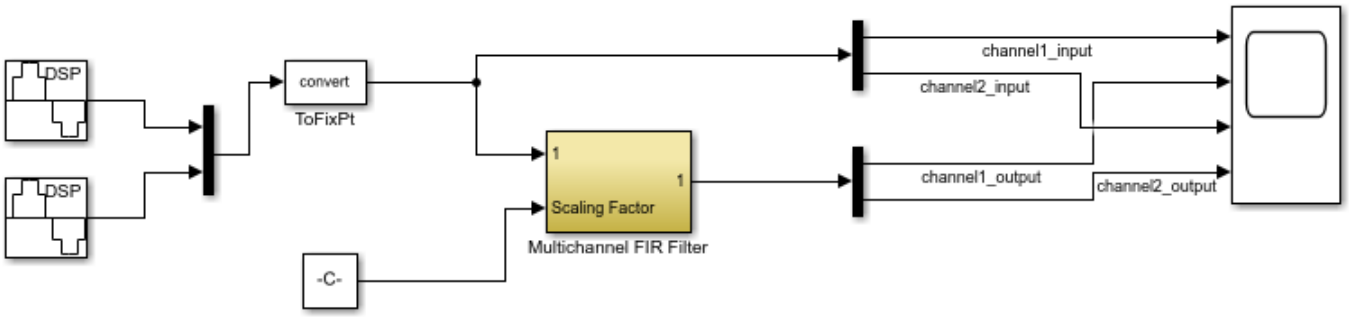


In the **Code Generation Report** window, click **High-level Resource Report**. The generated HDL code now uses one multiplier.

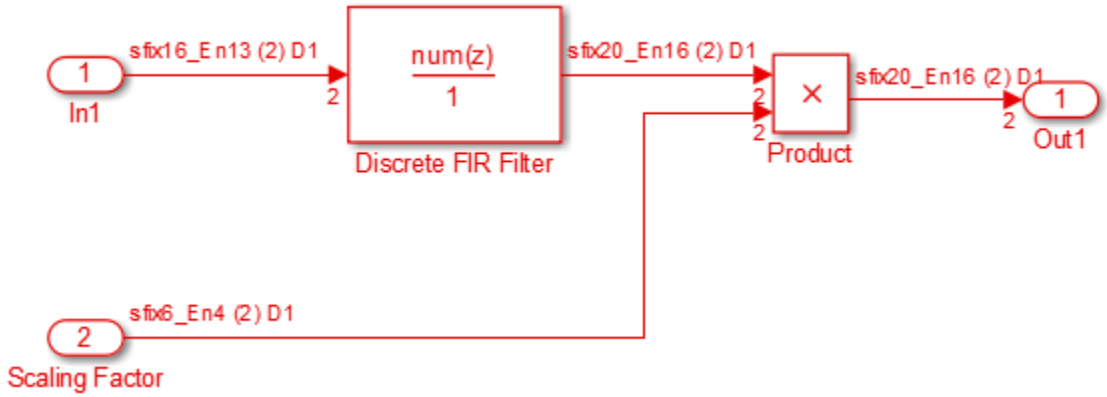
Multipliers	1
Adders/Subtractors	48
Registers	102
Total 1-Bit Registers	2457
RAMs	0
Multiplexers	7
I/O Bits	52

Area Reduction of Multichannel Filter Subsystem

To reduce the number of multipliers in the HDL implementation of a multichannel filter and surrounding logic, use the **StreamingFactor** HDL Coder™ optimization.

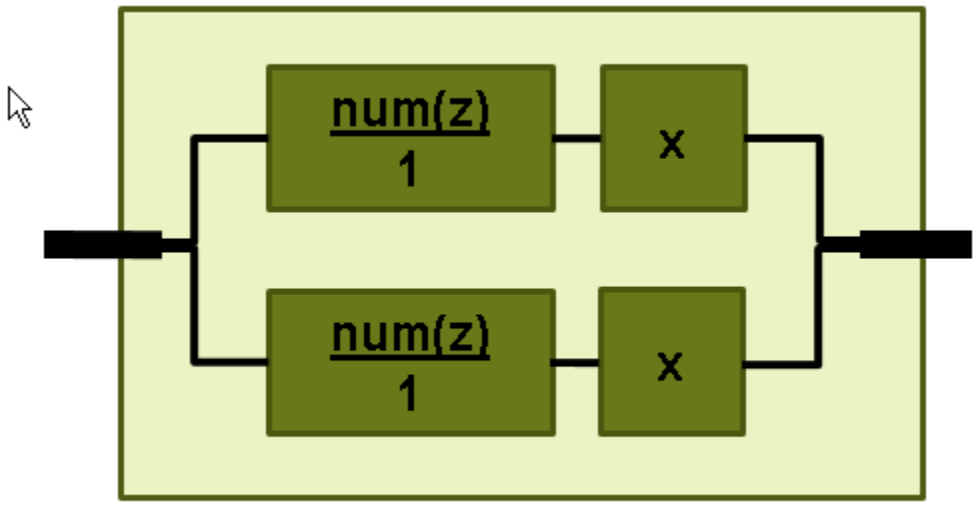


The model includes a two-channel sinusoidal signal source feeding a filter subsystem targeted for HDL code generation.

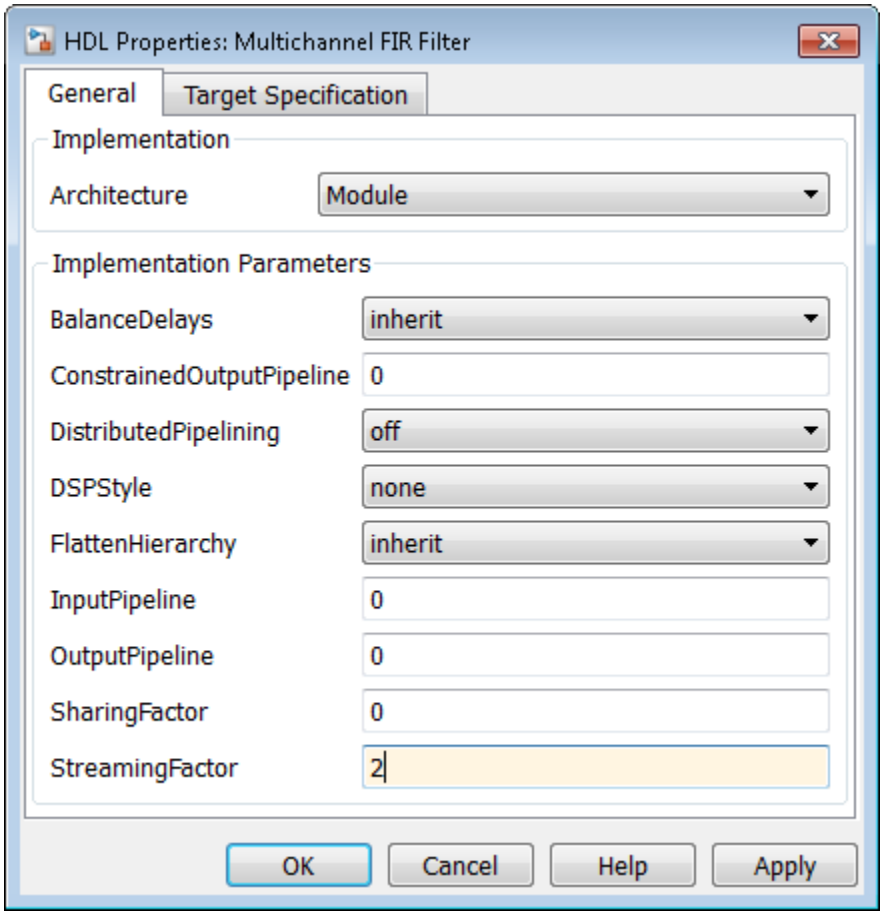


The subsystem contains a Discrete FIR Filter block and a constant multiplier. The multiplier is included to show the optimizations operating over all eligible logic in a subsystem.

The filter has 44 symmetric coefficients. With no optimizations enabled, the generated HDL code takes advantage of symmetry. The nonoptimized HDL implementation uses 46 multipliers: 22 for each channel of the filter and 1 for each channel of the Product block.



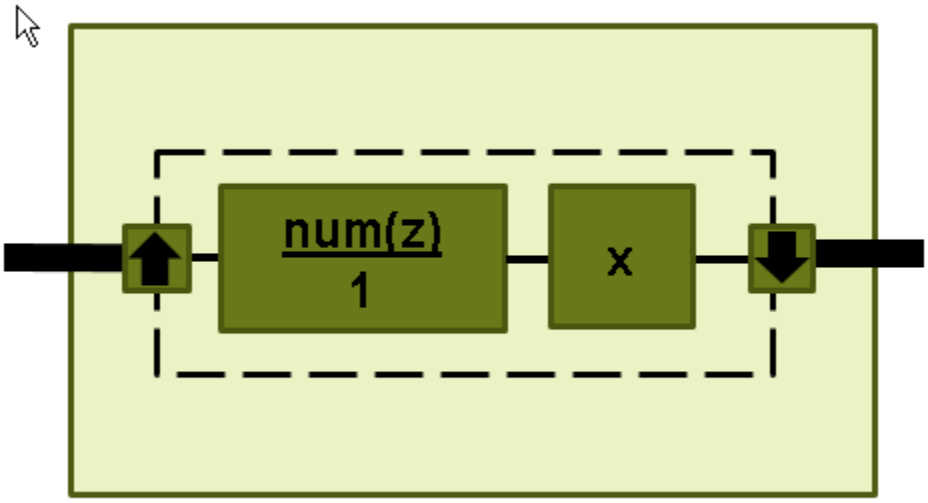
To enable streaming optimization for the Multichannel FIR Filter Subsystem, right-click the subsystem and select **HDL Code > HDL Block Properties**.



Set the **StreamingFactor** to 2, because this design is a two-channel system.

To observe the effect of the optimization, under **Configuration Parameters > HDL Code Generation**, select **Generate resource utilization report** and **Generate optimization report**. Then, to generate HDL code, right-click the Multichannel FIR Filter Subsystem and select **HDL Code > Generate HDL for Subsystem**.

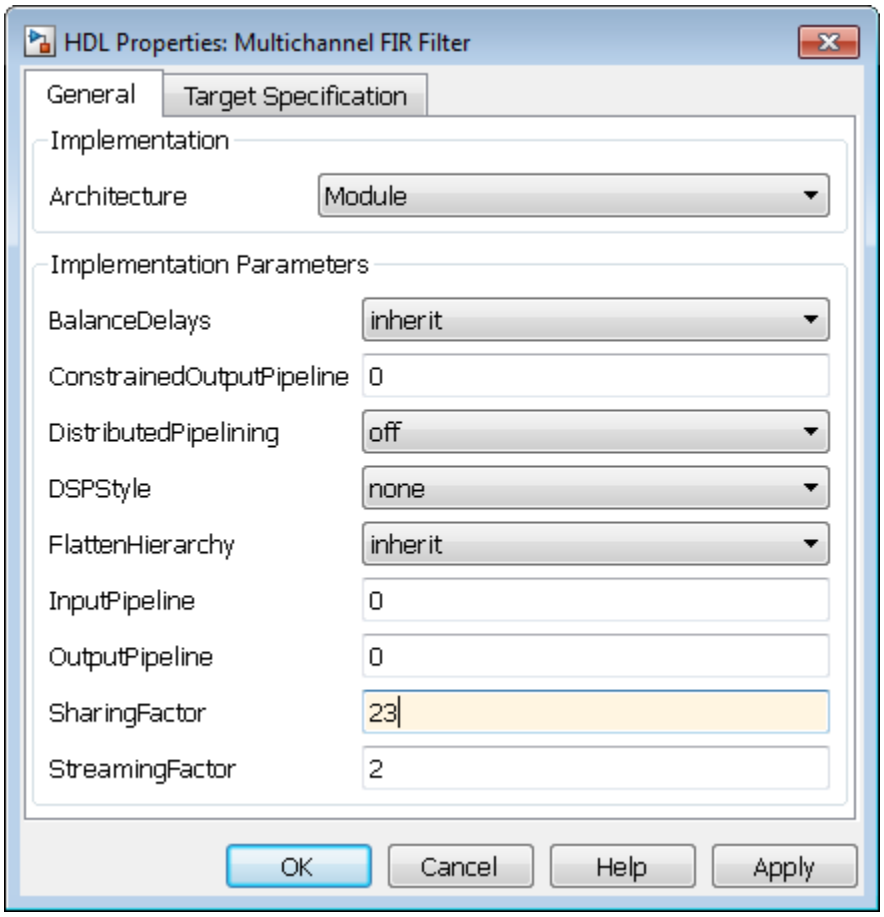
With the streaming factor applied, the logic for one channel is instantiated once and run at twice the rate of the original model.



In the **Code Generation Report** window, click **High-level Resource Report**. The generated HDL code now uses 23 multipliers, compared to 46 in the nonoptimized code. The multipliers in the filter kernel and subsequent scaling are shared between the channels.

Multipliers	23
Adders/Subtractors	44
Registers	92
RAMs	0
Multiplexers	28

To apply **SharingFactor** to multichannel filters, set the **SharingFactor** to 23.



The optimized HDL now uses only 2 multipliers. The optimization tools do not share multipliers of different sizes.

Summary

Multipliers	2
Adders/Subtractors	47
Registers	166
Total 1-Bit Registers	3566
RAMs	0
Multiplexers	37
I/O Bits	80

Detailed Report

Report for Subsystem: *Multichannel FIR Filter*

Multipliers (2)

```
16x16-bit Multiply : 1
[+] 16x6-bit Multiply : 1
```

See Also

More About

- “Resource Sharing” (HDL Coder)
- “Streaming” (HDL Coder)
- “Clock-Rate Pipelining” (HDL Coder)

Links to Category Pages

- “Signal Management Library” on page 20-2
- “Sinks Library” on page 20-3
- “Math Functions Library” on page 20-4
- “Filtering Library” on page 20-5

Signal Management Library

You can find the relevant blocks in the following pages:

- “Buffers, Switches, and Counters”
- “Signal Attributes and Indexing”
- “Signal Operations”

Sinks Library

You can find the relevant blocks in the following pages:

- “Signal Input and Output”
- “Scopes and Data Logging”

Math Functions Library

You can find the relevant blocks in the following pages:

- “Array and Matrix Mathematics”
- “Linear Algebra”

Filtering Library

You can find the relevant blocks in the following pages:

- “Filter Design”
- “Single-Rate Filters”
- “Multirate and Multistage Filters”
- “Adaptive Filters”

Designing Lowpass FIR Filters

- “Lowpass FIR Filter Design” on page 21-2
- “Controlling Design Specifications in Lowpass FIR Design” on page 21-7
- “Designing Filters with Non-Equiripple Stopband” on page 21-12
- “Minimizing Lowpass FIR Filter Length” on page 21-16

Lowpass FIR Filter Design

This example shows how to design a lowpass FIR filter using `fdesign`. An ideal lowpass filter requires an infinite impulse response. Truncating or windowing the impulse response results in the so-called window method of FIR filter design.

A Lowpass FIR Filter Design Using Various Windows

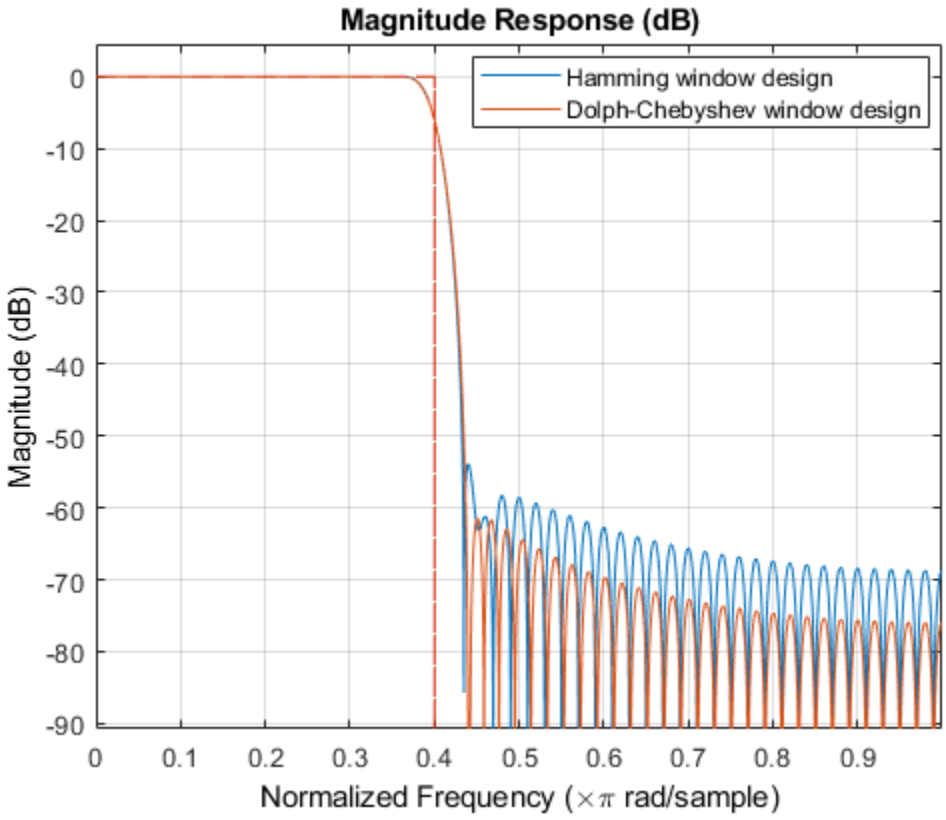
FIR filters are widely used due to the powerful design algorithms that exist for them, their inherent stability when implemented in non-recursive form, the ease with which one can attain linear phase, their simple extensibility to multirate cases, and the ample hardware support that exists for them among other reasons. This example showcases functionality in the DSP System Toolbox™ for the design of low pass FIR filters with a variety of characteristics. Many of the concepts presented here can be extended to other responses such as highpass, bandpass, etc.

Consider a simple design of a lowpass filter with a cutoff frequency of 0.4*pi radians per sample:

```
Fc = 0.4;
N = 100;
Hf = fdesign.lowpass('N,Fc',N,Fc);
```

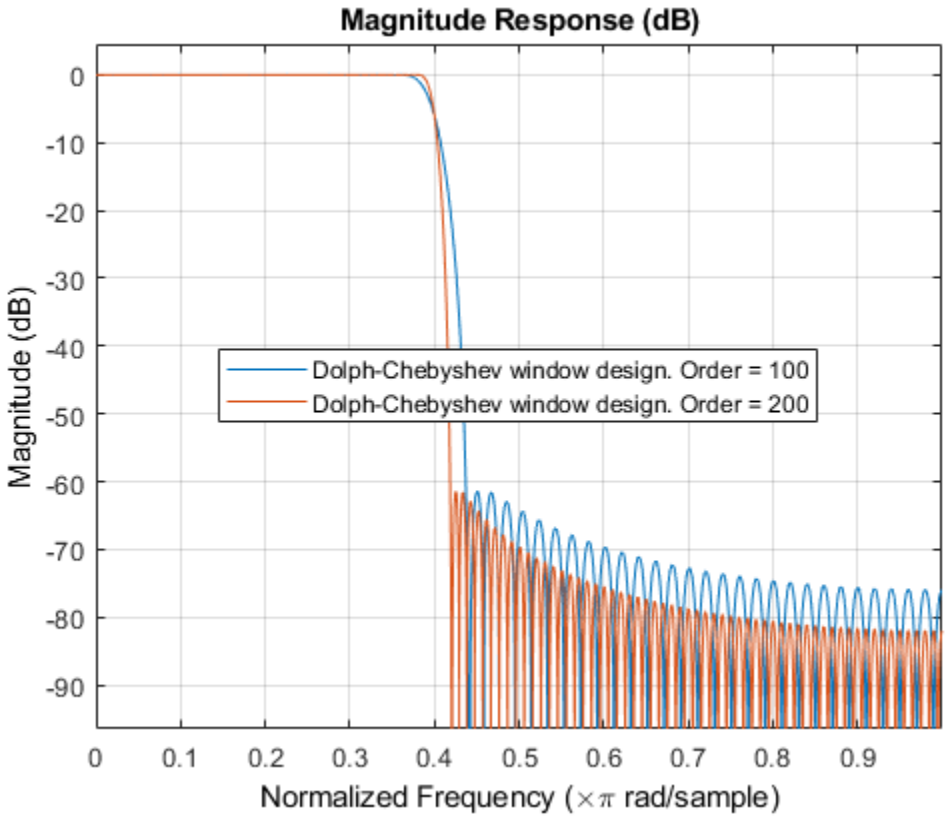
We can design this lowpass filter using the window method. For example, we can use a Hamming window or a Dolph-Chebyshev window:

```
Hd1 = design(Hf,'window','window',@hamming,'systemobject',true);
Hd2 = design(Hf,'window','window',{@chebwin,50}, ...
             'systemobject',true);
hfvt = fvtool(Hd1,Hd2,'Color','White');
legend(hfvt,'Hamming window design', ...
       'Dolph-Chebyshev window design')
```



The choice of filter was arbitrary. Since ideally the order should be infinite, in general, a larger order results in a better approximation to ideal at the expense of a more costly implementation. For instance, with a Dolph-Chebyshev window, we can decrease the transition region by increasing the filter order:

```
Hf.FilterOrder = 200;
Hd3 = design(Hf,'window','window',{@chebwin,50},...
             'systemobject',true);
hfvt2 = fvtool(Hd2,Hd3,'Color','White');
legend(hfvt2,'Dolph-Chebyshev window design. Order = 100',...
    'Dolph-Chebyshev window design. Order = 200')
```



Minimum Order Lowpass Filter Design

In order to determine a suitable filter order, it is necessary to specify the amount of passband ripple and stopband attenuation that will be tolerated. It is also necessary to specify the width of the transition region around the ideal cutoff frequency. The latter is done by setting the passband edge frequency and the stopband edge frequency. The difference between the two determines the transition width.

```
Fp = 0.38;
Fst = 0.42;
Ap = 0.06;
Ast = 60;
setspecs(Hf,'Fp,Fst,Ap,Ast',Fp,Fst,Ap,Ast);
```

We can still use the window method, along with a Kaiser window, to design the low pass filter.

```
Hd4 = design(Hf,'kaiserwin','systemobject',true);
measure(Hd4)
```

```
ans =
Sample Rate      : N/A (normalized frequency)
Passband Edge    : 0.38
3-dB Point       : 0.39539
6-dB Point       : 0.4
Stopband Edge    : 0.42
Passband Ripple  : 0.016058 dB
Stopband Atten.  : 60.092 dB
```

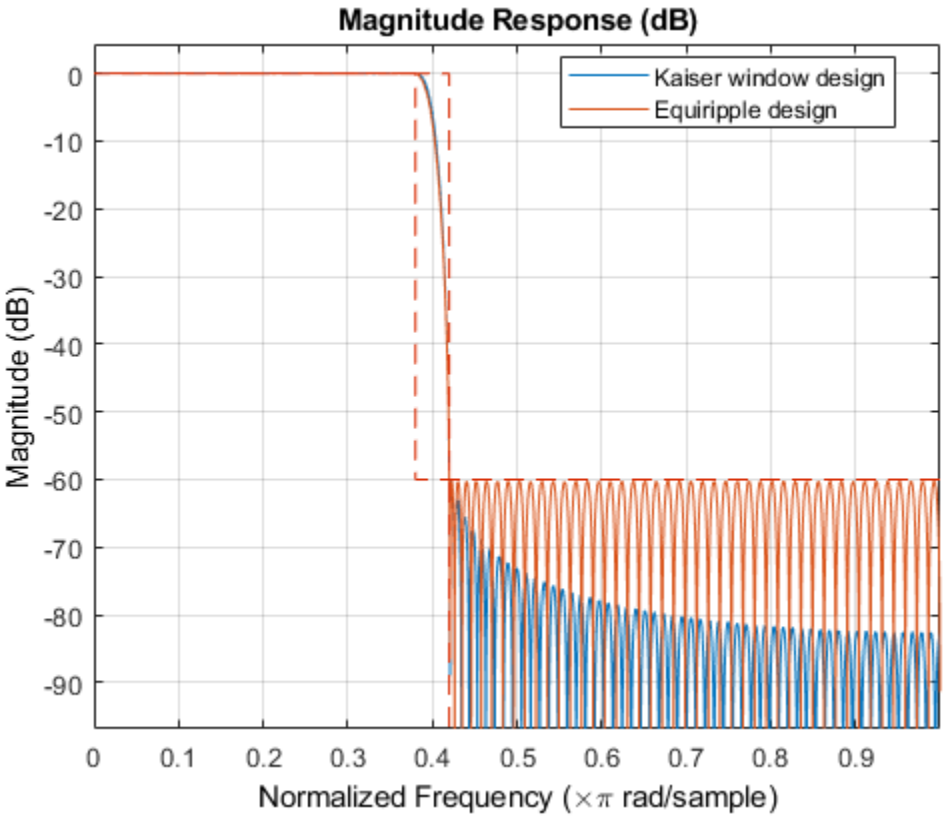
```
Transition Width : 0.04


ans =

Sampling Frequency : N/A (normalized frequency)
Passband Edge      : 0.38
3-dB Point         : 0.39539
6-dB Point         : 0.4
Stopband Edge      : 0.42
Passband Ripple    : 0.016058 dB
Stopband Atten.    : 60.092 dB
Transition Width   : 0.04
```

One thing to note is that the transition width as specified is centered around the cutoff frequency of 0.4 pi. This will become the point at which the gain of the lowpass filter is half the passband gain (or the point at which the filter reaches 6 dB of attenuation).

Optimal Minimum Order Designs

The Kaiser window design is not an optimal design and as a result the filter order required to meet the specifications using this method is larger than it needs to be. Equiripple designs result in the lowpass filter with the smallest possible order to meet a set of specifications.

```
Hd5 = design(Hf,'equiripple','systemobject',true);
hfvt3 = fvtool(Hd4,Hd5,'Color','White');
legend(hfvt3,'Kaiser window design','Equiripple design')
```



In this case, 146 coefficients are needed by the equiripple design while 183 are needed by the Kaiser window design.

See Also

`design` | `fdesign` | `fdesign.lowpass` | `fvtool`

More About

- “Design a Filter in Fdesign — Process Overview” on page 4-2
- “Controlling Design Specifications in Lowpass FIR Design” on page 21-7
- “Designing Filters with Non-Equiripple Stopband” on page 21-12
- “Minimizing Lowpass FIR Filter Length” on page 21-16

Controlling Design Specifications in Lowpass FIR Design

This example shows how to control the filter order, passband ripple, stopband attenuation, and transition region width of a lowpass FIR filter.

Controlling the Filter Order and Passband Ripples and Stopband Attenuation

When targeting custom hardware, it is common to find cases where the number of coefficients is constrained to a set number. In these cases, minimum order designs are not useful because there is no control over the resulting filter order. As an example, suppose that only 101 coefficients could be used and the passband ripple/stopband attenuation specifications need to be met. We can still use equiripple designs for these specifications. However, we lose control over the transition width which will increase. This is the price to pay for reducing the order while maintaining the passband ripple/ stopband attenuation specifications.

Consider a simple design of a lowpass filter with a cutoff frequency of 0.4*pi radians per sample:

```
Ap = 0.06;
Ast = 60;
Fp = 0.38;
Fst = 0.42;
Hf=fdesign.lowpass('Fp,Fst,Ap,Ast',Fp,Fst,Ap,Ast);
```

Design an equiripple filter:

```
Hd1 = design(Hf,'equiripple','systemobject',true);
```

Set the number of coefficients to 101, which means setting the order to 100:

```
N = 100;
Fc = 0.4;
setspecs(Hf,'N,Fc,Ap,Ast',N,Fc,Ap,Ast);
```

Design a second equiripple filter with the given constraint:

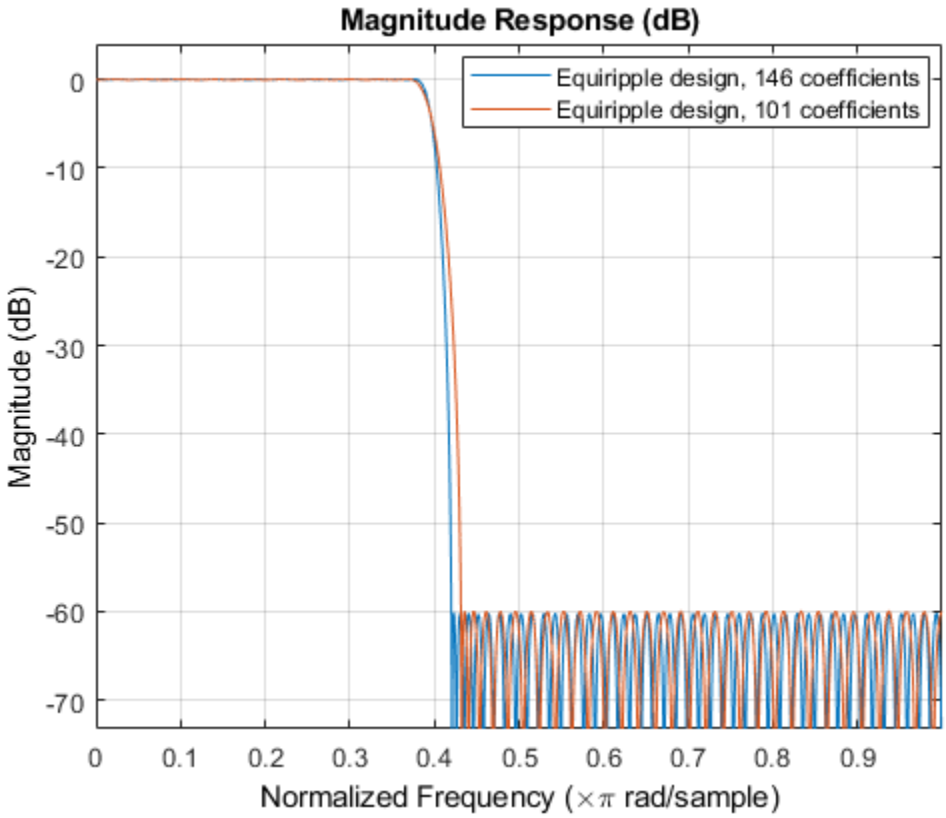
```
Hd2 = design(Hf,'equiripple','systemobject',true);
```

Measure the filter variables of the second equiripple filter, and compare the graphs of the first and second filters:

```
measure(Hd2)
```

```
ans =
Sample Rate      : N/A (normalized frequency)
Passband Edge    : 0.37316
3-dB Point       : 0.39285
6-dB Point       : 0.4
Stopband Edge    : 0.43134
Passband Ripple  : 0.06 dB
Stopband Atten.  : 60 dB
Transition Width : 0.058177
```

```
hfvt = fvtool(Hd1,Hd2,'Color','White');
legend(hfvt,'Equiripple design, 146 coefficients', ...
       'Equiripple design, 101 coefficients')
```



The transition has increased by almost 50%. This is not surprising given the almost 50% difference between 101 coefficients and 146 coefficients.

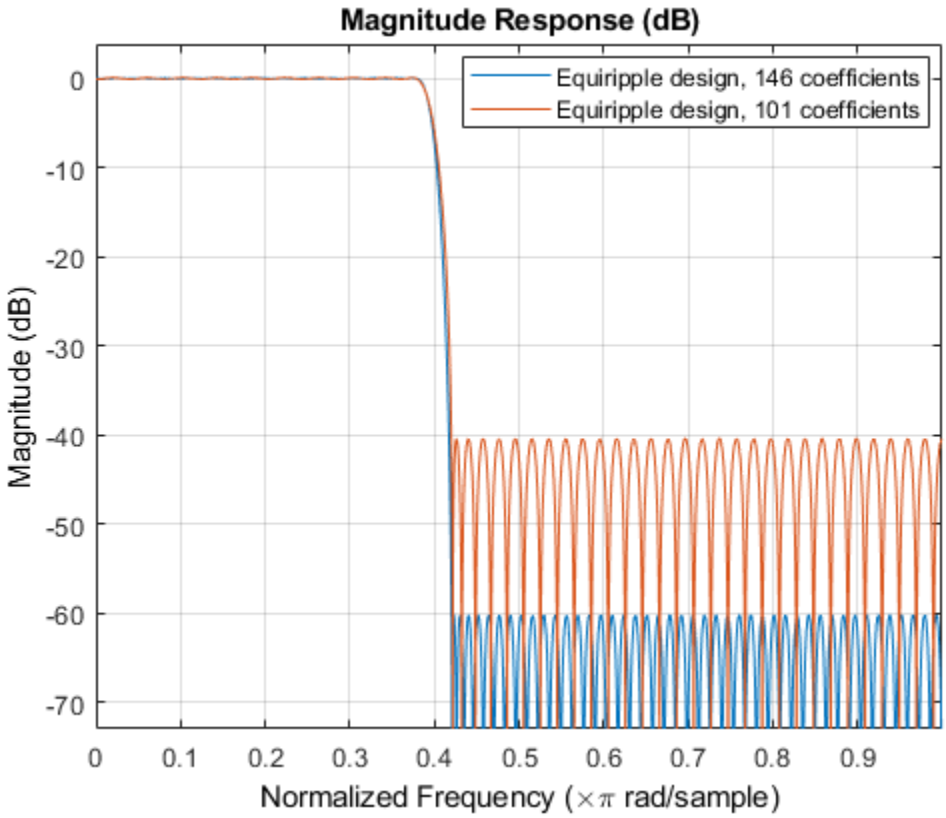
Controlling the Transition Region Width

Another option when the number of coefficients is set is to maintain the transition width at the expense of control over the passband ripple/stopband attenuation.

```
setspecs(Hf,'N,Fp,Fst',N,Fp,Fst);
Hd3 = design(Hf,'equiripple','systemobject',true);
measure(Hd3)
```

```
ans =
Sample Rate      : N/A (normalized frequency)
Passband Edge    : 0.38
3-dB Point       : 0.39407
6-dB Point       : 0.4
Stopband Edge    : 0.42
Passband Ripple  : 0.1651 dB
Stopband Atten.  : 40.4369 dB
Transition Width : 0.04
```

```
hfvt2 = fvtool(Hd1,Hd3,'Color','White');
legend(hfvt2,'Equiripple design, 146 coefficients',...
      'Equiripple design, 101 coefficients')
```



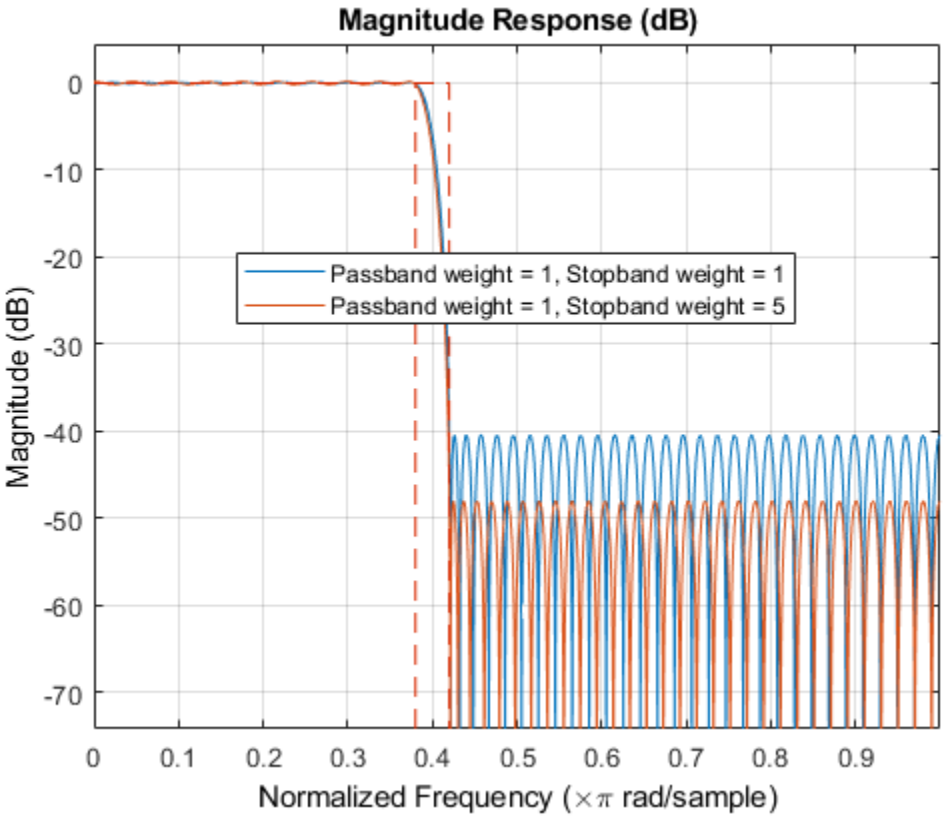
The differences between using 146 coefficients and using 101 coefficients is reflected in a larger passband ripple and a smaller stopband attenuation.

It is possible to increase the attenuation in the stopband while keeping the same filter order and transition width by the use of weights. Weights are a way of specifying the relative importance of the passband ripple versus the stopband attenuation. By default, passband and stopband are equally weighted (a weight of one is assigned to each). If we increase the stopband weight, we can increase the stopband attenuation at the expense of increasing the stopband ripple as well.

```
  Hd4 = design(Hf,'equiripple','Wstop',5,'systemobject',true);
  measure(Hd4)
```

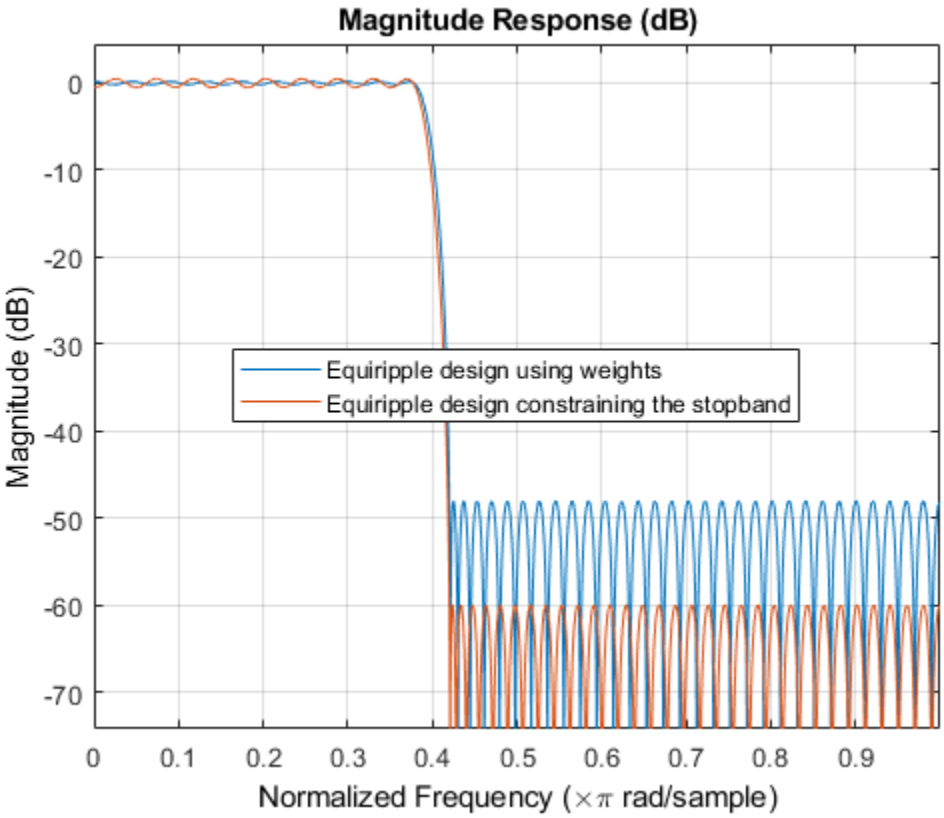
```
ans =
Sample Rate      : N/A (normalized frequency)
Passband Edge    : 0.38
3-dB Point       : 0.39143
6-dB Point       : 0.39722
Stopband Edge    : 0.42
Passband Ripple  : 0.34529 dB
Stopband Atten.  : 48.0068 dB
Transition Width : 0.04
```

```
  hfvt3 = fvtool(Hd3,Hd4,'Color','White');
  legend(hfvt3,'Passband weight = 1, Stopband weight = 1',...
        'Passband weight = 1, Stopband weight = 5')
```



Another possibility is to specify the exact stopband attenuation desired and lose control over the passband ripple. This is a powerful and very desirable specification. One has control over most parameters of interest.

```
setspecs(Hf,'N,Fp,Fst,Ast',N,Fp,Fst,Ast);
Hd5 = design(Hf,'equiripple','systemobject',true);
hfvt4 = fvtool(Hd4,Hd5,'Color','White');
legend(hfvt4,'Equiripple design using weights',...
      'Equiripple design constraining the stopband')
```


Magnitude Response (dB)
Magnitude (dB)
0
-10
-20
-30
-40
-50
-60
-70
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Normalized Frequency (×π rad/sample)
Equiripple design using weights
Equiripple design constraining the stopband

Designing Filters with Non-Equiripple Stopband

This example shows how to design lowpass filters with stopbands that are not equiripple.

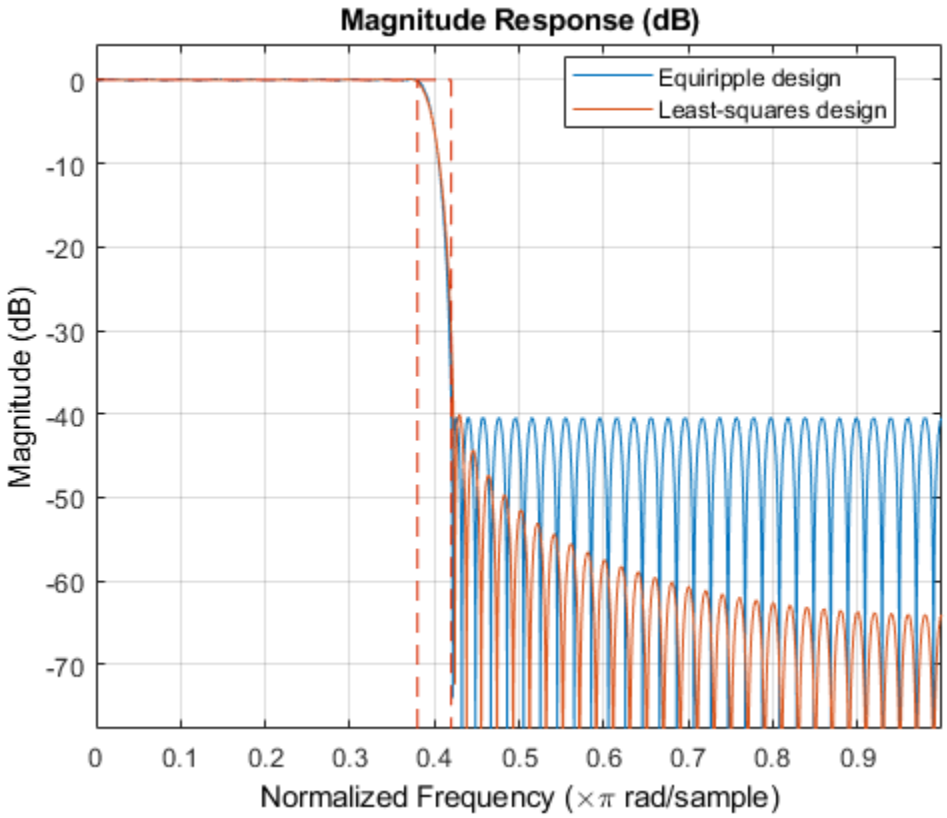
Optimal Non-Equiripple Lowpass Filters

To start, set up the filter parameters and use `fdesign` to create a constructor for designing the filter.

```
N = 100;
Fp = 0.38;
Fst = 0.42;
Hf = fdesign.lowpass('N,Fp,Fst',N,Fp,Fst);
```

Equiripple designs achieve optimality by distributing the deviation from the ideal response uniformly. This has the advantage of minimizing the maximum deviation (ripple). However, the overall deviation, measured in terms of its energy tends to be large. This may not always be desirable. When low pass filtering a signal, this implies that remnant energy of the signal in the stopband may be relatively large. When this is a concern, least-squares methods provide optimal designs that minimize the energy in the stopband.

```
Hd1 = design(Hf,'equiripple','systemobject',true);
Hd2 = design(Hf,'firls','systemobject',true);
hfvt = fvtool(Hd1,Hd2,'Color','White');
legend(hfvt,'Equiripple design','Least-squares design')
```

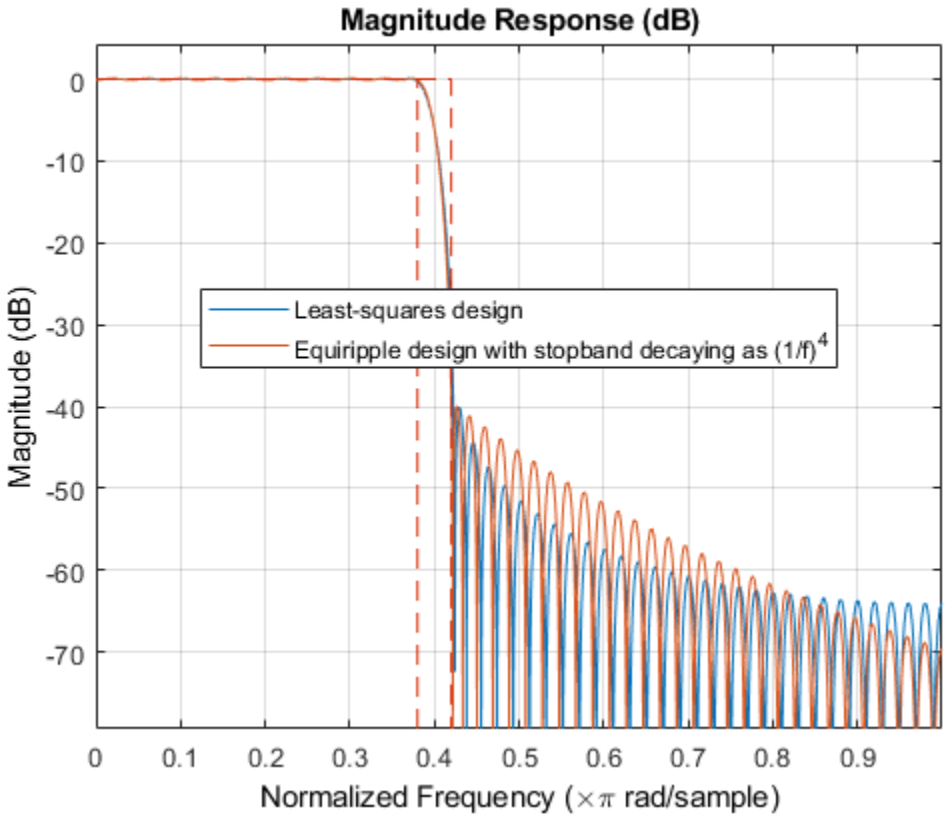


Notice how the attenuation in the stopband increases with frequency for the least-squares designs while it remains constant for the equiripple design. The increased attenuation in the least-squares case minimizes the energy in that band of the signal to be filtered.

Equiripple Designs with Increasing Stopband Attenuation

An often undesirable effect of least-squares designs is that the ripple in the passband region close to the passband edge tends to be large. For low pass filters in general, it is desirable that passband frequencies of a signal to be filtered are affected as little as possible. To this extent, an equiripple passband is generally preferable. If it is still desirable to have an increasing attenuation in the stopband, we can use design options for equiripple designs to achieve this.

```
Hd3 = design(Hf,'equiripple','StopbandShape','1/f',...
             'StopbandDecay',4,'systemobject',true);
hfvt2 = fvtool(Hd2,Hd3,'Color','White');
legend(hfvt2,'Least-squares design',...
       'Equiripple design with stopband decaying as (1/f)^4')
```



Notice that the stopbands are quite similar. However the equiripple design has a significantly smaller passband ripple,

```
mls = measure(Hd2);
meq = measure(Hd3);
mls.Apass
```

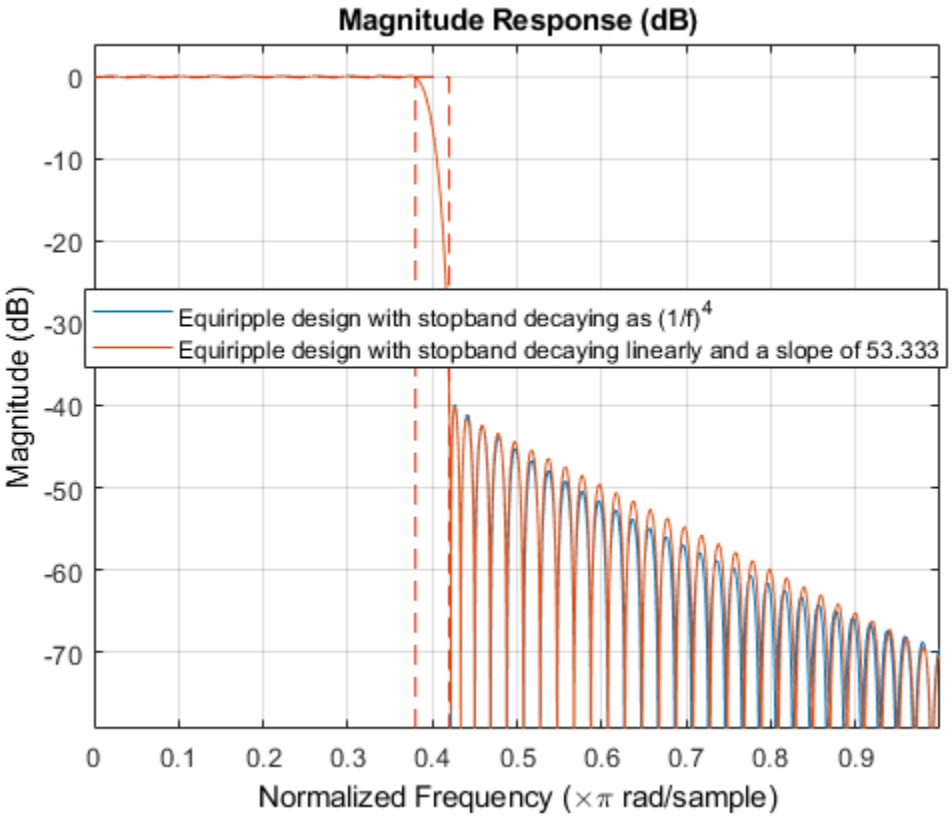
```
ans = 0.3504
```

```
meq.Apass
```

```
ans = 0.1867
```

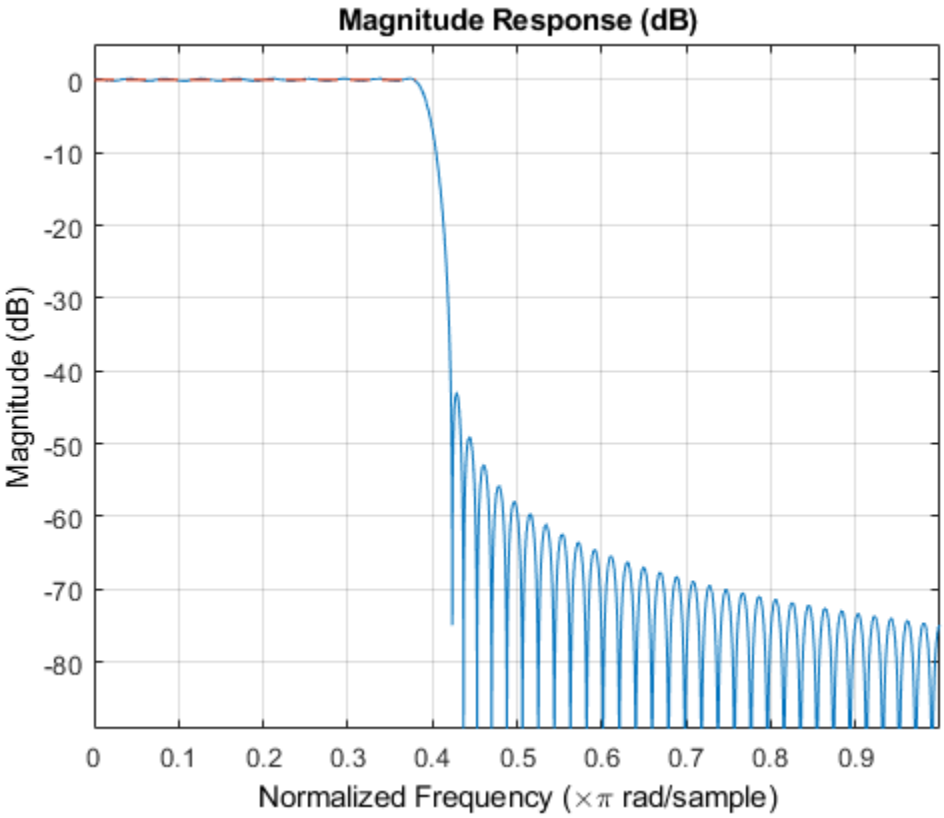
Filters with a stopband that decays as (1/f)^M will decay at 6M dB per octave. Another way of shaping the stopband is using a linear decay. For example given an approximate attenuation of 38 dB at 0.4*pi, if an attenuation of 70 dB is desired at pi, and a linear decay is to be used, the slope of the line is given by (70-38)/(1-0.4) = 53.333. Such a design can be achieved from:

```
Hd4 = design(Hf,'equiripple','StopbandShape','linear',...
             'StopbandDecay',53.333,'systemobject',true);
hfvt3 = fvtool(Hd3,Hd4,'Color','White');
legend(hfvt3,'Equiripple design with stopband decaying as (1/f)^4',...
    'Equiripple design with stopband decaying linearly and a slope of 53.333')
```



Yet another possibility is to use an arbitrary magnitude specification and select two bands (one for the passband and one for the stopband). Then, by using weights for the second band, it is possible to increase the attenuation throughout the band.

```
N = 100;
B = 2;  % number of bands
F = [0 .38 .42:.02:1];
A = [1 1 zeros(1,length(F)-2)];
W = linspace(1,100,length(F)-2);
Harb = fdesign.arbmag('N,B,F,A',N,B,F(1:2),A(1:2),F(3:end),...
                      A(3:end));
Ha = design(Harb,'equiripple','B2Weights',W,...
            'systemobject',true);
fvtool(Ha,'Color','White')
```


Magnitude Response (dB)
Magnitude (dB)
0
-10
-20
-30
-40
-50
-60
-70
-80
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Normalized Frequency (×π rad/sample)

Minimizing Lowpass FIR Filter Length

This example shows how to minimize the number coefficients, by designing minimum-phase or minimum-order filters.

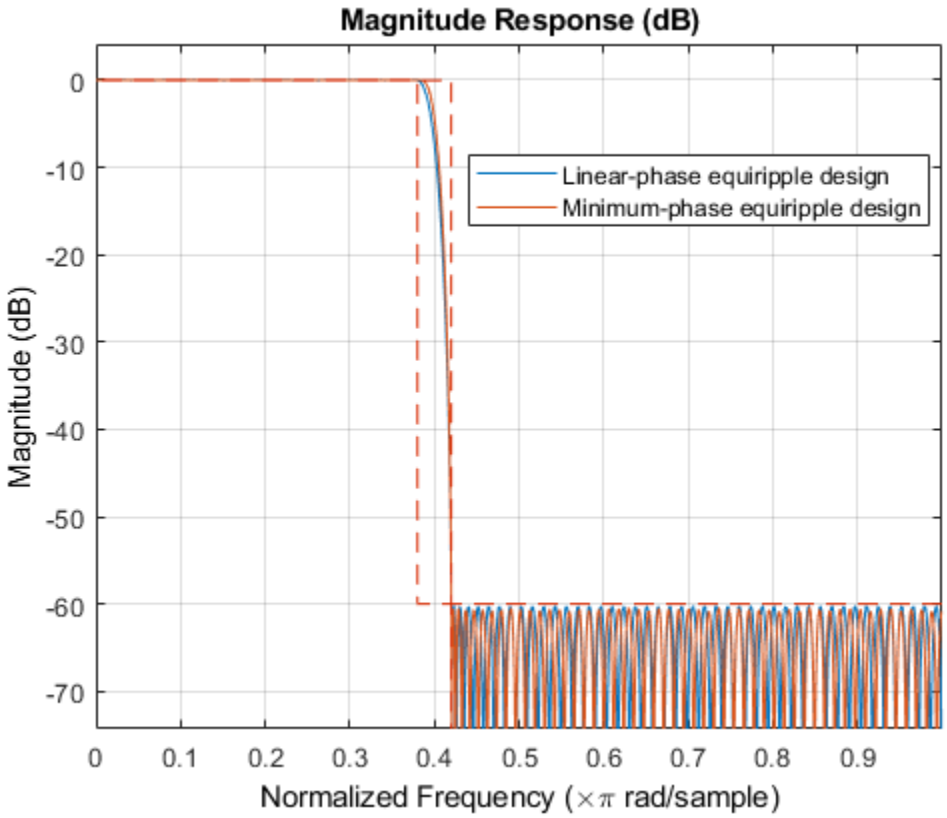
Minimum-Phase Lowpass Filter Design

To start, set up the filter parameters and use `fdesign` to create a constructor for designing the filter.

```
N = 100;
Fp = 0.38;
Fst = 0.42;
Ap = 0.06;
Ast = 60;
Hf = fdesign.lowpass('Fp,Fst,Ap,Ast',Fp,Fst,Ap,Ast);
```

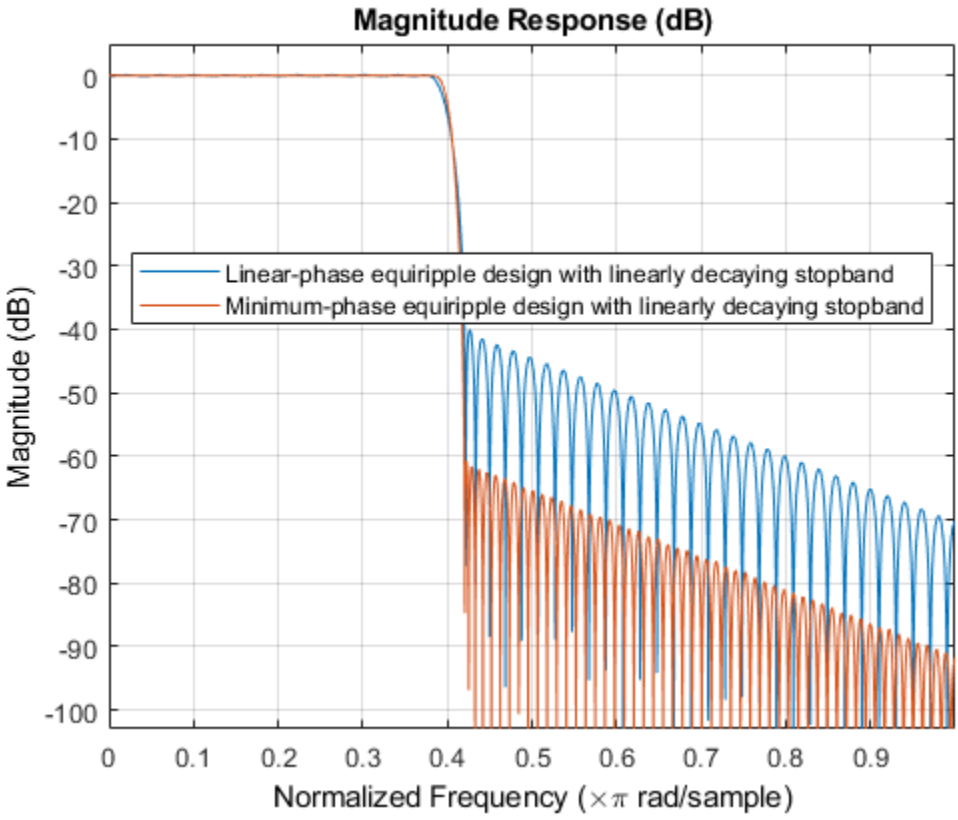
So far, we have only considered linear-phase designs. Linear phase is desirable in many applications. Nevertheless, if linear phase is not a requirement, minimum-phase designs can provide significant improvements over linear phase counterparts. For instance, returning to the minimum order case, a minimum-phase/minimum-order design for the same specifications can be computed with:

```
Hd1 = design(Hf,'equiripple','systemobject',true);
Hd2 = design(Hf,'equiripple','minphase',true,...
             'systemobject',true);
hfvt = fvtool(Hd1,Hd2,'Color','White');
legend(hfvt,'Linear-phase equiripple design',...
       'Minimum-phase equiripple design')
```



Notice that the number of coefficients has been reduced from 146 to 117. As a second example, consider the design with a stopband decaying in linear fashion. Notice the increased stopband attenuation. The passband ripple is also significantly smaller.

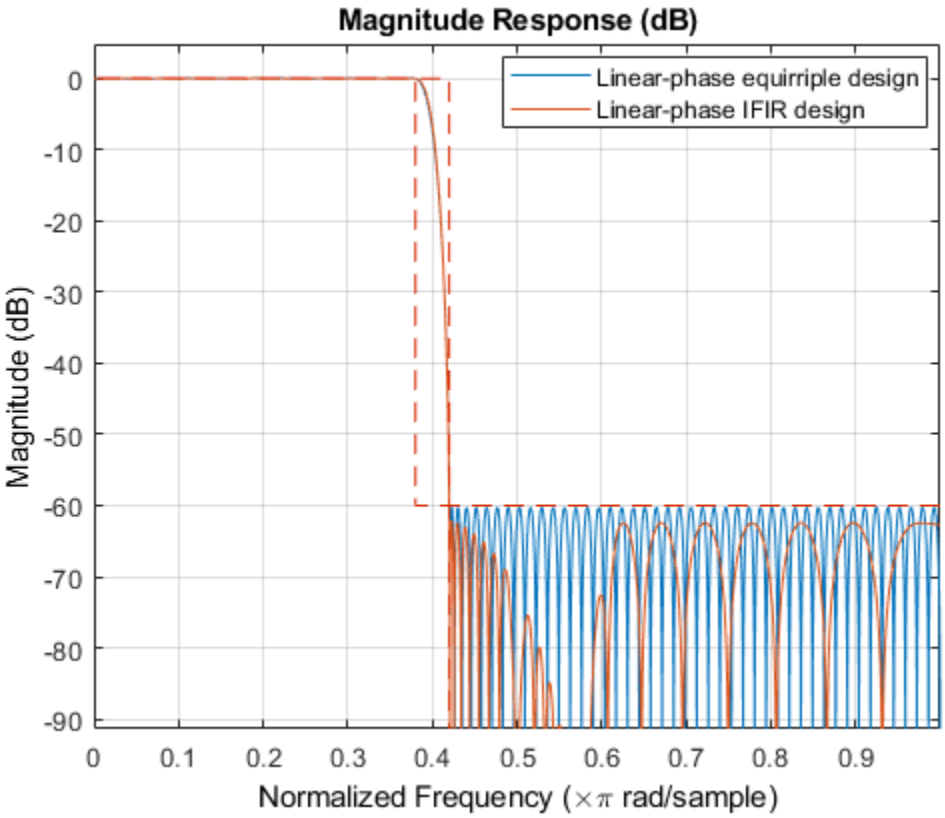
```
setspecs(Hf,'N,Fp,Fst',N,Fp,Fst);
Hd3 = design(Hf,'equiripple','StopbandShape','linear',...
    'StopbandDecay',53.333,'systemobject',true);
setspecs(Hf,'Fp,Fst,Ap,Ast',Fp,Fst,Ap,Ast);
Hd4 = design(Hf,'equiripple','StopbandShape','linear',...
    'StopbandDecay',53.333,'minphase',true,'systemobject',true);
hfvt2 = fvtool(Hd3,Hd4,'Color','White');
legend(hfvt2,'Linear-phase equiripple design with linearly decaying stopband',...
    'Minimum-phase equiripple design with linearly decaying stopband')
```



Minimum-Order Lowpass Filter Design Using Multistage Techniques

A different approach to minimizing the number of coefficients that does not involve minimum-phase designs is to use multistage techniques. Here we show an interpolated FIR (IFIR) approach.

```
Hd5 = ifir(Hf);
hfvt3 = fvtool(Hd1,Hd5,'Color','White');
legend(hfvt3,'Linear-phase equirriple design',...
      'Linear-phase IFIR design')
```



The number of nonzero coefficients required in the IFIR case is 111. Less than both the equiripple linear-phase and minimum-phase designs.

Filter Designer: A Filter Design and Analysis App

- “Using Filter Designer” on page 22-2
- “Importing a Filter Design” on page 22-24

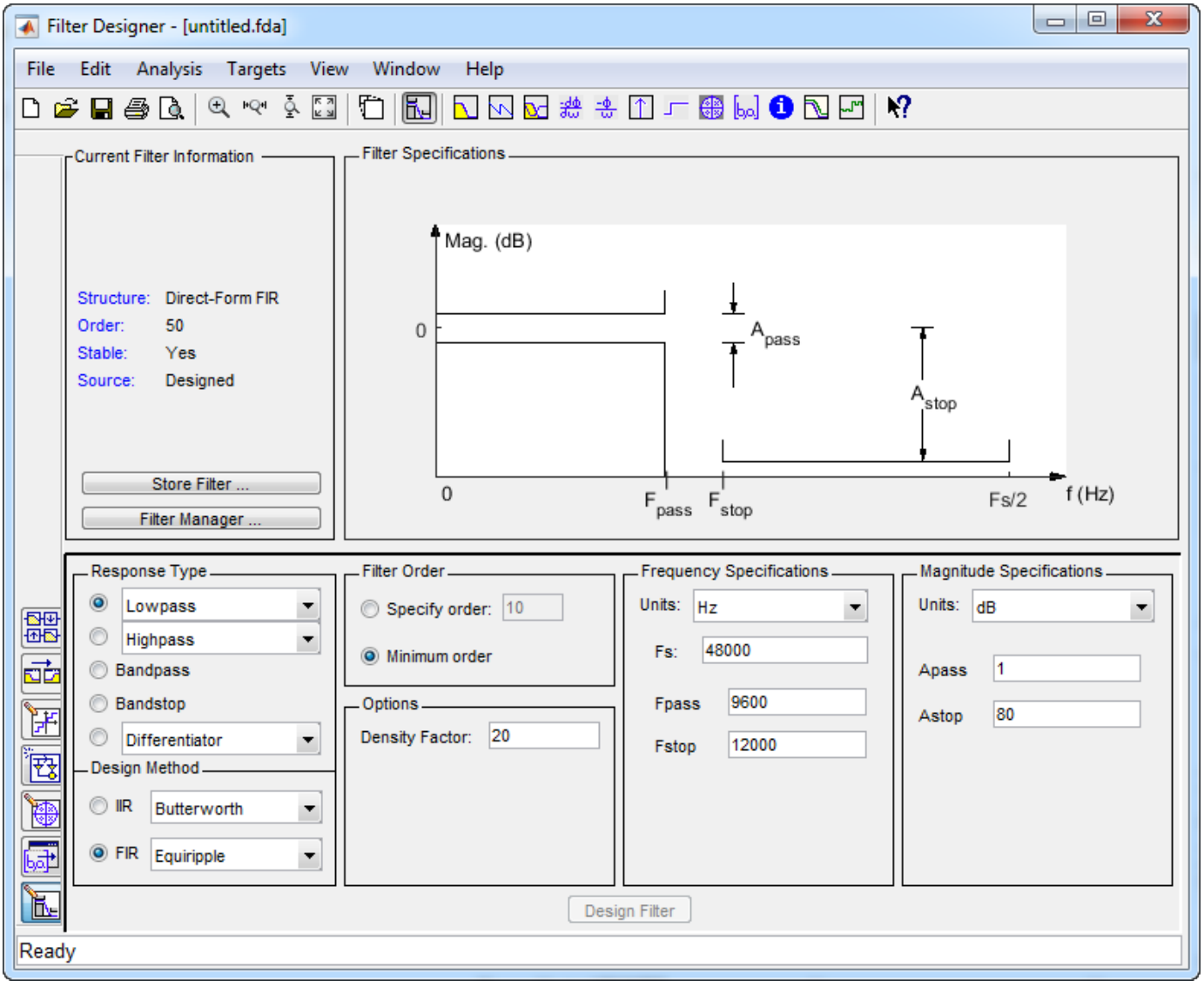
Using Filter Designer

To open filter designer, type

```
filterDesigner
```

at the MATLAB command prompt.

The filter designer opens with the `Design filter` panel displayed.



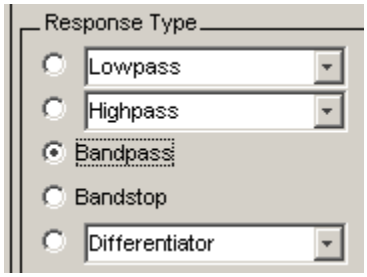
Note that when you open filter designer, **Design Filter** is not enabled. You must make a change to the default filter design in order to enable **Design Filter**. This is true for each time you want to change the filter design. Changes to radio button items or drop down menu items such as those under **Response Type** or **Filter Order** enable **Design Filter** immediately. Changes to specifications in text boxes such as **Fs**, **Fpass**, and **Fstop** require you to click outside the text box to enable **Design Filter**.

Choosing a Response Type

You can choose from several response types:

- Lowpass
- Raised cosine
- Highpass
- Bandpass
- Bandstop
- Differentiator
- Multiband
- Hilbert transformer
- Arbitrary magnitude
- Arbitrary Group Delay
- Peaking
- Notching

To design a bandpass filter, select the radio button next to **Bandpass** in the Response Type region of the app.

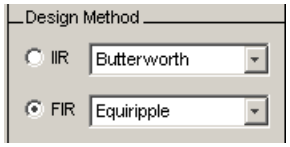


Note Not all filter design methods are available for all response types. Once you choose your response type, this may restrict the filter design methods available to you. Filter design methods that are not available for a selected response type are removed from the Design Method region of the app.

Choosing a Filter Design Method

You can use the default filter design method for the response type that you've selected, or you can select a filter design method from the available FIR and IIR methods listed in the app.

To select the Remez algorithm to compute FIR filter coefficients, select the **FIR** radio button and choose `Equiripple` from the list of methods.



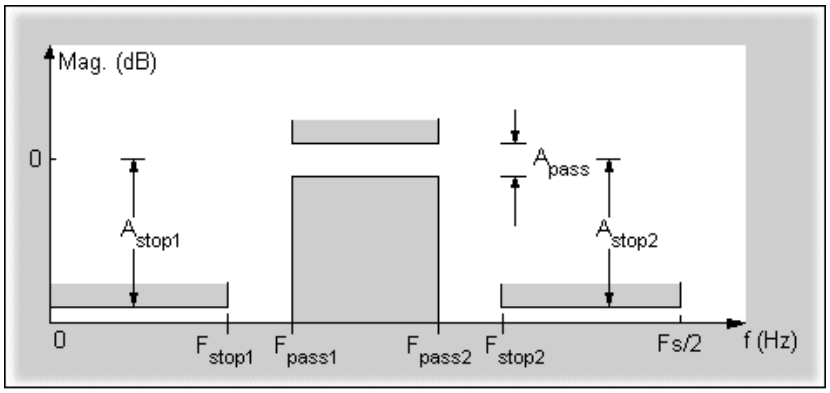
Setting the Filter Design Specifications

- “Viewing Filter Specifications” on page 22-4
- “Filter Order” on page 22-4
- “Options” on page 22-5
- “Bandpass Filter Frequency Specifications” on page 22-5
- “Bandpass Filter Magnitude Specifications” on page 22-6

Viewing Filter Specifications

The filter design specifications that you can set vary according to response type and design method. The display region illustrates filter specifications when you select **Analysis > Filter Specifications** or when you click the **Filter Specifications** toolbar button.

You can also view the filter specifications on the Magnitude plot of a designed filter by selecting **View > Specification Mask**.

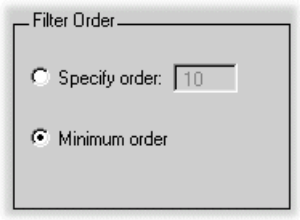


Filter Order

You have two mutually exclusive options for determining the filter order when you design an equiripple filter:

- **Specify order**: You enter the filter order in a text box.
- **Minimum order**: The filter design method determines the minimum order filter.

Select the **Minimum order** radio button for this example.



Note that filter order specification options depend on the filter design method you choose. Some filter methods may not have both options available.

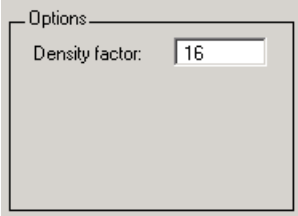
Options

The available options depend on the selected filter design method. Only the FIR Equiripple and FIR Window design methods have settable options. For FIR Equiripple, the option is a **Density Factor**. See `firpm` for more information. For FIR Window the options are **Scale Passband**, **Window** selection, and for the following windows, a settable parameter:

Window	Parameter
Chebyshev (`chebwin`)	Sidelobe attenuation
Gaussian (`gausswin`)	Alpha
Kaiser (`kaiser`)	Beta
Taylor (`taylorwin`)	Nbar and Sidelobe level
Tukey (`tukeywin`)	Alpha
User Defined	Function Name, Parameter

You can view the window in the Window Visualization Tool (`WVTool`) by clicking the **View** button.

For this example, set the **Density factor** to 16.



Bandpass Filter Frequency Specifications

For a bandpass filter, you can set

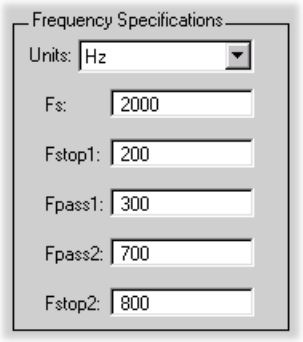
- Units of frequency:
  - Hz
  - kHz
  - MHz
  - GHz
  - Normalized (0 to 1)
- Sampling frequency
- Passband frequencies
- Stopband frequencies

You specify the passband with two frequencies. The first frequency determines the lower edge of the passband, and the second frequency determines the upper edge of the passband.

Similarly, you specify the stopband with two frequencies. The first frequency determines the upper edge of the first stopband, and the second frequency determines the lower edge of the second stopband.

For this example:

- Keep the units in **Hz** (default).
- Set the sampling frequency (**Fs**) to `2000` Hz.
- Set the end of the first stopband (**Fstop1**) to `200` Hz.
- Set the beginning of the passband (**Fpass1**) to `300` Hz.
- Set the end of the passband (**Fpass2**) to `700` Hz.
- Set the beginning of the second stopband (**Fstop2**) to `800` Hz.



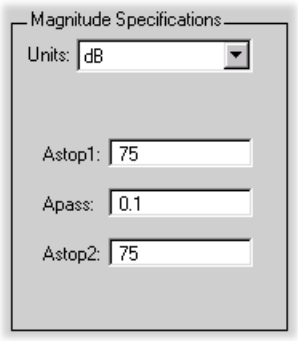
Bandpass Filter Magnitude Specifications

For a bandpass filter, you can specify the following magnitude response characteristics:

- Units for the magnitude response (dB or linear)
- Passband ripple
- Stopband attenuation

For this example:

- Keep **Units** in `dB` (default).
- Set the passband ripple (**Apass**) to `0.1` dB.
- Set the stopband attenuation for both stopbands (**Astop1**, **Astop2**) to `75` dB.



Computing the Filter Coefficients

Now that you've specified the filter design, click the **Design Filter** button to compute the filter coefficients.

Notice that the Design Filter button is disabled once you've computed the coefficients for your filter design. This button is enabled again once you make any changes to the filter specifications.

Analyzing the Filter

- “Displaying Filter Responses” on page 22-7
- “Using Data Tips” on page 22-8
- “Drawing Spectral Masks” on page 22-9
- “Changing the Sampling Frequency” on page 22-10
- “Displaying the Response in FVTool” on page 22-10

Displaying Filter Responses

You can view the following filter response characteristics in the display region or in a separate window.

- Magnitude response
- Phase response
- Magnitude and Phase responses
- Group delay response
- Phase delay response
- Impulse response
- Step response
- Pole-zero plot
- Zero-phase response — available from the *y*-axis context menu in a Magnitude or Magnitude and Phase response plot.
- Magnitude Response Estimate
- Round-off Noise Power Spectrum

The **Magnitude Response Estimate** and **Round-off Noise Power Spectrum** analyses use filter internals.

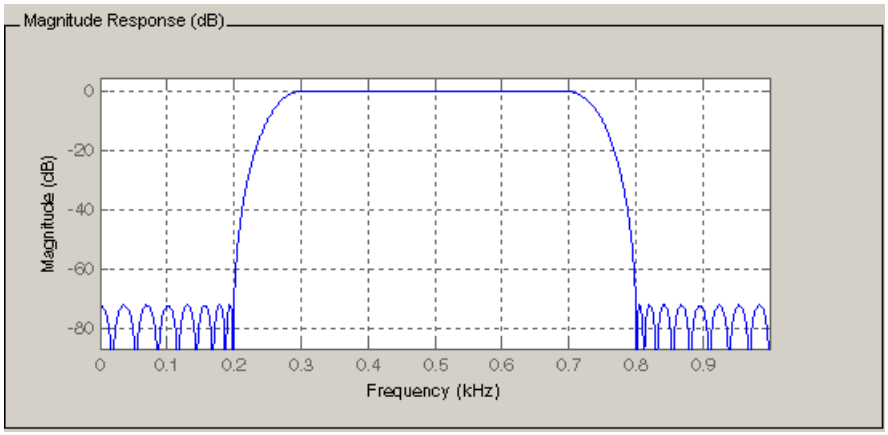
For descriptions of the above responses and their associated toolbar buttons and other filter designer toolbar buttons, see `FVTool`.

You can display two responses in the same plot by selecting **Analysis > Overlay Analysis** and selecting an available response. A second *y*-axis is added to the right side of the response plot. (Note that not all responses can be overlaid on each other.)

You can also display the filter coefficients and detailed filter information in this region.

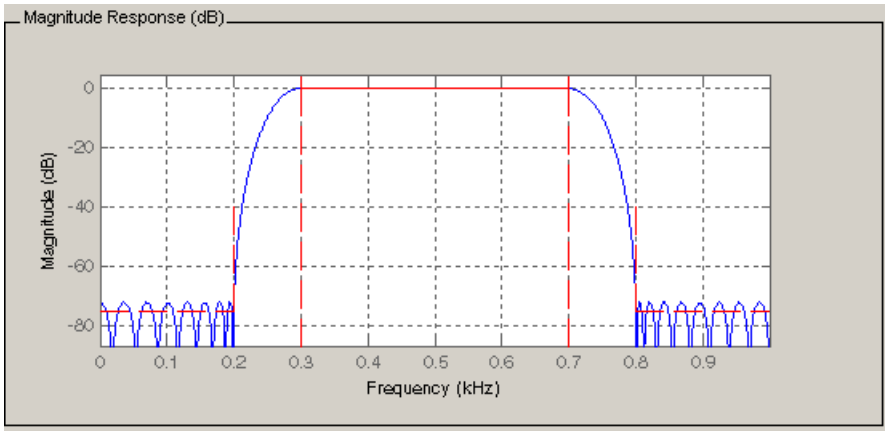
For all the analysis methods, except zero-phase response, you can access them from the **Analysis** menu, the Analysis Parameters dialog box from the context menu, or by using the toolbar buttons. For zero-phase, right-click the *y*-axis of the plot and select **Zero-phase** from the context menu.

For example, to look at the filter's magnitude response, select the **Magnitude Response** button on the toolbar.



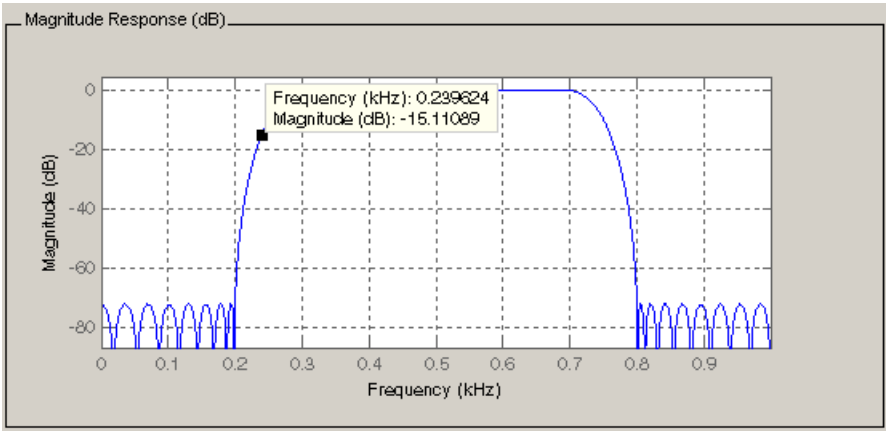
You can also overlay the filter specifications on the Magnitude plot by selecting **View > Specification Mask**.

Note You can use specification masks in FVTool only if FVTool was launched from filter designer.



Using Data Tips

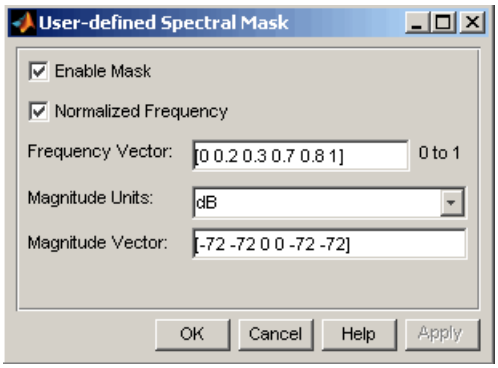
You can click the response to add plot data tips that display information about particular points on the response.



For information on using data tips, see “Interactively Explore Plotted Data” (MATLAB).

Drawing Spectral Masks

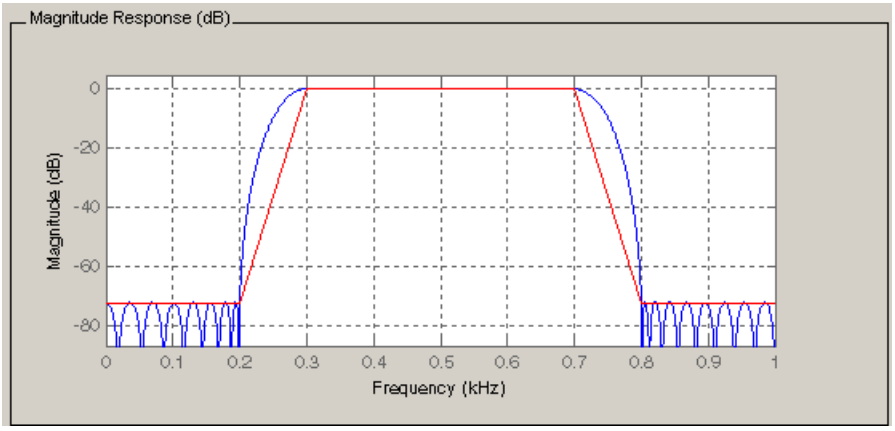
To add spectral masks or rejection area lines to your magnitude plot, click **View > User-defined Spectral Mask**.



The mask is defined by a frequency vector and a magnitude vector. These vectors must be the same length.

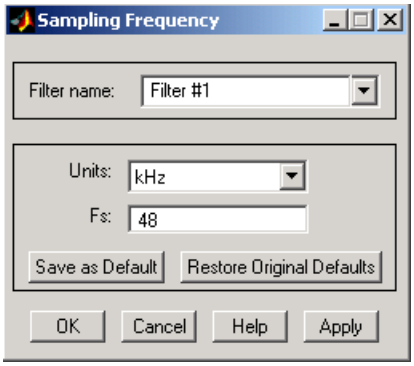
- `Enable Mask` — Select to turn on the mask display.
- `Normalized Frequency` — Select to normalize the frequency between 0 and 1 across the displayed frequency range.
- `Frequency Vector` — Enter a vector of *x*-axis frequency values.
- `Magnitude Units` — Select the desired magnitude units. These units should match the units used in the magnitude plot.
- `Magnitude Vector` — Enter a vector of *y*-axis magnitude values.

The magnitude response below shows a spectral mask.



Changing the Sampling Frequency

To change the sampling frequency of your filter, right-click any filter response plot and select **Sampling Frequency** from the context menu.



To change the filter name, type the new name in **Filter name**. (In `FVTool`, if you have multiple filters, select the desired filter and then enter the new name.)

To change the sampling frequency, select the desired unit from **Units** and enter the sampling frequency in **Fs**. (For each filter in `fvtool`, you can specify a different sampling frequency or you can apply the sampling frequency to all filters.)

To save the displayed parameters as the default values to use when filter designer or FVTool is opened, click **Save as Default**.

To restore the default values, click **Restore Original Defaults**.

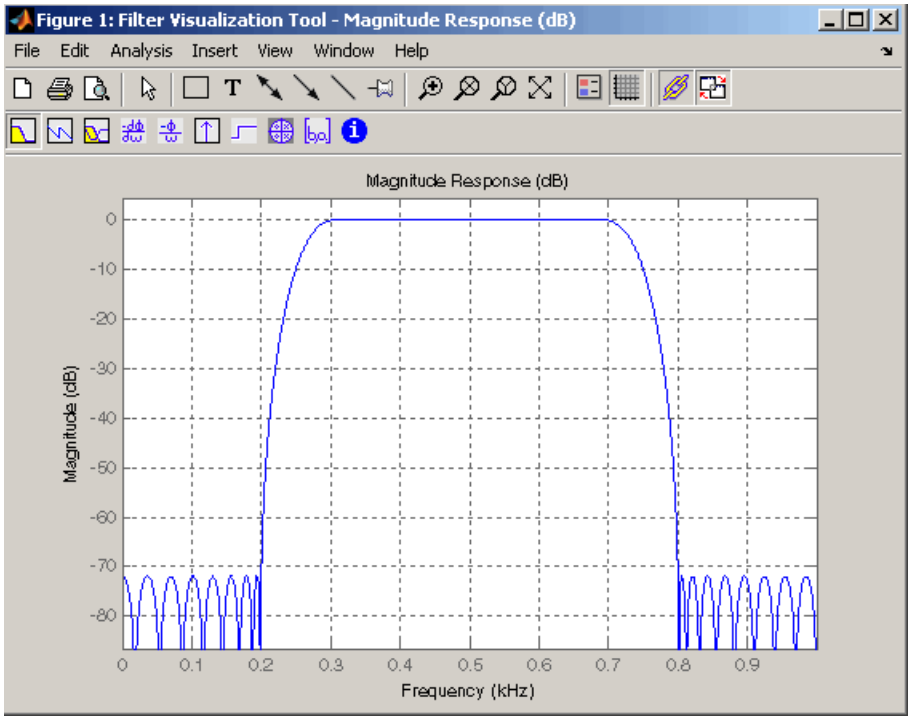
Displaying the Response in FVTool

To display the filter response characteristics in a separate window, select **View > Filter Visualization Tool** (available if any analysis, except the filter specifications, is in the display region) or click the **Full View Analysis** button:

This launches the Filter Visualization Tool (`fvtool`).

Note If Filter Specifications are shown in the display region, clicking the **Full View Analysis** toolbar button launches a MATLAB figure window instead of FVTool. For details, see "Add Annotations to Chart" (MATLAB). The associated menu item is **Print to figure**, which is enabled only if the filter specifications are displayed.

You can use this tool to annotate your design, view other filter characteristics, and print your filter response. You can link filter designer and fvtool so that changes made in filter designer are immediately reflected in fvtool. See `FVTool` for more information.



Editing the Filter Using the Pole/Zero Editor

- "Displaying the Pole-Zero Plot" on page 22-11
- "Changing the Pole-Zero Plot" on page 22-12

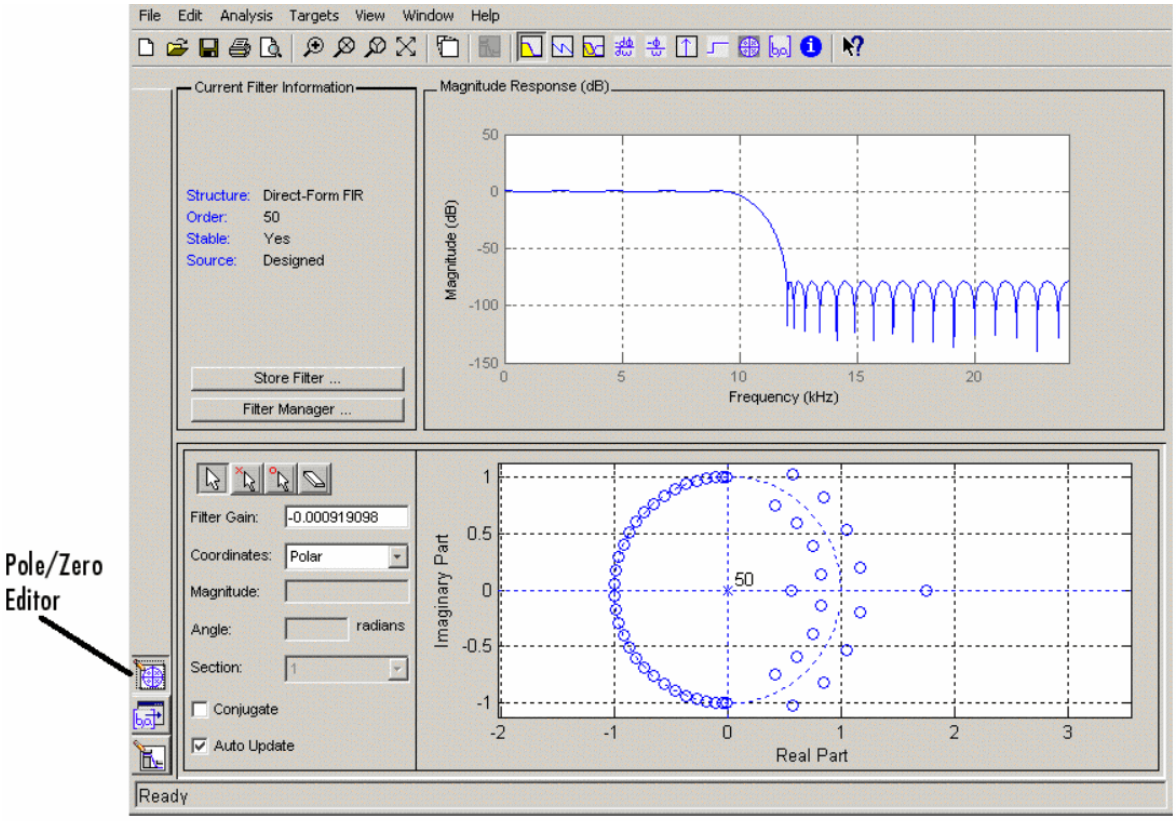
Displaying the Pole-Zero Plot

You can edit a designed or imported filter's coefficients by moving, deleting, or adding poles and/or zeros using the Pole/Zero Editor panel.

Note You cannot generate MATLAB code (**File > Generate MATLAB code**) if your filter was designed or edited with the Pole/Zero Editor.

You cannot move quantized poles and zeros. You can only move the reference poles and zeros.

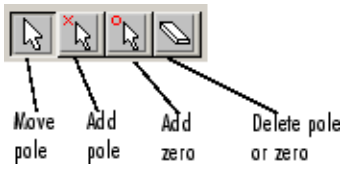
Click the **Pole/Zero Editor** button in the sidebar or select **Edit > Pole/Zero Editor** to display this panel.



Poles are shown using x symbols and zeros are shown using o symbols.

Changing the Pole-Zero Plot

Plot mode buttons are located to the left of the pole/zero plot. Select one of the buttons to change the mode of the pole/zero plot. The Pole/Zero Editor has these buttons from left to right: move pole, add pole, add zero, and delete pole or zero.

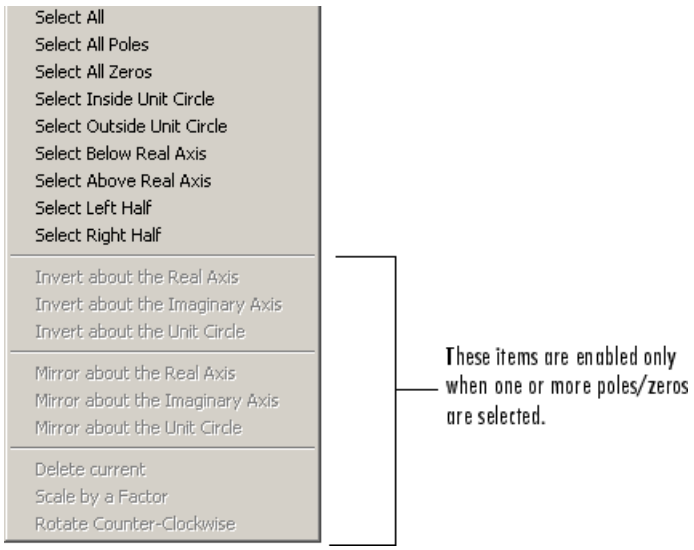


The following plot parameters and controls are located to the left of the pole/zero plot and below the plot mode buttons.

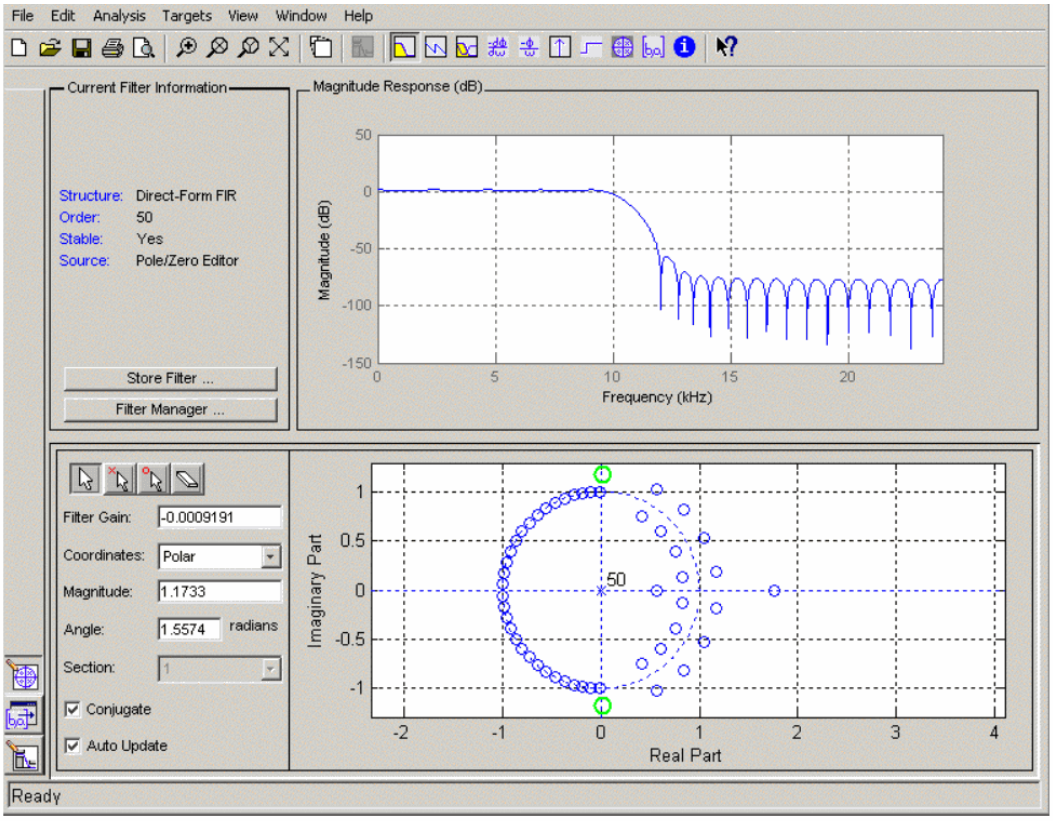
- **Gain** — factor to compensate for the filter's pole(s) and zero(s) gains
- **Coordinates** — units (`Polar` or `Rectangular`) of the selected pole or zero
- **Magnitude** — if polar coordinates is selected, magnitude of the selected pole or zero
- **Angle** — if polar coordinates is selected, angle of selected pole(s) or zero(s)
- **Real** — if rectangular coordinates is selected, real component of selected pole(s) or zero(s)
- **Imaginary** — if rectangular coordinates is selected, imaginary component of selected pole or zero

- **Section** — for multisection filters, number of the current section
- **Conjugate** — creates a corresponding conjugate pole or zero or automatically selects the conjugate pole or zero if it already exists.
- **Auto update** — immediately updates the displayed magnitude response when poles or zeros are added, moved, or deleted.

The **Edit > Pole/Zero Editor** has items for selecting multiple poles/zeros, for inverting and mirroring poles/zeros, and for deleting, scaling and rotating poles/zeros.



Moving one of the zeros on the vertical axis produces the following result:



- The selected zero pair is shown in green.
- When you select one of the zeros from a conjugate pair, the Conjugate check box and the conjugate are automatically selected.
- The Magnitude Response plot updates immediately because **Auto update** is active.

Converting the Filter Structure

- “Converting to a New Structure” on page 22-14
- “Converting to Second-Order Sections” on page 22-15

Converting to a New Structure

You can use **Edit > Convert Structure** to convert the current filter to a new structure. All filters can be converted to the following representations:

- Direct-form I
- Direct-form II
- Direct-form I transposed
- Direct-form II transposed
- Lattice ARMA

Note If you have DSP System Toolbox product installed, you will see additional structures in the Convert structure dialog box.

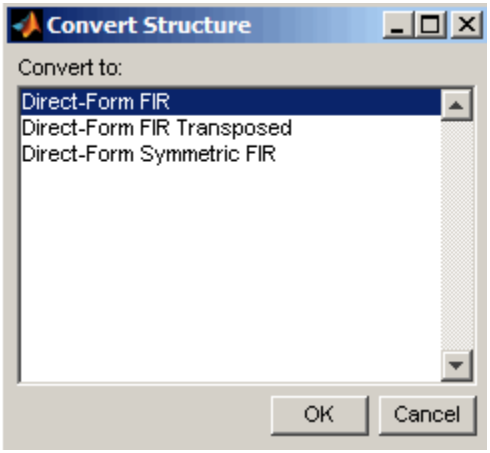
In addition, the following conversions are available for particular classes of filters:

- Minimum phase FIR filters can be converted to Lattice minimum phase
- Maximum phase FIR filters can be converted to Lattice maximum phase
- Allpass filters can be converted to Lattice allpass
- IIR filters can be converted to Lattice ARMA

Note Converting from one filter structure to another may produce a result with different characteristics than the original. This is due to the computer's finite-precision arithmetic and the variations in the conversion's roundoff computations.

For example:

- Select **Edit > Convert Structure** to open the Convert structure dialog box.
- Select `Direct-form I` in the list of filter structures.



Converting to Second-Order Sections

You can use **Edit > Convert to Second-Order Sections** to store the converted filter structure as a collection of second-order sections rather than as a monolithic higher-order structure.

Note The following options are also used for **Edit > Reorder and Scale Second-Order Sections**, which you use to modify an SOS filter structure.

The following **Scale** options are available when converting a direct-form II structure only:

- `None` (default)
- `L-2` (L^2 norm)
- `L-infinity` (L^∞ norm)

The **Direction** (`Up` or `Down`) determines the ordering of the second-order sections. The optimal ordering changes depending on the **Scale** option selected.

For example:

- Select **Edit** > **Convert to Second-Order Sections** to open the Convert to SOS dialog box.
- Select `L-infinity` from the **Scale** menu for L^∞ norm scaling.
- Leave `Up` as the **Direction** option.

Note To convert from second-order sections back to a single section, use **Edit** > **Convert to Single Section**.

Exporting a Filter Design

- "Exporting Coefficients or Objects to the Workspace" on page 22-16
- "Exporting Coefficients to an ASCII File" on page 22-16
- "Exporting Coefficients or Objects to a MAT-File" on page 22-16
- "Exporting to a Simulink Model" on page 22-17
- "Other Ways to Export a Filter" on page 22-19

Exporting Coefficients or Objects to the Workspace

You can save the filter either as filter coefficients variables or as a filter System object variable. To save the filter to the MATLAB workspace:

1. Select **File** > **Export**. The Export dialog box appears.
2. Select `Workspace` from the **Export To** menu.
3. Select `Coefficients` from the **Export As** menu to save the filter coefficients or select `System Objects` to save the filter in a filter System object.
4. For coefficients, assign variable names using the **Numerator** (for FIR filters) or **Numerator** and **Denominator** (for IIR filters), or **SOS Matrix** and **Scale Values** (for IIR filters in second-order section form) text boxes in the Variable Names region.

   For System objects, assign the variable name in the **Discrete Filter (**or **Quantized Filter)** text box. If you have variables with the same names in your workspace and you want to overwrite them, select the **Overwrite Variables** check box.
5. Click the **Export** button.

Exporting Coefficients to an ASCII File

To save filter coefficients to a text file,

1. Select **File** > **Export**. The Export dialog box appears.
2. Select `Coefficients File (ASCII)` from the **Export To** menu.
3. Click the **Export** button. The Export Filter Coefficients to .FCF File dialog box appears.
4. Choose or enter a filename and click the **Save** button.

The coefficients are saved in the text file that you specified, and the MATLAB Editor opens to display the file. The text file also contains comments with the MATLAB version number, the Signal Processing Toolbox version number, and filter information.

Exporting Coefficients or Objects to a MAT-File

To save filter coefficients or a filter object as variables in a MAT-file:

1. Select **File** > **Export**. The Export dialog box appears.
2. Select `MAT-file` from the **Export To** menu.
3. Select `Coefficients` from the **Export As** menu to save the filter coefficients or select `Objects` to save the filter in a filter object.
4. For coefficients, assign variable names using the **Numerator** (for FIR filters) or **Numerator** and **Denominator** (for IIR filters), or **SOS Matrix** and **Scale Values** (for IIR filters in second-order section form) text boxes in the Variable Names region.

   For objects, assign the variable name in the **Discrete Filter (**or **Quantized Filter)** text box. If you have variables with the same names in your workspace and you want to overwrite them, select the **Overwrite Variables** check box.
5. Click the **Export** button. The Export to a MAT-File dialog box appears.
6. Choose or enter a filename and click the **Save** button.

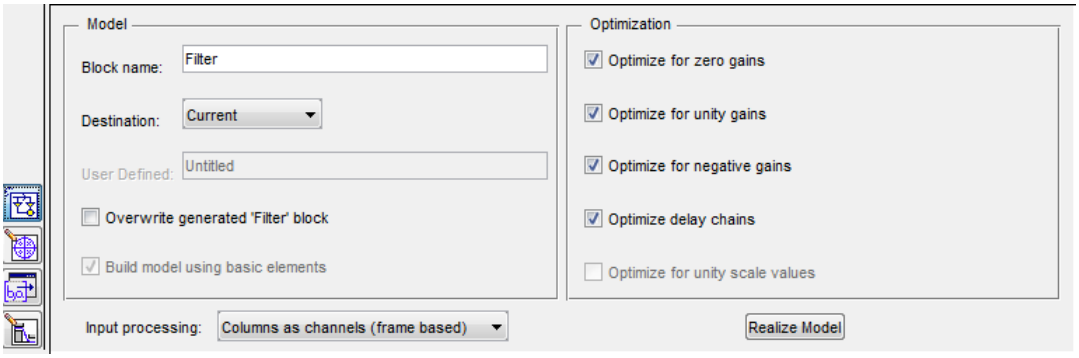
Exporting to a Simulink Model

If you have the Simulink product installed, you can export a Simulink block of your filter design and insert it into a new or existing Simulink model.

You can export a filter designed using any filter design method available in the filter designer app.

Note If you have the DSP System Toolbox and Fixed-Point Designer installed, you can export a CIC filter to a Simulink model.

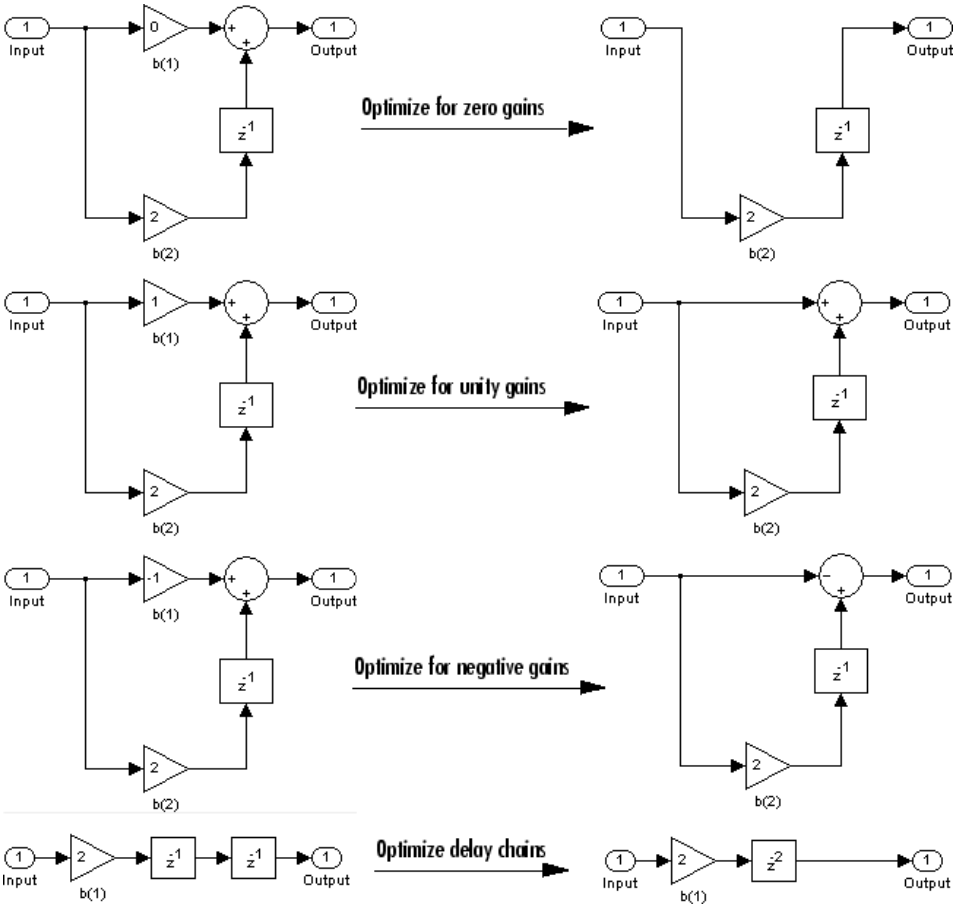
1. After designing your filter, click the **Realize Model** sidebar button or select **File** > **Export to Simulink Model**. The Realize Model panel is displayed.



2. Specify the name to use for your block in **Block name**.
3. To insert the block into the current (most recently selected) Simulink model, set the **Destination** to `Current`. To inset the block into a new model, select `New`. To insert the block into a user-defined subsystem, select `User defined`.
4. If you want to overwrite a block previously created from this panel, check **Overwrite generated `Filter' block**.
5. If you select the **Build model using basic elements** check box, your filter is created as a subsystem (Simulink) block, which uses separate sub-elements. In this mode, the following optimization(s) are available:

- `Optimize for zero gains` — Removes zero-valued gain paths from the filter structure.
- `Optimize for unity gains` — Substitutes a wire (short circuit) for gains equal to 1 in the filter structure.
- `Optimize for negative gains` — Substitutes a wire (short circuit) for gains equal to -1 and changes corresponding additions to subtractions in the filter structure.
- `Optimize delay chains` — Substitutes delay chains composed of `n` unit delays with a single delay of `n`.
- `Optimize for unity scale values` — Removes multiplications for scale values equal to 1 from the filter structure.

The following illustration shows the effects of some of the optimizations:



Optimization Effects

Note The **Build model using basic elements** check box is enabled only when you have a DSP System Toolbox license and your filter can be designed using digital filter blocks from that library. For more information, see the Filter Realization Wizard.

6 Set the **Input processing** parameter to specify whether the generated filter performs sample- or frame-based processing on the input. Depending on the type of filter you design, one or both of the following options may be available:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

7 Click the **Realize Model** button to create the filter block. When the **Build model using basic elements** check box is selected, filter designer implements the filter as a subsystem block using Sum, Gain, and Delay blocks.

If you double-click the Simulink Filter block, the filter structure is displayed.

Other Ways to Export a Filter

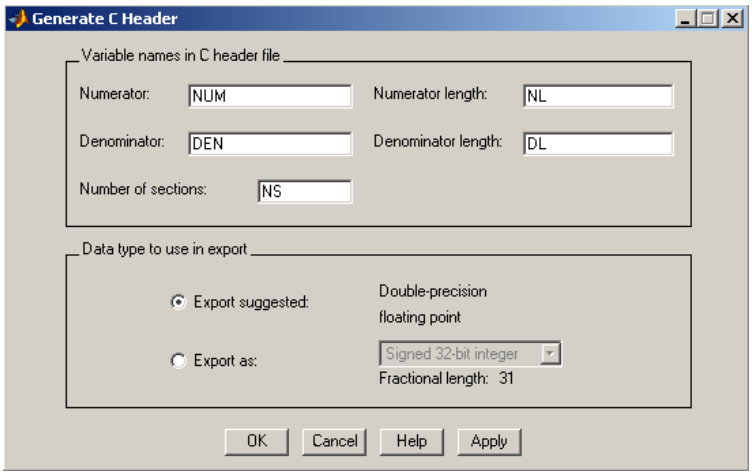
You can also send your filter to a C header file or generate MATLAB code to construct your filter from the command line. For detailed instructions, see the following sections:

- “Generating a C Header File” on page 22-19
- “Generating MATLAB Code” on page 22-20

Generating a C Header File

You may want to include filter information in an external C program. To create a C header file with variables that contain filter parameter data, follow this procedure:

1 Select **Targets** > **Generate C Header**. The Generate C Header dialog box appears.



2 Enter the variable names to be used in the C header file. The particular filter structure determines the variables that are created in the file

Filter Structure	Variable Parameter
Direct-form I Direct-form II Direct-form I transposed Direct-form II transposed	Numerator, Numerator length*, Denominator, Denominator length*, and Number of sections (inactive if filter has only one section)

Filter Structure	Variable Parameter
Lattice ARMA	Lattice coeffs, Lattice coeffs length*, Ladder coeffs, Ladder coeffs length*, Number of sections (inactive if filter has only one section)
Lattice MA	Lattice coeffs, Lattice coeffs length*, and Number of sections (inactive if filter has only one section)
Direct-form FIR Direct-form FIR transposed	Numerator, Numerator length*, and Number of sections (inactive if filter has only one section)

***length** variables contain the total number of coefficients of that type.

Note Variable names cannot be C language reserved words, such as “for.”

3 Select **Export Suggested** to use the suggested data type or select **Export As** and select the desired data type from the pull-down.

Note If you do not have DSP System Toolbox software installed, selecting any data type other than double-precision floating point results in a filter that does not exactly match the one you designed in the filter designer. This is due to rounding and truncating differences.

4 Click **OK** to save the file and close the dialog box or click **Apply** to save the file, but leave the dialog box open for additional C header file definitions.

Generating MATLAB Code

You can generate MATLAB code that constructs the filter you designed in filter designer from the command line. Select **File > Generate MATLAB Code > Filter Design Function** and specify the filename in the Generate MATLAB code dialog box.

Note You cannot generate MATLAB code through **File > Generate MATLAB Code > Filter Design Function (with System Objects)** or through **File > Generate MATLAB Code > Data Filtering Function (with System Objects)**, if your filter was designed or edited with the Pole/Zero Editor.

The following is generated MATLAB code when you choose **File > Generate MATLAB Code > Data Filtering Function (with System Objects)** for the equiripple bandpass filter designed in this example.

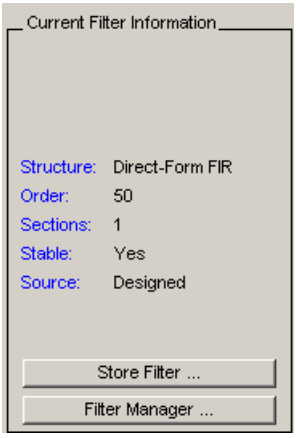
```
function Hd = ExFilter
%EXFILTER Returns a discrete-time filter object.

% MATLAB Code
% Generated by MATLAB(R) 9.1 and the DSP System Toolbox 9.3.
% Generated on: 17-Nov-2016 14:55:28

% Equiripple Bandpass filter designed using the FIRPM function.

% All frequency values are in Hz.
Fs = 2000;  % Sampling Frequency

Fstop1 = 200;              % First Stopband Frequency
Fpass1 = 300;              % First Passband Frequency
Fpass2 = 700;              % Second Passband Frequency
Fstop2 = 800;              % Second Stopband Frequency
Dstop1 = 0.000177827941;   % First Stopband Attenuation
Dpass  = 0.0057563991496;  % Passband Ripple
Dstop2 = 0.000177827941;   % Second Stopband Attenuation
```

```
dens   = 16;               % Density Factor

% Calculate the order from the parameters using FIRPMORD.
[N, Fo, Ao, W] = firpmord([Fstop1 Fpass1 Fpass2 Fstop2]/(Fs/2), [0 1 ...
    0], [Dstop1 Dpass Dstop2]);

% Calculate the coefficients using the FIRPM function.
b  = firpm(N, Fo, Ao, W, {dens});
Hd = dsp.FIRFilter( ...
    'Numerator', b);

% [EOF]
```

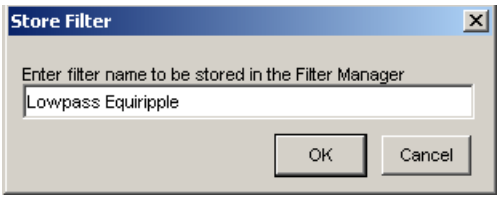
Managing Filters in the Current Session

You can store filters designed in the current filter designer session for cascading together, exporting to FVTool or for recalling later in the same or future filter designer sessions.

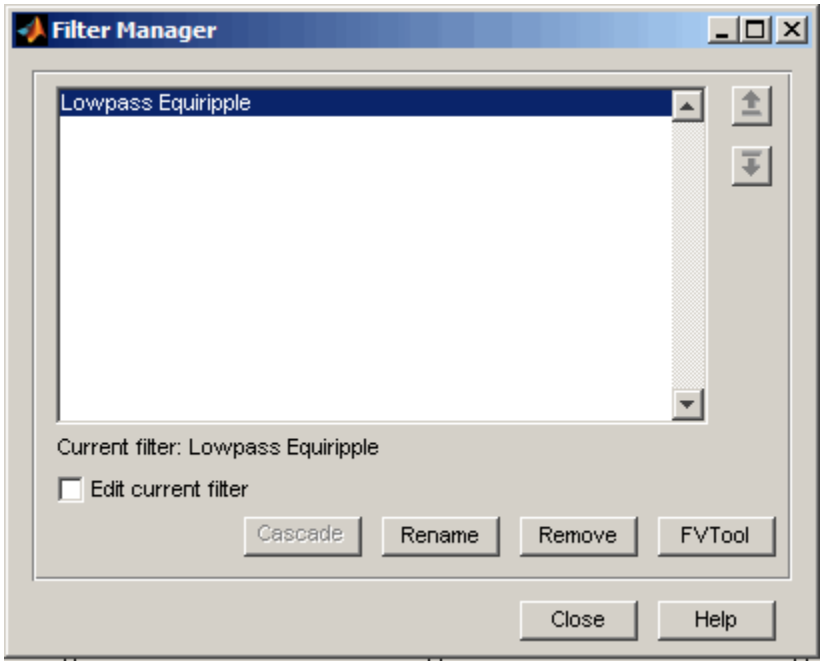
You store and access saved filters with the **Store filter** and **Filter Manager** buttons, respectively, in the Current Filter Information pane.



Store Filter — Displays the Store Filter dialog box in which you specify the filter name to use when storing the filter in the Filter Manager. The default name is the type of the filter.



Filter Manager — Opens the Filter Manager.



The current filter is listed below the listbox. To change the current filter, highlight the desired filter. If you select **Edit current filter**, filter designer displays the currently selected filter specifications. If you make any changes to the specifications, the stored filter is updated immediately.

To cascade two or more filters, highlight the desired filters and press **Cascade**. A new cascaded filter is added to the Filter Manager.

To change the name of a stored filter, press **Rename**. The Rename filter dialog box is displayed.

To remove a stored filter from the Filter Manager, press **Delete**.

To export one or more filters to FVTool, highlight the filter(s) and press **FVTool**.

Saving and Opening Filter Design Sessions

You can save your filter design session as a MAT-file and return to the same session another time.

Select the **Save session** button to save your session as a MAT-file. The first time you save a session, a Save Filter Design Session browser opens, prompting you for a session name.

For example, save this design session as `TestFilter.fda` in your current working directory by typing `TestFilter` in the **File name** field.

The `.fda` extension is added automatically to all filter design sessions you save.

Note You can also use the **File > Save session** and **File > Save session as** to save a session.

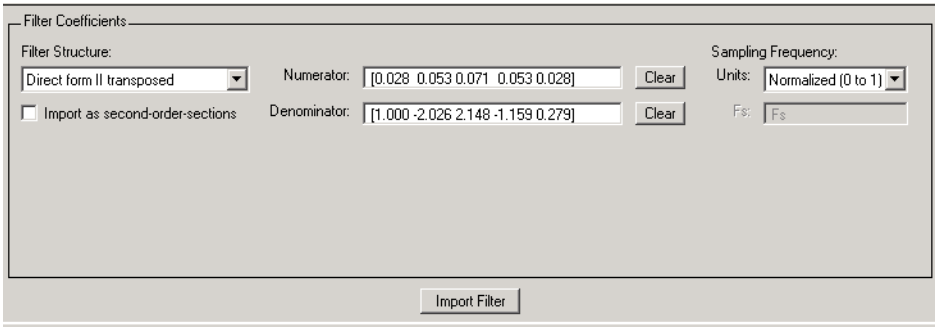
You can load existing sessions into the Filter Design and Analysis Tool by selecting the **Open session** button, or **File > Open session** . A Load Filter Design Session browser opens that allows you to select from your previously saved filter design sessions.

Importing a Filter Design

In this section...
“Import Filter Panel” on page 22-24 “Filter Structures” on page 22-24

Import Filter Panel

The Import Filter panel allows you to import a filter. You can access this region by clicking the **Import Filter** button in the sidebar.



The imported filter can be in any of the representations listed in the **Filter Structure** pull-down menu. You can import a filter as second-order sections by selecting the check box.

Specify the filter coefficients in **Numerator** and **Denominator**, either by entering them explicitly or by referring to variables in the MATLAB workspace.

Select the frequency units from the following options in the **Units** menu, and for any frequency unit other than Normalized, specify the value or MATLAB workspace variable of the sampling frequency in the **Fs** field.

To import the filter, click the **Import Filter** button. The display region is automatically updated when the new filter has been imported.

You can edit the imported filter using the Pole/Zero Editor panel.

Filter Structures

The available filter structures are:

- Direct Form, which includes direct-form I, direct-form II, direct-form I transposed, direct-form II transposed, and direct-form FIR
- Lattice, which includes lattice allpass, lattice MA min phase, lattice MA max phase, and lattice ARMA

The structure that you choose determines the type of coefficients that you need to specify in the text fields to the right.

Direct-form

For direct-form I, direct-form II, direct-form I transposed, and direct-form II transposed, specify the filter by its transfer function representation

$$H(z) = \frac{b(1) + b(2)z^{-1} + b(3)z^{-2} + ...b(m+1)z^{-m}}{a(1) + a(2)z^{-1} + a(3)Z^{-3} + ...a(n+1)z^{-n}}$$

- The **Numerator** field specifies a variable name or value for the numerator coefficient vector `b`, which contains `m+1` coefficients in descending powers of z.
- The **Denominator** field specifies a variable name or value for the denominator coefficient vector `a`, which contains `n+1` coefficients in descending powers of z. For FIR filters, the **Denominator** is `1`.

Filters in transfer function form can be produced by all of the Signal Processing Toolbox filter design functions (such as `fir1`, `fir2`, `firpm`, `butter`, `yulewalk`). See “Transfer Function” (Signal Processing Toolbox) for more information.

Importing as second-order sections

For all direct-form structures, except direct-form FIR, you can import the filter in its second-order section representation:

$$H(z) = G \prod_{k=1}^{L} \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2}}{a_{0k} + a_{1k}z^{-1} + a_{2k}z^{-2}}$$

The **Gain** field specifies a variable name or a value for the gain G, and the **SOS Matrix** field specifies a variable name or a value for the L-by-6 SOS matrix

$$SOS = \begin{pmatrix} b_{01} & b_{11} & b_{21} & 1 & a_{11} & a_{22} \\ b_{02} & b_{12} & b_{22} & 1 & a_{12} & a_{22} \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ b_{0L} & b_{1L} & b_{2L} & 1 & a_{1L} & a_{2L} \end{pmatrix}$$

whose rows contain the numerator and denominator coefficients b_{ik} and a_{ik} of the second-order sections of $H(z)$.

Filters in second-order section form can be produced by functions such as `tf2sos`, `zp2sos`, `ss2sos`, and `sosfilt`. See “Second-Order Sections (SOS)” (Signal Processing Toolbox) for more information.

Lattice

For lattice allpass, lattice minimum and maximum phase, and lattice ARMA filters, specify the filter by its lattice representation:

- For lattice allpass, the **Lattice coeff** field specifies the lattice (reflection) coefficients, `k(1)` to `k(N)`, where `N` is the filter order.
- For lattice MA (minimum or maximum phase), the **Lattice coeff** field specifies the lattice (reflection) coefficients, `k(1)` to `k(N)`, where `N` is the filter order.

- For lattice ARMA, the **Lattice coeff** field specifies the lattice (reflection) coefficients, `k(1)` to `k(N)`, and the **Ladder coeff** field specifies the ladder coefficients, `v(1)` to `v(N+1)`, where `N` is the filter order.

Filters in lattice form can be produced by `tf2latc`. See “Lattice Structure” (Signal Processing Toolbox) for more information.

Designing a Filter in the Filter Builder GUI

Filter Builder Design Process

In this section...
“Introduction to Filter Builder” on page 23-2
“Design a Filter Using Filter Builder” on page 23-2
“Select a Response” on page 23-2
“Select a Specification” on page 23-4
“Select an Algorithm” on page 23-5
“Customize the Algorithm” on page 23-6
“Analyze the Design” on page 23-7
“Realize or Apply the Filter to Input Data” on page 23-7

Introduction to Filter Builder

The `filterBuilder` function provides a graphical interface to the `fdesign` object-object oriented filter design paradigm and is intended to reduce development time during the filter design process. `filterBuilder` uses a specification-centered approach to find the best algorithm for the desired response.

Note `filterBuilder` requires the Signal Processing Toolbox. The functionality of `filterBuilder` is greatly expanded by the DSP System Toolbox. Many of the features described or displayed below are only available if the DSP System Toolbox is installed. You may verify your installation by typing `ver` at the command prompt.

Design a Filter Using Filter Builder

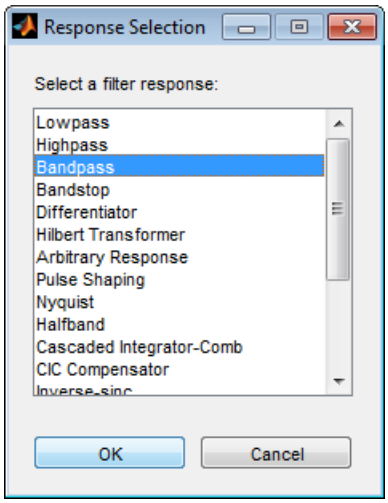
The basic workflow in using `filterBuilder` is to choose the constraints and specifications of the filter, and to use those as a starting point in the design. Postponing the choice of algorithm for the filter allows the best design method to be determined automatically, based upon the desired performance criteria. The following are the details of each of the steps for designing a filter with `filterBuilder`.

Select a Response

When you open the `filterBuilder` tool by typing:

```
filterBuilder
```

at the MATLAB command prompt, the **Response Selection** dialog box appears, listing all possible filter responses available in DSP System Toolbox.

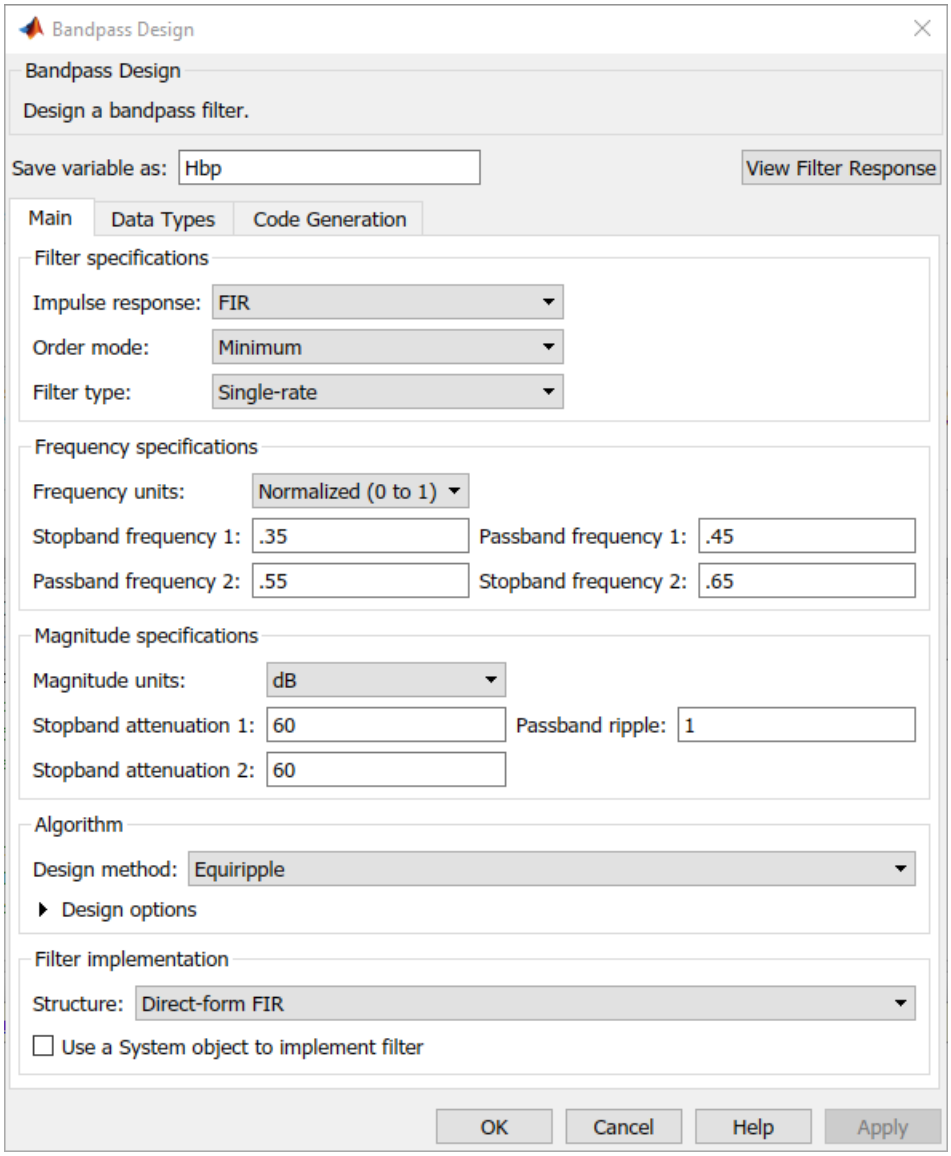


Note This step cannot be skipped because it is not automatically completed for you by the software. You must select a response to initiate the filter design process.

After you choose a response, say bandpass, you start the design of the Specifications Object, and the Bandpass Design dialog box appears. This dialog box contains a **Main** pane, a **Data Types** pane and a **Code Generation** pane. The specifications of your filter are generally set in the **Main** pane of the dialog box.

The **Data Types** pane provides settings for precision and data types, and the **Code Generation** pane contains options for various implementations of the completed filter design.

For the initial design of your filter, you will mostly use the **Main** pane.



The **Bandpass Design** dialog box contains all the parameters you need to determine the specifications of a bandpass filter. The parameters listed in the **Main** pane depend upon the type of filter you are designing. However, no matter what type of filter you have chosen in the **Response Selection** dialog box, the filter design dialog box contains the **Main**, **Data Types**, and **Code Generation** panes.

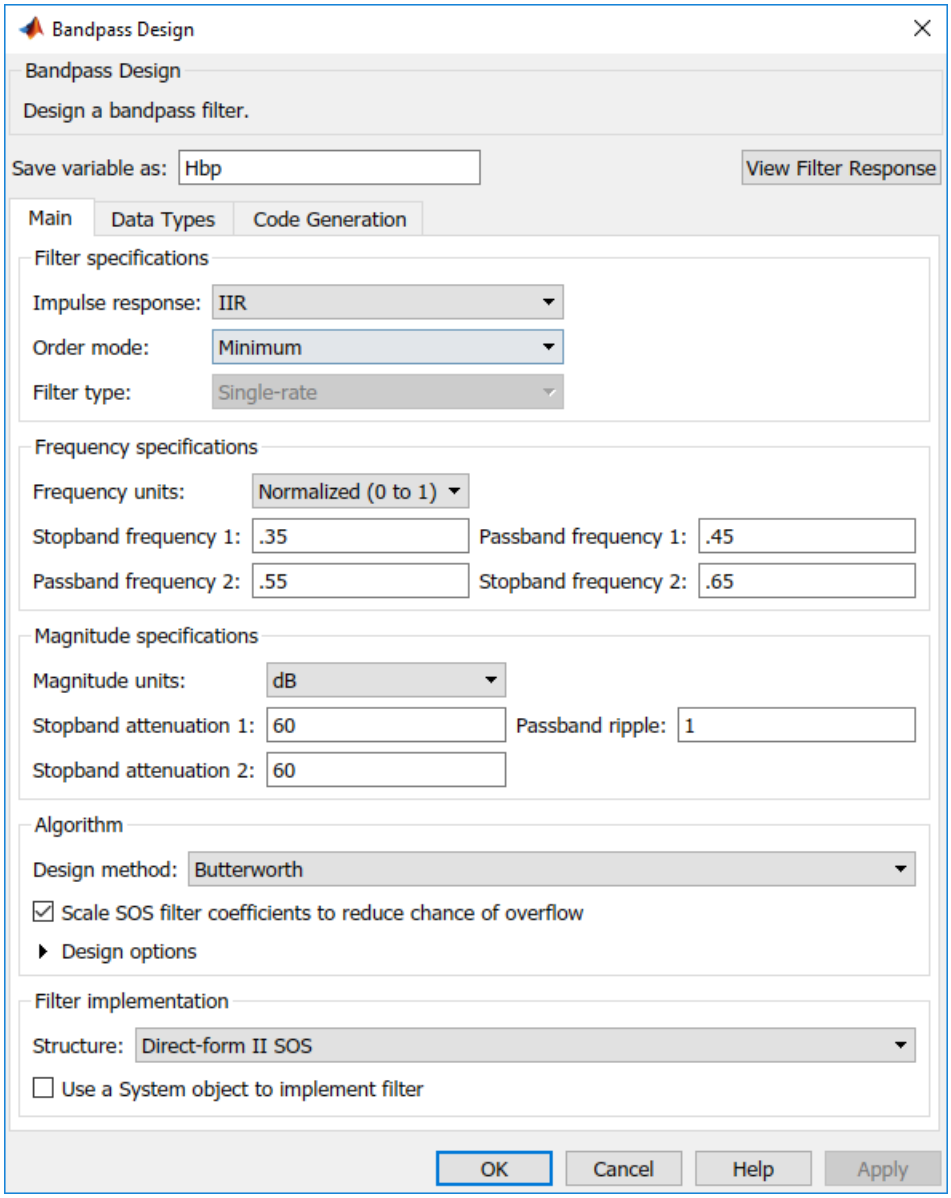
Select a Specification

To choose the specification for the bandpass filter, you can begin by selecting an **Impulse Response**, **Order Mode**, and **Filter Type** in the **Filter Specifications** frame of the **Main Pane**. You can further specify the response of your filter by setting frequency and magnitude specifications in the appropriate frames on the **Main Pane**.

Note **Frequency**, **Magnitude**, and **Algorithm** specifications are interdependent and may change based upon your **Filter Specifications** selections. When choosing specifications for your filter, select your Filter Specifications first and work your way down the dialog box- this approach ensures that the best settings for dependent specifications display as available in the dialog box.

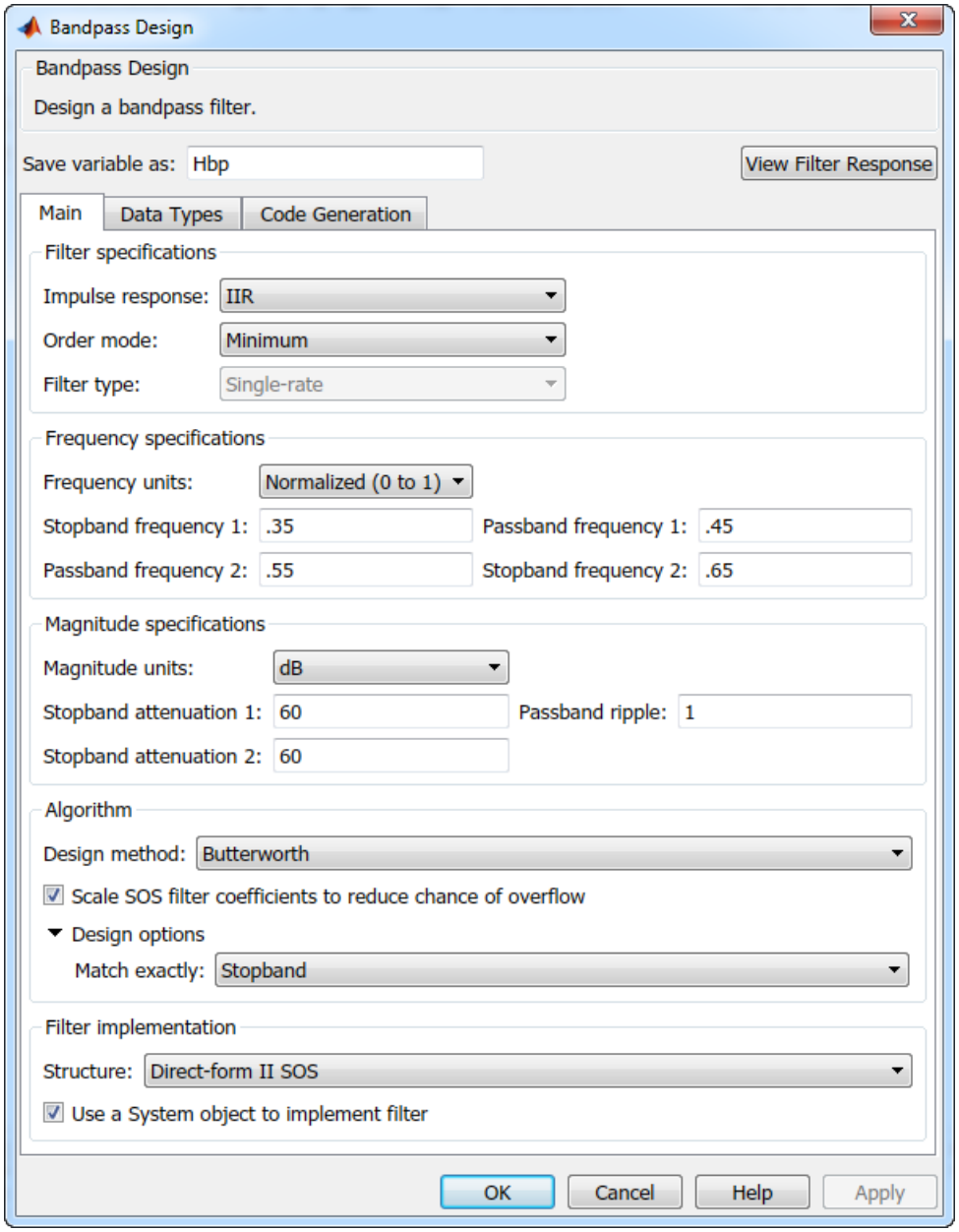
Select an Algorithm

The algorithms available for your filter depend upon the filter response and design parameters you have selected in the previous steps. For example, in the case of a bandpass filter, if the impulse response selected is IIR and the **Order Mode** field is set to `Minimum`, the design methods available are `Butterworth`, `Chebyshev` type I or II, or `Elliptic`, whereas if the **Order Mode** field is set to `Specify`, the design method available is `IIR least p-norm`.



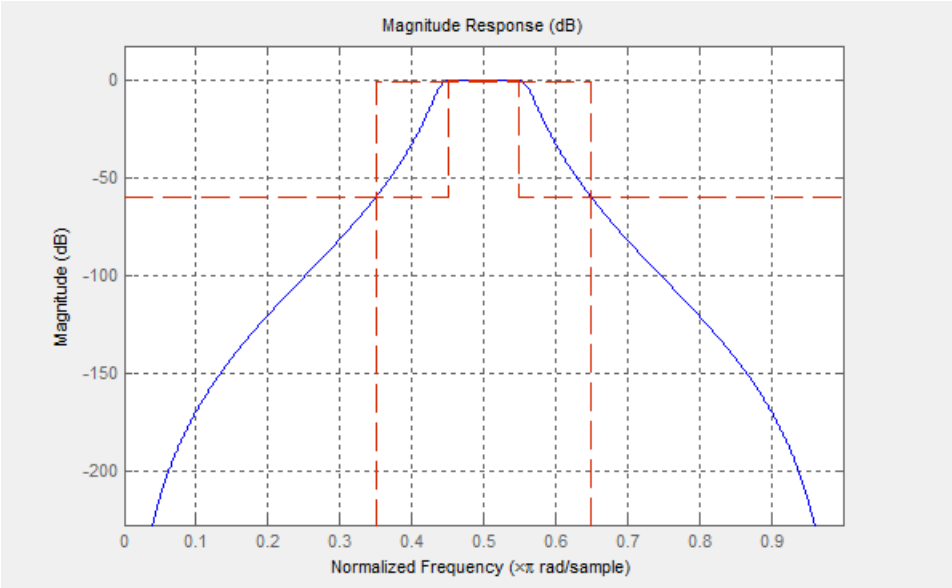
Customize the Algorithm

By expanding the **Design options** section of the **Algorithm** frame, you can further customize the algorithm specified. The options available will depend upon the algorithm and settings that have already been selected in the dialog box. In the case of a bandpass IIR filter using the `Butterworth` method, design options such as **Match Exactly** are available. Select the **Use a System object to implement filter** check box to generate a System object for the filter designed. With these settings, the `filterBuilder` generates a `dsp.BiquadFilter` **System object**.



Analyze the Design

To analyze the filter response, click on the View Filter Response button. The Filter Visualization Tool opens displaying the magnitude plot of the filter response.



Realize or Apply the Filter to Input Data

When you have achieved the desired filter response through design iterations and analysis using the **Filter Visualization Tool**, apply the filter to the input data. Again, this step is never automatically performed for you by the software. To filter your data, you must explicitly execute this step. In the **Bandpass Design** dialog box, click **OK** and DSP System Toolbox creates the filter System object and exports it to the MATLAB workspace.

The filter is then ready to be used to filter actual input data. To filter input data, `x`, enter the following in the MATLAB command prompt:

```
>> y = Hbp(x);
```

Tip If you have Simulink, you have the option of exporting this filter to a Simulink block using the `realizemdl` command. To get help on this command, type:

```
>> help realizemdl
```

Visualize Data and Signals

Learn how to display data and signals with DSP System Toolbox.

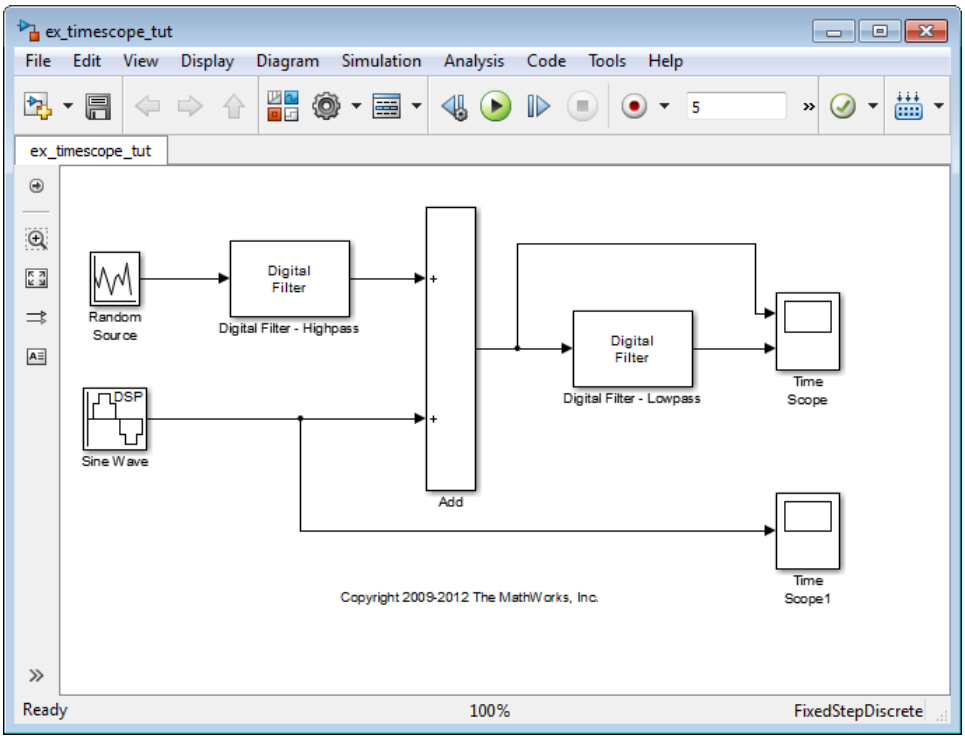
Display Time-Domain Data

In this section...
“Configure the Time Scope Properties” on page 24-3
“Use the Simulation Controls” on page 24-6
“Modify the Time Scope Display” on page 24-7
“Inspect Your Data (Scaling the Axes and Zooming)” on page 24-8
“Manage Multiple Time Scopes” on page 24-10

The following tutorial shows you how to configure the Time Scope blocks in the `ex_timescope_tut` model to display time-domain signals. To get started with this tutorial, open the model by typing

```
ex_timescope_tut
```

at the MATLAB command line.



Use the following workflow to configure the Time Scope blocks in the `ex_timescope_tut` model:

1. “Configure the Time Scope Properties” on page 24-3
2. “Use the Simulation Controls” on page 24-6
3. “Modify the Time Scope Display” on page 24-7
4. “Inspect Your Data (Scaling the Axes and Zooming)” on page 24-8
5. “Manage Multiple Time Scopes” on page 24-10

Configure the Time Scope Properties

The Configuration Properties dialog box provides a central location from which you can change the appearance and behavior of the Time Scope block. To open the Configuration Properties dialog box, you must first open the Time Scope window by double-clicking the Time Scope block in your model. When the window opens, select **View > Configuration Properties**. Alternatively, in the Time Scope toolbar, click the Configuration Properties button.

The Configuration Properties dialog box has four different tabs, **Main**, **Time**, **Display**, and **Logging**, each of which offers you a different set of options. For more information about the options available on each of the tabs, see the Time Scope block reference page.

Note As you progress through this workflow, notice the blue question mark icon ( ) in the lower-left corner of the subsequent dialog boxes. This icon indicates that context-sensitive help is available. You can get more information about any of the parameters on the dialog box by right-clicking the parameter name and selecting **What's This?**

Configure Appearance and Specify Signal Interpretation

First, you configure the appearance of the Time Scope window and specify how the Time Scope block should interpret input signals. In the Configuration Properties dialog box, click the **Main** tab. Choose the appropriate parameter settings for the **Main** tab, as shown in the following table.

Parameter	Setting
Open at simulation start	Checked
Number of input ports	2
Input processing	`Columns as channels (frame based)`
Maximize axes	`Auto`
Axes scaling	`Manual`

In this tutorial, you want the block to treat the input signal as frame-based, so you must set the **Input processing** parameter to `Columns as channels (frame based)`.

Configure Axes Scaling and Data Alignment

The **Main** tab also allows you to control when and how Time Scope scales the axes. These options also control how Time Scope aligns your data with respect to the axes. Click the link labeled **Configure...** to the right of the **Axes scaling** parameter to see additional options for axes scaling. After you click this button, the label changes to **Hide...** and new parameters appear. The following table describes these additional options.

Parameter	Description

Axes scaling	Specify when the scope automatically scales the axes. You can select one of the following options:
	• `Manual` — When you select this option, the scope does not automatically scale the axes. You can manually scale the axes in any of the following ways:
	  • Select **Tools > Axes Scaling Properties**.
	  • Press one of the **Scale Axis Limits** toolbar buttons.
	  • When the scope figure is the active window, press **Ctrl** and **A** simultaneously.
	• `Auto` — When you select this option, the scope scales the axes as needed, both during and after simulation. Selecting this option shows the **Do not allow Y-axis limits to shrink** check box.
	• `After N Updates` — Selecting this option causes the scope to scale the axes after a specified number of updates. This option is useful and more efficient when your scope display starts with one axis scale, but quickly reaches a different steady state axis scale. Selecting this option shows the **Number of updates** edit box.
	By default, this property is set to `Auto`. This property is Tunable (Simulink).

Scale axes limits at stop	Select this check box to scale the axes when the simulation stops. The *y*-axis is always scaled. The *x*-axis limits are only scaled if you also select the **Scale X-axis limits** check box.
Data range (%)	Allows you to specify how much white space surrounds your signal in the Time Scope window. You can specify a value for both the *y*- and *x*-axis. The higher the value you enter for the *y*-axis **Data range (%)**, the tighter the *y*-axis range is with respect to the minimum and maximum values in your signal. For example, to have your signal cover the entire *y*-axis range when the block scales the axes, set this value to `100`.
Align	Allows you to specify where the block should align your data with respect to each axis. You can choose to have your data aligned with the top, bottom, or center of the *y*-axis. Additionally, if you select the **Autoscale X-axis limits** check box, you can choose to have your data aligned with the right, left, or center of the *x*-axis.

Set the parameters to the values shown in the following table.

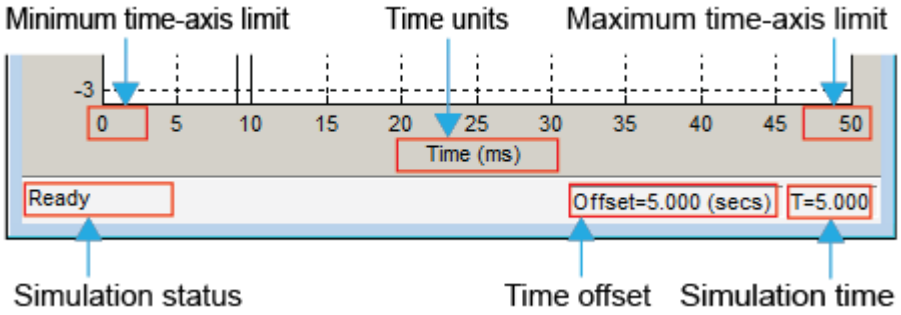
Parameter	Setting
Axes scaling	`Manual`
Scale axes limits at stop	Checked
Data range (%)	`80`
Align	`Center`
Autoscale X-axis limits	Unchecked

Set Time Domain Properties

In the Configuration Properties dialog box, click the **Time** tab. Set the parameters to the values shown in the following table.

Parameter	Setting
Time span	`One frame period`
Time span overrun action	`Wrap`
Time units	`Metric (based on Time Span)`
Time display offset	`0`
Time-axis labels	`All`
Show time-axis label	Checked

The **Time span** parameter allows you to enter a numeric value, a variable that evaluates to a numeric value, or select the `One frame period` menu option. You can also select the `Auto` menu option; in this mode, Time Scope automatically calculates the appropriate value for time span from the difference between the simulation “Start time” (Simulink) and “Stop time” (Simulink) parameters. The actual range of values that the block displays on the time axis depends on the value of both the **Time span** and **Time display offset** parameters. See the following figure.



If the **Time display offset** parameter is a scalar, the value of the minimum time-axis limit is equal to the **Time display offset**. In addition, the value of the maximum time--axis limit is equal to the sum of the **Time display offset** parameter and the **Time span** parameter. For information on the other parameters in the Time Scope window, see the Time Scope reference page.

In this tutorial, the values on the time--axis range from `0` to `One frame period`, where `One frame period` is `0.05` seconds (50 ms).

Set Display Properties

In the Configuration Properties dialog box, click the **Display** tab. Set the parameters to the values shown in the following table.

Parameter	Setting
Active display	1
Title	

Parameter	Setting
Show legend	Checked
Show grid	Checked
Plot signal(s) as magnitude and phase	Unchecked
Y-limits (Minimum)	`-2.5`
Y-limits (Maximum)	`2.5`
Y-label	`Amplitude`

Set Logging Properties

In the Configuration Properties dialog box, click the **Logging** tab. Set **Log data to workspace** to unchecked.

Click **OK** to save your changes and close the Configuration Properties dialog box.

Note If you have not already done so, repeat all of these procedures for the Time Scope1 block (except leave the **Number of input ports** on the **Main** tab as `1`) before continuing with the other sections of this tutorial.

Use the Simulation Controls

One advantage to using the Time Scope block in your models is that you can control model simulation directly from the Time Scope window. The buttons on the Simulation Toolbar of the Time Scope window allow you to play, pause, stop, and take steps forward or backward through model simulation. Alternatively, there are several keyboard shortcuts you can use to control model simulation when the Time Scope is your active window.

You can access a list of keyboard shortcuts for the Time Scope by selecting **Help** > **Keyboard Command Help**. The following procedure introduces you to these features.

1. If the Time Scope window is not open, double-click the block icon in the `ex_timescope_tut` model. Start model simulation. In the Time Scope window, on the Simulation Toolbar, click the

   Run button ( ) on the Simulation Toolbar. You can also use one of the following keyboard shortcuts:

   - **Ctrl+T**
   - **P**
   - **Space**
2. While the simulation is running and the Time Scope is your active window, pause the simulation. Use either of the following keyboard shortcuts:

   - **P**
   - **Space**

   Alternatively, you can pause the simulation in one of two ways:

   - In the Time Scope window, on the Simulation Toolbar, click the Pause button ( ).

- From the Time Scope menu, select **Simulation** > **Pause**.

3 With the model simulation still paused, advance the simulation by a single time step. To do so, in the Time Scope window, on the Simulation Toolbar, click the Next Step button ( ).

Next, try using keyboard shortcuts to achieve the same result. Press the **Page Down** key to advance the simulation by a single time step.

4 Resume model simulation using any of the following methods:

- From the Time Scope menu, select **Simulation** > **Continue**.
- In the Time Scope window, on the Simulation Toolbar, click the Continue button ( ).
- Use a keyboard shortcut, such as **P** or **Space**.

Modify the Time Scope Display

You can control the appearance of the Time Scope window using options from the display or from the **View** menu. Among other capabilities, these options allow you to:

- Control the display of the legend
- Edit the line properties of your signals
- Show or hide the available toolbars

Change Signal Names in the Legend

You can change the name of a signal by double-clicking the signal name in the legend. By default, the Time Scope names the signals based on the block they are coming from. For this example, set the signal names as shown in the following table.

Block Name	Original Signal Name	New Signal Name
Time Scope	Add	Noisy Sine Wave
Time Scope	Digital Filter – Lowpass	Filtered Noisy Sine Wave
Time Scope1	Sine Wave	Original Sine Wave

Modify Axes Colors and Line Properties

Use the Style dialog box to modify the appearance of the axes and the lines for each of the signals in your model. In the Time Scope menu, select **View** > **Style**.

1 Change the **Plot Type** parameter to `Auto` for each Time Scope block. This setting ensures that Time Scope displays a line graph if the signal is continuous and a stairstep graph if the signal is discrete.

2 Change the **Axes colors** parameters for each Time Scope block. Leave the axes background color as black and set the ticks, labels, and grid colors to white.

3 Set the **Properties for line** parameter to the name of the signal for which you would like to modify the line properties. Set the line properties for each signal according to the values shown in the following table.

Block Name	Signal Name	Line	Line Width	Marker	Color
Time Scope	Noisy Sine Wave	————	0.5	none	White
Time Scope	Filtered Noisy Sine Wave	————	0.5	◇	Red
Time Scope1	Original Sine Wave	————	0.5	✳	Yellow

Show and Hide Time Scope Toolbars

You can also use the options on the **View** menu to show or hide toolbars on the Time Scope window. For example:

- To hide the simulation controls, select **View > Toolbar**. Doing so removes the simulation toolbar from the Time Scope window and also removes the check mark from next to the **Toolbar** option in the **View** menu.
- You can choose to show the simulation toolbar again at any time by selecting **View > Toolbar**.

Verify that all toolbars are visible before moving to the next section of this tutorial.

Inspect Your Data (Scaling the Axes and Zooming)

Time Scope has plot navigation tools that allow you to scale the axes and zoom in or out on the Time Scope window. The axes scaling tools allow you to specify when and how often the Time Scope scales the axes.

So far in this tutorial, you have configured the Time Scope block for manual axes scaling. Use one of the following options to manually scale the axes:

- From the Time Scope menu, select **Tools > Scale Axes Limits**.
- Press the Scale Axes Limits toolbar button ( ).
- With the Time Scope as your active window, press **Ctrl + A**.

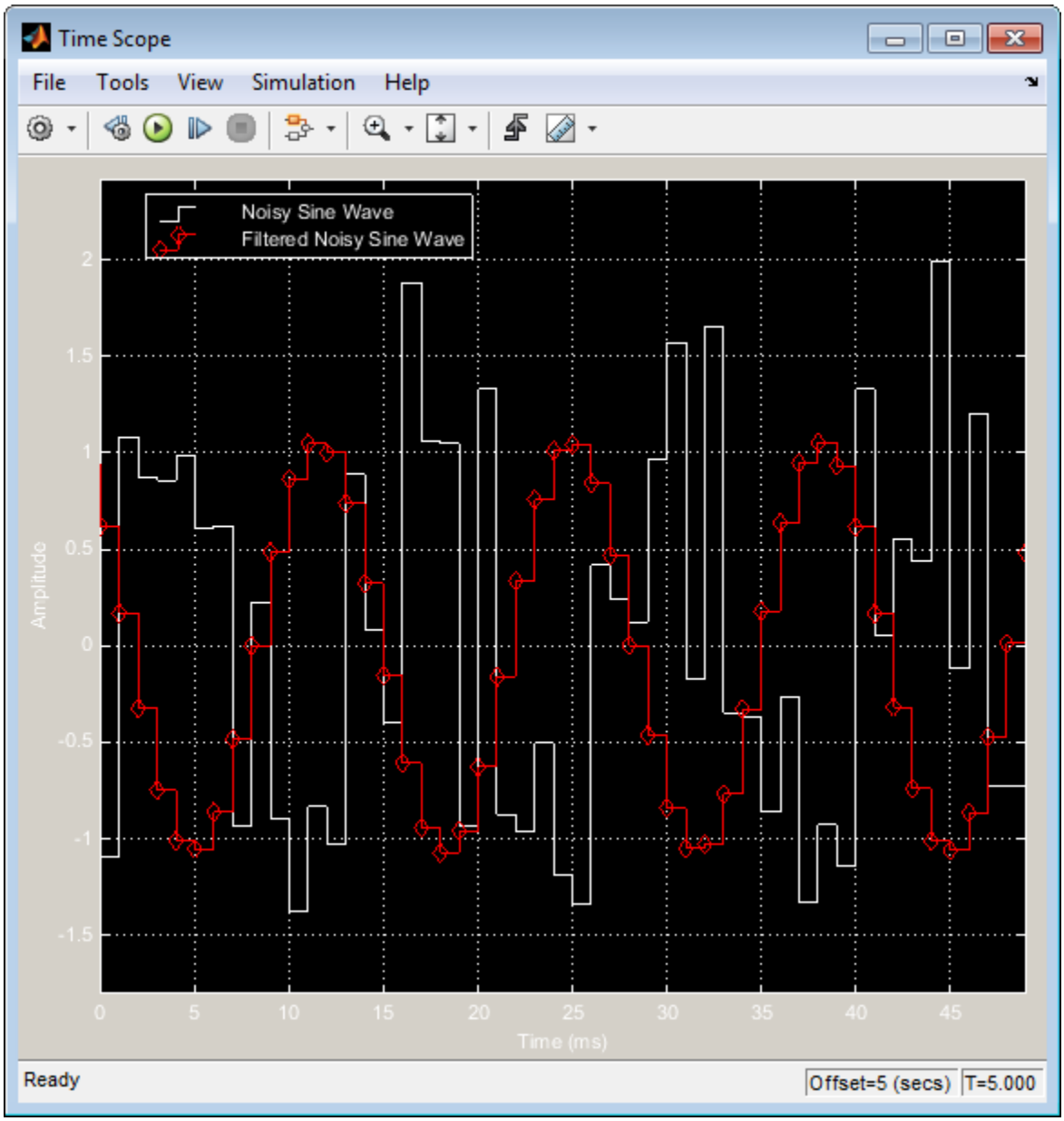
Adjust White Space Around the Signal

You can control how much space surrounds your signal and where your signal appears in relation to the axes. To adjust the amount of space surrounding your signal and realign it with the axes, you must first open the Tools—Plot Navigation Properties dialog box. From the Time Scope menu, select **Tools > Axes Scaling Properties** .

In the Tools:Plot Navigation options dialog box, set the **Data range (%)** and **Align** parameters. In a previous section, you set these parameters to `80` and `Center`, respectively.

- To decrease the amount of space surrounding your signal, set the **Data range (%)** parameter on the Tools:Plot Navigation Options dialog box to `90`.
- To align your signal with the bottom of the Y-axis, set the **Align** parameter to `Bottom`.

The next time you scale the axes of the Time Scope window, the window appears as follows.



Use the Zoom Tools

The zoom tools allow you to zoom in simultaneously in the directions of both the *x*- and *y*-axes , or in either direction individually. For example, to zoom in on the signal between 5010 ms and 5020 ms, you can use the **Zoom X** option.

- To activate the **Zoom X** tool, select **Tools** > **Zoom X**, or press the corresponding toolbar button ( ). The Time Scope indicates that the **Zoom X** tool is active by depressing the toolbar button and placing a check mark next to the **Tools** > **Zoom X** menu option.
- To zoom in on the region between 5010 ms and 5020 ms, in the Time Scope window, click and drag your cursor from the 10 ms mark to the 20 ms mark.

- While zoomed in, to activate the **Pan** tool, select **Tools > Pan**, or press the corresponding toolbar button ( ).
- To zoom out of the Time Scope window, right-click inside the window, and select **Zoom Out**. Alternatively, you can return to the original view of your signal by right-clicking inside the Time Scope window and selecting **Reset to Original View**.

Manage Multiple Time Scopes

The Time Scope block provides tools to help you manage multiple Time Scope blocks in your models. The model used throughout this tutorial, `ex_timescope_tut`, contains two Time Scope blocks, labeled `Time Scope` and `Time Scope1`. The following sections discuss the tools you can use to manage these Time Scope blocks.

Open All Time Scope Windows

When you have multiple windows open on your desktop, finding the one you need can be difficult. The Time Scope block offers a **View > Bring All Time Scopes Forward** menu option to help you manage your Time Scope windows. Selecting this option brings all Time Scope windows into view. If a Time Scope window is not currently open, use this menu option to open the window and bring it into view.

To try this menu option in the `ex_timescope_tut` model, open the Time Scope window, and close the Time Scope1 window. From the **View** menu of the Time Scope window, select **Bring All Time Scopes Forward**. The Time Scope1 window opens, along with the already active Time Scope window. If you have any Time Scope blocks in other open Simulink models, then these also come into view.

Open Time Scope Windows at Simulation Start

When you have multiple Time Scope blocks in your model, you may not want all Time Scope windows to automatically open when you start simulation. You can control whether or not the Time Scope window opens at simulation start by selecting **File > Open at Start of Simulation** from the Time Scope window. When you select this option, the Time Scope GUI opens automatically when you start the simulation. When you do not select this option, you must manually open the scope window by double-clicking the corresponding Time Scope block in your model.

Find the Right Time Scope Block in Your Model

Sometimes, you have multiple Time Scope blocks in your model and need to find the location of one that corresponds to the active Time Scope window. In such cases, you can use the **View > Highlight Simulink Block** menu option or the corresponding toolbar button ( ). When you do so, the model window becomes your active window, and the corresponding Time Scope block flashes three times in the model window. This option can help you locate Time Scope blocks in your model and determine to which signals they are attached.

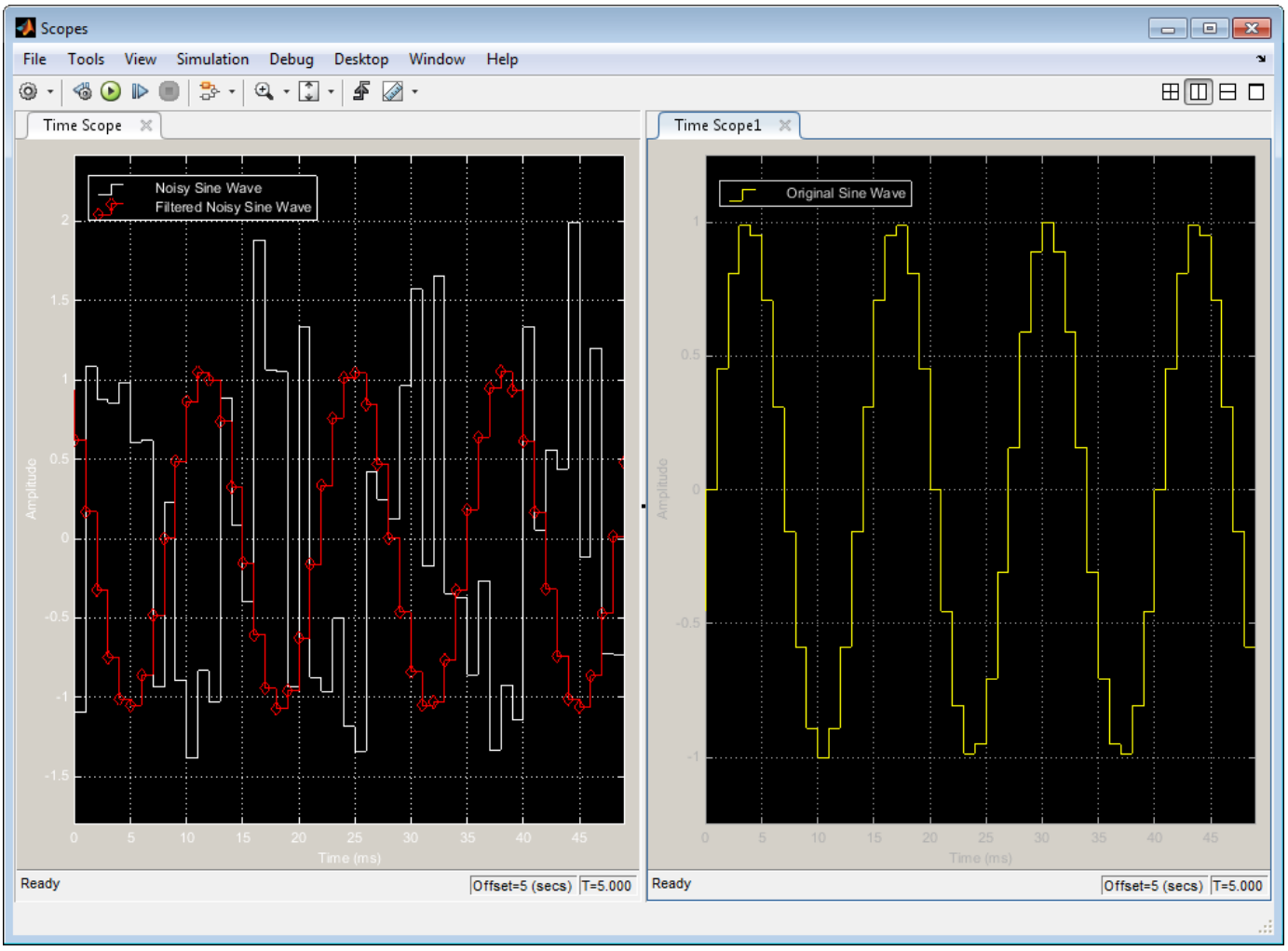
To try this feature, open the Time Scope window, and on the simulation toolbar, click the Highlight Simulink Block button. Doing so opens the `ex_timescope_tut` model. The Time Scope block flashes three times in the model window, allowing you to see where in your model the block of interest is located.

Docking Time Scope Windows in the Scopes Group Container

When you have multiple Time Scope blocks in your model you may want to see them in the same window and compare them side-by-side. In such cases, you can select the Dock Time Scope button (↘) at the top-right corner of the Time Scope window for the Time Scope block.

The Time Scope window now appears in the Scopes group container. Next, press the Dock Time Scope button at the top-right corner of the Time Scope window for the Time Scope1 block.

By default, the Scopes group container is situated above the MATLAB Command Window. However, you can undock the Scopes group container by pressing the Show Actions button (▼) at the top-right corner of the container and selecting **Undock**. The Scopes group container is now independent from the MATLAB Command Window.



Once docked, the Scopes group container displays the toolbar and menu bar of the Time Scope window. If you open additional instances of Time Scope, a new Time Scope window appears in the Scopes group container.

You can undock any instance of Time Scope by pressing the corresponding Undock button ( ) in the title bar of each docked instance. If you close the Scopes group container, all docked instances of Time Scope close but the Simulink model continues to run.

Close All Time Scope Windows

If you save your model with Time Scope windows open, those windows will reopen the next time you open the model. Reopening the Time Scope windows when you open your model can increase the amount of time it takes your model to load. If you are working with a large model, or a model containing multiple Time Scopes, consider closing all Time Scope windows before you save and close that model. To do so, use the **File > Close All Time Scope Windows** menu option.

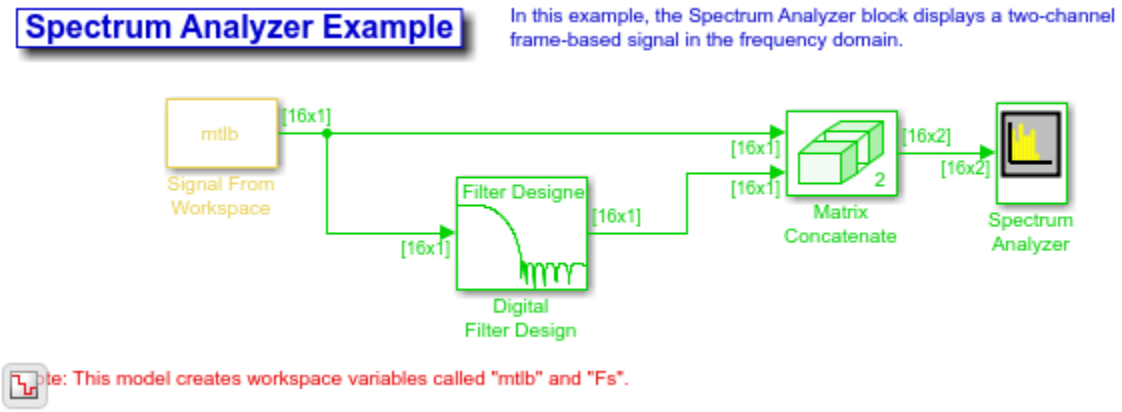
To use this menu option in the `ex_timescope_tut` model, open the Time Scope or Time Scope1 window, and select **File > Close All Time Scope Windows**. Both the Time Scope and Time Scope1 windows close. If you now save and close the model, the Time Scope windows do not automatically open the next time you open the model. You can open Time Scope windows at any time by double-clicking a Time Scope block in your model. Alternatively, you can choose to automatically open the Time Scope windows at simulation start. To do so, from the Time Scope window, select **File > Open at Start of Simulation**.

Display Frequency-Domain Data in Spectrum Analyzer

This example shows how you can use a Spectrum Analyzer block to display the frequency content of two frame-based signals simultaneously. The Spectrum Analyzer block computes the Fast Fourier Transform (FFT) of the input signal internally, transforming the signal into the frequency domain.

Open the `ex_spectrumanalyzer_tut` model.

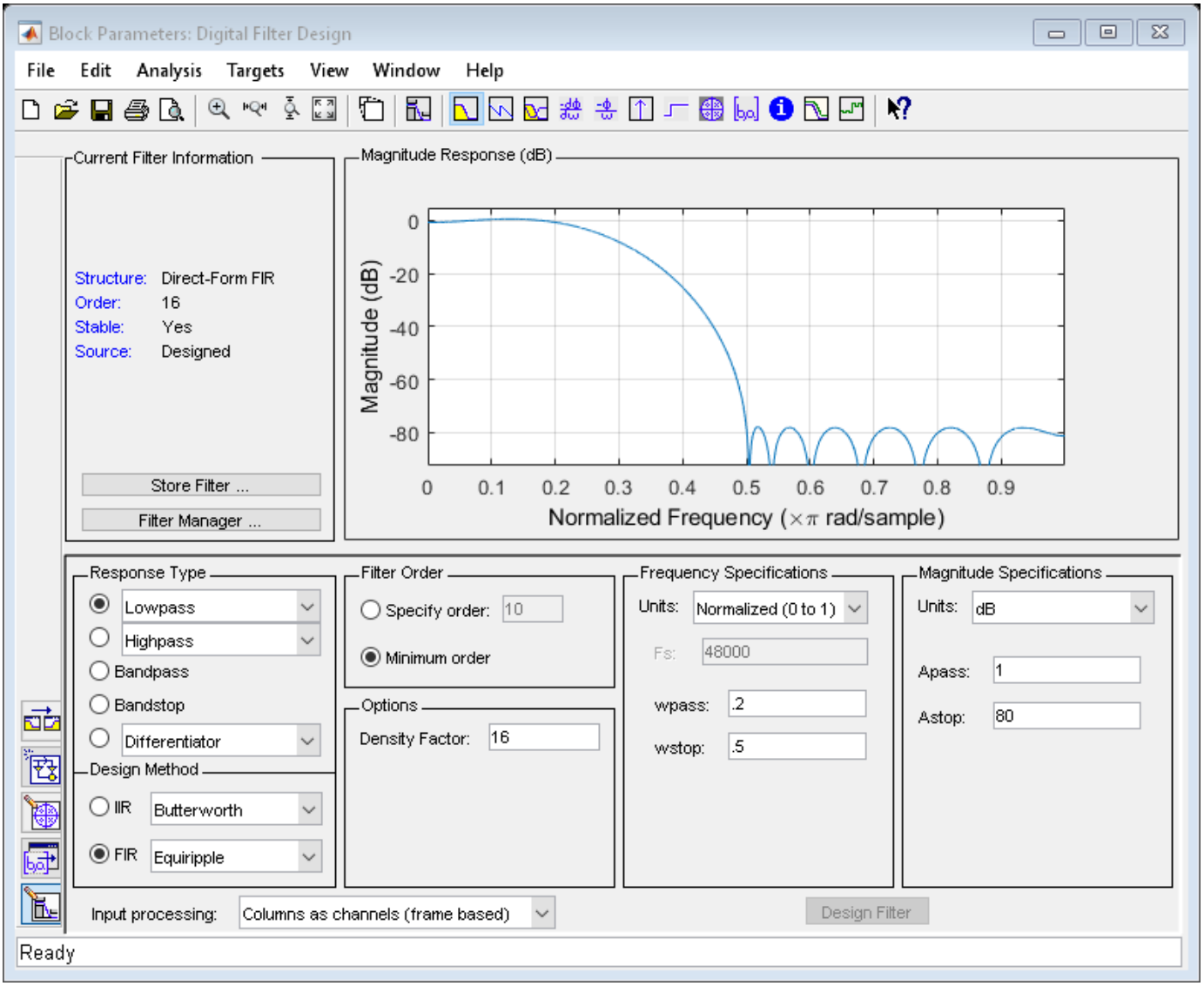
```
model = 'ex_spectrumanalyzer_tut';
open_system(model)
```



The Signal From Workspace block repeatedly outputs the input signal, `mtlb`, as a frame-based signal with a sample period of 1 second.

The Digital Filter Design block filters the input signal, using the default parameters.

```
open_system([model '/Digital Filter Design'])
```



The Matrix Concatenate block combines the two signals so that each column corresponds to a different signal.

The frequency of the signals are displayed in the Spectrum Analyzer. The Spectrum Analyzer uses 128 samples from each input channel to calculate a new windowed data segment, as shown in this equation:

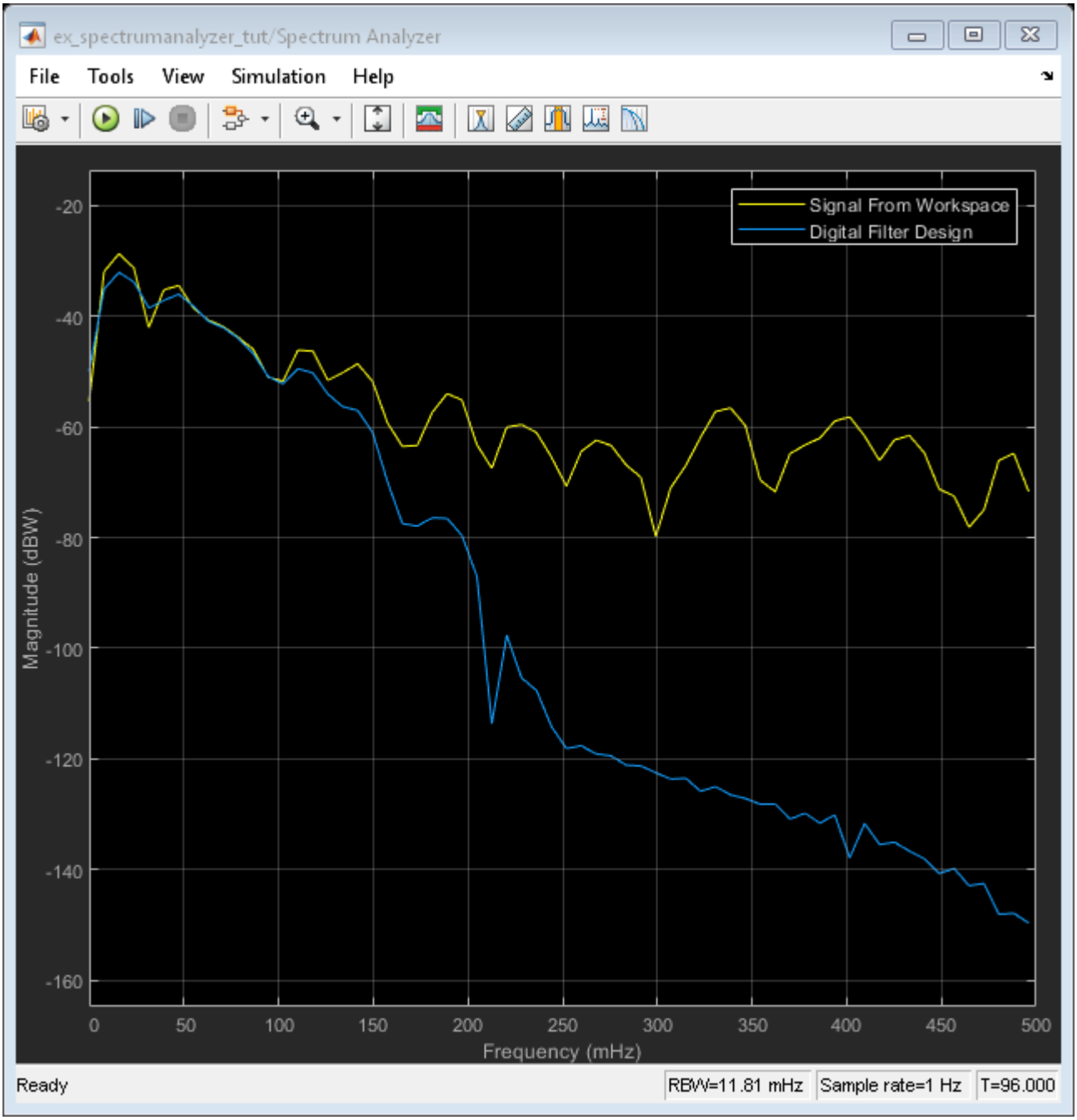
$$N_s = \frac{NENBW \times F_s}{RBW} = \frac{1.512 \times 1}{11.8125 \times 10^{-3}} = 128$$

Therefore, the FFT also has a length of 128 frequency points. Also, because **Overlap (%)** is set to `50`, there is a buffer overlap length of 64 samples in each spectral estimate, as shown in the following equation:

$$O_L = \frac{O_P}{100} \times L = \frac{50}{100} \times 128 = 64$$

Run the model and view power frequency of the signals in the Spectrum Analyzer. The power spectrum of the first input signal, from column one, is the yellow line. The power spectrum of the second input signal, from column two, is the blue line.

```
sim(model)
open_system([model '/Spectrum Analyzer'])
```



Visualize Central Limit Theorem in Array Plot

This example shows how to use and configure the `dsp.ArrayPlot` System object to visualize the Central Limit Theorem. This theorem states that if you take a large number of random samples from a population, the distribution of the means of the samples approaches a normal distribution.

Display a Uniform Distribution

The population for this example is a uniform distribution of random numbers between 0 and 1. Generate a sample set of the values in MATLAB using the `rand` function. Find their distributions using the `histcounts` function.

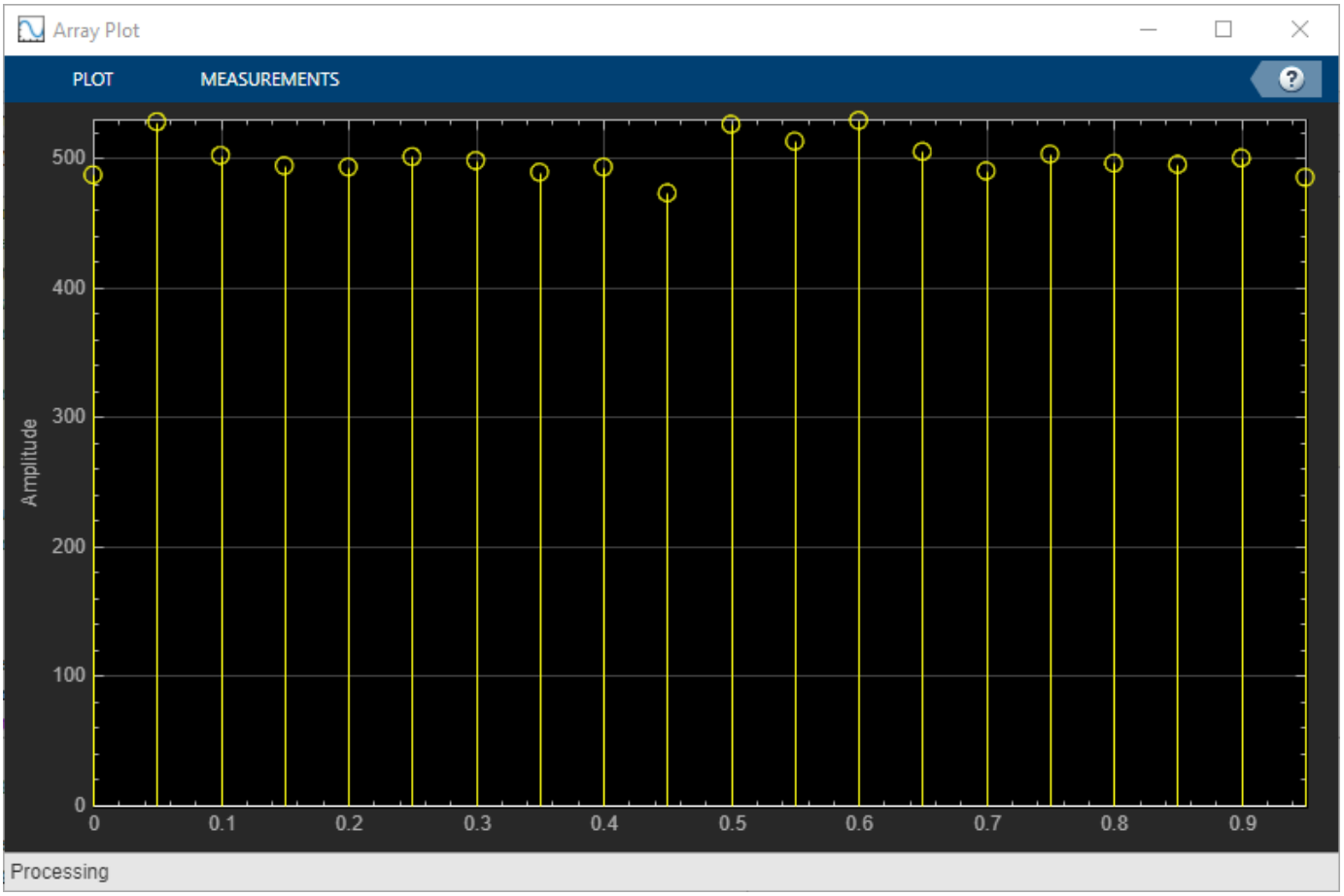
```
numsamples = 1e4;
numbins = 20;
r = rand(numsamples,1);
hst = histcounts(r,numbins);
```

Create a new array plot object and configure the properties of the array plot object to plot a histogram.

```
scope = dsp.ArrayPlot;
scope.XOffset = 0;
scope.SampleIncrement = 1/numbins;
scope.PlotType = 'Stem';
scope.YLimits = [0, max(hst)+1];
```

Call the scope to plot the uniform distribution.

```
scope(hst')
```



Display the Distribution of Multiple Samples

Next, simulate the calculation of multiple uniformly distributed random samples. Because the population is a uniformly distributed set of values between 0 and 1, we can simulate the sampling and calculation of sample means by generating random values between 0 and 1. As the number of random samples increases, the distribution of the means more closely resembles a normal curve. Run the release method to let property values and input characteristics change.

```
hide(scope);
release(scope);
```

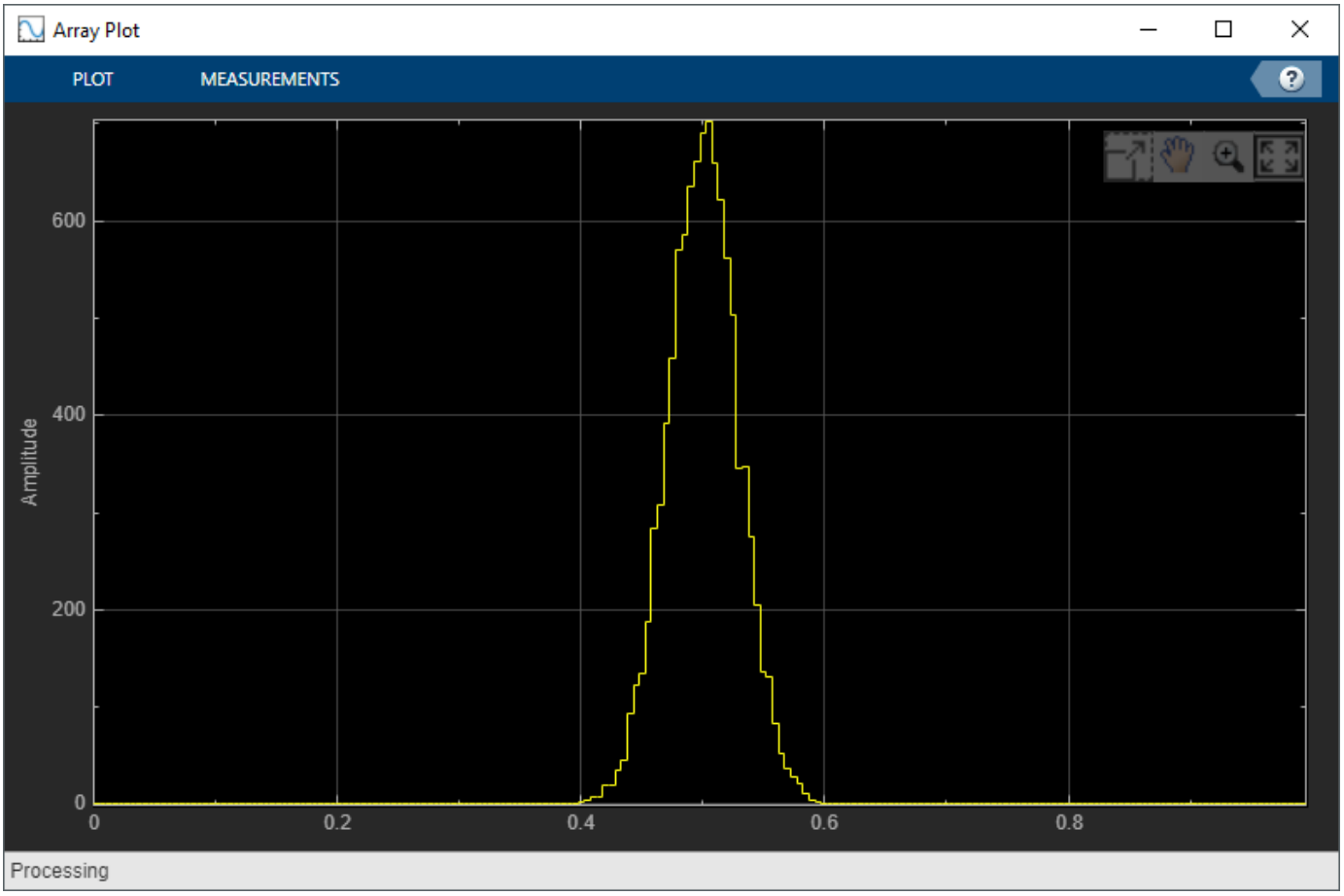
Change the configuration of the Array Plot properties for the display of a distribution function.

```
numbins = 201;
numtrials = 100;
r = zeros(numsamples,1);
scope.SampleIncrement = 1/numbins;
scope.PlotType = 'Stairs';
```

Call the scope repeatedly to plot the distribution of the samples.

```
show(scope);
for ii = 1:numtrials
    r = rand(numsamples,1)+r;
    hst = histcounts(r/ii,0:1/numbins:1);
```

```
    scope.YLimits = [min(hst)-1, max(hst)+1];
    scope(hst')
    pause(0.1);
end
```



When the simulation has finished, the Array Plot figure displays a bell curve, indicating a distribution that is close to normal.

Inspect Your Data by Zooming

The zoom tools allow you to zoom in simultaneously in the directions of both the *x*- and *y*-axes or in either direction individually. For example, to zoom in on the distribution between 0.3 and 0.7, you can use the Zoom X option.

- To activate the Zoom X tool, select **Tools > Zoom X**, or press the corresponding toolbar button. You can determine if the Zoom X tool is active by looking for an indented toolbar button or a check mark next to the **Tools > Zoom X** menu option.
- Next, zoom in on the region between 0.3 and 0.7. In the Array Plot window, click on the 0.3-second mark and drag to the 0.7-second mark.

See Also

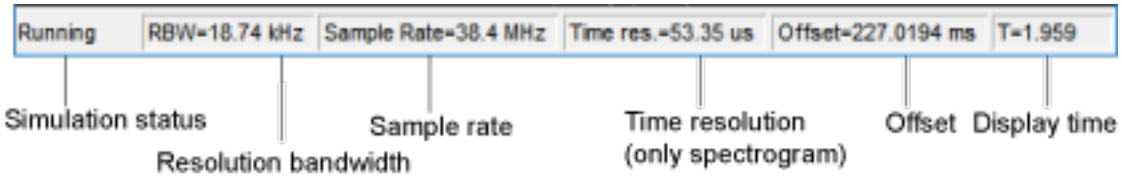
`dsp.ArrayPlot` | `histcounts` | `rand`

Configure Spectrum Analyzer

Use the Spectrum Analyzer to visualize frequency. You can customize the spectrum analyzer display to show the data and measurement information needed.

Signal and Spectrum Computation Information

The Spectrum Analyzer shows the spectrum computation settings for the current visualization. In the scope status bar, check the *Resolution Bandwidth*, *Time Resolution*, and *Offset* indicators for this information. The values specified by these indicators can change depending on your settings in the **Spectrum Settings** panel. You can also view the simulation status and the amount of time data that correspond to the current display. Check the *Simulation status* and *Display time* indicators for this information.



- *Resolution Bandwidth* — The smallest positive frequency or frequency interval that can be resolved.

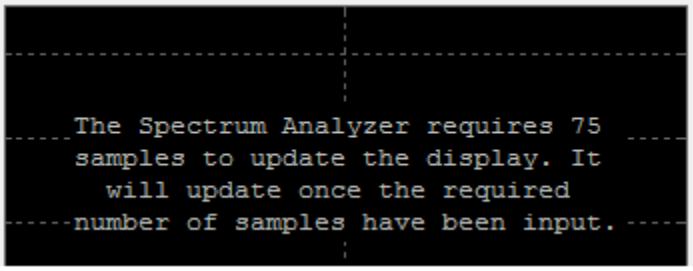
  Details

  Spectrum Analyzer sets the resolution bandwidth based on the `FrequencyResolutionMethod` property setting on the **Main options** pane of the **Spectrum Settings** panel. If `FrequencyResolutionMethod` is `RBW (Hz)` then the specified value of RBW is used. You can also get or set this value from the `RBW` property when `RBWSource` is set to `'Property'`. By default, the `RBW (Hz)` parameter on the **Main options** pane and the related `RBWSource` property are set to `'Auto'`. In this case, the Spectrum Analyzer determines the appropriate value to ensure that there are 1024 RBW intervals over the specified **Frequency Span**.

  You can set the resolution bandwidth to whatever value you choose. For this reason, there is a minimum boundary on the number of input samples required to compute a spectral update. This number of input samples required to compute one spectral update is shown as **Samples/update** in the **Main options** pane. This value is directly related to *RBW* by the following equation:

  $$N_{samples} = \frac{\left(1 - \frac{O_p}{100}\right) \times NENBW \times F_s}{RBW}$$

  Overlap percentage, O_p, is the value of the **Overlap %** parameter in the **Window Options** pane of the **Spectrum Settings** panel. *NENBW* is the normalized effective noise bandwidth, a factor of the windowing method used, which is shown in the **Window Options** pane. F_s is the sample rate. In some cases, the number of samples provided in the input are not sufficient to achieve the resolution bandwidth that you specify. When this situation occurs, Spectrum Analyzer shows a warning message on the display.



Spectrum Analyzer removes this message and displays a spectral estimate when enough data has been input.

If the `FrequencyResolutionMethod` property setting on the **Main options** pane of the **Spectrum Settings** is `Window length`, you specify the window length and the resulting RBW is

$$\frac{NENBW \times F_s}{N_{window}}$$

The **Samples/update** in this case is directly related to *RBW* by the following equation:

$$N_{samples} = \left(1 - \frac{O_p}{100}\right) N_{window}$$

- *Time Resolution* — The time resolution for a spectrogram line.

  Details

  Time resolution is the amount of data, in seconds, used to compute a spectrogram line. The **Time Resolution** parameter is available only when the spectrum **View** is `Spectrogram`. The minimum attainable resolution is the amount of data time required to compute a single spectral estimate. When the `SpectrumType` property is set to `'Spectrogram'`, you can get or set the minimum attainable resolution value from the `TimeResolution` property. See the time resolution table in the `TimeResolution` property description.

- *Offset* — The constant frequency offset to apply to the entire spectrum or a vector of frequency offsets to apply to each spectrum for multiple inputs.

  Details

  Spectrum Analyzer adds this constant offset or the vector of offsets to the values on the *frequency*-axis using the value of **Offset** on the **Trace options** pane of the **Spectrum Settings** panel. You can also set the offset from the `FrequencyOffset` property. The offset is the current time value at the middle of the interval of the line displayed at 0 seconds. The actual time of a particular spectrogram line is the offset minus the *y*-axis time listing. The offset is displayed on the plot only when the spectrum **View** is `Spectrogram`.

- *Simulation Status* — Provides the status of the model simulation.

  Details

  The status can be one of the following conditions:

  - `Processing` — Occurs after you construct the `SpectrumAnalyzer` object and before you run the `release` method.
  - `Stopped` — Occurs after you run the `release` method.

The *Simulation Status* is part of the status bar in the Spectrum Analyzer window. You can choose to hide or display the entire status bar. From the Spectrum Analyzer menu, select **View > Status Bar**.

- *Display time* — The amount of time that has progressed since the last update to the Spectrum Analyzer display.

  Details

  Every time you call the scope, the simulation time increases by the number of rows in the input signal divided by the sample rate, as given by the following formula:

  $$t_{sim} = t_{sim-1} + \frac{length(0:length(\text{xsine})) - 1}{SampleRate}$$

  . At the beginning of a simulation, you can modify the **SampleRate** parameter on the **Main options** pane of the **Spectrum Settings** panel. You can also set the sample rate using the `SampleRate` property. The display time is updated each time the display is updated. When `ReducePlotRate` is `true`, the simulation time and display time might differ. If at the end of a for loop that includes the Spectrum Analyzer, the times differ, you can call the `release` method to update the display with any data left in the buffer. Note, however, that if the remaining data is not a complete window interval, the display is not updated.

  The *Display time* indicator is a component of the status bar in the Spectrum Analyzer window. You can choose to hide or display the entire status bar. From the Spectrum Analyzer menu, select **View > Status Bar** .

- *Frequency span* — The range of values shown on the *frequency*-axis on the Spectrum Analyzer window.

  Details

  Spectrum Analyzer sets the frequency span using the values of parameters on the **Main options** pane of the **Spectrum Settings** panel.

  - `Span(Hz)` and `CF(Hz)` visible — The *Frequency span* value equals the **Span** parameter in the **Main options** pane. You can also get or set this value from the `Span` property when the `FrequencySpan` property is set to `'Span and Center Frequency'`.
  - `FStart(Hz)` and `FStop(Hz)` — The *Frequency span* value equals the difference of the **FStop** and **FStart** parameters in the **Main options** pane, as given by the formula: $f_{span} = f_{stop} - f_{start}$. You can also get or set these values from the `StartFrequency` and `StopFrequency` properties when the `FrequencySpan` property is set to `'Start and stop frequencies'`.

  By default, the **Full Span** check box in the **Main options** pane is enabled, and its equivalent `FrequencySpan` property is set to `'Full'`. In this case, the Spectrum Analyzer computes and plots the spectrum over the entire Nyquist frequency interval. When the **Two-sided spectrum** check box in the **Trace options** pane is enabled, and its equivalent `PlotAsTwoSidedSpectrum` property is `true`, the Nyquist interval is, in hertz:

  $$\left[-\frac{SampleRate}{2}, \frac{SampleRate}{2}\right) + FrequencyOffset$$

  If you set the `PlotAsTwoSidedSpectrum` property to `false`, the Nyquist interval is in hertz:

  $$\left[0, \frac{SampleRate}{2}\right) + FrequencyOffset$$

Reduce Plot Rate to Improve Performance

By default, Spectrum Analyzer updates the display at fixed intervals of time at a rate not exceeding 20 hertz. If you want Spectrum Analyzer to plot a spectrum on every simulation time step, you can disable the **Simulation** > **Reduce Plot Rate to Improve Performance** option.

Note When this option is selected, the Spectrum Analyzer may display a misleading spectrum in some situations. For example, if the input signal is wide-band with non-stationary behavior, such as a chirp signal, Spectrum Analyzer might display a stationary spectrum. The reason for this behavior is that Spectrum Analyzer buffers the input signal data and only updates the display periodically at approximately 20 times per second. Therefore, Spectrum Analyzer does not render changes to the spectrum that occur and elapse between updates, which gives the impression of an incorrect spectrum. To ensure that spectral estimates are as accurate as possible, clear the **Reduce Plot Rate to Improve Performance** check box. When you clear this box, Spectrum Analyzer calculates spectra whenever there is enough data, rendering results correctly.

Generate a MATLAB Script

You can change Spectrum Analyzer settings using menus and options in the interface of the scope, or by changing properties at the command line. If you change settings in the `dsp.SpectrumAnalyzer` interface, you can generate the corresponding command line settings to use later.

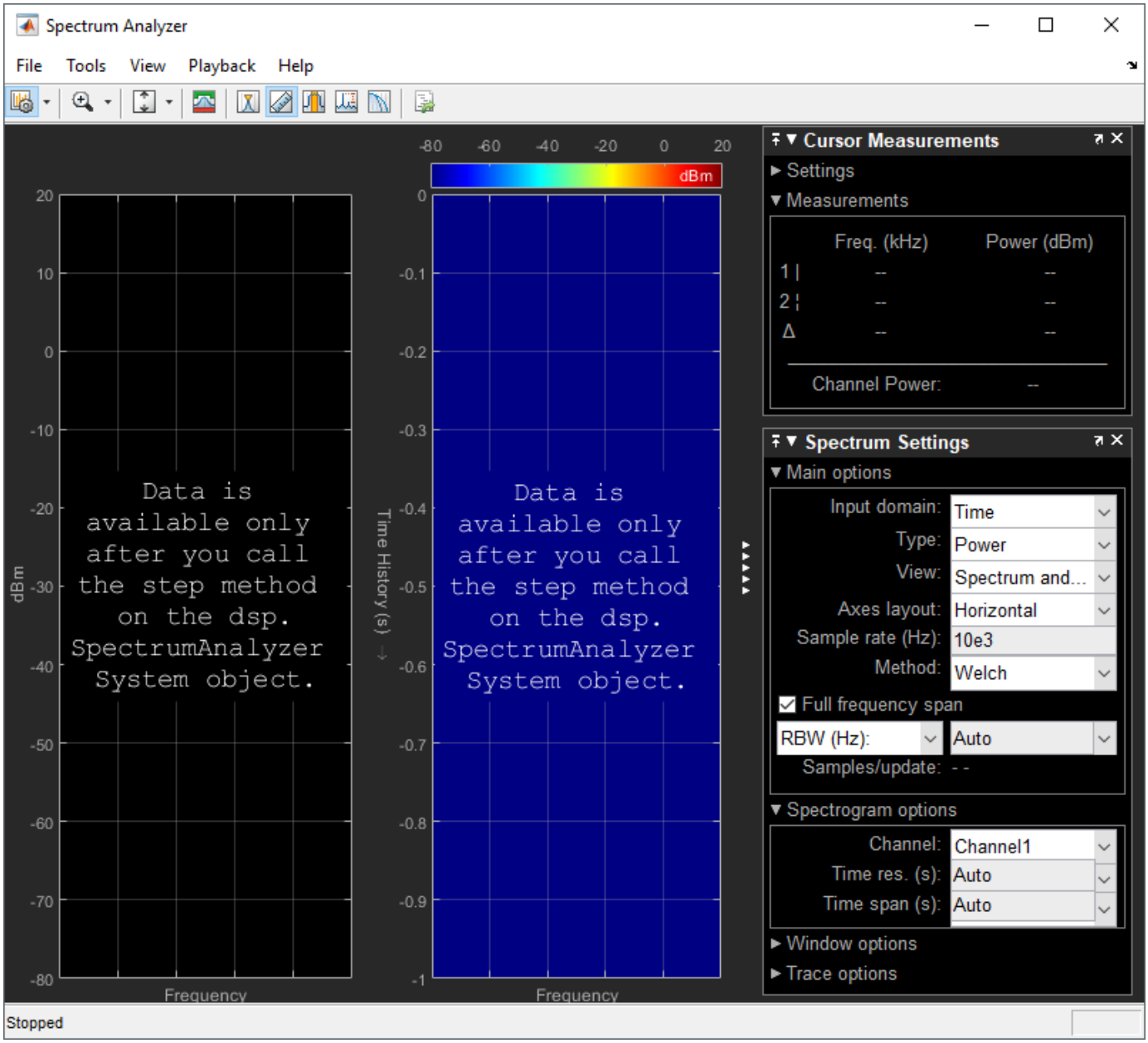
Note The script only generates commands for settings that are available from the command line, applicable to the current visualization, and changed from the default value.

This example shows how to generate a script after making changes to the `dsp.SpectrumAnalyzer` in the interface:

1. Create a `dsp.SpectrumAnalyzer` System object.

   ```
   scope = dsp.SpectrumAnalyzer();
   show(scope);
   ```

2. Set options in the Spectrum Analyzer. For this example, turn on the Cursor Measurements. Also in the Spectrum Settings, change the **View** type to **Spectrum and spectrogram** and set the **Axes Layout** to **Horizontal**.



3 Generate a script to recreate the `dsp.SpectrumAnalyzer` with the same modified settings. Either select **File > Generate MATLAB Script** or enter:

```
generateScript(scope);
```

A new editor window opens with code to regenerate the same scope.

```
% Creation Code for 'dsp.SpectrumAnalyzer'.
% Generated by Spectrum Analyzer on 10-Mar-2019 16:25:49 -0500.

specScope = dsp.SpectrumAnalyzer('ViewType','Spectrum and spectrogram', ...
    'AxesLayout','Horizontal');
% Cursor Measurements Configuration
specScope.CursorMeasurements.Enable = true;
```

Spectral Masks

Add upper and lower masks to the Spectrum Analyzer to visualize spectrum limits and compare spectrum values to specification values.

To open the **Spectral Mask** pane, in the toolbar, select the spectral mask button, .

Set Up Spectral Masks

In the Spectrum Analyzer window:

1. In the **Spectral Mask** pane, select a **Masks** option.
2. In the **Upper limits** or **Lower limits** box, enter the mask limits as a constant scalar, an array, or a workspace variable name.
3. (Optional) Select additional properties:
   - **Reference level** — Set a reference level for the mask. Enter a specific value or select `Spectrum peak`.
   - **Channel** — Select a channel to use for the mask reference.
   - **Frequency offset** — Set a frequency offset for mask.

From the command-line, to add a spectral mask to the `dsp.SpectrumAnalyzer` System object or the `SpectrumAnalyzerConfiguration` block configuration object:

1. Create a `SpectralMaskSpecfication` object.
2. Set properties, such as `EnabledMasks`, `LowerMask`, or `UpperMask`. For a full list of properties, see SpectralMask (block) and SpectralMask (System object).
3. In the `dsp.SpectrumAnalyzer` or `SpectrumAnalyzerConfiguration` object, set the `SpectralMask` property equal to your `SpectralMaskSpecfication` object.

For example:

```
mask = SpectralMaskSpecification();
mask.EnabledMasks = 'Upper';
mask.UpperMask = 10;
scope = dsp.SpectrumAnalyzer();
scope.SpectralMask = mask;
scope.SpectralMask

ans =

  SpectralMaskSpecification with properties:

           EnabledMasks: 'Upper'
              UpperMask: 10
              LowerMask: -Inf
         ReferenceLevel: 'Custom'
    CustomReferenceLevel: 0
     MaskFrequencyOffset: 0

  Events for class SpectralMaskSpecification: MaskTestFailed
```

Check Spectral Masks

You can check the status of the spectral mask in several different ways:

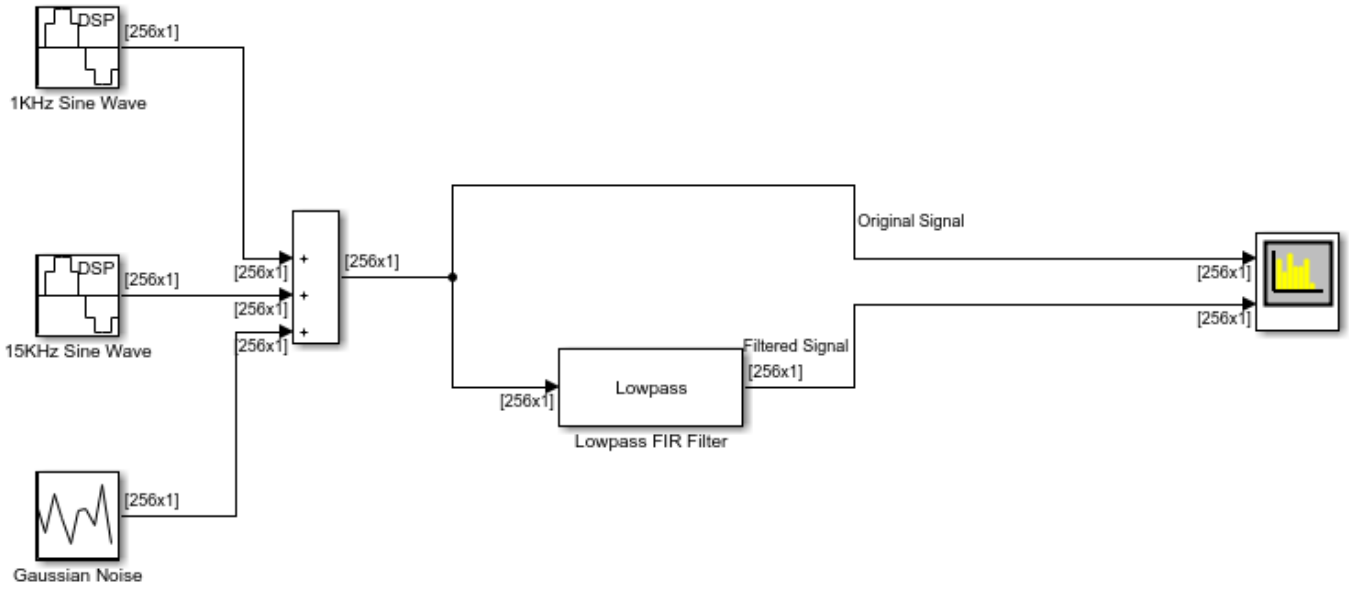
- In the Spectrum Analyzer window, select the spectral mask button, . In the **Spectral Mask** pane, the **Statistics** section shows statistics about how often the masks fail, which channels have caused a failure, and which masks are currently failing.
- To get the current status of the spectral masks, call `getSpectralMaskStatus`.
- To perform an action every time the mask fails, use the `MaskTestFailed` event. To trigger a function when the mask fails, create a listener to the `MaskTestFailed` event and define a callback function to trigger. For more details about using events, see “Events” (MATLAB).

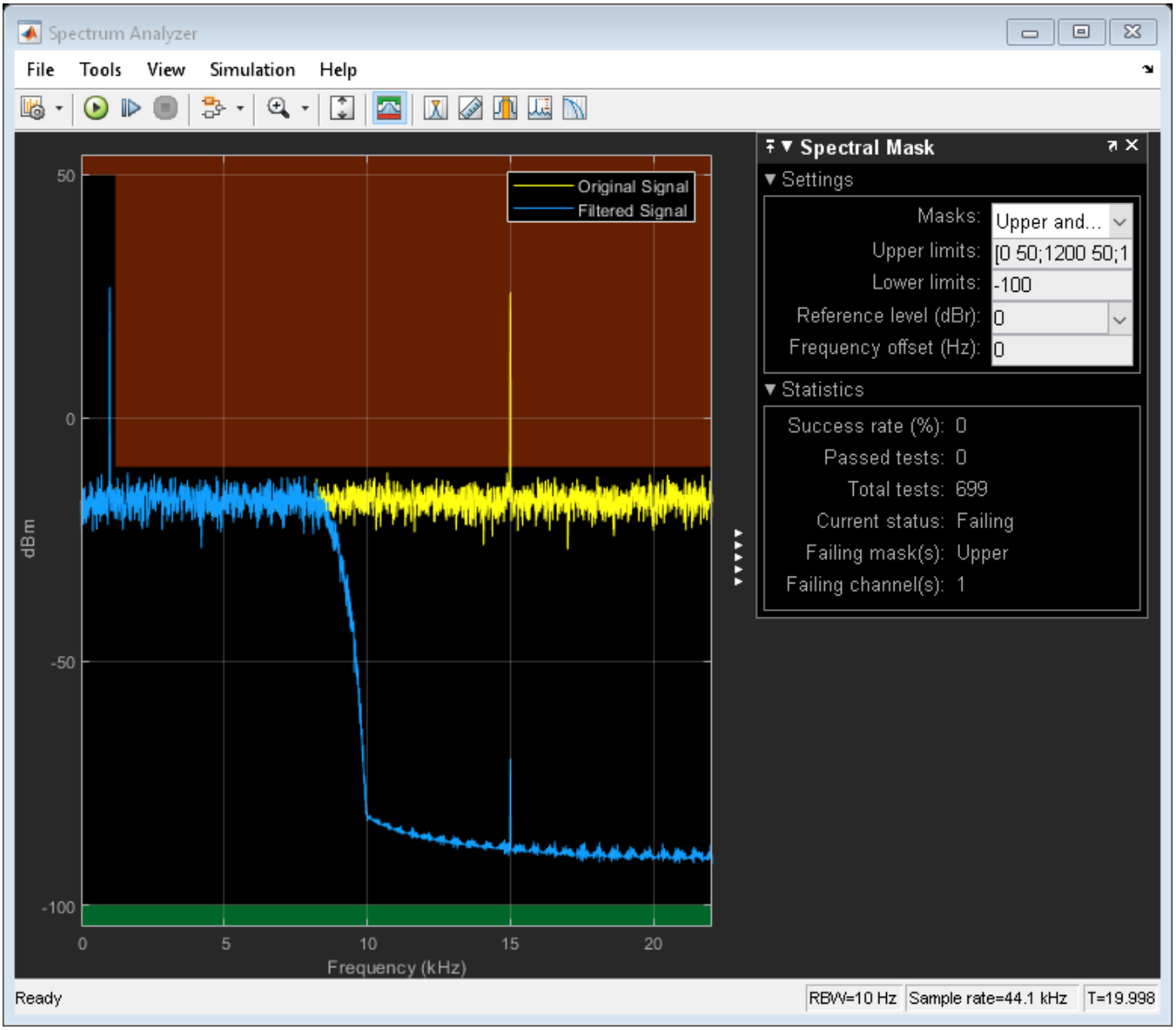
Spectral Mask in Spectrum Analyzer Block

This example shows how to create a new model based on the `dsp_basic_filter` template, add a spectral mask to its Spectrum Analyzer block, and run the model.

Masks are overlaid on the spectrum. If the mask is green, the signal is passing. If the mask is red, the signal is failing. The **Spectral Mask** panel shows what percentage of the time the mask is succeeding, which mask is failing, how many times the mask(s) failed, and which channels are causing the failure.

```
[~,mdl] = fileparts(tempname);
open_system(new_system(mdl,'FromTemplate','dsp_basic_filter'));
saBlock = find_system(mdl,'BlockType','SpectrumAnalyzer');

scopeConfig = get_param(saBlock{1},'ScopeConfiguration');
upperMask = [0 50; 1200 50; 1200 -10; 24000 -10];
scopeConfig.SpectralMask.UpperMask = upperMask;
scopeConfig.SpectralMask.LowerMask = -100;
scopeConfig.SpectralMask.EnabledMasks = 'Upper and lower';

sim(mdl,'StopTime','20');
```





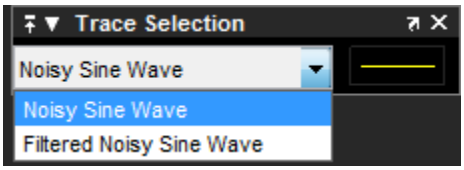
Measurements Panels

The Measurements panels are the panels that appear on the right side of the Spectrum Analyzer. These measurements allow you to interact with the frequency values.

Trace Selection Panel

When you use the scope to view multiple signals, the Trace Selection panel appears. Use this panel to select which signal to measure. To open the Trace Selection panel:

- From the menu, select **Tools** > **Measurements** > **Trace Selection**.
- Open a measurement panel.

- 

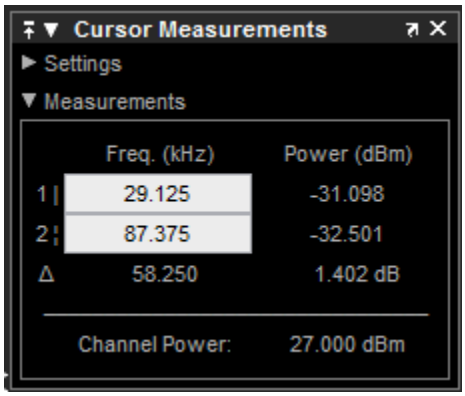
Cursor Measurements Panel

The **Cursor Measurements** panel displays screen cursors. The panel provides two types of cursors for measuring signals. Waveform cursors are vertical cursors that track along the signal. Screen cursors are both horizontal and vertical cursors that you can place anywhere in the display.

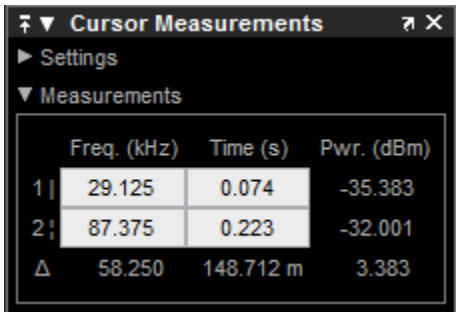
Note If a data point in your signal has more than one value, the cursor measurement at that point is undefined and no cursor value is displayed.

In the Scope menu, select **Tools > Measurements > Cursor Measurements**. Alternatively, in the Scope toolbar, click the Cursor Measurements button.

The **Cursor Measurements** panel for the spectrum and dual view:

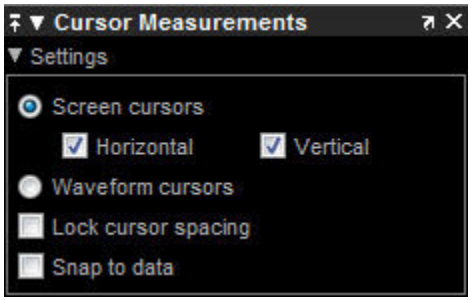


The **Cursor Measurements** panel for the spectrogram view. You must pause the spectrogram display before you can use cursors.



You can use the mouse or the left and right arrow keys to move vertical or waveform cursors and the up and down arrow keys for horizontal cursors.

In the **Settings** pane, you can modify the type of screen cursors used for calculating measurements. When more than one signal is displayed, you can assign cursors to each trace individually.



- **Screen Cursors** — Shows screen cursors (for spectrum and dual view only).
- **Horizontal** — Shows horizontal screen cursors (for spectrum and dual view only).
- **Vertical** — Shows vertical screen cursors (for spectrum and dual view only).
- **Waveform Cursors** — Shows cursors that attach to the input signals (for spectrum and dual view only).
- **Lock Cursor Spacing** — Locks the frequency difference between the two cursors.
- **Snap to Data** — Positions the cursors on signal data points.

The **Measurements** pane displays the frequency (Hz) , time (s), and power (dBm) value measurements. Time is displayed only in spectrogram mode. **Channel Power** shows the total power between the cursors.

- **1** — Shows or enables you to modify the frequency, time (for spectrograms only), or both, at cursor number one.
- **2** — Shows or enables you to modify the frequency, time (for spectrograms only), or both, at cursor number two.
- **Δ** — Shows the absolute value of the difference in the frequency, time (for spectrograms only), or both, and power between cursor number one and cursor number two.
- **Channel Power** — Shows the total power in the channel defined by the cursors.

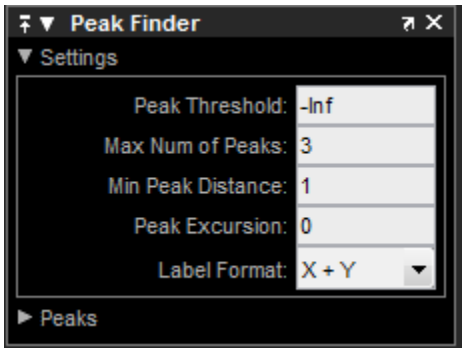
The letter after the value associated with a measurement indicates the abbreviation for the appropriate International System of Units (SI) prefix.

Peak Finder Panel

The **Peak Finder** panel displays the maxima, showing the *x*-axis values at which they occur. Peaks are defined as a local maximum where lower values are present on both sides of a peak. Endpoints are not considered peaks. This panel allows you to modify the settings for peak threshold, maximum number of peaks, and peak excursion.

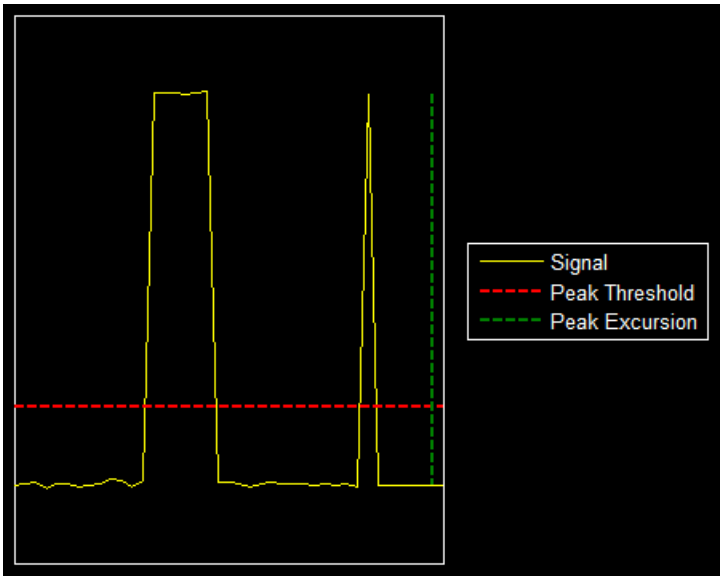
- From the menu, select **Tools** > **Measurements** > **Peak Finder**.
- On the toolbar, click the Peak Finder button.

The **Settings** pane enables you to modify the parameters used to calculate the peak values within the displayed portion of the input signal. For more information on the algorithms this pane uses, see the `findpeaks` function reference.



Properties to set:

- **Peak Threshold** — The level above which peaks are detected. This setting is equivalent to the `MINPEAKHEIGHT` parameter, which you can set when you run the `findpeaks` function.
- **Max Num of Peaks** — The maximum number of peaks to show. The value you enter must be a scalar integer from 1 through 99. This setting is equivalent to the `NPEAKS` parameter, which you can set when you run the `findpeaks` function.
- **Min Peaks Distance** — The minimum number of samples between adjacent peaks. This setting is equivalent to the `MINPEAKDISTANCE` parameter, which you can set when you run the `findpeaks` function.
- **Peak Excursion** — The minimum height difference between a peak and its neighboring samples. Peak excursion is illustrated alongside peak threshold in the following figure.



The peak threshold is a minimum value necessary for a sample value to be a peak. The peak excursion is the minimum difference between a peak sample and the samples to its left and right in the time domain. In the figure, the green vertical line illustrates the lesser of the two height differences between the labeled peak and its neighboring samples. This height difference must be greater than the **Peak Excursion** value for the labeled peak to be classified as a peak. Compare this setting to peak threshold, which is illustrated by the red horizontal line. The amplitude must be above this horizontal line for the labeled peak to be classified as a peak.

The peak excursion setting is equivalent to the `THRESHOLD` parameter, which you can set when you run the `findpeaks` function.

- **Label Format** — The coordinates to display next to the calculated peak values on the plot. To see peak values, you must first expand the **Peaks** pane and select the check boxes associated with individual peaks of interest. By default, both *x*-axis and *y*-axis values are displayed on the plot. Select which axes values you want to display next to each peak symbol on the display.
  - `X+Y` — Display both *x*-axis and *y*-axis values.
  - `X` — Display only *x*-axis values.
  - `Y` — Display only *y*-axis values.

The **Peaks** pane displays the largest calculated peak values. It also shows the coordinates at which the peaks occur, using the parameters you define in the **Settings** pane. You set the **Max Num of Peaks** parameter to specify the number of peaks shown in the list.

The numerical values displayed in the **Value** column are equivalent to the `pks` output argument returned when you run the `findpeaks` function. The numerical values displayed in the second column are similar to the `locs` output argument returned when you run the `findpeaks` function.

The Peak Finder displays the peak values in the **Peaks** pane. By default, the **Peak Finder** panel displays the largest calculated peak values in the **Peaks** pane in decreasing order of peak height.

Use the check boxes to control which peak values are shown on the display. By default, all check boxes are cleared and the **Peak Finder** panel hides all the peak values. To show or hide all the peak values on the display, use the check box in the top-left corner of the **Peaks** pane.

The Peaks are valid for any units of the input signal. The letter after the value associated with each measurement indicates the abbreviation for the appropriate International System of Units (SI) prefix, such as *m* for *milli-*. For example, if the input signal is measured in volts, an *m* next to a measurement value indicates that this value is in units of millivolts.

Channel Measurements Panel

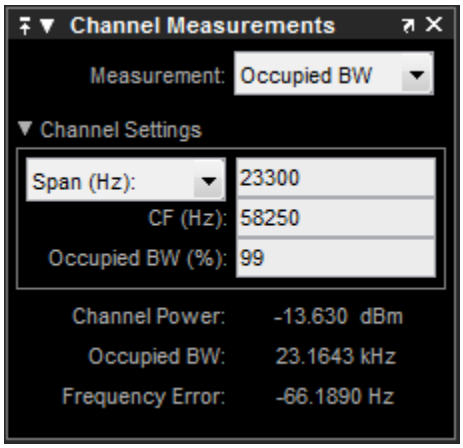
The **Channel Measurements** panel displays occupied bandwidth or adjacent channel power ratio (ACPR) measurements.

- From the menu, select **Tools > Measurements > Channel Measurements**.
- On the toolbar, click the Channel Measurements button.

In addition to the measurements, the **Channel Measurements** panel has an expandable **Channel Settings** pane.

- **Measurement** — The type of measurement data to display. Available options are `Occupied BW` or `ACPR`. See “Algorithms” for information on how Occupied BW is calculated. ACPR is the adjacent channel power ratio, which is the ratio of the main channel power to the adjacent channel power.

When you select `Occupied BW` as the **Measurement**, the following fields appear.

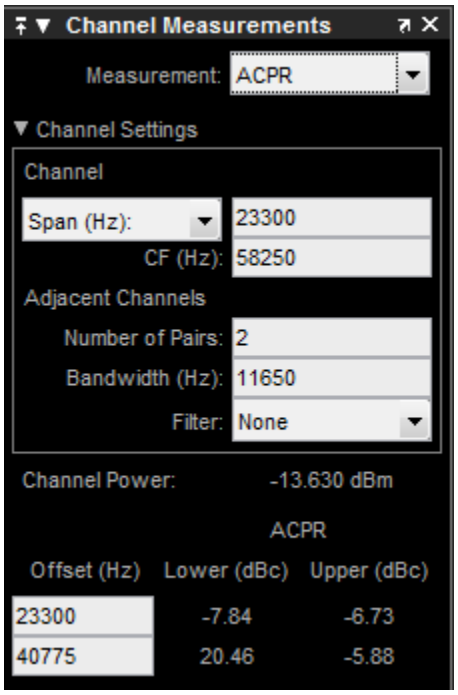


- **Channel Settings** — Modify the parameters for calculating the channel measurements.

  Channel Settings for Occupied BW

  - Select the frequency span of the channel, `Span(Hz)`, and specify the center frequency **CF (Hz)** of the channel. Alternatively, select the starting frequency, `FStart(Hz)`, and specify the starting frequency and ending frequency (**FStop (Hz)**) values of the channel.
  - **CF (Hz)** — The center frequency of the channel.
  - **Occupied BW (%)** — The percentage of the total integrated power of the spectrum centered on the selected channel frequency over which to compute the occupied bandwidth.
- **Channel Power** — The total power in the channel.
- **Occupied BW** — The bandwidth containing the specified **Occupied BW (%)** of the total power of the spectrum. This setting is available only if you select `Occupied BW` as the **Measurement** type.
- **Frequency Error** — The difference between the center of the occupied band and the center frequency (**CF**) of the channel. This setting is available only if you select `Occupied BW` as the **Measurement** type.

When you select `ACPR` as the **Measurement**, the following fields appear.



- **Channel Settings** — Enables you to modify the parameters for calculating the channel measurements.

  Channel Settings for ACPR

  - Select the frequency span of the channel, `Span (Hz)`, and specify the center frequency **CF (Hz)** of the channel. Alternatively, select the starting frequency, `FStart(Hz)`, and specify the starting frequency and ending frequency (**FStop (Hz)**) values of the channel.
  - **CF (Hz)** — The center frequency of the channel.
  - **Number of Pairs** — The number of pairs of adjacent channels.
  - **Bandwidth (Hz)** — The bandwidth of the adjacent channels.
  - **Filter** — The filter to use for both main and adjacent channels. Available filters are `None`, `Gaussian`, and `RRC` (root-raised cosine).
- **Channel Power** — The total power in the channel.
- **Offset (Hz)** — The center frequency of the adjacent channel with respect to the center frequency of the main channel. This setting is available only if you select `ACPR` as the **Measurement** type.
- **Lower (dBc)** — The power ratio of the lower sideband to the main channel. This setting is available only if you select `ACPR` as the **Measurement** type.
- **Upper (dBc)** — The power ratio of the upper sideband to the main channel. This setting is available only if you select `ACPR` as the **Measurement** type.

Distortion Measurements Panel

The **Distortion Measurements** panel displays harmonic distortion and intermodulation distortion measurements.

- From the menu, select **Tools > Measurements > Distortion Measurements**.

- On the toolbar, click the Distortion Measurements button.

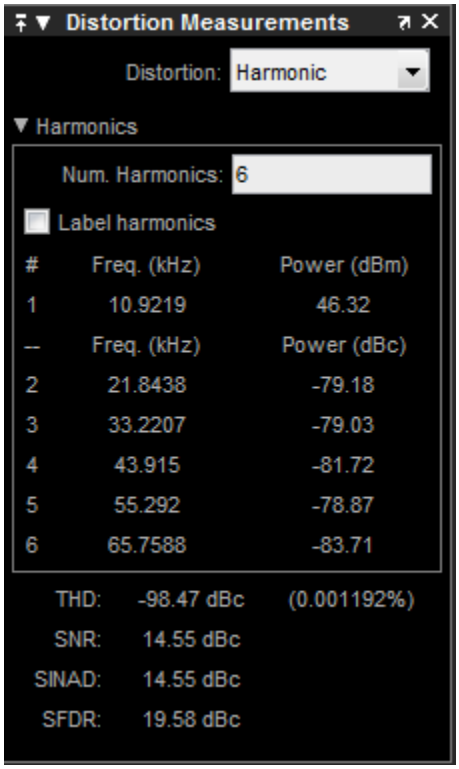
The **Distortion Measurements** panel has an expandable **Harmonics** pane, which shows measurement results for the specified number of harmonics.

Note For an accurate measurement, ensure that the fundamental signal (for harmonics) or primary tones (for intermodulation) is larger than any spurious or harmonic content. To do so, you may need to adjust the resolution bandwidth (`RBW`) of the spectrum analyzer. Make sure that the bandwidth is low enough to isolate the signal and harmonics from spurious and noise content. In general, you should set the RBW so that there is at least a 10dB separation between the peaks of the sinusoids and the noise floor. You may also need to select a different spectral window to obtain a valid measurement.

- **Distortion** — The type of distortion measurements to display. Available options are `Harmonic` or `Intermodulation`. Select `Harmonic` if your system input is a single sinusoid. Select `Intermodulation` if your system input is two equal amplitude sinusoids. Intermodulation can help you determine distortion when only a small portion of the available bandwidth will be used.

  See “Distortion Measurements” for information on how distortion measurements are calculated.

When you select `Harmonic` as the **Distortion**, the following fields appear.

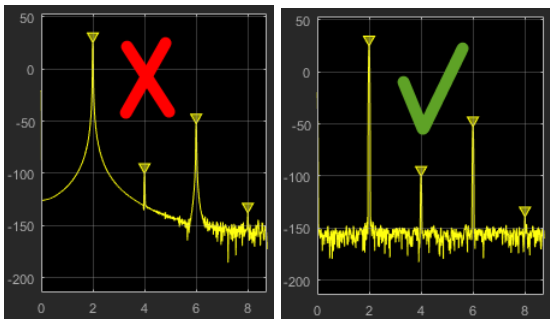


The harmonic distortion measurement automatically locates the largest sinusoidal component (fundamental signal frequency). It then computes the harmonic frequencies and power in each harmonic in your signal. Any DC component is ignored. Any harmonics that are outside the spectrum

analyzer's frequency span are not included in the measurements. Adjust your frequency span so that it includes all the desired harmonics.

Note To view the best harmonics, make sure that your fundamental frequency is set high enough to resolve the harmonics. However, this frequency should not be so high that aliasing occurs. For the best display of harmonic distortion, your plot should not show skirts, which indicate frequency leakage. Also, the noise floor should be visible.

For a better display, try a Kaiser window with a large sidelobe attenuation (e.g. between 100–300 db).



- **Num. Harmonics** — Number of harmonics to display, including the fundamental frequency. Valid values of **Num. Harmonics** are from 2 to 99. The default value is 6.
- **Label Harmonics** — Select **Label Harmonics** to add numerical labels to each harmonic in the spectrum display.
- **1** — The fundamental frequency, in hertz, and its power, in decibels of the measured power referenced to 1 milliwatt (dBm).
- **2, 3, ...** — The harmonics frequencies, in hertz, and their power in decibels relative to the carrier (dBc). If the harmonics are at the same level or exceed the fundamental frequency, reduce the input power.
- **THD** — The total harmonic distortion. This value represents the ratio of the power in the harmonics, D, to the power in the fundamental frequency, S. If the noise power is too high in relation to the harmonics, the THD value is not accurate. In this case, lower the resolution bandwidth or select a different spectral window.

  $$THD = 10 \cdot \log_{10}(D/S)$$

- **SNR** — Signal-to-noise ratio (SNR). This value represents the ratio of power in the fundamental frequency, S, to the power of all nonharmonic content, N, including spurious signals, in decibels relative to the carrier (dBc).

  $$SNR = 10 \cdot \log_{10}(S/N)$$

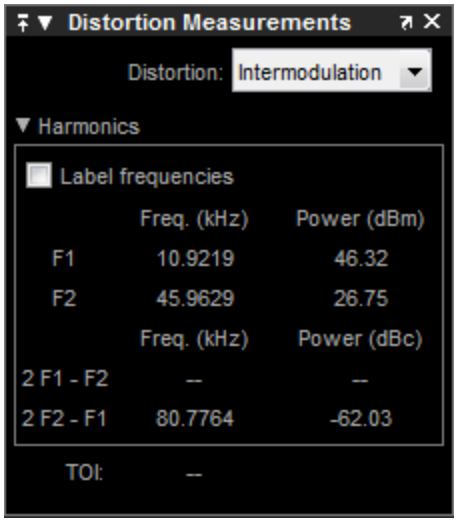
  If you see — as the reported SNR, the total non-harmonic content of your signal is less than 30% of the total signal.
- **SINAD** — Signal-to-noise-and-distortion. This value represents the ratio of the power in the fundamental frequency, S to all other content (including noise, N, and harmonic distortion, D), in decibels relative to the carrier (dBc).

  $$SINAD = 10 \cdot \log_{10}\left(\frac{S}{N+D}\right)$$

- **SFDR** — Spurious free dynamic range (SFDR). This value represents the ratio of the power in the fundamental frequency, *S*, to power of the largest spurious signal, *R*, regardless of where it falls in the frequency spectrum. The worst spurious signal may or may not be a harmonic of the original signal. SFDR represents the smallest value of a signal that can be distinguished from a large interfering signal. SFDR includes harmonics.

  $$SNR = 10 \cdot \log_{10}(S/R)$$

When you select `Intermodulation` as the **Distortion**, the following fields appear.



The intermodulation distortion measurement automatically locates the fundamental, first-order frequencies (F1 and F2). It then computes the frequencies of the third-order intermodulation products (2*F1-F2 and 2*F2-F1).

- **Label frequencies** — Select **Label frequencies** to add numerical labels to the first-order intermodulation product and third-order frequencies in the spectrum analyzer display.
- **F1** — Lower fundamental first-order frequency
- **F2** — Upper fundamental first-order frequency
- **2F1 - F2** — Lower intermodulation product from third-order harmonics
- **2F2 - F1** — Upper intermodulation product from third-order harmonics
- **TOI** — Third-order intercept point. If the noise power is too high in relation to the harmonics, the TOI value will not be accurate. In this case, you should lower the resolution bandwidth or select a different spectral window. If the TOI has the same amplitude as the input two-tone signal, reduce the power of that input signal.

CCDF Measurements Panel

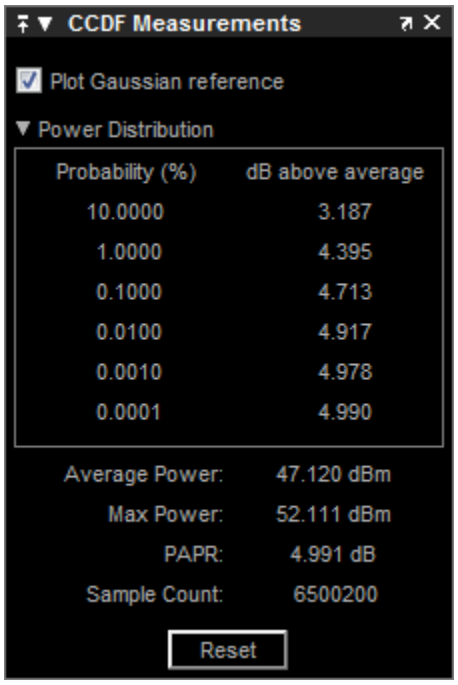
The **CCDF Measurements** panel displays complimentary cumulative distribution function measurements. CCDF measurements in this scope show the probability of a signal's instantaneous power being a specified level above the signal's average power. These measurements are useful indicators of a signal's dynamic range.

To compute the CCDF measurements, each input sample is quantized to 0.01 dB increments. Using a histogram 100 dB wide (10,000 points at 0.01 dB increments), the largest peak encountered is placed

in the last bin of the histogram. If a new peak is encountered, the histogram shifts to make room for that new peak.

To open this dialog box:

- From the menu, select **Tools > Measurements > CCDF Measurements**
- In the toolbar, click the CCDF Measurements button.



- **Plot Gaussian reference** — Show the Gaussian white noise reference signal on the plot.
- **Probability (%)** — The percentage of the signal that contains the power level above the value listed in the **dB above average** column
- **dB above average** — The expected minimum power level at the associated **Probability (%)**.
- **Average Power** — The average power level of the signal since the start of simulation or from the last reset.

  Max Power — The maximum power level of the signal since the start of simulation or from the last reset.
- **PAPR** — The ratio of the peak power to the average power of the signal. PAPR should be less that 100 dB to obtain accurate CCDF measurements. If PAPR is above 100 dB, only the highest 100 dB power levels are plotted in the display and shown in the distribution table.
- **Sample Count** — The total number of samples used to compute the CCDF.
- **Reset** — Clear all current CCDF measurements and restart.

Customize Visualization

To control the labels, minimum and maximum values, the legend, and gridlines, use the configuration properties. From the Spectrum Analyzer, select **View > Configuration Properties** or click the toolbar button .

In the **Style** dialog box, you can customize the style of spectrum display. This dialog box is not available for the spectrogram view. You can change the color of the spectrum plot, color of the background, and properties of the lines. To open this dialog box, select **View > Style**.

If you are viewing only the spectrum or the spectrogram, you only see the relevant options. For more details about these options, see “Configuration Properties” and “Style”.

Zoom and Pan

To zoom in and out of the plot, or pan to different area of the plot, use the zoom buttons in the toolbar or in the **Tools** menu.

You can set properties to zoom in/out automatically or scale the axes. In the Spectrum Analyzer menu, select **Tools > Axes Scaling**.

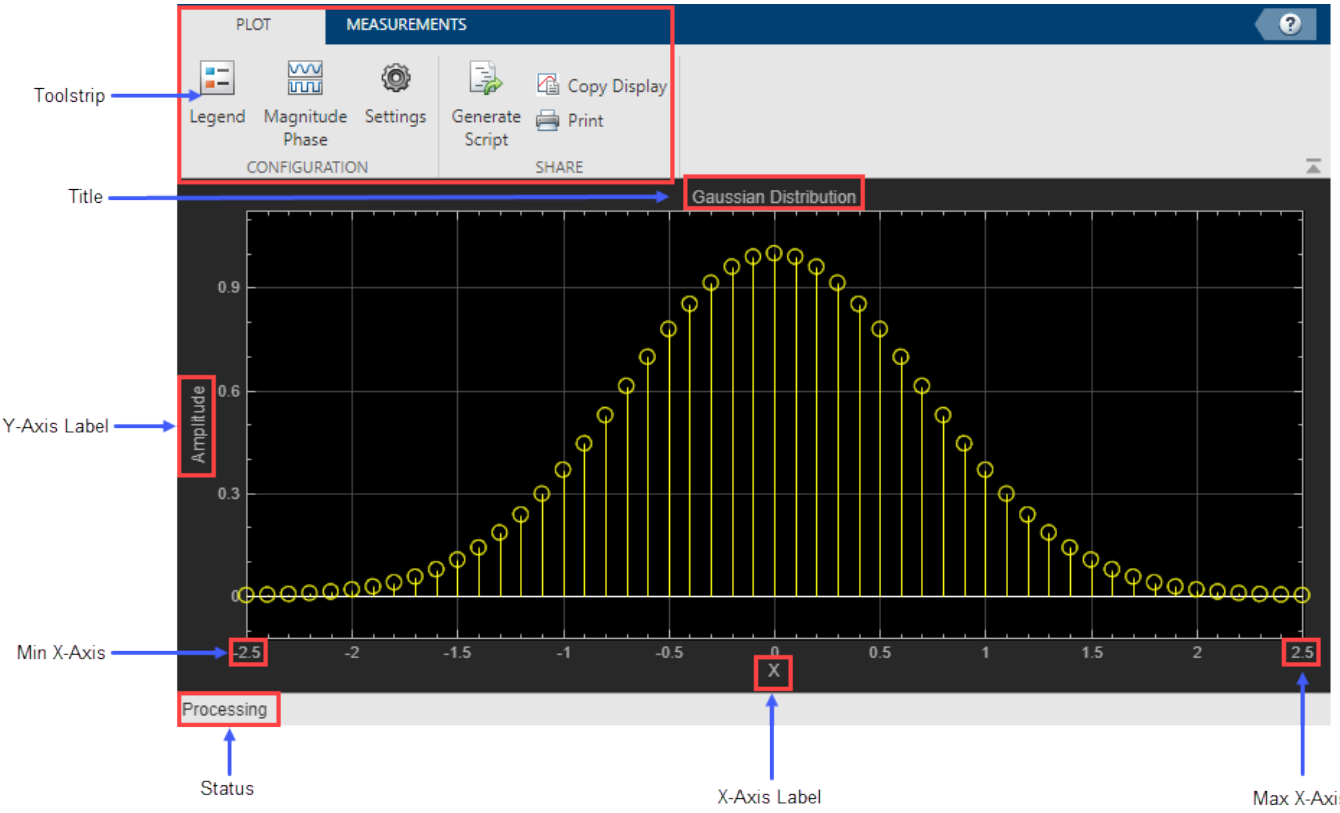
If you are viewing only the spectrum or the spectrogram, you see only the relevant options. If you are using the CCDF measurements, you will also see *x*-axis scaling options. For more details about these options, see “Axes Scaling”.

Configure Array Plot MATLAB Object

When you use the `dsp.ArrayPlot` object in MATLAB, you can configure many settings and tools from the interface. These sections show you how to use the Array Plot interface and the tools available.

Signal Display

This figure highlights the important aspects of the Array Plot window in MATLAB.



- **Min X-Axis** — Array Plot sets the minimum *x*-axis limit using the value of the `XOffset` property. To change the **X-offset** from the Array Plot window, click **Settings** and set the **X-offset**.
- **Max X-Axis** — Array Plot sets the maximum *x*-axis limit by summing the value of the **X-offset** parameter with the span of *x*-axis values. The relationship between the span of the x-axis data and the **Sample Increment** property is determined by this equation:

  $$x_{span} = \text{SampleIncrement} \times (\text{length}(x) - 1)$$

  To modify the `SampleIncrement` from the Array Plot window, click **Settings** and set the **Sample Increment**. If you set `SampleIncrement` to `0.1` and the input signal data has 51 samples, the scope displays values on the *x*-axis from `0` to `5`. If you also set the **X-offset** to `–2.5`, the scope displays values on the *x*-axis from `–2.5` to `2.5`. The values on the *x*-axis of the scope display remain the same throughout simulation.
- **Status** — Provides the current status of the plot. The status can be:

  - `Processing` — Occurs after you run the `step` method and before you run the `release` method.
  - `Stopped` — Occurs after you construct the scope object and before you first call the object. This status also occurs after you call `release`.
- **Title**, **X-Axis Label**, **Y-Axis Label** — You can customize the title and axes labels from **Settings** or by using the `Title`, `YLabel`, and `XLabel` properties.
- **Toolstrip** — The **Plot** tab contains buttons and settings to customize and share the array plot. The **Measurements** tab contains buttons and settings to turn on different measurement tools. Use the pin button to keep the toolstrip showing or the arrow button to hide the toolstrip.

Multiple Signal Names and Colors

By default, if the input signal has multiple channels, the scope uses an index number to identify each channel of that signal. For example, a 2-channel signal would have the following default names in the channel legend: `Channel 1`, `Channel 2`. To show the legend, on the **Plot** tab, click **Legend**. If there are a total of seven input channels, the legend appears in the display as:

By default, the scope has a black axes background and chooses line colors for each channel in a manner similar to the Simulink Scope block. When the scope axes background is black, it assigns each channel of each input signal a line color in the order shown in the above figure. If there are more than seven channels, then the scope repeats this order to assign line colors to the remaining channels. When the axes background is not black, the signals are colored in the following order:

To choose line colors or background colors, on the **Plot** tab click **Settings**.Use the **Axes** color drop-down to change the background of the plot. Click **Line** to choose a line to change, and the **Color** drop-down to change the line color of the selected line.

Configure Plot Settings

On the **Plot** tab, the Configuration section allows you to modify the plot.

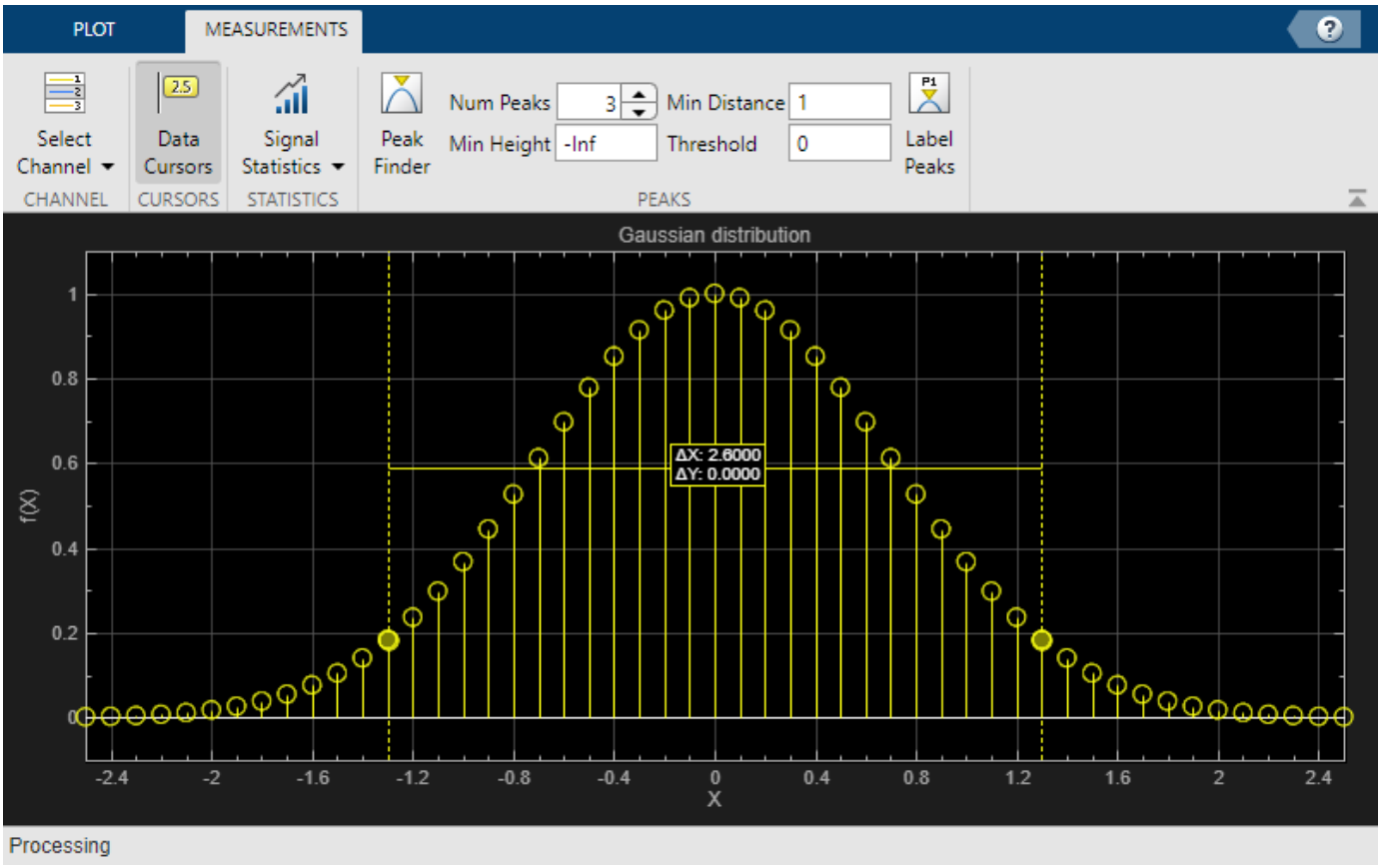
- The **Legend** button turns the legend on or off. When you show the legend, you can control which signals are shown. If you click a signal name in the legend, the signal is hidden from the plot and shown in grey on the legend. To redisplay the signal, click on the signal name again. This button corresponds to the `ShowLegend` property in the object.

- The **Magnitude Phase** button splits the magnitude and phase of the input signal and plots them on two separate axes within the same window. This button corresponds to the `PlotAsMagnitudePhase` property in the object.
- The **Settings** button opens the settings window which allows you to customize the *x*-axis, *y*-limits, plot labels, and signal colors.

Use dsp.ArrayPlot Measurements

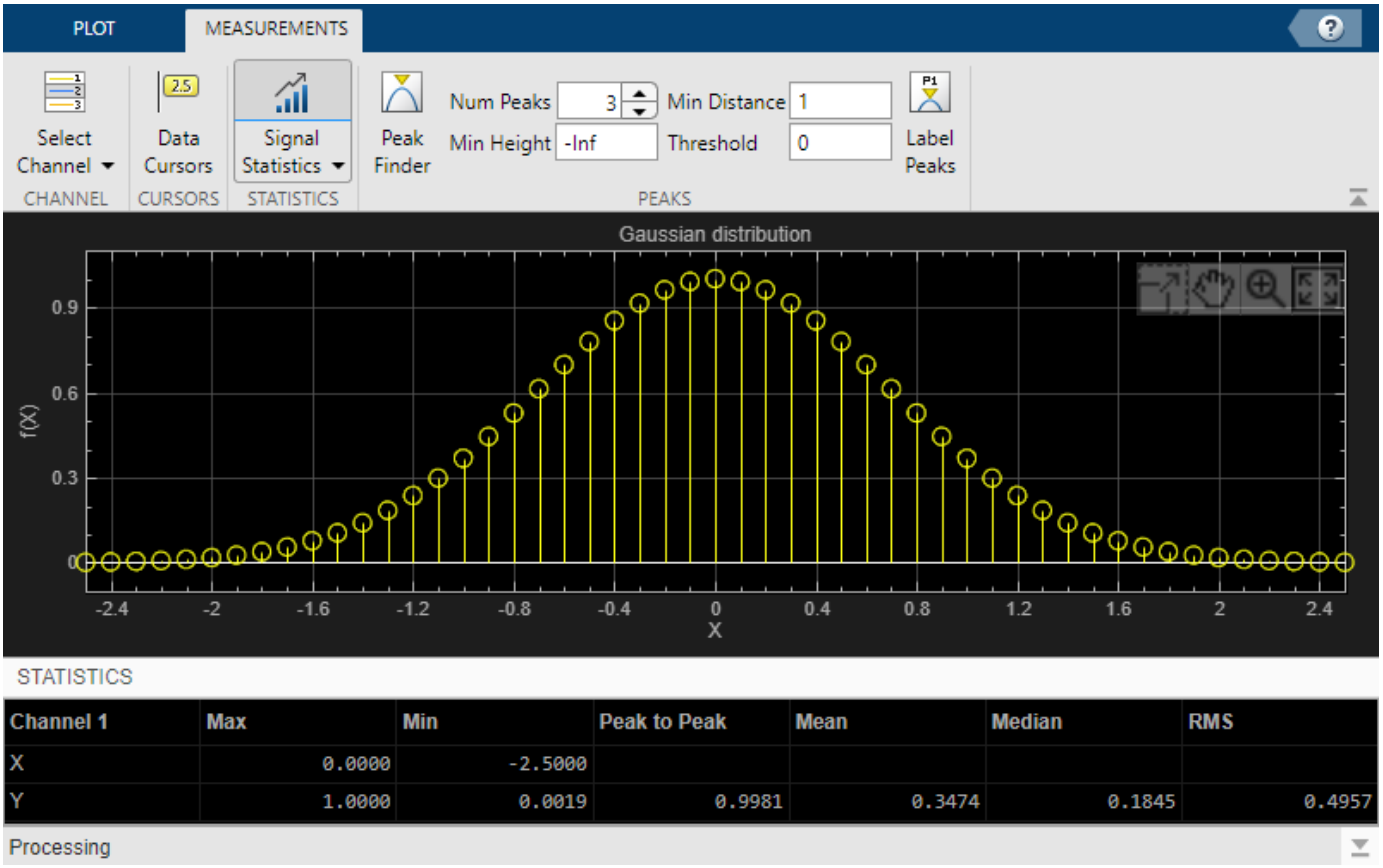
All measurements are made for a specified channel. By default, measurements are applied to the first channel. To change which channel is being measured, use the **Select Channel** drop-down in the **Measurements** tab.

Data Cursors



Use the **Data Cursors** button to display screen cursors. The cursors are vertical cursors that track along the selected signal. Between the two cursors, the difference between the *x*- and *y*-values of the signal at the two cursor points is displayed.

Signal Statistics

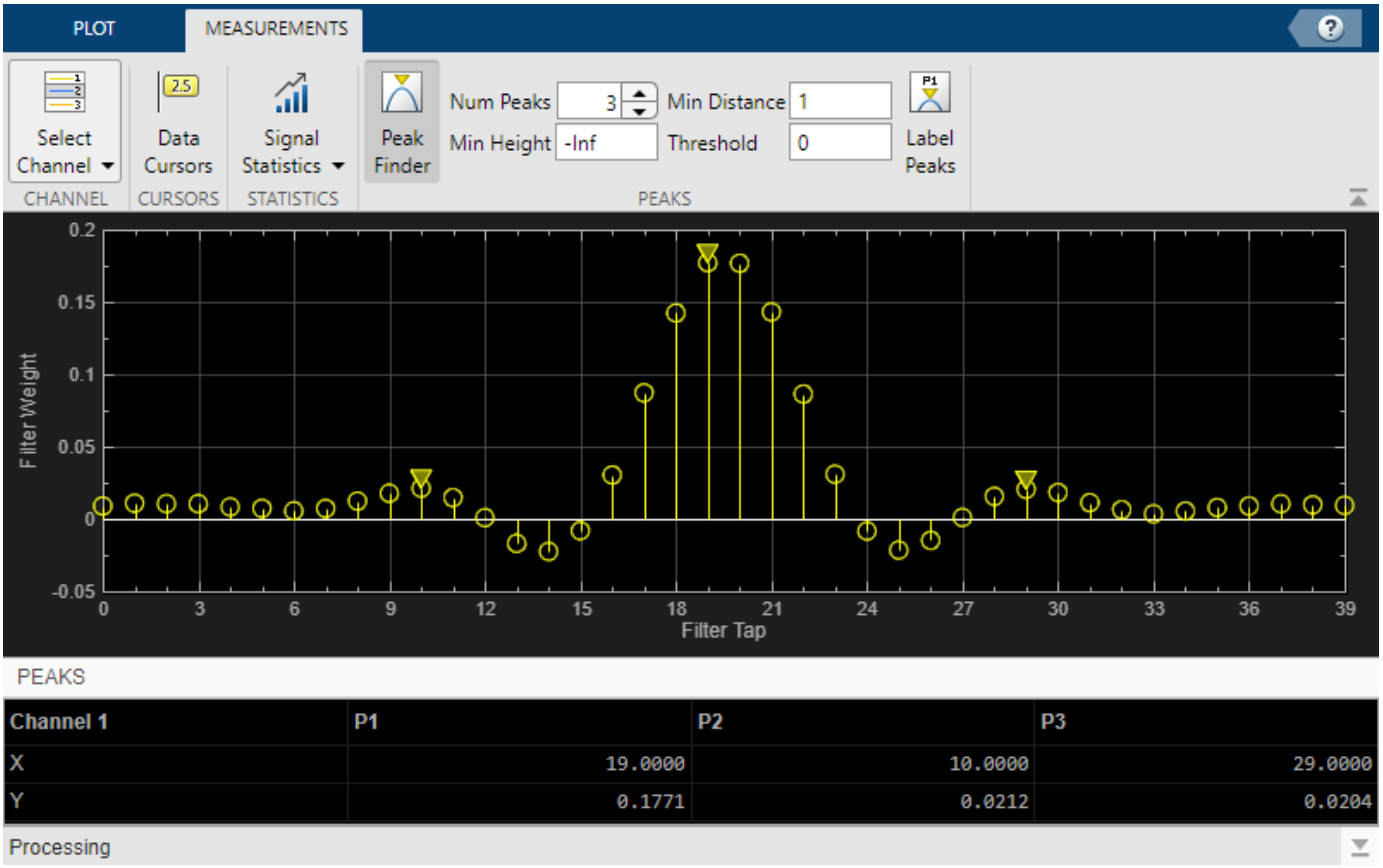


Use the **Signal Statistics** button to display various statistics about the selected signal at the bottom of the array plot window. You can hide or show the **Statistics** panel using the arrow button in the bottom right of the panel.

- **Max** — Maximum value within the displayed portion of the input signal.
- **Min** — Minimum value within the displayed portion of the input signal.
- **Peak to Peak** — Difference between the maximum and minimum values within the displayed portion of the input signal.
- **Mean** — Average or mean of all the values within the displayed portion of the input signal.
- **Median** — Median value within the displayed portion of the input signal.
- **RMS** — Root mean squared of the input signal.

To customize which statistics are shown and computed, use the **Signal Statistics** drop-down.

Peak Finder



Use the **Peak Finder** button to display peak values for the selected signal. Peaks are defined as a local maximum where lower values are present on both sides of a peak. Endpoints are not considered peaks. For more information on the algorithms used, see the `findpeaks` function reference.

When you turn on the peak finder measurements, an arrow appears on the plot at each maxima and a Peaks panel appears at the bottom of the array plot window showing the *x* and *y* values at each peak.

You can customize several peak finder settings:

- **Num Peaks** — The number of peaks to show. Must be a scalar integer from 1 through 99.
- **Min Height** — The minimum height difference between a peak and its neighboring samples.
- **Min Distance** — The minimum number of samples between adjacent peaks.
- **Threshold** — The level above which peaks are detected.
- **Label Peaks** — Show labels (**P1**, **P2**, ...) above the arrows on the plot.

Share or Save the Array Plot

If you want to save the array plot for future use or share it with others, use the buttons in the Share section of the **Plot** tab.

- **Generate Script** — Generate a script to regenerate your array plot with the same settings. An editor window opens with the code required to recreate your `dsp.ArrayPlot`.

- **Copy Display** — Copy the display to your clipboard. You can paste the image in another program to save or share.
- **Print** — Opens a print dialog box from which you can print out the plot image.

Scale Axes

To scale the plot axes, you can use the mouse to pan around the axes and the scroll button on your mouse to zoom in and out of the plot. Additionally, you can use the buttons that appear when you hover over the plot window.

- — Maximize the axes, hiding all labels and insetting the axes values.
- — Zoom in to the plot.
- — Pan around the axes.
- — Autoscale the axes to fit the shown data.

See Also

`dsp.ArrayPlot`

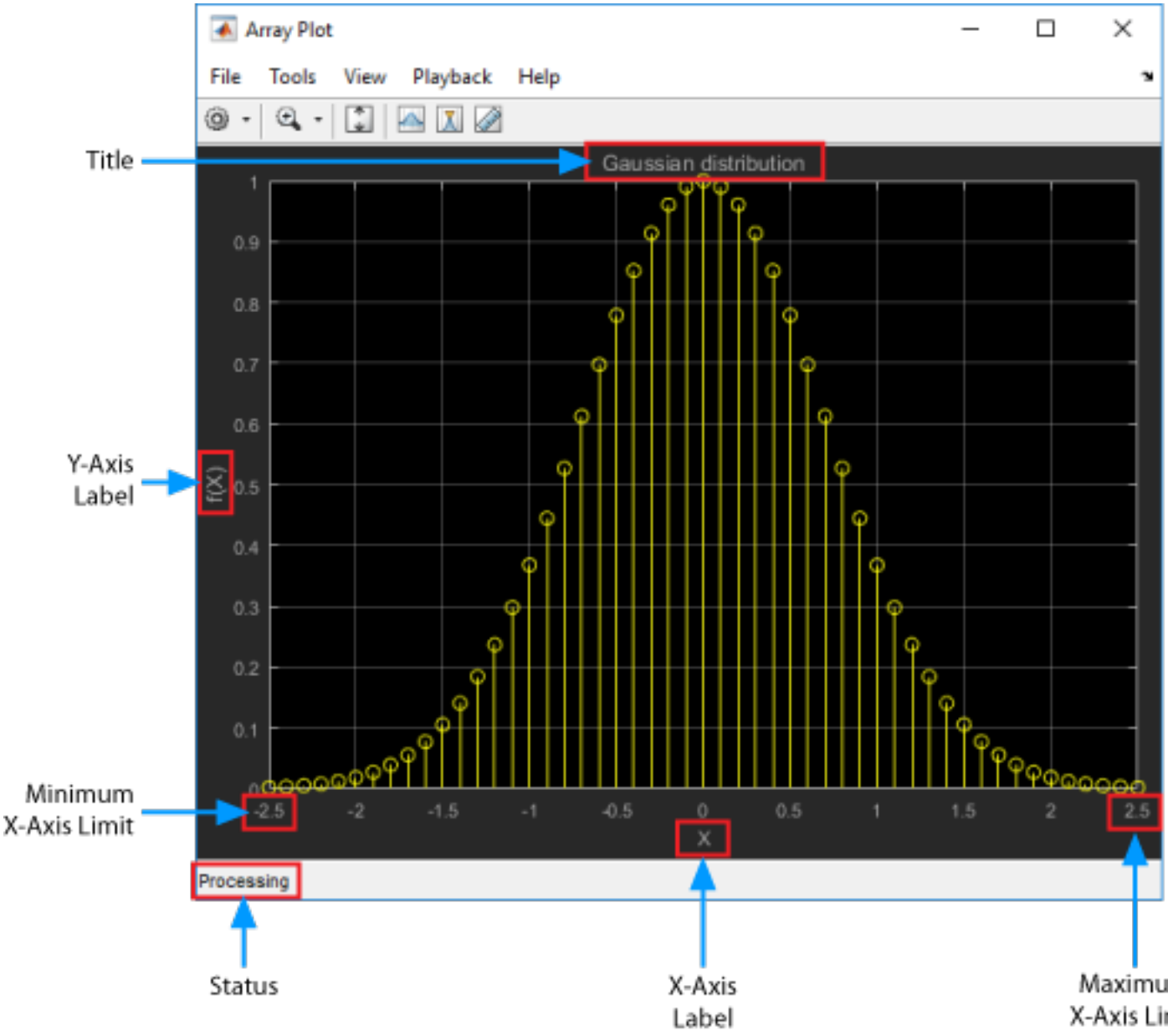
More About

- “Visualize Central Limit Theorem in Array Plot” on page 24-16
- “Reconstruction Through Two-Channel Filter Bank”

Configure Array Plot Block

Signal Display

The following figure highlights the important aspects of the Array Plot window.



- **Minimum x-axis limit** — Array Plot sets the minimum x-axis limit using the value of the **X-offset** property. To change the **X-offset**, select **View > Configuration Properties**. In the Configuration Properties, on the **Main** tab, modify the **X-offset**.
- **Maximum x-axis limit** — Array Plot sets the maximum x-axis limit by summing the value of **X-offset** parameter with the span of x-axis values. The relationship between the span of the x-axis data to the `SampleIncrement` property is determined by this equation:

x_{span} = SampleIncrement × (length(x) − 1)

If you set `SampleIncrement` to `0.1` and the input signal data has 51 samples. The scope displays values on the x-axis from `0` to `5`. If you also set the **X-offset** to `–2.5`, the scope displays values on the x-axis from `–2.5` to `2.5`. The values on the x-axis of the scope display remain the same throughout simulation.

- **Status** — Provides the current status of the plot. The status can be either of the following conditions:
  - `Processing` — Occurs after you run the model and before the simulation ends.
  - `Stopped` — Occurs after you construct the scope object and before you run the simulation. This status also occurs after the simulation ends.

Multiple Signal Names and Colors

By default, if the input signal has multiple channels, the scope uses an index number to identify each channel of that signal. For example, a 2-channel signal would have the following default names in the channel legend: `Channel 1`, `Channel 2`. To show the legend, select **View > Configuration Properties**, click the **Display** tab, and select the **Show Legend** check box. If there are a total of seven input channels, the following legend appears in the display.

By default, the scope has a black axes background and chooses line colors for each channel in a manner similar to the Simulink Scope block. When the scope axes background is black, it assigns each channel of each input signal a line color in the order shown in the above figure.

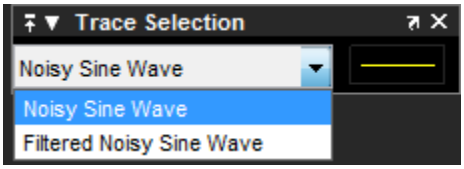
If there are more than seven channels, then the scope repeats this order to assign line colors to the remaining channels. To choose line colors for each channel, change the axes background color to any color except black. To change the axes background color to white, select **View > Style**, click the Axes background color button ( ), and select white from the color palette. Run the simulation again. The following legend appears in the display. This image shows the color order when the background is not black.

Array Plot Measurement Panels

The Measurements panels are the panels that appear to the right side of the Array Plot figure. These panels are labeled **Trace selection**, **Cursor Measurements**, **Signal Statistics**, and **Peak Finder**.

Trace Selection Panel

When you use the scope to view multiple signals, the Trace Selection panel appears. Use this panel to select which signal to measure. To open the Trace Selection panel:

- From the menu, select **Tools** > **Measurements** > **Trace Selection**.
- Open a measurement panel.
- 

Cursor Measurements Panel

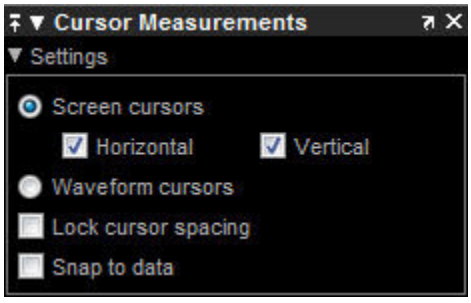
The **Cursor Measurements** panel displays screen cursors. The panel provides two types of cursors for measuring signals. Waveform cursors are vertical cursors that track along the signal. Screen cursors are both horizontal and vertical cursors that you can place anywhere in the display.

Note If a data point in your signal has more than one value, the cursor measurement at that point is undefined and no cursor value is displayed.

Display screen cursors with signal times and values. To open the Cursor measurements panel:

- From the menu, select **Tools** > **Measurements** > **Cursor Measurements**.
- On the toolbar, click the Cursor Measurements button.

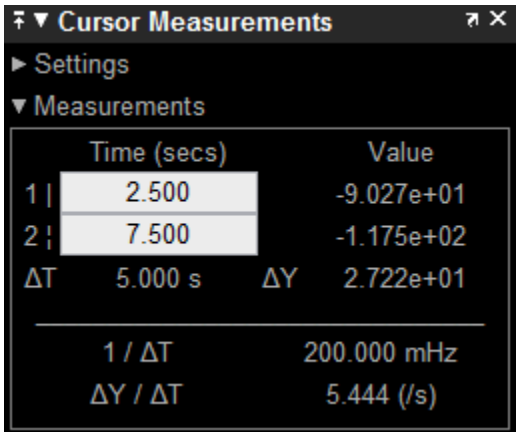
In the **Settings** pane, you can modify the type of screen cursors used for calculating measurements. When more than one signal is displayed, you can assign cursors to each trace individually.



- **Screen Cursors** — Shows screen cursors (for spectrum and dual view only).
- **Horizontal** — Shows horizontal screen cursors (for spectrum and dual view only).
- **Vertical** — Shows vertical screen cursors (for spectrum and dual view only).

- **Waveform Cursors** — Shows cursors that attach to the input signals (for spectrum and dual view only).
- **Lock Cursor Spacing** — Locks the frequency difference between the two cursors.
- **Snap to Data** — Positions the cursors on signal data points.

The Measurements pane displays time and value measurements.

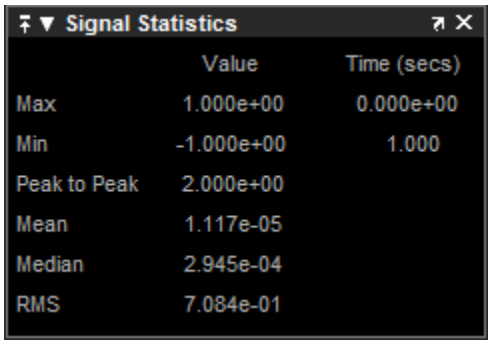


- **1** — View or modify the time or value at cursor number one (solid line cursor).
- **2** — View or modify the time or value at cursor number two (dashed line cursor).
- **ΔT** or **ΔX** — Shows the absolute value of the time (*x*-axis) difference between cursor number one and cursor number two.
- **ΔY** — Shows the absolute value of the signal amplitude difference between cursor number one and cursor number two.
- **1/ΔT** or **1/ΔX** — Shows the rate. The reciprocal of the absolute value of the difference in the times (*x*-axis) between cursor number one and cursor number two.
- **ΔY/ΔT** or **ΔY/ΔX** — Shows the slope. The ratio of the absolute value of the difference in signal amplitudes between cursors to the absolute value of the difference in the times (*x*-axis) between cursors.

Signal Statistics Panel

Display signal statistics for the signal selected in the **Trace Selection** panel. To open the Signal Statistics panel:

- From the menu, select **Tools > Measurements > Signal Statistics**.
- On the toolbar, click the Signal Statistics button.



The statistics shown are:

- **Max** — Maximum or largest value within the displayed portion of the input signal.
- **Min** — Minimum or smallest value within the displayed portion of the input signal.
- **Peak to Peak** — Difference between the maximum and minimum values within the displayed portion of the input signal.
- **Mean** — Average or mean of all the values within the displayed portion of the input signal.
- **Median** — Median value within the displayed portion of the input signal.
- **RMS** — Root mean squared of the input signal.

When you use the zoom options in the scope, the Signal Statistics measurements automatically adjust to the time range shown in the display. In the scope toolbar, click the **Zoom In** or **Zoom X** button to constrict the *x*-axis range of the display, and the statistics shown reflect this time range. For example, you can zoom in on one pulse to make the **Signal Statistics** panel display information about only that particular pulse.

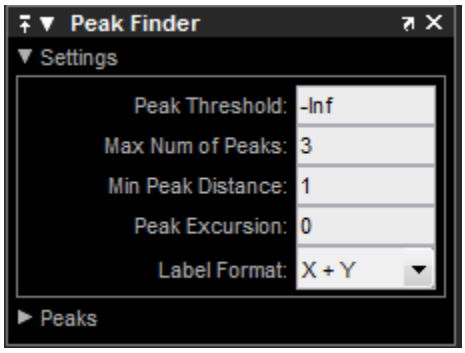
The Signal Statistics measurements are valid for any units of the input signal. The letter after the value associated with each measurement represents the appropriate International System of Units (SI) prefix, such as *m* for *milli-*. For example, if the input signal is measured in volts, an *m* next to a measurement value indicates that this value is in units of millivolts.

Peak Finder Panel

The **Peak Finder** panel displays the maxima, showing the *x*-axis values at which they occur. Peaks are defined as a local maximum where lower values are present on both sides of a peak. Endpoints are not considered peaks. This panel allows you to modify the settings for peak threshold, maximum number of peaks, and peak excursion.

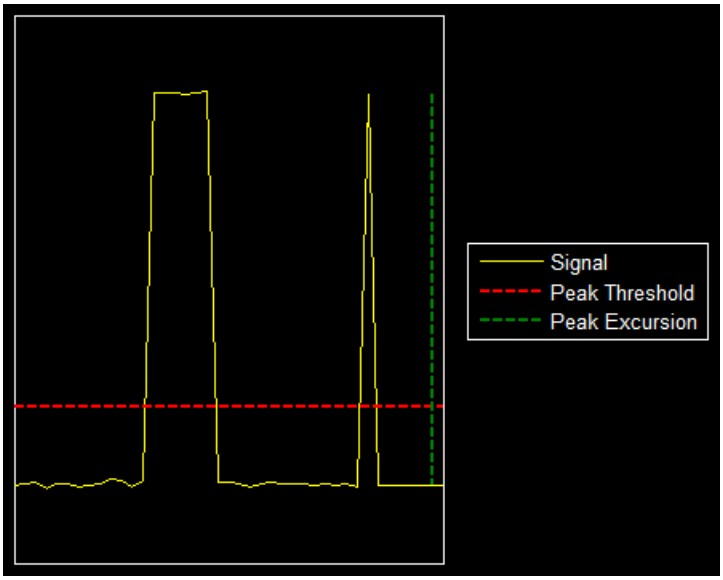
- From the menu, select **Tools > Measurements > Peak Finder**.
- On the toolbar, click the Peak Finder button.

The **Settings** pane enables you to modify the parameters used to calculate the peak values within the displayed portion of the input signal. For more information on the algorithms this pane uses, see the `findpeaks` function reference.



Properties to set:

- **Peak Threshold** — The level above which peaks are detected. This setting is equivalent to the `MINPEAKHEIGHT` parameter, which you can set when you run the `findpeaks` function.
- **Max Num of Peaks** — The maximum number of peaks to show. The value you enter must be a scalar integer from 1 through 99. This setting is equivalent to the `NPEAKS` parameter, which you can set when you run the `findpeaks` function.
- **Min Peaks Distance** — The minimum number of samples between adjacent peaks. This setting is equivalent to the `MINPEAKDISTANCE` parameter, which you can set when you run the `findpeaks` function.
- **Peak Excursion** — The minimum height difference between a peak and its neighboring samples. Peak excursion is illustrated alongside peak threshold in the following figure.



The peak threshold is a minimum value necessary for a sample value to be a peak. The peak excursion is the minimum difference between a peak sample and the samples to its left and right in the time domain. In the figure, the green vertical line illustrates the lesser of the two height differences between the labeled peak and its neighboring samples. This height difference must be greater than the **Peak Excursion** value for the labeled peak to be classified as a peak. Compare this setting to peak threshold, which is illustrated by the red horizontal line. The amplitude must be above this horizontal line for the labeled peak to be classified as a peak.

The peak excursion setting is equivalent to the `THRESHOLD` parameter, which you can set when you run the `findpeaks` function.

- **Label Format** — The coordinates to display next to the calculated peak values on the plot. To see peak values, you must first expand the **Peaks** pane and select the check boxes associated with individual peaks of interest. By default, both *x*-axis and *y*-axis values are displayed on the plot. Select which axes values you want to display next to each peak symbol on the display.

  - `X+Y` — Display both *x*-axis and *y*-axis values.
  - `X` — Display only *x*-axis values.
  - `Y` — Display only *y*-axis values.

The **Peaks** pane displays the largest calculated peak values. It also shows the coordinates at which the peaks occur, using the parameters you define in the **Settings** pane. You set the **Max Num of Peaks** parameter to specify the number of peaks shown in the list.

The numerical values displayed in the **Value** column are equivalent to the `pks` output argument returned when you run the `findpeaks` function. The numerical values displayed in the second column are similar to the `locs` output argument returned when you run the `findpeaks` function.

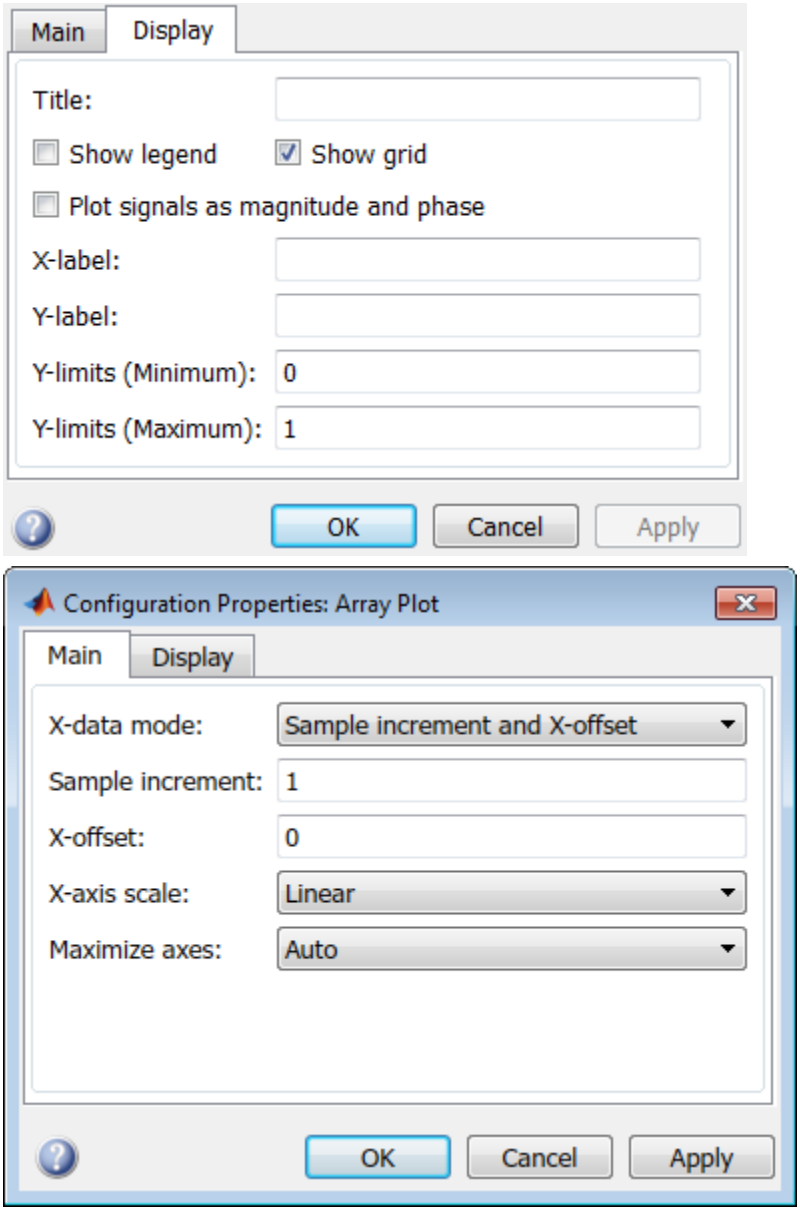
The Peak Finder displays the peak values in the **Peaks** pane. By default, the **Peak Finder** panel displays the largest calculated peak values in the **Peaks** pane in decreasing order of peak height.

Use the check boxes to control which peak values are shown on the display. By default, all check boxes are cleared and the **Peak Finder** panel hides all the peak values. To show or hide all the peak values on the display, use the check box in the top-left corner of the **Peaks** pane.

The Peaks are valid for any units of the input signal. The letter after the value associated with each measurement indicates the abbreviation for the appropriate International System of Units (SI) prefix, such as *m* for *milli-*. For example, if the input signal is measured in volts, an *m* next to a measurement value indicates that this value is in units of millivolts.

Configuration Properties

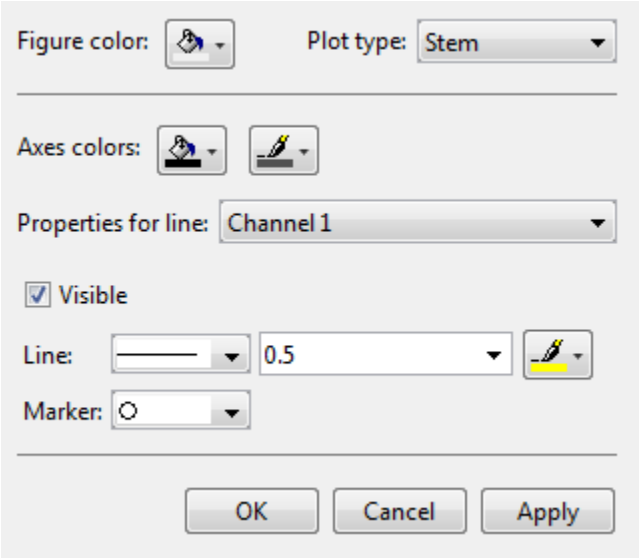
The Configuration Properties dialog box controls various properties about the Array Plot display. From the Array Plot menu, select **View > Configuration Properties** to open this dialog box.



For more details about the properties, see “Configuration Properties”.

Style Dialog Box

In the **Style** dialog box, you can customize the style of displays. You are able to change the color of the figure containing the displays, the background and foreground colors of display axes, and properties of lines in a display. From the scope menu, select **View > Style** to open this dialog box.



For more details about the properties, see “Style”.

Axes Scaling Properties

The Tools—Plot Navigation Properties dialog box allows you to automatically zoom in on and zoom out of your data. You can also scale the axes of the scope. In the scope menu, select **Tools > Axes Scaling > Axes Scaling Properties** to open this dialog box.

For more details about the properties, see “Axes Scaling Properties”.

Configure Time Scope

Signal Display

Time Scope uses the **Time span** and **Time display offset** parameters to determine the time range. To change the signal display settings, select **View > Configuration Properties** to bring up the Configuration Properties dialog box. Then, modify the values for the **Time span** and **Time display offset** parameters on the **Time** tab. For example, if you set the **Time span** to 25 seconds, the scope displays 25 seconds' worth of simulation data at a time. If you also set the **Time display offset** to 5 seconds, the scope displays values on the time axis from 5 to 30 seconds. The values on the time axis of the Time Scope display remain the same throughout simulation.

To communicate the simulation time that corresponds to the current display, the scope uses the *Time units*, *Time offset*, and *Simulation time* indicators on the scope window. The following figure highlights these and other important aspects of the Time Scope window.

Time Indicators

- *Minimum time-axis limit* — The Time Scope sets the minimum time-axis limit using the value of the **Time display offset** parameter on the **Main** tab of the Configuration Properties dialog box. If you specify a vector of values for the **Time display offset** parameter, the scope uses the smallest of those values to set the minimum time-axis limit.
- *Maximum time-axis limit* — The Time Scope sets the maximum time-axis limit by summing the value of **Time display offset** parameter with the value of the **Time span** parameter. If you specify a vector of values for the **Time display offset** parameter, the scope sets the maximum time-axis limit by summing the largest of those values with the value of the **Time span** parameter.
- *Time units* — The units used to describe the time-axis. The Time Scope sets the time units using the value of the **Time Units** parameter on the **Time** tab of the Configuration Properties dialog box. By default, this parameter is set to `Metric (based on Time Span)` and displays in metric units such as milliseconds, microseconds, minutes, days, etc. You can change it to `Seconds` to always display the time-axis values in units of seconds. You can change it to `None` to not display any units on the time axis. When you set this parameter to `None`, then Time Scope shows only the word `Time` on the time axis.

  To hide both the word `Time` and the values on the time axis, set the **Show time-axis labels** parameter to `None`. To hide both the word `Time` and the values on the time axis in all displays, except the bottom ones in each column of displays, set this parameter to `Bottom Displays Only`. This behavior differs from the Simulink Scope block, which always shows the values but never shows a label on the x-axis.

For more information, see "Configure the Time Scope Properties" on page 24-3.

Simulation Indicators

- *Simulation status* — Provides the status of the model simulation. The status can be either of the following conditions:
  - `Processing` — Occurs after you run the `step` method and before you run the `release` method.
  - `Stopped` — Occurs after you construct the scope object and before you first run the `step` method. This status also occurs after you run the `release` method.

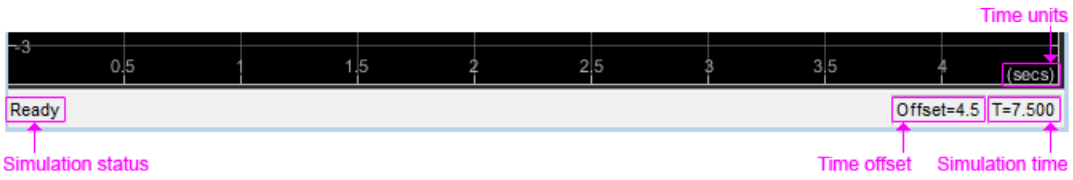
  The *Simulation status* is part of the status bar in the scope window. You can choose to hide or display the entire status bar. From the scope menu, select **View > Status Bar**.
- *Time offset* — The `Offset` value helps you determine the simulation times for which the scope is displaying data. The value is always in the range `0`≤ `Offset`≤ *Simulation time*. If the time offset is 0, the Scope does not display the `Offset` status field. Add the Time offset to the fixed time span values on the time-axis to get the overall simulation time.

  For example, if you set the **Time span** to `20` seconds, and you see an `Offset` of `0 (secs)` on the scope window. This value indicates that the scope is displaying data for the first `0` to `20` seconds of simulation time. If the **Offset** changes to `20 (secs)`, the scope displays data for simulation times from `20` seconds to `40` seconds. The scope continues to update the `Offset` value until the simulation is complete.
- *Simulation time* — The amount of time that the Time Scope has spent processing the input. Every time you call the scope, the simulation time increases by the number of rows in the input signal divided by the sample rate, as given by the following formula: $t_{sim} = t_{sim-1} + \frac{length(0:length(\text{xsine})) - 1}{SampleRate}$. You can set the sample rate using the `SampleRate` property. For frame-based inputs, the displayed Simulation time is the time at the beginning of the frame.

  The *Simulation time* is part of the status bar in the Time Scope window. You can choose to hide or display the entire status bar. From the Time Scope menu, select **View > Status Bar** .

Axes Maximization

When the scope is in maximized axes mode, the following figure highlights the important indicators on the scope window.



To toggle this mode, in the scope menu, select **View > Configuration Properties**. In the **Main** pane, locate the **Maximize axes** parameter.

Specify whether to display the scope in maximized axes mode. In this mode, each of the axes is expanded to fit into the entire display. To conserve space, labels do not appear in each display. Instead, tick-mark values appear on top of the plotted data. You can select one of the following options:

- `Auto` — In this mode, the axes appear maximized in all displays only if the `Title` and `YLabel` properties are empty for every display. If you enter any value in any display for either of these properties, the axes are not maximized.
- `On` — In this mode, the axes appear maximized in all displays. Any values entered into the `Title` and `YLabel` properties are hidden.
- `Off` — In this mode, none of the axes appear maximized.

The default setting is `Auto`.

Reduce Updates to Improve Performance

By default, the scope updates the displays periodically at a rate not exceeding 20 hertz. If you would like the scope to update on every simulation time step, you can disable the **Reduce Updates to Improve Performance** option. However, as a recommended practice, leave this option enabled because doing so can significantly improve the speed of the simulation.

In the Time Scope menu, select **Playback > Reduce Updates to Improve Performance** to clear the check box. Alternatively, use the **Ctrl+R** shortcut to toggle this setting. You can also set the `ReduceUpdates` property to `false` to disable this option.

Display Multiple Signals

Multi-Signal Input

You can configure the Time Scope to show multiple signals within the same display or on separate displays. By default, the signals appear as different-colored lines on the same display. The signals can have different dimensions, sample rates, and data types. Each signal can be either real or complex valued. You can set the number of input ports on the Time Scope in the following ways:

- Set the `NumInputPorts` property. This property is nontunable, so you should set it before you run the scope.
- Run the `show` method to open the scope window. In the scope menu, select **File > Number of Input Ports**.
- Run the `show` method to open the scope window. In the scope menu, select **View > Configuration Properties** and set the **Number of input ports** on the **Main** tab.

An input signal may contain multiple channels, depending on its dimensions. Multiple channels of data always appear as different-colored lines on the same display.

Multiple Signal Names and Colors

By default, if the input signal has multiple channels, the scope uses an index number to identify each channel of that signal. For example, a 2-channel signal would have the following default names in the channel legend: `Channel 1`, `Channel 2`. To show the legend, select **View > Configuration Properties**, click the **Display** tab, and select the **Show Legend** check box. If there are a total of seven input channels, the following legend appears in the display.

By default, the scope has a black axes background and chooses line colors for each channel in a manner similar to the Simulink Scope block. When the scope axes background is black, it assigns each channel of each input signal a line color in the order shown in the above figure.

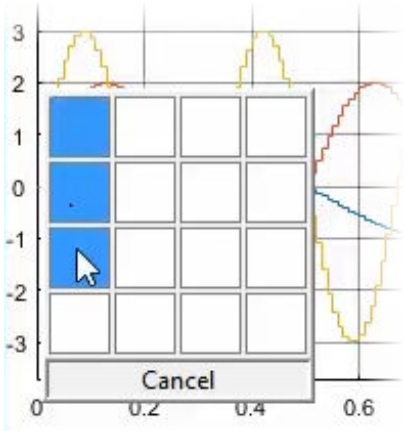
If there are more than seven channels, then the scope repeats this order to assign line colors to the remaining channels. To choose line colors for each channel, change the axes background color to any color except black. To change the axes background color to white, select **View > Style**, click the Axes background color button ( ), and select white from the color palette. Run the simulation again. The following legend appears in the display. This figure shows the color order when the background is not black.

Multiple Displays

You can display multiple channels of data on different displays in the scope window. In the scope toolbar, select **View > Layout**, or select the Layout button ( ).

Note The **Layout** menu item and button are not available when the scope is in snapshot mode.

You can tile the window into multiple displays. For example, if there are three inputs to the scope, you can display the signals in three separate displays. The layout grid shows a 4 by 4 grid, but you can select up to 16 by 16 by clicking and dragging within the layout grid.



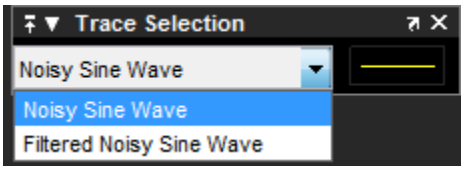
When you use the Layout option to tile the window into multiple displays, the display highlighted in blue is referred to as the active display. The scope dialog boxes reference the active display.

Time Scope Measurement Panels

The Measurements panels are the five panels that appear to the right side of the Scope GUI.

Trace Selection Panel

When you use the scope to view multiple signals, the Trace Selection panel appears. Use this panel to select which signal to measure. To open the Trace Selection panel:

- From the menu, select **Tools** > **Measurements** > **Trace Selection**.
- Open a measurement panel.
- 

Triggers Panel

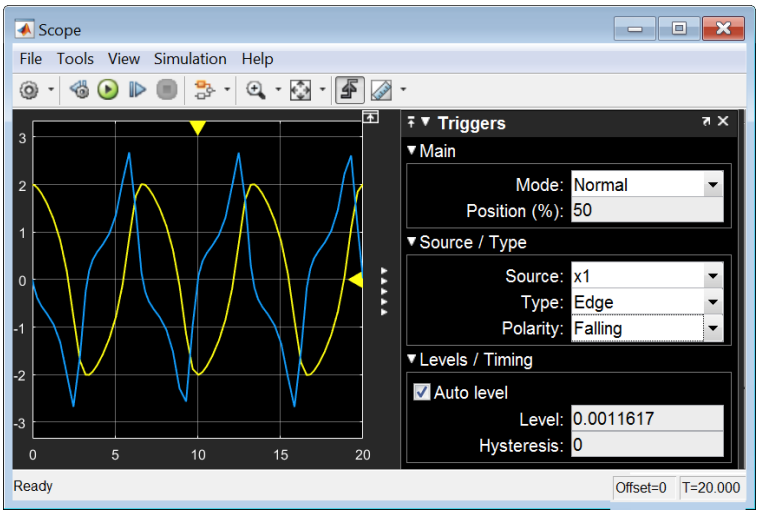
What Is the Trigger Panel

The Trigger panel defines a trigger event to synchronize simulation time with input signals. You can use trigger events to stabilize periodic signals such as a sine wave or capture non-periodic signals such as a pulse that occurs intermittently.

To open the Trigger panel:

1. Open a Scope block window.
2. On the toolbar, click the Triggers button .
3. Run a simulation.

   Triangle trigger pointers indicate the trigger time and trigger level of an event. The marker color corresponds to the color of the source signal.



Main Pane

Mode — Specify when the display updates.

- **Auto** — Display data from the last trigger event. If no event occurs after one time span, display the last available data.

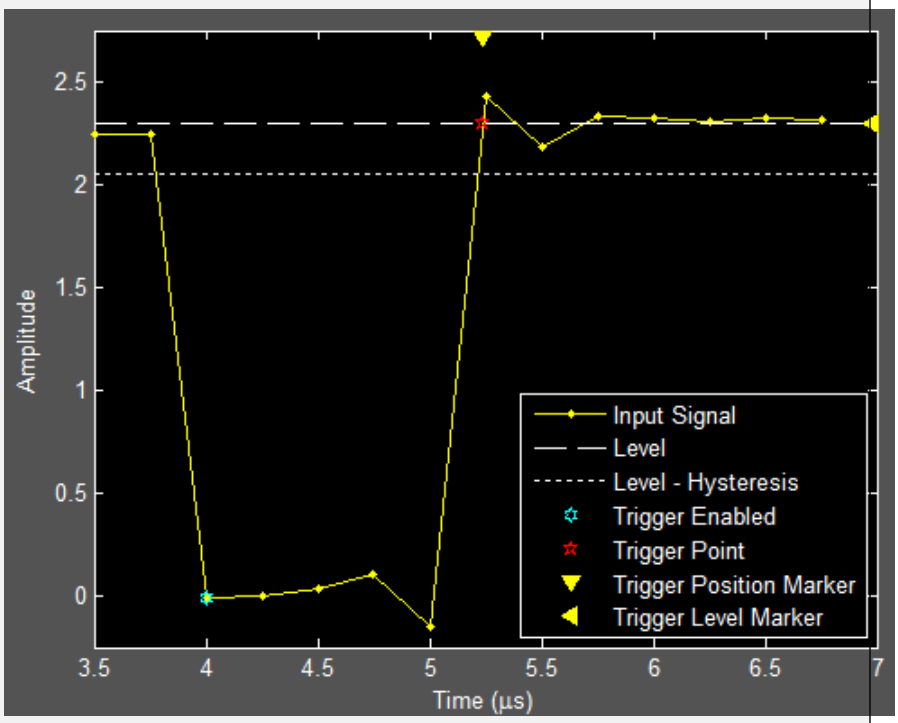
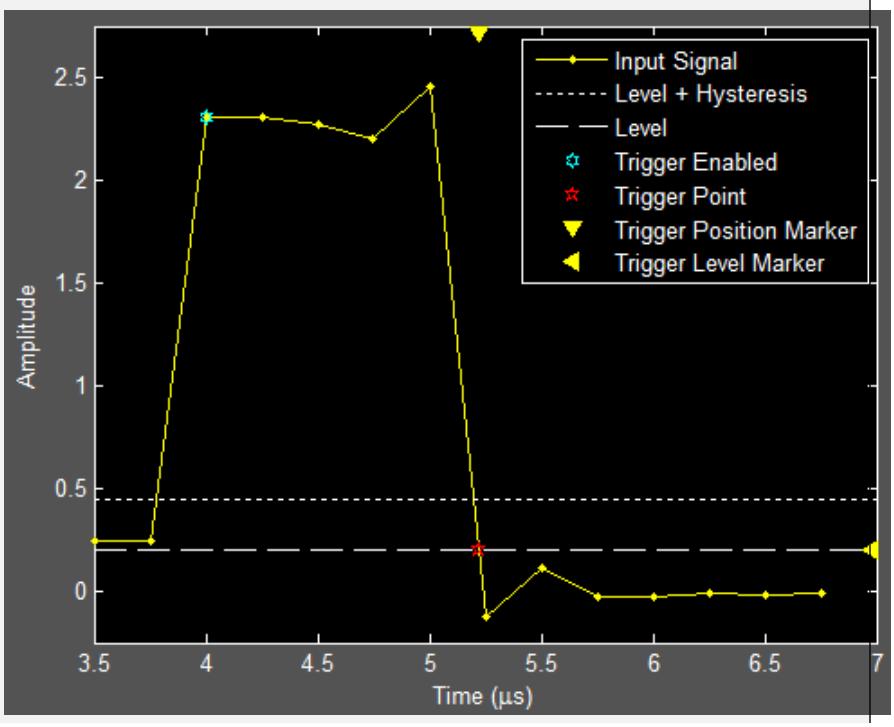
  Normal — Display data from the last trigger event. If no event occurs, the display remains blank.
- **Once** — Display data from the last trigger event and freeze the display. If no event occurs, the display remains blank. Click the **Rearm** button to look for the next trigger event.
- **Off** — Disable triggering.

Position (%) — Specify the position of the time pointer along the y-axis. You can also drag the time pointer to the left or right to adjust its position.

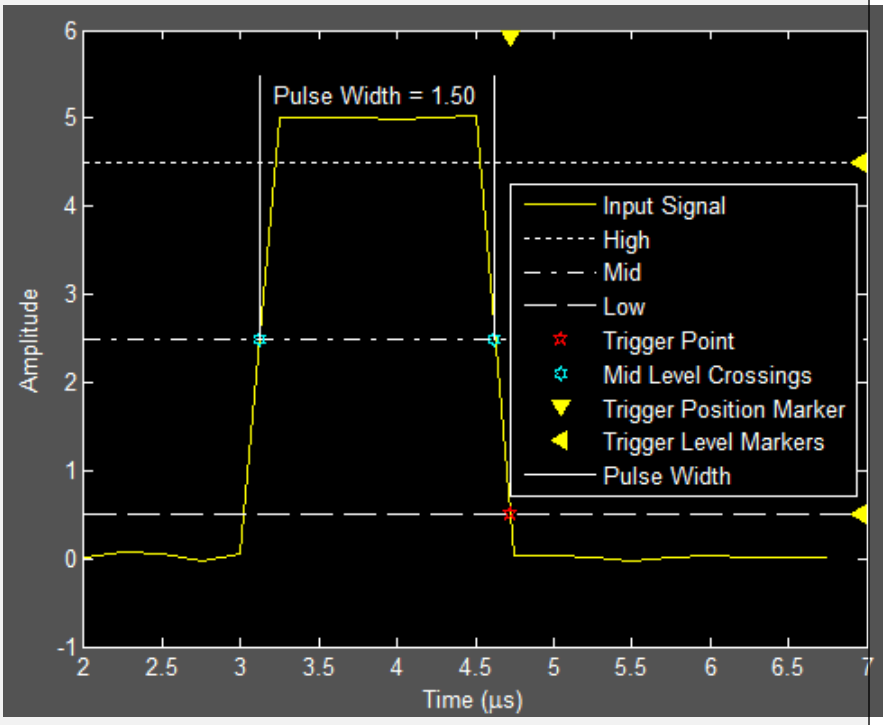
Source/Type and Levels/Timing Panes

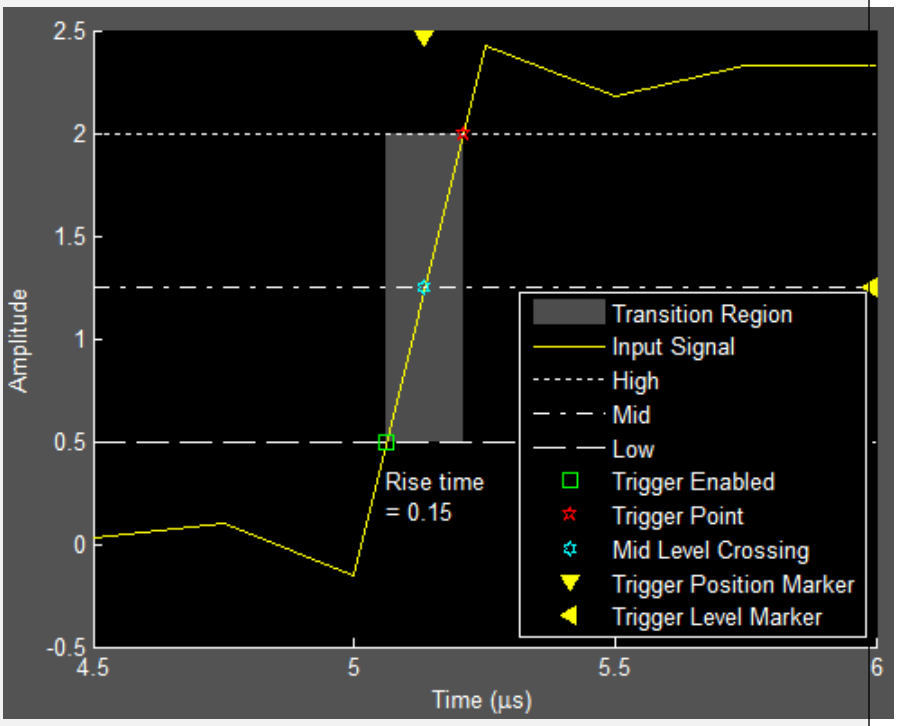
Source — Select a trigger signal. For magnitude and phase plots, select either the magnitude or the phase.

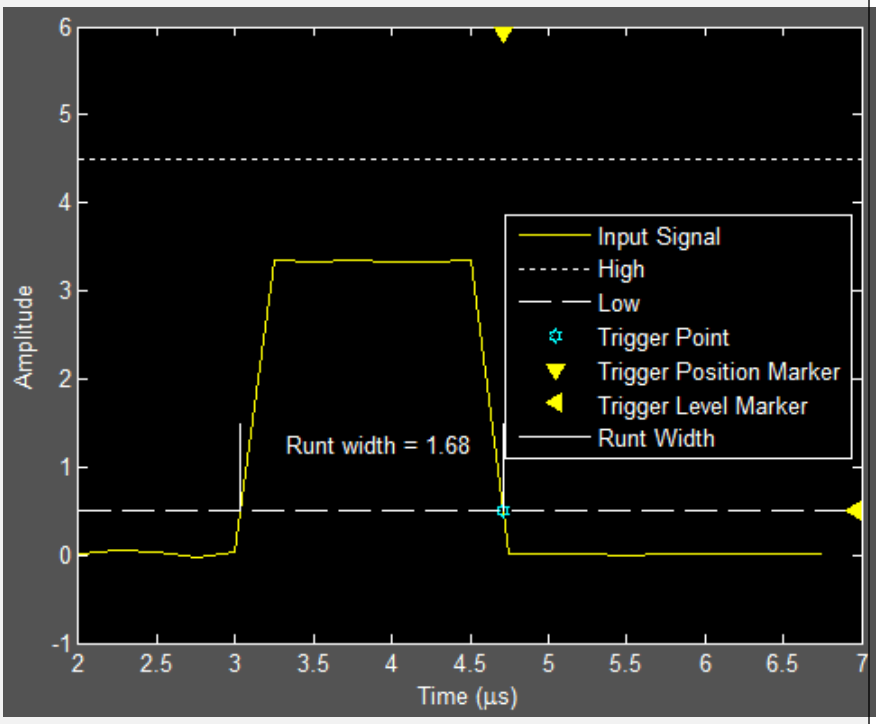
Type — Select the type of trigger.

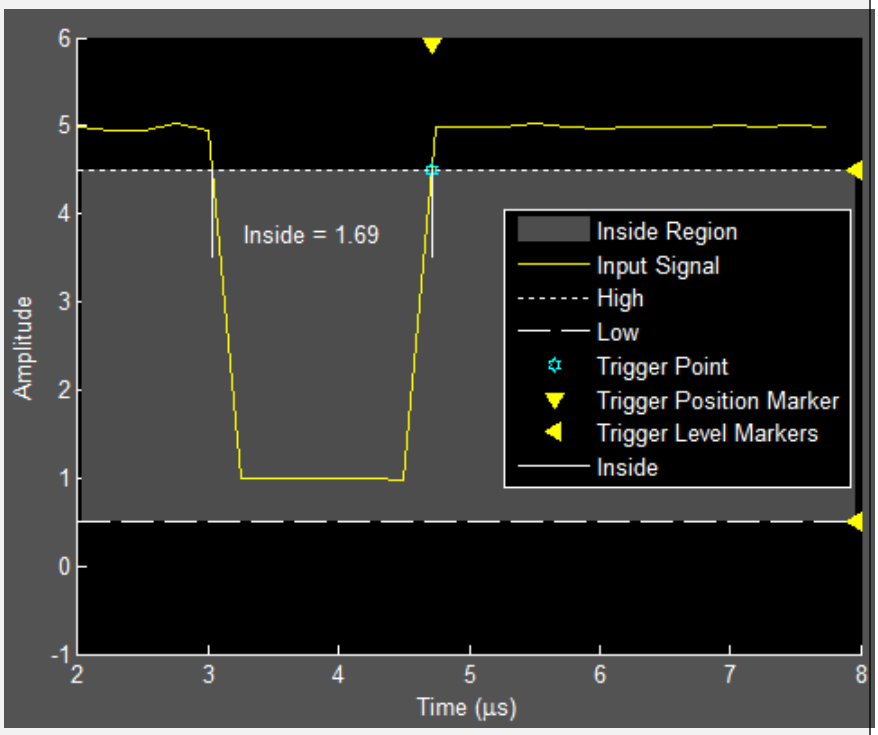
Trigger Type	Trigger Parameters
`Edge` — Trigger when the signal crosses a threshold.	**Polarity** — Select the polarity for an edge-triggered signal. • `Rising` — Trigger when the signal is increasing.  • `Falling` — Trigger when the signal value is decreasing. 

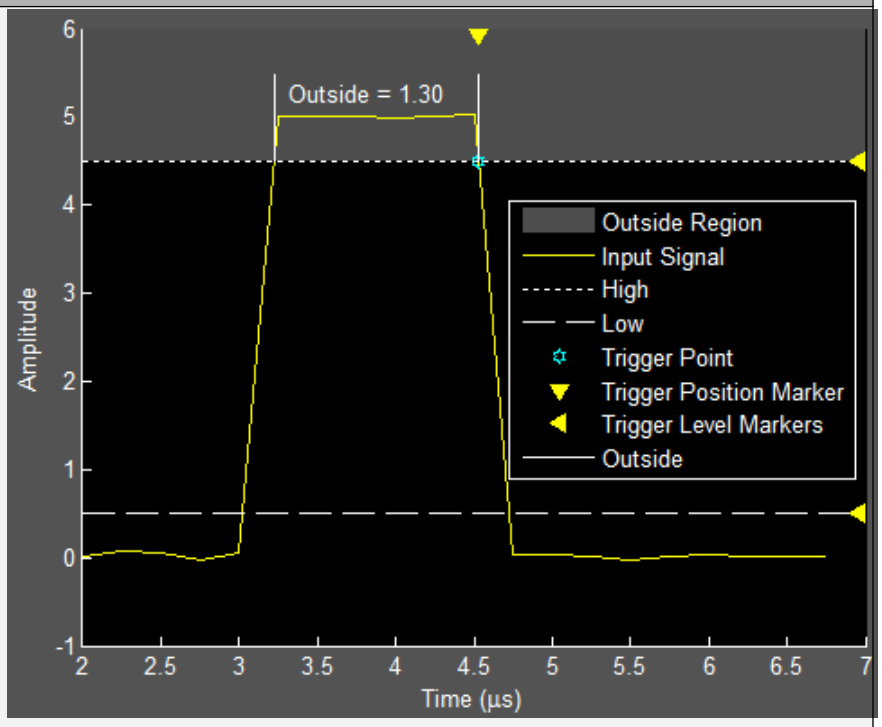
Trigger Type	Trigger Parameters
	• `Either` — Trigger when the signal is increasing or decreasing. **Level** — Enter a threshold value for an edge triggered signal. **Auto level** is 50% **Hysteresis** — Enter a value for an edge-triggered signal. See “Hysteresis of Trigger Signals” on page 24-67

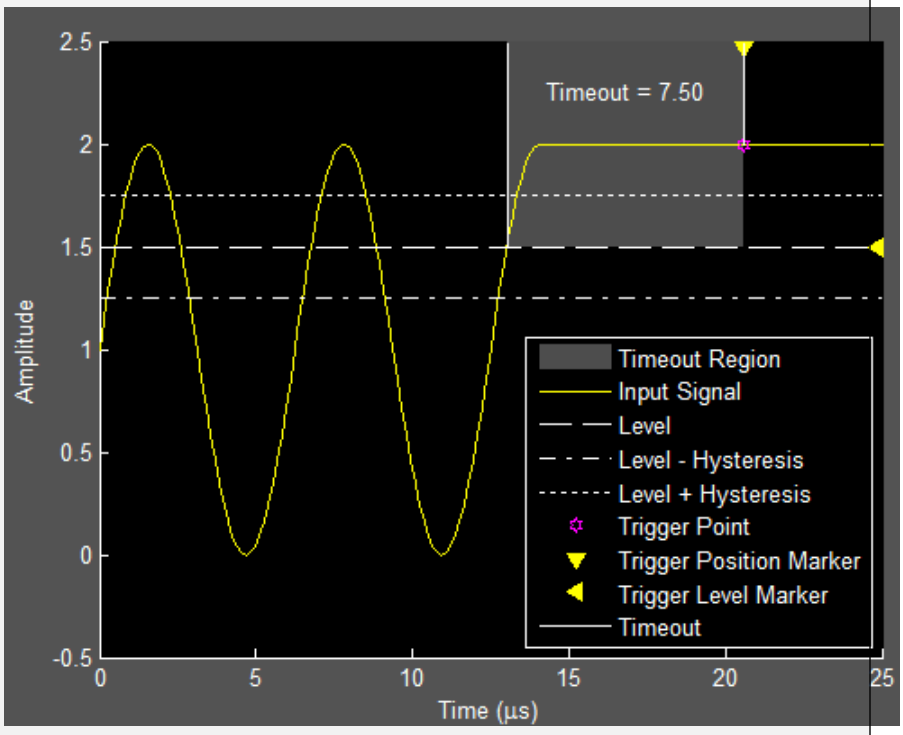
Trigger Type	Trigger Parameters
`Pulse Width` — Trigger when the signal crosses a low threshold and a high threshold twice within a specified time.	**Polarity** — Select the polarity for a pulse width-triggered signal. • `Positive` — Trigger on a positive-polarity pulse when the pulse crosses the low threshold for a second time.  • `Negative` — Trigger on a negative-polarity pulse when the pulse crosses the high threshold for a second time. • `Either` — Trigger on both positive-polarity and negative-polarity pulses. **Note** A glitch-trigger is a special type of a pulse width-trigger. A glitch-Trigger occurs for a pulse or spike whose duration is less than a specified amount. You can implement a glitch trigger by using a pulse width-trigger and setting the **Max Width** parameter to a small value. **High** — Enter a high value for a pulse width-triggered signal. **Auto level** is 90%. **Low** — Enter a low value for a pulse width-triggered signal. **Auto level** is 10%. **Min Width** — Enter the minimum pulse-width for a pulse width triggered signal. Pulse width is measured between the first and second crossings of the middle threshold.

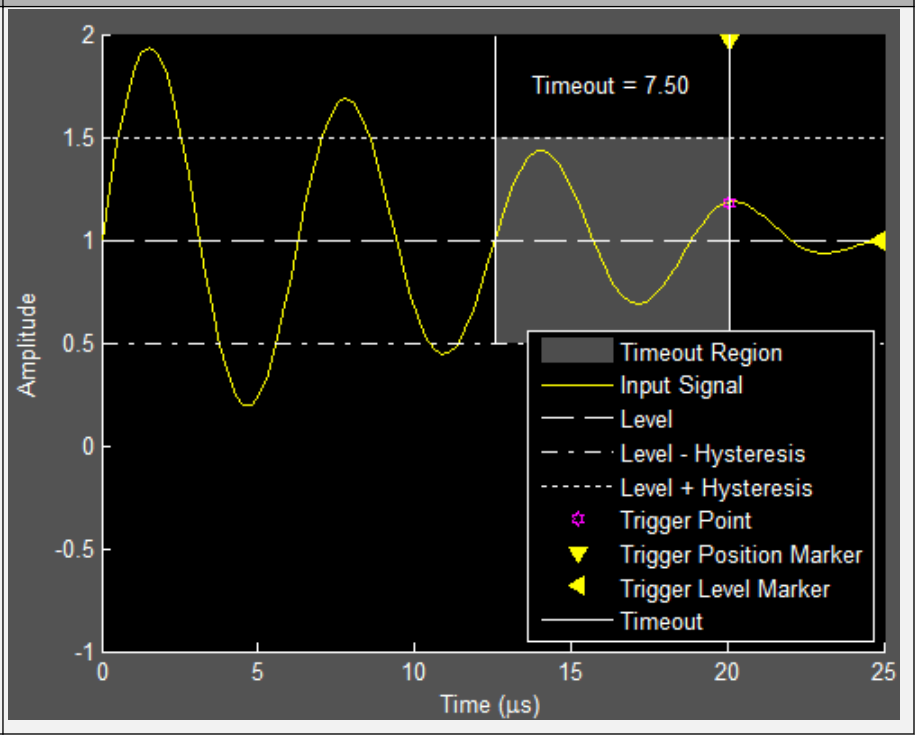
Trigger Type	Trigger Parameters
	Max Width — Enter the maximum pulse width for a pulse width triggered signal.
`Transition` — Trigger on the rising or falling edge of a signal that crosses the high and low levels within a specified time range.	**Polarity** — Select the polarity for a transition-triggered signal. • `Rise Time` — Trigger on an increasing signal when the signal crosses the high threshold.  • `Fall Time` — Trigger on a decreasing signal when the signal crosses the low threshold. • `Either` — Trigger on an increasing or decreasing signal. **High** — Enter a high value for a transition-triggered signal. **Auto level** is 90%. **Low** — Enter a low value for a transition-triggered signal. **Auto level** is 10%. **Min Time** — Enter a minimum time duration for a transition-triggered signal. **Max Time** — Enter a maximum time duration for a transition-triggered signal.

Trigger Type	Trigger Parameters
`Runt`— Trigger when a signal crosses a low threshold or a high threshold twice within a specified time.	**Polarity** — Select the polarity for a runt-triggered signal. • `Positive` — Trigger on a positive-polarity pulse when the signal crosses the low threshold a second time, without crossing the high threshold.  • `Negative` — Trigger on a negative-polarity pulse. • `Either` — Trigger on both positive-polarity and negative-polarity pulses. **High** — Enter a high value for a runt-triggered signal. **Auto level** is 90%. **Low** — Enter a low value for a runt-triggered signal. **Auto level** is 10%. **Min Width** — Enter a minimum width for a runt-triggered signal. Pulse width is measured between the first and second crossing of a threshold. **Max Width** — Enter a maximum pulse width for a runt-triggered signal.

Trigger Type	Trigger Parameters
`Window` — Trigger when a signal stays within or outside a region defined by the high and low thresholds for a specified time.	**Polarity** — Select the region for a window-triggered signal. • `Inside` — Trigger when a signal leaves a region between the low and high levels.  • `Outside` — Trigger when a signal enters a region between the low and high levels.

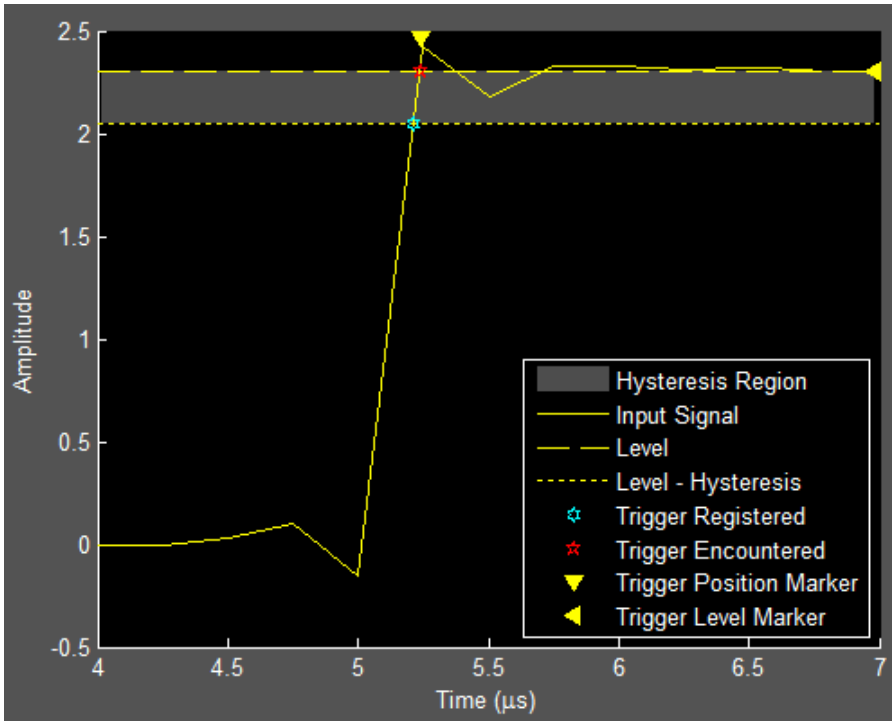
Trigger Type	Trigger Parameters
	 • `Either` — Trigger when a signal leaves or enters a region between the low and high levels. **High** — Enter a high value for a window-triggered signal. **Auto level** is 90%. **Low** — Enter a low value for a window-trigger signal. **Auto level** is 10%. **Min Time** — Enter the minimum time duration for a window-triggered signal. **Max Time** — Enter the maximum time duration for a window-triggered signal.

Trigger Type	Trigger Parameters
`Timeout` — Trigger when a signal stays above or below a threshold longer than a specified time	**Polarity** — Select the polarity for a timeout-triggered signal. • `Rising` — Trigger when the signal does not cross the threshold from below. For example, if you set **Timeout** to 7.50 seconds, the scope triggers 7.50 seconds after the signal crosses the threshold.  • `Falling` — Trigger when the signal does not cross the threshold from above. • `Either` — Trigger when the signal does not cross the threshold from either direction **Level** — Enter a threshold value for a timeout-triggered signal. **Hysteresis** — Enter a value for a timeout-triggered signal. See “Hysteresis of Trigger Signals” on page 24-67. **Timeout** — Enter a time duration for a timeout-triggered signal. Alternatively, a trigger event can occur when the signal stays within the boundaries defined by the hysteresis for 7.50 seconds after the signal crosses the threshold.

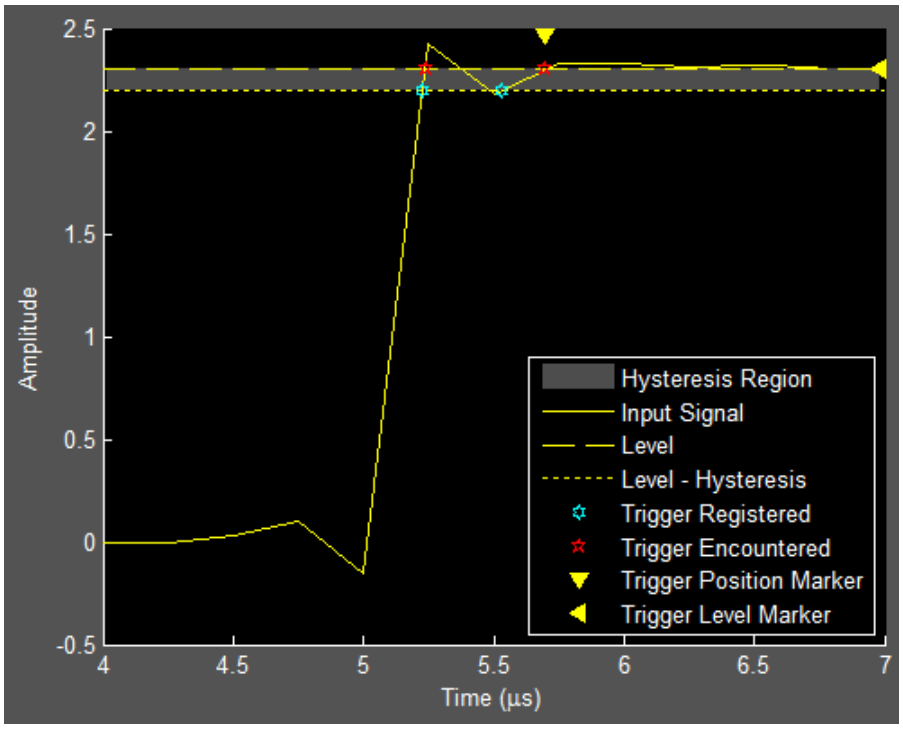
Trigger Type	Trigger Parameters
	

Hysteresis of Trigger Signals

Hysteresis (V) — Specify the hysteresis or noise reject value. This parameter is visible when you set **Type** to `Edge` or `Timeout`. If the signal jitters inside this range and briefly crosses the trigger level, the scope does not register an event. In the case of an edge trigger with rising polarity, the scope ignores the times that a signal crosses the trigger level within the hysteresis region.



You can reduce the hysteresis region size by decreasing the hysteresis value. In this example, if you set the hysteresis value to 0.07, the scope also considers the second rising edge to be a trigger event.



Delay/Holdoff Pane

Offset the trigger position by a fixed delay, or set the minimum possible time between trigger events.

- **Delay (s)** — Specify the fixed delay time by which to offset the trigger position. This parameter controls the amount of time the scope waits after a trigger event occurs before displaying a signal.
- **Holdoff (s)** — Specify the minimum possible time between trigger events. This amount of time is used to suppress data acquisition after a valid trigger event has occurred. A trigger holdoff prevents repeated occurrences of a trigger from occurring during the relevant portion of a burst.

Cursor Measurements Panel

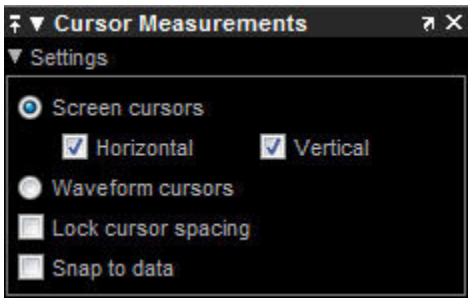
The **Cursor Measurements** panel displays screen cursors. The panel provides two types of cursors for measuring signals. Waveform cursors are vertical cursors that track along the signal. Screen cursors are both horizontal and vertical cursors that you can place anywhere in the display.

Note If a data point in your signal has more than one value, the cursor measurement at that point is undefined and no cursor value is displayed.

Display screen cursors with signal times and values. To open the Cursor measurements panel:

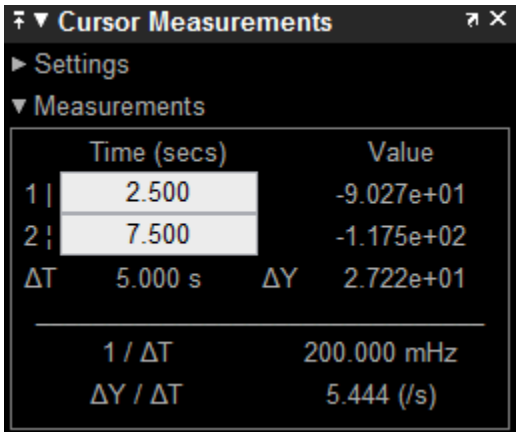
- From the menu, select **Tools > Measurements > Cursor Measurements**.
- On the toolbar, click the Cursor Measurements button.

In the **Settings** pane, you can modify the type of screen cursors used for calculating measurements. When more than one signal is displayed, you can assign cursors to each trace individually.



- **Screen Cursors** — Shows screen cursors (for spectrum and dual view only).
- **Horizontal** — Shows horizontal screen cursors (for spectrum and dual view only).
- **Vertical** — Shows vertical screen cursors (for spectrum and dual view only).
- **Waveform Cursors** — Shows cursors that attach to the input signals (for spectrum and dual view only).
- **Lock Cursor Spacing** — Locks the frequency difference between the two cursors.
- **Snap to Data** — Positions the cursors on signal data points.

The Measurements pane displays time and value measurements.

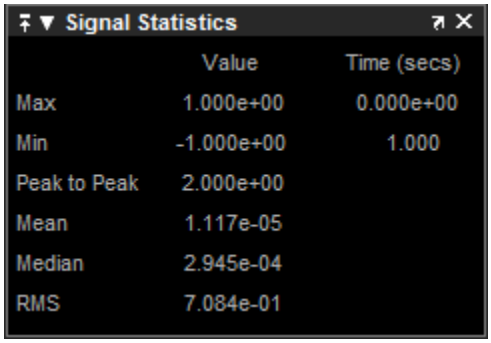


- **1** — View or modify the time or value at cursor number one (solid line cursor).
- **2** — View or modify the time or value at cursor number two (dashed line cursor).
- **ΔT** or **ΔX** — Shows the absolute value of the time (*x*-axis) difference between cursor number one and cursor number two.
- **ΔY** — Shows the absolute value of the signal amplitude difference between cursor number one and cursor number two.
- **1/ΔT** or **1/ΔX** — Shows the rate. The reciprocal of the absolute value of the difference in the times (*x*-axis) between cursor number one and cursor number two.
- **ΔY/ΔT** or **ΔY/ΔX** — Shows the slope. The ratio of the absolute value of the difference in signal amplitudes between cursors to the absolute value of the difference in the times (*x*-axis) between cursors.

Signal Statistics Panel

Display signal statistics for the signal selected in the **Trace Selection** panel. To open the Signal Statistics panel:

- From the menu, select **Tools > Measurements > Signal Statistics**.
- On the toolbar, click the Signal Statistics button.



The statistics shown are:

- **Max** — Maximum or largest value within the displayed portion of the input signal.
- **Min** — Minimum or smallest value within the displayed portion of the input signal.

- **Peak to Peak** — Difference between the maximum and minimum values within the displayed portion of the input signal.
- **Mean** — Average or mean of all the values within the displayed portion of the input signal.
- **Median** — Median value within the displayed portion of the input signal.
- **RMS** — Root mean squared of the input signal.

When you use the zoom options in the scope, the Signal Statistics measurements automatically adjust to the time range shown in the display. In the scope toolbar, click the **Zoom In** or **Zoom X** button to constrict the *x*-axis range of the display, and the statistics shown reflect this time range. For example, you can zoom in on one pulse to make the **Signal Statistics** panel display information about only that particular pulse.

The Signal Statistics measurements are valid for any units of the input signal. The letter after the value associated with each measurement represents the appropriate International System of Units (SI) prefix, such as *m* for *milli-*. For example, if the input signal is measured in volts, an *m* next to a measurement value indicates that this value is in units of millivolts.

Bilevel Measurements Panel

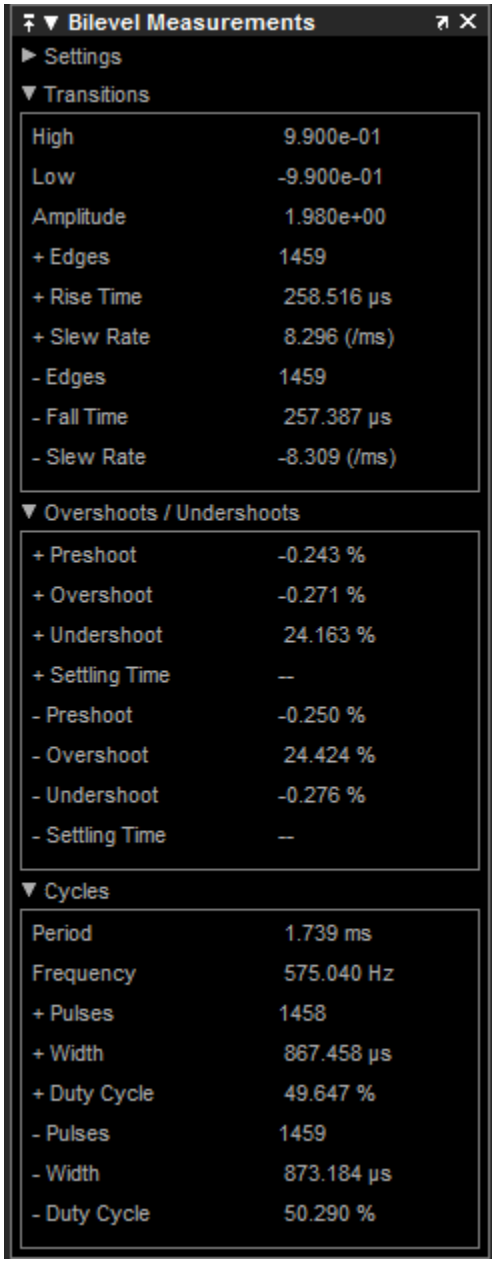
Bilevel Measurements

Display information about signal transitions, overshoots, undershoots, and cycles. To open the Bilevel Measurements panel:

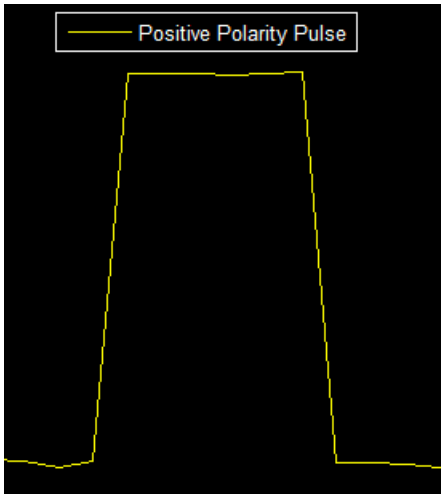
- From the menu, select **Tools > Measurements > Bilevel Measurements**.
- On the toolbar, click the Bilevel Measurements button.

Settings

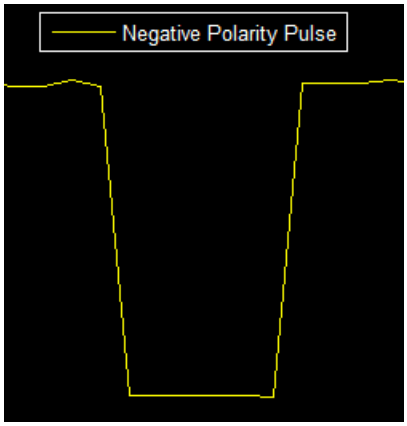
The **Settings** pane enables you to modify the properties used to calculate various measurements involving transitions, overshoots, undershoots, and cycles. You can modify the high-state level, low-state level, state-level tolerance, upper-reference level, mid-reference level, and lower-reference level.



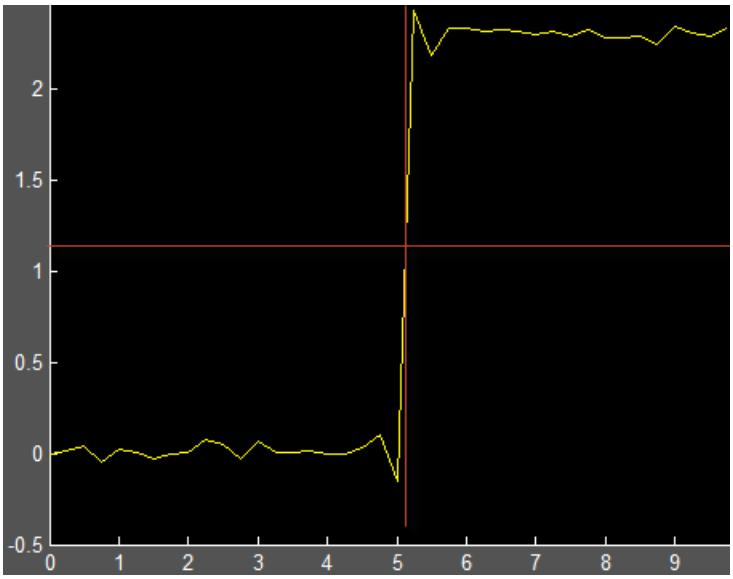
- **Auto State Level** — When this check box is selected, the Bilevel measurements panel detects the high- and low- state levels of a bilevel waveform. When this check box is cleared, you can enter in values for the high- and low- state levels manually.
    - **High** — Used to specify manually the value that denotes a positive polarity, or high-state level.



- **Low** — Used to specify manually the value that denotes a negative polarity, or low-state level.



- **State Level Tolerance** — Tolerance within which the initial and final levels of each transition must be within their respective state levels. This value is expressed as a percentage of the difference between the high- and low-state levels.
- **Upper Ref Level** — Used to compute the end of the rise-time measurement or the start of the fall time measurement. This value is expressed as a percentage of the difference between the high- and low-state levels.
- **Mid Ref Level** — Used to determine when a transition occurs. This value is expressed as a percentage of the difference between the high- and low- state levels. In the following figure, the mid-reference level is shown as the horizontal line, and its corresponding mid-reference level instant is shown as the vertical line.

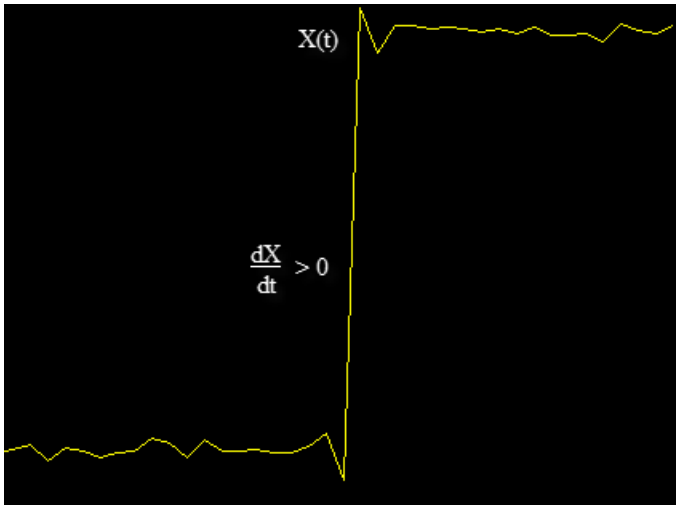


- **Lower Ref Level** — Used to compute the end of the fall-time measurement or the start of the rise-time measurement. This value is expressed as a percentage of the difference between the high- and low-state levels.
- **Settle Seek** — The duration after the mid-reference level instant when each transition occurs used for computing a valid settling time. This value is equivalent to the input parameter, `D`, which you can set when you run the `settlingtime` function. The settling time is displayed in the **Overshoots/Undershoots** pane.

Transitions Pane

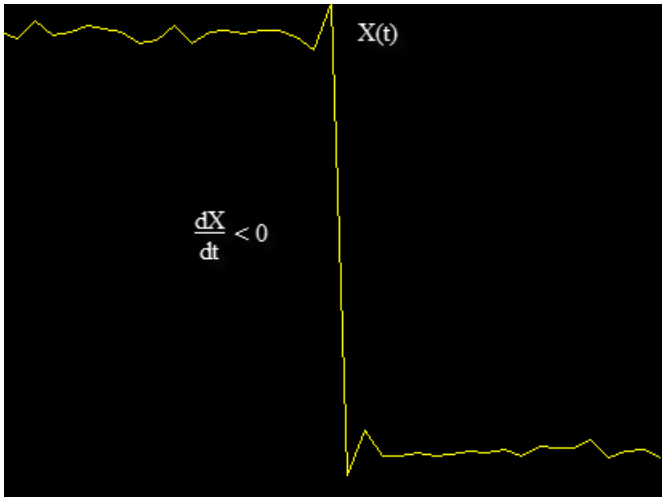
Display calculated measurements associated with the input signal changing between its two possible state level values, high and low.

A positive-going transition, or rising edge, in a bilevel waveform is a transition from the low-state level to the high-state level. A positive-going transition has a slope value greater than zero. The following figure shows a positive-going transition.



When there is a plus sign (+) next to a text label, the measurement is a rising edge, a transition from a low-state level to a high-state level.

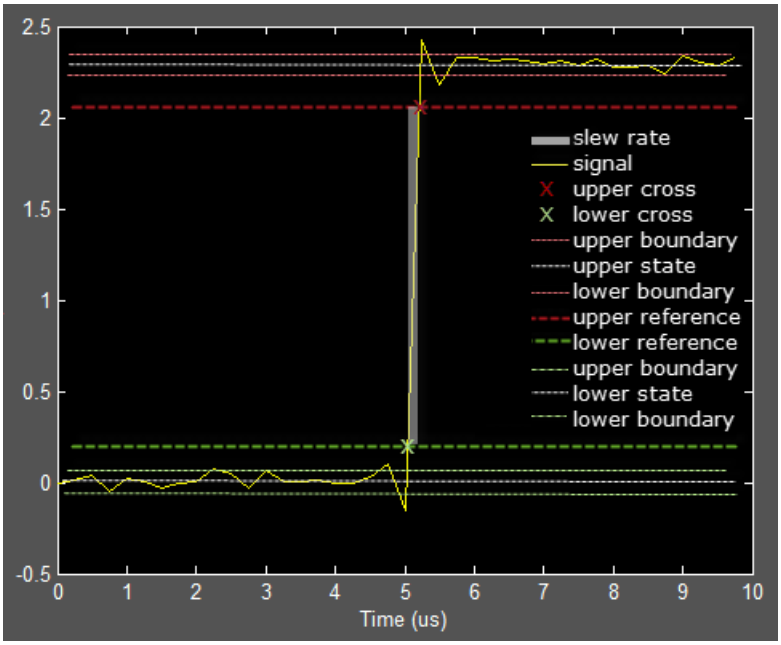
A negative-going transition, or falling edge, in a bilevel waveform is a transition from the high-state level to the low-state level. A negative-going transition has a slope value less than zero. The following figure shows a negative-going transition.



When there is a minus sign (–) next to a text label, the measurement is a falling edge, a transition from a high-state level to a low-state level.

The Transition measurements assume that the amplitude of the input signal is in units of volts. For the transition measurements to be valid, you must convert all input signals to volts.

- **High** — The high-amplitude state level of the input signal over the duration of the **Time Span** parameter. You can set **Time Span** in the **Main** pane of the Visuals—Time Domain Properties dialog box.
- **Low** — The low-amplitude state level of the input signal over the duration of the **Time Span** parameter. You can set **Time Span** in the **Main** pane of the Visuals—Time Domain Properties dialog box.
- **Amplitude** — Difference in amplitude between the high-state level and the low-state level.
- **+ Edges** — Total number of positive-polarity, or rising, edges counted within the displayed portion of the input signal.
- **+ Rise Time** — Average amount of time required for each rising edge to cross from the lower-reference level to the upper-reference level.
- **+ Slew Rate** — Average slope of each rising-edge transition line within the upper- and lower-percent reference levels in the displayed portion of the input signal. The region in which the slew rate is calculated appears in gray in the following figure.

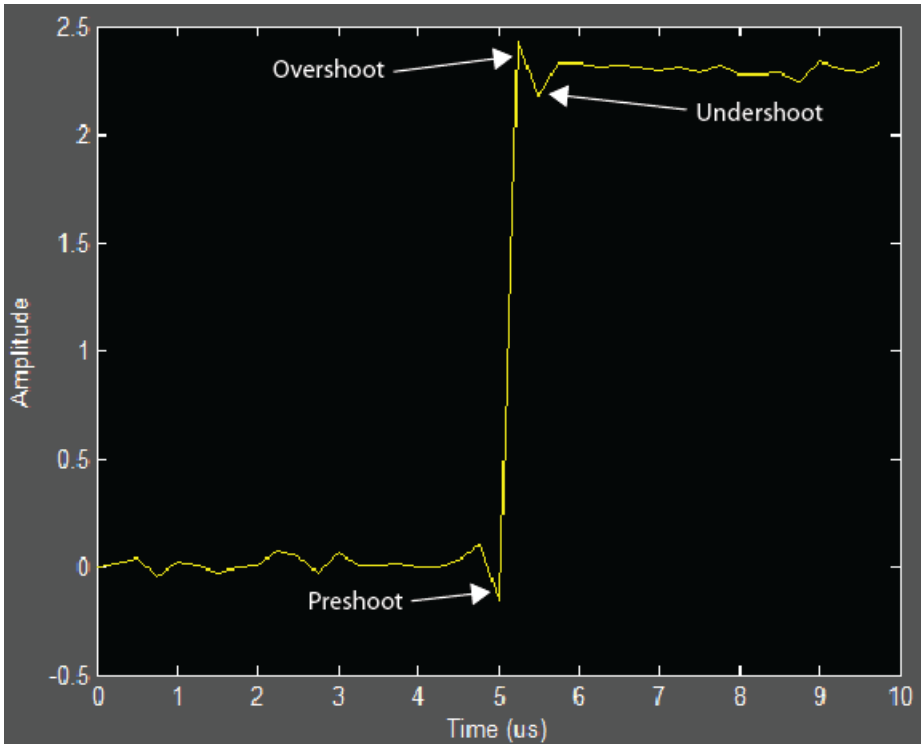


- **– Edges** — Total number of negative-polarity or falling edges counted within the displayed portion of the input signal.
- **– Fall Time** — Average amount of time required for each falling edge to cross from the upper-reference level to the lower-reference level.
- **– Slew Rate** — Average slope of each falling edge transition line within the upper- and lower-percent reference levels in the displayed portion of the input signal.

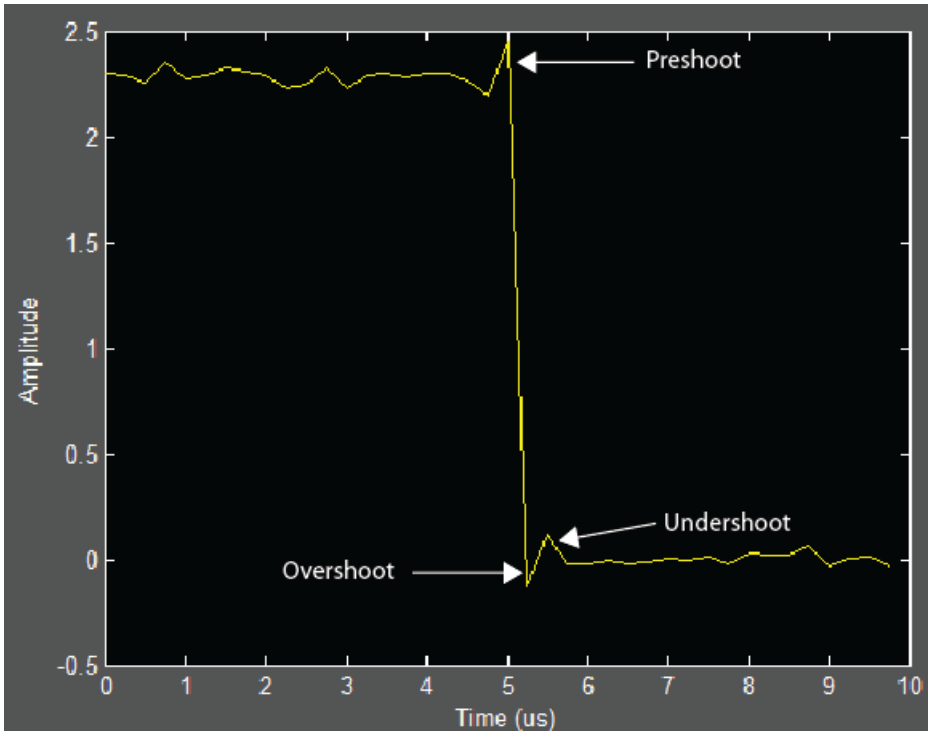
Overshoots / Undershoots Pane

The **Overshoots/Undershoots** pane displays calculated measurements involving the distortion and damping of the input signal. Overshoot and undershoot refer to the amount that a signal respectively exceeds and falls below its final steady-state value. Preshoot refers to the amount before a transition that a signal varies from its initial steady-state value.

This figure shows preshoot, overshoot, and undershoot for a rising-edge transition.

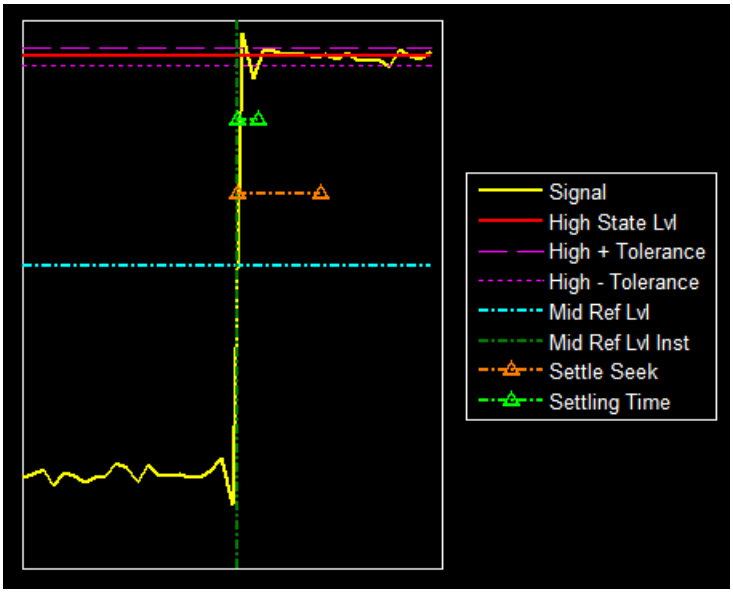


The next figure shows preshoot, overshoot, and undershoot for a falling-edge transition.



- **+ Preshoot** — Average lowest aberration in the region immediately preceding each rising transition.

- **+ Overshoot** — Average highest aberration in the region immediately following each rising transition.
- **+ Undershoot** — Average lowest aberration in the region immediately following each rising transition.
- **+ Settling Time** — Average time required for each rising edge to enter and remain within the tolerance of the high-state level for the remainder of the settle-seek duration. The settling time is the time after the mid-reference level instant when the signal crosses into and remains in the tolerance region around the high-state level. This crossing is illustrated in the following figure.



You can modify the settle-seek duration parameter in the **Settings** pane.

- **– Preshoot** — Average highest aberration in the region immediately preceding each falling transition.
- **– Overshoot** — Average highest aberration in the region immediately following each falling transition.
- **– Undershoot** — Average lowest aberration in the region immediately following each falling transition.
- **– Settling Time** — Average time required for each falling edge to enter and remain within the tolerance of the low-state level for the remainder of the settle-seek duration. The settling time is the time after the mid-reference level instant when the signal crosses into and remains in the tolerance region around the low-state level. You can modify the settle-seek duration parameter in the **Settings** pane.

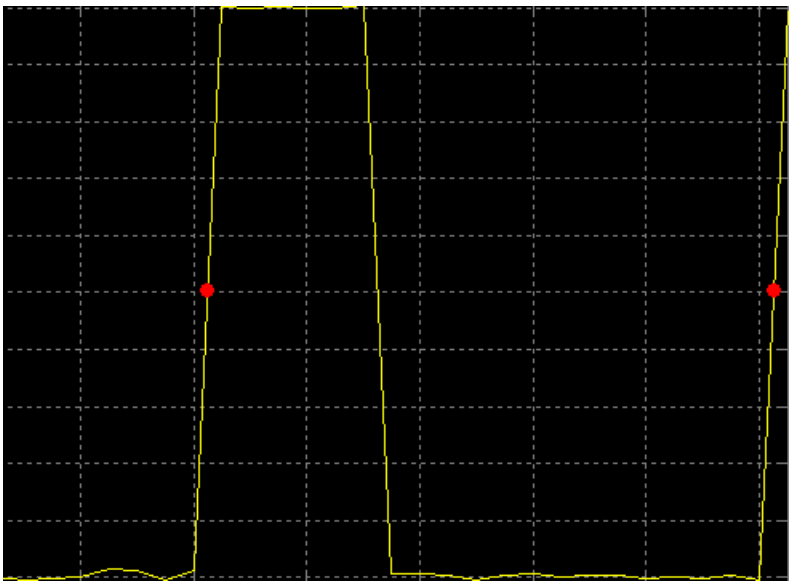
Cycles Pane

The **Cycles** pane displays calculated measurements pertaining to repetitions or trends in the displayed portion of the input signal.

Properties to set:

- **Period** — Average duration between adjacent edges of identical polarity within the displayed portion of the input signal. The Bilevel measurements panel calculates period as follows. It takes the difference between the mid-reference level instants of the initial transition of each positive-

polarity pulse and the next positive-going transition. These mid-reference level instants appear as red dots in the following figure.



- **Frequency** — Reciprocal of the average period. Whereas period is typically measured in some metric form of seconds, or seconds per cycle, frequency is typically measured in hertz or cycles per second.
- **+ Pulses** — Number of positive-polarity pulses counted.
- **+ Width** — Average duration between rising and falling edges of each positive-polarity pulse within the displayed portion of the input signal.
- **+ Duty Cycle** — Average ratio of pulse width to pulse period for each positive-polarity pulse within the displayed portion of the input signal.
- **– Pulses** — Number of negative-polarity pulses counted.
- **– Width** — Average duration between rising and falling edges of each negative-polarity pulse within the displayed portion of the input signal.
- **– Duty Cycle** — Average ratio of pulse width to pulse period for each negative-polarity pulse within the displayed portion of the input signal.

When you use the zoom options in the Scope, the bilevel measurements automatically adjust to the time range shown in the display. In the Scope toolbar, click the **Zoom In** or **Zoom X** button to constrict the *x*-axis range of the display, and the statistics shown reflect this time range. For example, you can zoom in on one rising edge to make the **Bilevel Measurements** panel display information about only that particular rising edge. However, this feature does not apply to the **High** and **Low** measurements.

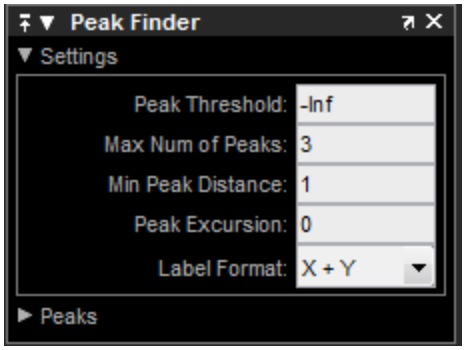
Peak Finder Panel

The **Peak Finder** panel displays the maxima, showing the *x*-axis values at which they occur. Peaks are defined as a local maximum where lower values are present on both sides of a peak. Endpoints are not considered peaks. This panel allows you to modify the settings for peak threshold, maximum number of peaks, and peak excursion.

- From the menu, select **Tools > Measurements > Peak Finder**.

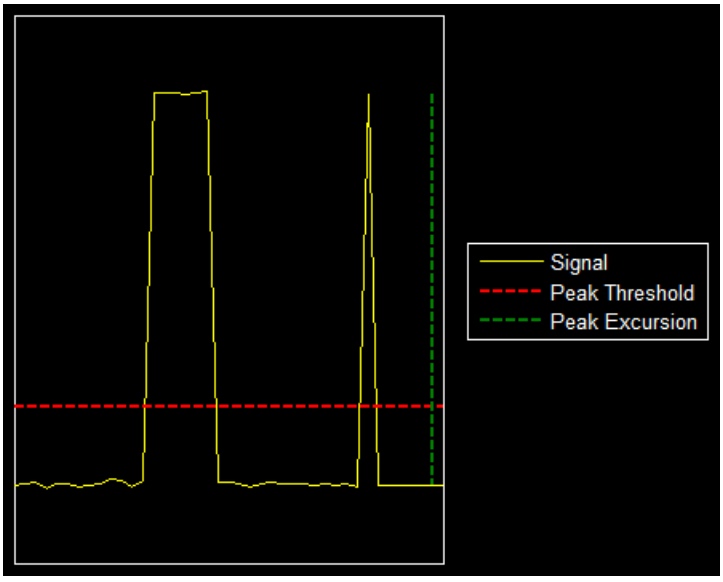
- On the toolbar, click the Peak Finder button.

The **Settings** pane enables you to modify the parameters used to calculate the peak values within the displayed portion of the input signal. For more information on the algorithms this pane uses, see the `findpeaks` function reference.



Properties to set:

- **Peak Threshold** — The level above which peaks are detected. This setting is equivalent to the `MINPEAKHEIGHT` parameter, which you can set when you run the `findpeaks` function.
- **Max Num of Peaks** — The maximum number of peaks to show. The value you enter must be a scalar integer from 1 through 99. This setting is equivalent to the `NPEAKS` parameter, which you can set when you run the `findpeaks` function.
- **Min Peaks Distance** — The minimum number of samples between adjacent peaks. This setting is equivalent to the `MINPEAKDISTANCE` parameter, which you can set when you run the `findpeaks` function.
- **Peak Excursion** — The minimum height difference between a peak and its neighboring samples. Peak excursion is illustrated alongside peak threshold in the following figure.



The peak threshold is a minimum value necessary for a sample value to be a peak. The peak excursion is the minimum difference between a peak sample and the samples to its left and right

in the time domain. In the figure, the green vertical line illustrates the lesser of the two height differences between the labeled peak and its neighboring samples. This height difference must be greater than the **Peak Excursion** value for the labeled peak to be classified as a peak. Compare this setting to peak threshold, which is illustrated by the red horizontal line. The amplitude must be above this horizontal line for the labeled peak to be classified as a peak.

The peak excursion setting is equivalent to the `THRESHOLD` parameter, which you can set when you run the `findpeaks` function.

- **Label Format** — The coordinates to display next to the calculated peak values on the plot. To see peak values, you must first expand the **Peaks** pane and select the check boxes associated with individual peaks of interest. By default, both *x*-axis and *y*-axis values are displayed on the plot. Select which axes values you want to display next to each peak symbol on the display.
  - `X+Y` — Display both *x*-axis and *y*-axis values.
  - `X` — Display only *x*-axis values.
  - `Y` — Display only *y*-axis values.

The **Peaks** pane displays the largest calculated peak values. It also shows the coordinates at which the peaks occur, using the parameters you define in the **Settings** pane. You set the **Max Num of Peaks** parameter to specify the number of peaks shown in the list.

The numerical values displayed in the **Value** column are equivalent to the `pks` output argument returned when you run the `findpeaks` function. The numerical values displayed in the second column are similar to the `locs` output argument returned when you run the `findpeaks` function.

The Peak Finder displays the peak values in the **Peaks** pane. By default, the **Peak Finder** panel displays the largest calculated peak values in the **Peaks** pane in decreasing order of peak height.

Use the check boxes to control which peak values are shown on the display. By default, all check boxes are cleared and the **Peak Finder** panel hides all the peak values. To show or hide all the peak values on the display, use the check box in the top-left corner of the **Peaks** pane.

The Peaks are valid for any units of the input signal. The letter after the value associated with each measurement indicates the abbreviation for the appropriate International System of Units (SI) prefix, such as *m* for *milli-*. For example, if the input signal is measured in volts, an *m* next to a measurement value indicates that this value is in units of millivolts.

Style Dialog Box

Select **View > Style** or the Style button ( ) in the dropdown below the Configuration Properties button to open the Style dialog box. In this dialog box, you can change the figure colors, background axes colors, foreground axes colors, and properties of lines in a display.

For more details about the properties, see “Style Properties”.

Axes Scaling Properties

The Axes Scaling Properties dialog box provides you with the ability to automatically zoom in on and zoom out of your data, and to scale the axes of the Time Scope. In the Time Scope menu, select **Tools > Axes Scaling > Axes Scaling Properties** to open this dialog box.

For more details about the properties, see “Axes Scaling Properties”.

Sources — Streaming Properties

The Sources – Streaming Properties dialog box lets you control the number of input signal samples that Time Scope holds in memory. In the Time Scope menu, select **View > Data History Properties** to open this dialog box.

Buffer length

Specify the size of the buffer that the scope holds in its memory cache. Memory is limited by available memory on your system. If your signal has *M* rows of data and *N* data points in each row, *M* x *N* is the number of data points per time step. Multiply this result by the number of time steps for your model to obtain the required buffer length. For example, if you have 10 rows of data with each row having 100 data points and your run will be 10 time steps, you should enter 10,000 (which is 10 x 100 x 10) as the buffer length.

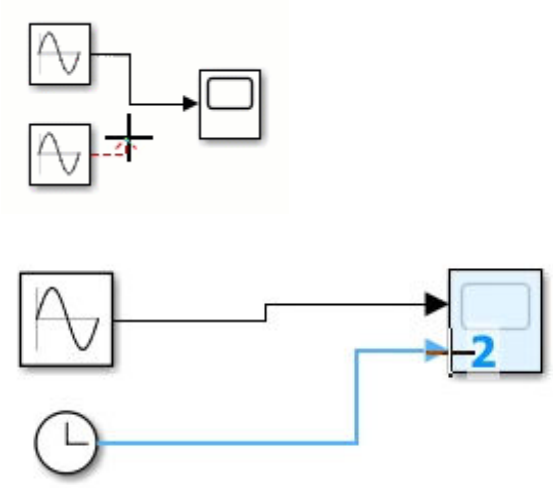
Common Scope Block Tasks

In this section...
“Connect Multiple Signals to a Scope” on page 24-83
“Save Simulation Data Using Scope Block” on page 24-85
“Pause Display While Running” on page 24-87
“Copy Scope Image” on page 24-87
“Plot an Array of Signals” on page 24-89
“Scopes in Referenced Models” on page 24-89
“Scopes Within an Enabled Subsystem” on page 24-92
“Show Signal Units on a Scope Display” on page 24-92
“Select Number of Displays and Layout” on page 24-95
“Dock and Undock Scope Window to MATLAB Desktop” on page 24-95

To visualize your simulation results over time, use a Scope block or Time Scope block

Connect Multiple Signals to a Scope

To connect multiple signals to a scope, drag additional signals to the scope block. An additional port is created automatically.



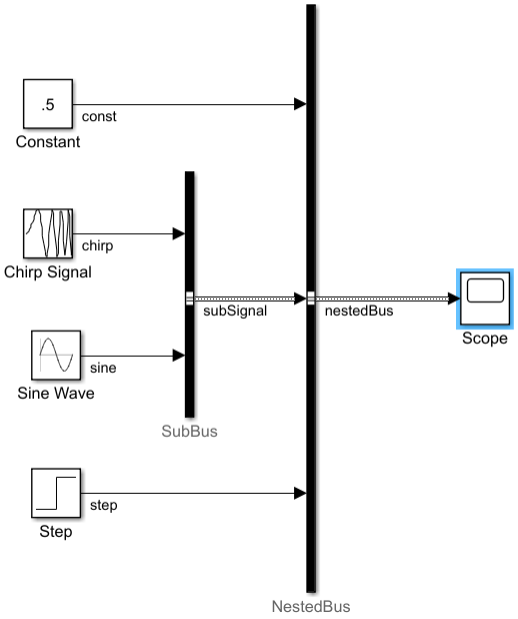
To specify the number of input ports:

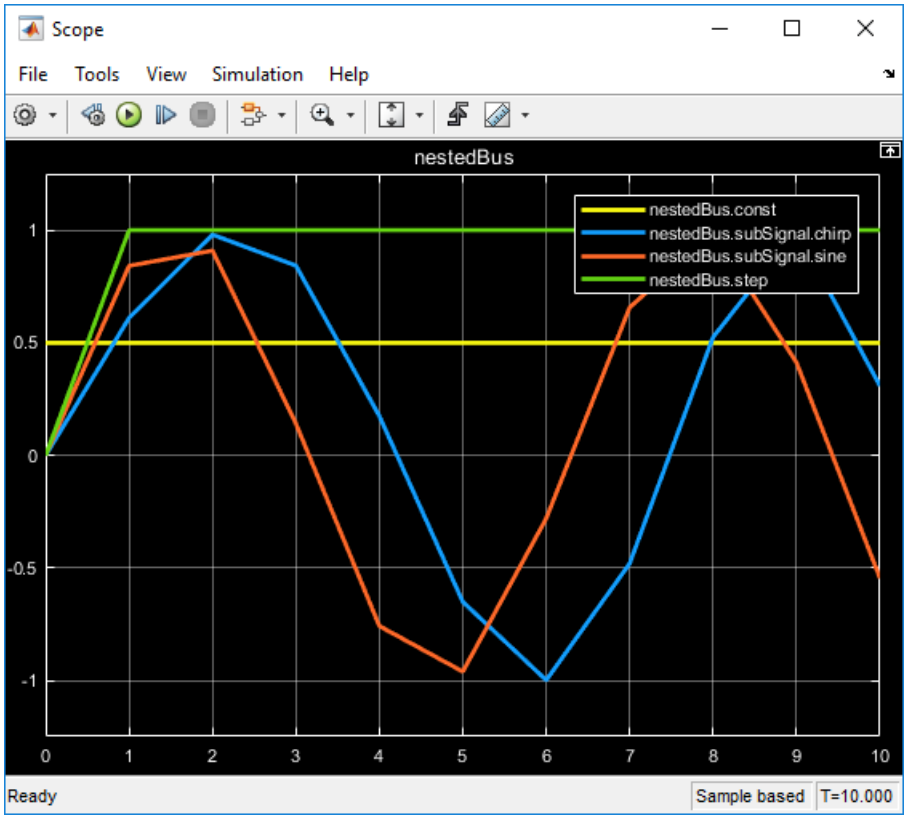
1. Open a scope window.
2. From the toolbar, select **File** > **Number of Input Ports** > **More**.
3. Enter the number of input ports, up to 96.

Signals from Nonvirtual Buses and Arrays of Buses

You can connect signals from nonvirtual buses and arrays of buses to a Scope block. To display the bus signals, use normal or accelerator simulation mode. The Scope block displays each bus element

signal, in the order the elements appear in the bus, from the top to the bottom. Nested bus elements are flattened. For example, in this model the `nestedBus` signal has the `const`, `subSignal`, and `step` signals as elements. The `subSignal` sub-bus has the `chirp` and `sine` signals as its bus elements. In the Scope block, the two elements of the `subSignal` bus display between the `const` and `step` signals.



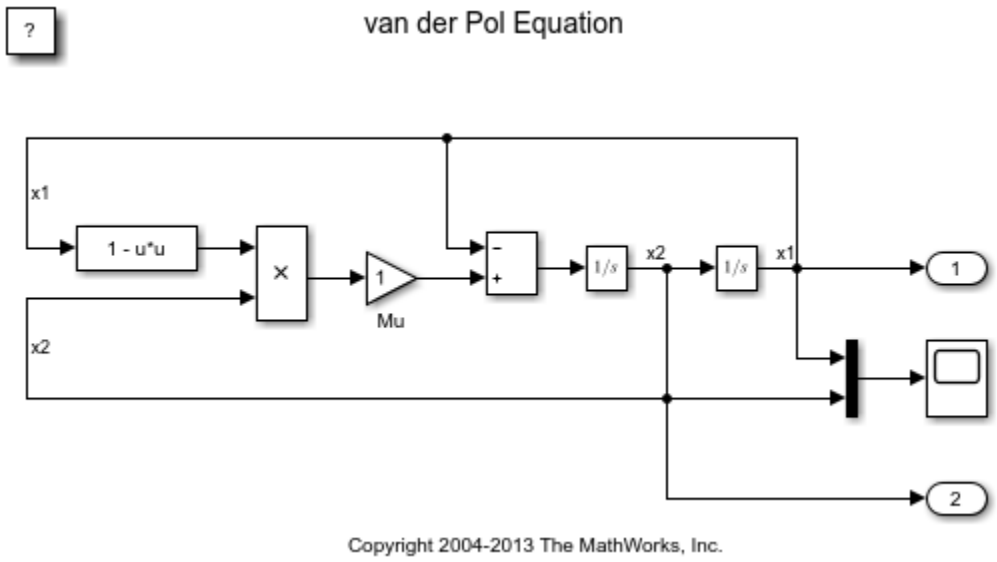


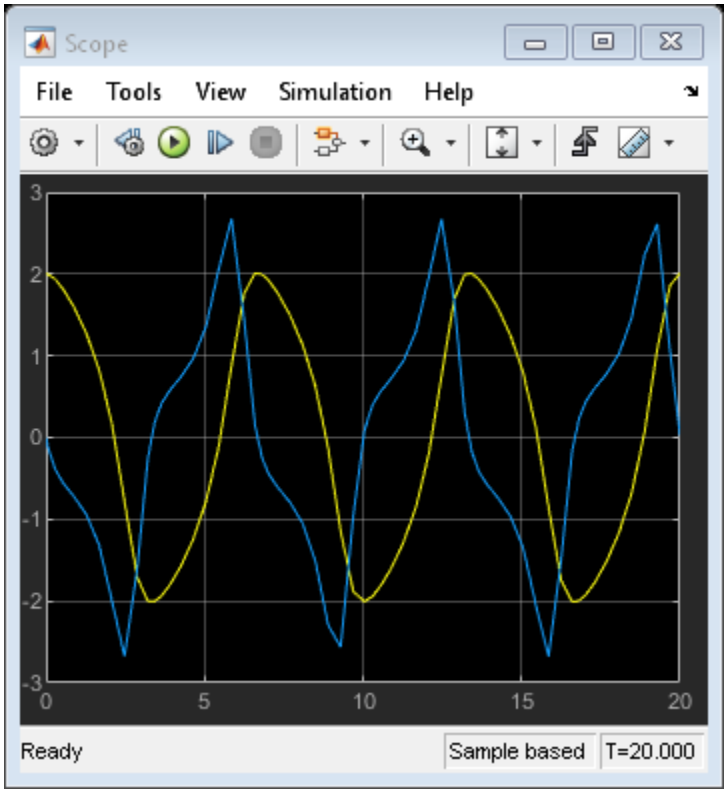
Save Simulation Data Using Scope Block

This example shows how to save signals to the MATLAB Workspace using the Scope block.

Using the `vdp` model, turn on data logging to the workspace. You can follow the commands below, or in the Scope Configuration Parameters - Logging tab, turn on **Log data to workspace**.

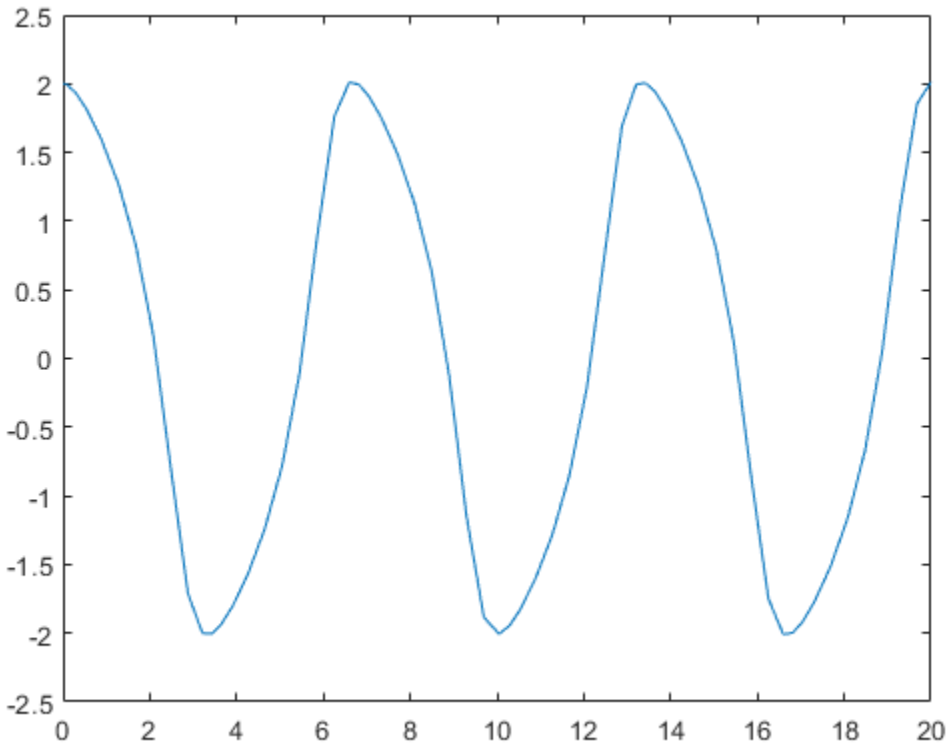
```
vdp
scopeConfig = get_param('vdp/Scope','ScopeConfiguration');
scopeConfig.DataLogging = true;
scopeConfig.DataLoggingSaveFormat = 'Dataset';
out = sim('vdp');
```





In the MATLAB Command window, view the logged data from the `SimulationOutput` object.

```
x1_data = out.ScopeData{1}.Values.Data(:,1);
x1_time = out.ScopeData{1}.Values.Time;
plot(x1_time,x1_data)
```



Pause Display While Running

Use the Simulink Snapshot to pause the scope display while the simulation keeps running in the background.

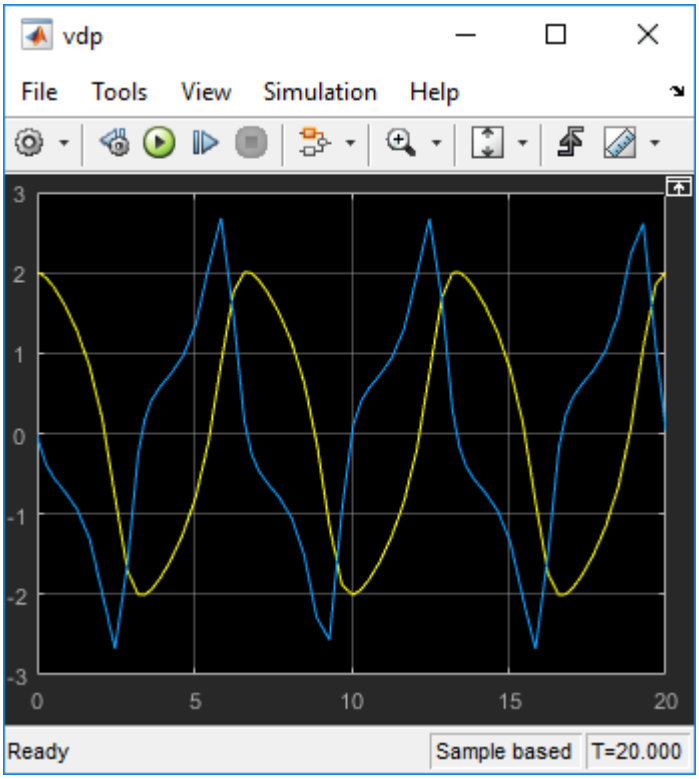
1. Open a scope window and start the simulation.
2. Select **Simulation** > **Simulink Snapshot**.

   The scope window status in the bottom left is **Frozen**, but the simulation continues to run in the background.
3. Interact with the paused display. For example, use measurements, copy the scope image, or zoom in or out.
4. To unfreeze the display, select **Simulation** > **Simulink Snapshot** again.

Copy Scope Image

This example uses the model vdp to demonstrate how to copy and paste a scope image.

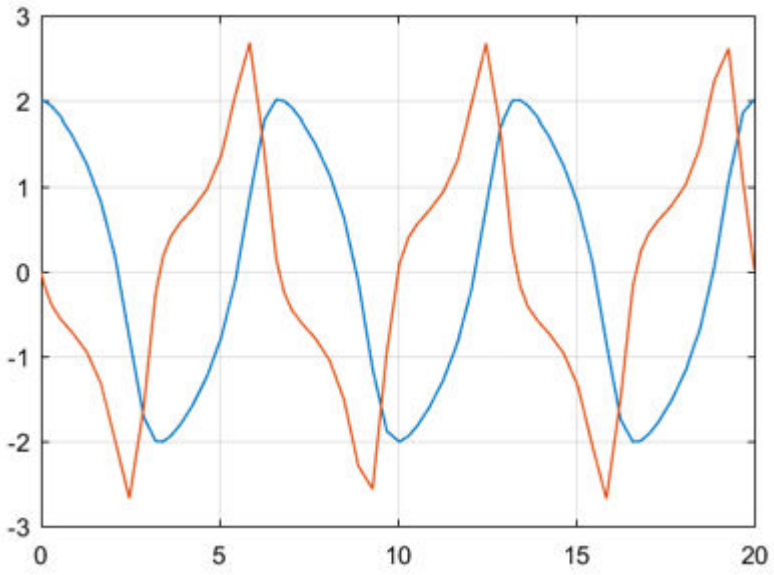
1. Add a scope block to your model.
2. Connect signals to scope ports. See “Connect Multiple Signals to a Scope” on page 24-83. For example, in the `vdp` model, connect the signals `x1` and `x2` to a scope.
3. Open the scope window and run the simulation.



4 Select **File > Copy to Clipboard**.

5 Paste the image into a document.

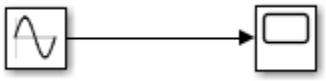
The van der Pol Equation results from my model:



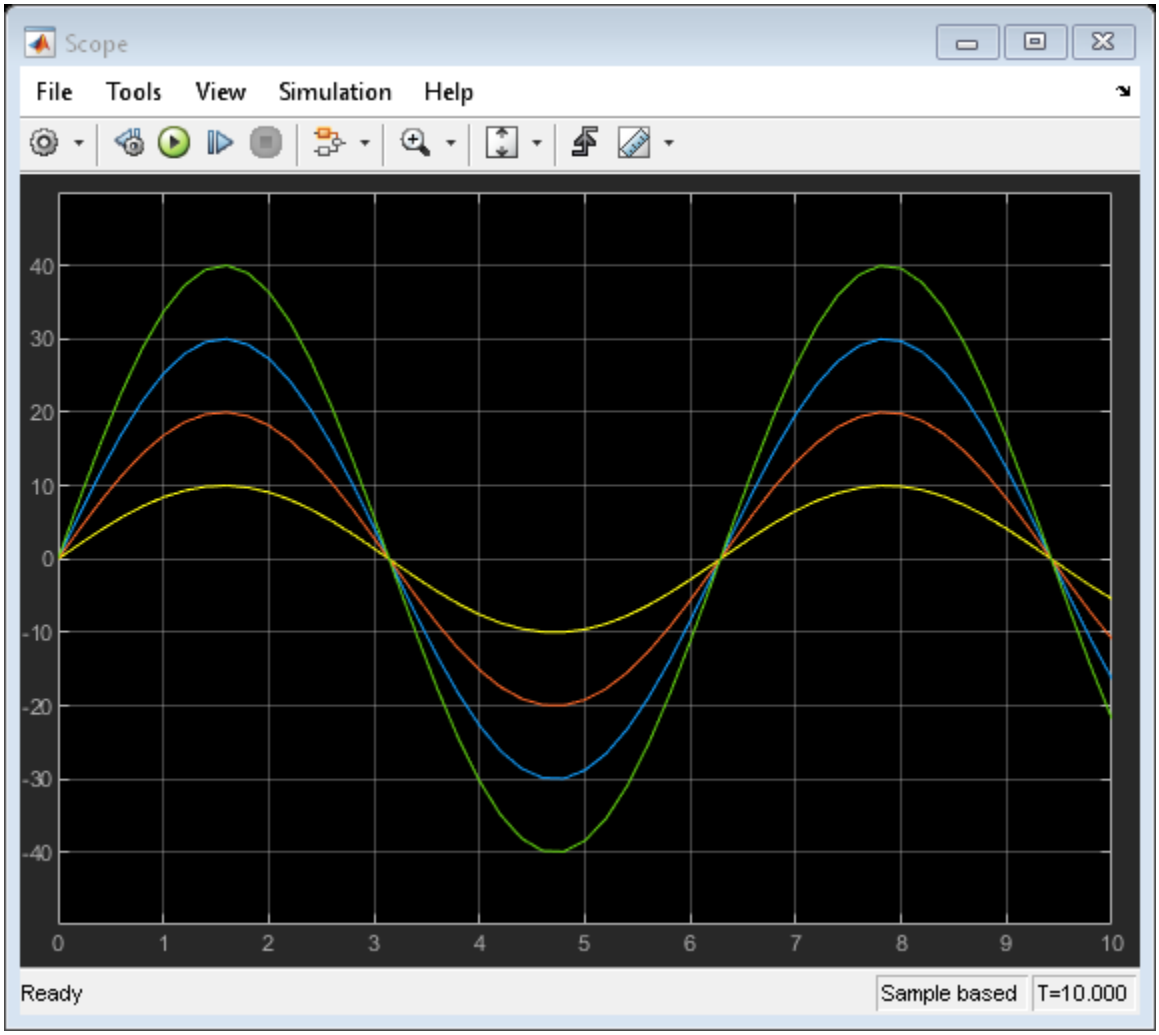
By default, **Copy to Clipboard** saves a printer-friendly version of the scope with a white background and visible lines. If you want to paste the exact scope plot displayed, select **View > Style**, then select the **Preserve colors for copy to clipboard** check box.

Plot an Array of Signals

This example shows how the scope plots an array of signals.



In this simple model, a Sine Wave block is connected to a scope block. The Sine Wave block outputs four signals with the amplitudes `[10, 20; 30 40]`. The scope displays each sine wave in the array separately in the matrix order (1,1), (2,1), (1,2), (2,2).



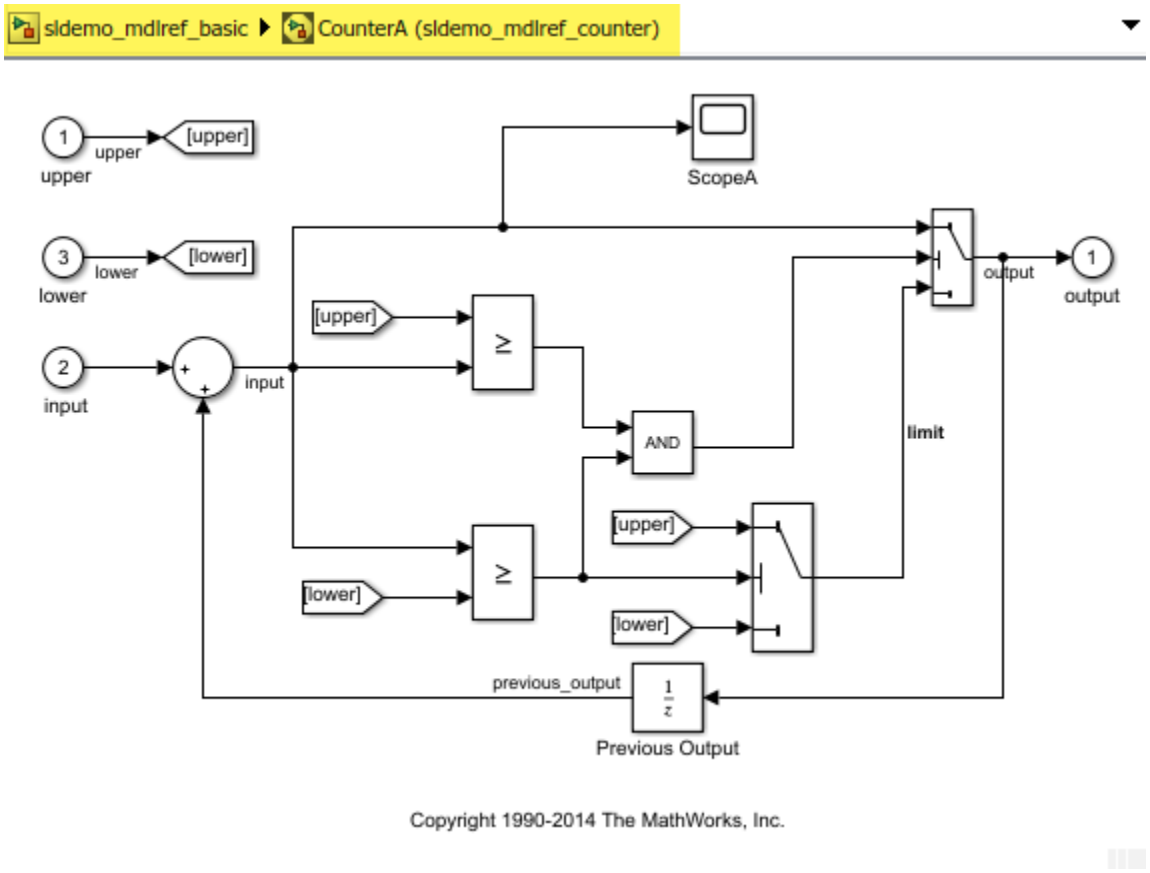
Scopes in Referenced Models

This example shows the behavior of scopes in referenced models. When you use a scope in a referenced model, you see different output in the scope depending on where you started the simulation: from the top model or the scope in the referenced model.

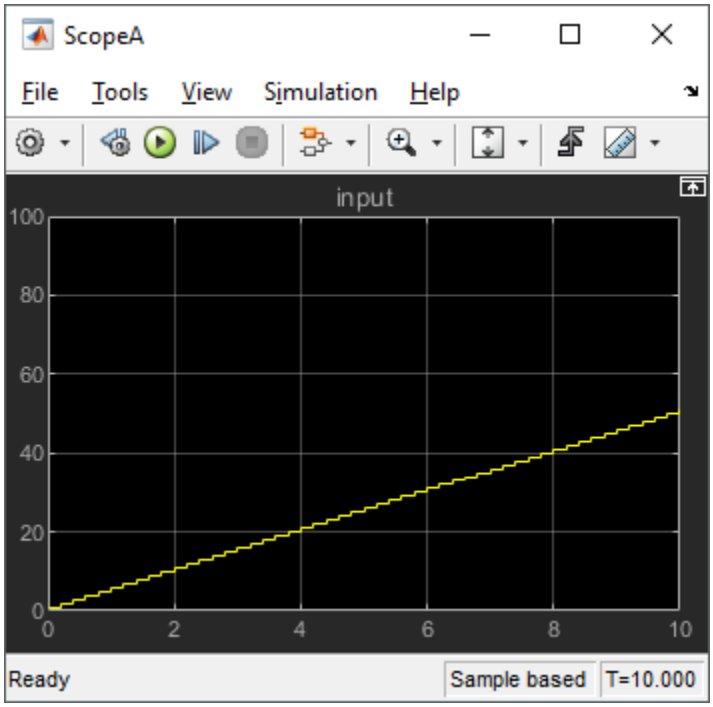
Note Scope windows display simulation results for the most recently opened top model. Playback controls in scope blocks and viewers simulate the model containing that block or viewer.

This example uses the `sldemo_mdlref_counter` model both as a top model and as a referenced model from the `sldemo_mdlref_basic` model.

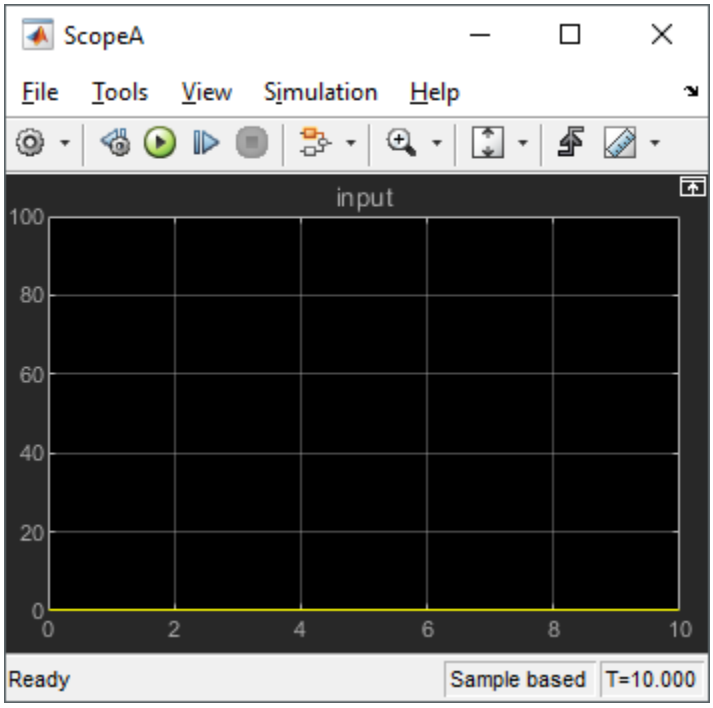
Open the `sldemo_mdlref_basic` model and double-click the CounterA block. The `sldemo_mdlref_counter` model opens as a referenced model, as evidenced by the breadcrumb above the canvas.



Run the simulation using the main run button, then open up the ScopeA scope. The scope visualizes the data from the entire model.



If you rerun the simulation using the run button in the scope, the scope only visualizes data as if the referenced model is opened in isolation. Playback controls in scope blocks and viewers simulate the model containing that block or viewer. In this case, the referenced model input, without the top model, is zero the entire time.

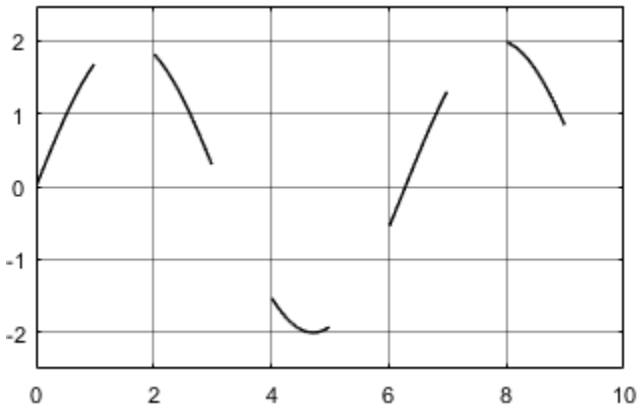


Note If you click run from the scope, the model does not show that the model is running in the background. For the simulation status, look at the status bar in the scope.

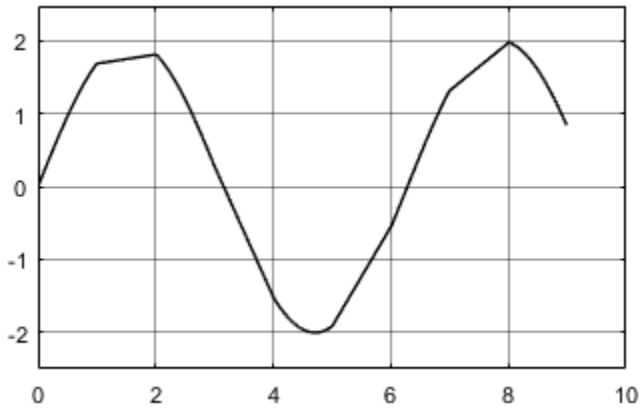
Scopes Within an Enabled Subsystem

When placed within an Enabled Subsystem block, scopes behave differently depending on the simulation mode:

- Normal mode — A scope plots data when the subsystem is enabled. The display plot shows gaps when the subsystem is disabled.

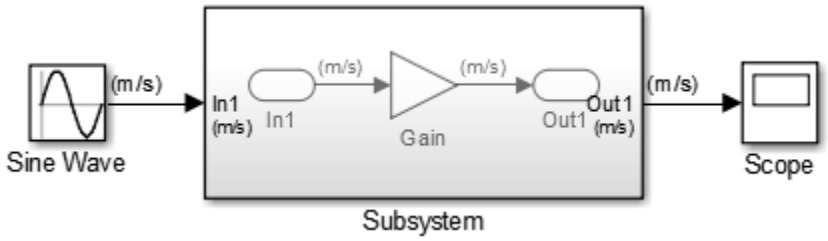


- External, Accelerator, and Rapid modes — A scope plots data when the subsystem is enabled. The display connects the gaps with straight lines.



Show Signal Units on a Scope Display

You can specify signal units at a model component boundary (Subsystem and Model blocks) using Inport and Outport blocks. See “Unit Specification in Simulink Models” (Simulink) . You can then connect a Scope block to an Outport block or a signal originating from an Outport block. In this example, the **Unit** property for the Out1 block was set to `m/s`.

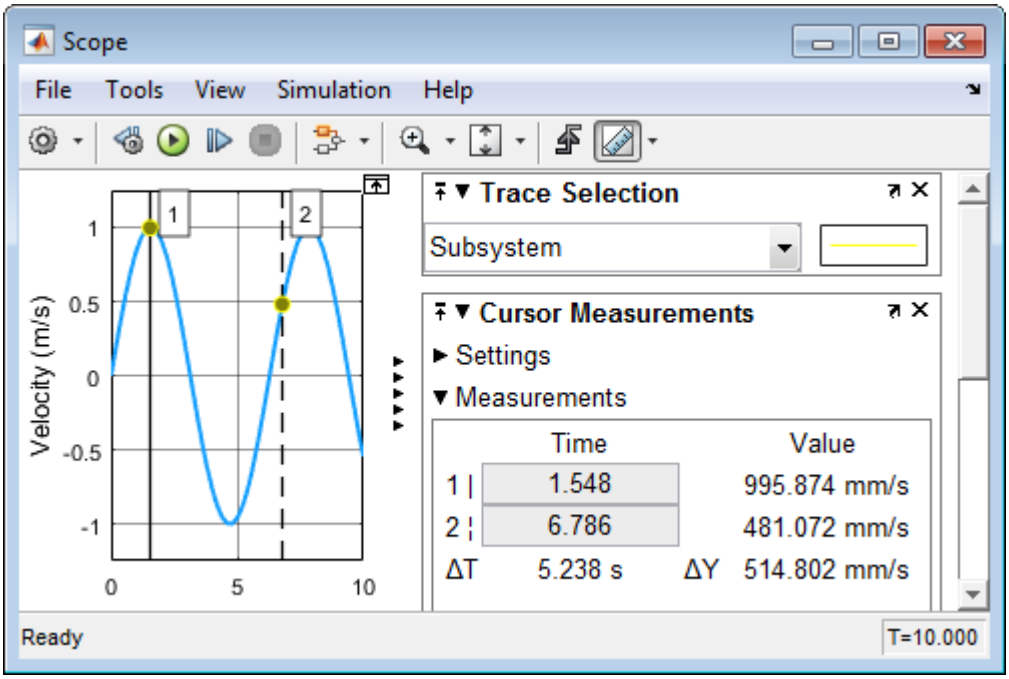


Show Units on a Scope Display

1. From the Scope window toolbar, select the Configuration Properties button .
2. In the Configuration Properties: Scope dialog box, select the **Display** tab.
3. In the **Y-label** box, enter a title for the y-axis followed by `(%<SignalUnits>)`. For example, enter

   ```
   Velocity (%<SignalUnits>)
   ```
4. Click **OK** or **Apply**.

   Signal units display in the *y*-axis label as meters per second (`m/s`) and in the Cursor Measurements panel as millimeters per second (`mm/s`).



From the Simulink toolstrip, you can also select **Debug** > **Information Overlays** > **Units**. You do not have to enter `(%<SignalUnits>)` in the **Y-Label** property.

Show Units on a Scope Display Programmatically

1. Get the scope properties. In the Command Window, enter

   ```
   load_system('my_model')
   s = get_param('my_model/Scope','ScopeConfiguration');
   ```

2 Add a *y*-axis label to the first display.

```
s.ActiveDisplay = 1
s.YLabel = 'Velocity (%<SignalUnits>)';
```

You can also set the model parameter `ShowPortUnits` to `'on'`. All scopes in your model, with and without `(%<SignalUnits>)` in the **Y-Label** property, show units on the displays.

```
load_system('my_model')
get_param('my_model','ShowPortUnits')

ans =
off

set_param('my_model', 'ShowPortUnits','on')

ans =
on
```

Determine Units from a Logged Data Object

When saving simulation data from a scope with the `Dataset` format, you can find unit information in the `DataInfo` field of the timeseries object.

Note Scope support for signal units is only for the `Dataset` logging format and not for the legacy logging formats `Array`, `Structure`, and `Structure With Time`.

1 From the Scope window toolbar, select the Configuration Properties button .

2 In the Configuration Properties window, select the **Logging** tab.

3 Select the **Log data to workspace** check box. In the text box, enter a variable name for saving simulation data. For example, enter `ScopeData`.

4 From the Scope window toolbar, select the run button .

5 In the Command Window, enter

```
ScopeData.getElement(1).Values.DataInfo

Package: tsdata
Common Properties:
             Units: m/s (Simulink.SimulationData.Unit)
     Interpolation: linear (tsdata.interpolation)
```

Connect Signals with Different Units to a Scope

When there are multiple ports on a scope, Simulink ensures that each port receives data with only one unit. If you try to combine signals with different units (for example by using a Bus Creator block), Simulink returns an error.

Scopes show units depending on the number of ports and displays:

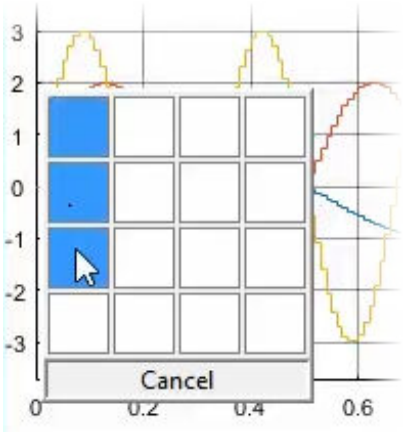
- **Number of ports equal to the number of displays** — One port is assigned to one display with units for the port signal shown on the *y*-axis label.

- **Greater than the number of displays** — One port is assigned to one display, with the last display assigned the remaining signals. Different units are shown on the last *y*-axis label as a comma-separated list.

Select Number of Displays and Layout

1. From a Scope window, select the Configuration Properties button .
2. In the Configuration Properties dialog box, select the **Main** tab, and then select the **Layout** button.
3. Select the number of displays and the layout you want.

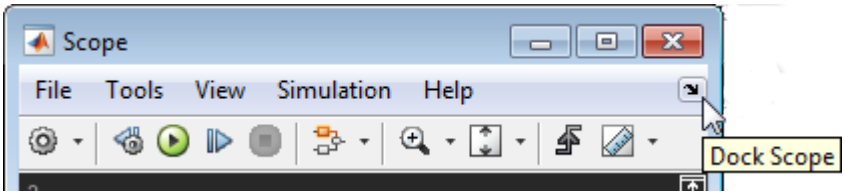
   You can select more than four displays in a row or column. Click within the layout, and then drag your mouse pointer to expand the layout to a maximum of 16 rows by 16 columns.



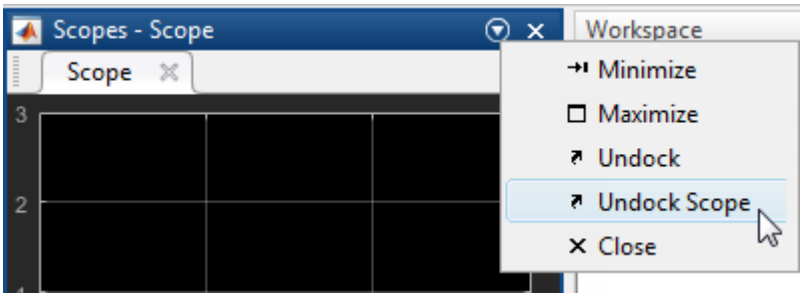
4. Click to apply the selected layout to the Scope window.

Dock and Undock Scope Window to MATLAB Desktop

1. In the right corner of a Scope window, click the Dock Scope button.



   The Scope window is placed above the Command Window in the MATLAB desktop.
2. Click the Show Scope Actions button, and then click **Undock Scope**.



See Also

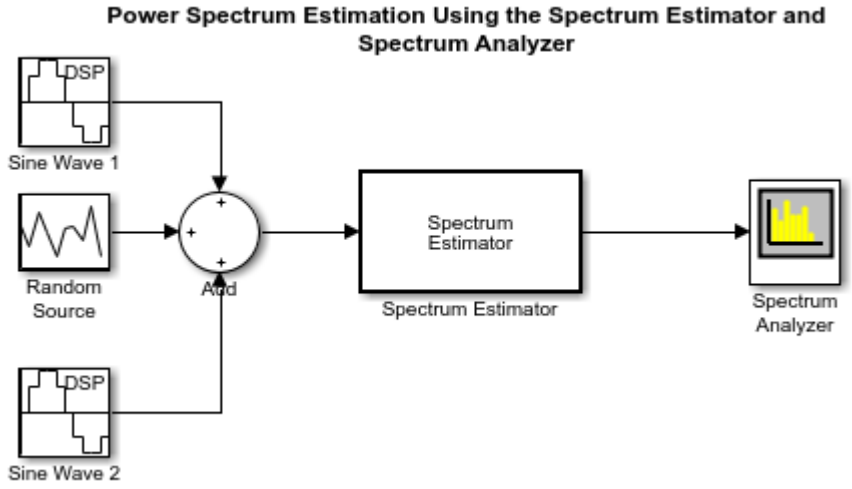
Floating Scope | Scope | Scope Viewer

Related Examples

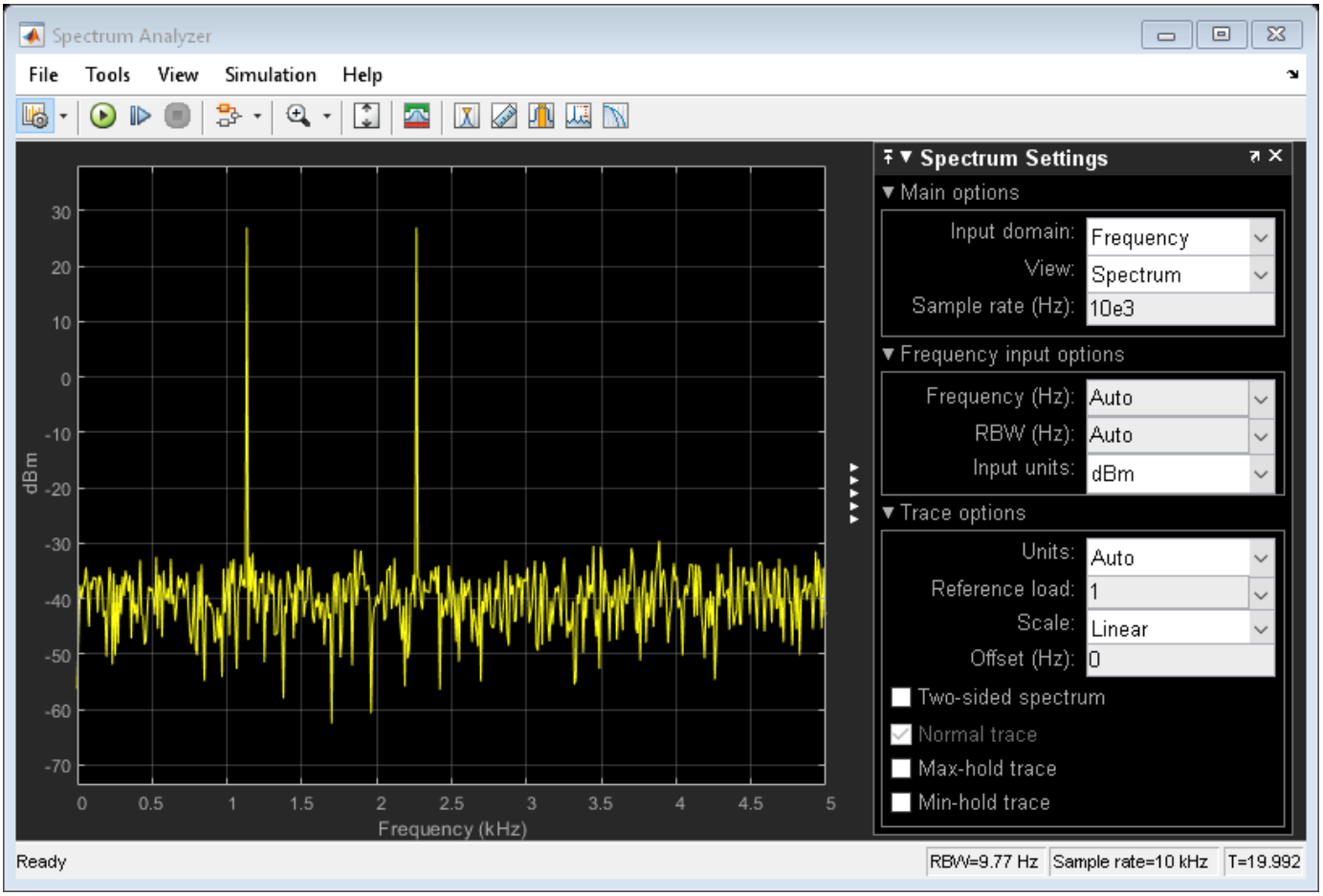
- “Scope Blocks and Scope Viewer Overview” (Simulink)
- “Floating Scope and Scope Viewer Tasks” (Simulink)

Display Frequency Input on Spectrum Analyzer

This example shows how to visualize frequency input signals with the Spectrum Analyzer block.



To visualize frequency-domain input signals using a Spectrum Analyzer block, in the Spectrum Settings, set **Input domain** to Frequency. In this example, also clear the **Two-sided spectrum** check box.



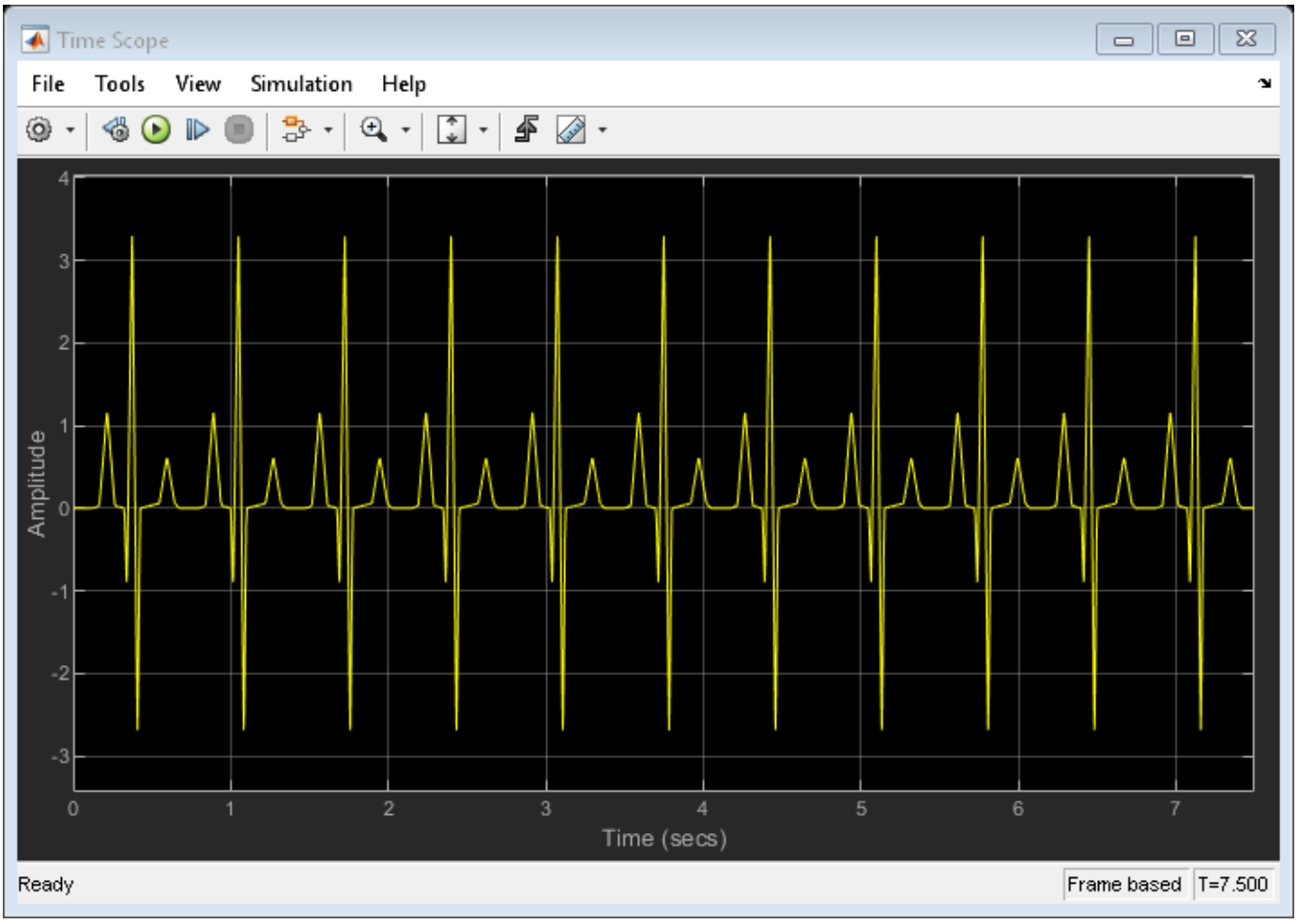
You can see two peaks at 5 kHz and 10 kHz. To measure these peaks, use the Peak Finder measurement tool.

Use Peak Finder to Find Heart Rate from ECG Input

This example shows how to use the Time Scope Peak Finder panel to measure the heart rate from an ECG.

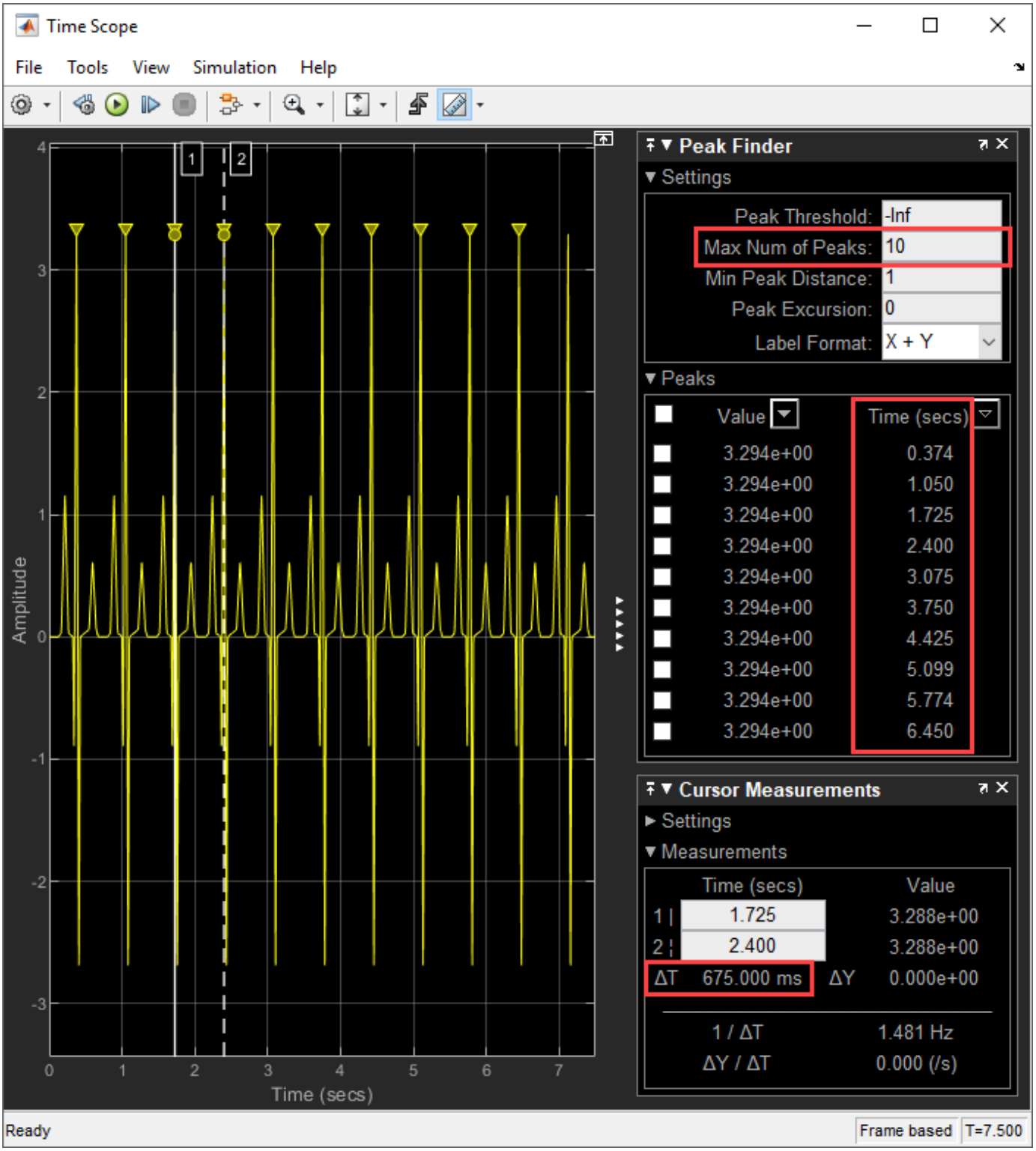
To emulate a heart beat, the model Preload creates the variable `mhb` in the MATLAB® workspace. This variable is then called by the Signal From Workspace block. To see all the model Preload commands, open the **Model Explorer** and look at the Callback functions.

Run your model to see the time domain output.



1 To show the **Peak Finder** panel, in the Time Scope menu, select **Tools > Measurements > Peak Finder**.

2. Expand the Settings section, enter 10 for **Max Num of Peaks**. The Time Scope Peaks section now displays a list of 10 peak amplitude values, and the times at which they occur.
3. Turn on the Cursor Measurements by selecting **Tools > Measurements > Cursor Measurements**.
4. Set the cursor time values to two consecutive peak times, for example 1.725 and 2.4. The time difference between all peaks is 675 milliseconds.



Therefore, you can calculate the heart rate of the ECG signal:

$$\frac{1\ beat}{675\ ms} * \frac{60000\ ms}{1\ min} = 88.88\ bpm$$

Logic Analyzer

- “Inspect and Measure Transitions Using the Logic Analyzer” on page 25-2
- “Configure Logic Analyzer” on page 25-8

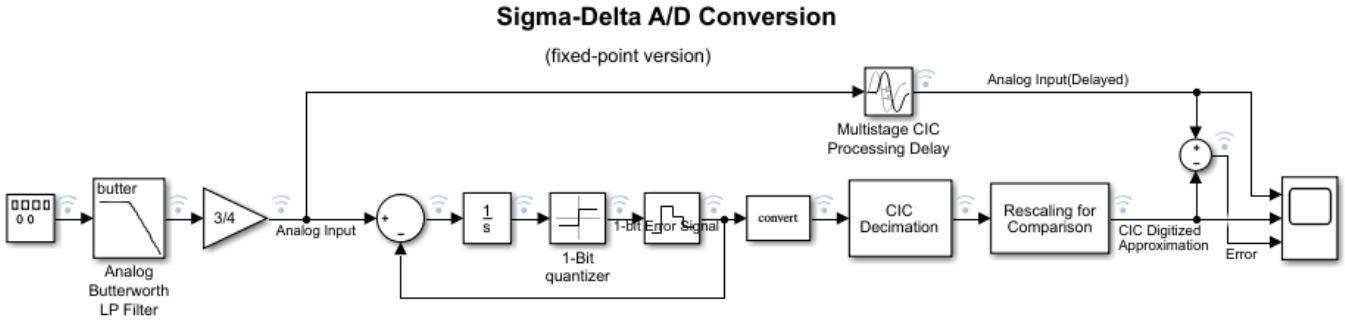
Inspect and Measure Transitions Using the Logic Analyzer

In this section...
“Open a Simulink Model” on page 25-2 “Open the Logic Analyzer” on page 25-2 “Configure Global Settings and Visual Layout” on page 25-3 “Set Stepping Options” on page 25-4 “Run Model” on page 25-5 “Configure Individual Wave Settings” on page 25-5 “Inspect and Measure Transitions” on page 25-5 “Step Through Simulation” on page 25-7 “Save Logic Analyzer Settings” on page 25-7

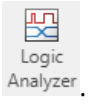
In this tutorial, you explore key functionality of the **Logic Analyzer**, such as choosing and configuring signals to visualize, stepping through a simulation, and measuring transitions.

Open a Simulink Model

To follow along with this tutorial, open the Sigma-Delta A/D Conversion (fixed-point version) model (`dspsdadc_fixpt`).

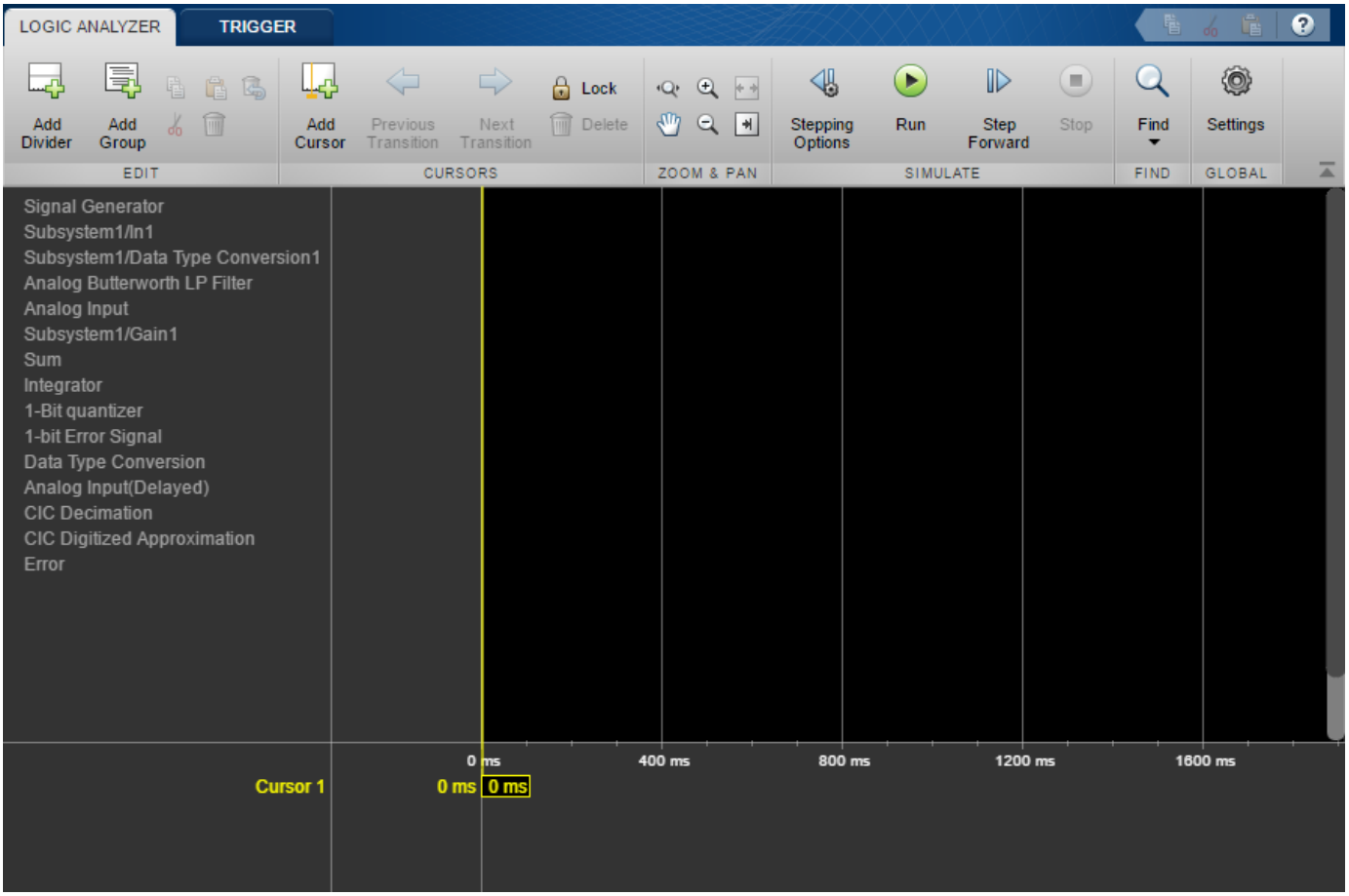


Open the Logic Analyzer

From the Simulink toolstrip, on the **Simulation** tab, in the Review Results gallery, click the Logic Analyzer button 

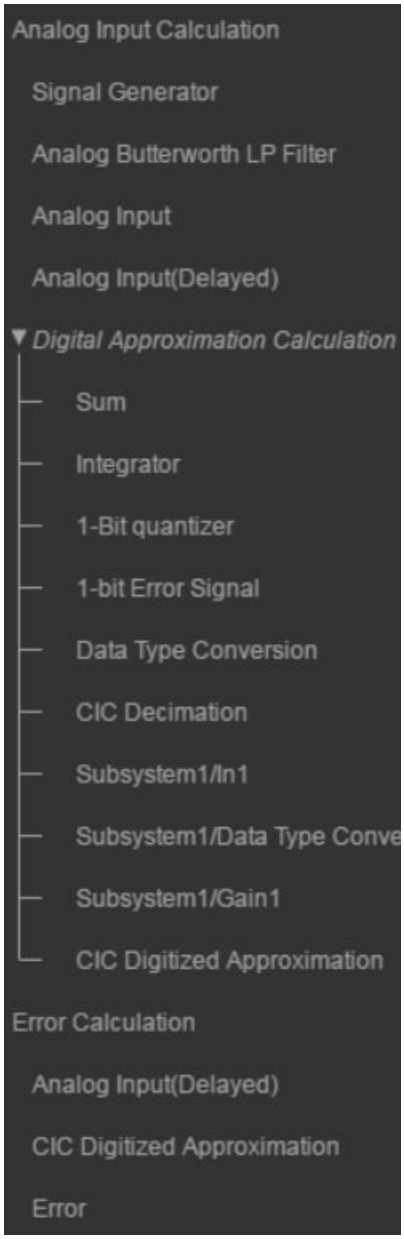
.

The **Logic Analyzer** opens with the selected signals shown in the channel display.



Configure Global Settings and Visual Layout

1. Click **Settings**. Set the **Height** to 20 and the **Spacing** to 10, and then click **OK**.
2. From the **Logic Analyzer** toolstrip, click **Add Divider**. A divider named `Divider` is added to the bottom of your channels. You can use dividers to separate signals.
3. Double-click `Divider` and rename `Divider` as `Analog Input Calculation`. Drag the divider to the top of the channels pane.
4. Add another divider and name it `Error Calculation`.
5. From the **Logic Analyzer** toolstrip, click **Add Group**. A group named `Group1` is added to the bottom of your channels. You can use groups to group signals in a collapsible tree structure. Double-click `Group1` and rename it as `Digital Approximation Calculation`.
6. You can visualize the same signal in multiple places. Right-click the `Analog Input(Delayed)` signal and select **Copy**. Paste this signal under the `Error Calculation` divider. Repeat the process for the `CIC Digitized Approximation` signal. Organize your dividers and signals as shown in the screen shot, and then collapse the `Digital Approximation Calculation` group.

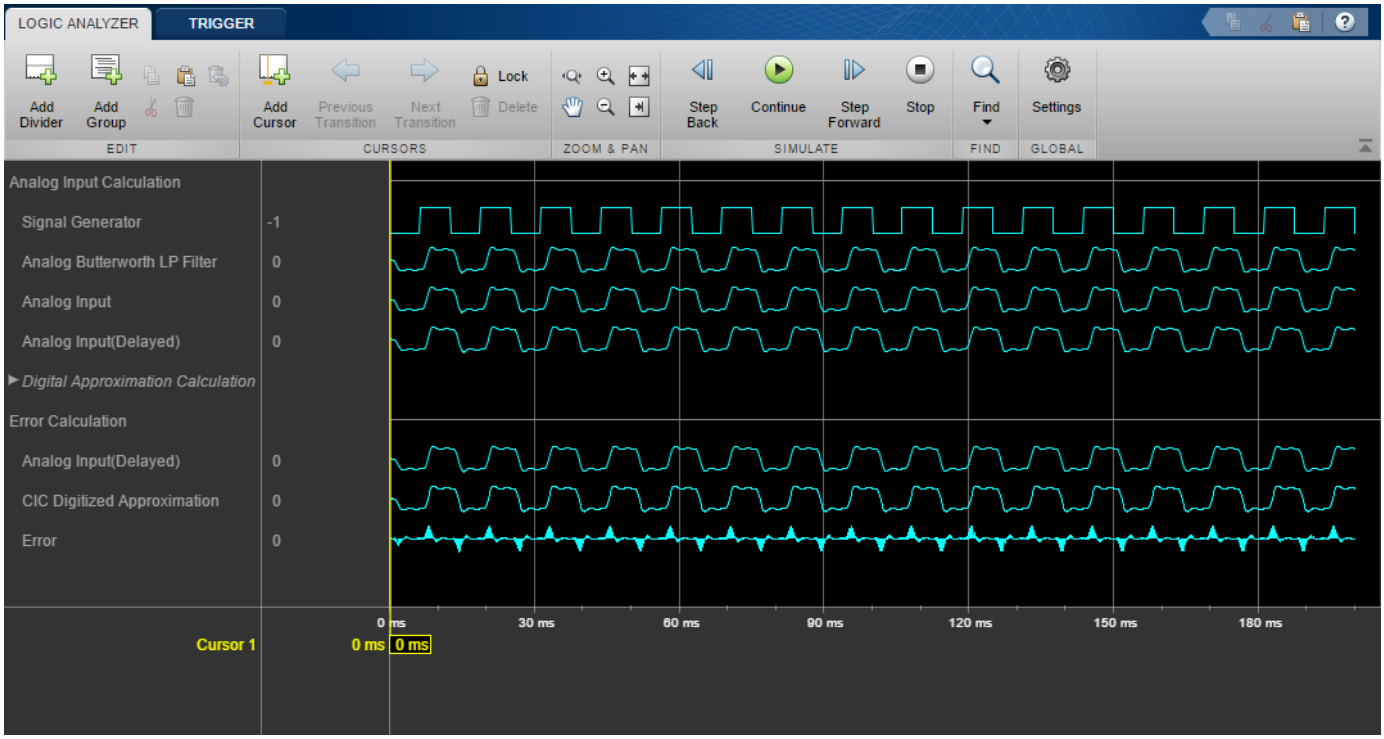


Set Stepping Options

1. From the **Logic Analyzer** toolstrip, click **Stepping Options**.
2. Select the **Enable stepping back** option. Specify the **Maximum number of saved back steps** as 2 and the **Interval between stored back steps** as `100` steps. When you run the simulation, a snapshot of the model is taken every 100 steps. Only the last snapshot is saved.
3. Set **Move back/forward by** to `100` **steps**.
4. Select the **Pause simulation when time reaches** option. Specify the simulation to pause after `0.2` seconds of model time has elapsed, and then click **OK**.

Run Model

1. To run the model, click **Run** on the **Logic Analyzer** toolstrip. The model runs for 0.2 seconds of model time and then pauses.
2. Click to fit your data to the time range.

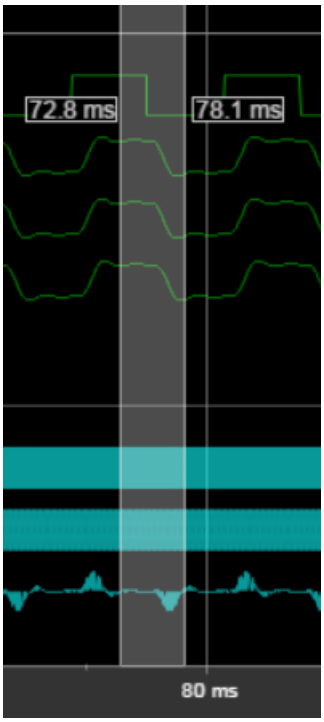


Configure Individual Wave Settings

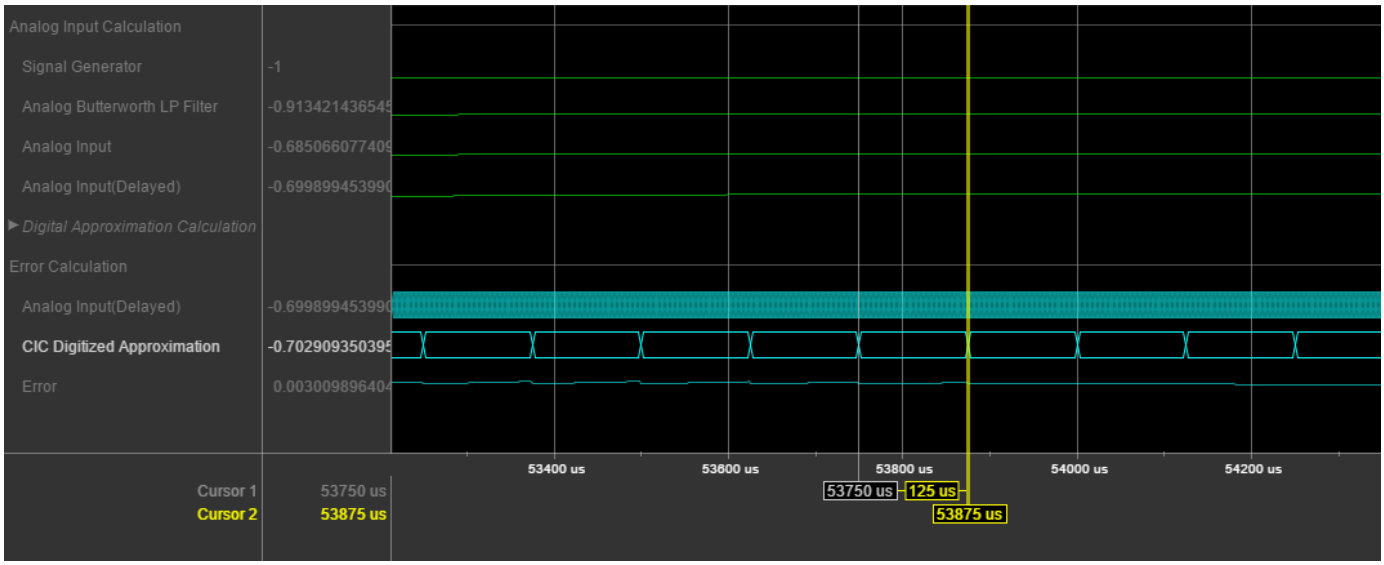
1. Select all waves under your `Analog Input Calculation` divider. Then on the **Waves** tab, select a new **Wave Color** for the selected waves.
2. Under the `Error Calculation` divider, select the `Analog Input(Delayed)` and `CIC Digitized Approximation` waves. On the **Waves** tab, modify the **Format** to `Digital`. The selected waves are now displayed as digital transitions.

Inspect and Measure Transitions

1. On the **Logic Analyzer** toolstrip, click and then drag-and-drop start and end points to zoom in time.



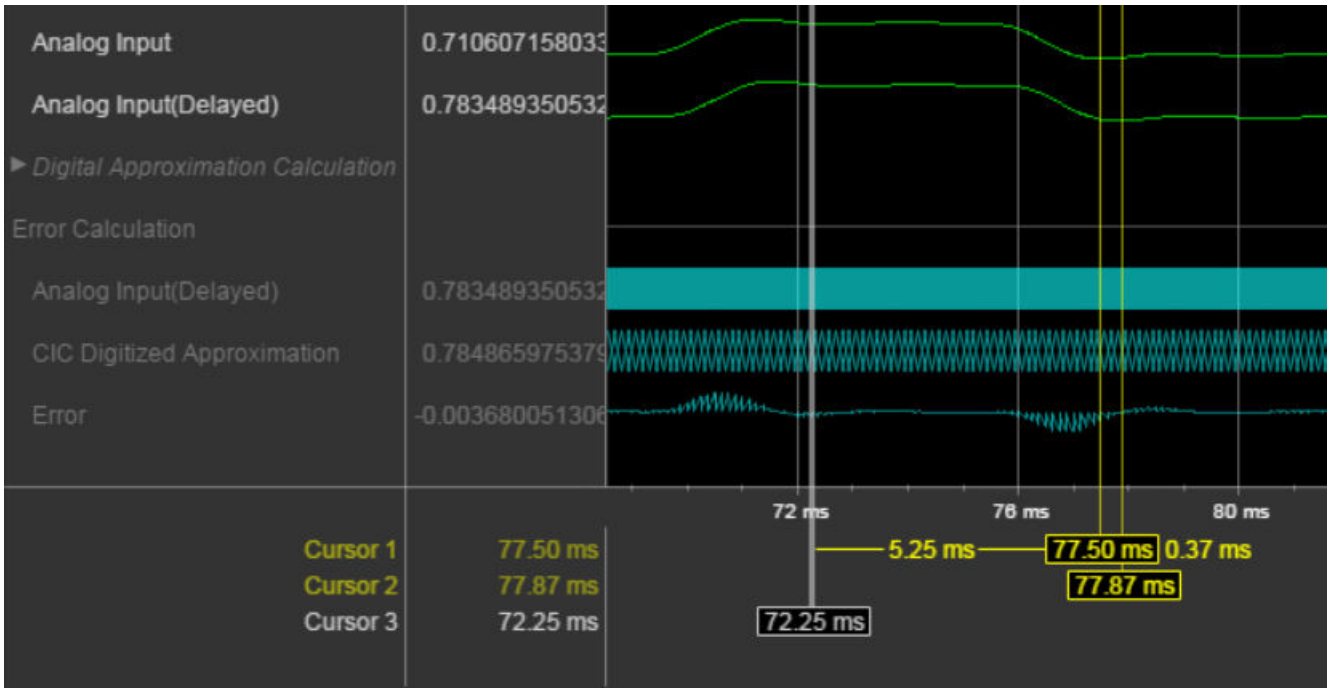
2. For waves displayed as digital, you can use the **Next Transition** and **Previous Transition** buttons. To move the active cursor to the next transition, click **Next Transition**.
3. Click **Lock** to lock the active cursor in place.
4. Click **Add Cursor** to add another cursor to the axes. The cursor shows its current position in time, and the difference from all surrounding cursors in time.



5. Right-click the second cursor you added and select **Delete Cursor**.
6. Press the space bar to zoom out.

7 Add another cursor and line it up with a low point of the `Analog Input` wave in your `Analog Input Calculation` division. Use the value displayed in the wave value pane to fine-tune the cursor position in time.

8 Add another cursor and line it up with the corresponding low point of the `Analog Input(Delayed)` wave in your `Analog Input Calculation` division.



Step Through Simulation

1 To move the simulation forward 100 steps, click **Step Forward**. The time axis adjusts so that you can see the most recent data.

2 To move the simulation backward 100 steps, click **Step Back**. The **Step Back** button becomes disabled because you specified saving only two back step.

Save Logic Analyzer Settings

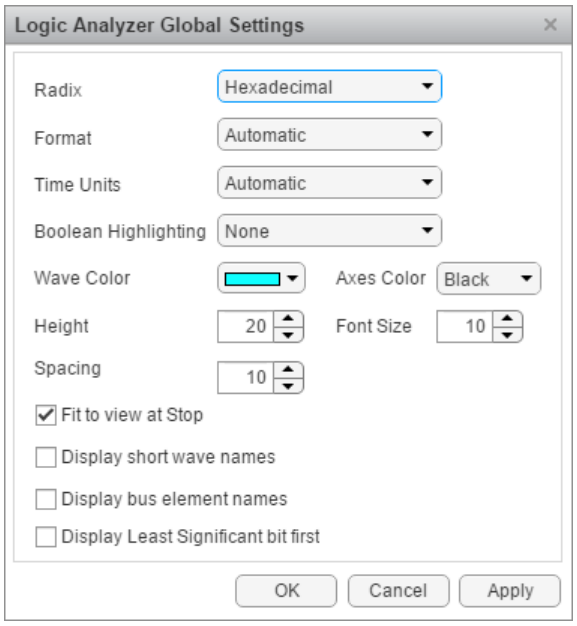
When you save your model, the logic analyzer settings are also saved for that model.

See Also

Logic Analyzer | `dsp.LogicAnalyzer`

Configure Logic Analyzer

Open the **Logic Analyzer** and select **Settings** from the toolstrip. A global settings dialog box opens. Any setting you change for an individual signal supersedes the global setting. The Logic Analyzer saves any setting changes with the model (Simulink) or System object (MATLAB).



Set the display **Radix** of your signals as one of the following:

- `Hexadecimal` — Displays values as symbols from zero to nine and A to F
- `Octal` — Displays values as numbers from zero to seven
- `Binary` — Displays values as zeros and ones
- `Signed decimal` — Displays the signed, stored integer value
- `Unsigned decimal` — Displays the stored integer value

Set the display **Format** as one of the following:

- `Automatic` — Displays floating point signals in `Analog` format and integer and fixed-point signals in `Digital` format. Boolean signals are displayed as zero or one.
- `Analog` — Displays values as an analog plot
- `Digital` — Displays values as digital transitions

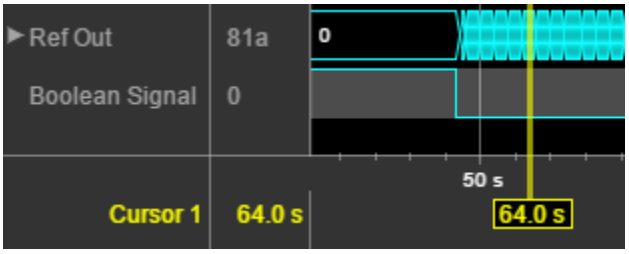
Set the display **Time Units** to one of the following:

- `Automatic` — Uses a time scale appropriate to the time range shown in the current plot
- `seconds`
- `milliseconds`
- `microseconds`
- `nanoseconds`
- `picoseconds`

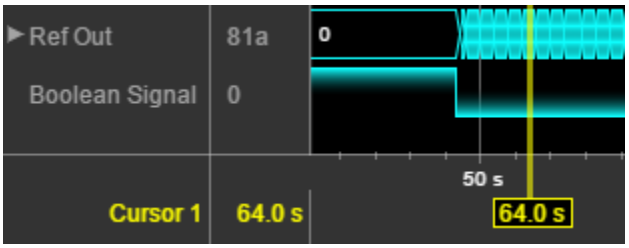
- `femtoseconds`

Set the **Boolean Highlighting** to one of the following:

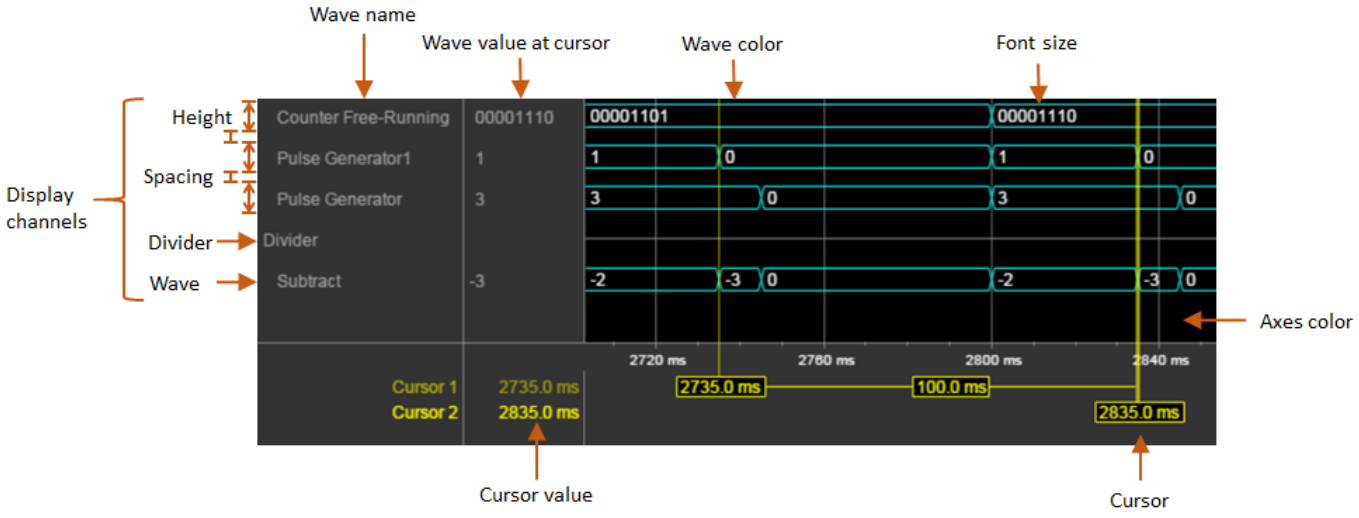
- `None`
- `Rows` — Adds a highlighted background for the entire Boolean signal row.



  Select **Highlight boolean values** to add highlighting to Boolean signals.
- `Gradient`— Adds color highlighting to Boolean signals based on value. If the signal value is `true`, the highlight fades out below. If the signal value is `false`, the signal fades out above. With this option, you can visually deduce the value of the signal.



Inspect the graphic for an explanation of the global settings: `Wave Color`, `Axes Color`, `Height`, `Font Size`, and `Spacing`. `Font Size` applies only to the text within the axes.



By default, when your simulation stops, the Logic Analyzer shows all the data for the simulation time on one screen. If you do not want this behavior, clear **Fit to view at Stop**. This option is disabled for long simulation times.

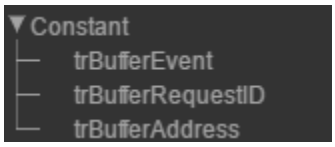
To display the short names of waves without path information, select **Display short wave names**.

You can expand fixed-point and integer signals and view individual bits. The **Display Least Significant bit first** option enables you to reverse the order of the displayed bits.

If you stream logged bus signals to the Logic Analyzer, you can display the names of the signals inside the bus using the **Display bus element names** option. To show bus element names:

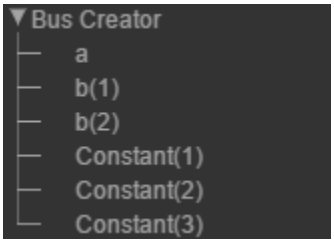
1. Add the bus signal for logging.
2. In the Logic Analyzer settings, select the **Display bus element names** check box.
3. Run the simulation.

When you expand the bus signals, you will see the bus signal names.

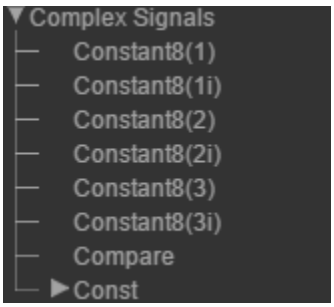


Some special situations:

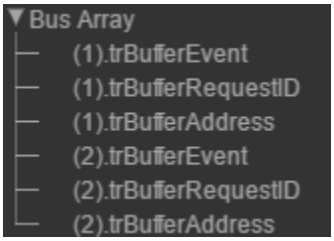
- If the signal has no name, the Logic Analyzer shows the block name instead.
- If the bus is a bus object, the Logic Analyzer shows the bus element names specified in the Bus Object Editor.
- If one of the bus elements contains an array, each element of the array is appended with the element index.

- If a bus element contains an array with complex elements, the real and complex values (`i`) are split.

- Bus signals passed through a Gain block are labeled `Gain(1)`, `Gain(2)`,...`Gain(n)`.
- If the bus contains an array of buses, the Logic Analyzer prepends the element name with the bus array index.


Bus Array
(1).trBufferEvent
(1).trBufferRequestID
(1).trBufferAddress
(2).trBufferEvent
(2).trBufferRequestID
(2).trBufferAddress

Statistics and Linear Algebra

- “What Are Moving Statistics?” on page 26-2
- “Sliding Window Method and Exponential Weighting Method” on page 26-5
- “Measure Statistics of Streaming Signals” on page 26-14
- “How Is a Moving Average Filter Different from an FIR Filter?” on page 26-17
- “Energy Detection in the Time Domain” on page 26-21
- “Remove High-Frequency Noise from Gyroscope Data” on page 26-24
- “Measure Pulse and Transition Characteristics of Streaming Signals” on page 26-26
- “Linear Algebra and Least Squares” on page 26-34

What Are Moving Statistics?

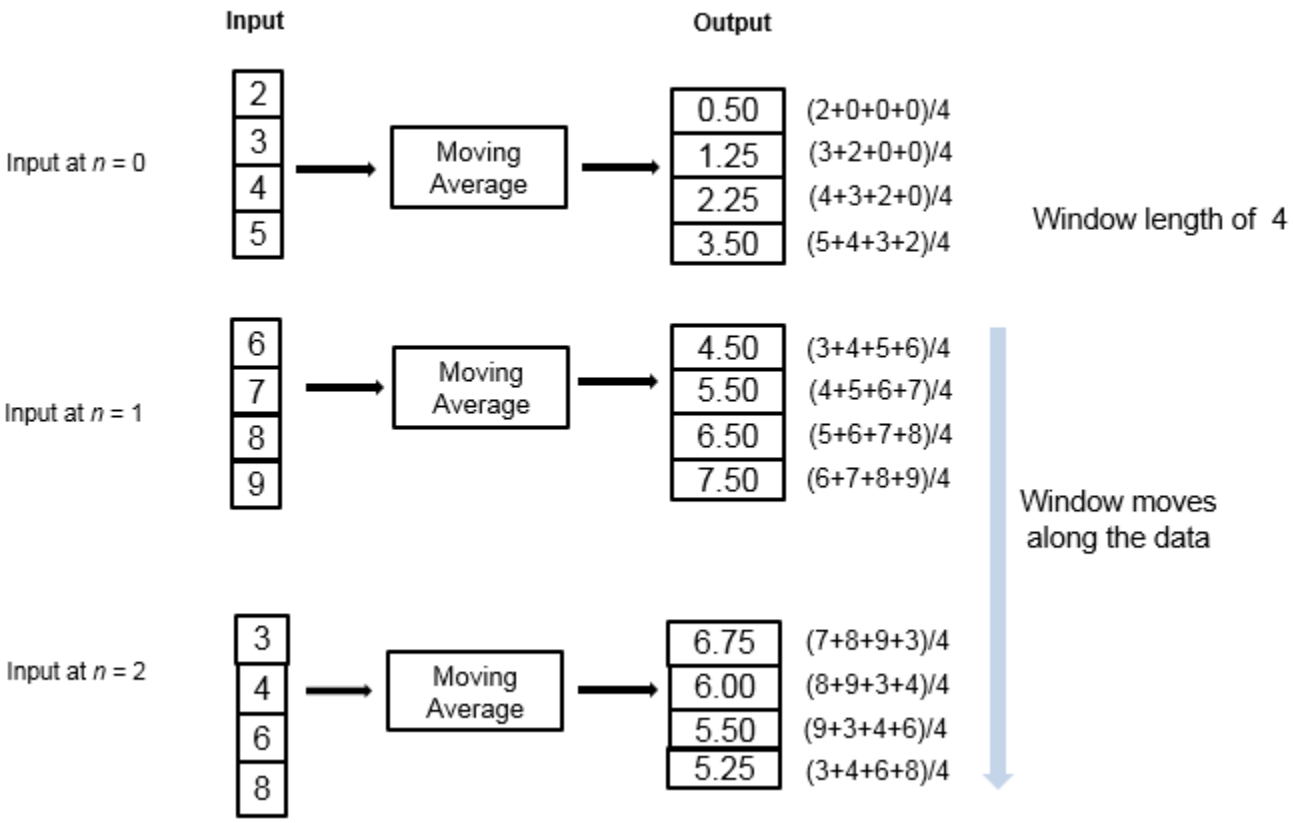
Note Simulink is not supported in MATLAB Online.

You can measure statistics of streaming signals in MATLAB and Simulink along each independent data channel using the moving statistics System objects and blocks. Statistics such as average, RMS, standard deviation, variance, median, maximum, and minimum change as the data changes constantly with time. With every data sample that comes in, the System objects and blocks compute the statistics over the current sample and a specific window of past samples. This window "moves" as new data comes in.

MATLAB System object	Simulink Block	Statistic Computed
`dsp.MedianFilter`	Median Filter	Moving median
`dsp.MovingAverage`	Moving Average	Moving average
`dsp.MovingMaximum`	Moving Maximum	Moving maximum
`dsp.MovingMinimum`	Moving Minimum	Moving minimum
`dsp.MovingRMS`	Moving RMS	Moving RMS
`dsp.MovingStandardDeviation`	Moving Standard Deviation	Moving standard deviation
`dsp.MovingVariance`	Moving Variance	Moving variance

These System objects and blocks compute the moving statistic using one or both of the sliding window method and exponential weighting method. For more details on these methods, see “Sliding Window Method and Exponential Weighting Method” on page 26-5.

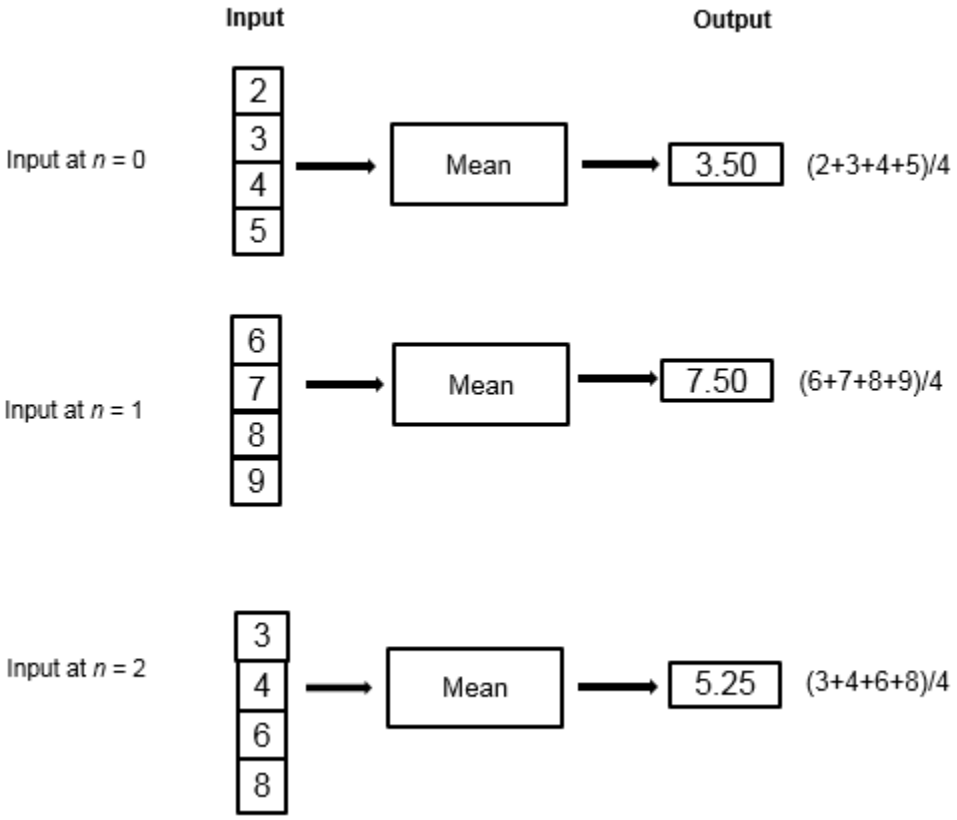
Consider an example of computing the moving average of a streaming input data using the sliding window method. The algorithm uses a window length of 4. At the first time step, the algorithm fills the window with three zeros to represent the first three samples. In the subsequent time steps, to fill the window, the algorithm uses samples from the previous data frame. The moving statistic algorithms have a state and remember the previous data.



If the data is stationary, use the stationary statistics blocks to compute the statistics over the entire data in Simulink. Stationary blocks include Autocorrelation, Correlation, Maximum, Mean, Median, Minimum, RMS, Sort, Standard Deviation, and Variance.

These blocks do not maintain a state. When a new data sample comes in, the algorithm computes the statistic over the entire data and has no influence from the previous state of the block.

Consider an example of computing the stationary average of streaming input data using the Mean block in Simulink. The Mean block is configured to find the mean value over each column.



At each time step, the algorithm computes the average over the entire data that is available in the current time step and does not use data from the previous time step. The stationary statistics blocks are more suitable for data that is already available rather than for streaming data.

See Also

More About

- “Measure Statistics of Streaming Signals” on page 26-14
- “Sliding Window Method and Exponential Weighting Method” on page 26-5
- “How Is a Moving Average Filter Different from an FIR Filter?” on page 26-17
- “Signal Statistics”
- “Energy Detection in the Time Domain” on page 26-21
- “Remove High-Frequency Noise from Gyroscope Data” on page 26-24
- “Measure Pulse and Transition Characteristics of Streaming Signals” on page 26-26

Sliding Window Method and Exponential Weighting Method

In this section...
“Sliding Window Method” on page 26-5 “Exponential Weighting Method” on page 26-7

Note Simulink is not supported in MATLAB Online.

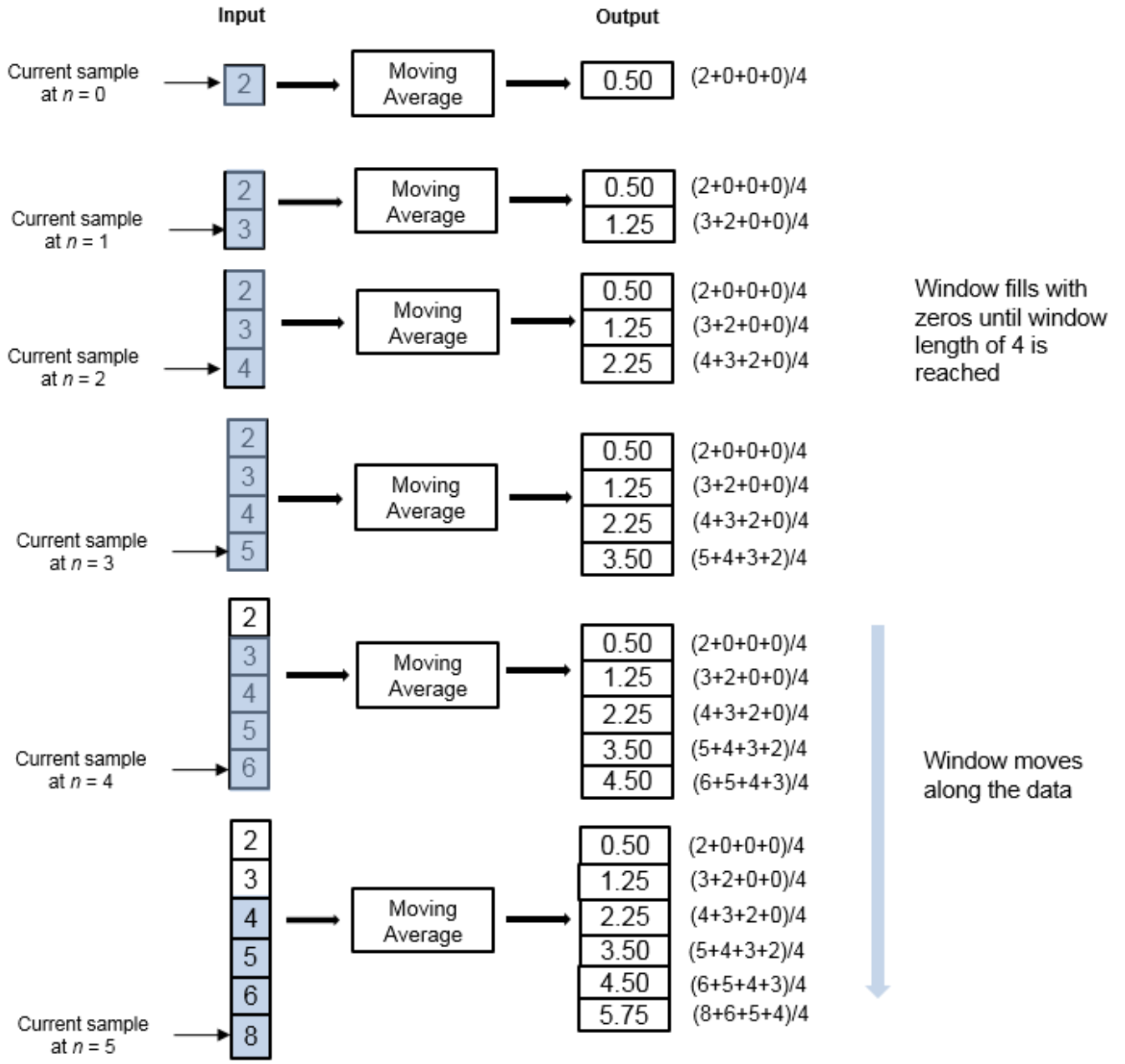
The moving objects and blocks compute the moving statistics of streaming signals using one or both of the sliding window method and exponential weighting method. The sliding window method has a finite impulse response, while the exponential weighting method has an infinite impulse response. To analyze a statistic over a finite duration of data, use the sliding window method. The exponential weighting method requires fewer coefficients and is more suitable for embedded applications.

Object, Block	Sliding Window Method	Exponential Weighting Method
`dsp.MedianFilter`, Median Filter	✓	
`dsp.MovingAverage`, Moving Average	✓	✓
`dsp.MovingMaximum`, Moving Maximum	✓	
`dsp.MovingMinimum`, Moving Minimum	✓	
`dsp.MovingRMS`, Moving RMS	✓	✓
`dsp.MovingStandardDeviation`, Moving Standard Deviation	✓	✓
`dsp.MovingVariance`, Moving Variance	✓	✓

Sliding Window Method

In the sliding window method, a window of specified length, *Len*, moves over the data, sample by sample, and the statistic is computed over the data in the window. The output for each input sample is the statistic over the window of the current sample and the *Len* - 1 previous samples. In the first-time step, to compute the first *Len* - 1 outputs when the window does not have enough data yet, the algorithm fills the window with zeros. In the subsequent time steps, to fill the window, the algorithm uses samples from the previous data frame. The moving statistic algorithms have a state and remember the previous data.

Consider an example of computing the moving average of a streaming input data using the sliding window method. The algorithm uses a window length of 4. With each input sample that comes in, the window of length 4 moves along the data.



The window is of finite length, making the algorithm a finite impulse response filter. To analyze a statistic over a finite duration of data, use the sliding window method.

Effect of Window Length

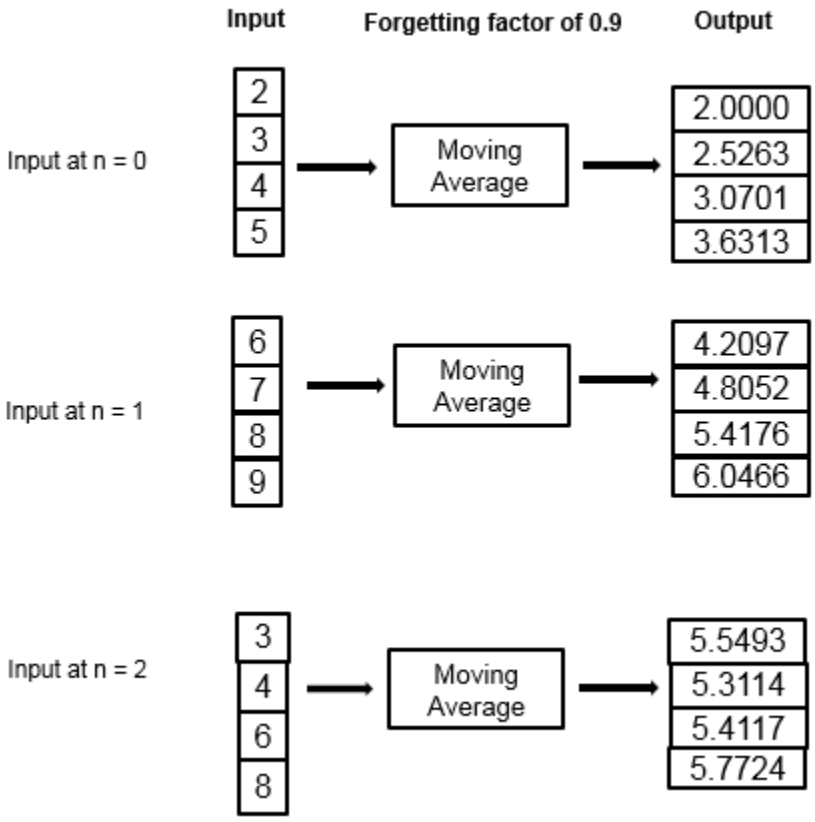
The window length defines the length of the data over which the algorithm computes the statistic. The window moves as the new data comes in. If the window is large, the statistic computed is closer to the stationary statistic of the data. For data that does not change rapidly, use a long window to get a smoother statistic. For data that changes fast, use a smaller window.

Exponential Weighting Method

The exponential weighting method has an infinite impulse response. The algorithm computes a set of weights, and applies these weights to the data samples recursively. As the age of the data increases, the magnitude of the weighting factor decreases exponentially and never reaches zero. In other words, the recent data has more influence on the statistic at the current sample than the older data. Due to the infinite impulse response, the algorithm requires fewer coefficients, making it more suitable for embedded applications.

The value of the forgetting factor determines the rate of change of the weighting factors. A forgetting factor of 0.9 gives more weight to the older data than does a forgetting factor of 0.1. To give more weight to the recent data, move the forgetting factor closer to 0. For detecting small shifts in rapidly varying data, a smaller value (below 0.5) is more suitable. A forgetting factor of 1.0 indicates infinite memory. All the previous samples are given an equal weight. The optimal value for the forgetting factor depends on the data stream. For a given data stream, to compute the optimal value for forgetting factor, see [1].

Consider an example of computing the moving average using the exponential weighting method. The forgetting factor is 0.9.



The moving average algorithm updates the weight and computes the moving average recursively for each data sample that comes in by using the following recursive equations.

$$w_{N,\lambda} = \lambda w_{N-1,\lambda} + 1$$

$$\bar{x}_{N,\lambda} = \left(1 - \frac{1}{w_{N,\lambda}}\right)\bar{x}_{N-1,\lambda} + \left(\frac{1}{w_{N,\lambda}}\right)x_N$$

- λ — Forgetting factor.
- $w_{N,\lambda}$ — Weighting factor applied to the current data sample.
- x_N — Current data input sample.
- $\bar{x}_{N-1,\lambda}$ — Moving average at the previous sample.
- $\left(1 - \frac{1}{w_{N,\lambda}}\right)\bar{x}_{N-1,\lambda}$ — Effect of the previous data on the average.
- $\bar{x}_{N,\lambda}$ — Moving average at the current sample.

Data	**Weight** $w_{N,\lambda} = \lambda w_{N-1,\lambda} + 1$	**Average** $\bar{x}_{N,\lambda} = \left(1 - \frac{1}{w_{N,\lambda}}\right)\bar{x}_{N-1,\lambda} + \left(\frac{1}{w_{N,\lambda}}\right)x_N$
Frame 1		
2	1. For $N = 1$, this value is 1.	2
3	0.9×1+1 = 1.9	(1–(1/1.9))×2+(1/1.9)×3 = 2.5263
4	0.9×1.9+1 = 2.71	(1–(1/2.71))×2.52+(1/2.71)×4 = 3.0701
5	0.9×2.71+1 = 3.439	(1–(1/3.439))×3.07+(1/3.439)×5 = 3.6313
Frame 2		
6	0.9×3.439+1 = 4.095	(1–(1/4.095))×3.6313+(1/4.095)×6 = 4.2097
7	0.9×4.095+1 = 4.6855	(1–(1/4.6855))×4.2097+(1/4.6855)×7 = 4.8052
8	0.9×4.6855+1 = 5.217	(1–(1/5.217))×4.8052+(1/5.217)×8 = 5.4176
9	0.9×5.217+1 = 5.6953	(1–(1/5.6953))×5.4176+(1/5.6953)×9 = 6.0466
Frame 3		
3	0.9×5.6953+1 = 6.1258	(1–(1/6.1258))×6.0466+(1/6.1258)×3 = 5.5493
4	0.9×6.1258+1 = 6.5132	(1–(1/6.5132))×5.5493+(1/6.5132)×4 = 5.3114

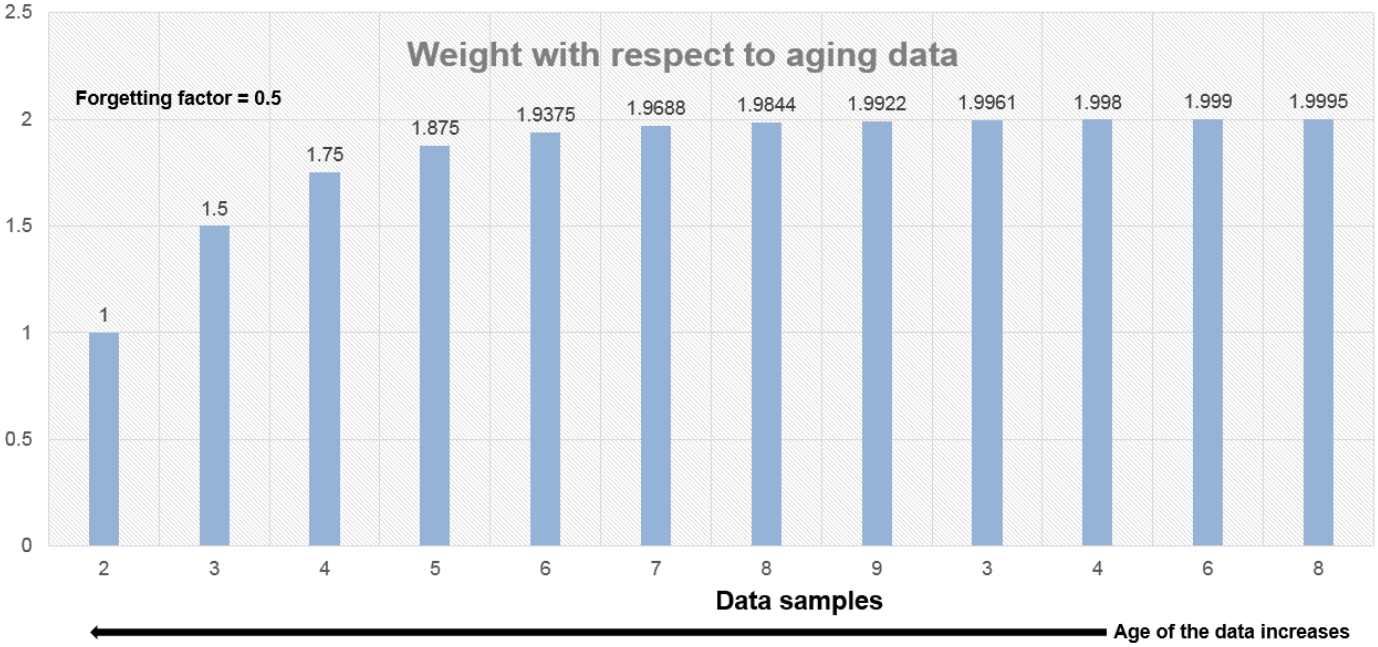
Data	**Weight** $w_{N,\lambda} = \lambda w_{N-1,\lambda} + 1$	**Average** $\bar{x}_{N,\lambda} = \left(1 - \frac{1}{w_{N,\lambda}}\right)\bar{x}_{N-1,\lambda} + \left(\frac{1}{w_{N,\lambda}}\right)x_N$
6	0.9×6.5132+1 = 6.8619	(1–(1/6.8619)) ×5.3114+(1/6.8619)×6 = 5.4117
8	0.9×6.8619+1 = 7.1751	(1–(1/7.1751)) ×5.4117+(1/7.1751)×8 = 5.7724

The moving average algorithm has a state and remembers the data from the previous time step.

For the first sample, when $N = 1$, the algorithm chooses $w_{N,\lambda} = 1$. For the next sample, the weighting factor is updated and the average is computed using the recursive equations.

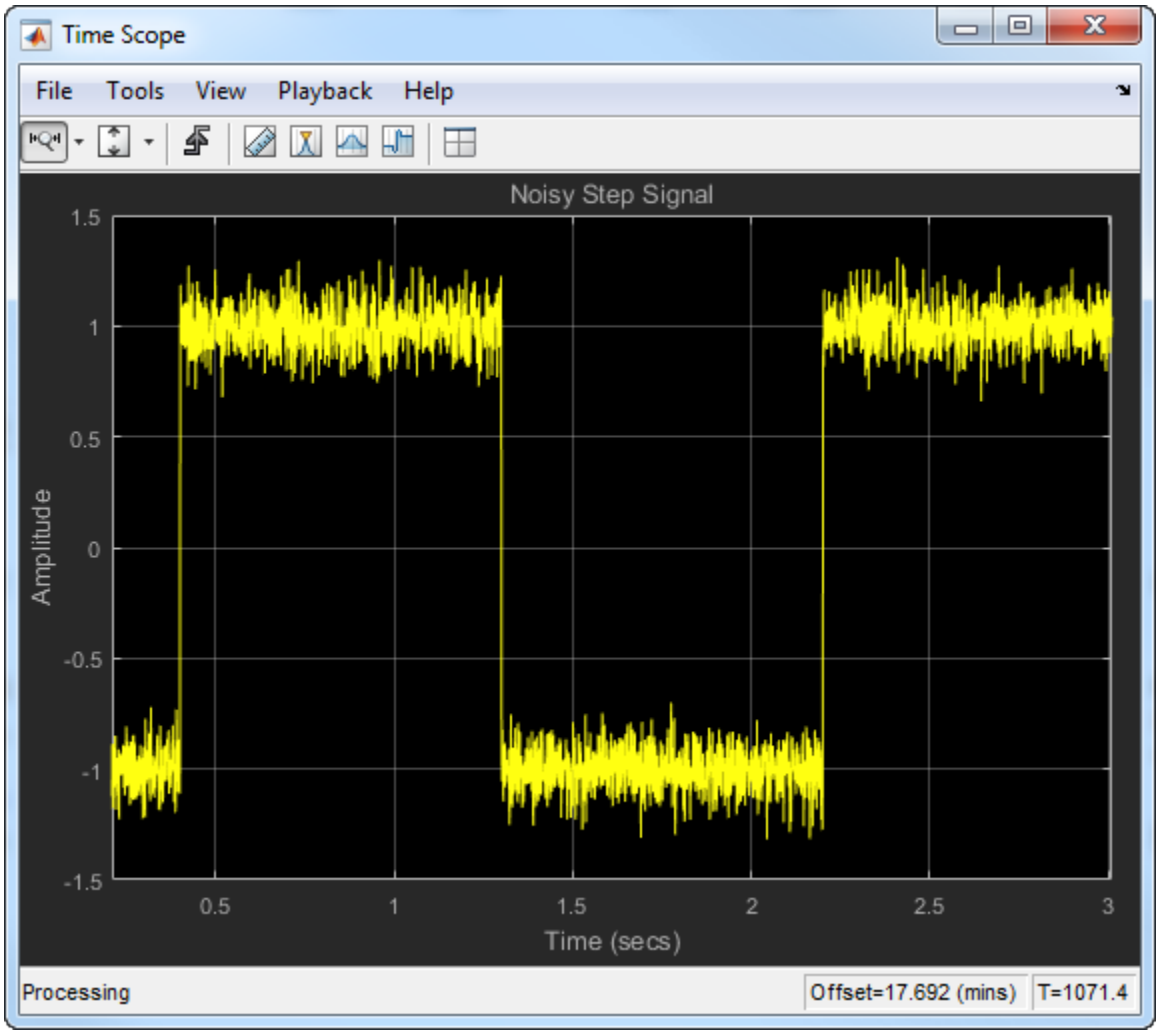
As the age of the data increases, the magnitude of the weighting factor decreases exponentially and never reaches zero. In other words, the recent data has more influence on the current average than the older data.

When the forgetting factor is 0.5, the weights applied to the older data are lower than when the forgetting factor is 0.9.

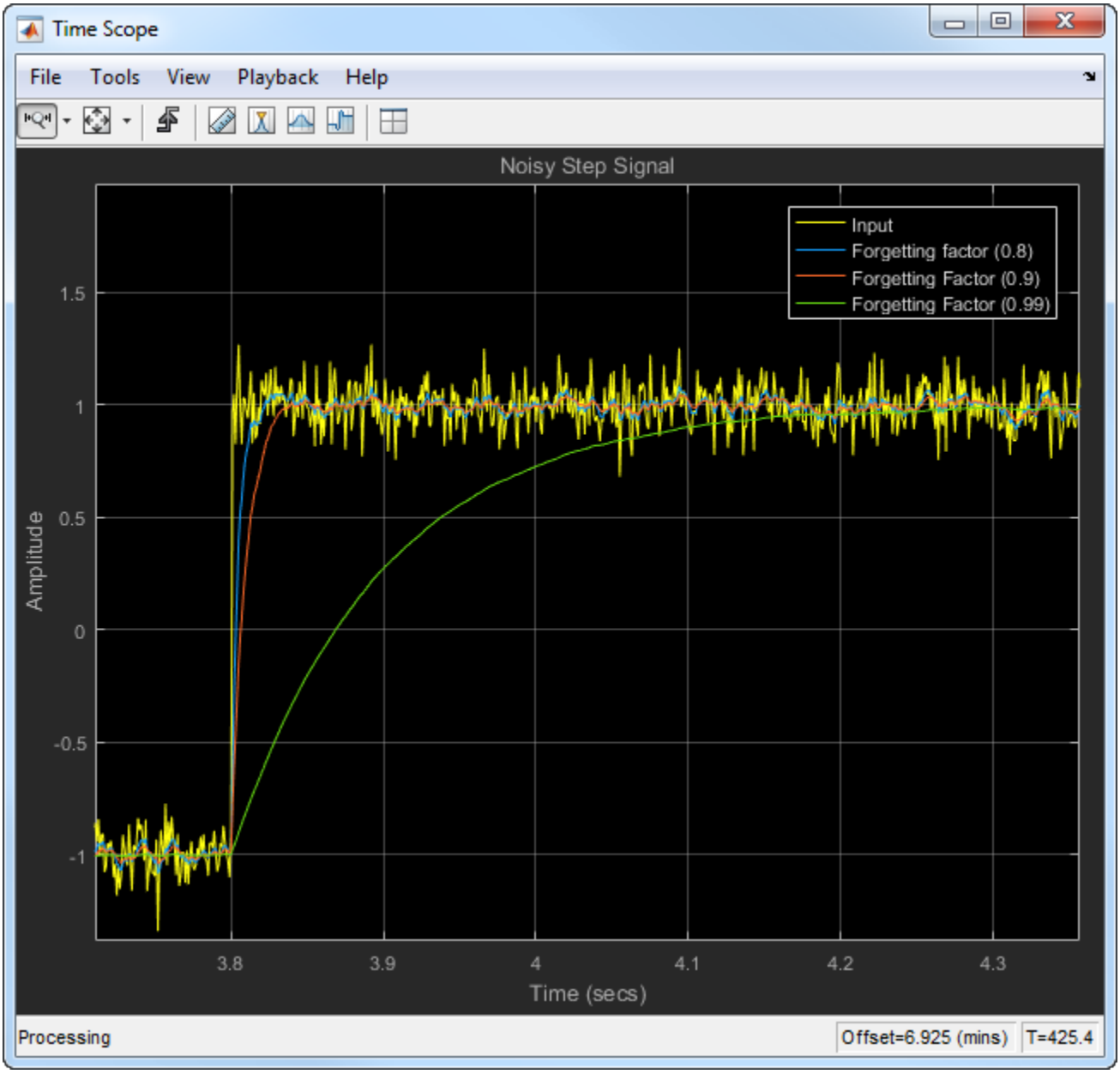


When the forgetting factor is 1, all the data samples are weighed equally. In this case, the exponentially weighted method is the same as the sliding window method with an infinite window length.

When the signal changes rapidly, use a lower forgetting factor. When the forgetting factor is low, the effect of the past data is lesser on the current average. This makes the transient sharper. As an example, consider a rapidly varying noisy step signal.



Compute the moving average of this signal using the exponentially weighted method. Compare the performance of the algorithm with forgetting factors 0.8, 0.9, and 0.99.



When you zoom in on the plot, you can see that the transient in the moving average is sharp when the forgetting factor is low. This makes it more suitable for data that changes rapidly.

For more information on the moving average algorithm, see the **Algorithms** section in the `dsp.MovingAverage` System object or the Moving Average block page.

For more information on other moving statistic algorithms, see the **Algorithms** section in the respective System object and block pages.

References

[1] Bodenham, Dean. “Adaptive Filtering and Change Detection for Streaming Data.” PH.D. Thesis. Imperial College, London, 2012.

See Also

More About

- “What Are Moving Statistics?” on page 26-2
- “Measure Statistics of Streaming Signals” on page 26-14
- “How Is a Moving Average Filter Different from an FIR Filter?” on page 26-17
- “Signal Statistics”
- “Energy Detection in the Time Domain” on page 26-21
- “Remove High-Frequency Noise from Gyroscope Data” on page 26-24
- “Measure Pulse and Transition Characteristics of Streaming Signals” on page 26-26

Measure Statistics of Streaming Signals

In this section...
“Compute Moving Average Using Only MATLAB Functions” on page 26-14
“Compute Moving Average Using System Objects” on page 26-15

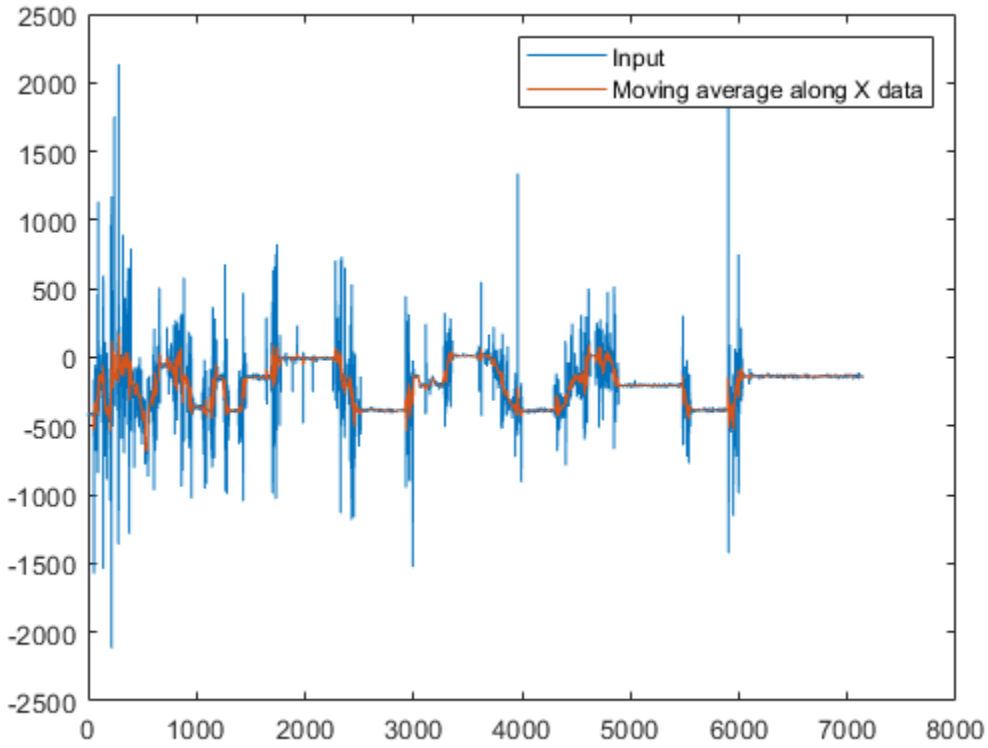
The moving statistics System objects measure statistics of streaming signals in MATLAB. You can also use functions such as `movmean`, `movmedian`, `movstd`, and `movvar` to measure the moving statistics. These functions are more suitable for one-time computations on data that is available in a batch. Unlike System objects, the functions are not designed to handle large streams of data.

Compute Moving Average Using Only MATLAB Functions

This example shows how to compute the moving average of a signal using the `movmean` function.

The `movmean` function computes the 10-point moving average of the noisy data coming from an accelerometer. The three columns in this data represent the linear acceleration of the accelerometer in the *X*-axis, *Y*-axis, and *Z*-axis, respectively. All the data is available in a MAT file. Plot the moving average of the *X*-axis data.

```
winLen = 10;
accel = load('LSM9DS1accelData73.mat');
movAvg = movmean(accel.data,winLen,'Endpoints','fill');
plot([accel.data(:,1),movAvg(:,1)]);
legend('Input','Moving average along X data');
```



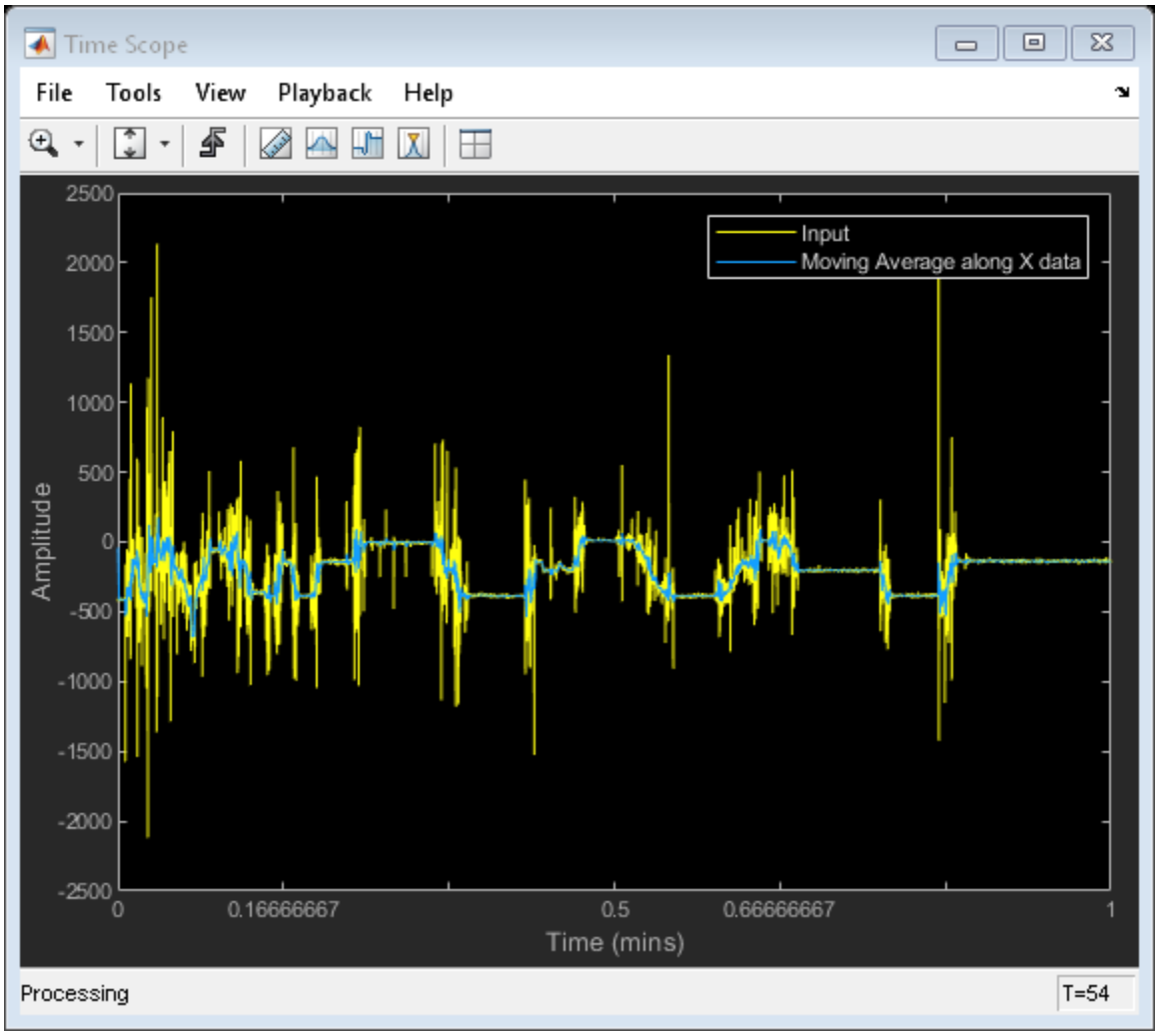
The data is not very large (7140 samples in each column) and is entirely available for processing. The `movmean` function is designed to handle such one-time computations. However, if the data is very large, such as in the order of GB, or if the data is a live stream that needs to be processed in real time, then use System objects. The System objects divide the data into segments called frames and process each frame in an iteration loop seamlessly. This approach is memory efficient, because only one frame of data is processed at any given time. Also, the System objects are optimized to handle states internally.

Compute Moving Average Using System Objects

Create a `dsp.MovingAverage` System object? to compute the 10-point moving average of the streaming signal. Use a `dsp.MatFileReader` System object to read data from the accelerometer MAT file. View the moving average output in the time scope.

The System objects automatically index the data into frames. Choose a frame size of 714 samples. There are 7140 samples or 10 frames of data in each column of the MAT file. Each iteration loop computes the moving average of 1 frame of data.

```
frameSize = 714;
reader = dsp.MatFileReader('SamplesPerFrame',frameSize,...
    'Filename','LSM9DS1accelData73.mat','VariableName','data');
movAvg = dsp.MovingAverage(10);
scope = dsp.TimeScope('NumInputPorts',2,'SampleRate',119,'YLimits',[-2500 2500],...
    'ChannelNames',{'Input','Moving Average along X data'},'TimeSpan',60,'ShowLegend',true);

while ~isDone(reader)
    accel = reader();
    avgData = movAvg(accel);
    scope(accel(:,1),avgData(:,1));
end
```



The processing loop is very simple. The System Objects handle data indexing and states automatically.

See Also

More About

- “What Are Moving Statistics?” on page 26-2
- “Sliding Window Method and Exponential Weighting Method” on page 26-5
- “How Is a Moving Average Filter Different from an FIR Filter?” on page 26-17
- “Signal Statistics”
- “Energy Detection in the Time Domain” on page 26-21
- “Remove High-Frequency Noise from Gyroscope Data” on page 26-24
- “Measure Pulse and Transition Characteristics of Streaming Signals” on page 26-26

How Is a Moving Average Filter Different from an FIR Filter?

The moving average filter is a special case of the regular FIR filter. Both filters have finite impulse responses. The moving average filter uses a sequence of scaled 1s as coefficients, while the FIR filter coefficients are designed based on the filter specifications. They are not usually a sequence of 1s.

The moving average of streaming data is computed with a finite sliding window:

$$movAvg = \frac{x[n] + x[n-1] + ... + x[n-N]}{N+1}$$

$N + 1$ is the length of the filter. This algorithm is a special case of the regular FIR filter with the coefficients vector, $[b_0, b_1, ..., b_N]$.

$$FIROutput = b_0x[n] + b_1x[n-1] + ... + b_Nx[n-N]$$

To compute the output, the regular FIR filter multiplies each data sample with a coefficient from the $[b_0, b_1, ..., b_N]$ vector and adds the result. The moving average filter does not use any multipliers. The algorithm adds all the data samples and multiplies the result with 1 / *filterLength*.

Frequency Response of Moving Average Filter and FIR Filter

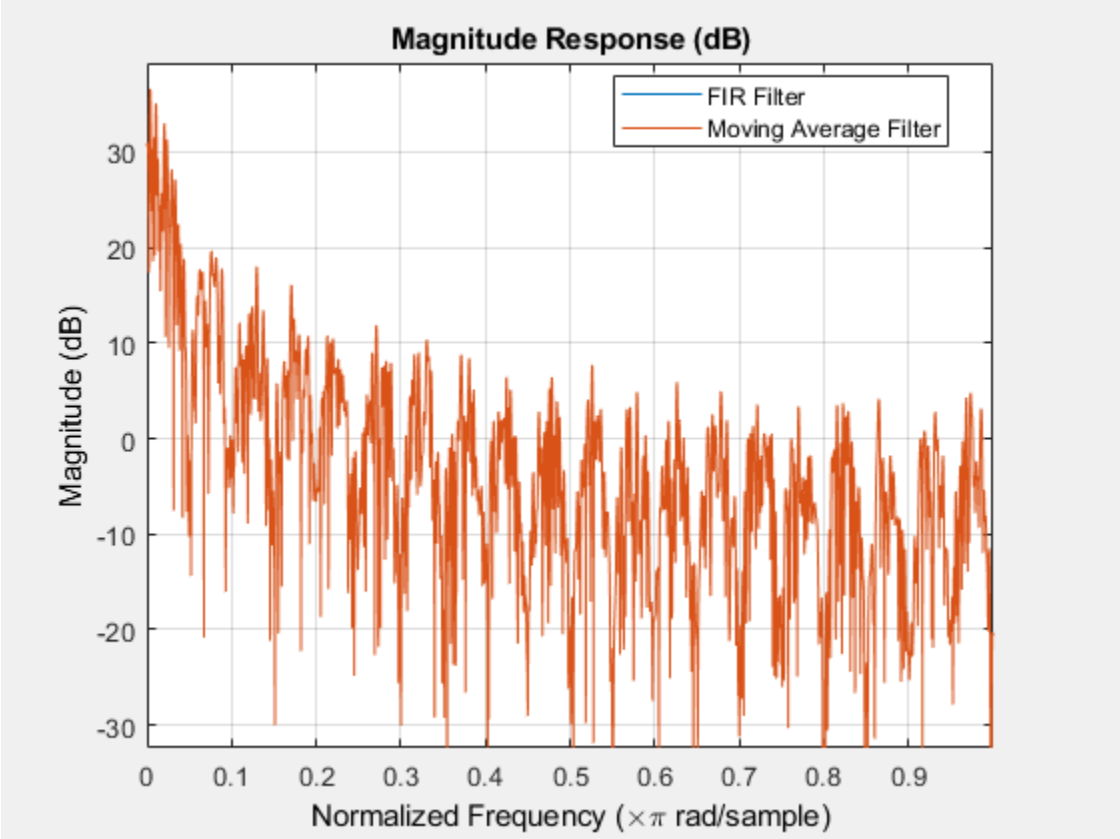
Compare the frequency response of the moving average filter with that of the regular FIR filter. Set the coefficients of the regular FIR filter as a sequence of scaled 1's. The scaling factor is 1/| filterLength|.

Create a `dsp.FIRFilter` System object™ and set its coefficients to 1/40. To compute the moving average, create a `dsp.MovingAverage` System object with a sliding window of length 40. Both filters have the same coefficients. The input is Gaussian white noise with a mean of 0 and a standard deviation of 1.

```
filter = dsp.FIRFilter('Numerator',ones(1,40)/40);
mvgAvg = dsp.MovingAverage(40);
input = randn(1024,1);
filterOutput = filter(input);
mvgAvgOutput = mvgAvg(input);
```

Visualize the frequency response of both filters by using fvtool.

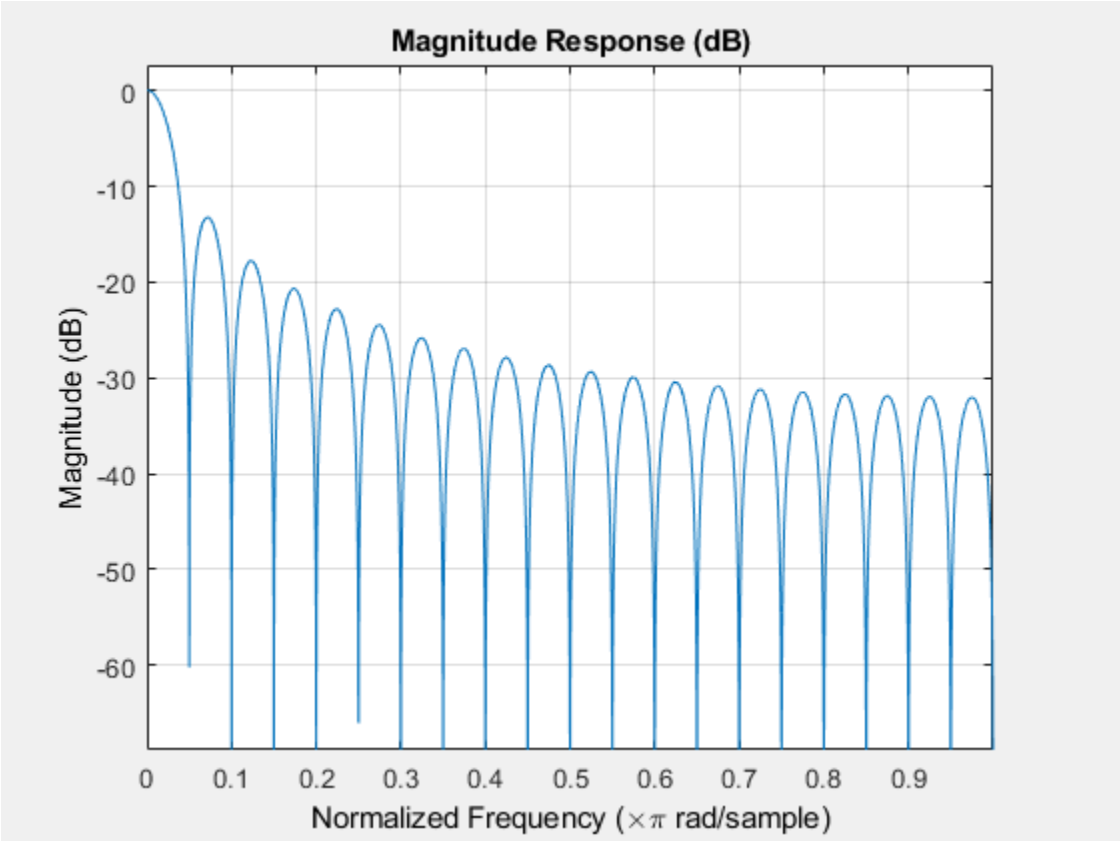
```
hfvt = fvtool(filterOutput,1,mvgAvgOutput,1);
legend(hfvt,'FIR Filter','Moving Average Filter');
```



The frequency responses match exactly, which proves that the moving average filter is a special case of the FIR filter.

For comparison, view the frequency response of the filter without noise.

```
fvtool(filter);
```

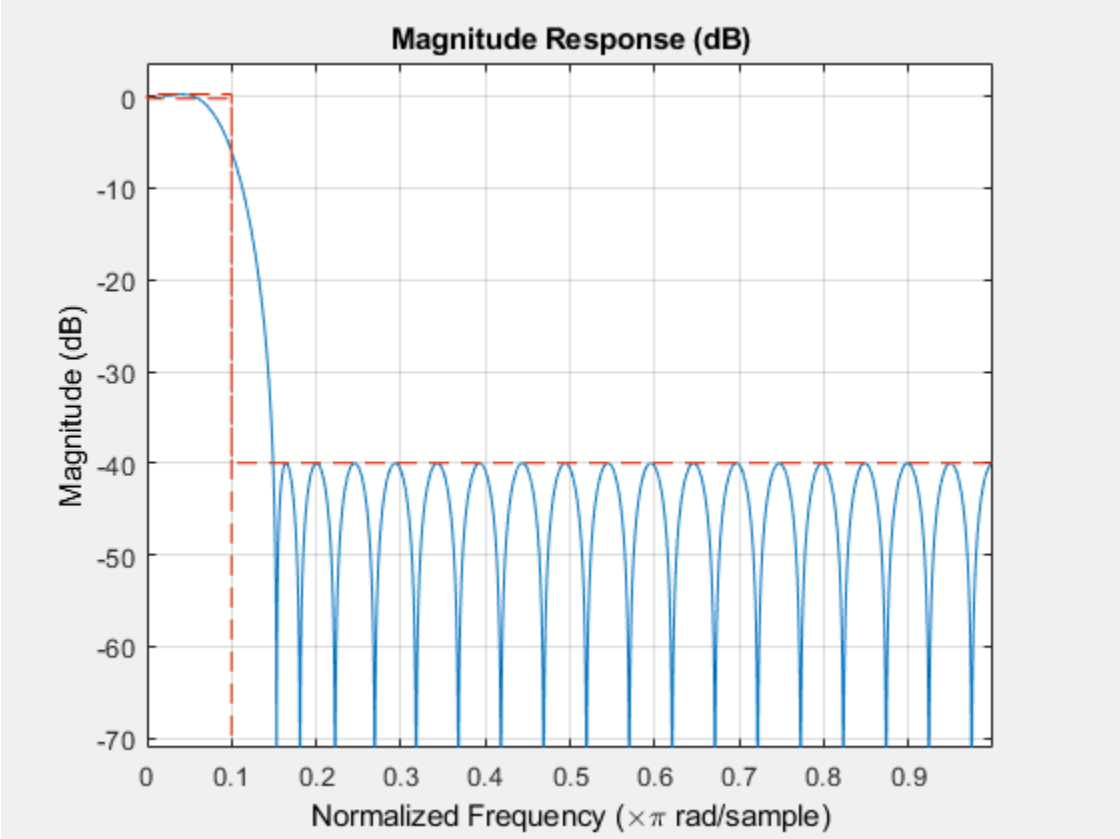


Compare the filter's frequency response to that of the ideal filter. You can see that the main lobe in the passband is not flat and the ripples in the stopband are not constrained. The moving average filter's frequency response does not match the frequency response of the ideal filter.

To realize an ideal FIR filter, change the filter coefficients to a vector that is not a sequence of scaled 1s. The frequency response of the filter changes and tends to move closer to the ideal filter response.

Design the filter coefficients based on predefined filter specifications. For example, design an equiripple FIR filter with a normalized cutoff frequency of 0.1, a passband ripple of 0.5, and a stopband attenuation of 40 dB. Use `fdesign.lowpass` to define the filter specifications and the `design` method to design the filter.

```
FIReq = fdesign.lowpass('N,Fc,Ap,Ast',40,0.1,0.5,40);
filterCoeff = design(FIReq,'equiripple','SystemObject',true);
fvtool(filterCoeff)
```



The filter's response in the passband is almost flat (similar to the ideal response) and the stopband has constrained equiripples.

See Also

More About

- “What Are Moving Statistics?” on page 26-2
- “Measure Statistics of Streaming Signals” on page 26-14
- “Sliding Window Method and Exponential Weighting Method” on page 26-5
- “Signal Statistics”
- “Energy Detection in the Time Domain” on page 26-21
- “Remove High-Frequency Noise from Gyroscope Data” on page 26-24
- “Measure Pulse and Transition Characteristics of Streaming Signals” on page 26-26

Energy Detection in the Time Domain

This example shows how to detect the energy of a discrete-time signal over a finite interval using the RMS value of the signal. By definition, the RMS value over a finite interval $-N \le n \le N$ is given by:

$$RMS = \sqrt{\frac{1}{2N+1}\sum_{n=-N}^{N}|x(n)|^2}$$

The energy of a discrete-time signal over a finite interval $-N \le n \le N$ is given by:

$$E_N = \sum_{n=-N}^{N}|x(n)|^2$$

To determine the signal energy from the RMS value, square the RMS value and multiply the result by the number of samples that are used to compute the RMS value.

$$E_N = RMS^2 \times (2N+1)$$

To compute the RMS value in MATLAB and Simulink, use the moving RMS System object and block, respectively.

Detect Signal Energy

This example shows how to compute the energy of a signal from the signal's RMS value and compares the energy value with a specified threshold. Detect the event when the signal energy is above the threshold.

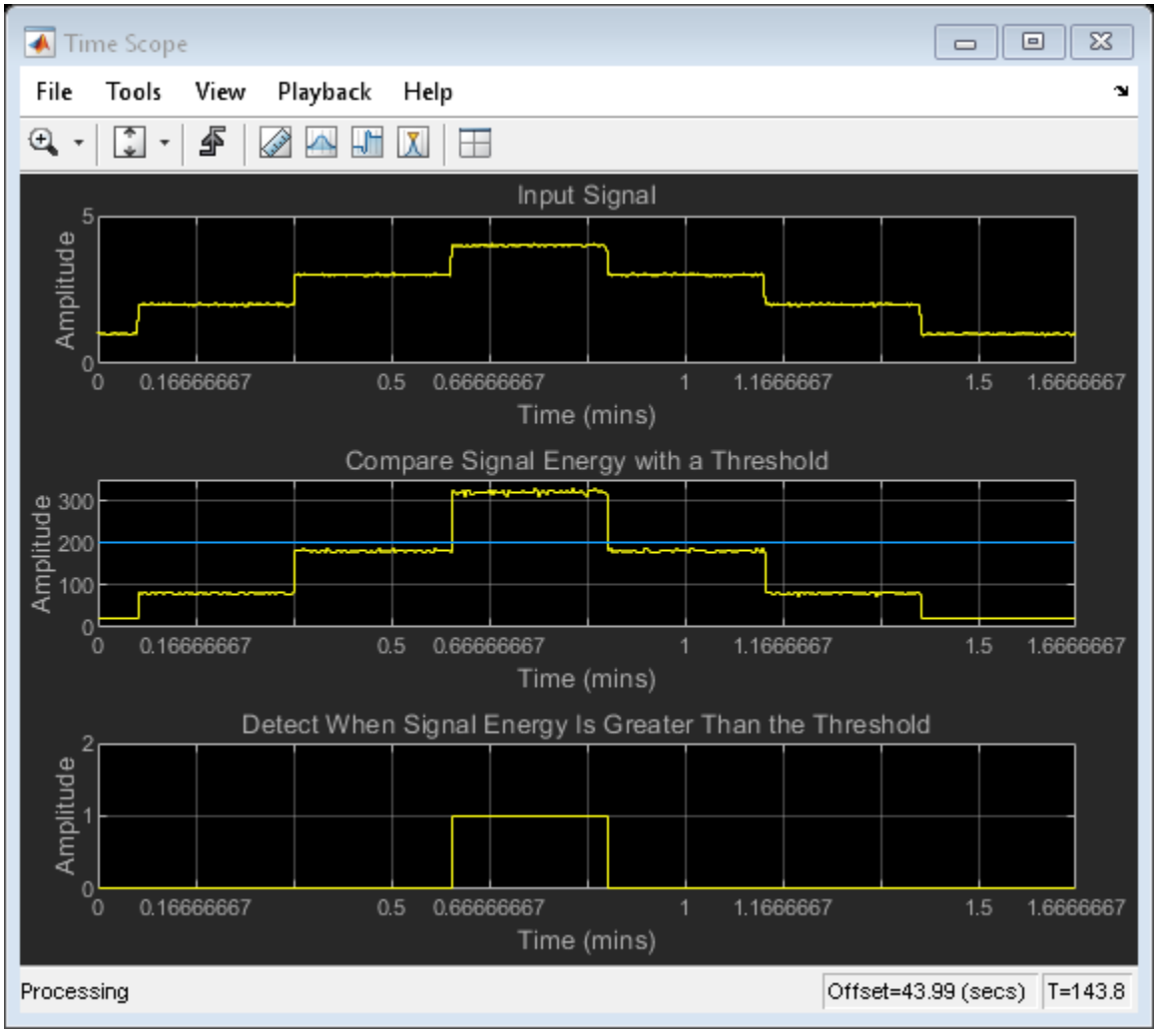
Create a `dsp.MovingRMS` System object? to compute the moving RMS of the signal. Set this object to use the sliding window method with a window length of 20. Create a `dsp.TimeScope` object to view the output.

```
FrameLength = 20;
Fs = 100;
movrmsWin = dsp.MovingRMS(20);
scope  = dsp.TimeScope('SampleRate',Fs,...
    'TimeSpanOverrunAction','Scroll',...
    'TimeSpan',100,...
    'ShowGrid',true,...
    'YLimits',[-1.0 350],'LayoutDimensions',[3 1],'NumInputPorts',3);

scope.ActiveDisplay = 1;
scope.YLimits = [0 5];
scope.Title = 'Input Signal';

scope.ActiveDisplay = 2;
scope.Title = 'Compare Signal Energy with a Threshold';

scope.ActiveDisplay = 3;
scope.YLimits = [0 2];
scope.PlotType = 'Stairs';
scope.Title = 'Detect When Signal Energy Is Greater Than the Threshold';
```

Create the input signal. The signal is a noisy staircase with a frame length of 20. The threshold value is 200. Compute the energy of the signal by squaring the RMS value and multiplying the result with

the window length. Compare the signal energy with the threshold value. Detect the event, and when the signal energy crosses the threshold, mark it as 1.

```
count = 1;
Vect = [1/8 1/2 1 2 3 4 3 2 1];
threshold = 200;
for index = 1:length(Vect)
    V = Vect(index);
    for i = 1:80
        x = V + 0.1 * randn(FrameLength,1);
        y1 = movrmsWin(x);
        y1ener = (y1(end)^2)*20;
        event = (y1ener>threshold);
        scope(y1,[y1ener,threshold],event);
    end
end
```



You can customize the energy mask into a pattern that varies by more than a scalar threshold. You can also record the time for which the signal energy stays above or below the threshold.

See Also

More About

- “What Are Moving Statistics?” on page 26-2
- “Measure Statistics of Streaming Signals” on page 26-14
- “Sliding Window Method and Exponential Weighting Method” on page 26-5
- “Signal Statistics”
- “How Is a Moving Average Filter Different from an FIR Filter?” on page 26-17
- “Remove High-Frequency Noise from Gyroscope Data” on page 26-24
- “Measure Pulse and Transition Characteristics of Streaming Signals” on page 26-26

Remove High-Frequency Noise from Gyroscope Data

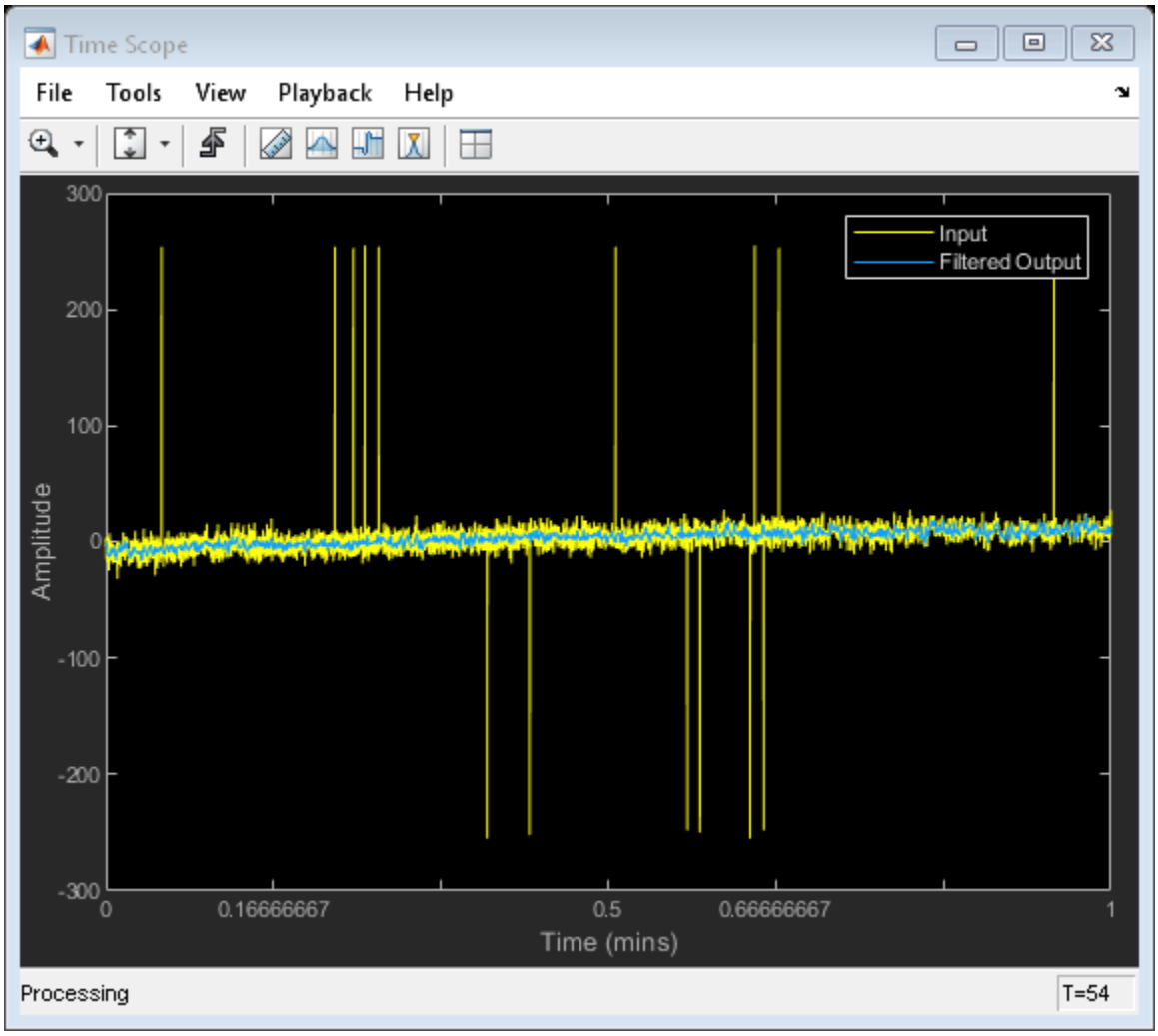
This example shows how to remove the high-frequency outliers from a streaming signal using the `dsp.MedianFilter` System object?.

Use the `dsp.MatFileReader` System object to read the gyroscope MAT file. The gyroscope MAT file contains 3 columns of data, with each column containing 7140 samples. The three columns represent the *X*-axis, *Y*-axis, and *Z*-axis data from the gyroscope motion sensor. Choose a frame size of 714 samples so that each column of the data contains 10 frames. The `dsp.MedianFilter` System object uses a window length of 10. Create a `dsp.TimeScope` object to view the filtered output.

```
reader = dsp.MatFileReader('SamplesPerFrame',714,'Filename','LSM9DS1gyroData73.mat',...
    'VariableName','data');
medFilt = dsp.MedianFilter(10);
scope = dsp.TimeScope('NumInputPorts',1,'SampleRate',119,'YLimits',[-300 300],...
    'ChannelNames',{'Input','Filtered Output'},'TimeSpan',60,'ShowLegend',true);
```

Filter the gyroscope data using the `dsp.MedianFilter` System object. View the filtered *Z*-axis data in the time scope.

```
for i = 1:10
    gyroData = reader();
    filteredData = medFilt(gyroData);
    scope([gyroData(:,3),filteredData(:,3)]);
end
```



The original data contains several outliers. Zoom in on the data to confirm that the median filter removes all the outliers.

See Also

More About

- “What Are Moving Statistics?” on page 26-2
- “Measure Statistics of Streaming Signals” on page 26-14
- “Sliding Window Method and Exponential Weighting Method” on page 26-5
- “Signal Statistics”
- “How Is a Moving Average Filter Different from an FIR Filter?” on page 26-17
- “Energy Detection in the Time Domain” on page 26-21
- “Measure Pulse and Transition Characteristics of Streaming Signals” on page 26-26

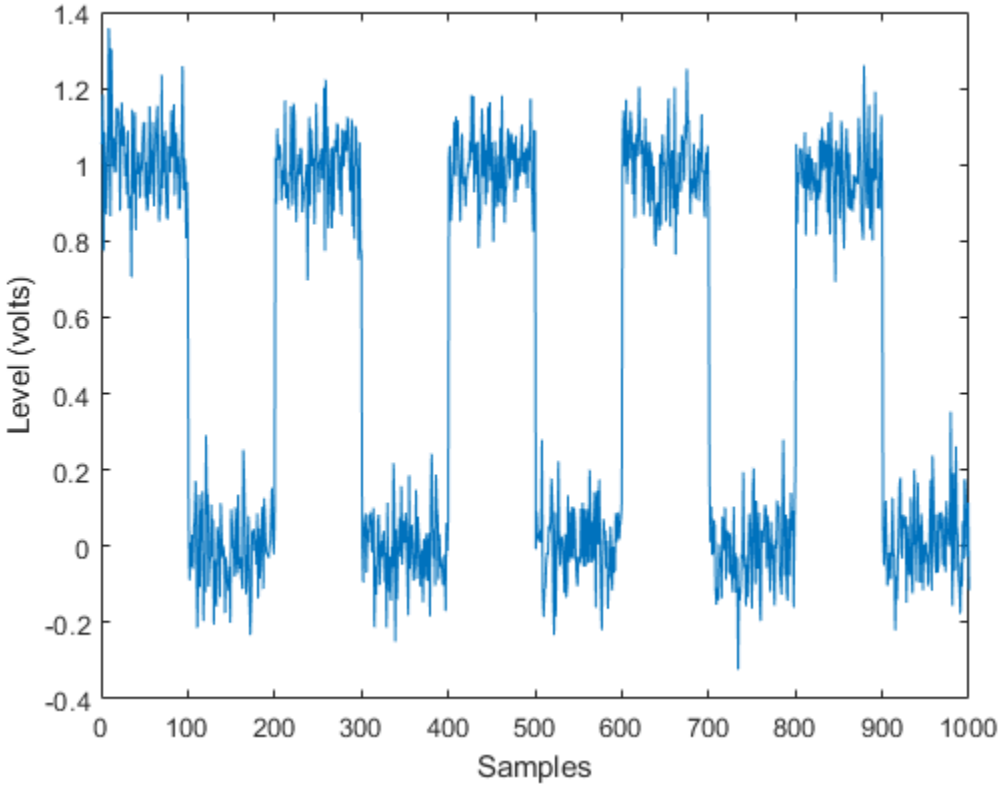
Measure Pulse and Transition Characteristics of Streaming Signals

This example measures the pulse and transition metrics of a noisy rectangular pulse. Pulse metrics include rise time, fall time, pulse width, and pulse period. Transition metrics include middle-cross events, overshoot, and undershoot of the posttransition aberration regions of the noisy rectangular pulse.

Generate a Rectangular Pulse

Generate a noisy rectangular pulse. The noise is a white Gaussian noise with zero mean and a standard deviation of 0.1. Store the data in `rectData`.

```
t = 0:.01:9.99;   % time vector
w = 1; % pulse width
d = w/2:w*2:10; % delay vector
y2 = pulstran(t,d,'rectpuls',w);
rectData = y2'+0.1*randn(1000,1); % rectangular pulse with noise
plot(rectData);
xlabel('Samples');
ylabel('Level (volts)');
```



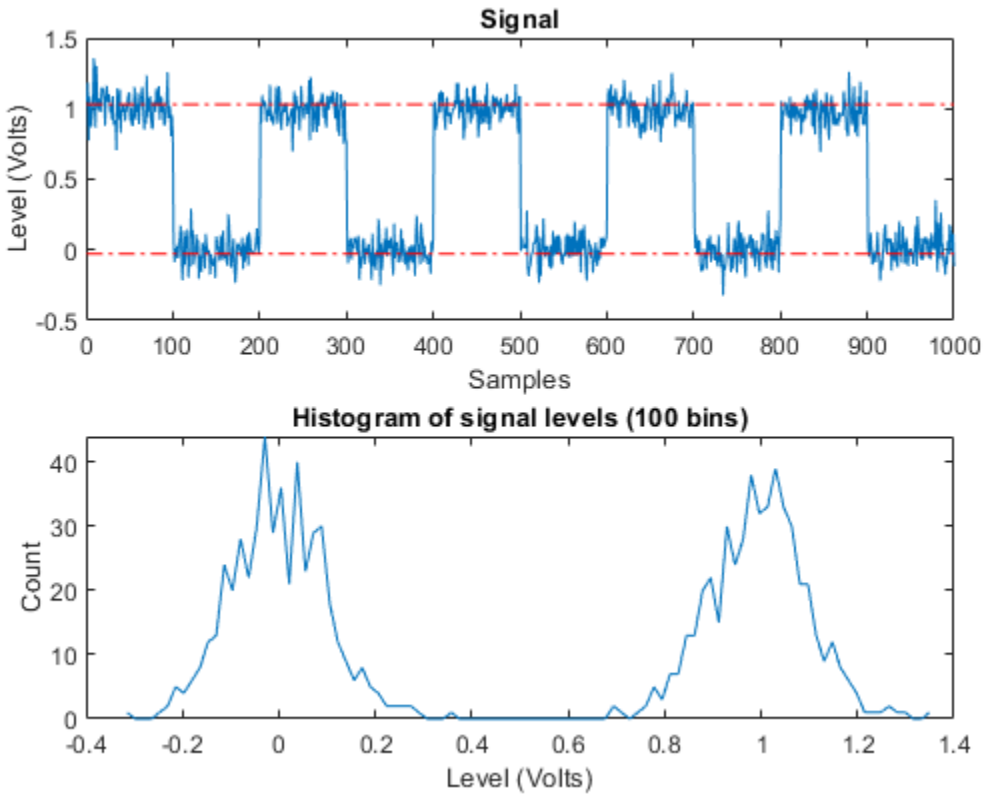
Measure State Levels

The `dsp.StateLevels` System object uses the histogram method to estimate the state levels of a bilevel waveform. The histogram method involves the following steps:

1. Determine the maximum and minimum amplitudes of the data.
2. For the specified number of histogram bins, determine the bin width, which is the ratio of the amplitude range to the number of bins.
3. Sort the data values into the histogram bins.
4. Identify the lowest indexed histogram bin and the highest indexed histogram bin with nonzero counts.
5. Divide the histgram into two subhistograms.
6. Compute the state levels by determining the mode or mean of the upper and lower histograms.

Plot the state levels of the rectangular pulse.

```
sLevel = dsp.StateLevels;
levels = sLevel(rectData);
figure(1);
plot(sLevel);
```

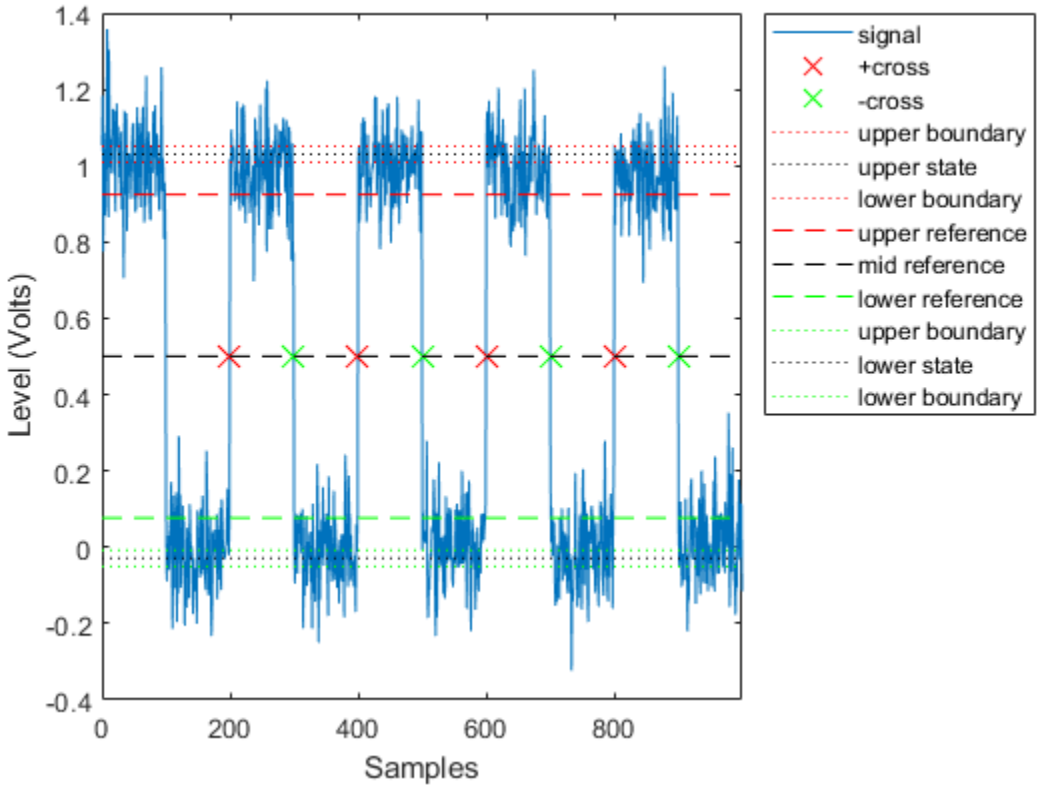


Compute Pulse Metrics

Using the `dsp.PulseMetrics` System object, you can compute metrics such as the rise time, fall time, pulse width, and pulse period of the rectangular pulse. Plot the pulse with the state levels and reference levels.

```
pMetrics = dsp.PulseMetrics('StateLevels',levels,'CycleOutputPort',true);
[pulse,cycle] = pMetrics(rectData);
```

```
plot(pMetrics);
xlabel('Samples');
```



Rise Time is the duration between the instants where the rising transition of each pulse crosses from the lower to the upper reference levels. View the rise time of each pulse.

```
pulse.RiseTime
```

```
ans = 4×1

    0.8864
    0.8853
    1.6912
    1.7727
```

Fall time is the duration between the instants where the falling transition of each pulse crosses from the upper to the lower reference levels. View the fall time of each pulse.

```
pulse.FallTime
```

```
ans = 4×1

    2.4263
    0.7740
    1.7339
    0.9445
```

Width is the duration between the mid-reference level crossings of the first and second transitions of each pulse. View the width of each pulse.

```
pulse.Width
```

```
ans = 4×1

   99.8938
  100.0856
  100.1578
  100.1495
```

Period is the duration between the first transition of the current pulse and the first transition of the next pulse. View the period of each pulse.

```
cycle.Period
```

```
ans = 3×1

  199.9917
  199.9622
  199.9291
```

Polarity

The `Polarity` property of the `pMetrics` object is set to `'Positive'`. The object therefore computes the pulse metrics starting from the first positive transition.

Running Metrics

If the `RunningMetrics` property is set to `true`, the object treats the data as a continuous stream of running data. If there is an incomplete pulse at the end, the object returns the metrics of the last pulse in the next process step, once it has enough data to complete the pulse. If the `RunningMetrics` property is set to false, the object treats each call to process independently. If the last pulse is incomplete, the object computes whatever metrics it can return with the available data. For example, if the pulse is half complete, the object can return the rise time of the last pulse, but not the pulse period.

Given that the polarity is positive and the running metrics are set to false, the rectangular pulse has:

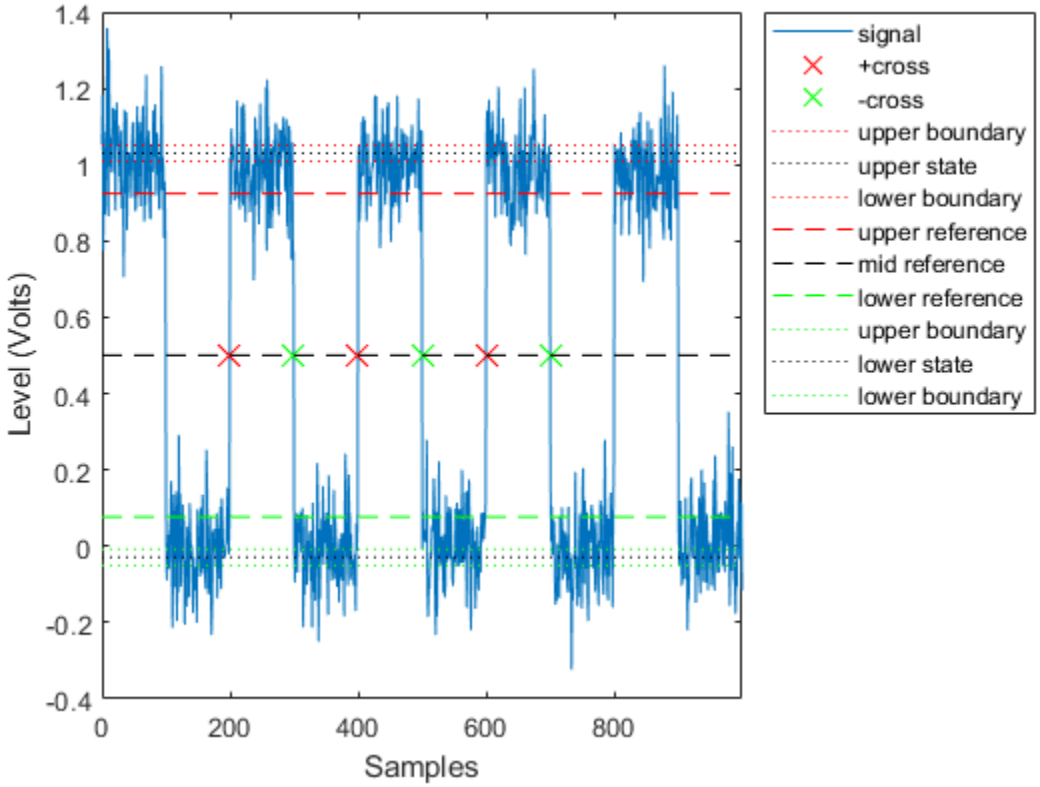
- The first positive transition at 200 seconds
- Three complete pulses and two incomplete pulses (first and the last)
- Four positive transitions and four negative transitions

Depending on the metric, the number of elements in the metric vector is equal to either the number of transitions or the number of complete pulses. The rise time vector has four elements, which matches the number of transitions. The cycle period has three elements, which matches the number of complete pulses.

Set the `RunningMetrics` property to `true`.

```
release(pMetrics);
pMetrics.RunningMetrics = true;
[pulse,cycle] = pMetrics(rectData);
```

```
plot(pMetrics);
xlabel('Samples');
```



The pMetrics object has three positive transitions and three negative transitions. The object waits to complete the last pulse before it returns the metrics for the last pulse.

Divide the input data into two frames with 500 samples in each frame. Compute the pulse metrics of the data in running mode. The number of iteration loops correspond to the number of data frames processed.

```
release(pMetrics);
framesize = 500;
for i = 1:2
    data = rectData((((i-1)*framesize)+1):i*framesize);
    [pulse,cycle] = pMetrics(data);
    pulse.RiseTime
end
```

```
ans = 0.8864

ans = 2×1

    0.8853
    1.6912
```

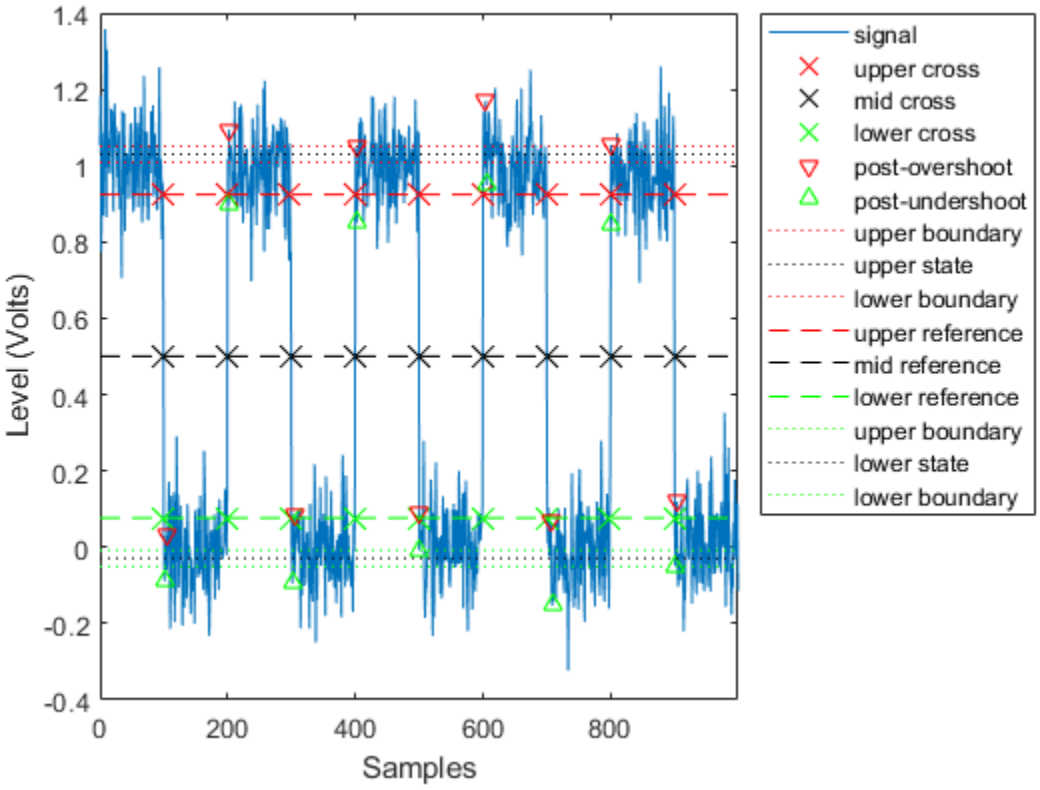
The first frame contains one complete pulse and 2 incomplete pulses. The rise time value displayed in the first iteration step corresponds to the rising transition in this complete pulse. The rise time of the

last complete pulse is not displayed in this iteration step. The algorithm waits to complete the pulse data. The second data frame contains the remaining part of this pulse and another complete pulse. The rise time vector in the second iteration step has two elements - first value corresponds to the rising transition of the incomplete pulse in the previous step, and the second value corresponds to the rising transition of the complete pulse in the current step.

Compute Transition Metrics

The transition metrics correspond to the metrics of the first and second transitions. Using the `dsp.TransitionMetrics` System object, you can determine the middlecross events, and compute the post-overshoot and post-undershoot of the rectangular pulse. To measure the postshoot metrics, set the `PostshootOutputPort` property of `dsp.TransitionMetrics` to true.

```
tMetrics = dsp.TransitionMetrics('StateLevels',levels,'PostshootOutputPort',true);
[transition,postshoot] = tMetrics(rectData);
plot(tMetrics);
xlabel('Samples');
```



Middle-cross events are instants in time where the pulse transitions cross the middle reference level. View the middle-cross events of the pulse.

```
transition.MiddleCross
```

```
ans = 9×1

   99.4345
  199.4582
```

```
  299.3520
  399.4499
  499.5355
  599.4121
  699.5699
  799.3412
  899.4907
```

Overshoots and *undershoots* are expressed as a percentage of the difference between state levels. Overshoots and undershoots that occur after the posttransition aberration region are called post-overshoots and post-undershoots. The overshoot value of the rectangular pulse is the maximum of the overshoot values of all the transitions. View the post-overshoots of the pulse.

```
postshoot.Overshoot
```

```
ans = 9×1

    5.6062
    6.1268
   10.8393
    1.8311
   11.2240
   13.2285
    9.2560
    2.2735
   14.0357
```

The undershoot value of the rectangular pulse is the minimum of the undershoot values of all the transitions.

```
postshoot.Undershoot
```

```
ans = 9×1

    5.6448
   12.5596
    6.2156
   16.8403
   -1.9859
    7.6490
   11.7320
   17.3856
    2.0221
```

See Also

Objects
`dsp.PulseMetrics` | `dsp.StateLevels` | `dsp.TransitionMetrics`

Functions
`falltime` | `overshoot` | `pulseperiod` | `pulsewidth` | `risetime` | `statelevels` | `undershoot`

More About

- “Measurement of Pulse and Transition Characteristics” (Signal Processing Toolbox)
- “What Are Moving Statistics?” on page 26-2

Linear Algebra and Least Squares

In this section...
“Linear Algebra Blocks” on page 26-34
“Linear System Solvers” on page 26-34
“Matrix Factorizations” on page 26-35
“Matrix Inverses” on page 26-36

Linear Algebra Blocks

The Matrices and Linear Algebra library provides three large sublibraries containing blocks for linear algebra; Linear System Solvers, Matrix Factorizations, and Matrix Inverses. A fourth library, Matrix Operations, provides other essential blocks for working with matrices.

Linear System Solvers

The Linear System Solvers library provides the following blocks for solving the system of linear equations A*X* = B:

- Autocorrelation LPC
- Cholesky Solver
- Forward Substitution
- LDL Solver
- Levinson-Durbin
- LU Solver
- QR Solver
- SVD Solver

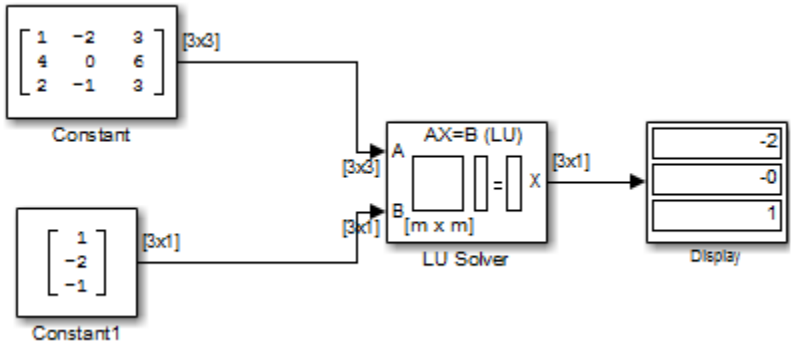
Some of the blocks offer particular strengths for certain classes of problems. For example, the Cholesky Solver block is adapted for a square Hermitian positive definite matrix A, whereas the Backward Substitution block is suited for an upper triangular matrix A.

Solve AX=B Using the LU Solver Block

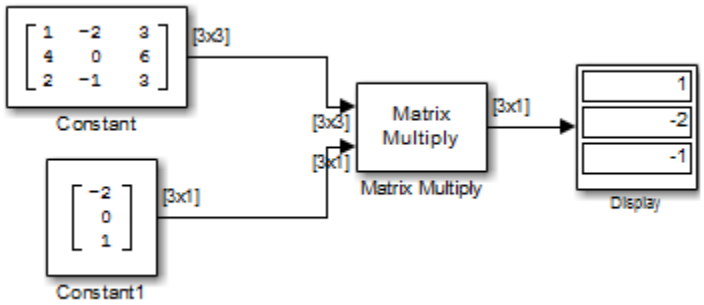
In the following ex_lusolver_tut model, the LU Solver block solves the equation A*x* = b, where

$$\mathbf{A} = \begin{bmatrix} 1 & -2 & 3 \\ 4 & 0 & 6 \\ 2 & -1 & 3 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 1 \\ -2 \\ -1 \end{bmatrix}$$

and finds *x* to be the vector `[-2 0 1]'`.



You can verify the solution by using the Matrix Multiply block to perform the multiplication A*x*, as shown in the following ex_matrixmultiply_tut1 model.



Matrix Factorizations

The Matrix Factorizations library provides the following blocks for factoring various kinds of matrices:

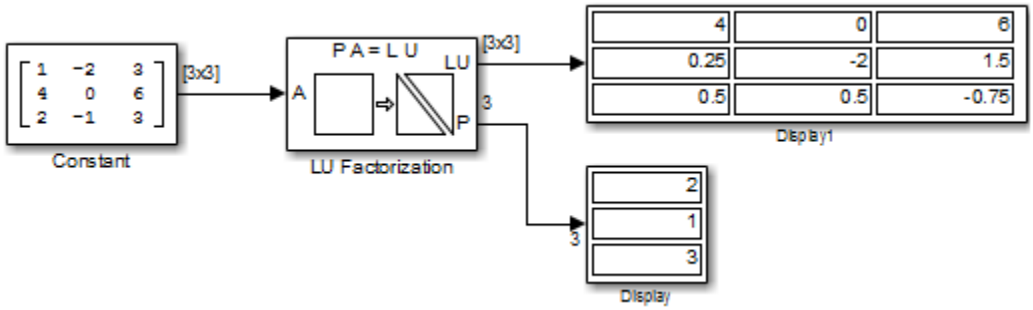
- Cholesky Factorization
- LDL Factorization
- LU Factorization
- QR Factorization
- Singular Value Decomposition

Some of the blocks offer particular strengths for certain classes of problems. For example, the Cholesky Factorization block is suited to factoring a Hermitian positive definite matrix into triangular components, whereas the QR Factorization is suited to factoring a rectangular matrix into unitary and upper triangular components.

Factor a Matrix into Upper and Lower Submatrices Using the LU Factorization Block

In the following ex_lufactorization_tut model, the LU Factorization block factors a matrix A_p into upper and lower triangular submatrices U and L, where A_p is row equivalent to input matrix A, where

$$A = \begin{bmatrix} 1 & -2 & 3 \\ 4 & 0 & 6 \\ 2 & -1 & 3 \end{bmatrix}$$



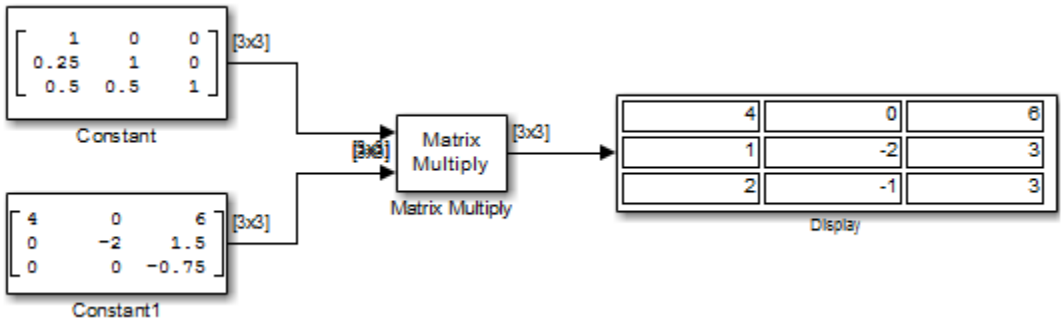
The lower output of the LU Factorization, P, is the permutation index vector, which indicates that the factored matrix A_p is generated from A by interchanging the first and second rows.

$$\mathbf{A}_p = \begin{bmatrix} 4 & 0 & 6 \\ 1 & -2 & 3 \\ 2 & -1 & 3 \end{bmatrix}$$

The upper output of the LU Factorization, LU, is a composite matrix containing the two submatrix factors, U and L, whose product LU is equal to A_p.

$$\mathbf{U} = \begin{bmatrix} 4 & 0 & 6 \\ 0 & -2 & 1.5 \\ 0 & 0 & -0.75 \end{bmatrix} \quad \mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ 0.25 & 1 & 0 \\ 0.5 & 0.5 & 1 \end{bmatrix}$$

You can check that LU = A_p with the Matrix Multiply block, as shown in the following ex_matrixmultiply_tut2 model.



Matrix Inverses

The Matrix Inverses library provides the following blocks for inverting various kinds of matrices:

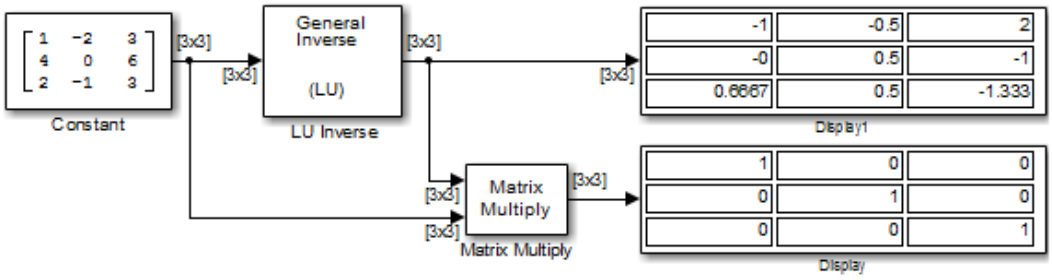
- Cholesky Inverse
- LDL Inverse
- LU Inverse
- Pseudoinverse

Find the Inverse of a Matrix Using the LU Inverse Block

In the following ex_luinverse_tut model, the LU Inverse block computes the inverse of input matrix A, where

$$\mathbf{A} = \begin{bmatrix} 1 & -2 & 3 \\ 4 & 0 & 6 \\ 2 & -1 & 3 \end{bmatrix}$$

and then forms the product A^{-1}A, which yields the identity matrix of order 3, as expected.



As shown above, the computed inverse is

$$\mathbf{A}^{-1} = \begin{bmatrix} -1 & -0.5 & 2 \\ 0 & 0.5 & -1 \\ 0.6667 & 0.5 & -1.333 \end{bmatrix}$$

Bibliography

References — Advanced Filters

[1] Antoniou, A., *Digital Filters: Analysis, Design, and Applications*, Second Edition, McGraw-Hill, Inc., 1993.

[2] Chirlian, P.M., *Signals and Filters*, Van Nostrand Reinhold, 1994.

[3] Fliege, N.J., *Multirate Digital Signal Processing*, John Wiley and Sons, 1994.

[4] Jackson, L., *Digital Filtering and Signal Processing with MATLAB Exercises*, Third edition, Springer, 1995.

[5] Lapsley, P., J. Bier, A. Sholam, and E.A. Lee, *DSP Processor Fundamentals: Architectures and Features*, IEEE Press, 1997.

[6] McClellan, J.H., C.S. Burrus, A.V. Oppenheim, T.W. Parks, R.W. Schafer, and H.W. Schuessler, *Computer-Based Exercises for Signal Processing Using MATLAB 5*, Prentice-Hall, 1998.

[7] Mayer-Baese, U., *Digital Signal Processing with Field Programmable Gate Arrays*, Springer, 2001, refer to the BiQuad block diagram on pp. 126 and the IIR Butterworth example on pp. 140.

[8] Moler, C., “Floating points: IEEE Standard unifies arithmetic model.” *Cleve's Corner*, The MathWorks, Inc., 1996. *See* https://www.mathworks.com/company/newsletter/pdf/Fall96Cleve.pdf.

[9] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989.

[10] Shajaan, M., and J. Sorensen, “Time-Area Efficient Multiplier-Free Recursive Filter Architectures for FPGA Implementation,” IEEE International Conference on Acoustics, Speech, and Signal Processing, 1996, pp. 3269-3272.

References — Frequency Transformations

[1] Constantinides, A.G., “Spectral Transformations for Digital Filters,” *IEEE Proceedings*, Vol. 117, No. 8, pp. 1585-1590, August 1970.

[2] Nowrouzian, B., and A.G. Constantinides, “Prototype Reference Transfer Function Parameters in the Discrete-Time Frequency Transformations,” *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, Vol. 2, pp. 1078-1082, August 1990.

[3] Feyh, G., J.C. Franchitti, and C.T. Mullis, “Allpass Filter Interpolation and Frequency Transformation Problem,“ *Proceedings 20th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, California, pp. 164-168, November 1986.

[4] Krukowski, A., G.D. Cain, and I. Kale, “Custom Designed High-Order Frequency Transformations for IIR Filters,” *38th Midwest Symposium on Circuits and Systems (MWSCAS'95)*, Rio de Janeiro, Brazil, August 1995.

Audio I/O User Guide

Run Audio I/O Features Outside MATLAB and Simulink

You can deploy these audio input and output features outside the MATLAB and Simulink environments:

System Objects

- `audioDeviceWriter`
- `dsp.AudioFileReader`
- `dsp.AudioFileWriter`

Blocks

- Audio Device Writer
- From Multimedia File
- To Multimedia File

The generated code for the audio I/O features relies on prebuilt dynamic library files included with MATLAB. You must account for these extra files when you run audio I/O features outside the MATLAB and Simulink environments. To run a standalone executable generated from a model or code containing the audio I/O features, set your system environment using commands specific to your platform.

Platform	Command

Mac	`setenv DYLD_LIBRARY_PATH "$ {DYLD_LIBRARY_PATH}:$MATLABROOT/bin/ maci64" (csh/tcsh)`
	
	`export DYLD_LIBRARY_PATH= $LD_LIBRARY_PATH:$MATLABROOT/bin/ maci64 (Bash)`

Linux	`setenv LD_LIBRARY_PATH $ {LD_LIBRARY_PATH}:$MATLABROOT/bin/ glnxa64 (csh/tcsh)`
	
	`export LD_LIBRARY_PATH= $LD_LIBRARY_PATH:$MATLABROOT/bin/ glnxa64 (Bash)`

Windows	`set PATH=%PATH%;%MATLABROOT%\bin\win64`

The path in these commands is valid only on systems that have MATLAB installed. If you run the standalone app on a machine with only MCR, and no MATLAB installed, replace `$MATLABROOT/bin/...` with the path to the MCR.

To run the code generated from the above System objects and blocks on a machine does not have MCR or MATLAB installed, use the `packNGo` function. The `packNGo` function packages all relevant files in a compressed zip file so that you can relocate, unpack, and rebuild your project in another development environment with no MATLAB installed.

You can use the `packNGo` function at the command line or the **Package** option in the MATLAB Coder app. The files are packaged in a compressed file that you can relocate and unpack using a

standard zip utility. For more details on how to pack the code generated from MATLAB code, see “Package Code for Other Development Environments” (MATLAB Coder). For more details on how to pack the code generated from Simulink blocks, see the `packNGo` function.

See Also

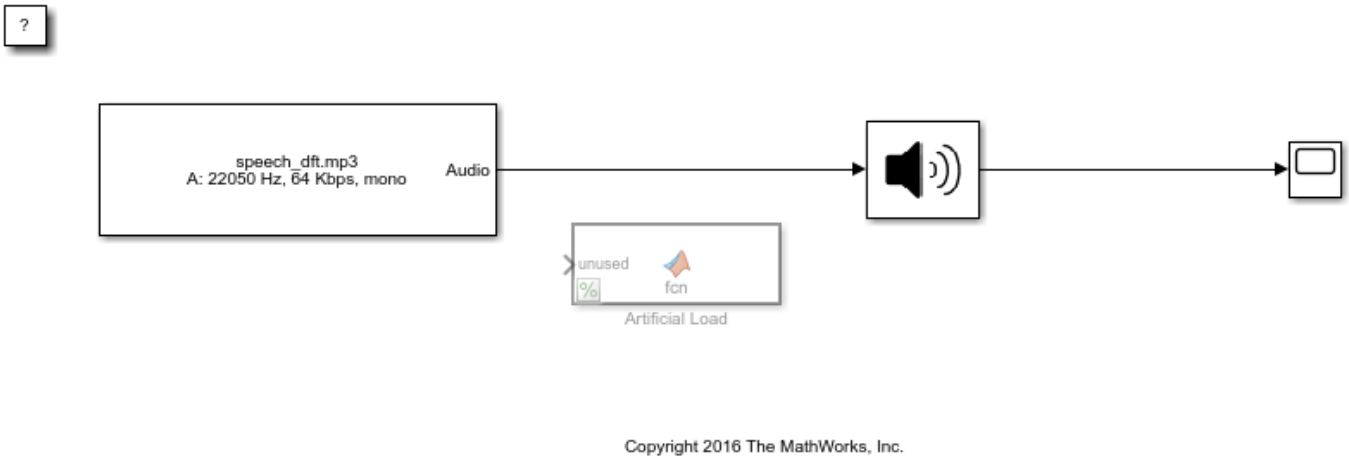
More About

- “Understanding C Code Generation in DSP System Toolbox” on page 18-8
- “MATLAB Programming for Code Generation” (MATLAB Coder)

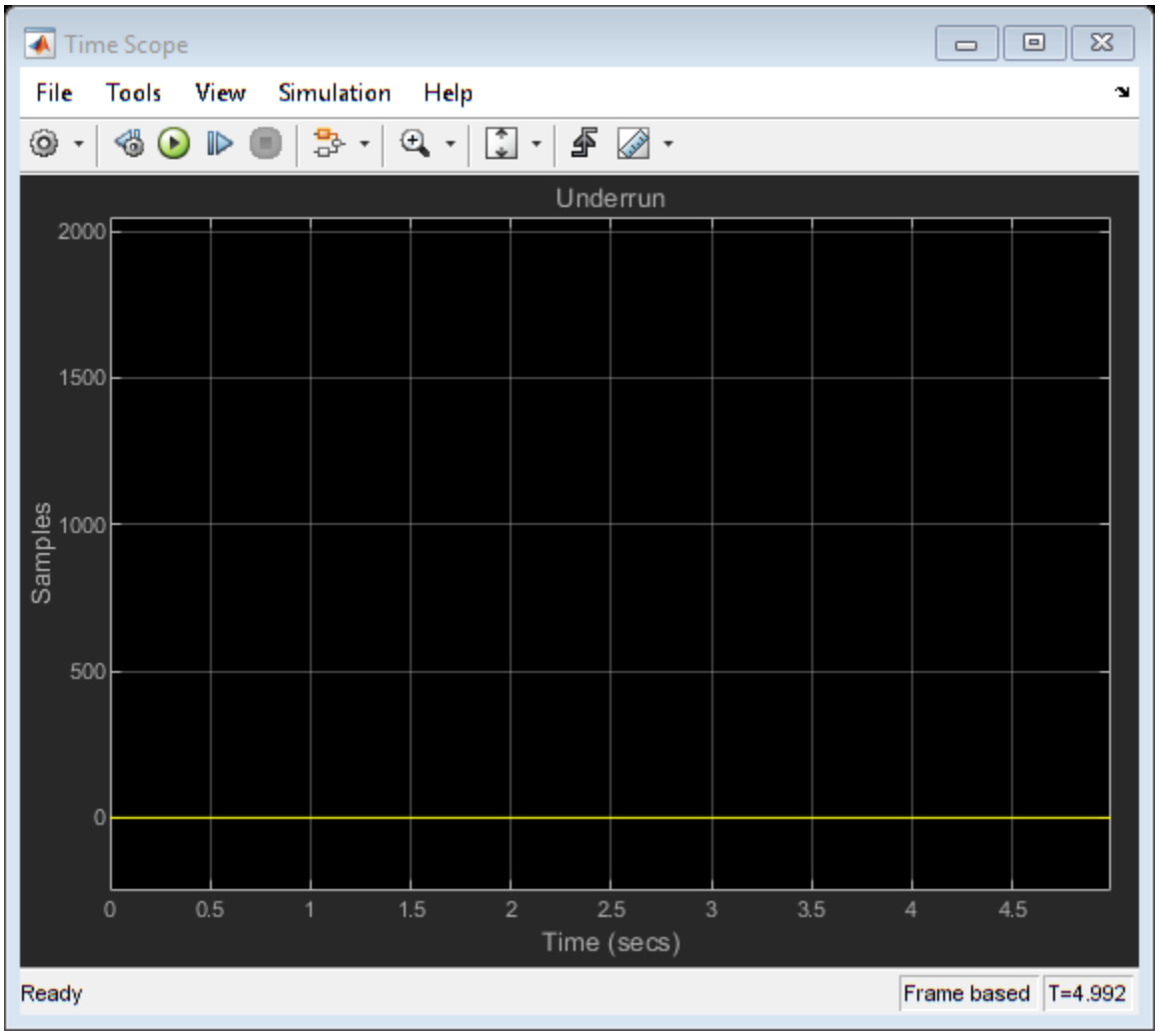
Block Example Repository

Decrease Underrun

Examine the Audio Device Writer block in a Simulink® model, determine underrun, and decrease underrun.



1. Run the model. The Audio Device Writer sends an audio stream to your computer's default audio output device. The Audio Device Writer block sends the number of samples underrun to your Time Scope.



2. Uncomment the Artificial Load block. This block performs computations that slow the simulation.

3. Run the model. If your device writer is dropping samples:

a. Stop the simulation.

b. Open the From Multimedia File block.

c. Set the **Samples per frame** parameter to `1024`.

d. Close the block and run the simulation.

If your model continues to drop samples, increase the frame size again. The increased frame size increases the buffer size used by the sound card. A larger buffer size decreases the possibility of underruns at the cost of higher audio latency.

See Also

From Multimedia File | Time Scope