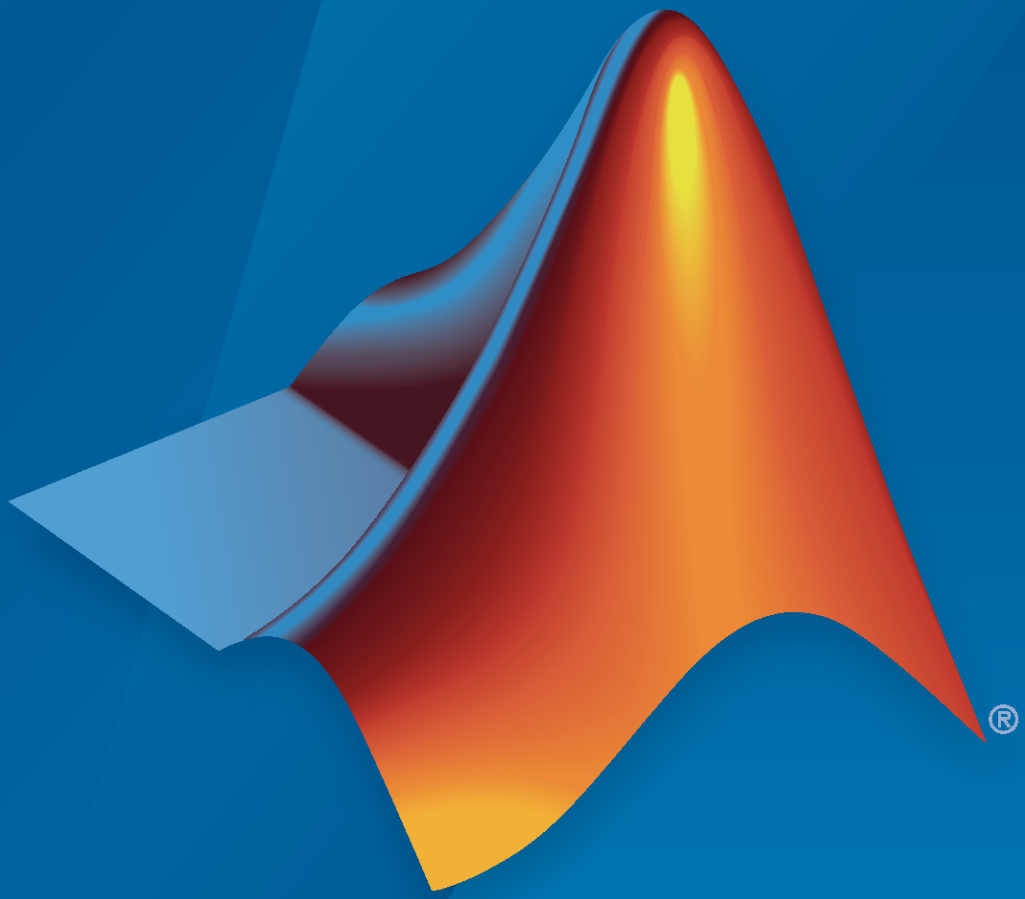
DSP System Toolbox™

User's Guide



MATLAB® & SIMULINK®

R2021b

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 2011–2021 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)
September 2020	Online only	Revised for Version 9.11 (R2020b)
March 2021	Online only	Revised for Version 9.12 (R2021a)
September 2021	Online only	Revised for Version 9.13 (R2021b)

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

Multirate Filtering in MATLAB and Simulink 1-36
- Implement an FIR Decimator in MATLAB 1-36
- Implement an FIR Decimator in Simulink 1-39
- Sample Rate Conversion 1-43

Tunable Lowpass Filtering of Noisy Input in Simulink 1-47
- Open Lowpass Filter Model 1-47
- Simulate the Model 1-49

Signal Processing Algorithm Acceleration in MATLAB 1-53
- FIR Filter Algorithm 1-53
- Accelerate the FIR Filter Using codegen 1-54
- Accelerate the FIR Filter Using dspunfold 1-55
- Kalman Filter Algorithm 1-56
- Accelerate the Kalman Filter Using codegen 1-58
- Accelerate the Kalman Filter Using dspunfold 1-59

Signal Processing Acceleration through Code Generation 1-61
- FIR Filter Algorithm 1-61
- Accelerate the FIR Filter Using codegen 1-62
- Accelerate the FIR Filter Using dspunfold 1-63

Multithreaded MEX File Generation 1-66

Fixed-Point Filter Design in MATLAB **1-73**

Visualizing Multiple Signals Using Logic Analyzer **1-79**
Model Programmable FIR Filter **1-79**
Simulation **1-80**
Use the Logic Analyzer **1-81**
Modify the Display **1-82**

Signal Visualization and Measurements in MATLAB **1-87**

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 3-6
Inspect Signal Rates Using Color Coding 3-7
Use Model Data Editor or Probe to Inspect Signals 3-15

Convert Sample and Frame Rates in Simulink Using Rate Conversion Blocks 3-16
Rate Conversion Blocks 3-16
Rate Conversion by Frame-Rate Adjustment 3-17
Rate Conversion by Frame-Size Adjustment 3-20

Convert Sample and Frame Rates in Simulink Using Frame Rebuffering Blocks 3-23
Frame Rebuffering Blocks 3-23
Buffer Signals by Preserving Sample Period 3-24
Buffer Signals by Altering the Sample Period 3-26

Buffering and Frame-Based Processing 3-29
Buffer Input into Frames 3-29
Buffer Signals into Frames with Overlap 3-31
Buffer Frame Inputs into Other Frame Inputs 3-33
Buffer Delay and Initial Conditions 3-35
Unbuffer Frame Signals into Sample Signals 3-36

Delay and Latency 3-40
Computational Delay 3-40
Algorithmic Delay 3-41
Zero Algorithmic Delay 3-41
Basic Algorithmic Delay 3-43
Excess Algorithmic Delay (Tasking Latency) 3-45
Predict Tasking Latency 3-46

Variable-Size Signal Support DSP System Objects 3-51
Variable-Size Signal Support Example 3-51
DSP System Toolbox System Objects That Support Variable-Size Signals 3-51

4 DSP System Toolbox Featured Examples

Wavelet Denoising 4-3

LPC Analysis and Synthesis of Speech 4-7

Streaming Signal Statistics 4-12

High Resolution Spectral Analysis in MATLAB 4-16

High Resolution Spectral Analysis in Simulink 4-33

Zoom FFT 4-39

Outlier Removal Techniques with Streaming ECG Signals 4-49

Sigma-Delta A/D Conversion 4-53

GSM Digital Down Converter in Simulink 4-55

Overlap-Add/Save 4-61

Designing Low Pass FIR Filters 4-64

Classic IIR Filter Design 4-79

Efficient Narrow Transition-Band FIR Filter Design 4-89

IIR Filter Design Given a Prescribed Group Delay 4-96

FIR Nyquist (L-th band) Filter Design 4-105

FIR Halfband Filter Design 4-112

Arbitrary Magnitude Filter Design 4-128

Design of Peaking and Notching Filters 4-141

Fractional Delay Filters Using Farrow Structures 4-151

Least Pth-norm Optimal FIR Filter Design 4-159

Least Pth-Norm Optimal IIR Filter Design 4-169

Multistage Rate Conversion 4-177

Complex Bandpass Filter Design 4-190

Design of Fractional Delay FIR Filters 4-196

Design of Decimators and Interpolators 4-211

Multistage Halfband IIR Filter Design **4-229**

Efficient Sample Rate Conversion Between Arbitrary Factors **4-234**

Reconstruction Through Two-Channel Filter Banks **4-241**

Adaptive Line Enhancer (ALE) **4-248**

Adaptive Noise Canceling (ANC) Applied to Fetal Electrocardiography **4-256**

Adaptive Noise Cancellation Using RLS Adaptive Filtering **4-259**

System Identification Using RLS Adaptive Filtering **4-264**

Acoustic Noise Cancellation (LMS) **4-270**

Adaptive Filter Convergence **4-272**

Noise Canceler (RLS) **4-275**

Time-Delay Channel Estimation Through Adaptive Filtering **4-278**

Time Scope Measurements **4-281**

Spectrum Analyzer Measurements **4-290**

Generate a Multithreaded MEX File from a MATLAB Function Using Unfolding **4-297**

Generate Standalone Executable And Interact With It Using UDP **4-303**

Code Generation for Parametric Audio Equalizer **4-306**

Generate DSP Applications with MATLAB Compiler **4-314**

Optimized Fixed-Point FIR Filters **4-320**

Floating-Point to Fixed-Point Conversion of IIR Filters **4-328**

GSM Digital Down Converter in MATLAB **4-346**

Cochlear Implant Speech Processor **4-353**

Three-Channel Wavelet Transmultiplexer **4-358**

Arbitrary Magnitude and Phase Filter Design **4-364**

G.729 Voice Activity Detection **4-381**

IF Subsampling with Complex Multirate Filters **4-385**

Design and Analysis of a Digital Down Converter **4-395**

Comparison of LDM, CVSD, and ADPCM **4-404**

Digital Up and Down Conversion for Family Radio Service **4-409**

Parametric Audio Equalizer **4-426**

Envelope Detection **4-430**

DTMF Generator and Receiver **4-438**

WWV Digital Receiver - Synchronization and Detection **4-441**

Real-Time ECG QRS Detection **4-448**

Internet Low Bitrate Codec (iLBC) for VoIP **4-454**

5 Filter Analysis, Design, and Implementation

Design a Filter in Fdesign — Process Overview **5-2**
Process Flow Diagram and Filter Design Methodology **5-2**

Use Filter Designer with DSP System Toolbox Software **5-9**
Design Advanced Filters in Filter Designer **5-9**
Access the Quantization Features of Filter Designer **5-11**
Quantize Filters in Filter Designer **5-13**
Analyze Filters with a Noise-Based Method **5-18**
Scale Second-Order Section Filters **5-22**
Reorder the Sections of Second-Order Section Filters **5-25**
View SOS Filter Sections **5-28**
Import and Export Quantized Filters **5-32**
Generate MATLAB Code **5-35**
Import XILINX Coefficient (.COE) Files **5-35**
Transform Filters Using Filter Designer **5-36**
Design Multirate Filters in Filter Designer **5-42**
Realize Filters as Simulink Subsystem Blocks **5-50**

Digital Frequency Transformations **5-53**
Details and Methodology **5-53**
Frequency Transformations for Real Filters **5-58**
Frequency Transformations for Complex Filters **5-67**

Digital Filter Design Block **5-76**
Overview of the Digital Filter Design Block **5-76**
Select a Filter Design Block **5-77**
Create a Lowpass Filter in Simulink **5-78**
Create a Highpass Filter in Simulink **5-78**
Filter High-Frequency Noise in Simulink **5-79**

Filter Realization Wizard **5-83**
Overview of the Filter Realization Wizard **5-83**
Design and Implement a Fixed-Point Filter in Simulink **5-83**

Set the Filter Structure and Number of Filter Sections 5-90
Optimize the Filter Structure 5-90

Digital Filter Implementations 5-93
Using Digital Filter Blocks 5-93
Implement a Lowpass Filter in Simulink 5-93
Implement a Highpass Filter in Simulink 5-94
Filter High-Frequency Noise in Simulink 5-95
Specify Static Filters 5-98
Specify Time-Varying Filters 5-99
Specify the SOS Matrix (Biquadratic Filter Coefficients) 5-99

Removing High-Frequency Noise from an ECG Signal 5-101

Minimax FIR Filter Design 5-103

6 Adaptive Filters

Overview of Adaptive Filters and Applications 6-2
Adaptive Filters in DSP System Toolbox 6-2
Choosing an Adaptive Filter 6-4
Mean Squared Error Performance 6-5
Common Applications 6-5

System Identification of FIR Filter Using LMS Algorithm 6-9

System Identification of FIR Filter Using Normalized LMS Algorithm . . 6-17

Compare Convergence Performance Between LMS Algorithm and Normalized LMS Algorithm 6-20

Noise Cancellation Using Sign-Data LMS Algorithm 6-22

Compare RLS and LMS Adaptive Filter Algorithms 6-26

Inverse System Identification Using RLS Algorithm 6-29

Signal Enhancement Using LMS and NLMS Algorithms 6-34

Noise Cancellation in Simulink Using Normalized LMS Adaptive Filter 6-43
Create an Acoustic Environment in Simulink 6-43
LMS Filter Configuration for Adaptive Noise Cancellation 6-44
Modify Adaptive Filter Parameters During Model Simulation 6-47

7 Multirate and Multistage Filters

Overview of Multirate Filters **7-2**
Decimation and Interpolation **7-2**
Decimators **7-3**
Interpolators **7-4**
Sample Rate Converters **7-6**

Overview of Multistage Filters **7-11**
Multistage Decimator **7-11**
Multistage Interpolator **7-12**
Determine the number of stages and rate conversion factor for each stage **7-12**

Overview of Filter Banks **7-14**
Analysis Filter Bank (Channelizer) **7-14**
Synthesis Filter Bank (Channel synthesizer) **7-15**
Two-Channel (Halfband) Filter Bank **7-16**

Two-Channel Filter Bank Using Halfband Decimators and Halfband Interpolators **7-19**

Channelize and Synthesize Sine Wave in MATLAB **7-26**

Multilevel Filter Banks **7-28**
Dyadic Analysis Filter Banks **7-28**
Dyadic Synthesis Filter Banks **7-30**

8 Dataflow

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

Model Multirate Signal Processing Systems Using Dataflow **8-10**

Multicore Simulation and Code Generation of Dataflow Domains **8-12**
Simulation of Dataflow Domains **8-12**
Code Generation of Dataflow Domains **8-12**
Types of Parallelism **8-12**
Improve Simulation Throughput with Multicore Simulation **8-14**
Generate Multicore Code from a Dataflow Subsystem **8-16**

Multicore Execution using Dataflow Domain **8-19**

Multicore Code Generation for Dataflow Domain **8-27**

Perform Multicore Analysis for Dataflow **8-34**
Select the Cost Calculation Method **8-35**
Manually Change Block Costs **8-37**
Specify Analysis Constraints and Run Analysis **8-37**
Review Results **8-37**

Multicore Analysis Using a Dataflow Domain **8-41**

9 Simulink HDL Optimized Block Examples in DSP System Toolbox

Generate a Sine Wave **9-2**

Implement CIC Decimation Filter for HDL **9-5**

Fully Parallel Systolic FIR Filter Implementation **9-8**

Partly Serial Systolic FIR Filter Implementation **9-12**

Optimize Programmable FIR Filter Resources **9-16**

FIR Decimation for FPGA **9-21**

High Throughput Channelizer for FPGA **9-23**

Implement atan2 Function for HDL **9-31**

Downsample a Signal **9-34**

Control Data Rate Using the Ready and Request Ports **9-37**

Implement FFT for FPGA Using FFT HDL Optimized Block **9-40**

Automatic Delay Matching for the Latency of FFT HDL Optimized Block **9-44**

10 Simulink Block Examples in Multirate and Multistage Filters

FIR Decimation Using Single-Rate Processing **10-2**

FIR Decimation Using Multirate Frame-Based Processing **10-3**

Polyphase Implementation of FIR Decimation Block **10-4**

Two-Stage Multirate Narrow Lowpass Filter **10-6**

FIR Interpolation Using Single-Rate Processing 10-8

FIR Interpolation Using Multirate Frame-Based Processing 10-9

Polyphase Implementation of FIR Interpolation Block 10-10

11 Simulink Block Examples in Scopes and Data Logging Category

Obtain Measurement Data Programmatically for dsp.SpectrumAnalyzer System object 11-2

Obtain Measurements Data Programmatically for Spectrum Analyzer Block 11-5

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

Write and Read Binary Files in Simulink 12-2

Write and Read Matrix Data from Binary Files in Simulink 12-6

Write and Read Fixed-Point Data from Binary Files in Simulink 12-8

Write and Read Character Data from Binary Files in Simulink 12-10

Change the Endianness of the Data in Simulink 12-11

13 Simulink Block Examples in Signal Generation and Operations Category

Delay Signal Using Multitap Fractional Delay 13-2

Bidirectional Linear Sweep 13-7

Unidirectional Linear Sweep 13-10

When Sweep Time Is Greater than Target Time 13-12

Sweep with Negative Frequencies 13-14

Aliased Sweep **13-17**

Generate Discrete Impulse with Three Channels **13-19**

Generate Unit-Diagonal and Identity Matrices **13-20**

Generate Five-Phase Output from the Multiphase Clock Block **13-21**

Count Down Through Range of Numbers **13-23**

Import Two-Channel Signal From Workspace **13-25**

Import 3-D Array From Workspace **13-26**

Generate Sample-Based Sine Waves **13-27**

Generate Frame-Based Sine Waves **13-28**

Design an NCO Source Block **13-29**

Generate Constant Ramp Signal **13-32**

Averaged Power Spectrum of Pink Noise **13-33**

Downsample a Signal **13-35**

Sample and Hold a Signal **13-38**

Generate and Apply Hamming Window **13-41**

Convert Sample Rate of Speech Signal **13-44**

Unwrap Signal **13-46**

Convolution of Two Inputs **13-48**

Select Rows or Columns from Matrices **13-50**

Convert 2-D Matrix to 1-D Array **13-51**

Pad or Truncate Matrix with Constant Values **13-52**

Extract the Phase of Sine Wave **13-54**

Queues **13-56**

14 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? **14-2**

Why Does the dsp.AsyncBuffer Object Error When You Call read Before write? 14-7

Buffering Input with Overlap 14-9

15 Simulink Block Examples in DSP System Toolbox

Synthesize and Channelize Audio in Simulink 15-2

Filter input with Butterworth Filter in Simulink 15-9

SSB Modulation 15-10

Wavelet Reconstruction and Noise Reduction 15-15

16 Simulink Block Examples in DSP System Toolbox

Compute the Maximum 16-2

Compute the Running Maximum 16-4

Compute the Minimum 16-6

Compute the Running Minimum 16-8

Compute the Mean 16-10

Compute the Running Mean 16-12

Compute the Histogram of Real and Complex Data 16-14

Extract Submatrix from Input Signal 16-19

Compute Difference of a Matrix 16-21

Compute Maximum Column Sum of Matrix 16-22

Extract Diagonal of Matrix 16-23

Generate Diagonal Matrix from Vector Input 16-24

Permute Matrix by Row or Column 16-25

LDL Factorization of 3-by-3 Hermitian Positive Definite Matrix 16-26

Compute Power Measurements of Voltage Signal in Simulink 16-27

Compute Matrix Exponential **16-30**

Compute Moving RMS of Noisy Step Signal **16-31**

Compute RMS of Noisy Step Signal **16-34**

17 Simulink Block Examples in Transforms and Spectral Analysis Category

Analyze a Subband of Input Frequencies Using Zoom FFT **17-2**

Group Delay Estimation in Simulink **17-4**

High Resolution Filter-Bank-Based Power Spectrum Estimation **17-7**

Estimate Data Series Using Forward Linear Predictor **17-11**

Continuous-Time Transfer Function Estimation **17-13**

18 Transforms, Estimation, and Spectral Analysis

Transform Time-Domain Data into Frequency Domain **18-2**

Transform Frequency-Domain Data into Time Domain **18-5**

Linear and Bit-Reversed Output Order **18-7**
FFT and IFFT Blocks Data Order **18-7**
Find the Bit-Reversed Order of Your Frequency Indices **18-7**

Calculate Channel Latencies Required for Wavelet Reconstruction **18-9**
Analyze Your Model **18-9**
Calculate the Group Delay of Your Filters **18-10**
Reconstruct the Filter Bank System **18-11**
Equalize the Delay on Each Filter Path **18-12**
Update and Run the Model **18-13**
References **18-14**

Estimate the Power Spectrum in MATLAB **18-15**
Estimate the Power Spectrum Using dsp.SpectrumAnalyzer **18-15**
Convert the Power Between Units **18-22**
Estimate the Power Spectrum Using dsp.SpectrumEstimator **18-24**

Estimate the Power Spectrum in Simulink **18-28**
Estimate the Power Spectrum Using the Spectrum Analyzer **18-28**
Convert the Power Between Units **18-37**
Estimate Power Spectrum Using the Spectrum Estimator Block **18-39**

Estimate the Transfer Function of an Unknown System **18-44**
Estimate the Transfer Function in MATLAB **18-44**
Estimate the Transfer Function in Simulink **18-48**

View the Spectrogram Using Spectrum Analyzer **18-52**
Colormap **18-53**
Display **18-54**
Resolution Bandwidth (RBW) **18-54**
Time Resolution **18-57**
Convert the Power Between Units **18-57**
Scale Color Limits **18-59**

Spectral Analysis **18-61**
Welch’s Algorithm of Averaging Modified Periodograms **18-61**
Filter Bank **18-64**

Streaming Power Spectrum Estimation Using Welch's Method **18-65**

19 Fixed-Point Design

Fixed-Point Signal Processing **19-2**
Fixed-Point Features **19-2**
Benefits of Fixed-Point Hardware **19-2**
Benefits of Fixed-Point Design with System Toolboxes Software **19-2**

Fixed-Point Concepts and Terminology **19-4**
Fixed-Point Data Types **19-4**
Scaling **19-5**
Precision and Range **19-6**

Arithmetic Operations **19-8**
Modulo Arithmetic **19-8**
Two's Complement **19-8**
Addition and Subtraction **19-9**
Multiplication **19-10**
Casts **19-12**

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

Simulink Blocks in DSP System Toolbox that Support Fixed-Point Design **19-19**

System Objects Supported by Fixed-Point Converter App **19-20**

Convert dsp.FIRFilter Object to Fixed-Point Using the Fixed-Point Converter App **19-21**
Create DSP Filter Function and Test Bench **19-21**

Convert the Function to Fixed-Point 19-22

Specify Fixed-Point Attributes for Blocks 19-28
Fixed-Point Block Parameters 19-28
Specify System-Level Settings 19-30
Inherit via Internal Rule 19-30
Specify Data Types for Fixed-Point Blocks 19-37

Quantizers 19-43
Scalar Quantizers 19-43
Vector Quantizers 19-46

Create an FIR Filter Using Integer Coefficients 19-50
Define the Filter Coefficients 19-50
Build the FIR Filter 19-50
Set the Filter Parameters to Work with Integers 19-51
Create a Test Signal for the Filter 19-52
Filter the Test Signal 19-52
Truncate the Output WordLength 19-54
Scale the Output 19-56
Configure Filter Parameters to Work with Integers Using the set2int Method 19-59

Fixed-Point Precision Rules for Avoiding Overflow in FIR Filters 19-62
Output Limits for FIR Filters 19-62
Fixed-Point Precision Rules 19-64
Polyphase Interpolators and Decimators 19-65

20 C Code Generation

Functions and System Objects in DSP System Toolbox that Support C Code Generation 20-2

Simulink Blocks in DSP System Toolbox that Support C Code Generation 20-4

Understanding C Code Generation in DSP System Toolbox 20-6
Generate C and C++ code from MATLAB code 20-6
Generate C and C++ Code from a Simulink Model 20-6
Shared Library Dependencies 20-7
Generate C Code for ARM Cortex-M and ARM Cortex-A Processors 20-8
Generate Code for Mobile Devices 20-8

Generate C Code from MATLAB Code 20-10
Set Up the Compiler 20-10
Break Out the Computational Part of the Algorithm into a MATLAB Function 20-10
Make Code Suitable for Code Generation 20-11
Compare the MEX Function with the Simulation 20-13
Generate a Standalone Executable 20-13
Read and Verify the Binary File Data 20-15

Relocate Code to Another Development Environment **20-16**

Relocate Code Generated from MATLAB Code to Another Development Environment **20-17**
Package the Code **20-17**
Prebuilt Dynamic Library Files (.dll) **20-17**

Generate C Code from Simulink Model **20-19**
Open the Model **20-19**
Configure Model for Code Generation **20-20**
Simulate the Model **20-20**
Generate Code from the Model **20-21**
Build and Run the Generated Code **20-22**

Relocate Code Generated from a Simulink Model to Another Development Environment **20-24**
Package the Code **20-24**
Prebuilt Dynamic Library Files (.dll) **20-26**

How To Run a Generated Executable Outside MATLAB **20-27**

Use Generated Code to Accelerate an Application Deployed with MATLAB Compiler **20-30**

How Is dspunfold Different from parfor? **20-41**
DSP Algorithms Involve States **20-41**
dspunfold Introduces Latency **20-41**
parfor Requires Significant Restructuring in Code **20-41**
parfor Used with dspunfold **20-41**

Workflow for Generating a Multithreaded MEX File using dspunfold . **20-43**
Workflow Example **20-43**

Why Does the Analyzer Choose the Wrong State Length? **20-47**
Reason for Verification Failure **20-48**
Recommendation **20-48**

Why Does the Analyzer Choose a Zero State Length? **20-49**
Recommendation **20-49**

Array Plot with Android Devices **20-50**

System objects in DSP System Toolbox that Support SIMD Code Generation **20-55**

Generate High Performance SIMD Code on Intel from MATLAB Algorithms in DSP System Toolbox **20-57**

Simulink Blocks in DSP System Toolbox that Support SIMD Code Generation **20-59**

Generate High Performance SIMD Code on Intel from Simulink Blocks in DSP System Toolbox **20-64**
Compare the Performance of SIMD Code with Generated Plain C Code **20-65**

In-Place Memory Optimization **20-67**

21 HDL Code Generation

Find Blocks That Support HDL Code Generation **21-2**
Blocks **21-2**
System Objects **21-3**

High Throughput HDL Algorithms **21-4**
Blocks with HDL Support for Frame Input **21-4**

HDL Filter Architectures **21-6**
Fully Parallel Architecture **21-6**
Serial Architectures **21-7**
Frame-Based Architecture **21-8**

Subsystem Optimizations for Filters **21-11**
Sharing **21-11**
Streaming **21-11**
Pipelining **21-11**
Area Reduction of Multichannel Filter Subsystem **21-12**
Area Reduction of Filter Subsystem **21-17**

Multichannel FIR Filter for FPGA **21-21**

Programmable FIR Filter for FPGA **21-24**

Implementing the Filter Chain of a Digital Down-Converter in HDL . . **21-30**

HDL Implementation of a Digital Down-Converter for LTE **21-50**

HDL Implementation of a Digital Up-Converter for LTE **21-69**

22 Links to Category Pages

Signal Management Library **22-2**

Sinks Library **22-3**

Math Functions Library **22-4**

Filtering Library **22-5**

23 Designing Lowpass FIR Filters

Lowpass FIR Filter Design **23-2**

Controlling Design Specifications in Lowpass FIR Design **23-7**

Designing Filters with Non-Equiripple Stopband **23-12**

Minimizing Lowpass FIR Filter Length **23-16**

24 Filter Designer: A Filter Design and Analysis App

Using Filter Designer **24-2**
- Choosing a Response Type **24-3**
- Choosing a Filter Design Method **24-3**
- Setting the Filter Design Specifications **24-4**
- Computing the Filter Coefficients **24-6**
- Analyzing the Filter **24-7**
- Editing the Filter Using the Pole/Zero Editor **24-11**
- Converting the Filter Structure **24-14**
- Exporting a Filter Design **24-16**
- Generating a C Header File **24-19**
- Generating MATLAB Code **24-20**
- Managing Filters in the Current Session **24-21**
- Saving and Opening Filter Design Sessions **24-22**

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

25 Designing a Filter in the Filter Builder GUI

Filter Builder Design Process **25-2**
- Introduction to Filter Builder **25-2**
- Design a Filter Using Filter Builder **25-2**
- Select a Response **25-2**
- Select a Specification **25-4**
- Select an Algorithm **25-5**
- Customize the Algorithm **25-6**
- Analyze the Design **25-7**
- Realize or Apply the Filter to Input Data **25-7**

Display Time-Domain Data **26-2**
Configure the Time Scope Properties **26-3**
Use the Simulation Controls **26-6**
Modify the Time Scope Display **26-7**
Inspect Your Data (Scaling the Axes and Zooming) **26-8**
Manage Multiple Time Scopes **26-10**
Display Complex-Valued Input Signal **26-12**
Display Input Signal of Changing Size **26-12**
Display Simulink Enumeration Input Signal **26-12**

Display Frequency-Domain Data in Spectrum Analyzer **26-18**

Visualize Central Limit Theorem in Array Plot **26-21**

Configure Spectrum Analyzer **26-24**
Signal and Spectrum Computation Information **26-24**
Generate a MATLAB Script **26-27**
Spectral Masks **26-29**
Measurements Panels **26-31**
Customize Visualization **26-41**
Zoom and Pan **26-42**

Configure Array Plot **26-43**
Signal Display **26-43**
Multiple Signal Names and Colors **26-45**
Configure Plot Settings **26-45**
Use Array Plot Measurements **26-46**
Share or Save the Array Plot **26-48**
Scale Axes **26-49**
Set Additional Properties **26-49**
Find the Array Plot Block in Your Model **26-49**

Configure Time Scope Block **26-50**
Signal Display **26-50**
Display Multiple Signals **26-52**
Time Scope Measurement Panels **26-54**
Style Dialog Box **26-78**
Axes Scaling Properties **26-78**
Sources — Streaming Properties **26-79**

Configure Time Scope MATLAB Object **26-80**
Signal Display **26-80**
Multiple Signal Names and Colors **26-81**
Configure Scope Settings **26-81**
Use timescope Measurements and Triggers **26-82**
Share or Save the Time Scope **26-98**
Scale Axes **26-99**

Common Scope Block Tasks **26-100**
Connect Multiple Signals to a Scope **26-100**
Save Simulation Data Using Scope Block **26-102**

Pause Display While Running 26-104
Copy Scope Image 26-104
Plot an Array of Signals 26-106
Scopes in Referenced Models 26-107
Scopes Within an Enabled Subsystem 26-111
Modify x-axis of Scope 26-112
Show Signal Units on a Scope Display 26-115
Select Number of Displays and Layout 26-117
Dock and Undock Scope Window to MATLAB Desktop 26-118

Display Frequency Input on Spectrum Analyzer 26-119

Use Peak Finder to Find Heart Rate from ECG Input 26-121

Configure Array Plot From the Command-Line 26-125

Waterfall Tasks 26-127
Scope Trigger Function 26-127
Scope Transform Function 26-129
Exporting Data 26-129
Capturing Data 26-129
Linking Scopes 26-130
Selecting Data 26-131
Zooming 26-132
Rotating the Display 26-132
Scaling the Axes 26-132
Saving Scope Settings 26-132

27 Logic Analyzer

Inspect and Measure Transitions Using the Logic Analyzer 27-2
Open a Simulink Model 27-2
Open the Logic Analyzer 27-2
Configure Global Settings and Visual Layout 27-3
Set Stepping Options 27-4
Run Model 27-5
Configure Individual Wave Settings 27-5
Inspect and Measure Transitions 27-5
Step Through Simulation 27-7
Save Logic Analyzer Settings 27-7

Configure Logic Analyzer 27-8

28 Statistics and Linear Algebra

What Are Moving Statistics? 28-2

Sliding Window Method and Exponential Weighting Method **28-5**
Sliding Window Method **28-5**
Exponential Weighting Method **28-7**

Measure Statistics of Streaming Signals **28-14**
Compute Moving Average Using Only MATLAB Functions **28-14**
Compute Moving Average Using System Objects **28-15**

How Is a Moving Average Filter Different from an FIR Filter? **28-17**
Frequency Response of Moving Average Filter and FIR Filter **28-17**

Energy Detection in the Time Domain **28-21**
Detect Signal Energy **28-21**

Remove High-Frequency Noise from Gyroscope Data **28-24**

Linear Algebra and Least Squares **28-26**
Linear Algebra Blocks **28-26**
Linear System Solvers **28-26**
Matrix Factorizations **28-27**
Matrix Inverses **28-28**

29 Bibliography

References — Advanced Filters **29-2**

References — Frequency Transformations **29-3**

30 Audio I/O User Guide

Run Audio I/O Features Outside MATLAB and Simulink **30-2**

31 Block Example Repository

Decrease Underrun **31-2**

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
- “Multirate Filtering in MATLAB and Simulink” on page 1-36
- “Tunable Lowpass Filtering of Noisy Input in Simulink” on page 1-47
- “Signal Processing Algorithm Acceleration in MATLAB” on page 1-53
- “Signal Processing Acceleration through Code Generation” on page 1-61
- “Multithreaded MEX File Generation” on page 1-66
- “Fixed-Point Filter Design in MATLAB” on page 1-73
- “Visualizing Multiple Signals Using Logic Analyzer” on page 1-79
- “Signal Visualization and Measurements in MATLAB” on page 1-87

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

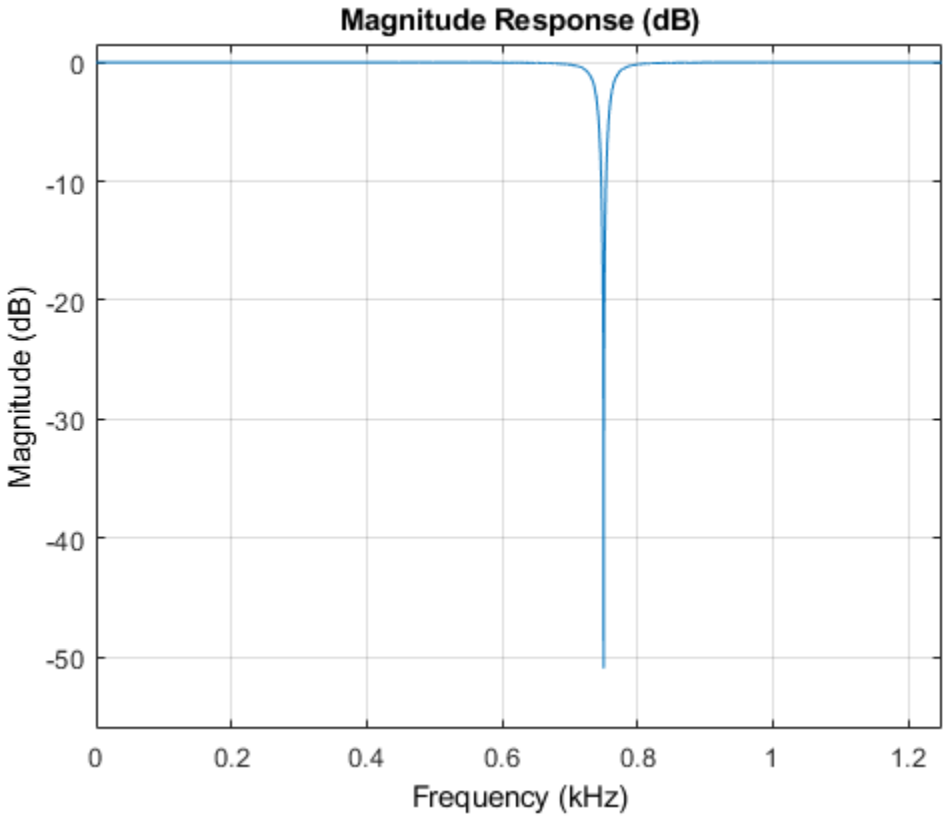
Initialize the sine wave source to generate the sine wave, the notch-peak filter to filter the sine wave, and the spectrum analyzer to show the filtered signal. The input sine wave has two frequencies: one at 100 Hz, and the other at 1000 Hz. Create two `dsp.SineWave` objects, one to generate the 100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

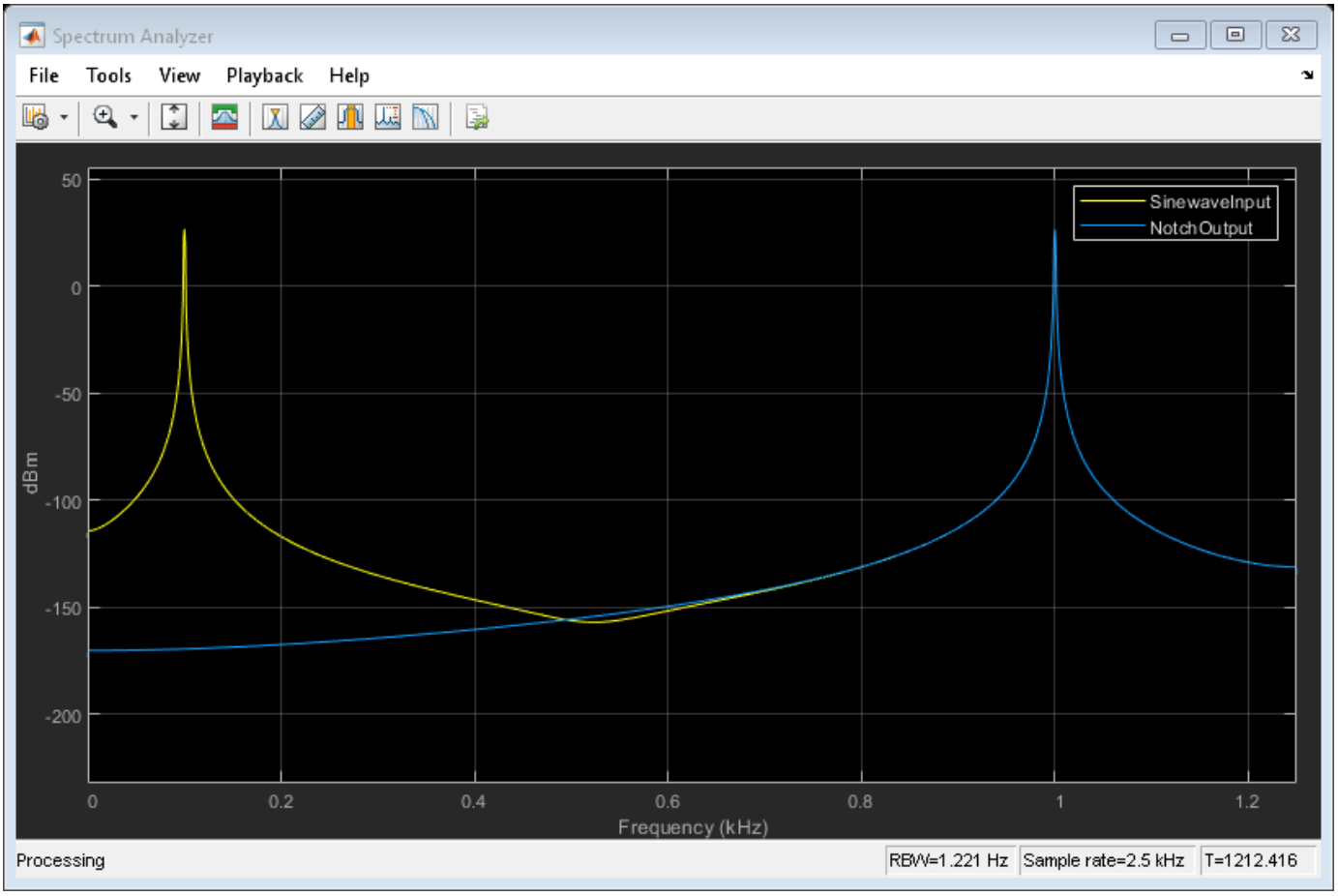


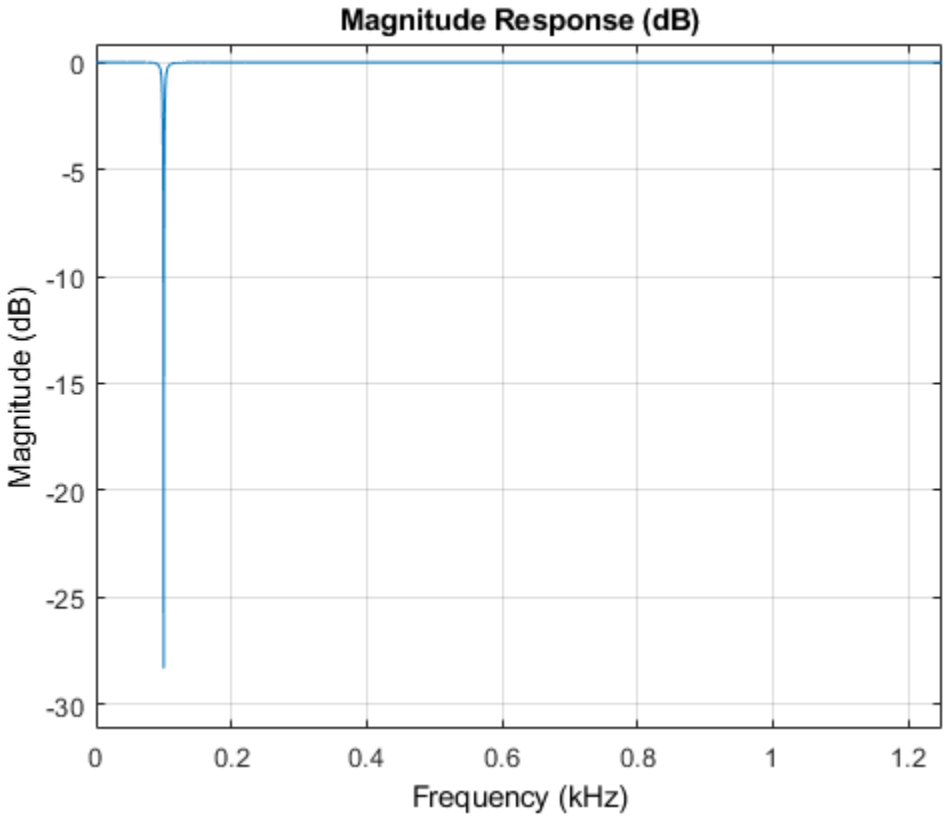
Stream In and Process Signal

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 | "Multirate Filtering in MATLAB and Simulink" 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',...
    'Lowpass 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.

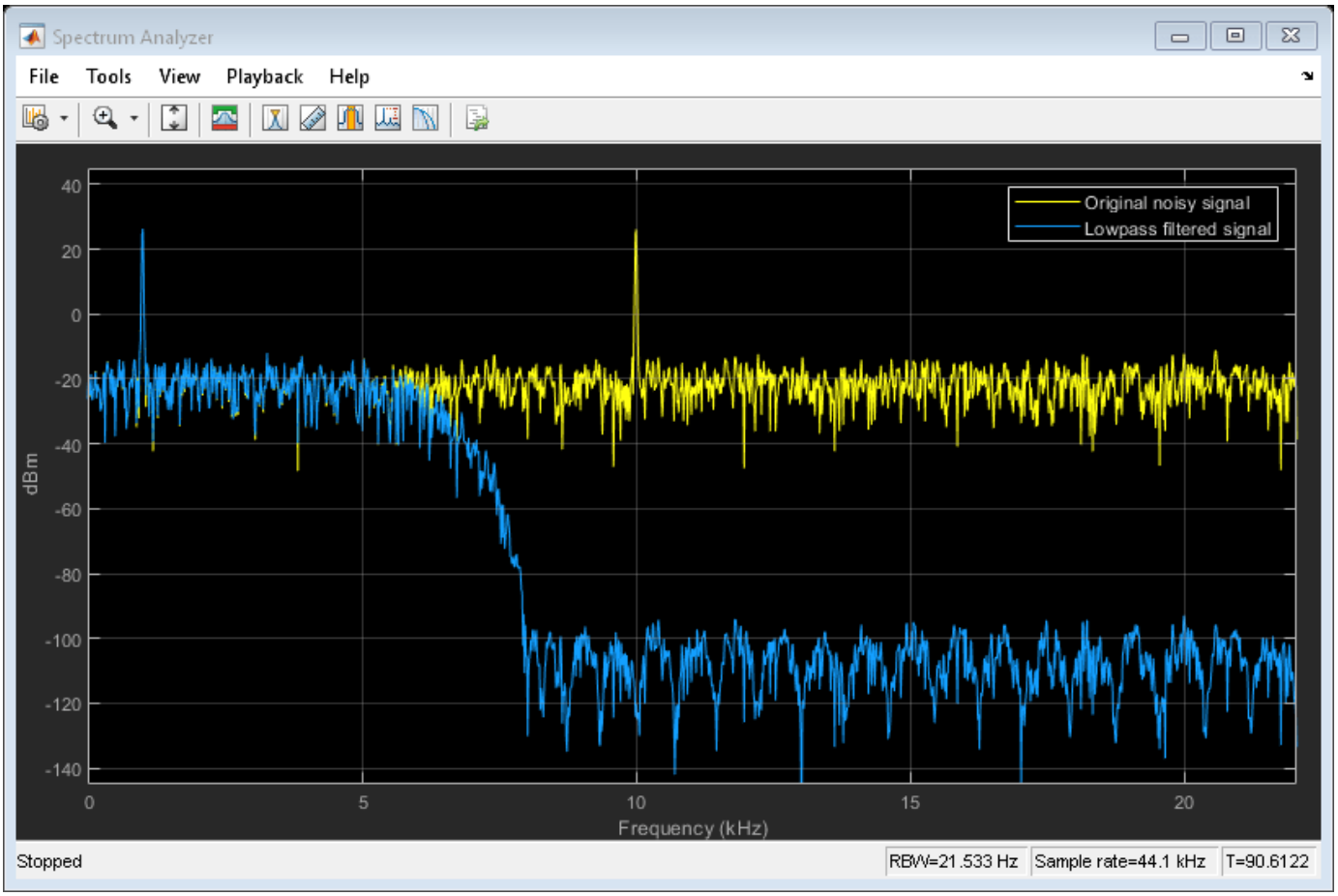
```
Sine1.SamplesPerFrame = 4000;
Sine2.SamplesPerFrame = 4000;
```

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 | "Multirate Filtering in MATLAB and Simulink" on page 1-36

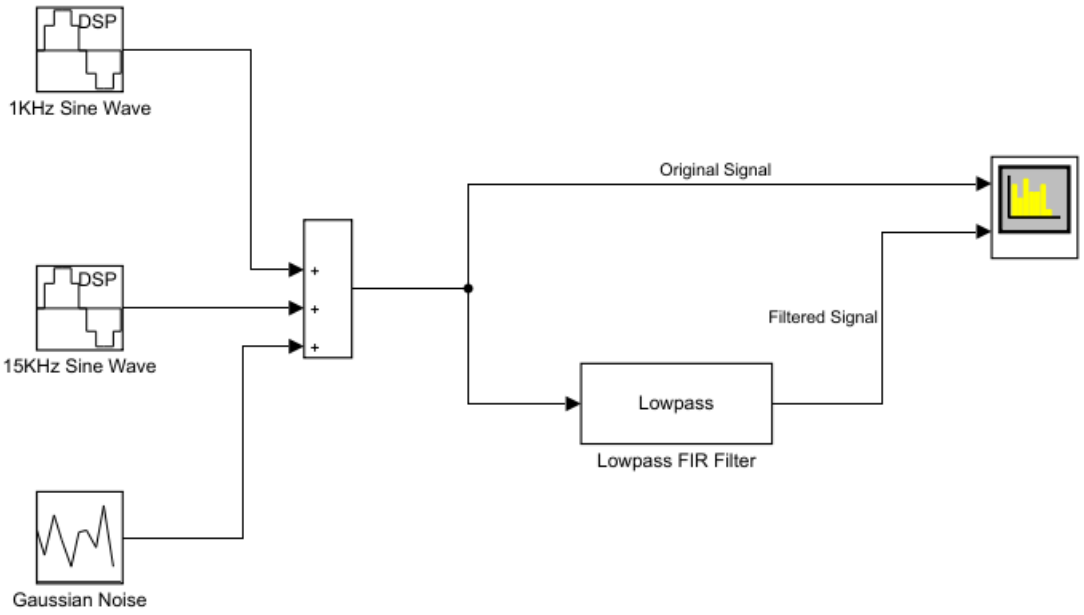
Filter Frames of a Noisy Sine Wave Signal in Simulink

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.

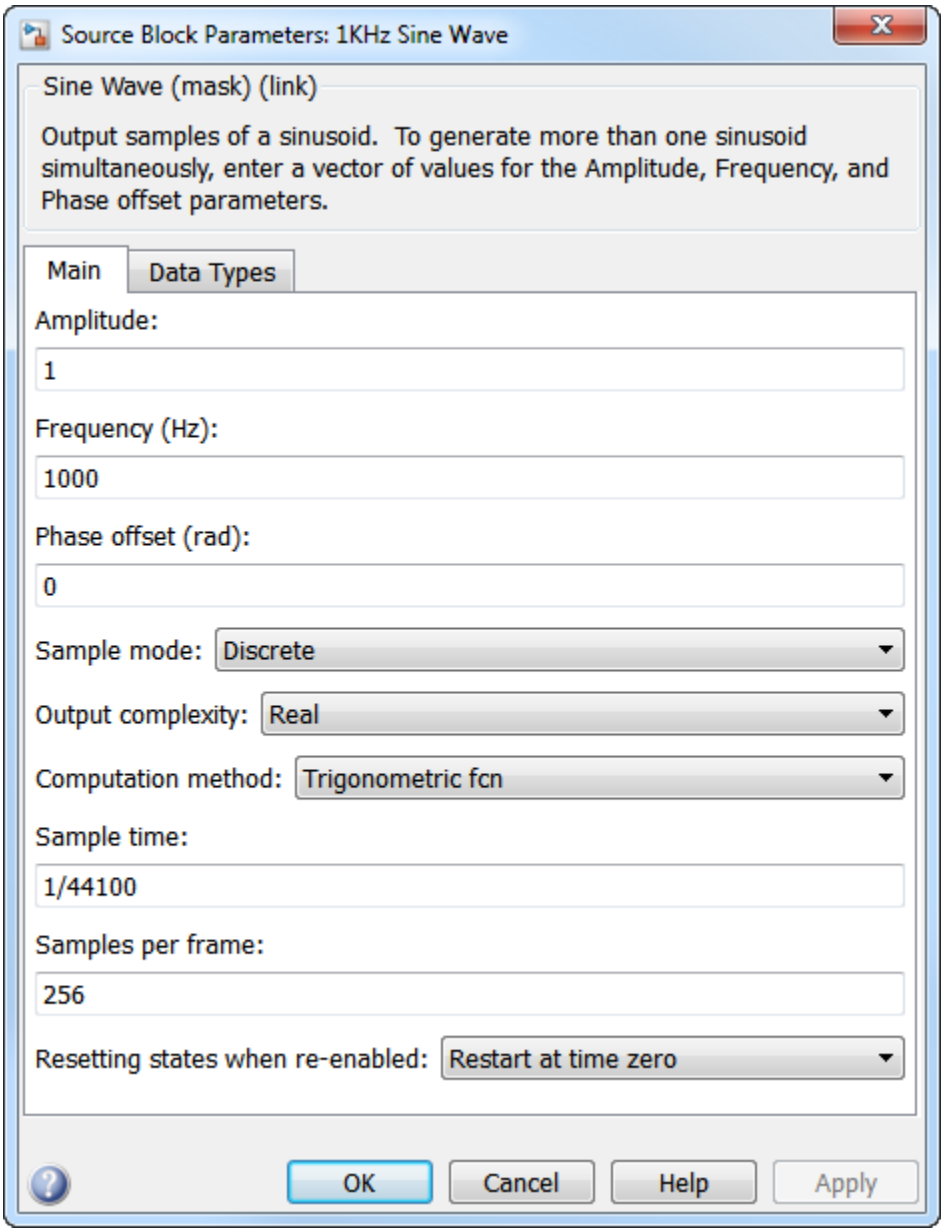


The new model using the template settings and contents appears in the Simulink Editor. The model is only in memory until you save it.

Inspect Model

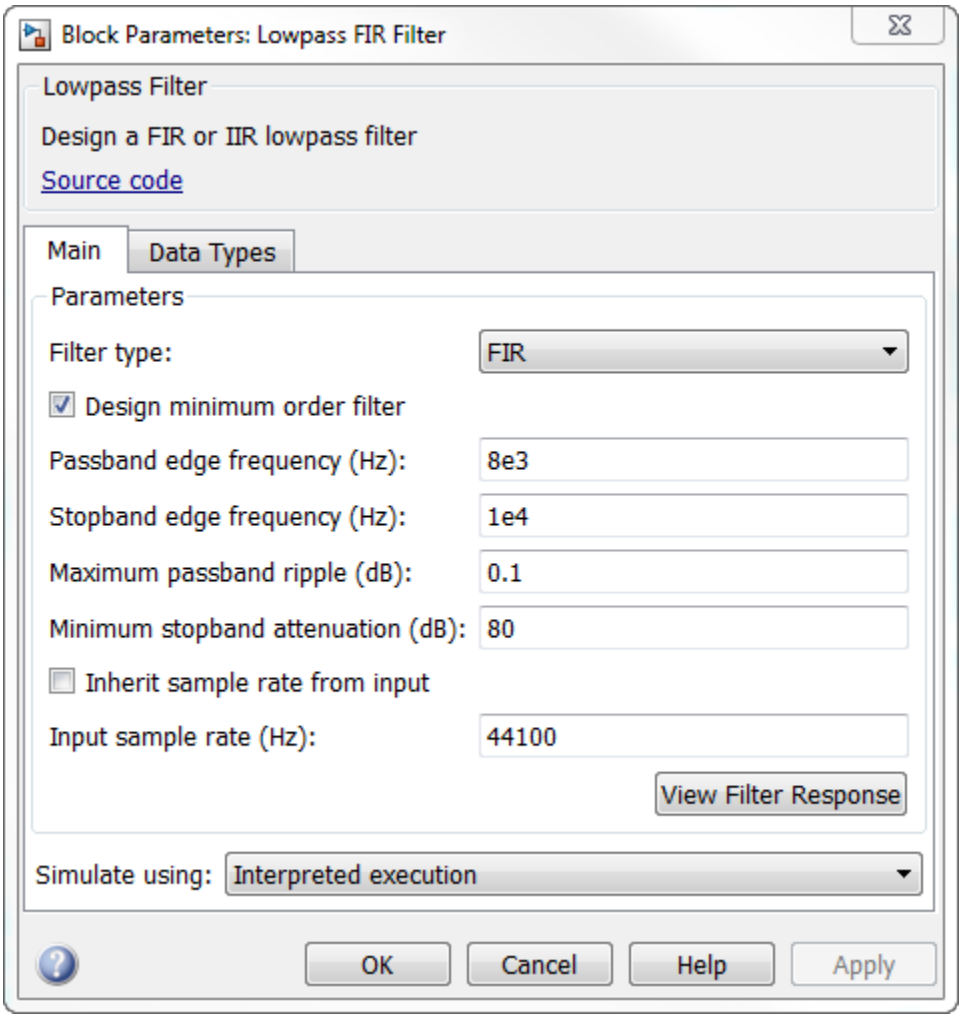
Input Signal

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.



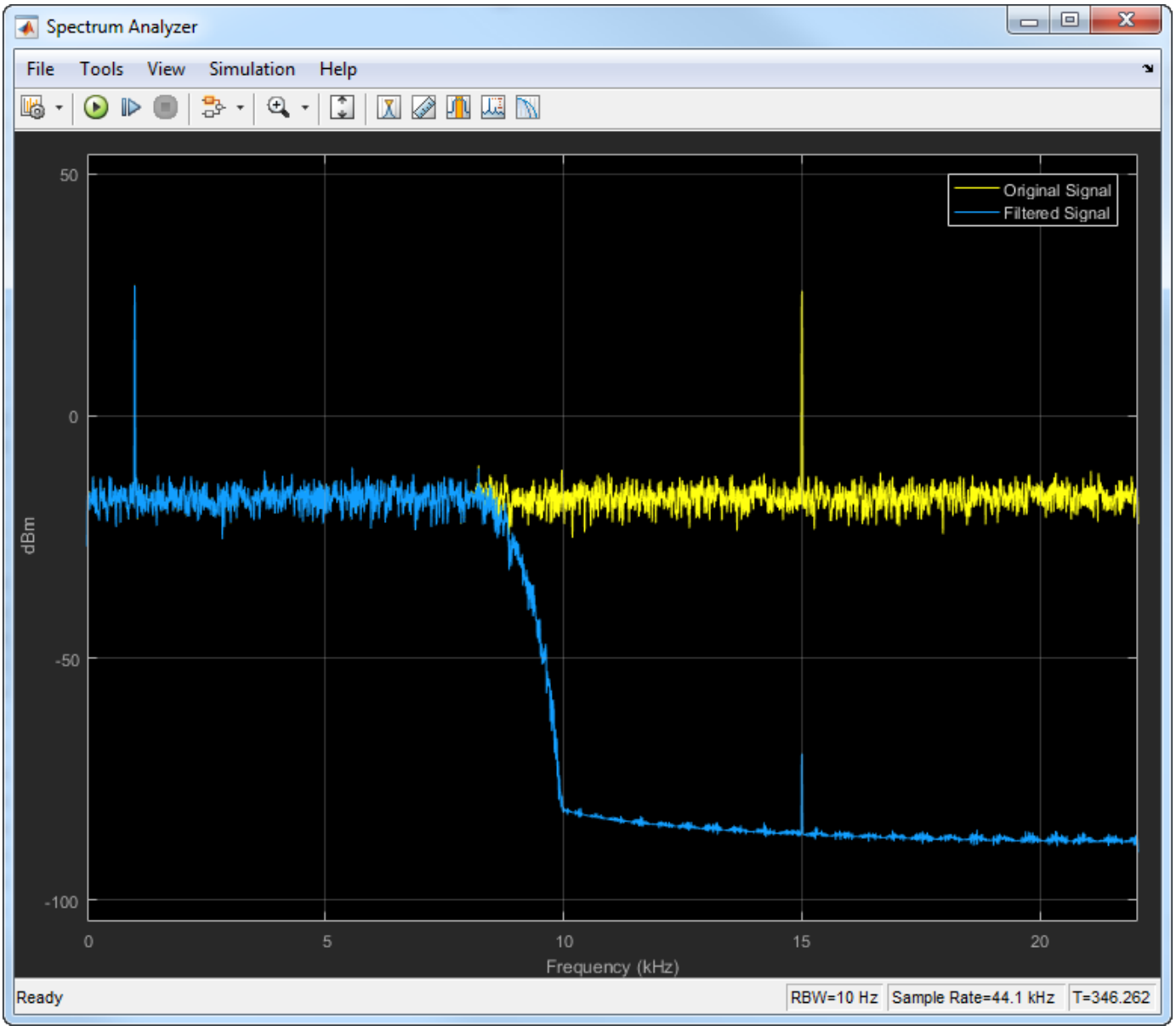
The Lowpass Filter block uses frame-based processing to process data one frame at a time. Each frame of data contains sequential samples from an independent channel. Frame-based processing is advantageous for many signal processing applications because you can process multiple samples at once. By buffering 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” on page 4-64.

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.

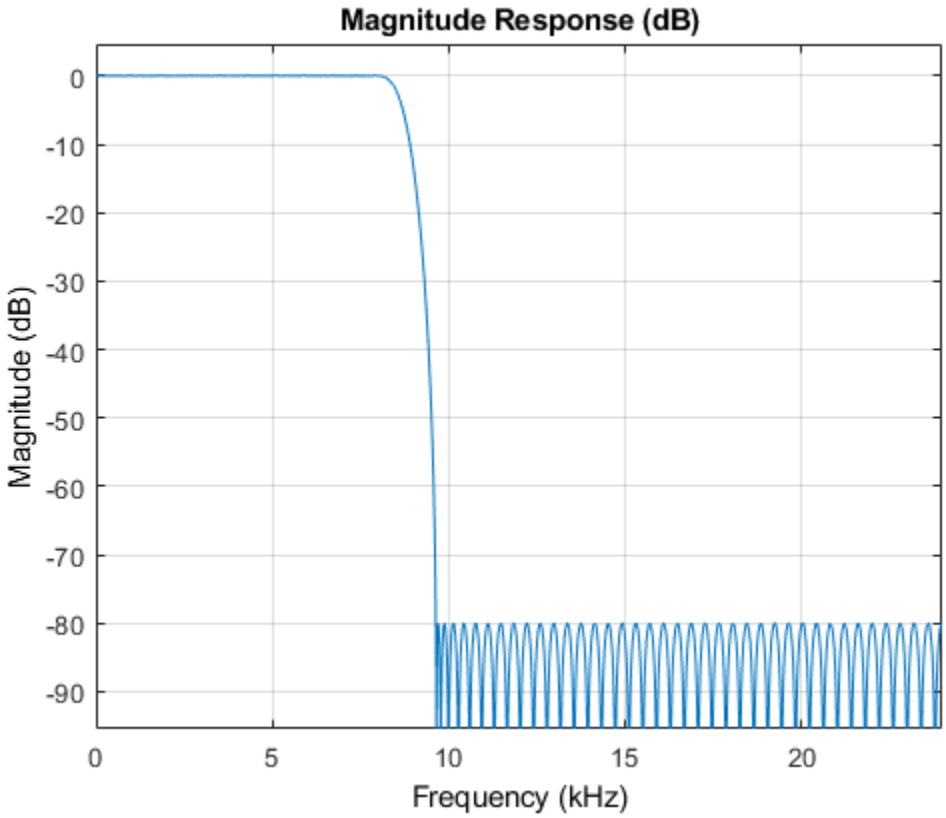
```
N = 120;
Fs = 48e3;
Fp = 8e3;
Ap = 0.01;
Ast = 80;
```

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

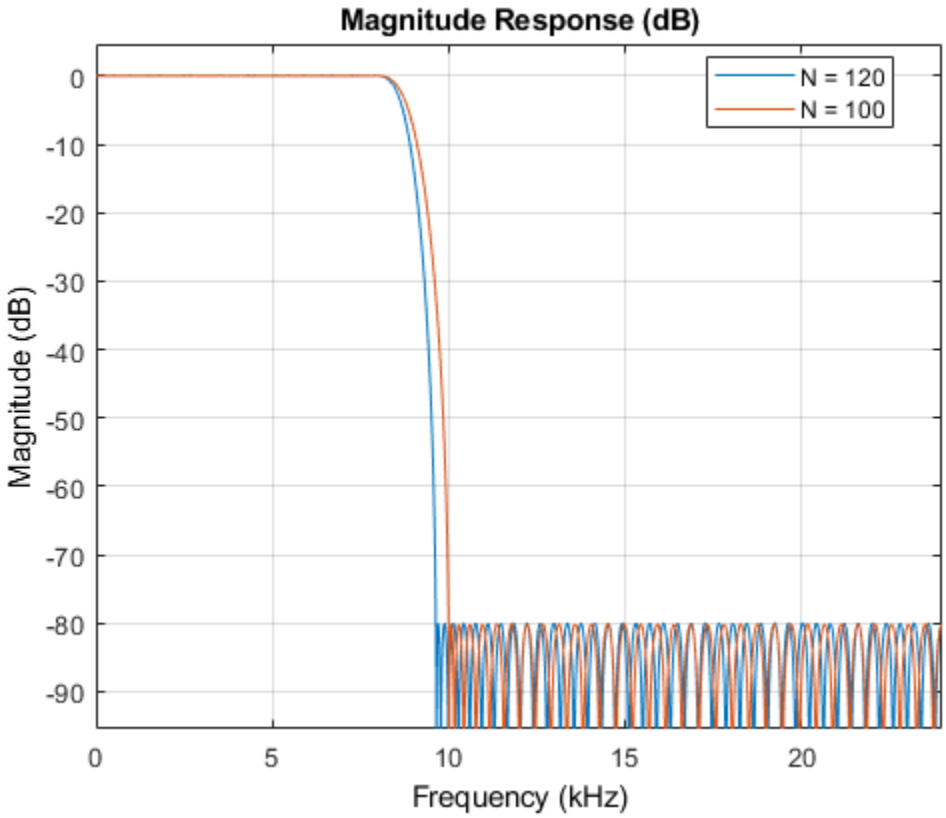
Another design function for optimal equiripple filters is `firgr`. `firgr` can design a filter that meets passband/stopband ripple constraints as well as a specified transition width with the smallest possible filter order. For example, if the stopband-edge frequency is specified as 10 kHz, the resulting 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')
```

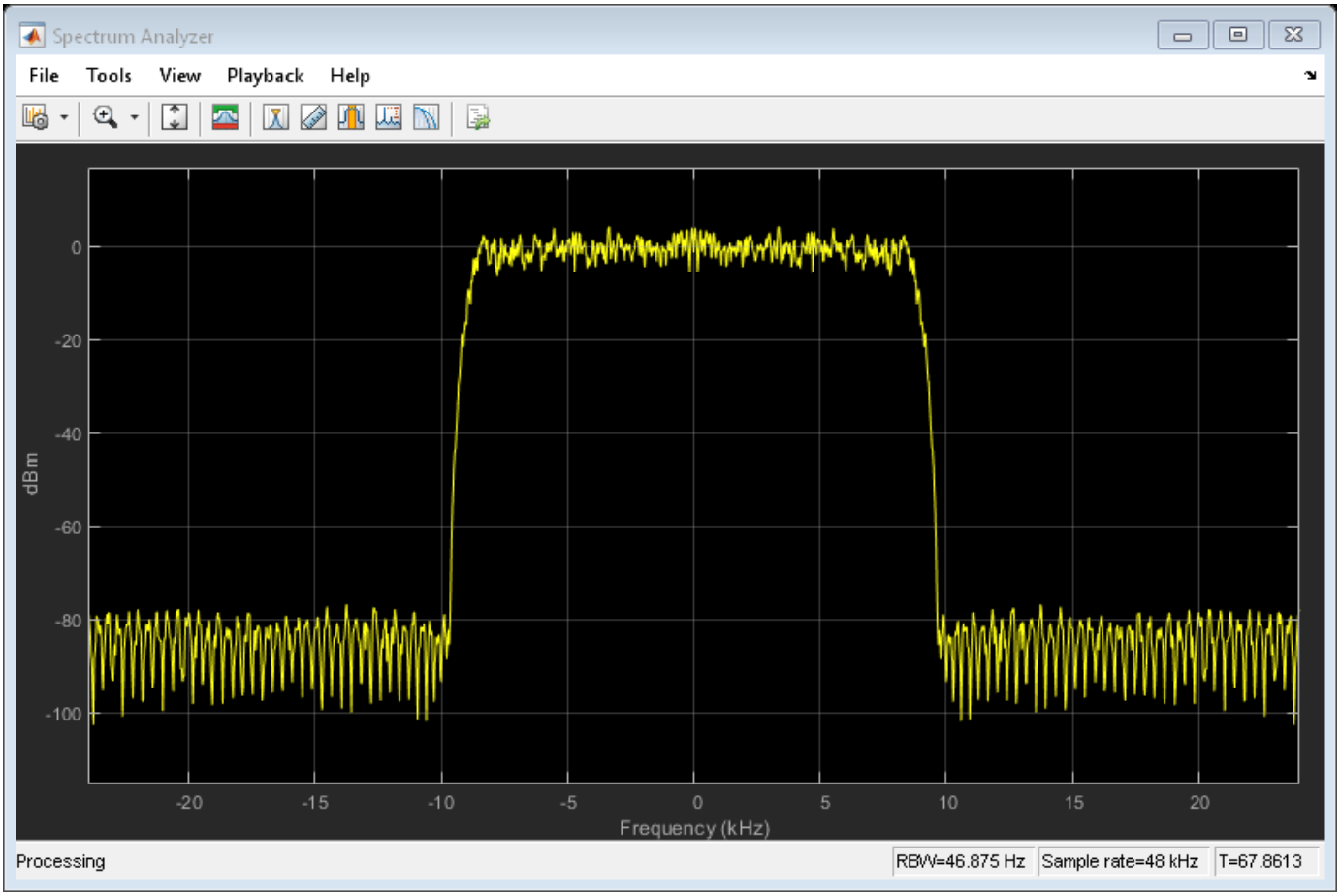


Filtering Data

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);
```

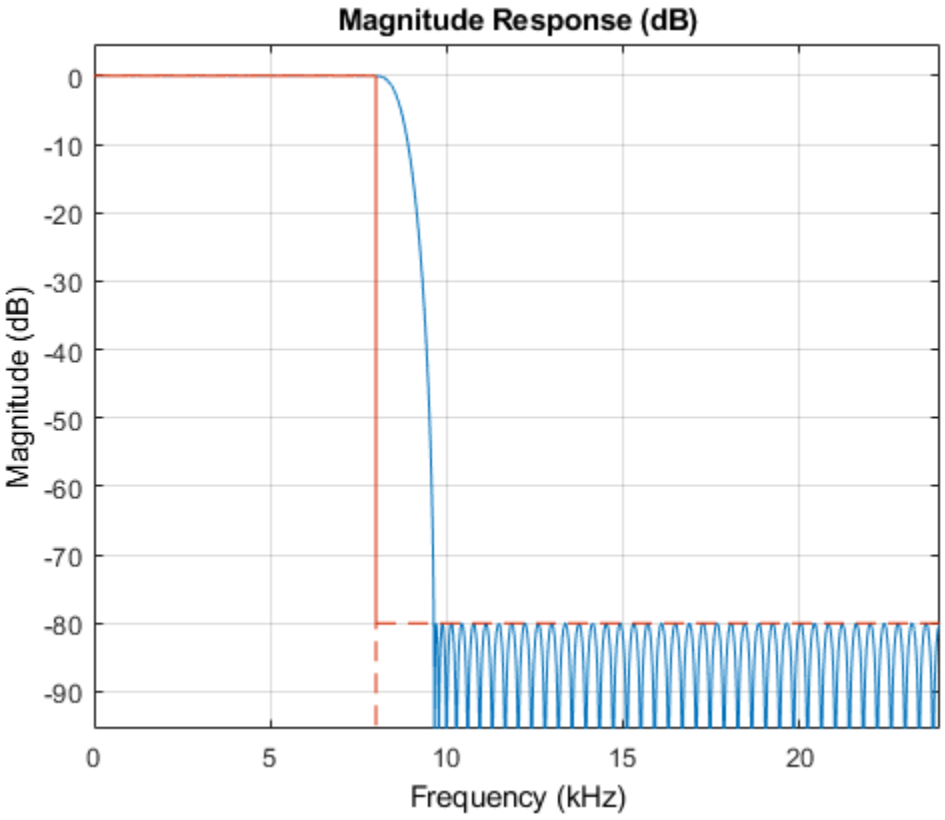
The coefficients in `LP_FIR` are identical to the coefficients in `NUM`.

```
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.

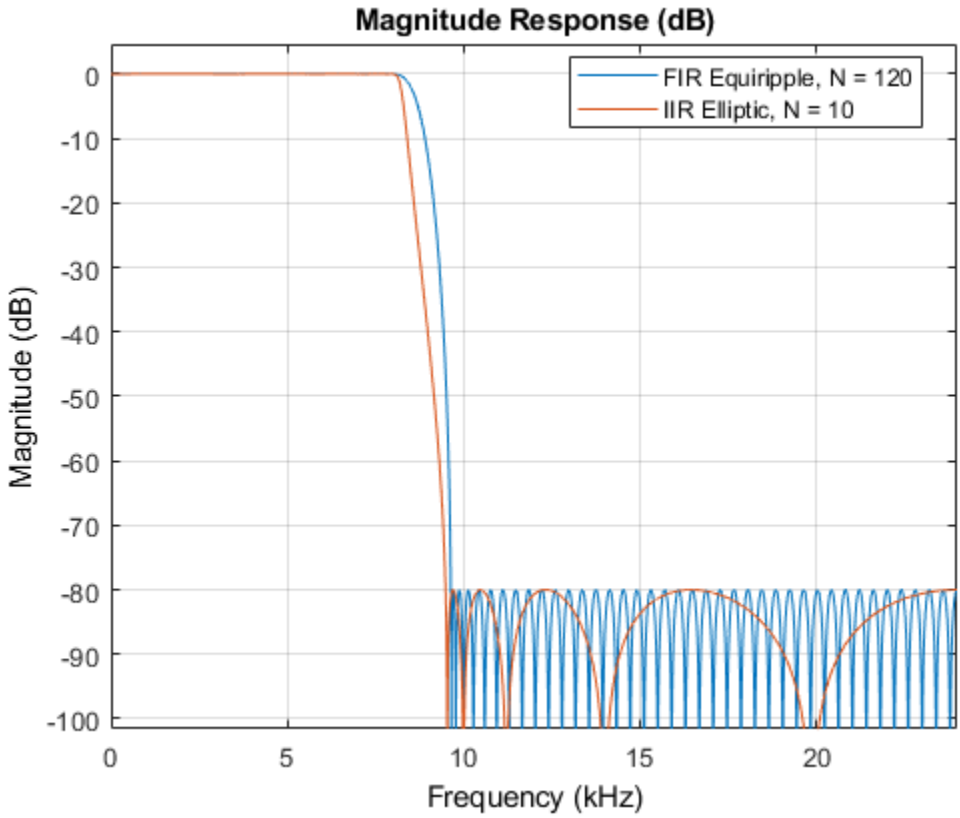
```
N = 10;
LP_IIR = dsp.LowpassFilter('SampleRate',Fs,...
    'FilterType','IIR',...
    'DesignForMinimumOrder',false,...
    'FilterOrder',N,...
    'PassbandFrequency',Fp,...
    'PassbandRipple',Ap,...
    'StopbandAttenuation',Ast);
```

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.

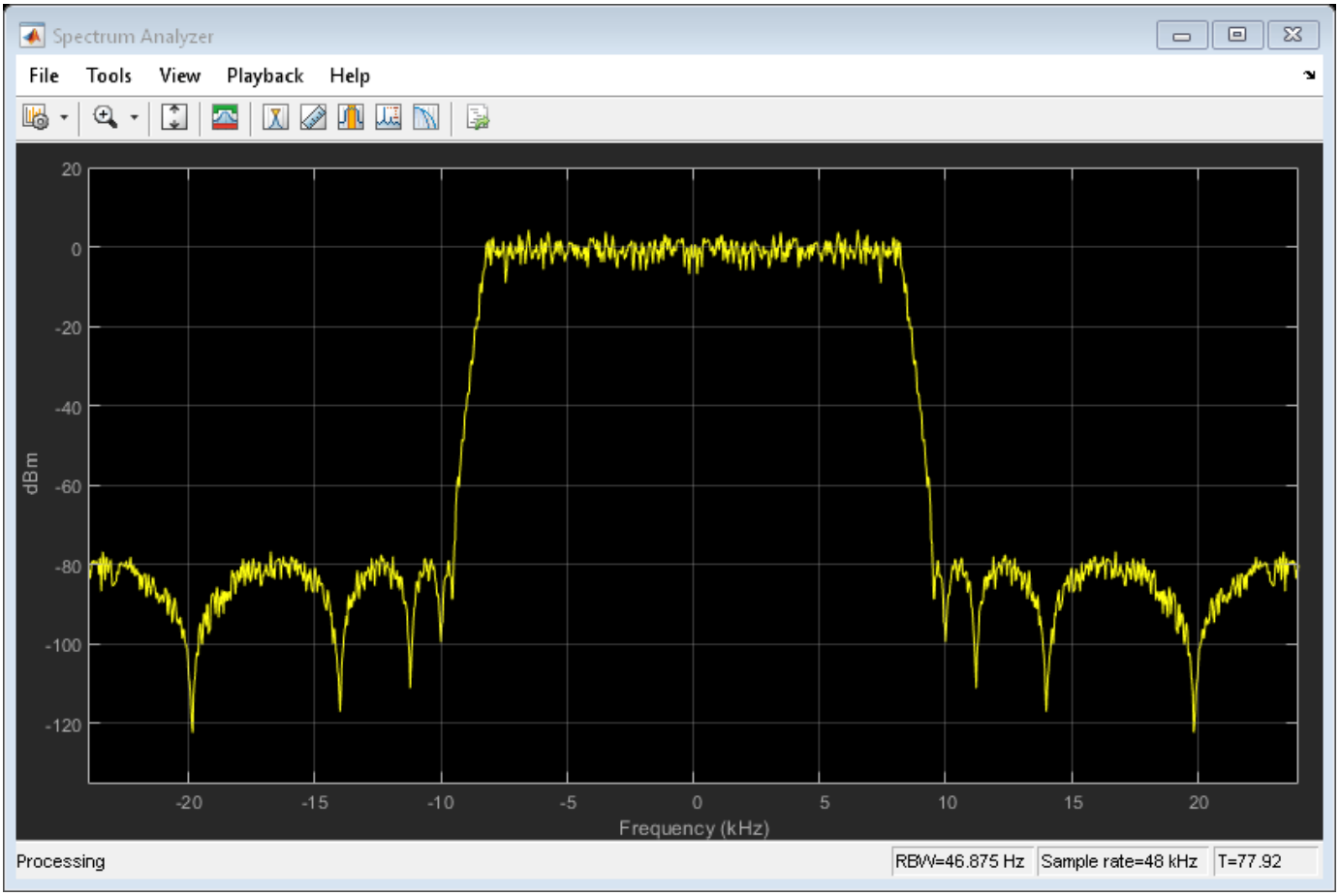
Running the IIR Filters

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 1-47
- “Multirate Filtering in MATLAB and Simulink” 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.

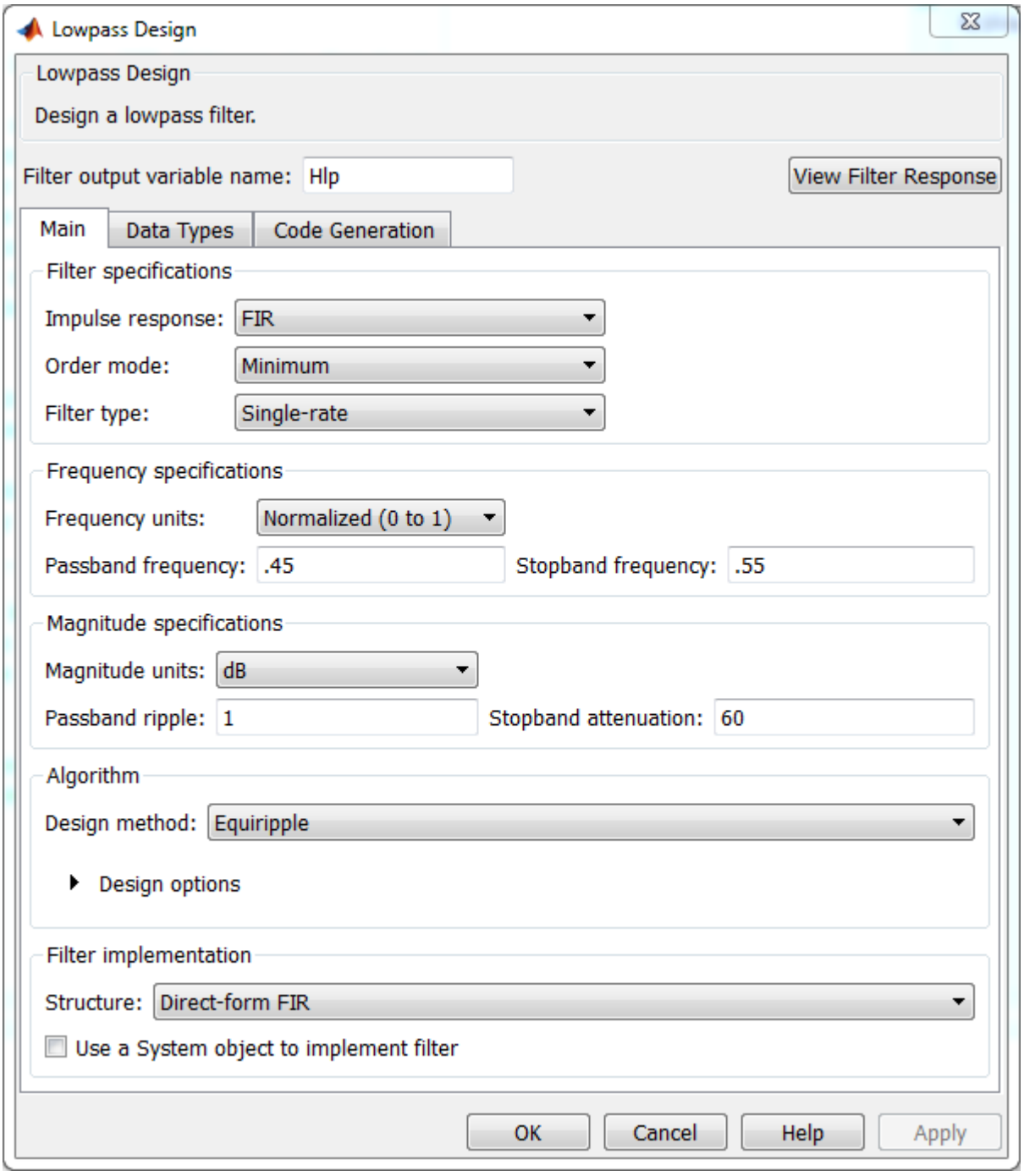
The example then shows how to design and use lowpass filters in Simulink using the interface available from the Lowpass Filter block..

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.

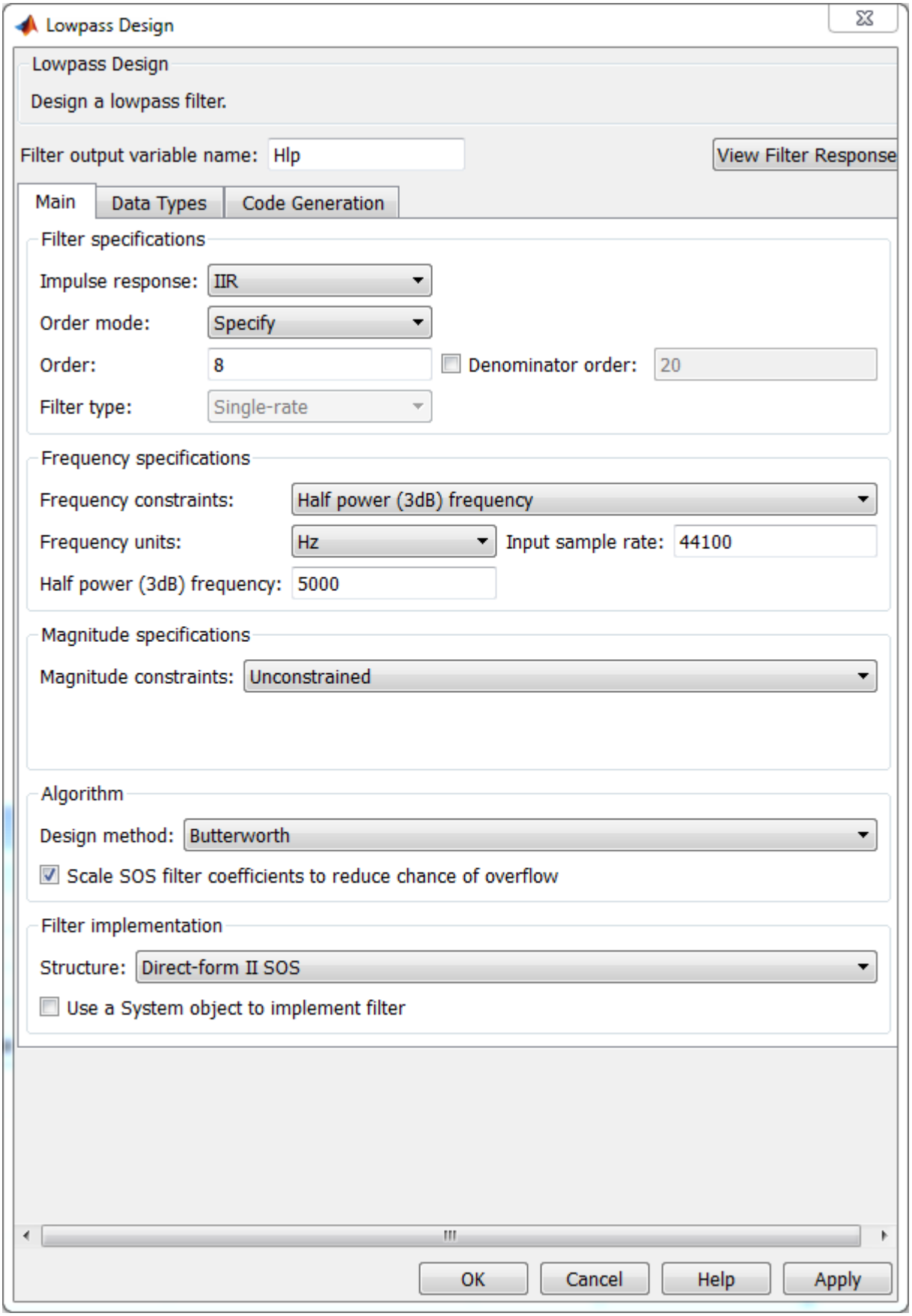


Butterworth Filter

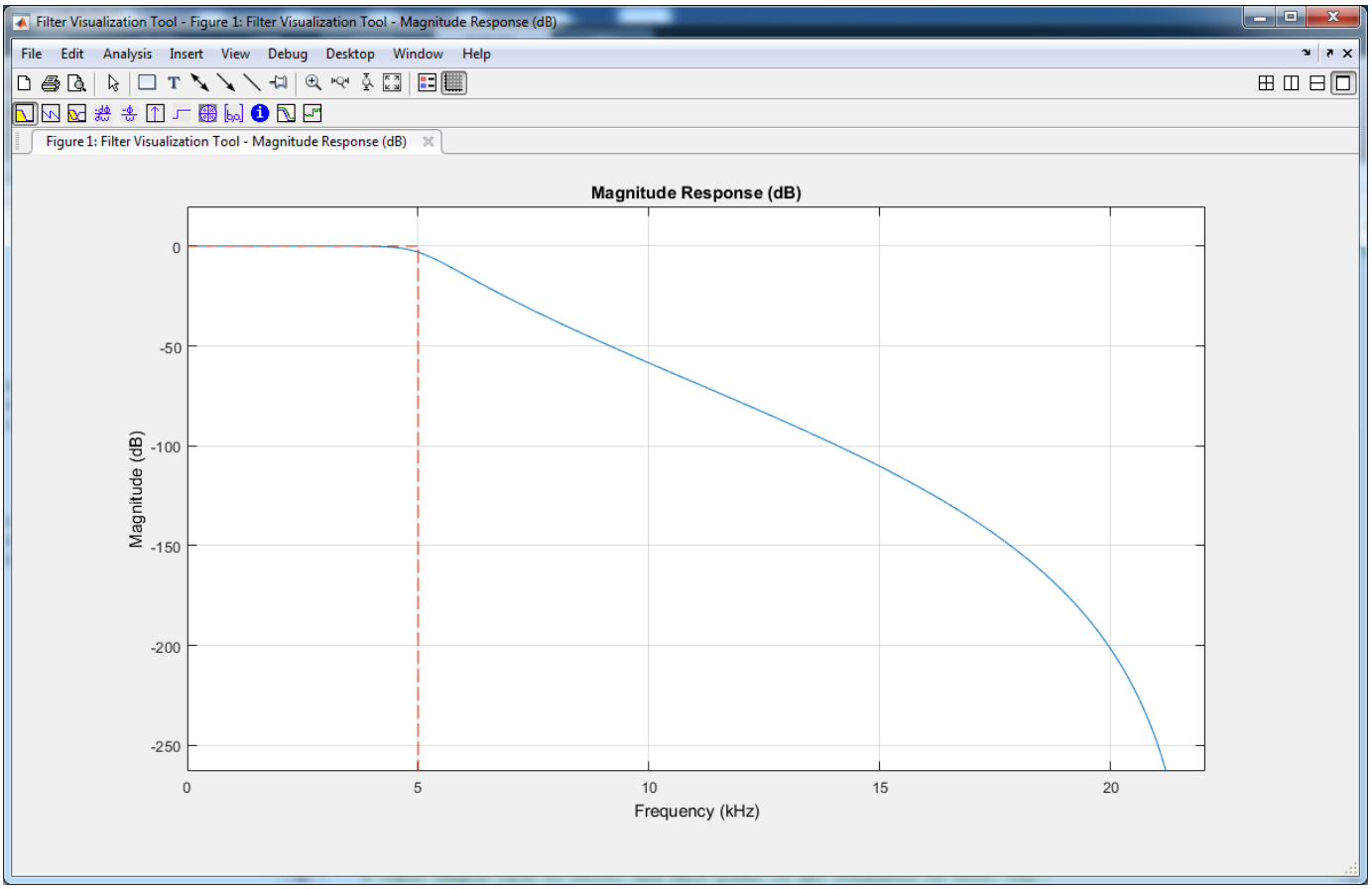
Design an eighth order Butterworth lowpass filter with a cutoff frequency of 5 kHz, assuming a sample rate of `44.1` KHz.

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

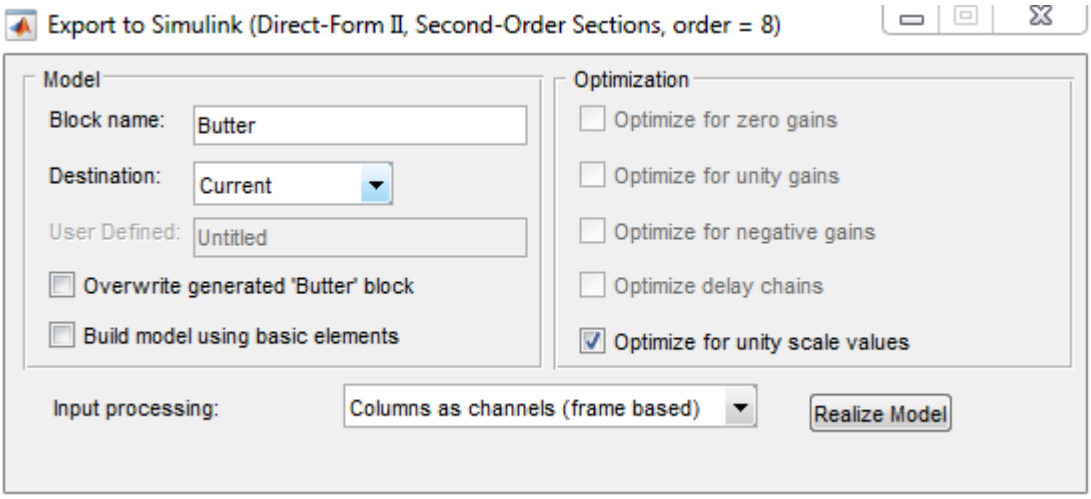
frequencies in Hz, set **Frequency units** to `Hz`, **Input sample rate** to `44100`, and **Half power (3 dB) frequency** to `5000`. Set the **Design method** to `Butterworth`.


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

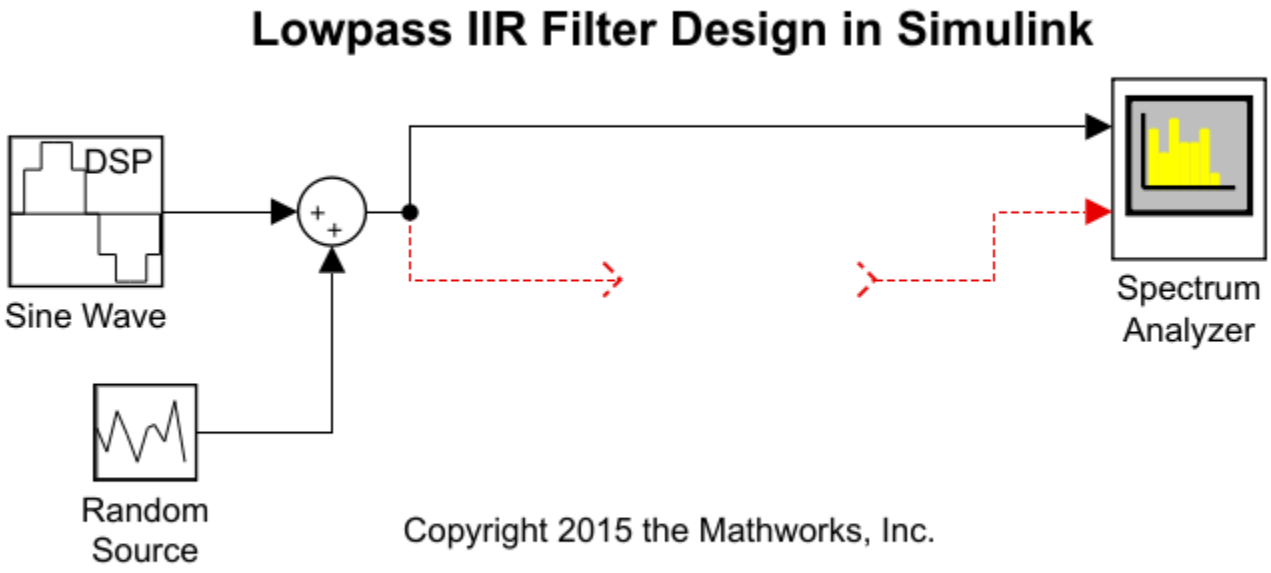
Click **Apply**. To visualize the filter's frequency response, click **View Filter Response**. The filter is maximally flat. There is no ripple in the passband or in the stopband. The filter response is within the specification mask (the red dotted line).



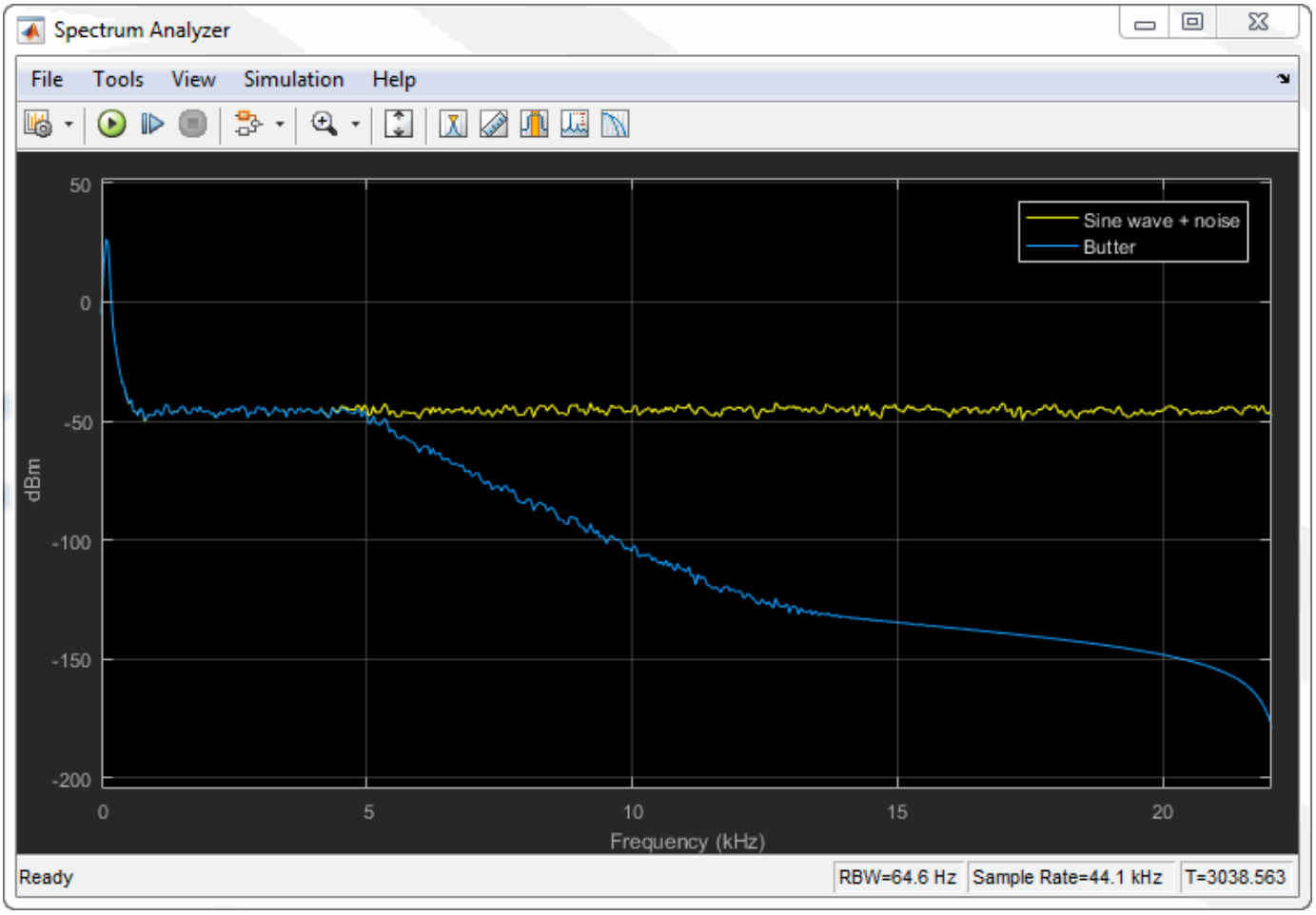
Generate a block from this design and use it in a model. Open the model `ex_iir_design`. In **Filter Builder**, on the **Code Generation** tab, click **Generate Model**. In the Export to Simulink window, specify the **Block name** as `Butter` and **Destination** as `Current`. You can also choose to build 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

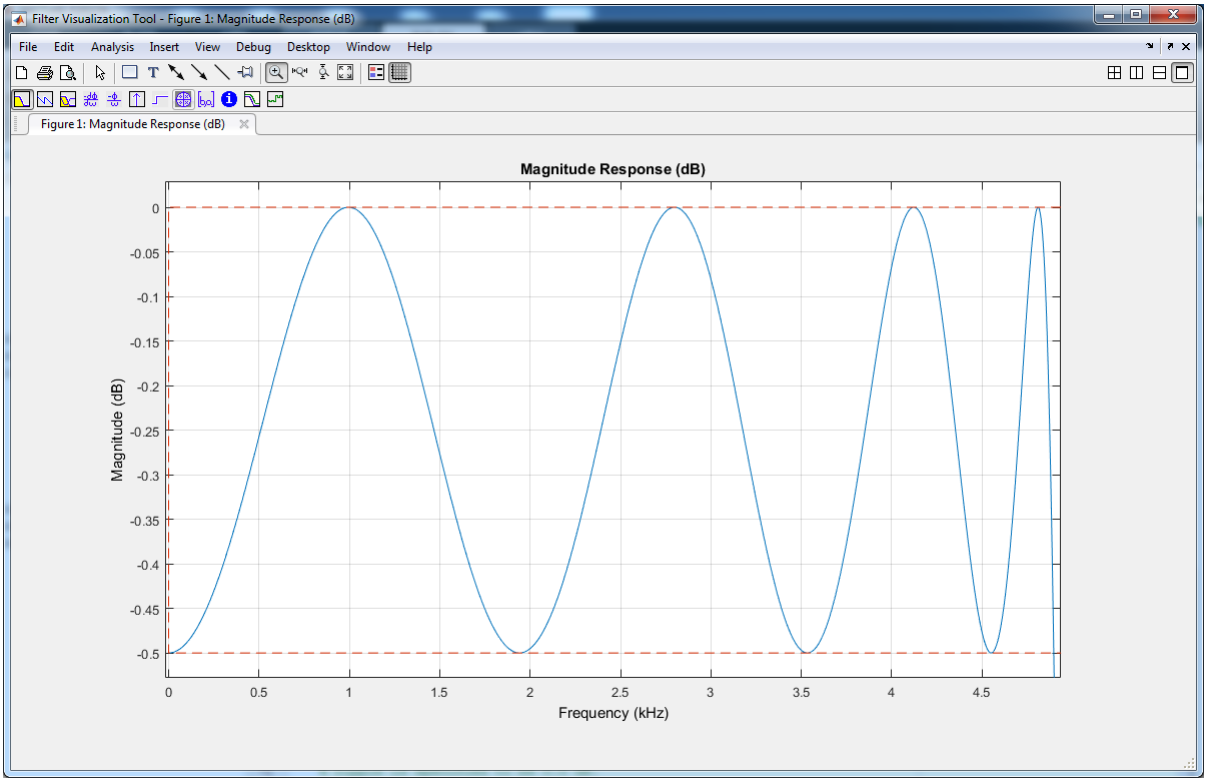
Now design a Chebyshev Type I filter. A Chebyshev type I design allows you to control the passband. There are still no ripples in the stopband. Larger ripples enable a steeper roll-off. In this model, the peak-to-peak ripple is specified as 0.5 dB.

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**.

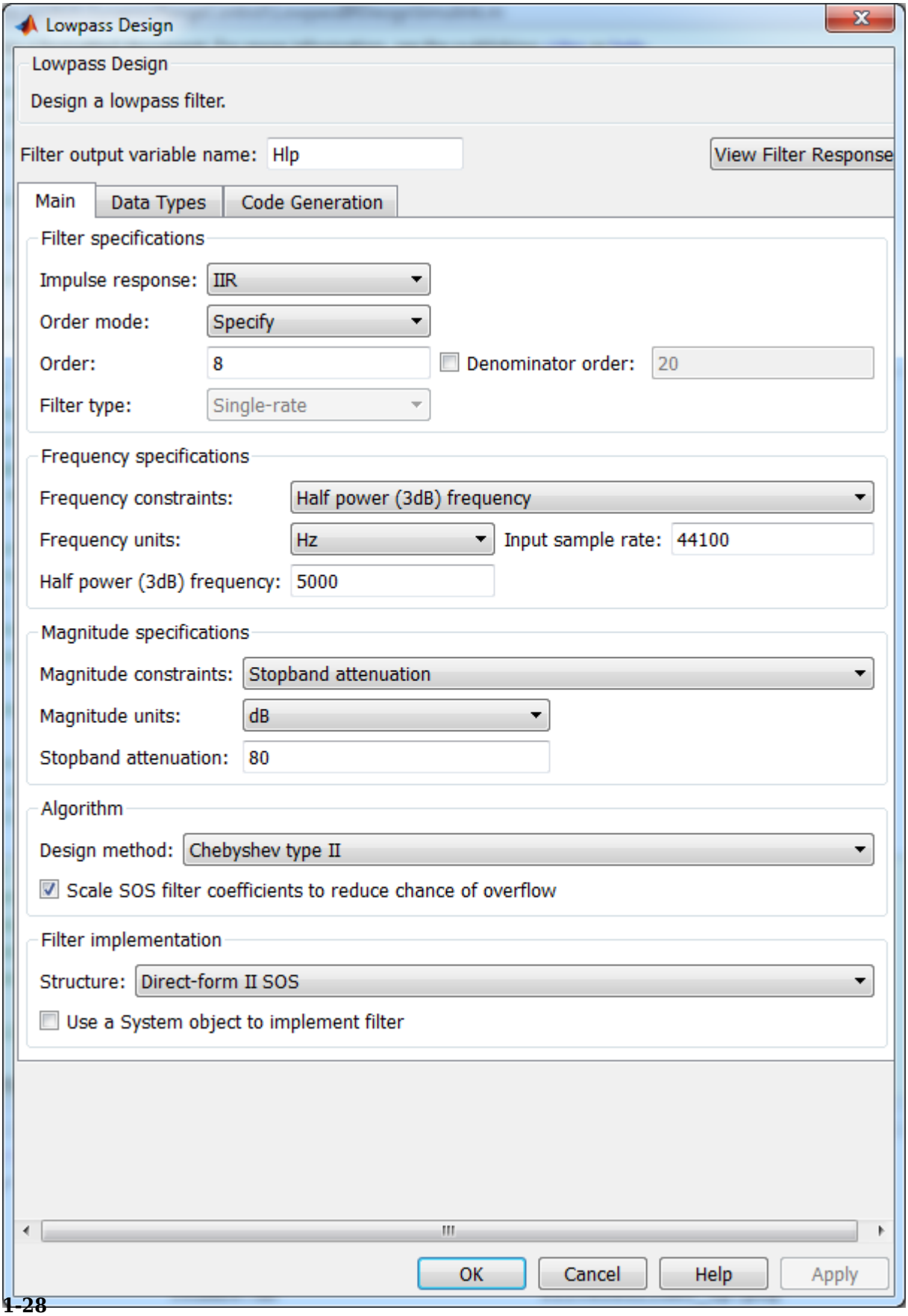
Zooming in on the passband, you can see that the ripples are contained in the range [-0.5, 0] dB.



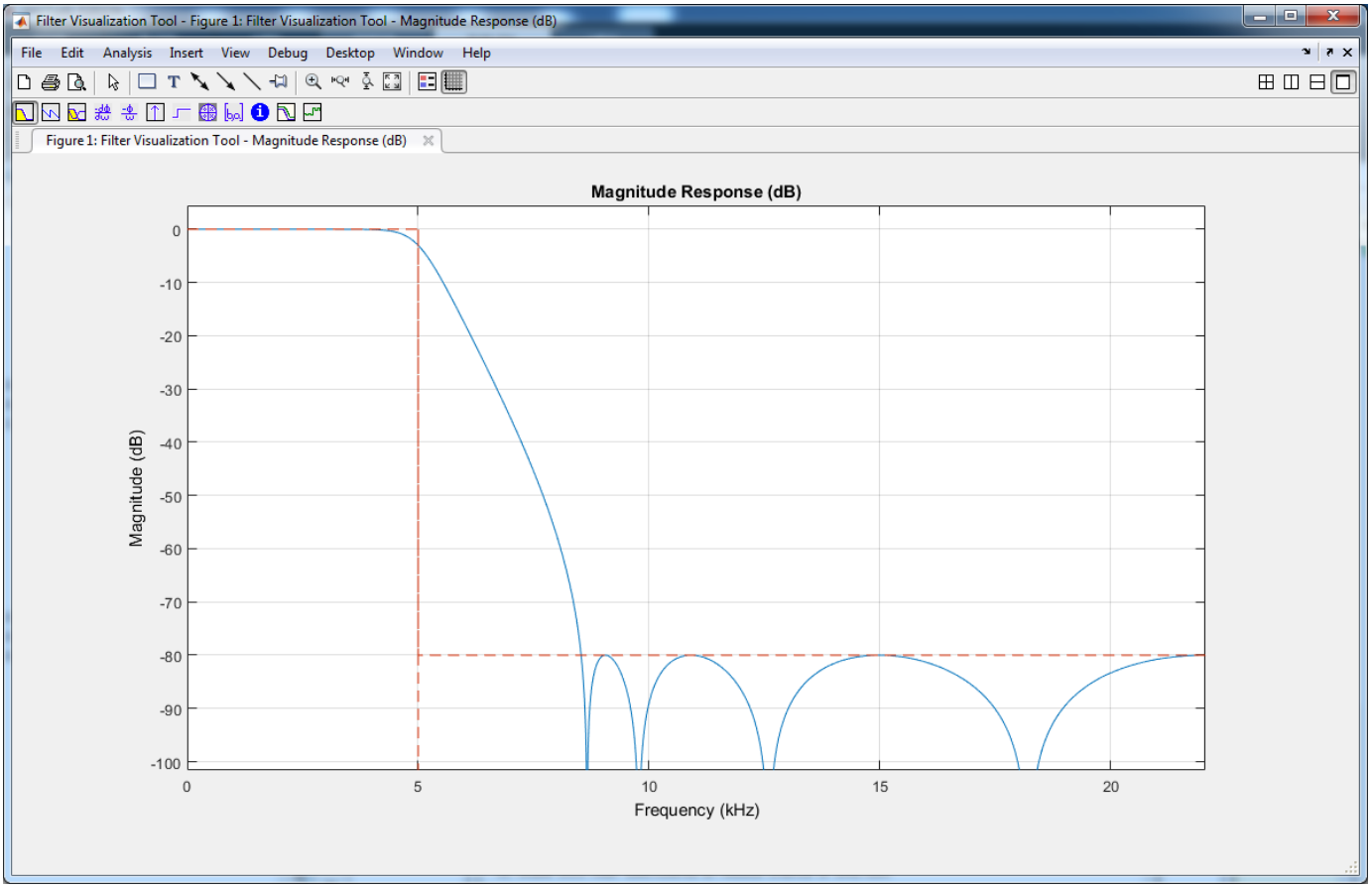
Similar to the Butterworth filter, you can generate a block from this design by clicking **Generate Model** on the **Code Generation** tab, and then clicking **Realize model**.

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

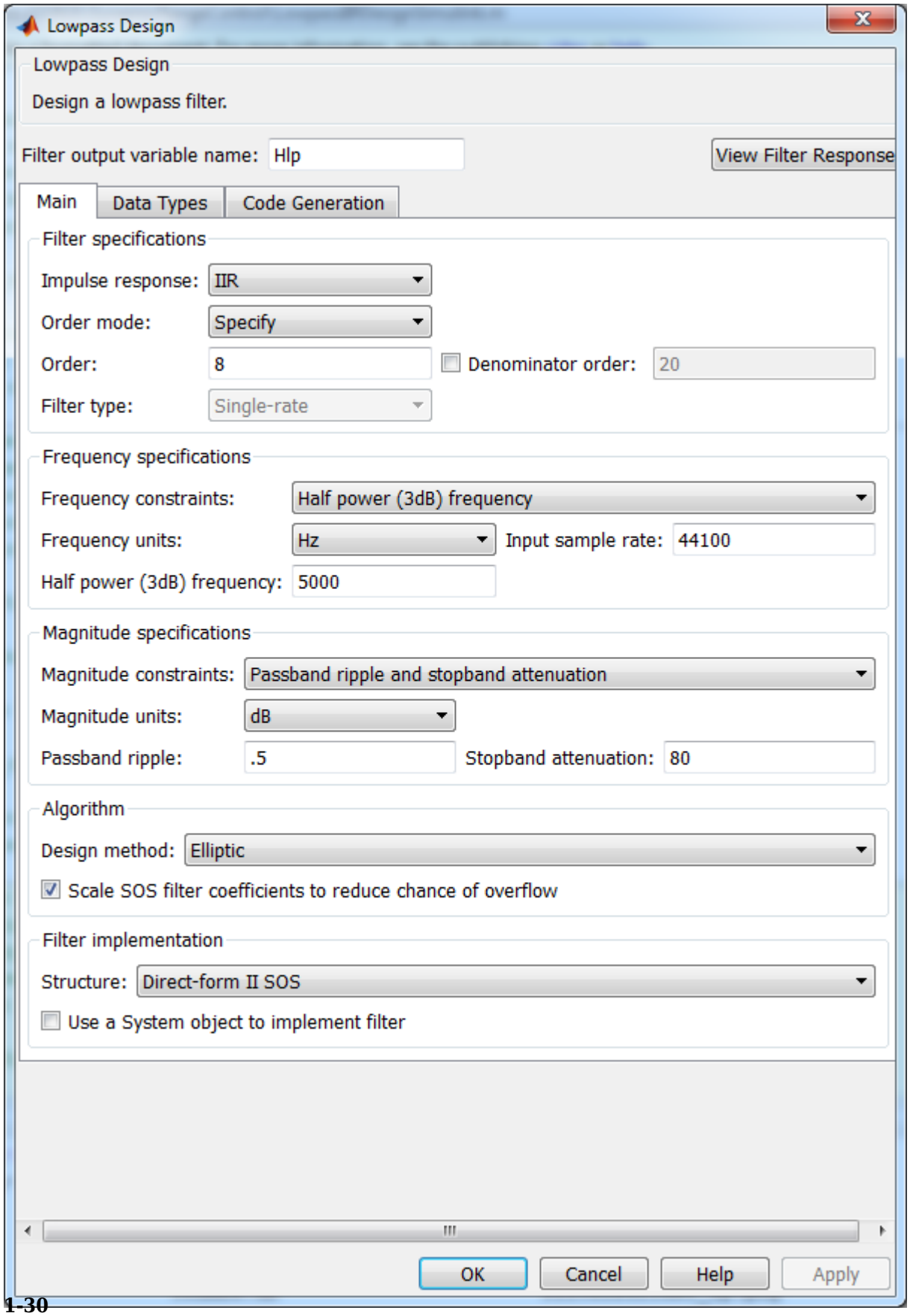
Click **View Filter Response**.

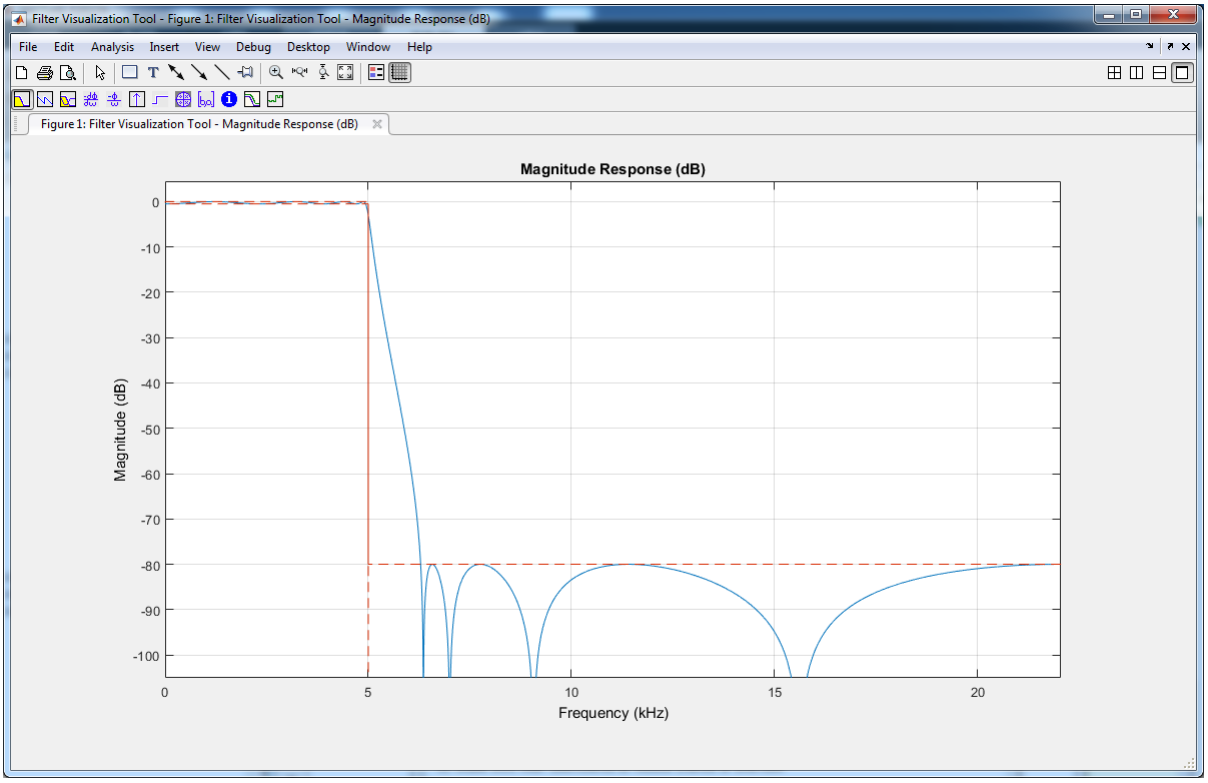


To generate a block from this design, on the **Code Generation** tab, click **Generate Model**, and then click **Realize model**.

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**.

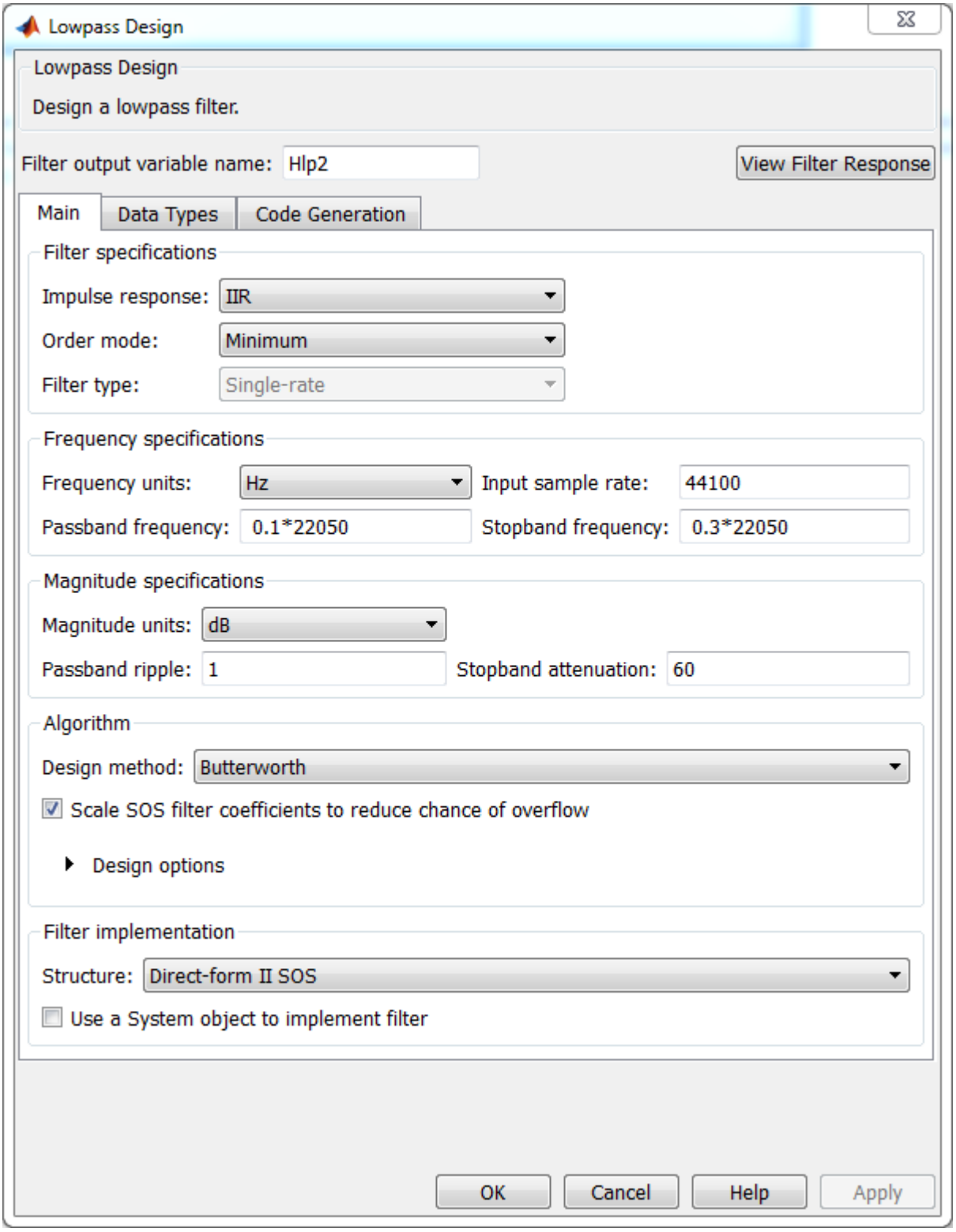

Lowpass Design
Lowpass Design
Design a lowpass filter.
Filter output variable name: Hlp
View Filter Response
Main
Data Types
Code Generation
Filter specifications
Impulse response: IIR
Order mode: Specify
Order: 8
Denominator order: 20
Filter type: Single-rate
Frequency specifications
Frequency constraints: 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



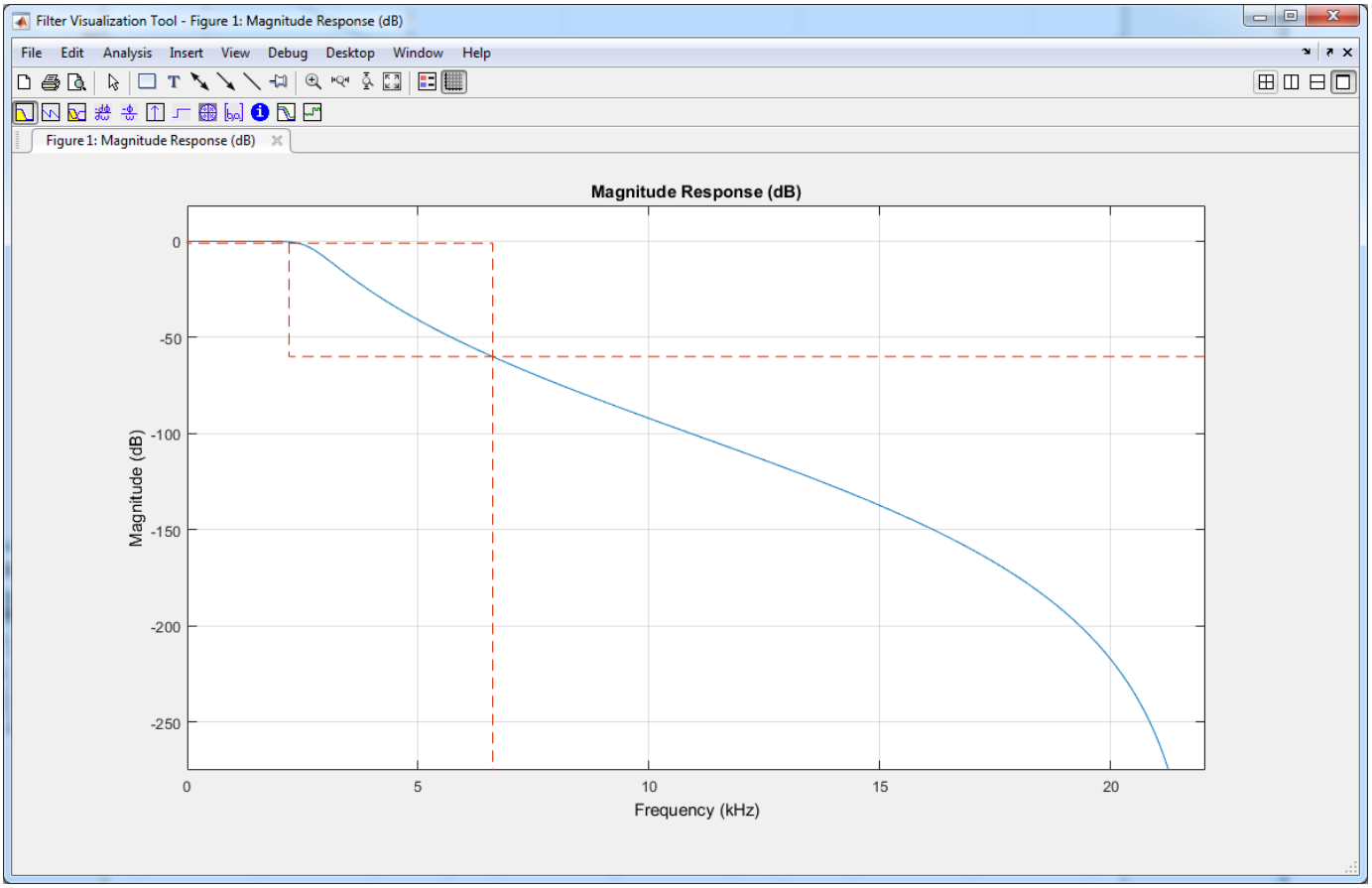
To generate a block from this design, on the **Code Generation** tab, click **Generate Model**, and then click **Realize model**.

Minimum-Order Designs

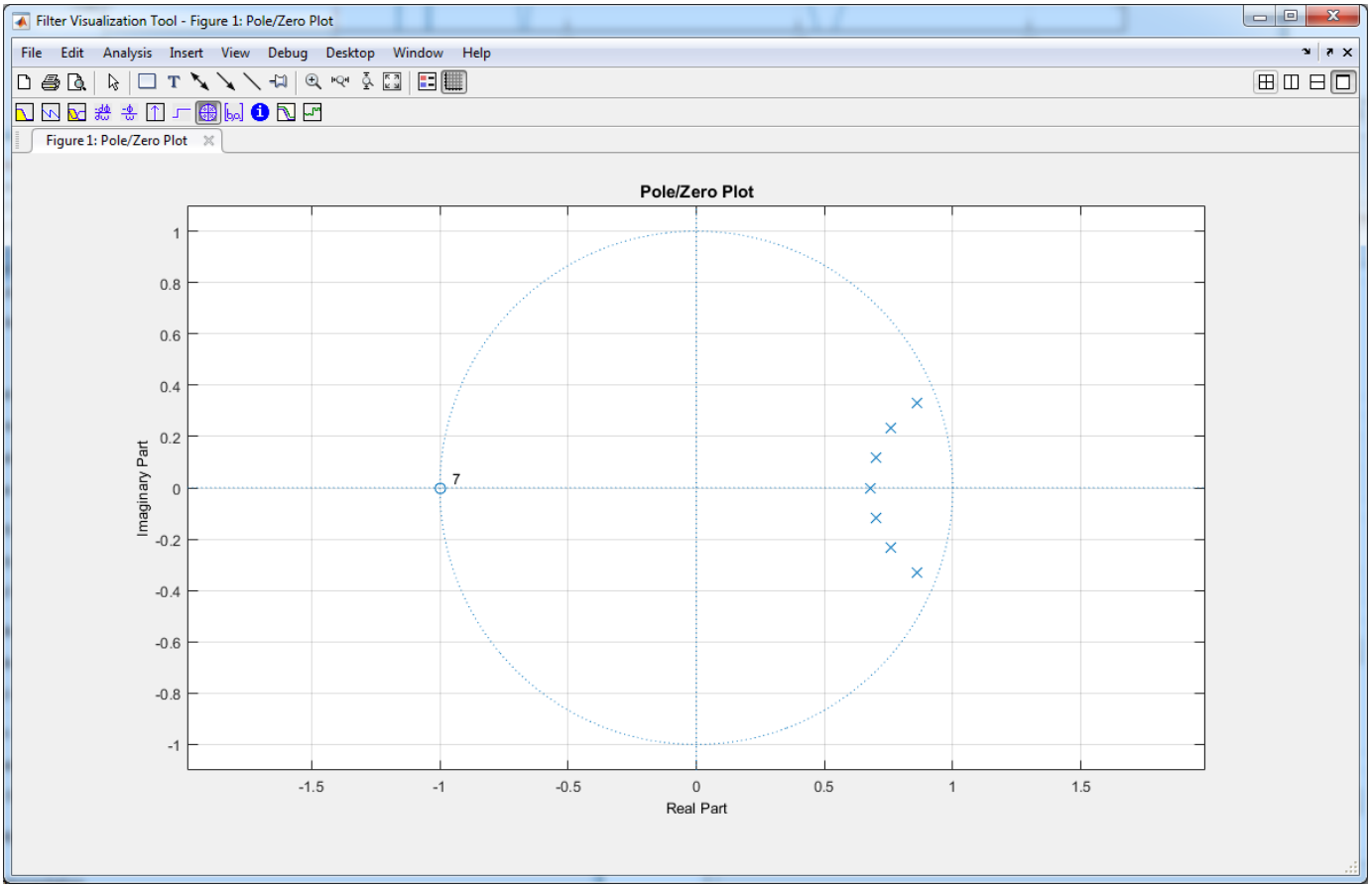
To specify the passband and stopband in terms of frequencies and the amount of tolerable ripple, use a minimum-order design. As an example, verify that the **Order mode** of the Butterworth filter is set to `Minimum`, and set **Design method** to `Butterworth`. Set the passband and stopband frequencies 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.

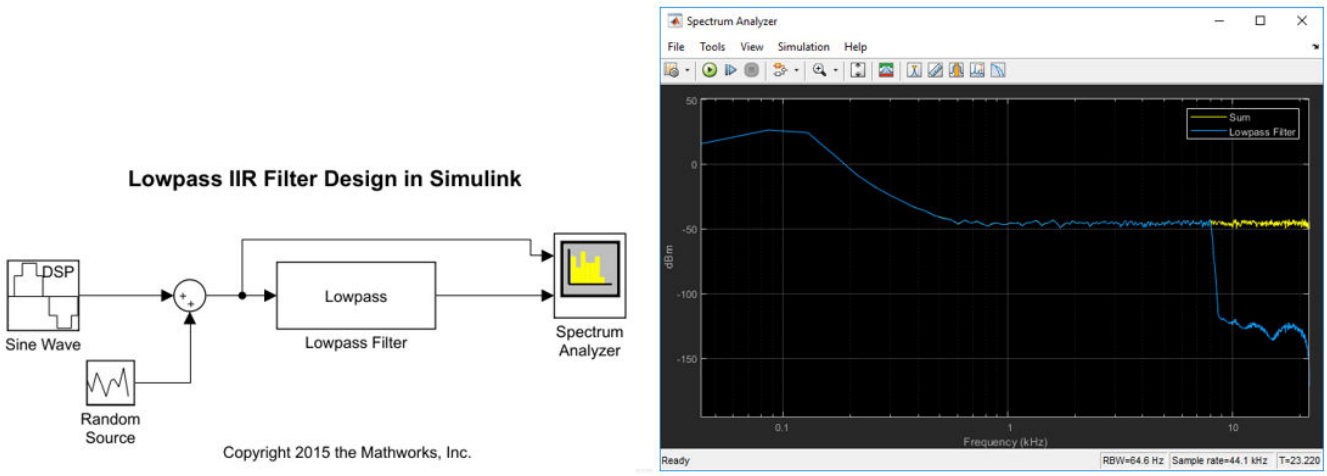


Lowpass Filter Block

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 1-47 example for a model that uses this block.

See Also

Related Examples

- “Tunable Lowpass Filtering of Noisy Input in Simulink” on page 1-47
- “Lowpass Filter Design in MATLAB” on page 1-12
- “Multirate Filtering in MATLAB and Simulink” on page 1-36

Multirate Filtering in MATLAB and Simulink

Multirate filters are digital filters that change the sample rate of an sampled input signal. The process of rate conversion involves an upsampler, a downsampler, and a lowpass filter to process the signal.

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

The `designMultirateFIR` function automatically designs an anti-aliasing FIR filter based on the rate conversion factor that you specify. The inputs to the `designMultirateFIR` function are the interpolation factor and the decimation factor. Optionally, you can provide the half-polyphase length or transition width and stopband attenuation. To design a decimator, set the interpolation factor to 1. Similarly, to design an interpolator, set the decimation factor to 1.

To implement the multirate filters in MATLAB, use the coefficients returned by the `designMultirateFIR` function as inputs to the `dsp.FIRDecimator`, `dsp.FIRInterpolator`, and `dsp.FIRRateConverter` System objects.

```
b = designMultirateFIR(1,4);
firDecim = dsp.FIRDecimator(4,b)
```

```
firDecim =

  dsp.FIRDecimator with properties:

    DecimationFactor: 4
     NumeratorSource: 'Property'
           Numerator: [0 -2.2355e-05 -5.0269e-05 -5.2794e-05 0 1.0256e-04 1.9352e-04 … ]
           Structure: 'Direct form'
```

Alternatively, you can set the `SystemObject` flag of the `designMultirateFIR` function to `true`. The function designs and automatically creates the appropriate rate conversion object.

```
firDecim = designMultirateFIR(1,4,'SystemObject',true)
```

```
firDecim =

  dsp.FIRDecimator with properties:

    DecimationFactor: 4
     NumeratorSource: 'Property'
           Numerator: [0 -2.2355e-05 -5.0269e-05 -5.2794e-05 0 1.0256e-04 1.9352e-04 … ]
           Structure: 'Direct form'
```

In Simulink, compute these coefficients using the `designMultirateFIR` function in the default **Auto** mode of the FIR Decimation, FIR Interpolation, and FIR Rate Conversion blocks. You can also specify these coefficients as parameters or pass them through an input port.

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

Implement an FIR Decimator in MATLAB

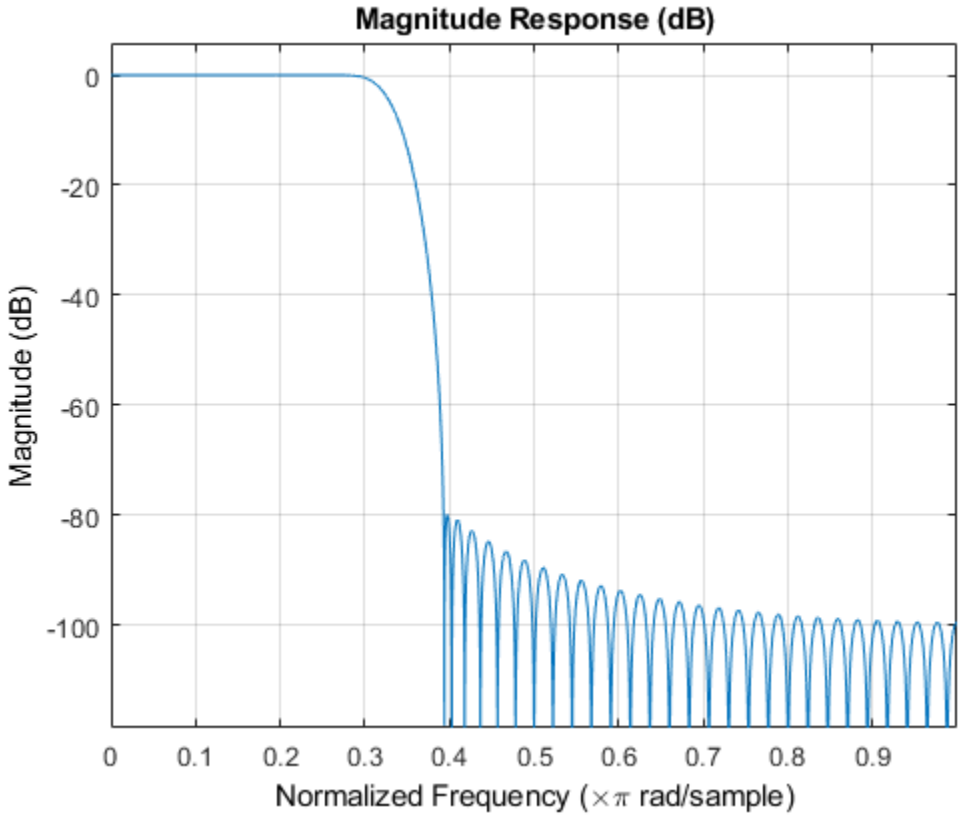
To implement an FIR Decimator, you must first design it by using the `designMultirateFIR` function. Specify the decimation factor of interest (usually greater than 1) and an interpolation factor equal to 1. You can use the default half-polyphase length of 12 and the default stopband attenuation of 80 dB. Alternatively, you can also specify the half-polyphase length, transition width, 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` function shows the magnitude response. Navigate through the Filter Visualization Tool 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. The noise is a white Gaussian noise with zero mean and a standard deviation of 1e-5. The decimated output will have one-third the sample rate as the input. Initialize two `dsp.SpectrumAnalyzer` System objects, one for the input and the other for the output.

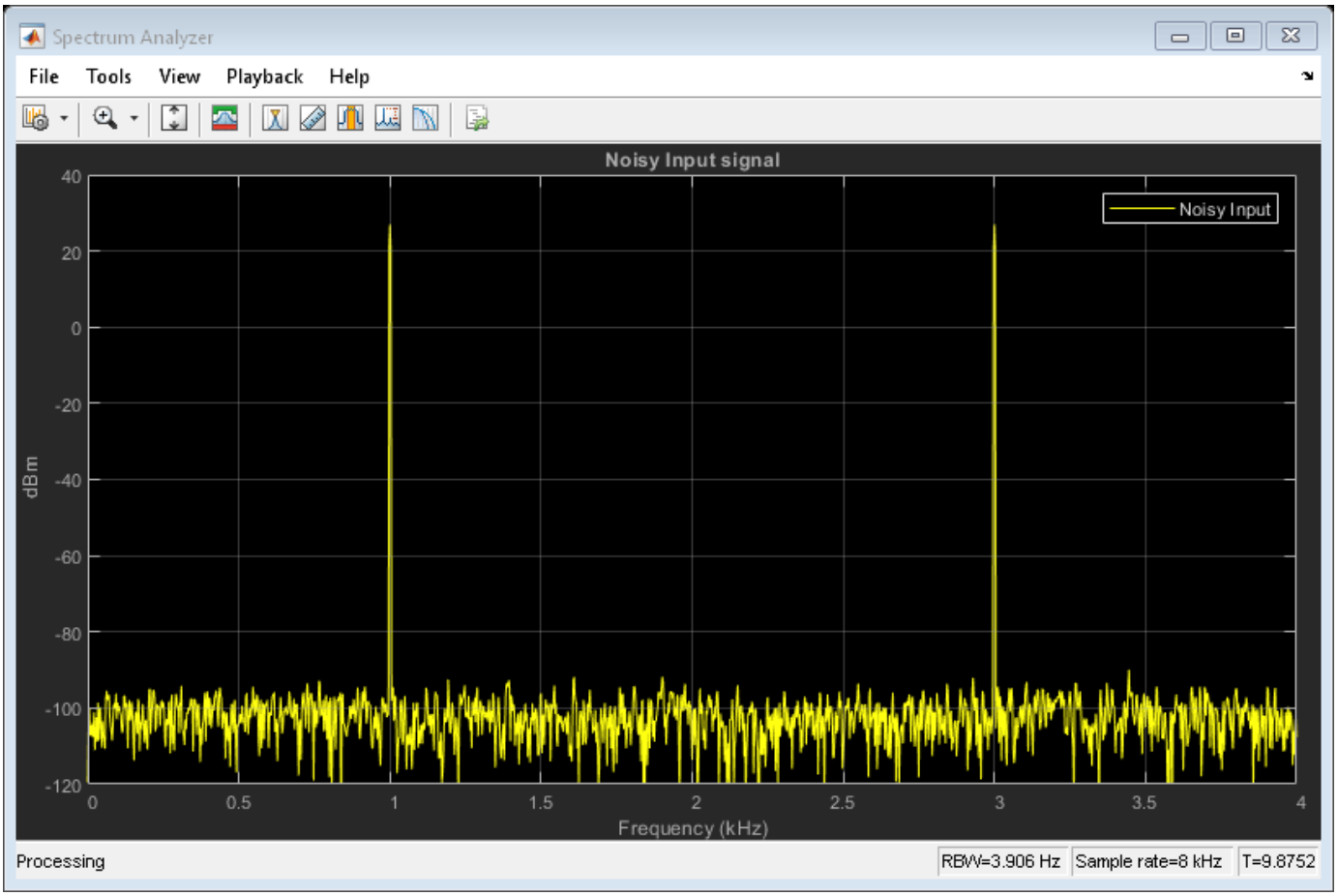
```
f1 = 1000;
f2 = 3000;
Fs = 8000;
source = dsp.SineWave('Frequency',[f1,f2],'SampleRate',Fs,...
    'SamplesPerFrame',1026);

specanainput = dsp.SpectrumAnalyzer('SampleRate',Fs,...
    'PlotAsTwoSidedSpectrum',false,...
```

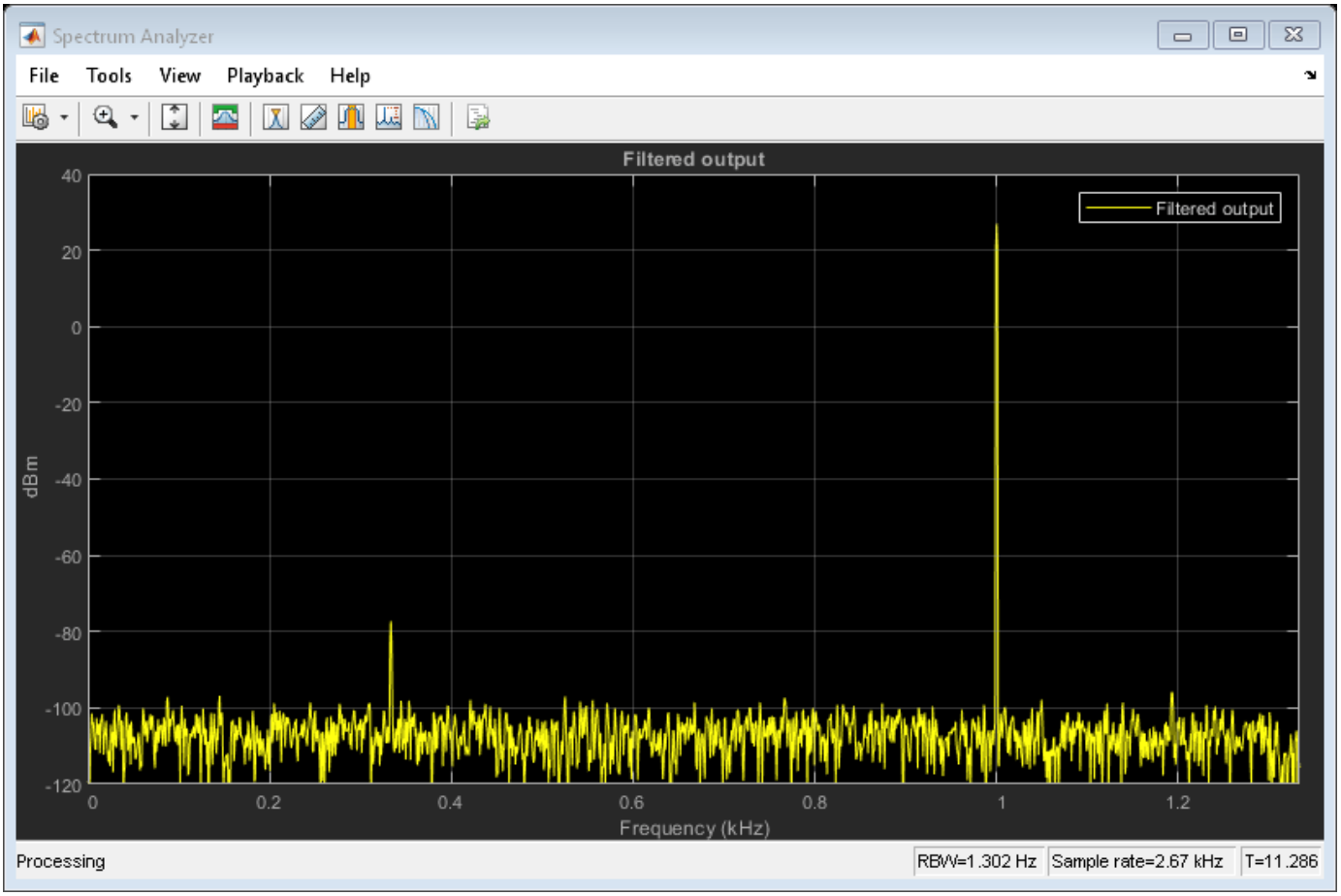
```
    'ShowLegend',true,'YLimits',[-120 40],...
    'Title','Noisy Input signal',...
    'ChannelNames', {'Noisy Input'});
specanaoutput = dsp.SpectrumAnalyzer('SampleRate',Fs/3,...
    'PlotAsTwoSidedSpectrum',false,...
    'ShowLegend',true,'YLimits',[-120 40],...
    'Title','Filtered output',...
    'ChannelNames', {'Filtered output'});
```

Stream the input and filter it in a processing loop.

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)`.

```
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 0.3π rad/sample. With a sampling frequency of 8000 Hz, that value 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.

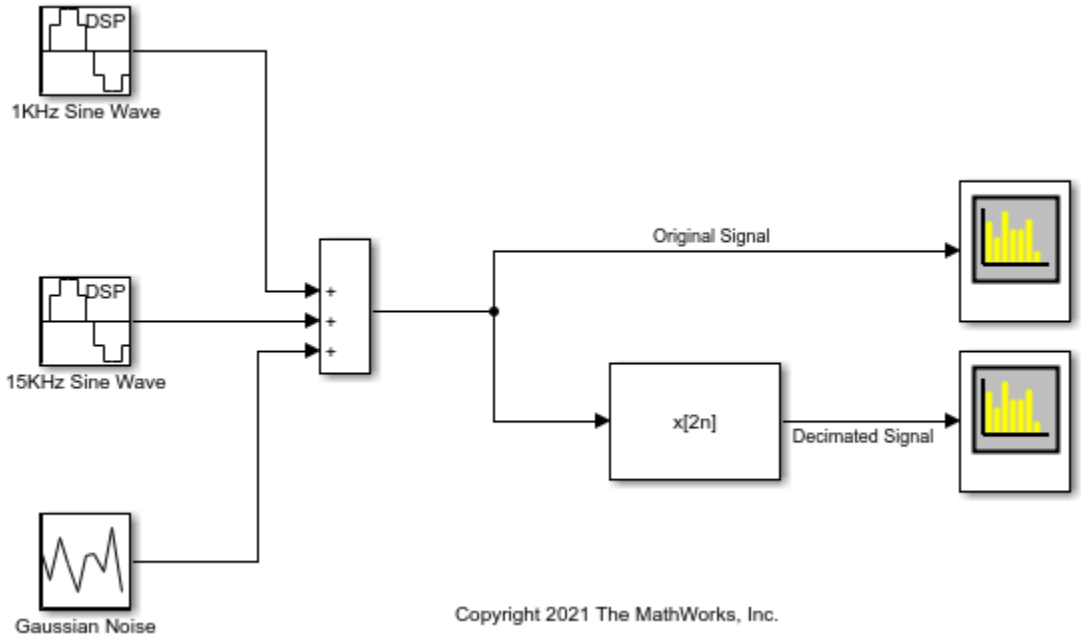
Similarly, you can design an FIR interpolator and an FIR rate Converter by providing appropriate inputs to the `designMultirateFIR` function. To implement the filters, pass the designed coefficients to the `dsp.FIRInterpolator` and `dsp.FIRRateConverter` objects.

Implement an FIR Decimator in Simulink

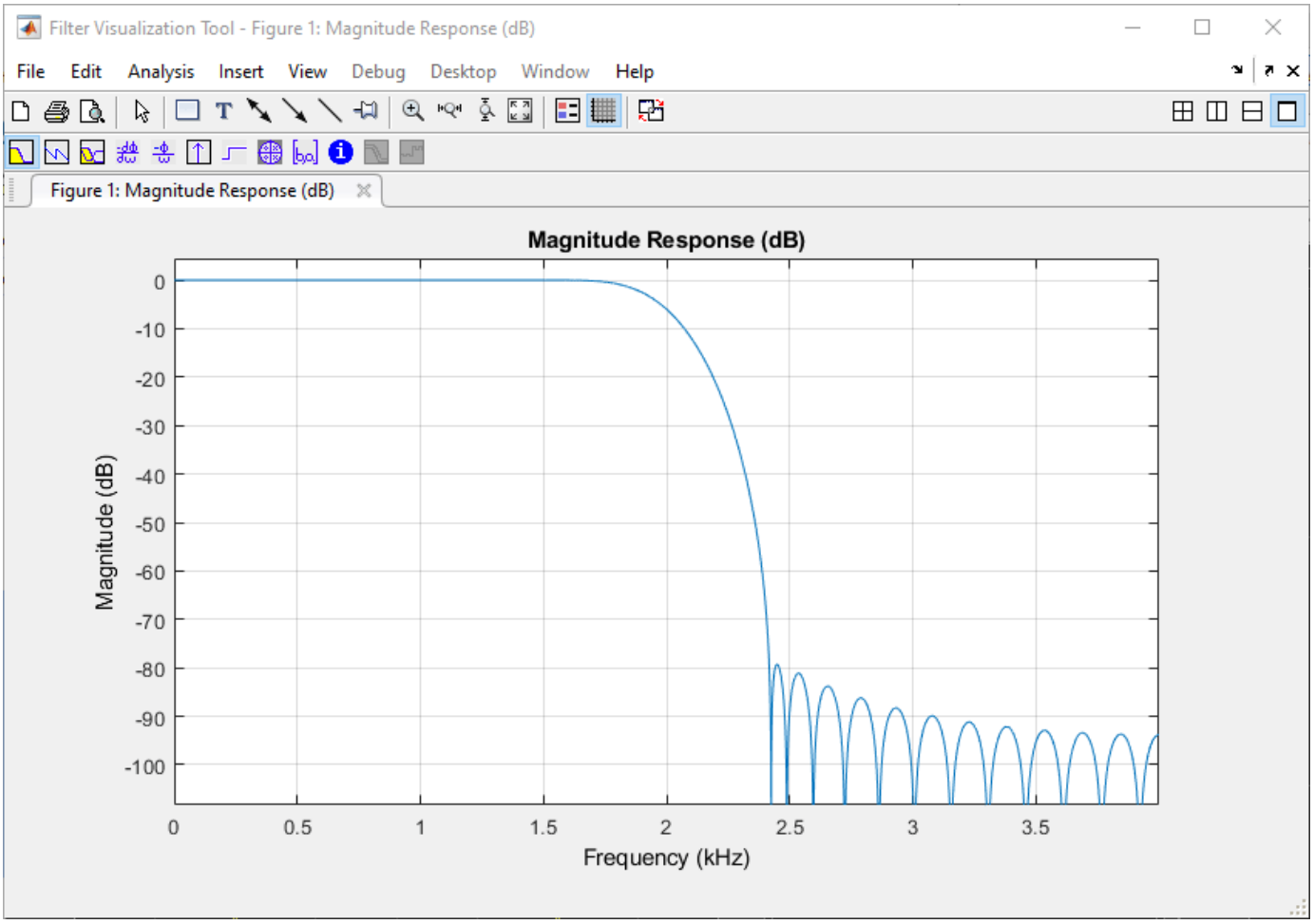
You can design and implement the FIR multirate filters in Simulink™ using the FIR Decimation, FIR Interpolation, and FIR Rate Conversion blocks.

Open the model `'multiratefiltering.slx'`.

The input signal is a noisy sinusoidal signal with two frequencies: one at 1000 Hz and the other at 3000 Hz. The sample rate of the signal is 8000 Hz and the **Samples per frame** parameter of the Sine Wave blocks is set to 1026. The noise added to the signal is a white Gaussian noise with zero mean and a variance of 1e-10.

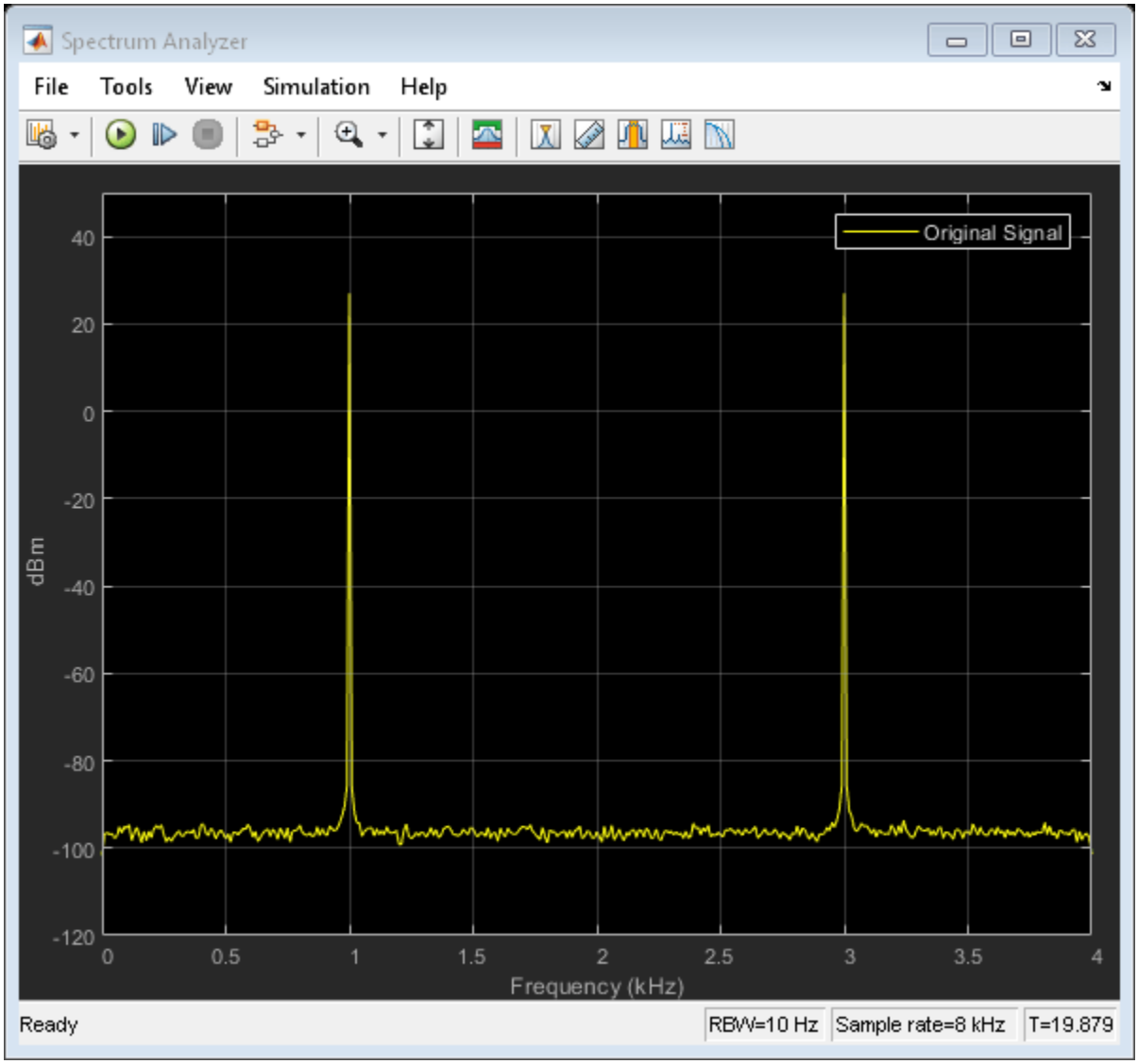


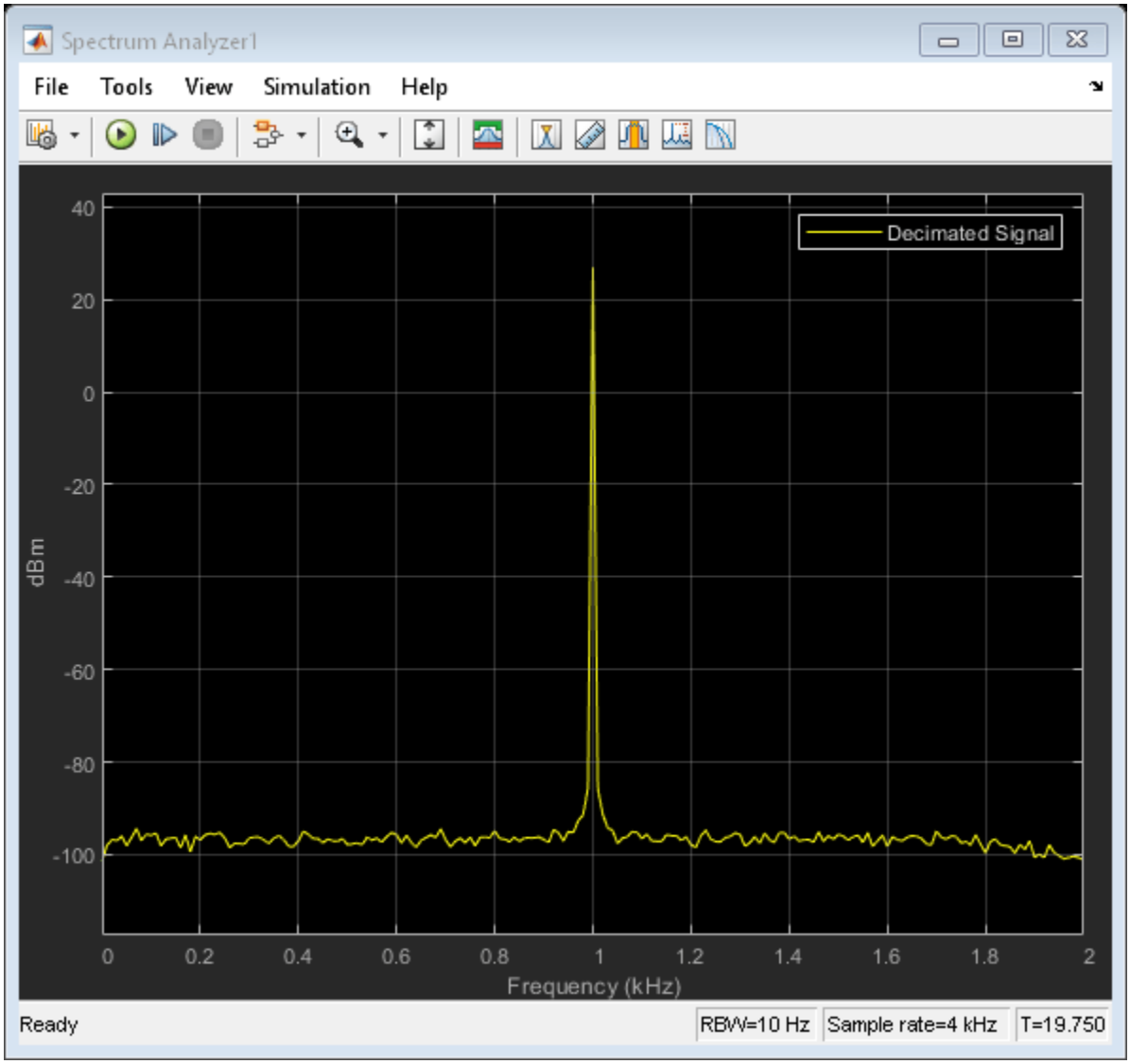
The **Coefficient source** parameter of the FIR Decimation block is set to **Dialog parameters**, and the **FIR filter coefficients** parameter is set to `designMultirateFIR(1,2)`. The function uses the default half-polyphase length of 12 and the default stopband attenuation of 80 dB. The magnitude response of the designed filter looks like as follows:



Run the model.

The first spectrum analyzer shows the spectrum of the original signal, while the second spectrum analyzer shows the spectrum of the decimated signal. Because the **Rate options** parameter in the FIR Decimation block is set to `Allow multirate processing`, the frame size of the signal is the same at the input and output of the FIR Decimation block, while the sample rate changes. For more details on this mode, see “Rate Conversion by Frame-Rate Adjustment” on page 3-17.


Spectrum Analyzer
File
Tools
View
Simulation
Help
Original Signal
40
20
0
-20
-40
-60
-80
-100
-120
dBm
0
0.5
1
1.5
2
2.5
3
3.5
4
Frequency (kHz)
Ready
RBW=10 Hz
Sample rate=8 kHz
T=19.879



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 sample rate conversion. To perform an efficient multistage rate conversion, use the `dsp.SampleRateConverter` object that:

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);
```

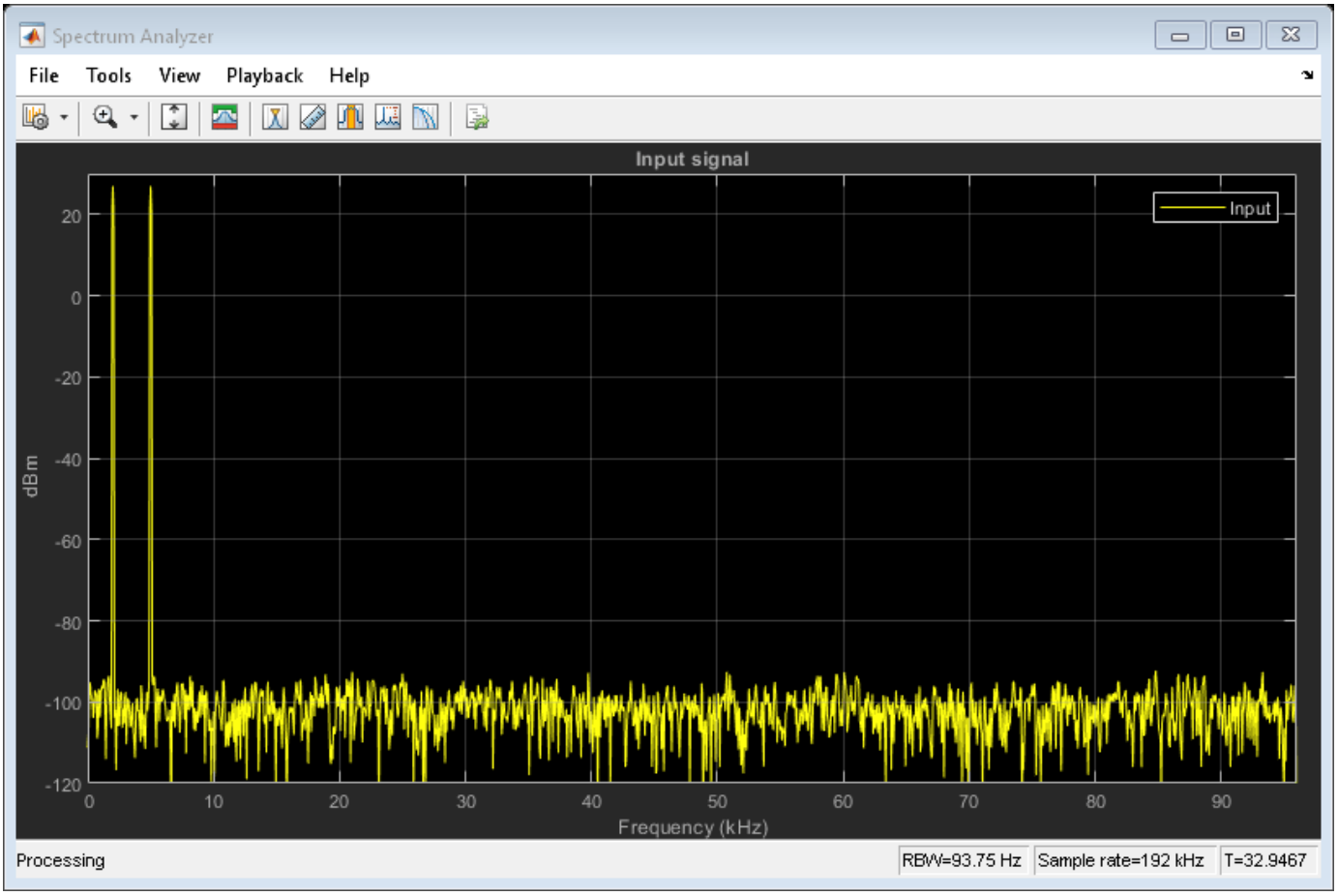
Initialize two spectrum analyzers, one to see the spectrum of the input signal and the other to see the spectrum of the rate converted output signal. The `'PlotAsTwoSidedSpectrum'` property of the `dsp.SpectrumAnalyzer` objects is set to `'false'`, indicating that the spectrum shown is one-sided in the range [0 *Fs*/2], where *Fs* is the sample rate of the signal.

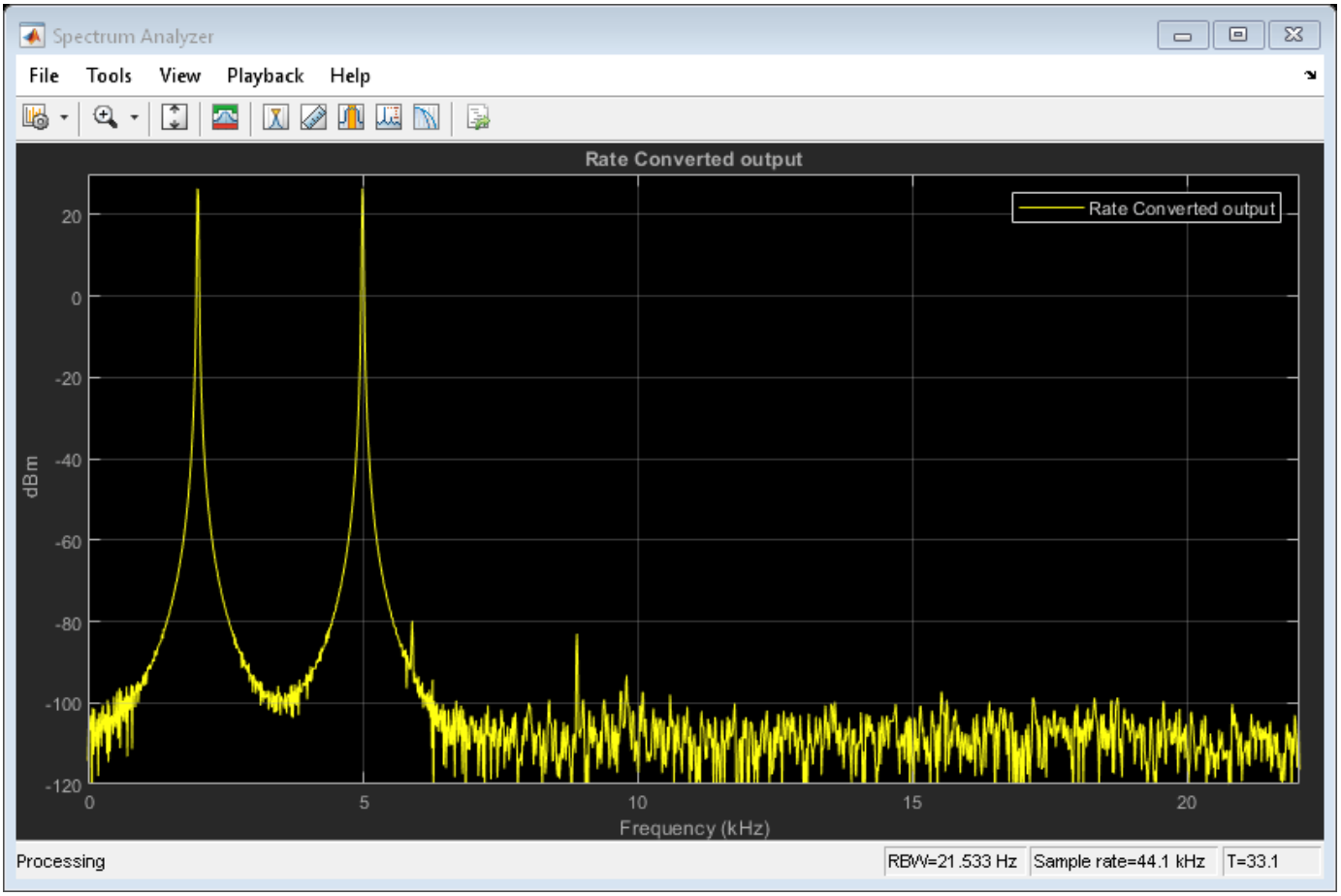
```
Fsin = SRC.InputSampleRate;
Fsout = SRC.OutputSampleRate;
specanainput = dsp.SpectrumAnalyzer('SampleRate',Fsin,...
    'PlotAsTwoSidedSpectrum',false,...
    'ShowLegend',true,'YLimits',[-120 30],...
    'Title','Input signal',...
    'ChannelNames', {'Input'});
specanaoutput = dsp.SpectrumAnalyzer('SampleRate',Fsout,...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

Stream the input signal and convert the sample rate of the signal using the sample rate converter. View the spectra of both the input and output signals in the two spectrum analyzers.

The spectrum analyzers show the spectrum 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 changes from 192 kHz to 44.1 kHz.

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





References

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

See Also

More About

- “Overview of Multirate Filters” on page 7-2
- “Design of Decimators and Interpolators” on page 4-211
- “Overview of Multistage Filters” on page 7-11
- “Multilevel Filter Banks” on page 7-28

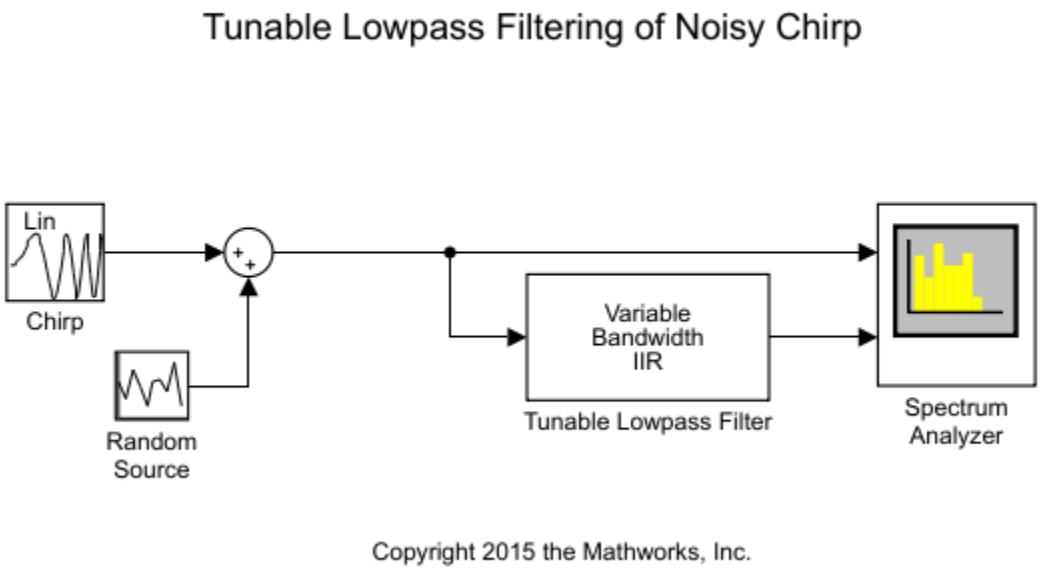
Tunable Lowpass Filtering of Noisy Input in Simulink

In this section...
“Open Lowpass Filter Model” on page 1-47 “Simulate the Model” on page 1-49

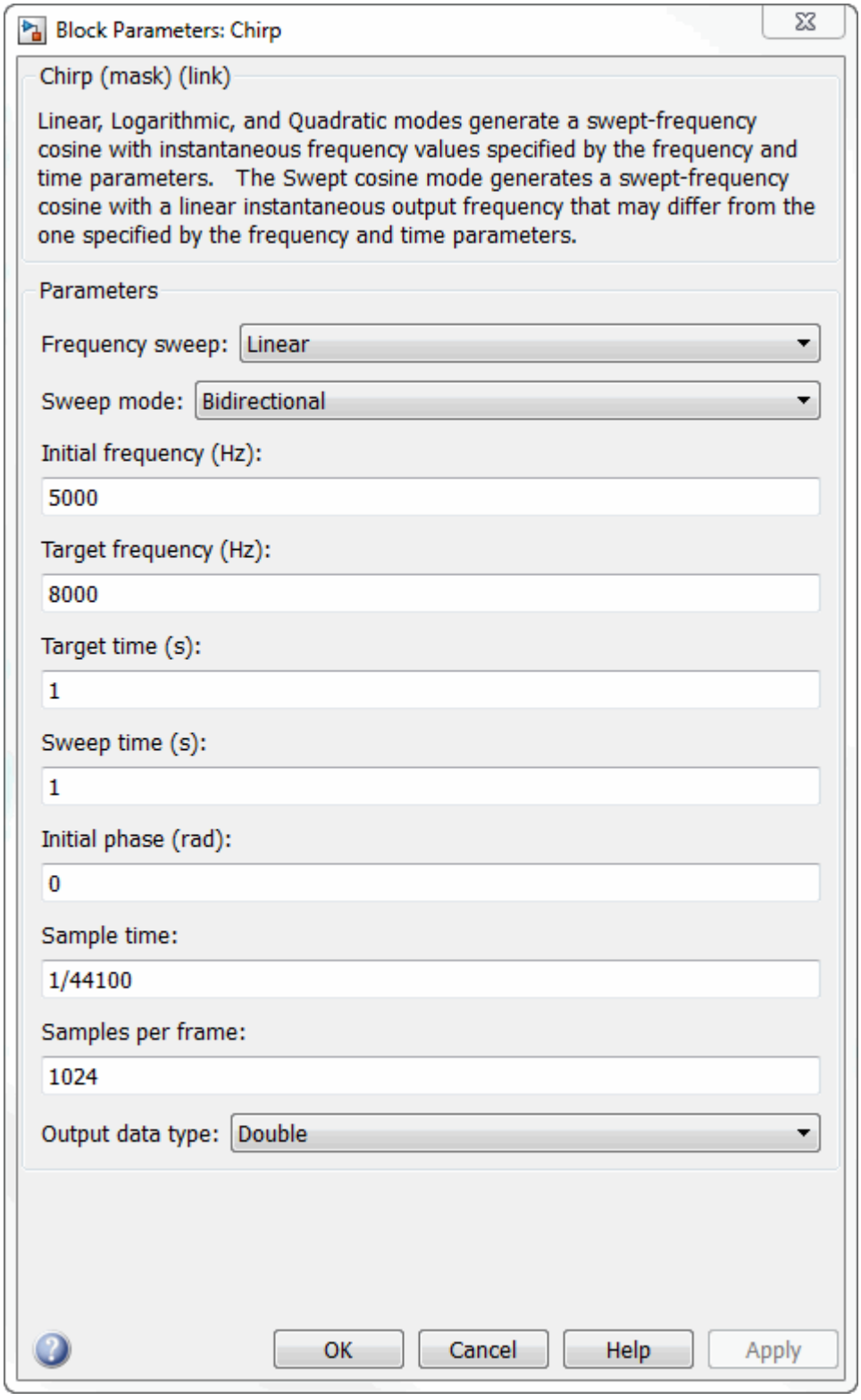
This example shows how to filter a noisy chirp signal with a lowpass filter that has a tunable passband frequency. The filter is a Variable Bandwidth IIR Filter block with **Filter type** set to `Lowpass`. This type of filter enables you to change the passband frequency during simulation without having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

Open Lowpass Filter Model

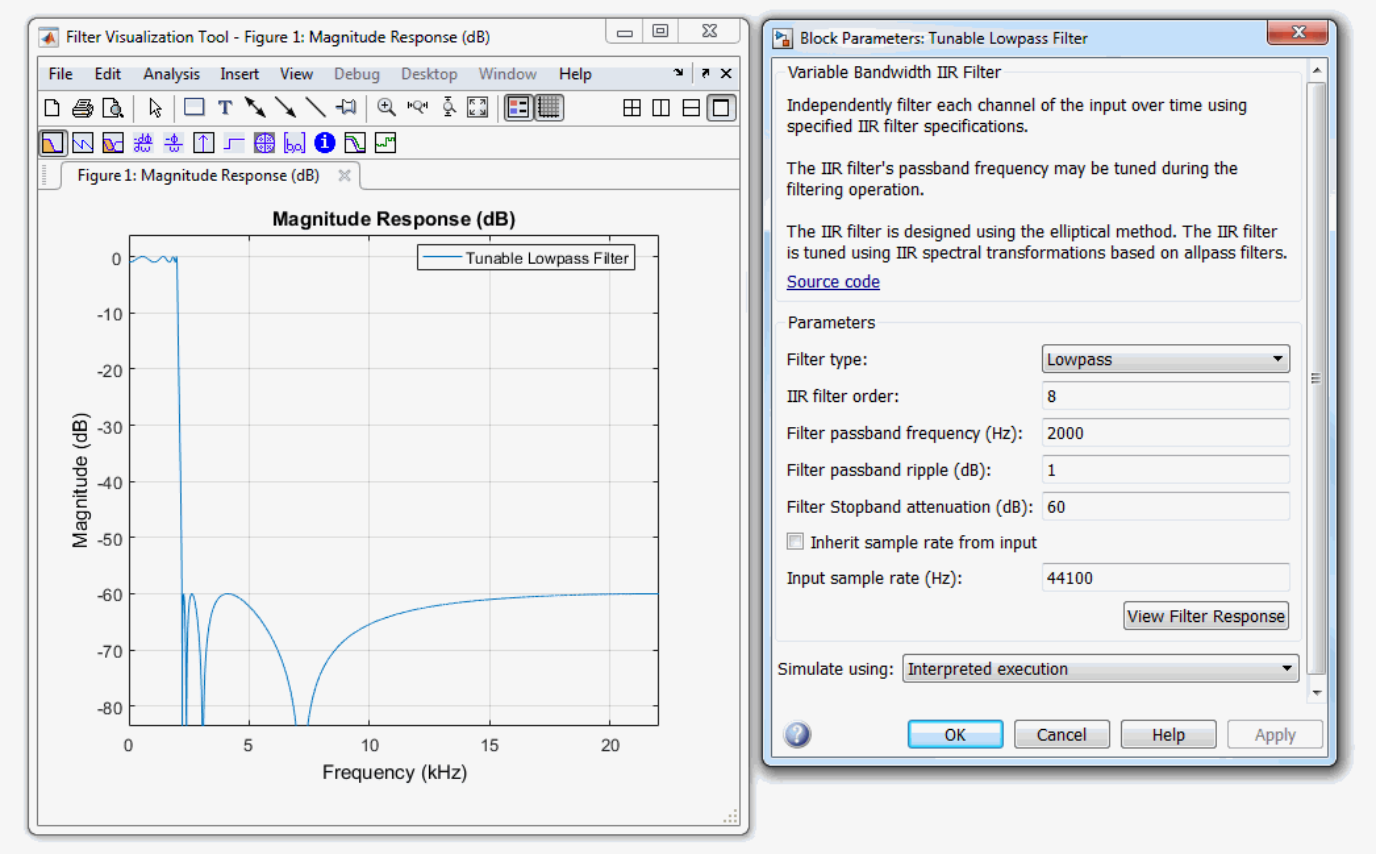
```
model = 'ex_tunable_chirp_lowpass';
open_system(model);
```



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.

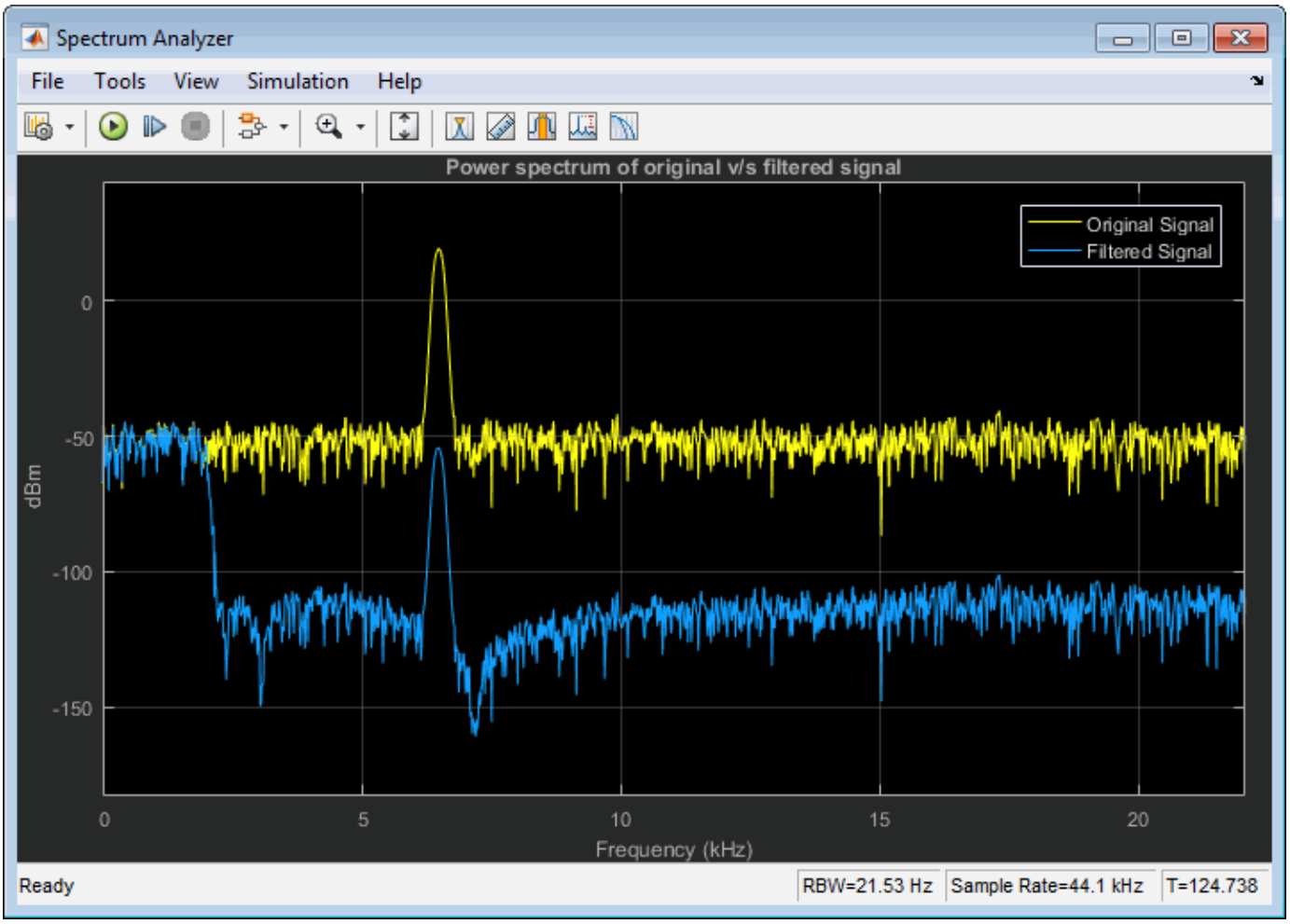


The Variable Bandwidth IIR Filter block has a lowpass frequency response, with the passband frequency set to 2000 Hz.



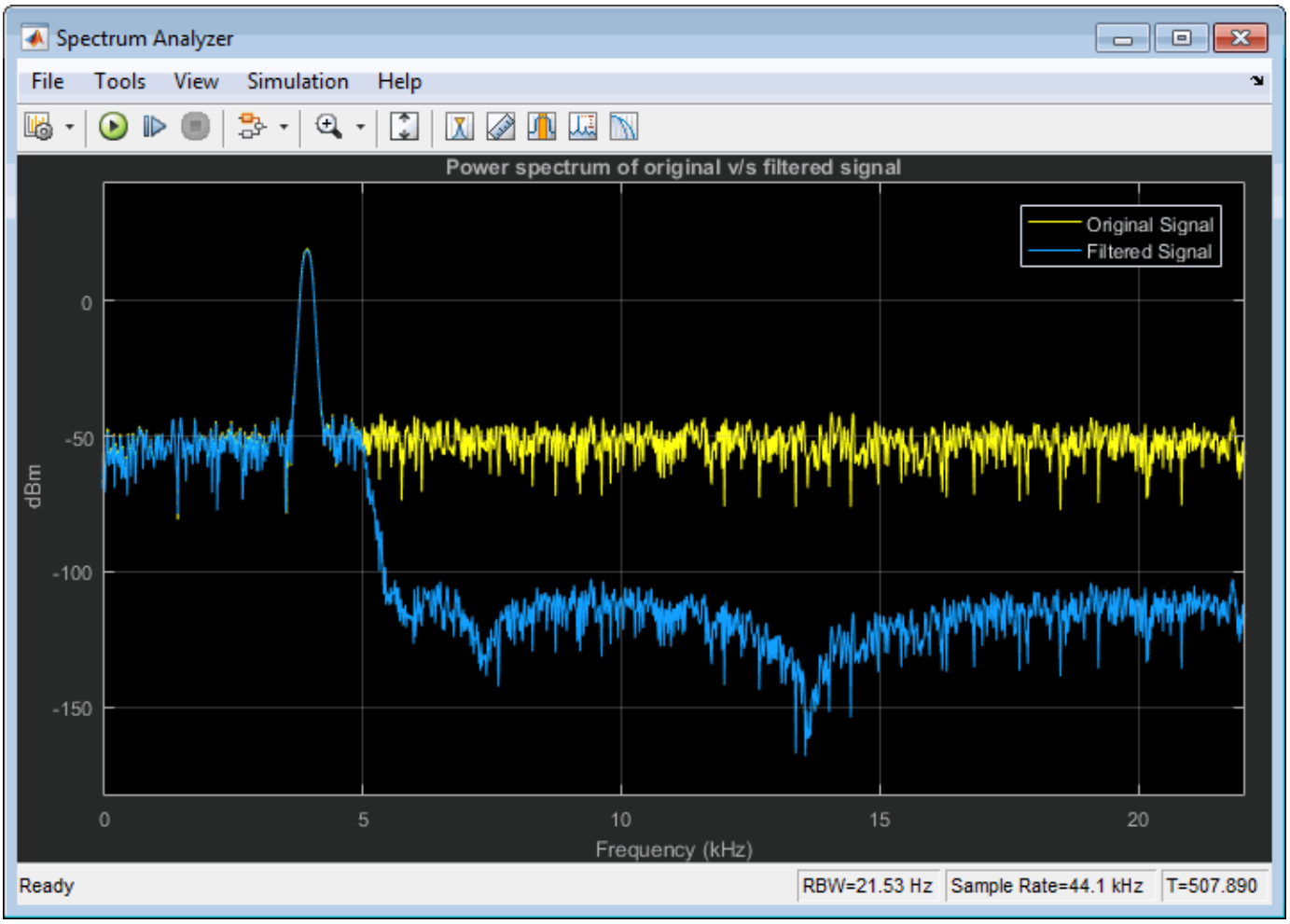
Simulate the Model

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.

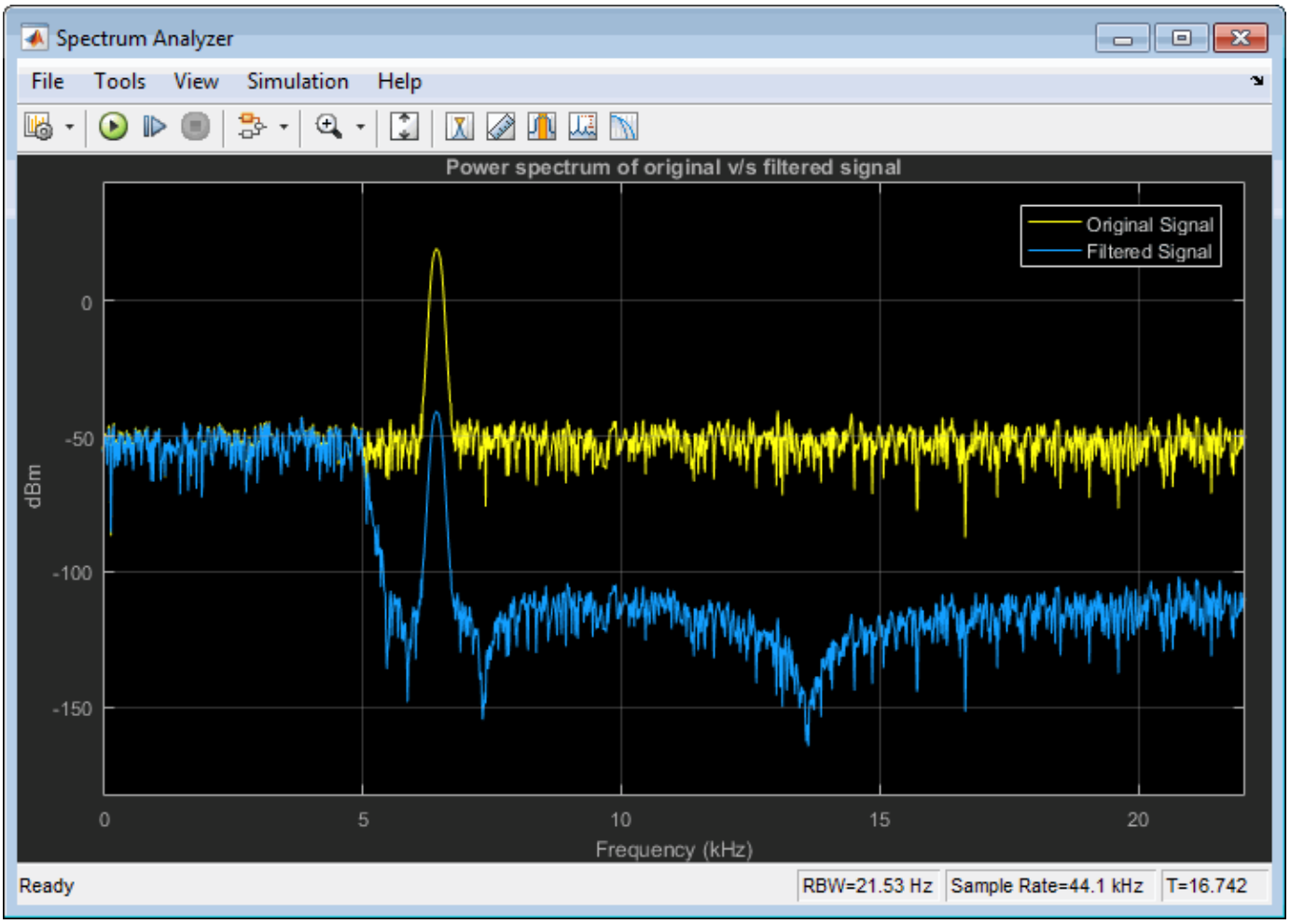


To tune the Passband frequency of the filter, in the Variable Bandwidth IIR Filter block dialog box, change **Filter passband frequency (Hz)** to 6000 Hz. Click **Apply** and the output of the Spectrum Analyzer changes immediately.

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 | “Multirate Filtering in MATLAB and Simulink” 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 1-53
“Accelerate the FIR Filter Using codegen” on page 1-54
“Accelerate the FIR Filter Using dspunfold” on page 1-55
“Kalman Filter Algorithm” on page 1-56
“Accelerate the Kalman Filter Using codegen” on page 1-58
“Accelerate the Kalman Filter Using dspunfold” on page 1-59

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 1-66
- “Generate a Multithreaded MEX File from a MATLAB Function Using Unfolding” on page 4-297
- “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 1-61
“Accelerate the FIR Filter Using codegen” on page 1-62
“Accelerate the FIR Filter Using dspunfold” on page 1-63

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 1-66
- “Generate a Multithreaded MEX File from a MATLAB Function Using Unfolding” on page 4-297

- “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 are collected using a host computer with six physical CPU cores.

Required MathWorks™ products:

- DSP System Toolbox™
- MATLAB Coder™

Using 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

```
type dspunfoldDCTExample.m
```

```
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));

end
```

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)'}
```

```
Code generation successful.
```

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
```

```
State length: 0 frames, Repetition: 1, Output latency: 12 frames, Threads: 6
Analyzing: dspunfoldDCTExample.m
Creating single-threaded MEX file: dspunfoldDCTExample_st.mexa64
Creating multi-threaded MEX file: dspunfoldDCTExample_mt.mexa64
Creating analyzer file: dspunfoldDCTExample_analyzer.p
```

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`.

```
type 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 = 2.9x
```

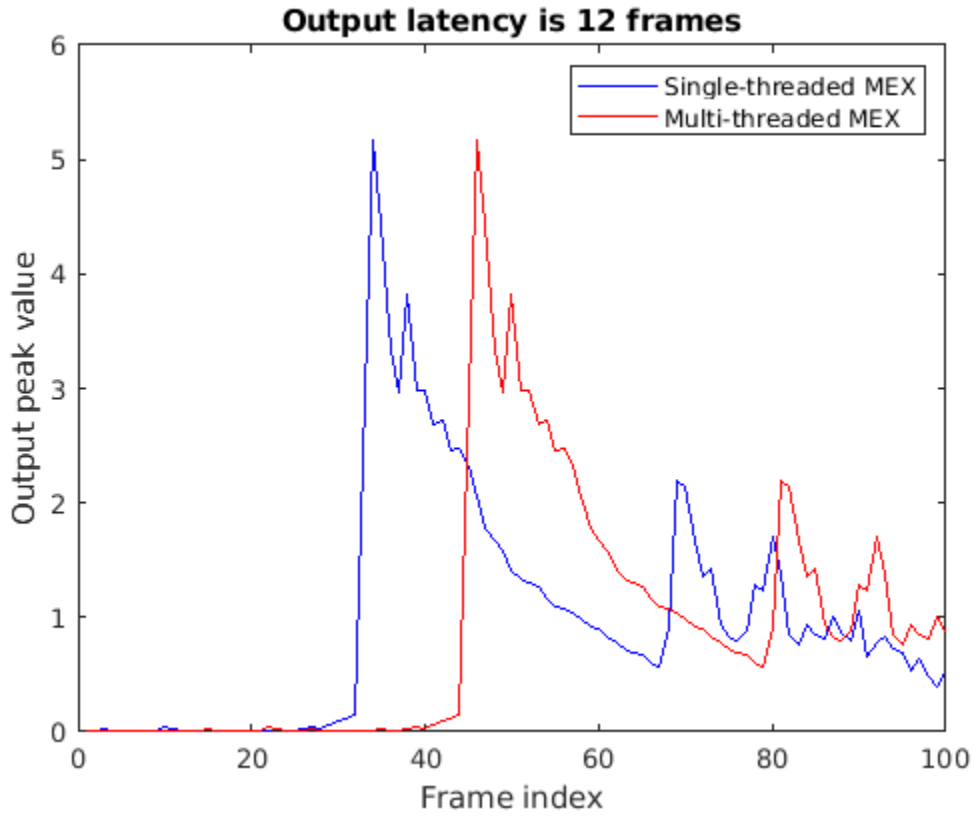
To improve the speedup even more, increase the repetition value. To modify the repetition value, use the `-r` flag. For more information on the repetition value, see the `dspunfold` function reference page. For an example on how to specify the repetition value, see the section 'Using dspunfold with a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

```
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 = step(FIRSTFIR,u,c1);
y = step(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(192,0.8);
firCoeffs2 = fir1(256,0.2,'High');
dspunfold dspunfoldFIRExample -args {(1:4096)',firCoeffs1,firCoeffs2} -s 1
```

```
State length: 1 frames, Repetition: 1, Output latency: 12 frames, Threads: 6
Analyzing: dspunfoldFIRExample.m
Creating single-threaded MEX file: dspunfoldFIRExample_st.mexa64
Creating multi-threaded MEX file: dspunfoldFIRExample_mt.mexa64
Creating analyzer file: dspunfoldFIRExample_analyzer.p
```

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 12 frames. To measure the speedup, execute `dspunfoldBenchmarkFIRExample`.

```
dspunfoldBenchmarkFIRExample;
```

```
Speedup = 1.4x
```

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 (192 + 256 = 448). Using the `-f` argument, mark the first input argument as true (frame), and the last two input arguments as false (nonframes)

```
dspunfold dspunfoldFIRExample -args {(1:4096)',firCoeffs1,firCoeffs2} -s 448 -f [true,false,false
```

```
State length: 448 samples, Repetition: 1, Output latency: 12 frames, Threads: 6
Analyzing: dspunfoldFIRExample.m
Creating single-threaded MEX file: dspunfoldFIRExample_st.mexa64
Creating multi-threaded MEX file: dspunfoldFIRExample_mt.mexa64
Creating analyzer file: dspunfoldFIRExample_analyzer.p
```

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 = 2.0x
```

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 {(1:4096)',firCoeffs1,firCoeffs2} ...
-s 448 -f [true,false,false] -r 5

State length: 448 samples, Repetition: 5, Output latency: 60 frames, Threads: 6
Analyzing: dspunfoldFIRExample.m
Creating single-threaded MEX file: dspunfoldFIRExample_st.mexa64
Creating multi-threaded MEX file: dspunfoldFIRExample_mt.mexa64
Creating analyzer file: dspunfoldFIRExample_analyzer.p
```

Again, measure the speedup for the resulting multithreaded MEX, using the `dspunfoldBenchmarkFIRExample` function. Speedup increases again, but the output latency is now 60 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 = 2.2x
```

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 {(1:4096)',firCoeffs1,firCoeffs2} ...
-s auto -f [true,false,false] -r 5

State length: [autodetect] samples, Repetition: 5, Output latency: 60 frames, Threads: 6
Analyzing: dspunfoldFIRExample.m
Creating single-threaded MEX file: dspunfoldFIRExample_st.mexa64
Searching for minimal state length (this might take a while)
Checking stateless ... Insufficient
Checking 4096 samples ... Sufficient
Checking 2048 samples ... Sufficient
Checking 1024 samples ... Sufficient
Checking 512 samples ... Sufficient
Checking 256 samples ... Insufficient
Checking 384 samples ... Insufficient
Checking 448 samples ... Sufficient
Checking 416 samples ... Insufficient
Checking 432 samples ... Insufficient
Checking 440 samples ... Insufficient
Checking 444 samples ... Insufficient
Checking 446 samples ... Insufficient
Checking 447 samples ... Insufficient
Minimal state length is 448 samples
Creating multi-threaded MEX file: dspunfoldFIRExample_mt.mexa64
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 448, 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(192,0.8);
firCoeffs1_2 = fir1(192,0.7);
firCoeffs1_3 = fir1(192,0.6);
firCoeffs2_1 = fir1(256,0.2,'High');
firCoeffs2_2 = fir1(256,0.1,'High');
firCoeffs2_3 = fir1(256,0.3,'High');
dspunfoldFIRExample_analyzer((1:4096*3)',[firCoeffs1_1;firCoeffs1_2;firCoeffs1_3],...
[firCoeffs2_1;firCoeffs2_2;firCoeffs2_3]);
```

```
Analyzing multi-threaded MEX file dspunfoldFIRExample_mt.mexa64. For best results, please refrai
Latency = 60 frames
Speedup = 2.4x
```

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 4096 doubles for its first input, while 4096 × 3 samples were provided to `dspunfoldFIRExample_analyzer`. The analyzer interprets this input as 3 frames of 4096 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 180 or 3 × latency, where latency is 60 in this case.

```
| input1 | input2 | input3

------+----------------+--------------+--------------

Step1 | (1:4096)' | firCoeffs1_1 | firCoeffs2_1

Step2 | (4097:8192)' | firCoeffs1_2 | firCoeffs2_2

Step3 | (8193:12288)' | firCoeffs1_3 | firCoeffs2_3

Step4 | (1:4096)' | 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 Multithreaded MEX File from a MATLAB Function Using Unfolding” on page 4-297 | `dspunfold` | “Workflow for Generating a Multithreaded MEX File using dspunfold” on page 20-43 | “Why Does the Analyzer Choose the Wrong State Length?” on page 20-47 | “How Is dspunfold Different from parfor?” on page 20-41

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:

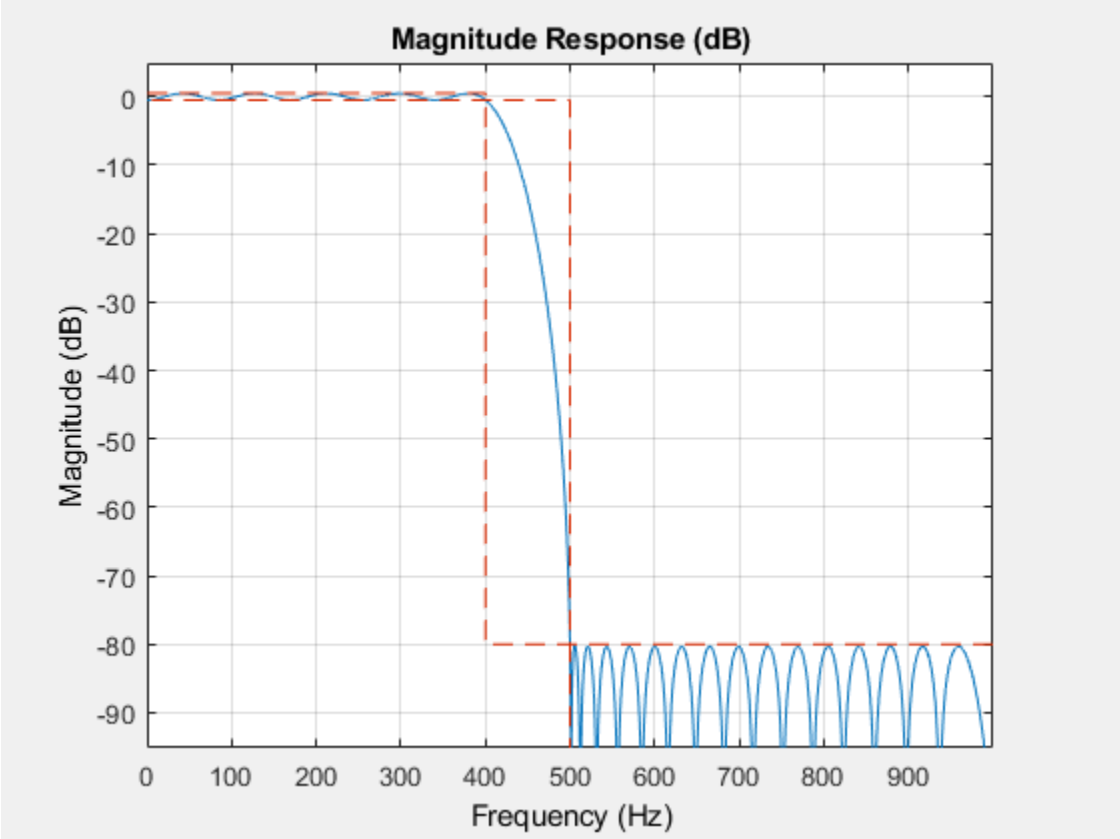
- Sample rate: 2000 Hz
- Center frequency: 450 Hz
- Transition width: 100 Hz
- Equiripple design
- Maximum 1 dB of ripple in the passband
- Minimum 80 dB of attenuation in the stopband

```
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: [-0.0013 -0.0055 -0.0112 -0.0125 -0.0048 0.0062 ... ]
    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')
```

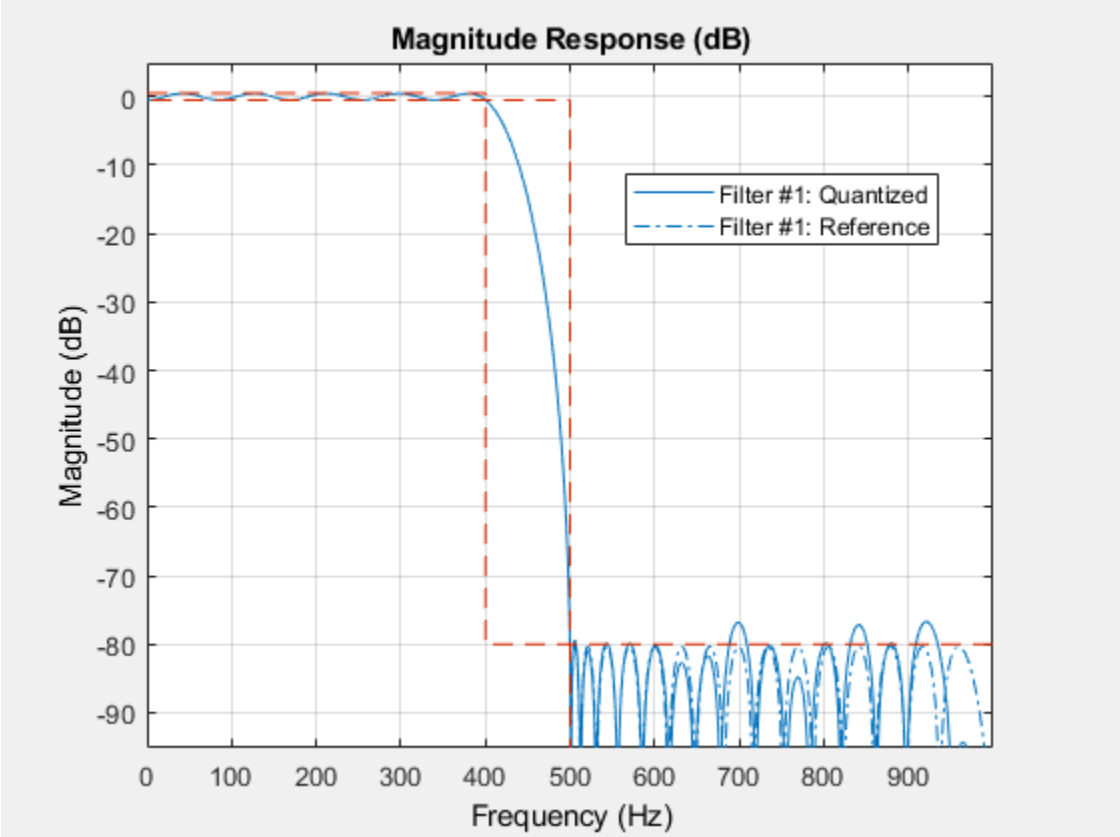
```
ans = 6.8994e-05
```

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)
```

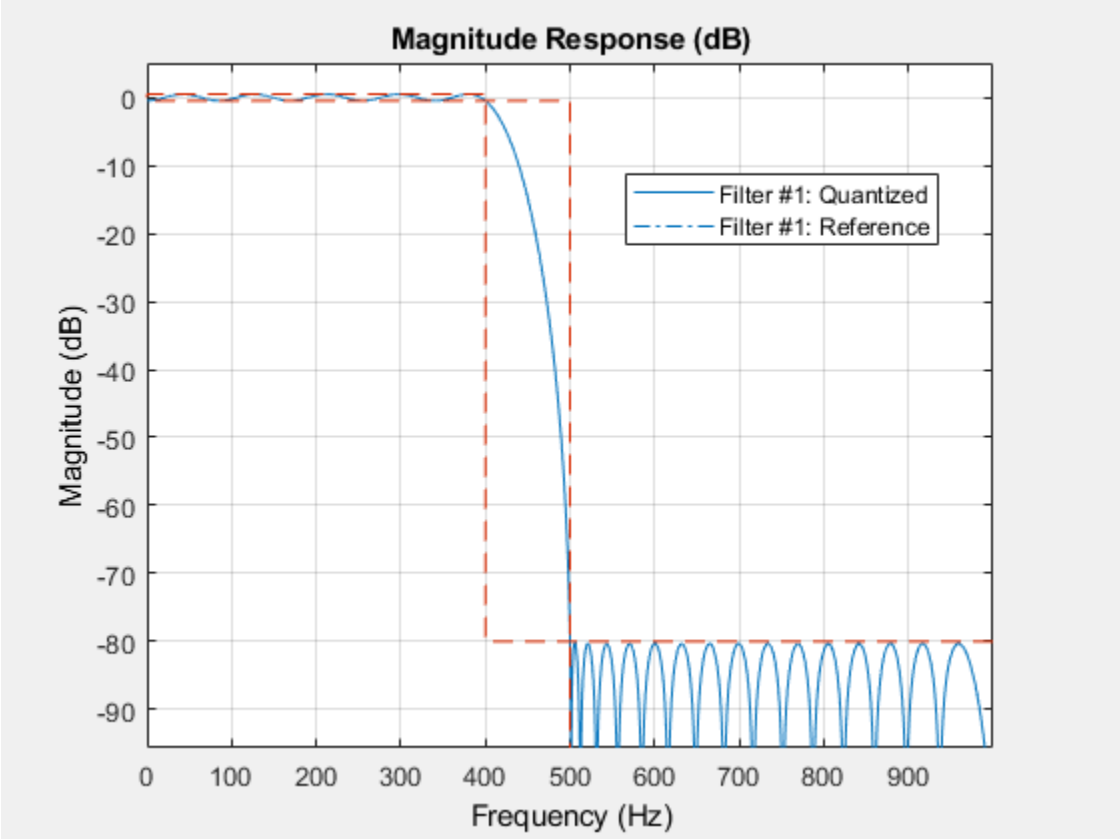
```
ans =
Sample Rate      : 2 kHz
Passband Edge    : 400 Hz
3-dB Point       : 416.2891 Hz
6-dB Point       : 428.1081 Hz
Stopband Edge    : 500 Hz
Passband Ripple  : 0.96325 dB
Stopband Atten.  : 76.6913 dB
Transition Width : 100 Hz
```

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
```

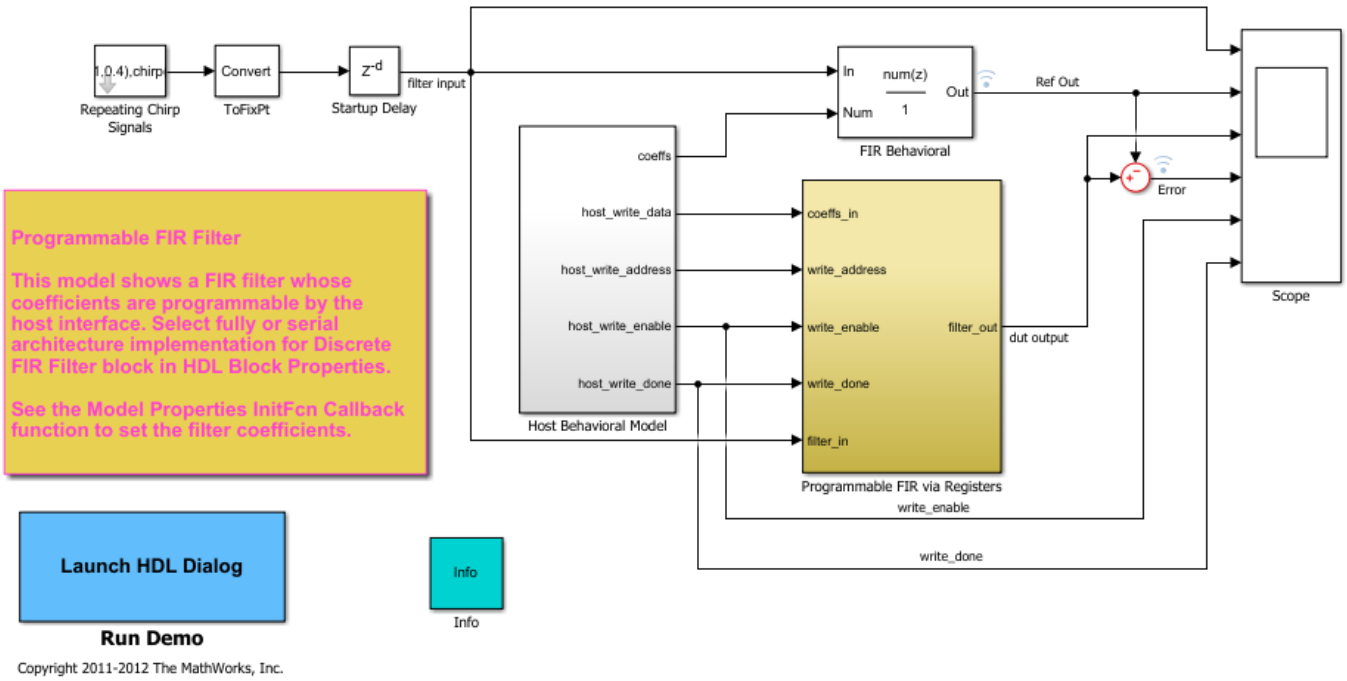
Visualizing Multiple Signals Using Logic Analyzer

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” on page 21-24.

Model Programmable FIR Filter

Open the example model.

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

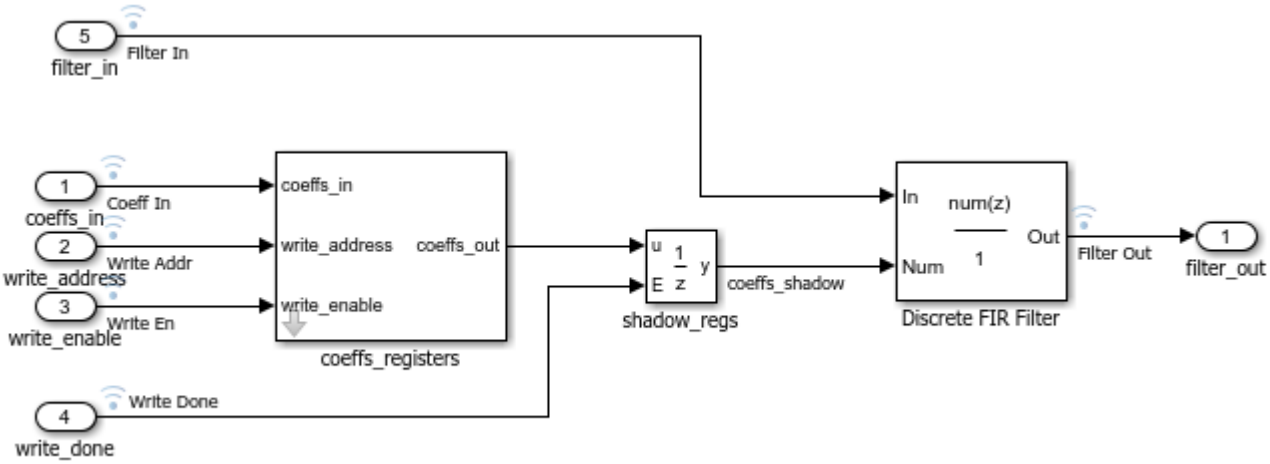


Consider two FIR filters, one with a lowpass response and the other with a highpass response. The coefficients can be specified using the **InitFcn*** callback function. To specify the callback, select **File > Model Properties > Model Properties**. In the dialog box, in the **Callbacks** tab, select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



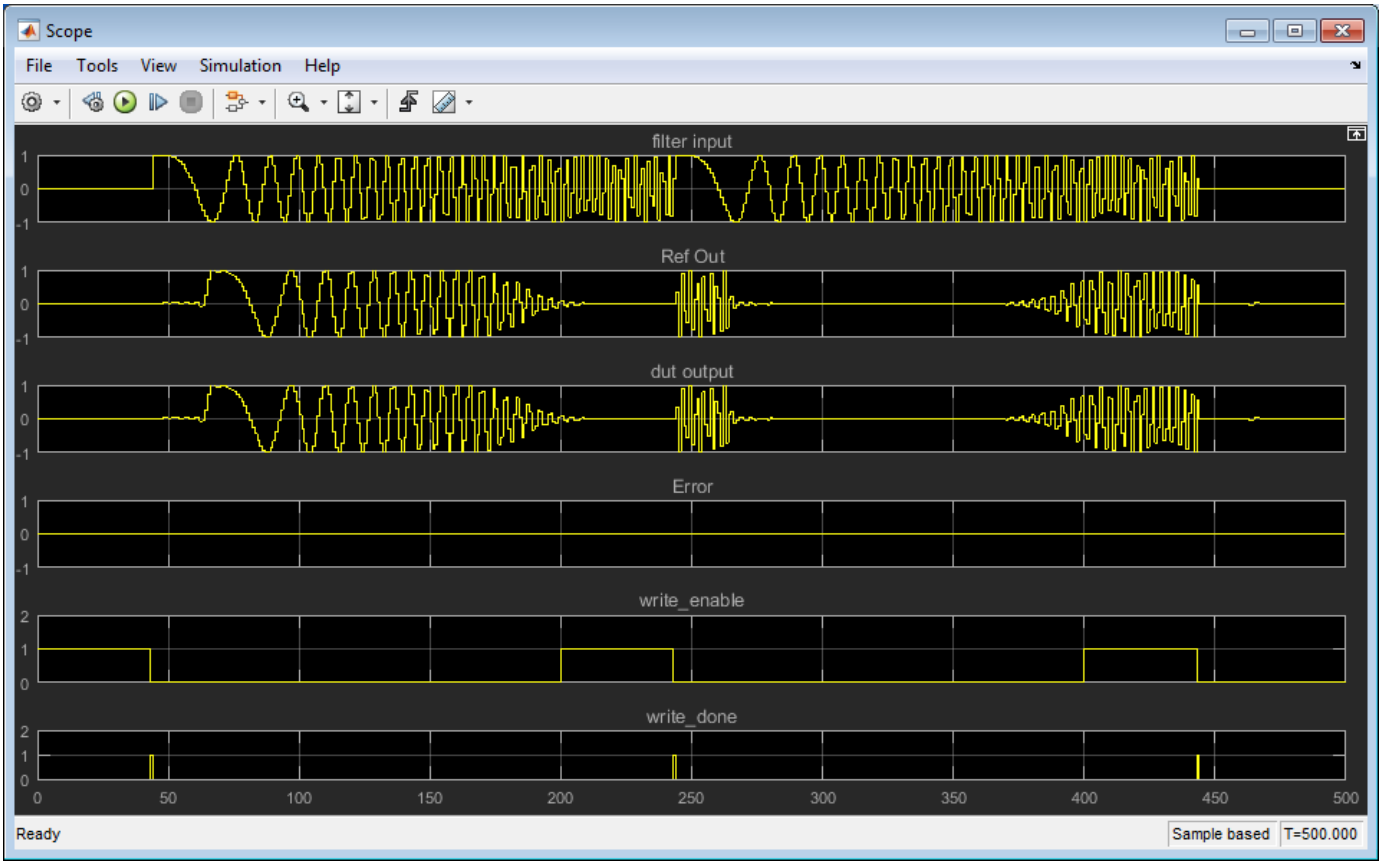
Simulation

Run the example model.

```
sim(modelname)
```

Open the scope.

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

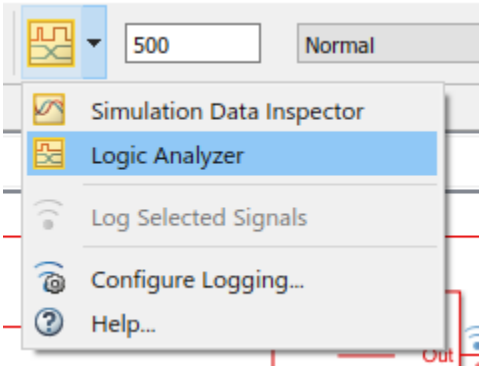


Compare the DUT (Design under Test) output with the reference output.

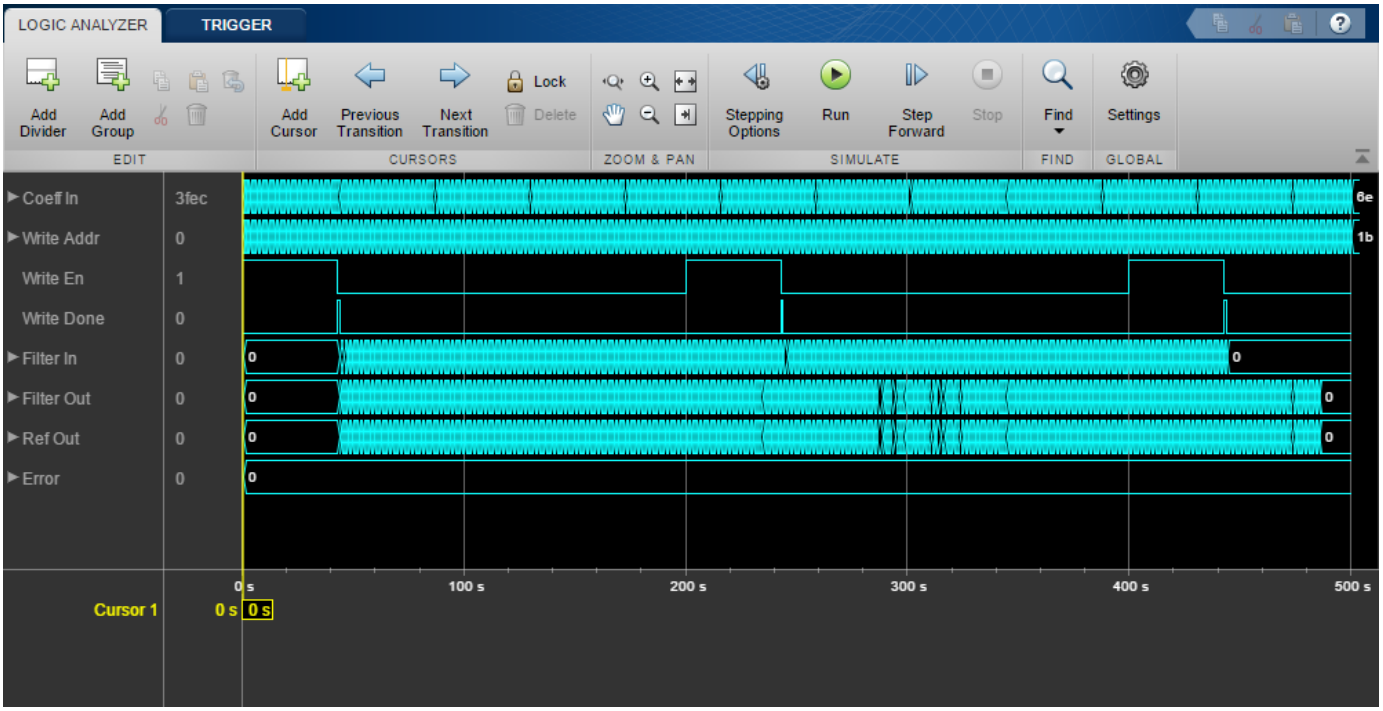
Use the Logic Analyzer

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**.

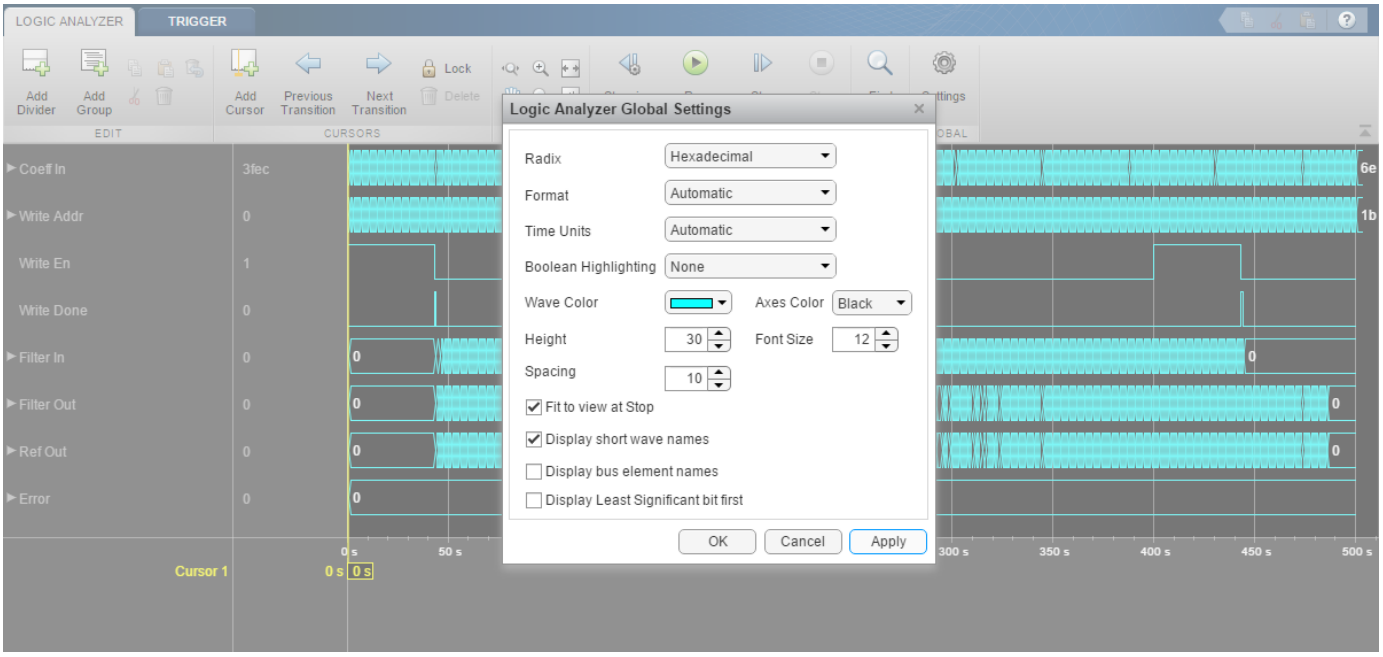


The Logic Analyzer displays waveforms of the selected signals.

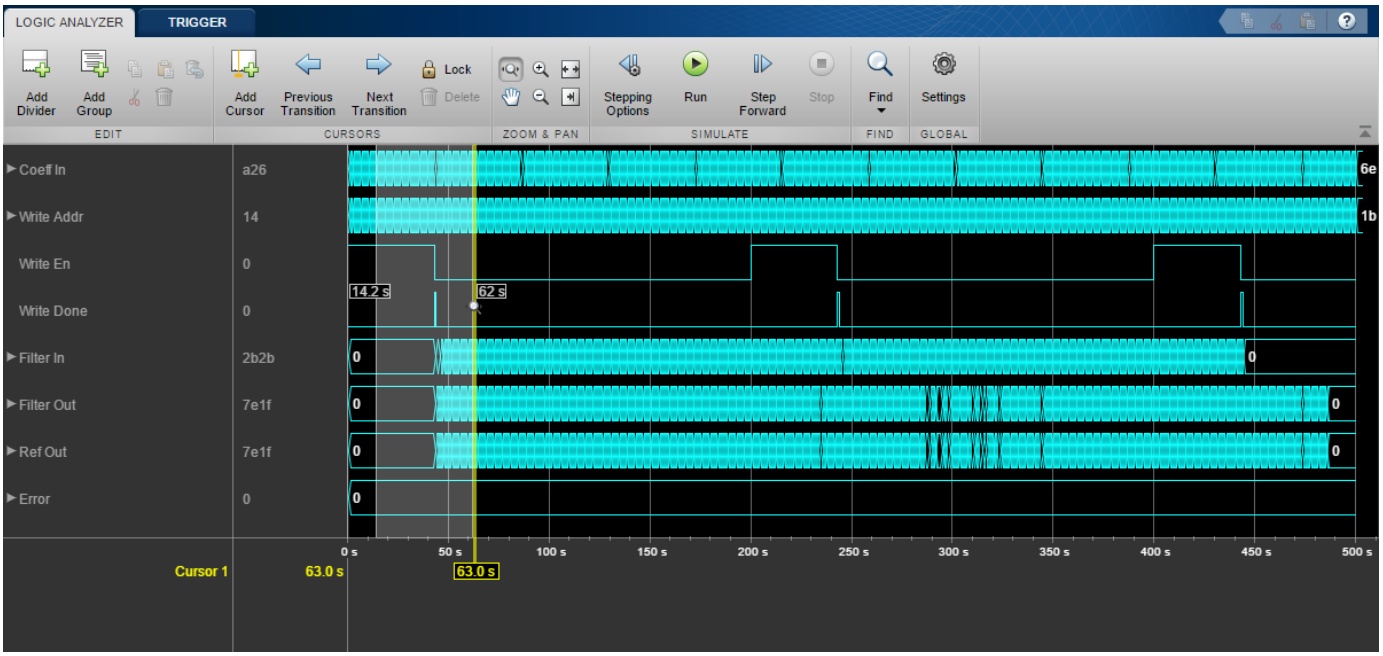


Modify the Display

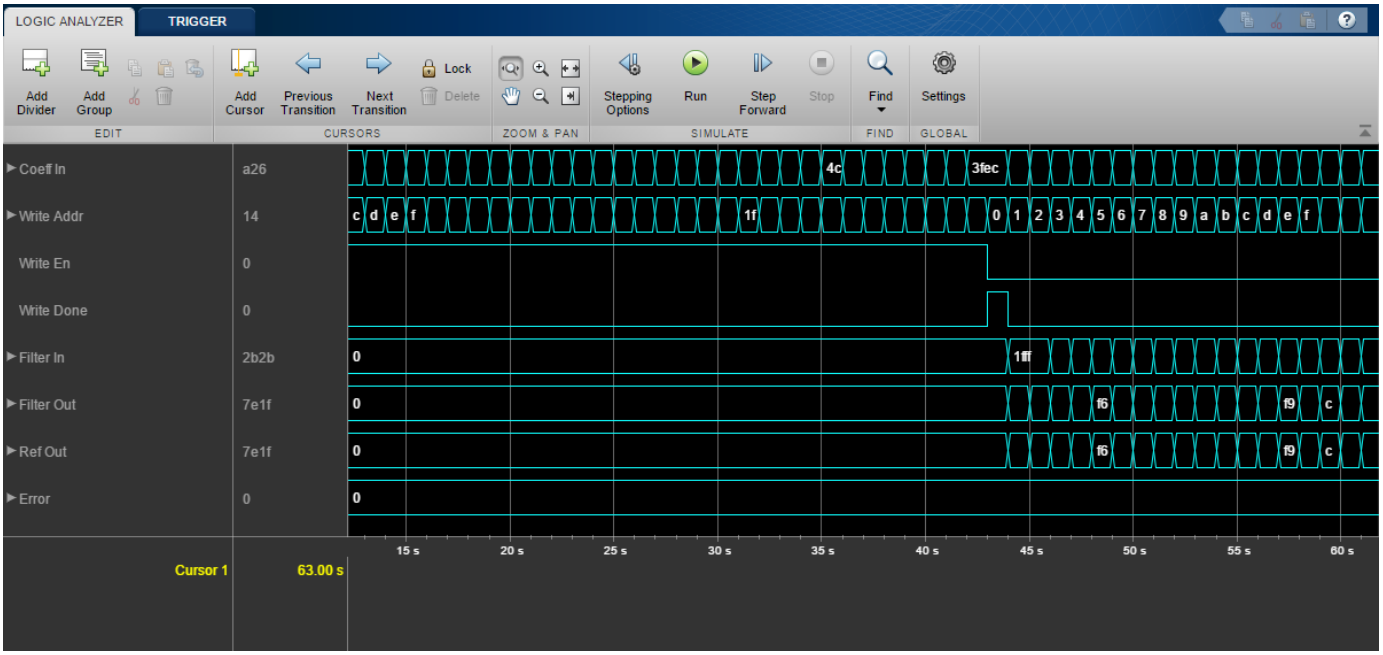
In the Logic Analyzer, you can modify the height of all the displayed channels, and the spacing between the channels. Click the **Settings** button. Then, modify the default height and spacing for each wave. Click **Apply** to show the new dimensions in the background.



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.

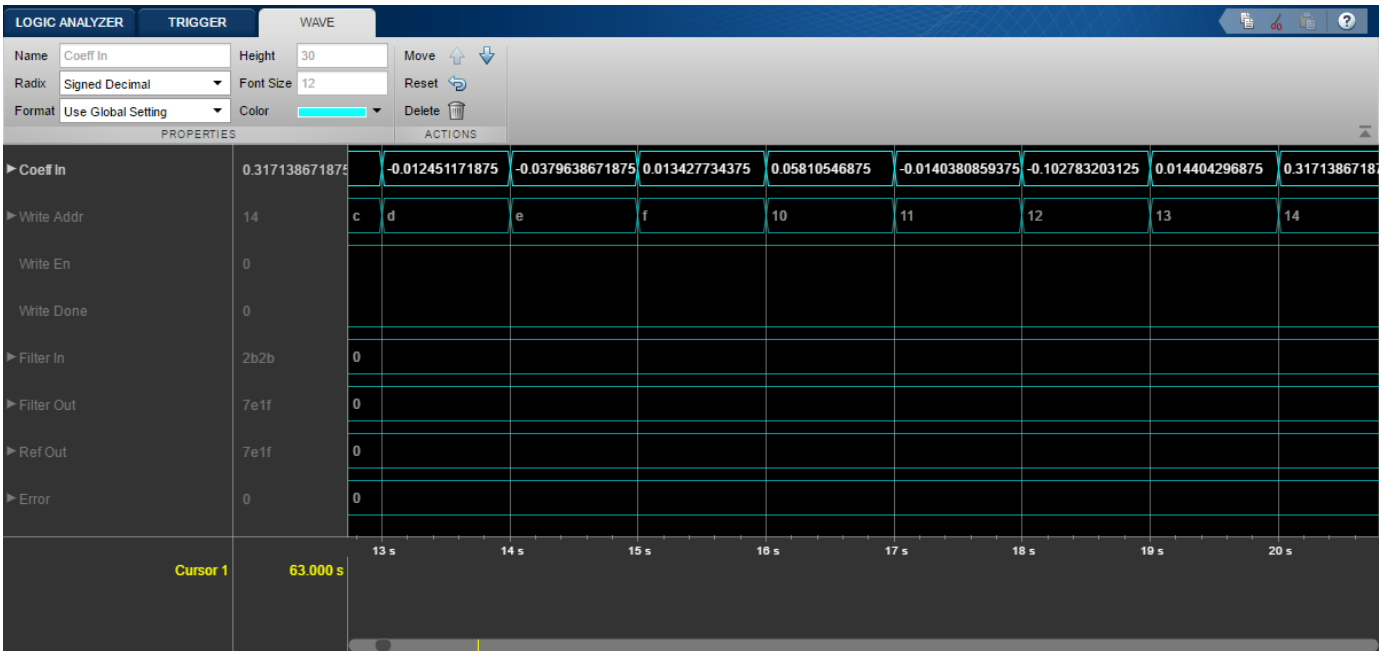


The Logic Analyzer now displays the time span you selected.

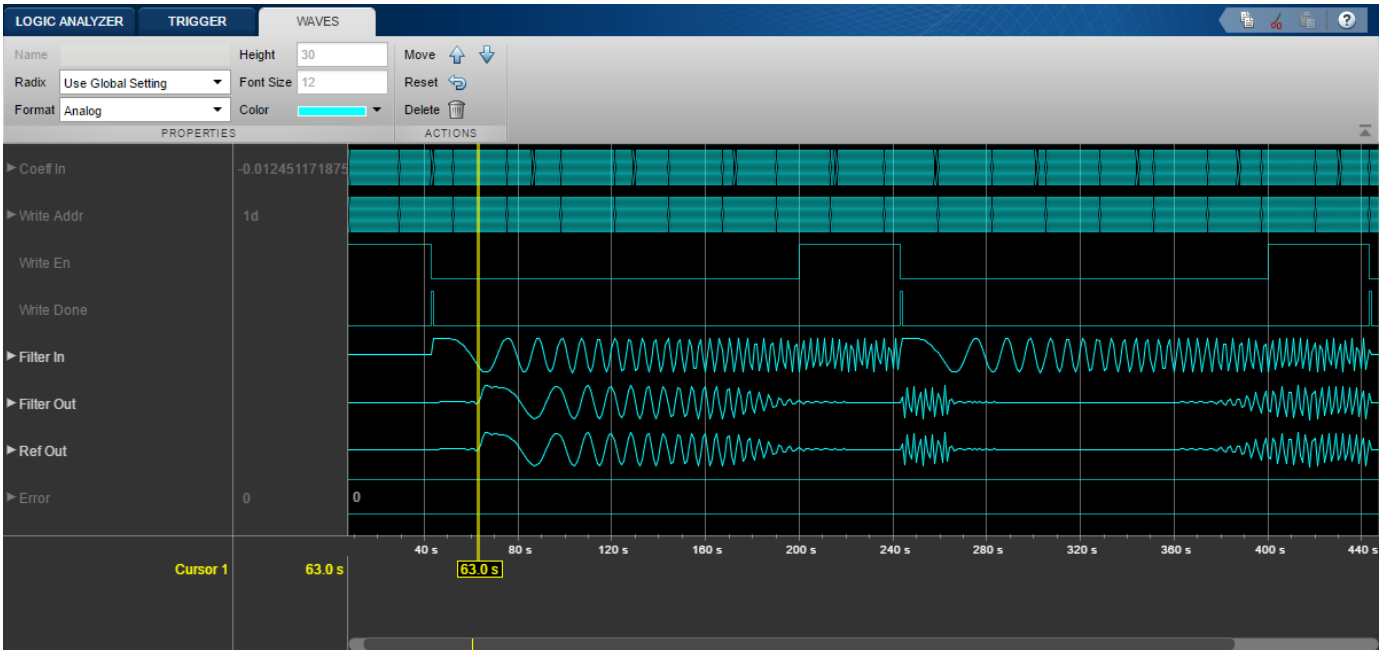


You can also control the display on a per-waveform basis. To modify an individual waveform, double-click the signal, select the signal, then click the **WAVE** tab to modify its settings.

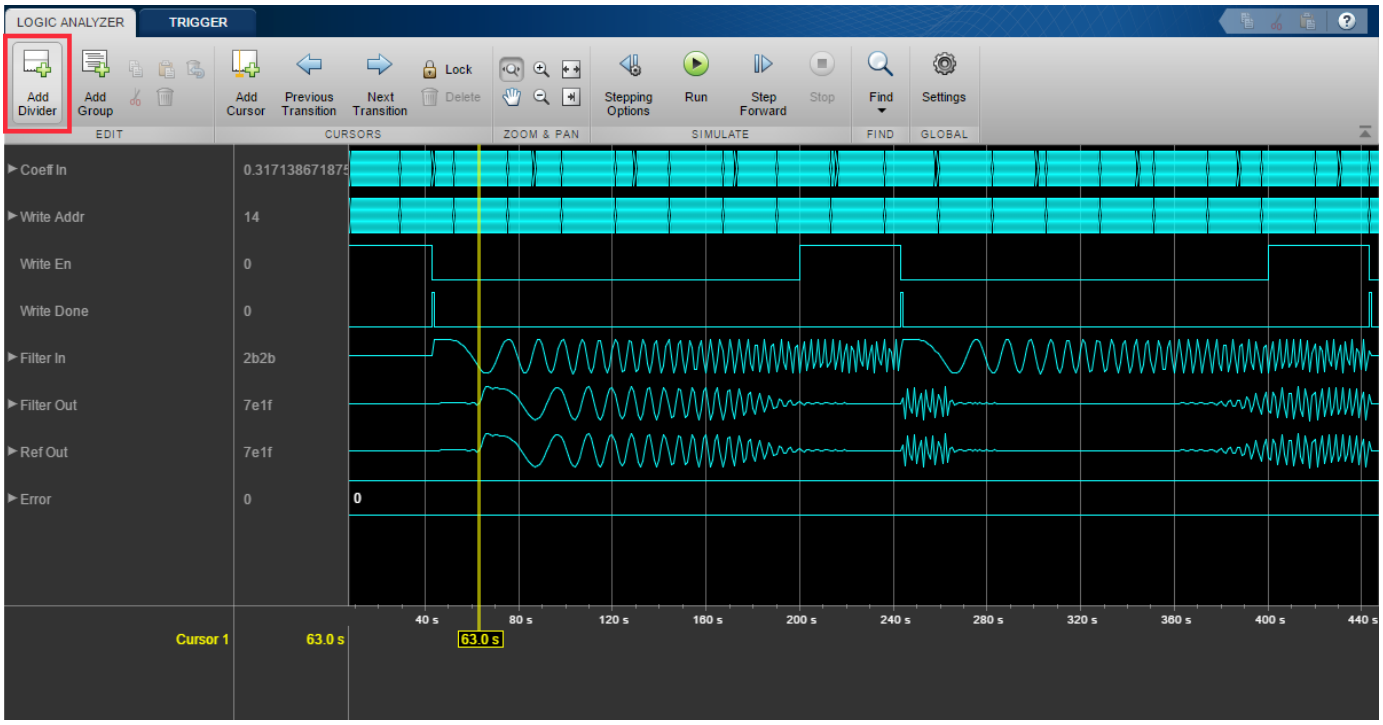
Display the `CoeffIn` signal in signed decimal mode. The conversion uses the fractional and integer bits as defined for this signal in your model.



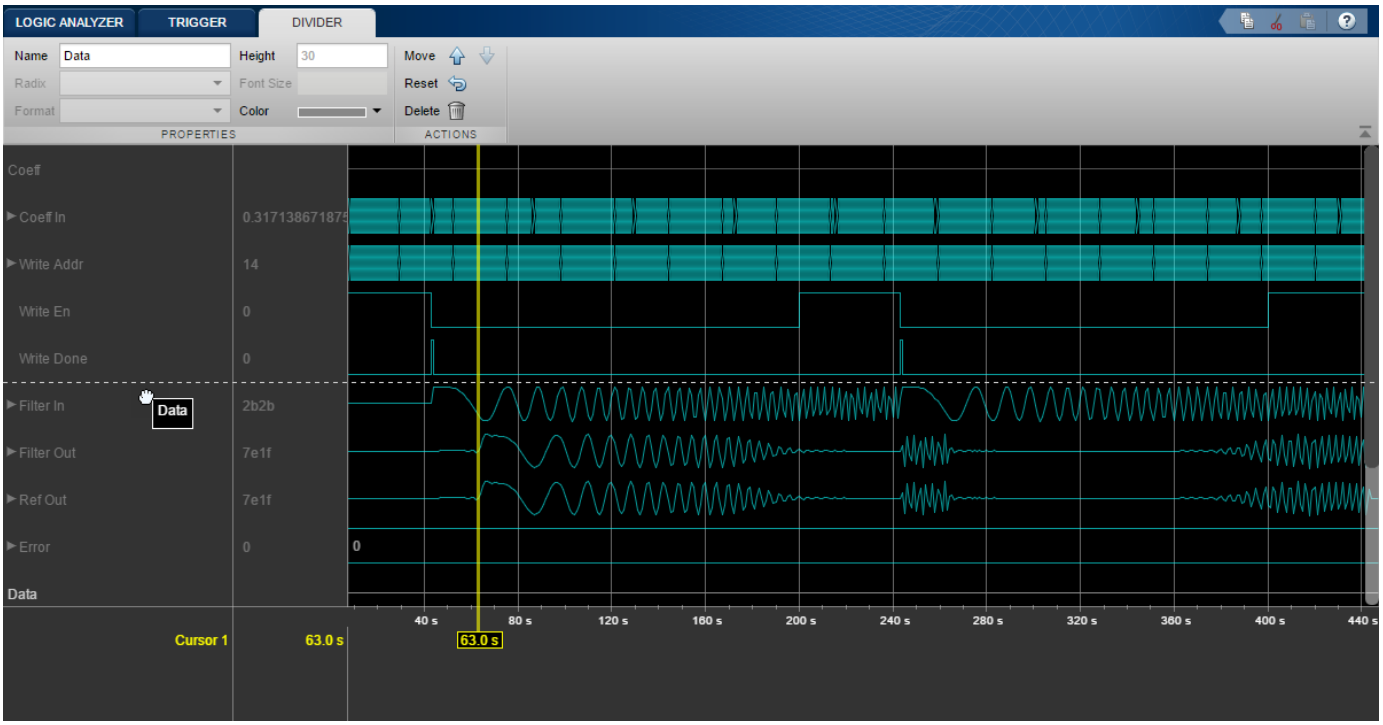
Another useful mode of visualization in the Logic Analyzer is the analog format. View the `Filter In`, `Filter Out`, and `Ref Out` signals in analog format.



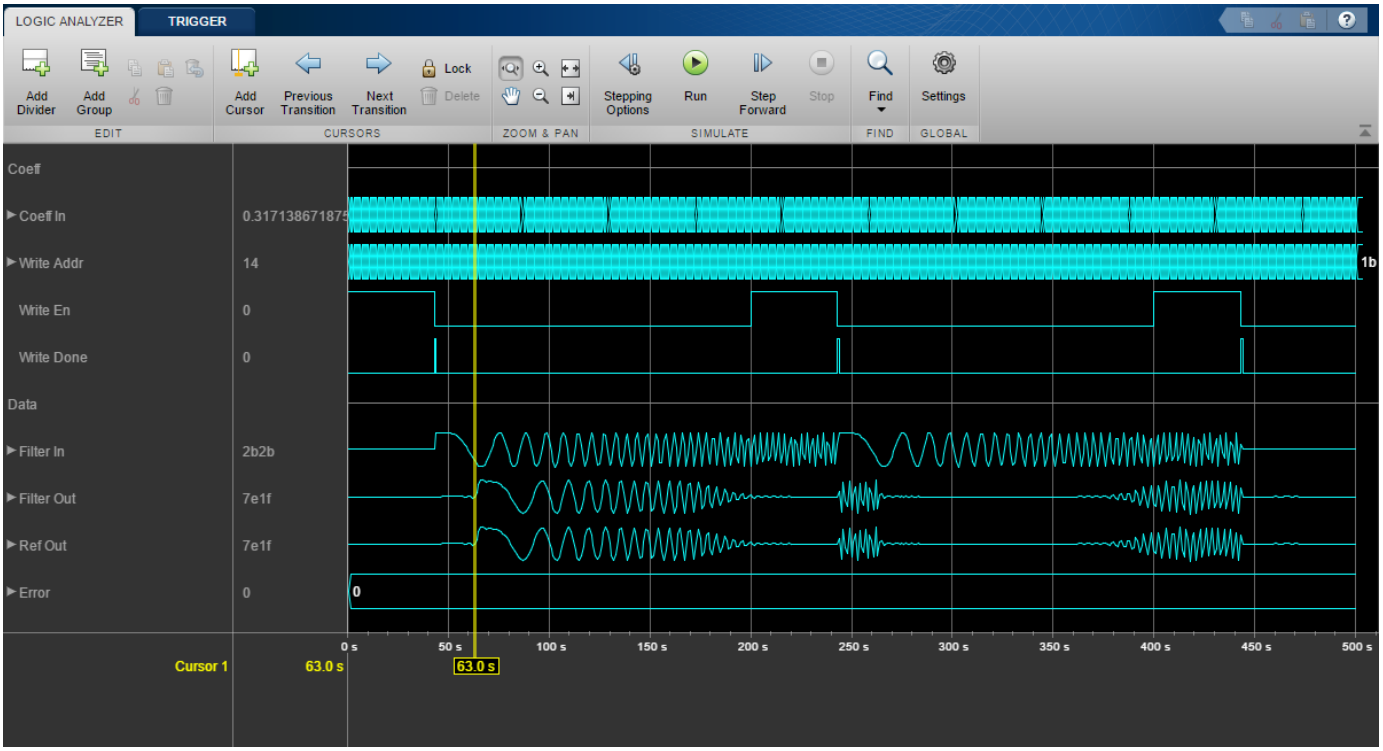
You can also add dividers to the display. Click the **Add Divider** button in the toolbar.



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**.

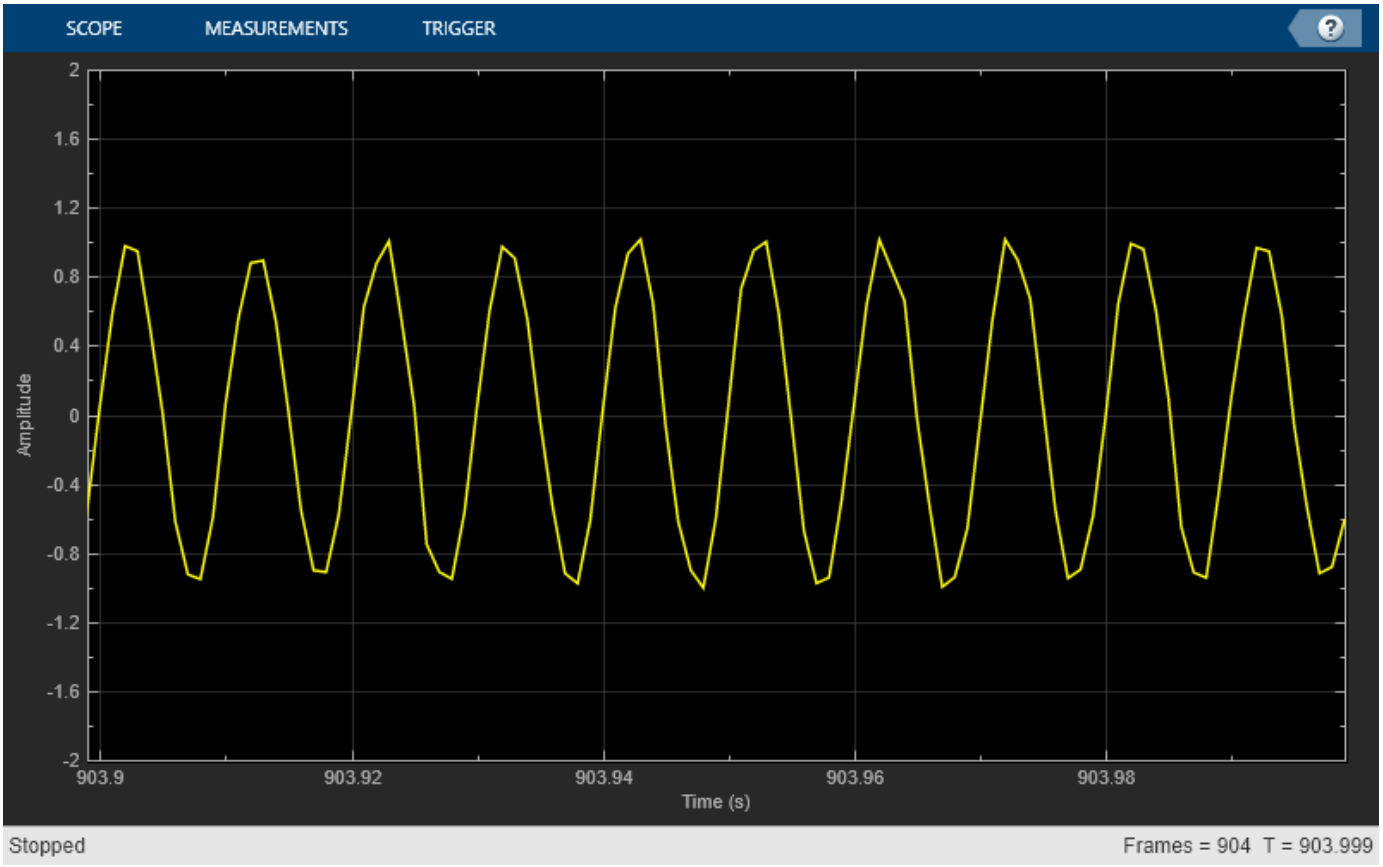
Signal Visualization and Measurements in MATLAB

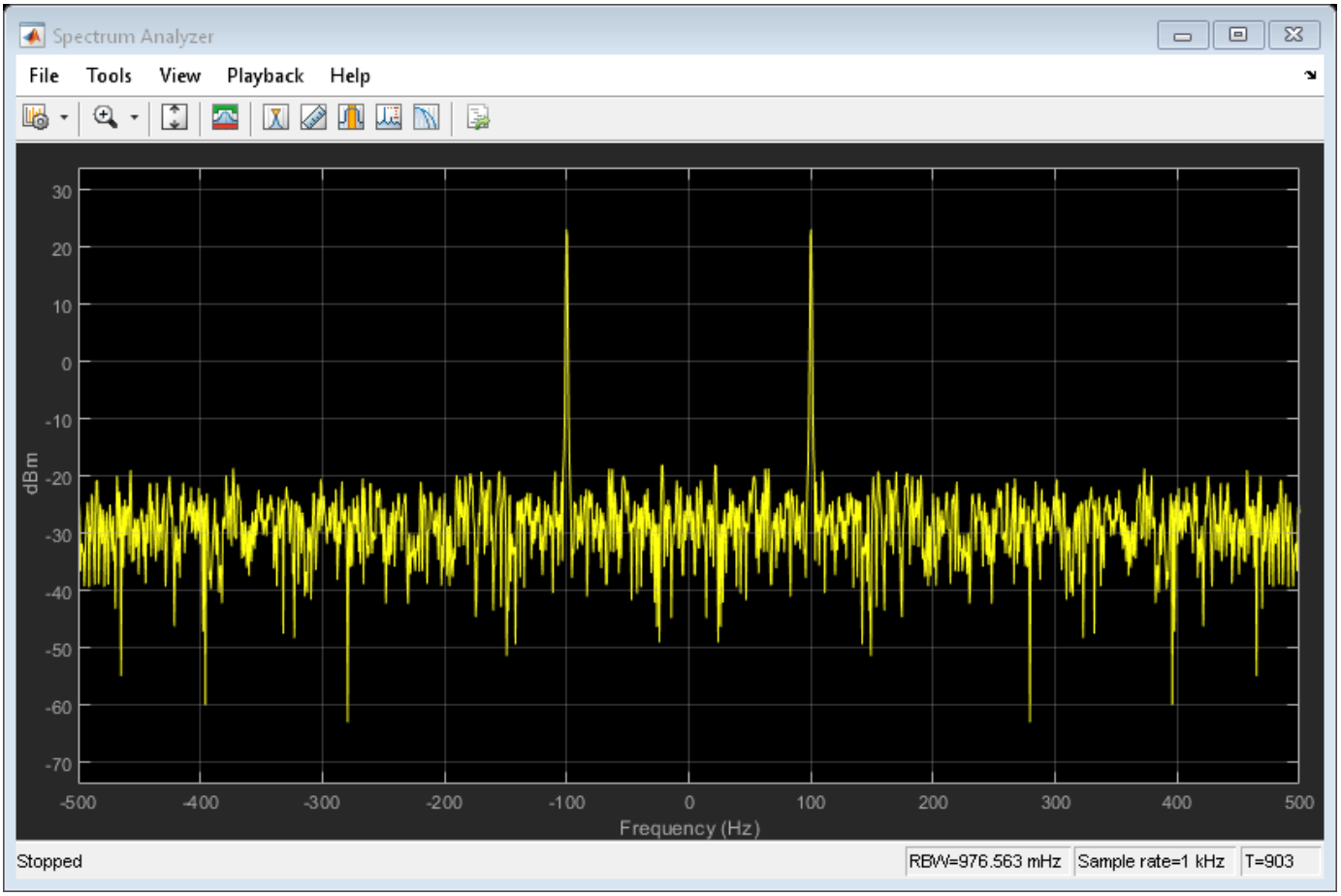
This example shows how to visualize and measure signals in the time and frequency domain in MATLAB® using a time scope and spectrum analyzer.

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 = timescope('SampleRate',Fs,...
    'TimeSpanSource','property',...
    'TimeSpan',0.1,...
    'YLimits',[-2, 2],...
    'ShowGrid',true);
SA = dsp.SpectrumAnalyzer('SampleRate',Fs);
tic;
while toc < 5
  sigData = SW() + 0.05*randn(SampPerFrame,1);
  SA(sigData);
  TS(sigData);
end
release(TS)
release(SA)
```


SCOPE
MEASUREMENTS
TRIGGER
2
1.6
1.2
0.8
0.4
0
-0.4
-0.8
-1.2
-1.6
-2
Amplitude
903.9
903.92
903.94
903.96
903.98
Time (s)
Stopped
Frames = 904 T = 903.999



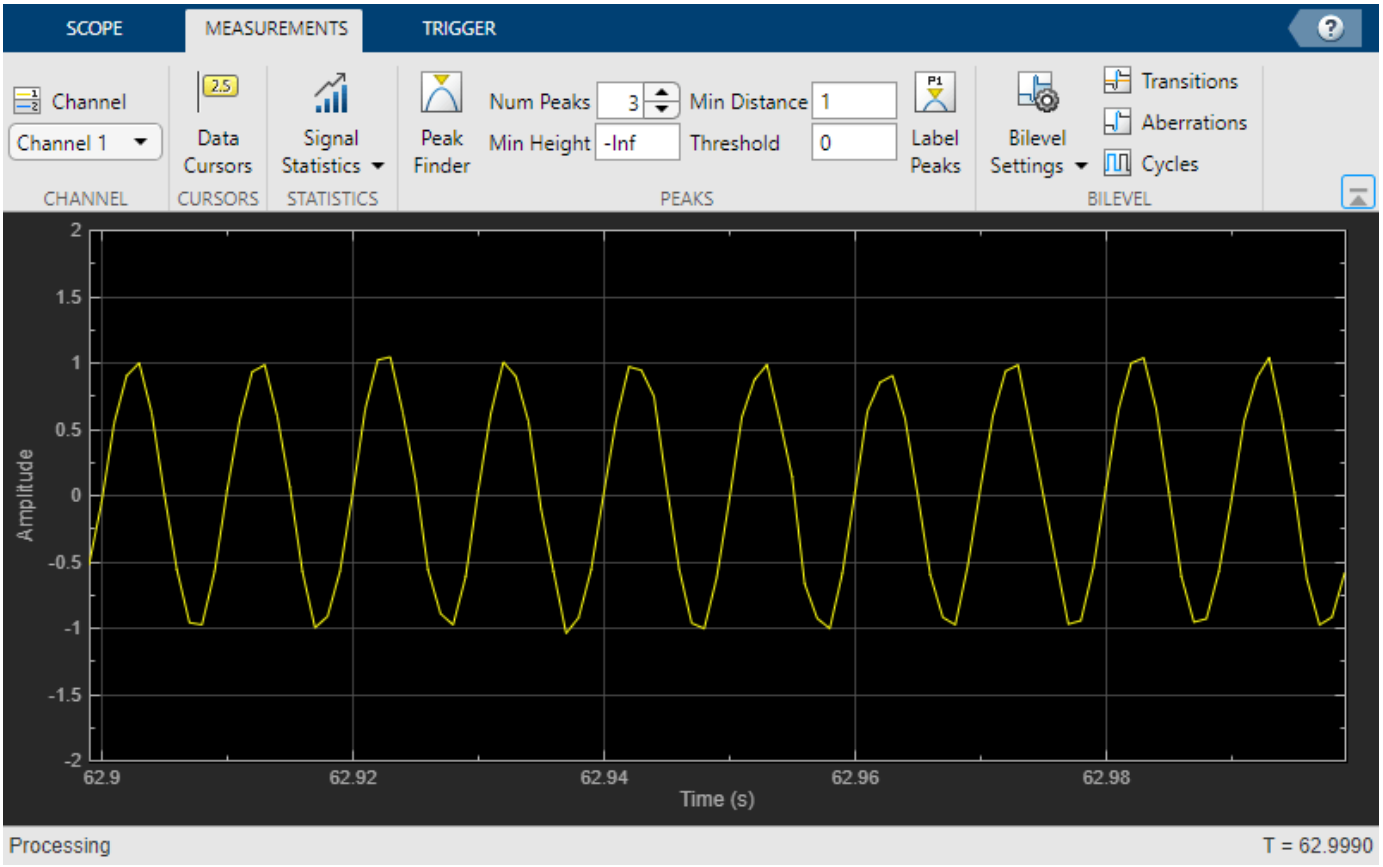
Time-Domain Measurements

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, aberrations, and cycles.
- **Peak Finder** - displays maxima and the times at which they occur.

You can enable and disable these measurements from the **Measurements** tab.

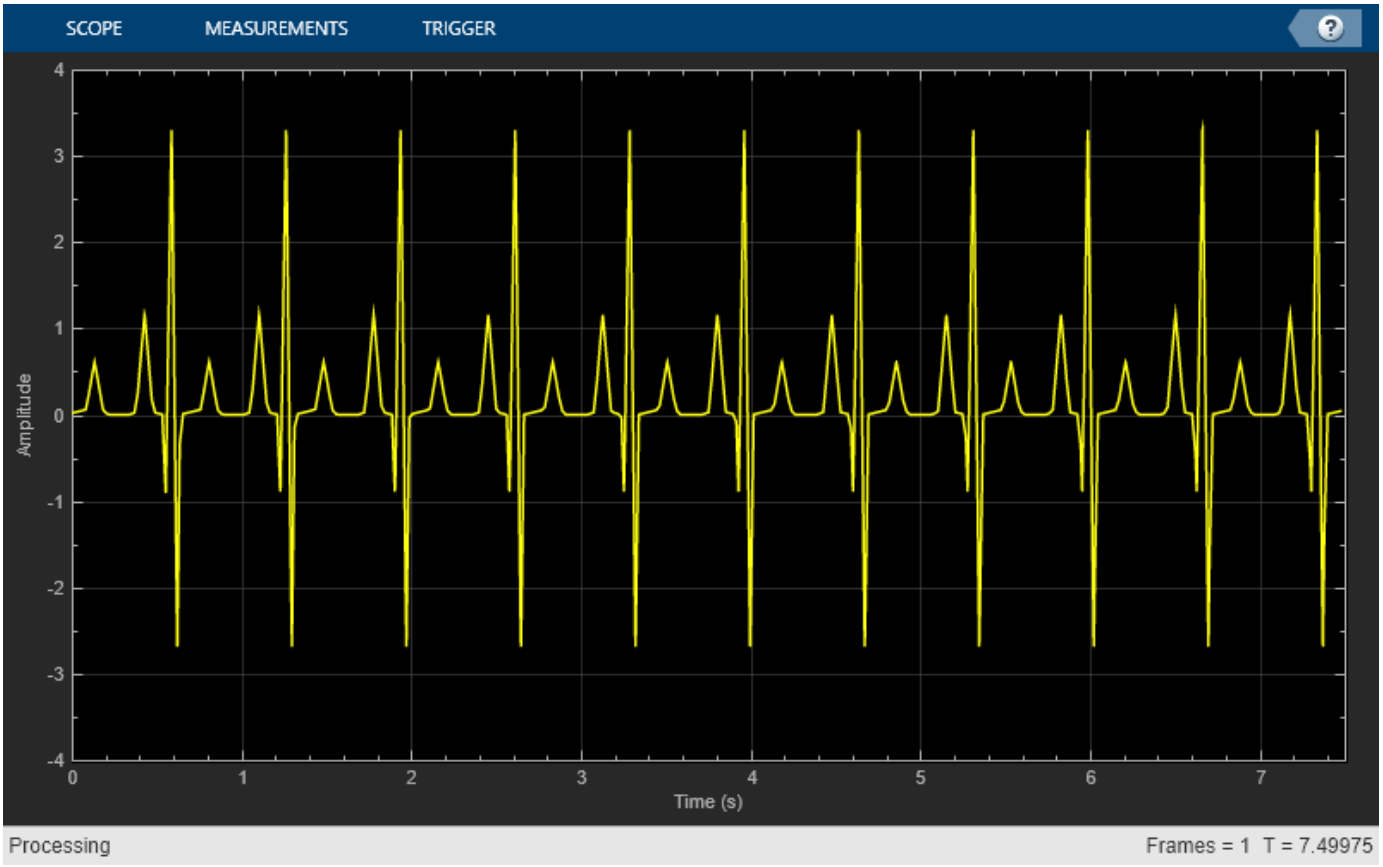


To illustrate the use of measurements in the time scope, simulate an ECG signal. Use the `ecg` function to generate 2700 samples of the signal. Use a Savitzky-Golay filter to smooth the signal and periodically extend the data to obtain approximately 11 periods.

```
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 and Data Cursor measurements. Assume a sample rate of 4 kHz.

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



Peak Measurements

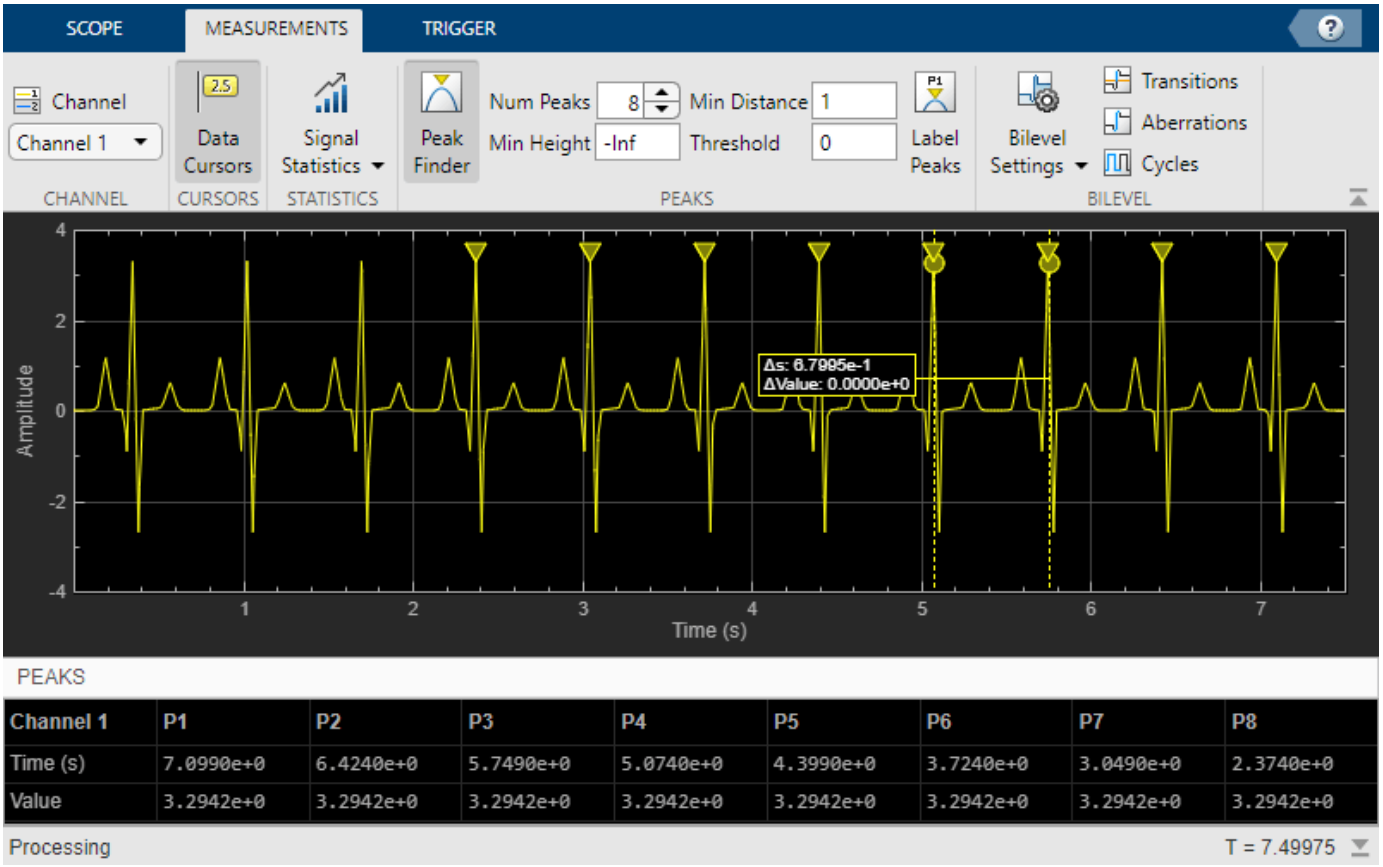
Enable **Peak Measurements** from the **Measurements** tab by clicking the corresponding toolstrip button. The Peaks pane appears at the bottom of the time scope window. For the **Num Peaks** property, enter 8 and press Enter. In the Peaks pane, the time scope displays a list of 8 peak amplitude values 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** from the **Measurements** tab by clicking the corresponding toolstrip button. The cursors appear on the time scope with a box showing the change in time and value between the two cursors. You can drag the cursors and use them to measure the time between events in the waveform. As you drag a cursor, the time an value at the cursor appears. This figure shows how to use cursors to measure the time interval between peaks in the ECG waveform. The ΔT measurement in the cursor box 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 various bilevel measurements from the **Measurements** tab. 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, settling time, pulse width, and duty cycle. To read more about these measurements, see “Configure Time Scope MATLAB Object” on page 26-80.

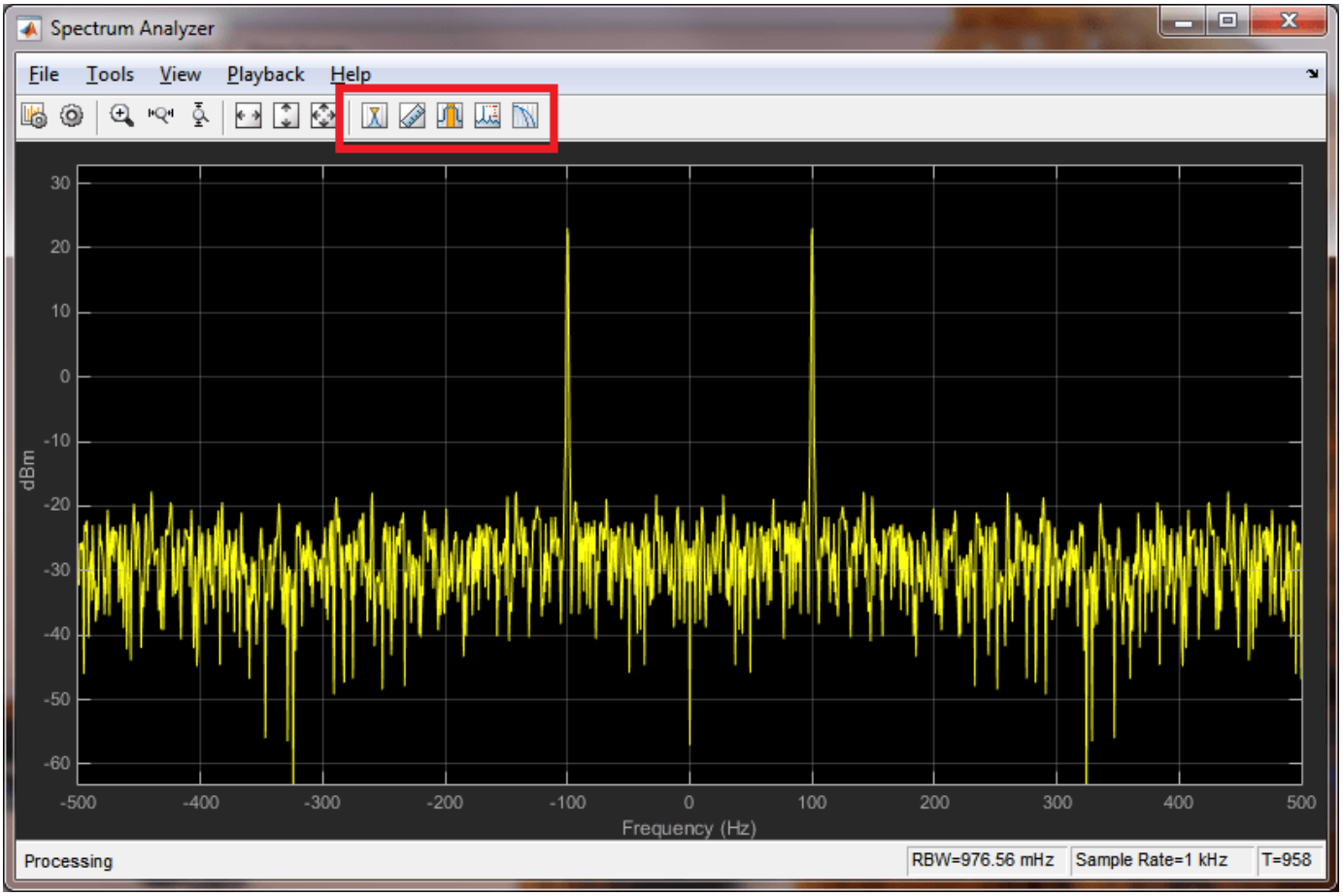
Frequency-Domain Measurements

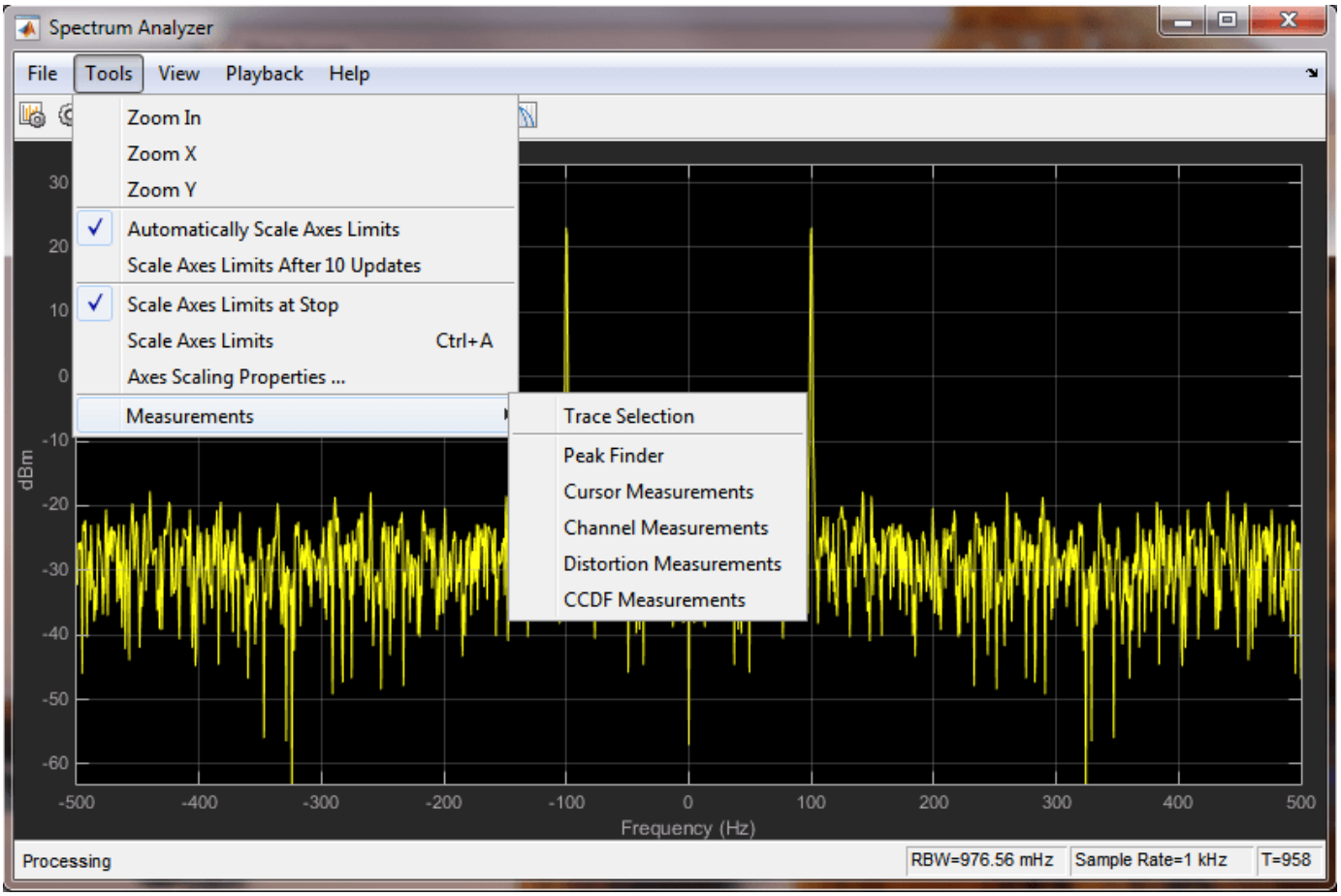
This section explains how to make frequency domain measurements with the spectrum analyzer.

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.


Spectrum Analyzer
File Tools View Playback Help
30
20
10
0
-10
-20
-30
-40
-50
-60
dBm
-500 -400 -300 -200 -100 0 100 200 300 400 500
Frequency (Hz)
Processing
RBW=976.56 mHz
Sample Rate=1 kHz
T=958



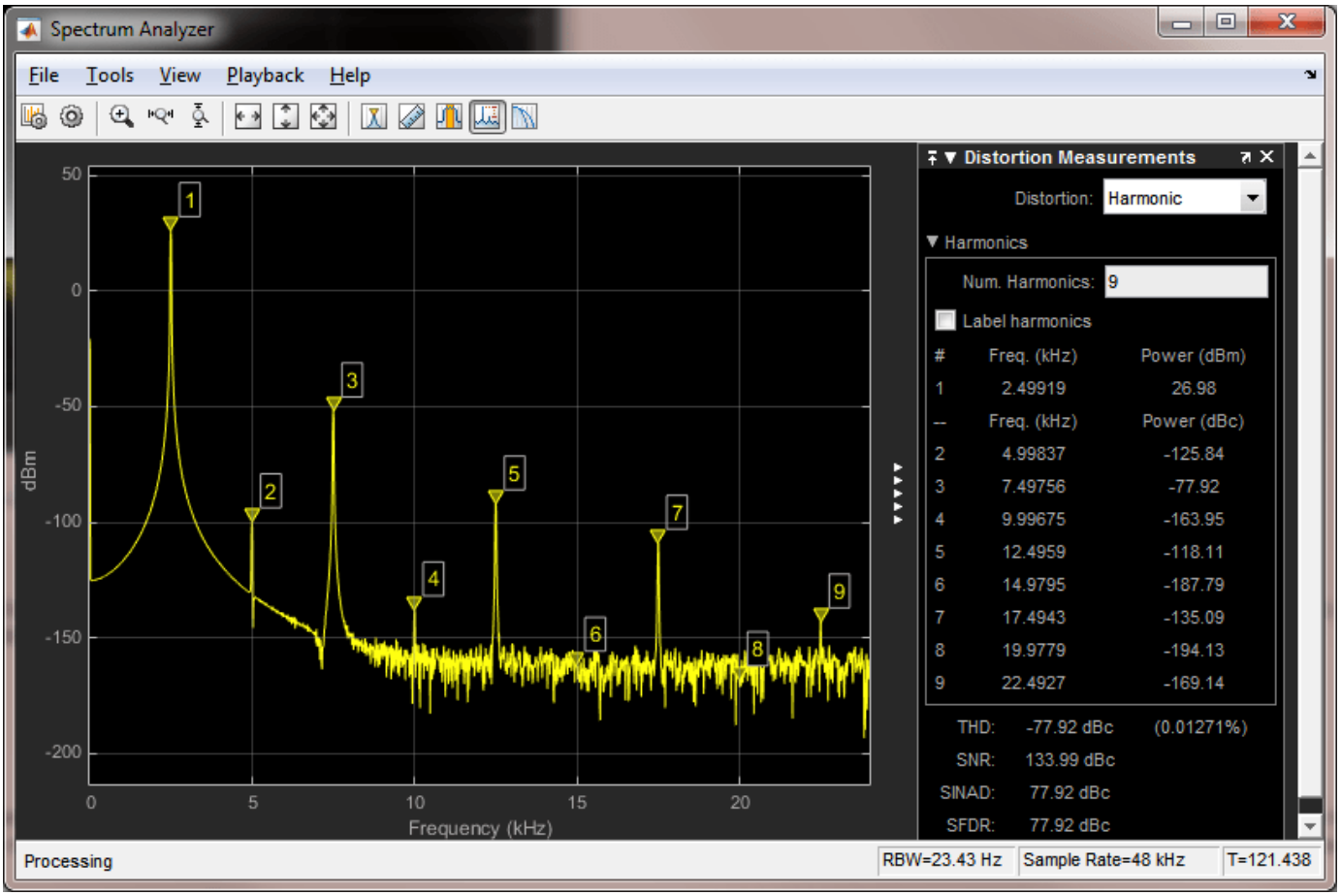
Distortion Measurements

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

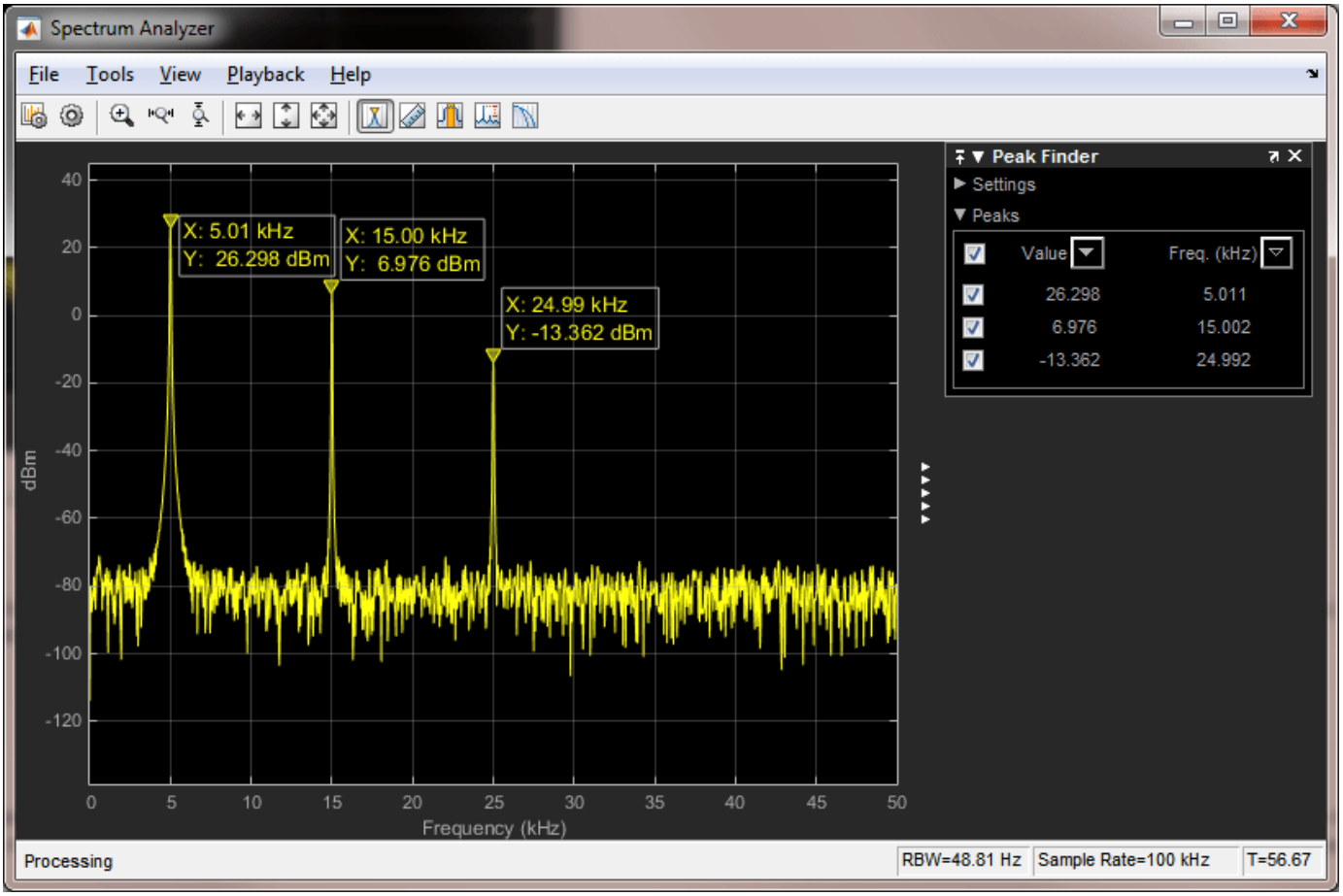
You can track time-varying spectral components by using the Peak Finder measurement dialog. You can show and optionally label up to 100 peaks. You can invoke the Peak Finder dialog from the **Tools > Measurements > Peak Finder** menu item, or by clicking the corresponding icon in the toolbar.

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” on page 4-290 example.

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

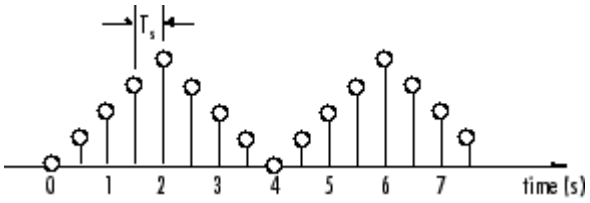
Discrete-Time Signals

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, $f = 1/T$.

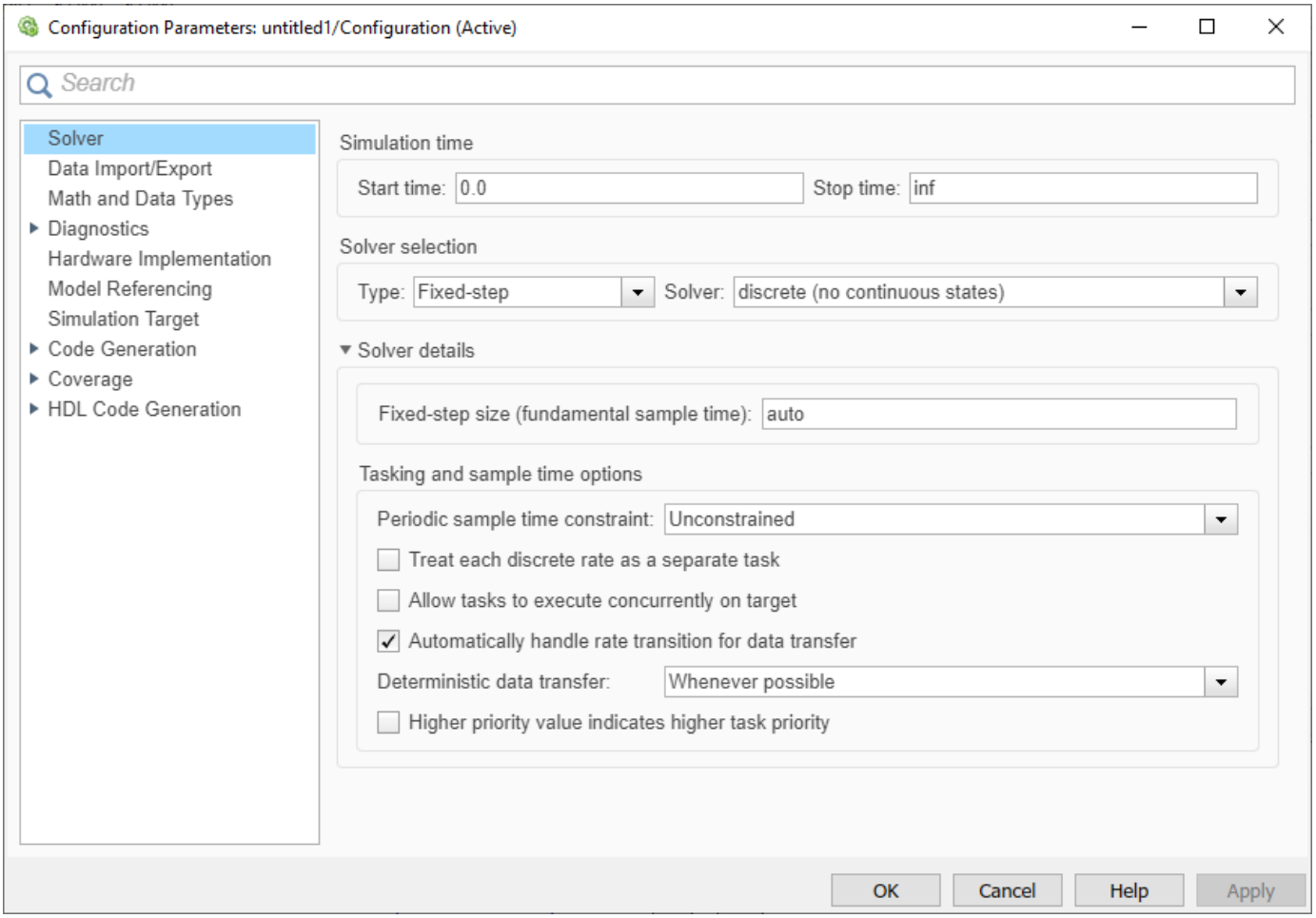
Term	Symbol	Units	Notes
Nyquist rate		Hz (cycles per second)	The minimum sample rate that avoids aliasing, usually twice the highest frequency in the signal being sampled.
Nyquist frequency	f_{nyq}	Hz (cycles per second)	Twice the highest frequency present in the signal.
Normalized frequency	f_n	Two 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:

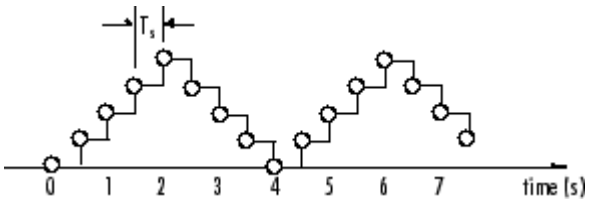
- If your model contains one sample time

- 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

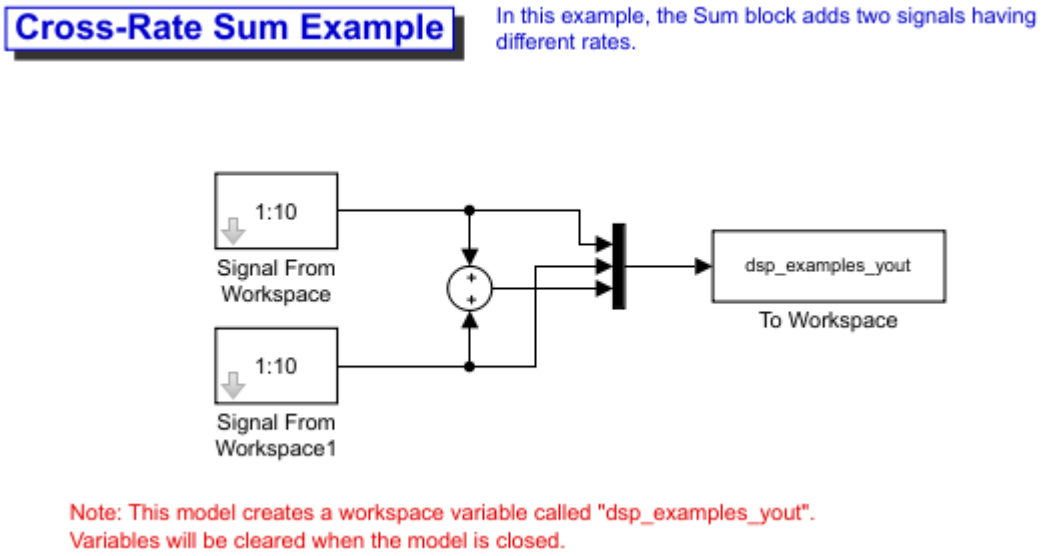
When the fixed-step, multitasking solver is selected, discrete signals in Simulink are undefined between sample times. Therefore, to perform cross-rate operations like the addition of two signals with different sample rates, you must convert the two signals to a common sample rate. Several

blocks in the Signal Operations and Multirate Filters libraries can accomplish this task. See “Convert Sample and Frame Rates in Simulink Using Rate Conversion Blocks” on page 3-16 for more information. Rate change can occur implicitly depending on the diagnostic settings. However, this is not recommended. See “Multitask data transfer” (Simulink), “Single task data transfer” (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**

data transfer 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.

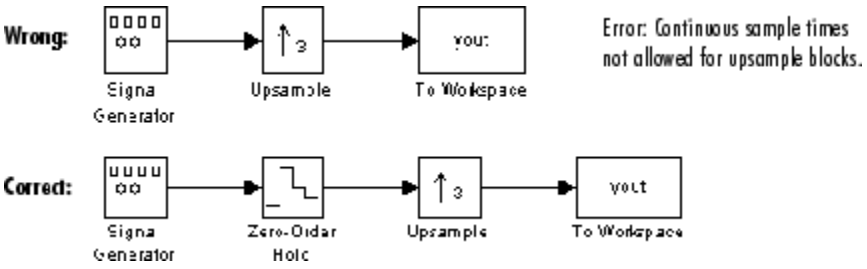
Continuous-Time Signals

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.



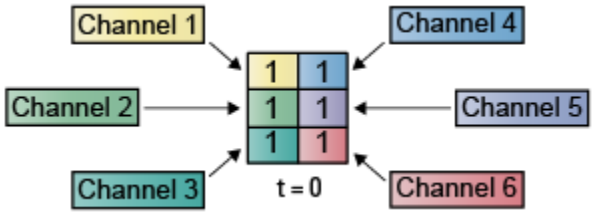
Continuous-Time Nonsource Blocks

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.

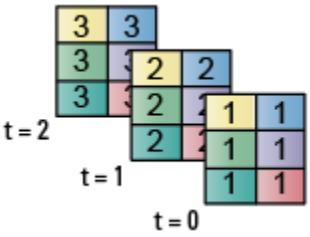
Create Signals for Sample-Based Processing

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.

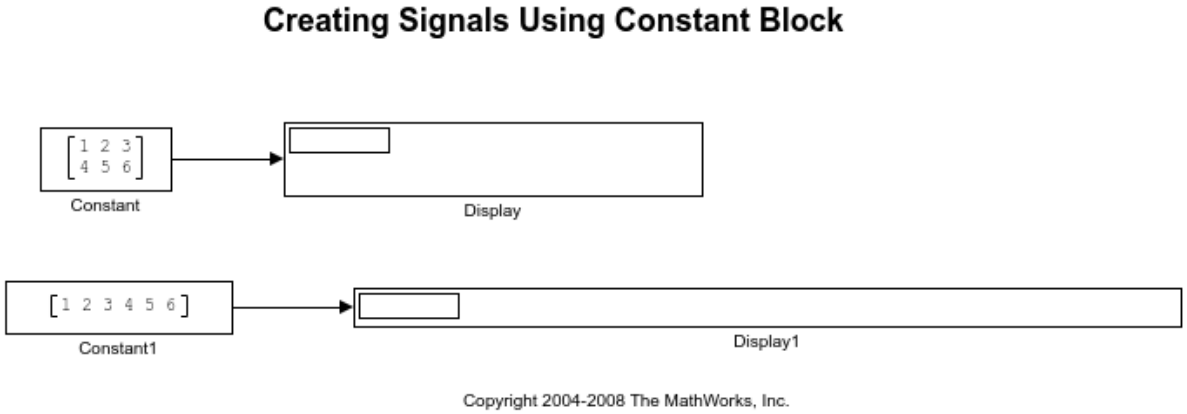
2 Double-click the Constant1 block, and set the block parameters as follows:

- **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:

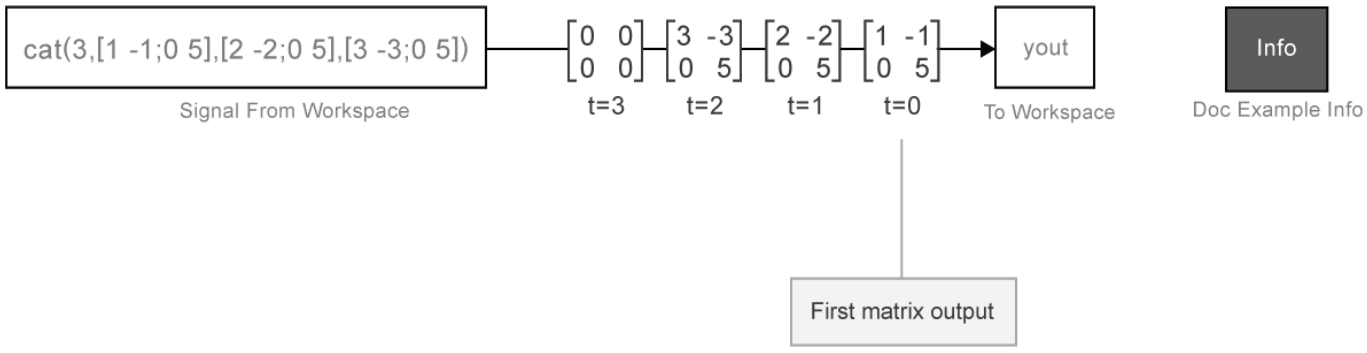
   - **Signal** = `cat(3,[1 -1;0 5],[2 -2;0 5],[3 -3;0 5])`
   - **Sample time** = `1`
   - **Samples per frame** = `1`
   - **Form output after final data value by** = `Setting to zero`

   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

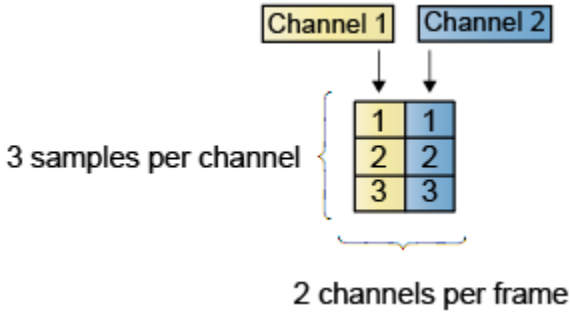
More About

- “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

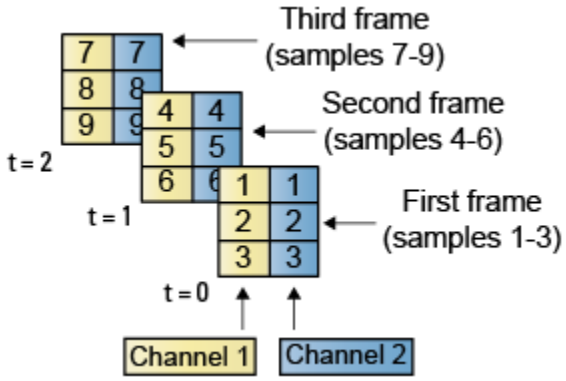
Create Signals for Frame-Based Processing

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

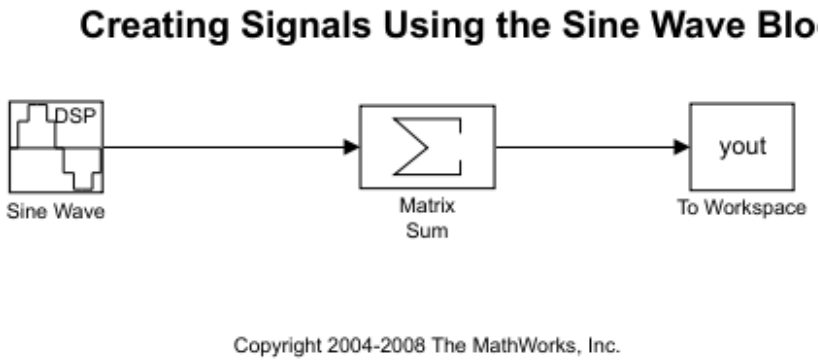
1. Create a new Simulink model.
2. From the Sources library, click-and-drag a Sine Wave block into the model.
3. From the Matrix Operations library, click-and-drag a Matrix Sum block into the model.
4. From the Simulink Sinks library, click-and-drag a To Workspace block into the model.
5. Connect the 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.

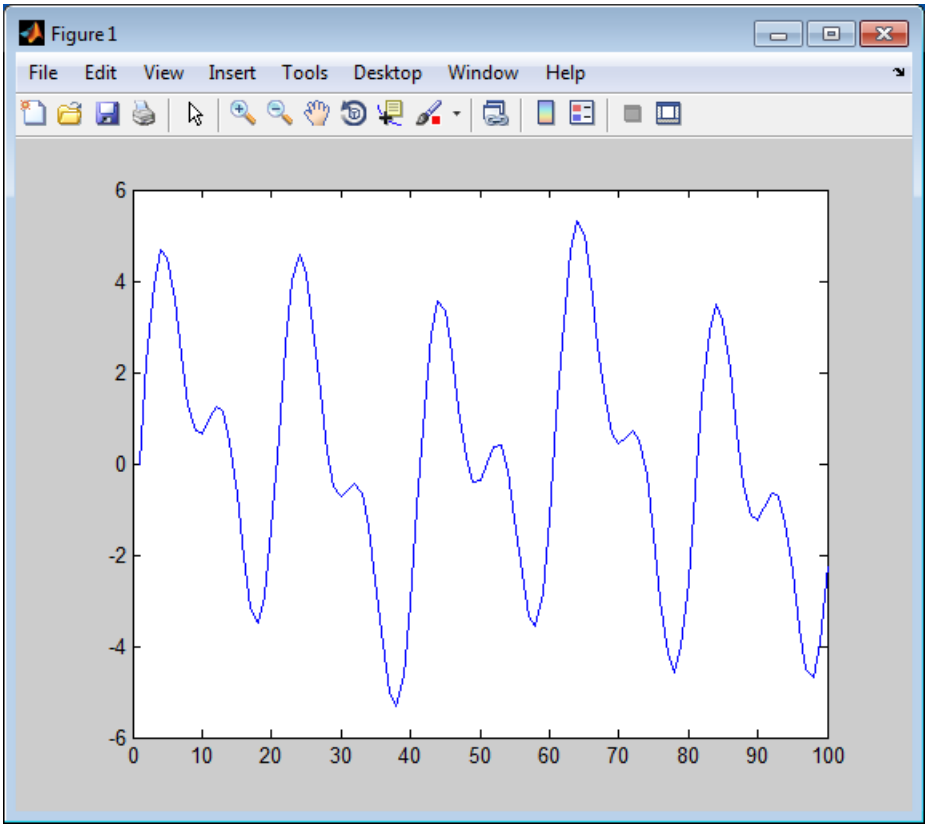
    Your model should now look similar to the following figure. You can also open the model by typing `ex_usingsinwaveblkfb` at the MATLAB command line.



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:

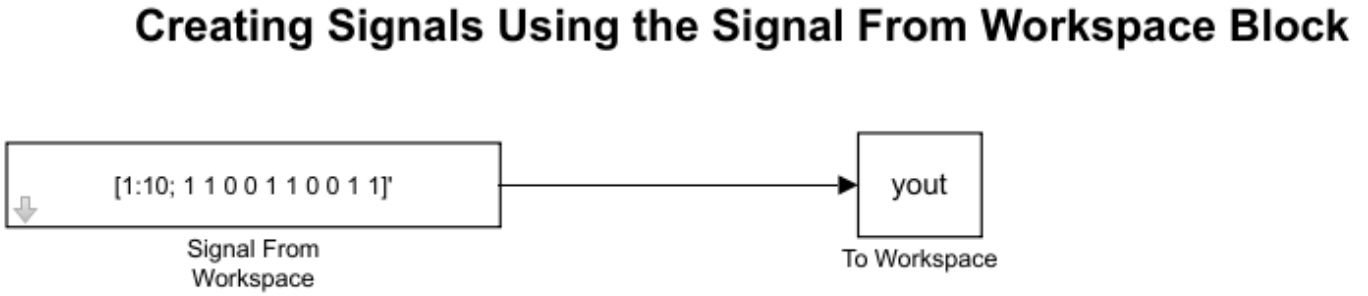
- Channel 1: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 0, 0,...
- Channel 2: 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0,...

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

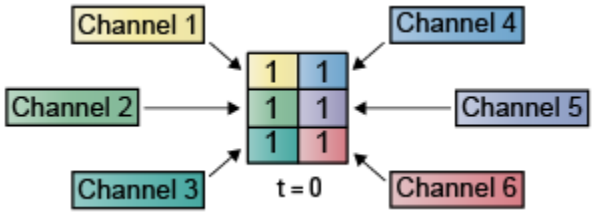
More About

- “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

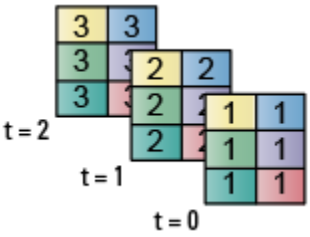
Create Multichannel Signals for Sample-Based Processing

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.

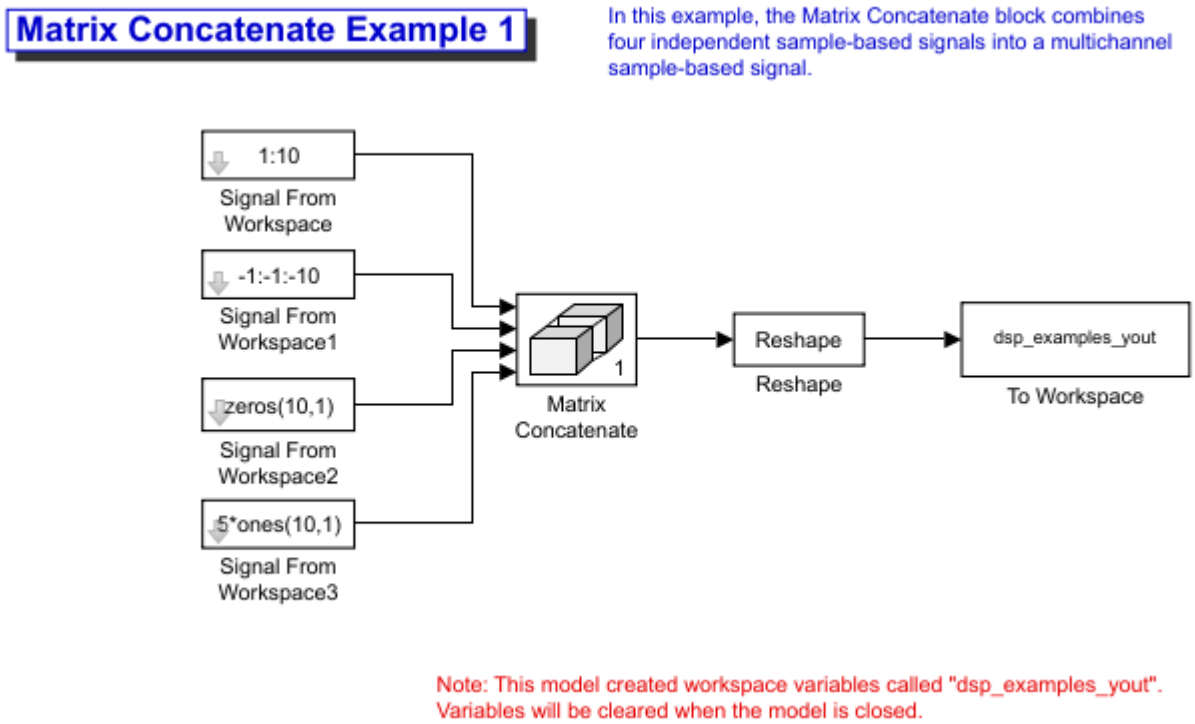
Create Multichannel Signals by Combining Single-Channel Signals

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`

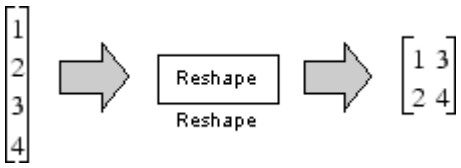
- **Output dimensions** = `[2,2]`

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`.

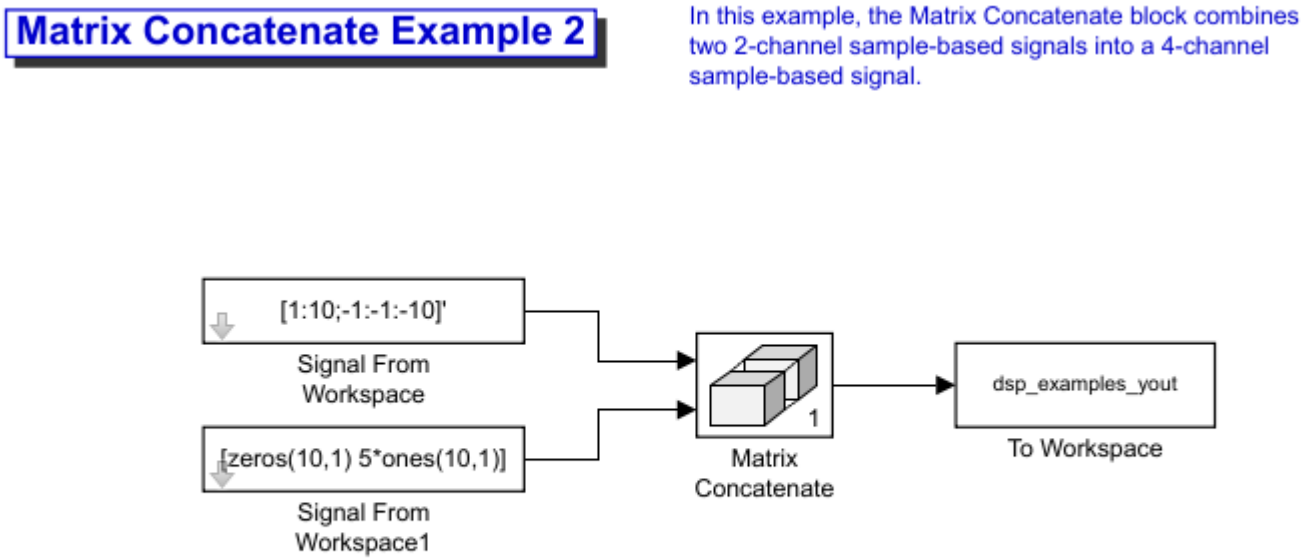
Create Multichannel Signals by Combining Multichannel Signals

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

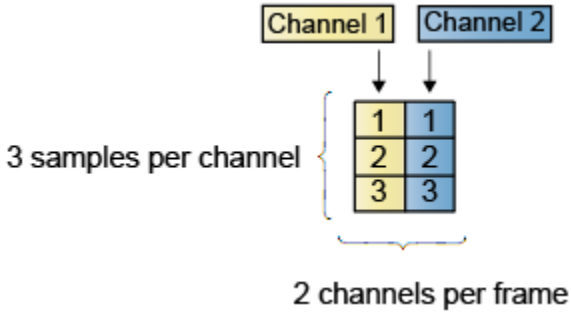
- “Sample- and Frame-Based Concepts” on page 3-2
- “Create Signals for Sample-Based Processing” on page 2-9
- “Create Signals for Frame-Based Processing” on page 2-13
- “Create Multichannel Signals for Frame-Based Processing” on page 2-23

- “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

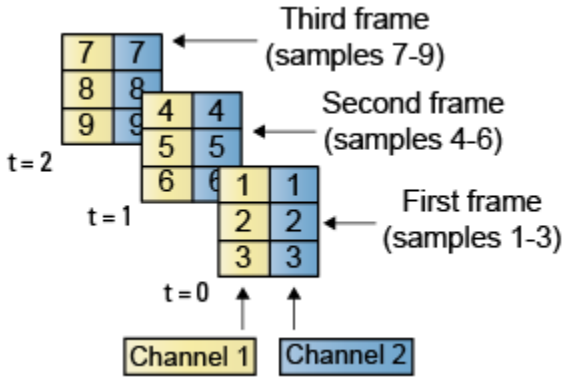
Create Multichannel Signals for Frame-Based Processing

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.



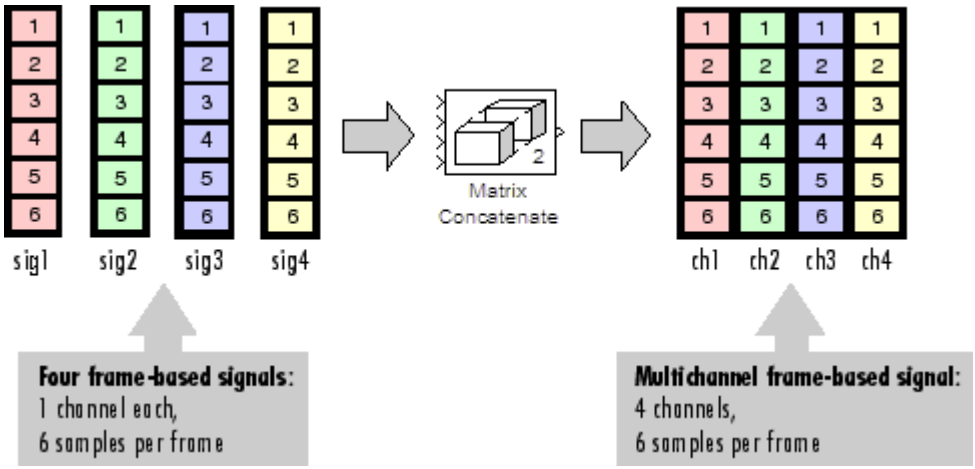
Using frame-based processing is advantageous for many signal processing applications because you can process multiple samples at once. By buffering your data into frames and processing multisample frames of data, you can often improve the computational time of your signal processing algorithms. To perform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



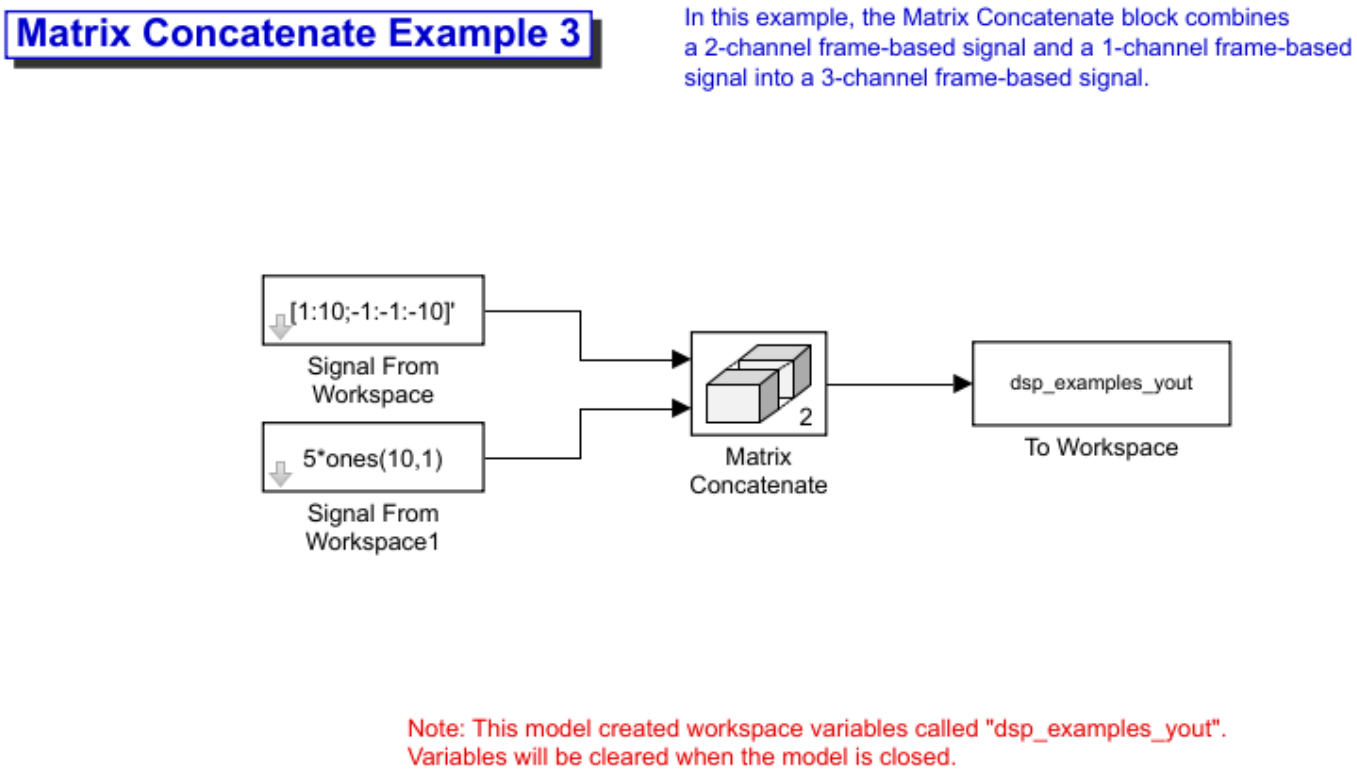
Create Multichannel Signals Using Concatenate Block

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

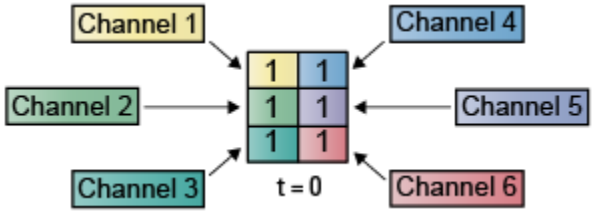
More About

- “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

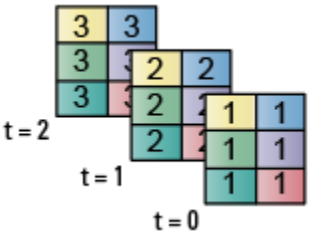
Deconstruct Multichannel Signals for Sample-Based Processing

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.



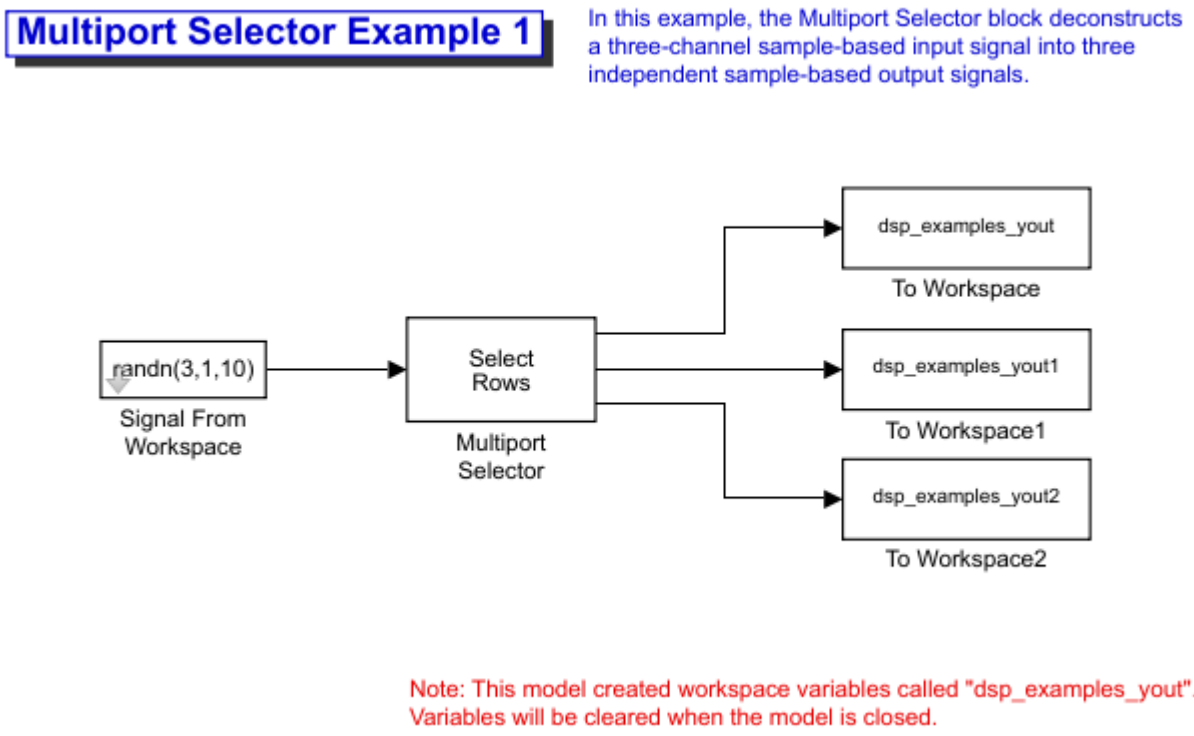
For more information about the recent changes to frame-based processing, see the “Frame-based processing changes” section of the *DSP System Toolbox Release Notes*.

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.

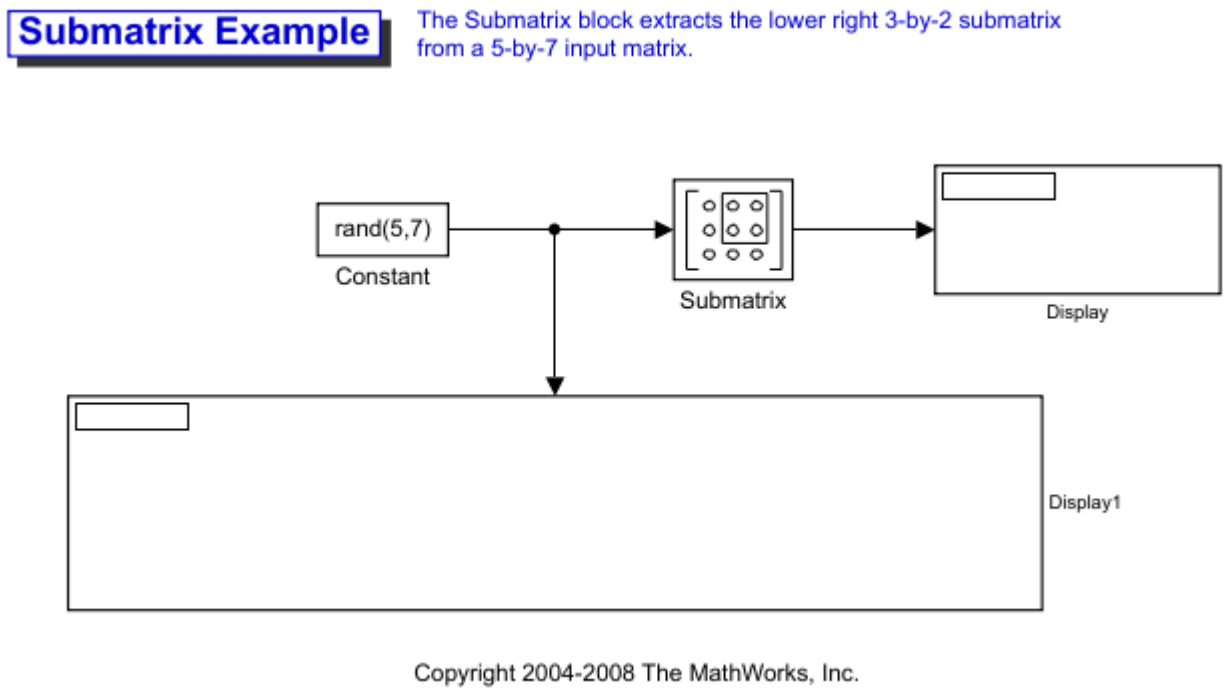
You have now successfully created three single-channel signals from a multichannel signal using a Multiport Selector block.

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**.

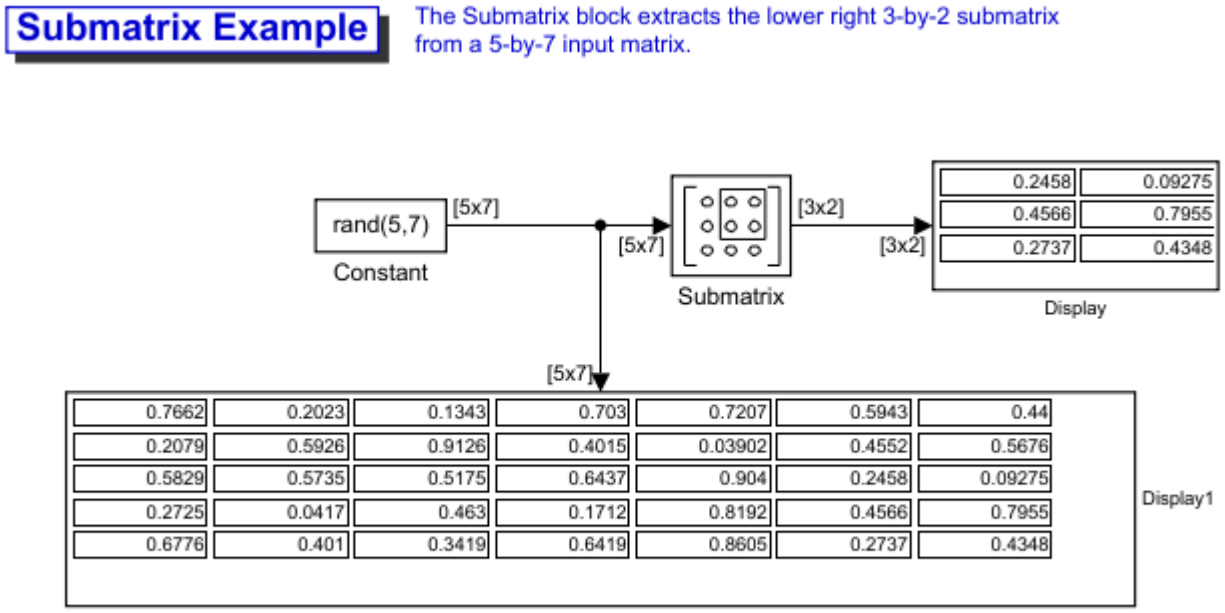
4 Double-click the Submatrix block. Set the block parameters as follows, and then click **OK**:

- **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.



Copyright 2004-2008 The MathWorks, Inc.

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

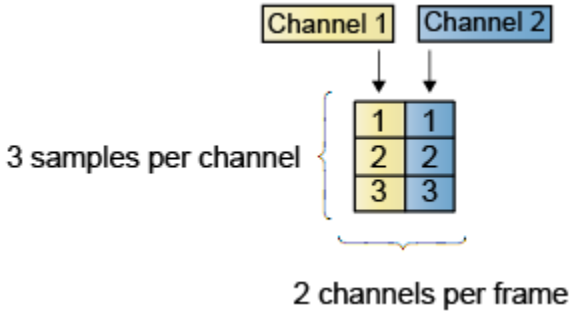
More About

- “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 Using Rate Conversion Blocks” on page 3-16

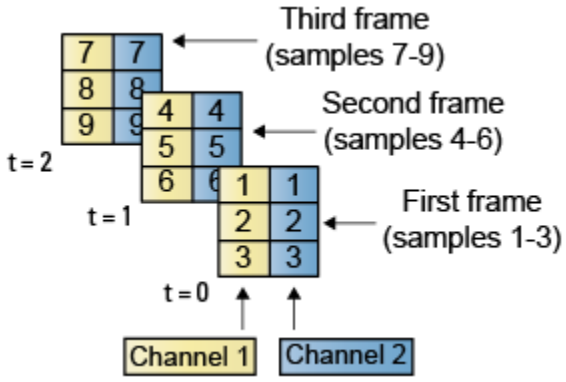
Deconstruct Multichannel Signals for Frame-Based Processing

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.

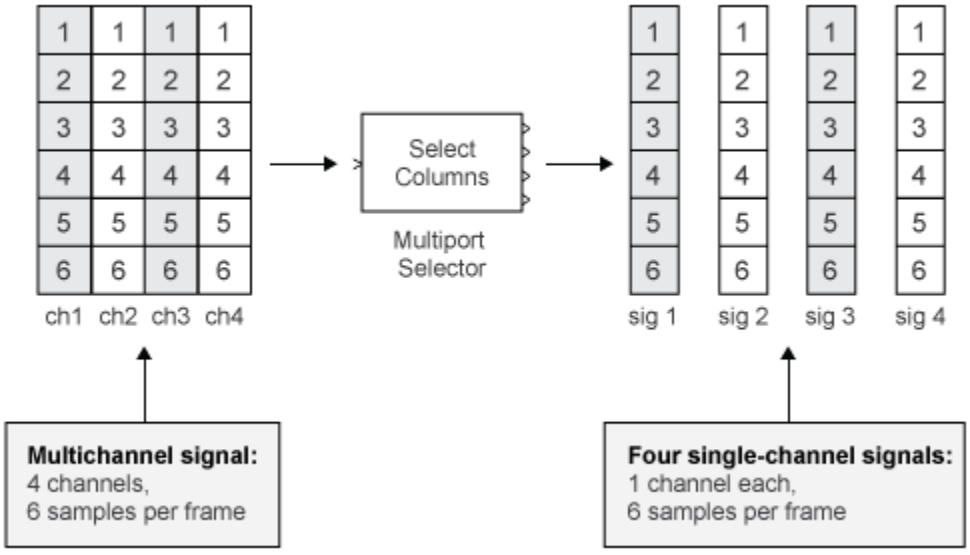
For more information about the recent changes to frame-based processing, see the “Frame-based processing changes” section of the *DSP System Toolbox Release Notes*.

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.

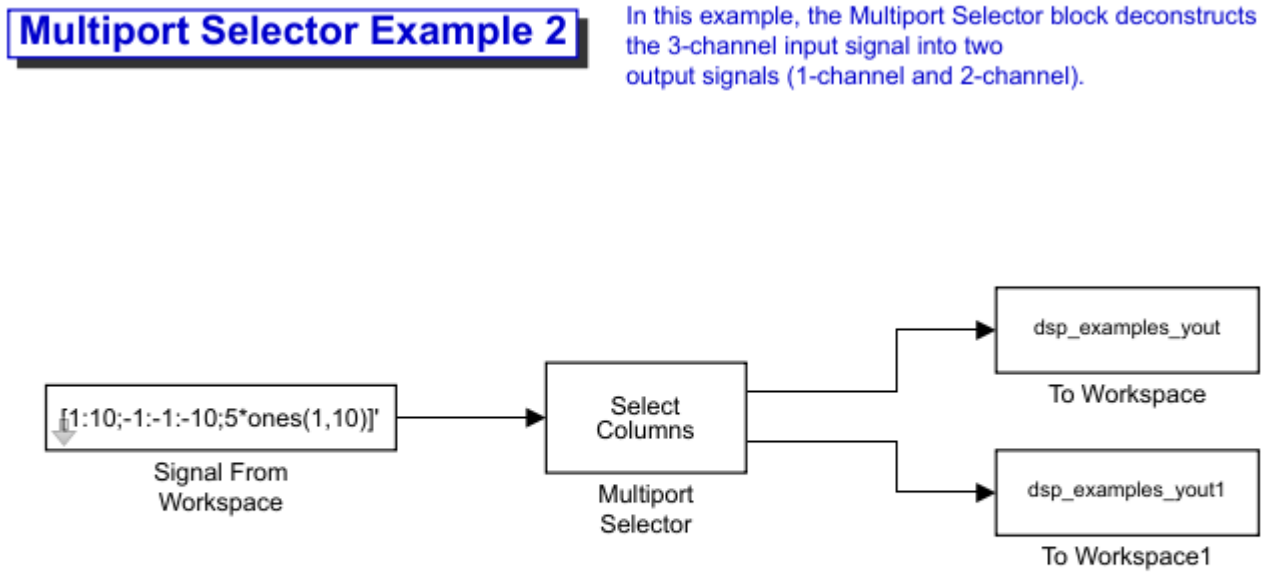
The figure below is a graphical representation of this process.



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**.

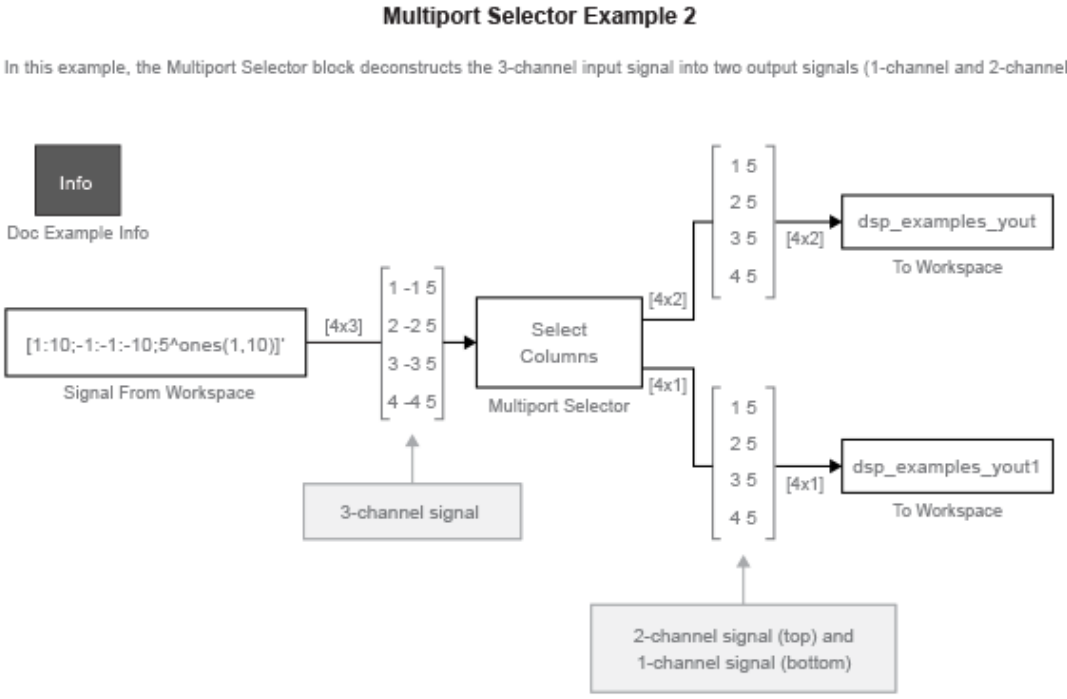
4 Double-click the Multiport Selector block. Set the block parameters as follows, and then click **OK**:

   - **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.

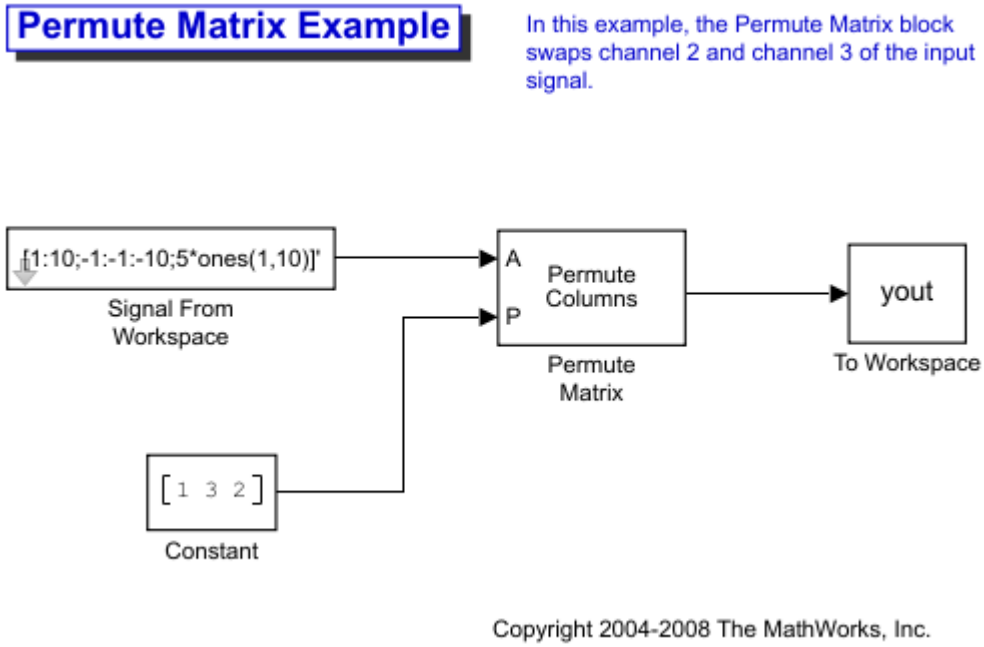
You have now successfully created a single-channel signal and a two-channel signal from a multichannel signal using the Multiport Selector block.

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.

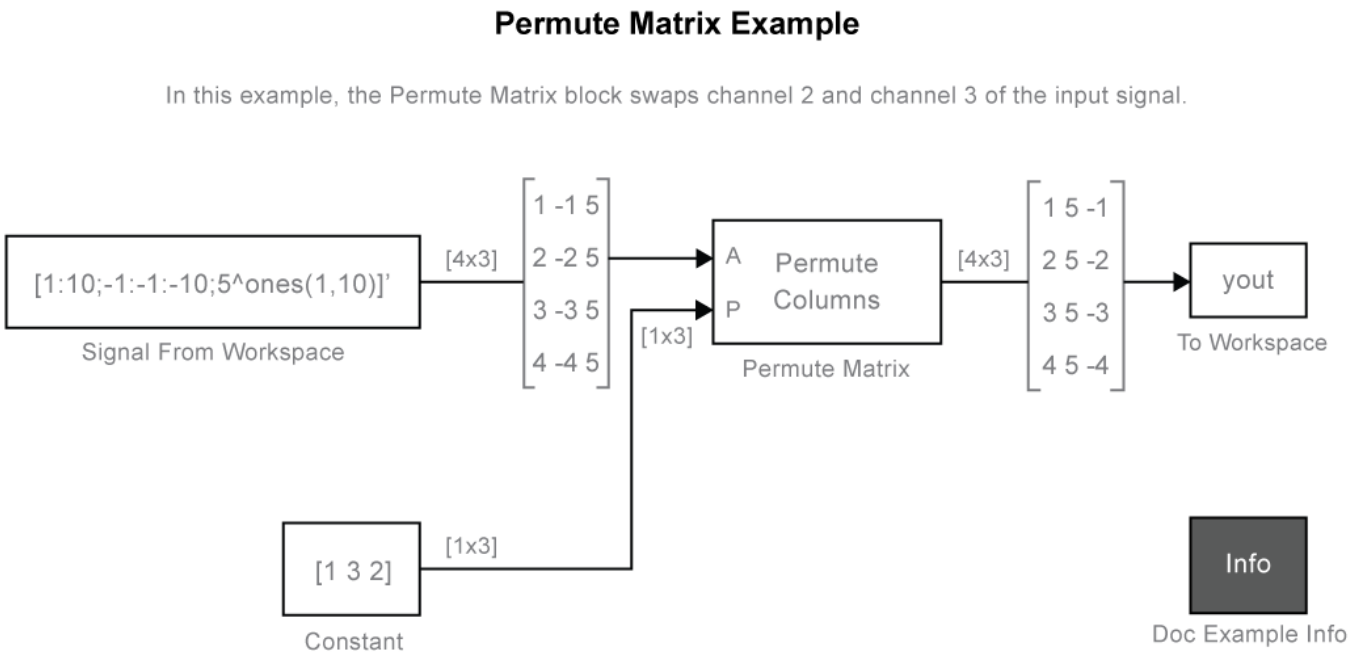
5 Double-click the Permute Matrix block. Set the block parameters as follows, and then click **OK**:

- **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

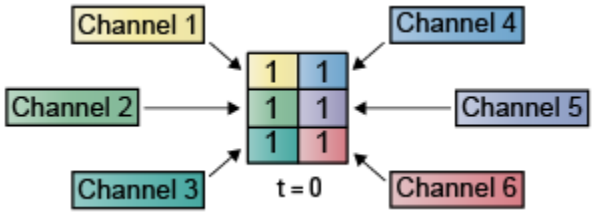
More About

- “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 Using Rate Conversion Blocks” on page 3-16

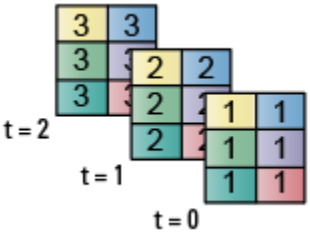
Import and Export Signals for Sample-Based Processing

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.

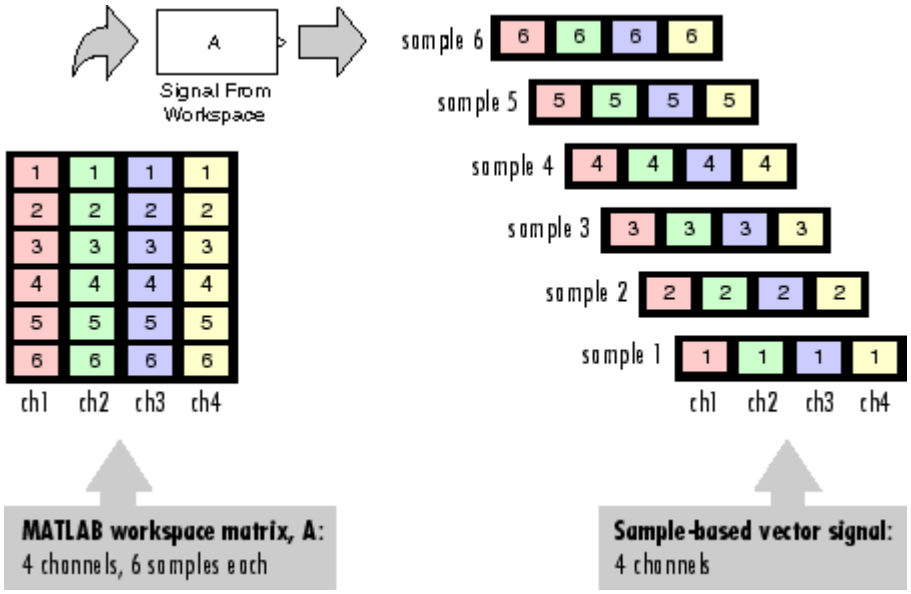


For more information about the recent changes to frame-based processing, see the “Frame-based processing changes” section of the *DSP System Toolbox Release Notes*.

Import Vector Signals for Sample-Based Processing

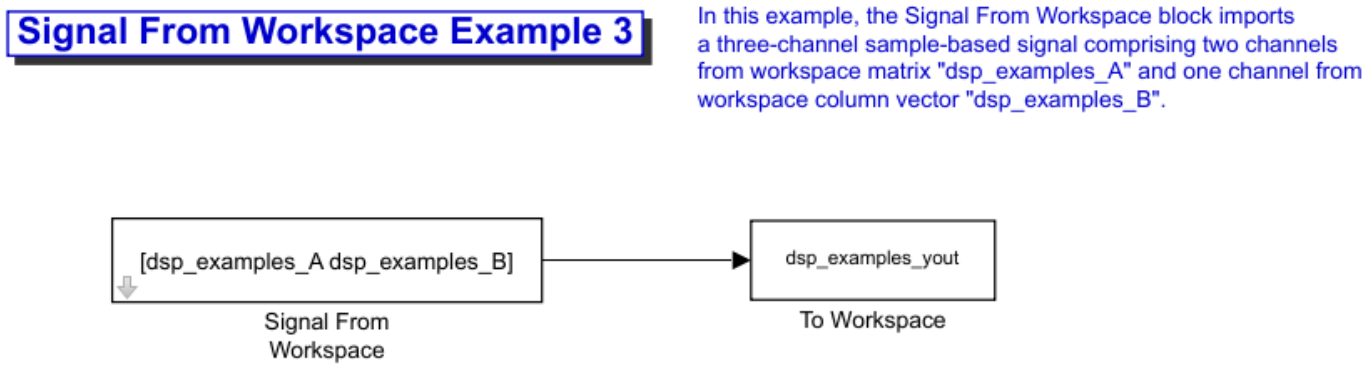
The Signal From Workspace block generates a vector signal for sample-based processing when the variable or expression in the **Signal** parameter is a matrix and the **Samples per frame** parameter is set to 1. Each column of the input matrix represents a different channel. Beginning with the first row 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]`

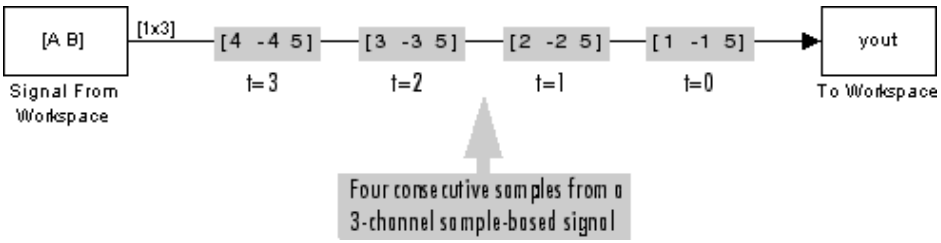
- **Sample time** = 1
- **Samples per frame** = 1
- **Form output after final data value** = `Setting to zero`

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.



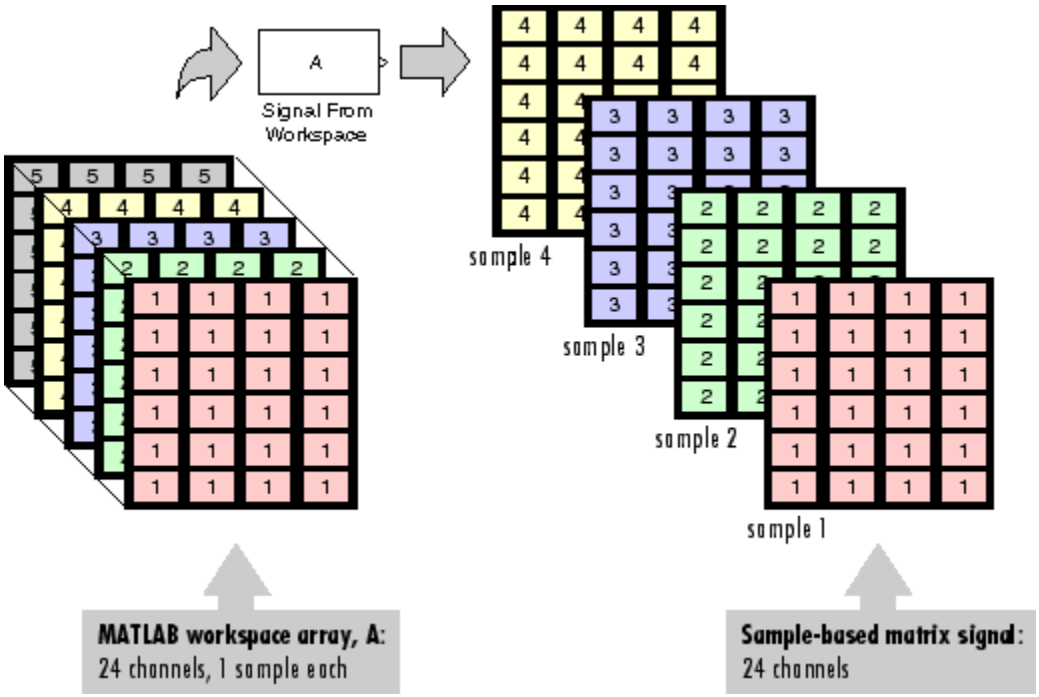
The first row of the input matrix `[A B]` is output at time `t=0`, the second row of the input matrix is output at time `t=1`, and so on.

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

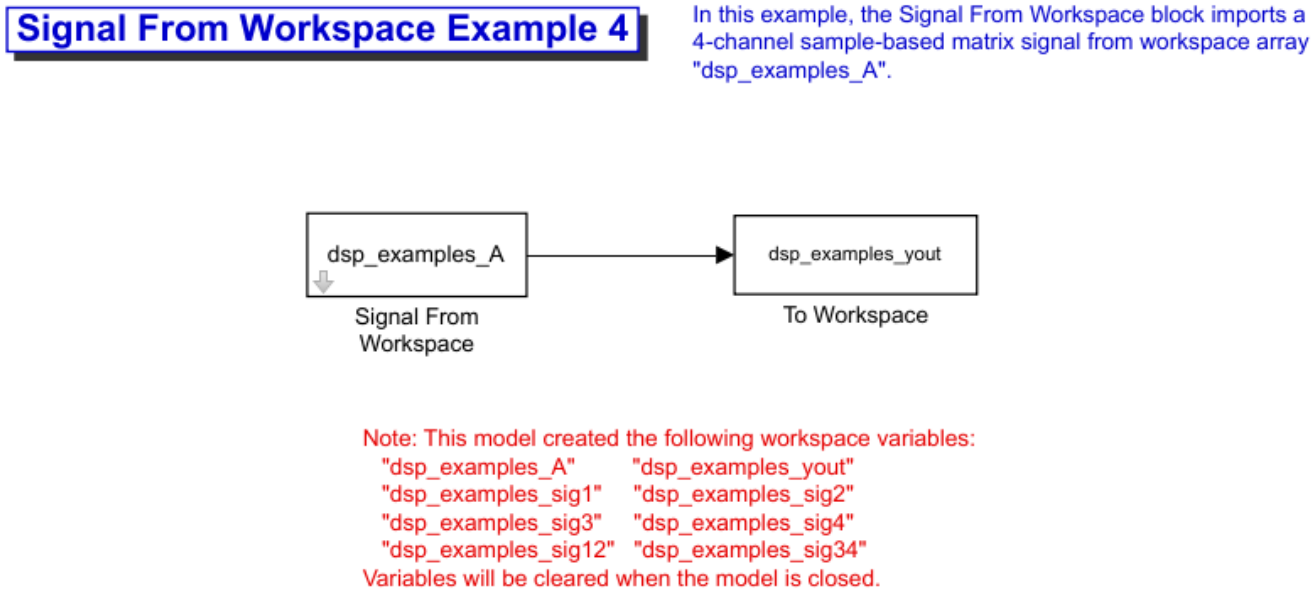
The Signal From Workspace block generates a matrix signal that is convenient for sample-based processing. Beginning with the first page of the array, the block outputs a single page of the array to the output at each sample time. Therefore, if the **Signal** parameter specifies an M-by-N-by-P array, 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**:

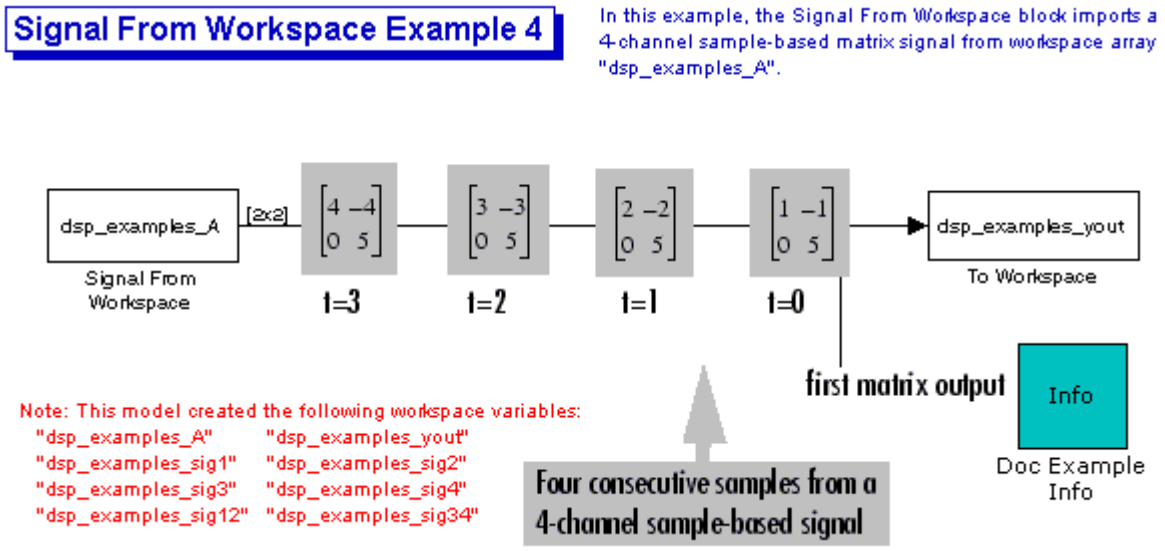
- **Signal** = `dsp_examples_A`
- **Sample time** = 1
- **Samples per frame** = 1
- **Form output after final data value** = `Setting to zero`

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.

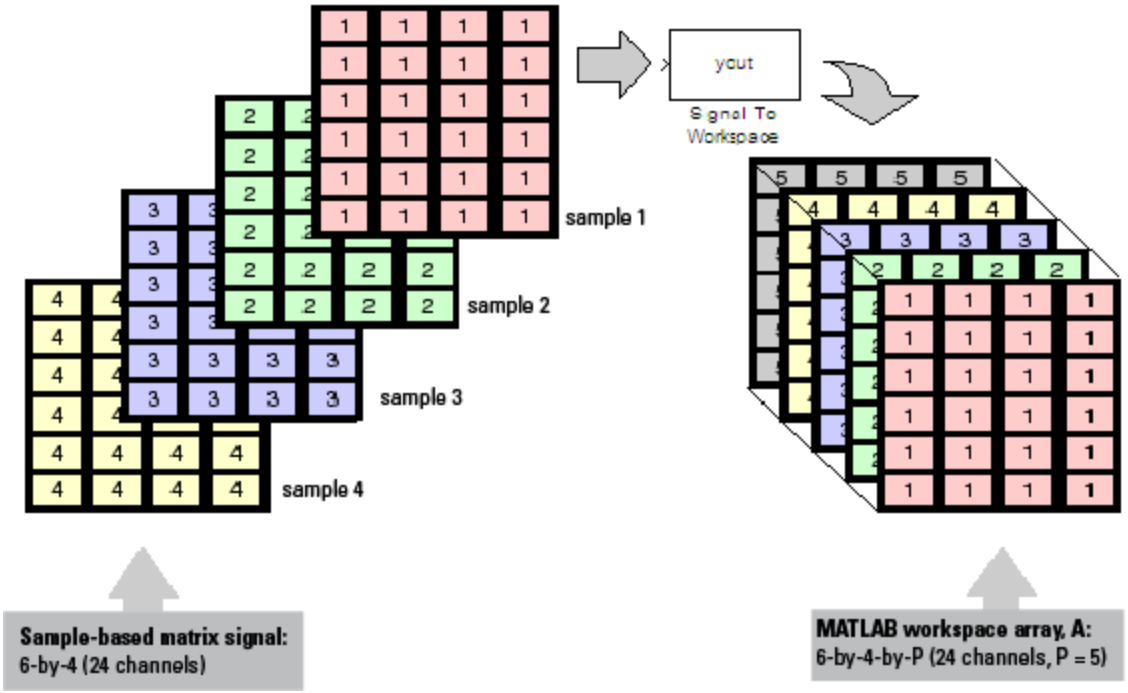
You have now successfully imported a 4-channel matrix signal into your model using the Signal From Workspace block.

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.

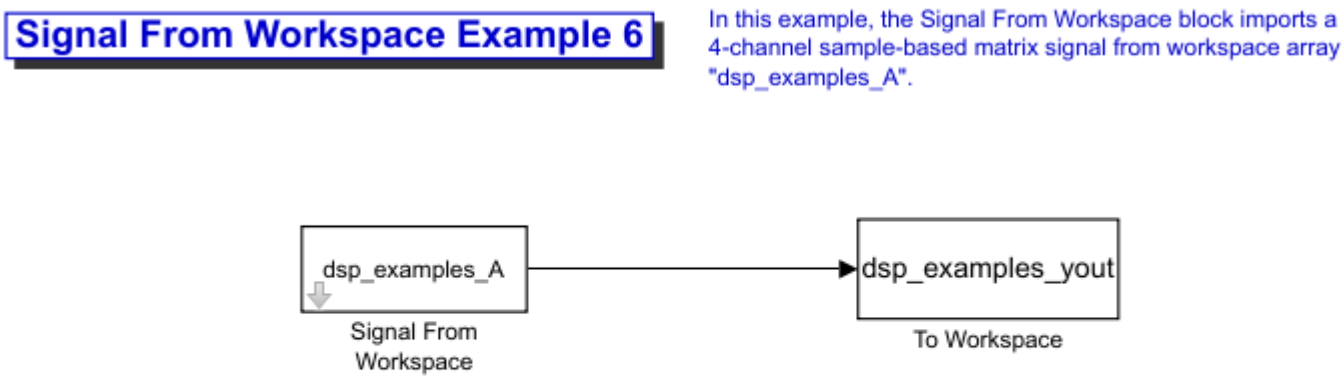
The following figure is the graphical illustration of this process using a 6-by-4 signal exported to workspace array A.



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

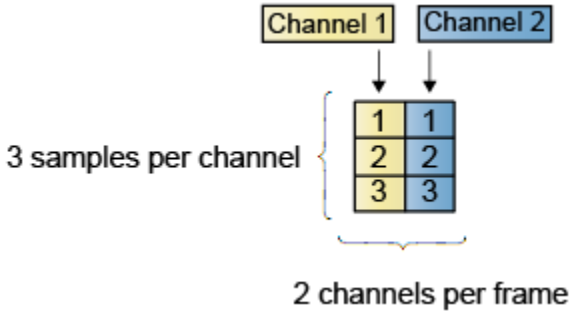
- “Sample- and Frame-Based Concepts” on page 3-2
- “Create Signals for Sample-Based Processing” on page 2-9
- “Create Signals for Frame-Based Processing” on page 2-13
- “Create Multichannel Signals for Sample-Based Processing” on page 2-18
- “Create Multichannel Signals for Frame-Based Processing” on 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 Using Rate Conversion Blocks” on page 3-16

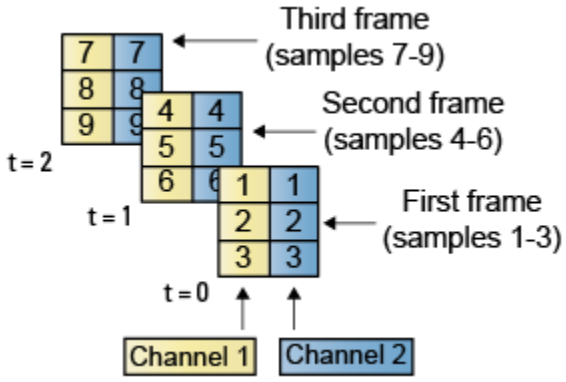
Import and Export Signals for Frame-Based Processing

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.



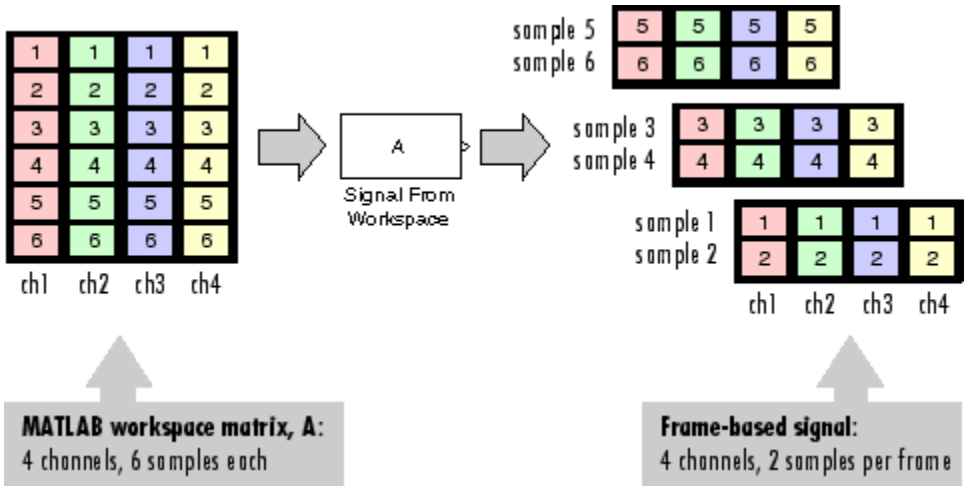
Using frame-based processing is advantageous for many signal processing applications because you can process multiple samples at once. By buffering your data into frames and processing multisample frames of data, you can often improve the computational time of your signal processing algorithms. To perform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

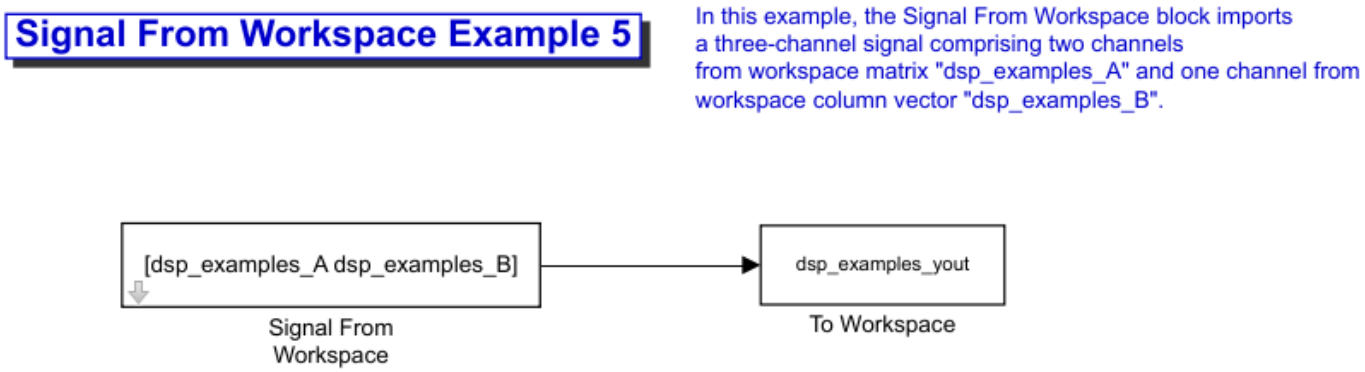
   ```
   ex_importfbsigs
   ```

   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:



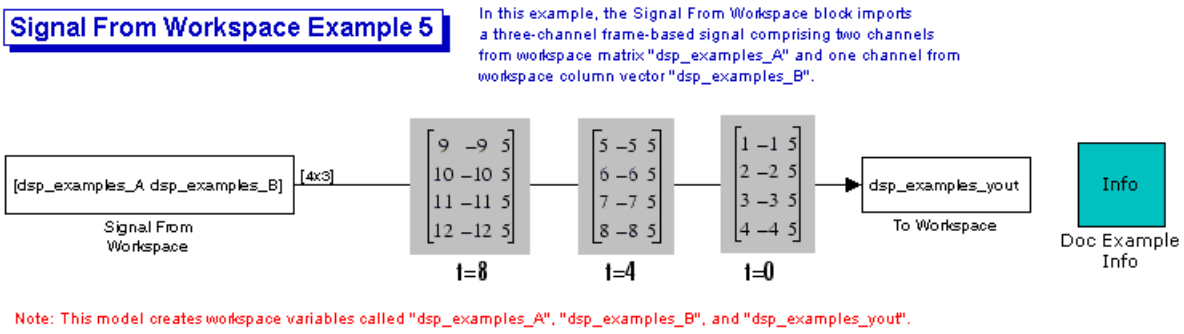
2 Double-click the Signal From Workspace block. Set the block parameters as follows, and then click **OK**:

- **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.



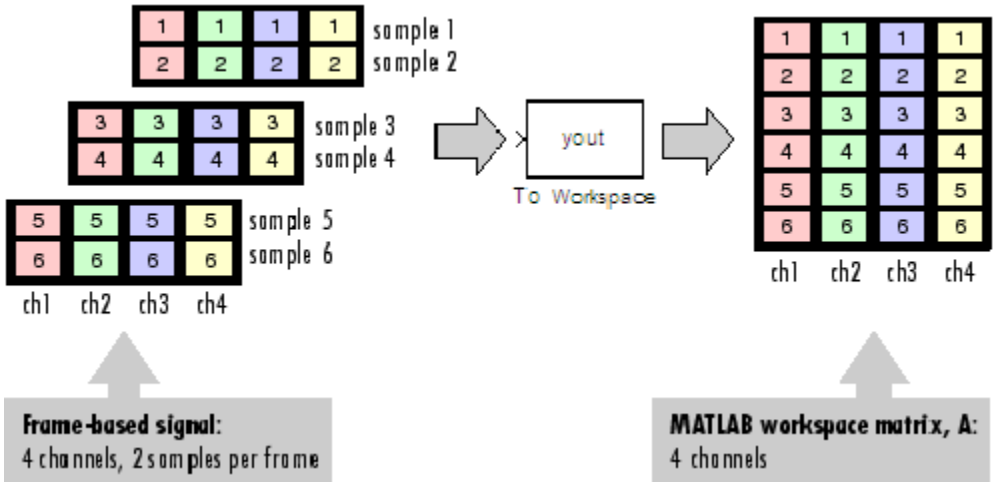
You have now successfully imported a three-channel frame signal into your model using the Signal From Workspace block.

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.

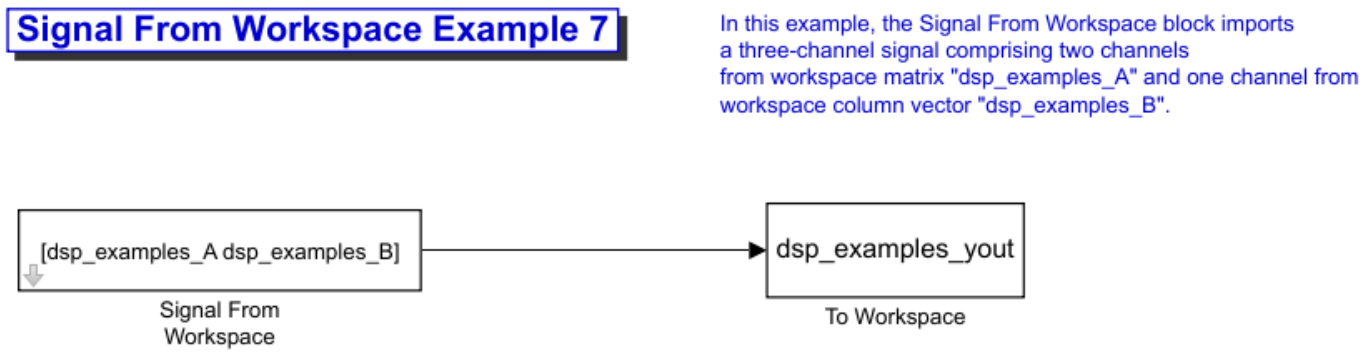
A signal with *N* channels and frame size M is represented by a sequence of *M*-by-*N* matrices. When this signal is input to the To Workspace block, the block creates a *P*-by-*N* array in the MATLAB workspace containing the *P* most recent samples from each channel. The number of rows, *P*, is specified 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.

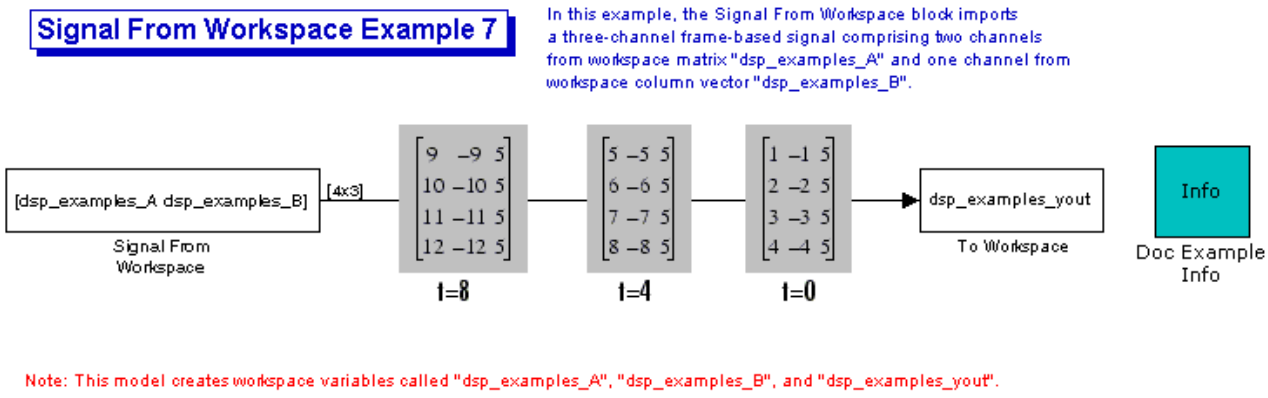
3 Double-click the To Workspace block. Set the block parameters as follows, and then click **OK**:

- **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 Using Rate Conversion Blocks” on page 3-16

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 Using Rate Conversion Blocks” on page 3-16
- “Convert Sample and Frame Rates in Simulink Using Frame Rebuffering Blocks” on page 3-23
- “Buffering and Frame-Based Processing” on page 3-29
- “Delay and Latency” on page 3-40
- “Variable-Size Signal Support DSP System Objects” on page 3-51

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

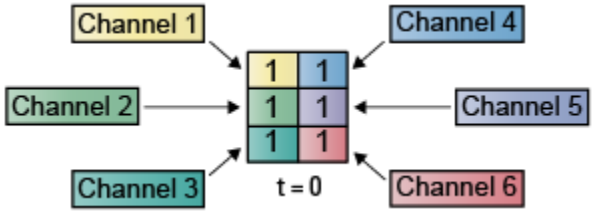
When you process signals using DSP System Toolbox software, you can do so in either a sample- or frame-based manner. When you are working with blocks in Simulink, you can specify, on a block-by-

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.

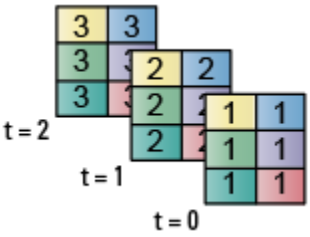
	Sample-Based Processing	Frame-Based Processing
Simulink — Blocks	**Input processing** = `Elements as channels (sample based)`	**Input processing** = `Columns as channels (frame based)`

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.



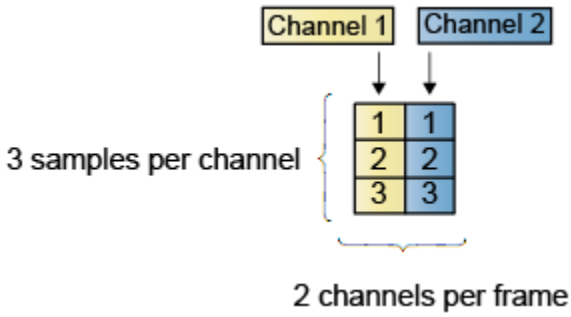
When you configure a block to perform sample-based processing, the block interprets scalar input as a single-channel signal. Similarly, the block interprets an *M*-by-*N* matrix as multichannel signal with *M***N* independent channels. For example, in sample-based processing, blocks interpret the following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



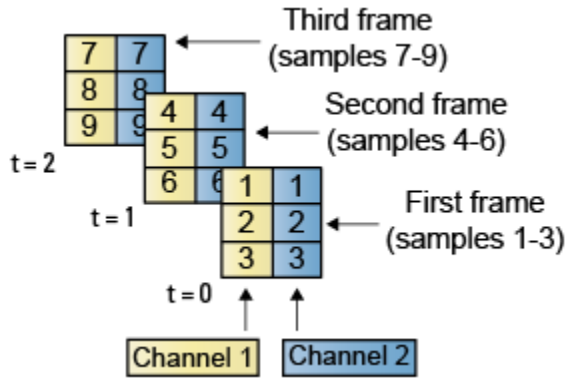
For more information about the recent changes to frame-based processing, see the “Frame-based processing changes” section of the *DSP System Toolbox Release Notes*.

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

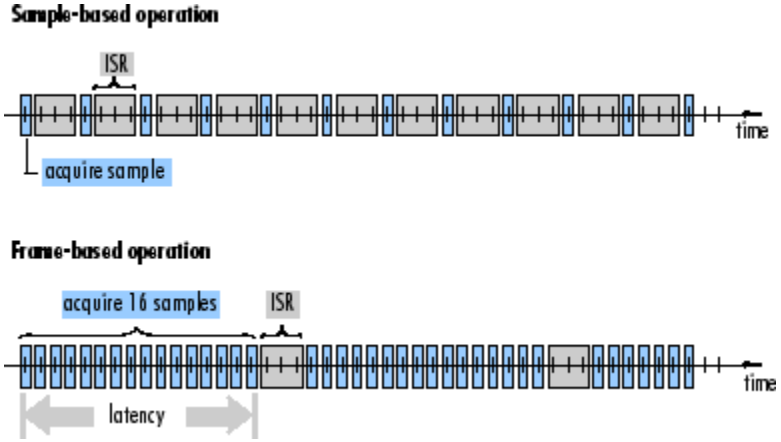
Frame-based processing is an established method of accelerating both real-time systems and model simulations.

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 3-40.

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

Related Examples

- Frame-Based Processing in Simulink

More About

- “Inspect Sample and Frame Rates in Simulink” on page 3-6
- “Convert Sample and Frame Rates in Simulink Using Rate Conversion Blocks” on page 3-16
- “Convert Sample and Frame Rates in Simulink Using Frame Rebuffering Blocks” on page 3-23

Inspect Sample and Frame Rates in Simulink

In this section...
“Sample Rate and Frame Rate” on page 3-6
“Inspect Signal Rates Using Color Coding” on page 3-7
“Use Model Data Editor or Probe to Inspect Signals” on page 3-15

Simulink enables you to model single-rate and multirate discrete systems. In a single-rate system, all the blocks in the model operate at the same rate. In a multirate system, different blocks operate at different rates. You can control the rate of block execution in a model by specifying the sample time (Simulink) (inverse of sample rate) at the block level. The sample time of a block indicates when the block generates outputs or updates its internal state. For more details on block execution rates, see “Sample Time” (Simulink). You can access sample time information interactively using tools such as Timing Legend and **Model Data Editor**.

This topic starts by defining basic concepts such as samples and frames, sample time (also known as sample period), sample rate, frame period, and frame rate, all in the context of a Simulink model. The topic then explains how to inspect sample rates and frame rates in a Simulink model using color coding and Timing Legend.

Sample Rate and Frame Rate

Samples and Frames

Sample is a value or a set of values of a signal at a given time instant. A frame can be a vector or matrix of samples of consecutive times stacked together. For more details on samples and frames, see “Sample- and Frame-Based Concepts” on page 3-2.

Sample Time (or Sample Period)

Sample time T_s of a block is the parameter that indicates when the block during simulation produces outputs and, if appropriate, updates its internal state. Sample time is the time interval between individual samples in a frame.

Sample Rate

The sample rate of a signal is the reciprocal of the sample time (or sample period) T_s. Mathematically, the sample rate can be represented using $1/T_s$.

In most cases, when you build a Simulink model, you need to set sample rates for only the source blocks. Simulink automatically computes the appropriate sample rates for the blocks connected to the source blocks.

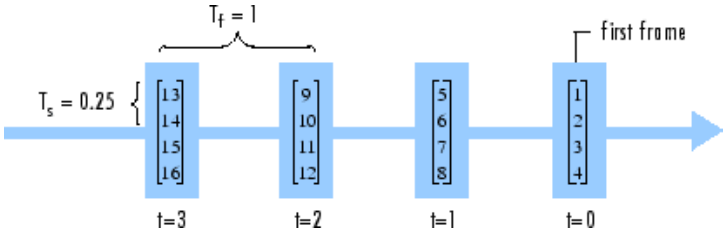
Frame Period and its Relationship to the Sample Period

The frame period T_f is given by the product of the sample time T_s and the frame size M, and it is represented using the following equation:

Frame Rate

The frame rate of a signal is the reciprocal of the frame period T_f and can be represented using $1/T_f$.

This diagram shows a single-channel signal with a frame size M of 4 and a frame period T_f of 1 second. The sample period T_s is therefore 1/4 or 0.25 seconds.



In most cases, the sequence sample time T_s is more 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 time, a larger frame size corresponds to a slower frame rate and vice versa.

Input and Output Sample and Frame Period

In Simulink, blocks can have different rates for their inputs and outputs (rate conversion blocks). The frame period and frame rate at the input and output of the block is calculated based on the sample time and the signal frame size. In most cases, when you build a Simulink model, you need to set sample rates for only the source blocks. Simulink automatically computes the appropriate sample rates for the blocks connected to the source blocks.

The *input frame period* (T_{fi}) of a signal is the time interval between consecutive frame inputs to a block. Similarly, the *output frame period* (T_{fo}) is the time at which the block updates the frame vector or matrix value at the output port.

More specifically, the sample time of inputs (T_{si}) and outputs (T_{so}) are related to their respective frame periods by the following equations:

where M_i and M_o are the input and output frame sizes, respectively.

Mathematically, the input frame rate is given by $1/T_{fi}$ and the output frame rate is given by $1/T_{fo}$.

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 the **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 the **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. For more details, see “Sample- and Frame-Based Concepts” on page 3-2.

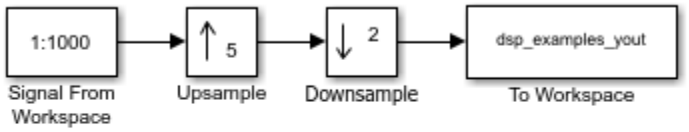
Inspect Signal Rates Using Color Coding

You can inspect the sample rate and frame rate of signals in a Simulink model using color coding. When you enable color coding, each sample time type (Simulink) in your model has one or more colors associated with it. Furthermore, you can annotate these signals based on their sample time and show this data in a Timing Legend. The Timing Legend contains the sample time color, annotation, and value for each sample time in the model. For more details, see “View Sample Time Information” (Simulink).

The following two models enable you to inspect sample rate and frame rate of signals using the color coding technique.

View Signal Sample Rate Using Sample Time Color Coding

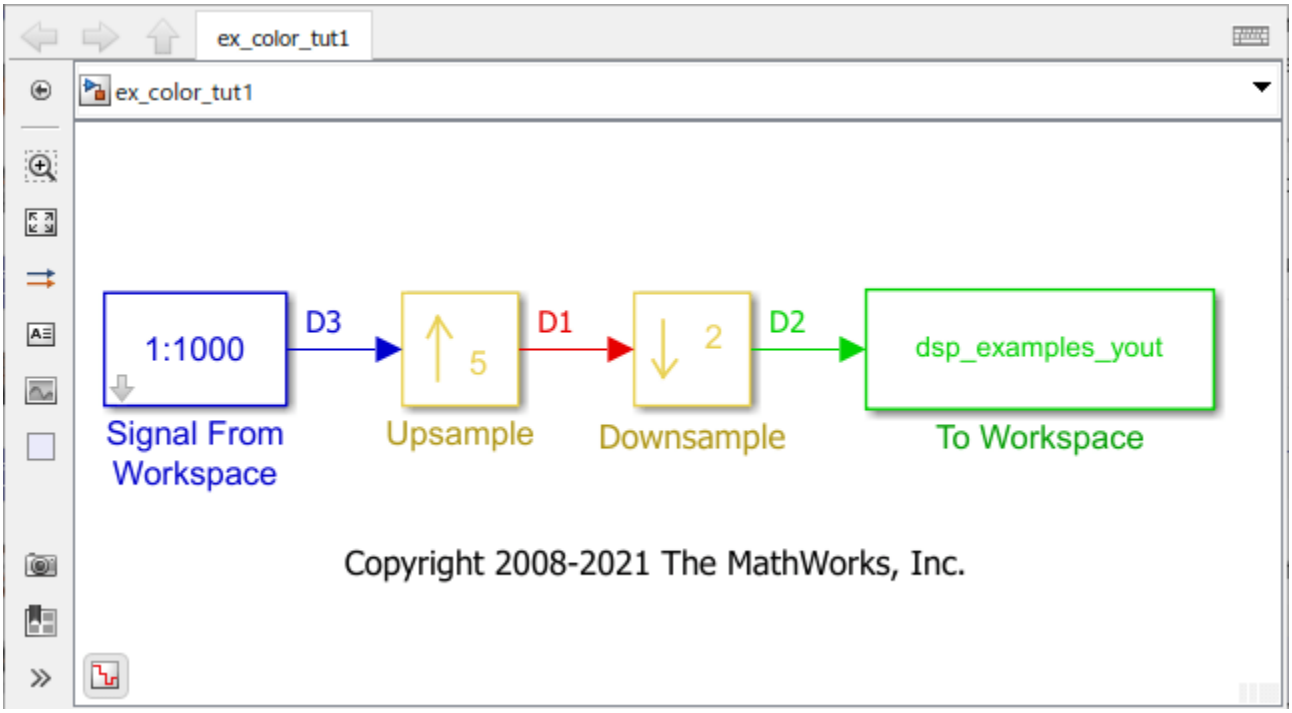
Open the model `'ex_color_tut1'`. In this model, the input signal is upsampled by a factor of 5 using the Upsample block, and the signal downsampled by a factor of 2 using the Downsample block. The overall rate conversion factor is 5/2. The Signal From Workspace block that generates the input signal has the **Sample time** parameter set to 2 seconds, and the **Samples per frame** parameter is set to 1. The **Input processing** parameter in the Upsample and the Downsample blocks is set to `Elements as channels (sample based)`. This setting enables the blocks to treat each sample they receive as an independent channel.



Copyright 2008-2021 The MathWorks, Inc.

Enable Sample Time Color Coding

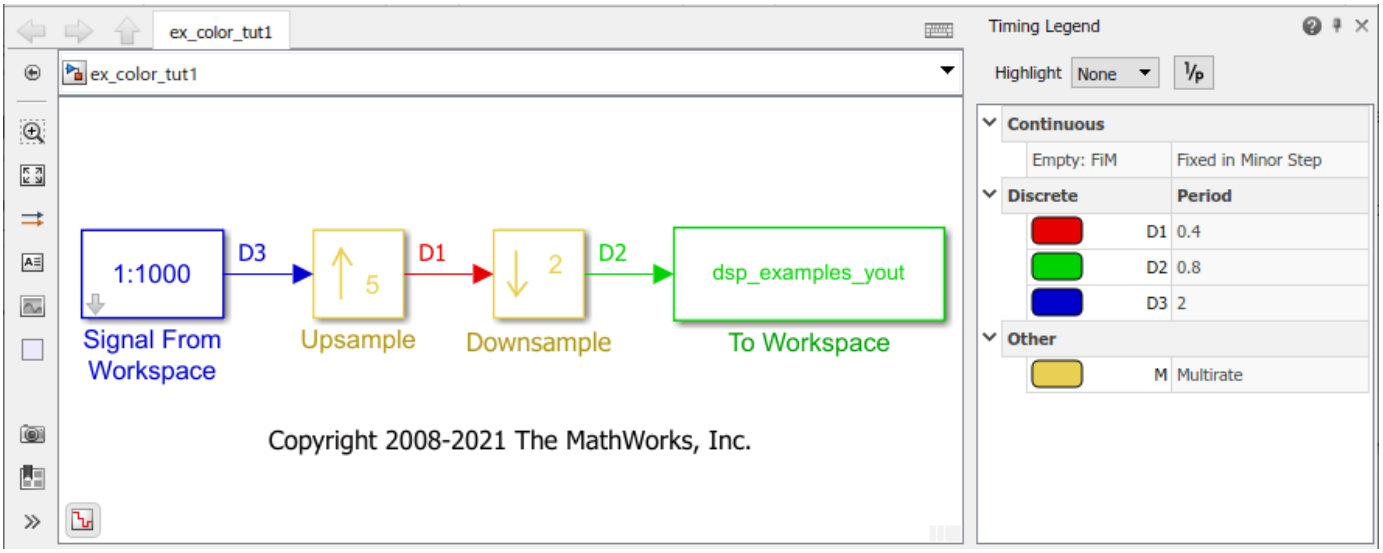
To enable sample time color coding in this model, on the **Debug** tab, select **Information Overlays** > **Colors**. This selection turns on sample time color coding. Simulink now assigns each sample time in the model a different color. In addition, to enable annotation for all sample times, select **Information Overlays** > **Text**. Selecting both **Colors** and **Text** displays both the colors and the annotations. Run the model. Every signal in this model has a different sample rate. Therefore, each signal is assigned a different color.



Inspect Sample Time Using Timing Legend

You can view the information on the sample time color, annotations, and the value of each sample time in the model using a Timing Legend. To enable this legend, select **Information Overlays > Timing Legend**. Simulink updates the model diagram and opens the Timing Legend. The Timing Legend shows the values of the frame period of the signal. In this example, because the Upsample and Downsample blocks treat each sample as an independent channel, and the **Samples per frame** parameter of the source block is set to 1 and the frame period of the signal is the same as the sample period (sample time) of the signal.

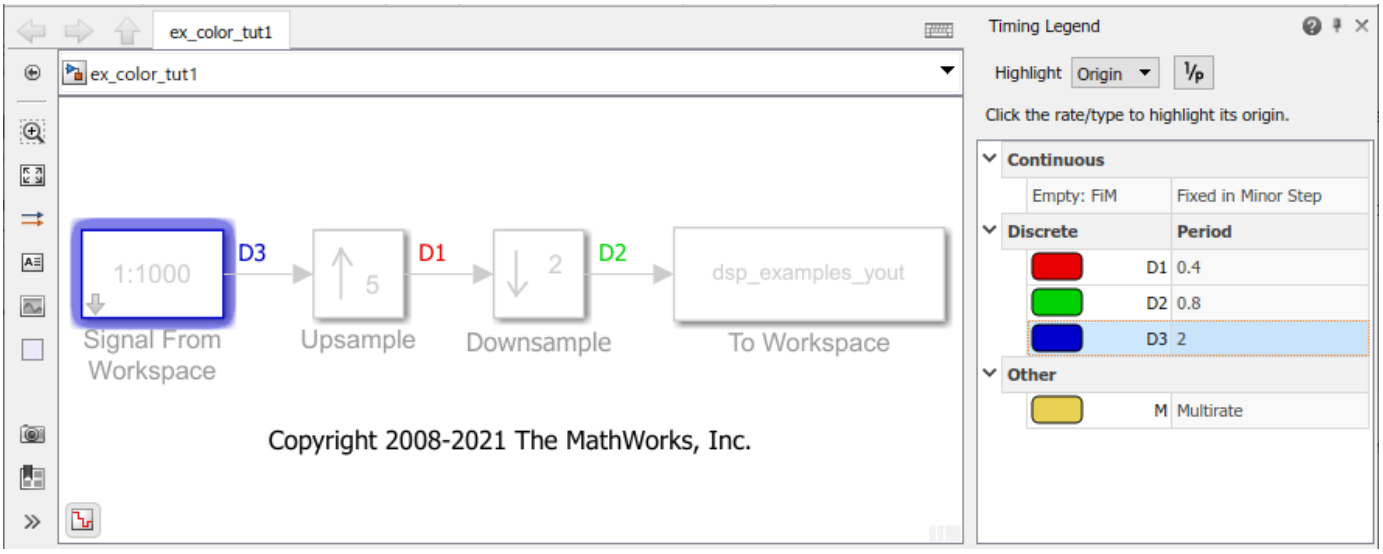
As you can see from the frame period values in the timing legend, signal D3 originating from the source block is the slowest with a sample time of 2 seconds. The Upsample block increases the sample rate of the signal by a factor of 5. As a result, the sample time reduces by a factor of 5 and becomes 0.4 seconds. Furthermore, the signal is downsampled by a factor of 2, making the sample time of the output signal D2 0.8 seconds.



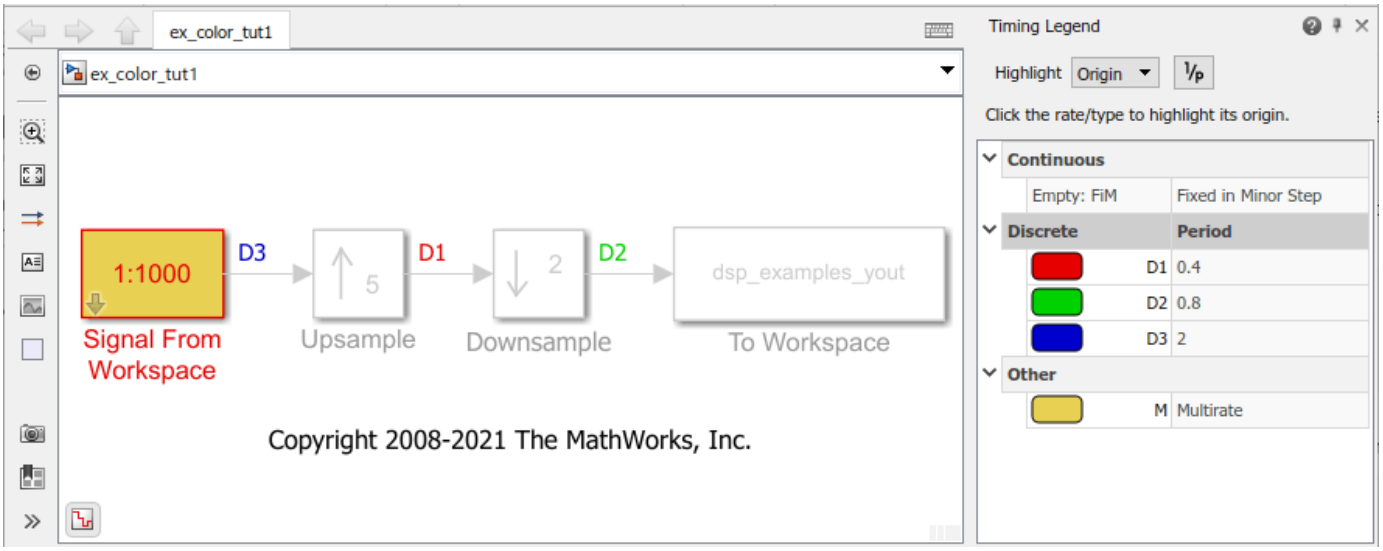
The legend provides two highlighting options:

- Highlighting the blocks and signals that the sample time originates from.
- Highlighting all the blocks and signals that contain the selected sample time.

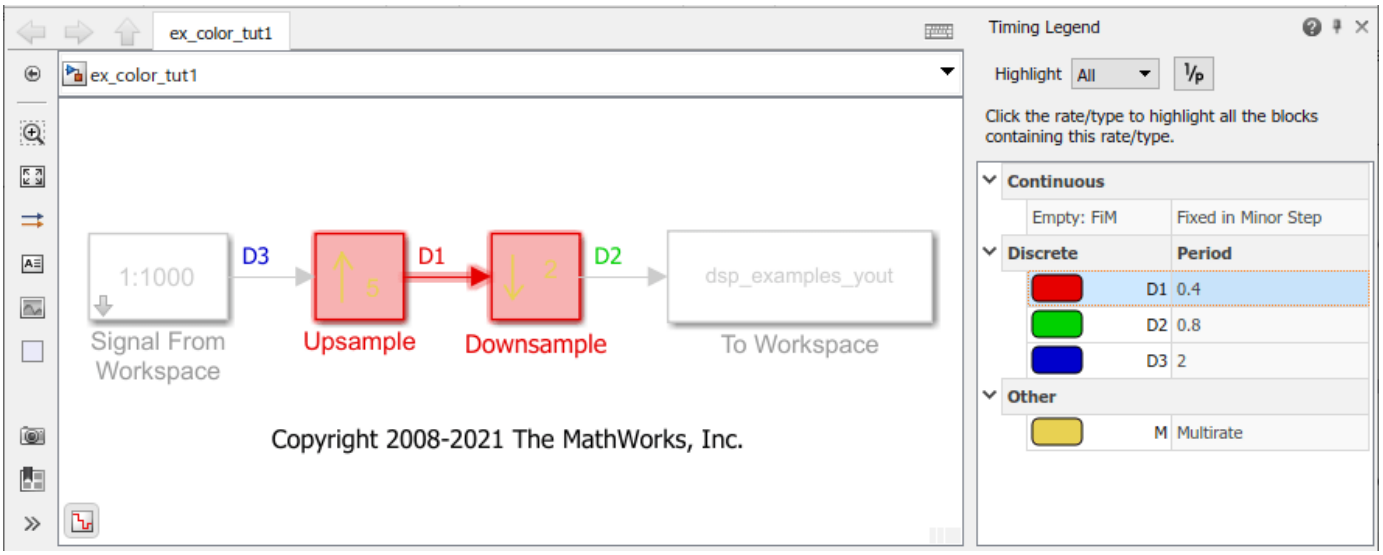
To enable highlighting the origin of the sample times, click the **Origin** option from the Highlight menu. Select a specific sample time from the list (D3 in this case) to see the source of this sample time.



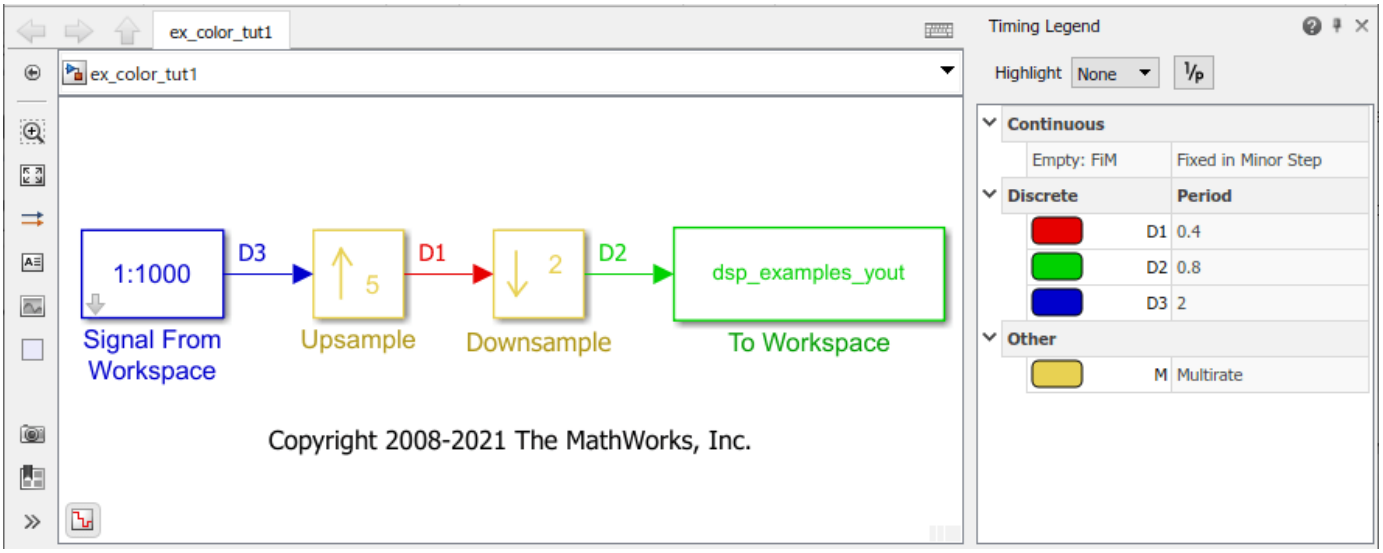
You can also click the type of the sample time to highlight all sources of a particular type of sample time.



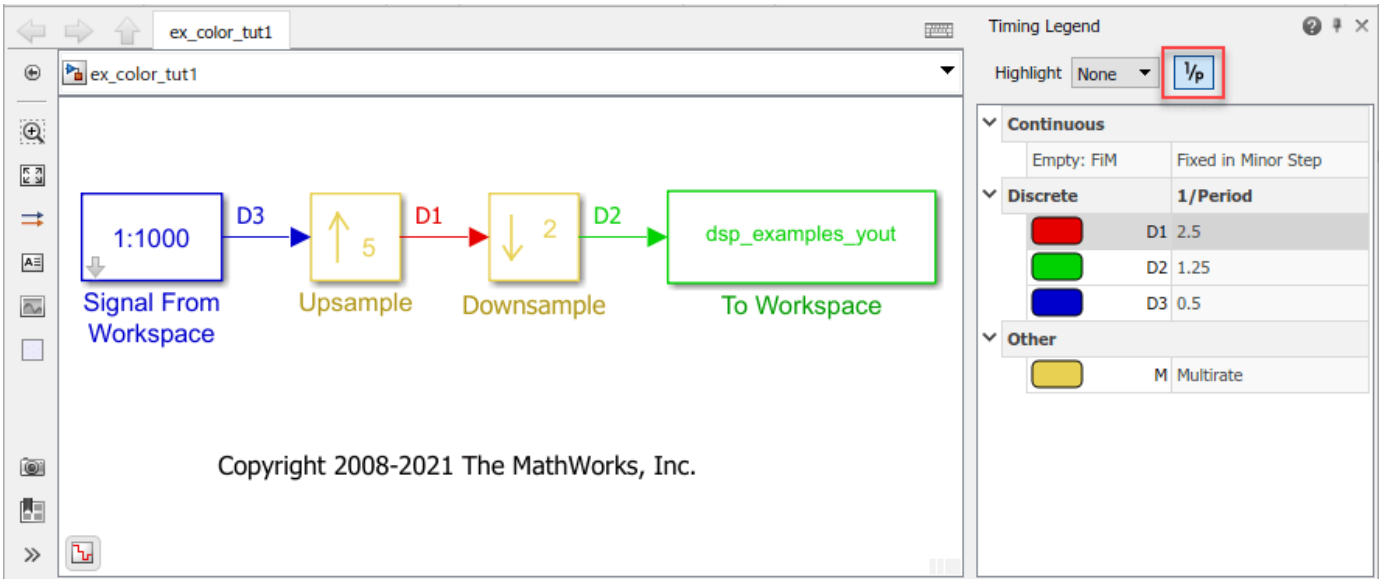
To enable highlighting of all the blocks that contain a selected sample time, click the **All** option from the Highlight menu.



The None option from the Highlight menu clears current highlighting.

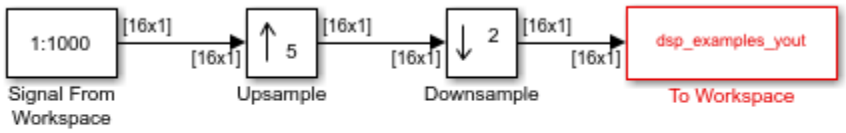


You can also display the sample rate (1/sample period) of the signal by clicking on the 1/p button. The timing legend displays 1/period values.



View Signal Frame Rate Using Sample Time Color Coding

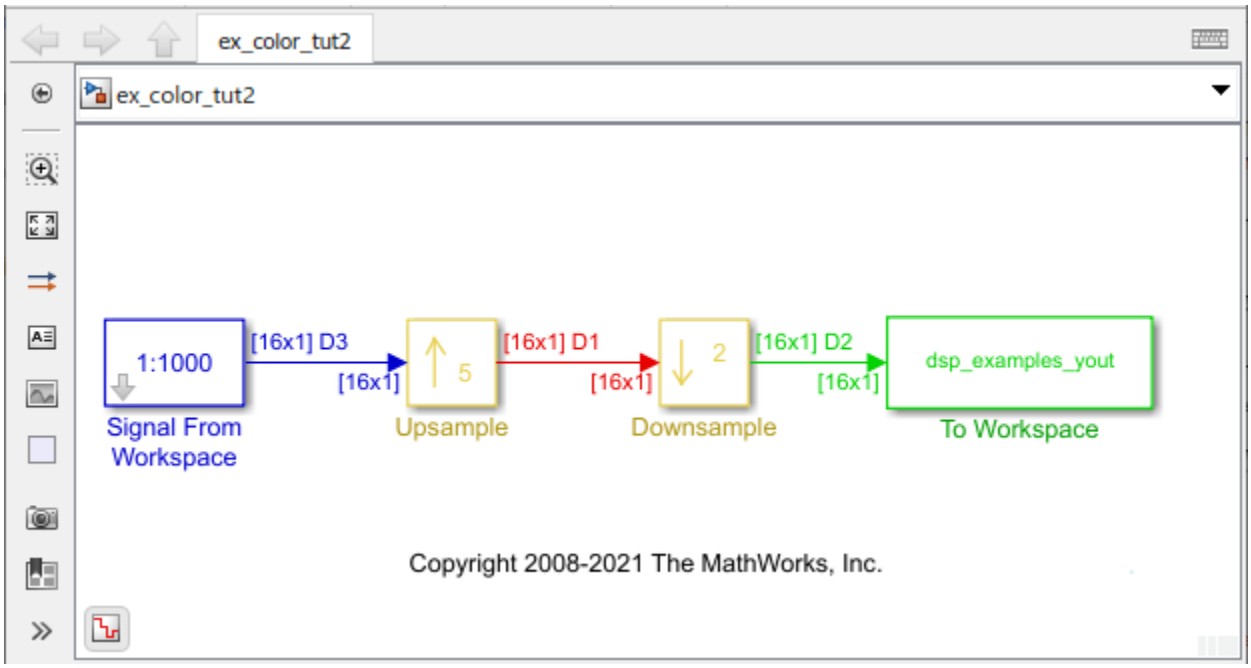
Open the model `'ex_color_tut2'`. The Signal From Workspace block has the **Sample time** parameter set to 1, and the **Samples per frame** parameter is set to 16. Each frame in the generated signal contains 16 samples. The **Input processing** parameter in the Upsample and the Downsample blocks is set to `Columns as channels (frame based)` and the **Rate options** parameter is set to `Allow multirate processing`. This setting enables the Upsample and Downsample blocks to operate in the multirate mode and treat data as frames of size 16.



Copyright 2008-2021 The MathWorks, Inc.

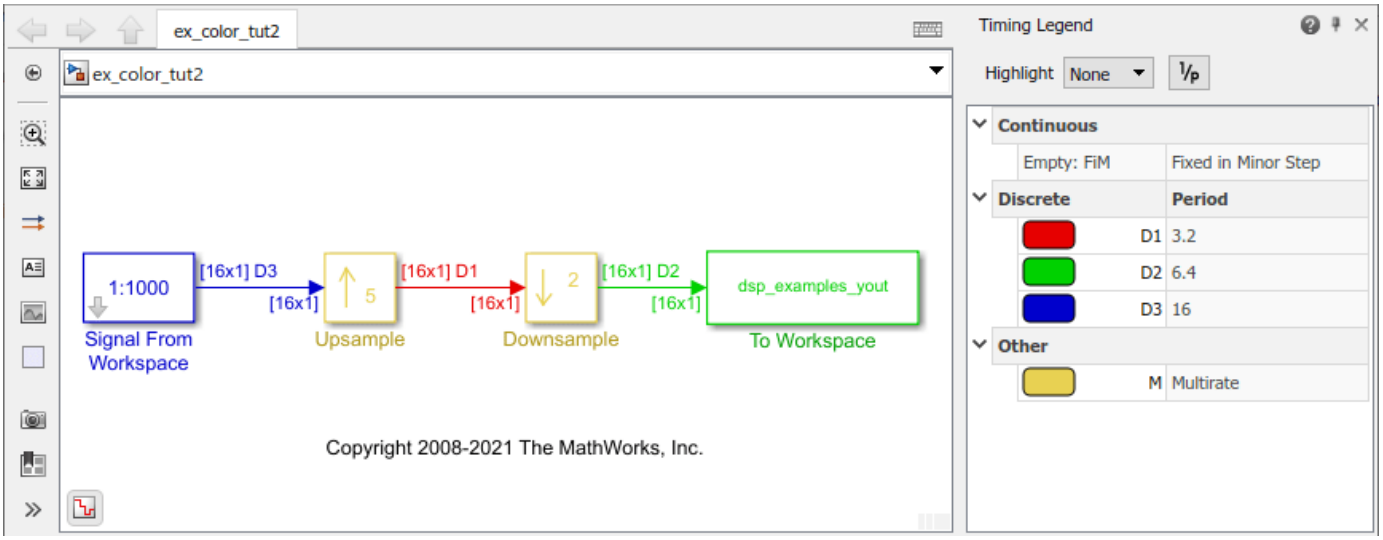
Enable Sample Time Color Coding

On the **Debug** tab, select **Information Overlays** > **Colors** and **Text**. This selection turns on sample time color coding and annotations. Run the model. Simulink assigns each frame rate a different color.



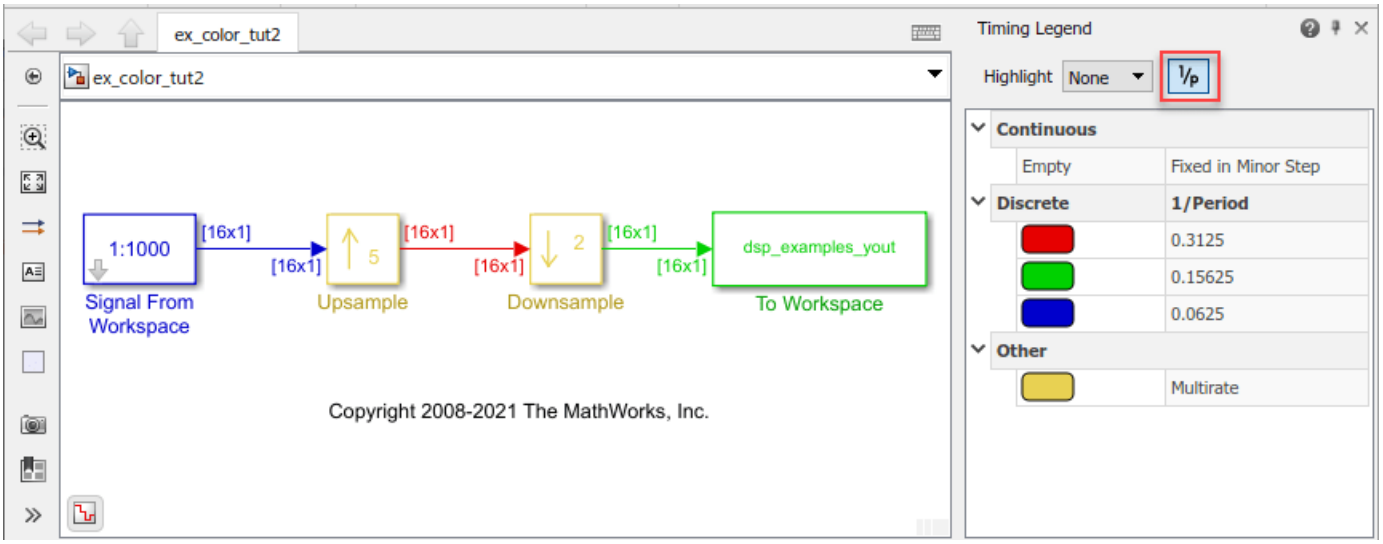
Inspect Frame Rate Using Timing Legend

Enable the Timing Legend by selecting **Information Overlays** > **Timing Legend**. Simulink updates the model and opens the Timing Legend. You can view the value of the frame period for each signal in the model, the color associated with the frame period, and the corresponding annotation. For example, the frame period of the input signal (denoted by D3 in the Timing Legend) is given by $M_i \times T_{si}$ or 16×1, which equals 16 seconds. The D1 signal is upsampled by a factor of 5. As a result, the frame period of D1 is 16/5 or 3.2 seconds. The signal D2 that follows is downsampled by 2, and the resultant frame period is 3.2 x 2 = 6.4 seconds.



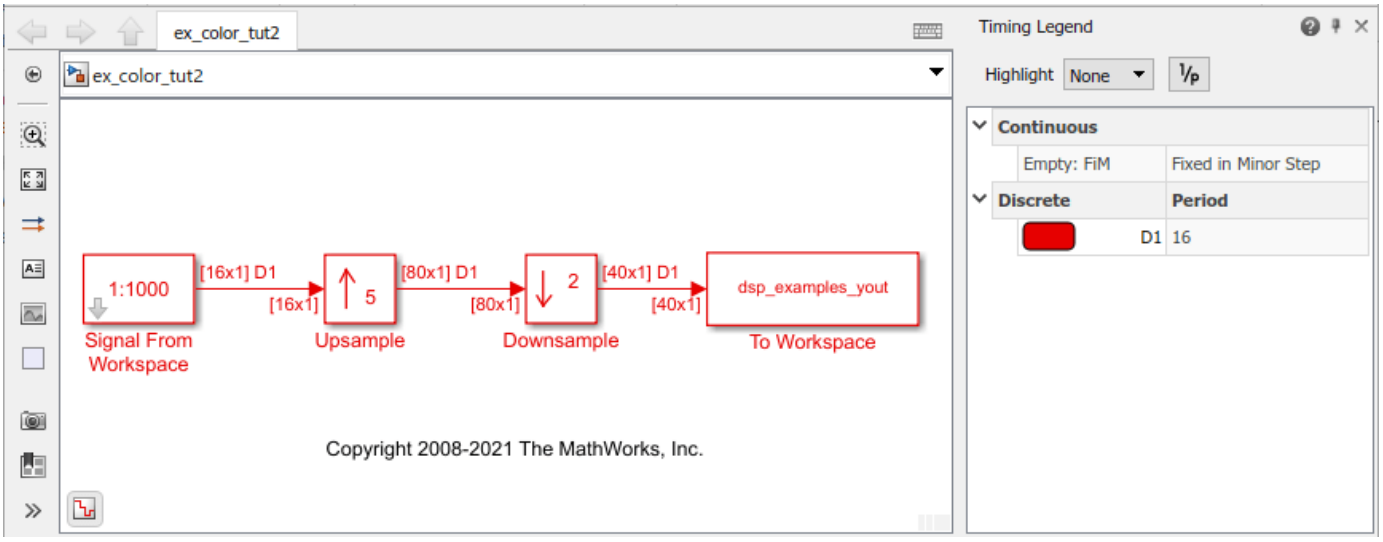
Because the **Rate options** parameter in the Upsample and Downsample blocks is set to `Allow multirate processing`, each of these blocks changes the frame rate. Therefore, each frame signal in the model is assigned a different color.

You can view the frame rate of the signal by clicking the 1/p button in the timing legend.



Double-click on the Upsample block and the Downsample block, and change the **Rate options** parameter to `Enforce single-rate processing`.

Run the model. Every signal now has the same frame period and is therefore coded with the same color.



Use Model Data Editor or Probe to Inspect Signals

There are two other techniques that you can use to inspect signals in your model: Model Data Editor and Probe blocks.

Model Data Editor

The **Model Data Editor** (on the **Modeling** tab, click **Model Data Editor**) shows information about model data (signals, parameters, and states) in a sortable, searchable table. The **Sample Time** column shows the sample time specified for each signal in a model. After you update the block diagram, the column also shows the specific sample that each signal uses (for example, for signals where you specify inherited sample time, the column value is `-1`). You can also use this column to specify sample times.

Probe blocks

You can connect Probe blocks to the signal that you want to inspect. The Probe blocks show signal parameters such as signal width, sample time, signal complexity, and signal dimensions.

See Also

More About

- “Sample- and Frame-Based Concepts” on page 3-2
- “View Sample Time Information” (Simulink)
- “Convert Sample and Frame Rates in Simulink Using Rate Conversion Blocks” on page 3-16
- “Convert Sample and Frame Rates in Simulink Using Frame Rebuffering Blocks” on page 3-23

Convert Sample and Frame Rates in Simulink Using Rate Conversion Blocks

In this section...
“Rate Conversion Blocks” on page 3-16
“Rate Conversion by Frame-Rate Adjustment” on page 3-17
“Rate Conversion by Frame-Size Adjustment” on page 3-20

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 also changes either the sample rate or the frame rate of the signal. For more details on the frame rebuffering technique, see “Convert Sample and Frame Rates in Simulink Using Frame Rebuffering Blocks” on page 3-23.

This topic contains two models that show how to change the sample rate of a signal using the direct rate conversion blocks. The following is a list of rate conversion blocks in DSP System Toolbox that are not based on filters.

- Downsample
- Repeat
- Upsample

Note that the upsampling and downsampling operations can introduce imaging and aliasing in the frequency domain, respectively. To prevent that from happening, use direct rate conversion blocks that are filter-based. For a list of the direct rate conversion blocks and the topics that show how to use these blocks, see “Multirate and Multistage Filters”.

Direct Rate Conversion Using the Rate options Parameter

In certain rate conversion blocks, the **Rate options** parameter determines whether the block operates in the single-rate mode or the multirate mode.

When **Rate options** parameter is set to:

- `Enforce single-rate processing`: The block operates in the single-rate mode. The input and output sample rates of the block remain the same. However, the signal frame size changes according to the rate conversion factor specified in the block dialog box.
- `Allow multirate processing`: The block operates in the multirate mode. The input and output frame sizes of the block remain the same. However, the signal frame rate changes according to the rate conversion factor specified in the block dialog box.

  When a Simulink model contains signals with various frame rates, the model is called *multirate*. For more information on multirate models, see “Excess Algorithmic Delay (Tasking Latency)” on page 3-45. Also see “Time-Based Scheduling and Code Generation” (Simulink Coder).

The following two sections show how to change the sample rate of a signal using the two rate conversion modes in the Downsample block.

Rate Conversion by Frame-Rate Adjustment

Adjust the frame rate of the signal by setting the **Rate options** parameter of the rate conversion block to `Allow multirate processing`. In this mode, the sample rate of the signal changes by changing the frame rate of the signal while keeping the frame size constant.

- $T_{fo} \neq T_{fi}$
- $M_o = M_i$

where,

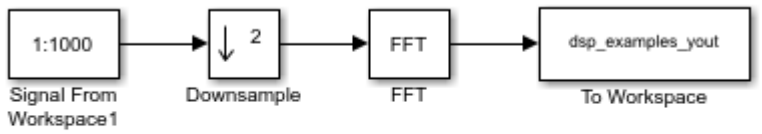
- T_{fo} is the output frame rate.
- T_{fi} is the input frame rate.
- M_o is the output frame size.
- M_i is the input frame size.

The sample rate of the output signal $1/T_{so}$ is given by the following equation:

$$1/T_{so} = M_o/T_{fo}$$

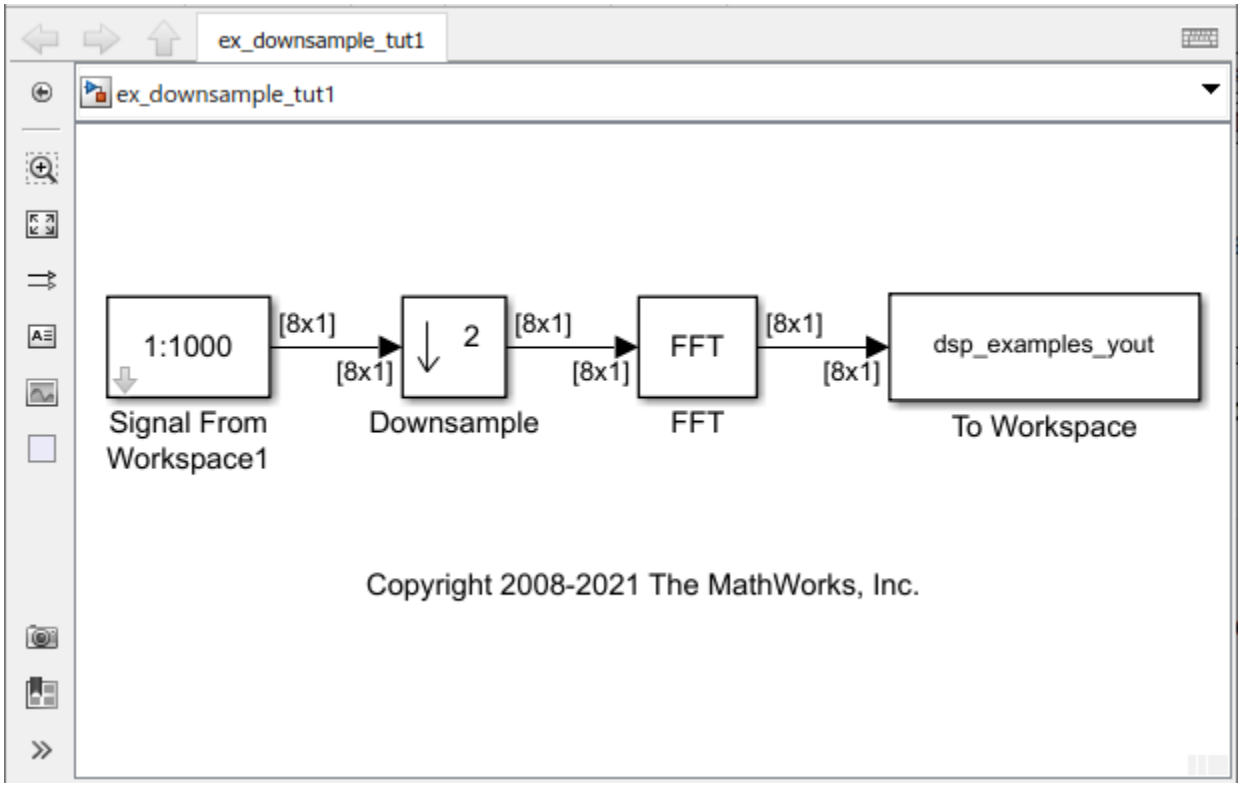
The `'ex_downsample_tut1'` model shows rate conversion by frame-rate adjustment.

Open the model. In this model, the input signal is downsampled by a factor of 2 using the Downsample block. The Signal From Workspace block that generates the input signal has the **Sample time** parameter set to 0.125 seconds and the **Samples per frame** parameter set to 8. Therefore, the data generated by the block has a sample time of 0.125 seconds and a frame size of 8. The **Input processing** parameter in the Downsample block is set to `Columns as channels (frame based)`, and the **Rate options** parameter is set to `Allow multirate processing`. This setting enables the Downsample block to operate in the multirate mode and treat data as frames of size 8.

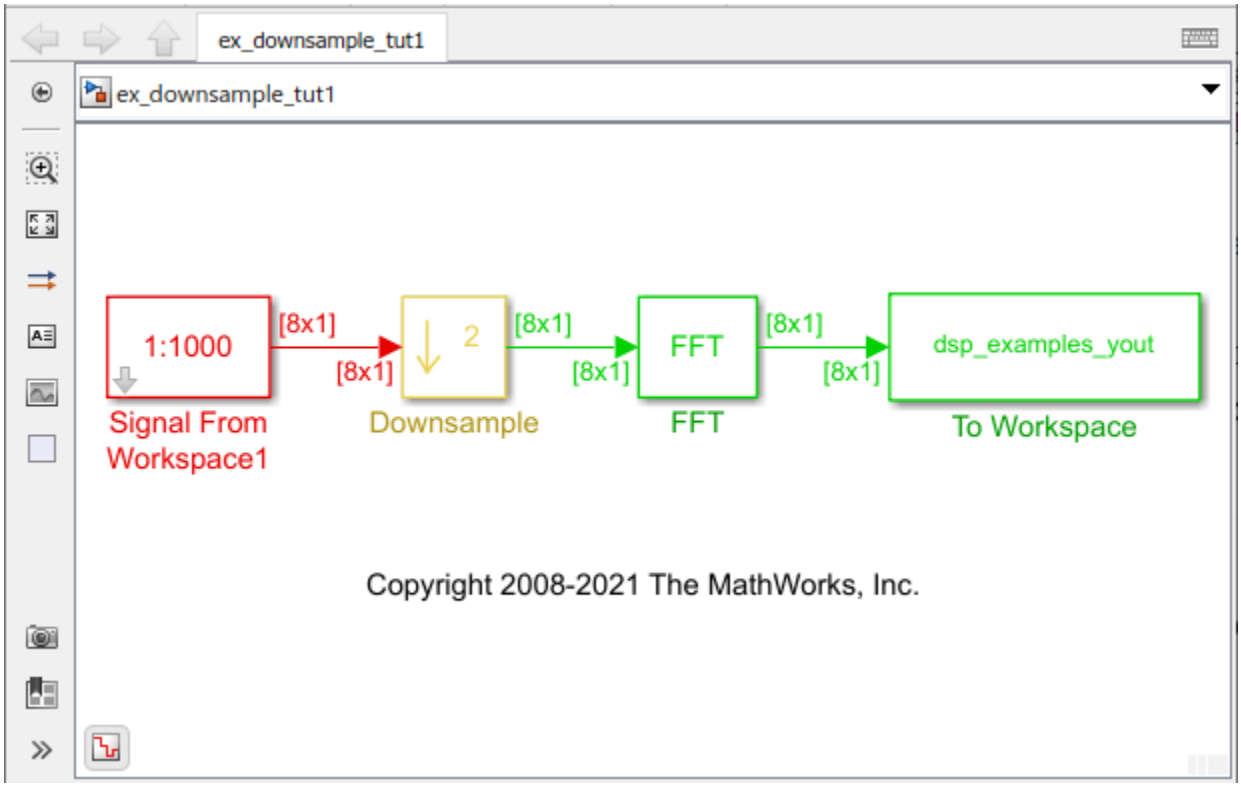


Copyright 2008-2021 The MathWorks, Inc.

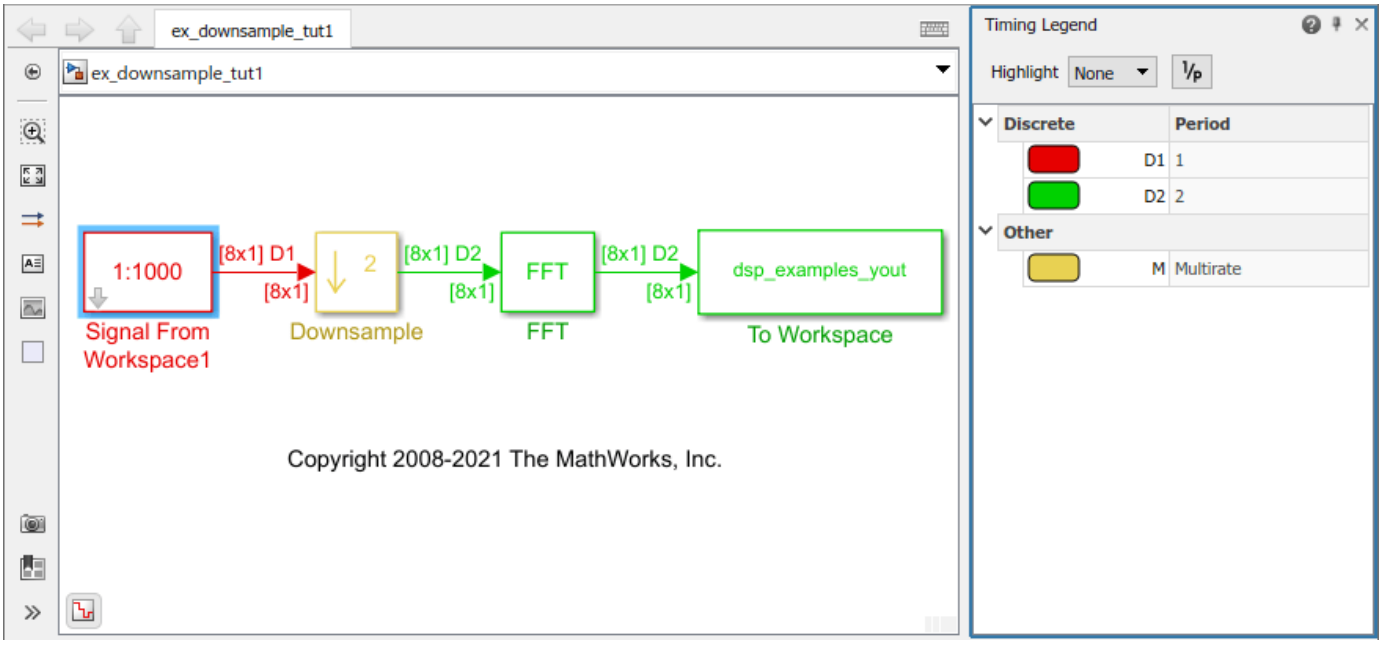
In the **Debug** tab, select **Information Overlays > Signal Dimensions**. When you run the model, the dimensions of the signals appear next to the lines connecting the blocks. The signal dimensions in the model confirm that the frame size of the signal remains the same between the input and output of the Downsample block.



To see the change in frame rate, enable sample time color coding by selecting **Information Overlays > Colors**.



Furthermore, to enable the annotation and Timing Legend, select **Information Overlays > Text** and **Timing Legend**. In the Timing Legend, you can view the value of the frame period for each signal in the model, the color associated with the frame period, and the corresponding annotation. For example, the frame period of the input signal (denoted by D1 in the Timing Legend) is given by $T_{fi} = M_i \times T_{si}$ or 8×0.125, which equals 1 second.



The Timing Legend in the model verifies that the output from the Downsample block has a frame period T_{fo} of 2 seconds, which is twice the frame period of the input T_{fi}. However, because the frame rate of the input $1/T_{fi}$ is 1 frame per second, and 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 seconds without any change to the frame size.

Rate Conversion by Frame-Size Adjustment

Adjust the frame size of the signal by setting the **Rate options** parameter of the rate conversion block to `Enforce single-rate processing`. In this mode, the sample rate of the signal changes by changing the frame size of the signal while keeping the frame rate constant.

- $M_o \neq M_i$
- $T_{fo} = T_{fi}$

where,

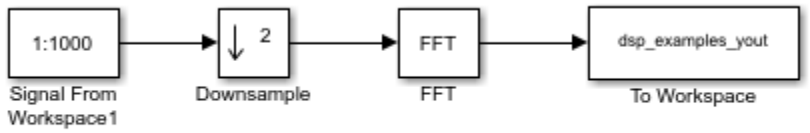
- M_o is the output frame size.
- M_i is the input frame size.
- T_{fo} is the output frame rate.
- T_{fi} is the input frame rate.

The sample rate of the output signal $1/T_{so}$ is given by the following equation:

$$1/T_{so} = M_o/T_{fo}$$

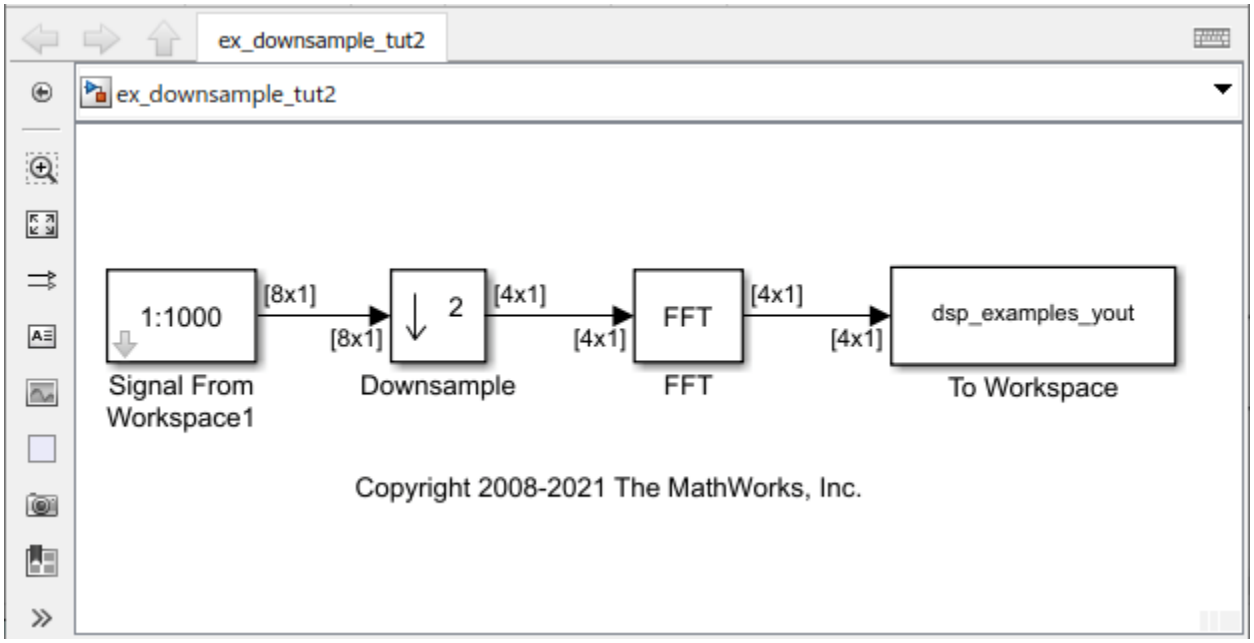
The `'ex_downsample_tut2'` model shows rate conversion by frame-size adjustment.

Open the model. In this model, the input signal is downsampled by a factor of 2 using the Downsample block. The Signal From Workspace block that generates the input signal has the **Sample time** parameter set to 0.125 seconds, and the **Samples per frame** parameter is set to 8. Therefore, the data generated by the block has a sample time of 0.125 seconds and a frame size of 8. The **Input processing** parameter in the Downsample block is set to `Columns as channels (frame based)`, and the **Rate options** parameter is set to `Enforce single-rate processing`. This setting enables the Downsample block to operate in the single-rate mode.

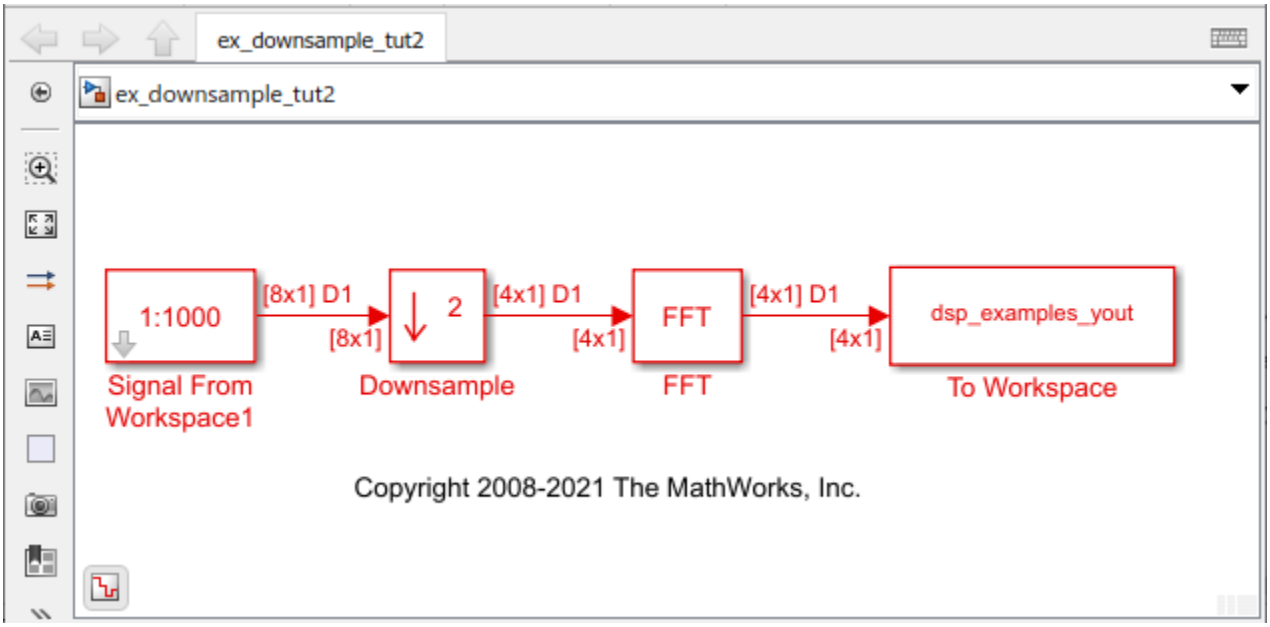


Copyright 2008-2021 The MathWorks, Inc.

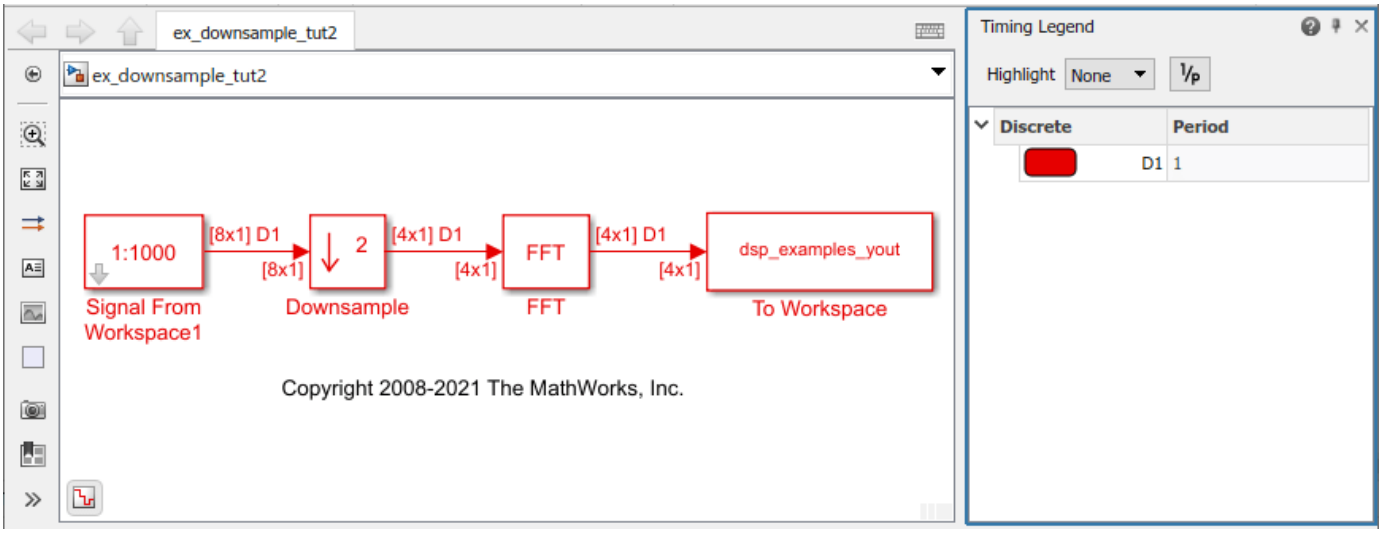
In the **Debug** tab, select **Information Overlays > Signal Dimensions**. When you run the model, the dimensions of the signals appear next to the lines connecting the blocks. The signal dimensions in the model confirm that the frame size of the signal decreases by a factor of 2 between the input and output of the Downsample block.



To see the effect on the frame rate, enable sample time color coding by selecting **Information Overlays > Colors**. You can see that all the blocks and signals are in the same color because they are operating at the same rate.



Furthermore, to enable the annotation and Timing Legend, select **Information Overlays > Text** and **Timing Legend**. In the Timing Legend, you can view the value of the frame period for each signal in the model, the color associated with the frame period, and the corresponding annotation. For example, the frame period of the input signal (denoted by D1 in the Timing Legend) is given by $T_{fi} = M_i \times T_{si}$ or 8×0.125, which equals 1 second. 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, which is half the frame size of the input. As a result, the sample period of the output, $T_{so} = T_{fo}/M_o$ is 1/4 or 0.25 seconds. This process occurs without any change to the frame rate $T_{fi} = T_{fo}$.

See Also

More About

- “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 Using Frame Rebuffering Blocks” on page 3-23

Convert Sample and Frame Rates in Simulink Using Frame Rebuffering Blocks

Frame Rebuffering Blocks

There are two common types of operations that impact the frame and sample rates of a signal: Frame rebuffering and direct rate conversion. Frame rebuffering, which is used to alter the frame size of a signal in order to improve simulation throughput, usually also changes either the sample rate or the frame rate of the signal. Direct rate conversions such as upsampling and downsampling can be implemented by altering either the frame rate or the frame size of a signal. For more details on the direct rate conversion technique, see “Convert Sample and Frame Rates in Simulink Using Rate Conversion Blocks” on page 3-16.

This topic contains two models that show how to change the sample rate of a signal using the frame rebuffering blocks. The following is a list of frame rebuffering blocks in DSP System Toolbox.

- Buffer
- Delay Line
- Unbuffer
- Variable Selector

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 can rebuffer the signal to a frame size more appropriate for the intended operations without introducing any change to the data or sample rate.

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 sample rate of the signal T_s. This operation usually results in a change to the frame rate of the signal T_f according to the following equation:

However, this equation is true only if no samples are added to or deleted from the original signal. Therefore, this equation 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 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 data and sample period only when its **Buffer overlap** parameter is set to `0`. The output frame period T_{fo} is given by the following equation:

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 data and sample period.

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 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 when you want to select a subset of samples from each input frame for processing.

The following blocks alter a signal by adjusting its frame size. 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:

- Buffer –– 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 the following equation:

  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

- Delay Line –– 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 and equal the input sample period, $T_{fo} = T_{fi} = T_{si}$. The new output sample period is:

- Variable Selector –– The Variable Selector block can remove, add, and 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:

  where M_o is the length of the block 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 Sample Period

In this example, the Buffer block rebuffers the signal to a larger frame size. The Unbuffer block unbuffers the input frame into a sequence of scalar values.

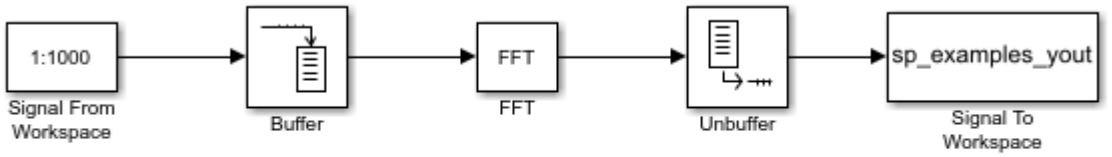
A signal with a sample period of 0.125 seconds is rebuffered from a frame size of 8 to a frame size of 16. This rebuffering process doubles the frame period from 1 to 2 seconds, but does not change the sample period of the signal, $T_{so} = T_{si} = 0.125$. The signal is then unbuffered into a sequence of sample outputs using the Unbuffer block. The frame period then changes to 0.125 seconds, which is equal to the value of the sample period of the signal.

This process does not add or delete samples from the original signal.

Open the model `'ex_buffer_tut1'`.

The Signal From Workspace block has the **Sample time** parameter set to 0.125, and the **Samples per frame** parameter is set to 8. Each frame in the generated signal contains 8 samples and has a sample period of 0.125 seconds.

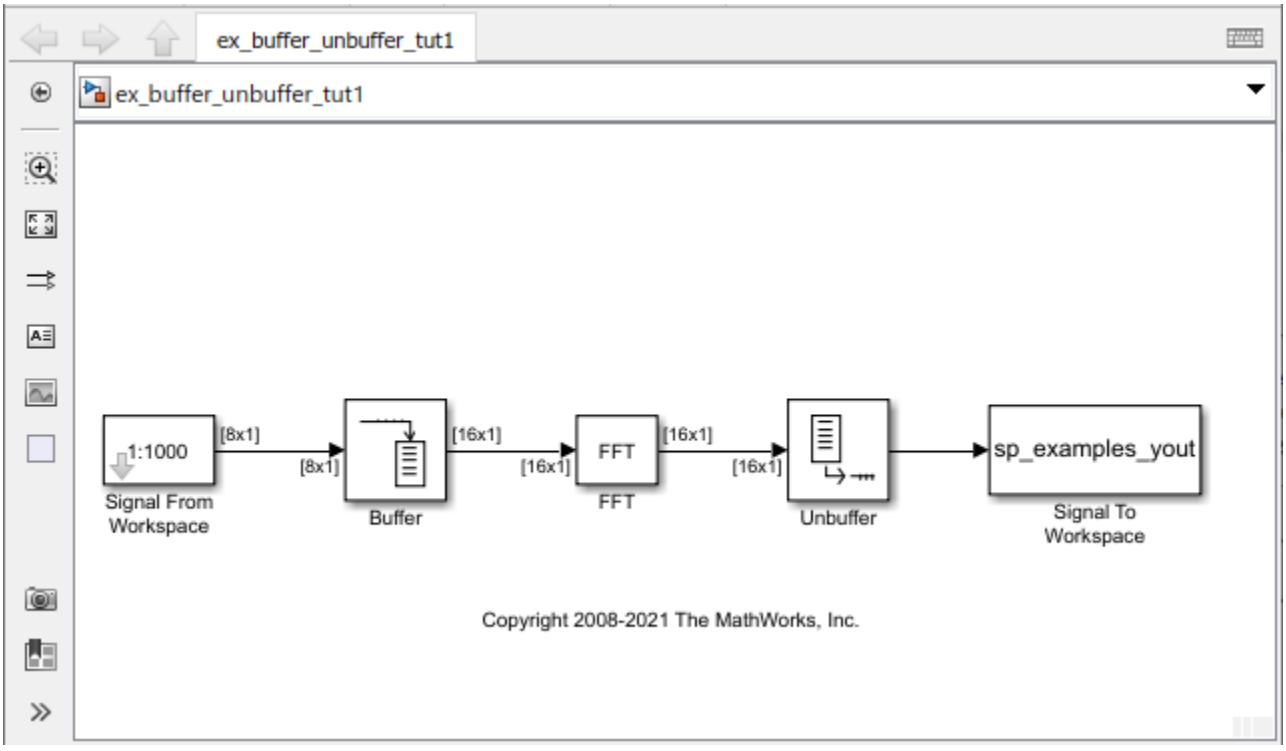
The Buffer block has the **Output buffer size (per channel)** parameter set to 16, and the **Buffer overlap** parameter is set to 0. The Buffer block rebuffers the signal from a frame size of 8 to a frame size of 16.



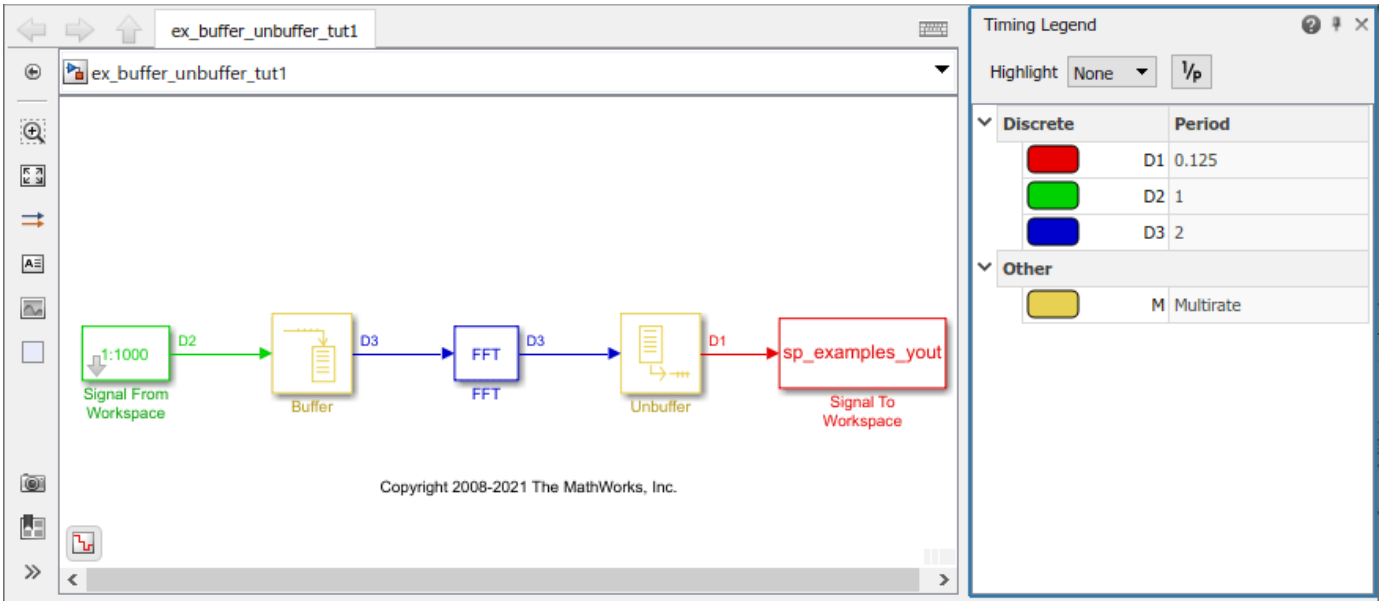
Copyright 2008-2021 The MathWorks, Inc.

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. The signal dimensions in the model confirm the following:

- Buffer block changes the frame size of the signal from 8 to 16.
- Unbuffer block unbuffers the signal into a sequence of scalar outputs.



To view the effect on the frame period of the signal, enable color coding, annotations, and timing legend by selecting **Information Overlays > Colors**, **Text**, **Timing Legend**. In the Timing Legend, you can view the value of the frame period for each signal in the model, the color associated with the frame period, and the corresponding annotation.



As you can see, the input frame period of the signal (denoted by D2 in the model), is given by $T_{fi} = M_i \times T_{si}$ or 8×0.125 and equals 1 second. The Buffer block doubles the frame period from 1 to 2 seconds. The Unbuffer block that follows unbuffers the signal into a sequence of scalar outputs. The frame period of the unbuffered sequence equals 0.125 seconds, which matches the sample period of the signal.

Buffer Signals by Altering the Sample Period

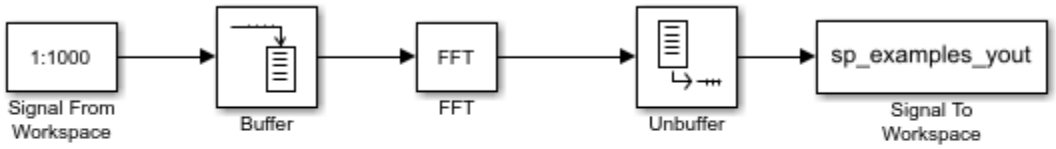
In this example, the Buffer block rebuffers the signal to a larger frame size while overlapping 4 samples per frame.

Some forms of buffering alter the signal data or sample period in addition to adjusting the frame size. In the following example, a signal with a sample period of 0.125 seconds is rebuffered from a frame size of 8 to a frame size of 16 with a buffer overlap of 4 samples.

Open the model `'ex_buffer_tut2'`.

The Signal From Workspace block has the **Sample time** parameter set to 0.125, and the **Samples per frame** parameter is set to 8. Each frame in the generated signal contains 8 samples and has a sample period of 0.125 seconds.

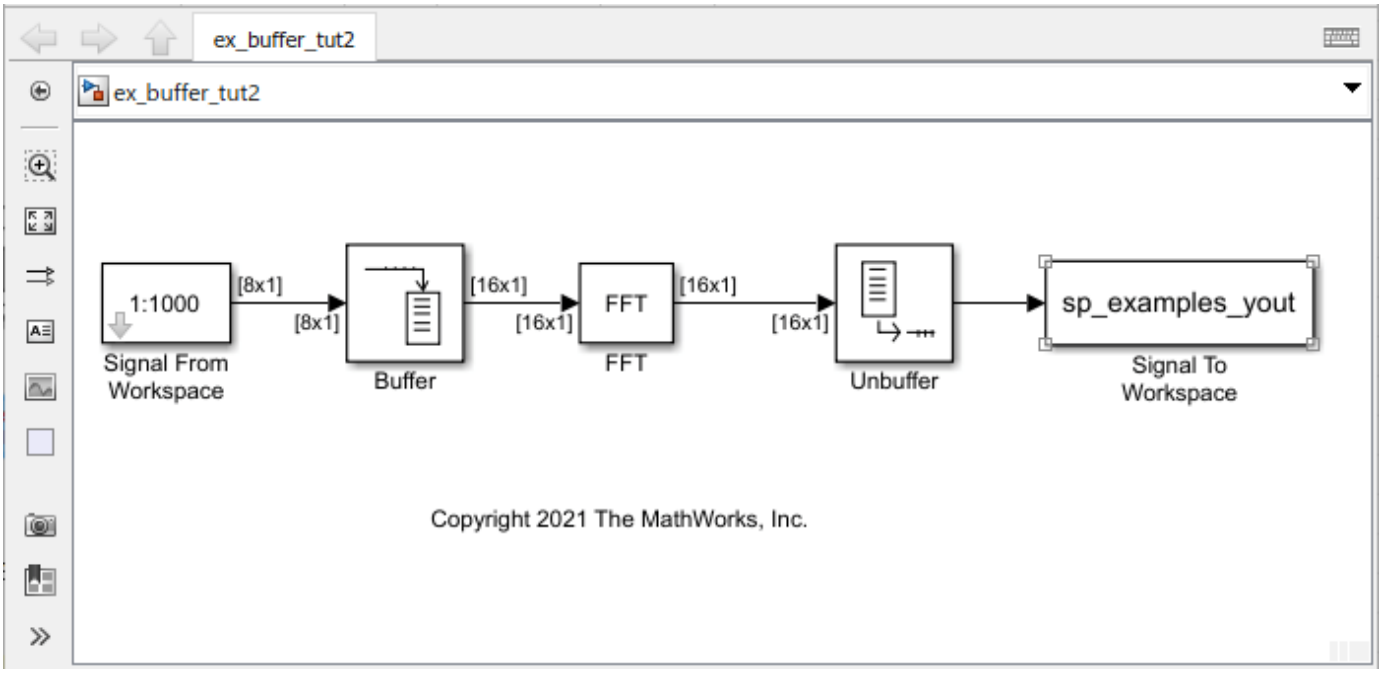
The Buffer block has the **Output buffer size (per channel)** parameter set to 16, and the **Buffer overlap** parameter is set to 4. The Buffer block rebuffers the signal from a frame size of 8 to a frame size of 16. After the initial output, the first four samples of each output frame are made up of the last four samples from the previous output frame.



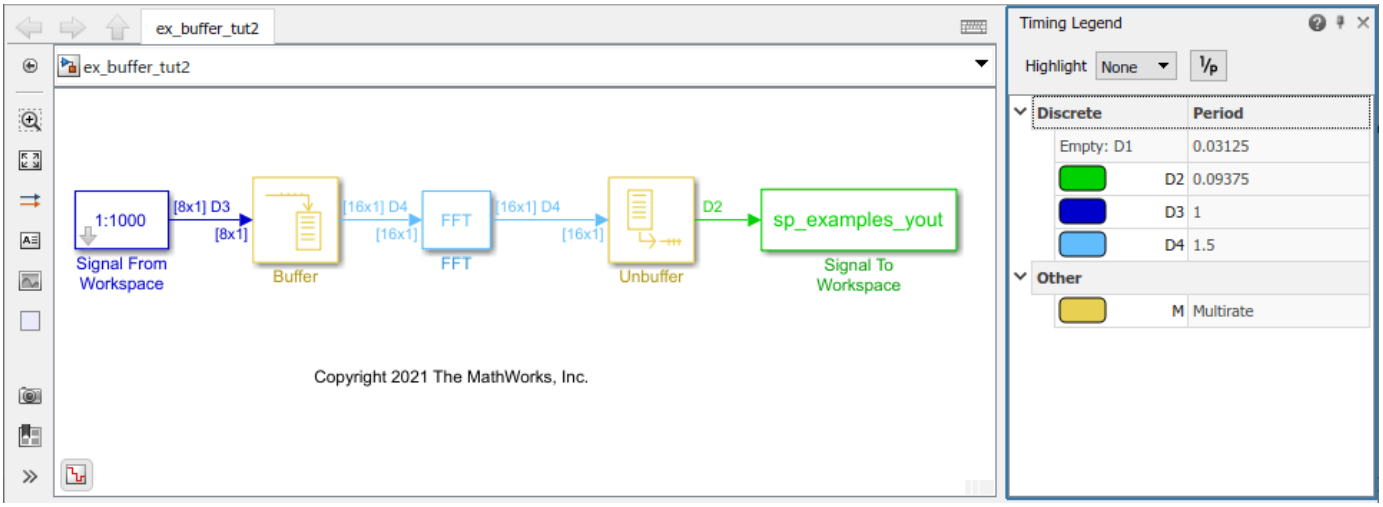
Copyright 2021 The MathWorks, Inc.

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. The signal dimensions in the model confirm the following:

- Buffer block changes the frame size of the signal from 8 to 16.
- Unbuffer block unbuffers the signal into a sequence of scalar outputs.



To view the effect on the frame period of the signal, enable color coding, annotations, and timing legend by selecting **Information Overlays > Colors**, **Text**, **Timing Legend**. In the Timing Legend, you can view the value of the frame period for each signal in the model, the color associated with the frame period, and the corresponding annotation.



Accounting for the overlap, the output frame period T_{fo} of the Buffer block is given by the following equation:

$$T_{fo} = (M_o - L) \times T_{si}$$

where M_o is the output frame size and equals 16, L is the overlap and equals 4, and T_{si} is the input sample period and equals 0.125 seconds. Substituting these values, the output frame period of the Buffer block T_{fo} becomes $(16 - 4) \times 0.125$ or 1.5 seconds. The corresponding sample period of the signal $T_{so} = T_{fo}/M_o$ equals $1.5/16$ or 0.0938 seconds. When you unbuffer the signal into a sequence of sample outputs, the frame period of the signal (shown as D2 in the model) matches the sample period value of 0.0938 seconds. Thus, both the data and the sample period of the signal 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
- “Convert Sample and Frame Rates in Simulink Using Rate Conversion Blocks” on page 3-16

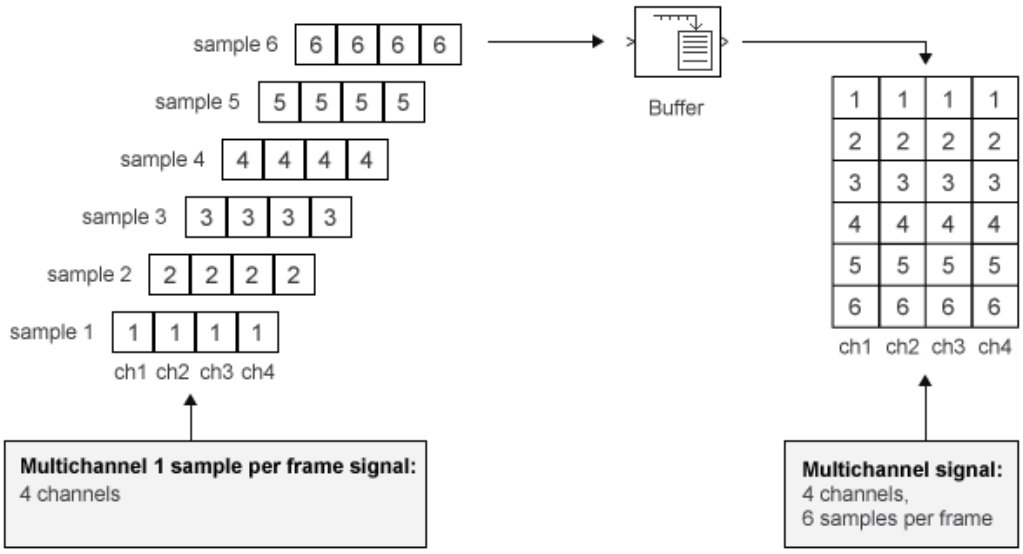
Buffering and Frame-Based Processing

In this section...
“Buffer Input into Frames” on page 3-29
“Buffer Signals into Frames with Overlap” on page 3-31
“Buffer Frame Inputs into Other Frame Inputs” on page 3-33
“Buffer Delay and Initial Conditions” on page 3-35
“Unbuffer Frame Signals into Sample Signals” on page 3-36

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.

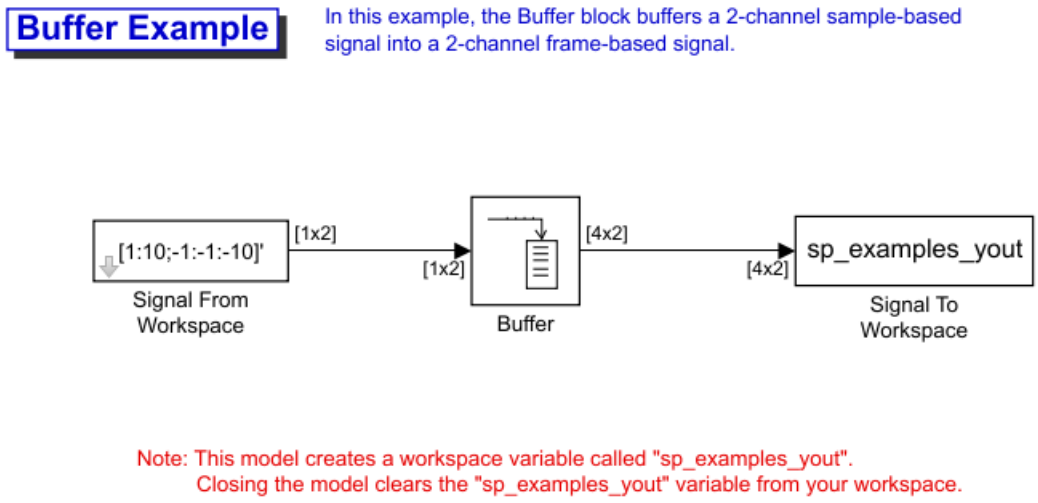
The following figure is a graphical representation of a signal with frame size 1 being converted into a signal of frame size *L* by the Buffer block.



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:

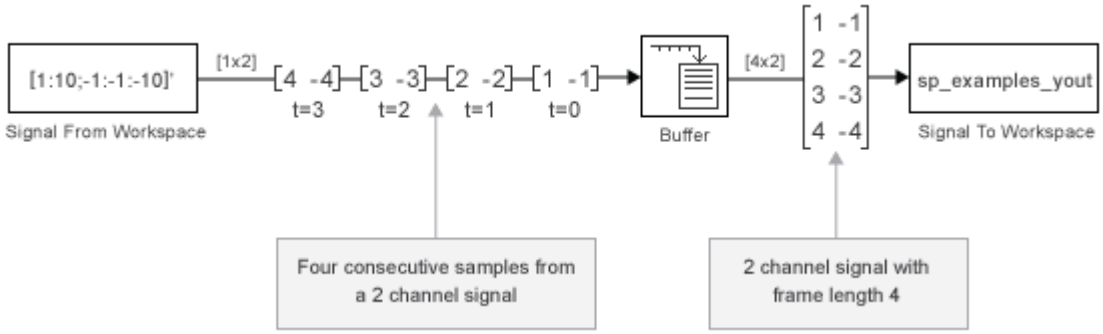
   - **Signal** = `[1:10;-1:-1:-10]'`
   - **Sample time** = `1`
   - **Samples per frame** = `1`
   - **Form output after final data value** = `Setting to zero`

   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.

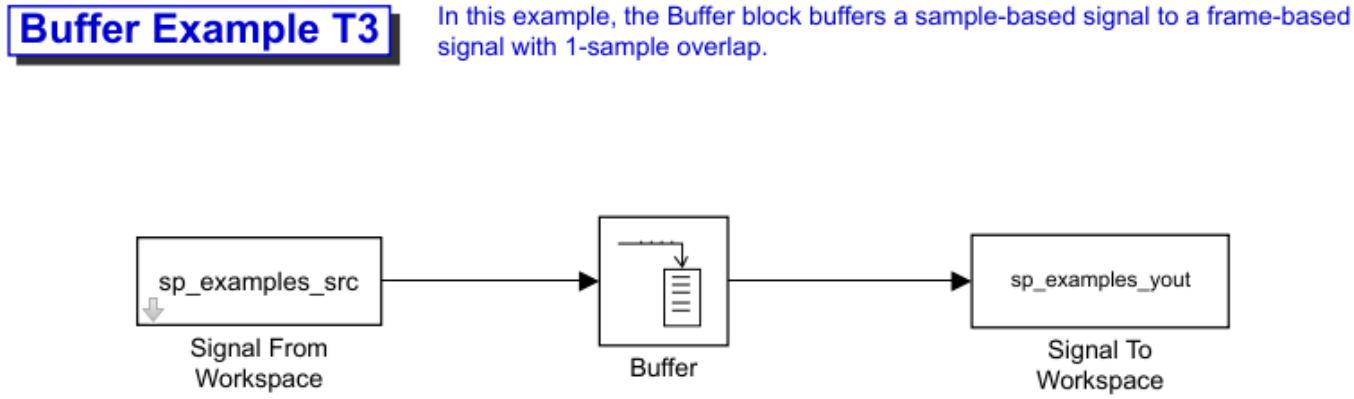
The **Buffer overlap** parameter of the Buffer block specifies the number of overlap points, L. In the overlap case ($L > 0$), the frame period for the output is $(M_o\text{-}L)*T_{si}$, where T_{si} is the input sample period and M_o is the **Buffer size**.

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:

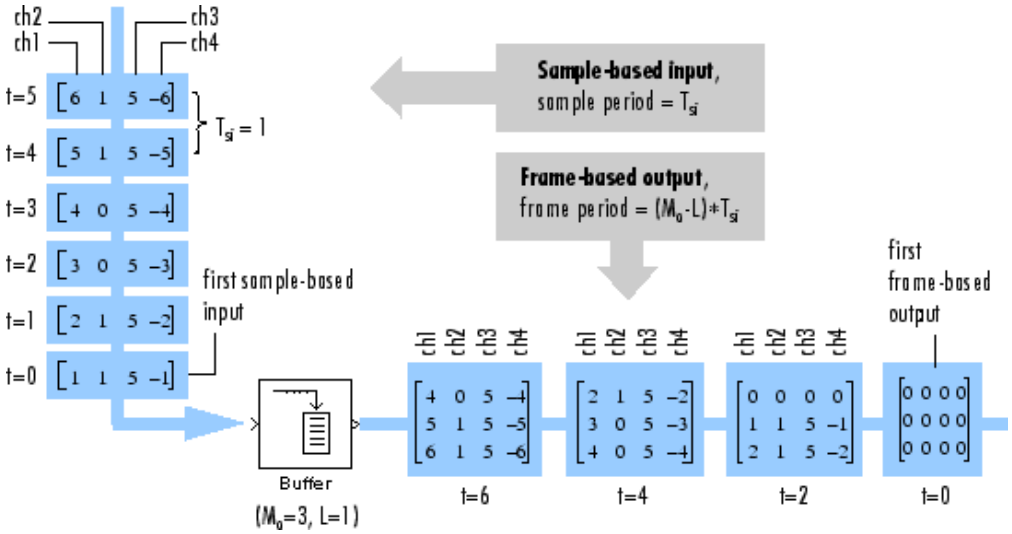
   - **Signal** = `sp_examples_src`
   - **Sample time** = `1`
   - **Samples per frame** = `1`
   - **Form output after final data value by** = `Setting to zero`

   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 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 latency.

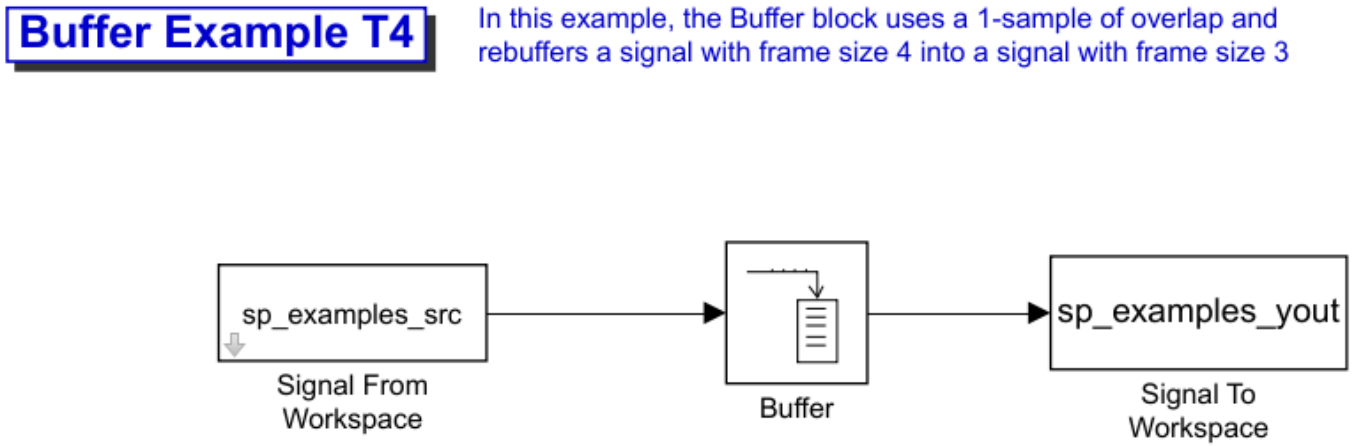
See “Excess Algorithmic Delay (Tasking Latency)” on page 3-45 for general information about algorithmic delay. For instructions on how to calculate buffering delay, see “Buffer Delay and Initial Conditions” on page 3-35.

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.

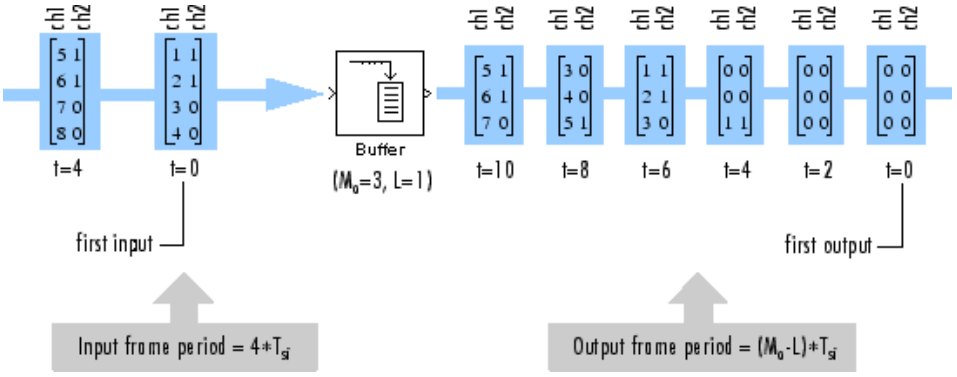
6 Set the block parameters as follows, and then click **OK**:

- **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 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 latency.

See “Excess Algorithmic Delay (Tasking Latency)” on page 3-45 for general information about algorithmic delay. For instructions on how to calculate buffering delay, and see “Buffer Delay and Initial Conditions” on page 3-35.

Buffer Delay and Initial Conditions

In the examples “Buffer Signals into Frames with Overlap” on page 3-31 and “Buffer Frame Inputs into Other Frame Inputs” on page 3-33, 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 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 3-33 using the following command at the MATLAB command line:

```
d = rebuffer_delay(4,3,1)
d = 8
```

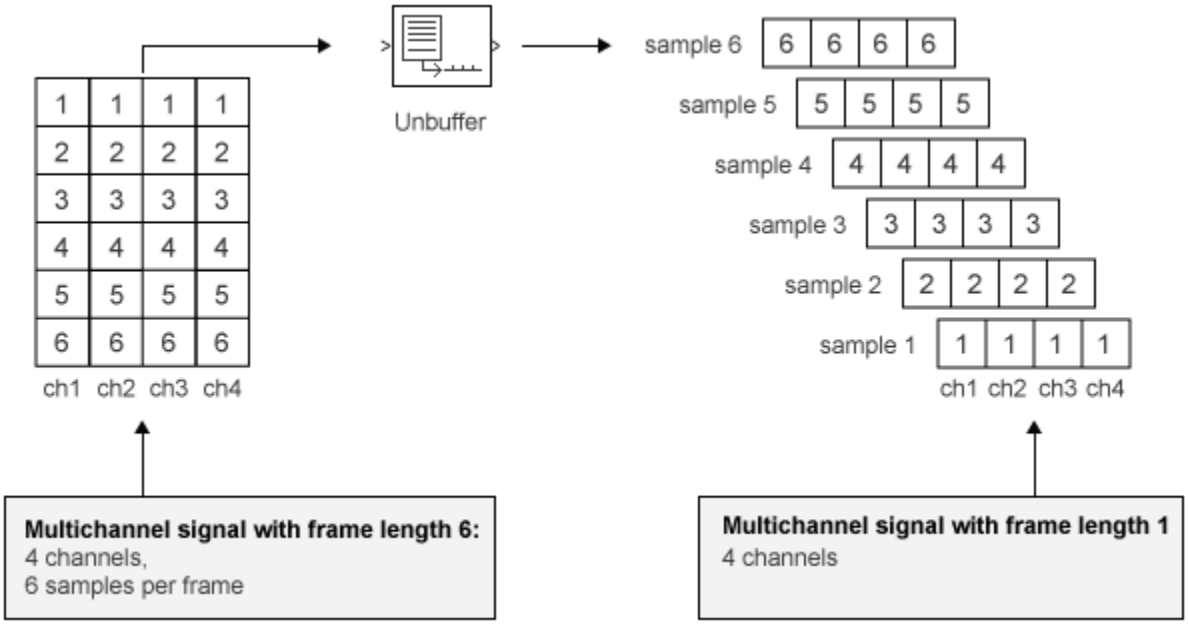
This result agrees with the block output in that example. Notice that this model was simulated in Simulink multitasking mode.

For more information about delay, see “Excess Algorithmic Delay (Tasking Latency)” on page 3-45. 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.

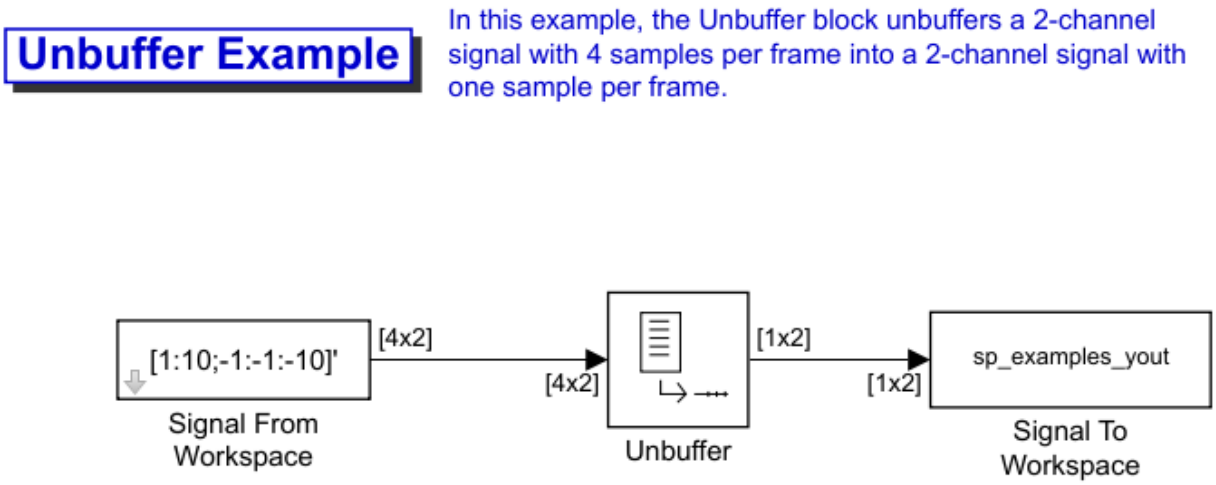
$$T_{so} = T_{fi}/M_i$$

The Unbuffer block always preserves the signal sample period ($T_{so} = T_{si}$). See “Convert Sample and Frame Rates in Simulink Using Frame Rebuffering Blocks” on page 3-23 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.

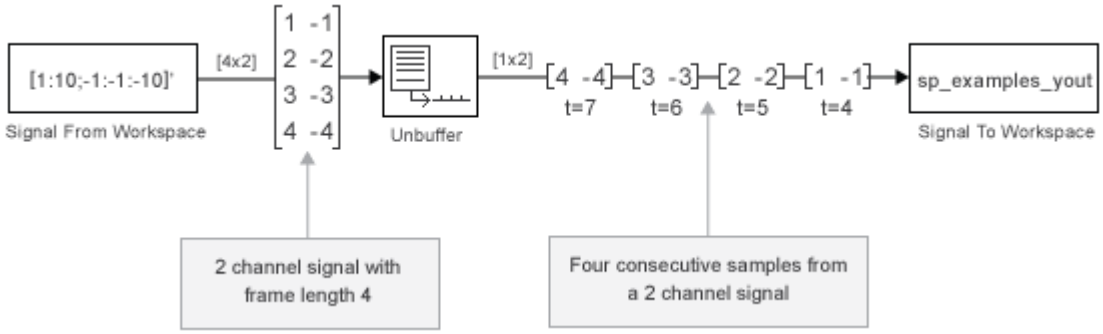


2. Double-click the Signal From Workspace block. The **Source Block Parameters: Signal From Workspace** dialog box opens.
3. Set the block parameters as follows:
   - **Signal** = `[1:10;-1:-1:-10]'`
   - **Sample time** = `1`
   - **Samples per frame** = `4`
   - **Form output after final data value 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 3-40

Delay and Latency

In this section...
“Computational Delay” on page 3-40
“Algorithmic Delay” on page 3-41
“Zero Algorithmic Delay” on page 3-41
“Basic Algorithmic Delay” on page 3-43
“Excess Algorithmic Delay (Tasking Latency)” on page 3-45
“Predict Tasking Latency” on page 3-46

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 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 delay. It is generally expressed in terms of the number of samples by which a block output lags behind the corresponding input. This delay is directly related to the time elapsed on the Simulink timer during the execution of the block.

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 block delay using the following categories:

- “Zero Algorithmic Delay” on page 3-41
- “Basic Algorithmic Delay” on page 3-43
- “Excess Algorithmic Delay (Tasking Latency)” on page 3-45

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.

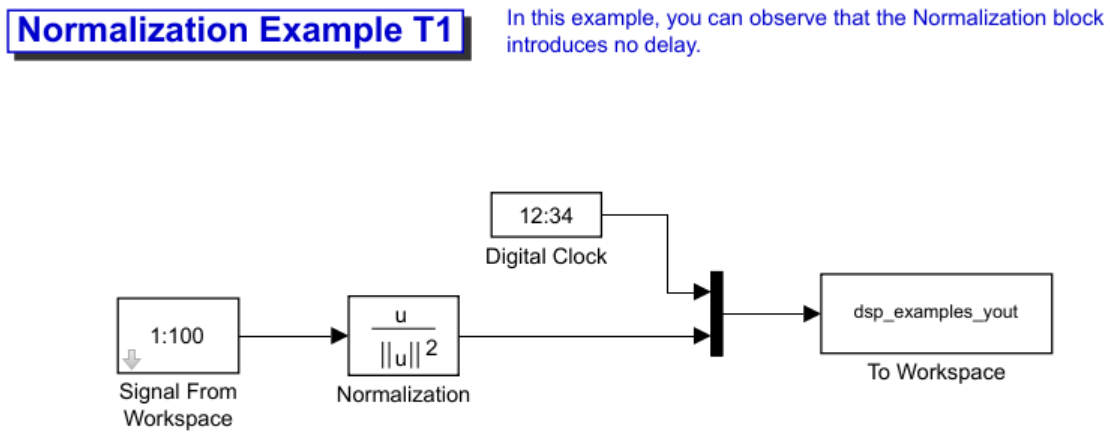
Zero Algorithmic Delay

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

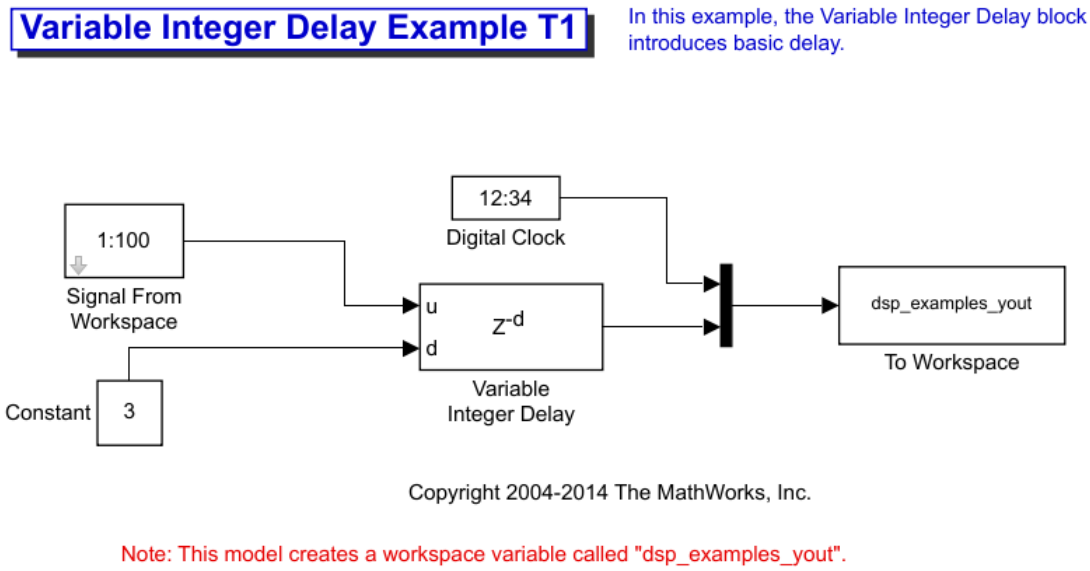
When several blocks with zero algorithmic delay are connected in a feedback loop, Simulink may report an algebraic loop error and performance may generally suffer. You can prevent algebraic loops by injecting at least one sample of delay into a feedback loop , for example, by including a Delay block 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 algorithmic delay of the block. 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 In port of the block at time $t = 0$ is released to the output at time $t = 3$.

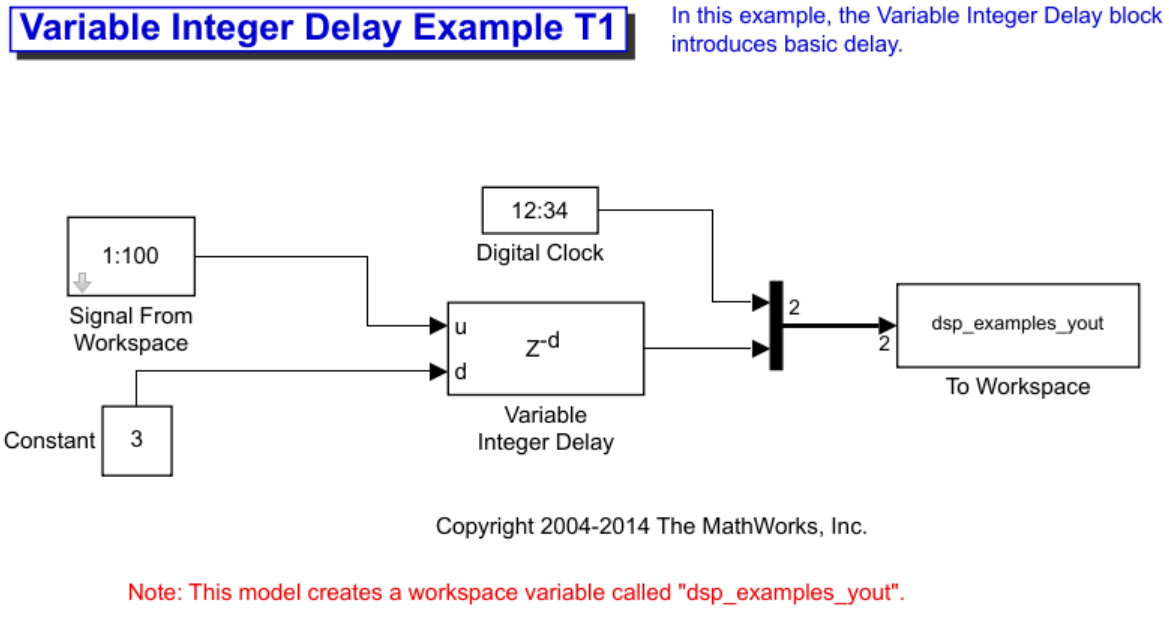
7 Double-click the Variable Integer Delay block. The **Function Block Parameters: Variable Integer Delay** dialog box opens.

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 **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 algorithm. This excess algorithmic delay is called tasking latency, because it arises from synchronization requirements of the Simulink tasking mode. The overall algorithmic delay of a block 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 3-45
- “Block Rate Type” on page 3-46
- “Model Rate Type” on page 3-46
- “Block Input Processing Mode” on page 3-46

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 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

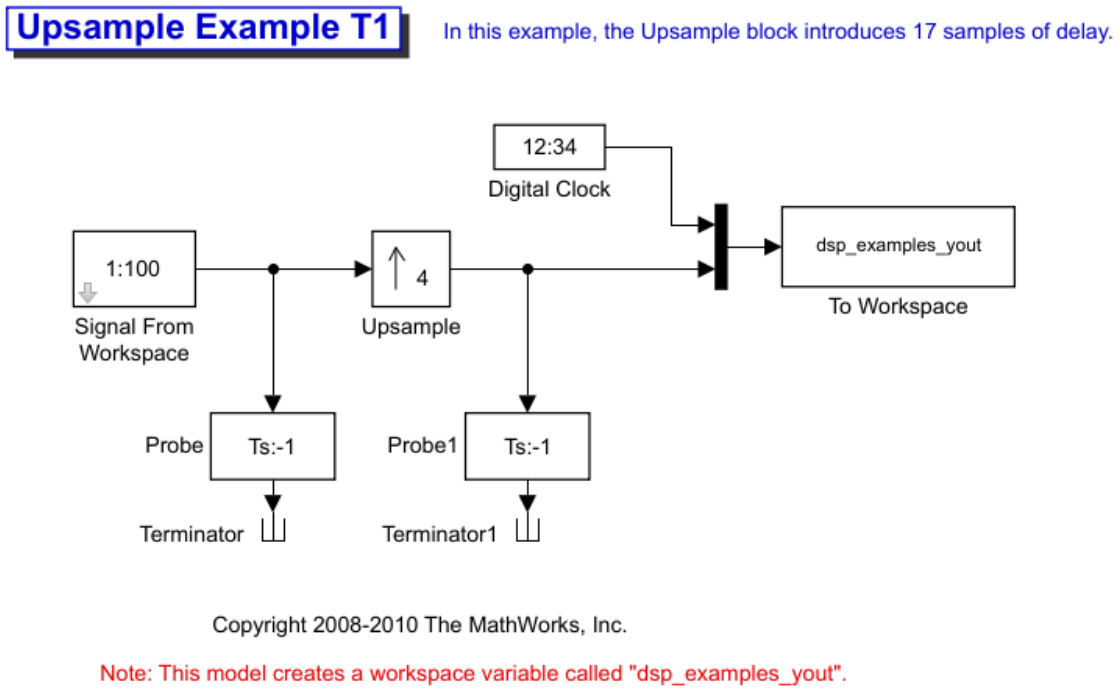
Many blocks can operate in either sample-based or frame-based processing modes. To choose, you can set the **Input processing** parameter of the block to `Columns as channels (frame based)` or `Elements as channels (sample based)`.

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 reference page. In this topic, you use the Upsample block 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.

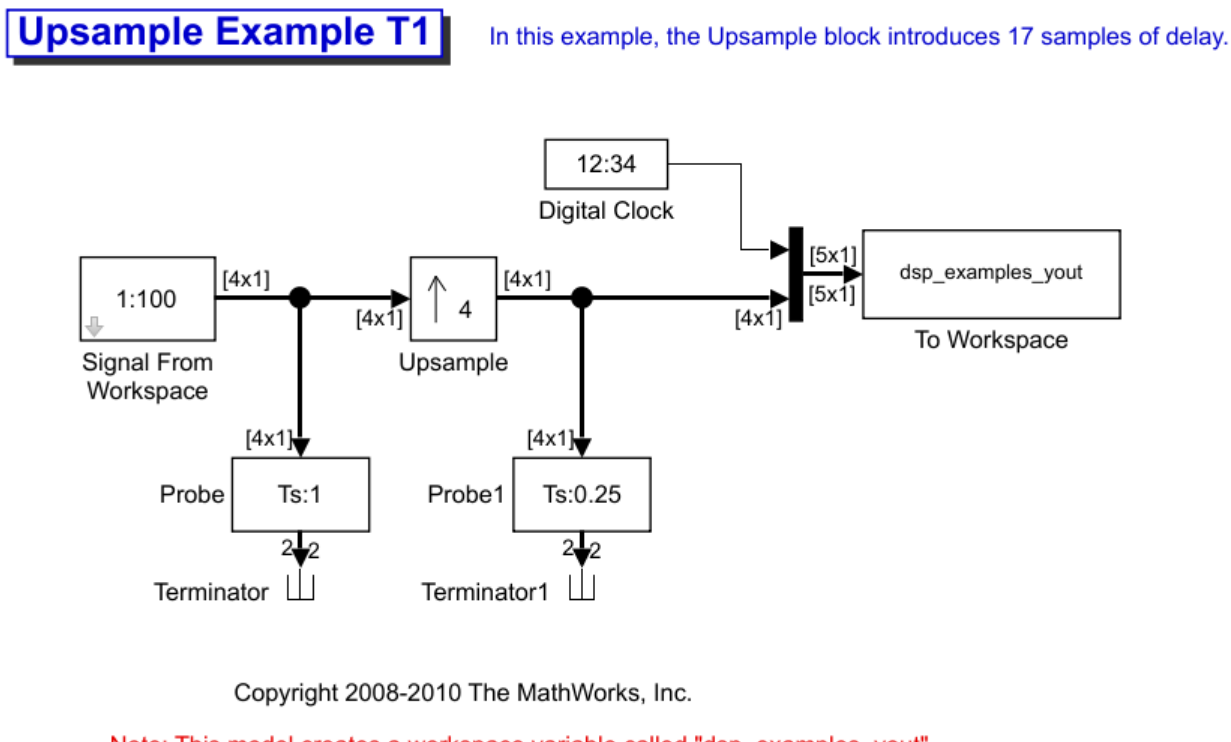
9 Double-click the Digital Clock block. The **Source Block Parameters: Digital Clock** dialog box opens.

10 Set the **Sample time** parameter to `0.25`, and then click **OK**.

This matches the sample period of the Upsample block 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 =
```

time				
0	-1.0000	0	0	0
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
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

1st output frame

5th output frame

"Latency and Initial Conditions" in the Upsample block reference page indicates that when Simulink is in multitasking mode, the first sample of the block 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 **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 =
```

time				
0	1.0000	0	0	0
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
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

1st output frame

5th output frame

"Latency and Initial Conditions" in the Upsample block 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 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 3-29

Variable-Size Signal Support DSP System Objects

In this section...
“Variable-Size Signal Support Example” on page 3-51
“DSP System Toolbox System Objects That Support Variable-Size Signals” on page 3-51

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.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.PeakToRMS`
Signal Operations
`dsp.DCBlocker`
`dsp.Delay`
`dsp.VariableFractionalDelay`
`dsp.PhaseExtractor`
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.

DSP System Toolbox Featured Examples

- “Wavelet Denoising” on page 4-3
- “LPC Analysis and Synthesis of Speech” on page 4-7
- “Streaming Signal Statistics” on page 4-12
- “High Resolution Spectral Analysis in MATLAB” on page 4-16
- “High Resolution Spectral Analysis in Simulink” on page 4-33
- “Zoom FFT” on page 4-39
- “Outlier Removal Techniques with Streaming ECG Signals” on page 4-49
- “Sigma-Delta A/D Conversion” on page 4-53
- “GSM Digital Down Converter in Simulink” on page 4-55
- “Overlap-Add/Save” on page 4-61
- “Designing Low Pass FIR Filters” on page 4-64
- “Classic IIR Filter Design” on page 4-79
- “Efficient Narrow Transition-Band FIR Filter Design” on page 4-89
- “IIR Filter Design Given a Prescribed Group Delay” on page 4-96
- “FIR Nyquist (L-th band) Filter Design” on page 4-105
- “FIR Halfband Filter Design” on page 4-112
- “Arbitrary Magnitude Filter Design” on page 4-128
- “Design of Peaking and Notching Filters” on page 4-141
- “Fractional Delay Filters Using Farrow Structures” on page 4-151
- “Least Pth-norm Optimal FIR Filter Design” on page 4-159
- “Least Pth-Norm Optimal IIR Filter Design” on page 4-169
- “Multistage Rate Conversion” on page 4-177
- “Complex Bandpass Filter Design” on page 4-190
- “Design of Fractional Delay FIR Filters” on page 4-196
- “Design of Decimators and Interpolators” on page 4-211
- “Multistage Halfband IIR Filter Design” on page 4-229
- “Efficient Sample Rate Conversion Between Arbitrary Factors” on page 4-234
- “Reconstruction Through Two-Channel Filter Banks” on page 4-241
- “Adaptive Line Enhancer (ALE)” on page 4-248
- “Adaptive Noise Canceling (ANC) Applied to Fetal Electrocardiography” on page 4-256
- “Adaptive Noise Cancellation Using RLS Adaptive Filtering” on page 4-259
- “System Identification Using RLS Adaptive Filtering” on page 4-264
- “Acoustic Noise Cancellation (LMS)” on page 4-270
- “Adaptive Filter Convergence” on page 4-272
- “Noise Canceler (RLS)” on page 4-275

- “Time-Delay Channel Estimation Through Adaptive Filtering” on page 4-278
- “Time Scope Measurements” on page 4-281
- “Spectrum Analyzer Measurements” on page 4-290
- “Generate a Multithreaded MEX File from a MATLAB Function Using Unfolding” on page 4-297
- “Generate Standalone Executable And Interact With It Using UDP” on page 4-303
- “Code Generation for Parametric Audio Equalizer” on page 4-306
- “Generate DSP Applications with MATLAB Compiler” on page 4-314
- “Optimized Fixed-Point FIR Filters” on page 4-320
- “Floating-Point to Fixed-Point Conversion of IIR Filters” on page 4-328
- “GSM Digital Down Converter in MATLAB” on page 4-346
- “Cochlear Implant Speech Processor” on page 4-353
- “Three-Channel Wavelet Transmultiplexer” on page 4-358
- “Arbitrary Magnitude and Phase Filter Design” on page 4-364
- “G.729 Voice Activity Detection” on page 4-381
- “IF Subsampling with Complex Multirate Filters” on page 4-385
- “Design and Analysis of a Digital Down Converter” on page 4-395
- “Comparison of LDM, CVSD, and ADPCM” on page 4-404
- “Digital Up and Down Conversion for Family Radio Service” on page 4-409
- “Parametric Audio Equalizer” on page 4-426
- “Envelope Detection” on page 4-430
- “DTMF Generator and Receiver” on page 4-438
- “WWV Digital Receiver - Synchronization and Detection” on page 4-441
- “Real-Time ECG QRS Detection” on page 4-448
- “Internet Low Bitrate Codec (iLBC) for VoIP” on page 4-454

Wavelet Denoising

This example shows how to use the `DyadicAnalysis` and `DyadicSynthesis` System objects to remove noise from a signal.

Introduction

Wavelets have an important application in signal denoising. After wavelet decomposition, the high frequency subbands contain most of the noise information and little signal information. In this example, soft thresholding is applied to the different subbands. The threshold is set to higher values for high frequency subbands and lower values for low frequency subbands.

Initialization

Creating and initializing the System objects before they are used in a processing loop is critical in obtaining optimal performance.

```
load dspwlets; % load wavelet coefficients and noisy signal
Threshold = [3 2 1 0];
```

Create a `SignalSource` System object to output the noisy signal.

```
signalGenerator = dsp.SignalSource(noisdopp.', 64);
```

Create and configure a `DyadicAnalysisFilterBank` System object for wavelet decomposition of the signal.

```
dyadicAnalysis = dsp.DyadicAnalysisFilterBank( ...
    'CustomLowpassFilter', lod, ...
    'CustomHighpassFilter', hid, ...
    'NumLevels', 3);
```

Create three `Delay` System objects to compensate for the system delay introduced by the wavelet components.

```
delay1 = dsp.Delay(3*(length(lod)-1));
delay2 = dsp.Delay(length(lod)-1);
delay3 = dsp.Delay(7*(length(lod)-1));
```

Create and configure a `DyadicSynthesisFilterBank` System object for wavelet reconstruction of the signal.

```
dyadicSynthesis = dsp.DyadicSynthesisFilterBank( ...
    'CustomLowpassFilter', lor, ...
    'CustomHighpassFilter', hir, ...
    'NumLevels', 3);
```

Create time scope System object to plot the original, denoised and residual signals.

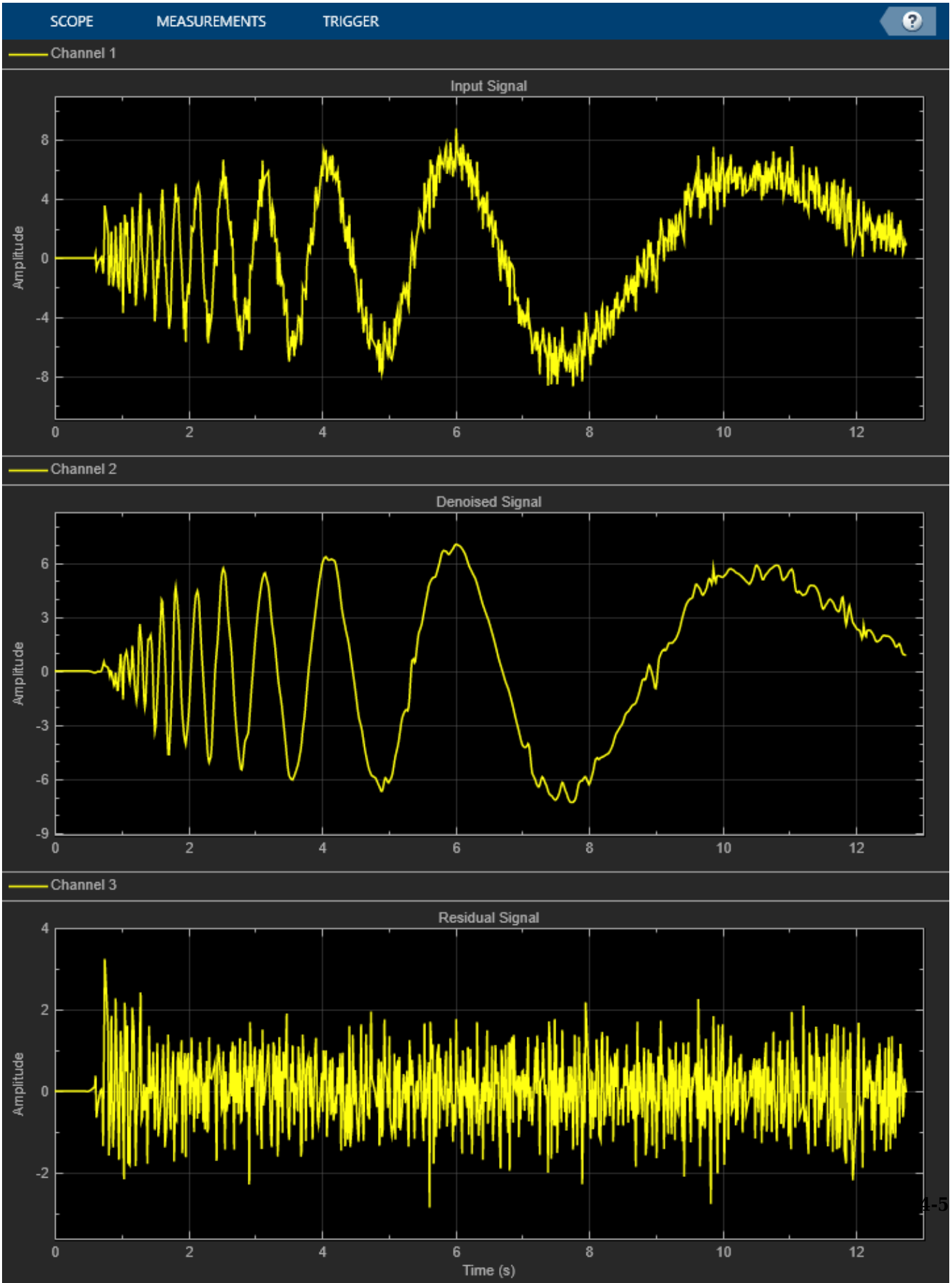
```
scope = timescope('Name', 'Wavelet Denoising', ...
  'SampleRate', fs, ...
  'TimeSpanSource','property',...
  'TimeSpan', 13, ...
  'LayoutDimensions',[3 1], ...
  'TimeAxisLabels', 'Bottom');
pos = scope.Position;
scope.Position = [pos(1) pos(2)-(0.5*pos(4)) 0.9*pos(3) 2*pos(4)];
```

```
% Set properties for each display
scope.ActiveDisplay = 1;
scope.Title = 'Input Signal';

scope.ActiveDisplay = 2;
scope.Title = 'Denoised Signal';

scope.ActiveDisplay = 3;
scope.Title = 'Residual Signal';
```

Stream Processing Loop

Create a processing loop to denoise the input signal. This loop uses the System objects you instantiated above.

```
for ii = 1:length(noisdopp)/64
    sig = signalGenerator();      % Input noisy signal
    S = dyadicAnalysis(sig);      % Dyadic analysis

    % separate into four subbands
    S1 = S(1:32);  S2 = S(33:48);  S3 = S(49:56);  S4 = S(57:64);

    % Delay to compensate for the dyadic analysis filters
    S1 = delay1(S1);
    S2 = delay2(S2);

    S1 = dspDeadZone(S1, Threshold(1));
    S2 = dspDeadZone(S2, Threshold(2));
    S3 = dspDeadZone(S3, Threshold(3));
    S4 = dspDeadZone(S4, Threshold(4));

    % Dyadic synthesis (on concatenated subbands)
    S = dyadicSynthesis([S1; S2; S3; S4]);

    sig_delay = delay3(sig);   % Delay to compensate for analysis/synthesis.
    Error = sig_delay - S;

    % Plot the results
    scope(sig_delay, S, Error);
end
release(scope);
```


SCOPE
MEASUREMENTS
TRIGGER
Channel 1
Input Signal
Amplitude
8
4
0
-4
-8
0
2
4
6
8
10
12
Channel 2
Denoised Signal
Amplitude
6
3
0
-3
-6
-9
0
2
4
6
8
10
12
Channel 3
Residual Signal
Amplitude
4
2
0
-2
0
2
4
6
8
10
12
Time (s)

Summary

This example used signal processing System objects such as the `DyadicAnalysisFilterBank` and `DyadicSynthesisFilterBank` to denoise a noisy signal using user-specified thresholds. The Input Signal window shows the original noisy signal, the Denoised Signal window shows the signal after suppression of noise, and the Residue Signal window displays the error between the original and denoised signal.

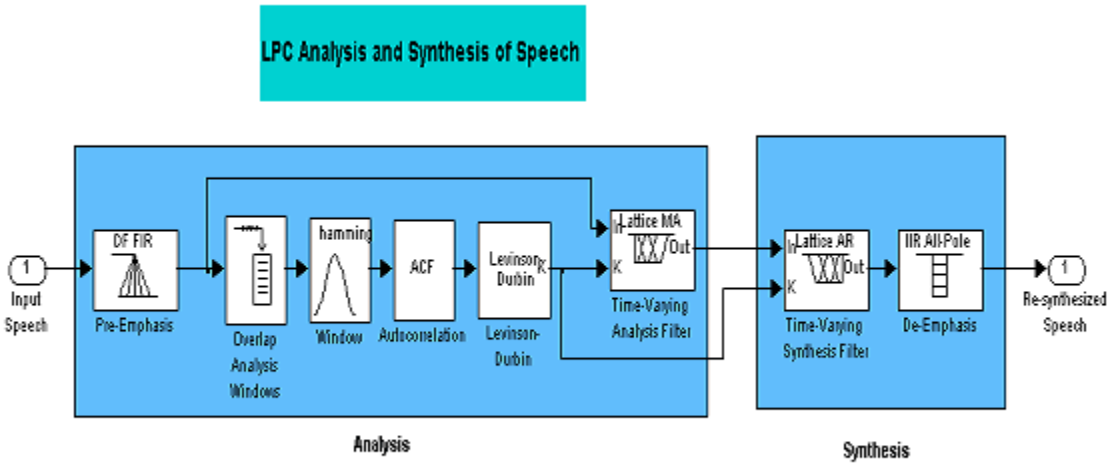
LPC Analysis and Synthesis of Speech

This example shows how to implement a speech compression technique known as Linear Prediction Coding (LPC) using DSP System Toolbox™ functionality available at the MATLAB® command line.

Introduction

In this example you implement LPC analysis and synthesis (LPC coding) of a speech signal. This process consists of two steps: analysis and synthesis. In the analysis section, you extract the reflection coefficients from the signal and use it to compute the residual signal. In the synthesis section, you reconstruct the signal using the residual signal and reflection coefficients. The residual signal and reflection coefficients require less number of bits to code than the original speech signal.

The block diagram below shows the system you will implement.



In this simulation, the speech signal is divided into frames of size 3200 samples, with an overlap of 1600 samples. Each frame is windowed using a Hamming window. Twelfth-order autocorrelation coefficients are found, and then the reflection coefficients are calculated from the autocorrelation coefficients using the Levinson-Durbin algorithm. The original speech signal is passed through an analysis filter, which is an all-zero filter with coefficients as the reflection coefficients obtained above. The output of the filter is the residual signal. This residual signal is passed through a synthesis filter which is the inverse of the analysis filter. The output of the synthesis filter is the original signal.

Initialization

Here you initialize some of the variables like the frame size and also instantiate the System objects used in your processing. These objects also pre-compute any necessary variables or tables resulting in efficient processing calls later inside a loop.

Initialize variables.

```
frameSize = 1600;
fftLen = 2048;
```

Here you create a System object to read from an audio file and determine the file's audio sampling rate.

```
audioReader = dsp.AudioFileReader('SamplesPerFrame', frameSize, ...
            'OutputDataType', 'double');

fileInfo = info(audioReader);
Fs = fileInfo.SampleRate;
```

Create an FIR digital filter System object used for pre-emphasis.

```
preEmphasisFilter = dsp.FIRFilter(...
        'Numerator', [1 -0.95]);
```

Create a buffer System object and set its properties such that you get an output of twice the length of the frameSize with an overlap length of frameSize.

```
signalBuffer = dsp.AsyncBuffer(2*frameSize);
```

Create an FIR digital filter System object used for analysis. Also create two all-pole digital filter System objects used for synthesis and de-emphasis.

```
analysisFilter = dsp.FIRFilter(...
                    'Structure','Lattice MA',...
                    'ReflectionCoefficientsSource', 'Input port');

synthesisFilter = dsp.AllpoleFilter('Structure','Lattice AR');

deEmphasisFilter = dsp.AllpoleFilter('Denominator',[1 -0.95]);
```

Create a System object to play the resulting audio.

```
audioWriter = audioDeviceWriter('SampleRate', Fs);

% Setup plots for visualization.
scope = dsp.SpectrumAnalyzer('SampleRate', Fs, ...
    'PlotAsTwoSidedSpectrum', false, 'YLimits', [-140, 0], ...
    'FrequencyResolutionMethod', 'WindowLength', 'WindowLength', fftLen,...
    'FFTLengthSource', 'Property', 'FFTLength', fftLen, ...
    'Title', 'Linear Prediction of Speech', ...
    'ShowLegend', true, 'ChannelNames', {'Signal', 'LPC'});
```

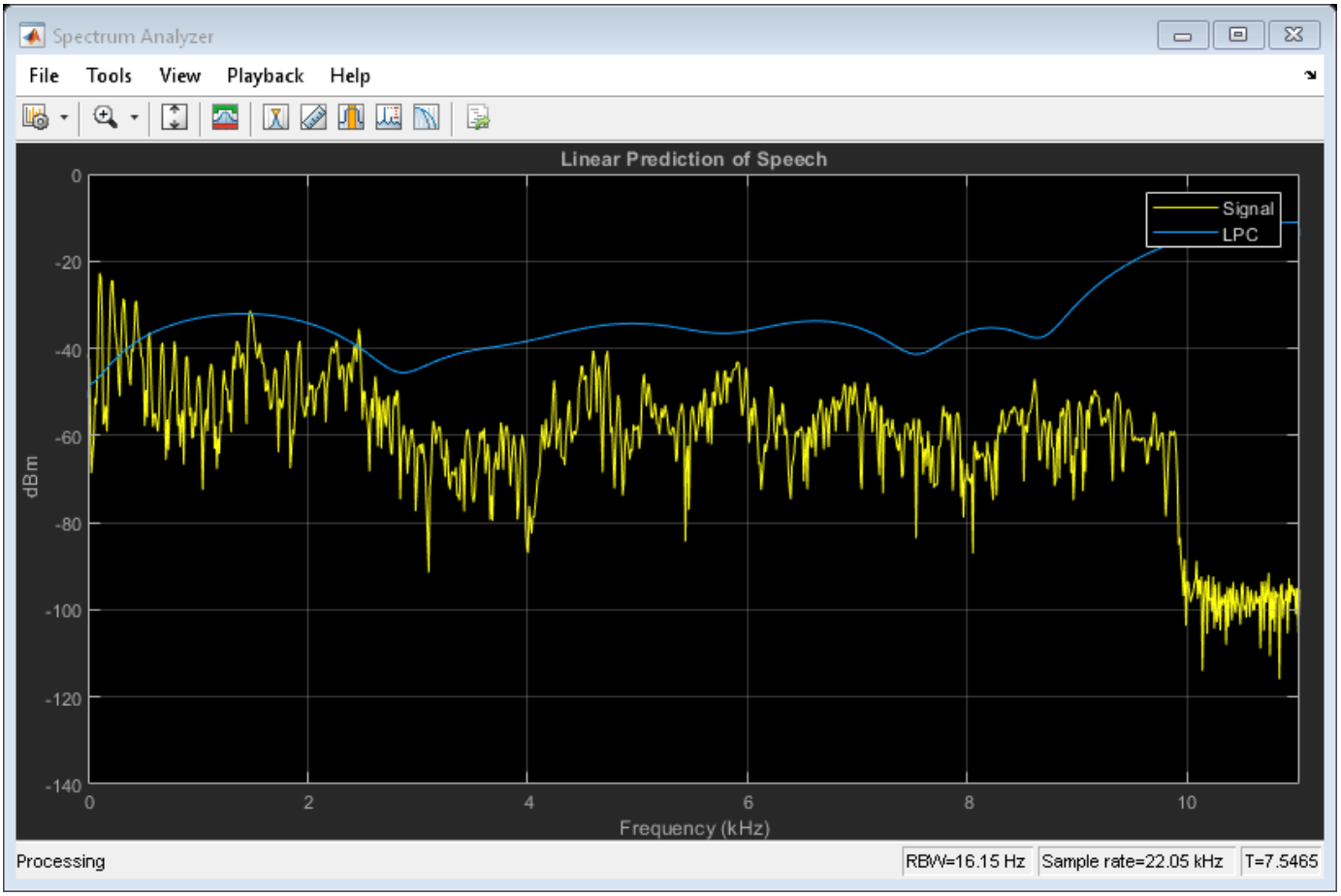
Stream Processing Loop

Here you call your processing loop where you do the LPC analysis and synthesis of the input audio signal using the System objects you have instantiated.

The loop is stopped when you reach the end of the input file, which is detected by the `AudioFileReader` System object.

```
while ~isDone(audioReader)
    % Read audio input
    sig = audioReader();

    % Analysis
    % Note that the filter coefficients are passed in as an argument to the
    % analysisFilter System object.
    sigpreem   = preEmphasisFilter(sig);
```

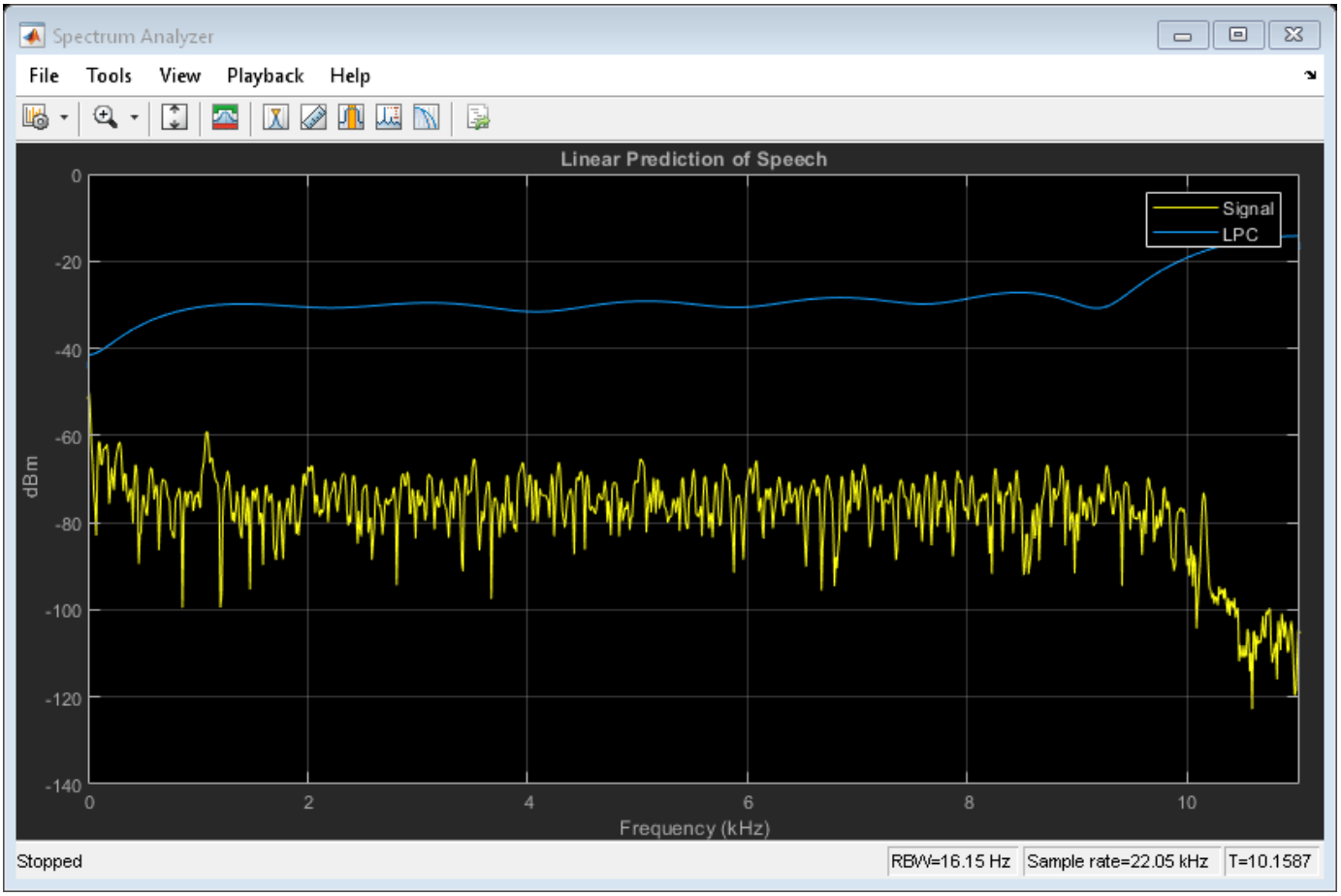
```
        write(signalBuffer,sigpreem);
        sigbuf     = read(signalBuffer,2*frameSize, frameSize);
        hammingwin = hamming(2*frameSize);
        sigwin     = hammingwin.*sigbuf;

        % Autocorrelation sequence on [0:13]
        sigacf = xcorr(sigwin, 12, 'biased');
        sigacf = sigacf(13:end);

        % Compute the reflection coefficients from auto-correlation function
        % using the Levinson-Durbin recursion. The function outputs both
        % polynomial coefficients and reflection coefficients. The polynomial
        % coefficients are used to compute and plot the LPC spectrum.
        [sigA, ~, sigK] = levinson(sigacf); % Levinson-Durbin
        siglpc          = analysisFilter(sigpreem, sigK);

        % Synthesis
        synthesisFilter.ReflectionCoefficients = sigK.';
        sigsyn = synthesisFilter(siglpc);
        sigout = deEmphasisFilter(sigsyn);

        % Play output audio
        audioWriter(sigout);

        % Update plots
        sigA_padded = zeros(size(sigwin), 'like', sigA.'); % Zero-padded to plot
        sigA_padded(1:size(sigA.',1), :) = sigA.';
        scope([sigwin, sigA_padded]);
    end
```



Release

Here you call the release method on the System objects to close any open files and devices.

```
release(audioReader);
pause(10*audioReader.SamplesPerFrame/audioReader.SampleRate); % Wait until audio finishes playing
release(audioWriter);
release(scope);
```



Conclusion

You have seen here the implementation of speech compression technique using Linear Prediction Coding. The implementation used the DSP System Toolbox functionality available at the MATLAB command line. The code involves only calling of the successive System objects with appropriate input arguments. This involves no error prone manual state tracking which may be the case for instance for a MATLAB implementation of Buffer.

Streaming Signal Statistics

This example shows how to perform statistical measurements on an input data stream using DSP System Toolbox™ functionality available at the MATLAB® command line. You will compute the signal statistics minimum, maximum, mean, variance and peak-to-RMS and the signal power spectrum density and plot them.

Introduction

This example computes signal statistics using DSP System Toolbox System objects. These objects handle their states automatically reducing the amount of hand code needed to update states reducing the possible chance of coding errors.

These System objects pre-compute many values used in the processing. This is very useful when you are processing signals of same properties in a loop. For example, in computing an FFT in a spectrum estimation application, the values of sine and cosine can be computed and stored once you know the properties of the input and these values can be reused for subsequent calls. Also the objects check only whether the input properties are of same type as previous inputs in each call.

Initialization

Here you initialize some of the variables used in the code and instantiate the System objects used in your processing. These objects also pre-compute any necessary variables or tables resulting in efficient processing calls later inside a loop.

```
frameSize = 1024; % Size of one chunk of signal to be processed in one loop
Fs = 48e3;        % Sample rate
numFrames = 100;  % number of frames to process
```

The input signal in this example is white Gaussian noise passed through a lowpass FIR filter. Create the FIR Filter System object used to filter the noise signal:

```
fir = dsp.FIRFilter('Numerator',fir1(32,.3));
```

Create a Spectrum Estimator System object to estimate the power spectrum density of the input.

```
spect = dsp.SpectrumEstimator('SampleRate',Fs,...
                              'SpectrumType','Power density',...
                              'FrequencyRange','onesided',...
                              'Window','Kaiser');
```

Create System objects to calculate mean, variance, peak-to-RMS, minimum and maximum and set them to running mode. These objects are a subset of statistics System objects available in the product. In running mode, you compute the statistics of the input for its entire length in the past rather than the statistics for just the current input.

```
runMean       = dsp.MovingAverage('SpecifyWindowLength', false);
runVar        = dsp.MovingVariance('SpecifyWindowLength', false);
runPeaktoRMS  = dsp.PeakToRMS('RunningPeakToRMS',true);
runMin        = dsp.MovingMinimum('SpecifyWindowLength', false);
runMax        = dsp.MovingMaximum('SpecifyWindowLength', false);
```

Initialize scope System objects, used to visualize statistics and spectrum

```
meanScope = timescope('SampleRate',Fs,...
                      'TimeSpanSource','property',...
```

```
                        'TimeSpan',numFrames*frameSize/Fs,...
                        'Title','Running Mean',...
                        'YLabel','Mean',...
                        'YLimits',[-0.1 .1],...
                        'Position',[43   308   420   330]);
p2rmsScope = timescope('SampleRate',Fs,...
                        'TimeSpanSource','property',...
                        'TimeSpan',numFrames*frameSize/Fs,...
                        'Title','Running Peak-To-RMS',...
                        'YLabel','Peak-To-RMS',...
                        'YLimits',[0 5],...
                        'Position',[480   308   420   330]);
minmaxScope = timescope('SampleRate',Fs,...
                         'TimeSpanSource','property', ...
                         'TimeSpan',numFrames*frameSize/Fs,...
                         'ShowGrid',true,...
                         'Title',...
                             'Signal with Running Minimum and Maximum',...
                         'YLabel','Signal Amplitude',...
                         'YLimits',[-3 3],...
                         'Position',[43   730   422   330]);
spectrumScope = dsp.ArrayPlot('SampleIncrement',...
                              .5 * Fs/(frameSize/2 + 1 ),...
                              'PlotType','Line',...
                              'Title','Power Spectrum Density',...
                              'XLabel','Frequency (Hz)',...
                              'YLabel','Power/Frequency (dB/Hz)',...
                              'YLimits',[-120 -30],...
                              'Position',[475 730 420 330]);
```

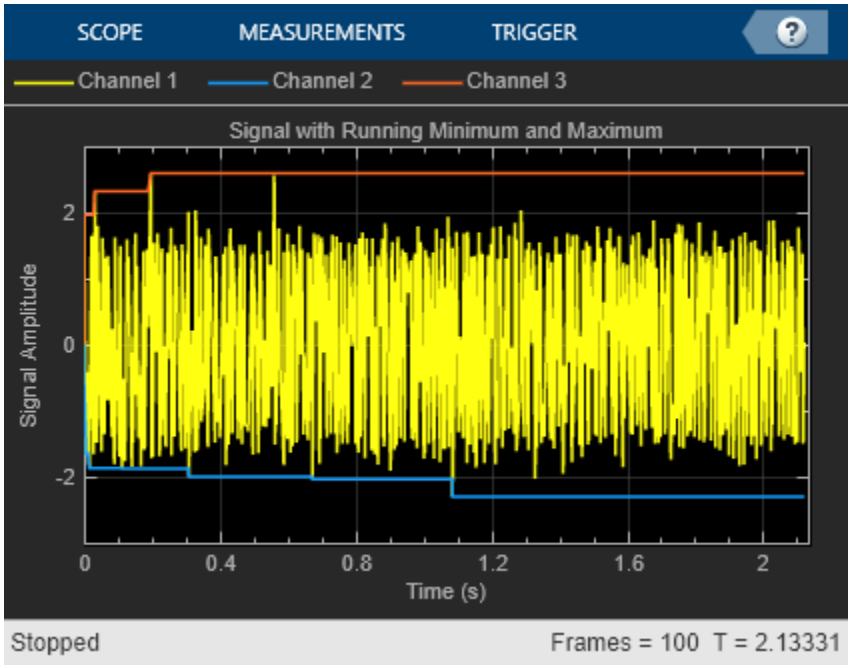
Stream Processing Loop

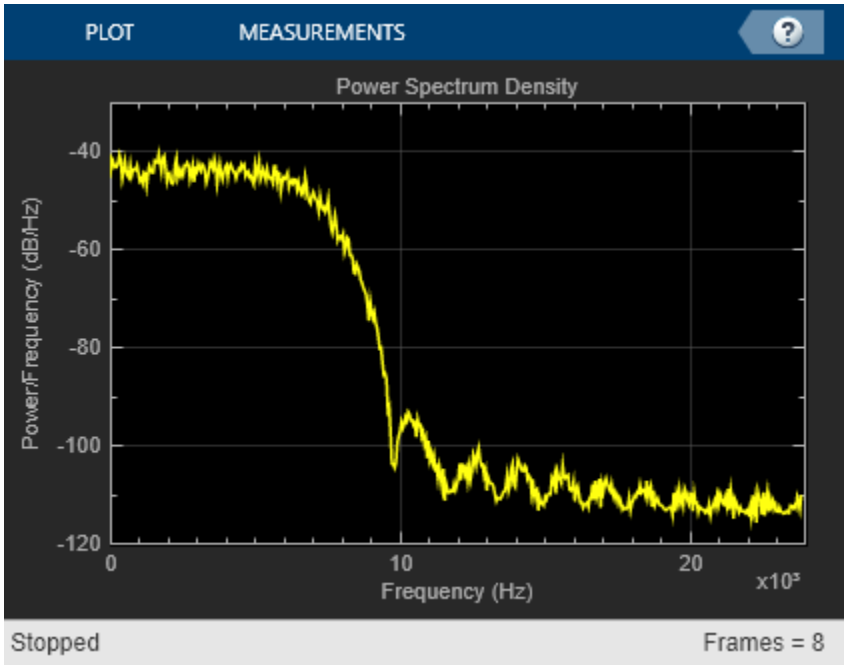
Here you call your processing loop which will filter white Gaussian noise and calculate its mean, variance, peak-to-RMS, min, max and spectrum using the System objects.

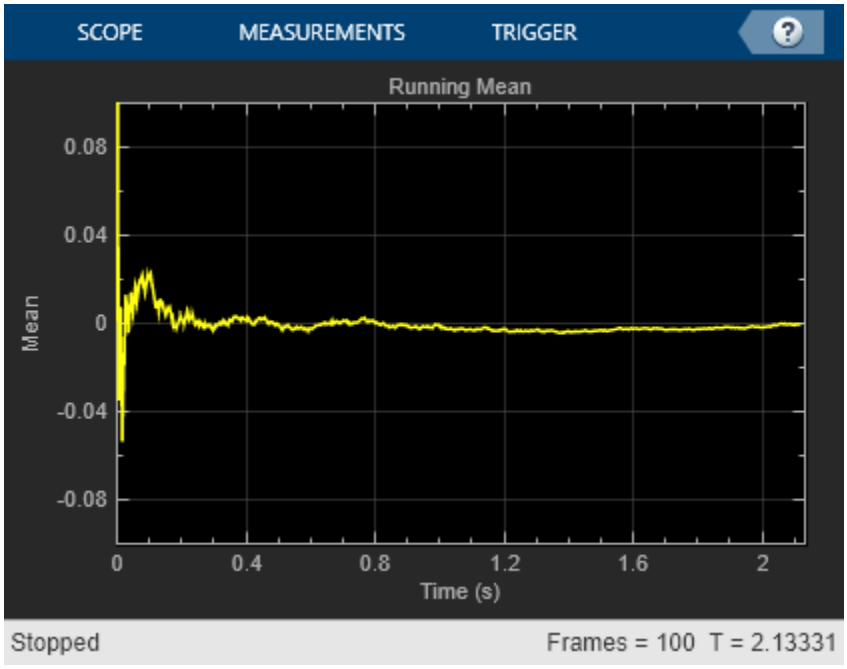
Note that the System objects are called inside the loop. Since the input data properties do not change, this enables reuse of objects here. This reduces memory usage.

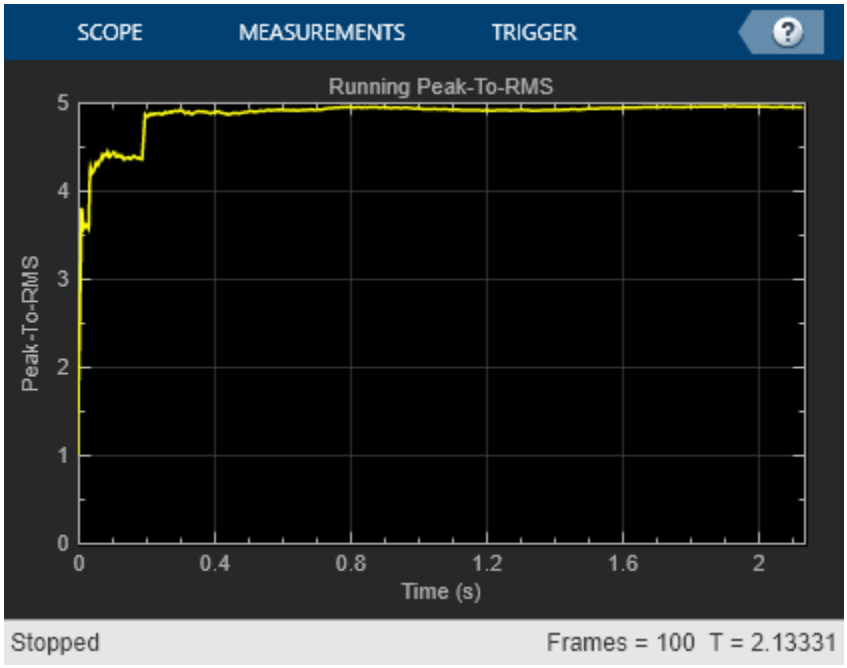
```
for i=1:numFrames
    % Pass white Gaussian noise through FIR Filter
    sig = fir(randn(frameSize,1));

    % Compute power spectrum density
    ps = spect(sig);

    % The runMean System object keeps track of the information about past
    % samples and gives you the mean value reached until now. The same is
    % true for runMin and runMax System objects.
    meanval  = runMean(sig);
    variance = runVar( sig);
    peak2rms = runPeaktoRMS(sig);
    minimum  = runMin( sig);
    maximum  = runMax(sig);

    % Plot the data you have processed
    minmaxScope([sig,minimum,maximum]);
    spectrumScope(10*log10(ps));
    meanScope(meanval);
```

```
    p2rmsScope(peak2rms);
end
release(minmaxScope);
release(spectrumScope);
release(meanScope);
release(p2rmsScope);
```









Conclusion

You have seen visually that the code involves just calling successive System objects with appropriate input arguments and does not involve maintaining any more variables like indices or counters to compute the statistics. This helps in quicker and error free coding. Pre-computation of constant variables inside the objects generally leads to faster processing time.

High Resolution Spectral Analysis in MATLAB

This example shows how to perform high resolution spectral analysis in MATLAB® using an efficient filter bank sometimes referred to as a channelizer. For comparison purposes, a traditional averaged modified periodogram (Welch's) method is also shown. For a similar example in Simulink™, see “High Resolution Spectral Analysis in Simulink” on page 4-33.

Resolution in Spectral Analysis

Resolution in this context refers to the ability to distinguish between two spectral components that lie in the vicinity of each other. Resolution depends on the length of the time-domain segment used to compute the spectrum. When windowing is used on the time-domain segment as is the case with modified periodograms, the type of window used also affects the resolution.

The classical tradeoff with different windows is one of resolution vs. sidelobe attenuation. Rectangular windows provide the highest resolution but very poor (~14 dB) sidelobe attenuation. Poor sidelobe attenuation can result in spectral components being buried by the windowing operation and thus is undesirable. Hann windows provide good sidelobe attenuation at the expense of lower frequency resolution. Parameterizable windows such as Kaiser allow to control the tradeoff by changing the window parameter.

Instead of using averaged modified periodograms (Welch's method), a higher resolution estimate can be achieved by using a filter bank approach that emulates how analog spectrum analyzers work. The main idea is to divide the signal into different frequency bins using a filter bank and computing the average power of each subband signal.

Filter Bank-Based Spectrum Estimation

For this example, 512 different bandpass filters need to be used to get the same resolution afforded by the rectangular window. In order to implement the 512 bandpass filters efficiently, a polyphase analysis filter bank (a.k.a. channelizer) is used. This works by taking a prototype lowpass filter with a bandwidth of Fs/N where N the desired frequency resolution (512 in this example), and implementing the filter in polyphase form much like an FIR decimator is implemented. Instead of adding the results of all the branches as in the decimator case, each branch is used as an input to an N-point FFT. It can be shown that each output of the FFT corresponds a modulated version of a lowpass filter, thus implementing a bandpass filter. The main drawback of the filter bank approach is increased computation due to the polyphase filter as well as slower adaptation to changing signals due to the states of that filter. More details can be found in the book 'Multirate Signal Processing for Communications Systems' by fredric j. harris. Prentice Hall PTR, 2004.

In this example, 100 averages of the spectrum estimate are used throughout. The sampling frequency is set to 1 MHz. It is assumed that we are working with frames of 64 samples which will need to be buffered in order to perform the spectrum estimation.

```
NAvg = 100;
Fs = 1e6;
FrameSize = 64;
NumFreqBins = 512;
filterBankRBW = Fs/NumFreqBins;
```

`dsp.SpectrumAnalyzer` implements a filter bank-based spectrum estimator when `Method` is set accordingly. Internally, it uses `dsp.Channelizer` which implements the polyphase filtering plus FFT (and can be used for other applications besides spectrum analysis, e.g. multicarrier communications).

```
filterBankSA = dsp.SpectrumAnalyzer(...
    'Method','Filter bank',...
    'NumTapsPerBand',24,...
    'SampleRate',Fs,...
    'RBWSource','Property',...
    'RBW',filterBankRBW,...
    'SpectralAverages',NAvg,...
    'PlotAsTwoSidedSpectrum',false,...
    'YLimits',[-150 50],...
    'YLabel','Power',...
    'Title','Filter bank Power Spectrum Estimate',...
    'Position',[50 375 800 450]);
```

Test Signal

In this example, the test signal is acquired in 64-sample frames. For spectral analysis purposes, the larger the frame, the better the resolution.

The test signal consists of two sine waves plus white Gaussian noise. Changing the number of frequency bins, amplitude, frequency, and noise power values is instructive and encouraged.

```
sinegen = dsp.SineWave('SampleRate',Fs,...
    'SamplesPerFrame',FrameSize);
```

Initial Test Case

To start, compute the filter bank spectral estimate for sine waves of amplitude 1 and 2 and frequencies of 200 kHz and 250 kHz, respectively. The white Gaussian noise has an average power (variance) of 1e-12. Note that the onesided noise floor of -114 dBm is accurately shown in the spectral estimate.

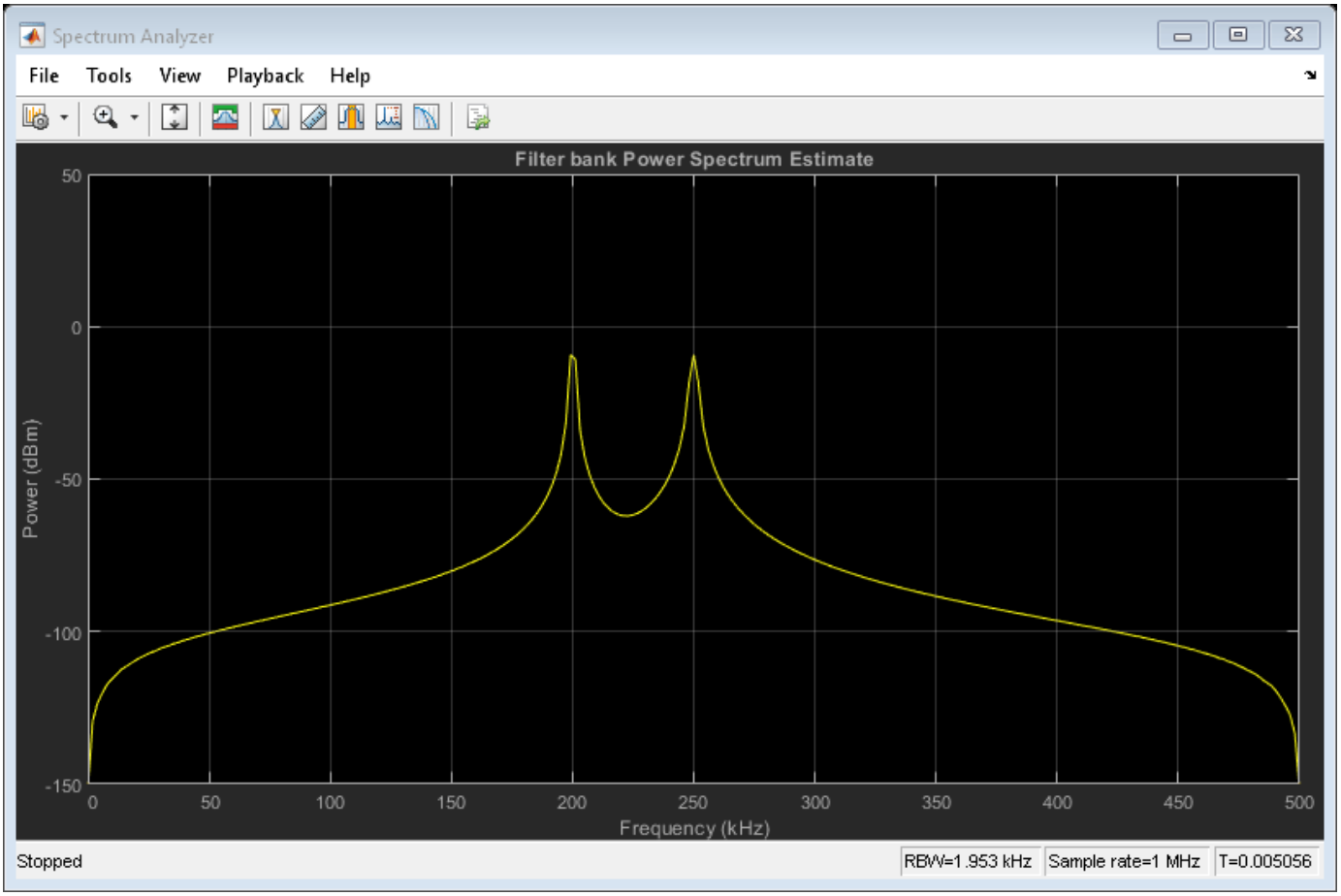
```
release(sinegen)
sinegen.Amplitude = [1 2];
sinegen.Frequency = [200000 250000];

noiseVar = 1e-12;
% -114 dBm onesided
noiseFloor = 10*log10((noiseVar/(NumFreqBins/2))/1e-3);
fprintf('Noise Floor\n');
```

```
Noise Floor
```

```
fprintf('Filter bank noise floor = %.2f dBm\n\n',noiseFloor);
```

```
Filter bank noise floor = -114.08 dBm
```

```
timesteps = 10 * ceil(NumFreqBins / FrameSize);
for t = 1:timesteps
    x = sum(sinegen(),2) + sqrt(noiseVar)*randn(FrameSize,1);
    filterBankSA(x);
end

release(filterBankSA)
```



Numerical Computation Using Spectrum Estimator

`dsp.SpectrumEstimator` can be used to compute the filter bank spectrum estimates.

In order to feed the spectrum estimator a longer frame, a buffer gathers 512 samples before computing the spectral estimate. Although not used in this example, the buffer allows for overlapping which can be used to increase the number of averages obtained from a given set of data.

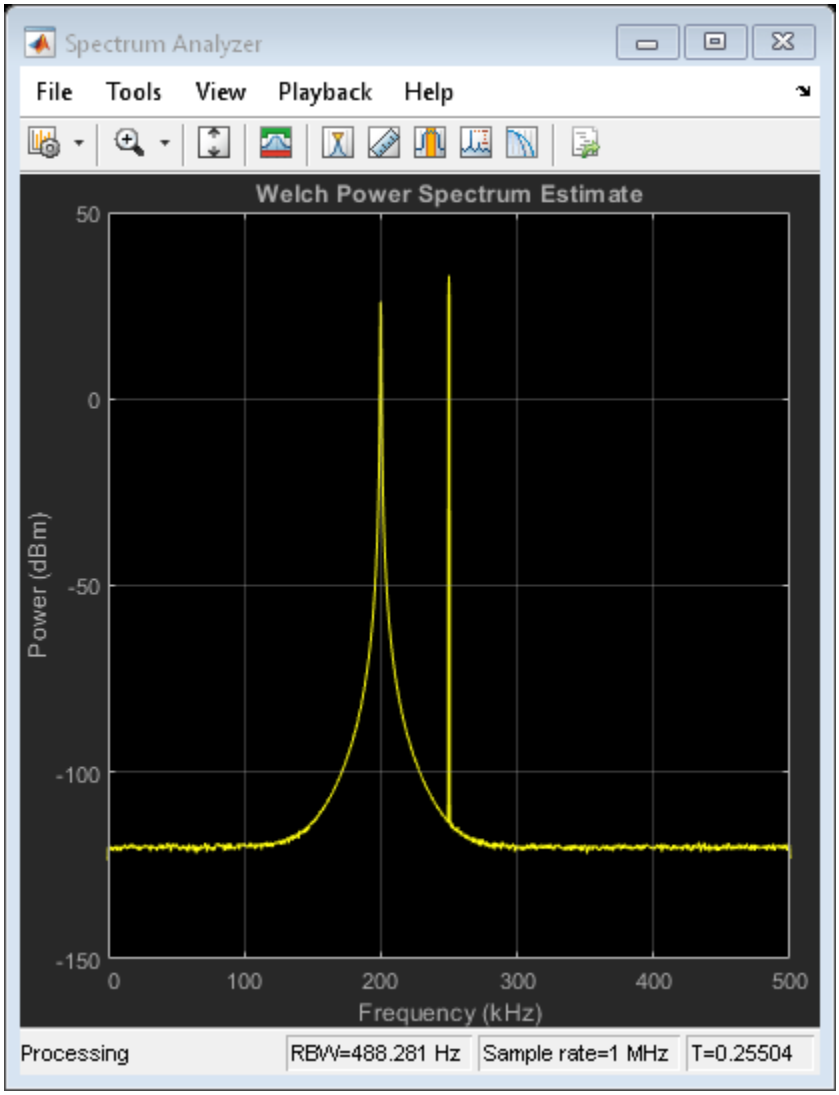
```
filterBankEstimator = dsp.SpectrumEstimator(...
    'Method','Filter bank',...
    'NumTapsPerBand',24,...
    'SampleRate',Fs,...
    'SpectralAverages',NAvg,...
    'FrequencyRange','onesided',...
    'PowerUnits','dBm');

buff    = dsp.AsyncBuffer;

release(sinegen)

timesteps = 10 * ceil(NumFreqBins / FrameSize);
for t = 1:timesteps
    x     = sum(sinegen(),2) + sqrt(noiseVar)*randn(FrameSize,1);
    write(buff,x);      % Buffer data
    if buff.NumUnreadSamples >= NumFreqBins
```

```
        xbuff = read(buff,NumFreqBins);
        Pfbse = filterBankEstimator(xbuff);
    end
end
```

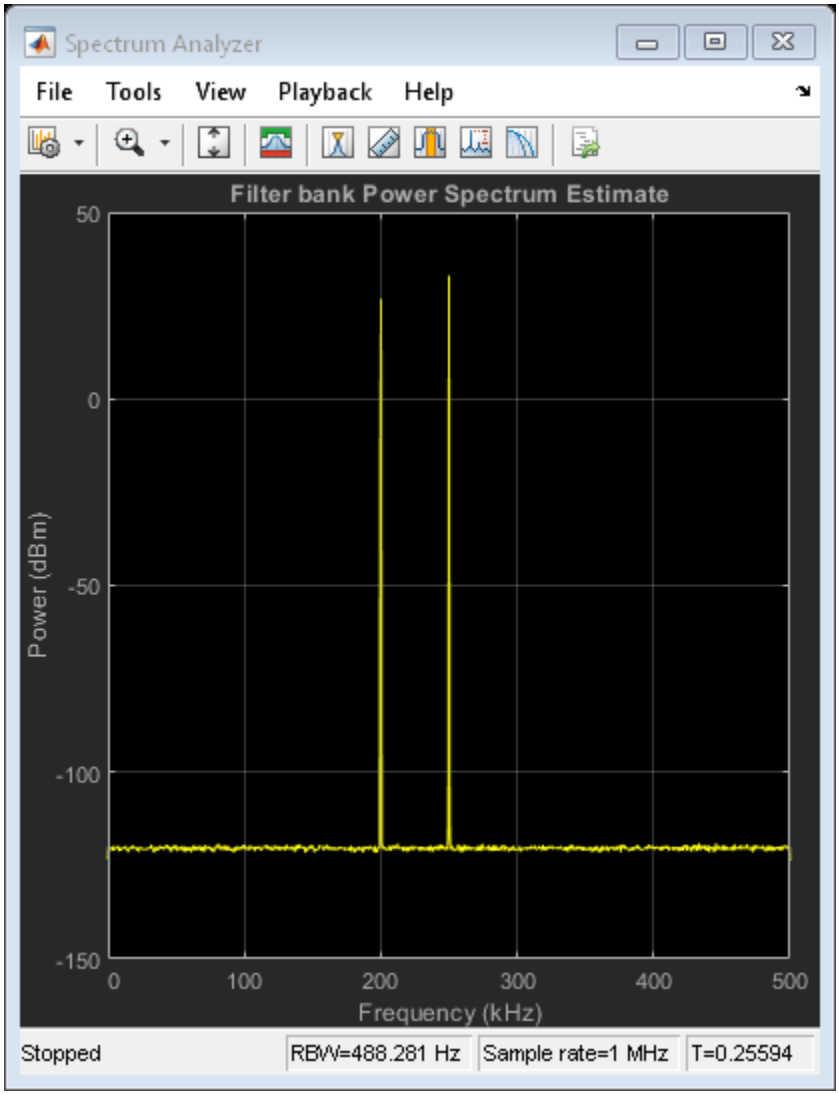
Compare Spectrum Estimates Using Different Methods

Compute the welch and the filter bank spectral estimate for sine waves of amplitude 1 and 2 and frequencies of 200 kHz and 250 kHz, respectively. The white Gaussian noise has an average power (variance) of 1e-12.

```
release(sinegen)
sinegen.Amplitude = [1 2];
sinegen.Frequency = [200000 250000];

filterBankSA.RBWSource = 'Auto';
filterBankSA.Position = [50 375 400 450];

welchSA = dsp.SpectrumAnalyzer(...
    'Method','Welch',...
    'SampleRate',Fs,...
    'SpectralAverages',NAvg,...
    'PlotAsTwoSidedSpectrum',false,...
    'YLimits',[-150 50],...
    'YLabel','Power',...
    'Title','Welch Power Spectrum Estimate',...
    'Position',[450 375 400 450]);

noiseVar = 1e-12;

timesteps = 500 * ceil(NumFreqBins / FrameSize);
for t = 1:timesteps
    x = sum(sinegen(),2) + sqrt(noiseVar)*randn(FrameSize,1);
    filterBankSA(x);
    welchSA(x);
end
```




```
release(filterBankSA)
```




```
RBW = 488.28;
hannNENBW = 1.5;

welchNSamplesPerUpdate = Fs*hannNENBW/RBW;
filterBankNSamplesPerUpdate = Fs/RBW;

fprintf('Samples/Update\n');
```

```
Samples/Update
```

```
fprintf('Welch Samples/Update = %.3f Samples\n',...
    welchNSamplesPerUpdate);
```

```
Welch Samples/Update = 3072.008 Samples
```

```
fprintf('Filter bank Samples/Update = %.3f Samples\n\n',...
    filterBankNSamplesPerUpdate);
```

```
Filter bank Samples/Update = 2048.005 Samples
```

```
welchNoiseFloor = 10*log10((noiseVar/(welchNSamplesPerUpdate/2))/1e-3);
filterBankNoiseFloor = 10*log10((noiseVar/(filterBankNSamplesPerUpdate/2))/1e-3);

fprintf('Noise Floor\n');
```

```
Noise Floor
```

```
fprintf('Welch noise floor       = %.2f dBm\n',welchNoiseFloor);
```

```
Welch noise floor       = -121.86 dBm
```

```
fprintf('Filter bank noise floor = %.2f dBm\n\n',filterBankNoiseFloor);
```

```
Filter bank noise floor = -120.10 dBm
```

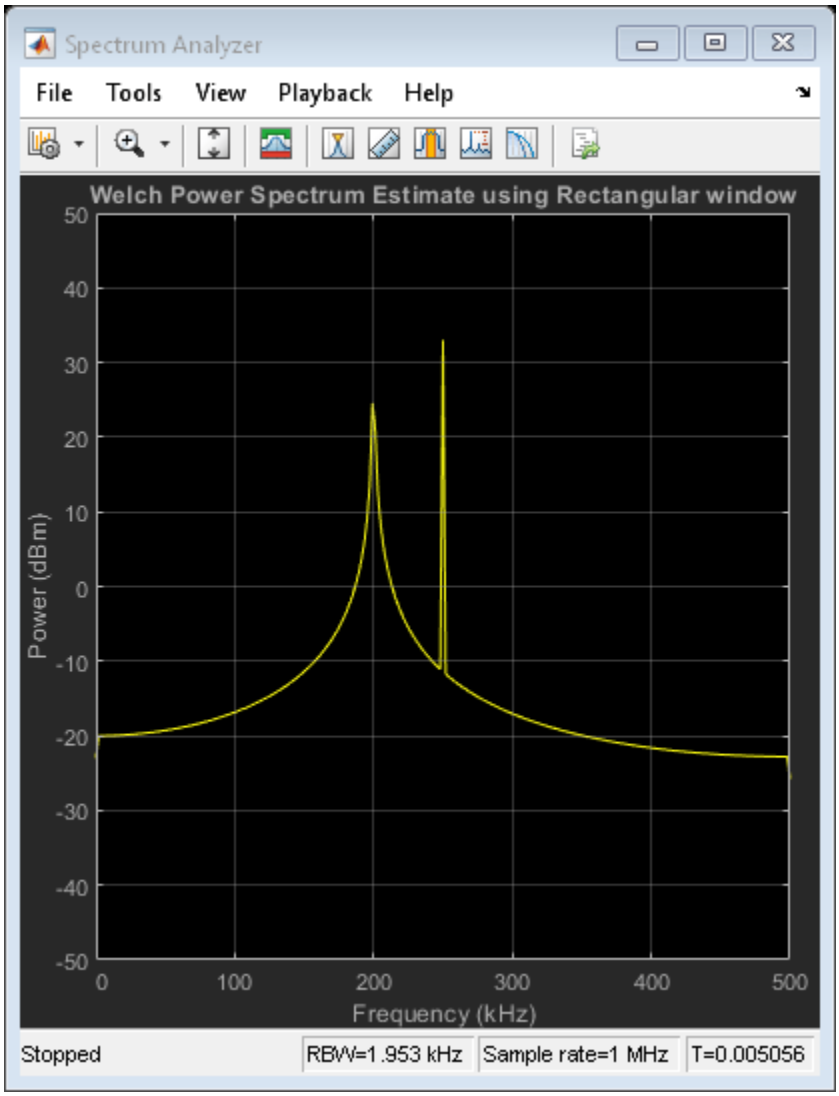
Both Welch and Filter bank-based spectrum estimate detected the two tones at 200 kHz and 250 kHz. Filter bank-based spectrum estimate has better isolation of tones. For the same Resolution Bandwidth (RBW), averaged modified periodogram (Welch's) method requires 3073 samples to compute the spectrum compared to 2048 required by filter bank-based estimate. Note that the onesided noise floor of -120 dBm is accurately shown in the filter bank spectral estimate.

Compare Modified Periodograms Using Different Windows

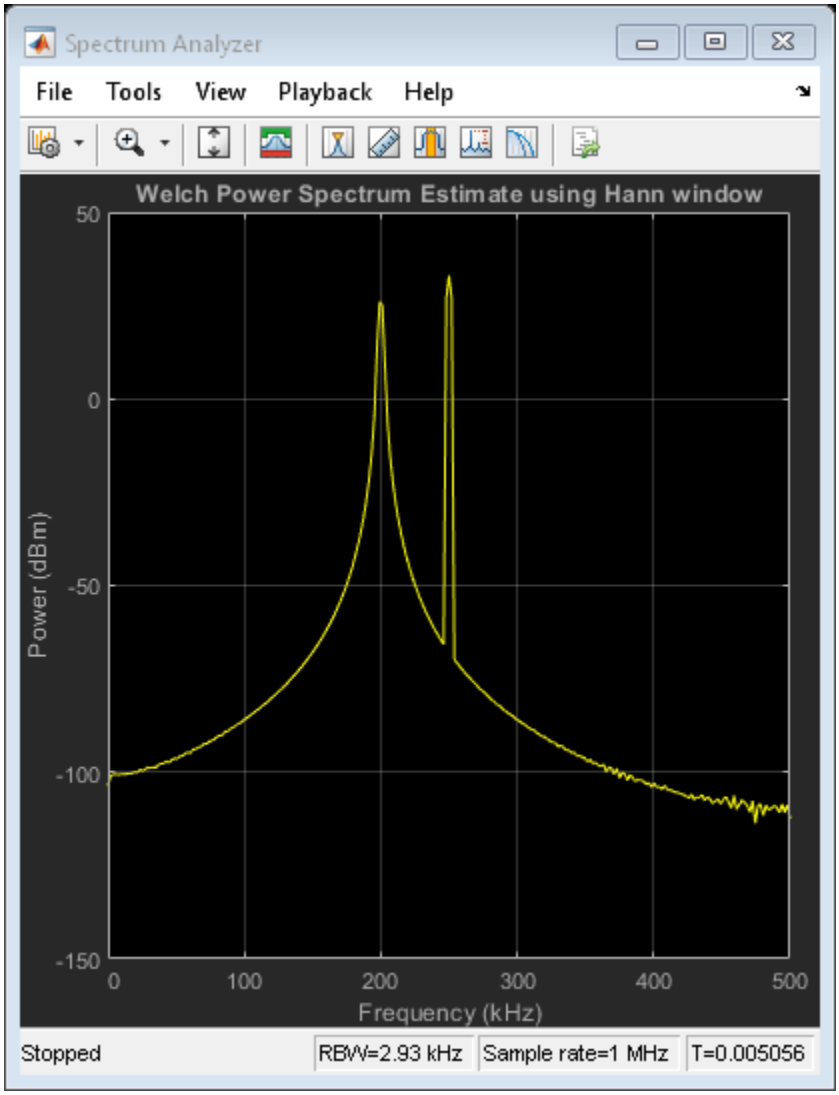
Consider two spectrum analyzers in which the only difference is the window used: rectangular or Hann.

```
rectRBW = Fs/NumFreqBins;
hannNENBW = 1.5;
hannRBW = Fs*hannNENBW/NumFreqBins;

rectangularSA = dsp.SpectrumAnalyzer(...
    'SampleRate',Fs,...
    'Window','Rectangular',...
    'RBWSource','Property',...
    'RBW',rectRBW,...
    'SpectralAverages',NAvg,...
    'PlotAsTwoSidedSpectrum',false,...
    'YLimits',[-50 50],...
    'YLabel','Power',...
    'Title','Welch Power Spectrum Estimate using Rectangular window',...
    'Position',[50 375 400 450]);

hannSA = dsp.SpectrumAnalyzer(...
    'SampleRate',Fs,...
    'Window','Hann',...
    'RBWSource','Property',...
    'RBW',hannRBW,...
    'SpectralAverages',NAvg,...
    'PlotAsTwoSidedSpectrum',false,...
    'YLimits',[-150 50],...
    'YLabel','Power',...
    'Title','Welch Power Spectrum Estimate using Hann window',...
    'Position',[450 375 400 450]);

release(sinegen)
sinegen.Amplitude = [1 2]; % Try [0 2] as well
sinegen.Frequency = [200000 250000];
```

```
noiseVar = 1e-12;
timesteps = 10 * ceil(NumFreqBins / FrameSize);
for t = 1:timesteps
    x = sum(sinegen(),2) + sqrt(noiseVar)*randn(FrameSize,1);
    rectangularSA(x);
    hannSA(x);
end
release(rectangularSA)
```




```
release(hannSA)
```



The rectangular window provides a narrow mainlobe at the expense of low sidelobe attenuation. In contrast, the Hann window provides a broader mainlobe in exchange for the much larger sidelobe attenuation. The broader mainlobe is particularly noticeable at 250 kHz. Both windows exhibit large rolloffs around the frequencies at which the sine waves lie. This can mask low power signals of interest above the noise floor. That problem is virtually non-existent in the filter bank case.

Changing the amplitudes to [0 2] rather than [1 2] effectively means there is a single sine wave of 250 kHz along with noise. This case is interesting because the rectangular window behaves particularly well when the 200 kHz sine wave is not interfering. The reason is that 250 kHz is one of the 512 frequencies that divide 1 MHz evenly. In that case, the time domain replicas introduced by the frequency sampling inherent in the FFT make a perfect periodic extension of the time-limited data segment used for the power spectrum computation. In general, for sine waves with arbitrary frequencies, this is not the case. This dependence on the frequency of the sine wave along with the susceptibility to signal interference is another drawback of the modified periodogram approach.

Resolution Bandwidth (RBW)

The resolution bandwidth for each analyzer can be computed once the input length is known. The RBW indicates the bandwidth over which the power component is computed. That is to say, each element of the power spectrum estimate represents the power in Watts, dBW, or dBm across a bandwidth of width RBW centered around the frequency corresponding to the element of the estimate. The power value of each element in the power spectrum estimate is found by integrating the power-density over the frequency band spanned by the RBW value. A lower RBW indicates a higher resolution since the power is computed over a finer grid (a smaller bandwidth). Rectangular windows have the highest resolution of all windows. In the case of the Kaiser window, the RBW depends on the sidelobe attenuation used.

```
fprintf('RBW\n')
```

```
RBW
```

```
fprintf('Welch-Rectangular  RBW = %.3f Hz\n',rectRBW);
```

```
Welch-Rectangular  RBW = 1953.125 Hz
```

```
fprintf('Welch-Hann         RBW = %.3f Hz\n',hannRBW);
```

```
Welch-Hann         RBW = 2929.688 Hz
```

```
fprintf('Filter bank        RBW = %.3f Hz\n\n',filterBankRBW);
```

```
Filter bank        RBW = 1953.125 Hz
```

In the case of setting the amplitudes to [0 2], i.e. the case in which there is a single sine wave at 250 kHz, it is interesting to understand the relation between the RBW and the noise floor. The expected noise floor is 10*log10((noiseVar/(NumFreqBins/2))/1e-3) or about -114 dBm. The spectral estimate corresponding to the rectangular window has the expected noise floor, but the spectral estimate using the Hann window has a noise floor that is about 2 dBm higher than expected. The reason for this is that the spectral estimate is compute at 512 frequency points but the power spectrum is integrated over the RBW of the particular window. For the rectangular window, the RBW is exactly 1 MHz/512 so that the spectral estimate contains independent estimates of the power for each frequency bin. For the Hann window, the RBW is larger, so that the spectral estimate contains overlapping power from one frequency bin to the next. This overlapping power increases the power value in each bin and elevates the noise floor. The amount can be computed analytically as follows:

```
hannNoiseFloor = 10*log10((noiseVar/(NumFreqBins/2)*hannRBW/rectRBW)/1e-3);
fprintf('Noise Floor\n');
```

```
Noise Floor
```

```
fprintf('Hann noise floor = %.2f dBm\n\n',hannNoiseFloor);
```

```
Hann noise floor = -112.32 dBm
```

Sinusoids in Close Proximity to Each Other

To illustrate the resolution issue consider the following case. The sinusoid frequencies are changed to 200 kHz and 205 kHz. The filter bank estimate remains accurate. Focusing on the window-based estimators, because of the broader mainlobe in the Hann window, the two sinusoids are harder to distinguish when compared to the rectangular window estimate. The fact is that neither of the two estimates is particularly accurate. Note that 205 kHz is basically at the limit of what we can distinguish from 200 kHz. For closer frequencies, all three estimators will fail to separate the two

spectral components. The only way to separate closer components is to have larger frame sizes and therefore a larger number of `NumFrequencyBands` in the case of the filter bank estimator.

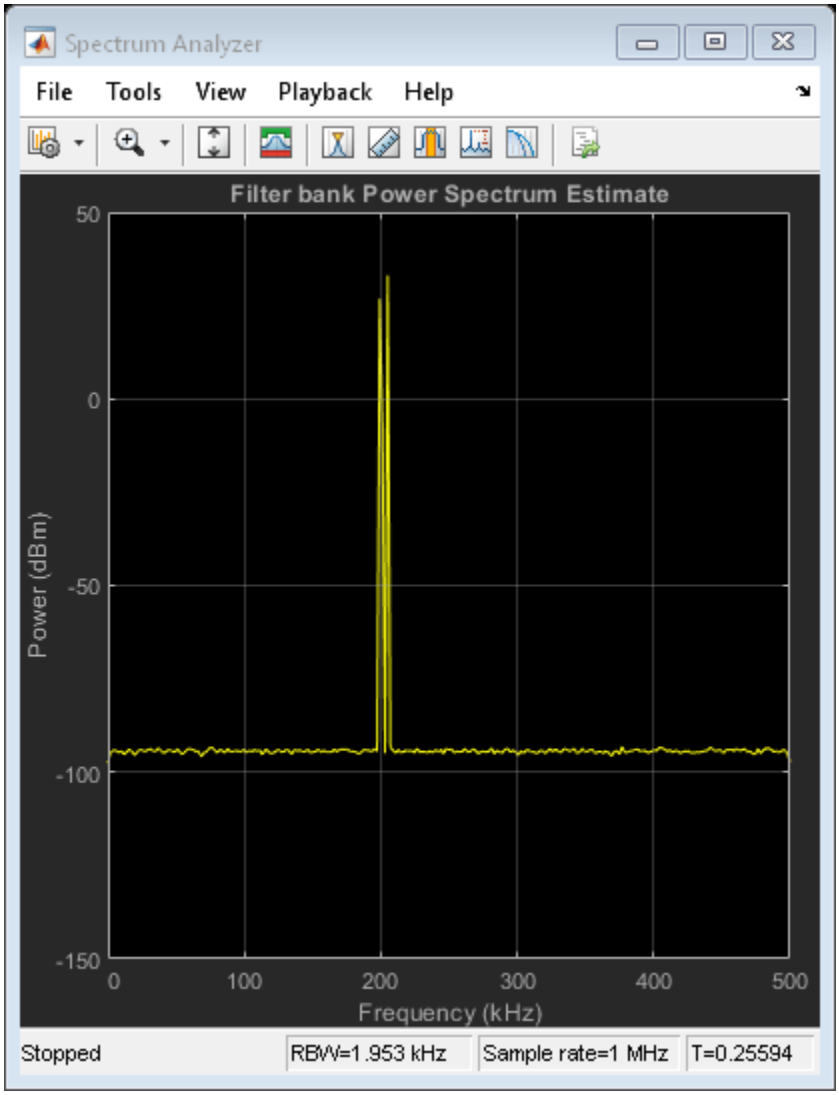
```
release(sinegen)
sinegen.Amplitude = [1 2];
sinegen.Frequency = [200000 205000];

filterBankSA.RBWSource = 'Property';
filterBankSA.RBW       =  filterBankRBW;
filterBankSA.Position = [850 375 400 450];

noiseVar = 1e-10;
noiseFloor = 10*log10((noiseVar/(NumFreqBins/2))/1e-3); % -94 dBm onesided
fprintf('Noise Floor\n');
```

```
Noise Floor
```

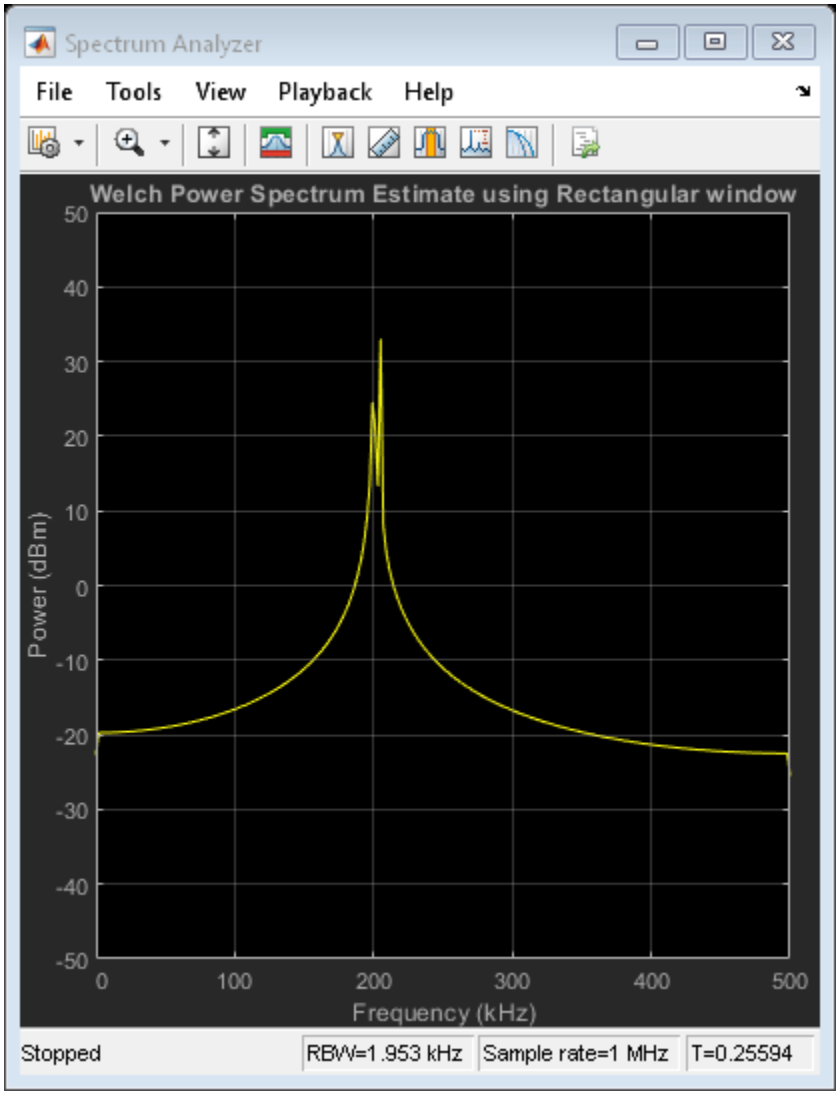
```
fprintf('Noise floor = %.2f dBm\n\n',noiseFloor);
```

```
Noise floor = -94.08 dBm
```

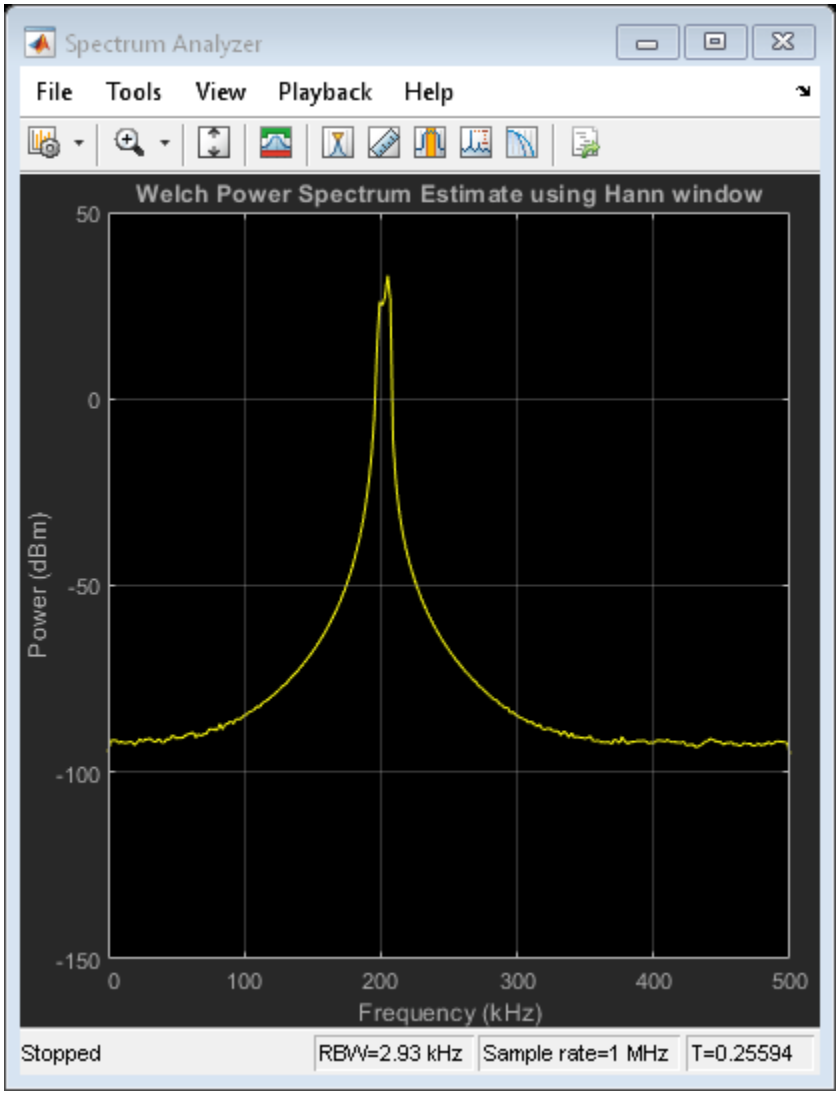
```
timesteps = 500 * ceil(NumFreqBins / FrameSize);
for t = 1:timesteps
    x  = sum(sinegen(),2) + sqrt(noiseVar)*randn(FrameSize,1);
    filterBankSA(x);
    rectangularSA(x);
    hannSA(x);
end

release(filterBankSA)
```




```
release(rectangularSA)
```




```
release(hannSA)
```



Detecting Low Power Sinusoidal Components

Next, re-run the previous scenario but add a third sinusoid at 170 kHz with a very small amplitude. This third sinusoid is missed completely by the rectangular window estimate and the Hann window estimate. The filter bank estimate provides both better resolution and better isolation of tones, so that the three sine waves are clearly visible.

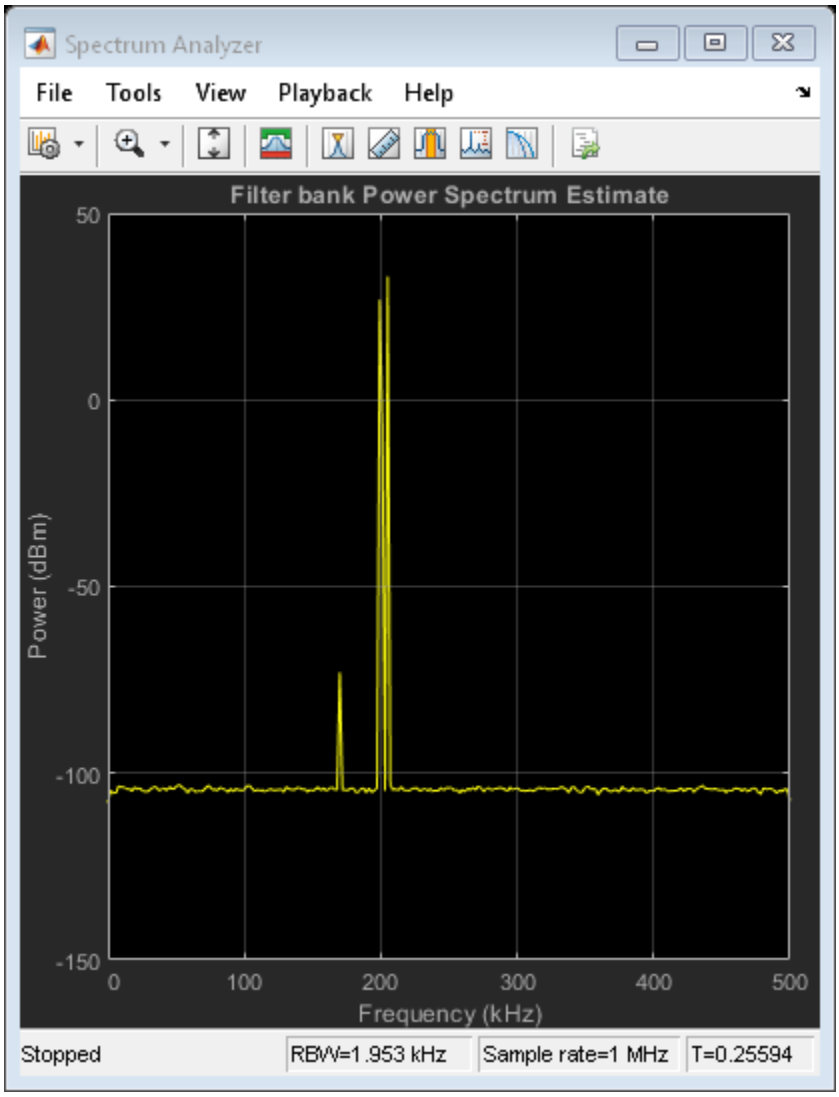
```
release(sinegen)
sinegen.Amplitude = [1e-5 1 2];
sinegen.Frequency = [170000 200000 205000];

noiseVar = 1e-11;
noiseFloor = 10*log10((noiseVar/(NumFreqBins/2))/1e-3); % -104 dBm onesided
fprintf('Noise Floor\n');
```

```
Noise Floor
```

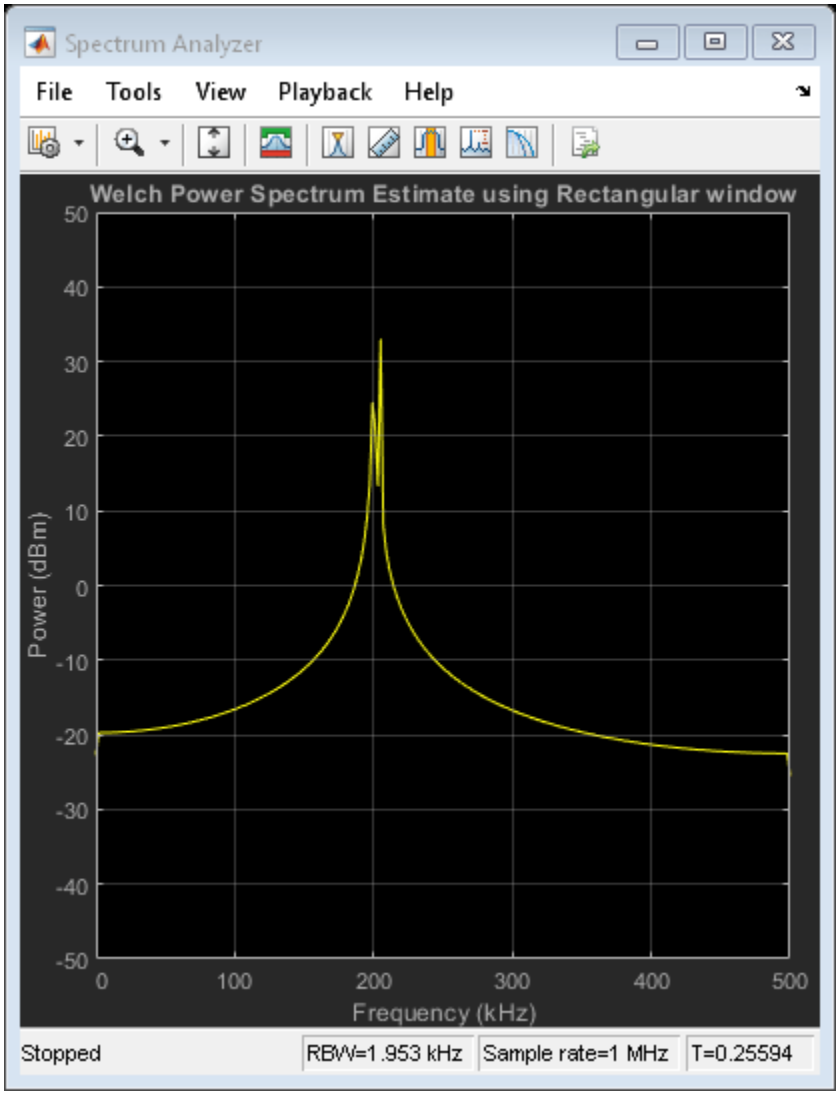
```
fprintf('Noise floor = %.2f dBm\n\n',noiseFloor);
```

```
Noise floor = -104.08 dBm
```

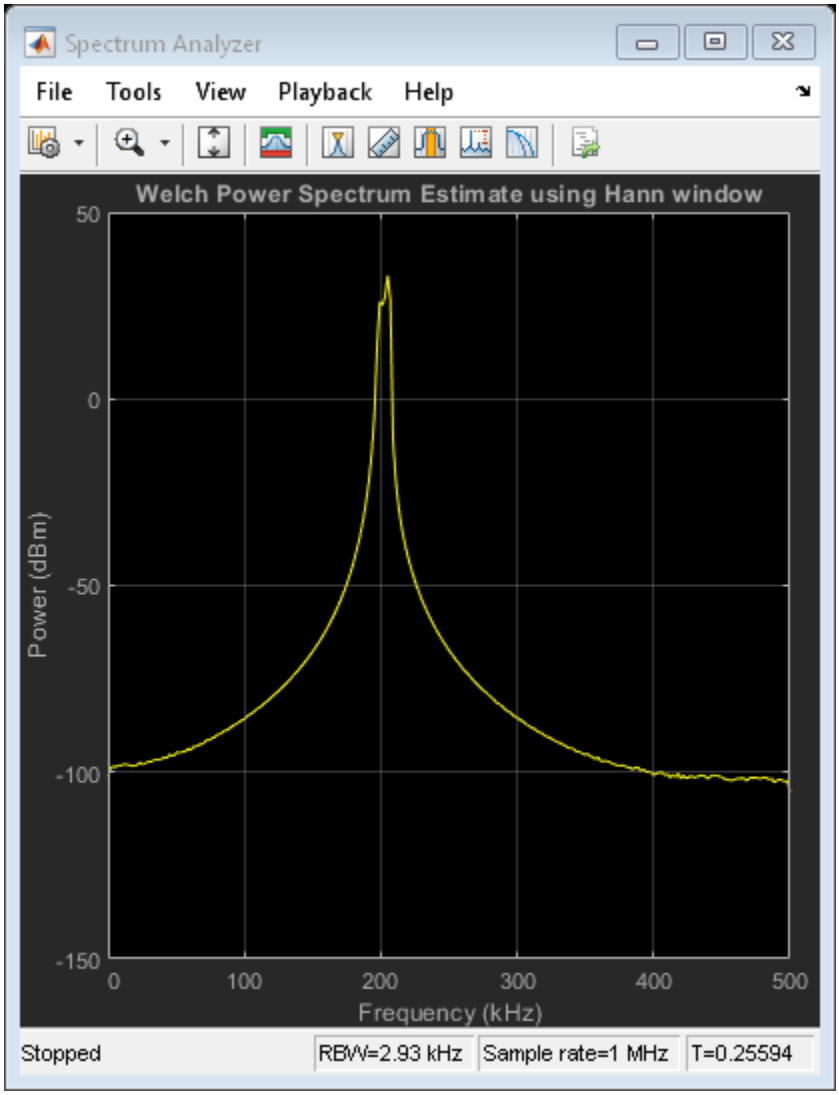
```
timesteps = 500 * ceil(NumFreqBins / FrameSize);
for t = 1:timesteps
    x  = sum(sinegen(),2) + sqrt(noiseVar)*randn(FrameSize,1);
    filterBankSA(x);
    rectangularSA(x);
    hannSA(x);
end

release(filterBankSA)
```




```
release(rectangularSA)
```




```
release(hannSA)
```



See Also

Related Examples

- “High Resolution Spectral Analysis in Simulink” on page 4-33
- “High Resolution Filter-Bank-Based Power Spectrum Estimation” on page 17-7
- “Streaming Power Spectrum Estimation Using Welch's Method” on page 18-65

External Websites

- Filter Bank Method: A Better Approach to Spectral Analysis

High Resolution Spectral Analysis in Simulink

This example shows how to perform high resolution spectral analysis in Simulink™ using the Spectrum Analyzer block and the Spectrum Estimator block.

For comparison purposes, a traditional averaged modified periodogram (Welch's) method is also shown. For a similar example in MATLAB®, see “High Resolution Spectral Analysis in MATLAB” on page 4-16.

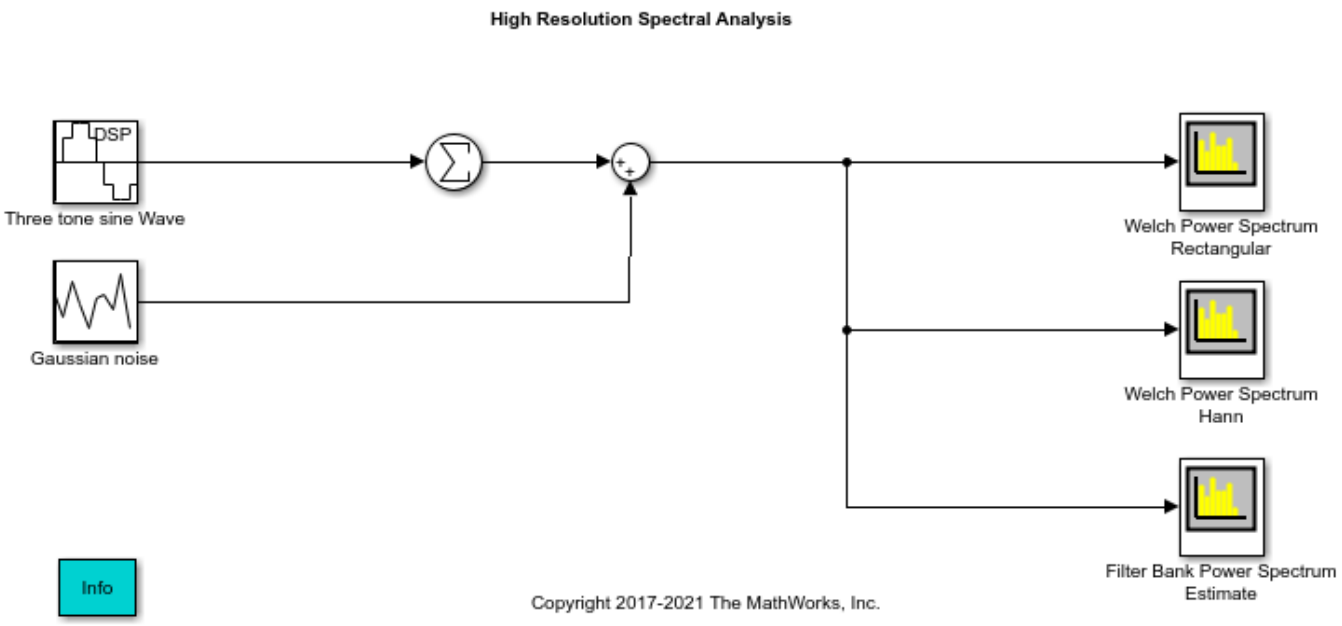
Using Spectrum Analyzer

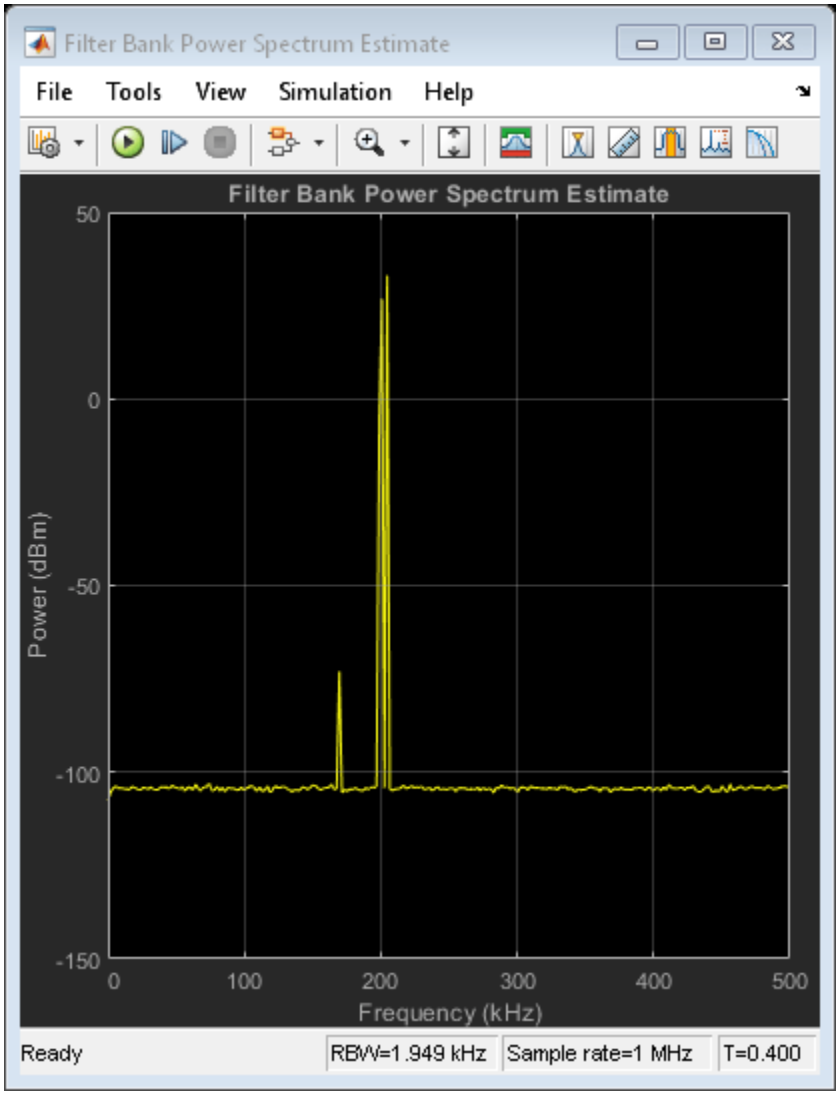
The `SpectrumAnalyzerFilterBank` model illustrates the high resolution capabilities of filter bank-based spectral estimation compared to the Welch's method. The filter bank-based spectral estimation has lower noise floor.

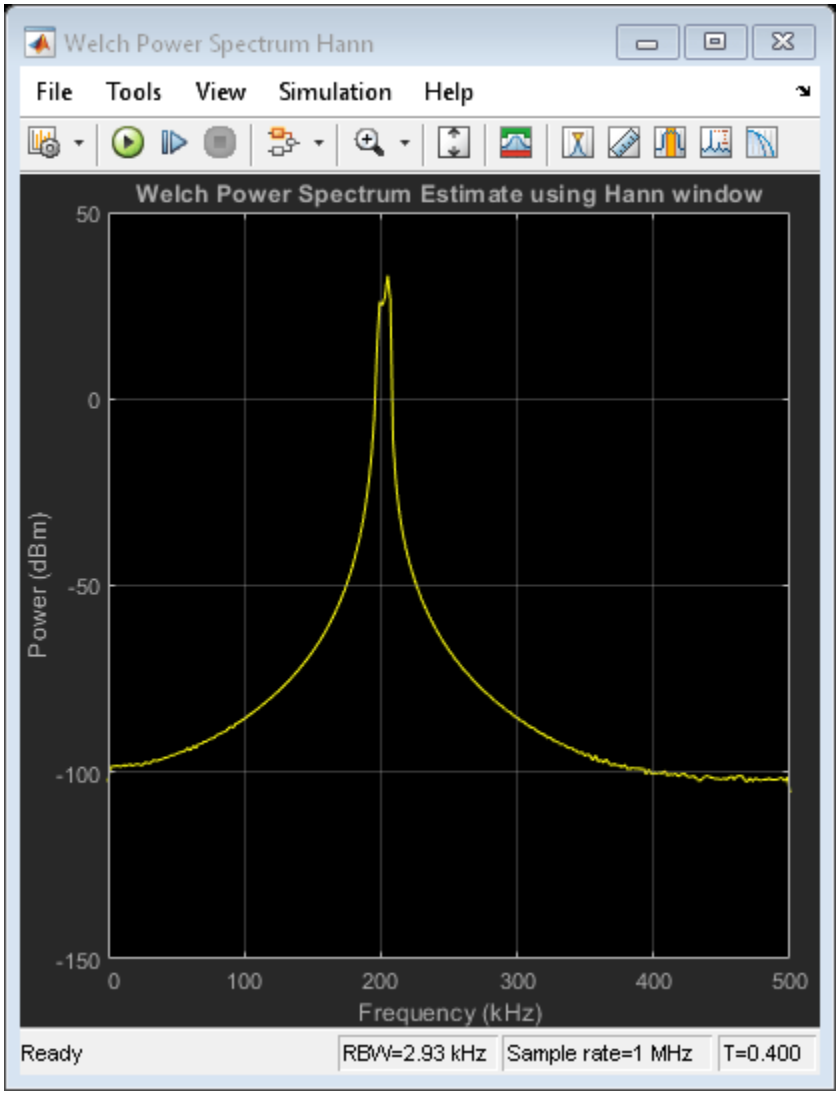
Consider the following case. Three sinusoids at 170 kHz, 200 kHz and 205kHz with the amplitudes [1e-5 1 2]. The first sinusoid is completely missed by the rectangular window estimate. The filter bank estimate provides better resolution and better isolation of three tones.

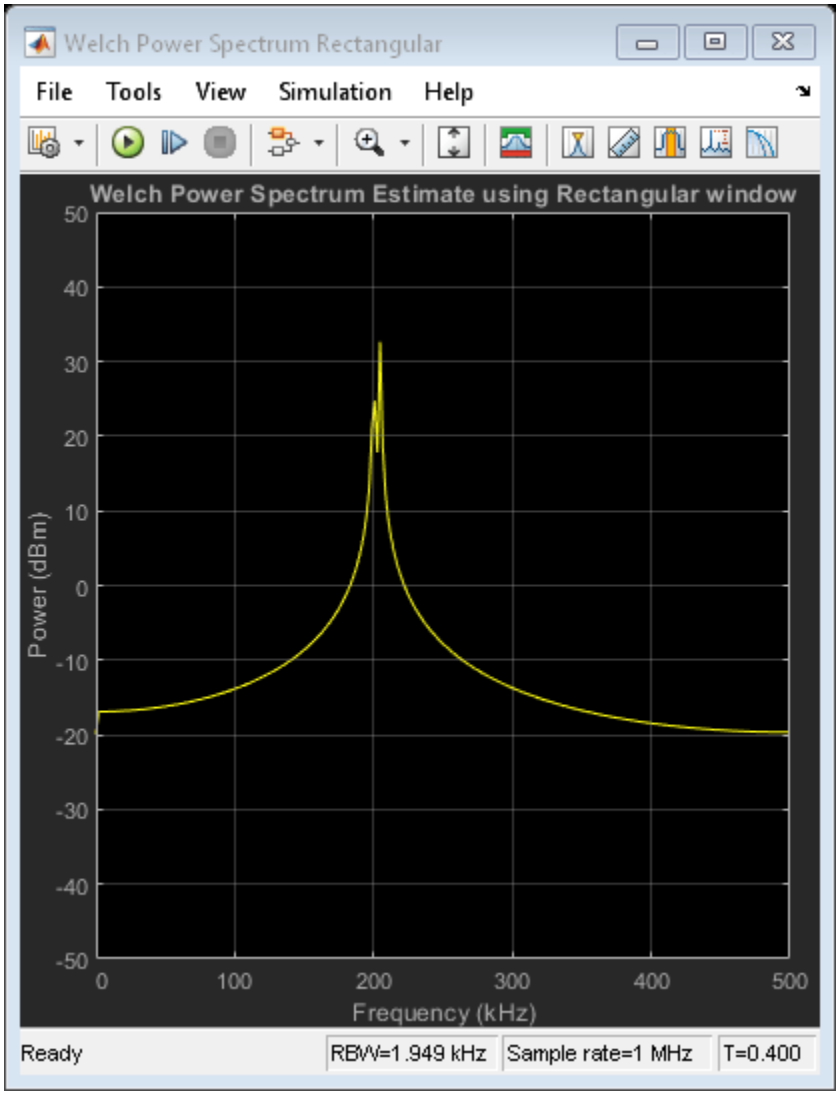
Open and simulate the `SpectrumAnalyzerFilterBank` model.

```
open_system('SpectrumAnalyzerFilterBank');
sim('SpectrumAnalyzerFilterBank');
```




Filter Bank Power Spectrum Estimate
File
Tools
View
Simulation
Help
Filter Bank Power Spectrum Estimate
50
0
-50
-100
-150
Power (dBm)
0
100
200
300
400
500
Frequency (kHz)
Ready
RBW=1.949 kHz
Sample rate=1 MHz
T=0.400


Welch Power Spectrum Hann
File
Tools
View
Simulation
Help
Welch Power Spectrum Estimate using Hann window
50
0
-50
-100
-150
Power (dBm)
0
100
200
300
400
500
Frequency (kHz)
Ready
RBW=2.93 kHz
Sample rate=1 MHz
T=0.400



Close the model.

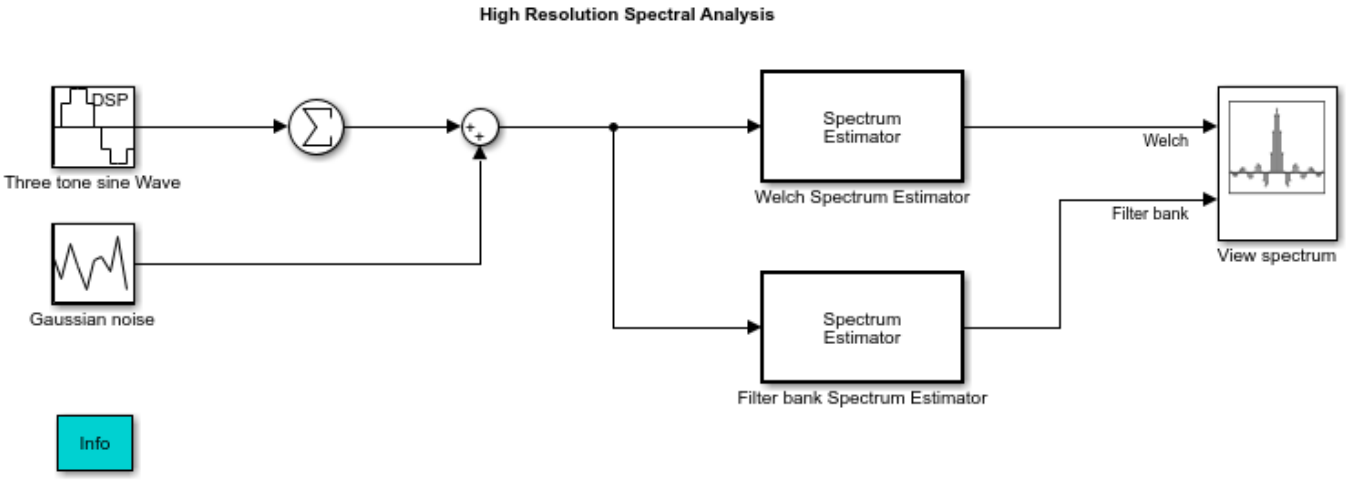
```
bdclose('SpectrumAnalyzerFilterBank');
```

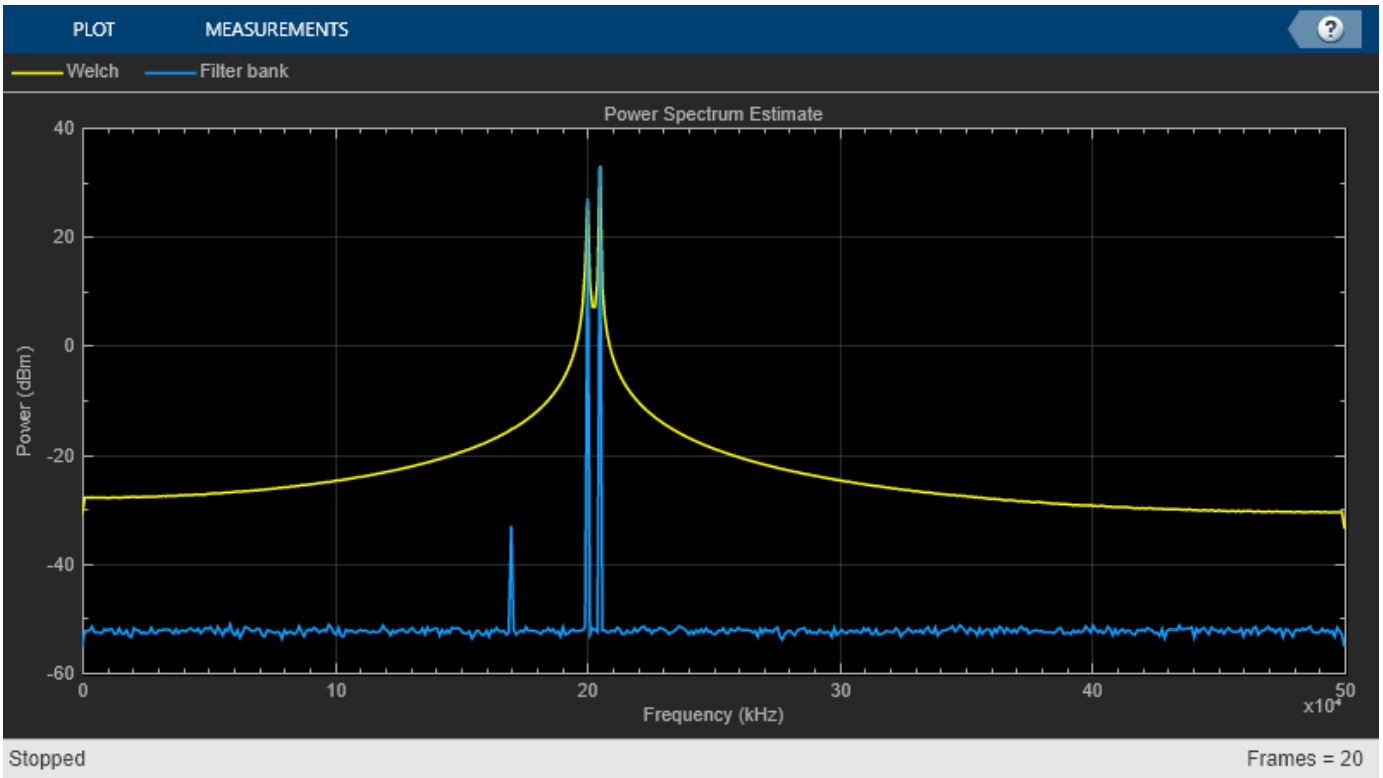
Using Spectrum Estimator

Numerical computations for high resolution spectral estimation shown above can also be modeled in Simulink using the Spectrum Estimator block. The `SpectrumEstimatorFilterBank` model illustrates the high resolution capabilities of filter bank-based spectrum estimation and lower noise floor compared to the Welch's method, using Simulink. Array plot is used to visualize the results. Array plot provides a convenient way of plotting the spectrum estimates. Values are shown in dBm, but Watts or dBW could easily be used instead.

Open and simulate the `SpectrumEstimatorFilterBank` model.

```
open_system('SpectrumEstimatorFilterBank');
sim('SpectrumEstimatorFilterBank');
```





Close the model.

```
bdclose('SpectrumEstimatorFilterBank');
```

See Also

Related Examples

- “High Resolution Spectral Analysis in MATLAB” on page 4-16
- “High Resolution Filter-Bank-Based Power Spectrum Estimation” on page 17-7
- “Streaming Power Spectrum Estimation Using Welch's Method” on page 18-65

External Websites

- Filter Bank Method: A Better Approach to Spectral Analysis

Zoom FFT

This example showcases zoom FFT, which is a signal processing technique used to analyze a portion of a spectrum at high resolution. DSP System Toolbox™ offers this functionality in MATLAB™ through the `dsp.ZoomFFT` System object, and in Simulink through the zoom FFT library block.

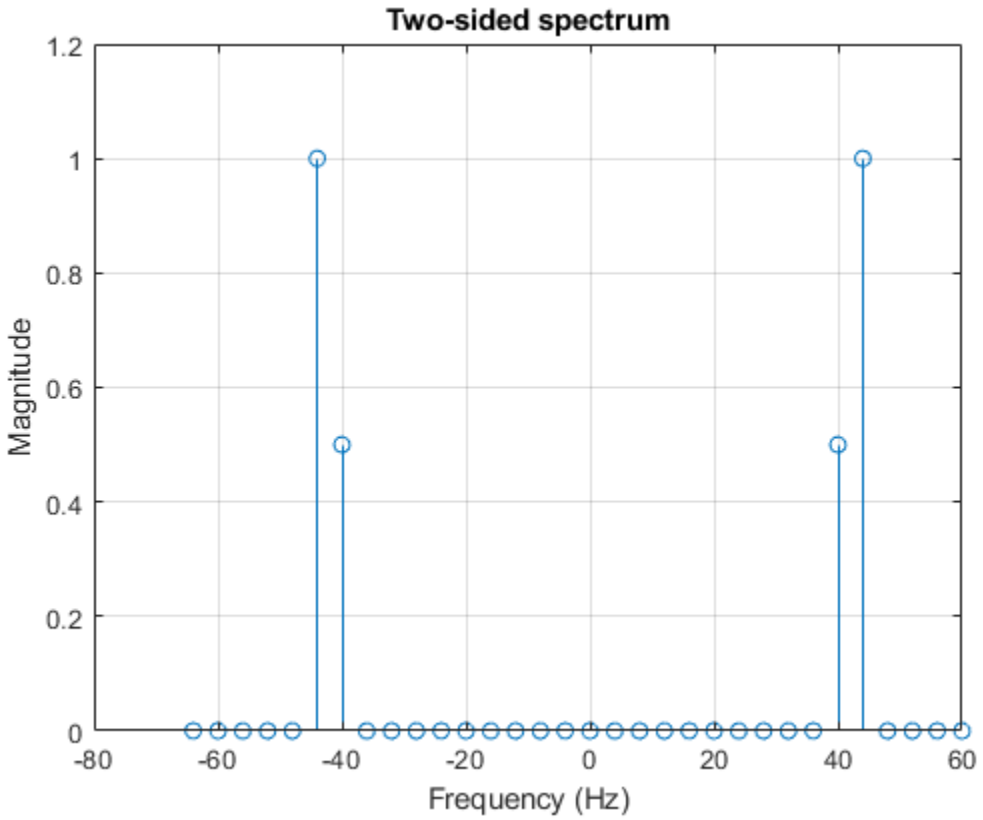
Regular FFT

A signal's resolution is bounded by its length. We will illustrate this fact by a simple example. Consider a signal formed by the sum of two sine waves:

```
L  = 32;        % Frame size
Fs = 128;       % Sample rate
res = Fs/L;     % Frequency resolution
f1 = 40;        % First sine wave frequency
f2 = f1 + res;  % Second sine wave frequency

sn1 = dsp.SineWave('Frequency',f1,'SampleRate',Fs,'SamplesPerFrame',L);
sn2 = dsp.SineWave('Frequency',f2,'SampleRate',Fs,'SamplesPerFrame',L,...
                   'Amplitude',2);

x = sn1() + sn2();
```

Let's compute the FFT of x and plot the magnitude of the FFT. Note that the two sine waves are in adjacent samples of the FFT. This means that you cannot discriminate between frequencies closer than Fs/L.

```
X = fft(x);
stem(Fs/L*(0:length(X)-1)-Fs/2,abs(fftshift(X))/L)
grid on;
xlabel('Frequency (Hz)')
ylabel('Magnitude')
title('Two-sided spectrum')
```



Zoom FFT

Suppose you have an application for which you are only interested in a sub-band of the Nyquist interval. The idea behind zoom FFT is to 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. The shorter signal comes from decimating the original signal. The savings come from being able to compute a much shorter FFT while achieving the same resolution. This is intuitive: for a decimation factor of D, the new sampling rate is Fsd = Fs/D, and the new frame size (and FFT length) is Ld = L/D, so the resolution of the decimated signal is Fsd/Ld = Fs/L.

DSP System Toolbox offers zoom FFT functionality for MATLAB and Simulink, through the dsp.ZoomFFT System object and the zoom FFT library block, respectively. The next sections will discuss the zoom FFT algorithm in more detail.

The Mixer Approach

Before discussing the algorithm used in dsp.FFT, we present the mixer approach, which is a popular zoom FFT method.

For the example here, assume you are only interested in the interval [16 Hz, 48 Hz].

```
BWOfInterest = 48 - 16;
Fc           = (16 + 48)/2; % Center frequency of bandwidth of interest
```

The achievable decimation factor is:

```
BWFactor = floor(Fs/BWOfInterest);
```

The mixer approach consists of first shifting the band of interest down to DC using a mixer, and then performing lowpass filtering and decimation by a factor of BWFactor (using an efficient polyphase FIR decimation structure).

Let's design the decimation filter's coefficients using the function `designMultirateFIR`:

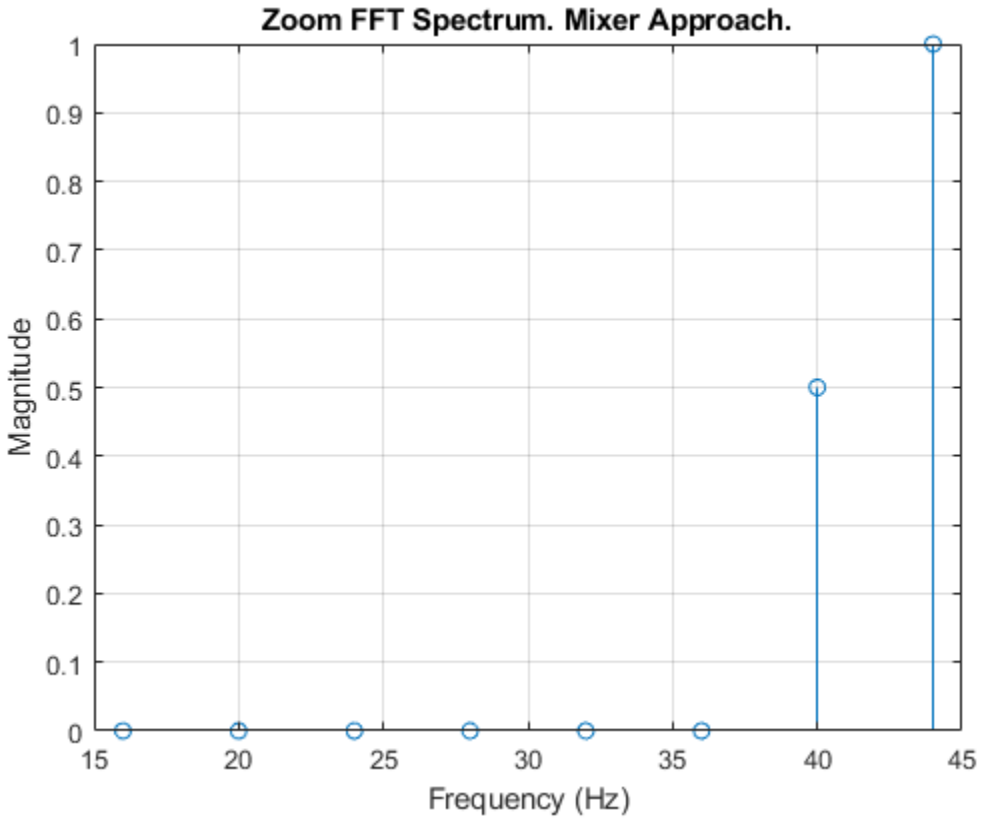
```
B = designMultirateFIR(1,BWFactor);
D = dsp.FIRDecimator('DecimationFactor',BWFactor,'Numerator',B);
```

Now, let's mix the signal down to DC, and filter it through the FIR decimator:

```
for k = 1:10
    % Run a few times to eliminate transient in filter
    x = sn1()+sn2();
    % Mix down to DC
    indVect = (0:numel(x)-1).' + (k-1) * size(x,1);
    y = x .* exp(-2*pi*indVect*Fc*1j/Fs);
    % Filter through FIR decimator
    xd = D(y);
end
```

Let's now take the FFT of the filtered signal (note that the FFT length is reduced by BWFactor, or the decimation length, compared to regular FFT, while maintaining the same resolution):

```
fftlen = numel(xd);
Xd = fft(xd);
figure
Ld  = L/BWFactor;
Fsd = Fs/BWFactor;
F   = Fc + Fsd/fftlen*(0:fftlen-1)-Fsd/2;
stem(F,abs(fftshift(Xd))/Ld)
grid on
xlabel('Frequency (Hz)')
ylabel('Magnitude')
title('Zoom FFT Spectrum. Mixer Approach.')
```



The complex-valued mixer adds an extra multiplication for each high-rate sample, which is not efficient. the next section presents an alternative, more efficient, zoom FFT approach.

Bandpass Sampling

An alternative zoom FFT method takes advantage of a known result from bandpass filtering (also sometimes called under-sampling): Assume we are interested in the band [F1,F2] of a signal with sampling rate Fs Hz. If we pass the signal through a complex (one-sided) bandpass filter centered at Fc = (F1+F2)/2 and with bandwidth BW = F2 - F1, and then downsample it by a factor of D = floor(Fs/BW), we will bring down the desired band to baseband.

In general, if Fc cannot be expressed in the form k*Fs/D (where K is an integer), then the shifted, decimated spectrum will not be centered at DC. In fact, the center frequency Fc will be translated to [2]:

Fd = Fc - (Fs/D)*floor((D*Fc + Fs/2)/Fs)

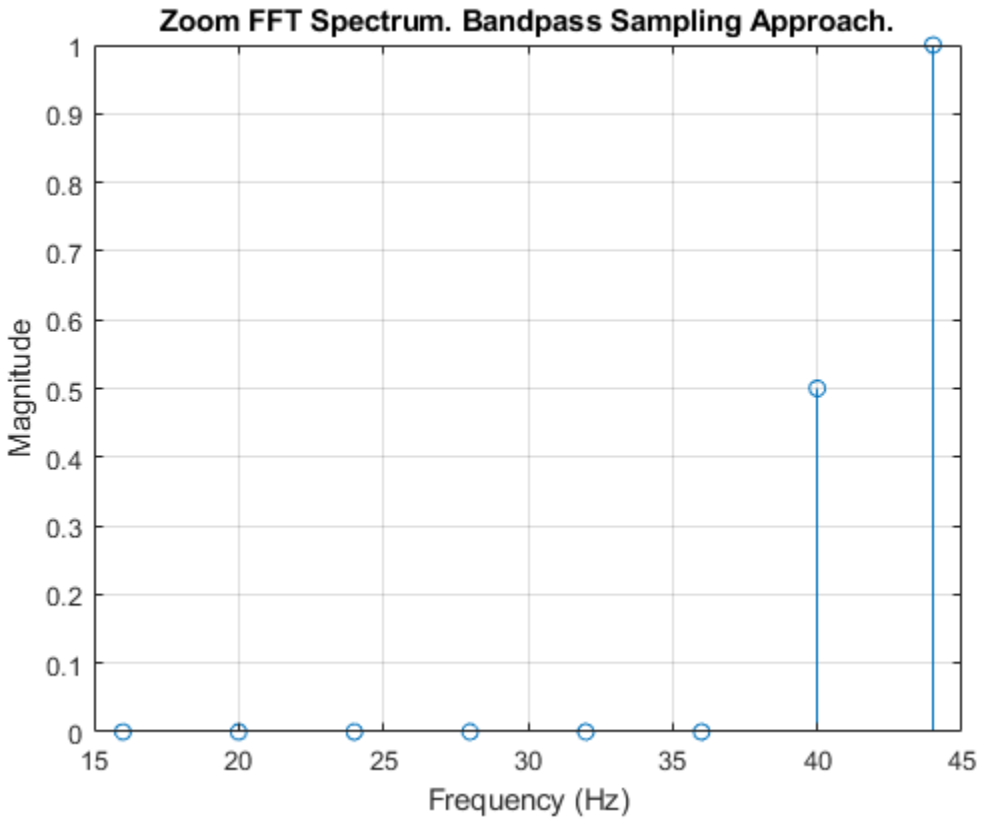
In this case, we can use a mixer (running at the low sample rate of the decimated signal) to center the desired band to zero Hertz.

Using the example from the previous section, we obtain the coefficients of the complex bandpass filter by modulating the coefficients of the designed lowpass filter:

```
Bbp     = B .*exp(1j*(Fc / Fs)*2*pi*(0:numel(B)-1));
D = dsp.FIRDecimator('DecimationFactor',BWFactor,'Numerator',Bbp);
```

Let's now perform the filtering and the FFT:

```
for k = 1:10
    % Run a few times to eliminate transient in filter
    x = sn1()+sn2();
    xd = D(x);
end
Xd = fft(xd);
figure
stem(F,abs(fftshift(Xd))/Ld)
grid on
xlabel('Frequency (Hz)')
ylabel('Magnitude')
title('Zoom FFT Spectrum. Bandpass Sampling Approach.')
```



Using a Multirate, Multistage Bandpass Filter

The filter from the previous section is a single-stage filter. We can achieve a less computationally complex filter by using a multistage design instead. This is the approach followed in dsp.ZoomFFT.

Let's consider the following example, where the input sample rate is 48 kHz, and the bandwidth of interest is the interval [1500,2500] Hz. The achievable decimation factor is then floor(48000/1000) = 48.

Let's first design a single-stage decimator:

```
Fs       = 48e3;
Fc       = 2000; % Bandpass filter center frequency
BW       = 1e3;  % Bandwidth of interest
```

```
Ast      = 80;  % Stopband attenuation
BWFactor = floor(Fs/BW);
B        = designMultirateFIR(1,BWFactor,12,80);
Bbp      = B .*exp(1j*(Fc / Fs)*2*pi*(0:numel(B)-1));
D_single_stage = dsp.FIRDecimator('DecimationFactor',BWFactor,'Numerator',Bbp);
```

Now, let's design the filter using a multistage design, while maintaining the same stopband attenuation and transition bandwidth as the single-stage case (see `kaiserord` for details on the transition width computation):

```
tw = (Ast - 7.95) / ( numel(B) * 2.285);
fmd = fdesign.decimator(BWFactor,'Nyquist',BWFactor,...
                        'TW,Ast', tw * Fs / (2*pi),Ast,Fs);
D_multi_stage = multistage(fmd,'HalfbandDesignMethod','equiripple','SystemObject',true);
fprintf('Number of filter stages: %d\n', getNumStages(D_multi_stage) );
fprintf('Stage 1: Decimation factor = %d , filter length = %d\n',...
        D_multi_stage.Stage1.DecimationFactor,...
        numel(D_multi_stage.Stage1.Numerator));
fprintf('Stage 2: Decimation factor = %d , filter length = %d\n',...
        D_multi_stage.Stage2.DecimationFactor,...
        numel(D_multi_stage.Stage2.Numerator));
fprintf('Stage 3: Decimation factor = %d , filter length = %d\n',...
        D_multi_stage.Stage3.DecimationFactor,...
        numel(D_multi_stage.Stage3.Numerator));
```

```
Number of filter stages: 3
Stage 1: Decimation factor = 6 , filter length = 33
Stage 2: Decimation factor = 2 , filter length = 11
Stage 3: Decimation factor = 4 , filter length = 101
```

Note that D_multi_stage is a three-stage multirate lowpass filter. We transform it to a bandpass filter by performing a frequency shift on the coefficients of each stage, while taking the cumulative decimation factor into account:

```
num1 = D_multi_stage.Stage1.Numerator;
num2 = D_multi_stage.Stage2.Numerator;
num3 = D_multi_stage.Stage3.Numerator;
N1 = length(num1)-1;
N2 = length(num2)-1;
N3 = length(num3)-1;
% Decimation factor between stage 1 & 2:
M12 = D_multi_stage.Stage1.DecimationFactor;
% Decimation factor between stage 2 & 3:
M23 = D_multi_stage.Stage2.DecimationFactor;
num1     = num1 .*exp(1j*(Fc / Fs)*2*pi*(0:N1));
num2     = num2 .*exp(1j*(Fc * M12 / Fs)*2*pi*(0:N2));
num3     = num3 .*exp(1j*(Fc * M12 * M23 / Fs)*2*pi*(0:N3));
D_multi_stage.Stage1.Numerator = num1;
D_multi_stage.Stage2.Numerator = num2;
D_multi_stage.Stage3.Numerator = num3;
```

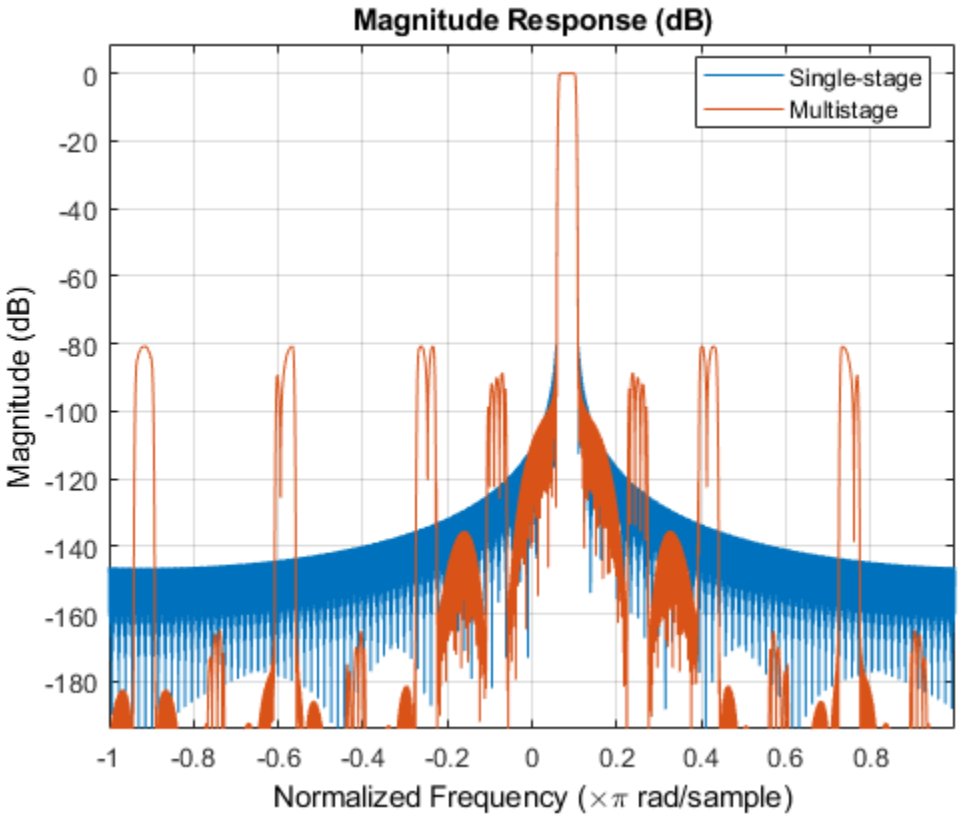
Let's compare the cost of the single-stage and multistage filters:

```
fprintf('Single-stage filter cost:\n\n')
cost(D_single_stage)
fprintf('Multistage filter cost:\n')
cost(D_multi_stage)
```

```
Single-stage filter cost:


ans =

  struct with fields:

                 NumCoefficients: 1129
                      NumStates: 1104
    MultiplicationsPerInputSample: 23.5208
          AdditionsPerInputSample: 23.5000

Multistage filter cost:

ans =

  struct with fields:

                 NumCoefficients: 113
                      NumStates: 140
    MultiplicationsPerInputSample: 7.0208
          AdditionsPerInputSample: 6.7500
```

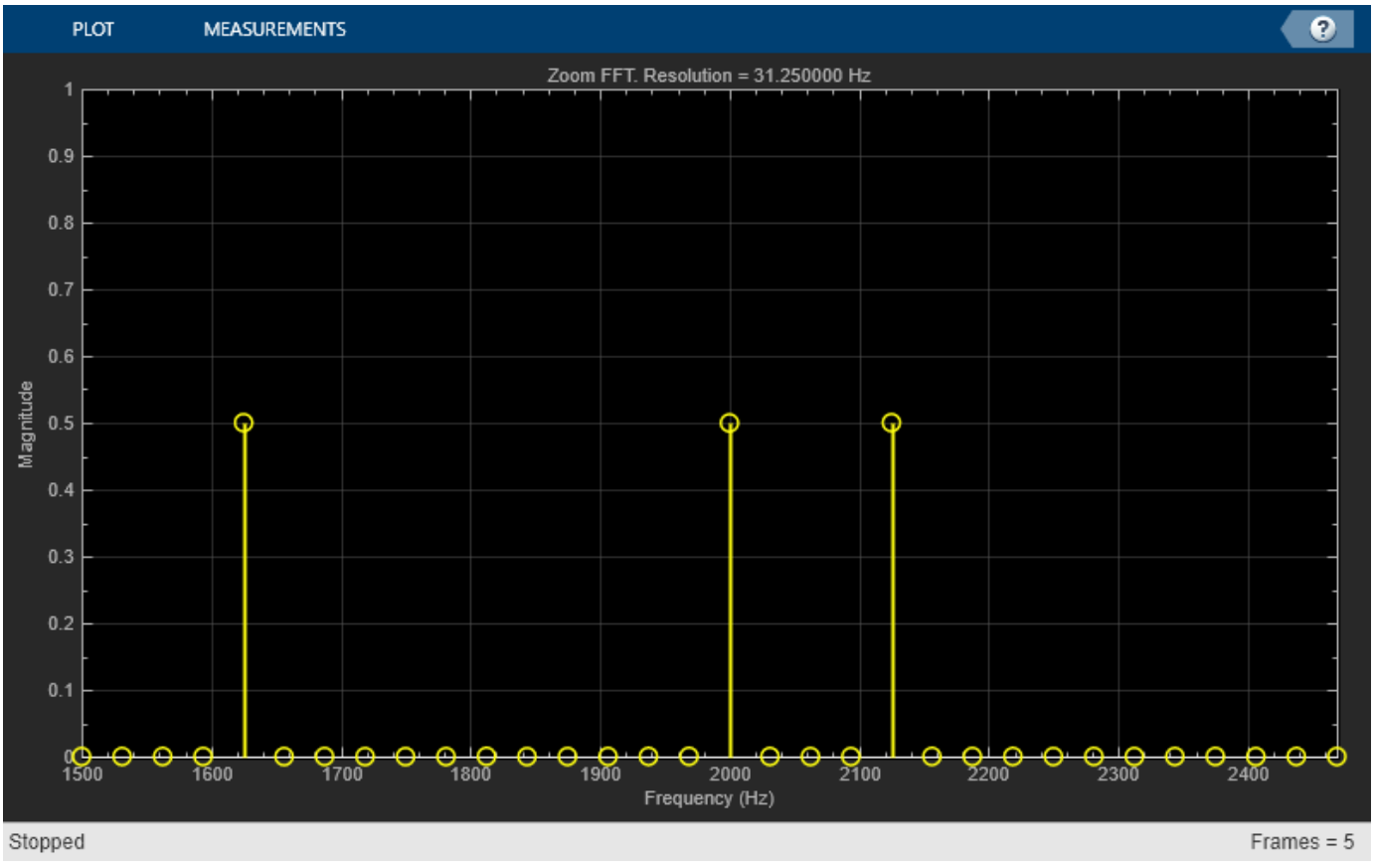
Let's also compare the frequency response of the two filters:

```
vis = fvtool(D_single_stage,D_multi_stage,'DesignMask','off','legend','on');
legend(vis,'Single-stage','Multistage')
```



Let's use the multistage filter to perform zoom FFT:

```
fftlen = 32;
L      = BWFactor * fftlen;
tones  = [1625 2000 2125]; % sine wave tones
sn     = dsp.SineWave('SampleRate',Fs,'Frequency',tones,...
                      'SamplesPerFrame',L);
Fsd    = Fs / BWFactor;
% Frequency points at which FFT is computed
F      = Fc + Fsd/fftlen*(0:fftlen-1)-Fsd/2;
ap = dsp.ArrayPlot('XDataMode','Custom',...
                   'CustomXData',F,...
                   'YLabel','Magnitude',...
                   'XLabel','Frequency (Hz)',...
                   'YLimits',[0 1],...
                   'Title',sprintf('Zoom FFT. Resolution = %f Hz',(Fs/BWFactor)/fftlen));
for k=1:100
    x = sum(sn(),2) +  1e-2 * randn(L,1);
    y = D_multi_stage(x);
    z = fft(y,fftlen,1);
    z = fftshift(z);
    ap( abs(z)/fftlen )
end
release(ap)
```

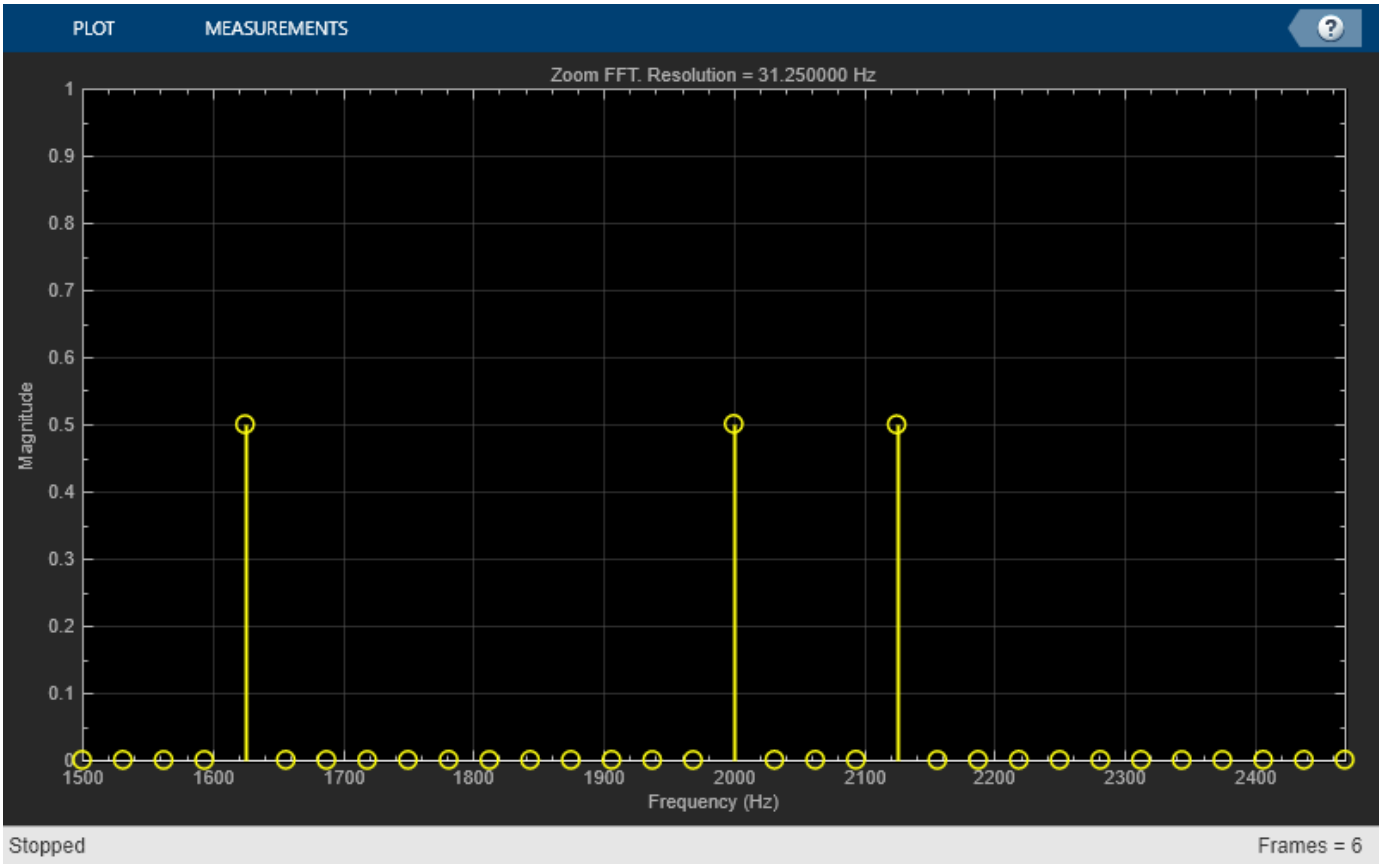


Using dsp.ZoomFFT

dsp.ZoomFFT is a System object that implements zoom FFT based on the multirate multistage bandpass filter highlighted in the previous section. You specify the desired center frequency and decimation factor, and dsp.ZoomFFT will design the filter and apply it to the input signal.

Let's use dsp.ZoomFFT to zoom into the sine tones from the previous section's example:

```
zfft = dsp.ZoomFFT(BWFactor,Fc,Fs);
for k=1:100
    x = sum(sn(),2) +  1e-2 * randn(L,1);
    z = zfft(x);
    z = fftshift(z);
    ap( abs(z)/fftlen)
end
release(ap)
```



Zoom FFT Simulink Block

The zoom FFT block brings the functionality of dsp.ZoomFFT to Simulink. In the model dspzoomfft, we use the zoom FFT block to inspect the frequency band [800 Hz, 1600 Hz] of an input signal sampled at 44100 Hz.

References

[1] Multirate Signal Processing - Harris (Prentice Hall).

[2] Computing Translated Frequencies in digitizing and Downsampling Analog Bandpass - Lyons (https://www.dsprelated.com/showarticle/523.php)

Outlier Removal Techniques with Streaming ECG Signals

This example explores different outlier removal filters and uses an electrocardiogram (ECG) signal as input.

Introduction

There are many different outlier removal techniques because a rigid definition of an outlier does not exist.

The three techniques explored in this example are:

- `dsp.MovingAverage`
- `dsp.MedianFilter`
- `dsp.HampelFilter`

ECG Signal Source

The ECG signal used in this example is taken from the MIT-BIH Arrhythmia Database. The signal is sampled at 360 Hz. The signal was shifted and scaled to convert it from the raw 12-bit ADC values to real-world values.

For more information on ECG signals, please see the example “Real-Time ECG QRS Detection” on page 4-448.

Setup

First, create a stream from the ECG signal using the dsp.MatFileReader. Next, create a scope to visualize the raw and filtered signals.

```
Fs = 360;
frameSize = 500;
fileName = 'ecgsig.mat';
winLen = 13; % Window length for the filters.

fileReader = dsp.MatFileReader('Filename',fileName, ...
    'VariableName','ecgsig','SamplesPerFrame',frameSize);
scope = timescope('SampleRate',Fs,'TimeSpanSource','property', ...
    'TimeSpan',2,'YLimits',[-1.5 1.5], ...
    'LayoutDimensions',[2 1]);
scope.ActiveDisplay = 1;
scope.Title = 'Raw Signal';
scope.ActiveDisplay = 2;
scope.Title = 'Filtered Signal';
```

Outlier Removal Performance of Moving Average Filter

The moving average filter calculates a running mean on the specified window length. This is a relatively simple calculation compared to the other two filters. However, this will smooth both the signal and the outliers. This causes the peak in the ECG signal to be smoothed to roughly a third of its original magnitude.

```
movAvg = dsp.MovingAverage(winLen);
while ~isDone(fileReader)
    x = fileReader();
```

```
    y = movAvg(x);
    scope(x,y);
end

% Clean up
release(scope);
reset(fileReader);
reset(scope);
```

Outlier Removal Performance of Median Filter

The Median Filter is sometimes a better choice since it is less sensitive to outliers than the Moving Average Filter. However, as can be seen in the scope below, it can cause "steps" to appear at extremes in the signal where the local median does not change. This means that the window length of the filter must be carefully considered.

```
medFilt = dsp.MedianFilter(winLen);
while ~isDone(fileReader)
    x = fileReader();
    y = medFilt(x);
    scope(x,y);
end

% Clean up
release(scope);
reset(fileReader);
reset(scope);
```

Outlier Removal Performance of Hampel Filter

The Hampel Filter has an additional threshold parameter that can be set. Below, it is set to one, meaning that any sample that is more than one standard deviation away from the local median will be classified as an outlier. Both the threshold and the window length can be changed to remove outliers from the input signal without distorting the original signal.

```
thres = 1;
hampFilt = dsp.HampelFilter(winLen,thres);
while ~isDone(fileReader)
    x = fileReader();
    y = hampFilt(x);
    scope(x,y);
end

% Clean up
release(scope);
reset(fileReader);
reset(scope);
```

Conclusion

All three of the above filters can be used for outlier removal. The noise distribution of the outliers and the window length both effect the filter's performance. This must be taken into consideration when selecting a filter for outlier removal in a specific application.

References

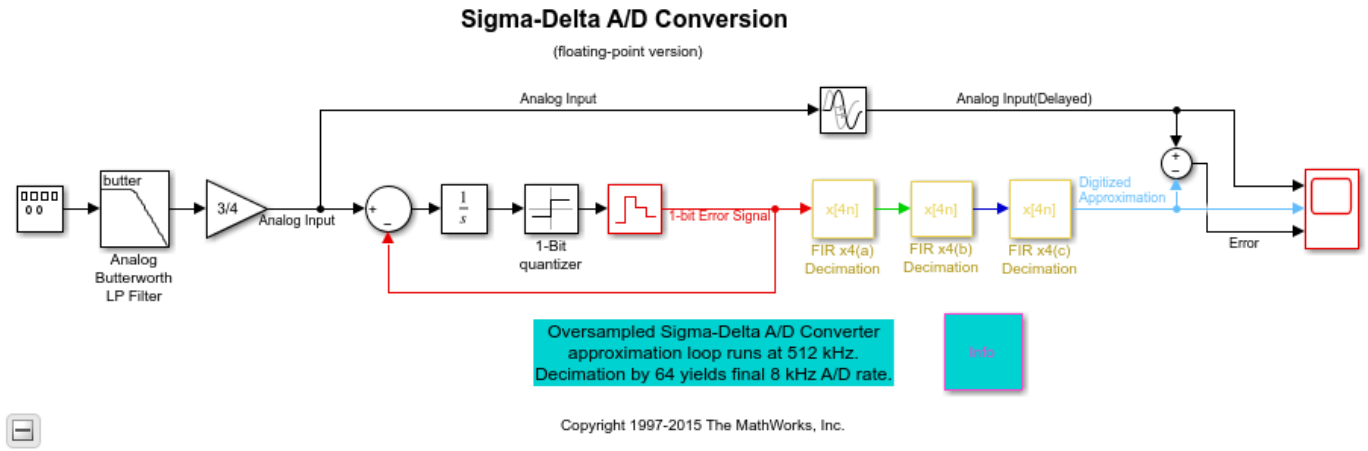
Goldberger AL, Amaral LAN, Glass L, Hausdorff JM, Ivanov PCh, Mark RG, Mietus JE, Moody GB, Peng C-K, Stanley HE. "PhysioBank, PhysioToolkit, and PhysioNet: Components of a New Research Resource for Complex Physiologic Signals." Circulation 101(23):e215-e220, 2000. `http://circ.ahajournals.org/cgi/content/full/101/23/e215`

Moody GB, Mark RG. "The impact of the MIT-BIH Arrhythmia Database." IEEE Eng in Med and Biol 20(3):45-50 (May-June 2001).

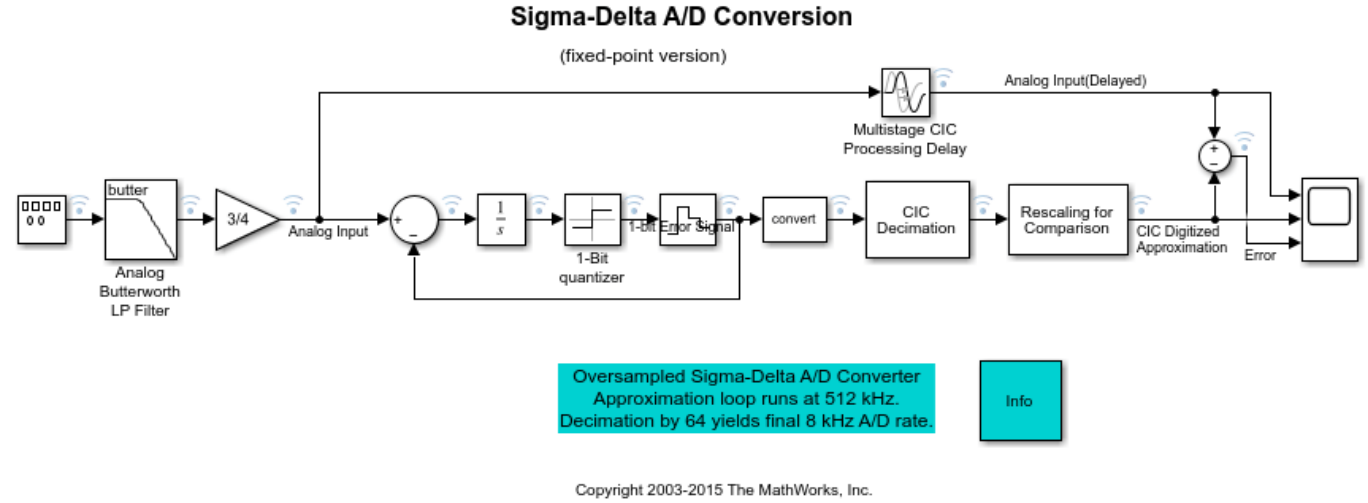
Sigma-Delta A/D Conversion

This example shows how to model analog-to-digital conversion using a sigma-delta algorithm implementation.

Floating-point Example Model



Fixed-point Example Model



Exploring the Example

The Oversampled Sigma-Delta A/D Converter is a noise-shaping quantizer. The main purpose of noise-shaping is to reshape the spectrum of quantization noise so that most of the noise is filtered out of the relevant frequency band, for example, the audio band for speech applications. The main objective is to trade bits for samples; that is, to increase the sampling rate but reduce the number of bits per sample. The resulting increase in quantization noise is compensated by a noise-shaping quantizer. This quantizer pushes the added quantization noise out of the relevant frequency band and thereby preserves a desired level of signal quality. This reduction in the number of bits simplifies the structure of A/D and D/A converters.

As seen in this example, the analog input is prefiltered by an antialiasing prefilter whose structure is simplified because of oversampling. The input signal is oversampled by a factor of 64. The Integrator, 1-Bit Quantizer, and Zero-Order Hold blocks comprise a two-level analog to digital converter (ADC). The output of the Zero-Order Hold is then subtracted from the analog input. The feedback, or approximation, loop causes the quantization noise generated by the ADC to be highpass filtered, pushing its energy towards the higher frequencies (`64*fs/2`) and away from the relevant signal band. The decimation stage reduces the sampling rate back to 8 KHz. During this process, it removes the high frequency quantization noise that was introduced by the feedback loop and removes any undesired frequency components beyond `fs/2` (4 KHz) that were not removed by the simple analog prefilter.

Decimator Design

The example versions illustrate two possible decimator design solutions.

The floating-point version model uses a cascade of three polyphase FIR decimators. This approach reduces computation and memory requirements as compared to a single decimator by using lower-order filters. Each decimator stage reduces the sampling rate by a factor of four. The latency introduced by the filters is used to set the appropriate 'Time Delay' in the 'Transport Delay' block. The three FIR Decimation filters each introduce a latency of 16 samples, due to the group delay of the filter (the actual value of 15.5 is rounded up to the nearest integer number of samples). Due to the decimation operation the total latency introduced by the three filters is as follows: 16 (first filter) + 4*16 (second filter) + 16*16 (third filter) to give a final total delay of 336. The denominator of the 'Time delay' parameter is the base rate of the model (512 kHz).

The fixed-point version uses a five-section CIC decimator to reduce the sampling rate by the same factor of 64. While not as flexible as a FIR decimator, the CIC decimator has the advantage of not requiring any multiply operations. It is implemented using only additions, subtractions, and delays. Therefore, it is a good choice for a hardware implementation where computational resources are limited. The CIC Decimator introduces a latency of 158 samples, which is the group delay of the filter (157.5) rounded up to the nearest integer. This is the value used in 'Time Delay' parameter of the 'Multistage CIC Processing Delay' block.

References

Orfanidis, S. J. **Introduction To Signal Processing**, Prentice Hall, 1996.

Available Example Versions

Floating-point version: dspsdadc

Fixed-point version: dspsdadc_fixpt

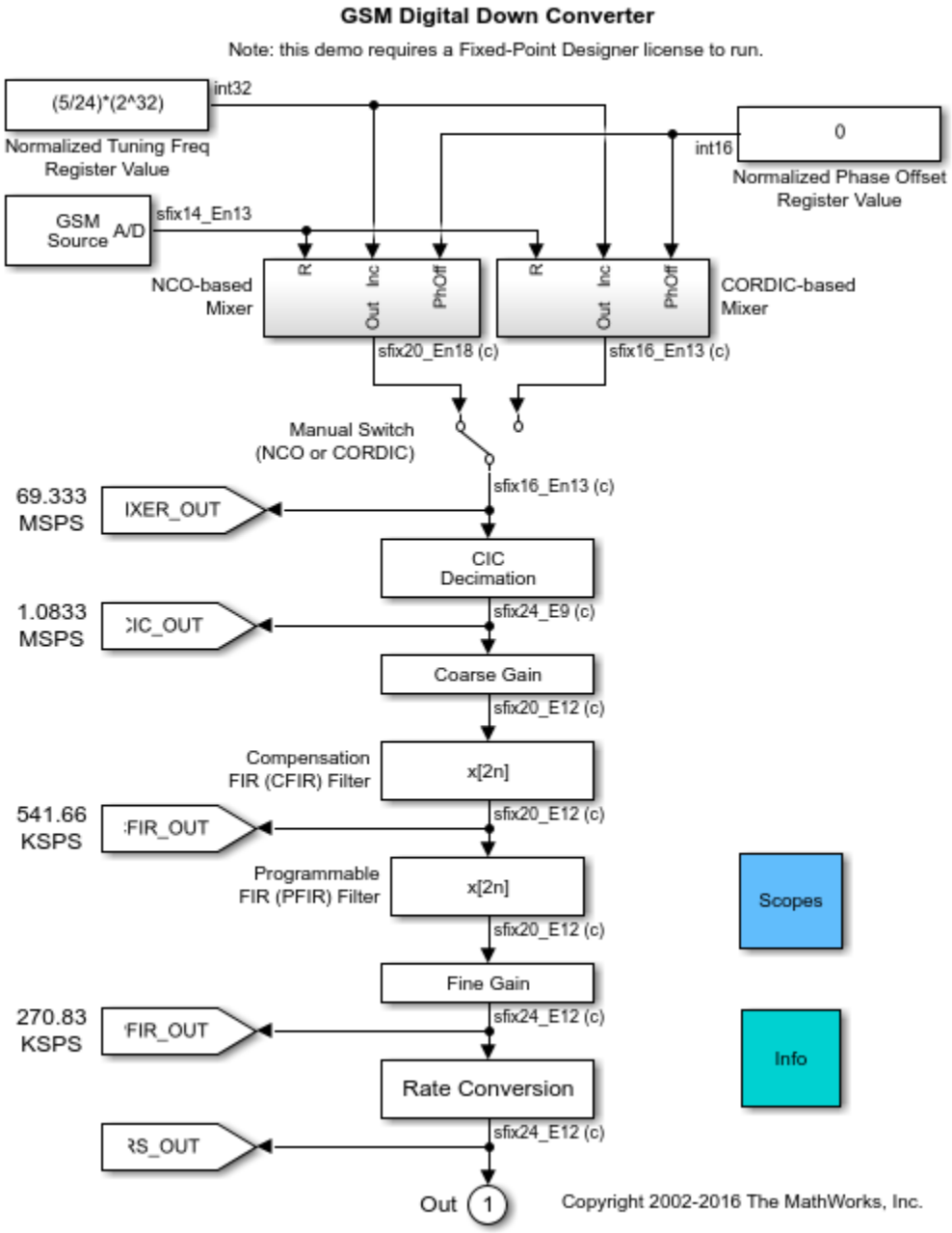
GSM Digital Down Converter in Simulink

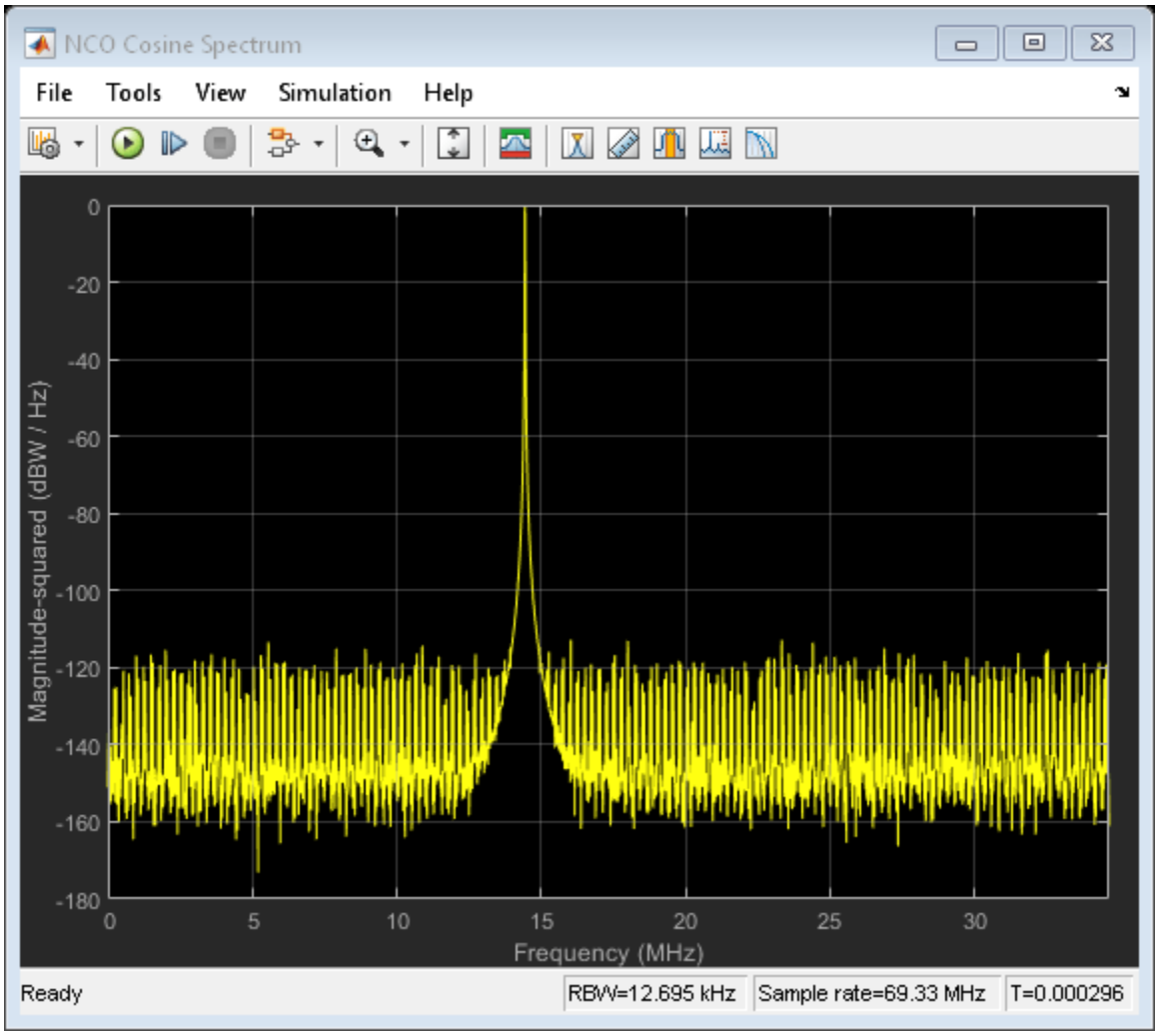
This example shows how to simulate steady-state behavior of a fixed-point digital down converter for GSM (Global System for Mobile) baseband conversions. The example model uses blocks from Simulink® and the DSP System Toolbox™ to emulate the operation of the TI GC4016 Quad Digital Down Converter (DDC).

The DDC performs:

- Digital mixing (down conversion) of the input signal
- Narrow band low-pass filtering and decimation
- Gain adjustment and final resampling of the data stream

In this model, the DDC accepts a high sample-rate (69.333 MSPS) bandpass signal. The DDC produces a low sample-rate (270.83 KSPS) baseband signal, ready for demodulation.


GSM Digital Down Converter
Note: this demo requires a Fixed-Point Designer license to run.
(5/24)*(2^32)
int32
Normalized Tuning Freq
Register Value
0
int16
Normalized Phase Offset
Register Value
GSM
Source
A/D
sfix14_En13
NCO-based
Mixer
R
Inc
PhOff
Out
CORDIC-based
Mixer
sfix20_En18 (c)
sfix16_En13 (c)
Manual Switch
(NCO or CORDIC)
sfix16_En13 (c)
69.333
MSPS
IXER_OUT
CIC
Decimation
1.0833
MSPS
IC_OUT
sfix24_E9 (c)
Coarse Gain
sfix20_E12 (c)
Compensation
FIR (CFIR) Filter
x[2n]
541.66
KSPS
FIR_OUT
sfix20_E12 (c)
Programmable
FIR (PFIR) Filter
x[2n]
sfix20_E12 (c)
Scopes
Fine Gain
270.83
KSPS
FIR_OUT
sfix24_E12 (c)
Info
Rate Conversion
S_OUT
sfix24_E12 (c)
Out
1
Copyright 2002-2016 The MathWorks, Inc.



Changing the GSM Source

You can switch between a chirp and a sinusoid signal using the `GSM Source` block in the example model. You can replace this block with a different source to model your application, however you will have to adjust the parameters of the downstream mixer subsystems.

Adjusting the Normalized Tuning Frequency and Phase Offset Values

To assure that your GSM source signal gets received and mixed down with minimum error, you should adjust the `Normalized Tuning Freq Register Value` and `Normalized Phase Offset Register Value`.

Since this example is simulating the TI GC4016 Quad Digital Down Converter, these values must be entered in a particular format. The `Normalized Tuning Freq Register Value` should be a signed twos-complement 32-bit integer, representing a normalized range between 0 and the sampling frequency. Use positive frequency values for down conversion. The `Normalized Phase Offset Register Value` should be an unsigned 16-bit integer, also representing a normalized range. For more details, please refer to the TI GC4016 Quad Digital Down Converter documentation and the DSP System Toolbox `NCO` library block reference documentation.

Comparing the NCO-based and CORDIC-based Mixer Implementations

View the Digital Mixer Real Output scope and Mixer Output Comparison scope to compare the NCO-based Mixer implementation outputs to the CORDIC-based Mixer implementation outputs. Both implementations can be made to produce similar output values, however the implementation choice is based largely on available hardware resources and performance constraints. In general, NCO-based approaches trade off lookup table size (read-only memory resources) with speed performance, whereas CORDIC-based approaches may trade off speed performance for smaller memory resources, based on the number of CORDIC kernel iterations needed.

Adjusting the NCO-based Mixer Parameters

Look at the output of the NCO Cosine Spectrum Analyzer block to observe the effects of tuning NCO-based Mixer subsystem block parameters.

Dithering

To spread the spurious frequencies throughout the available bandwidth, you can add a dither signal to the accumulator phase values. In this example, the dither signal is generated by a PN Sequence Generator consisting binary shift registers and exclusive-or gates (internal to the NCO block). The number of dither bits is automatically determined by

```
number of dither bits = accumulator word length - table address word length
```

When you increase the number of dither bits beyond the optimal value, the noise floor begins to rise. When you decrease the number of dither bits below the optimal value, the appearance of spurious frequencies will decrease the spurious free dynamic range of the NCO system.

For more information, please see the DSP System Toolbox `NCO` library block reference documentation.

Adjusting the CORDIC-based Mixer Parameters

Look at the output of the CORDIC Cosine Spectrum Analyzer block to observe the effects of tuning CORDIC-based Mixer subsystem block parameters.

Phase Accumulator with Dither Generator

The Phase Accumulator with Dither Generator subsystem computes the angle `Theta` input of the CORDIC Complex Rotate function. Look at the output of the CORDIC Cosine Spectrum Analyzer block to observe the effects of tuning the Phase Accumulator with Dither Generator subsystem parameters.

As in the NCO-based Mixer described above, you can add a dither signal to the phase accumulator values to spread the spurious frequencies throughout the available bandwidth. The dither signal is generated by a PN Sequence Generator consisting of binary shift registers and exclusive-or gates (internal to the Phase Accumulator with Dither Generator). The number of dither bits was chosen to be 15 to closely match the cosine spectrum performance of the NCO-based Mixer.

CORDIC Complex Rotate

The CORDIC Complex Rotate computes `u * exp(j*theta)` using a CORDIC rotation algorithm. Refer to the Fixed-Point Designer™ documentation to learn about the `CORDICROTATE` function. Also please refer to the references listed below for more information on using CORDIC-based digital mixer approaches.

Adjusting Decimation Filter Parameters

The CIC Decimator, Compensation FIR, and Programmable FIR blocks are used together to achieve:

- A high decimation ratio
- Aliasing attenuation
- Application-specific filtering

You can use Filter Designer to visualize and analyze the filters. Refer to the Signal Processing Toolbox™ documentation to learn about Filter Designer.

Double-clicking on the CIC Decimator block in the example model lets you see the implementation of the filter. To customize the DDC, you can change the CIC filter by editing the CIC Decimation block parameters.

CIC Decimation filters are implemented using integer overflow "wrap" arithmetic to perform the decimation filtering within their cascaded integrator-comb structures. This type of filter is economical for implementation on hardware such as FPGAs and ASICs, because the only arithmetic operation required is summing; no multiplies are required. For more information on CIC filters please refer to the references below.

The Compensation FIR block adjusts for roll-off of the CIC passband, and the Programmable FIR block filters the signal to meet the requirements of the GSM baseband spectral mask. You can adjust the gain and coefficients of these filters.

The input gain to Compensation FIR filter is set through the `COARSE` gain parameter. The TI GC4016 Quad Digital Down Converter requires input from a `COARSE` parameter to shift the output of the CIC filter by 0 - 7 bits, according to `2^COARSE`. Thus, you may enter 0 - 7 for the `COARSE` gain parameter in the Coarse Gain block mask.

The gain at the output of the Programmable FIR block is set through the `FINE` gain parameter. The TI GC4016 Quad Digital Down Converter requires input from a `FINE` parameter to shift the signal by 1 - 4 bits, according to `FINE/1024`. Thus, you may enter `1` to `16383` for the `FINE` gain parameter in the Fine Gain block mask.

Adjusting Rate Conversion Block Parameters

This final stage of the DDC can be used to change the rate of the output of the DDC to match the baseband frequency of your particular system's demodulator input. The Rate Conversion block is a fixed-point filter that acts similarly to the FIR Rate Conversion block in the DSP System Toolbox. The Rate Conversion block's `NDELAY` parameter is the interpolation factor, and the `NDEC` parameter is the decimation factor.

Analyzing the DDC

You can use scopes and the Fixed-Point Tool to observe and analyze the results of your simulation.

Scopes

Double-click on the Scopes block in the example model to gain access to the following scopes:

- NCO Cosine Spectrum
- CORDIC Cosine Spectrum

- Digital Mixer Real Output
- Mixer Output Comparison
- CIC Decimator Output
- Compensation FIR Output
- Programmable FIR Output
- Resampler Output

Fixed-Point Tool

Invoke the Fixed-Point Tool interface for the example by going to the Analysis menu and selecting Fixed-Point Tool. This interface allows you to see the maximum values, minimum values, and overflows for fixed-point blocks in any subsystem in the example model. Refer to the Simulink and Fixed-Point Designer™ documentation for more information on the Fixed-Point Tool.

More Information

More information on CIC filters can be found here:

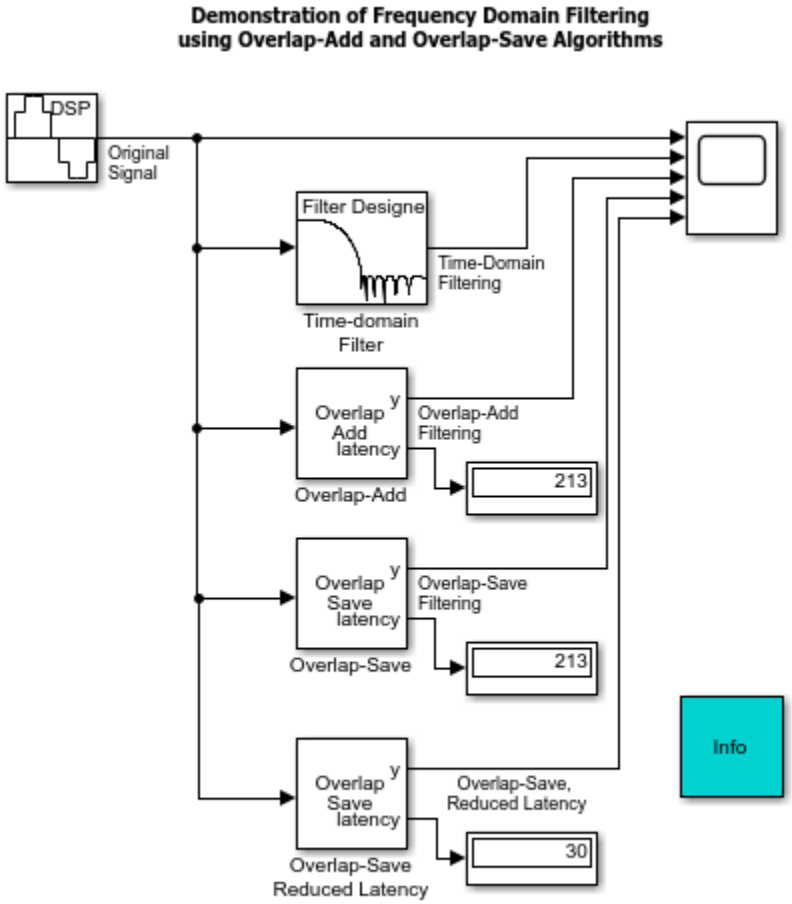
- Hogenauer, E. B., "An Economical Class of Digital Filters for Decimation and Interpolation," **IEEE® Transactions on Acoustics, Speech, and Signal Processing**, ASSP-29(2):155 - 162, 1981.

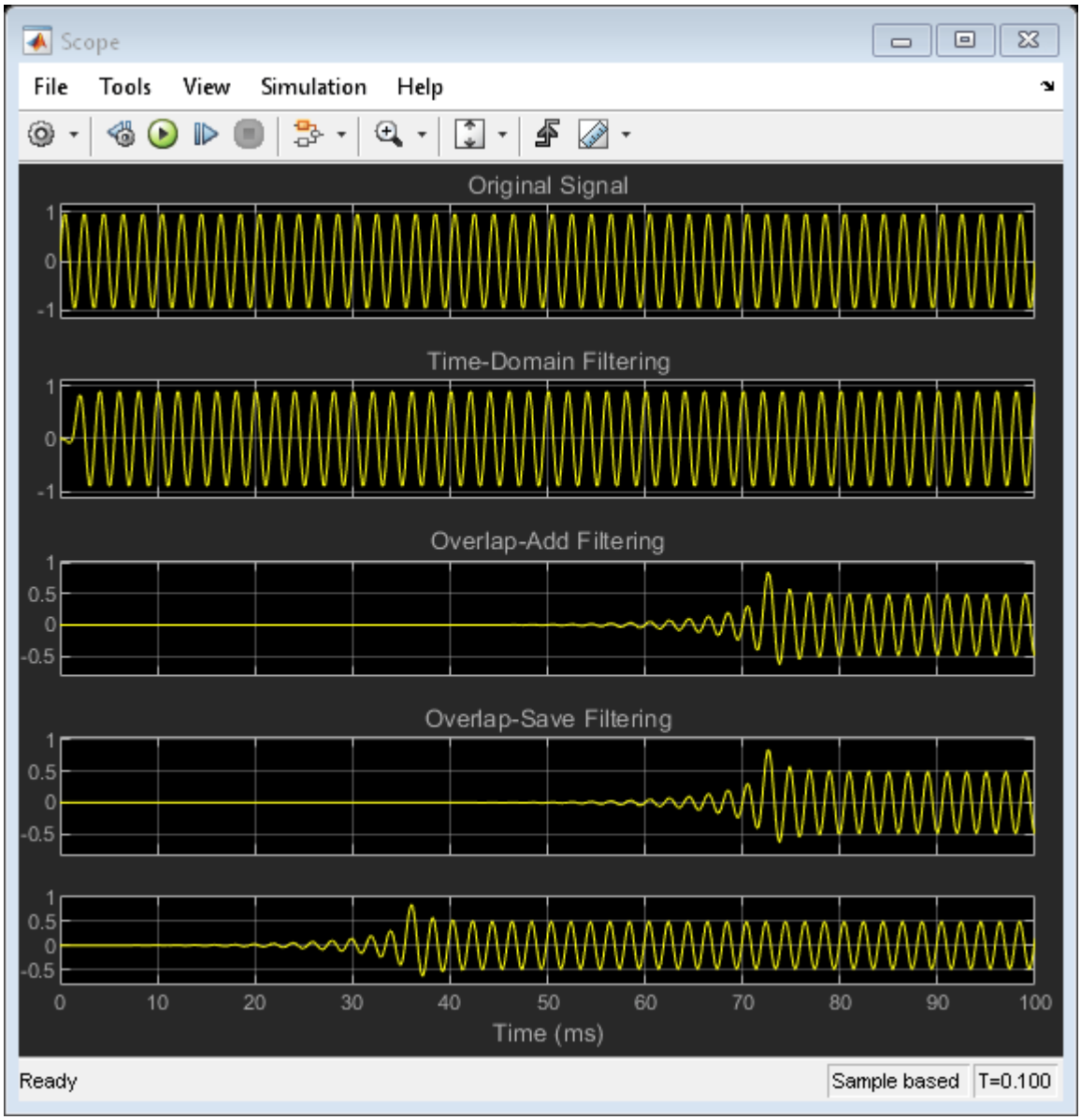
More information on CORDIC-based down conversion can be found here:

- Lohning, M., Hentschel, T., and Fettweis, G., "Digital Down Conversion in Software Radio Terminals", **Proceedings of the Tenth European Signal Processing Conference (EUSIPCO)**, 1517 - 1520, 2000.
- Valss, J., Sansaloni, T., Perez-Pascual, A., Torres, V., and Almenar, V., "The Use of CORDIC in Software Defined Radios: A Tutorial", **IEEE Communications Magazine**, 46 - 50, September 2006.
- Yang, S., Wu, Z., and Ren, G., "Design and Implementation of FPGA-Based FSK IF Digital Receiver", **1st International Symposium on Systems and Control in Aerospace and Astronautics (ISSCAA)**, 819 - 821, January 2006.
- Andraka, Ray, "A survey of CORDIC algorithm for FPGA based computers", **Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays**, 191 - 200, Feb. 22-24, 1998.
- Volder, Jack E., "The CORDIC Trigonometric Computing Technique", **IRE Transactions on Electronic Computers**, Volume EC-8, 330 - 334, September 1959.

Overlap-Add/Save

This example shows how to filter a sinusoid with the Overlap-Add and Overlap-Save FFT methods using the Frequency-Domain FIR filter block.





The overlap-add algorithm [1] filters the input signal in the frequency domain. The input is divided into non-overlapping blocks which are linearly convolved with the FIR filter coefficients. The linear convolution of each block is computed by multiplying the discrete Fourier transforms (DFTs) of the block and the filter coefficients, and computing the inverse DFT of the product. For filter length `M` and FFT size `N`, the last `M-1` samples of the linear convolution are added to the first `M-1` samples of the next input sequence. The first `N-M+1` samples of each summation result are output in sequence.

The overlap-save algorithm [2] also filters the input signal in the frequency domain. The input is divided into overlapping blocks which are circularly convolved with the FIR filter coefficients. The circular convolution of each block is computed by multiplying the DFTs of the block and the filter coefficients, and computing the inverse DFT of the product. For filter length `M` and FFT size `N`, the first `M-1` points of the circular convolution are invalid and discarded. The output consists of the remaining `N-M+1` points, which are equivalent to the true convolution.

Overlap-save and overlap-add introduce a processing latency of N-M+1 samples. You can reduce this latency by partitioning the numerator into shorter segments, applying overlap-add or overlap-save over the partitions, and then combining the results to obtain the filtered output [3]. The latency is reduced to the partition length, at the expense of additional computation compared to traditional overlap-save/overlap-add (though still numerically more efficient than time-domain filtering for long filters). In this model, we use a partition length of 30, which reduces the latency from 213 samples for traditional overlap-add/overlap-save to 30 samples.

References

[1] Overlap-Add Algorithm: Proakis and Manolakis, **Digital Signal Processing**, 3rd ed, Prentice-Hall, Englewood Cliffs, NJ, 1996, pp. 430 - 433.

[2] Overlap-Save Algorithm: Oppenheim and Schafer, **Discrete-Time Signal Processing**, Prentice-Hall, Englewood Cliffs, NJ, 1989, pp. 558 - 560.

[3] T. G. Stockham Jr., "High-speed convolution and correlation", Proc. 1966 Spring Joint Computer Conf., AFIPS, Vol 28, 1966, pp. 229-233.

Designing Low Pass FIR Filters

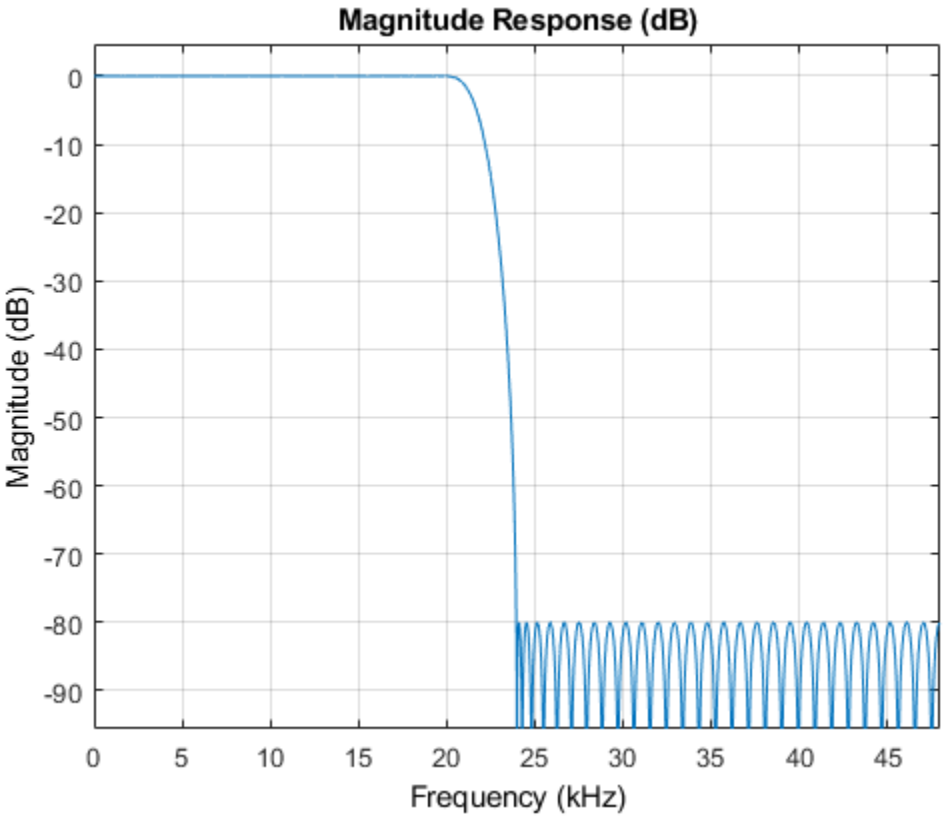
This example shows how to design lowpass FIR filters. Many of the concepts presented here can be extended to other responses such as highpass, bandpass, etc.

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 lowpass FIR filters with a variety of characteristics.

Obtaining Lowpass FIR Filter Coefficients

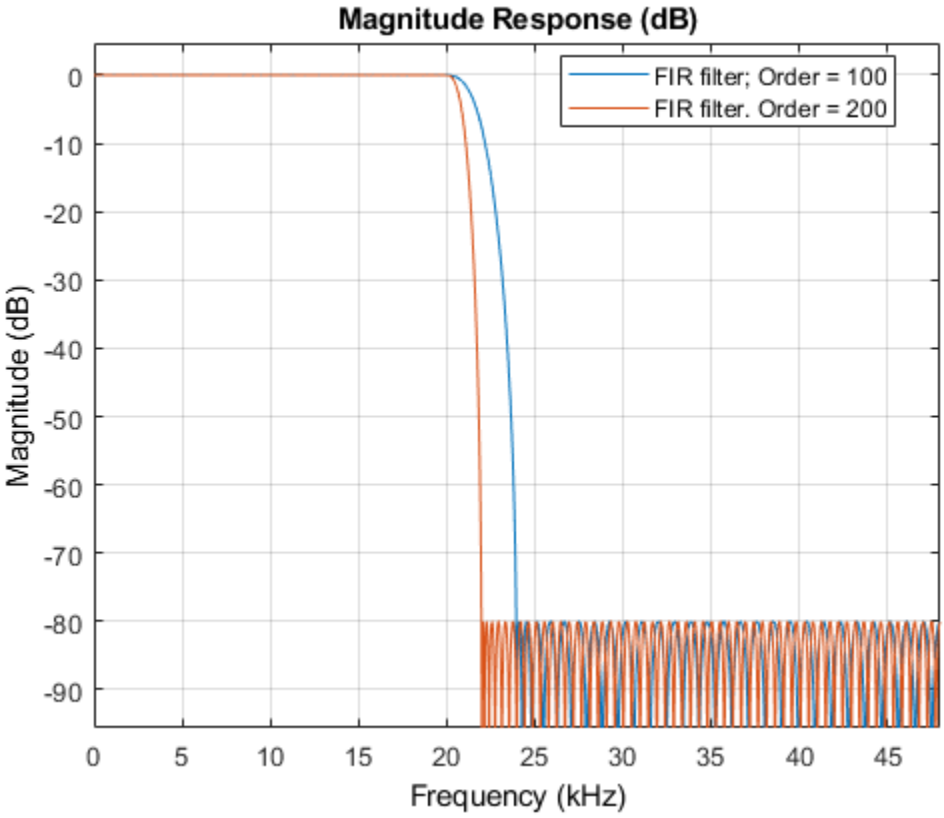
“Lowpass Filter Design in MATLAB” on page 1-12 provides an overview on designing lowpass filters with DSP System Toolbox. To summarize, two functions are presented that return a vector of FIR filter coefficients: `firceqrip` and `firgr`. `firceqrip` is used when the filter order (equivalently the filter length) is known and fixed.

```
N   = 100;        % FIR filter order
Fp  = 20e3;       % 20 kHz passband-edge frequency
Fs  = 96e3;       % 96 kHz sampling frequency
Rp  = 0.00057565; % Corresponds to 0.01 dB peak-to-peak ripple
Rst = 1e-4;       % Corresponds to 80 dB stopband attenuation

eqnum = firceqrip(N,Fp/(Fs/2),[Rp Rst],'passedge'); % eqnum = vec of coeffs
fvtool(eqnum,'Fs',Fs,'Color','White') % Visualize filter
```



The choice of a filter order of 100 was arbitrary. In general, a larger order results in a better approximation to ideal at the expense of a more costly implementation. Doubling the order roughly reduces the filter's transition width in half (assuming all other parameters remain the same).

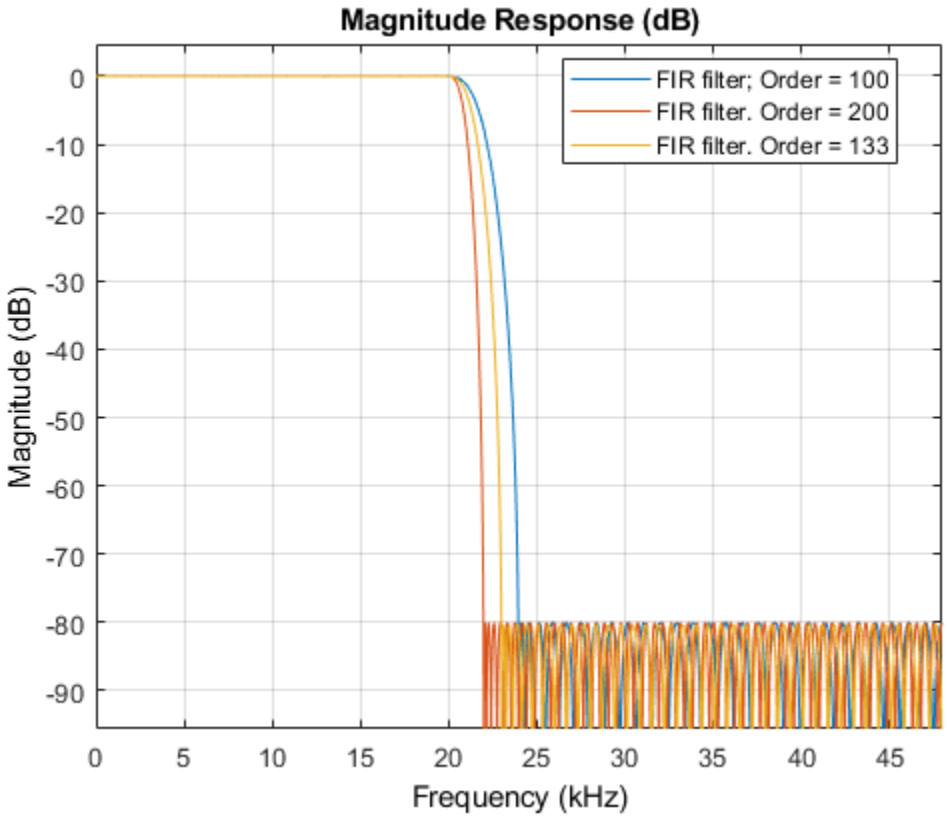
```
N2 = 200; % Change filter order from 100 to 200
eqNum200 = firceqrip(N2,Fp/(Fs/2),[Rp Rst],'passedge');
fvt = fvtool(eqnum,1,eqNum200,1,'Fs',Fs,'Color','White');
legend(fvt,'FIR filter; Order = 100','FIR filter. Order = 200')
```



Minimum-Order Lowpass Filter Design

Instead of specifying the filter order, `firgr` can be used to determine the minimum-order required to meet the design specifications. In order to do so, it is necessary to specify the width of the transition region. This is done by setting the stopband edge frequency.

```
Fst = 23e3;  % Transition Width = Fst - Fp
numMinOrder = firgr('minorder',[0,Fp/(Fs/2),Fst/(Fs/2),1],[1 1 0 0],...
    [Rp Rst]);
fvt = fvtool(eqnum,1,eqNum200,1,numMinOrder,1,'Fs',Fs,'Color','White');
legend(fvt,'FIR filter; Order = 100','FIR filter. Order = 200',...
    'FIR filter. Order = 133')
```

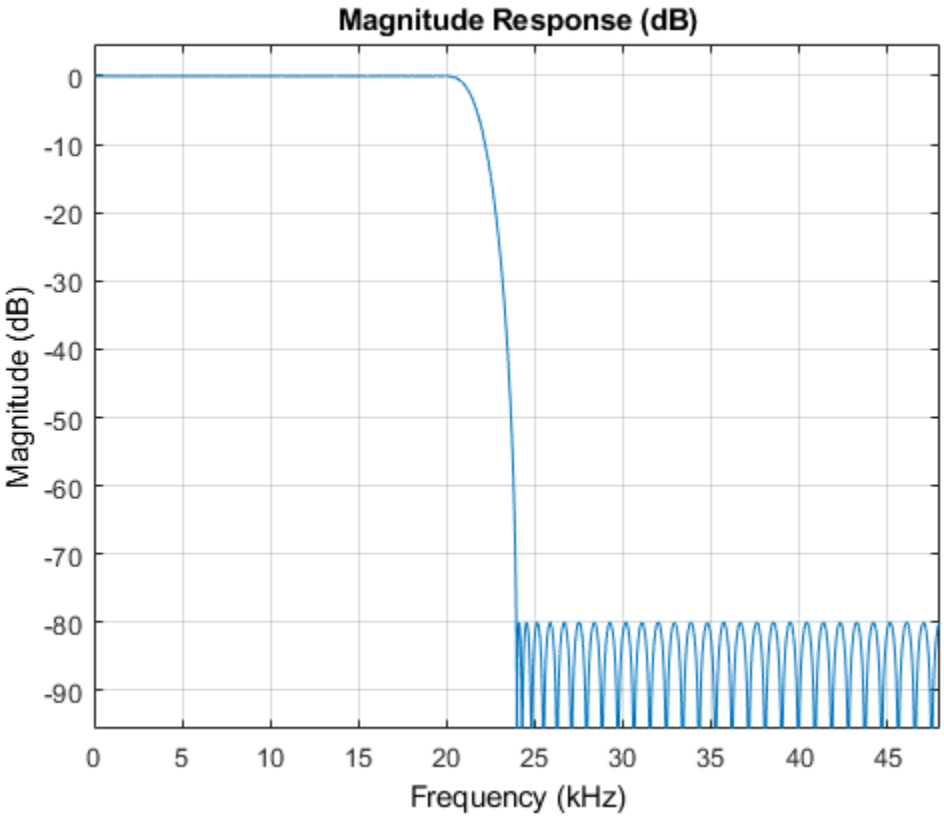


It is also possible to design filters of minimum even order ('mineven') or minimum odd order ('minodd') through the `firgr` function.

Implementing the Lowpass FIR Filter

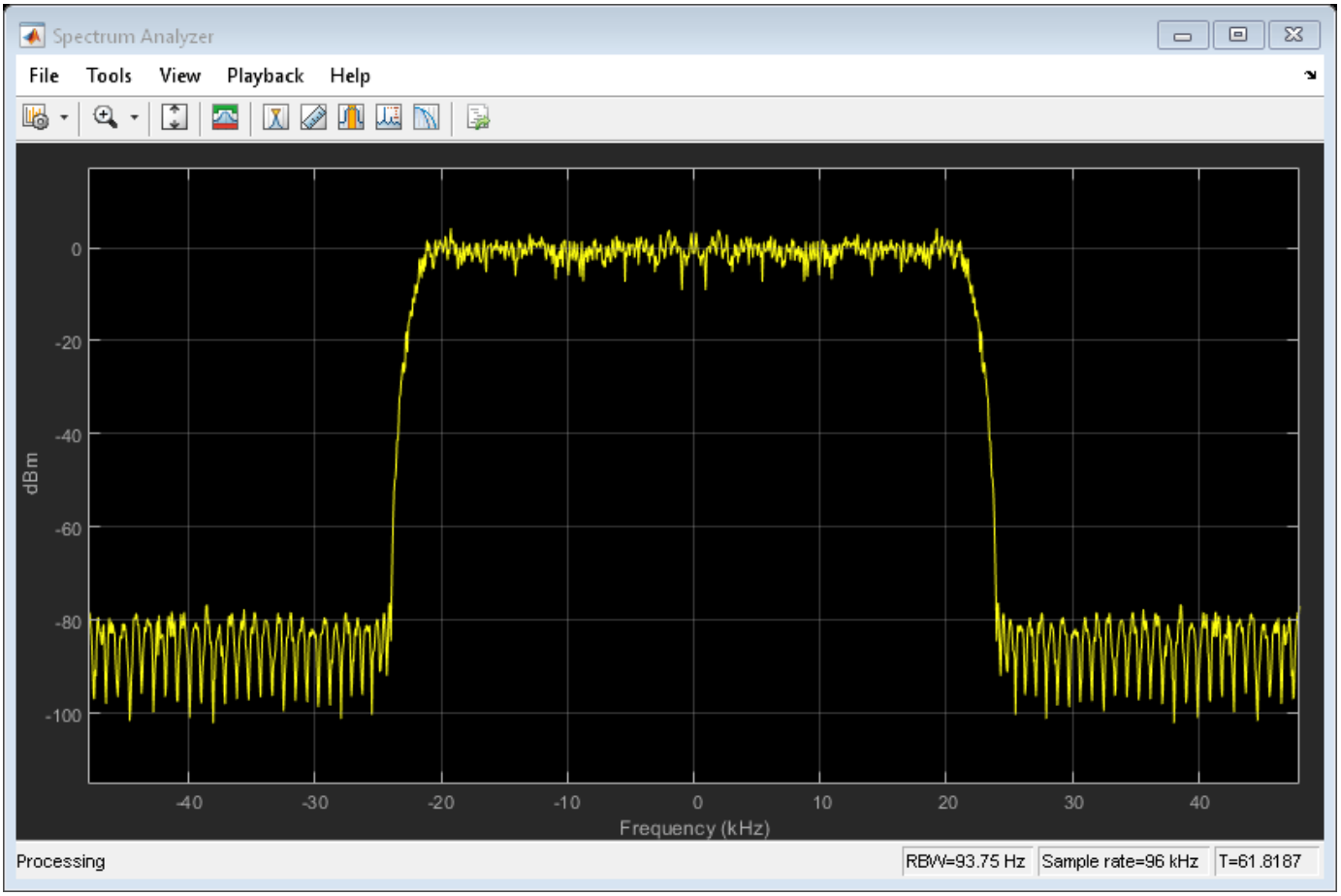
Once the filter coefficients have been obtained, the filter can be implemented with `dsp.FIRFilter`. This supports double/single precision floating-point data as well as fixed-point data. It also supports C and HDL code generation as well as optimized code generation for ARM® Cortex® M and ARM Cortex A.

```
lowpassFIR = dsp.FIRFilter('Numerator',eqnum); %or eqNum200 or numMinOrder
fvtool(lowpassFIR,'Fs',Fs,'Color','White')
```



In order to perform the actual filtering, call the FIR directly like a function. The following code filters Gaussian white noise and shows the resulting filtered signal in a spectrum analyzer for 10 seconds.

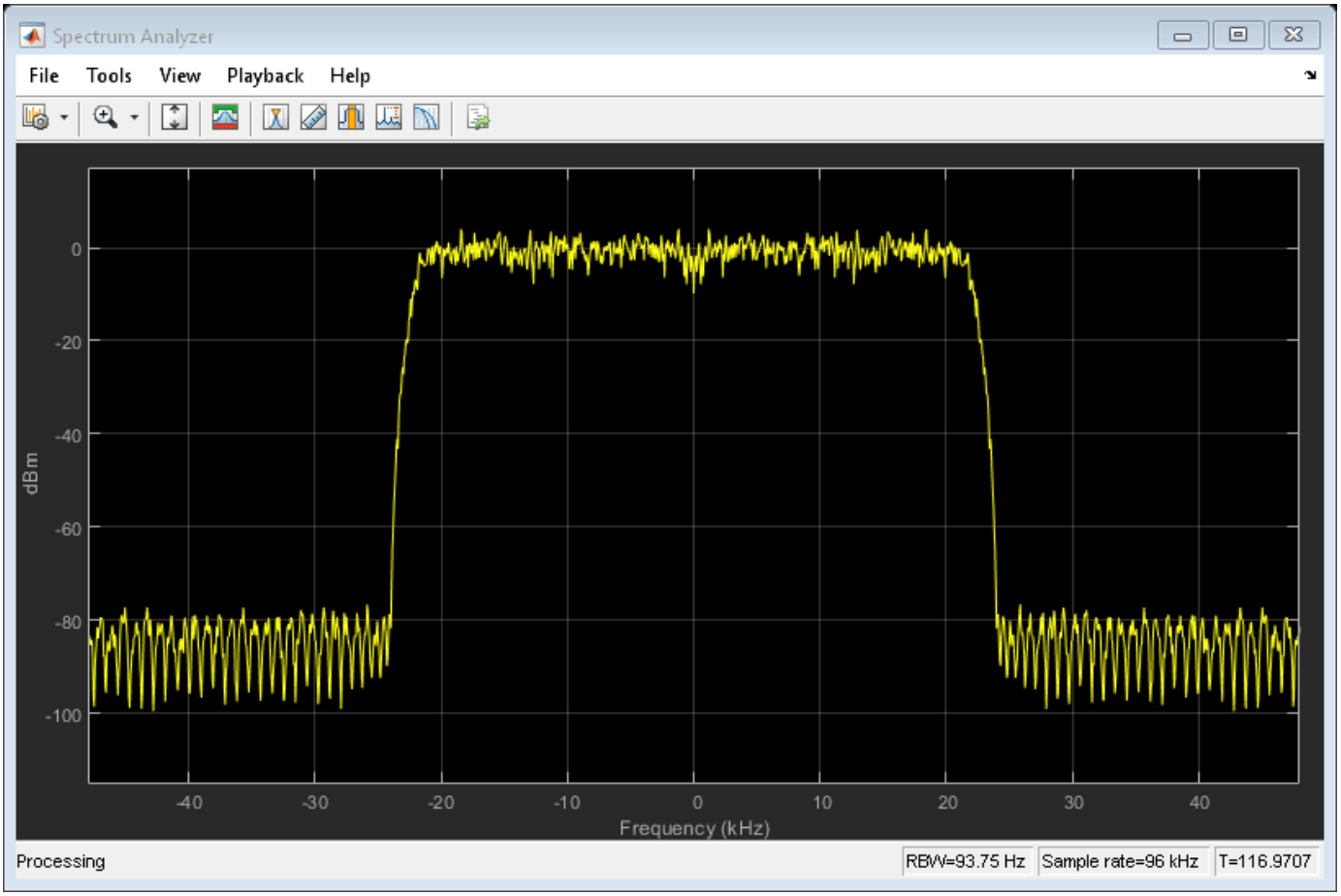
```
scope  = dsp.SpectrumAnalyzer('SampleRate',Fs,'SpectralAverages',5);
tic
while toc < 10
    x = randn(256,1);
    y = lowpassFIR(x);
    scope(y);
end
```



Designing and Implementing the Filter in One Step

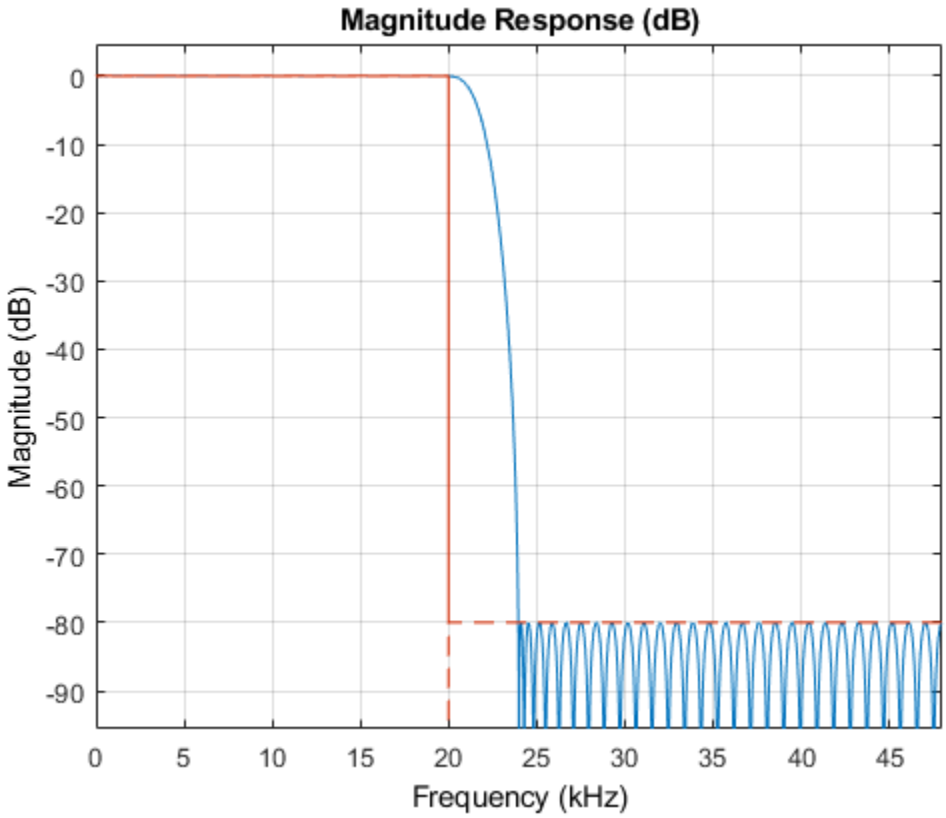
As a convenience, designing and implementing the filter can be done in a single step using `dsp.LowpassFilter`. This also supports floating-point, fixed-point, C code generation, and ARM Cortex M and ARM Cortex A optimizations.

```
lowpassFilt = dsp.LowpassFilter('DesignForMinimumOrder',false, ...
    'FilterOrder',N,'PassbandFrequency',Fp,'SampleRate',Fs,...
    'PassbandRipple',0.01, 'StopbandAttenuation',80);
tic
while toc < 10
    x = randn(256,1);
    y = lowpassFilt(x);
    scope(y);
end
```



Notice that as a convenience, specifications are entered directly using dB values. The passband ripple can be examined by selecting the "View" menu in FVTool and then selecting "Passband". `dsp.LowpassFilter` can also be used for IIR (biquad) designs.

```
fvtool(lowpassFilt,'Fs',Fs,'Color','White')
```



Obtaining the Filter Coefficients

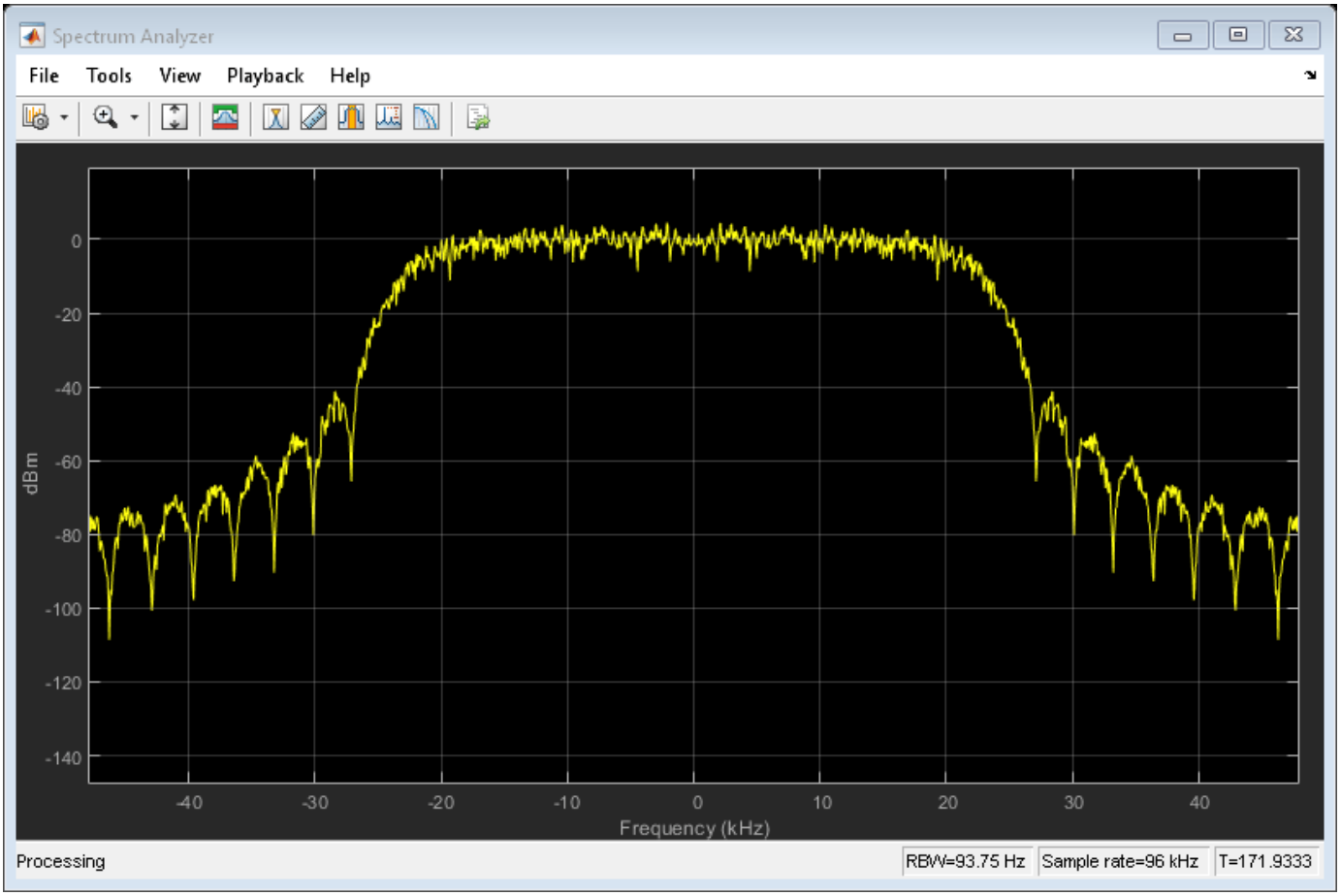
The filter coefficients can be extracted from `dsp.LowpassFilter` by using the `tf` function.

```
eqnum = tf(lowpassFilt);
```

Tunable Lowpass FIR Filters

Lowpass FIR filters in which the cutoff frequency can be tuned at run-time can be implemented using 'dsp.VariableBandwidthFIRFilter'. These filters do not provide the same granularity of control over the filter's response characteristic, but they do allow for dynamic frequency response.

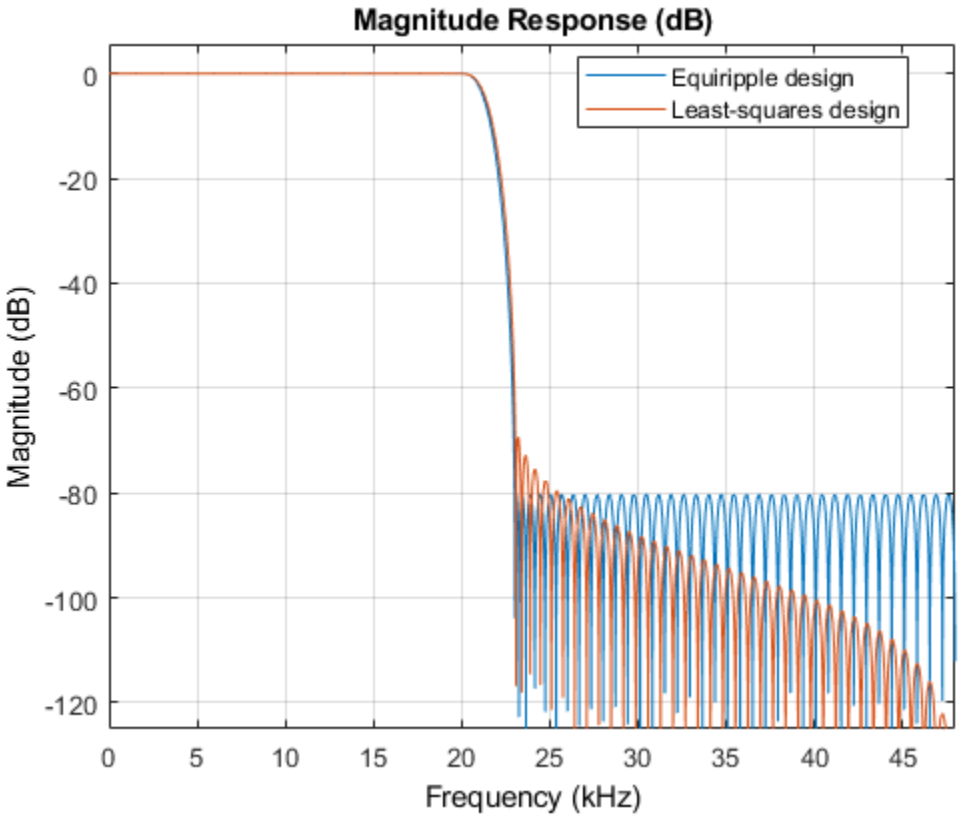
```
vbwFilter = dsp.VariableBandwidthFIRFilter('CutoffFrequency',1e3);
tic
told = 0;
while toc < 10
    t = toc;
    if floor(t) > told
        % Add 1 kHz every second
        vbwFilter.CutoffFrequency = vbwFilter.CutoffFrequency + 1e3;
    end
    x = randn(256,1);
    y = vbwFilter(x);
    scope(y);
    told = floor(t);
end
```



Advanced Design Options: Optimal Non-Equiripple Lowpass Filters

So far all designs used have been optimal equiripple designs. 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 lowpass 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. `fdesign.lowpass` can be used to design least-squares and other kinds of lowpass filters. The following code compares a least-squares FIR design to an FIR equiripple design with the same filter order and transition width:

```
lowpassSpec = fdesign.lowpass('N,Fp,Fst',133,Fp,Fst,Fs);
lsFIR = design(lowpassSpec,'firls','SystemObject',true);
LP_MIN = dsp.FIRFilter('Numerator',numMinOrder);
fvt = fvtool(LP_MIN,lsFIR,'Fs',Fs,'Color','White');
legend(fvt,'Equiripple design','Least-squares design')
```

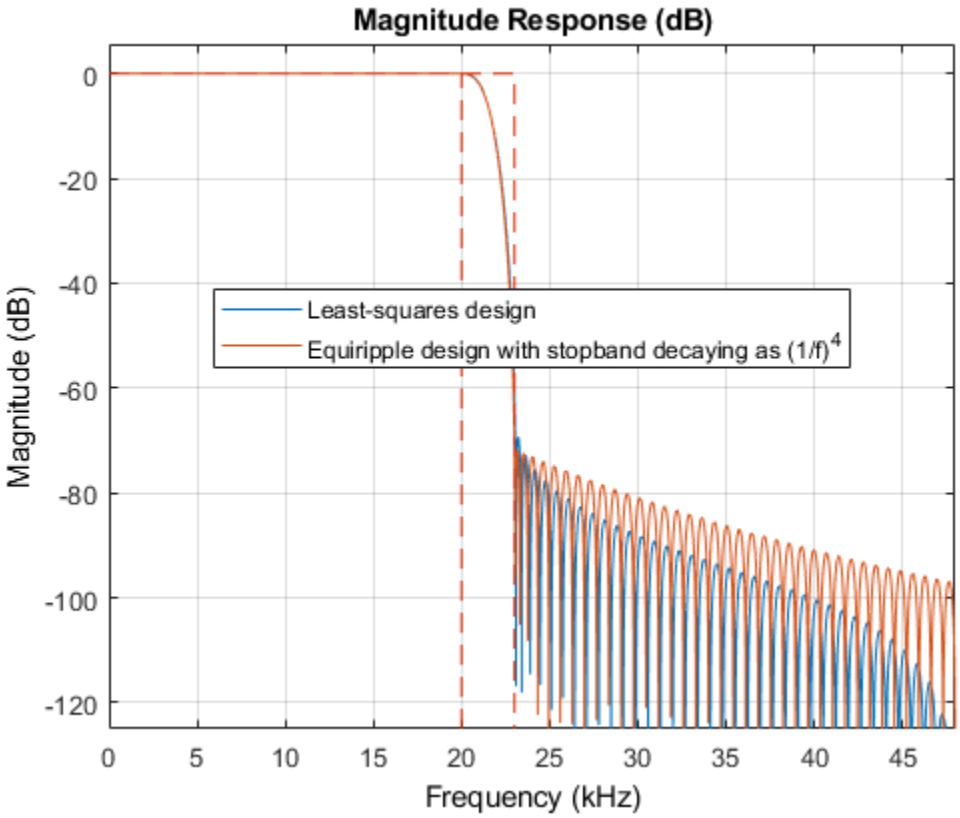


Notice how the attenuation in the stopband increases with frequency for the least-squares designs while it remains constant for the equiripple design. The increased attenuation in the least-squares case minimizes the energy in that band of the signal to be filtered.

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 lowpass 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, equiripple design options provide a way to achieve this.

```
FIR_eqrip_slope = design(lowpassSpec,'equiripple','StopbandShape','1/f',...
    'StopbandDecay',4,'SystemObject',true);
fvt = fvtool(lsFIR,FIR_eqrip_slope,'Fs',Fs,'Color','White');
legend(fvt,'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 in the vicinity of the passband-edge frequency, 20 kHz:

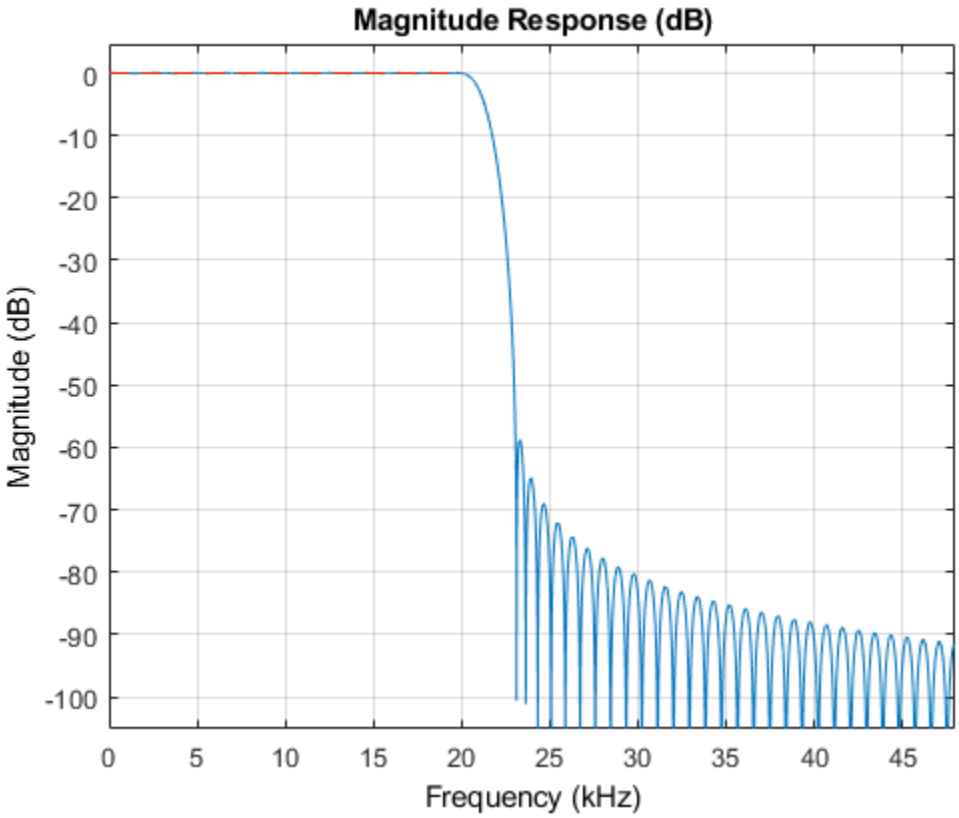
```
mls = measure(lsFIR);
meq = measure(FIR_eqrip_slope);
mls.Apass
meq.Apass
```

```
ans =

    0.0121


ans =

    0.0046
```

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. For more information on this and other arbitrary magnitude designs see “Arbitrary Magnitude Filter Design” on page 4-128.

```
B   = 2; % Number of bands
F   = [0 Fp linspace(Fst,Fs/2,40)];
A   = [1 1 zeros(1,length(F)-2)];
```

```
W   = linspace(1,100,length(F)-2);
lowpassArbSpec = fdesign.arbmag('N,B,F,A',N,B,F(1:2),A(1:2),F(3:end), ...
    A(3:end),Fs);
lpfilter = design(lowpassArbSpec,'equiripple','B2Weights',W, ...
    'SystemObject',true);
fvtool(lpfilter,'Fs',Fs,'Color','White');
```



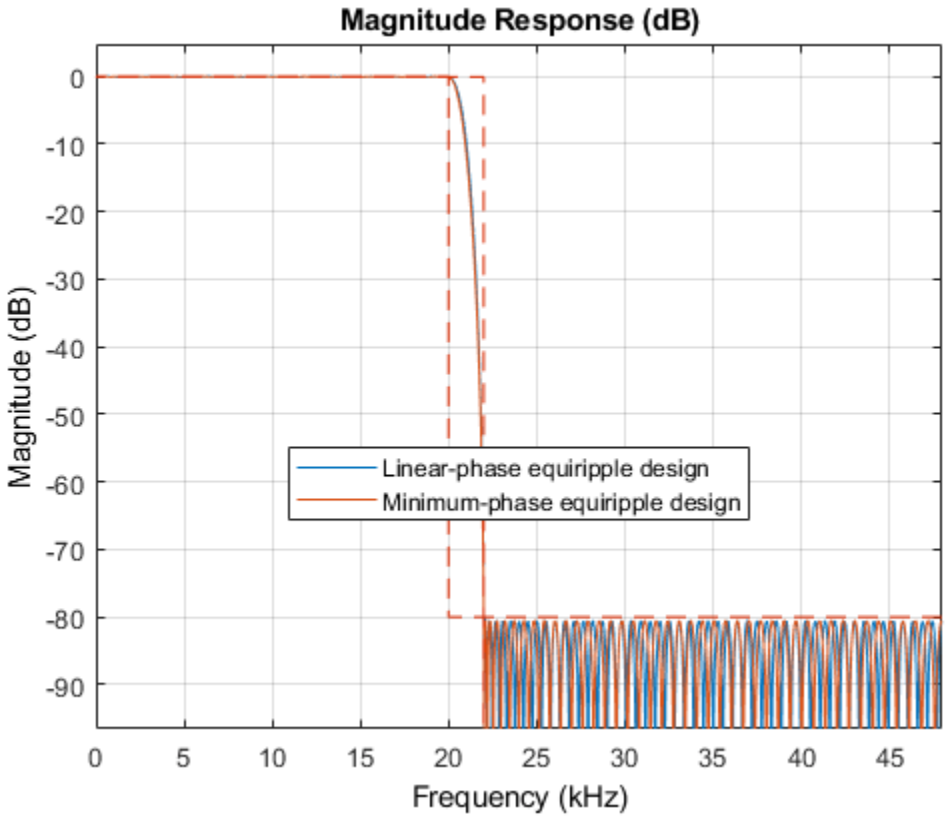
Minimum-Phase Lowpass Filter Design

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. However, minimum-phase designs are not always numerically robust. Always check the design with FVTool.

As an example of the advantages of minimum-phase designs, consider the comparison of a linear-phase design with a minimum-phase design that meets the same design specifications:

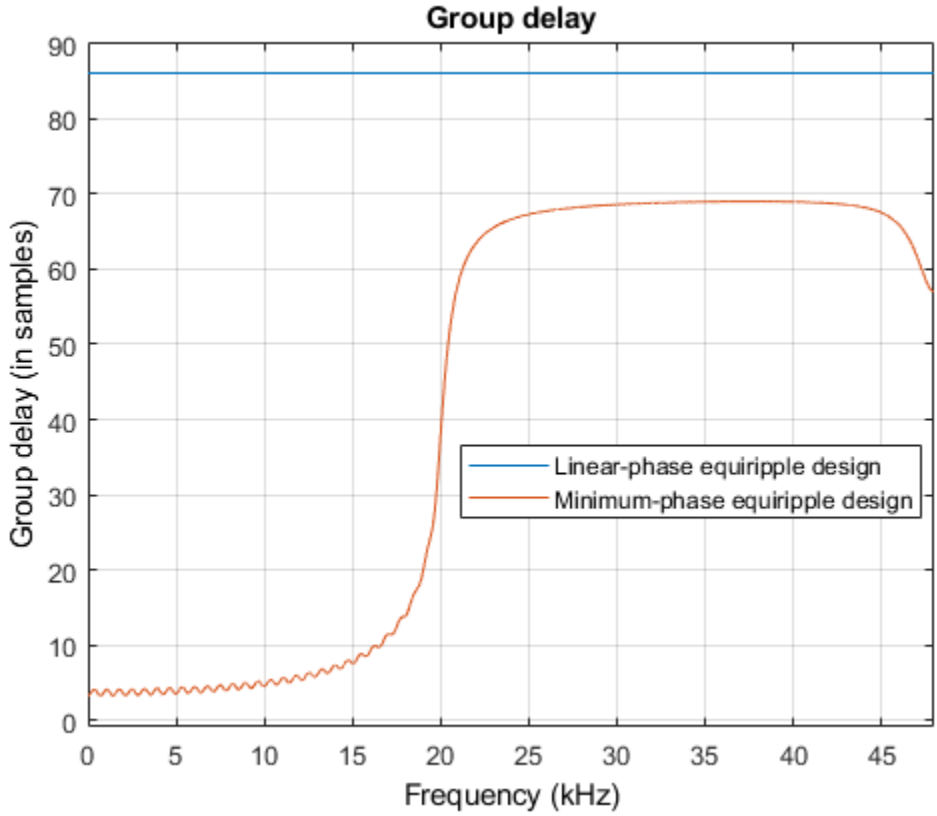
```
Fp  = 20e3;
Fst = 22e3;
Fs  = 96e3;
Ap  = 0.06;
Ast = 80;
lowpassSpec = fdesign.lowpass('Fp,Fst,Ap,Ast',Fp,Fst,Ap,Ast,Fs);
linphaseSpec =  design(lowpassSpec,'equiripple','SystemObject',true);
eqripSpec =  design(lowpassSpec,'equiripple','minphase',true,...
    'SystemObject',true);
fvt = fvtool(linphaseSpec,eqripSpec,'Fs',Fs,...
```

```
    'Color','White');
legend(fvt,...
    'Linear-phase equiripple design',...
    'Minimum-phase equiripple design')
```



Notice that the number of coefficients has been reduced from 173 to 141. The group-delay plot also reveals advantages of the minimum-phase design. Notice how the group-delay is much smaller (in particular in the passband region).

```
fvt = fvtool(linphaseSpec,eqripSpec,'Fs',Fs,...
    'Analysis','grpdelay','Color','White');
legend(fvt,...
    'Linear-phase equiripple design',...
    'Minimum-phase equiripple design')
```



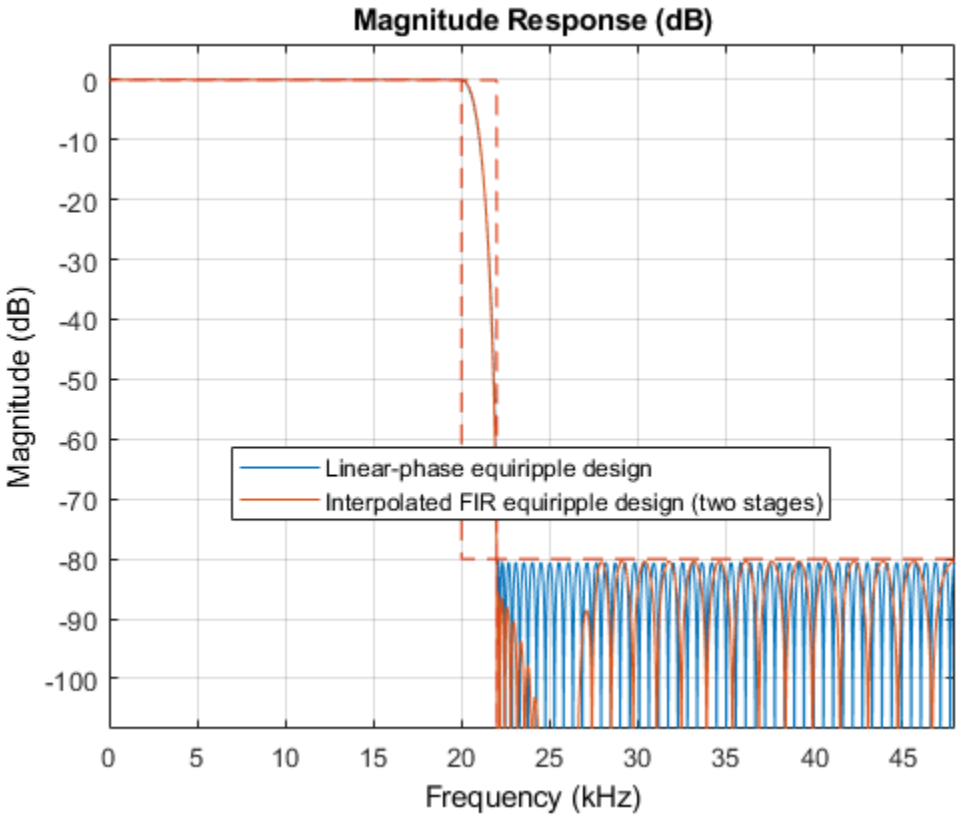
Minimum-Order Lowpass Filter Design Using Multistage Techniques

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. This approach breaks down the design problem into designing two filters in cascade. For this example, the design requires 151 coefficients rather than 173. For more information on this, see “Arbitrary Magnitude Filter Design” on page 4-128.

```
Fp  = 20e3;
Fst = 22e3;
Fs  = 96e3;
Ap  = 0.06;
Ast = 80;
lowpassSpec = fdesign.lowpass('Fp,Fst,Ap,Ast',Fp,Fst,Ap,Ast,Fs);
interpFilter = design(lowpassSpec,'ifir','SystemObject',true);
cost(interpFilter)
fvt = fvtool(linphaseSpec,interpFilter,'Fs',Fs,'Color','White');
legend(fvt,...
    'Linear-phase equiripple design',...
    'Interpolated FIR equiripple design (two stages)')
```

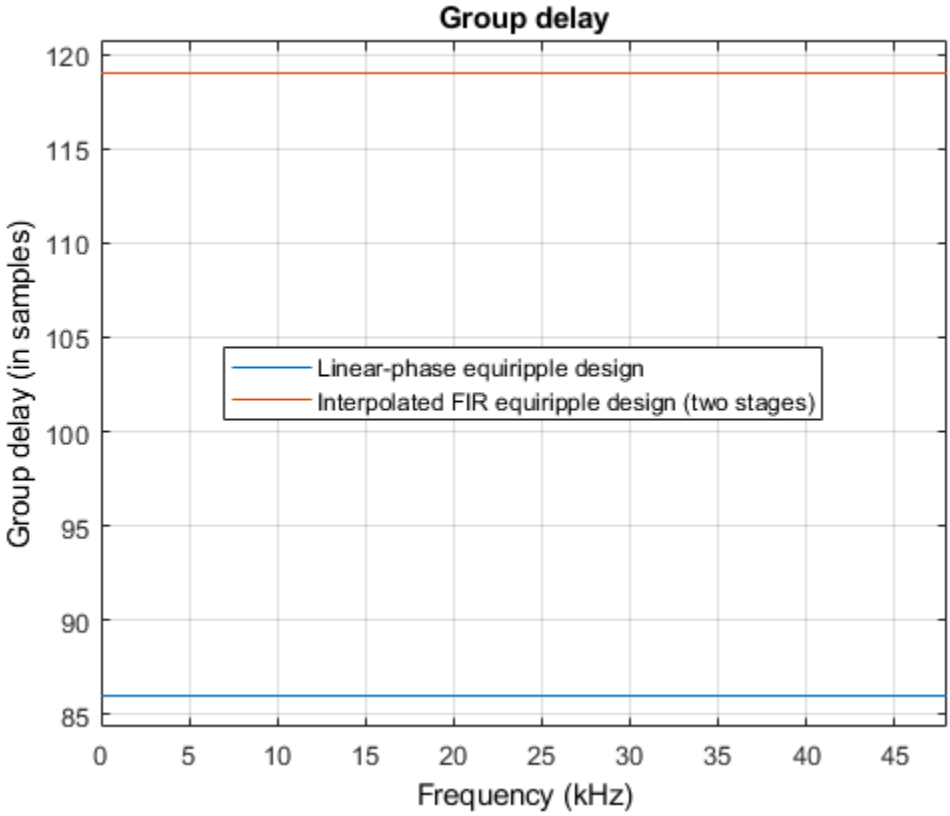
```
ans =

  struct with fields:

                 NumCoefficients: 151
```

```
                    NumStates: 238
MultiplicationsPerInputSample: 151
      AdditionsPerInputSample: 149
```



In this case, the group-delay plot reveals disadvantages of the IFIR design. While IFIR designs do provide linear phase, the group delay is in general larger than a comparable single-stage design.

```
fvt = fvtool(linphaseSpec,interpFilter,'Fs',Fs,'Analysis','grpdelay',...
    'Color','White');
legend(fvt,...
    'Linear-phase equiripple design',...
    'Interpolated FIR equiripple design (two stages)')
```



Lowpass Filter Design for Multirate Applications

Lowpass filters are extensively used in the design of decimators and interpolators. See “Design of Decimators and Interpolators” on page 4-211 for more information on this and “Multistage Rate Conversion” on page 4-177 for multistage techniques that result in very efficient implementations.

Classic IIR Filter Design

This example shows how to design classic IIR filters. The initial focus is on the situation for which the critical design parameter is the cutoff frequency at which the filter's power decays to half (-3 dB) the nominal passband value.

The example illustrates how easy it is to replace a Butterworth design with either a Chebyshev or an elliptic filter of the same order and obtain a steeper rolloff at the expense of some ripple in the passband and/or stopband of the filter. After this, minimum-order designs are explored.

Lowpass Filters

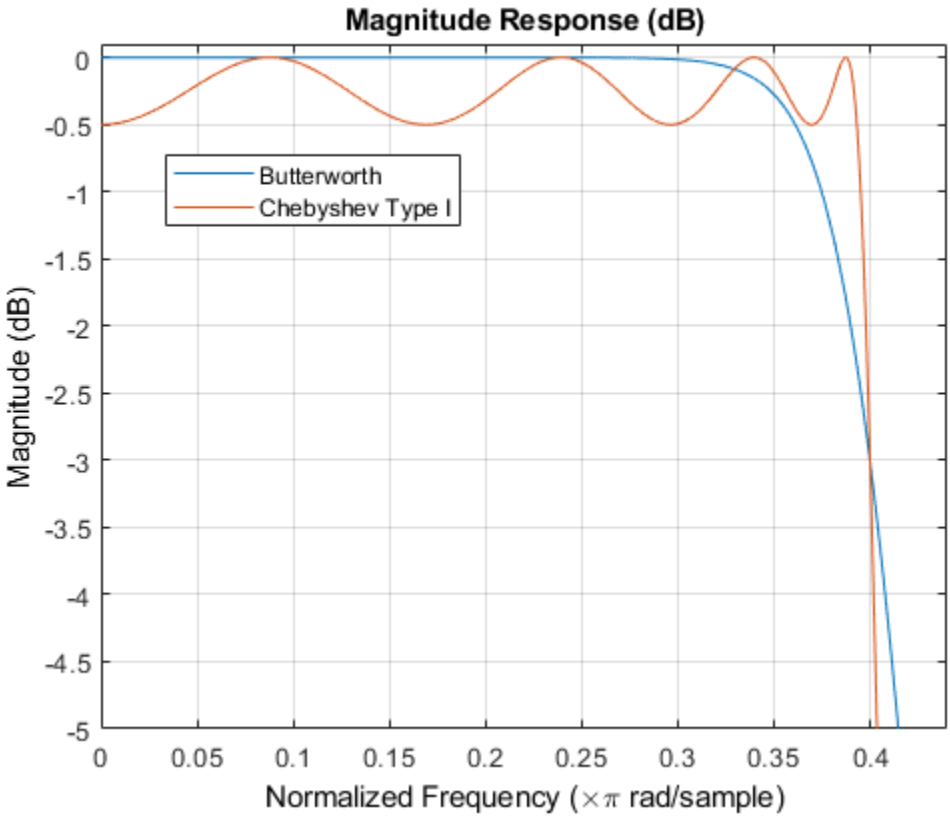
Let's design an 8th order filter with a normalized cutoff frequency of 0.4pi. First, we design a Butterworth filter which is maximally flat (no ripple in the passband or in the stopband):

```
N = 8;
F3dB = .4;
d = fdesign.lowpass('N,F3dB',N,F3dB);
Hbutter = design(d,'butter','SystemObject',true)
```

```
Hbutter =
  dsp.BiquadFilter with properties:

                  Structure: 'Direct form II'
            SOSMatrixSource: 'Property'
                  SOSMatrix: [4x6 double]
                ScaleValues: [5x1 double]
          InitialConditions: 0
    OptimizeUnityScaleValues: true

  Show all properties
```

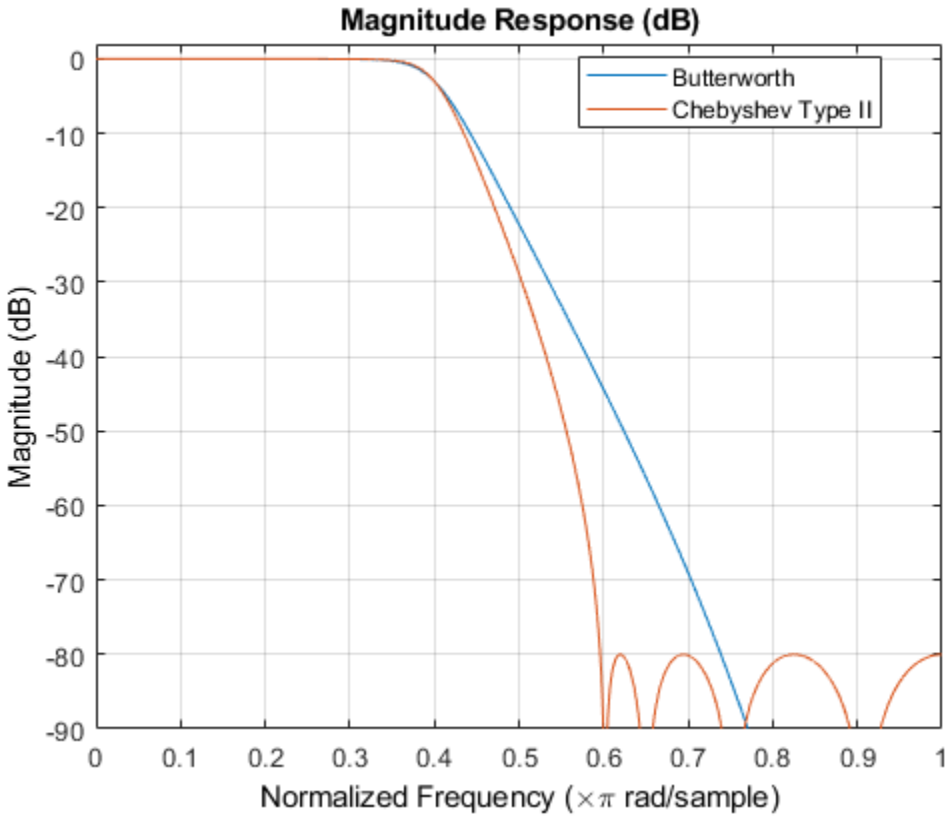
A Chebyshev Type I design allows for the control of ripples in the passband. There are still no ripples in the stopband. Larger ripples enable a steeper rolloff. Here, we specify peak-to-peak ripples of 0.5dB:

```
Ap = .5;
setspecs(d,'N,F3dB,Ap',N,F3dB,Ap);
Hcheby1 = design(d,'cheby1','SystemObject',true);
hfvt = fvtool(Hbutter,Hcheby1,'Color','white');
axis([0 .44 -5 .1])
legend(hfvt,'Butterworth','Chebyshev Type I');
```



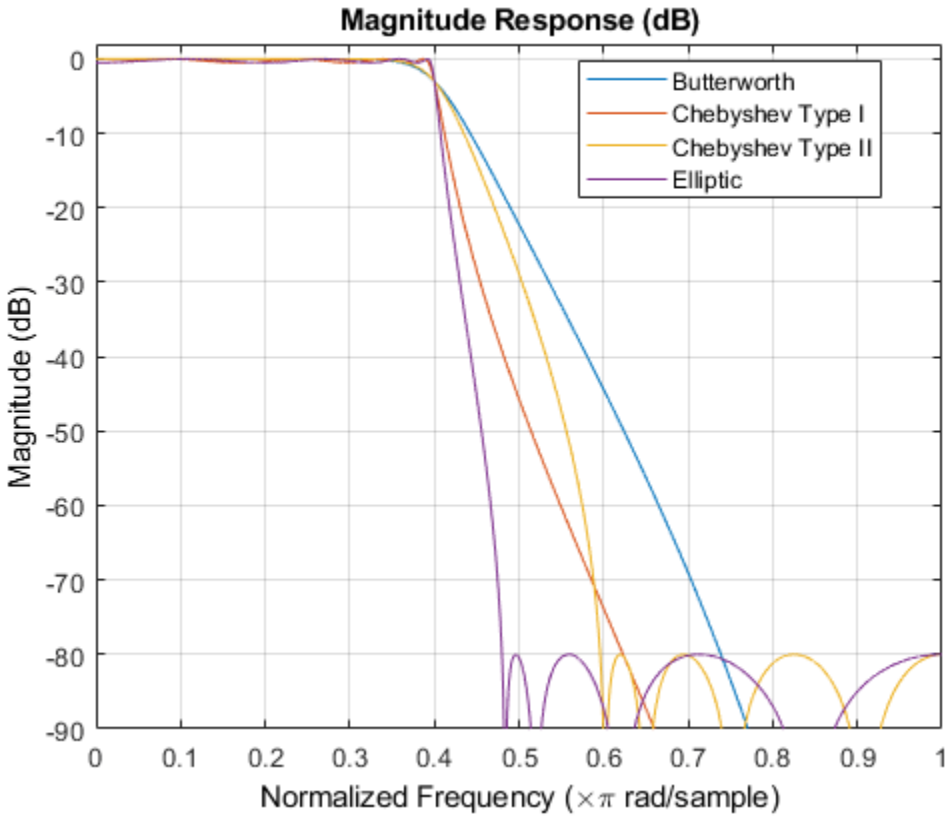
A Chebyshev Type II design allows for the control of the stopband attenuation. There are no ripples in the passband. A smaller stopband attenuation enables a steeper rolloff. In this example, we specify a stopband attenuation of 80 dB:

```
Ast = 80;
setspecs(d,'N,F3dB,Ast',N,F3dB,Ast);
Hcheby2 = design(d,'cheby2','SystemObject',true);
hfvt = fvtool(Hbutter,Hcheby2,'Color','white');
axis([0 1 -90 2])
legend(hfvt,'Butterworth','Chebyshev Type II');
```



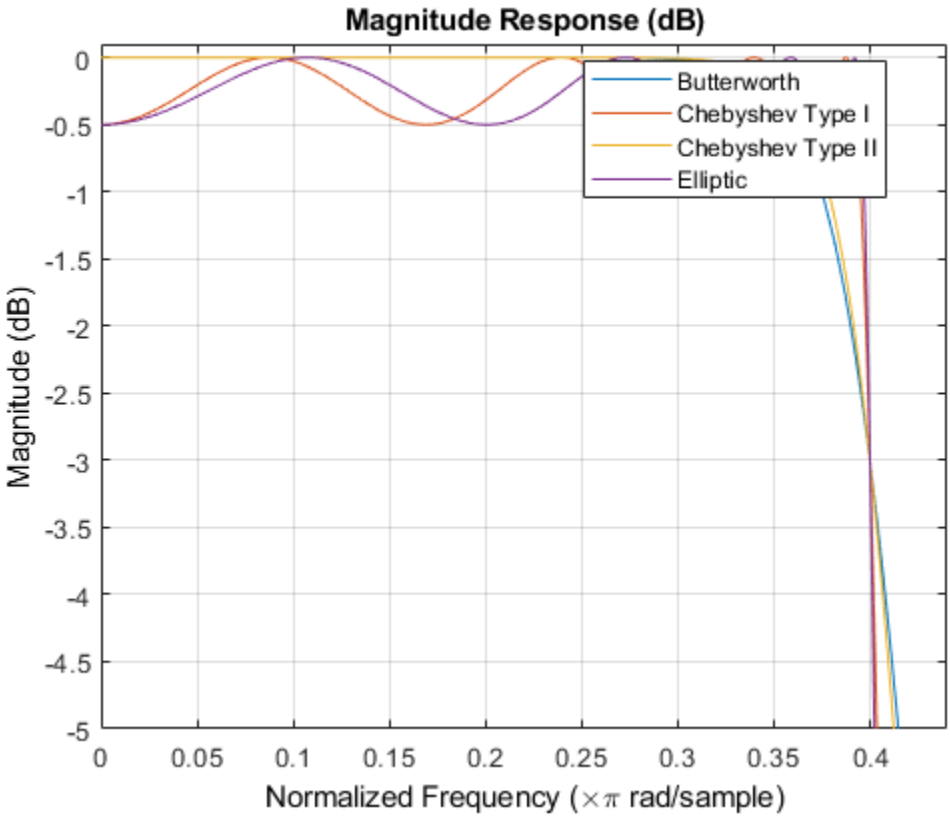
Finally, an elliptic filter can provide the steeper rolloff compared to previous designs by allowing ripples both in the stopband and the passband. To illustrate that, we reuse the same passband and stopband characteristic as above:

```
setspecs(d,'N,F3dB,Ap,Ast',N,F3dB,Ap,Ast);
Hellip = design(d,'ellip','SystemObject',true);
hfvt = fvtool(Hbutter,Hcheby1,Hcheby2,Hellip,'Color','white');
axis([0 1 -90 2])
legend(hfvt, ...
    'Butterworth','Chebyshev Type I','Chebyshev Type II','Elliptic');
```



By zooming in the passband, we verify that all filters have the same -3dB frequency point and that only Butterworth and Chebyshev Type II designs have a perfectly flat passband:

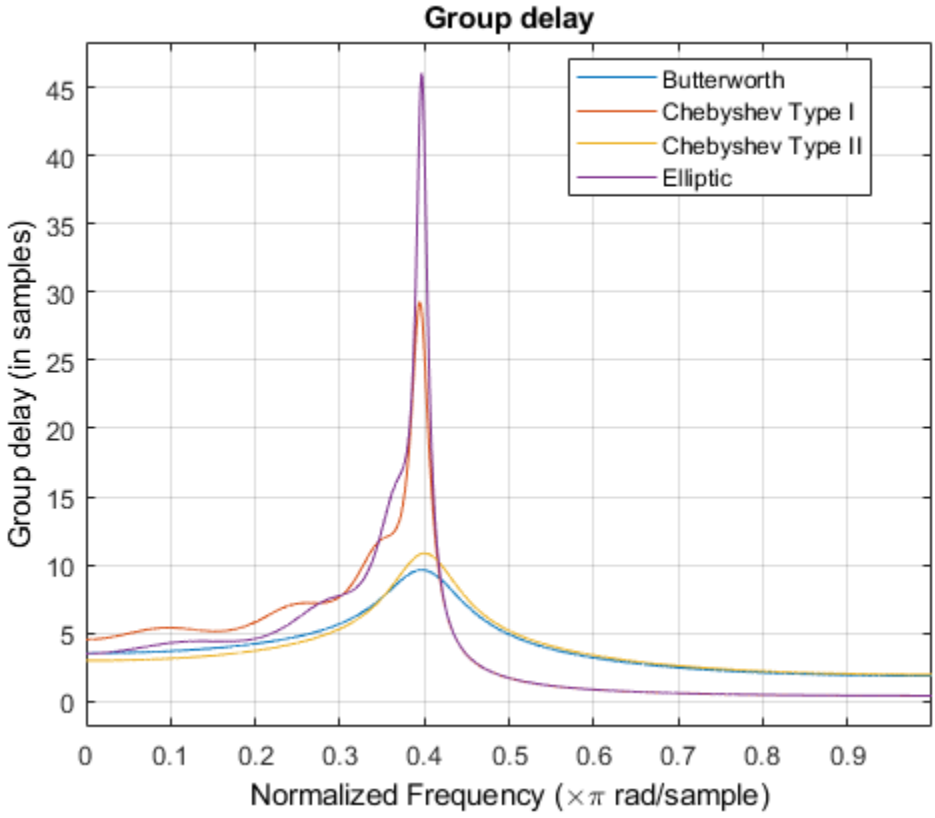
```
axis([0 .44 -5 .1])
```



Phase Consideration

If phase is an issue, it is useful to notice that Butterworth and Chebyshev Type II designs are also the ones introducing a lesser distortion (their group delay is flatter):

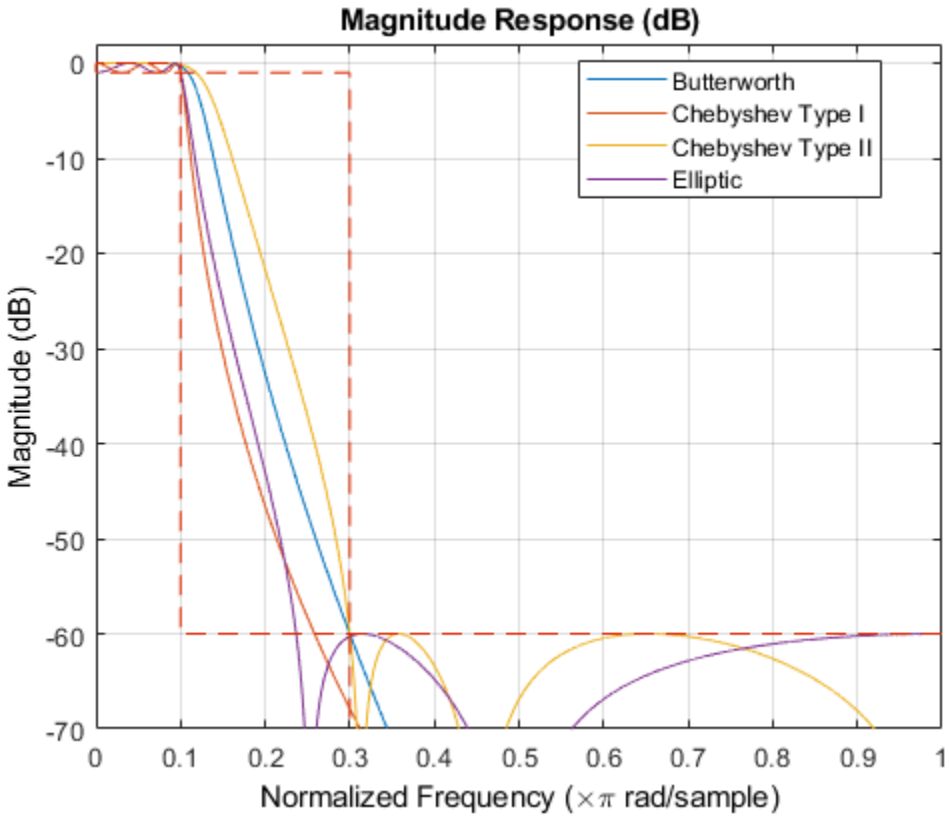
```
hfvt.Analysis = 'grpdelay';
```



Minimum Order Designs

In cases where the 3dB cutoff frequency is not of primary interest but instead both the passband and stopband are fully specified in terms of frequencies and the amount of tolerable ripples, we can use a minimum order design technique:

```
Fp = .1;
Fst = .3;
Ap = 1;
Ast = 60;
setspecs(d,'Fp,Fst,Ap,Ast',Fp,Fst,Ap,Ast);
Hbutter = design(d,'butter','SystemObject',true);
Hcheby1 = design(d,'cheby1','SystemObject',true);
Hcheby2 = design(d,'cheby2','SystemObject',true);
Hellip = design(d,'ellip','SystemObject',true);
hfvt = fvtool(Hbutter,Hcheby1,Hcheby2,Hellip, 'DesignMask', 'on',...
    'Color','white');
axis([0 1 -70 2])
legend(hfvt, ...
    'Butterworth','Chebyshev Type I','Chebyshev Type II','Elliptic');
```



A 7th order filter is necessary to meet the specification with a Butterworth design whereas a 5th order is sufficient with either Chebyshev techniques. The order of the filter can even be reduced to 4 with an elliptic design:

```
order(Hbutter)
```

```
ans = 7
```

```
order(Hcheby1)
```

```
ans = 5
```

```
order(Hcheby2)
```

```
ans = 5
```

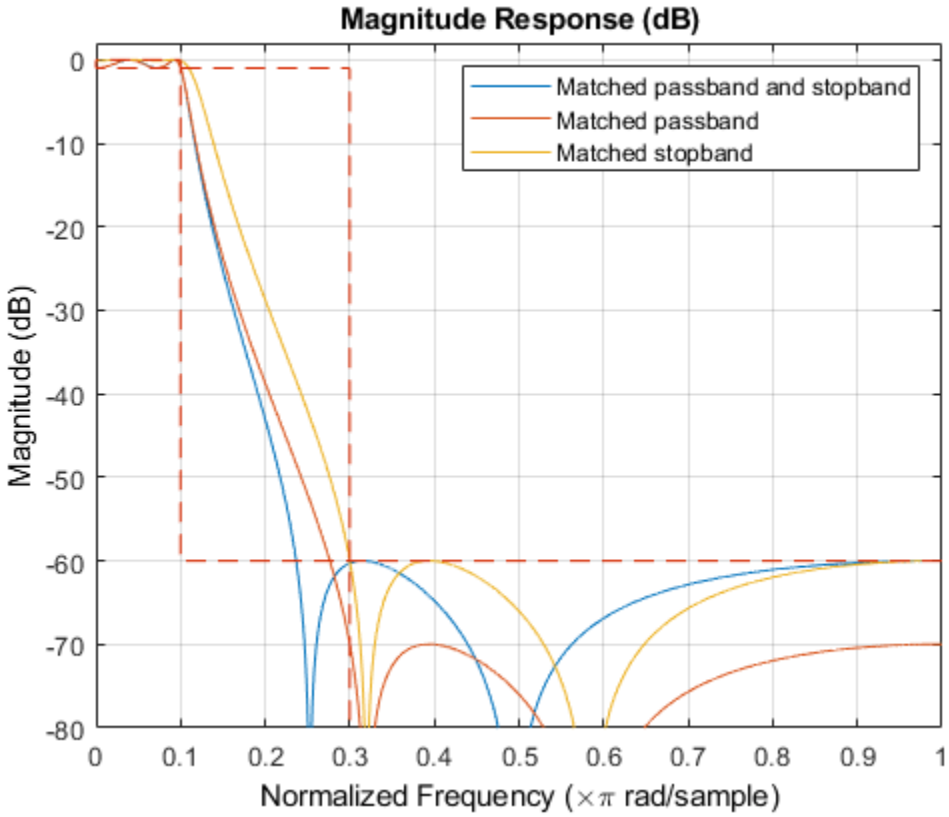
```
order(Hellip)
```

```
ans = 4
```

Matching Exactly the Passband or Stopband Specifications

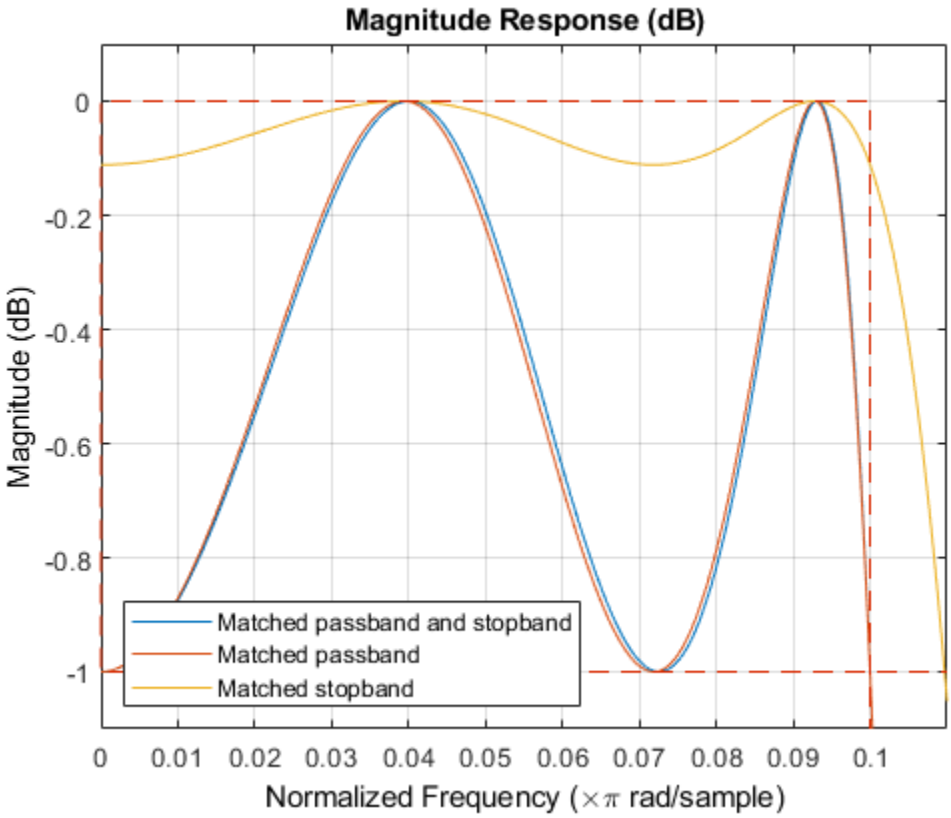
With minimum-order designs, the ideal order needs to be rounded to the next integer. This additional fractional order allows the algorithm to actually exceed the specifications. We can use the `MatchExactly` flag to constraint the design algorithm to match exactly one band. The other band will exceed its specifications. By default, Chebyshev Type I designs match the passband, Butterworth and Chebyshev Type II match the stopband and the attenuations of both bands are matched by the elliptic design (while the stopband edge frequency is exceeded):

```
Hellipmin1    = design(d, 'ellip', 'MatchExactly', 'passband',...
    'SystemObject',true);
Hellipmin2 = design(d, 'ellip', 'MatchExactly', 'stopband',...
    'SystemObject',true);
hfvt = fvtool(Hellip, Hellipmin1, Hellipmin2, 'DesignMask', 'on',...
    'Color','white');
axis([0 1 -80 2]);
legend(hfvt, 'Matched passband and stopband', ...
    'Matched passband', 'Matched stopband', ...
    'Location', 'Northeast')
```



Zoom in the passband to compare passband edges. The matched passband and matched both designs have an attenuation of exactly 1 dB at Fpass = .1:

```
axis([0 .11 -1.1 0.1]);
legend(hfvt, 'Location', 'Southwest')
```



We verify that the resulting order of the filters did not change:

```
order(Hellip)
```

```
ans = 4
```

```
order(Hellipmin1)
```

```
ans = 4
```

```
order(Hellipmin2)
```

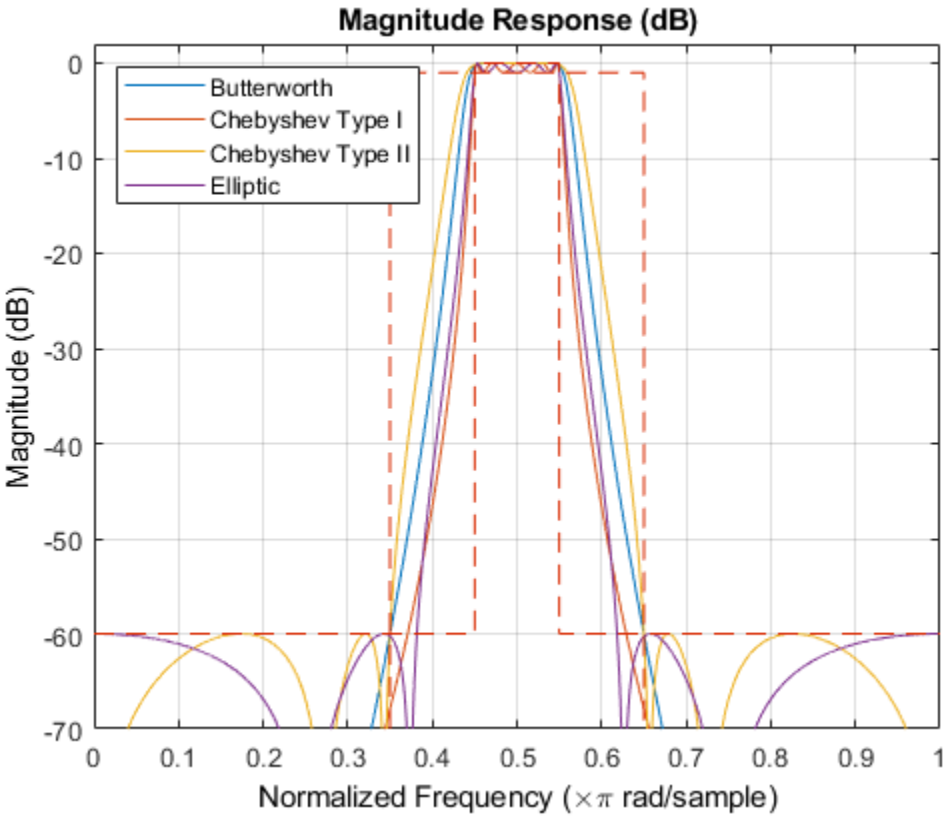
```
ans = 4
```

Highpass, Bandpass and Bandstop Filters

The results presented above can be extended to highpass, bandpass and bandstop response types. For example, here are minimum order bandpass filters:

```
d = fdesign.bandpass('Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2', ...
    .35,.45,.55,.65,60,1,60);

Hbutter = design(d,'butter','SystemObject',true);
Hcheby1 = design(d,'cheby1','SystemObject',true);
Hcheby2 = design(d,'cheby2','SystemObject',true);
Hellip = design(d,'ellip','SystemObject',true);
hfvt = fvtool(Hbutter,Hcheby1,Hcheby2,Hellip, 'DesignMask', 'on',...
    'Color','white');
axis([0 1 -70 2])
```

```
legend(hfvt, ...
    'Butterworth','Chebyshev Type I','Chebyshev Type II','Elliptic',...
    'Location', 'Northwest')
```



Efficient Narrow Transition-Band FIR Filter Design

This example shows how to design efficient FIR filters with very narrow transition-bands using multistage techniques. The techniques can be extended to the design of multirate filters. See “Multistage Rate Conversion” on page 4-177 for an example of that.

Design of a Lowpass Filter with Narrow Transition Bandwidth

One of the drawbacks of using FIR filters is that the filter order tends to grow inversely proportional to the transition bandwidth of the filter. Consider the following design specifications (where the ripples are given in linear units):

```
Fpass = 2866.5;  % Passband edge
Fstop = 3087;    % Stopband edge
Apass = 0.0174;  % Passband ripple, 0.0174 dB peak to peak
Astop = 66.0206; % Stopband ripple, 66.0206 dB of minimum attenuation
Fs    = 44.1e3;

lowpassSpec = fdesign.lowpass(Fpass,Fstop,Apass,Astop,Fs);
```

A regular linear-phase equiripple design that meets the specs can be designed with:

```
eqripFilter = design(lowpassSpec,'equiripple','SystemObject',true);
cost(eqripFilter)
```

```
ans = struct with fields:
                  NumCoefficients: 695
                        NumStates: 694
    MultiplicationsPerInputSample: 695
          AdditionsPerInputSample: 694
```

The filter length required turns out to be 694 taps.

Interpolated FIR (IFIR) Design

The IFIR design algorithm achieves an efficient design for the above specifications in the sense that it reduces the total number of multipliers required. To do this, the design problem is broken into two stages, a filter which is upsampled to achieve the stringent specifications without using many multipliers, and a filter which removes the images created when upsampling the previous filter.

```
interpFilter = design(lowpassSpec,'ifir','SystemObject',true);
```

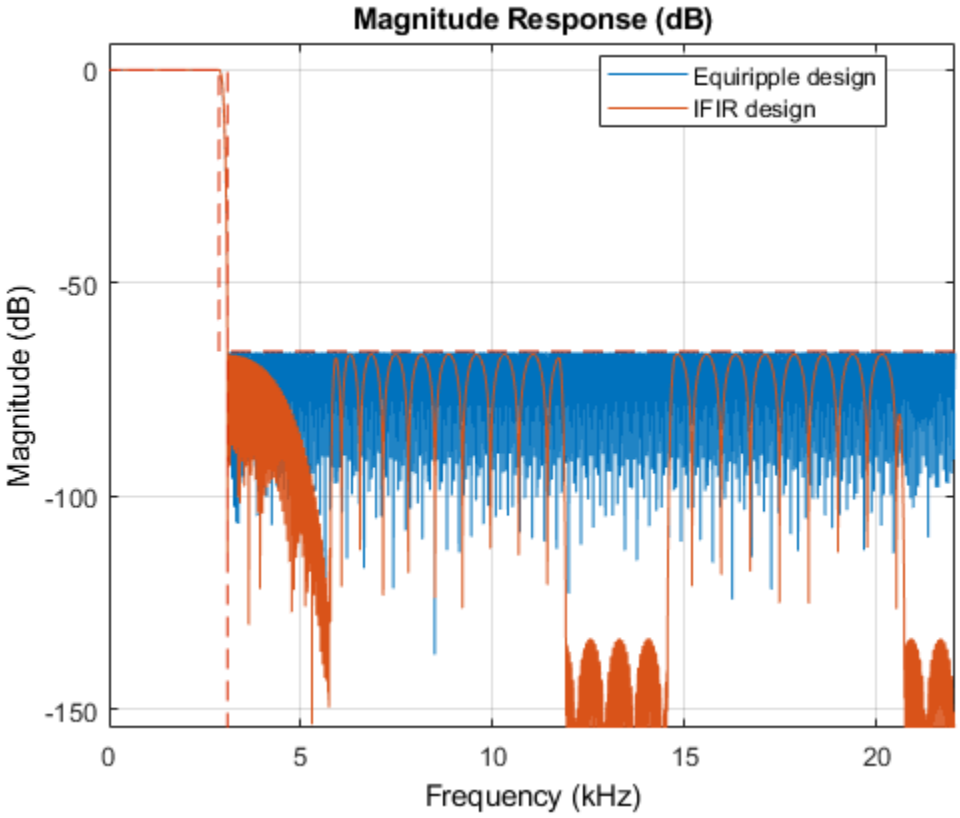
Apparently we have made things worse. Instead of a single filter with 694 multipliers, we now have two filters with a total of 804 multipliers. However, close examination of the second stage shows that only about one multiplier in 5 is non-zero. The actual total number of multipliers has been reduced from 694 to 208.

```
cost(interpFilter)
```

```
ans = struct with fields:
                  NumCoefficients: 208
                        NumStates: 802
    MultiplicationsPerInputSample: 208
          AdditionsPerInputSample: 206
```

Let's compare the responses of the two designs:

```
fvt = fvtool(eqripFilter,interpFilter,'Color','White');
legend(fvt,'Equiripple design', 'IFIR design','Location','Best')
```



Manually Controlling the Upsampling Factor

In the previous example, we automatically determined the upsampling factor used such that the total number of multipliers was minimized. It turned out that for the given specifications, the optimal upsampling factor was 5. However, if we examine the design options:

```
opts = designopts(lowpassSpec,'ifir')
```

```
opts = struct with fields:
      FilterStructure: 'dffir'
     UpsamplingFactor: 'auto'
    JointOptimization: 0
         SystemObject: 0
```

we can see that we can control the upsampling factor. For example, if we wanted to upsample by 4 rather than 5:

```
opts.UpsamplingFactor = 4;
opts.SystemObject = true;
upfilter = design(lowpassSpec,'ifir',opts);
cost(upfilter)
```

```
ans = struct with fields:
                  NumCoefficients: 217
```

```
                      NumStates: 767
    MultiplicationsPerInputSample: 217
          AdditionsPerInputSample: 215
```

We would obtain a design that has a total of 217 non-zero multipliers.

Using Joint Optimization

It is possible to design the two filters used in IFIR conjunctly. By doing so, we can save a significant number of multipliers at the expense of a longer design time (due to the nature of the algorithm, the design may also not converge altogether in some cases):

```
opts.UpsamplingFactor = 'auto'; % Automatically determine the best factor
opts.JointOptimization = true;
ifirFilter = design(lowpassSpec,'ifir',opts);
cost(ifirFilter)
```

```
ans = struct with fields:
                  NumCoefficients: 172
                        NumStates: 726
    MultiplicationsPerInputSample: 172
          AdditionsPerInputSample: 170
```

For this design, the best upsampling factor found was 6. The number of non-zero multipliers is now only 152

Using Multirate/Multistage Techniques to Achieve Efficient Designs

For the designs discussed so far, single-rate techniques have been used. This means that the number of multiplications required per input sample (MPIS) is equal to the number of non-zero multipliers. For instance, the last design we showed requires 152 MPIS. The single-stage equiripple design we started with required 694 MPIS.

By using multirate/multistage techniques which combine decimation and interpolation we can also obtain efficient designs with a low number of MPIS. For decimators, the number of multiplications required per input sample (on average) is given by the number of multipliers divided by the decimation factor.

```
multistageFilter = design(lowpassSpec,'multistage','SystemObject',true)
```

```
multistageFilter =
  dsp.FilterCascade with properties:

    Stage1: [1x1 dsp.FIRDecimator]
    Stage2: [1x1 dsp.FIRDecimator]
    Stage3: [1x1 dsp.FIRDecimator]
    Stage4: [1x1 dsp.FIRInterpolator]
    Stage5: [1x1 dsp.FIRInterpolator]
    Stage6: [1x1 dsp.FIRInterpolator]
```

```
cost(multistageFilter)
```

```
ans = struct with fields:
                  NumCoefficients: 396
                        NumStates: 352
```

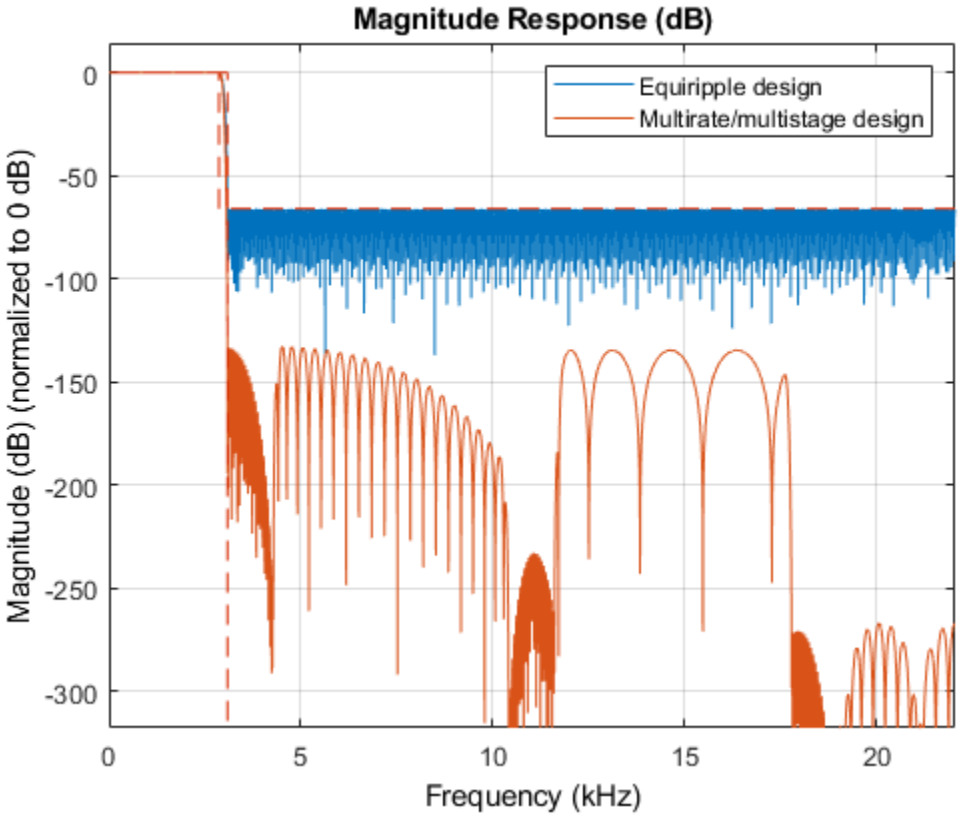
```
MultiplicationsPerInputSample: 73
      AdditionsPerInputSample: 70.8333
```

The first stage has 21 multipliers, and a decimation factor of 3. Therefore, the number of MPIS is 7. The second stage has a length of 45 and a cumulative decimation factor of 6 (that is the decimation factor of this stage multiplied by the decimation factor of the first stage; this is because the input samples to this stage are already coming in at a rate 1/3 the rate of the input samples to the first stage). The average number of multiplications per input sample (reference to the input of the overall multirate/multistage filter) is thus 45/6=7.5. Finally, given that the third stage has a decimation factor of 1, the average number of multiplications per input for this stage is 130/6=21.667. The total number of average MPIS for the three decimators is 36.5.

For the interpolators, it turns out that the filters are identical to the decimators. Moreover, their computational cost is the same. Therefore the total number of MPIS for the entire multirate/multistage design is 73.

Now we compare the responses of the equiripple design and this one:

```
fvt = fvtool(eqripFilter,multistageFilter,'Color','White', ...
    'NormalizeMagnitudeto1','on');
legend(fvt,'Equiripple design', 'Multirate/multistage design', ...
    'Location','NorthEast')
```



Notice that the stopband attenuation for the multistage design is about double that of the other designs. This is because it is necessary for the decimators to attenuate out of band components by

the required 66 dB in order to avoid aliasing that would violate the required specifications. Similarly, the interpolators need to attenuate images by 66 dB in order to meet the specifications of this problem.

Manually Controlling the Number of Stages

The multirate/multistage design that was obtained consisted of 6 stages. The number of stages is determined automatically by default. However, it is also possible to manually control the number of stages that result. For example:

```
lp4stage=design(lowpassSpec,'multistage','NStages',4,'SystemObject',true);
cost(lp4stage)
```

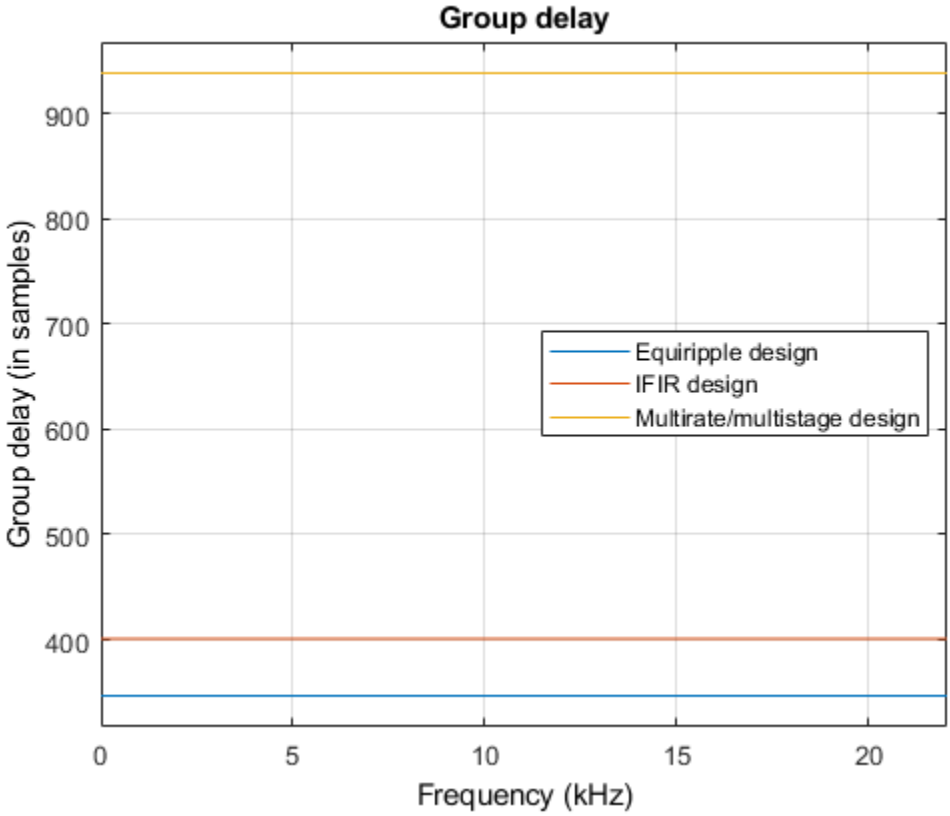
```
ans = struct with fields:
                  NumCoefficients: 516
                        NumStates: 402
    MultiplicationsPerInputSample: 86
          AdditionsPerInputSample: 84.5000
```

The average number of MPIS for this case is 86

Group Delay

We can compute the group delay for each design. Notice that the multirate/multistage design introduces the most delay (this is the price to pay for a less computationally expensive design). The IFIR design introduces more delay than the single-stage equiripple design, but less so than the multirate/multistage design.

```
fvt = fvtool(eqripFilter,interpFilter,multistageFilter,...
    'Analysis','grpdelay','Color','White');
legend(fvt, 'Equiripple design','IFIR design',...
    'Multirate/multistage design');
```

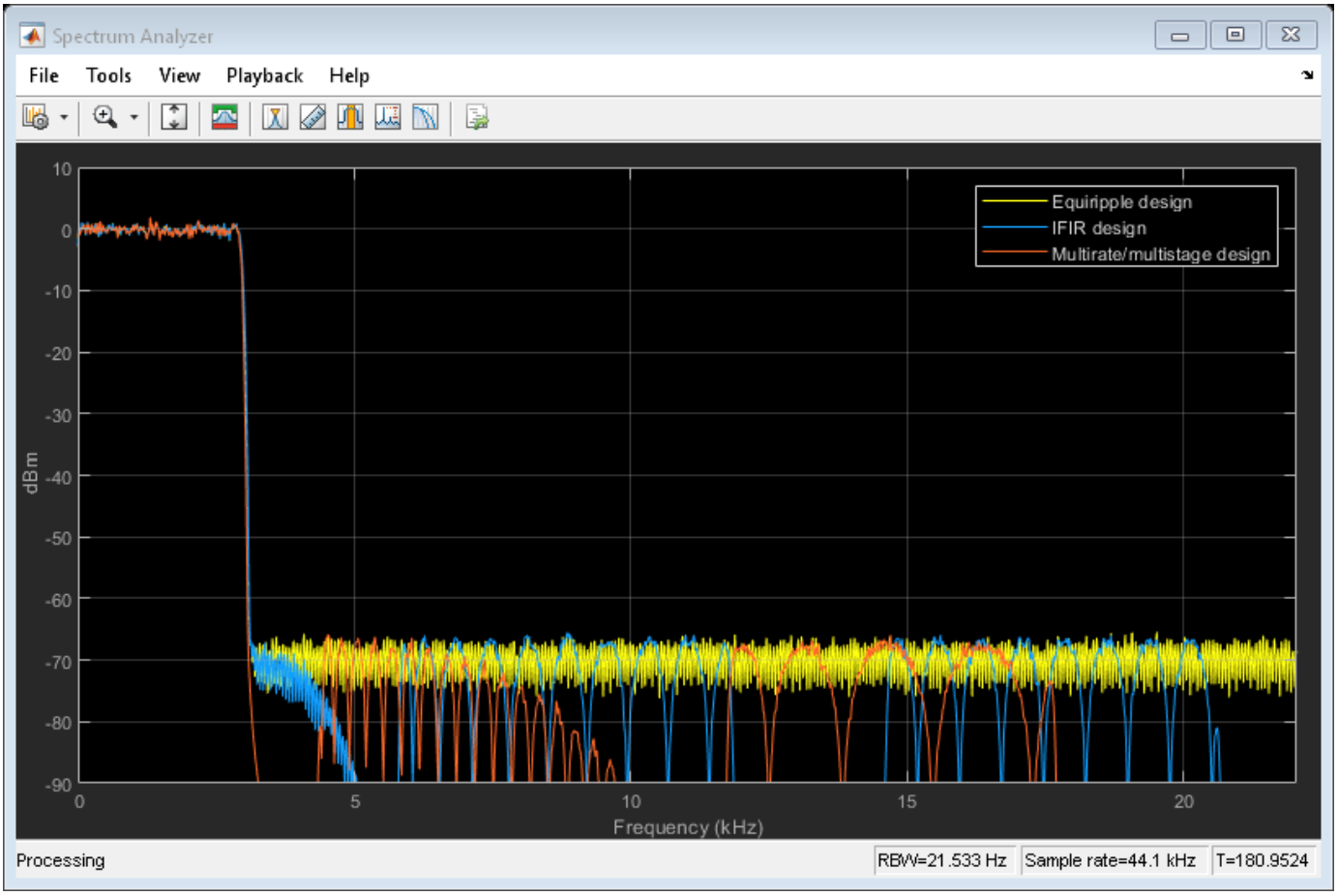


Filtering a Signal

We now show by example that the IFIR and multistage/multirate design perform comparably to the single-stage equiripple design while requiring much less computation. In order to perform filtering for cascades, it is necessary to call generateFilteringCode(ifirFilter) and generateFilteringCode(multistageFilter). This has been done here already in order to create HelperIFIRFilter and HelperMultiFIRFilter.

```
scope = dsp.SpectrumAnalyzer('SpectralAverages',50,'SampleRate',Fs,...
    'PlotAsTwoSidedSpectrum',false,'YLimits', [-90 10],...
    'ShowLegend',true','ChannelNames',{'Equiripple design',...
    'IFIR design','Multirate/multistage design'});
tic,
while toc < 20
    % Run for 20 seconds
    x = randn(6000,1);

    % Filter using single stage FIR filter
    y1 = eqripFilter(x);

    % Filter using IFIR filter
    y2 = HelperIFIRFilter(x);

    % Filter multi-stage/multi-rate FIR filters
    y3 = HelperMultiFIRFilter(x);

    % Compare the output from both approaches
```

```
        scope([y1,y2,y3])
    end
```



IIR Filter Design Given a Prescribed Group Delay

This example shows how to design arbitrary group delay filters using the `fdesign.arbgrpdelay` filter designer. This designer uses a least-Pth constrained optimization algorithm to design allpass IIR filters that meet a prescribed group delay.

`fdesign.arbgrpdelay` can be used for group delay equalization.

Arbitrary Group Delay Filter Designer

You can use fdesign.arbgrpdelay to design an allpass filter with a desired group delay response. The desired group delay is specified in a relative sense. The actual group delay depends on the filter order (the higher the order, the higher the delay). However, if you subtract the offset in the group delay due to the filter order, the group delay of the designed filter tends to match the desired group delay. The following code provides an example using two different filter orders.

```
N = 8;          % Filter order
N2 = 10;        % Alternate filter order
F = [0 0.1 1]; % Frequency vector
Gd = [2 3 1];  % Desired group delay
R = 0.99;       % Pole-radius constraint
```

Note that in an allpass filter, the numerator is always the reversed version of its denominator. For this reason, you cannot specify different numerator and denominator orders in fdesign.arbgrpdelay.

The following code shows a single band arbitrary group delay design with the desired group delay values, Gd, at the specified frequency points, F. In single band designs you specify the group delay over frequency values that cover the entire Nyquist interval [0 1]*pi rad/sample.

```
arbGrpSpec = fdesign.arbgrpdelay('N,F,Gd',N,F,Gd)
```

```
arbGrpSpec =
  arbgrpdelay with properties:

               Response: 'Arbitrary Group Delay'
          Specification: 'N,F,Gd'
            Description: {3x1 cell}
    NormalizedFrequency: 1
            FilterOrder: 8
            Frequencies: [0 0.1000 1]
             GroupDelay: [2 3 1]
```

```
arbGrpDelFilter1 = design(arbGrpSpec,'MaxPoleRadius',R, ...
    'SystemObject', true)
```

```
arbGrpDelFilter1 =
  dsp.BiquadFilter with properties:

                  Structure: 'Direct form II'
             SOSMatrixSource: 'Property'
                   SOSMatrix: [4x6 double]
                 ScaleValues: [5x1 double]
           InitialConditions: 0
    OptimizeUnityScaleValues: true
```

```
  Show all properties
```

Measure the total group delay at a set of frequency points from 0 to 1. Measure the nominal group delay of the filter using the measure method.

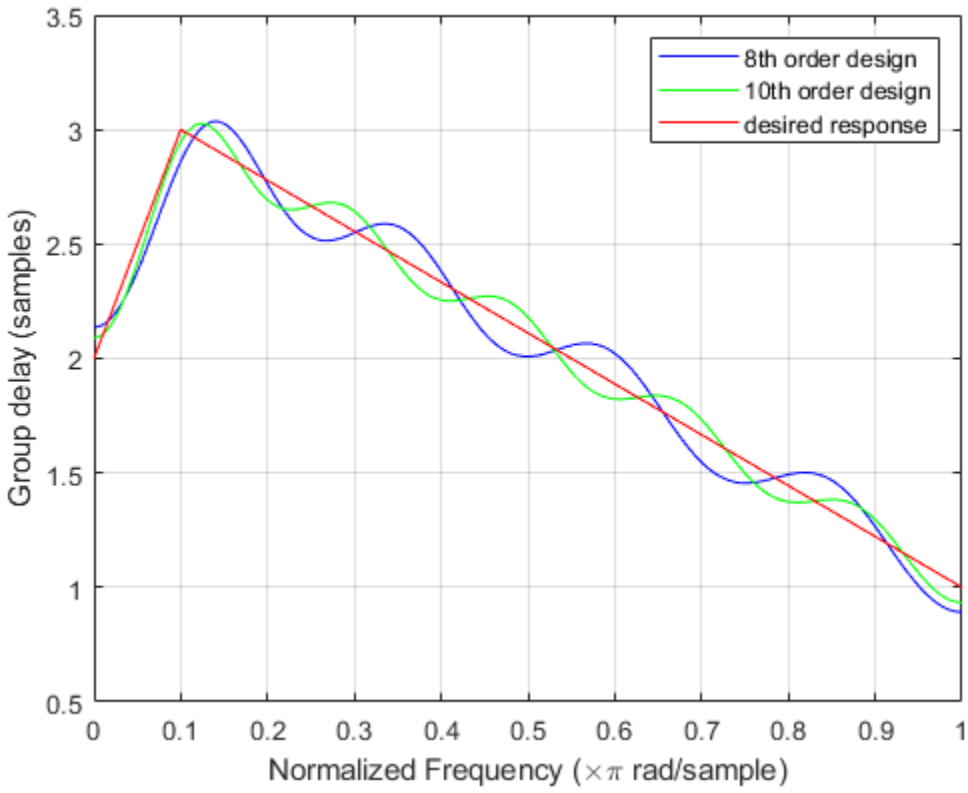
```
Fpoints = 0:0.001:1;
M1 = measure(arbGrpSpec,arbGrpDelFilter1,Fpoints);
```

Design another filter with an order equal to N2.

```
arbGrpSpec.FilterOrder = N2;
arbGrpDelFilter2 = design(arbGrpSpec,'MaxPoleRadius',R, ...
    'SystemObject', true);
M2 = measure(arbGrpSpec,arbGrpDelFilter2,Fpoints);
```

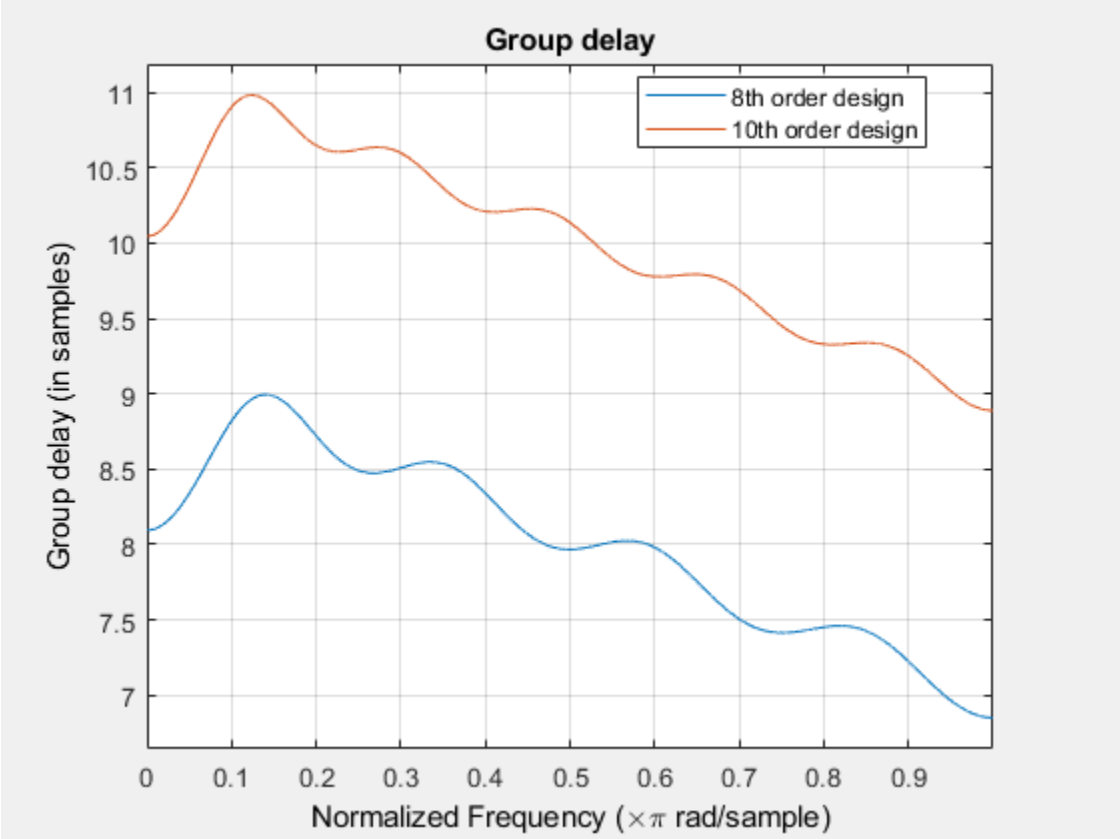
Plot the measured total group delay minus the nominal group delay.

```
plot(Fpoints, M1.TotalGroupDelay-M1.NomGrpDelay, 'b',...
     Fpoints, M2.TotalGroupDelay-M2.NomGrpDelay, 'g',...
     [0 0.1 1], [2 3 1], 'r');
xlabel('Normalized Frequency (\times\pi rad/sample)');
ylabel('Group delay (samples)'); grid on;
legend('8th order design','10th order design','desired response')
```



The following plot shows that the actual group delay of the two designs is different. The significance of this result is that one must find a compromise between a better fit to the desired relative group delay (less ripple) and a larger overall delay in the filter.

```
fvt = fvtool(arbGrpDelFilter1,arbGrpDelFilter2,'Analysis', 'grpdelay');
legend(fvt, '8th order design','10th order design')
```



Passband Group Delay Equalization

The primary use of fdesign.arbgrpdelay is to compensate for nonlinear-phase responses of IIR filters. Since the compensating filter is allpass, it can be cascaded with the filter you want to compensate without affecting its magnitude response. Since the cascade of the two filters is an IIR filter itself, it cannot have linear-phase (while being stable). However, it is possible to have approximately a linear phase response in the passband of the overall filter.

Lowpass Equalization

The following example uses fdesign.arbgrpdelay to equalize the group delay response of a lowpass elliptic filter without affecting its magnitude response.

You use a multiband design to specify desired group delay values over one or more bands of interest while leaving the group delay of all other frequency bands unspecified (don't care regions). In this example there is only one band of interest which equals the passband of the lowpass filter. You want to compensate the group delay in this band, and do not care about the resulting group delay values outside of it.

Design an elliptic filter with a passband frequency of 0.2π rad/sample. Measure the total group delay over the passband.

```
ellipFilter = design(fdesign.lowpass('N,Fp,Ap,Ast',4,0.2,1,40),...
    'ellip', 'SystemObject', true);
```

```
wncomp = 0:0.001:0.2;
g = grpdelay(ellipFilter,wncomp,2); % samples
g1 = max(g)-g;
```

Design an 8th order arbitrary group delay allpass filter. Use a multiband design and specify a single band.

```
allpassSpec = fdesign.arbgrpdelay('N,B,F,Gd',8,1,wncomp,g1)
```

```
allpassSpec =
  arbgrpdelay with properties:

              Response: 'Arbitrary Group Delay'
         Specification: 'N,B,F,Gd'
           Description: {4x1 cell}
    NormalizedFrequency: 1
           FilterOrder: 8
                NBands: 1
          B1Frequencies: [0 1.0000e-03 0.0020 0.0030 0.0040 0.0050 ... ]
           B1GroupDelay: [14.5838 14.5835 14.5824 14.5808 14.5784 ... ]
```

```
allpassFilter = design(allpassSpec,'iirlpnorm', 'SystemObject', true)
```

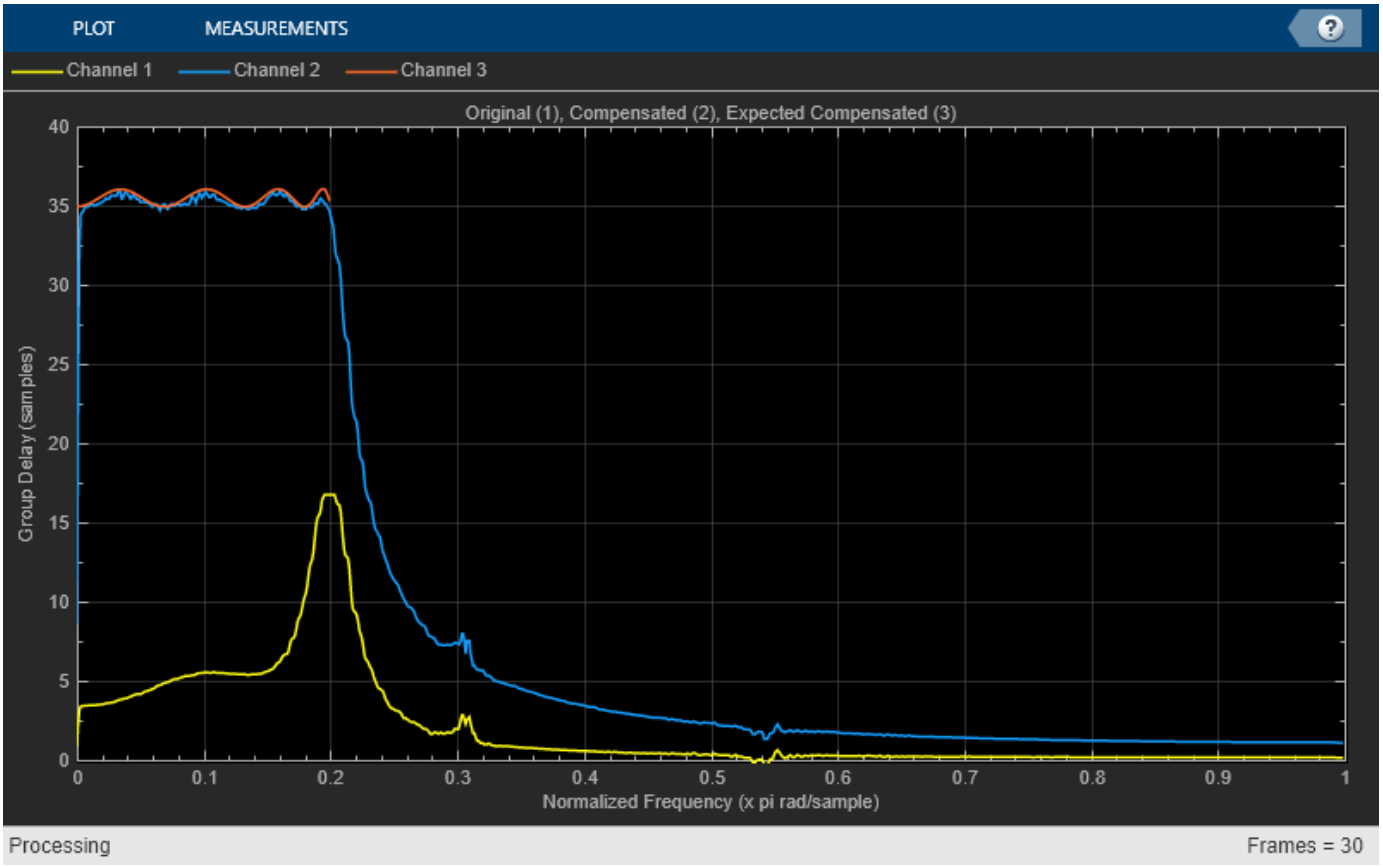
```
allpassFilter =
  dsp.BiquadFilter with properties:

                 Structure: 'Direct form II'
            SOSMatrixSource: 'Property'
                  SOSMatrix: [4x6 double]
                ScaleValues: [5x1 double]
          InitialConditions: 0
    OptimizeUnityScaleValues: true

  Show all properties
```

Cascade the original filter with the compensation filter to achieve the desired group delay equalization. Verify by processing white noise and estimating the group delay at the two output stages

```
samplesPerFrame = 2048;
wn = (2/samplesPerFrame) * (0:samplesPerFrame-1);
numRealPoints = samplesPerFrame/2 + 1;
tfEstimator = dsp.TransferFunctionEstimator('FrequencyRange','onesided',...
      'SpectralAverages',64);
scope = dsp.ArrayPlot('PlotType','Line','YLimits',[0 40],...
      'YLabel','Group Delay (samples)',...
      'XLabel','Normalized Frequency (x pi rad/sample)',...
      'SampleIncrement',2/samplesPerFrame,...
      'Title',['Original (1), Compensated (2), ',...
      'Expected Compensated (3)'], 'ShowLegend', true);

gdOrig = grpdelay(ellipFilter, numRealPoints);
gdComp = grpdelay(allpassFilter, numRealPoints);
range = wn < wncomp(end);
gdExp = nan(numRealPoints, 1);
gdExp(range) = gdOrig(range) + gdComp(range);
```

Stream random samples through filter cascade

```
Nframes = 300;
for k = 1:Nframes
    x = randn(samplesPerFrame,1);  % Input signal = white Gaussian noise

    y_orig = ellipFilter(x);       % Filter noise with original IIR filter
    y_corr = allpassFilter(y_orig);% Compensating filter

    Txy = tfEstimator([x, x],[y_orig, y_corr]);
    gdMeas = HelperMeasureGroupDelay(Txy, [], 20);
    scope([gdMeas, gdExp]);
end
```



Bandpass Equalization

Design a passband group delay equalizer for a bandpass Chebyshev filter with a passband region in the [0.3 0.4]*pi rad/sample interval. As in the previous example, there is only one band of interest which corresponds to the passband of the filter. Because you want to compensate the group delay in this band and do not care about the resulting group delay values outside of it, you use a multiband design and specify a single band.

Design a bandpass Chebyshev type-1 filter and measure its total group delay over the passband.

```
bandpassFilter = design(fdesign.bandpass('N,Fp1,Fp2,Ap',4,0.3,0.4,1), ...
    'cheby1', 'SystemObject', true);
wncomp = 0.3:0.001:0.4;
```

```
g = grpdelay(bandpassFilter,wncomp,2);
g1 = max(g)-g;
```

Design an 8th order arbitrary group delay filter. The pole radius is constrained to not exceed 0.95.

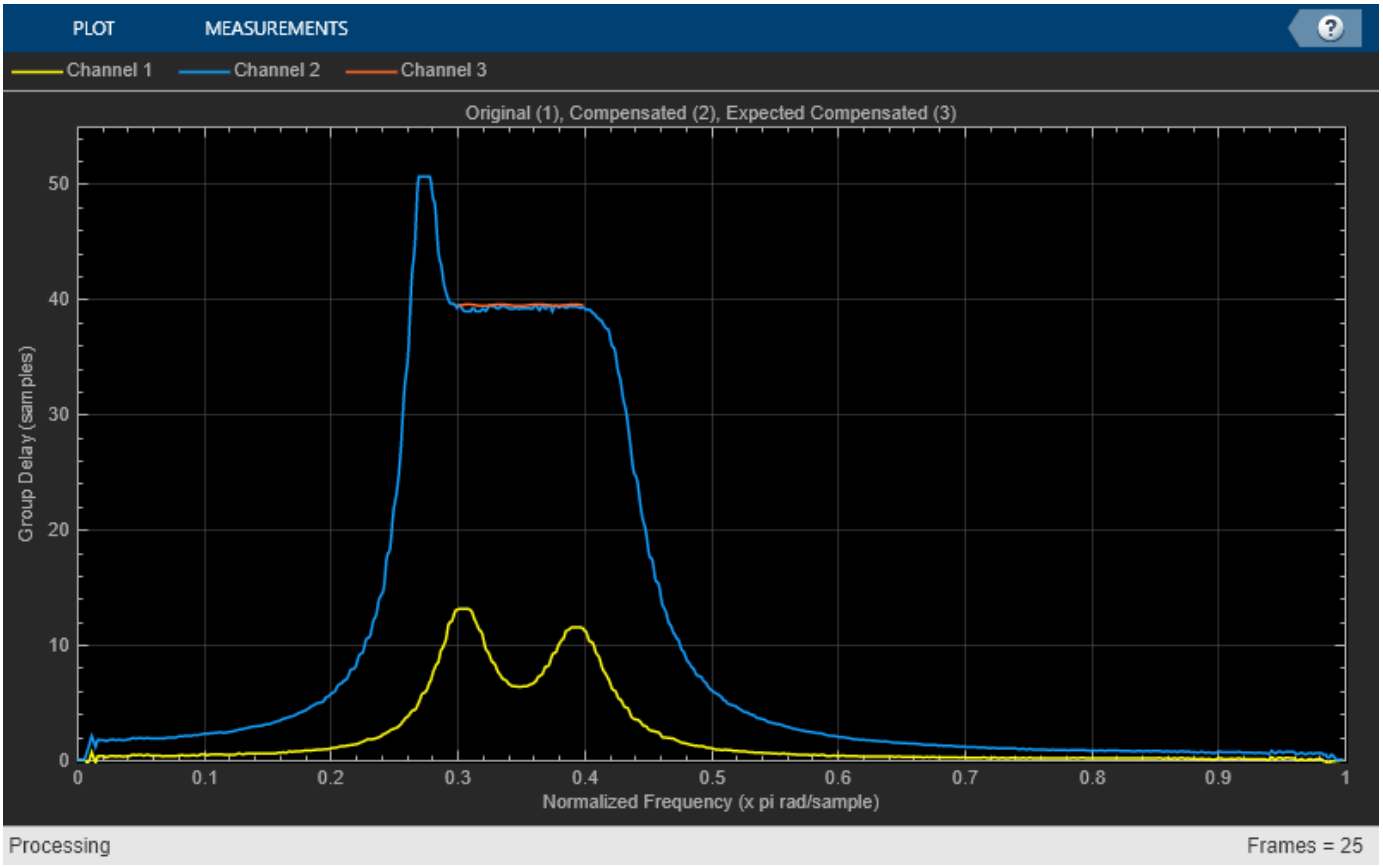
```
allpassSpec = fdesign.arbgrpdelay('N,B,F,Gd',8,1,wncomp,g1);
allpassFilter = design(allpassSpec,'iirlpnorm','MaxPoleRadius',0.95, ...
    'SystemObject', true);
```

Cascade the original filter with the compensation filter to achieve the desired group delay equalization. Verify by processing white noise and estimating the group delay at the two output stages.

```
gdOrig = grpdelay(bandpassFilter, numRealPoints);
gdComp = grpdelay(allpassFilter, numRealPoints);
range = wn > wncomp(1) & wn < wncomp(end);
gdExp = nan(numRealPoints,1); gdExp(range) = gdOrig(range) + gdComp(range);

release(scope), scope.YLimits = [0 55];
release(tfEstimator);
```

Stream random samples through filter cascade

```
for k = 1:Nframes
    x = randn(samplesPerFrame,1);  % Input signal = white Gaussian noise

    y_orig = bandpassFilter(x);    % Filter noise with original IIR filter
    y_corr = allpassFilter(y_orig);% Compensating filter

    Txy = tfEstimator([x, x],[y_orig, y_corr]);
    gdMeas = HelperMeasureGroupDelay(Txy, [], 20);
    scope([gdMeas, gdExp]);
end
```



The resulting filter has one pair of constrained poles. The group delay variation in the passband ([0.3 0.4]*pi rad/sample) is less than 0.2 samples.

Bandstop Equalization

Design a passband group delay equalizer for a bandstop Chebyshev filter operating with a sampling frequency of 1 KHz. The bandstop filter has two passband regions in the [0 150] Hz and [200 500] Hz intervals. You want to compensate the group delay in these bands so you use a multiband design and specify two bands.

Design a bandstop Chebyshev type-2 filter and measure its total group delay over the passbands. Convert the measured group delay to seconds because fdesign.arbgrpdelay expects group delay specifications in seconds when you specify a sampling frequency.

```
Fs = 1e3;
bandstopFilter = design(fdesign.bandstop('N,Fst1,Fst2,Ast',6,150,400,1,...
    Fs), 'cheby2', 'SystemObject', true);
f1 = 0.0:0.5:150; % Hz
g1 = grpdelay(bandstopFilter,f1,Fs).'/Fs; % seconds
f2 = 400:0.5:Fs/2; % Hz
g2 = grpdelay(bandstopFilter,f2,Fs).'/Fs; % seconds
maxg = max([g1 g2]);
```

Design a 14th order arbitrary group delay allpass filter. The pole radius is constrained to not exceed 0.95. The group delay specifications are given in seconds and the frequency specifications are given in Hertz.

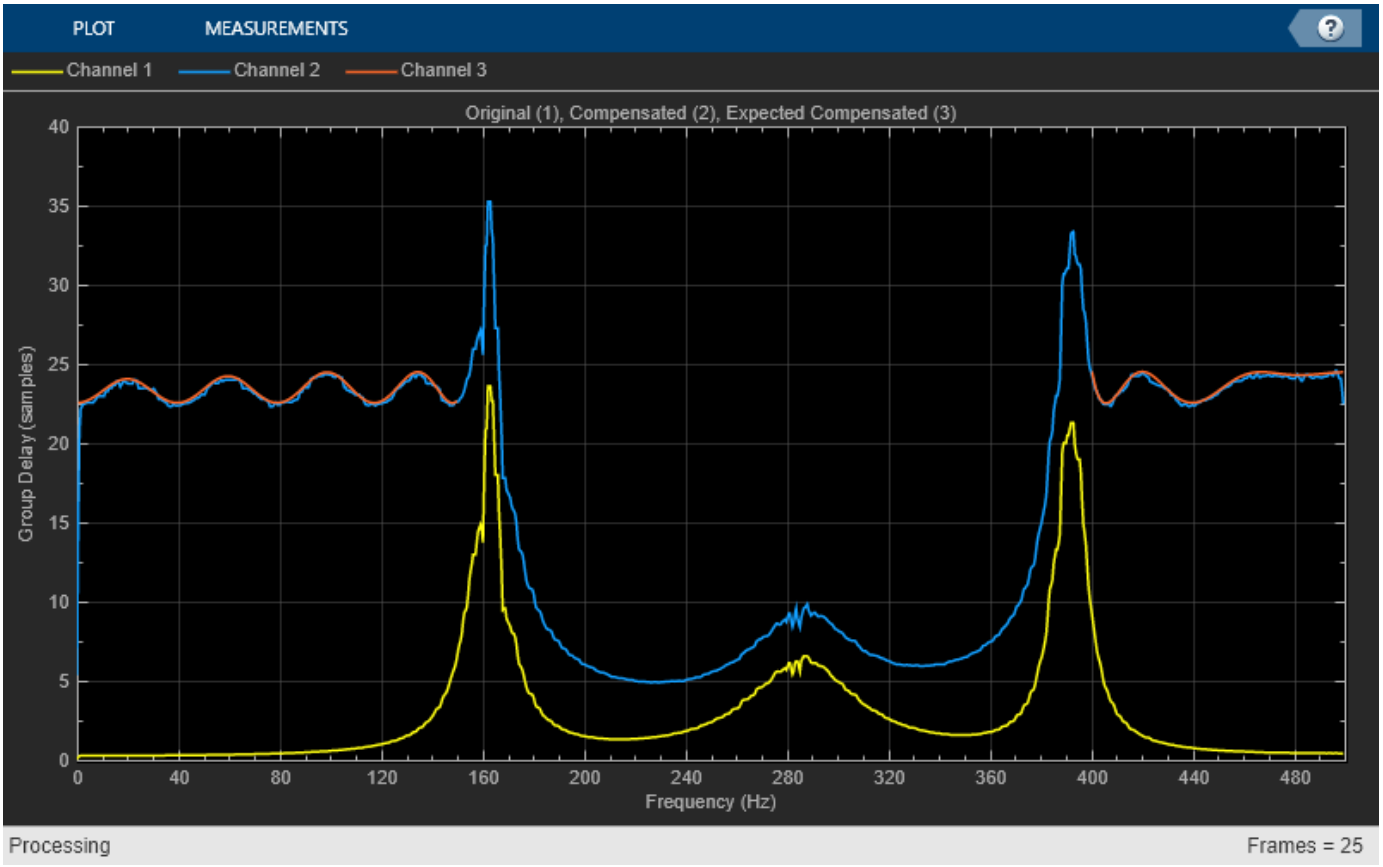
```
allpassSpec = fdesign.arbgrpdelay('N,B,F,Gd',14,2,f1,maxg-g1,f2,...
    maxg-g2,Fs);
allpassFilter = design(allpassSpec,'iirlpnorm','MaxPoleRadius',0.95,...
    'SystemObject', true);
```

Cascade the original filter with the compensation filter to process white noise and estimate the group delay at the two output stages.

```
gdOrig = grpdelay(bandstopFilter, numRealPoints);
gdComp = grpdelay(allpassFilter, numRealPoints);
fcomp = (Fs/samplesPerFrame) * (0:samplesPerFrame-1);
range = (fcomp>f1(1) & fcomp<f1(end)) | (fcomp>f2(1) & fcomp<f2(end));
gdExp = nan(numRealPoints,1); gdExp(range) = gdOrig(range) + gdComp(range);

release(scope),
    scope.YLimits = [0 40];
    scope.SampleIncrement = Fs/samplesPerFrame;
    scope.YLabel = 'Group Delay (samples)';
    scope.XLabel = 'Frequency (Hz)';
release(tfEstimator)
```

Stream random samples through filter cascade

```
Nframes = 300;
for k = 1:Nframes
    x = randn(samplesPerFrame,1);  % Input signal = white Gaussian noise

    y_orig = bandstopFilter(x);    % Filter noise with original IIR filter
    y_corr = allpassFilter(y_orig);% Compensating filter

    Txy = tfEstimator([x, x],[y_orig, y_corr]);
    gdMeas = HelperMeasureGroupDelay(Txy, [], 12);
    scope([gdMeas, gdExp]);
end
```



The resulting filter has one pair of constrained poles. The passbands have a group delay variation of less than 3 samples.

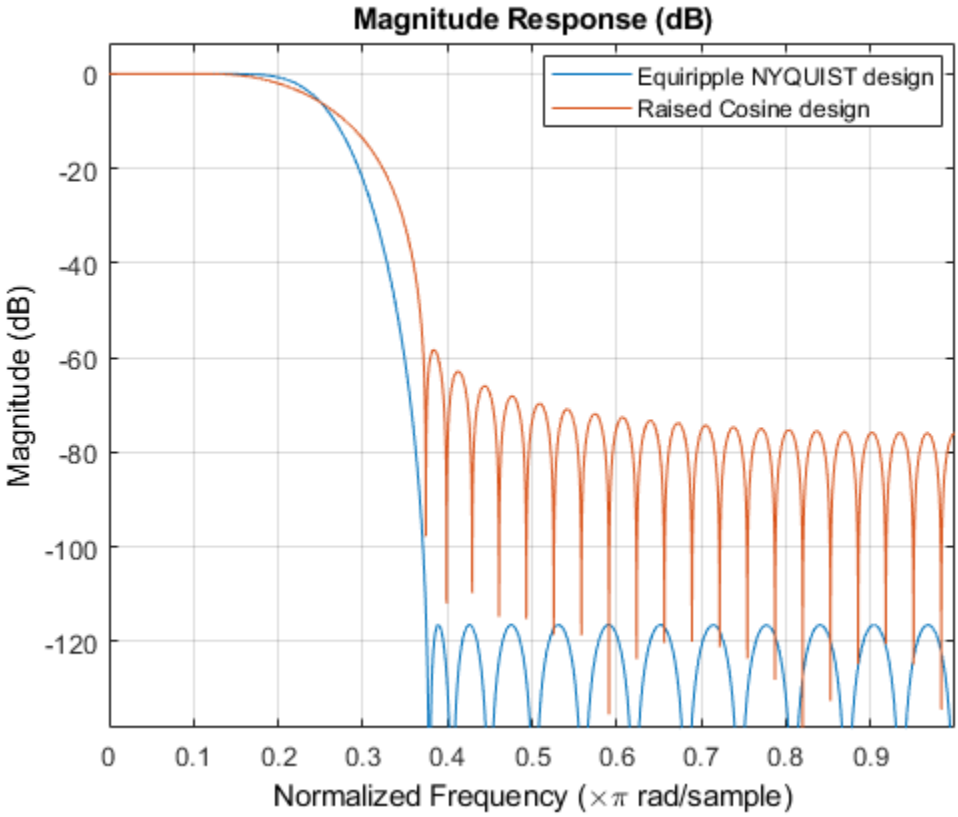
FIR Nyquist (L-th band) Filter Design

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');
```

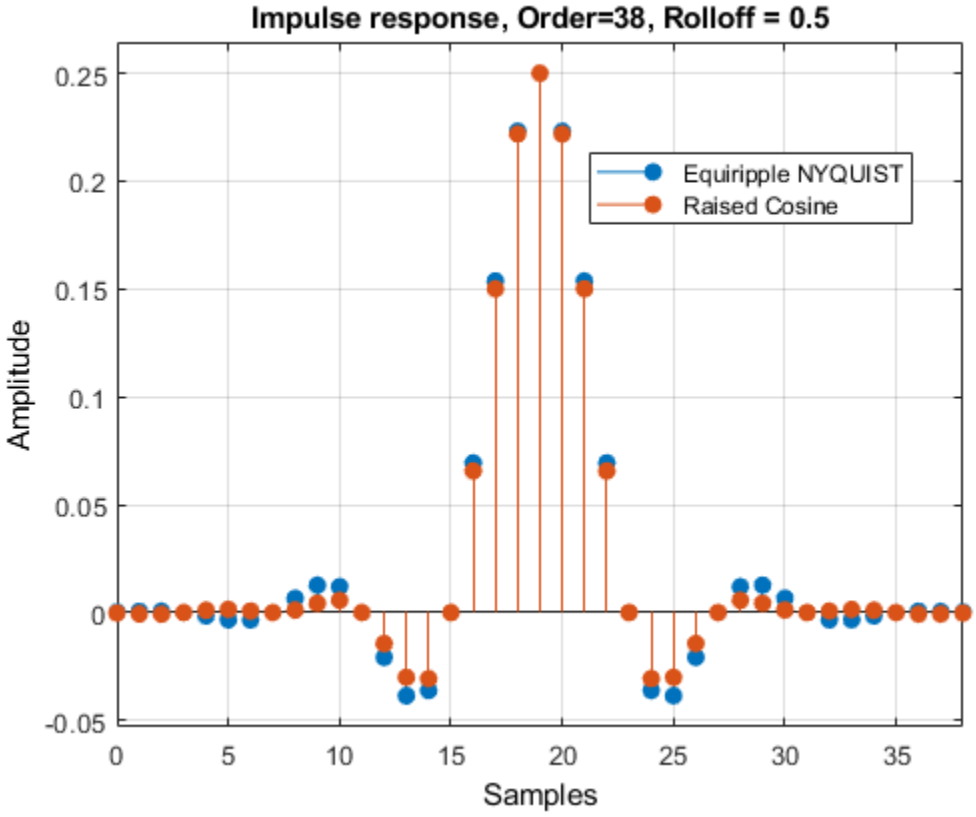


In fact, in this example it is necessary to increase the order of the raised-cosine design to about 1400 in order to attain similar attenuation.

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');
```

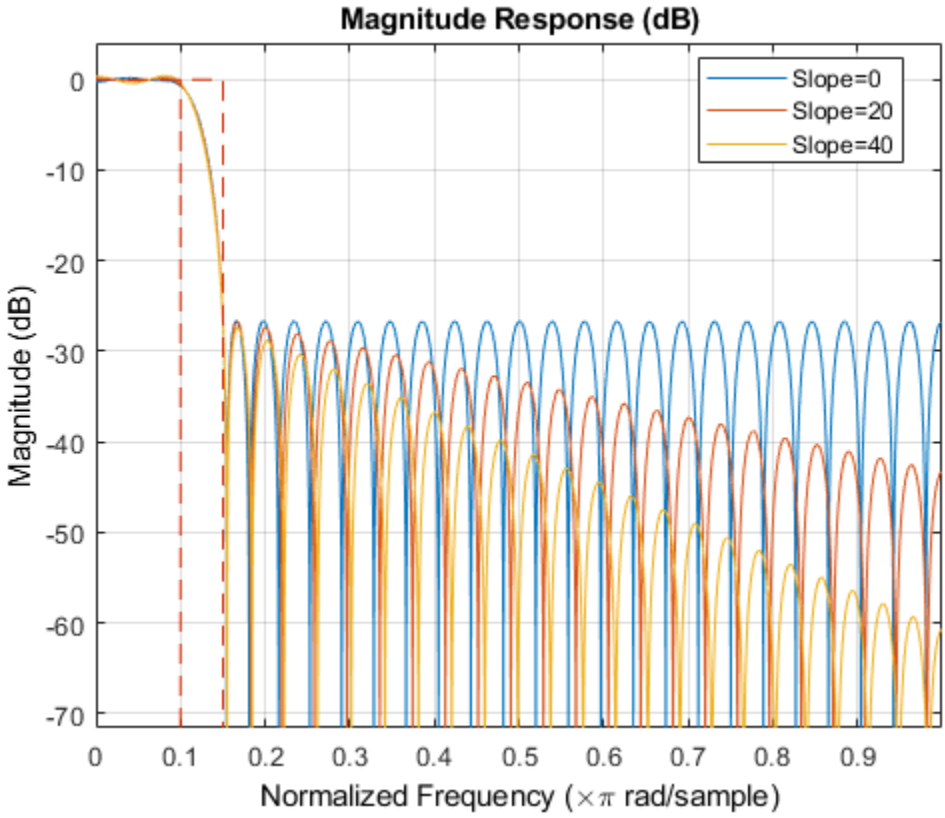


Nyquist Filters with a Sloped Stopband

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;
```

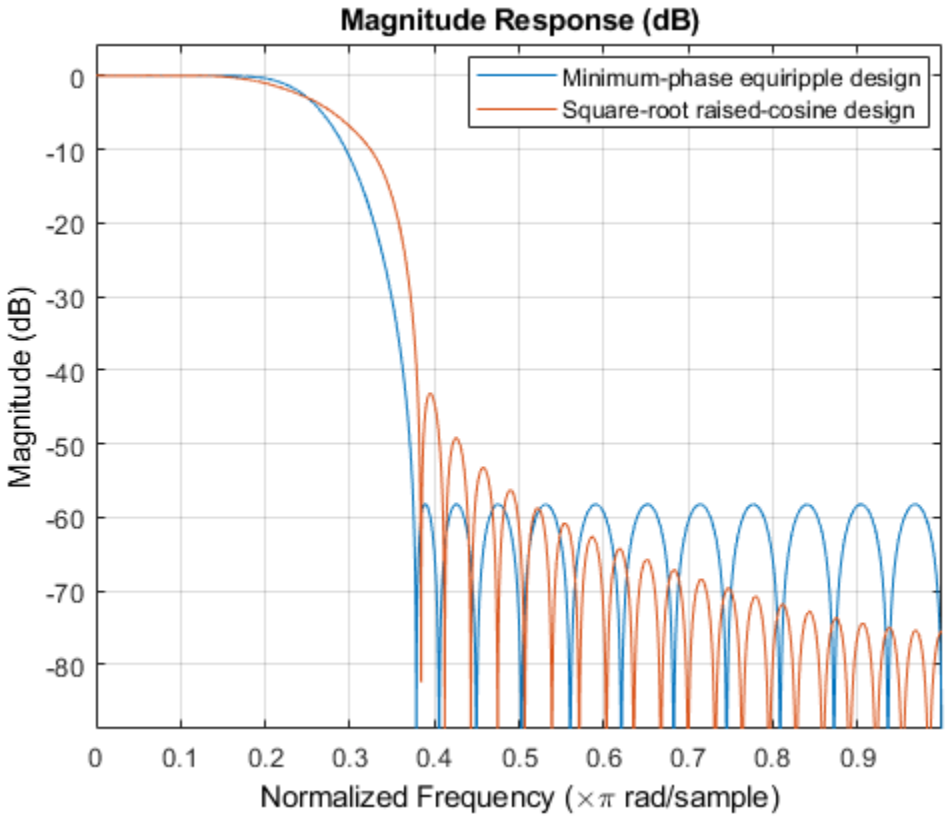
```
eq1 = design(f1,'equiripple','SystemObject',true);
eq2 = design(f1,'equiripple','StopbandShape','linear',...
    'StopbandDecay',20,'SystemObject',true);
eq3 = design(f1,'equiripple','StopbandShape','linear',...
    'StopbandDecay',40,'SystemObject',true);
fvt = fvtool(eq1,eq2,eq3,'Color','white');
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 transmitter's end and the other square root is placed at the receiver's end.

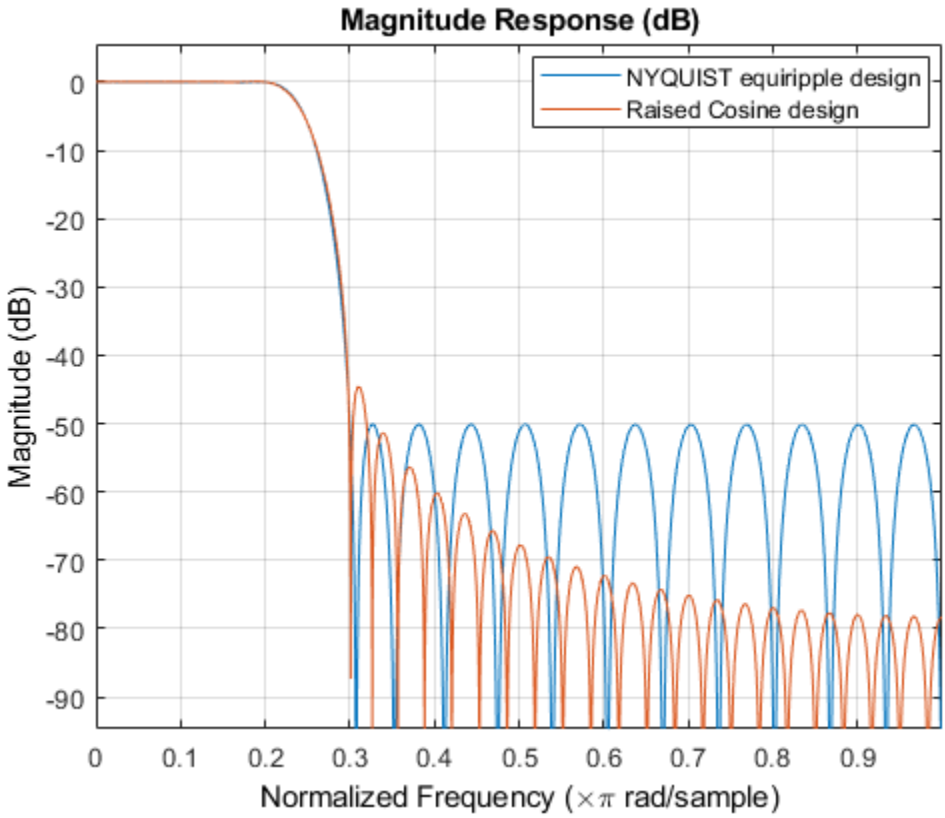
```
f1.FilterOrder = 30;
f1.Band = NBand;
f1.TransitionWidth = TW;
eq1 = design(f1,'equiripple','Minphase',true,'SystemObject',true);
coeffs = rcosdesign(R,N/NBand,NBand);
coeffs = coeffs / max(coeffs) * (-1/(pi*NBand) * (pi*(R-1) - 4*R));
srrc   = dsp.FIRFilter('Numerator',coeffs);
fvt = fvtool(eq1,srrc,'Color','white');
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');
```



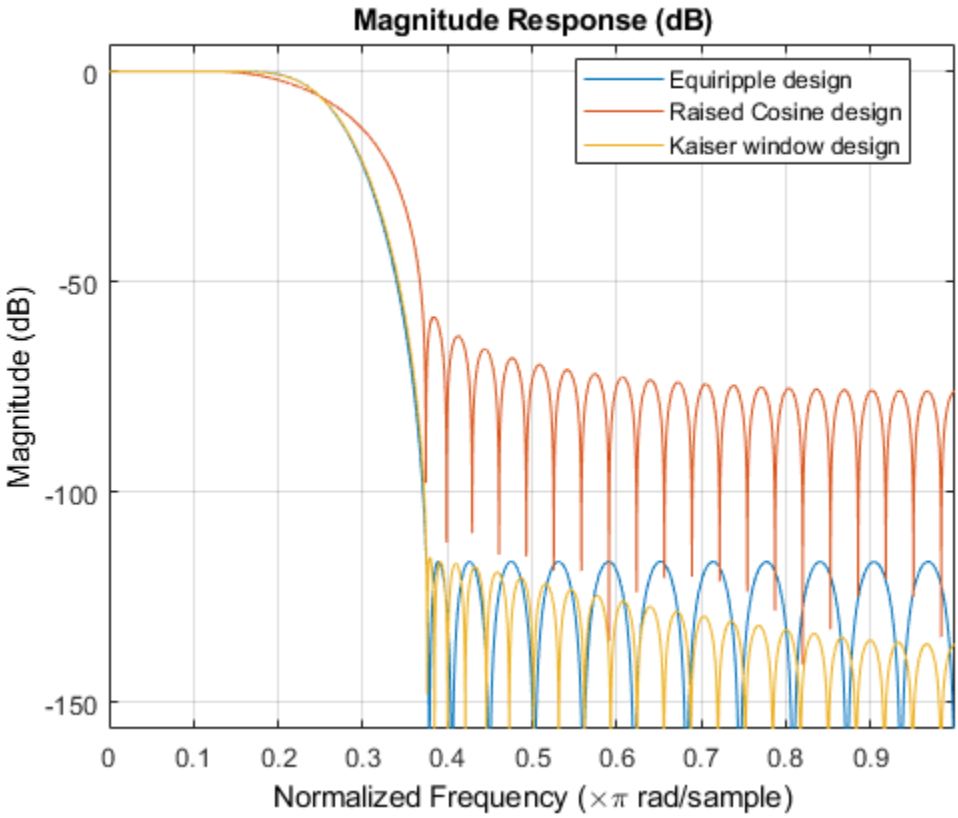
Windowed-Impulse-Response Nyquist Design

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.)

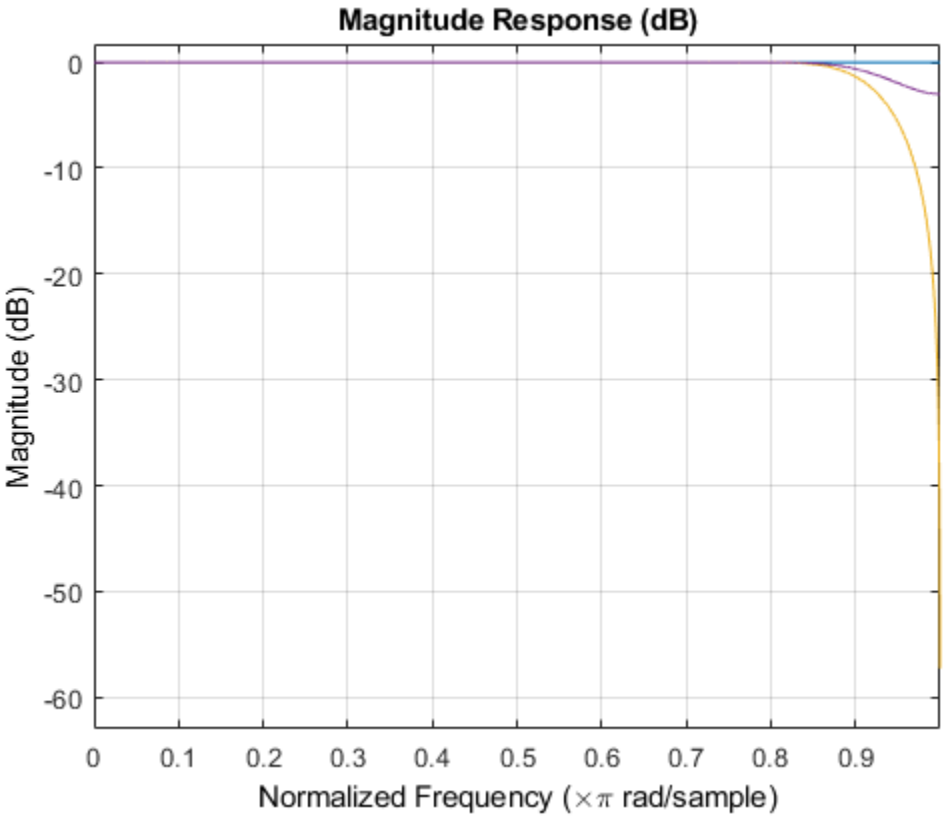
```
fvt = fvtool(eq,rc,kaiserFilt,'Color','white');
legend(fvt,'Equiripple design',...
    'Raised Cosine design','Kaiser window design');
```



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 25-2
- “Using Filter Designer” on page 24-2

FIR Halfband Filter Design

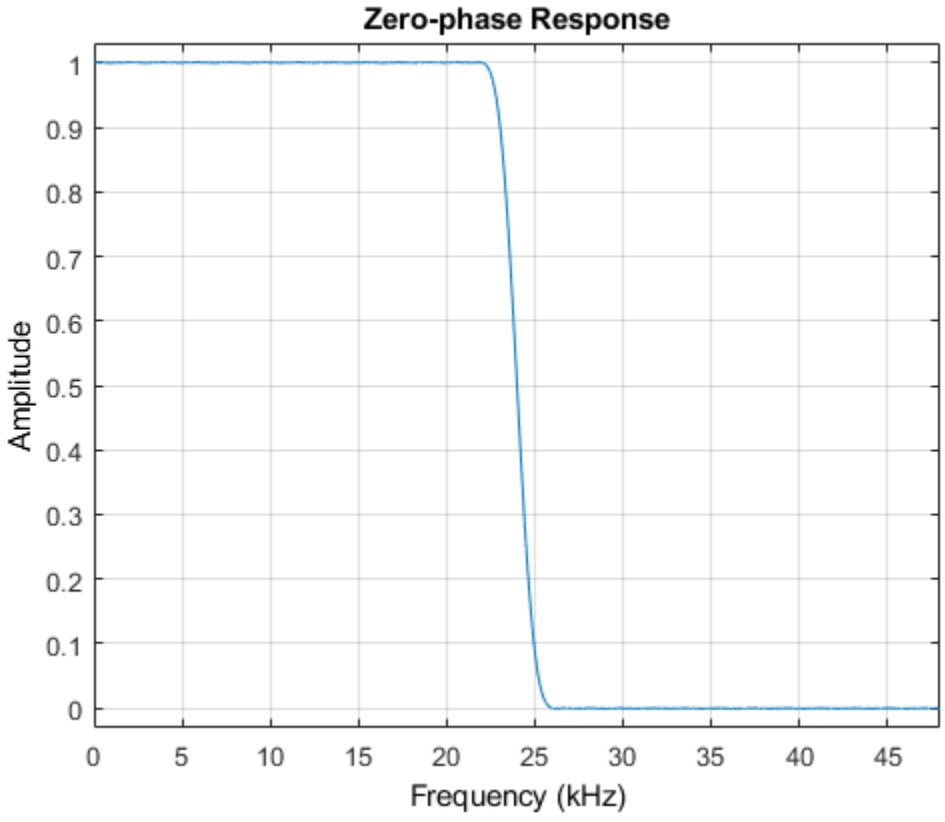
This example shows how to design FIR halfband filters. Halfband filters are widely used in multirate signal processing applications when interpolating/decimating by a factor of two. Halfband filters are implemented efficiently in polyphase form, because approximately half of its coefficients are equal to zero.

Halfband filters have two important characteristics, the passband and stopband ripples must be the same, and the passband-edge and stopband-edge frequencies are equidistant from the halfband frequency Fs/4 (Or pi/2 rad/sample in normalized frequency).

Obtaining the Halfband Coefficients

The `firhalfband` function returns the coefficients of an FIR halfband equiripple filter. As a simple example, consider a halfband filter whose dealing with data sampled at 96 kHz and a passband frequency of 22 kHz.

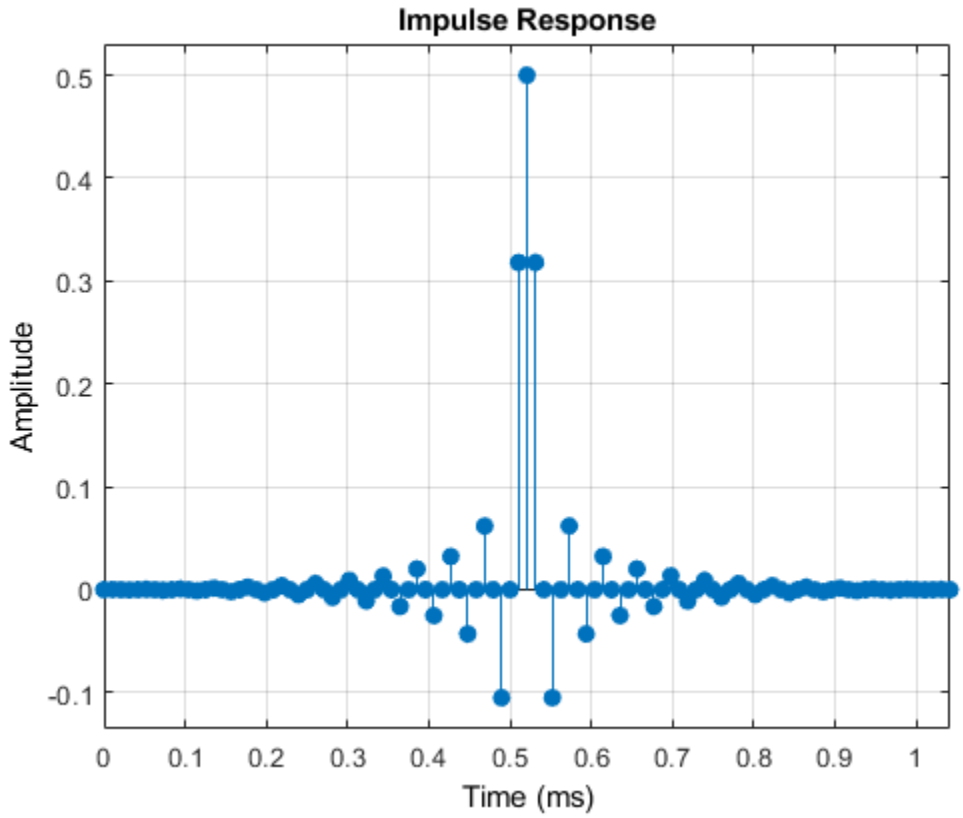
```
Fs  = 96e3;
Fp  = 22e3;
N   = 100;
num = firhalfband(N,Fp/(Fs/2));
fvt = fvtool(num,'Fs',Fs,'Color','white');
fvt.MagnitudeDisplay = 'Zero-phase';
```



By zooming in to the response, you can verify that the passband and stopband peak-to-peak ripples are the same. Also there is symmetry about the Fs/4 (24 kHz) point. The passband extends up to 22

kHz as specified and the stopband begins at 26 kHz. We can also verify that every other coefficient is equal to zero by looking at the impulse response. This makes the filter very efficient to implement for interpolation/decimation by a factor of 2.

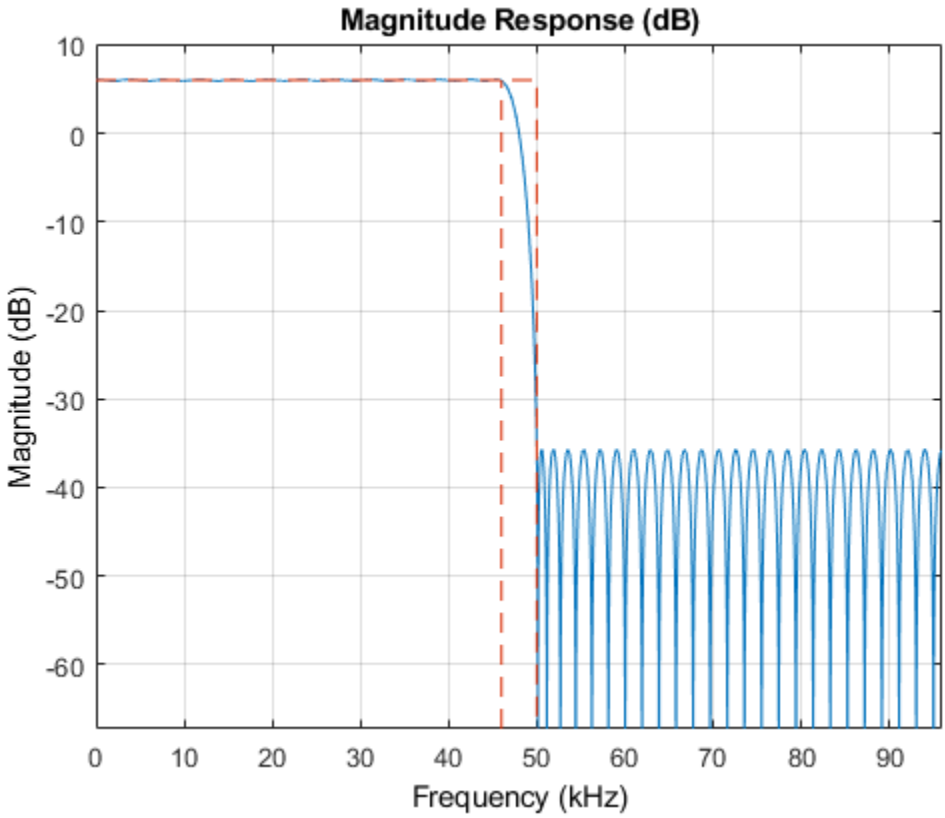
```
fvt.Analysis = 'impulse';
```



`dsp.FIRHalfbandInterpolator` and `dsp.FIRHalfbandDecimator`

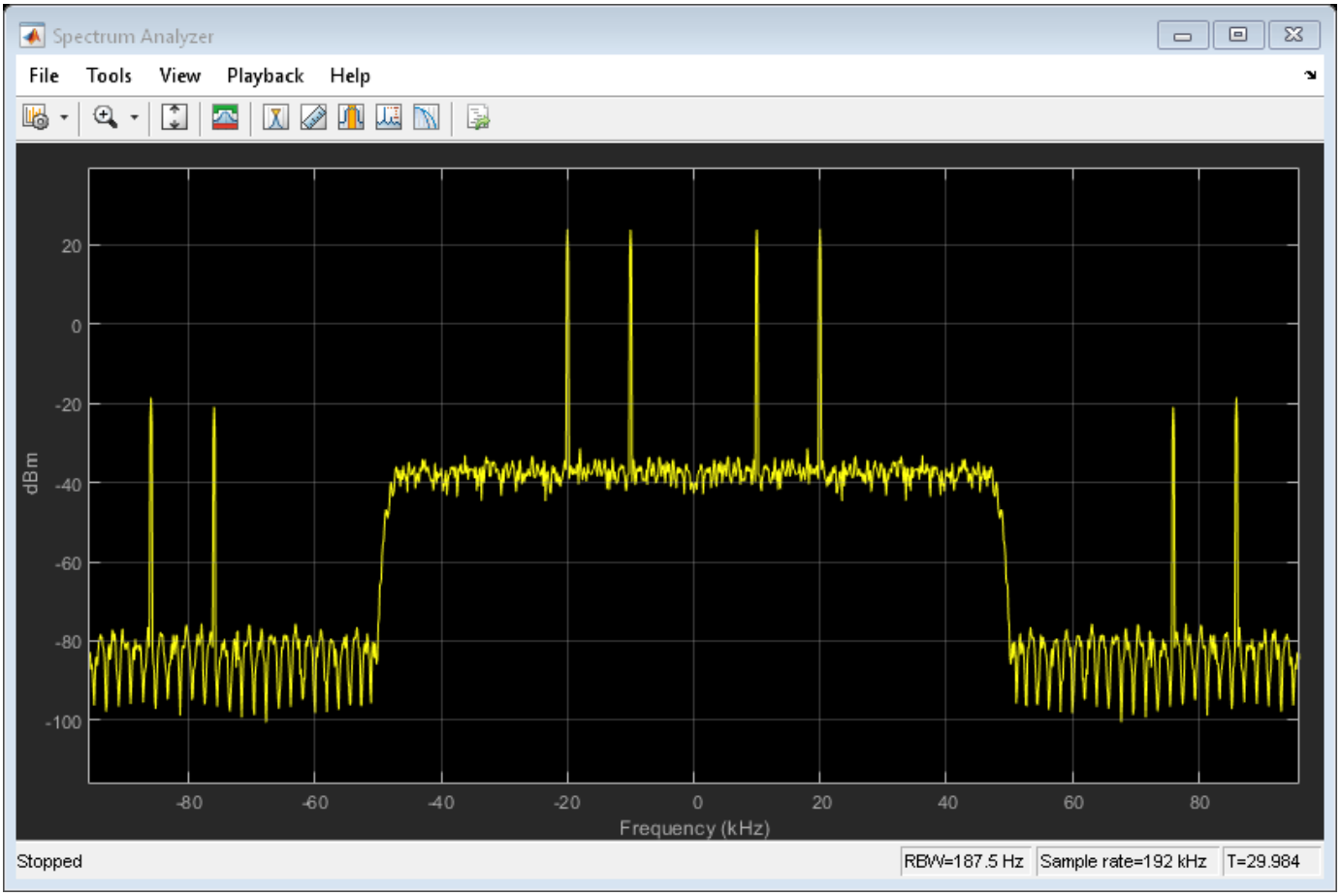
The `firhalfband` function provides several other design options. However, for most cases it is preferable to work directly with `dsp.FIRHalfbandInterpolator` and `dsp.FIRHalfbandDecimator`. These two System objects not only design the coefficients, but also provide efficient implementation of the polyphase interpolator/decimator. They support filtering double/single precision floating-point data as well as fixed-point data. They also support C and HDL code generation as well as optimized ARM® Cortex® M and Cortex A code generation.

```
halfbandInterpolator = dsp.FIRHalfbandInterpolator('SampleRate',Fs, ...
    'Specification','Filter order and transition width', ...
    'FilterOrder',N,'TransitionWidth',4000);
fvtool(halfbandInterpolator,'Fs',2*Fs,'Color','white');
```



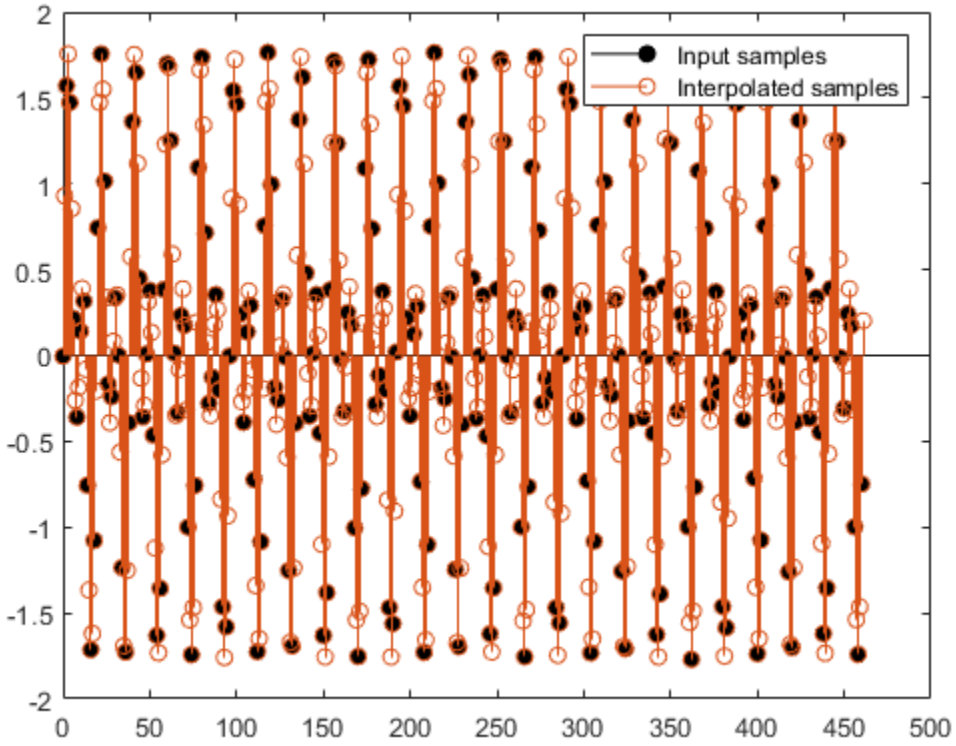
In order to perform the interpolation, the `dsp.FIRHalfbandInterpolator` System object is used. Because this is a multirate filter, it is important to define what is meant by the sample rate. For this and all other System objects, the sample rate refers to the sample rate of the input signal. However, FVTool defines the sample rate as the rate at which the filter is running. In the case of interpolation, you upsample and then filter (conceptually), therefore the sampling rate of FVTool needs to be specified as 2*Fs because of the upsampling by 2.

```
FrameSize = 256;
scope = dsp.SpectrumAnalyzer('SampleRate',2*Fs,'SpectralAverages',5);
sine1 = dsp.SineWave('Frequency',10e3','SampleRate',Fs, ...
    'SamplesPerFrame',FrameSize);
sine2 = dsp.SineWave('Frequency',20e3','SampleRate',Fs, ...
    'SamplesPerFrame',FrameSize);
tic
while toc < 10
    x = sine1() + sine2() + 0.01.*randn(FrameSize,1); %  96 kHz
    y = halfbandInterpolator(x);                      % 192 kHz
    scope(y);
end

release(scope);
```



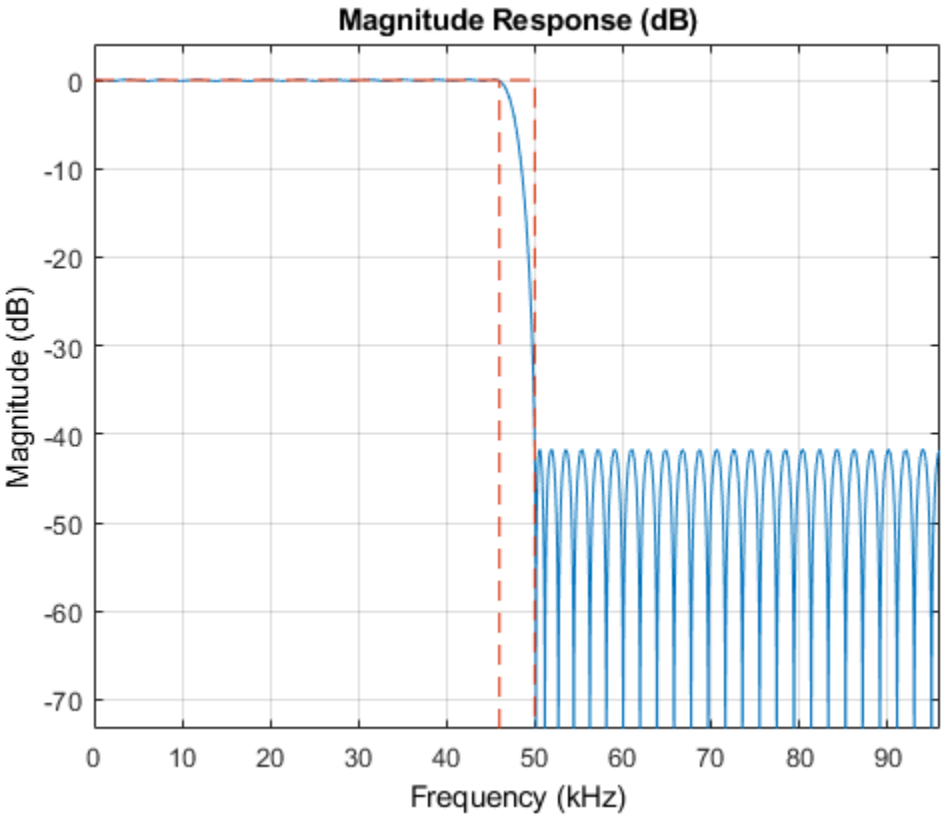
Notice that the spectral replicas are attenuated by about 40 dB which is roughly the attenuation provided by the halfband filter. Compensating for the group-delay in the filter, it is possible to plot the input and interpolated samples overlaid. Notice that the input samples are retained unchanged at the output of the filter. This is because one of the polyphase branches of the halfband is a pure delay branch which does not change the input samples.

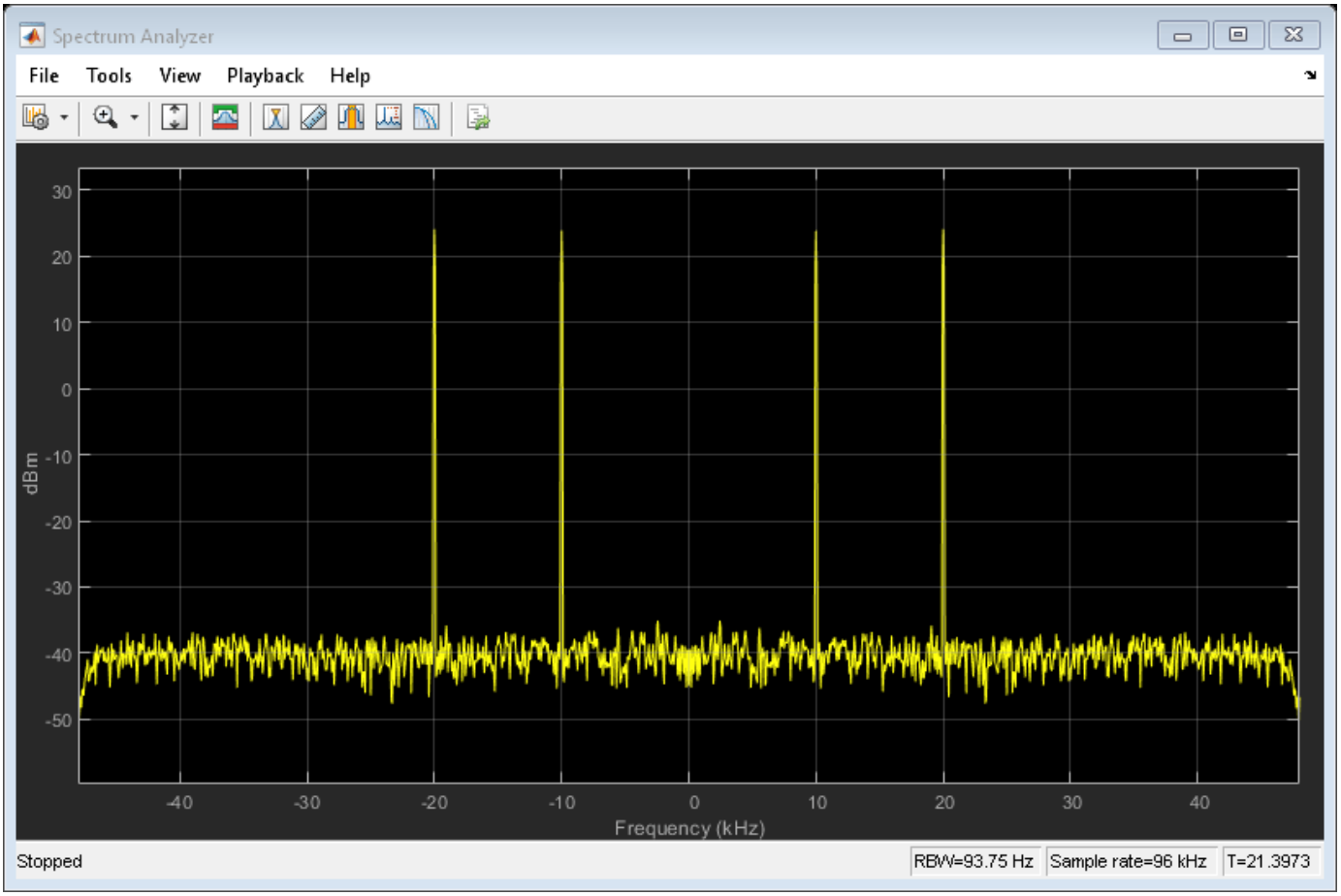
```
grpDel = 50;
n = 0:2:511;
stem(n(1:end-grpDel/2),x(1:end-grpDel/2),'k','filled')
hold on
nu = 0:511;
stem(nu(1:end-grpDel),y(grpDel+1:end))
legend('Input samples','Interpolated samples')
```



In the case of decimation, the sample rate specified in `dsp.FIRHalfbandDecimator` corresponds to the sample rate of the filter, since you filter and then downsample (conceptually). So for decimators, the Fs specified in FVTool does not need to be multiplied by any factor.

```
FrameSize = 256;
FsIn = 2*Fs;
halfbandDecimator = dsp.FIRHalfbandDecimator('SampleRate',FsIn, ...
    'Specification','Filter order and transition width', ...
    'FilterOrder',N,'TransitionWidth',4000);
fvtool(halfbandDecimator,'Fs',FsIn,'Color','white');
scope = dsp.SpectrumAnalyzer('SampleRate',Fs,'SpectralAverages',5);
sine1 = dsp.SineWave('Frequency',10e3','SampleRate',Fs, ...
    'SamplesPerFrame',FrameSize);
sine2 = dsp.SineWave('Frequency',20e3','SampleRate',Fs, ...
    'SamplesPerFrame',FrameSize);
tic
while toc < 10
    x = sine1() + sine2() + 0.01.*randn(FrameSize,1); %  96 kHz
    y = halfbandInterpolator(x);                      % 192 kHz
    xd = halfbandDecimator(y);                        %  96 kHz
    scope(xd);
end

release(scope);
```


Magnitude Response (dB)
Magnitude (dB)
0
-10
-20
-30
-40
-50
-60
-70
0
10
20
30
40
50
60
70
80
90
Frequency (kHz)



Obtaining the Filter Coefficients

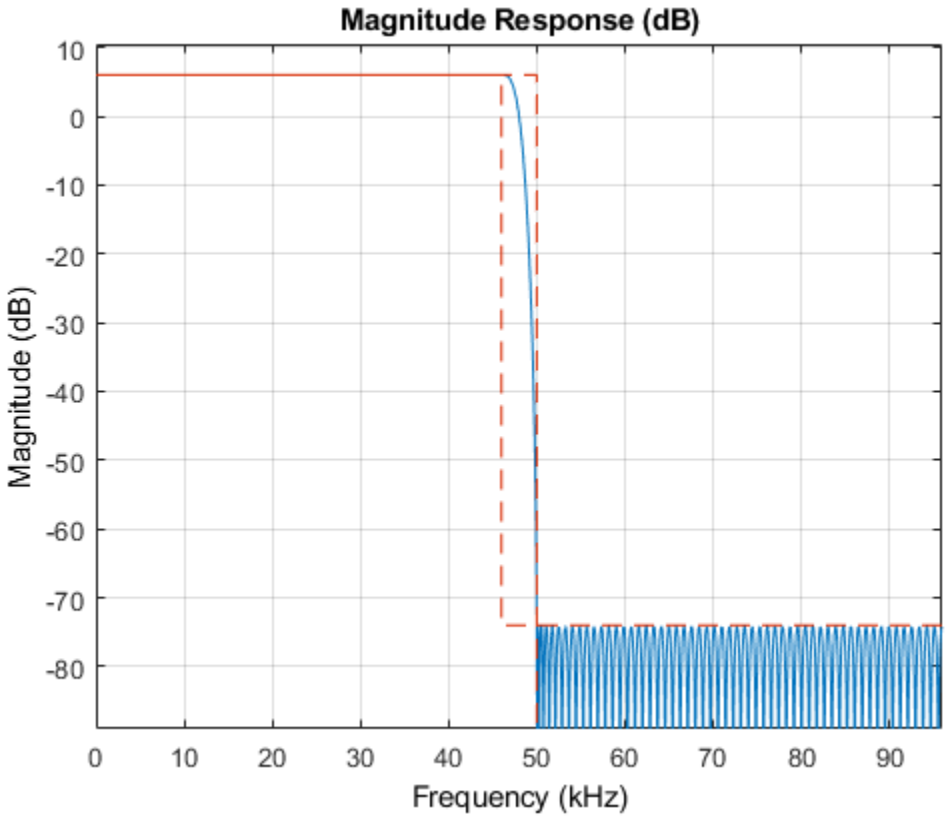
The filter coefficients can be extracted from the interpolator/decimator by using the `tf` function.

```
num = tf(halfbandInterpolator); % Or num = tf(halfbandDecimator);
```

Using Different Design Specifications

Instead of specifying the filter order and transition width, you can design a minimum-order filter that provides a given transition width as well as a given stopband attenuation.

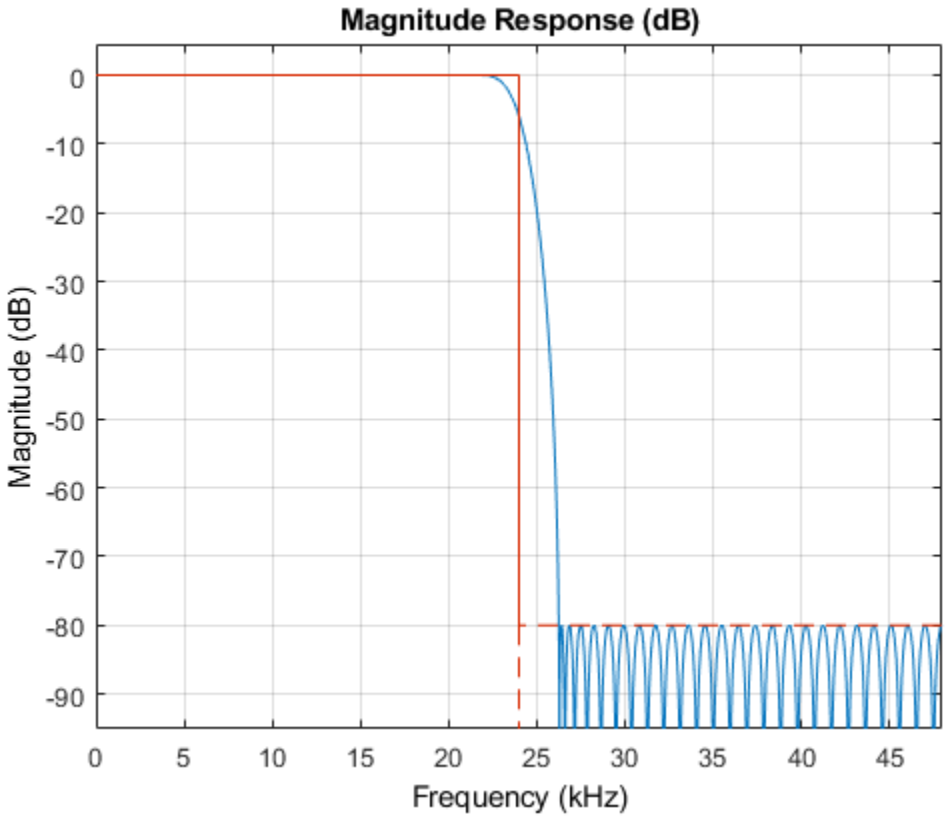
```
Ast = 80; % 80 dB
halfbandInterpolator = dsp.FIRHalfbandInterpolator('SampleRate',Fs, ...
    'Specification','Transition width and stopband attenuation', ...
    'StopbandAttenuation',Ast,'TransitionWidth',4000);
fvtool(halfbandInterpolator,'Fs',2*Fs,'Color','white');
```



Notice that as with all interpolators, the passband gain in absolute units is equal to the interpolation factor (2 in the case of halfbands). This corresponds to a passband gain of 6.02 dB.

It is also possible to specify the filter order and the stopband attenuation.

```
halfbandDecimator = dsp.FIRHalfbandDecimator('SampleRate',Fs, ...
    'Specification','Filter order and stopband attenuation', ...
    'StopbandAttenuation',Ast,'FilterOrder',N);
fvtool(halfbandDecimator,'Fs',Fs,'Color','white');
```



Unlike interpolators, decimators have a gain of 1 (0 dB) in the passband.

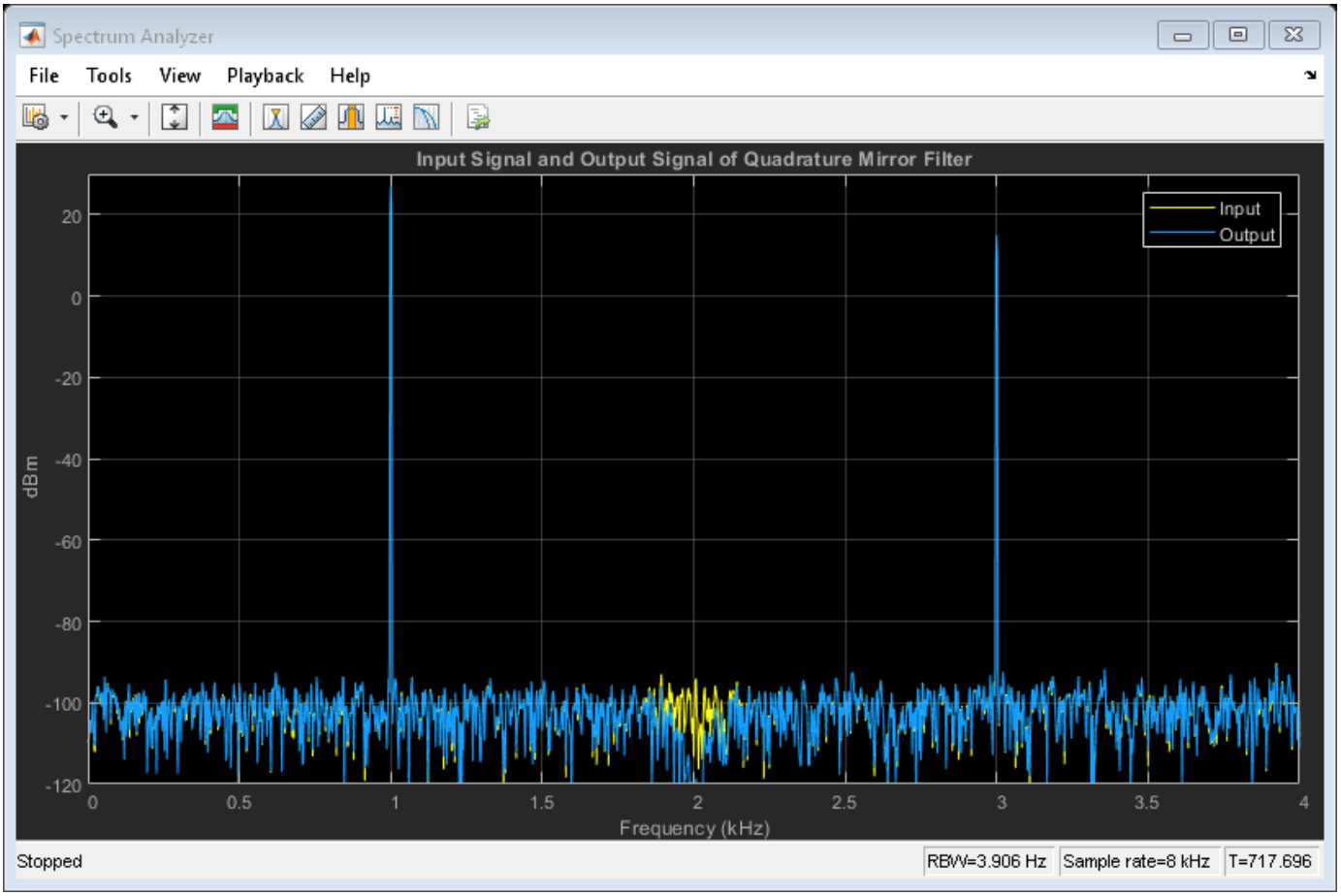
Using Halfband Filters for Filter Banks

Halfband interpolators and decimators can be used to efficiently implement synthesis/analysis filter banks. The halfband filters shown so far have all been lowpass filters. With a single extra adder, it is possible to obtain a highpass response in addition to the lowpass response and use the two responses for the filter bank implementation.

The following code simulates a quadrature mirror filter (QMF) bank. An 8 kHz signal consisting of 1 kHz and 3 kHz sine waves is separated into two 4 kHz signals using a lowpass/highpass halfband decimator. The lowpass signal retains the 1 kHz sine wave while the highpass signal retains the 3 kHz sine wave (which is aliased to 1 kHz after downsampling). The signals are then merged back together with a synthesis filter bank using a halfband interpolator. The highpass branch upconverts the aliased 1 kHz sine wave back to 3 kHz. The interpolated signal has an 8 kHz sampling rate.

```
Fs1 = 8000; % Units = Hz
Spec = 'Filter order and transition width';
Order = 52;
TW = 4.1e2; % Units = Hz

% Construct FIR Halfband Interpolator
halfbandInterpolator = dsp.FIRHalfbandInterpolator( ...
    'Specification',Spec, ...
    'FilterOrder',Order, ...
    'TransitionWidth',TW, ...
    'SampleRate',Fs1/2, ...
```

```
    'FilterBankInputPort',true);

% Construct FIR Halfband Decimator
halfbandDecimator = dsp.FIRHalfbandDecimator( ...
    'Specification',Spec, ...
    'FilterOrder',Order, ...
    'TransitionWidth',TW, ...
    'SampleRate',Fs1);

% Input
f1 = 1000;
f2 = 3000;
InputWave   = dsp.SineWave('Frequency',[f1,f2],'SampleRate',Fs1, ...
    'SamplesPerFrame',1024,'Amplitude',[1 0.25]);

% Construct Spectrum Analyzer object to view the input and output
scope = dsp.SpectrumAnalyzer('SampleRate',Fs1, ...
    'PlotAsTwoSidedSpectrum',false,'ShowLegend',true,'YLimits', ...
    [-120 30], ...
    'Title', ...
    'Input Signal and Output Signal of Quadrature Mirror Filter');
scope.ChannelNames = {'Input','Output'};

tic
while toc < 10
    Input = sum(InputWave(),2);
    NoisyInput = Input+(10^-5)*randn(1024,1);
    [Lowpass,Highpass] = halfbandDecimator(NoisyInput);
    Output = halfbandInterpolator(Lowpass,Highpass);
    scope([NoisyInput,Output]);
end

release(scope);
```



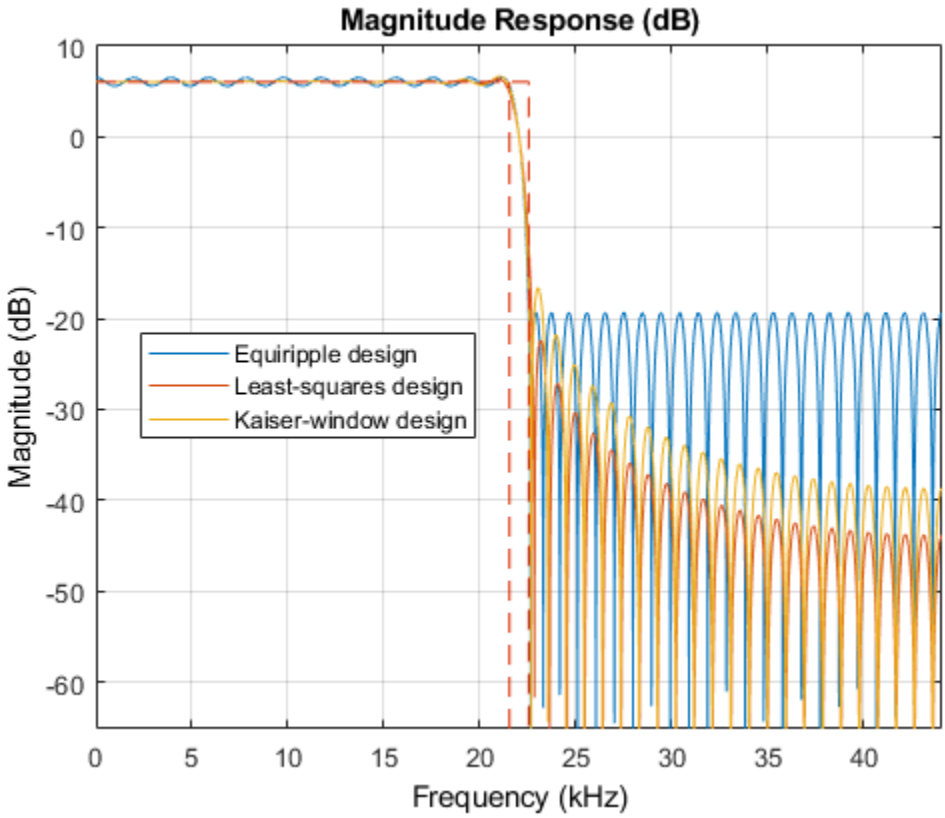
Advanced Design Options: Specifying Different Design Algorithms

All designs presented so far have been optimal equiripple designs. Using `fdesign.interpolator` and `fdesign.decimator`, other design algorithms are available.

```
Fs = 44.1e3;
N  = 90;
TW = 1000/Fs; % Transition width
filtSpecs = fdesign.interpolator(2,'halfband','N,TW',N,TW);
equiripplеHBFilter = design(filtSpecs,'equiripple','SystemObject',true);
leastSquaresHBFilter = design(filtSpecs,'firls','SystemObject',true);
kaiserHBFilter = design(filtSpecs,'kaiserwin','SystemObject',true);
```

You can compare the designs with FVTool. The different designs allow for trade offs between minimum stopband attenuation and more overall attenuation.

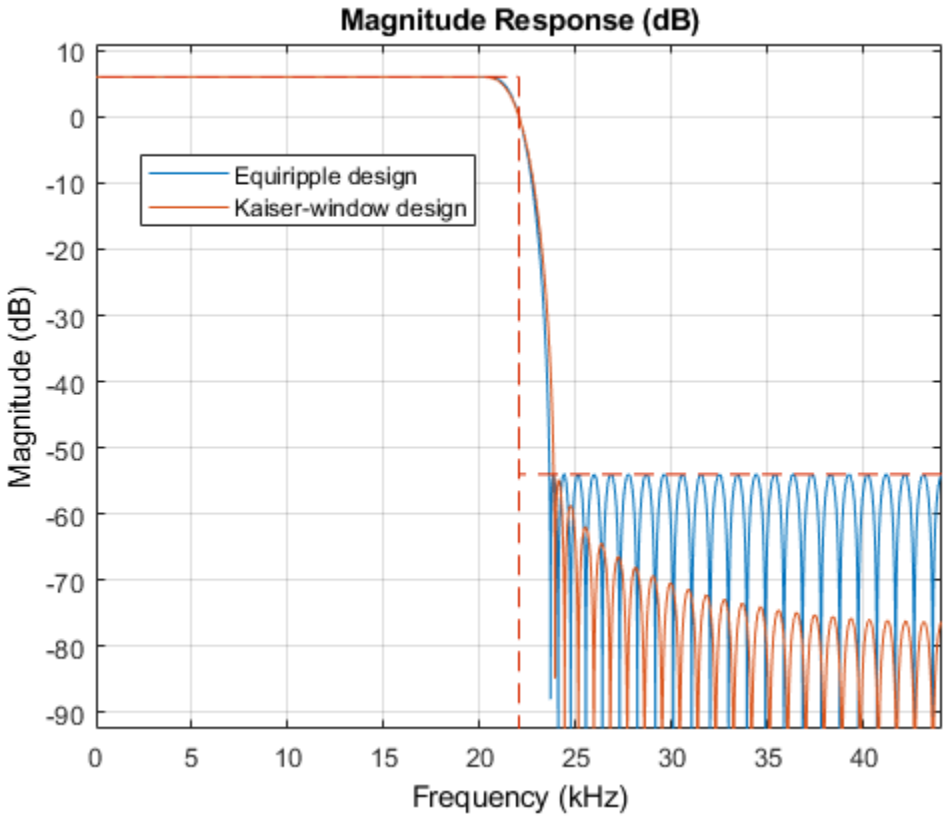
```
fvt = fvtool(equiripplеHBFilter,leastSquaresHBFilter,kaiserHBFilter, ...
    'Fs',2*Fs,'Color','white');
legend(fvt,'Equiripple design','Least-squares design', ...
    'Kaiser-window design')
```



Controlling the Stopband Attenuation

Alternatively, one can specify the order and the stopband attenuation. This allows for trade offs between overall stopband attenuation and transition width.

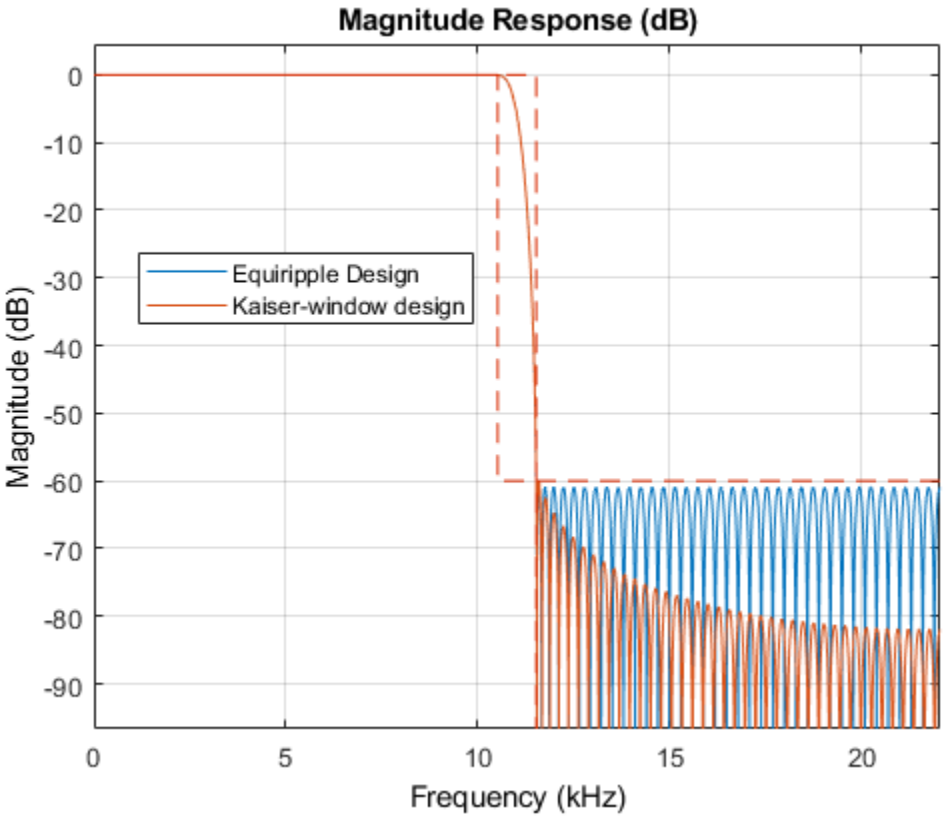
```
Ast  = 60; % Minimum stopband attenuation
filtSpecs = fdesign.interpolator(2,'halfband','N,Ast',N,Ast);
equiripplеHBFilter = design(filtSpecs,'equiripple','SystemObject',true);
kaiserHBFilter = design(filtSpecs,'kaiserwin','SystemObject',true);
fvt = fvtool(equiripplеHBFilter,kaiserHBFilter,'Fs',2*Fs,'Color','white');
legend(fvt,'Equiripple design','Kaiser-window design')
```



Minimum-Order Designs

Kaiser window designs can also be used in addition to equiripple designs when designing a filter of the minimum-order necessary to meet the design specifications. The actual order for the Kaiser window design is larger than that needed for the equiripple design, but the overall stopband attenuation is better in return.

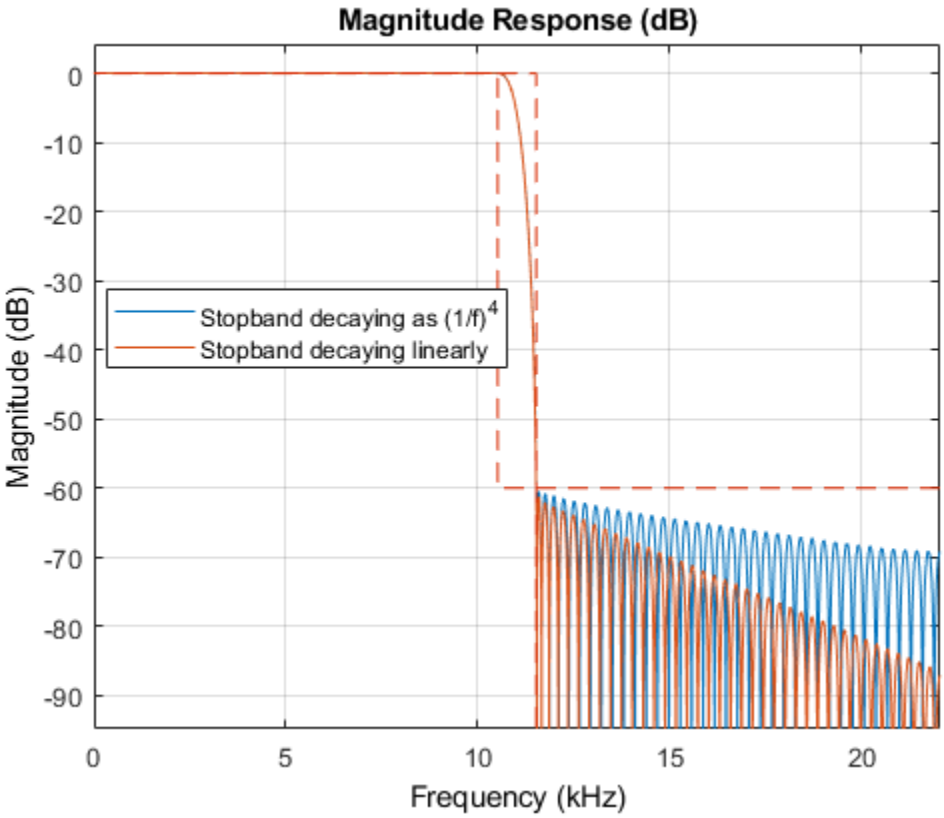
```
Fs = 44.1e3;
TW = 1000/(Fs/2); % Transition width
Ast = 60;  % 60 dB minimum attenuation in the stopband
filtSpecs = fdesign.decimator(2,'halfband','TW,Ast',TW,Ast);
equirippleHBFilter = design(filtSpecs,'equiripple','SystemObject',true);
kaiserHBFilter = design(filtSpecs,'kaiserwin','SystemObject',true);
fvt = fvtool(equirippleHBFilter,kaiserHBFilter,'Fs',Fs,'Color','white');
legend(fvt,'Equiripple Design','Kaiser-window design')
```



Equiripple Designs with Increasing Stopband Attenuation

Instead of designing Kaiser window filters, it is also possible to obtain increasing stopband attenuation with modified 'equiripple' designs.

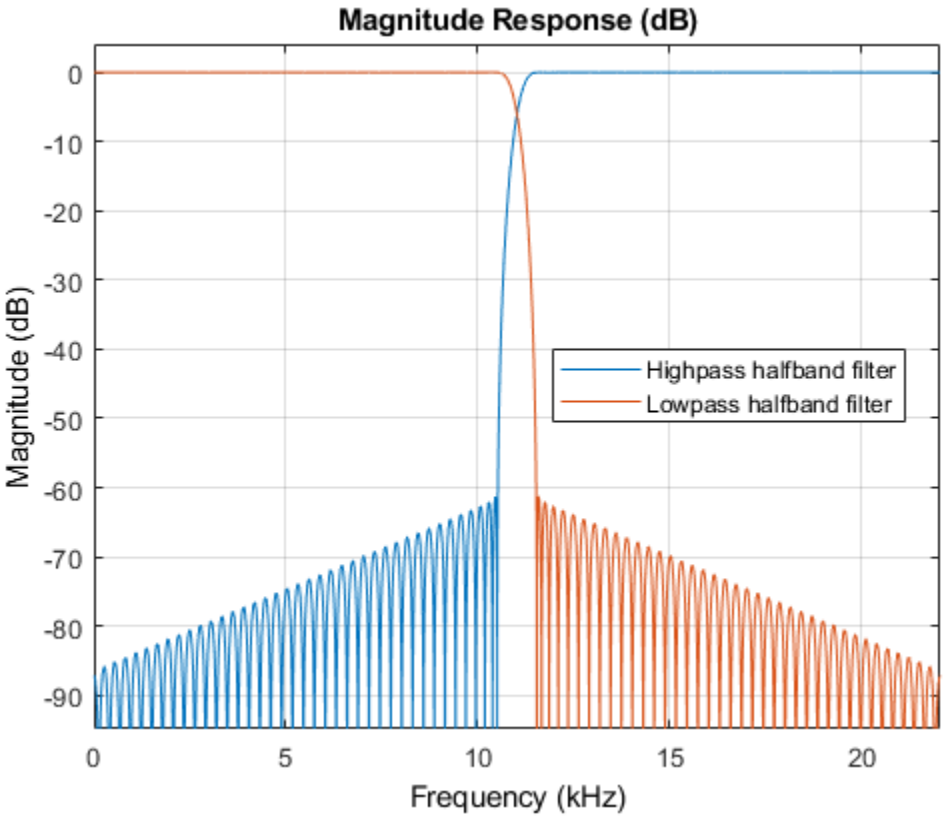
```
equiripplеHBFilter1 = design(filtSpecs,'equiripple', ...
    'StopbandShape','1/f','StopbandDecay',4,'SystemObject',true);
equiripplеHBFilter2 = design(filtSpecs,'equiripple', ...
    'StopbandShape','linear','StopbandDecay',53.333,'SystemObject',true);
fvt = fvtool(equiripplеHBFilter1,equiripplеHBFilter2, ...
    'Fs',Fs,'Color','white');
legend(fvt,'Stopband decaying as (1/f)^4','Stopband decaying linearly')
```



Highpass Halfband Filters

A highpass halfband filter can be obtained from a lowpass halfband filter by changing the sign of every second coefficient. Alternatively, one can directly design a highpass halfband by setting the 'Type' property to 'Highpass'.

```
filtSpecs = fdesign.decimator(2,'halfband', ...
    'Type','Highpass','TW,Ast',TW,Ast);
halfbandHPFilter = design(filtSpecs,'equiripple', ...
    'StopbandShape','linear','StopbandDecay',53.333,'SystemObject',true);
fvt = fvtool(halfbandHPFilter,equirippleHBFilter2,'Fs',Fs,'Color','white');
legend(fvt,'Highpass halfband filter','Lowpass halfband filter')
```


Magnitude Response (dB)
Magnitude (dB)
0
-10
-20
-30
-40
-50
-60
-70
-80
-90
0
5
10
15
20
Frequency (kHz)
Highpass halfband filter
Lowpass halfband filter

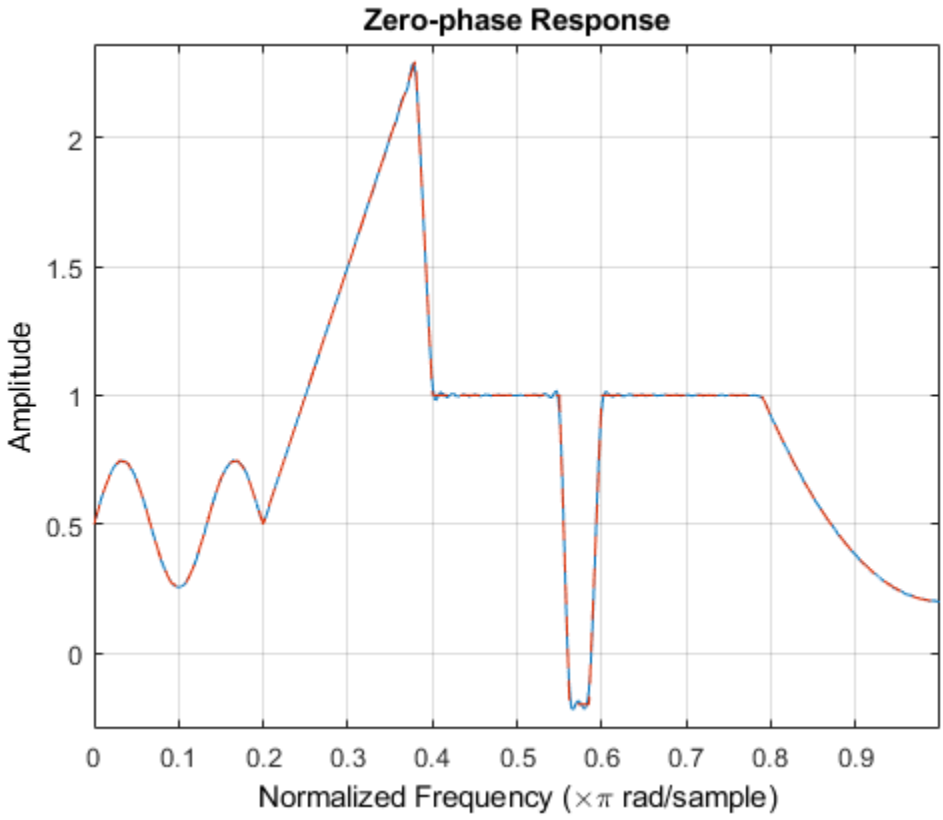
Arbitrary Magnitude Filter Design

This example shows how to design filters with arbitrary magnitude response. The family of filter design (FDESIGN) objects allow for the design of filters with various types of responses. Among these types, the arbitrary magnitude is the less specialized and most versatile one. The examples below illustrate how arbitrary magnitude designs can solve problems when other response types find limitations.

FIR Modeling with the Frequency Sampling Method

This section illustrates a case where the amplitude of the filter is defined over the complete Nyquist range (there are no relaxed or "don't care" regions). The example that follows uses a single (full) band specification type and the robust frequency sampling algorithm to design a filter whose amplitude is defined over three sections: a sinusoidal section, a piecewise linear section and a quadratic section. It is necessary to select a large filter order because the shape of the filter is quite complicated:

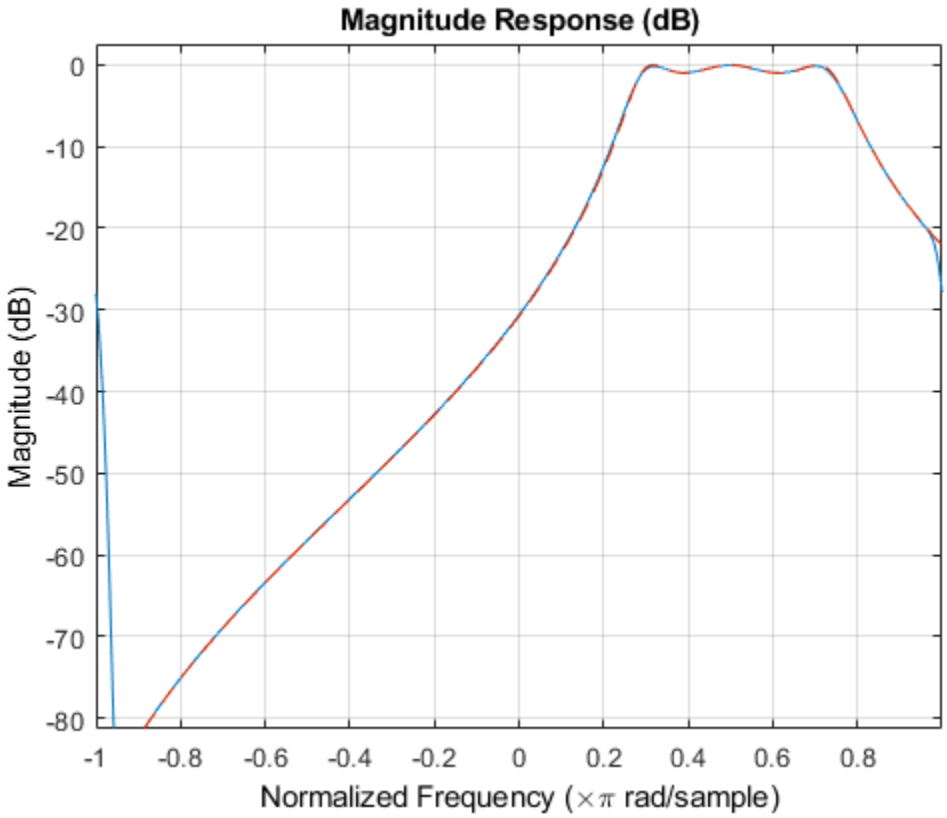
```
N = 300;
B1 = 0:0.01:0.18;
B2 = [.2 .38 .4 .55 .562 .585 .6 .78];
B3 = 0.79:0.01:1;
A1 = .5+sin(2*pi*7.5*B1)/4;    % Sinusoidal section
A2 = [.5 2.3 1 1 -.2 -.2 1 1]; % Piecewise linear section
A3 = .2+18*(1-B3).^2;          % Quadratic section
F = [B1 B2 B3];
A = [A1 A2 A3];
d = fdesign.arbmag('N,F,A',N,F,A);
Hd = design(d,'freqsamp','SystemObject',true);
fvtool(Hd,'MagnitudeDisplay','Zero-phase','Color','White');
```




```
close(gcf)
```

In the previous example, the normalized frequency points were distributed between 0 and pi rad/sample (extrema included). You can also specify negative frequencies and obtain complex filters. The following example models a complex RF bandpass filter and uses a Kaiser window to mitigate the effects of the Gibbs phenomenon that occurs due to the 70 dB magnitude gap between the -pi and pi rad/sample frequencies:

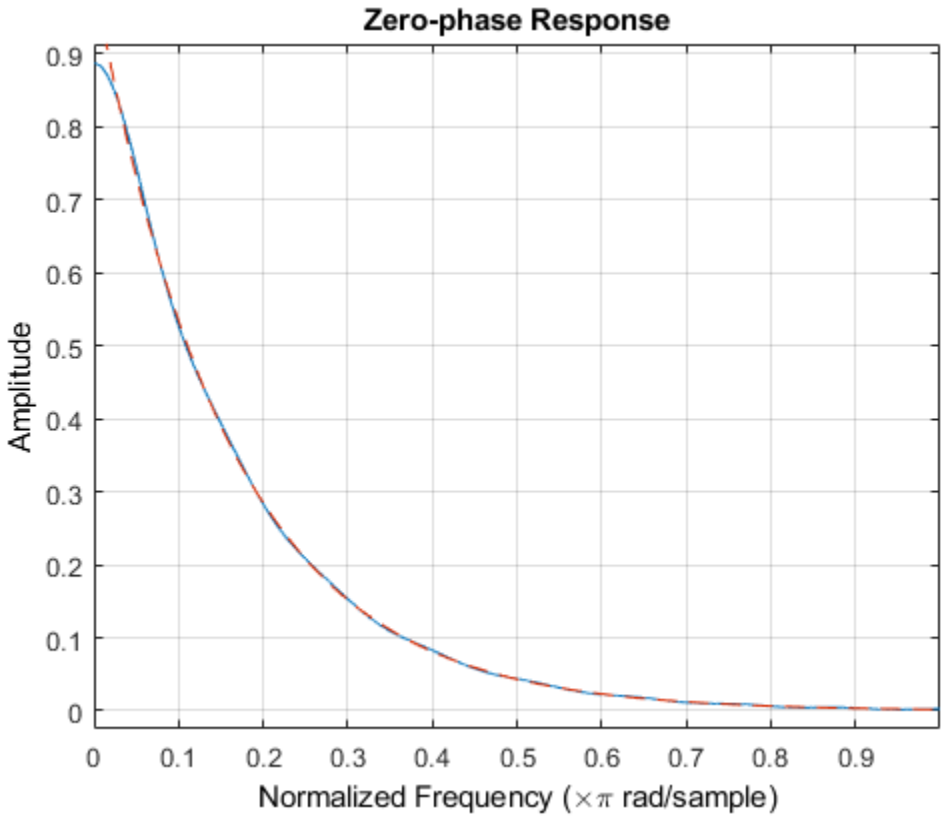
```
load cfir.mat; % load a predefined set of frequency and amplitude vectors
N = 200;
d = fdesign.arbmag('N,F,A',N,F,A);
Hd = design(d,'freqsamp', 'window' ,{@kaiser,20},'SystemObject',true);
fvtool(Hd,'FrequencyRange','[-pi, pi)','Color','White');
```



Modeling Smooth Functions with an Equiripple FIR Filter

The equiripple algorithm is well suited for modeling smooth functions as shown in the following example that models an exponential with a minimum order FIR filter. The example specifies a small ripple value across all frequencies and defines weights that increase proportionally to the desired amplitude to improve the performance at high frequencies:

```
F = linspace(0,1,100);
A = exp(-2*pi*F);
R = 0.045; % ripple
W = .1-20*log10(abs(A)); % weights
d = fdesign.arbmag('F,A,R',F,A,R);
Hd = design(d,'equiripple','weights',W,'SystemObject',true);
fvtool(Hd,'MagnitudeDisplay','Zero-phase', 'FrequencyRange','[0, pi)',...
    'Color','White');
```



Single-Band vs. Multi-Band Equiripple FIR Designs

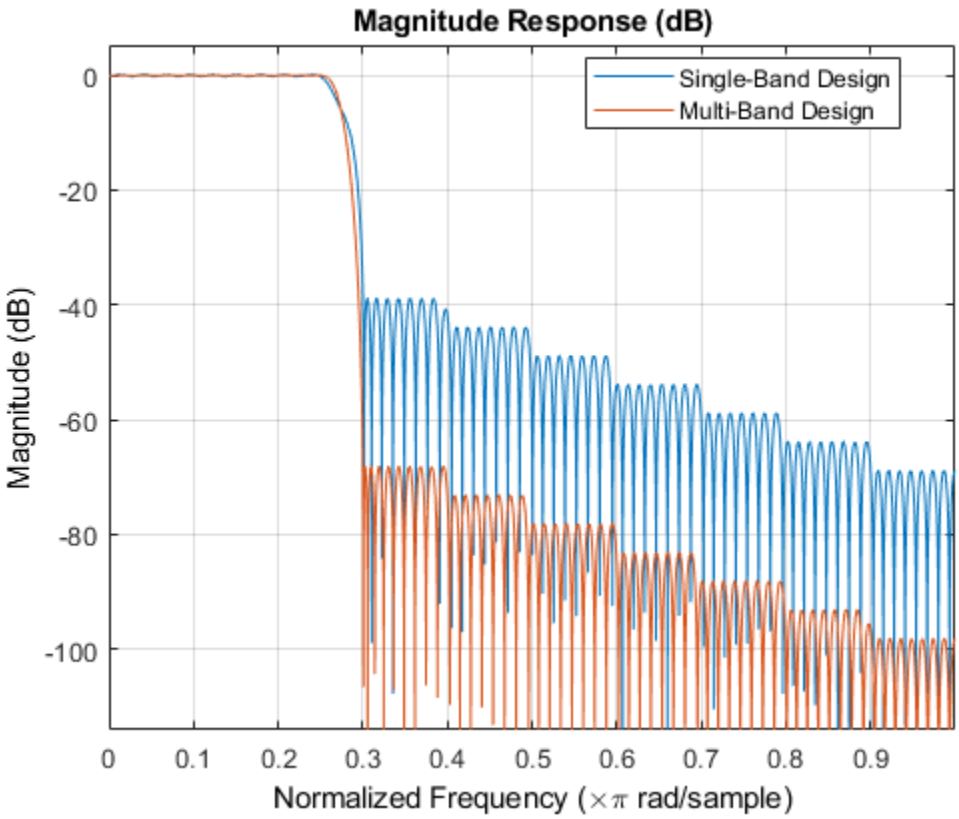
In certain applications, it might be of interest to shape the stopband of the filter to, for example, minimize the integrated side-lobe levels, or to improve the quantization robustness. The following example designs a lowpass filter with a staircase stopband. To achieve the design, it uses a distribution of weights that increase the attenuation of each step by 5 dB in the stopband:

```
N = 150;
F = [0 .25 .3 .4 .401 .5 .501 .6 .601 .7 .701 .8 .801 .9 .901 1];
A = [1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0];
W = 10.^([0 0 5 5 10 10 15 15 20 20 25 25 30 30 35 35]/20);
d = fdesign.arbmag('N,F,A',N,F,A);
Hd1 = design(d,'equiripple','weights',W,'SystemObject',true);
```

The following example presents an alternative design based on the use of a multi-band approach that defines two bands (passband and stopband) separated by a "don't care" region (or transition band):

```
B = 2;            % Number of bands
F1 = F(1:2);      % Passband
F2 = F(3:end);    % Stopband
% F(2:3)=[.25 .3] % Transition band
A1 = A(1:2);
A2 = A(3:end);
W1 = W(1:2);
W2 = W(3:end);
d = fdesign.arbmag('N,B,F,A',N,B,F1,A1,F2,A2);
Hd2 = design(d,'equiripple','B1Weights',W1,'B2Weights',W2,...
```

```
    'SystemObject',true);
hfvt = fvtool(Hd1,Hd2,'MagnitudeDisplay','Magnitude (dB)','Legend','On',...
    'Color','White');
legend(hfvt, 'Single-Band Design', 'Multi-Band Design');
```

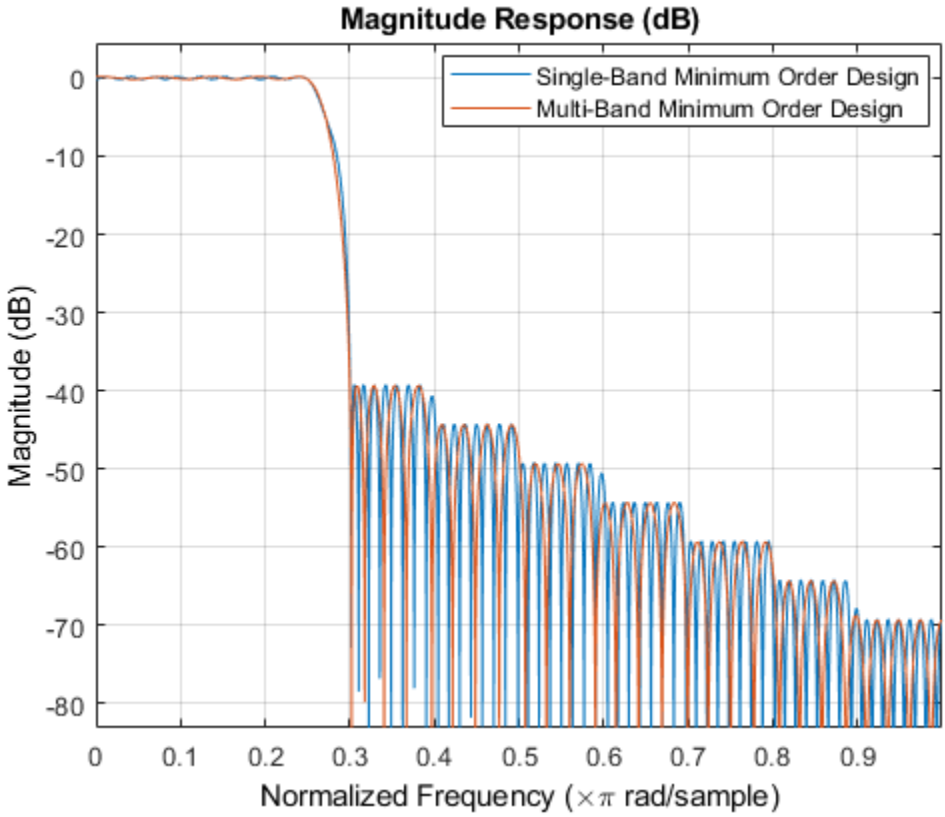


Notice the clear advantage of the multi-band approach. By relaxing constraints in the transition region, the equiripple algorithm converges to a solution with lower passband ripples and greater stopband attenuation. In other words, the frequency characteristics of the first filter could be matched with a lower order. The following example illustrates this last comment by obtaining equivalent filters using minimum order designs.

Minimum order designs require the specification of one ripple value per band. For this example, set the ripple to 0.0195 in all bands.

```
R = 0.0195;

% Single-band minimum order design
d = fdesign.arbmag('F,A,R',F,A,R);
Hd1 = design(d,'equiripple','Weights',W,'SystemObject',true);

% Multi-band minimum order design
d = fdesign.arbmag('B,F,A,R',B,F1,A1,R,F2,A2,R);
Hd2 = design(d,'equiripple','B1Weights',W1,'B2Weights',W2,...
    'SystemObject',true);

hfvt = fvtool(Hd1,Hd2,'Color','White');
```

```
legend(hfvt, 'Single-Band Minimum Order Design', ...
  'Multi-Band Minimum Order Design');
```



The passband ripple and stopband attenuation of both designs match. However, the single-band design has an order of 152 while the multi-band design has an order of 72.

```
order(Hd)
```

```
ans = 32
```

Constrained Multi-Band Equiripple Designs

Multi-band equiripple designs allow you to specify ripple constraints for different bands, specify single-frequency bands, and force specified frequency points to specified values.

Constrained Band Designs

The following example designs an 80th order passband filter with an attenuation of 60 dB in the first stopband and of 40 dB in the second stopband. By relaxing the attenuation of the second stopband, the ripple in the passband is reduced while maintaining the same filter order.

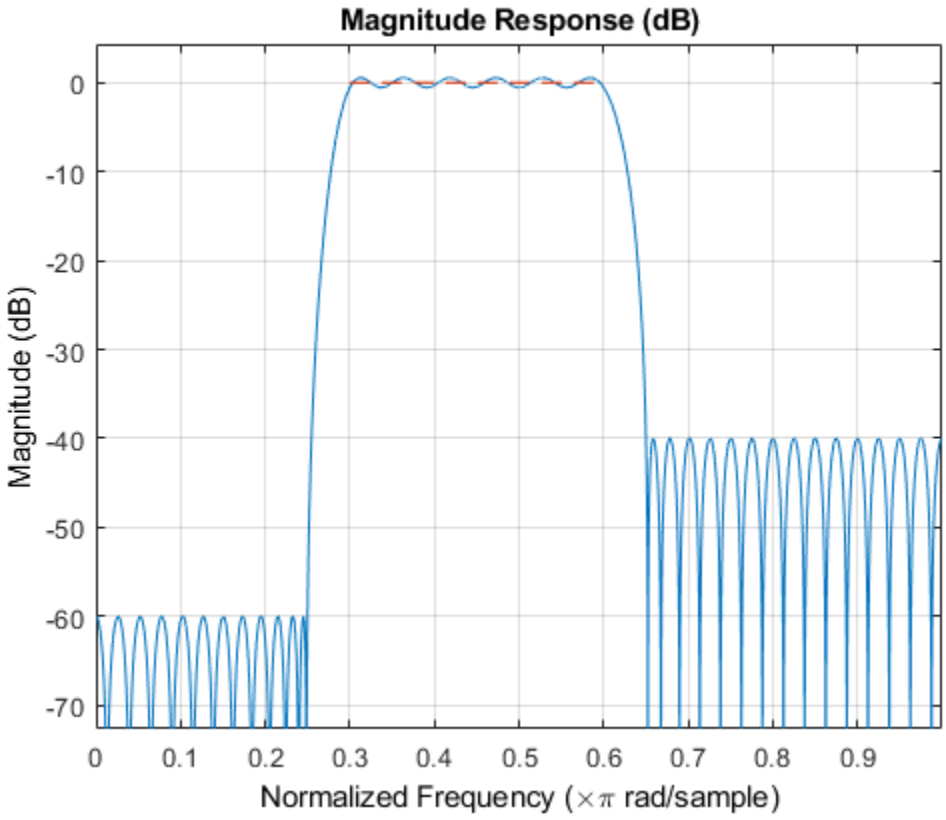
```
N = 80; % filter order
B = 3;  % number of bands

d = fdesign.arbmag('N,B,F,A,C',N,B,[0 0.25],[0 0],true,...
  [0.3 0.6],[1 1],false,[0.65 1],[0 0],true)
```

```
d =
  arbmag with properties:
```

```
            Response: 'Arbitrary Magnitude'
       Specification: 'N,B,F,A,C'
         Description: {4x1 cell}
NormalizedFrequency: 1
         FilterOrder: 80
              NBands: 3
       B1Frequencies: [0 0.2500]
        B1Amplitudes: [0 0]
       B1Constrained: 1
            B1Ripple: 0.2000
       B2Frequencies: [0.3000 0.6000]
        B2Amplitudes: [1 1]
       B2Constrained: 0
       B3Frequencies: [0.6500 1]
        B3Amplitudes: [0 0]
       B3Constrained: 1
            B3Ripple: 0.2000
```

The `B1Constrained` and `B3Constrained` properties have been set to true to specify that the first and third bands are constrained bands. Specify the ripple value for the ith constrained band using the `BiRipple` property:

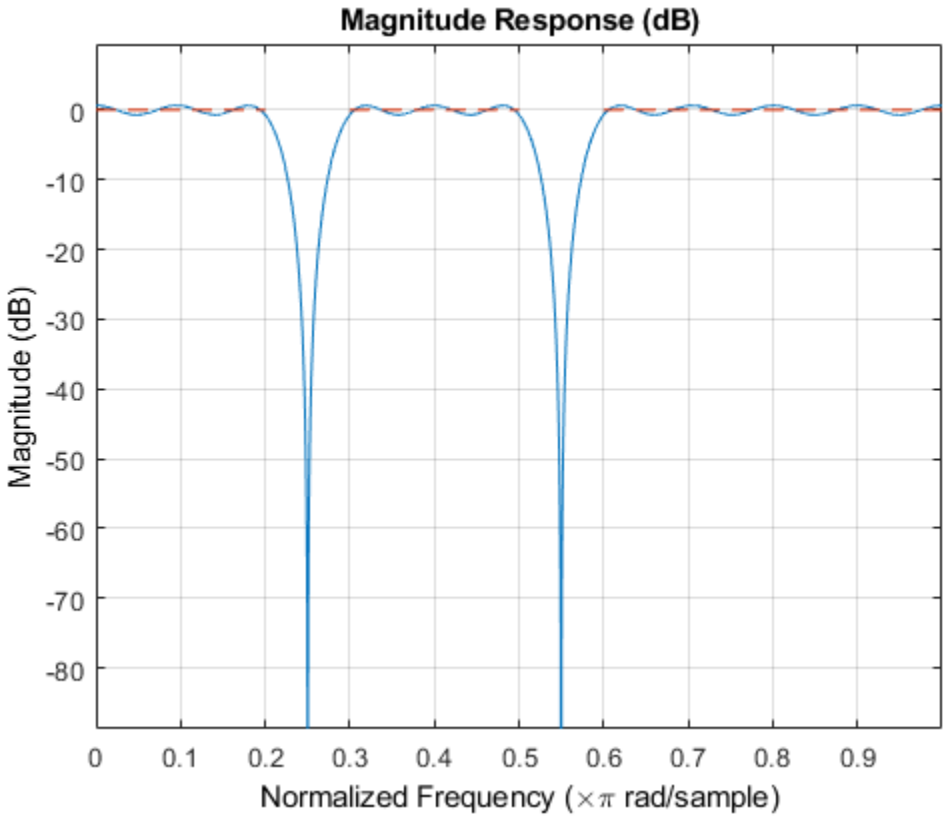
```
d.B1Ripple = 10^(-60/20); % Attenuation for the first stopband
d.B3Ripple = 10^(-40/20); % Attenuation for the second stopband

Hd = design(d,'equiripple','SystemObject',true)

Hd =
  dsp.FIRFilter with properties:

            Structure: 'Direct form'
      NumeratorSource: 'Property'
            Numerator: [-0.0036 0.0049 0.0052 -0.0022 -0.0097 -0.0044 ... ]
    InitialConditions: 0

  Show all properties


fvtool(Hd,'Legend','Off','Color','White');
```



Single-Frequency Bands

The following example designs a minimum order equiripple filter with two notches at exactly 0.25*pi and 0.55*pi rad/sample, and with a ripple of 0.15 in the passbands.

```
B = 5; % number of bands
d = fdesign.arbmag('B,F,A,R',B);

d.B1Frequencies = [0 0.2];
d.B1Amplitudes = [1 1];
d.B1Ripple = 0.15;
d.B2Frequencies = 0.25; % single-frequency band
d.B2Amplitudes = 0;
d.B3Frequencies = [0.3 0.5];
d.B3Amplitudes = [1 1];
d.B3Ripple = 0.15;
d.B4Frequencies = 0.55; % single-frequency band
d.B4Amplitudes = 0;
d.B5Frequencies = [0.6 1];
d.B5Amplitudes = [1 1];
d.B5Ripple = 0.15;
Hd = design(d,'equiripple','SystemObject',true);
fvtool(Hd,'Color','White');
```



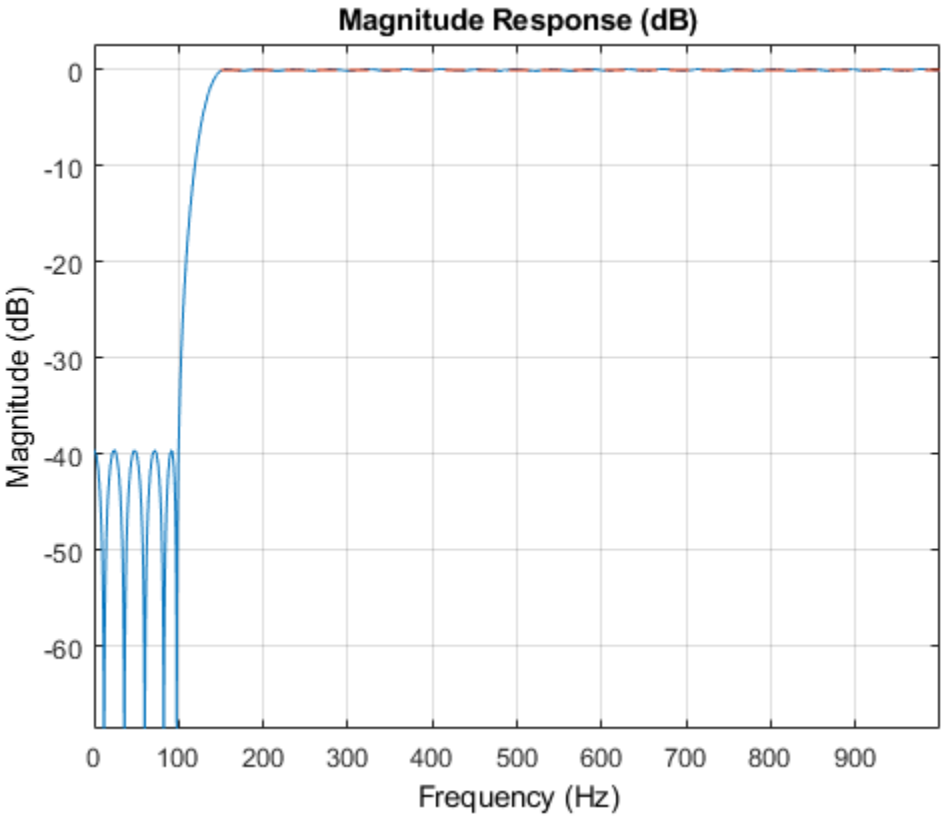
Forced Frequency Points

The following example designs a highpass filter with a stopband edge at 100 Hz, and a passband edge at 150 Hz. Suppose that you want to reject a strong 60 Hz interference without having to add an extra filter or without having to increase the filter order by a large amount. You can do this by forcing the magnitude response of the highpass filter to be 0 at 60 Hz:

```
B = 2;    % number of bands
N = 92;   % filter order
Fs = 2e3; % sampling frequency
d = fdesign.arbmag('N,B,F,A',N,B,[0 60 100],[0 0 0],[150 1000],[1 1],Fs);
```

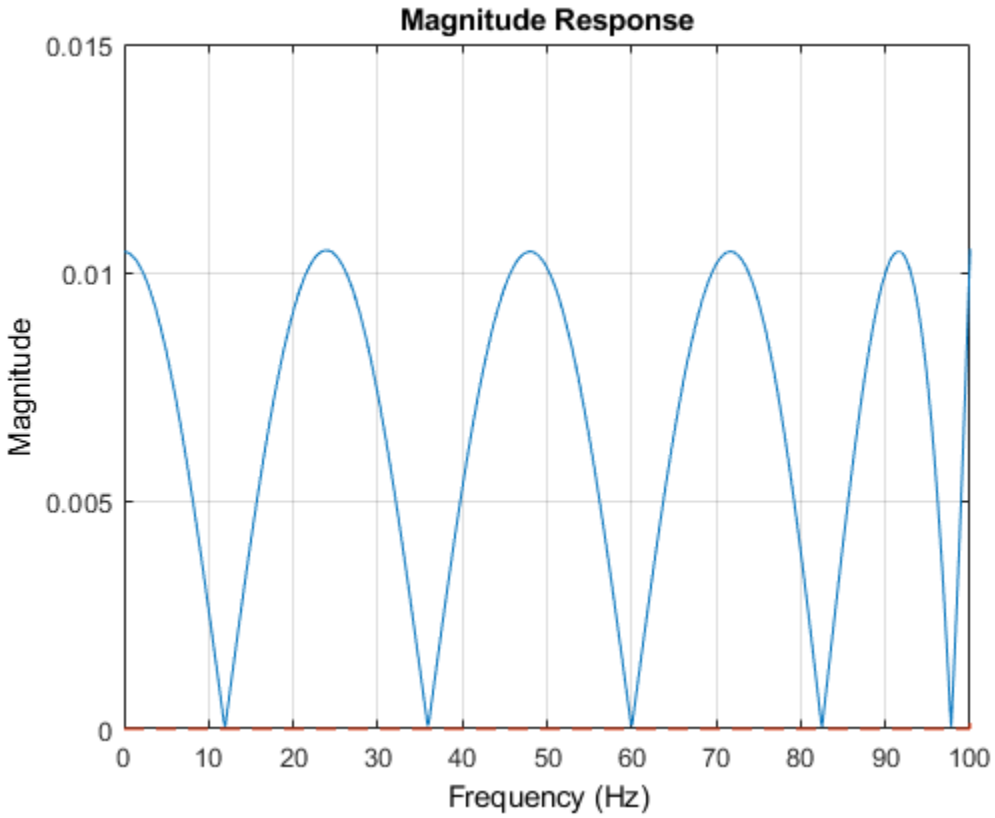
Use the `B1ForcedFrequencyPoints` design option to force the 60 Hz point to its specified amplitude value.

```
Hd = design(d,'equiripple','B1ForcedFrequencyPoints',60,...
    'SystemObject',true);
hfvt = fvtool(Hd,'Fs', Fs,'Color','White');
```



Zoom into the stopband of the highpass filter to observe that the amplitude is zero at the specified 60 Hz frequency point:

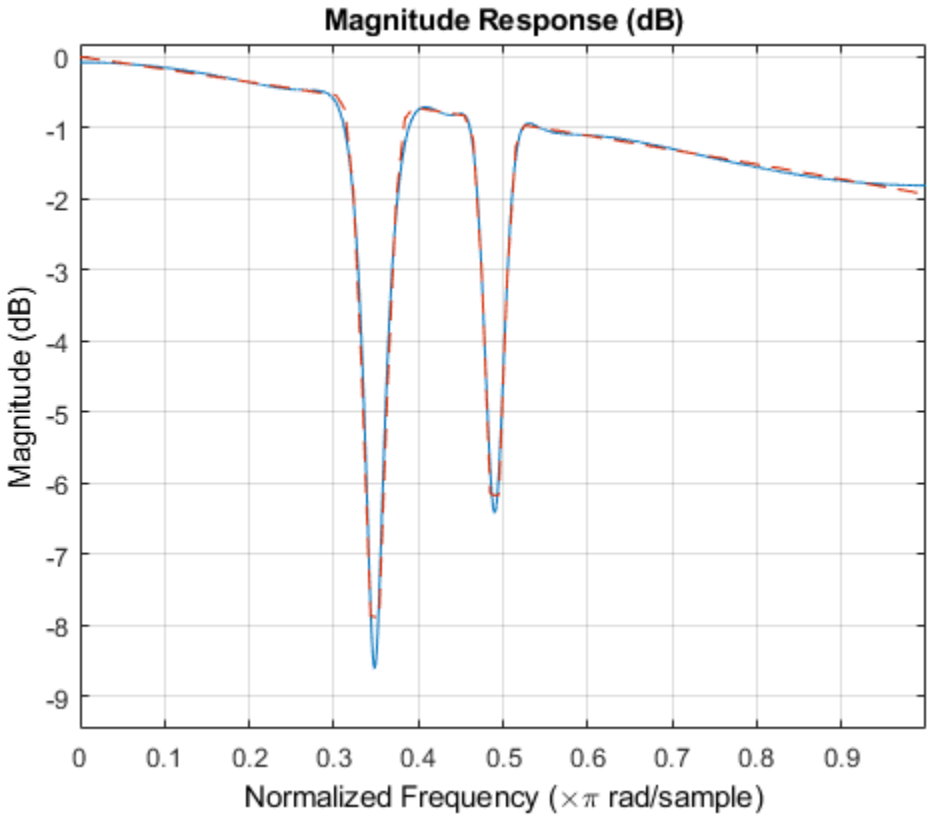
```
hfvt.MagnitudeDisplay = 'Magnitude';
xlim([0 100])
ylim([0 0.015])
```



Single-Band vs. Multi-Band IIR Designs

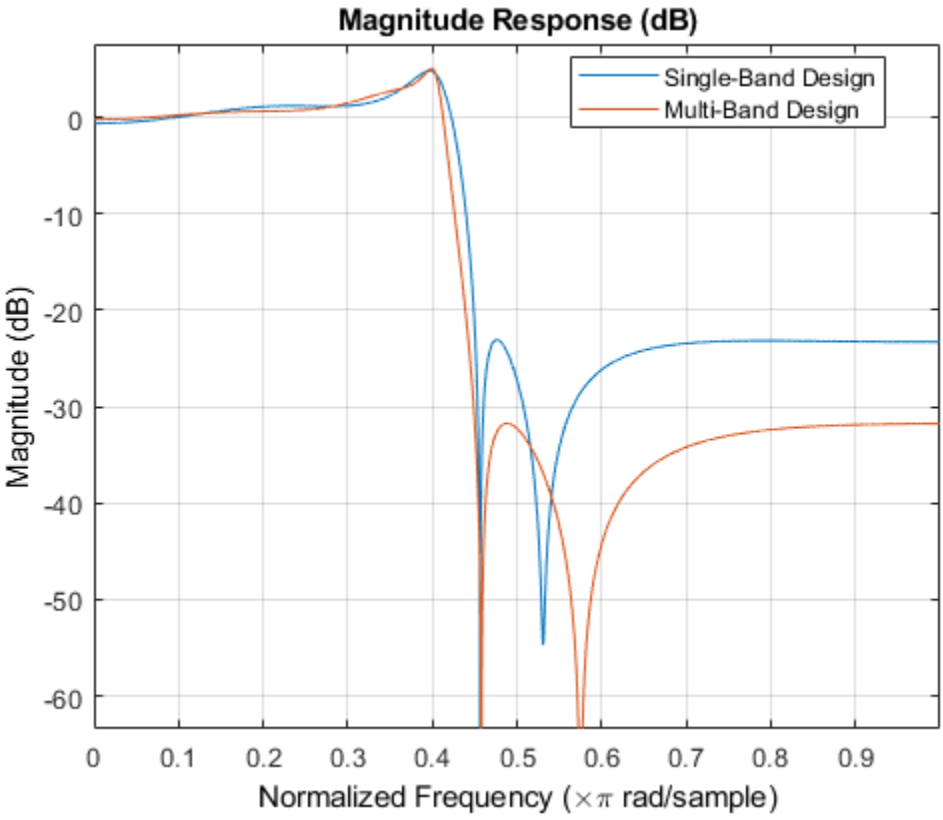
As in the FIR case, IIR design problems where a transition band cannot be easily identified are best resolved with a single (full) band specification approach. As an example, model the optical absorption of a gas (atomic Rubidium87 vapor):

```
Nb = 12;
Na = 10;
F = linspace(0,1,100);
As = ones(1,100)-F*0.2;
Absorb = [ones(1,30),(1-0.6*bohmanwin(10))',...
    ones(1,5), (1-0.5*bohmanwin(8))',ones(1,47)];
A = As.*Absorb;
d = fdesign.arbmag('Nb,Na,F,A',Nb,Na,F,A);
W = [ones(1,30) ones(1,10)*.2 ones(1,60)];
Hd = design(d, 'iirlpnorm', 'Weights', W, 'Norm', 2, 'DensityFactor',30,...
    'SystemObject',true);
fvtool(Hd, 'MagnitudeDisplay','Magnitude (dB)', ...
    'NormalizedFrequency','On','Color','White');
```



In other cases where constraints can be relaxed in one or more transition bands, the multi-band approach provides the same benefits as in the FIR case (namely better passband and stopband characteristics). The following example illustrates these differences by modeling a Rayleigh fading wireless communications channel:

```
Nb = 4;
Na = 6;
F = [0:0.01:0.4 .45 1];
A = [1.0./(1-(F(1:end-2)./0.42).^2).^0.25 0 0];
d = fdesign.arbmag('Nb,Na,F,A',Nb,Na,F,A); % single-band design
Hd1 = design(d,'iirlpnorm','SystemObject',true);

B = 2;
F1 = F(1:end-2);          % Passband
F2 = F(end-1:end);        % Stopband
% F(end-2:end-1)=[.4 .45] % Transition band
A1 = A(1:end-2);
A2 = A(end-1:end);
d = fdesign.arbmag('Nb,Na,B,F,A',Nb,Na,B,F1,A1,F2,A2); % multi-band design
Hd2 = design(d,'iirlpnorm','SystemObject',true);
hfvt = fvtool(Hd1,Hd2,'Color','White');
legend(hfvt, 'Single-Band Design', 'Multi-Band Design');
```


Magnitude Response (dB)
Single-Band Design
Multi-Band Design
Magnitude (dB)
0
-10
-20
-30
-40
-50
-60
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Normalized Frequency (×π rad/sample)

Design of Peaking and Notching Filters

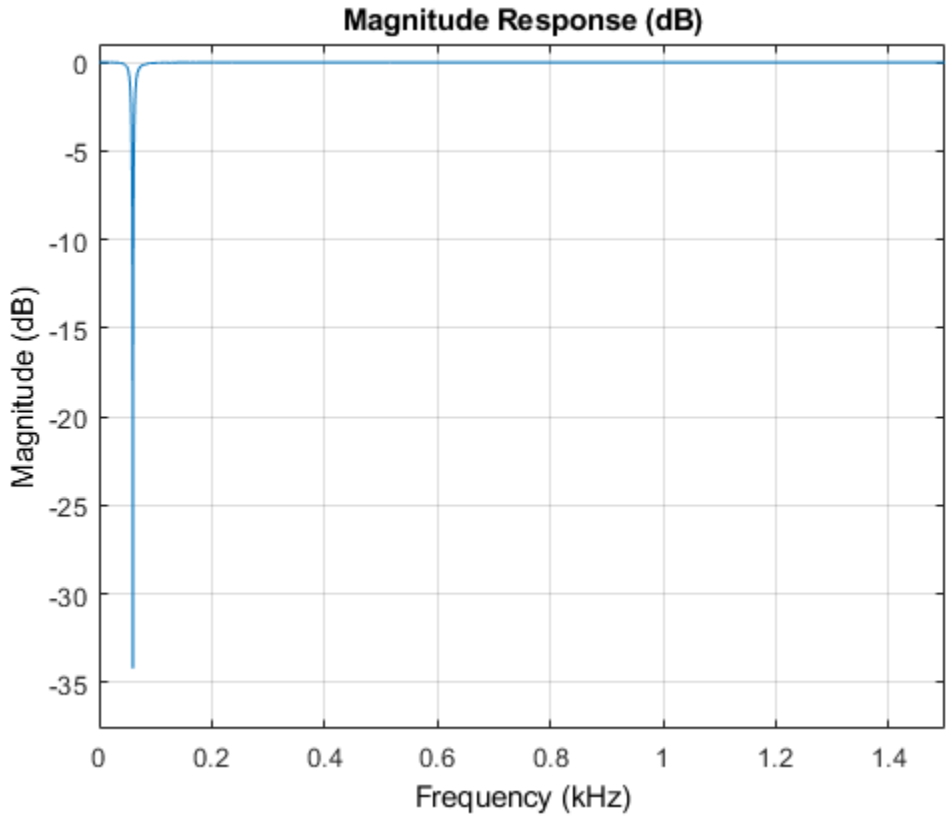
This example shows how to design peaking and notching filters. Filters that peak or notch at a certain frequency retain or eliminate a particular frequency component of a signal. The design parameters for the filter are the frequency at which the peak or notch is desired, and either the 3-dB bandwidth or the filter's Q factor. Moreover, given these specifications, by increasing the filter order, it is possible to obtain designs that more closely approximate an ideal filter.

Second Order Notch Filters

Suppose you need to eliminate a 60 Hz interference in a signal sampled at 3000 Hz. A notch filter can be used for such a purpose. The function `iirnotch` can be used to compute the coefficients of a second order notch filter.

Here is an example:

```
F0 = 60;   % Interference is at 60 Hz
Fs = 3000; % Sampling frequency is 3000 Hz
BW = 6;    % Choose a bandwidth factor of 6Hz
[num1,den1] = iirnotch(F0/(Fs/2),BW/(Fs/2));
fvtool(num1,den1,'Fs',Fs,'Color','white');
```

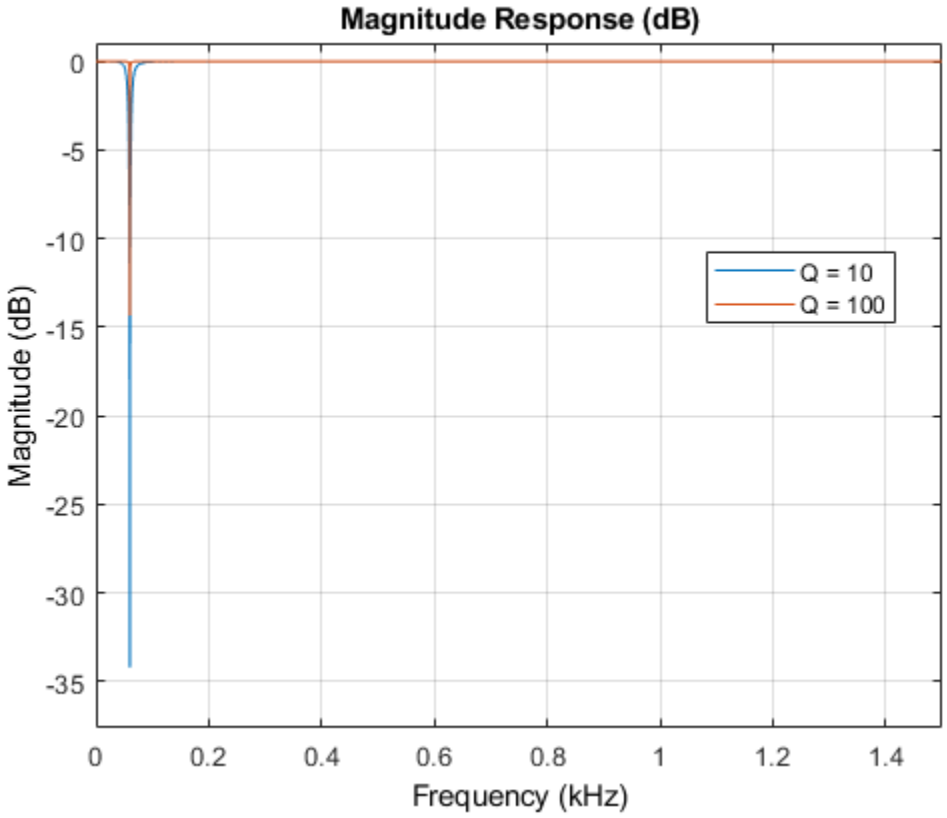


An equivalent way of designing the filter is to specify the quality factor and obtain the 3 dB bandwidth. Quality factor is defined as the ratio of the notch or peak frequency F_0 and the 3 dB bandwidth BW. Mathematically, Q factor is given by $Q = F_0/BW$. In the above case, the value of the quality factor is 10. Specifying the bandwidth is a more convenient way of achieving exactly the

desired shape for the designed filter. Ther Q factor of the filter is a measure of how well the desired frequency is isolated from other frequencies. For a fixed filter order, a higher Q factor is accomplished by pushing the poles closer to the zeros.

Visualize the magnitude response of the filter using `fvtool`.

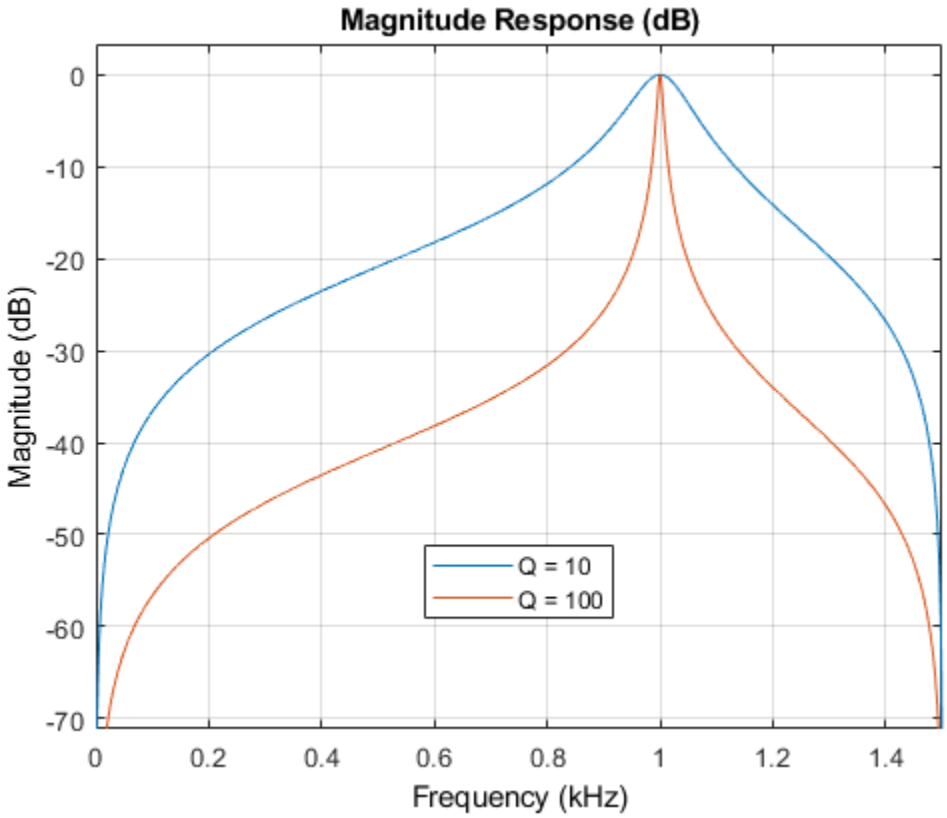
```
Q2 = 100;    % Choose a Q factor of 100
[num2,den2] = iirnotch(F0/(Fs/2),F0/(Q2*Fs/2));
fvt = fvtool(num1,den1,num2,den2,'Fs',Fs,'Color','white');
legend(fvt,'Q = 10','Q = 100');
```



Second order Peak Filters

Peaking filters are used to retain only a single frequency component (or a small band of frequencies) from a signal. The `iirpeak` function can be used to compute the coefficients of a second order peak filter.

```
F0 = 1000;   % Interference is at 60 Hz
Fs = 3000;   % Sampling frequency is 3000 Hz
Q1 = 10;
[num1,den1] = iirpeak(F0/(Fs/2),F0/(Q1*Fs/2));
Q2 = 100;
[num2,den2] = iirpeak(F0/(Fs/2),F0/(Q2*Fs/2));
fvt = fvtool(num1,den1,num2,den2,'Fs',Fs,'Color','white');
legend(fvt,'Q = 10','Q = 100');
```



Time varying Second Order Notch filter Implementation

Using time-varying filters requires changing the coefficients of the filter while the simulation runs. The DSP System Toolbox™ provides certain features such as the `iirnotch` function and the `dsp.NotchPeakFilter` object to design time-varying tunable notch filters. These features compute the filter coefficients directly.

Dynamic simulation with static filter

In order to implement a time varying filter, create a dynamic setup to simulate the filter and implement the filter with time-varying design parameters.

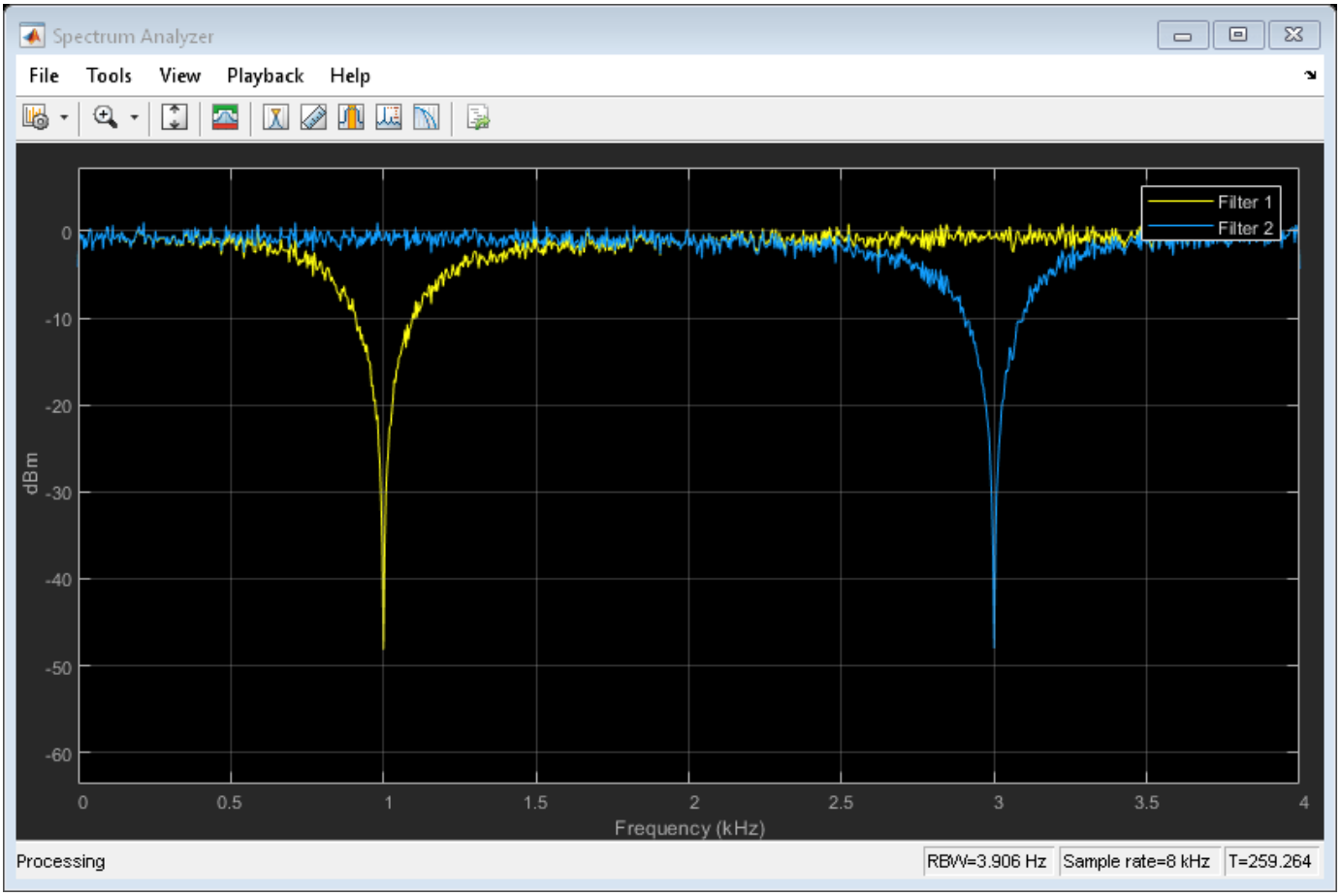
Start by creating a dynamic (streamed) simulation with filters whose coefficients do not change. Create two second-order notch filters, one using the `dsp.SOSFilter` object and the second using the `dsp.NotchFilter` object. In the first filter, set the center frequency to 1 kHz, and the bandwidth at -3 dB to 50 Hz. Calculate the coefficients of this filter directly using the `iirnotch` function. In the second filter, set the center frequency to 3 kHz and the bandwidth at -3 dB to 50 Hz. The sample rate for both filters is 8 kHz.

```
Fs = 8e3;    % 8 kHz sampling frequency
F01 = 1e3;   % Notch at 1 kHz for Filter 1
BW = 500;    % 500 Hz bandwidth for both filters
[b, a] = iirnotch(F01/(Fs/2), BW/(Fs/2))    % Filter 1 coefficients
```

```
b = 1×3
```

```
    0.8341   -1.1796    0.8341

a = 1×3

    1.0000   -1.1796    0.6682
```

```
sosFilter = dsp.SOSFilter(b,a);

F02 = 3e3;    % Notch at 3 kHz for Filter 2
npFilter = dsp.NotchPeakFilter('CenterFrequency',F02,'Bandwidth',BW,...
    'SampleRate',Fs);

scope = dsp.SpectrumAnalyzer('PlotAsTwoSidedSpectrum', false, ...
    'SampleRate', Fs, ...
    'AveragingMethod',"Exponential",...
    'ForgettingFactor',.95,...
    'Method','Filter bank',...
    'ChannelNames',{'Filter 1','Filter 2'},...
    'ShowLegend',true);

samplesPerFrame = 256;
nFrames = 8192;
for k = 1:nFrames
   x = randn(samplesPerFrame, 1);
   y1 = sosFilter(x);
   y2 = npFilter(x);
   scope([y1,y2]);
end
```

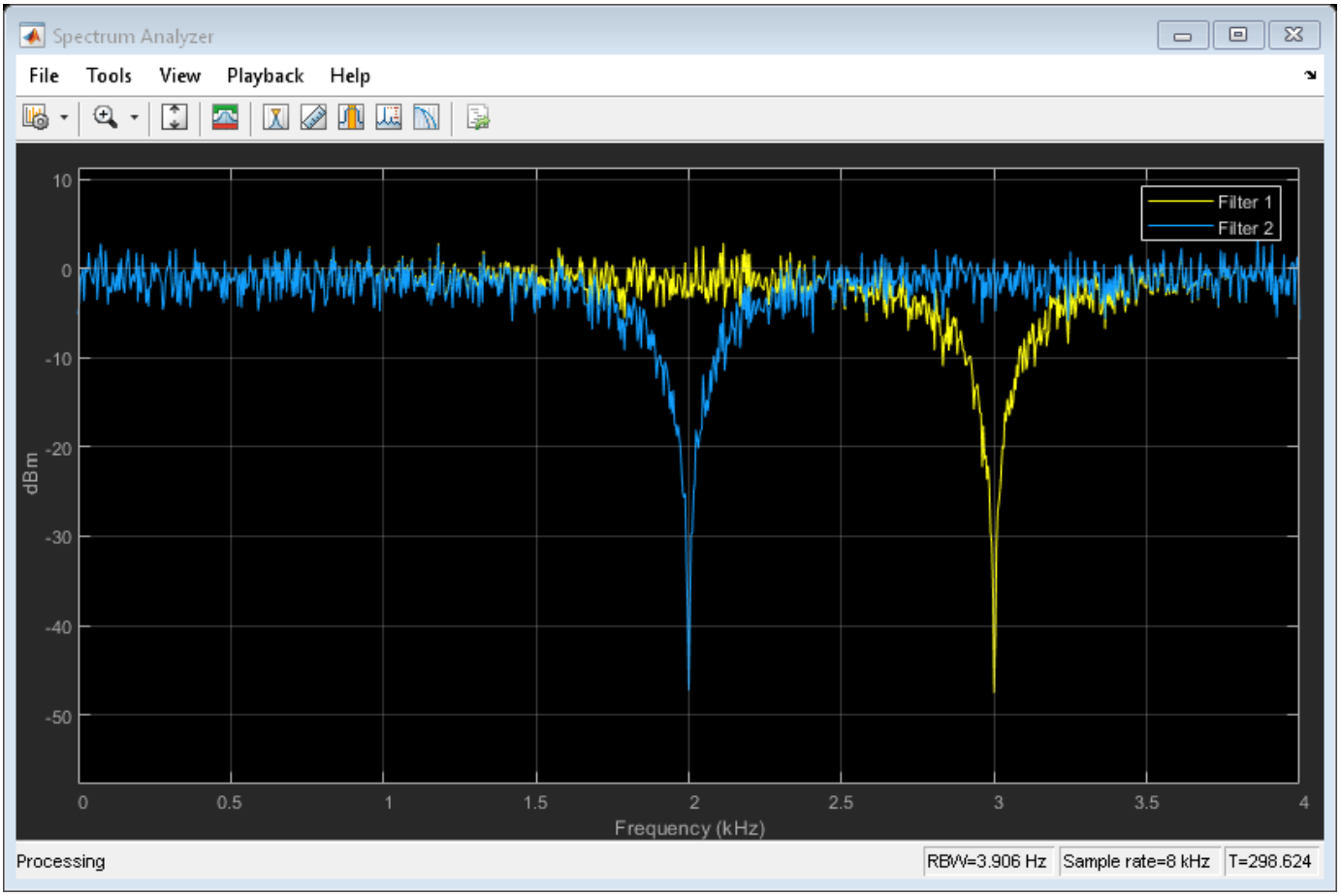


Dynamic simulation with time-varying filter

For a time-varying filter, the coefficients of time-varying filters change over time due to runtime changes in the design parameters (for example the center frequency for a notch filter). Create two second order notch filters with time varying design parameters. Similar to the above example, use the `iirnotch` function and the `dsp.SOSFilter` object to implement the first filter, and the `dsp.NotchFilter` object to implement the second filter. Vary the design parameters of both filters over time.

```
% Notch filter parameters - how they vary over time
Fs = 8e3;                          % 8 kHz sampling frequency
F01 = 1e3 * [0.5, 1, 1.5, 3];   % Notch frequencies for Filter 1
F02 = 1e3 * [3.5, 3, 2.5, 2];   % Notch frequencies for Filter 2
BW = 500 * ones(1,4);              % 500 Hz bandwidth for both filters

myChangingParams1 = struct('f0', num2cell(F01/(Fs/2)), 'bw', num2cell(BW/(Fs/2)));
myChangingParams2 = struct('F0', num2cell(F02), 'BW', num2cell(BW));
paramsChangeTimes = [0, 70, 140, 210]; % in seconds

% Simulation time management
nSamplesPerFrame = 256;
tEnd = 300;
nSamples = ceil(tEnd * Fs);
nFrames = floor(nSamples / nSamplesPerFrame);
```

```
% Object creation
sosFilter = dsp.SOSFilter; %Filter 1 object
npFilter = dsp.NotchPeakFilter('SampleRate',Fs);
scope = dsp.SpectrumAnalyzer('PlotAsTwoSidedSpectrum', false, ...
    'SampleRate', Fs, ...
    'AveragingMethod',"Exponential",...
    'ForgettingFactor',.75,......
    'RBWSource', 'Auto',...
    'Method','Filter bank',...
    'ChannelNames',{'Filter 1','Filter 2'},...
    'ShowLegend',true);
paramtbl1 = ParameterTimeTable('Time', paramsChangeTimes, ...
    'Values', myChangingParams1, ...
    'SampleRate', Fs/nSamplesPerFrame);
paramtbl2 = ParameterTimeTable('Time', paramsChangeTimes, ...
    'Values', myChangingParams2, ...
    'SampleRate', Fs/nSamplesPerFrame);

% Actual simulation loop
for frameIdx = 1:nFrames
    % Get current F0 and BW
    [params1, update1] = paramtbl1();
    [params2, update2] = paramtbl2();
    if(update1)
        % Recompute filter coefficients if parameters changed
        [b, a] = iirnotch(params1.f0, params1.bw);
        % Set filter coefficients to new values
        sosFilter.Numerator = b;
        sosFilter.Denominator = a;
    end
    if(update2)
        npFilter.CenterFrequency = params2.F0;
        npFilter.Bandwidth = params2.BW;
    end
    % Generate vector of white noise samples
    x = randn(nSamplesPerFrame, 1);
    % Filter noise
    y1 = sosFilter(x);
    y2 = npFilter(x);
    % Visualize spectrum
    scope([y1,y2]);
end
```



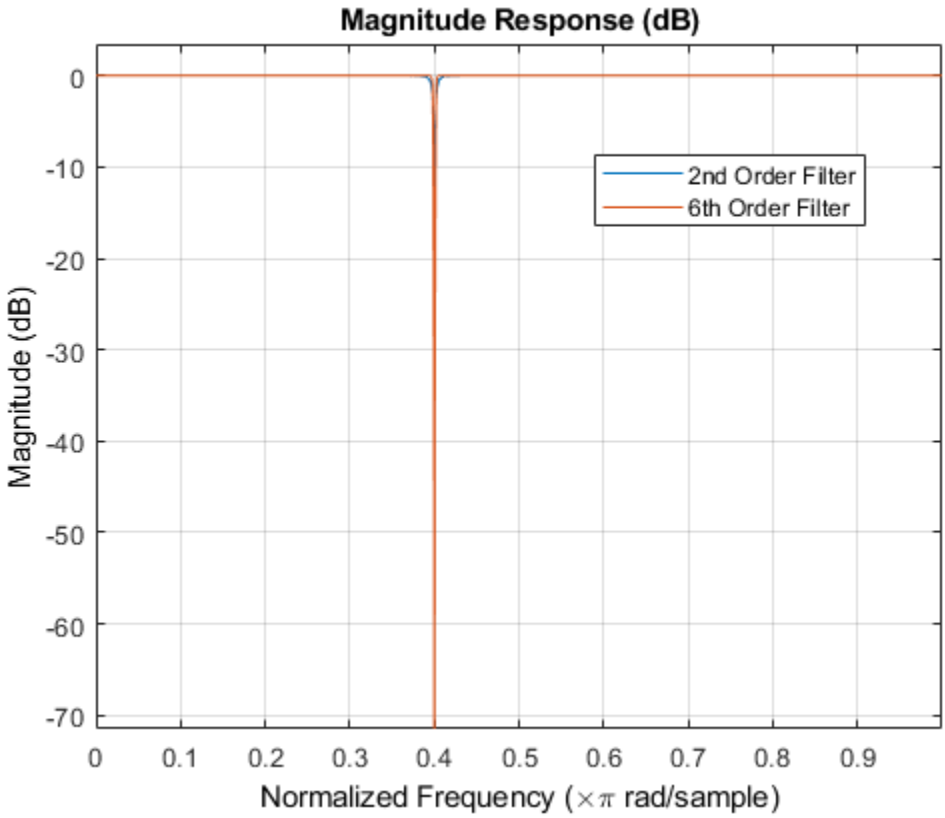
A tunable peaking filter can be implemented similarly using the `dsp.NotchPeakFilter` object or using the `iirpeak` function and `dsp.SOSFilter` object.

Note: These tunable peaking and notching filters support code generation.

Higher order Notch Filter

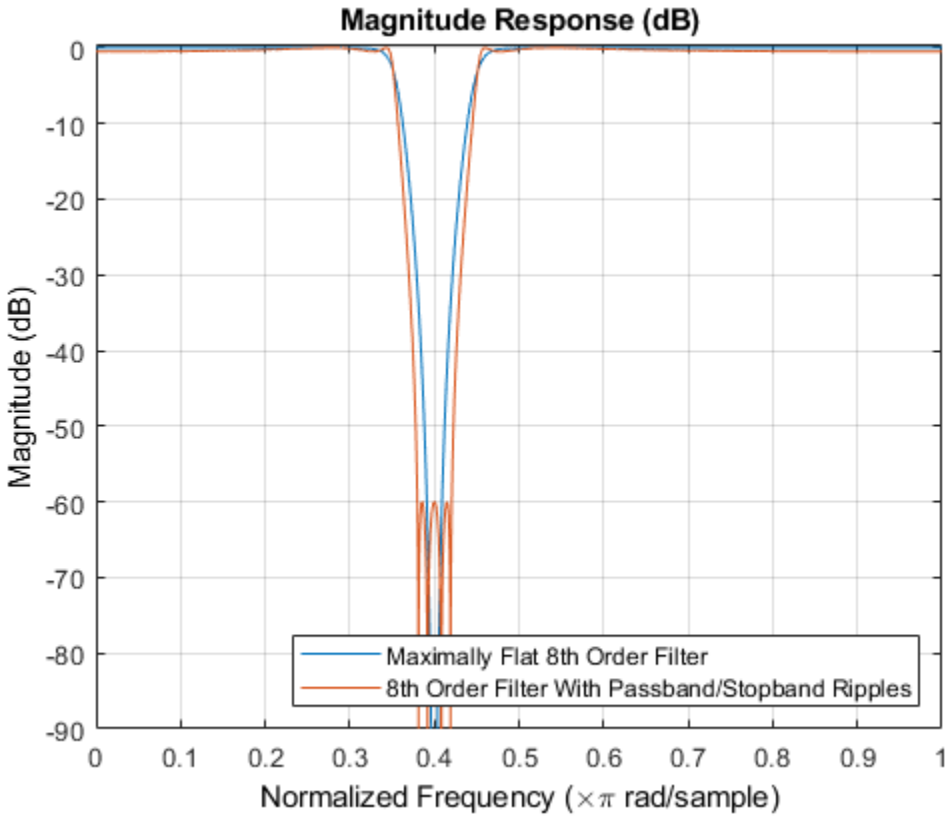
Since it is only possible to push the poles so far and remain stable, in order to improve the brickwall approximation of the filter, it is necessary to increase the filter order. A higher order notch filter can be designed using `fdesign.notch` filter specification object.

```
notchspec = fdesign.notch('N,F0,Q',2,.4,100);
notchfilt = design(notchspec,'SystemObject',true);
notchspec.FilterOrder = 6;
notchfilt1 = design(notchspec,'SystemObject',true);
fvt= fvtool(notchfilt, notchfilt1, 'Color','white');
legend(fvt,'2nd Order Filter','6th Order Filter');
```



For a given order, we can obtain sharper transitions by allowing for passband and/or stopband ripples.

```
N = 8; F0 = 0.4; BW = 0.1;
notchspec = fdesign.notch('N,F0,BW',N,F0,BW);
notchfilt = design(notchspec,'SystemObject',true);
notchspec1 = fdesign.notch('N,F0,BW,Ap,Ast',N,F0,BW,0.5,60);
notchfilt1 = design(notchspec1,'SystemObject',true);
fvt= fvtool(notchfilt, notchfilt1, 'Color','white');
legend(fvt,'Maximally Flat 8th Order Filter',...
    '8th Order Filter With Passband/Stopband Ripples', ...
    'Location','SouthEast');
axis([0 1 -90 0.5]);
```

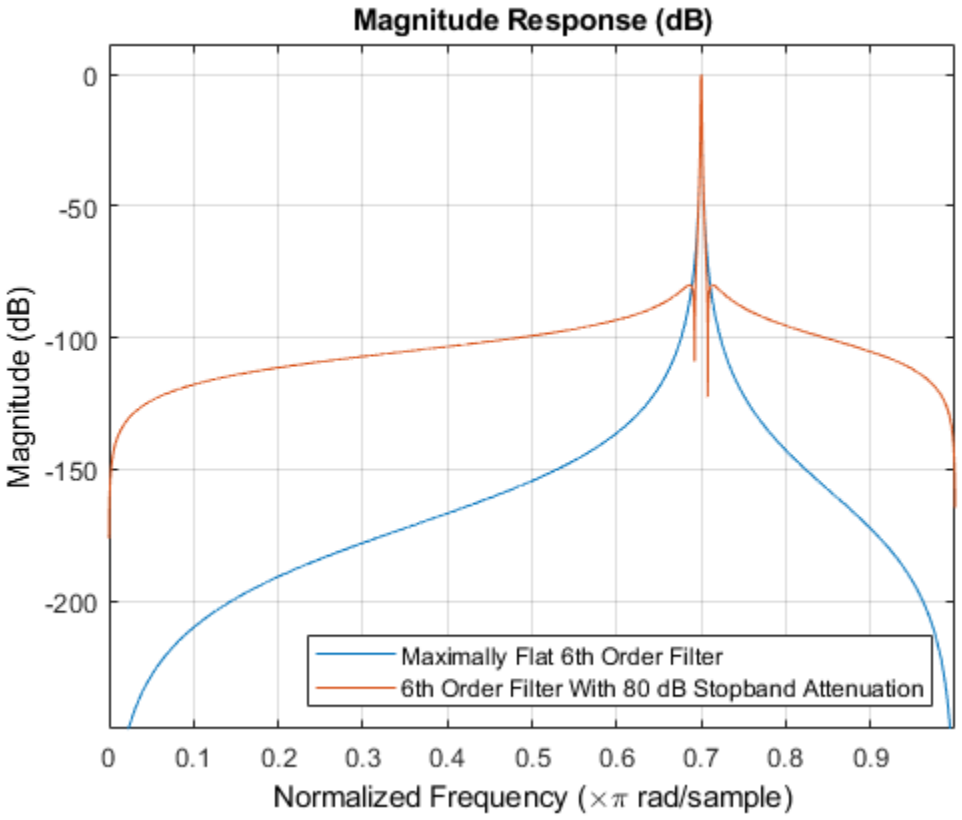


Higher order Peak Filters

A higher order peak filter can be designed using `fdesign.peak` filter specification object. All specifications and tradeoffs mentioned so far apply equally to peaking filters.

Here's an example of a higher order peaking filter:

```
N = 6; F0 = 0.7; BW = 0.001;
peakspec = fdesign.peak('N,F0,BW',N,F0,BW);
peakfilt = design(peakspec,'SystemObject',true);
peakspec1 = fdesign.peak('N,F0,BW,Ast',N,F0,BW,80);
peakfilt1 = design(peakspec1,'SystemObject',true);
fvt= fvtool(peakfilt, peakfilt1, 'Color','white');
legend(fvt,'Maximally Flat 6th Order Filter',...
    '6th Order Filter With 80 dB Stopband Attenuation','Location','SouthEast');
```


Magnitude Response (dB)
Magnitude (dB)
0
-50
-100
-150
-200
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Normalized Frequency (×π rad/sample)
Maximally Flat 6th Order Filter
6th Order Filter With 80 dB Stopband Attenuation

Fractional Delay Filters Using Farrow Structures

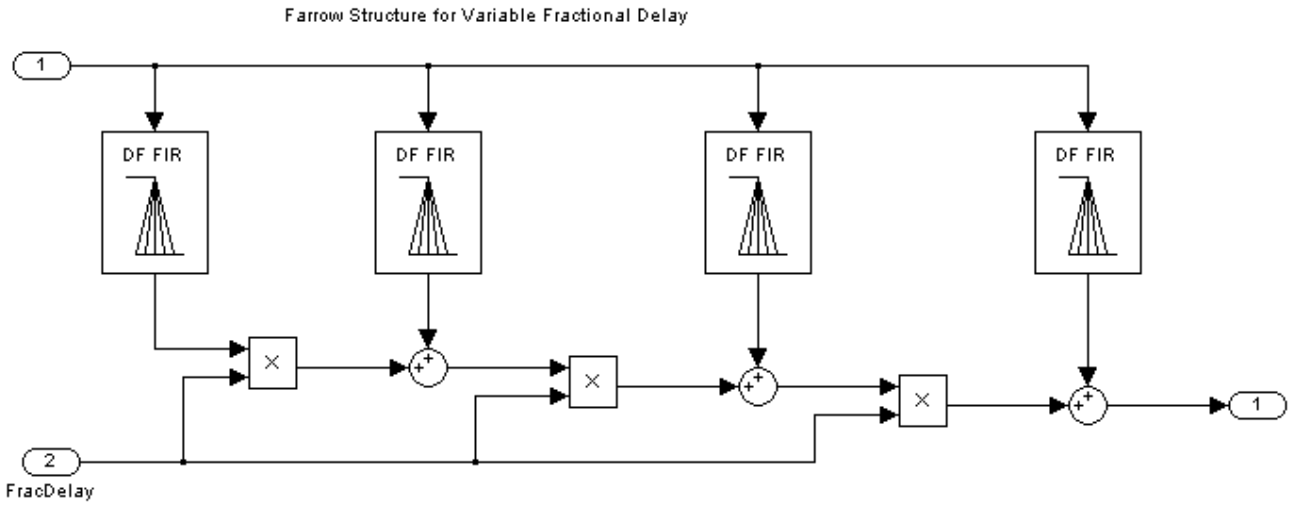
This example shows how to design digital fractional delay filters that are implemented using Farrow structures. Digital fractional delay (fracDelay) filters are useful tools to fine-tune the sampling instants of signals. They are, for example, typically found in the synchronization of digital modems where the delay parameter varies over time. This example illustrates the Farrow structure, a popular method for implementing time-varying FIR fracDelay filters.

Ideal Fractional Delay Filter

The ideal fractional delay filter is a linear phase allpass filter. Its impulse response is a time-shifted discrete sinc function that corresponds to a non causal filter. Since the impulse response is infinite, it cannot be made causal by a finite shift in time. It is therefore non realizable and must be approximated.

The Farrow Structure

To compute the output of a fractional delay filter, we need to estimate the values of the input signal between the existing discrete-time samples. Special interpolation filters can be used to compute new sample values at arbitrary points. Among those, polynomial-based filters are of particular interest because a special structure - the Farrow structure - permits simple handling of coefficients. In particular, the tunability of the Farrow structure makes its well-suited for practical hardware implementations.

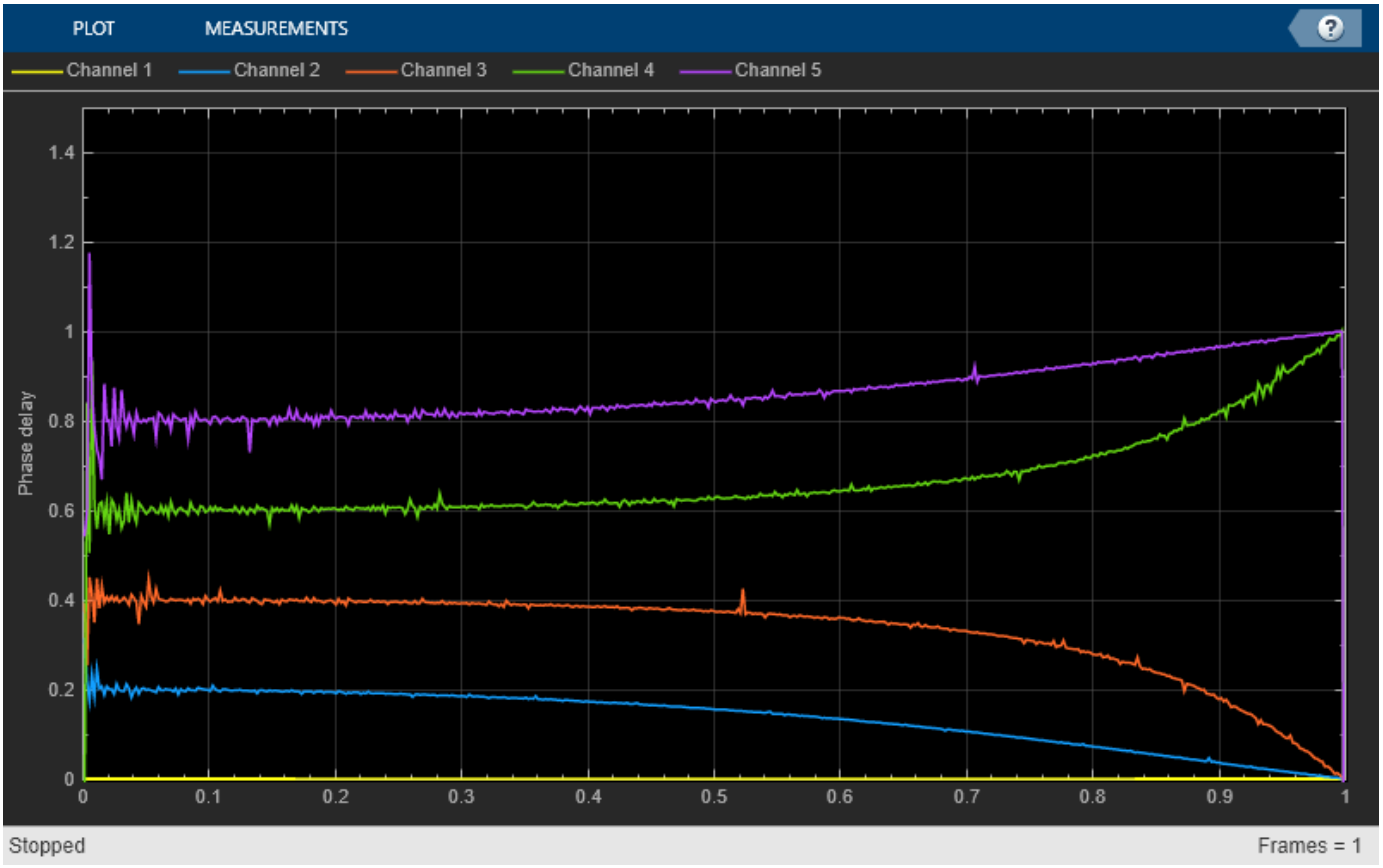


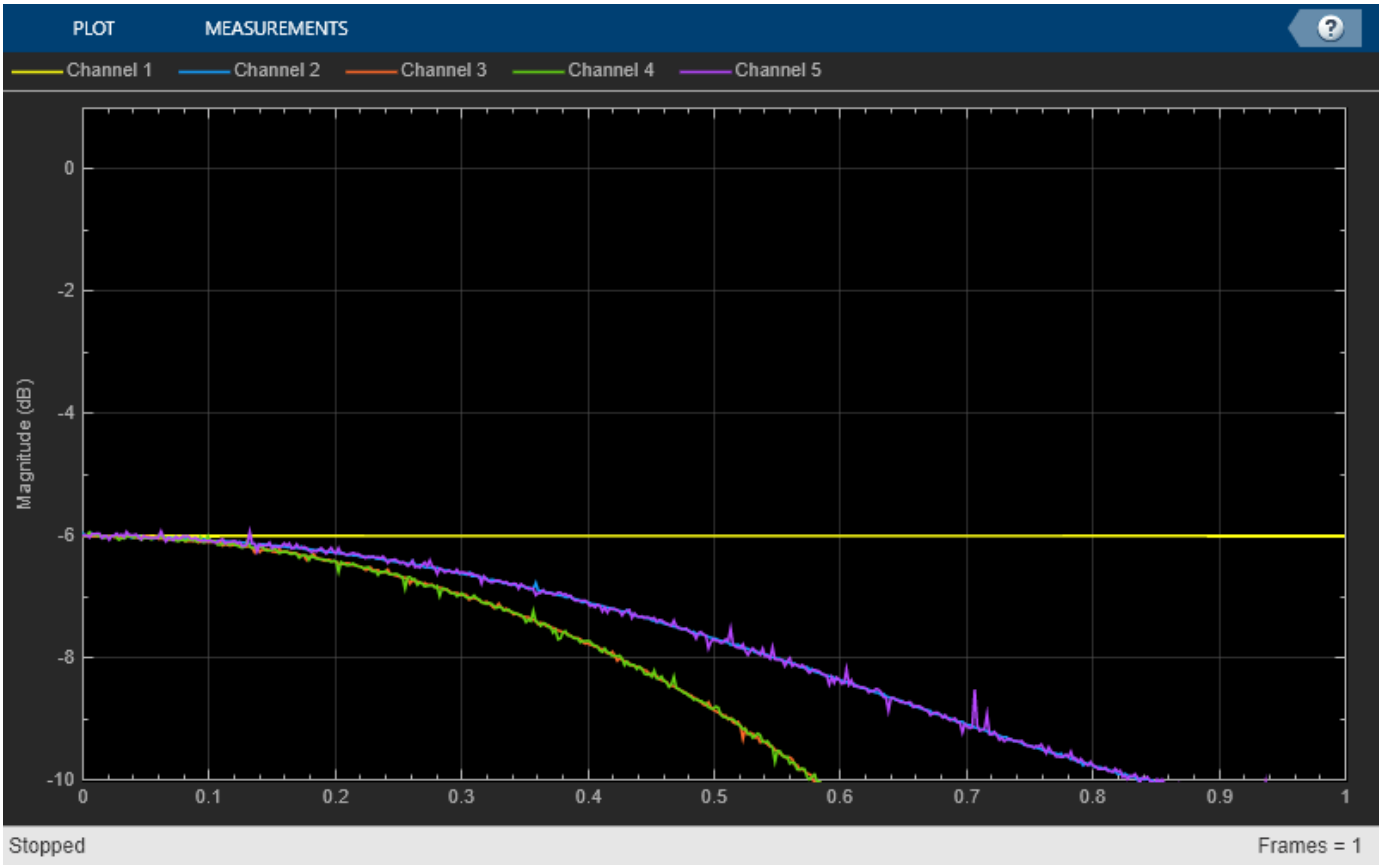
Maximally-Flat FIR Approximation (Lagrange Interpolation)

Lagrange interpolation is a time-domain approach that leads to a special case of polynomial-based filters. The output signal is approximated with a polynomial of degree M. The simplest case (M=1) corresponds to linear interpolation. Let's design and analyze a linear fractional delay filter that will split the unit delay by various fractions:

```
Nx = 1024;
Nf = 5;
yw = zeros(Nx,Nf);
```

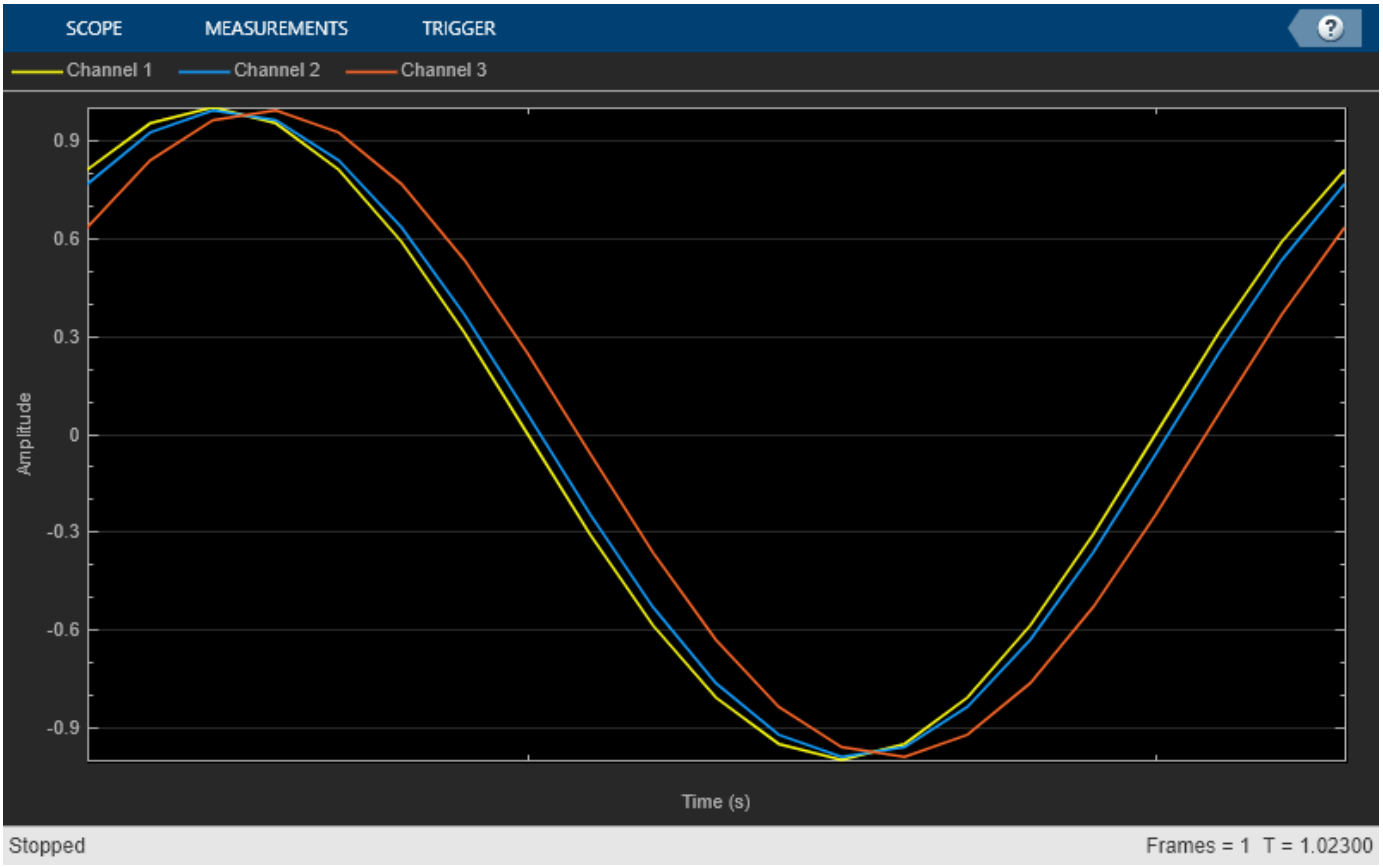
```
transferFuncEstimator = dsp.TransferFunctionEstimator( ...
    'SpectralAverages',25,'FrequencyRange','onesided');
arrPlotPhaseDelay = dsp.ArrayPlot('PlotType','Line','YLimits',[0 1.5], ...
    'YLabel','Phase delay','SampleIncrement',1/512);
arrPlotMag = dsp.ArrayPlot('PlotType','Line','YLimits',[-10 1], ...
    'YLabel','Magnitude (dB)','SampleIncrement',1/512);

fracDelay = dsp.VariableFractionalDelay;

xw = randn(Nx,Nf);
transferFuncEstimator(xw,yw);
w = getFrequencyVector(transferFuncEstimator,2*pi);
w = repmat(w,1,Nf);
tic,
while toc < 2
    yw = fracDelay(xw,[0 0.2 0.4 0.6 0.8]);
    H = transferFuncEstimator(xw,yw);
    arrPlotMag(20*log10(abs(H)))
    arrPlotPhaseDelay(-angle(H)./w)
end
release(transferFuncEstimator)
release(arrPlotMag)
release(arrPlotPhaseDelay)
release(fracDelay)
```





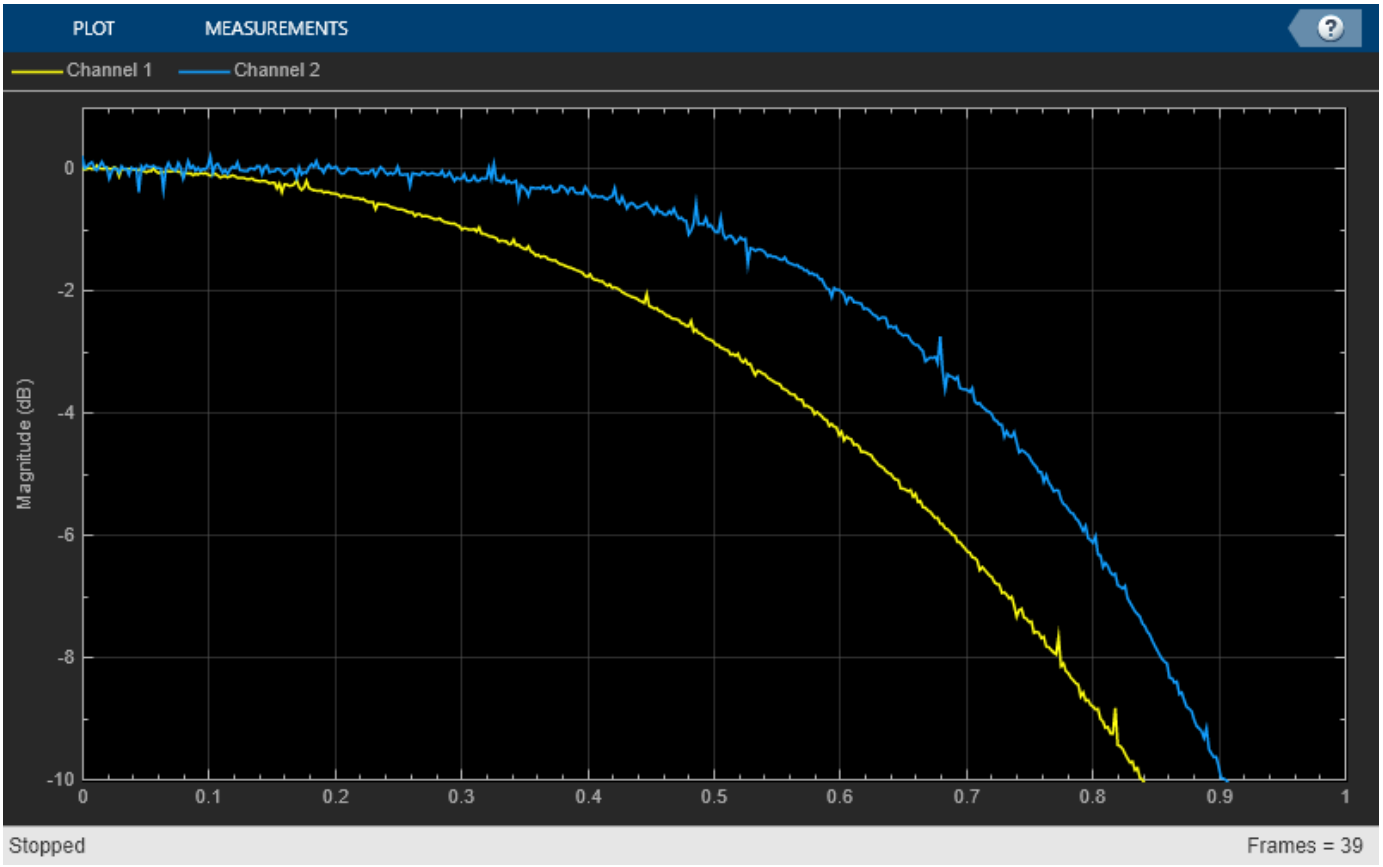
For any value of the delay, the ideal filter should have both a flat magnitude response and a flat phase delay response. The approximation is correct only for the lowest frequencies. This means that in practice the signals need to be over-sampled for the linear fractional delay to work correctly. Here you apply two different fractional delays to a sine wave and use the time scope to overlay the original sine wave and the two delayed versions. A delay of 0.2 samples with a sample rate of 1000 Hz, corresponds to a delay of 0.2 ms.

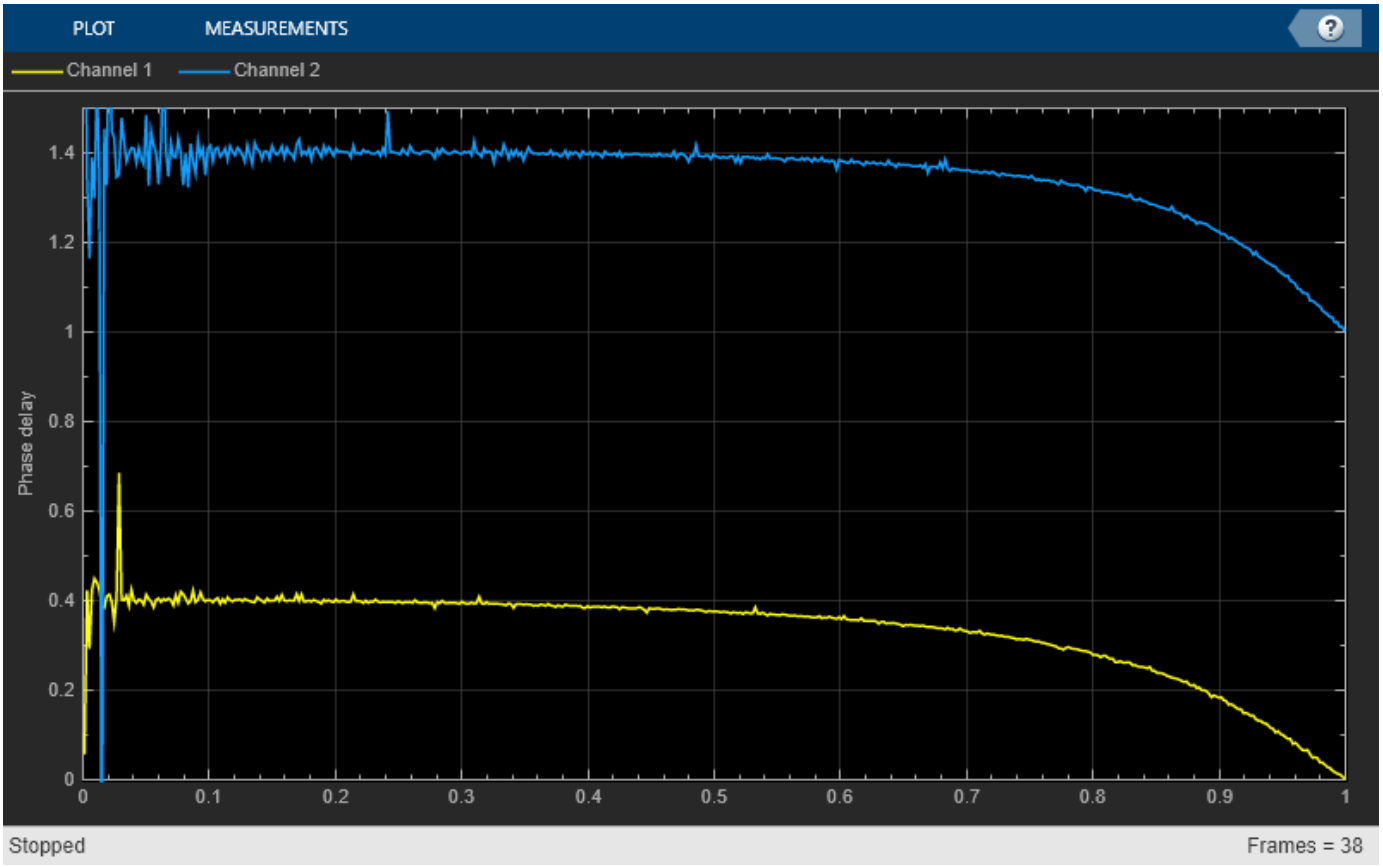
```
scope = timescope('SampleRate',1000, ...
                  'YLimits',[-1 1], ...
                  'TimeSpanSource','property',...
                  'TimeSpan',.02);
sine = dsp.SineWave('Frequency',50,'SamplesPerFrame',Nx);
tic,
while toc < 2
    x  = sine();
    y = fracDelay(x,[.2 .8]); % Delay by 0.2 ms and 0.8 ms
    scope([x,y(:,1),y(:,2)])
end
release(scope);
release(fracDelay)
```



Higher order Lagrange interpolators can be designed. Let's compare a cubic Lagrange interpolator with a linear one:

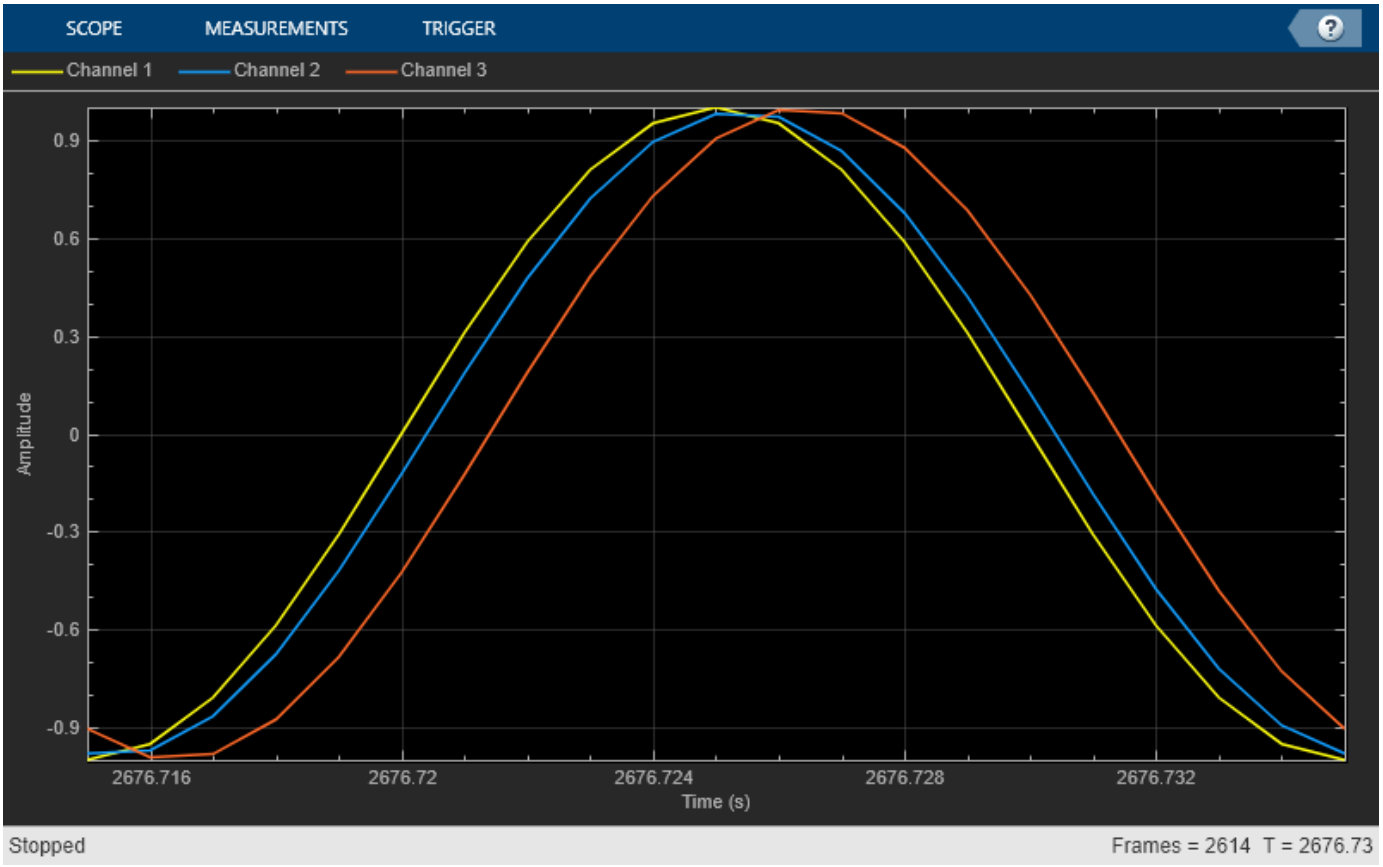
```
farrowFracDelay = dsp.VariableFractionalDelay( ...
    'InterpolationMethod','Farrow','MaximumDelay',1025);

Nf = 2;
yw = zeros(Nx,Nf);
xw = randn(Nx,Nf);
H = transferFuncEstimator(xw,yw);
w = getFrequencyVector(transferFuncEstimator,2*pi);
w = repmat(w,1,Nf);
tic,
while toc < 2
    % Run for 2 seconds
    yw(:,1) = fracDelay(xw(:,1),0.4);  % Delay by 0.4 ms
    yw(:,2) = farrowFracDelay(xw(:,2),1.4); % Delay by 1.4 ms
    H = transferFuncEstimator(xw,yw);
    arrPlotMag(20*log10(abs(H)))
    arrPlotPhaseDelay(-unwrap(angle(H))./w)
end
release(transferFuncEstimator)
release(arrPlotMag)
release(arrPlotPhaseDelay)
release(fracDelay)
```


PLOT
MEASUREMENTS
Channel 1
Channel 2
Magnitude (dB)
0
-2
-4
-6
-8
-10
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
Stopped
Frames = 39



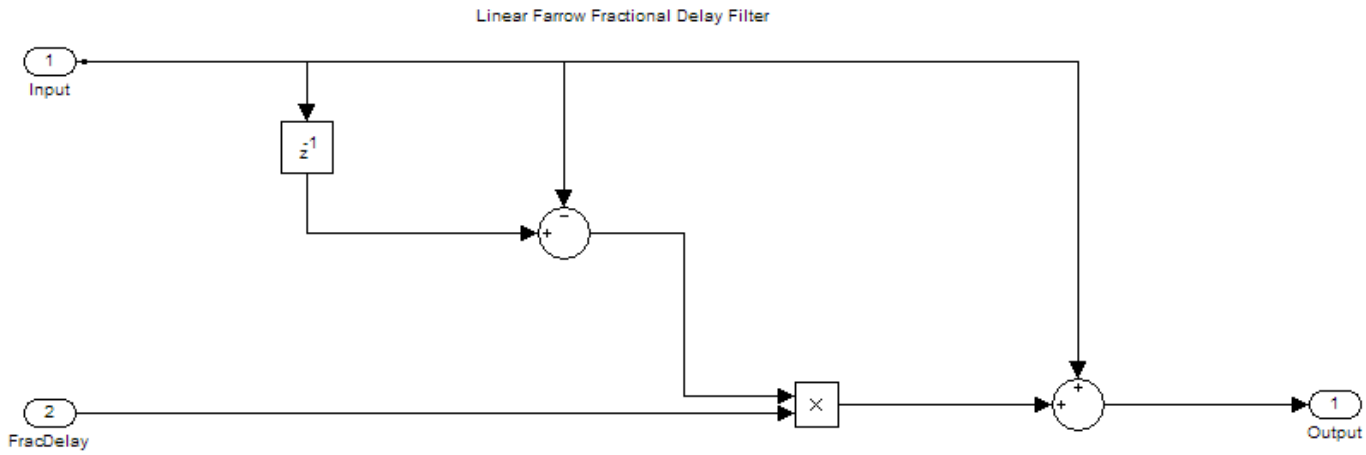
Increasing the order of the polynomials slightly increases the useful bandwidth when Lagrange approximation is used, the length of the differentiating filters i.e. the number of pieces of the impulse response (number of rows of the 'Coefficients' property) is equal to the length of the polynomials (number of columns of the 'Coefficients' property). Other design methods can be used to overcome this limitation. Also notice how the phase delay of the third order filter is shifted from 0.4 to 1.4 samples at DC. Since the cubic lagrange interpolator is a 3rd order filter, the minimum delay it can achieve is 1. For this reason, the delay requested is 1.4 ms instead of 0.4 ms for this case.

```
sine = dsp.SineWave('Frequency',50,'SamplesPerFrame',Nx);
tic,
while toc < 2
       x  = sine();
       y1 = fracDelay(x,0.4);
       y2 = farrowFracDelay(x,1.4);
       scope([x,y1,y2])
end
release(scope);
release(fracDelay)
```

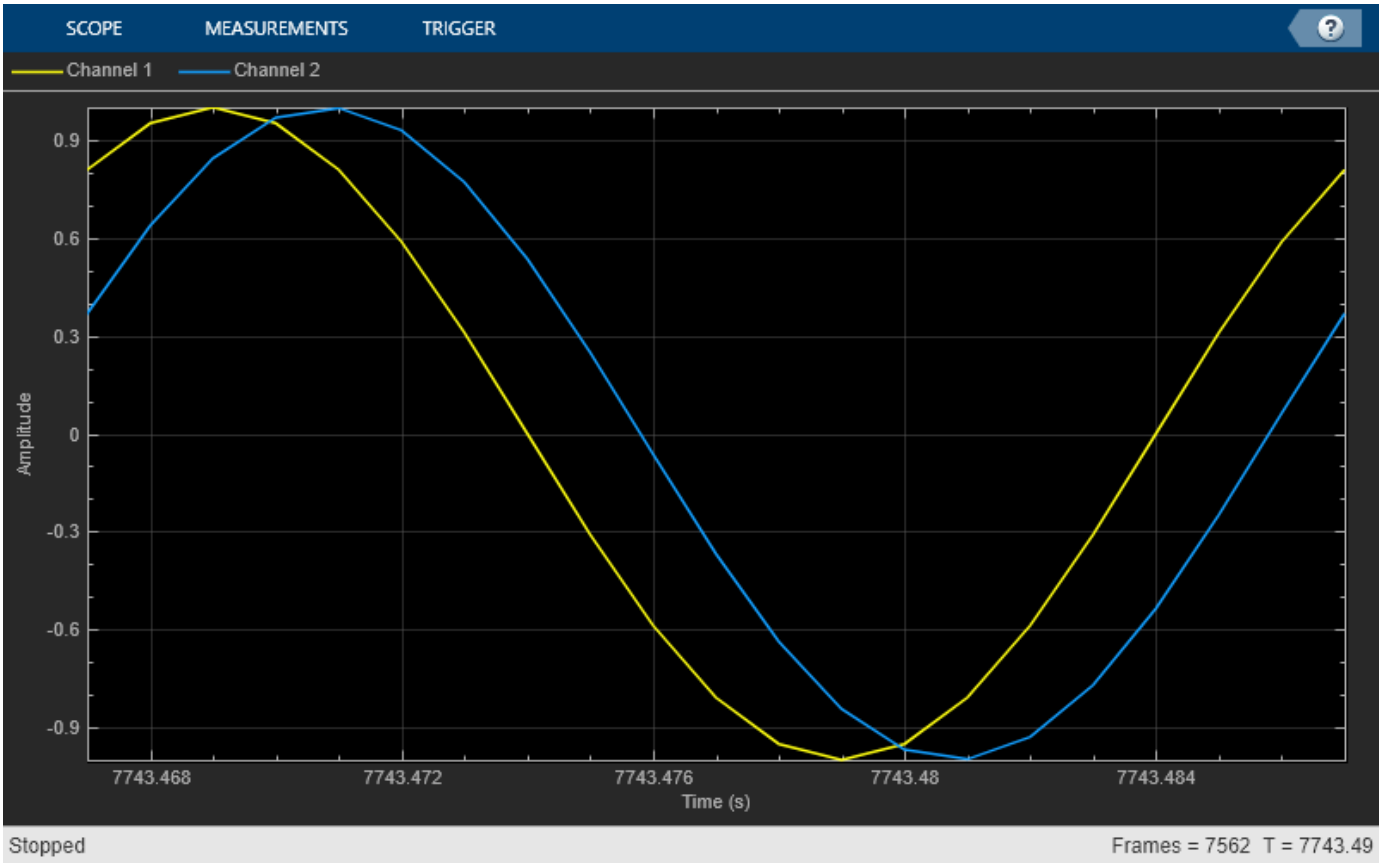


Time-Varying Fractional Delay

The advantage of the Farrow structure over a Direct-Form FIR resides in its tunability. In many practical applications, the delay is time-varying. For each new delay we would need a new set of coefficients in the Direct-Form implementation but with a Farrow implementation, the polynomial coefficients remain constant.



```
tic,
while toc < 5
    x  = sine();
    if toc < 1
        delay = 1;
    elseif toc < 2
        delay = 1.2;
    elseif toc < 3
        delay = 1.4;
    elseif toc < 4
        delay = 1.6;
    else
        delay = 1.8;
    end
    y = farrowFracDelay(x,delay);
    scope([x,y])
end
release(scope);
release(fracDelay)
```



See Also

Related Examples

- “Design of Fractional Delay FIR Filters” on page 4-196

Least Pth-norm Optimal FIR Filter Design

This example shows how to design least Pth-norm FIR filters with the FIRLPNORM function. This function uses a least-Pth unconstrained optimization algorithm to design FIR filters with arbitrary magnitude response.

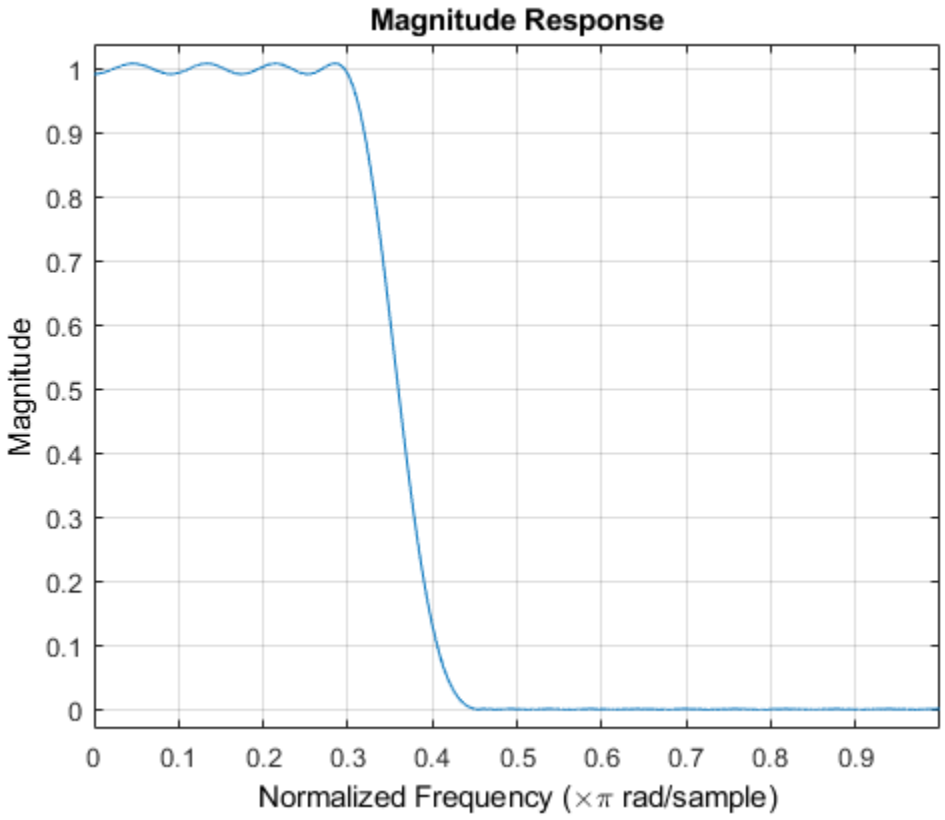
FIRLPNORM Syntax

The syntax for FIRLPNORM is similar to that of IIRLPNORM (see the least Pth-norm optimal IIR filter design example for details) except that the denominator order is not specified.

The function designs optimal FIR filters in the least-Pth sense. However the filter is not constrained to have linear-phase (although linear-phase is generally considered a good thing) i.e. the impulse response has no special symmetry properties.

However, the linear-phase constraint also results in filters with larger order than the more general nonlinear-phase designs (we should point out that in some hardware implementations, one can reduce the number of multipliers in half when implementing linear-phase filters because of the symmetry in the coefficients). For example, consider the following FIRLPNORM design

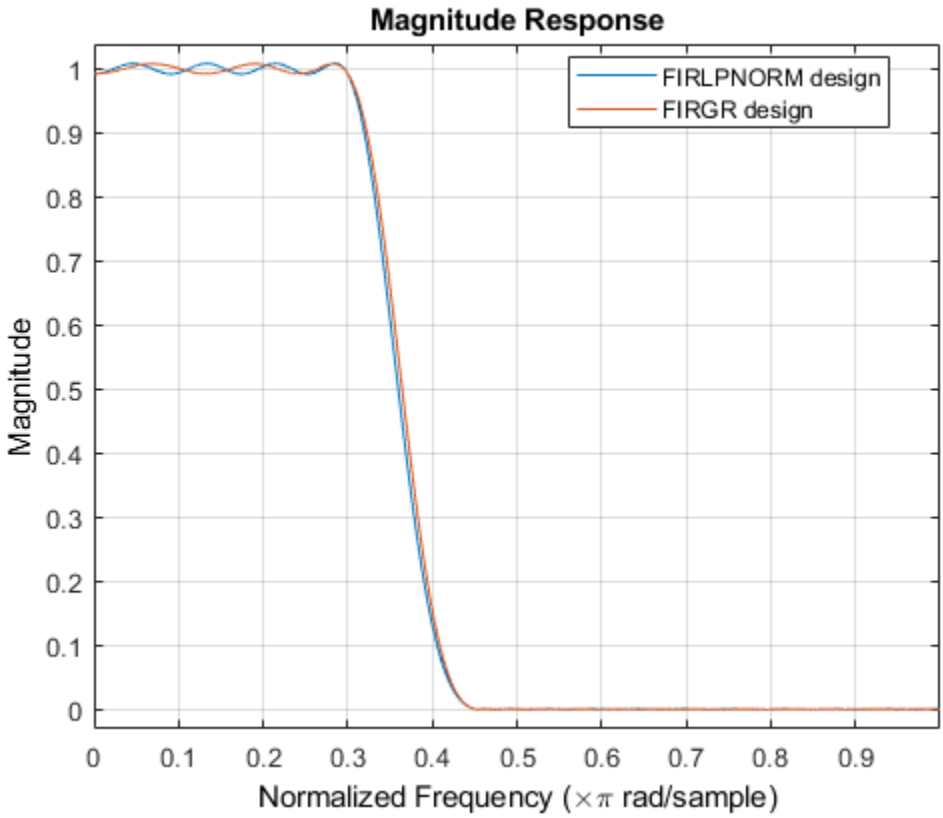
```
N = 30;
F = [0 0.3 0.45 1];
E = F;
A = [1 1 0 0];
W = [1 1 10 10];
b = firlpnorm(N,F,E,A,W);
h = fvtool(b);
h.MagnitudeDisplay = 'Magnitude';
h.Color = 'White';
```



If we zoom in, we can see that the filter has a passband peak ripple of about 0.008 and stopband peak ripple of about 0.000832. A FIRPM or FIRGR design with comparable specs will require a 37th order filter. This is especially significant considering that FIRGR will provide the lowest order linear-phase FIR filter that meets the specifications.

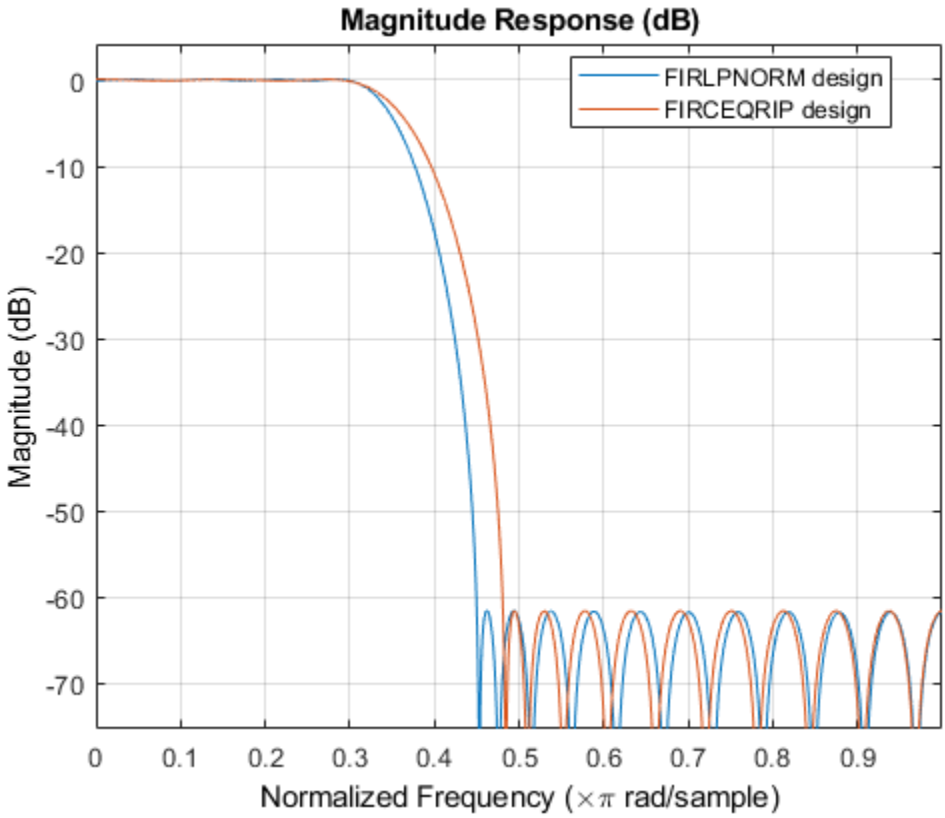
```
dev = [0.008 0.000832];
bgr = firgr('minorder',F,A,dev);
orderfirgr = length(b)-1;
fprintf('Order: %d\n',orderfirgr);
h = fvtool(b,1,bgr,1);
h.MagnitudeDisplay = 'Magnitude';
h.Color = 'White';
legend(h,'FIRLPNORM design','FIRGR design');
```

```
Order: 30
```



Another way to look at this is by using the FIRCEQRIP function which also designs linear-phase equiripple filters, but whose specifications are given in a different way to FIRGR (see the constrained equiripple FIR filter design example for details). If we want a linear-phase filter of 30th order that meets the passband and stopband ripple that the design from FIRLPNORM achieves we need to live with a larger transition width.

```
bceq = firceqrip(30,(F(2)+F(3))/2,dev);
h = fvtool(b,1,bceq,1,'Color','White');
legend(h,'FIRLPNORM design','FIRCEQRIP design');
```

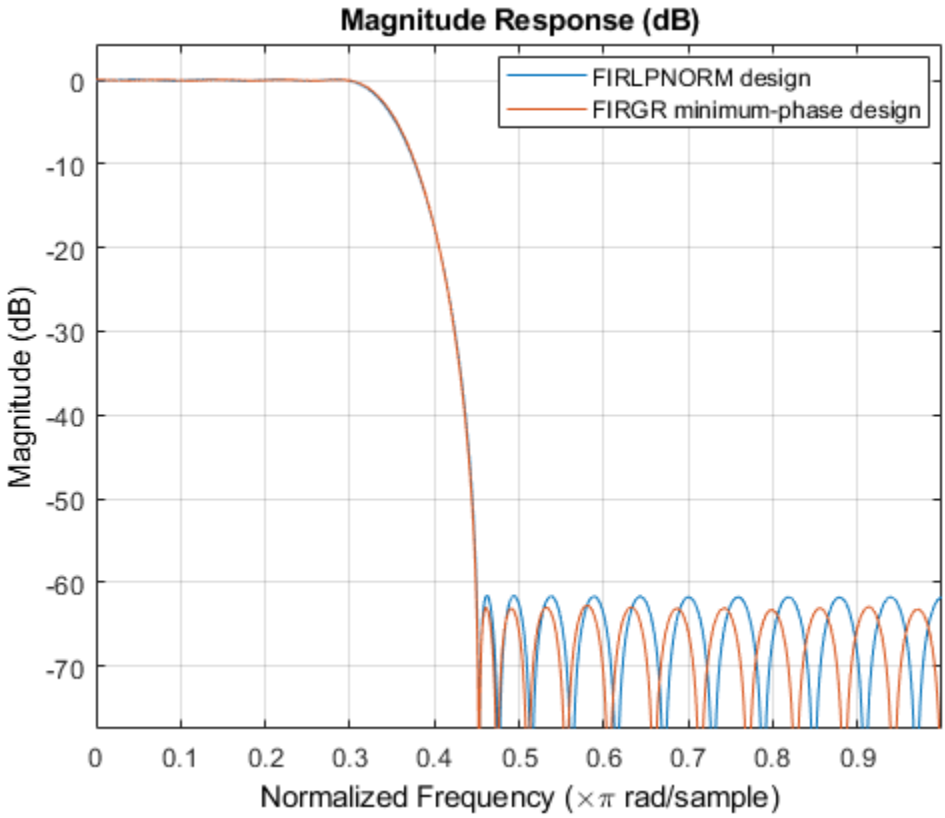


Minimum-Phase Designs

Of course it is also possible to design nonlinear-phase filters with FIRGR by specifying the 'minphase' option. Doing so, allows us to obtain an FIR filter of lower order than in the linear-phase case and still meet the required specs. However, even in this case, the result is a non-optimal nonlinear-phase filter because the filter order is larger than the minimum required for a nonlinear-phase equiripple filter to meet the specs as is evident from the following example.

```
bm = firgr('minorder',F,A,dev,'minphase');
orderfirgrmin = length(bm)-1;
fprintf('Order: %d\n',orderfirgrmin);
h = fvtool(b,1,bm,1,'Color','White');
legend(h,'FIRLPNORM design','FIRGR minimum-phase design');
```

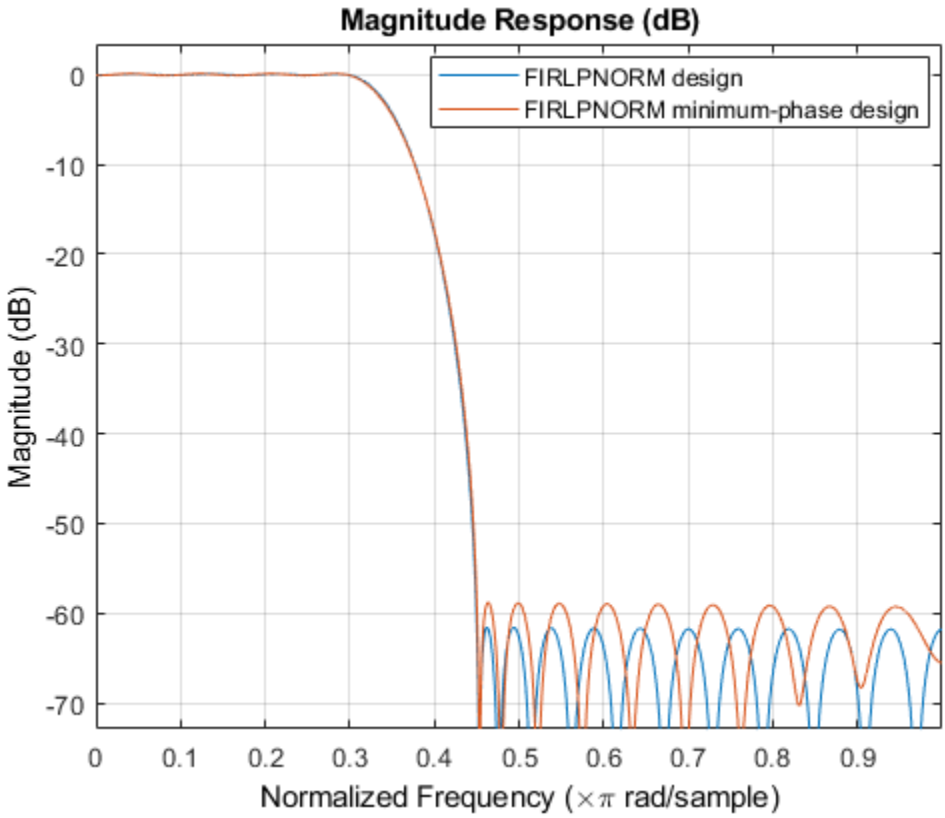
```
Order: 32
```



Minimum-Phase Designs with FIRLPNORM

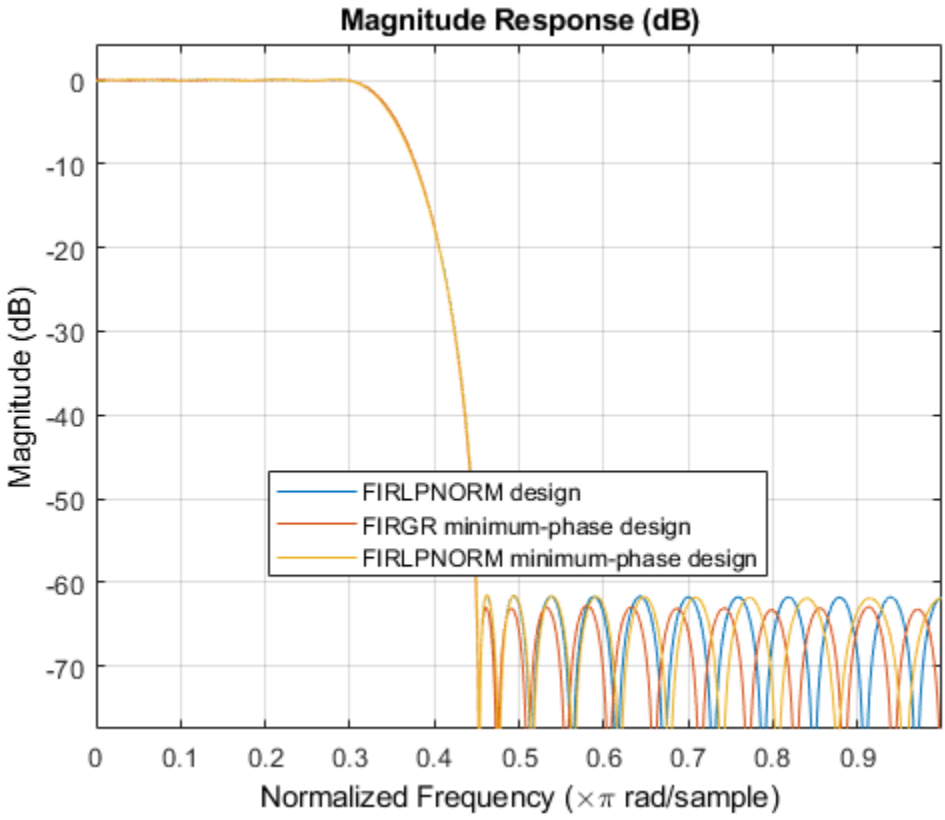
FIRLPNORM does allow for the option to constrain the zeros to lie on or inside the unit circle, resulting in a minimum-phase design. The constraint, however, results in larger passband ripple and less stopband attenuation than the unconstrained design.

```
bmlp = firlpnorm(30,F,E,A,W,'minphase');
h = fvtool(b,1,bmlp,1,'Color','White');
legend(h,'FIRLPNORM design','FIRLPNORM minimum-phase design');
```



If we increase the order to that of the minimum-phase filter designed with FIRGR we can see that we meet the specs met by both the 30th order FIRLPNORM (nonminimum-phase) design and the 32nd order FIRGR minimum-phase design.

```
bmlp = firlpnorm(orderfirgrmin,F,E,A,W,'minphase');
h = fvtool(b,1,bm,1,bmlp,1,'Color','White');
legend(h,'FIRLPNORM design',...
    'FIRGR minimum-phase design',...
    'FIRLPNORM minimum-phase design');
```

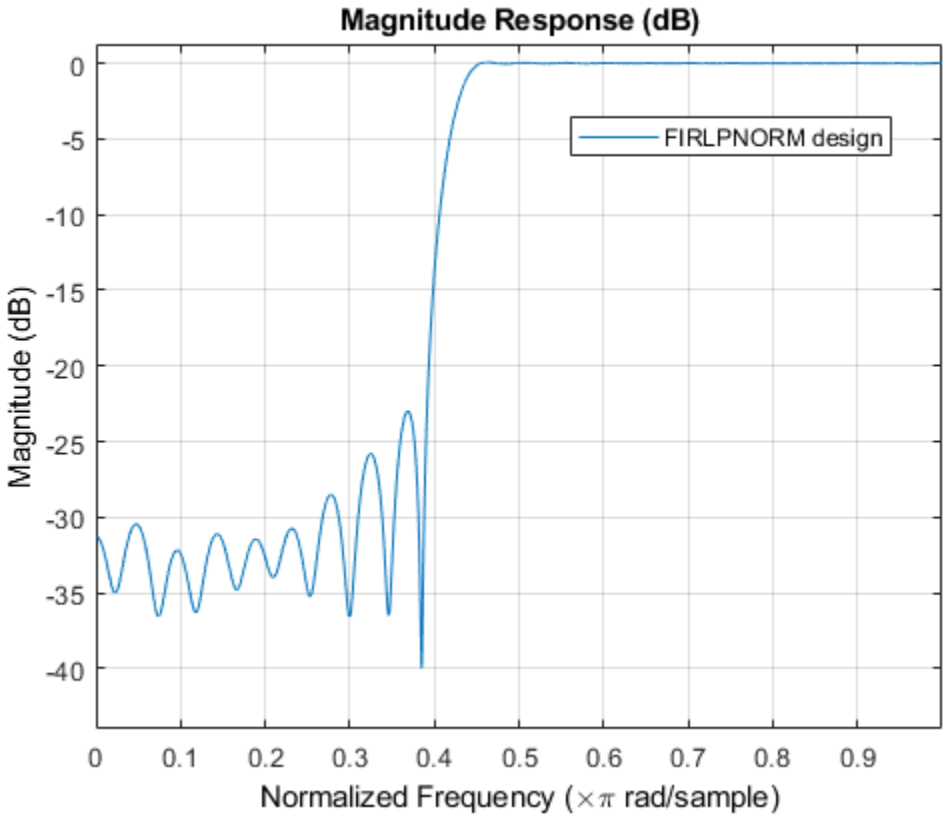


Changing the Pth-Norm

Like IIRLPNORM and IIRLPNORMC, FIRLPNORM allows for the specification of the Pth-norm used to optimize the filter. The Pth-norm is specified in the exact same way as in IIRLPNORM, i.e. a two element vector with Pinit and Pfinal as its elements. Pinit specifies the initial Pth-norm used by the algorithm (this aids in the convergence) and Pfinal specifies the final Pth-norm with which the filter is optimized.

For example, a least-squares design for the above specs can be obtained as follows:

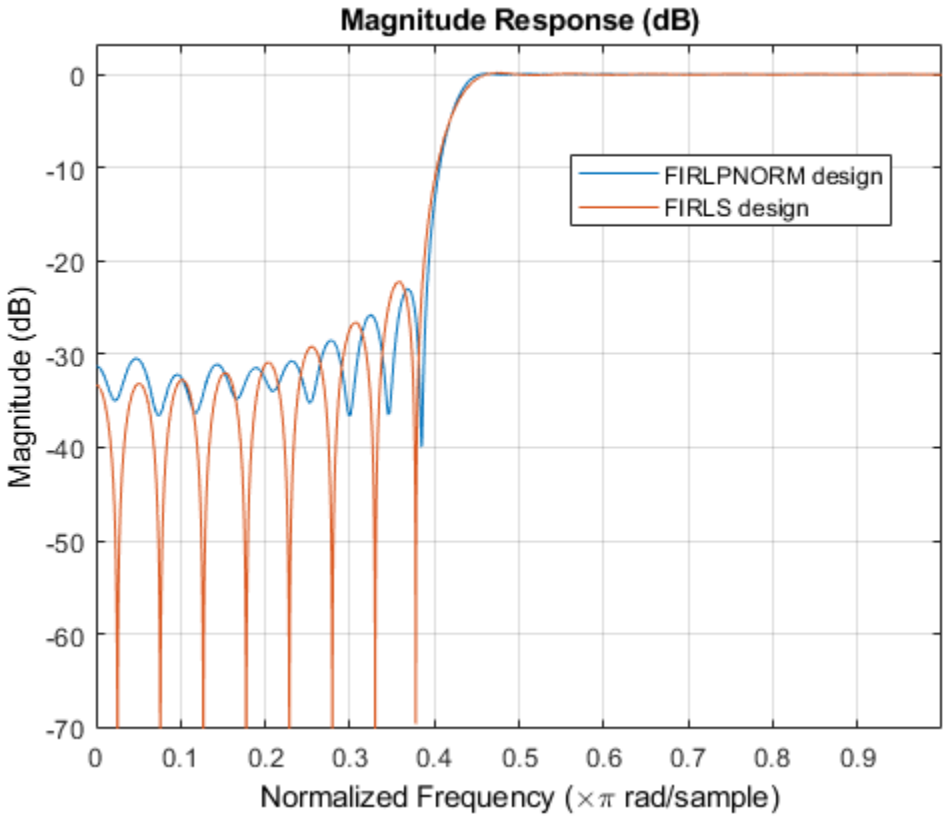
```
N = 40;
F = [0 0.4 0.45 1];
E = F;
A = [0 0 1 1];
W = [1 1 10 10];
P = [2 2];
bl2 = firlpnorm(N,F,E,A,W,P);
h = fvtool(bl2,1,'Color','White');
legend(h,'FIRLPNORM design')
```



Comparing to FIRLS

In comparison, we design a linear-phase least-squares filter using FIRLS. Once again, for the same filter order, the linear-phase constraint results in less stopband attenuation and a larger passband ripple.

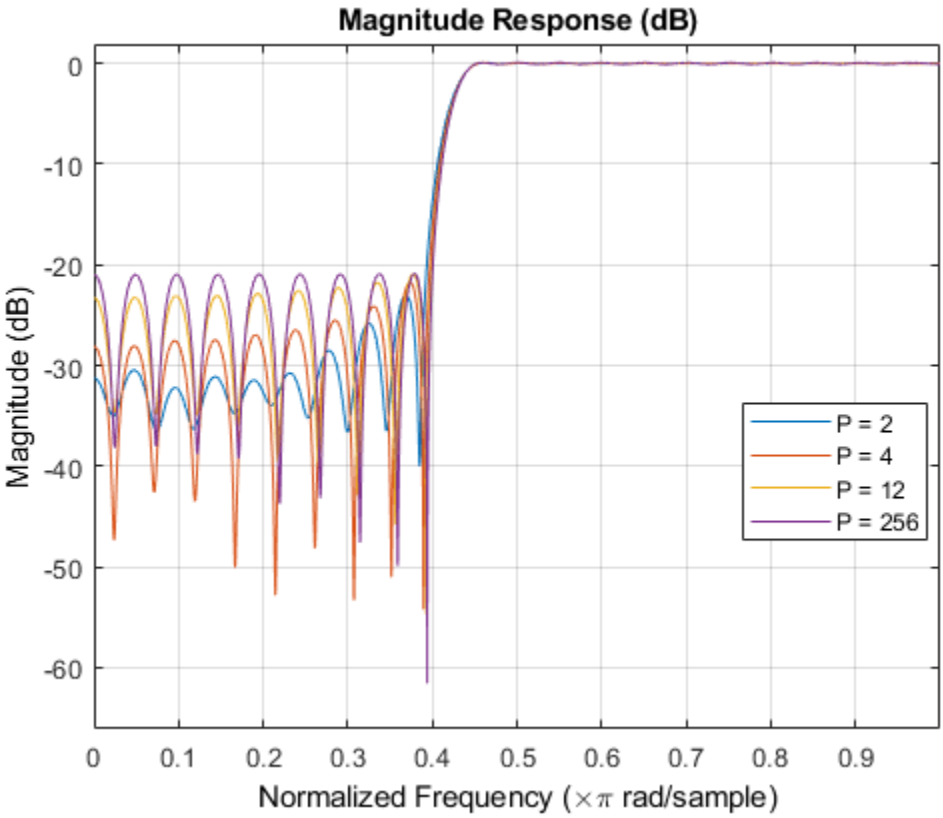
```
W = [1 20];
bls = firls(N,F,A,W);
h = fvtool(bl2,1,bls,1,'Color','White');
legend(h,'FIRLPNORM design','FIRLS design');
```



Other Norms

Equiripple designs are useful when one requires the smallest possible order to meet a set of design specifications. To meet the same specs with a least-squares design requires a higher order filter. However, the higher order does provide extra attenuation (less ripple) for a large portion of the stopband (passband). In fact least-squares design minimize the energy of the stopband. Compromises between equiripple design and least-squares design can be reached by using intermediate norms. For example we show the design of a filter with the same specs, but optimized for the following norms: 2, 4, 12, 256 (approx. infinity norm).

```
W = [1 1 10 10];
P4 = [2 4];
bl4 = firlpnorm(N,F,E,A,W,P4);
P12 = [2 12];
bl12 = firlpnorm(N,F,E,A,W,P12);
Pinf = [2 256];
blinf = firlpnorm(N,F,E,A,W,Pinf);
h = fvtool(bl2,1,bl4,1,bl12,1,blinf,1,'Color','White');
legend(h,'P = 2','P = 4','P = 12','P = 256');
```



In order to meet the minimum stopband attenuation of the equiripple (256-norm) case it is necessary to increase the order of the other designs.

Least Pth-Norm Optimal IIR Filter Design

This example shows how to design optimal IIR filters with arbitrary magnitude response using the least-Pth unconstrained optimization algorithm.

IIRLPNORM Fundamentals

The IIRLPNORM algorithm differs from the traditional IIR design algorithms in several aspects:

. The designs are done directly in the Z-domain. No need for bilinear transformation.

. The numerator and denominator order can be different.

. One can design IIR filters with arbitrary magnitude response in addition to the basic lowpass, highpass, bandpass, and bandstop.

Lowpass Design

For simple designs however (lowpass, highpass, etc.), we must specify passband and stopband frequencies. The transition band is considered as a don't-care band by the algorithm.

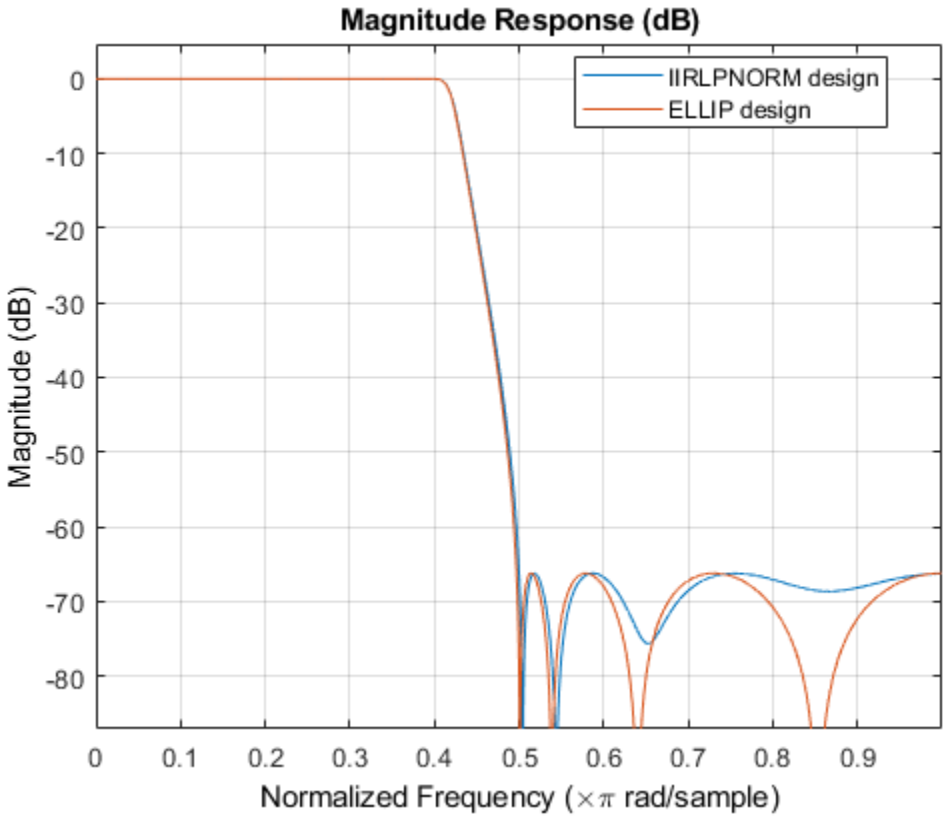
```
d = fdesign.lowpass('N,Fp,Fst',8,.4,.5) %#ok
```

```
d =

  lowpass with properties:

              Response: 'Lowpass'
         Specification: 'N,Fp,Fst'
           Description: {3x1 cell}
    NormalizedFrequency: 1
           FilterOrder: 8
                 Fpass: 0.4000
                 Fstop: 0.5000
```

```
hiirlpnorm = design(d,'iirlpnorm','SystemObject',true);
```

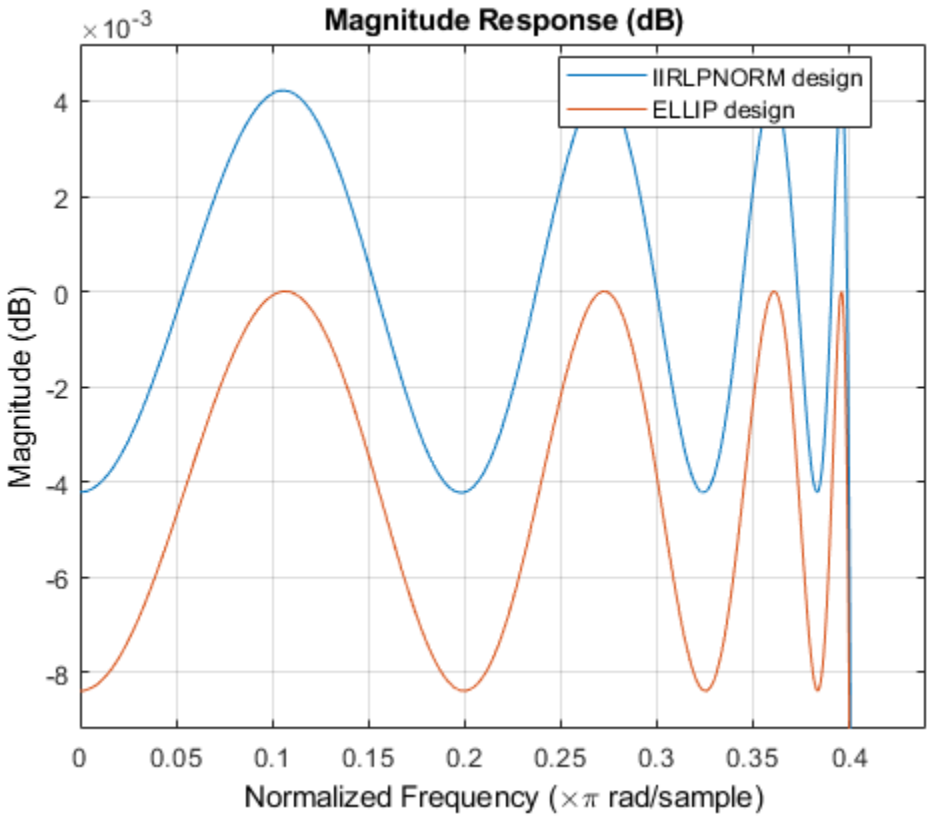
For comparison purposes, consider this elliptic filter design

```
d = fdesign.lowpass('N,Fp,Ap,Ast',8,.4,0.0084,66.25);
hellip = design(d, 'ellip','SystemObject',true);
hfvt = fvtool(hiirlpnorm,hellip,'Color','White');
legend(hfvt,'IIRLPNORM design','ELLIP design');
```



The response of the two filters is very similar. Zooming into the passband accentuates the point. However, the magnitude of the filter designed with IIRLPNORM is not constrained to be less than 0 dB.

```
axis([0 .44 -.0092 .0052])
```



Different Numerator, Denominator Orders

While we can get very similar designs as elliptic filters, IIRLPNORM provides greater flexibility. For instance, say we change the denominator order

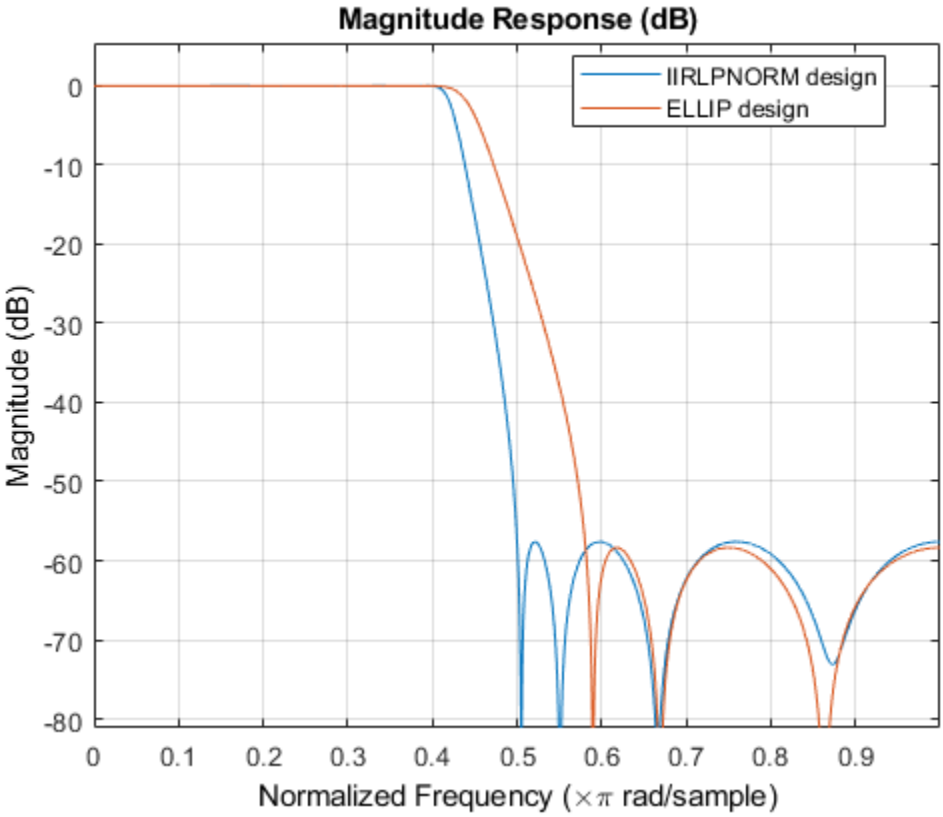
```
d = fdesign.lowpass('Nb,Na,Fp,Fst',8,6,.4,.5) %#ok
```

```
d =

  lowpass with properties:

              Response: 'Lowpass'
         Specification: 'Nb,Na,Fp,Fst'
           Description: {4x1 cell}
    NormalizedFrequency: 1
               NumOrder: 8
               DenOrder: 6
                  Fpass: 0.4000
                  Fstop: 0.5000
```

```
hiirlpnorm = design(d,'iirlpnorm','SystemObject',true);
```

With elliptic filters (and other classical IIR designs) we must change both the numerator and the denominator order.

```
d = fdesign.lowpass('N,Fp,Ap,Ast',6,.4,0.0084,58.36);
hellip = design(d, 'ellip','SystemObject',true);
hfvt = fvtool(hiirlpnorm,hellip,'Color','White');
legend(hfvt,'IIRLPNORM design','ELLIP design');
```



Clearly, the elliptic design (in green) now results in a much wider transition width.

Weighting the Designs

Similar to equiripple or least-square designs, we can weight the optimization criteria to alter the design as we see fit. However, unlike equiripple, we have the extra flexibility of providing different weights for each frequency point instead of for each frequency band.

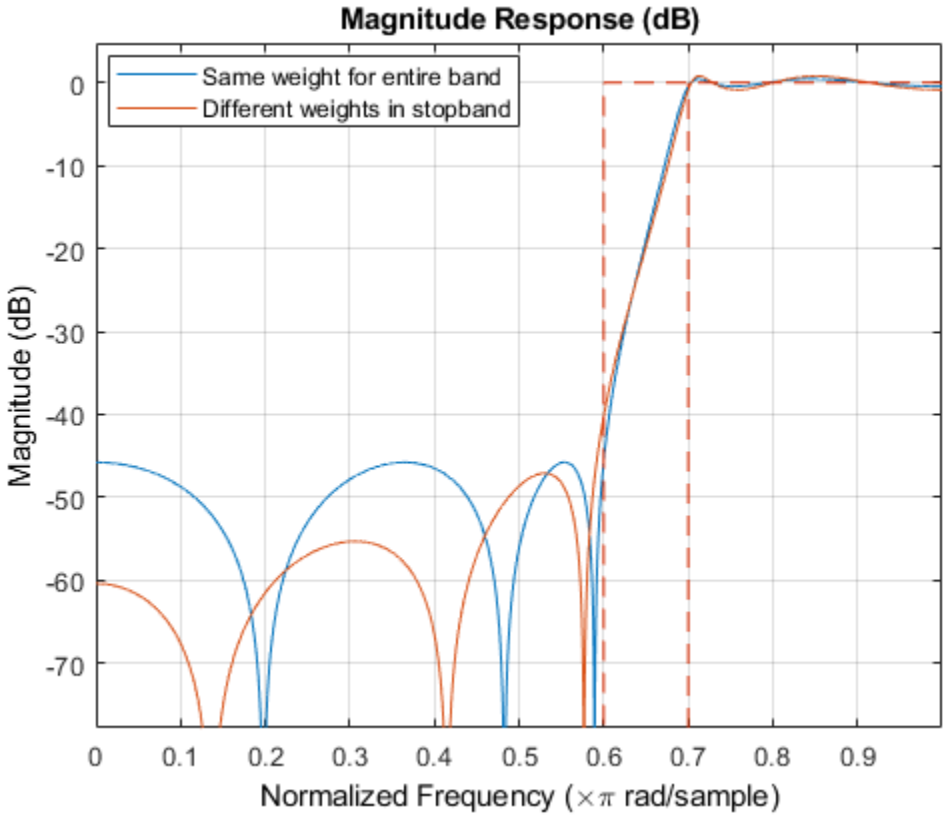
Consider the following two highpass filters:

```
d = fdesign.highpass('Nb,Na,Fst,Fp',6,4,.6,.7) %#ok
```

```
d =

  highpass with properties:

         Specification: 'Nb,Na,Fst,Fp'
              Response: 'Highpass'
           Description: {4x1 cell}
    NormalizedFrequency: 1
                NumOrder: 6
                DenOrder: 4
```

```
                Fstop: 0.6000
                Fpass: 0.7000
```

```
h1 = design(d,'iirlpnorm','Wpass',1,'Wstop',10,'SystemObject',true);
h2 = design(d,'iirlpnorm','Wpass',1,'Wstop',[100 10],'SystemObject',true);
hfvt = fvtool(h1,h2,'Color','White');
legend(hfvt,'Same weight for entire band',...
    'Different weights in stopband');
```



The first design uses the same weight per band (10 in the stopband, 1 in the passband). The second design uses a different weight per frequency point. This provides a simple way of attaining a sloped stopband which may be desirable in some applications. The extra attenuation over portions of the stopband comes at the expense of a larger passband ripple and transition width.

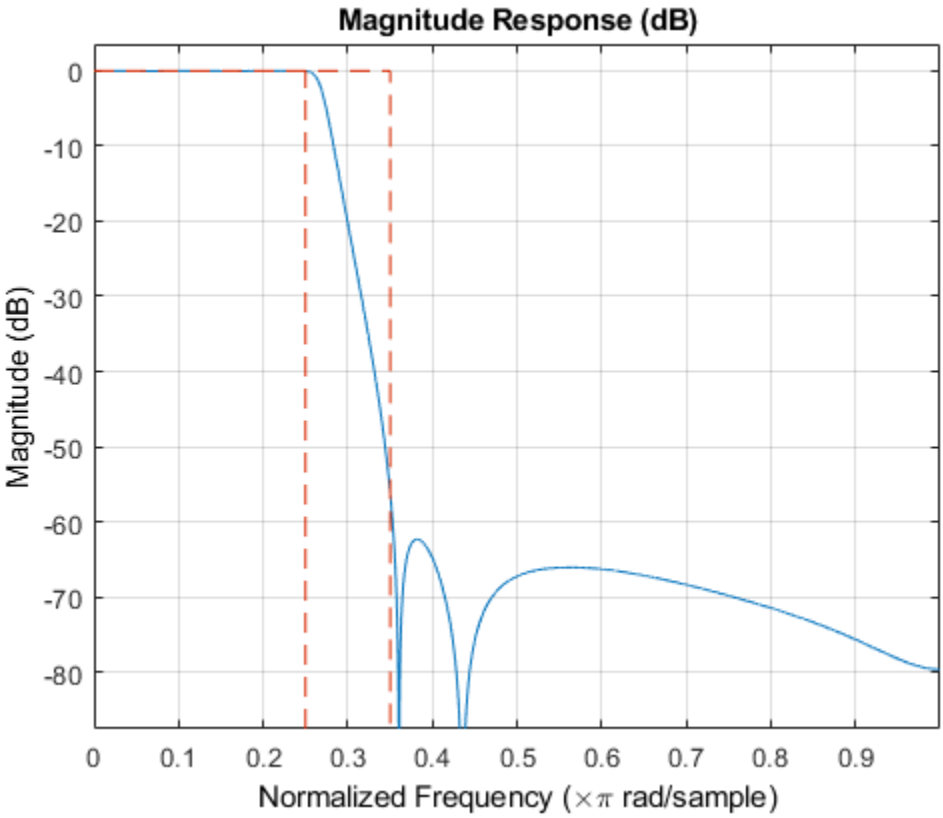
The Pth-Norm

Roughly speaking, the optimal design is achieved by minimizing the error between the actual designed filter and an ideal filter in the Pth-norm sense. Different values of the norm result in different designs. When specifying the P-th norm, we actually specify two values, 'InitNorm' and 'Norm' where 'InitNorm' is the initial value of the norm used by the algorithm and 'Norm' is the final (the actual) value for which the design is optimized. Starting the optimization with a smaller initial value aids in the convergence of the algorithm.

By default, the algorithm starts optimizing in the 2-norm sense but finally optimizes the design in the 128-norm sense. The 128-norm in practice yields a good approximation to the infinity-norm. So that

the designs tend to be equiripple. For a least-squares design, we should set the norm to 2. For instance, consider the following lowpass filter

```
d = fdesign.lowpass('Nb,Na,Fp,Fst',10,7,.25,.35);
design(d,'iirlpnorm','Norm',2,'SystemObject',true);
fig = gcf;
fig.Color = [1 1 1];
```



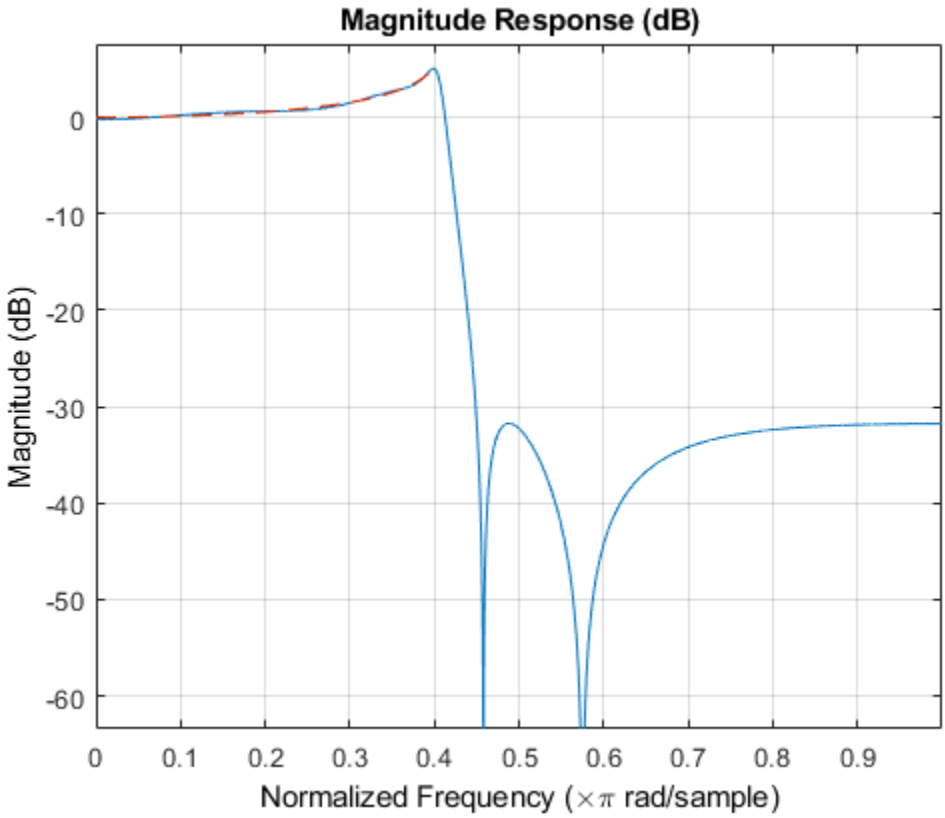
Arbitrary Shaped Magnitude

Another of the important features of IIRLPNORM is its ability to design filters other than the basic lowpass, highpass, bandpass and bandstop. See the “Arbitrary Magnitude Filter Design” on page 4-128 example for more information. We now show a few examples:

Rayleigh Fading Channel

Here's a filter for noise shaping when simulating a Rayleigh fading wireless communications channel

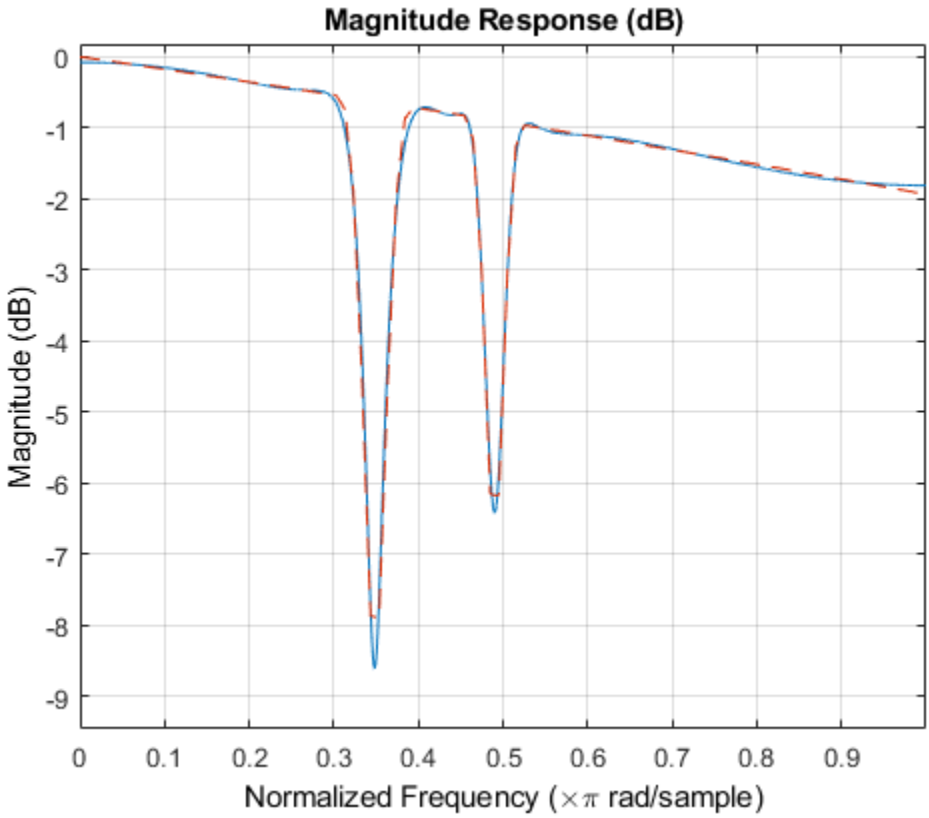
```
F1 = 0:0.01:0.4;
A1 = 1.0 ./ (1 - (F1./0.42).^2).^0.25;
F2 = [0.45 1];
A2 = [0 0];
d = fdesign.arbmag('Nb,Na,B,F,A',4,6,2,F1,A1,F2,A2);
design(d,'iirlpnorm','SystemObject',true);
fig = gcf;
fig.Color = [1 1 1];
```



Optical Absorption of Atomic Rubidium87 Vapor

The following design models the absorption of light in a certain gas. The resulting filter turns out to have approximately linear-phase:

```
Nb = 12;
Na = 10;
F = linspace(0,1,100);
As = ones(1,100)-F*0.2;
Absorb = [ones(1,30),(1-0.6*bohmanwin(10))',...
    ones(1,5), (1-0.5*bohmanwin(8))',ones(1,47)];
A = As.*Absorb;
d = fdesign.arbmag('Nb,Na,F,A',Nb,Na,F,A);
W = [ones(1,30) ones(1,10)*.2 ones(1,60)];
design(d, 'iirlpnorm', 'Weights', W, 'Norm', 2, 'DensityFactor', 30,...
    'SystemObject',true);
fig = gcf;
fig.Color = 'white';
```


Magnitude Response (dB)
Magnitude (dB)
0
-1
-2
-3
-4
-5
-6
-7
-8
-9
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Normalized Frequency (×π rad/sample)

Multistage Rate Conversion

Multistage rate conversion is an approach that splits rate conversion into several stages. For example, instead of decimation by a factor of 18, decimate by factor of 3, followed by another decimation by 3, and and then by a factor of 2. Using multiple stages reduces the computational complexity of filtered rate conversion. Furthermore, if one already has the converter units for the different prime factors, they can be used as building blocks for higher rates. This example demonstrates multistage rate conversion designs.

Single-Stage v.s. Multistage Conversion: Cost Analysis

Consider decimation system of rate $M = 8$. One can implement such a system in two ways:

- A single decimator of rate $M = 8$.
- A cascade of three half-rate decimators ($M = 2$)

A multistage cascade of filtered decimators (or interpolators) has a reduced single-stage form. The filter in the reduced form is called the single-stage equivalent filter, which encapsualtes the filters of all the stages. Thus, any multistage cascade FIR decimator can be represented as a single-stage FIR decimator. For more details, see [1]. However, while the two alternatives effectively perform the same decimation, they differ in their numerical complexites.

Evaluate the cost of implementing a multistage decimator using the `cost` function, and compare to the cost of implementing a single-stage decimator.

```
firDecim2_1 = dsp.FIRDecimator(2);
firDecim2_2 = dsp.FIRDecimator(2);
firDecim2_3 = dsp.FIRDecimator(2);
firDecim2cascade = dsp.FilterCascade(firDecim2_1,firDecim2_2,firDecim2_3);

cost2cascade = cost(firDecim2cascade)

firDecim8 = dsp.FIRDecimator(8);
cost8 = cost(firDecim8)
```

```
cost2cascade =

  struct with fields:

                 NumCoefficients: 75
                       NumStates: 138
    MultiplicationsPerInputSample: 21.8750
          AdditionsPerInputSample: 21


cost8 =

  struct with fields:

                 NumCoefficients: 169
                       NumStates: 184
    MultiplicationsPerInputSample: 21.1250
          AdditionsPerInputSample: 21
```

Cascading three decimators of rate M=2 consumes less memory (states and coefficients) compared to a single-stage decimator of M=8, making the multistage converter more memory efficient. The arithmetic load (operations per sample) of the single-stage and multistage implementation are equivalent. Note that the number of samples drops by half after each decimation stage. In conclusion, it is often better to split the decimation into multiple stages (given that the rate change factor is not a prime number, of course).

There is usually more than one way to factor a (non-prime) conversion rate, and even more degrees of freedom multistage design. The DSP System Toolbox (TM) offers several tools to simplify the design process. We will examine two of them in what follows.

Using designMultistageDecimator and designMultistageInterpolator Functions

The `designMultistageInterpolator` and `designMultistageDecimator` functions automatically determine an optimal configuation, that includes determining the number of stages along with their arrangements, lowpass parameters, etc. The result is a filter cascade system object, which encapsualtes all the stages. To illustrate, let us design a decimator of rate M=12.

```
M = 12;
fcDecMulti = designMultistageDecimator(M);
disp(cascadeInfoSummary(fcDecMulti))
```

```
Multistage FIR Decimator, M=12 (48.0kHz to 4.0kHz)
Equivalent lowpass cutoff: 4.0kHz, transition Width: 800.0Hz

Number of stages: 3
Stage1: FIR Decimator, M = 2 (48.0kHz to 24.0kHz), FIR Length = 11
Stage2: FIR Decimator, M = 2 (24.0kHz to 12.0kHz), FIR Length = 15
Stage3: FIR Decimator, M = 3 (12.0kHz to 4.0kHz),  FIR Length = 79
```

This particular design has 3 stages ($12 = 2 \times 2 \times 3$), where the lowpass of the last stage is the longest.

Repeat the design with a single-stage.

```
fcDecSingle = designMultistageDecimator(M,'NumStages',1);
disp(cascadeInfoSummary(fcDecSingle))
```

```
Multistage FIR Decimator, M=12 (48.0kHz to 4.0kHz)
Equivalent lowpass cutoff: 4.0kHz, transition Width: 800.0Hz

Number of stages: 1
Stage1: FIR Decimator, M = 12 (48.0kHz to 4.0kHz), FIR Length = 307
```

Compare the cost of the two implementations. Obivously, the multistage approach is more efficient.

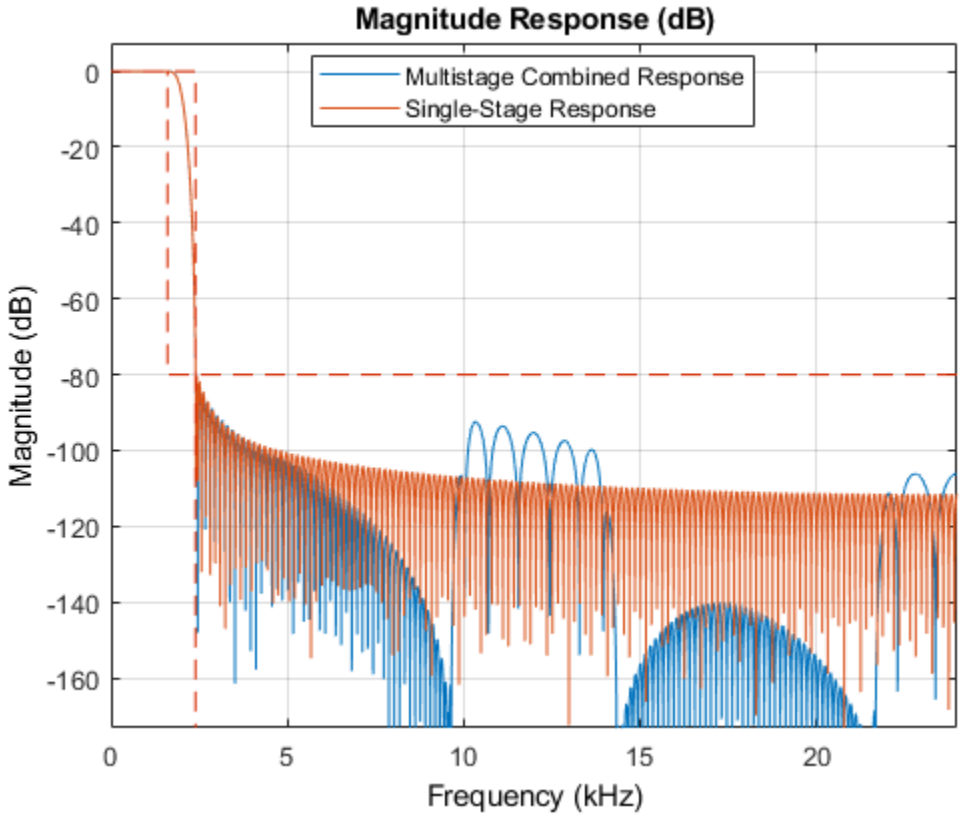
```
costMulti  = cost(fcDecMulti)
costSingle = cost(fcDecSingle)
```

```
costMulti =

  struct with fields:

                 NumCoefficients: 69
                       NumStates: 102
    MultiplicationsPerInputSample: 10.1667
```

```
          AdditionsPerInputSample: 9.3333


costSingle =

  struct with fields:

                 NumCoefficients: 283
                      NumStates: 300
    MultiplicationsPerInputSample: 23.5833
          AdditionsPerInputSample: 23.5000
```

Now, let us compare the combined frequency response of the decimation filters. While the filters of the two implementations differ in the stopband, the passband and transition band are nearly identical.

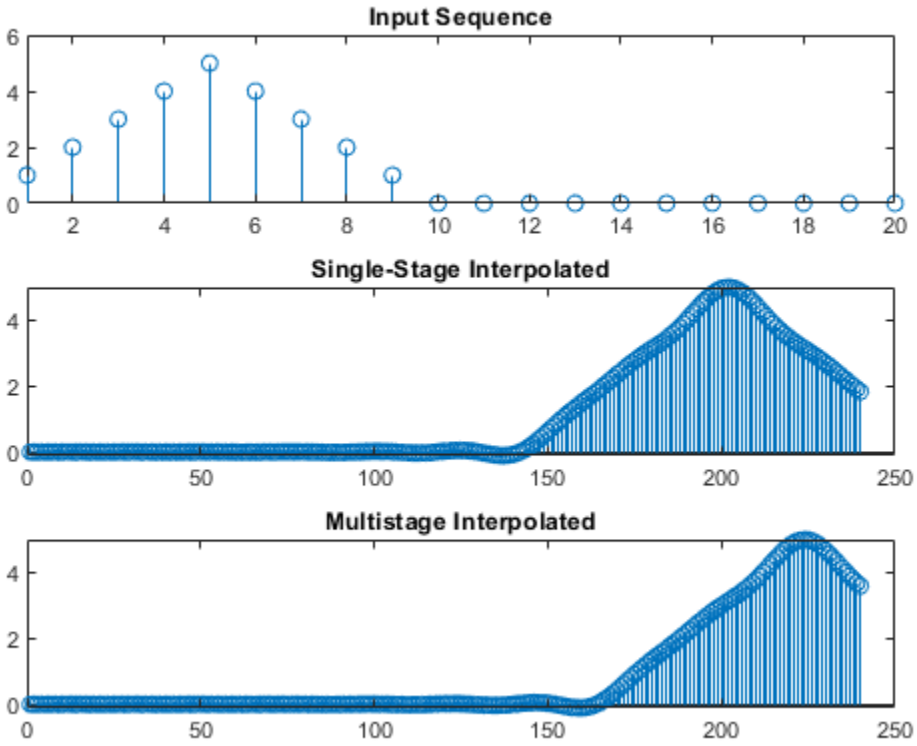
```
hfv = fvtool(fcDecMulti, fcDecSingle);
legend(hfv,'Multistage Combined Response', 'Single-Stage Response');
```



The same methodology applies for `designMultistageInterpolator`. Create two interpolators (single-stage and multistage) and compare their outputs. Note that the outputs are nearly identical, except a slightly longer latency of the multistage interpolator.

```
n = (1:20)';
x = (abs(n-5)<=5).*(5-abs(n-5));
```

```
L = 12;
fcIntrMulti  = designMultistageInterpolator(L);
fcIntrSingle = designMultistageInterpolator(L,'NumStages',1);

xInterpSingle = fcIntrSingle(x);
xInterpMulti  = fcIntrMulti(x);

release(fcIntrMulti);
release(fcIntrSingle);

subplot(3,1,1); stem(x); xlim([1,20]); title('Input Sequence');
subplot(3,1,2); stem(xInterpSingle); title('Single-Stage Interpolated')
subplot(3,1,3); stem(xInterpMulti); title('Multistage Interpolated')
```



Additional Design Parameters for the `designMultistageDecimator` and `designMultistageInterpolator` Functions

You can specify filter design parameters such as transition width and stopband attenuation to the `designMultistageDecimator` and `designMultistageInterpolator` functions. Such additional parameters allow for better control of the filter characteristics. The input sample rate *Fs* is assumed to be 1 by default, but this value can be customized as well.

Design a filter that reduces the input rate from 48 MHz to 1 MHz, a decimation factor of 48. The following are typical specifications for a lowpass filter that reduces the bandwidth accordingly.

```
Fs    = 48e6;
TW    = 100e3;
```

```
Astop = 80;     % Minimum stopband attenuation
M     = 48;     % Decimation factor
```

Here is a simple multistage design for these specs.

```
multiDecim = designMultistageDecimator(M,Fs,TW,Astop);
disp(cascadeInfoSummary(multiDecim))
```

```
Multistage FIR Decimator, M=48 (48.0MHz to 1.0MHz)
Equivalent lowpass cutoff: 1.0MHz, transition Width: 100.0kHz

Number of stages: 5
Stage1: FIR Decimator, M = 2 (48.0MHz to 24.0MHz), FIR Length = 7
Stage2: FIR Decimator, M = 2 (24.0MHz to 12.0MHz), FIR Length = 7
Stage3: FIR Decimator, M = 2 (12.0MHz to 6.0MHz),  FIR Length = 11
Stage4: FIR Decimator, M = 3 (6.0MHz to 2.0MHz),   FIR Length = 33
Stage5: FIR Decimator, M = 2 (2.0MHz to 1.0MHz),   FIR Length = 95
```

This is a 5-stage decimator cascade with the factors $2 \to 2 \to 2 \to 3 \to 2$ whose product is $2^4 \times 3 \ = \ 48$ as expected.

Design a similar filter with the default transition width and attenuation. The overall conversion rate is similar, but the transition width (and perhaps the ordering of the stages) can be different.

```
multiDecim_default = designMultistageDecimator(M,Fs);
disp(cascadeInfoSummary(multiDecim_default))
```

```
Multistage FIR Decimator, M=48 (48.0MHz to 1.0MHz)
Equivalent lowpass cutoff: 1.0MHz, transition Width: 200.0kHz

Number of stages: 5
Stage1: FIR Decimator, M = 2 (48.0MHz to 24.0MHz), FIR Length = 7
Stage2: FIR Decimator, M = 2 (24.0MHz to 12.0MHz), FIR Length = 7
Stage3: FIR Decimator, M = 2 (12.0MHz to 6.0MHz),  FIR Length = 11
Stage4: FIR Decimator, M = 2 (6.0MHz to 3.0MHz),   FIR Length = 15
Stage5: FIR Decimator, M = 3 (3.0MHz to 1.0MHz),   FIR Length = 79
```

Design a single-stage decimator using the same parameters.

```
singleDecim = designMultistageDecimator(M,Fs,TW,Astop,'NumStages',1);
disp(cascadeInfoSummary(singleDecim))
```

```
Multistage FIR Decimator, M=48 (48.0MHz to 1.0MHz)
Equivalent lowpass cutoff: 1.0MHz, transition Width: 100.0kHz

Number of stages: 1
Stage1: FIR Decimator, M = 48 (48.0MHz to 1.0MHz), FIR Length = 2411
```

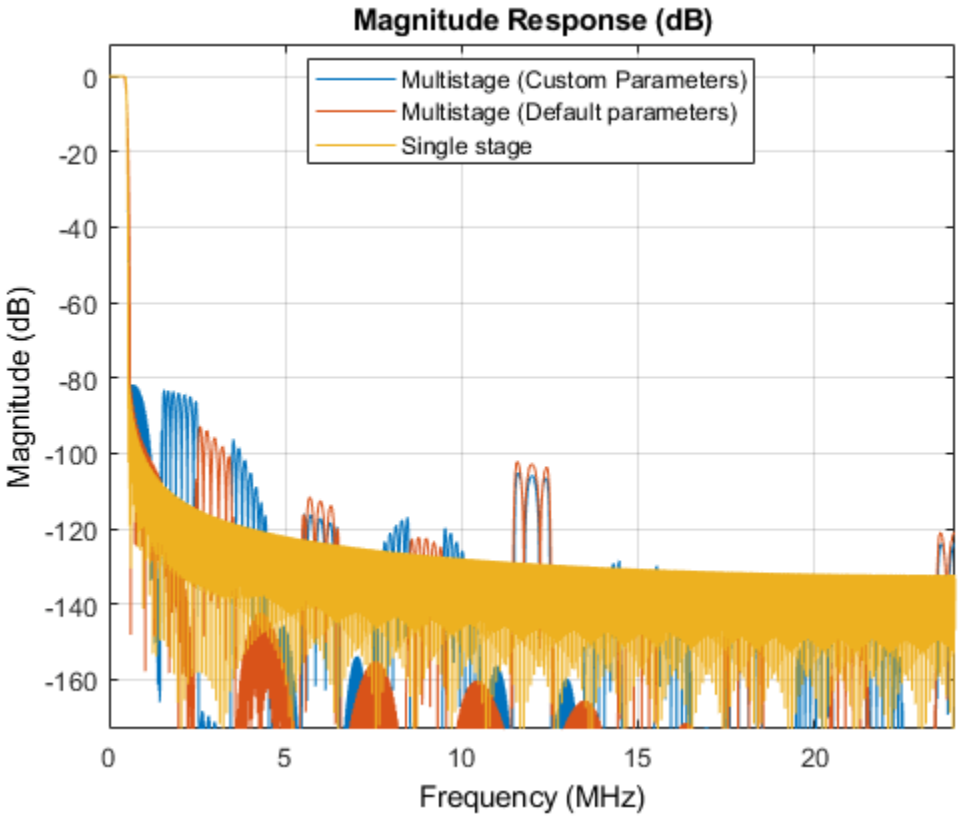
Compare the filter costs for the single-stage and the multistage implementations. The number of multiplications per input sample for the multistage approach is about 7, and roughly 49 for the single-stage implementation. In other words, using the multistage implementation reduces the number of multiplications by a factor of 7, which makes a significant difference. Similar differences can be observed in the number of coefficints (89 v.s. 2361), number of states (146 v.s. 2400), and additions per input sample (6 v.s. 49).

```
costMultiDecim  = cost(multiDecim)

costSingleDecim = cost(singleDecim)
```

```
costMultiDecim =

  struct with fields:

                 NumCoefficients: 89
                       NumStates: 146
    MultiplicationsPerInputSample: 6.6042
          AdditionsPerInputSample: 5.6667


costSingleDecim =

  struct with fields:

                 NumCoefficients: 2361
                       NumStates: 2400
    MultiplicationsPerInputSample: 49.1875
          AdditionsPerInputSample: 49.1667
```

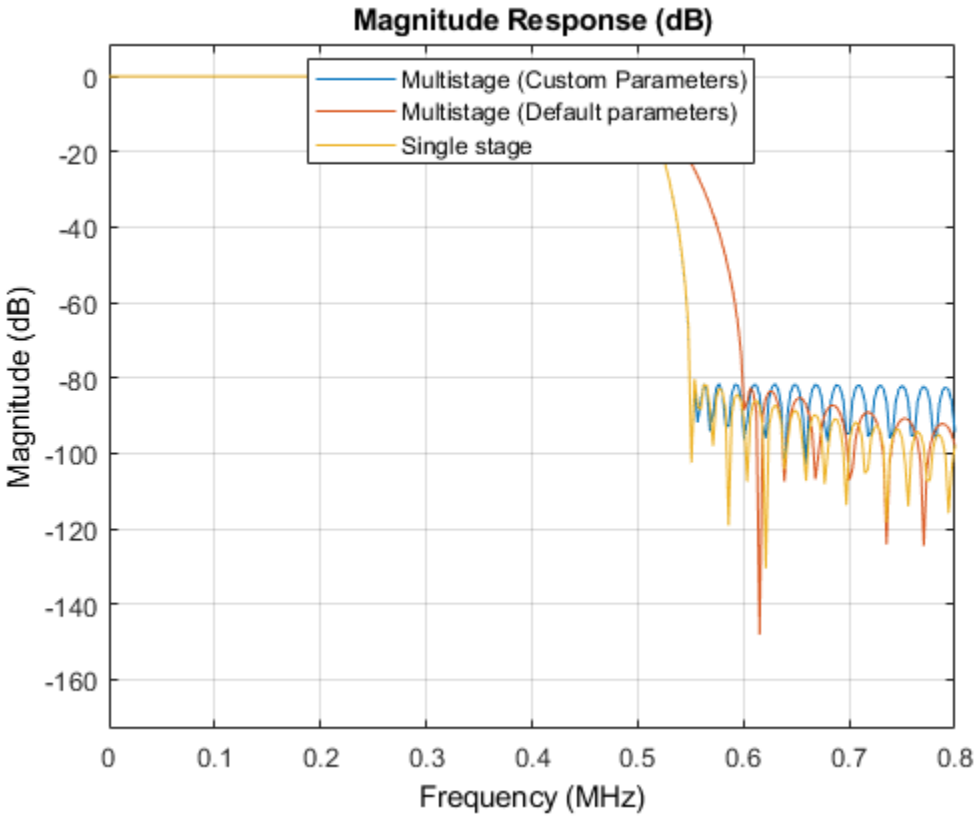
Compare the frequency responses of the single-stage implementation and the single-stage equivalents of the two multistage designs. The gain responses of the three implementations are very similar on the passband and transition band, and have negligible differences on the stopband. In spite of the significant cost difference, the lowpass filtering in all three designs is almost the same.

```
hfv = fvtool(multiDecim, multiDecim_default, singleDecim);
legend(hfv, 'Multistage (Custom Parameters)','Multistage (Default parameters)','Single stage')
```



The default design has a slightly larger transition width.

```
hfv = fvtool(multiDecim, multiDecim_default, singleDecim);
legend(hfv, 'Multistage (Custom Parameters)','Multistage (Default parameters)','Single stage')
xlim([0 0.8])
```



Optimial Designs for a Minimimal Total Number of Coefficients

By default, the design minimizes multiplications per input sample. It is also possible to minimize the number of coefficients. Set the property `MinTotalCoeffs = true` to use the latter cost.

```
minCoeffDecim = designMultistageDecimator(M,Fs,TW,Astop,'MinTotalCoeffs',true);
disp(cascadeInfoSummary(minCoeffDecim))
cost(minCoeffDecim)
```

```
Multistage FIR Decimator, M=48 (48.0MHz to 1.0MHz)
Equivalent lowpass cutoff: 1.0MHz, transition Width: 100.0kHz

Number of stages: 5
Stage1: FIR Decimator, M = 2 (48.0MHz to 24.0MHz), FIR Length = 7
Stage2: FIR Decimator, M = 3 (24.0MHz to 8.0MHz),  FIR Length = 17
Stage3: FIR Decimator, M = 2 (8.0MHz to 4.0MHz),   FIR Length = 11
Stage4: FIR Decimator, M = 2 (4.0MHz to 2.0MHz),   FIR Length = 23
Stage5: FIR Decimator, M = 2 (2.0MHz to 1.0MHz),   FIR Length = 95

ans =

  struct with fields:

                NumCoefficients: 87
                      NumStates: 147
    MultiplicationsPerInputSample: 6.8125
          AdditionsPerInputSample: 6
```

Compared to `multiDecim`, the number of coefficients in `minCoeffDecim` is lower, but the number of multiplications per input sample is higher.

Estimate vs. Design for Determining Cost

The objective function of the optimal design (either the number of coefficients or multiplications) that the designMultistageDecimator function uses depends on the FIR length of every stage. By default, this function evaluates the cost using an estimate of the FIR length rather than a true length, sometimes leading to a sub-optimal filter design.

A slower, but more precise method uses a cost based on the true FIR lengths obtained through actual designs of all filter candidates. Use the property `CostMethod='design'` to optimize for the accurate cost. Setting this property ensures that the design cost is indeed minimal.

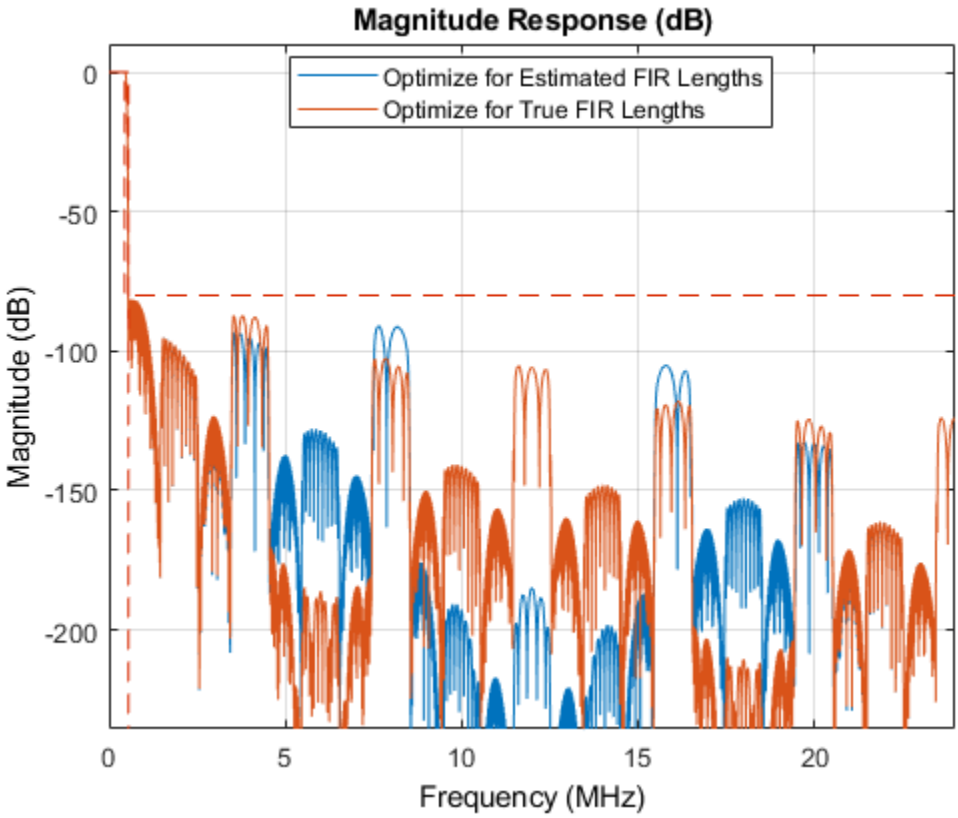
```
trueMinCostDecim = designMultistageDecimator(M,Fs,TW,Astop, 'CostMethod','design');

disp(cascadeInfoSummary(trueMinCostDecim))
```

```
Multistage FIR Decimator, M=48 (48.0MHz to 1.0MHz)
Equivalent lowpass cutoff: 1.0MHz, transition Width: 100.0kHz

Number of stages: 5
Stage1: FIR Decimator, M = 2 (48.0MHz to 24.0MHz), FIR Length = 7
Stage2: FIR Decimator, M = 2 (24.0MHz to 12.0MHz), FIR Length = 7
Stage3: FIR Decimator, M = 3 (12.0MHz to 4.0MHz),  FIR Length = 21
Stage4: FIR Decimator, M = 2 (4.0MHz to 2.0MHz),   FIR Length = 23
Stage5: FIR Decimator, M = 2 (2.0MHz to 1.0MHz),   FIR Length = 95
```

The estimated cost performs very well in many cases (as it does in this example).

```
cost(trueMinCostDecim)

hfv = fvtool(minCoeffDecim,trueMinCostDecim);
legend(hfv, 'Optimize for Estimated FIR Lengths', 'Optimize for True FIR Lengths')
```

```
ans =

  struct with fields:

                 NumCoefficients: 87
                       NumStates: 146
    MultiplicationsPerInputSample: 6.5625
          AdditionsPerInputSample: 5.6667
```



The `dsp.SampleRateConverter` System Object

The `dsp.SampleRateConverter` system object provides a convenient interface for arbitrary rate conversion, combining interpolation and decimation as needed.

```
src = dsp.SampleRateConverter('InputSampleRate',18,'OutputSampleRate',16,'Bandwidth',13);
info(src)
```

```
ans =

    'Overall Interpolation Factor    : 8
     Overall Decimation Factor       : 9
     Number of Filters               : 1
     Multiplications per Input Sample: 24.333333
     Number of Coefficients          : 219
     Filters:
        Filter 1:
        dsp.FIRRateConverter - Interpolation Factor: 8
                             - Decimation Factor   : 9
     '
```

The different stages can be extracted with the `getFilters` function:

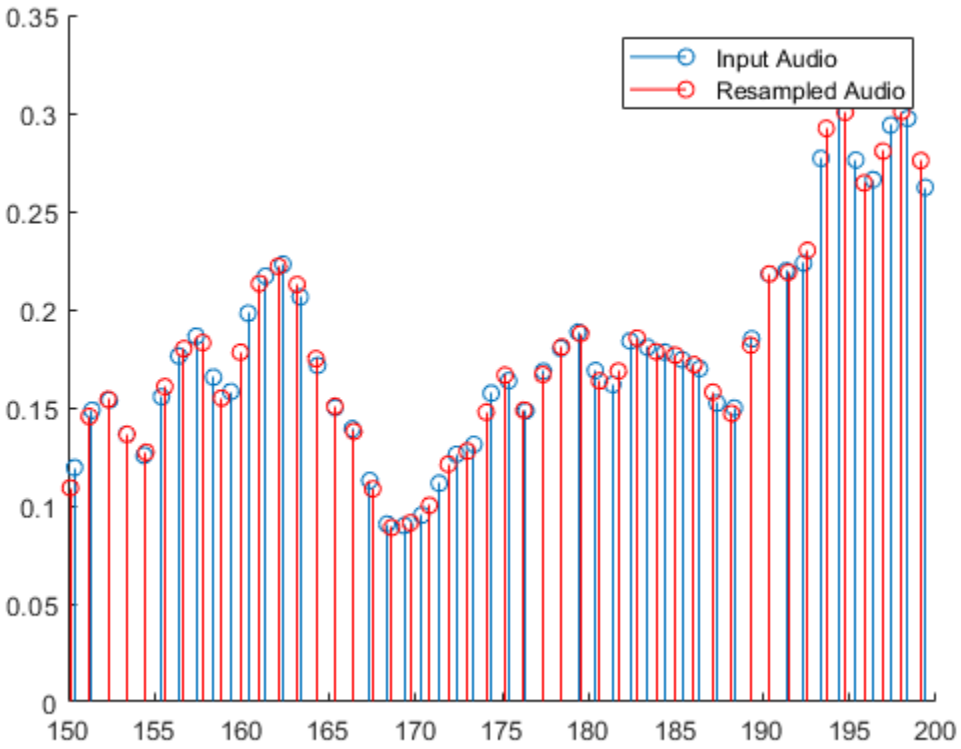
```
firs = getFilters(src)
```

```
firs =
```

```
  dsp.FilterCascade with properties:

    Stage1: [1x1 dsp.FIRRateConverter]
```

We can also specify absolute frequencies (rather than ratios). For example, the `dsp.SampleRateConverter` object can convert audio data sample rate from 48 kHz to 44.1 kHz.

```
src = dsp.SampleRateConverter('InputSampleRate',48000,'OutputSampleRate',44100);
[L,M] = getRateChangeFactors(src);

firs = getFilters(src);

reader = dsp.AudioFileReader('audio48kHz.wav','SamplesPerFrame',4*M);

x = reader();
xr = src(x);

% Obtain the rate conversion FIR
b = firs.Stage1.Numerator;

% Calculate the resampling delay
i0 = floor(length(b)/2)/L;

figure;
hold on;
stem((1:length(x))+i0,x);
stem(linspace(1,length(x),length(xr)),xr,'r');
hold off;
legend('Input Audio','Resampled Audio');
xlim([150,200])

release(reader);
```



Simplification by Rate Conversion Slack

Conversion ratios like $48k/44.1k$ (used in the previous section) requires a large upsample and downsample ratios, as even its reduced form is $L/M = 160/147$. The filters required for such a conversion are fairly long, introducing a significant latency in addition to the memory and computational load.

```
cost(src)
```

```
ans =

  struct with fields:

                NumCoefficients: 8587
                     NumStates: 58
  MultiplicationsPerInputSample: 53.6688
        AdditionsPerInputSample: 52.7500
```

We can mitigate the costly conversion by approximating the rate conversion factor. For example,

$$48kHz/44.1kHz \approx 48kHz/44kHz = 12/11.$$

The deviation of 100Hz is small, only 0.23 % of the absolute frequencies. The `dsp.SampleRateConverter` can automatically approximate the rate conversion factor by allowing

the output frequency to be perturbed. The perturbation tolerance is specified through the `'OutputRateTolerance'` property. The default tolerance is 0, meaning, no slack. In other words, slack means the deviation from the specified output rate value. Clearly, the approximated rate conversion has much smaller computational cost, and suffices for many applications, such as standard definition audio processing.

```
src_approx = dsp.SampleRateConverter('InputSampleRate',48000,...
             'OutputSampleRate',44100,'Bandwidth',13,...
             'OutputRateTolerance',0.01);
[L_approx,M_approx] = getRateChangeFactors(src_approx)

cost(src_approx)
```

```
L_approx =

    11


M_approx =

    12


ans =

  struct with fields:

                  NumCoefficients: 61
                        NumStates: 5
    MultiplicationsPerInputSample: 5.0833
          AdditionsPerInputSample: 4.1667
```

See Also

References

[1] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989.

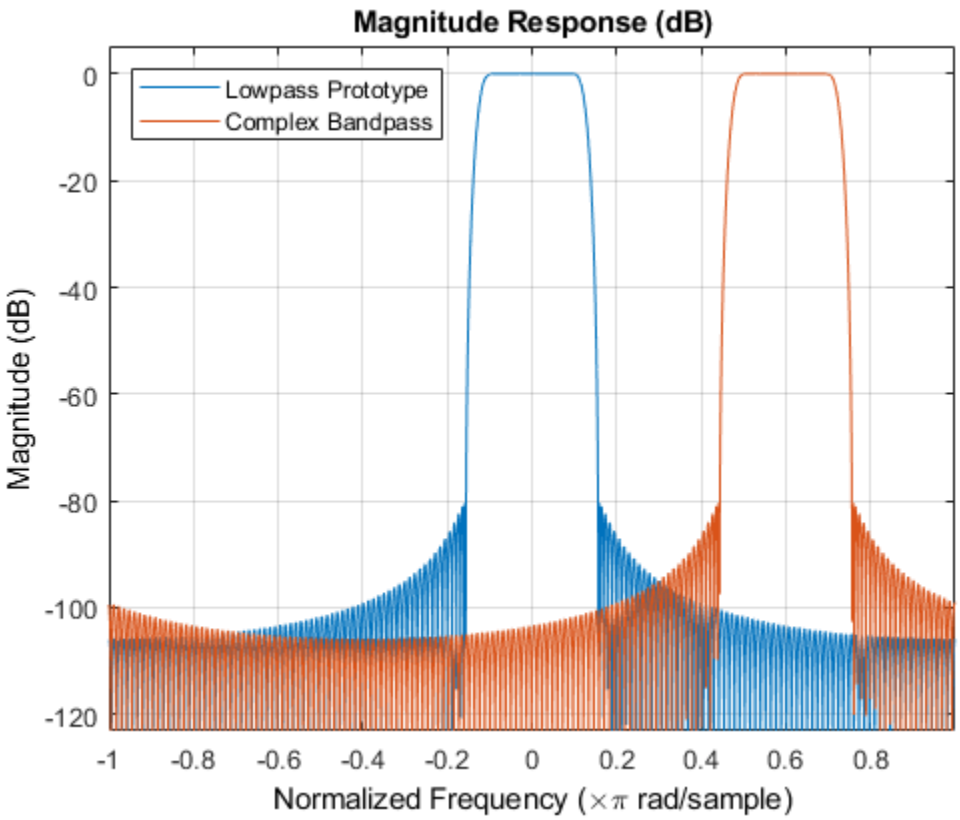
Complex Bandpass Filter Design

This example shows how to design complex bandpass filters. Complex bandpass filters are used in many applications from IF subsampling digital down converters to vestigial sideband modulation schemes for analog and digital television broadcast. One easy way to design a complex bandpass filter is to start with a lowpass prototype and apply a complex shift frequency transformation. In this example, we review several cases of lowpass prototypes from single-stage single-rate FIR filters to multistage multirate FIR filters to IIR filters.

Single-Stage Single-Rate FIR Design

In the case of a single-rate FIR design, we simply multiply each set of coefficients by (aka 'heterodyne with') a complex exponential. In the next example, we rotate the zeros of the lowpass Nyquist filter prototype by a normalized frequency of .6.

```
Hlp = design(fdesign.nyquist(8));     % Lowpass prototype
N = length(Hlp.Numerator)-1;
Fc = .6;                              % Desired frequency shift
j = complex(0,1);
Hbp = copy(Hlp);
Hbp.Numerator = Hbp.Numerator.*exp(j*Fc*pi*(0:N));
hfvt = fvtool(Hlp,Hbp,'Color','white');
legend(hfvt,'Lowpass Prototype','Complex Bandpass','Location','NorthWest')
```

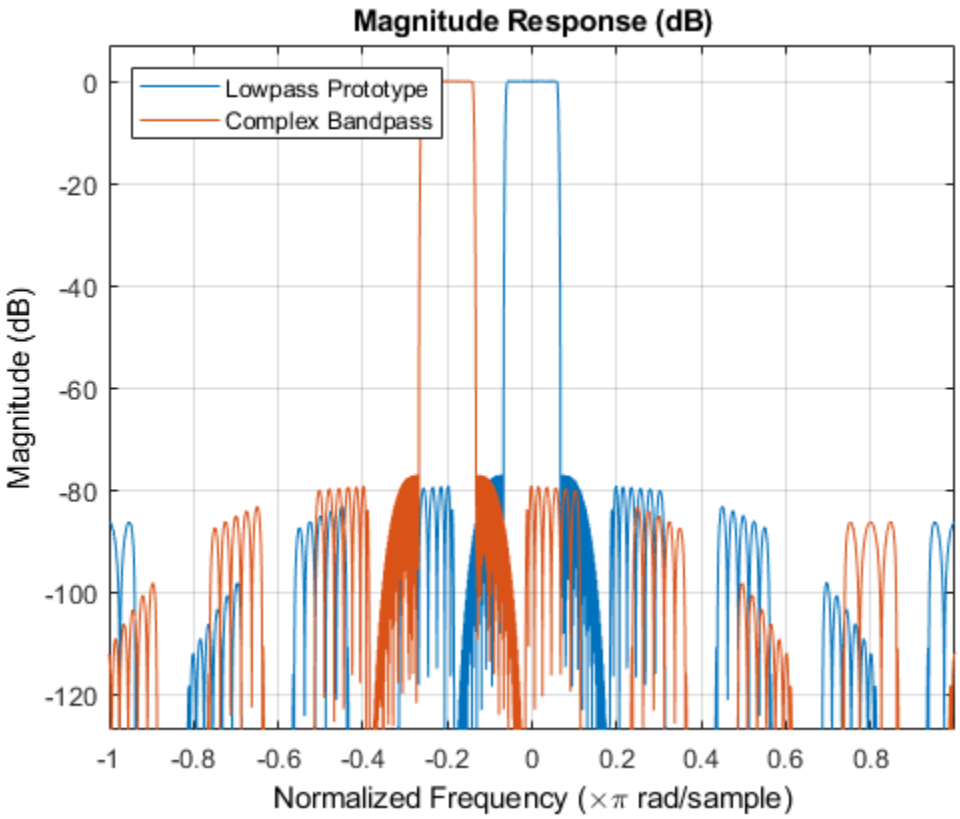


The same technique also applies to single-stage multirate filters.

Multirate Multistage FIR Design

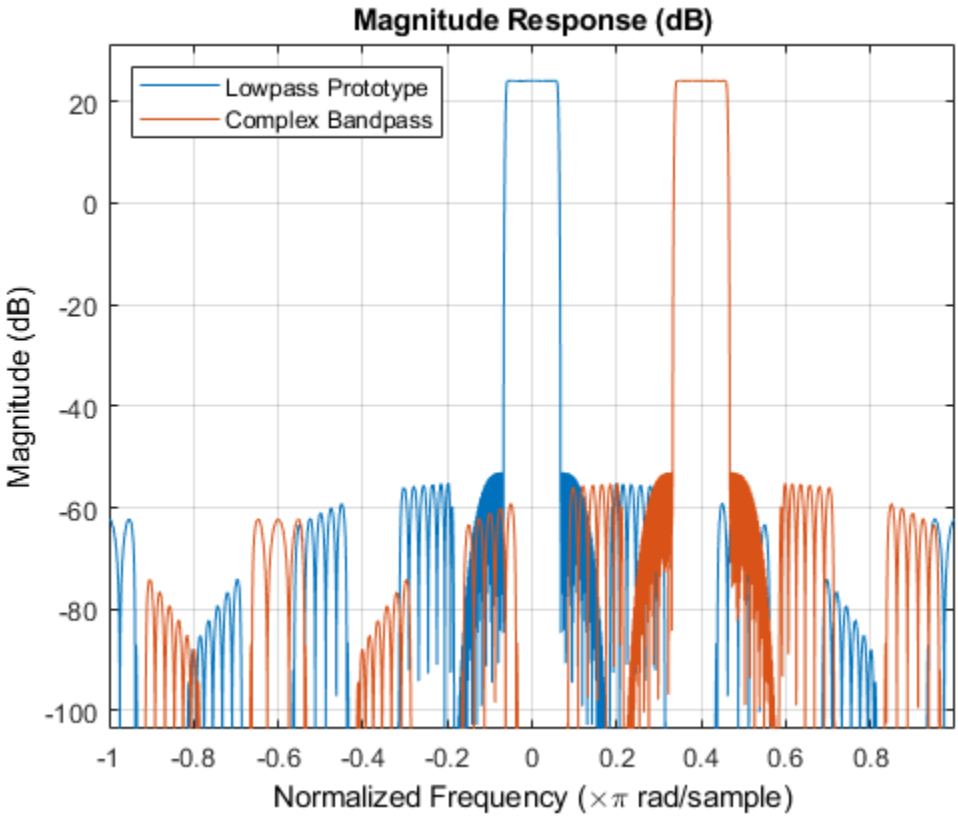
In the case of multirate multistage FIR filters, we need to account for the different relative frequencies each filter operates on. In the case of a multistage **decimator**, the desired frequency shift applies only to the **first** stage. Subsequent stages must also scale the desired frequency shift by their respective cumulative decimation factor.

```
f = fdesign.decimator(16,'nyquist',16,'TW,Ast',.01,75);
Hd = design(f,'multistage');
N1 = length(Hd.Stage(1).Numerator)-1;
N2 = length(Hd.Stage(2).Numerator)-1;
N3 = length(Hd.Stage(3).Numerator)-1;
M12 = Hd.Stage(1).DecimationFactor; % Decimation factor between stage 1 & 2
M23 = Hd.Stage(2).DecimationFactor; % Decimation factor between stage 2 & 3
Fc  = -.2;                          % Desired frequency shift
Fc1 = Fc;                       % Frequency shift applied to the first stage
Fc2 = Fc*M12;                   % Frequency shift applied to the second stage
Fc3 = Fc*M12*M23;               % Frequency shift applied to the third stage
Hdbp = copy(Hd);
Hdbp.Stage(1).Numerator = Hdbp.Stage(1).Numerator.*exp(j*Fc1*pi*(0:N1));
Hdbp.Stage(2).Numerator = Hdbp.Stage(2).Numerator.*exp(j*Fc2*pi*(0:N2));
Hdbp.Stage(3).Numerator = Hdbp.Stage(3).Numerator.*exp(j*Fc3*pi*(0:N3));

hfvt = fvtool([Hd,Hdbp],'Color','White');
legend(hfvt,'Lowpass Prototype','Complex Bandpass','Location','NorthWest')
```



Similarly, in the case of a multistage **interpolator**, the desired frequency shift applies only to the **last** stage. Previous stages must also scale the desired frequency shift by their respective cumulative interpolation factor.

```
f = fdesign.interpolator(16,'nyquist',16,'TW,Ast',.01,75);
Hi = design(f,'multistage');
N1 = length(Hi.Stage(1).Numerator)-1;
N2 = length(Hi.Stage(2).Numerator)-1;
N3 = length(Hi.Stage(3).Numerator)-1;
L12 = Hi.Stage(2).InterpolationFactor; % Interpolation factor
                                      % between stage 1 & 2
L23 = Hi.Stage(3).InterpolationFactor; % Interpolation factor
                                      % between stage 2 & 3
Fc = .4;                              % Desired frequency shift
Fc3 = Fc;                    % Frequency shift applied to the third stage
Fc2 = Fc*L23;                % Frequency shift applied to the second stage
Fc1 = Fc*L12*L23;            % Frequency shift applied to the first stage
Hibp = copy(Hi);
Hibp.Stage(1).Numerator = Hibp.Stage(1).Numerator.*exp(j*Fc1*pi*(0:N1));
Hibp.Stage(2).Numerator = Hibp.Stage(2).Numerator.*exp(j*Fc2*pi*(0:N2));
Hibp.Stage(3).Numerator = Hibp.Stage(3).Numerator.*exp(j*Fc3*pi*(0:N3));

hfvt = fvtool([Hi,Hibp],'Color','White');
legend(hfvt,'Lowpass Prototype','Complex Bandpass','Location','NorthWest')
```



We can design multistage bandpass filters easily by using the dsp.ComplexBandpassDecimator System object. The object designs the bandpass filter based on the specified decimation factor, center

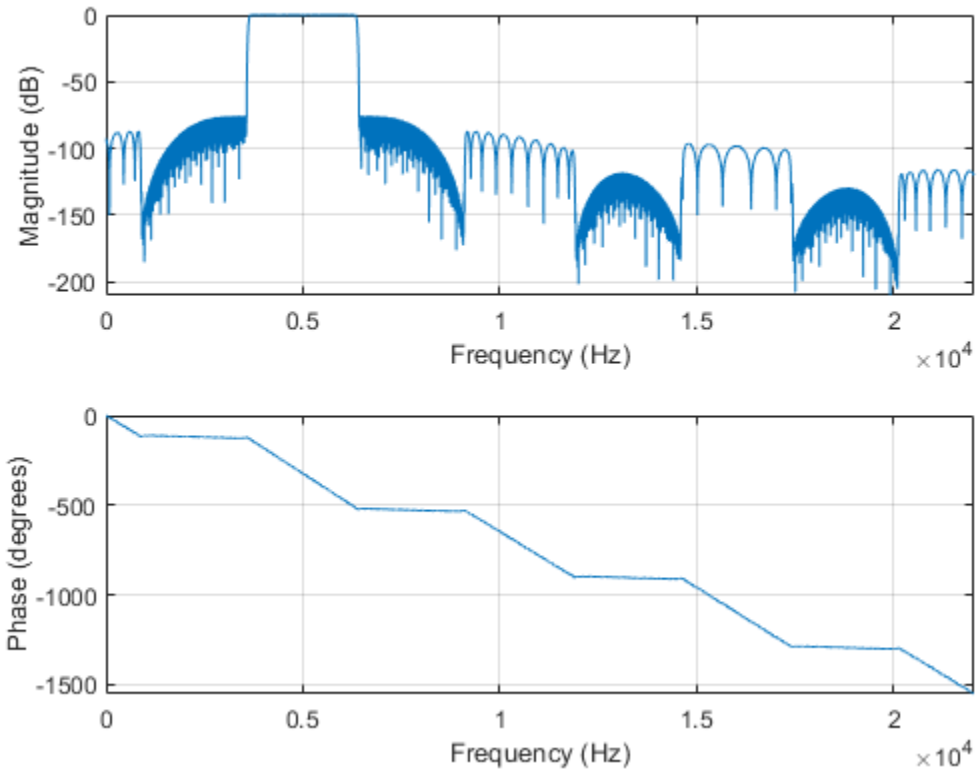
frequency, and sample rate. There is no need to translate lowpass coefficients to bandpass as we did in the section above: the object will do it for us.

Design a complex bandpass filter with a decimation factor of 16, a center frequency of 5 KHz, a sampling rate of 44.1 KHz, a transition width of 100 Hz, and a stopband attenuation of 75 dB:

```
bp = dsp.ComplexBandpassDecimator(16 , 5000, 'SampleRate',44100,...
                                  'TransitionWidth',100,...
                                  'StopbandAttenuation',75);
```

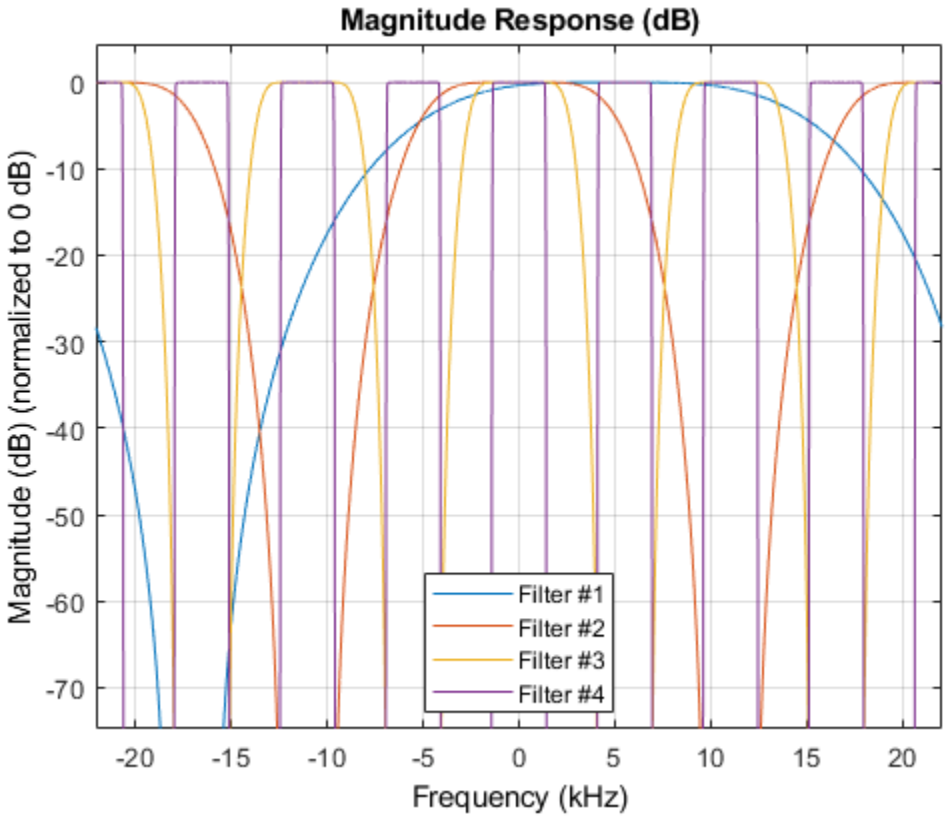
Visualize the filter response using `freqz`:

```
freqz(bp);
```



Visualize the response of the different filter stages using `visualizeFilterStages`:

```
visualizeFilterStages(bp);
```



Notice that only the first filter is shifted to 5 KHz. The subsequent filter stages are lowpass and have real coefficients. Set the `MinimizeComplexCoefficients` property to false to shift all filter stages to 5000 KHz.

Get the cost of the bandpass filter using `cost`:

```
cost(bp)
```

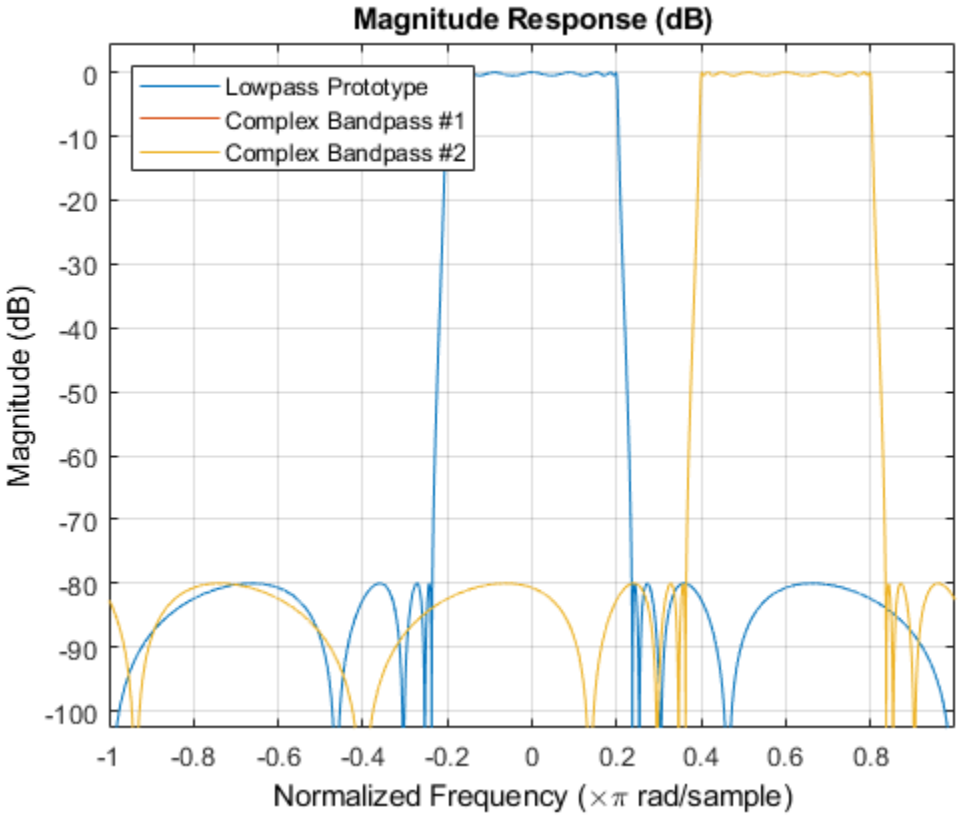
```
ans =

  struct with fields:

                     NumCoefficients: 144
                          NumStates: 272
    RealMultiplicationsPerInputSample: 27.8750
          RealAdditionsPerInputSample: 27
```

Single-Rate IIR Design

Finally in case of single-rate IIR designs, we can either use a complex shift frequency transformation or a lowpass to complex bandpass IIR transformation. In the latter case, the bandwidth of the bandpass filter may also be modified.

```
Fp = .2;
Hiirlp = design(fdesign.lowpass(Fp,.25,.5,80),'ellip');
```

```
Fc  = .6;                          % Desired frequency shift
Hiircbp = ciirxform(Hiirlp, ...    % Shift frequency transformation
    'zpkshiftc', 0, Fc);           % DC shifted to Fc
Hiircbp2 = iirlp2bpc(Hiirlp, ...   % Lowpass to complex bandpass transf.
    Fp, [Fc-Fp, Fc+Fp]);           % Lowpass passband frequency mapped
                                   % to bandpass passband frequencies

hfvt = fvtool([Hiirlp,Hiircbp,Hiircbp2],'Color','White');
legend(hfvt,'Lowpass Prototype','Complex Bandpass #1',...
    'Complex Bandpass #2','Location','NorthWest')
```



Design of Fractional Delay FIR Filters

Fractional delay filters are aimed at shifting a digital sequence by a noninteger value, through interpolation and resampling combined into a single convolution filter. This example demonstrates the design and implementation of fractional delay FIR filters using tools available in the DSP System Toolbox™.

Delay as a Convolution System

Integer Delays Delays

Consider the delay of a digital signal, $y[n] = x[n - D]$ where D is an integer. This operation can be represented as a convolution filter $y = h * x$, with a finite impulse response $h[n] = \delta[n - D]$. The corresponding transfer function is $H(z) = z^{-D}$, and the frequeny response is $H(\omega) = e^{-i\omega D}$. Programmatically, you can implement such an integer delay filter using the following MATLAB® code.

```
% Create the FIR
D = 3; % Delay value
h = [zeros(1,D) 1]
```

```
h = 1×4

     0     0     0     1
```

Shift a sequence by filtering it through the FIR h. Note the leading zeros at the beginning of the output, those signify the initial condition that is inherent to such filters.

```
x = (1:10)';
dfir = dsp.FIRFilter(h);
y = dfir(x)'
```

```
y = 1×10

     0     0     0     1     2     3     4     5     6     7
```

Non-Integer Delays via D/A Interpolation

Delaying a sequence $x[n - D]$ is not defined whenever D is not an integer. To make such fractional delays sensible, one needs to add an intermediate D/A interpolation stage so as to sample the output on the contniuum. That is, $y[n] = \widehat{x_n}(n - D)$ where $\widehat{x_n}$ denotes some D/A interpolation of the input sequnce x. The D/A interpolating function $\widehat{x_n}$ could depend on with n, *and could be thought of* as a representation of an underlying analog signal model from which the sequence x was sampled. This strategy is used in other resampling problems such as rate conversion.

This example will feature fractional delay filters using two interpolation models, both of which are offered as a part of the DSP Systems Toolbox.

1. The sinc-based interpolation model, which uses a bandlimited reconstruction for $\widehat{x_n}$.
2. The Lagrange-based interpolation model, which uses a polynomial reconstruction for $\widehat{x_n}$.

Bandlimited Fractional Delay Filters

The *Shannon-Whittaker* interpolation formula $\widehat{x}(t) = \sum_k x[k]\text{sinc}(t-k)$ models bandlimited signals. That is, the intermediate D/A conversion $\widehat{x}$ is a bandlimited reconstruction of the input sequence. For a delay value D, the fractional delay $y[n] = \widehat{x}(n-D)$, which uses the same $\widehat{x}$ for every n, can be represented as a convolution filter. This filter is called the *ideal bandlimited fractional delay filter, and its* impulse response is

$h_D[k] = \text{sinc}(k-D)$.

The corresponding frequency response (that is, DTFT) is given by $H_D(\omega) = e^{-i\omega D}$.

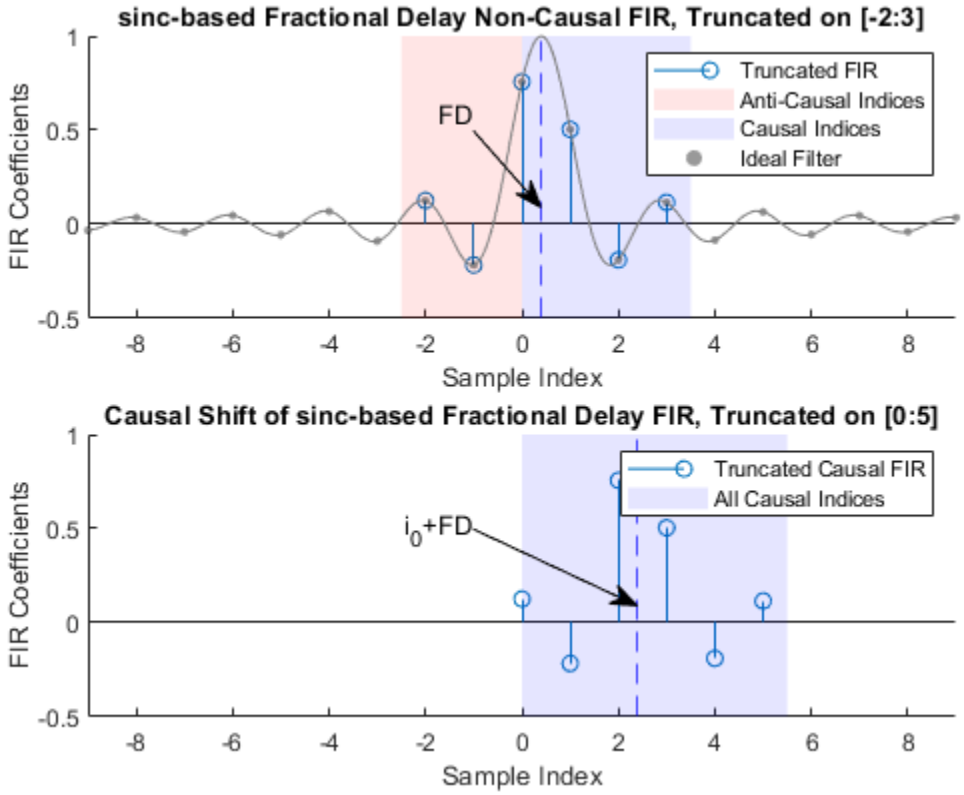
Causal FIR Approximation of the Ideal Bandlimited Shift Filter

The ideal *sinc* shift filter described in the previous section is an all-pass filter (i.e. $|H_d(\omega)| = 1$), but it has an infinite and non-causal impulse response h_D. In MATLAB, it cannot be represented as a vector, but rather as a function of an index k.

```
% Ideal Filter sequence
D = 0.4;
hIdeal = @(k) sinc(k-D);
```

For practical and computational purposes, the ideal filter can be truncated on a finite index window, at a cost of some bandwidth loss. For a target delay value of D and a desired length of N, the window of indices k satisfying $\left|D-k\right| \le \frac{N}{2}$ is symmetric about D, and captures the main lobe of the ideal filter. For $D = i_0 + FD$ where $0 \le FD \le 1$ and an integer i_0, the explicit window indices are $\{i_0 - \lfloor\frac{N-1}{2}\rfloor, \ldots, \ i_0 + \lfloor\frac{N}{2}\rfloor\}$. The integer i_0 is referred to as the *integer latency*, and can be chosen arbitrarily. To make the FIR causal, set $i_0 = \left\lfloor\frac{N-1}{2}\right\rfloor$, so the index window is $\{0, \ldots, N-1\}$. The code below depicts the rationale behind the causal FIR approximation.

```
% FIR approximation with causal shift
N = 6;
idxWindow = (-floor((N-1)/2):floor(N/2))';
i0 = -idxWindow(1); % Causal latency
hApprox = hIdeal(idxWindow);
plot_causal_fir("sinc",D,N,i0,hApprox,hIdeal);
```



Truncation of a sinc filter causes a ripple in the frequency response, which can be addressed by applying weights $\{w_k\}$ (such as Kaiser or Hamming) to the FIR coefficients .

Finally, the resulting FIR approximation model of the ideal bandlimited fractional delay filter is given below.

$$h[k] = w_k h_d[k] = \begin{cases} w_k sinc(k - FD - i_0) & 0 \le k \le N-1 \\ 0 & \text{otherwise} \end{cases}$$

You can design such a filter using the `designFracDelayFIR` function and the `dsp.VariableFractionalDelay` System object™ in `'FIR'` mode, both of which use Kaiser window weights.

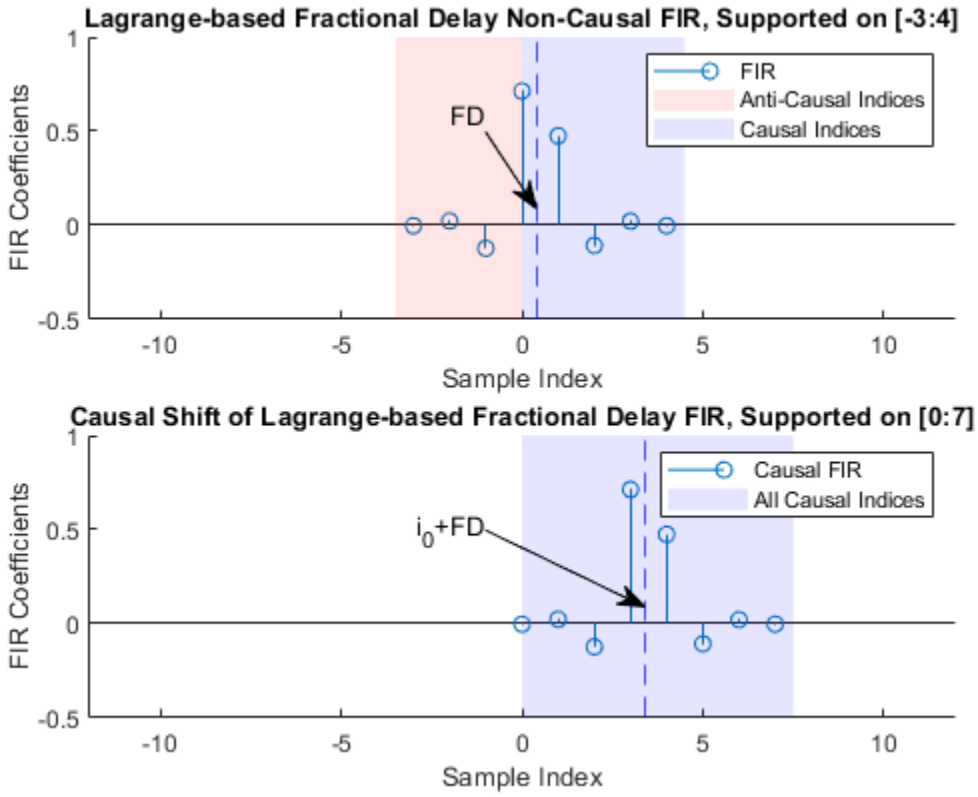
Lagrange-based Fractional Delay Filters

Lagrange-based fractional delay filters use polynmial fitting on a moving window of input samples. That is, $\widehat{x_n}(t)$ is a polynomial of some fixed degree K. Like the sinc-based delay filters, Lagrange-based delay filters can be formulated as a causal FIR convolution (i.e. $y = h * x$) of the length $N=K+1$, *and supported on* the index window$\{ -\lfloor\frac{N-1}{2}\rfloor, \ldots \lfloor\frac{N}{2}\rfloor\}$. Similarly to the sinc-based model, apply the causal latency $i_0 = \left\lfloor\frac{N-1}{2}\right\rfloor$. Given a fractional delay FD, the FIR coefficients $h[0], \ldots, h[K]$ of the (causal shifted) Lagrange delay filter can be obtained by solving a system of linear equaions, as written below. Those equations describe a standard Lagrange polynomial fitting problem.

$$\sum_{k=0}^{K} t_n^k h[k] = (FD)^n, \qquad n = 0, \ldots, \ K$$

Here, $t_0, \ldots, t_K$ are the enumerated indices of the sample window. The implementation is straightforward.

```
% Filter parameters
FD = 0.4;
K = 7;    % Polynomial degree
N = K+1;  % FIR Length
idxWindow = (-floor((N-1)/2):floor(N/2))';

% Define and solve Lagrange interpolation equations
V = idxWindow.^(0:K); % Vandermonde structure
C = FD.^(0:K);

hLagrange = C/V;  % Solve for the coefficients
i0 = -idxWindow(1); % Causal latency
plot_causal_fir("Lagrange",FD,N,i0,hLagrange);
```



This model can be implemented as a direct-form FIR filter if the delay value *FD* is fixed, or using a Farrow structure if the delay value is varying. There is a section below dedicated to the implementaiton of Lagrange interpolation using `dsp.VariableFractionalDelay` in `'Farrow'` mode.

Design And Implement sinc-Based Fractional Delay FIR Filters

The following section is focuses on designing and implementing sinc-based fractional delay filters.

The Function designFracDelayFIR in Length Design Mode

The function `designFracDelayFIR` provides a simple interface to design a fractional delay FIR filter of delay value *FD* and of length *N*.

```
FD = 0.32381;
N = 10;
h = designFracDelayFIR(FD,N)
```

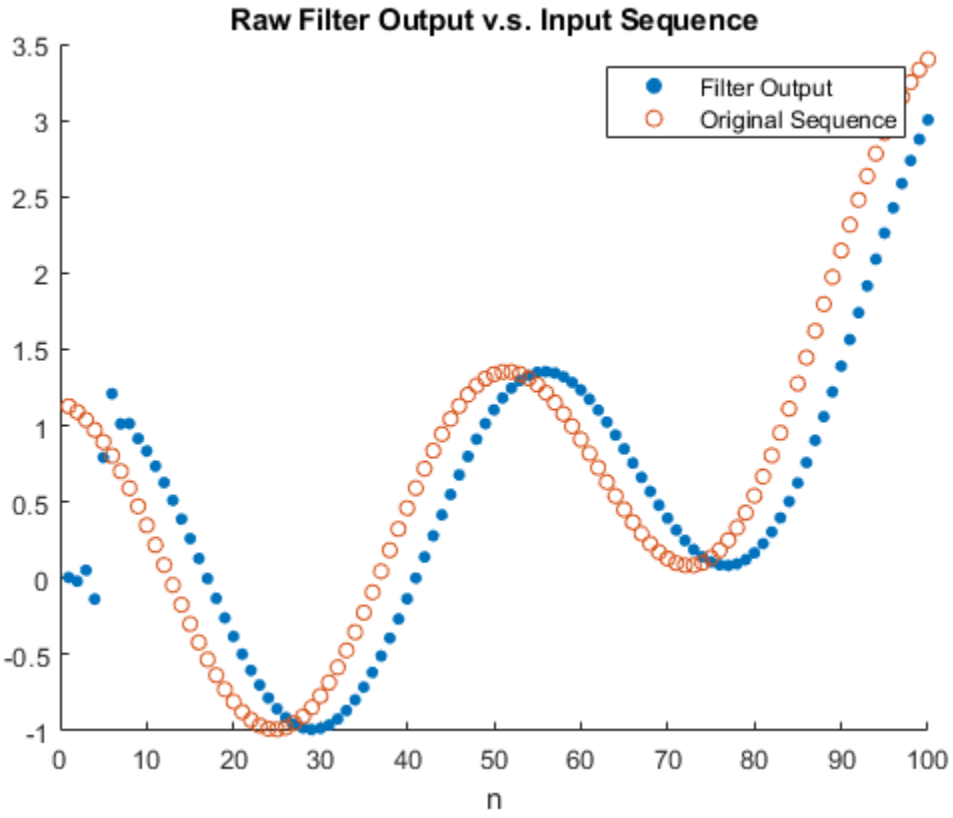
```
h = 1×10

    0.0046   -0.0221    0.0635   -0.1664    0.8198    0.3926   -0.1314    0.0552   -0.0200    0.0
```

The implementation itself can be done as a standard FIR filter, such as the `dsp.FIRFilter` System object.

```
% Create an FIR filter object
fdfir = dsp.FIRFilter(h);
```

Delay a signal by filtering it through the designed filter.

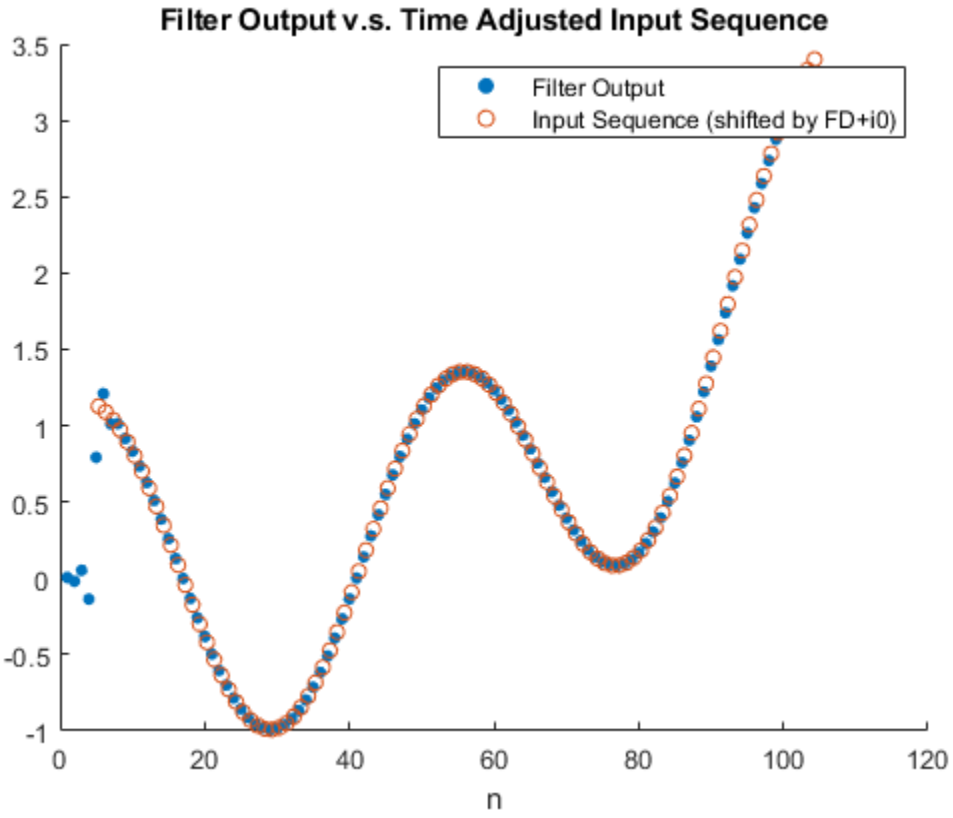
```
% Generate some input
n = (1:100)';
x = gen_input_signal(n);

% Filter the input signal
y = fdfir(x);
plot_sequences(n,x, n,y);
legend('Filter Output','Original Sequence')
title('Raw Filter Output v.s. Input Sequence')
```




```
release(fdfir);
```

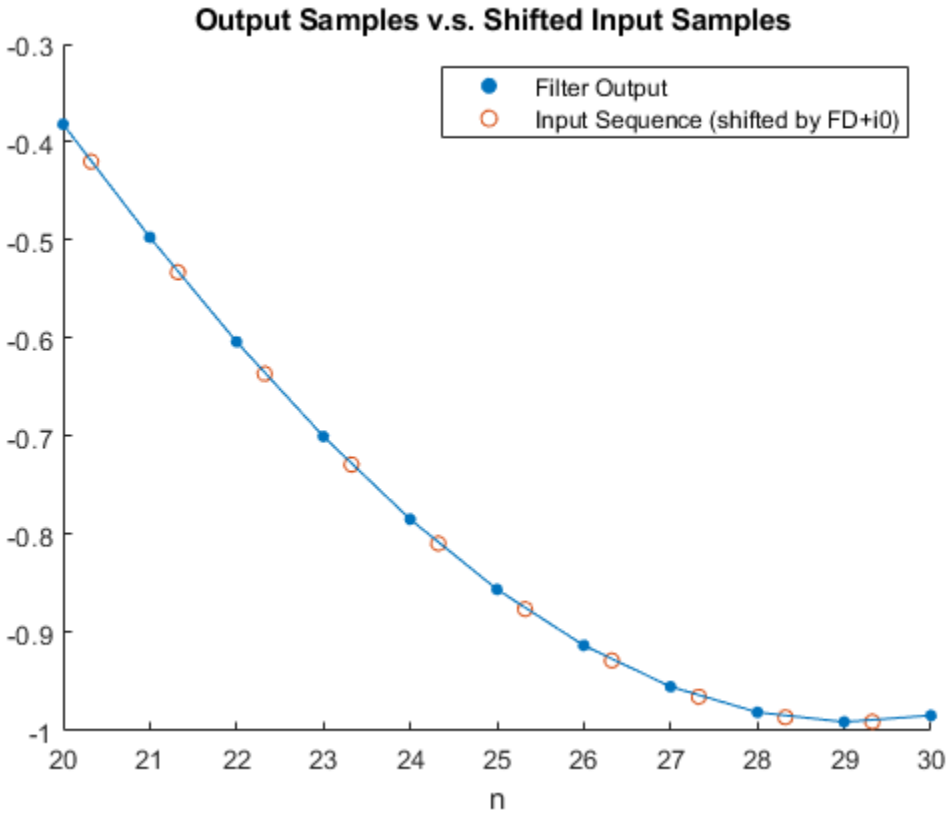
Notice that the actual filter delay is not *FD*, but rather $FD + i_0$ because of the causal integer latency i_0. That latency is returned from the `designFracDelayFIR` function as well. Shift the plot of the input sequence by $FD + i_0$ to align the filter output with the expected result.

```
[h,i0] = designFracDelayFIR(FD,N);
y = fdfir(x);
plot_sequences(n+i0+FD,x, n,y);
legend('Filter Output','Input Sequence (shifted by FD+i0)')
title('Filter Output v.s. Time Adjusted Input Sequence')
```



Note that the shifted input markers located at $(n + FD + i_0,\ x[n])$ generally do not coincide with output samples markers $(n,\ y[n])$, because $n + FD + i_0$ falls on noninteger values on the x-axis, whereas n is integer. Rather, the shifted input samples fall approximately on a line connecting each two consecutive output samples.

```
plot_sequences(n+i0+FD,x, n,y,'with line');
legend('Filter Output','Input Sequence (shifted by FD+i0)')
title('Output Samples v.s. Shifted Input Samples ')
xlim([20,30])
```



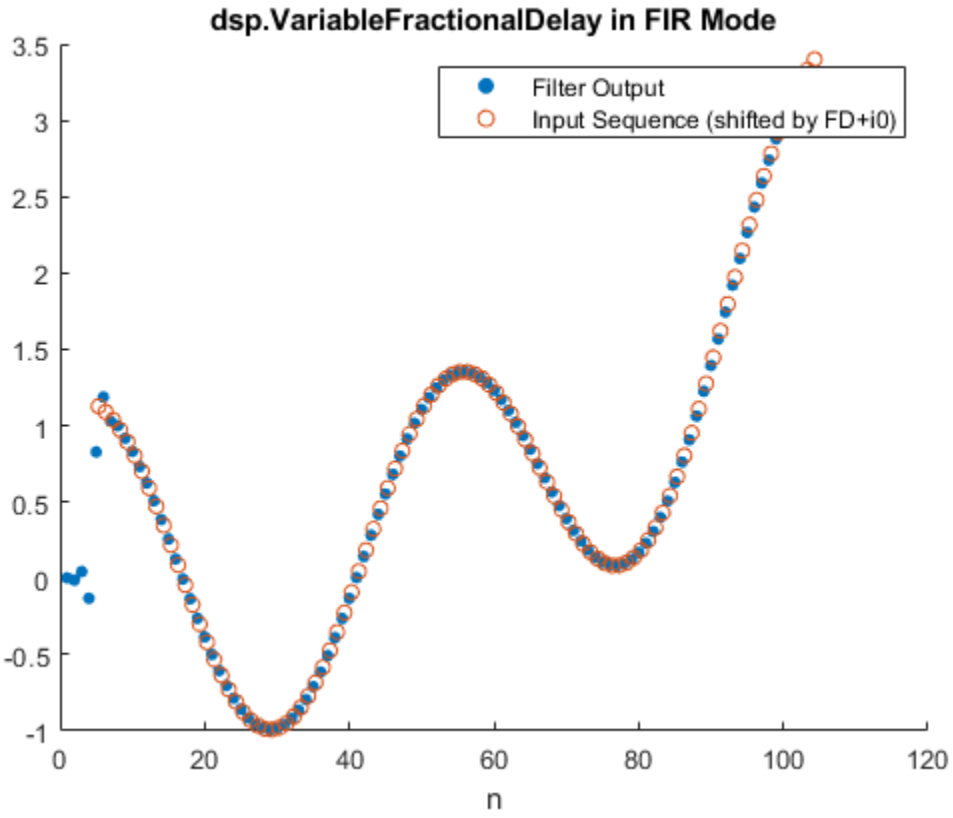
The `dsp.VariableFractionalDelay` System Object in `'FIR'` Mode

Similarly to `designFracDelayFIR`, the `dsp.VariableFractionalDelay` object can also design sinc-based delay filters when used with the `'FIR'` inteprolation mode. Begin by creating an instance of the System object. The FIR length is always even, and is specified as a half-length parameter.

```
vfd_fir = dsp.VariableFractionalDelay('InterpolationMethod','FIR','FilterHalfLength',N/2);
i0_vfd_fir = vfd_fir.FilterHalfLength;    % Interger latency
```

Pass the desired fractional delay as the second input argument to the object call. Make sure that the delay value you specify includes the integer latency.

```
y = vfd_fir(x,i0+FD);
release(vfd_fir)
plot_sequences(n+i0+FD,x, n,y);
legend('Filter Output','Input Sequence (shifted by FD+i0)')
title('dsp.VariableFractionalDelay in FIR Mode')
```



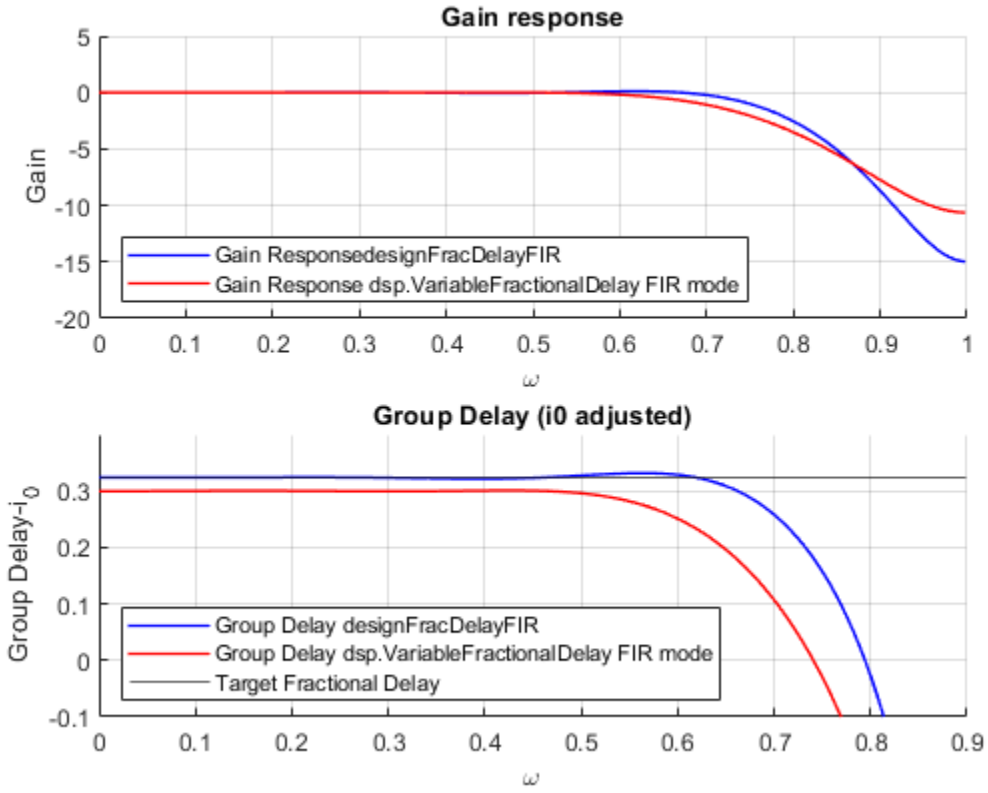
Comparison of `designFracDelayFIR` and `dsp.VariableFractionalDelay` in `'FIR'` Mode

Both `designFracDelayFIR` and `dsp.VariableFractionalDelay` in `'FIR'` mode provide sinc-based fractional delay filters, but their implementations are different.

- The `dsp.VariableFractionalDelay` approximates the delay value by a rational number $FD \approx {}^{k}/_{L}$ up to some tolerance, and then samples the fractional delay as the k-th phase of a (long) interpolation filter of length L. This requires increased memory use, and yields less accurate delay.
- By contrast, `designFracDelayFIR` generates the FIR coefficients directly, rather than sampling them from a longer FIR. This gives the precise fractional delay value, and costs less memory.
- The `designFracDelayFIR` has a simple function interface returning the FIR coefficients, leaving the filter implementation to the the user. The `dsp.VariableFractionalDelay` is System object meant to encapsualte the filter design and implementation entirely.

The use of `designFractionalDelayFIR` is preferred over `dsp.VariableFractionalDelay` in `'FIR'` mode for its simplicity, better performance, and efficiency. In the figure below, the filter designed with `dsp.VariableFractionalDelay` has a shorter bandwidth, and its group delay is off by ~0.02 from the nominal value.

```
% Obtain the FIR coefficients from the dsp.VariableFractionalDelay object
h_vfd_fir = vfd_fir([1;zeros(31,1)],i0_vfd_fir+FD);
release(vfd_fir);
plot_freq_and_gd(h,i0,[],"designFracDelayFIR", h_vfd_fir,i0_vfd_fir,[],"dsp.VariableFractionalDe
hold on;
yline(FD,'DisplayName','Target Fractional Delay');
ylim([-0.1,0.4])
```



Design And Implement Lagrange-based Delay Filters

Lagrange-based fractional delay filter are computationally cheap and can be implemented efficiently using the Farrow structure. The Farrow filter is a special type of FIR that is implemented using only elementary algebraic operations, such as scalar additions and multiplications. Unlike the sinc-based designs, Farrow filters do not require specialized functions (such as *sinc* or *Bessel*) to compute the delay FIR coefficients. This makes Farrow fractional delay filters particularly simple to implement on a basic hardware.

On the downside, Lagrange-based delay filters are limited to low orders, due to the highly unstable nature of polynomial approximations of high degree. This usually results with a lower bandwidth, when compared with a sinc-based filter.

The System Object `dsp.VariableFractionalDelay` in `'Farrow'` mode

Use the system object `dsp.VariableFractionalDelay` in `'Farrow'` mode to create and implement Farrow delay filters. Begin by creating an instance of the system object:

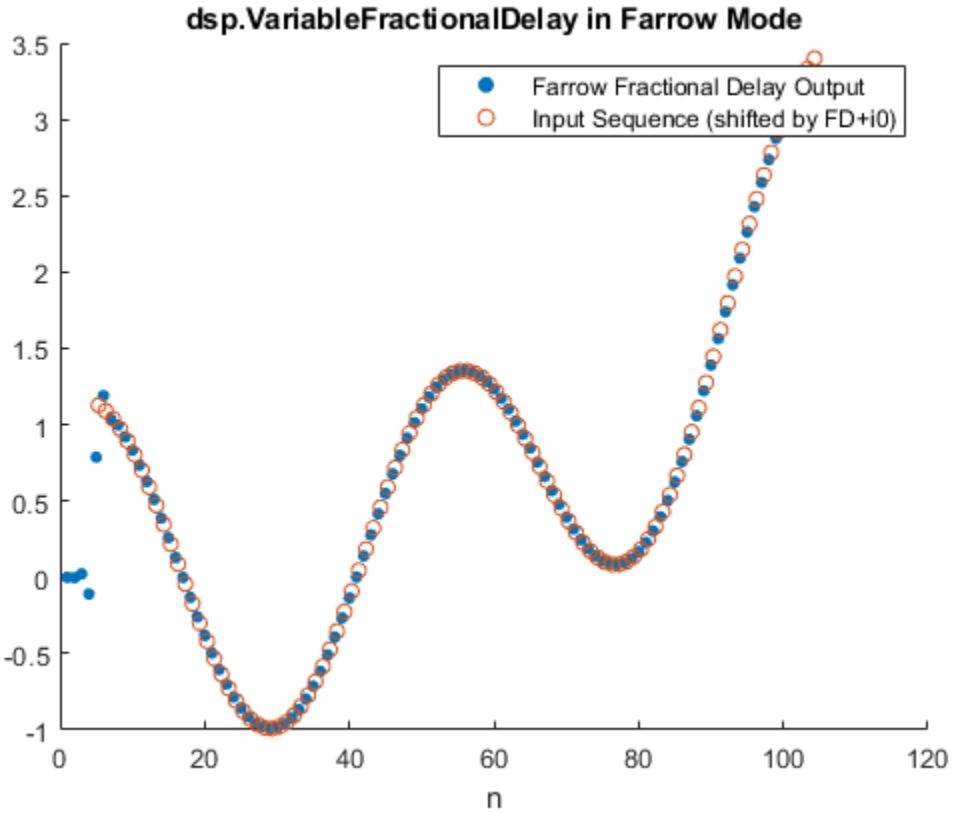
```
vfd = dsp.VariableFractionalDelay('InterpolationMethod','Farrow','FilterLength',8);
i0var = floor(vfd.FilterLength/2)  % Interger latency of the filter
```

```
i0var = 4
```

Apply the created object on the input signal, and plot the result.

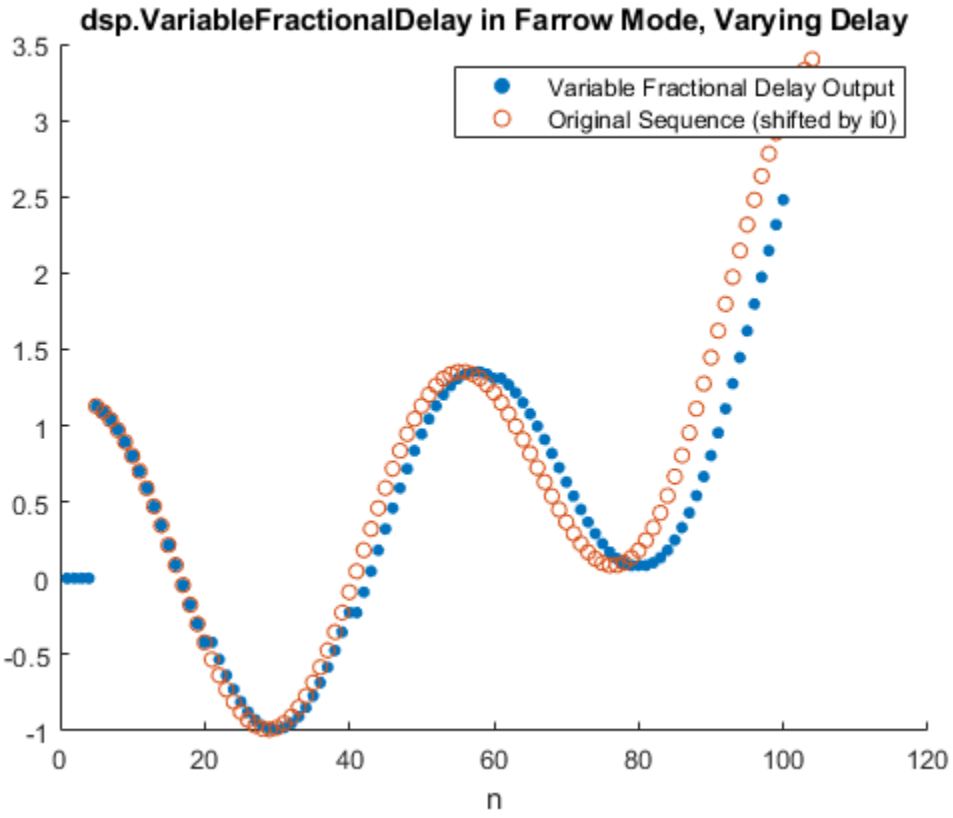
```
y = vfd(x,i0var+FD);
plot_sequences(n+i0var+FD,x, n,y);
```

```
legend('Farrow Fractional Delay Output','Input Sequence (shifted by FD+i0)')
title('dsp.VariableFractionalDelay in Farrow Mode')
```



You can also vary the fractional delay values. The code below operates on frames of 20 samples, while increasing the delay value with each frame. Note the increase of the delay in the output graph, corresponding to the changes in the delay values.

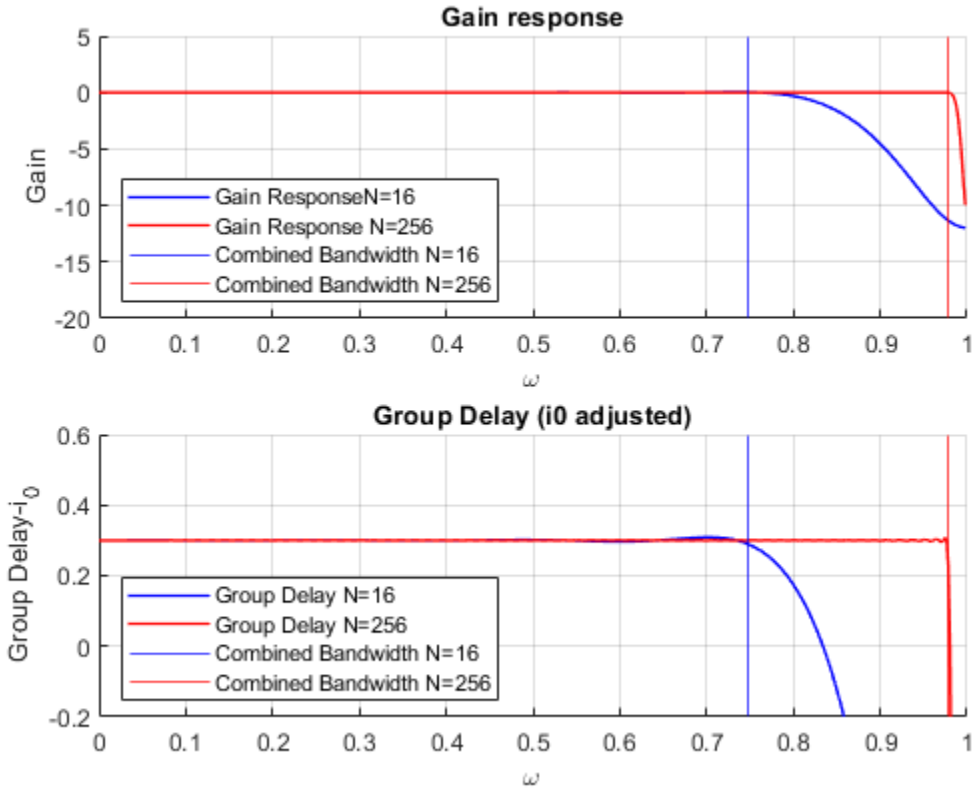
```
release(vfd)
FDs = i0var+5*(0:0.2:0.8); % Fractional delays vector
xsource = dsp.SignalSource(x,20);
ysink = dsp.AsyncBuffer;
for FD=FDs
    xk = xsource();
    yk = vfd(xk, FD);
    write(ysink,yk);
end
y = read(ysink);

plot_sequences(n+i0var,x, n,y);
legend('Variable Fractional Delay Output','Original Sequence (shifted by i0)')
title('dsp.VariableFractionalDelay in Farrow Mode, Varying Delay')
```



Bandwidth of FIR Fractional Delay Filters: Analysis and Design

Longer filters give better approximation of the ideal delay filter. Indeed, in terms of raw quadratic norms it is the case. However, we need a metric that is more practically meaningful, such as the bandwidth. The function `designFracDelayFIR` measures combined bandwidth, which is defined as the frequency range in which both the gain and the group delay are within 1% of their nominal values. The measured combined bandwidth can be obtained as a return value of the `designFracDelayFIR` function. Compare a filter of length 16 (blue) with a filter of length 256 (red) in the figure below. As expected, the longer filter have significantly higher combined bandwidth.

```
FD = 0.3;
N1 = 16;
N2 = 256;
[h1,i1,bw1] = designFracDelayFIR(FD, N1);
[h2,i2,bw2] = designFracDelayFIR(FD, N2);

plot_freq_and_gd(h1,i1,bw1,"N="+num2str(N1), h2,i2,bw2,"N="+num2str(N2));
ylim([-0.2,0.6])
```



The Function designFracDelayFIR in Bandwidth Design Mode

The bandwidth design mode of `designFracDelayFIR` can determine the required length for a given bandwidth. Specify the delay value and the desired target bandwidth as inputs to the function, and the function will find the appropriate length.

```
FD = 0.3;
bwLower = 0.9; % Target bandwidth lower limit
[h,i0fixed,bw] = designFracDelayFIR(FD,bwLower);
fdfir = dsp.FIRFilter(h);
info(fdfir)
```

```
ans = 6x35 char array
    'Discrete-Time FIR Filter (real)    '
    '-------------------------------    '
    'Filter Structure  : Direct-Form FIR'
    'Filter Length     : 52             '
    'Stable            : Yes            '
    'Linear Phase      : No             '
```

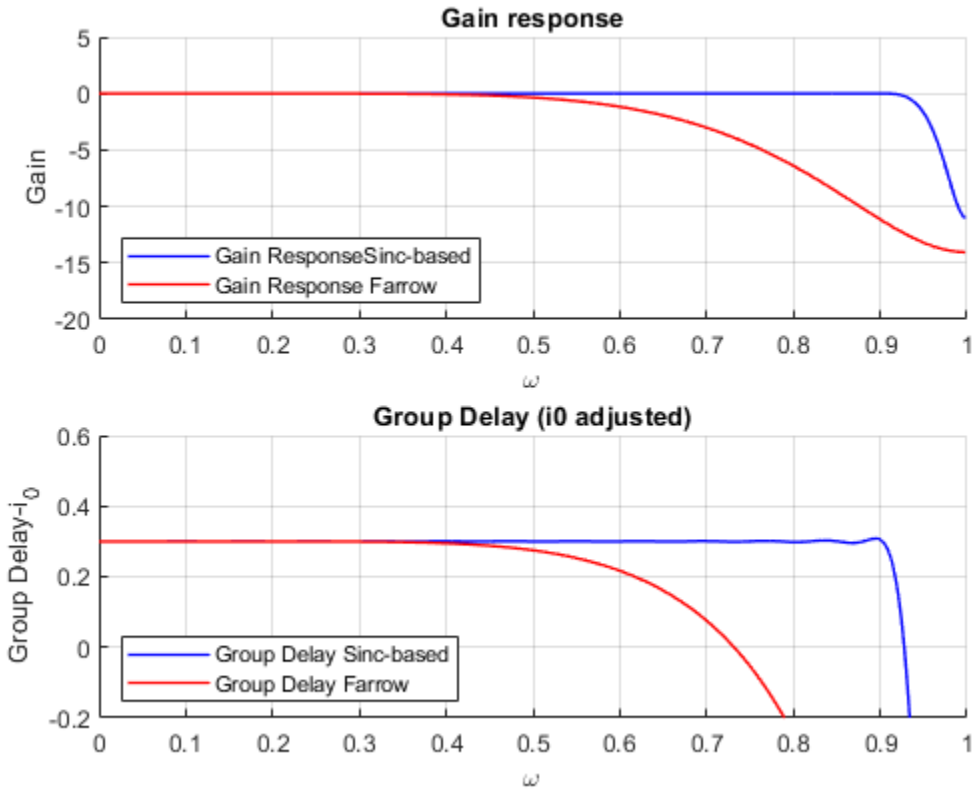
Note that `bwLower` is merely a lower bound for the combined bandwidth. The function returns a filter whose combined bandwidth is at least the value specified in `bwLow`.

Distortion in High Bandwidth Signals

In this section, we compare the performance of the two design points (long sinc v.s. short Lagrange) with a high bandwidth input. The `dsp.VariableFractionalDelay` in the previous section is an 8-

degree Farrow structure, effectively an FIR of length 9. The filter obtained by `designFracDelayFIR(FD,0.9)` has a length of 52 samples. Putting the two FIR frequency responses together on the same graph demonstrates the bandwidth difference between the two.

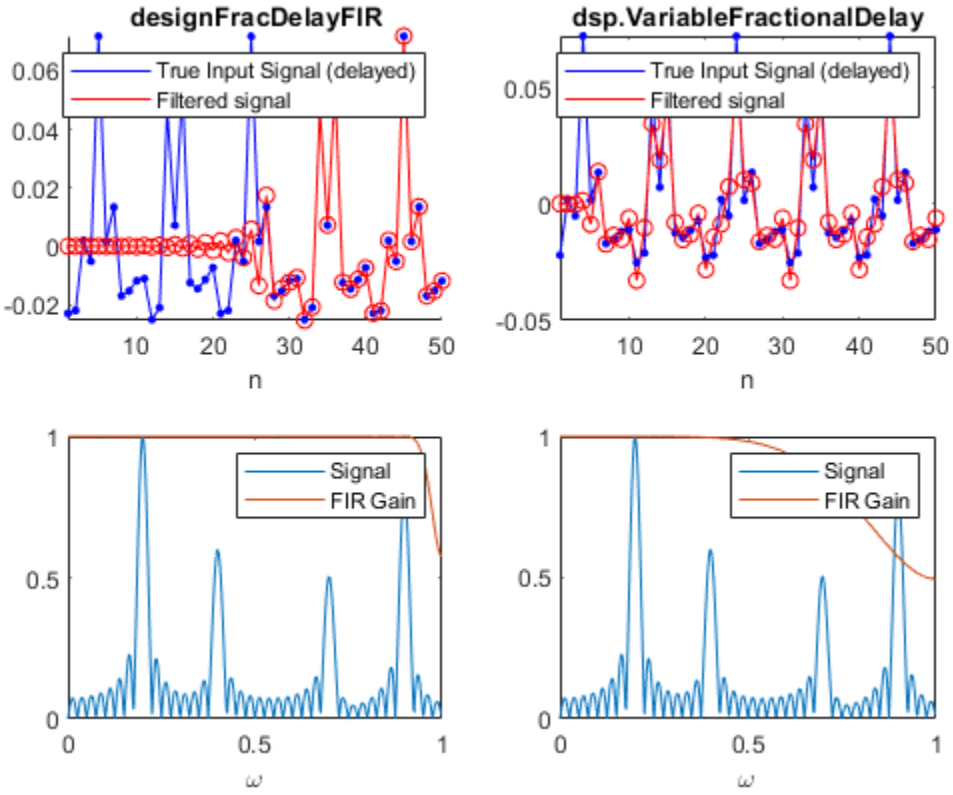
```
release(vfd);
hvar = vfd([1;zeros(31,1)],i0var+FD);
plot_freq_and_gd(h,i0fixed,bw,"Sinc-based", hvar,i0var,[],"Farrow");
ylim([-0.2,0.6])
```



Apply the two filters on a high bandwidth signal, as compared in figure below. Sinc on the left column, Farrow on the right. Time domain on top row, frequency on the bottom. The results are, as expected:

- The longer sinc filter has a higher bandwidth. The shorter Farrow filter has lower bandwidth.
- Signal distortion is virtually nonexistent using the longer sinc filter, but easily noticeable in the shorter Farrow filter.
- The higher accuracy comes at the expense of longer latency: approximately 25 samples v.s. only 4 in the shorter filter.

```
n=(1:80)';
x = high_bw_signal(n);

y1 = fdfir(x);
y2 = vfd(x,i0var+FD);

plot_signal_comparison(n,x,y1,y2,h,hvar,i0fixed,i0var,FD);
```



Which should be used: `dsp.VariableFractionalDelay` or `designFracDelayFIR` ?

This decision is largely based on filter requirements and the target platform.

- For a high bandwidth and accurate group delay response, use the `designFracDelayFIR` function. Keep in mind that this design process is more computationally intensive. Therefore, is it better suited to be deployed on a higher-end hardware, especially if realtime tuning of the delay value is desired. It is also suitable for lower-end hardware deployment, if the delay value is fixed, and the design can be done offline.
- For time-varying delay filters aimed at low-performance computational apparatus, use `dsp.VariableFractionalDelay` with the `'Farrow'` mode.

Design of Decimators and Interpolators

This example shows how to design filters for decimation and interpolation of discrete sequences.

The Role of Lowpass Filtering in Rate Conversion

Rate conversion is the process of changing the rate of a discrete signal to obtain a new discrete representation of the underlying continuous signal. The process involves uniform downsampling and upsampling. Uniform downsampling by a rate of *N* refers to taking every *N*-th sample of a sequence and discarding the rest of the samples. Uniform upsampling by a factor of *N* refers to the padding of *N*-1 zeros between every two consecutive samples.

```
x = 1:3
L = 3; % upsampling rate
M = 2; % downsampling rate

% Upsample and downsample
xUp = upsample(x,L)
xDown = downsample(x,M)
```

```
x =

     1     2     3


xUp =

     1     0     0     2     0     0     3     0     0


xDown =

     1     3
```

Both those basic operations introduce signal artifacts: downsampling introduces aliasing, and upsampling introduces imaging. To mitigate these effect, use lowpass filters.

- When downsampling by a rate of N, a lowpass filter applied *prior* to downsampling limits the input bandwidth, and thus eliminating spectrum aliasing. This is similar to an analog LPF used in A/D converters. Ideally, such an anti-aliasing filter has a unit gain and a cutoff frequency of $\omega_c = \frac{1}{N}\omega_N$, here ω_N is the Nyquist frequency of the signal. **Note:** the underlying sampling frequency is insignificant, we assume normalized frequencies (i.e. $\omega_N = 1$) throughout the discussion.
- When upsampling by a rate of N, a lowpass filter applied after upsampling is known as an anti-imaging filter. The filter removes the spectral images of the low-rate signal. Ideally, the cutoff frequency of this anti-imaging filter is $\omega_c = \frac{1}{N}$ (like its antialiasing counterpart), while its gain is N.

Both upsampling and downsampling operations of rate N require a lowpass filter with a normalized cutoff frequency of $\frac{1}{N}$. The only difference is in the required gain and the placement of the filter (before or after rate conversion).

The combination of upsampling a signal by a factor of L, followed by filtering, and then downsampling by a factor of M converts the sequence sample rate by a rational factor of $\frac{L}{M}$. This is obtained by upsampling by rate L followed by filtering, then downsampling by rate M. The order of rate conversion operation cannot be commuted. A single filter that combines anti-aliasing and anti-imaging is placed between the upsampling and the downsampling stages. This filter is a lowpass with the normalized cutoff frequency of $\omega_c = \min(1/L, 1/M)$ and a gain of L.

While any lowpass FIR design function (e.g. `fir1`, `firpm`, or `fdesign`) could design an appropriate anti-aliasing and anti-imaging filter, the function `designMultirateFIR` gives a convenient and a simplified interface. The next few sections show the use of these functions to design the filter and demonstrate why `designMultirateFIR` is the preferred way.

Filtered Rate Conversion: Decimators, Interpolators, and Rational Rate Converters

Filtered rate conversions includes decimators, interpolators, and rational rate converters, all of which are cascades of rate change blocks with filters in various configuations.

Filtered Rate Conversion using the `filter`, `upsample`, and `downsample` functions

Decimation refers to LTI filtering followed by uniform downsampling. An FIR decimator can be implemented as follows.

1. Design a an anti-aliasing lowpass filter *h*
2. Filter the input though *h*
3. Downsample the filtered sequence by a factor of *M*

```
% Define an input sequence
x = rand(60,1);

% Implement an FIR decimator
h = fir1(L*12*2,1/M); % an arbitrary filter
xDecim = downsample(filter(h,1,x), M);
```

Interpolation refers to a upsampling followed by filtering. The implementation is very similar to decimation.

```
xInterp = filter(h,1,upsample(x,L));
```

Lastly, rational rate conversion is comprised of an interpolator followed by a decimator (in that specific order).

```
xRC = downsample(filter(h,1,upsample(x,L) ), M);
```

Filtered Rate Conversion Using System Objects

For streaming data, the system objects `dsp.FIRInterpolator`, `dsp.FIRDecimator`, and `dsp.FIRRateConverter` encapsulate the rate change and filtering in a single object. For example, construction of an interpolator is done as follows.

```
firInterp = dsp.FIRInterpolator(L,h);
```

Then, feed a sequence to the newly created object by a step call.

```
xInterp = firInterp(x);
```

Design and use decimators and rate converters in a similar way.

```
firDecim = dsp.FIRDecimator(M,h); % Construct
xDecim = firDecim(x); % Decimate (step call)

firRC = dsp.FIRRateConverter(L,M,h); % Construct
xRC = firRC(x); % Convert rate (step call)
```

Using system objects is generally preferred, as they:

- Allow for a cleaner syntax.
- Keep a state, as filter initial coniditon for subsequent step calls.
- Most importantly, they utilize a very efficient polyphase algorithm.

To construct these object, you need the rate conversion factor, and the FIR coefficients. The following section describes how to generate appropriate FIR coefficients for a rate conversion filter.

Design a Rate Conversion Filter using `designMultirateFIR`

The function `designMultirateFIR(L,M)` automatically finds the apropriate scaling and cutoff frequency for a given rate conversion ratio L/M. Use the FIR coefficients returned by `designMultirateFIR` with `dsp.FIRDecimator` (if $L = 1$), `dsp.FIRInterpolator` (if $M = 1$), or `dsp.FIRRateConverter` (general case).

Let us design an interpolation filter:

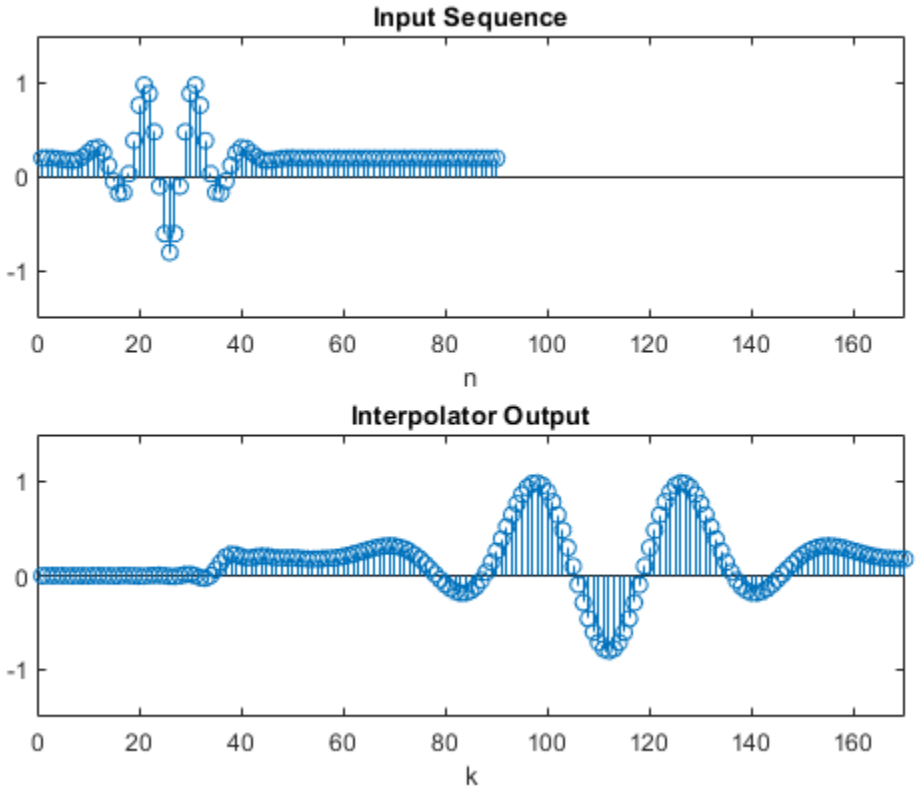
```
L = 3;
bInterp = designMultirateFIR(L,1); % Pure upsampling filter
firInterp = dsp.FIRInterpolator(L,bInterp);
```

Then, apply the interpolator to a sequence.

```
% Create a sequence
n = (0:89)';
f = @(t) cos(0.1*2*pi*t).*exp(-0.01*(t-25).^2)+0.2;
x = f(n);

% Apply interpolator
xUp = firInterp(x);
release(firInterp);
```

Let us first examine the raw output of the interpolator and compare with the original sequence.

```
plot_raw_sequences(x,xUp);
```

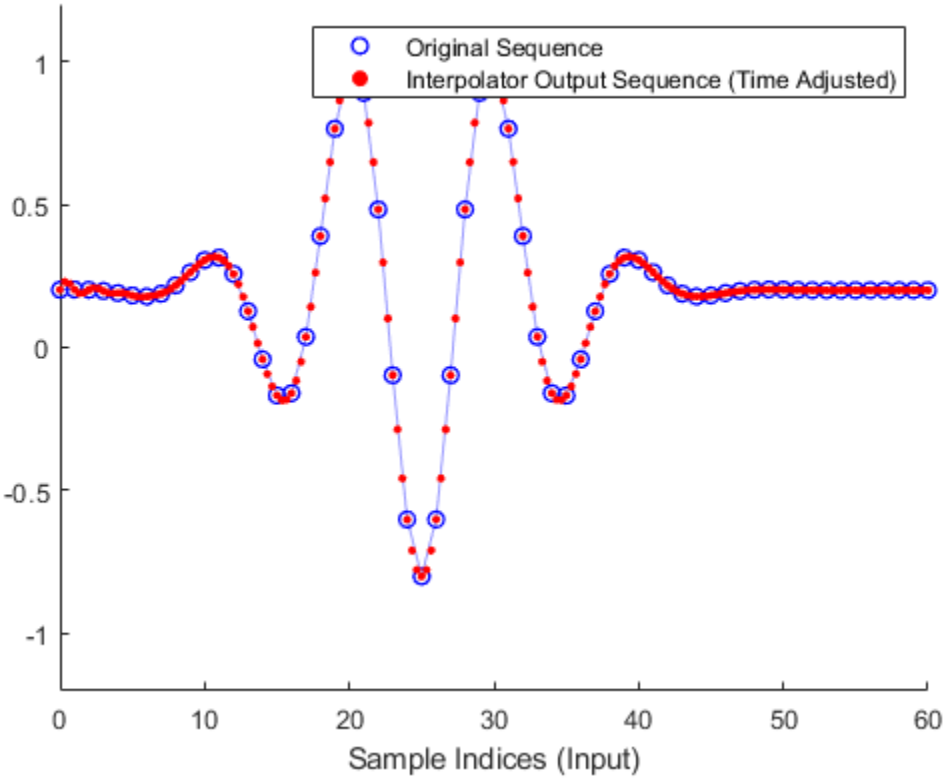


While there is some resemblence between the input `x` and the output `xUp`, there are several key differences. In the interpolated signal

- The time domain is stretched (as expected).
- The signal has a delay of half the length of the FIR `length(h)/2` (denoted i_0 henceforth).
- There is a transient response at the beginning.

To compare, align and scale the time domains of the two sequences. An interpolated sample `xUp[k]` corresponds to an input time $t[k] = \frac{1}{L}(k - i_0)$.

```
nUp = (0:length(xUp)-1);
i0 = length(bInterp)/2;
plot_scaled_sequences(n,x,(1/L)*(nUp-i0),xUp,["Original Sequence",...
            "Interpolator Output Sequence (Time Adjusted)"],[0,60]);
```

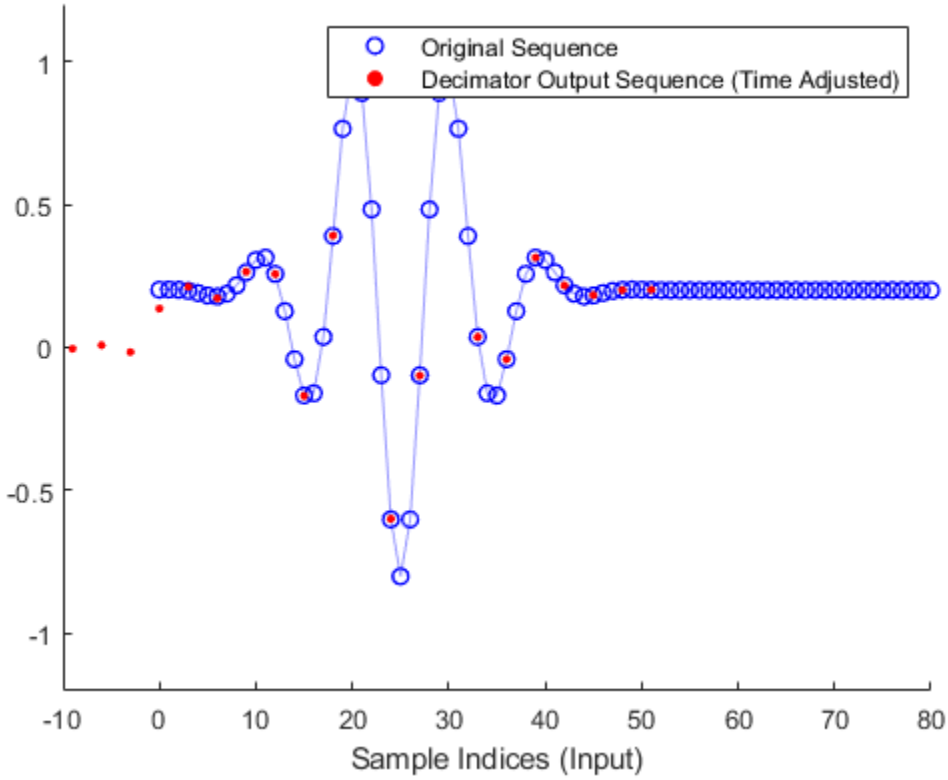


The same idea works for downsampling, where the time conversion is $t[k] = Mk - i_0$:

```
M = 3;
bDecim = designMultirateFIR(1,M); % Pure downsampling filter
firDecim = dsp.FIRDecimator(M,bDecim);
xDown = firDecim(x);
```

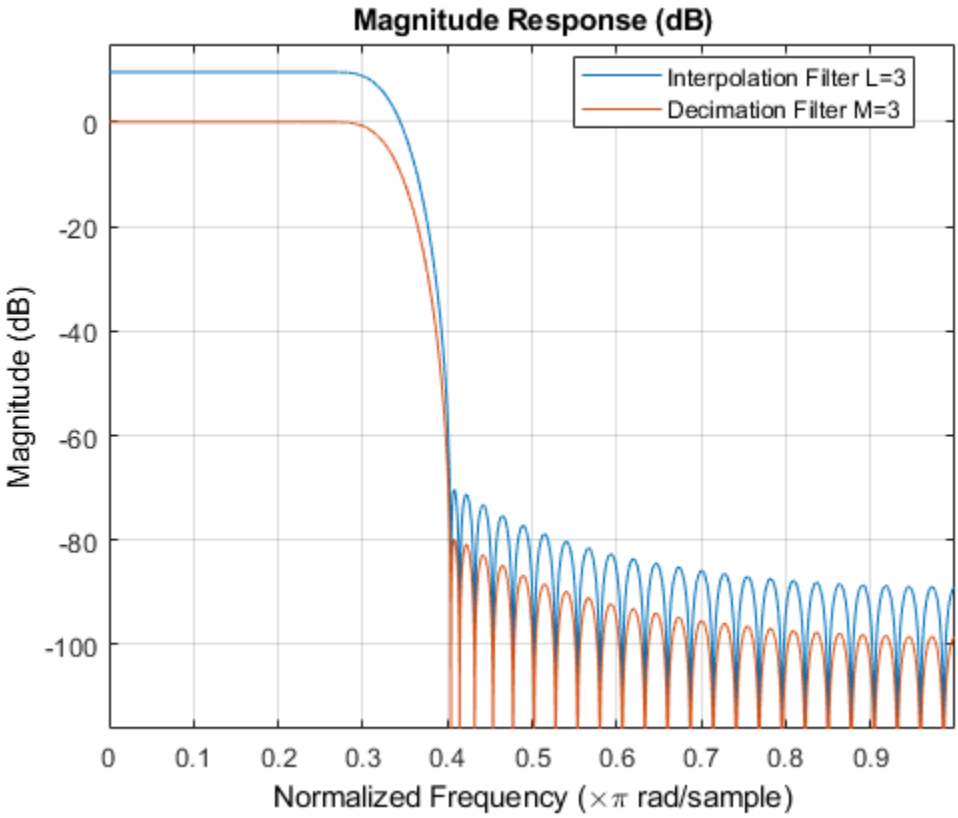
Plot them on the same scale and adjust for delay. Note they overlap perfectly.

```
i0 = length(bDecim)/2;
nDown = (0:length(xDown)-1);
plot_scaled_sequences(n,x,M*nDown-i0,xDown,["Original Sequence",...
            "Decimator Output Sequence (Time Adjusted)"],[-10,80]);
```



Visualize the magnitude responses of the upsampling and downampling filters using `fvtool`. The two FIR filters are identical in that case, up to a different gain.

```
hfv = fvtool(firInterp,firDecim); % Notice the gains in the passband
legend(hfv,"Interpolation Filter L="+num2str(L), ...
           "Decimation Filter M="+num2str(M));
```

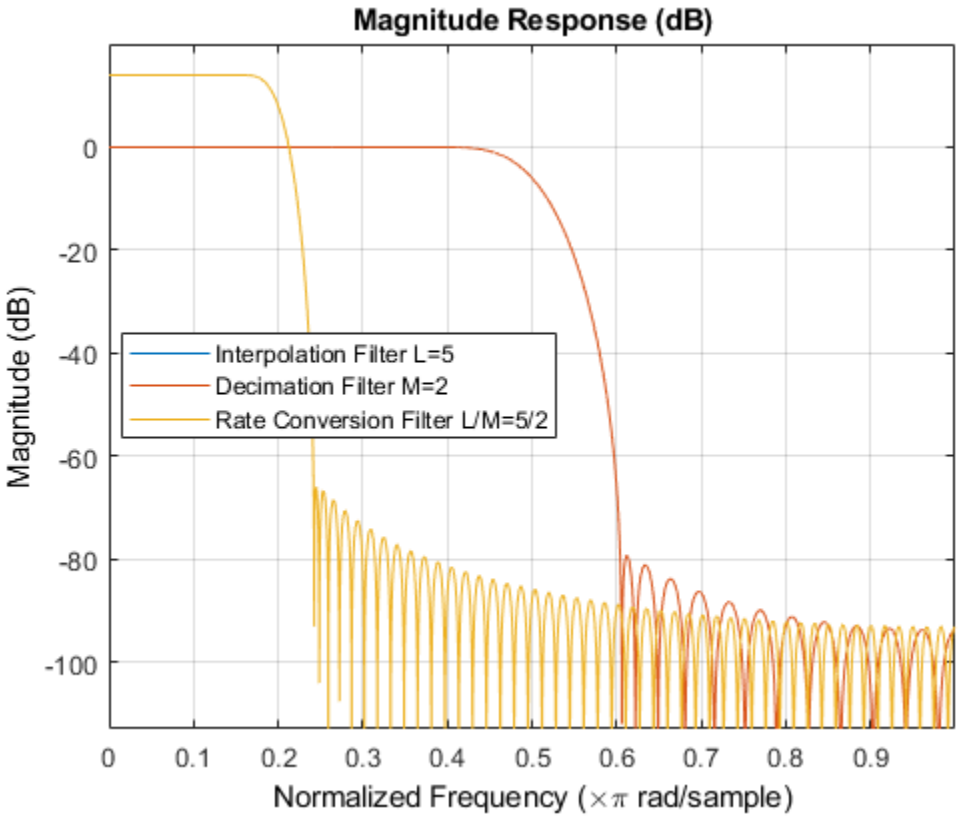


General rational conversions can be treated the same way as upsampling and downsampling. The cutoff is $\omega_c = \min(\frac{1}{L}, \frac{1}{M})$ and the gain is L. The function `designMultirateFIR` figures that out automatically.

```
L = 5;
M = 2;
b = designMultirateFIR(L,M);
firRC = dsp.FIRRateConverter(L,M,b);
```

Let us now compare the combined filter with the separate interpolation/decimation components.

```
firDecim = dsp.FIRDecimator(M,designMultirateFIR(1,M));
firInterp = dsp.FIRInterpolator(L,designMultirateFIR(L,1));

hfv = fvtool(firInterp,firDecim, firRC); % Notice the gains in the passband
legend(hfv,"Interpolation Filter L="+num2str(L),...
    "Decimation Filter M="+num2str(M), ...
    "Rate Conversion Filter L/M="+num2str(L)+"/"+num2str(M));
```

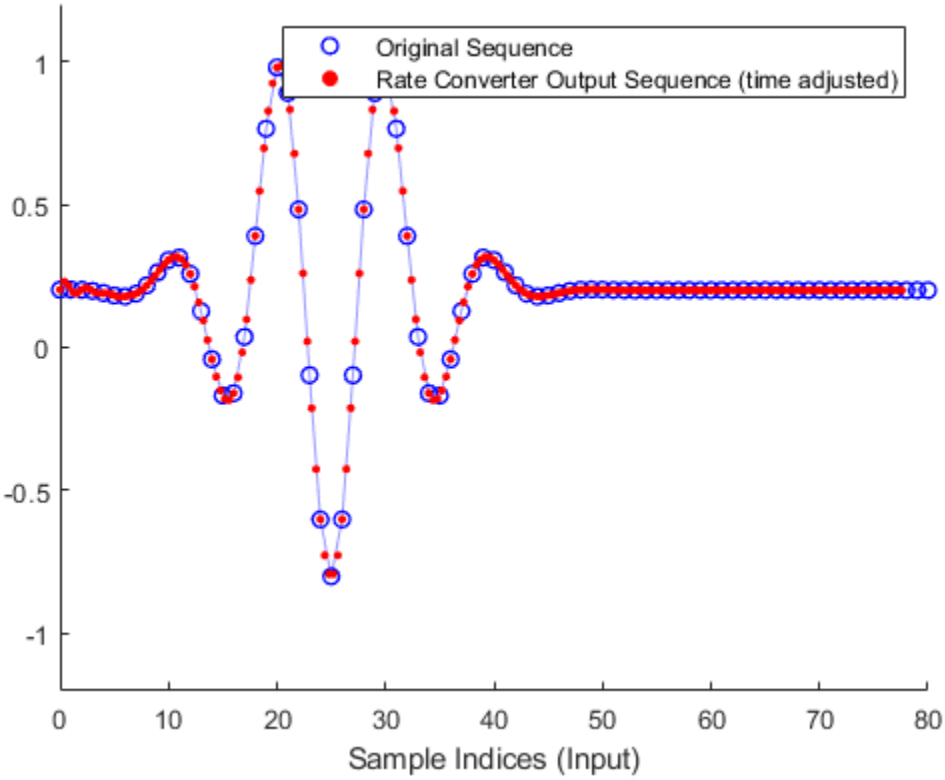


Once the `FIRRateConverter` is set up, perform the rate conversion by a step call.

```
xRC = firRC(x);
```

Plot the input and the filter output with time adjustment given by $t[k] = \frac{1}{L}(Mk - i_0)$.

```
nRC = (0:length(xRC)-1)';
i0 = length(b)/2;
plot_scaled_sequences(n,x,(1/L)*(M*nRC-i0),xRC,["Original Sequence",...
        "Rate Converter Output Sequence (time adjusted)"],[0,80]);
```



Adjusting the Lowpass FIR Design Parameters

Using `designMultirateFIR` you can also adjust the FIR length, transition width, and stopband attenuation.

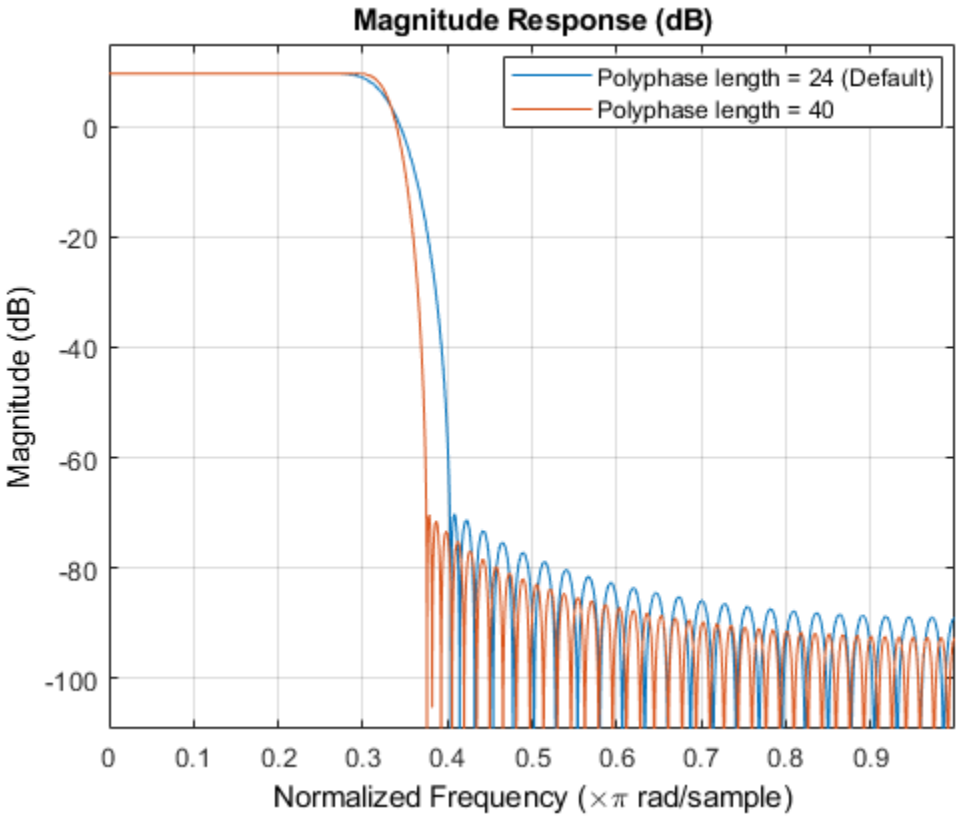
Adjusting the FIR Length

The FIR length can be controlled through *L*, *M*, and a third parameter *P* called half-polyphase length, whose default value is 12 (refer to “Output Arguments” for more details). Let us examine two design points.

```
% Unspecified half-length defaults to 12
b24 = designMultirateFIR(3,1);

halfPhaseLength = 20;
b40 = designMultirateFIR(3,1,halfPhaseLength);
```

Generally, larger half polyphase length yields steeper transitions.

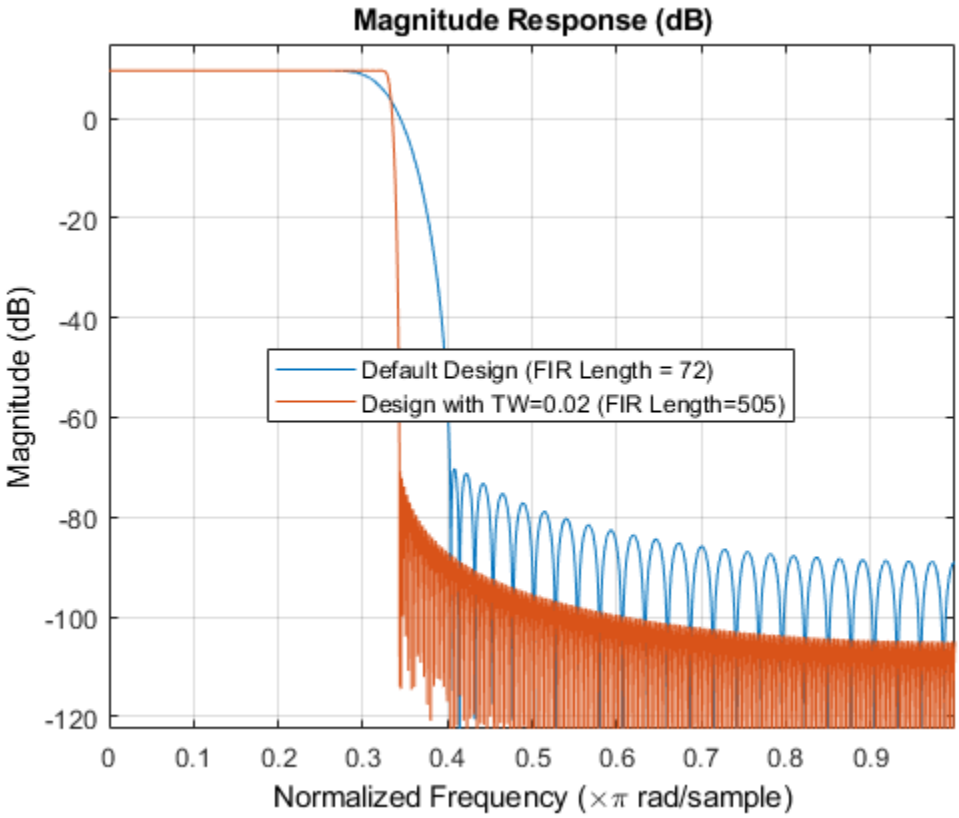
```
hfv = fvtool(b24,1,b40,1);
legend(hfv, 'Polyphase length = 24 (Default)','Polyphase length = 40');
```



Adjusting the Transition Width

Design the filter by specifying the desired transition width. The appropirate length will be derived automatically. Plot the resulting filter against the default design, and notice the difference in the transition width.

```
TW = 0.02;
bTW = designMultirateFIR(3,1,TW);

hfv = fvtool(b24,1,bTW,1);
legend(hfv, 'Default Design (FIR Length = 72)',"Design with TW="...
        +num2str(TW)+" (FIR Length="+num2str(length(bTW))+")");
```

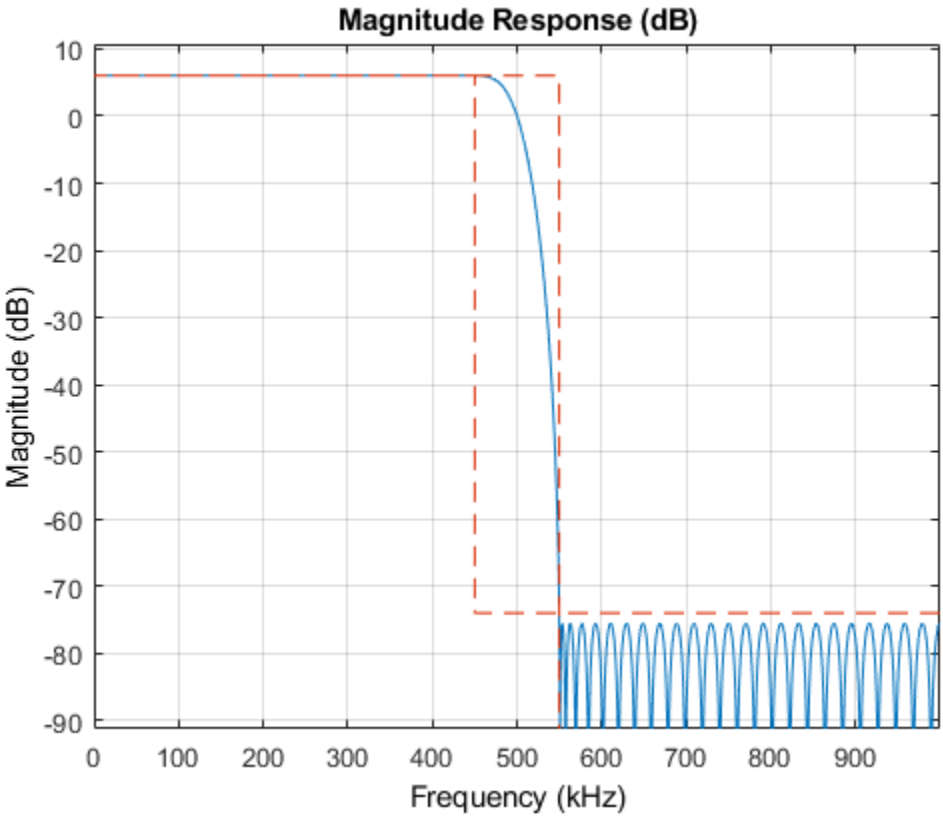


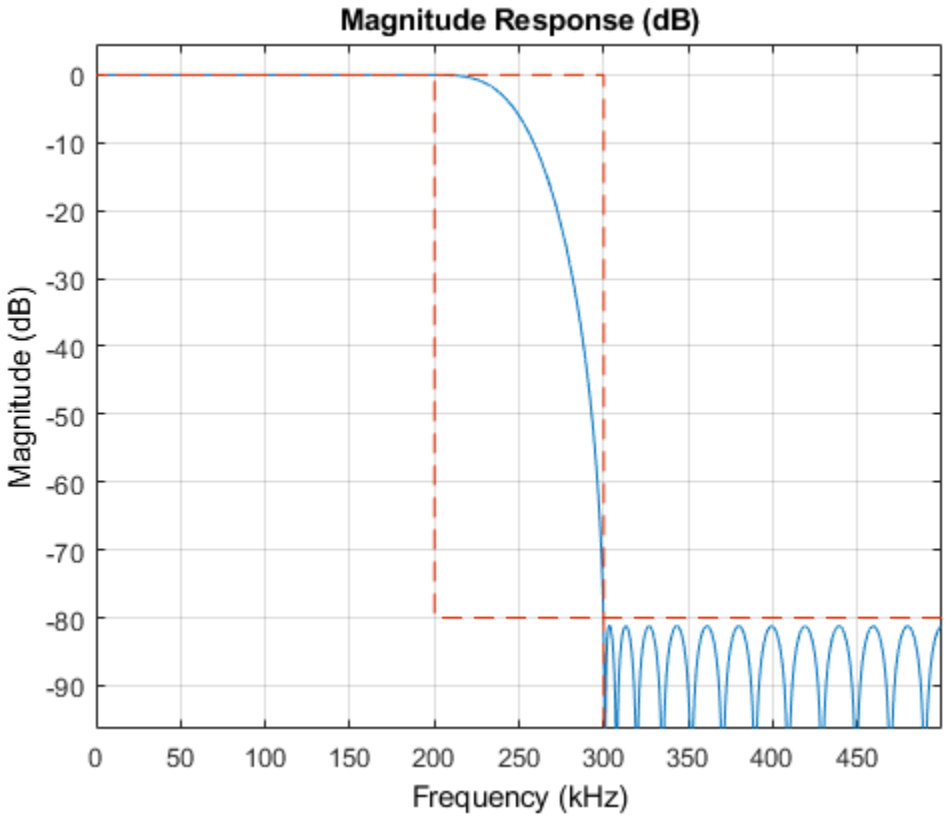
The Special Case of Rate Conversion by 2: Halfband Interpolators and Decimators

Using a half-band filter (i.e. $\omega_c = \frac{1}{2}$), you can perform sample rate conversion by a factor of 2. The `dsp.FIRHalfbandInterpolator` and `dsp.FIRHalfbandDecimator` objects perform interpolation and decimation by a factor of 2 using halfband filters. These system object are implemented using an efficient polyphase structure specific for that rate conversion. The IIR counterparts `dsp.IIRHalfbandInterpolator` and `dsp.IIRHalfbandDecimator` can be even more efficient. These system objects can also work with custom sample rates.

Visualize the magnitude response using `fvtool`. In the case of interpolation, the filter retains most of the spectrum from 0 to Fs/2 while attenuating spectral images. For decimation, the filter passes about half of the band, that is 0 to Fs/4, and attenuates the other half in order to minimize aliasing. The amount of attenuation can be set to any desired value for both interpolation and decimation. If unspecified, it defaults to 80 dB.

```
Fs = 1e6;
hbInterp = dsp.FIRHalfbandInterpolator('TransitionWidth',Fs/10,...
    'SampleRate',Fs);
fvtool(hbInterp) % Notice gain of 2 (6 dB) in the passband
hbDecim  = dsp.FIRHalfbandDecimator('TransitionWidth',Fs/10,...
    'SampleRate',Fs);
fvtool(hbDecim)
```


Magnitude Response (dB)
Magnitude (dB)
10
0
-10
-20
-30
-40
-50
-60
-70
-80
-90
0
100
200
300
400
500
600
700
800
900
Frequency (kHz)



Equiripple Design

The function `designMultirateFIR` utilizes window-based design of FIR lowpass. Other lowpass design methods can be applied as well, such as equiripple. For more control over the design process, use the `fdesign` filter design functions. The following example designs a decimator using the `fdesign.decimator` function.

```
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
fdDecim = fdesign.decimator(M,'lowpass',Fp,Fst,Ap,Ast,Fs) %#ok


fdDecim =

  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
```

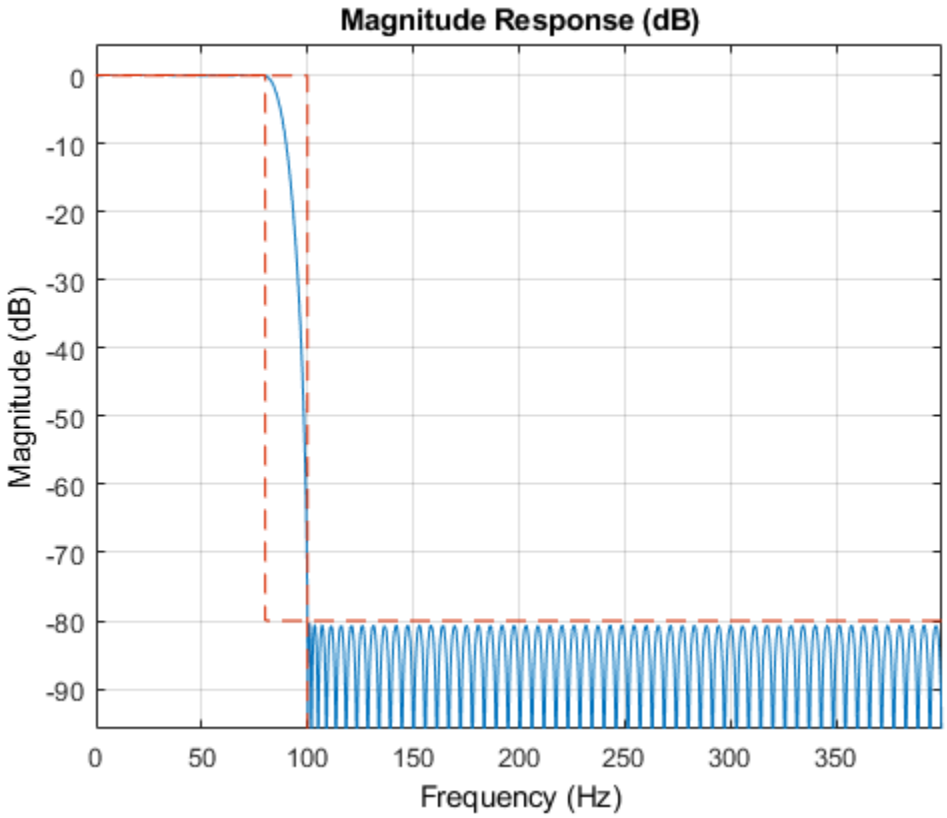
The specifications for the filter determine that a transition band of 20 Hz is acceptable between 80 and 100 Hz and that the minimum attenuation for out of band components is 80 dB. Also that the maximum distortion for the components of interest is 0.05 dB (half the peak-to-peak passband ripple). An equiripple filter that meets these specs can be easily obtained by the `fdesign` interface.

```
eqrDecim = design(fdDecim,'equiripple', 'SystemObject', true);
measure(eqrDecim)
```

```
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
```

Visualize the magnitude response confirms that the filter is an equiripple filter.

```
fvtool(eqrDecim)
```



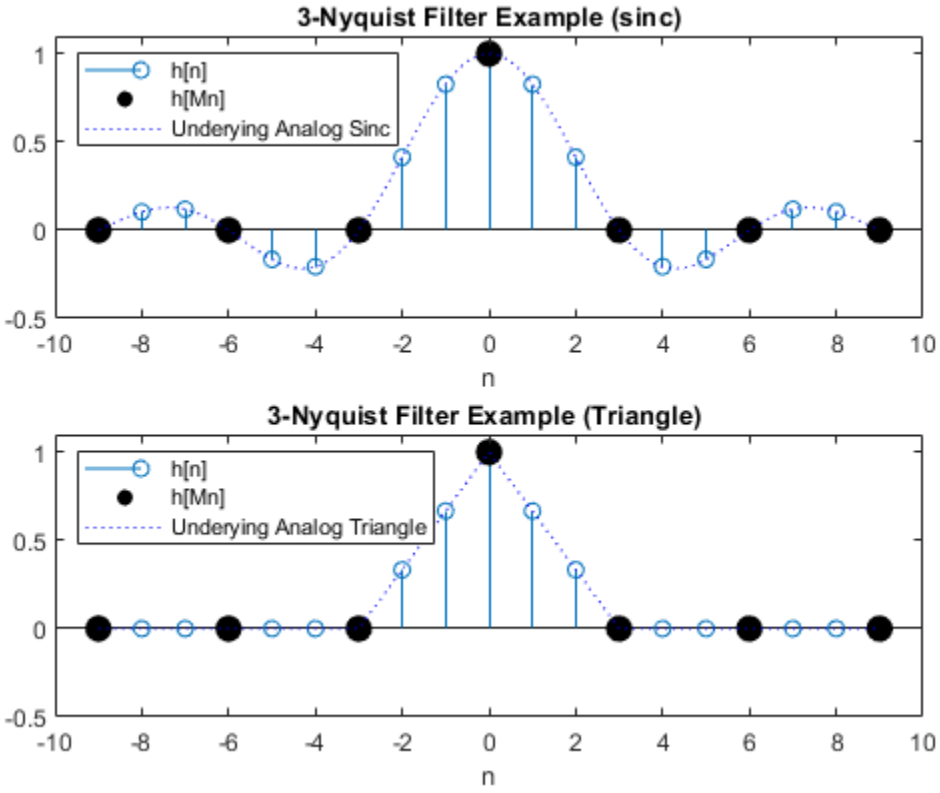
Nyquist Filters and Interpolation Consistency

A digital convolution filter h is called an L-th Nyquist filter if it is vanishing periodically every L samples, except the center index. In other words, the sampling h by a factor of L yields an impulse:

$$h[kL] = \delta_k$$

The L-th band ideal lowpass, $h[m] = sinc(\frac{m}{L})$, for example, is L-th Nyquist filter. Another example is a triangular window.

```
L=3;
t = linspace(-3*L,3*L,1024);
n = (-3*L:3*L);
hLP = @(t) sinc(t/L);
hTri = @(t) (1-abs(t/L)).*(abs(t/L)<=1);

plot_nyquist_filter(t,n,hLP,hTri,L);
```



The function `designMultirateFIR` yields Nyquist filters, since it is based on weighted and truncated versions of ideal Nyquist filters.

Nyquist filters are efficient to implement since an L-th fraction of the coefficients in these filters are zero, which reduces the number of required multiplications. This feature makes these filters efficient for both decimation and interpolation.

Interpolation Consistency

Nyquist filters retain the sample values of the input even after filtering. This behavior, which is called *interpolation consistency*, is not true in general, as will be shown below.

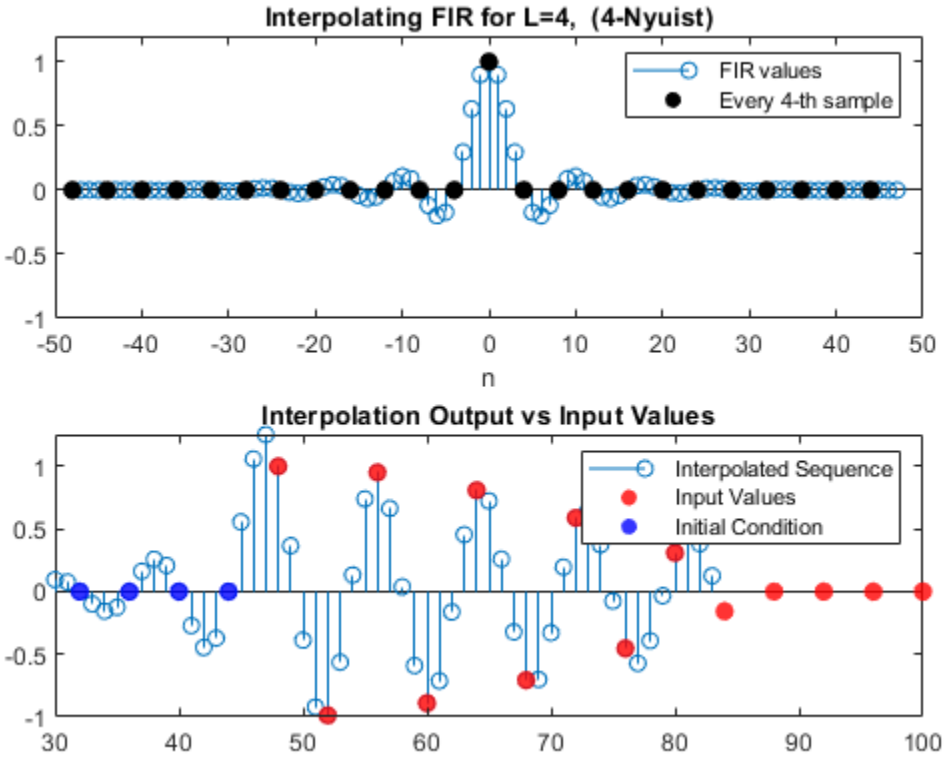
Interpolation consistency holds in Nyquist filter, since the coefficients equal to zero every L samples (except for at the center). The proof is straightforward. Assume that x_L is the upsampled version of x (with zeros inserted between samples) so that $x_L[Ln] = x[n]$, and that $y = h * x_L$ is the interpolated signal. Sample y uniformly and get the following equation.

$$y[nL] = (h * x_L)[Ln] = \sum_k h[Ln-k]x_L[k] = \sum_m \underbrace{h[Ln-Lm]}_{\delta[L(n-m)]} x_L[Lm] = x_L[Ln] = x[n]$$

Let us examine the effect of using a Nyquist filter for interpolation. The `designMultirateFIR` function produces Nyquist filters. As you can see in the depiction below, the input values coincide with the interpolated values.

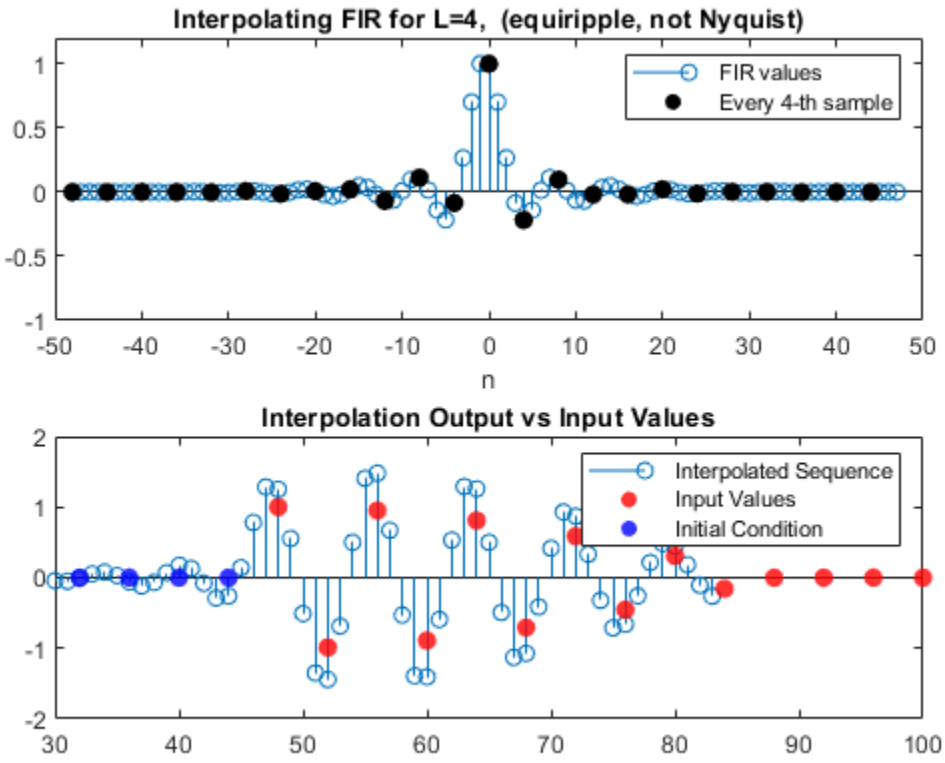
```
% Generate input
n = (0:20)';
```

```
xInput = (n<=10).*cos(pi*0.05*n).*(-1).^n;

L = 4;
hNyq = designMultirateFIR(L,1);
firNyq = dsp.FIRInterpolator(L,hNyq);
xIntrNyq = firNyq(xInput);
release(firNyq);
plot_shape_and_response(hNyq,xIntrNyq,xInput,L,num2str(L)+"-Nyuist");
```



This is not the case for other lowpass filters such as equiripple designs, as seen in the figure below. Note that the interpolated sequence does not coincide with the low-rate input values. On the other hand, distortion may be lower in non-Nyquist filters, as a tradeoff for interpolation consistency.

```
hNotNyq = firpm(length(hNyq)-1,[0 1/L 1.5/L 1],[1 1 0 0]);
hNotNyq = hNotNyq/max(hNotNyq); % Adjust gain
firIntrNotNyq = dsp.FIRInterpolator(L,hNotNyq);
xIntrNotNyq= firIntrNotNyq(xInput);
release(firIntrNotNyq);

plot_shape_and_response(hNotNyq,xIntrNotNyq,xInput,L,"equiripple, not Nyquist");
```



See Also

Functions
`designMultirateFIR` | `downsample` | `upsample` | `filter`

Objects
`dsp.FIRDecimator` | `dsp.FIRInterpolator` | `dsp.FIRRateConverter` | `dsp.FIRHalfbandDecimator` | `dsp.FIRHalfbandInterpolator`

Related Examples

- “FIR Nyquist (L-th band) Filter Design” on page 4-105
- “Multistage Rate Conversion” on page 4-177

Multistage Halfband IIR Filter Design

This example shows how to design multistage halfband IIR decimators.

Similar to FIR multirate filters, IIR halfband decimators/interpolators can be implemented using efficient polyphase structures. IIR polyphase filters present several interesting properties: they require a very small number of multipliers to implement, they are inherently stable, have low roundoff noise sensitivity and no limit cycles.

Butterworth and elliptic IIR filters can be designed with a halfband decimator/interpolator response type. Furthermore, it is possible to achieve almost linear phase response using specialized IIR design algorithms.

Cost Efficiency Case Study

A way of measuring a filter's computational cost is to determine how many multiplications need to be computed (on average) per input sample (MPIS). Consider a MPIS count case study: FIR vs IIR for the following filter specifications:

```
Fs  = 9.6e3;   % Sampling frequency: 9.6 kHz
TW  = 120;     % Transition width
Ast = 80;      % Minimum stopband attenuation: 80 dB
M   = 8;       % Decimation factor
NyquistDecimDesign = fdesign.decimator(M,'Nyquist',M,TW,Ast,Fs);
```

Multistage Halfband FIR Design

A way of obtaining efficient FIR designs is through the use of multirate multistage techniques. This design results in three FIR halfband decimators in cascade. Halfband filters are extremely efficient because every other coefficient is zero.

```
MultistageFIRDecim = design(NyquistDecimDesign,'multistage',...
           'HalfbandDesignMethod','equiripple','SystemObject',true);
cost(MultistageFIRDecim)
```

```
ans =

  struct with fields:

                 NumCoefficients: 69
                       NumStates: 126
    MultiplicationsPerInputSample: 12.8750
          AdditionsPerInputSample: 12
```

This method achieves computational costs of 12.875 MPIS on average.

Multistage Halfband IIR Design

Elliptic filters are the IIR equivalent of optimal equiripple filters. This design results in three IIR halfband decimators in cascade. Elliptic designs produce the most efficient IIR halfband designs.

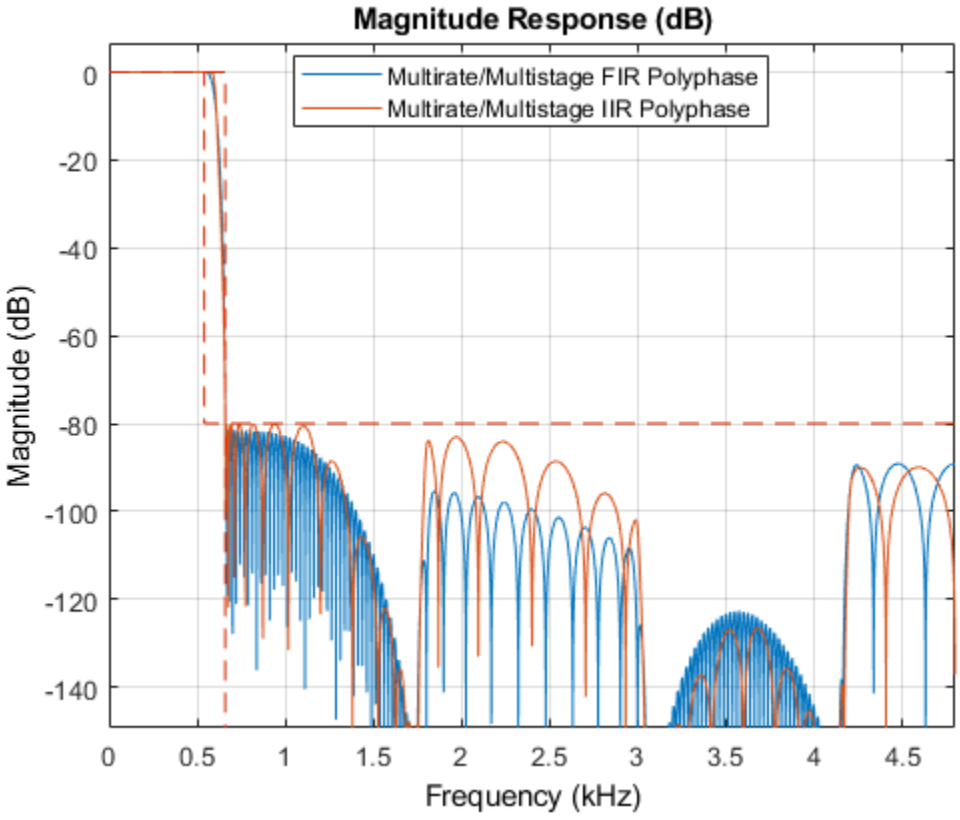
```
MultistageIIRDecim = design(NyquistDecimDesign,'multistage',...
    'HalfbandDesignMethod','ellip','SystemObject',true);
cost(MultistageIIRDecim)
```

```
ans =

  struct with fields:

                NumCoefficients: 11
                     NumStates: 17
    MultiplicationsPerInputSample: 2.5000
          AdditionsPerInputSample: 5
```

This method achieves computational costs of only 2.5 MPIS on average.

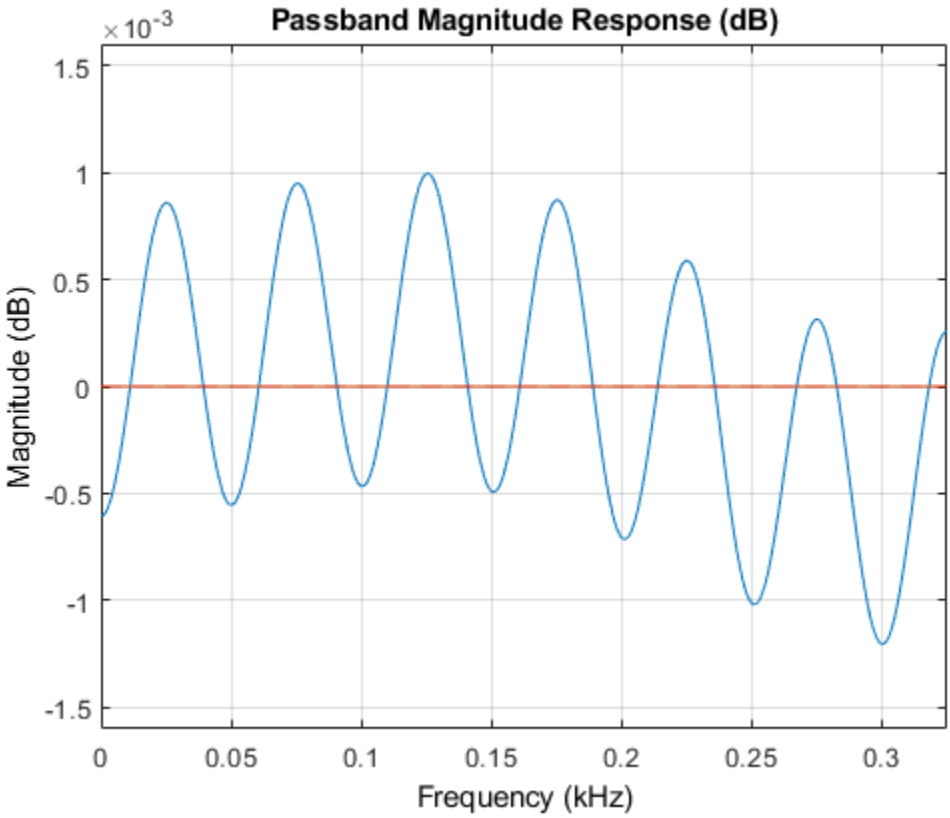
If we overlay the magnitude responses of the FIR and IIR multirate multistages filters, the two filters look very similar and both meet the specifications.

```
fvFig = fvtool(MultistageFIRDecim,MultistageIIRDecim,'Color','white');
legend(fvFig, 'Multirate/Multistage FIR Polyphase', ...
    'Multirate/Multistage IIR Polyphase')
```



Close inspection actually shows the passband ripples of the IIR filter to be far superior to that of the FIR filter. So computational cost savings don't come at the price of a degraded magnitude response.

```
fvtool(MultistageFIRDecim,MultistageIIRDecim,'Color','white');
title('Passband Magnitude Response (dB)')
axis([0 0.325 -0.0016 0.0016])
```



Quasi-Linear Phase Halfband IIR Designs

By modifying the structure used to implement each IIR halfband filter, it is possible to achieve almost linear phase designs using IIR filters. This design also results in three halfband decimators in cascade. However, each halfband is implemented in a specific way that includes a pure delay connected in parallel with an allpass filter. This constraint on the implementation helps provide the quasi linear phase response. This comes at the expense of a slight increase in computational cost compared to elliptic designs.

```
IIRLinearPhaseFilt = design(NyquistDecimDesign,'multistage',...
    'HalfbandDesignMethod','iirlinphase','SystemObject',true);
cost(IIRLinearPhaseFilt)
```

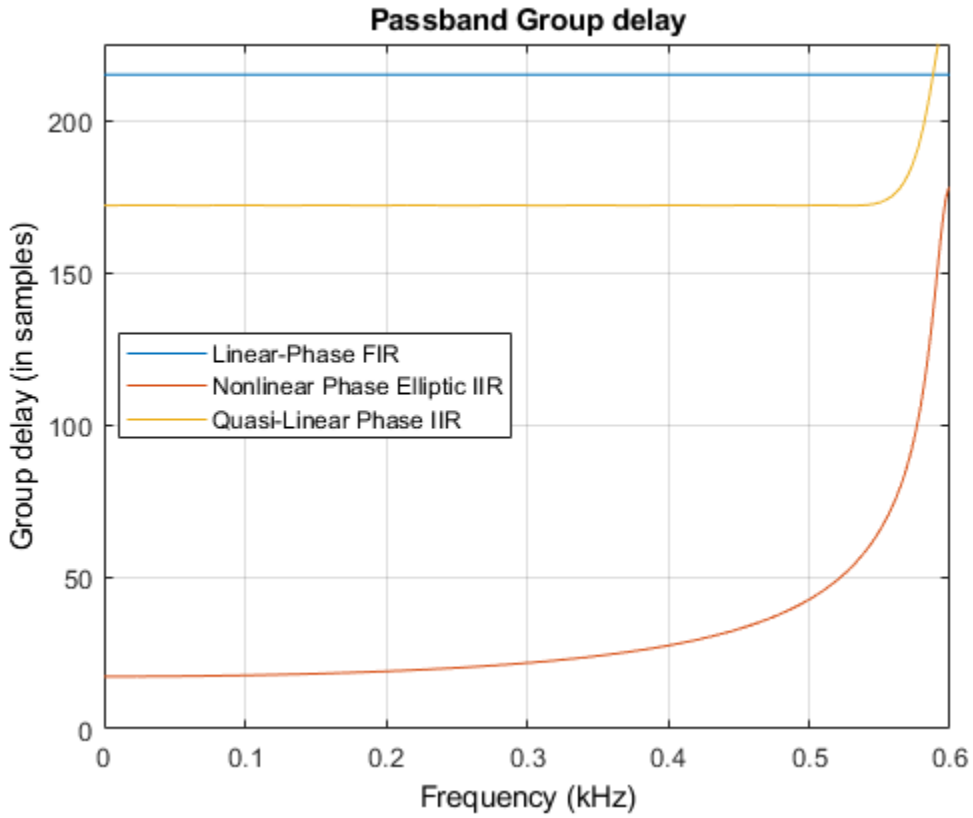
```
ans =

  struct with fields:

                NumCoefficients: 25
                      NumStates: 55
  MultiplicationsPerInputSample: 4.3750
        AdditionsPerInputSample: 8.7500
```

Although not as efficient as the elliptic case, the design is nevertheless more efficient than using FIR halfbands.

Group Delay Comparison

Overlaying the group delay of the three designs, and focusing on the passband of the filter (the area of interest), we can verify that the latter IIR design achieves quasi-linear phase (almost flat group delay) in that area. In contrast, the elliptic filter, while more efficient (and with a lower group delay overall), has a clearly nonlinear phase response.

```
fvFig = fvtool(MultistageFIRDecim,MultistageIIRDecim,IIRLinearPhaseFilt,...
    'Color','white','Analysis','grpdelay');
axis([0 0.6 0 225])
title('Passband Group delay')
legend(fvFig, 'Linear-Phase FIR', 'Nonlinear Phase Elliptic IIR',...
    'Quasi-Linear Phase IIR')
```



Fixed-Point Robustness

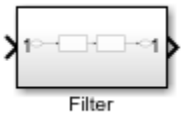
Polyphase IIR filters can be implemented in different ways. We have already encountered single-rate and multirate cascade allpass in previous sections. Now take a Hilbert transformer for example. A quasi linear-phase IIR Hilbert filter with a transition width of 96Hz and a maximum passband ripple of 0.1 dB can be implemented as a cascade wave digital filter using only 10 MPIS compared to 133 MPIS for an FIR equivalent:

```
HilbertDesign = fdesign.hilbert('TW,Ap',96,.1,Fs);
HilbertIIRFilt = design(HilbertDesign,'iirlinphase',...
                 'FilterStructure','cascadewdfallpass',...
                 'SystemObject',true);
cost(HilbertIIRFilt)
```

```
ans =

  struct with fields:

                NumCoefficients: 10
                     NumStates: 33
    MultiplicationsPerInputSample: 10
          AdditionsPerInputSample: 25
```

Wave digital filters have been proven to be very robust even when poles are close to the unit circle. They are inherently stable, have low roundoff noise properties and are free of limit cycles. To convert our IIR Hilbert filter to a fixed-point representation, we can use the realizemdl command and the Fixed-Point Tool to do the floating-point to fixed-point conversion of the Simulink model:

```
realizemdl(HilbertIIRFilt)
```



Summary

IIR filters have traditionally been considered much more efficient than their FIR counterparts in the sense that they require a much smaller number of coefficients in order to meet a given set of specifications.

Modern FIR filter design tools utilizing multirate/polyphase techniques have bridged the gap while providing linear-phase response along with good sensitivity to quantization effects and the absence of stability and limit cycles problems when implemented in fixed-point.

However, IIR polyphase filters enjoy most of the advantages that FIR filters have and require a very small number of multipliers to implement.

Efficient Sample Rate Conversion Between Arbitrary Factors

This example shows how to efficiently convert sample rates between arbitrary factors.

The need for sample rate conversion by an arbitrary factor arises in many applications (e.g. symbol synchronization in digital receivers, speech coding and synthesis, computer simulation of continuous-time systems, etc.). In this example, we will examine an example where cascades of polynomial-based and polyphase filters form an efficient solution when it is desired to convert the sampling rate of a signal from 8 kHz to 44.1 kHz.

Single Stage Polyphase Approach

Polyphase structures are generally considered efficient implementations of multirate filters. However in the case of fractional sample rate conversion, the number of phases, and therefore the filter order, can quickly become excessively high. To resample a signal from 8 kHz to 44.1 kHz, we interpolate by 441 and decimate by 80 (8*441/80=44.1).

```
sampRateConv = dsp.SampleRateConverter('Bandwidth',6e3, ...
    'InputSampleRate',8e3,'OutputSampleRate',44.1e3, ...
    'StopbandAttenuation',50);
```

This can be done in relatively efficient manner in two stages:

```
info(sampRateConv)
cost(sampRateConv)
```

```
ans =

    'Overall Interpolation Factor    : 441
     Overall Decimation Factor       : 80
     Number of Filters               : 2
     Multiplications per Input Sample: 95.175000
     Number of Coefficients          : 1774
     Filters:
        Filter 1:
        dsp.FIRRateConverter - Interpolation Factor: 147
                             - Decimation Factor   : 80
        Filter 2:
        dsp.FIRInterpolator  - Interpolation Factor: 3
     '


ans =

  struct with fields:

                 NumCoefficients: 1774
                       NumStates: 30
    MultiplicationsPerInputSample: 95.1750
          AdditionsPerInputSample: 89.6750
```

Although the number of operations per input sample is reasonable (roughly 95 multiplications - keeping in mind that the rate increases after the first stage to 14.7 kHz), 1774 coefficients would have to be stored in memory in this case.

Providing an Output Rate Tolerance

One way to mitigate the large number of coefficients could be to allow for a tolerance in the output sample rate if the exact rate is not critical. For example, specifying a tolerance of 1% results in an output rate of 44 kHz rather then 44.1 kHz. This now requires to interpolate by 11 and decimate by 2. It can be done efficiently with a single stage.

```
sampRateConvWithTol = dsp.SampleRateConverter('Bandwidth',6e3, ...
    'InputSampleRate',8e3,'OutputSampleRate',44.1e3, ...
    'StopbandAttenuation',50,'OutputRateTolerance',0.01);
cost(sampRateConvWithTol)
```

```
ans =

  struct with fields:

                 NumCoefficients: 120
                       NumStates: 12
    MultiplicationsPerInputSample: 60
          AdditionsPerInputSample: 55
```

In this case, 120 coefficients are needed and the number of multiplications per input sample is 60.

Single Stage Farrow Approach

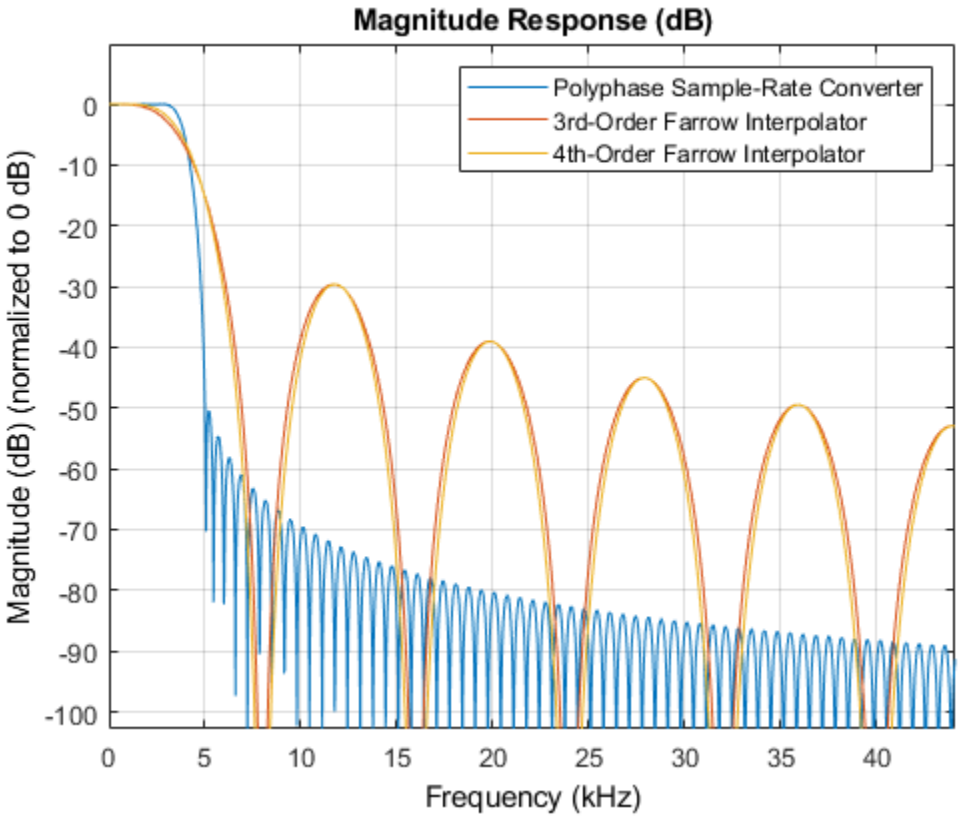
Polynomial-based filters are another way to overcome the problem of needing a large number of coefficients to be stored. Farrow structures are efficient implementations for such filters.

```
farrowSampRateConv_3rd = dsp.FarrowRateConverter('InputSampleRate',8e3, ...
    'OutputSampleRate',44.1e3,'PolynomialOrder',3);

farrowSampRateConv_4th = dsp.FarrowRateConverter('InputSampleRate',8e3, ...
    'OutputSampleRate',44.1e3,'PolynomialOrder',4);

cost(farrowSampRateConv_3rd)
cost(farrowSampRateConv_4th)
```

```
ans =

  struct with fields:

                 NumCoefficients: 16
                       NumStates: 3
    MultiplicationsPerInputSample: 66.1500
          AdditionsPerInputSample: 60.6375


ans =

  struct with fields:

                 NumCoefficients: 25
                       NumStates: 4
    MultiplicationsPerInputSample: 121.2750
          AdditionsPerInputSample: 99.2250
```

With 3rd-order polynomials, 16 coefficients are needed and about 66 multiplications per input sample. Fourth-order polynomials provide slightly better lowpass response at a higher cost: 25 coefficients and 121 multiplications per input sample.

```
filts = getFilters(sampRateConv);
W = linspace(0,44.1e3,2048);  % Define the frequency range analysis
Fs1 = 8e3*147;  % The equivalent single stage filter is clocked at 3.53 MHz
hfvt = fvtool(filts.Stage1,farrowSampRateConv_3rd, ...
    farrowSampRateConv_4th,'FrequencyRange','Specify freq. vector', ...
    'FrequencyVector',W,'Fs',[Fs1 3*Fs1 3*Fs1], ...
    'NormalizeMagnitudeto1','on','Color','white');
legend(hfvt,'Polyphase Sample-Rate Converter', ...
    '3rd-Order Farrow Interpolator','4th-Order Farrow Interpolator', ...
    'Location','NorthEast')
```



Providing an output rate tolerance does not significantly impact the implementation cost of the Farrow filter. However, it does change the interpolation and decimation factors in the same way it does for dsp.SampleRateConverter.

```
farrowSampRateConv_4th = dsp.FarrowRateConverter('InputSampleRate',8e3, ...
    'OutputSampleRate',44.1e3,'PolynomialOrder',4, ...
    'OutputRateTolerance',0.01);
info(farrowSampRateConv_4th)
cost(farrowSampRateConv_4th)
```

```
ans =
```

```
  12x52 char array

    'Discrete-Time FIR Multirate Filter (real)           '
    '----------------------------------------           '
    'Filter Structure      : Farrow Sample-Rate Converter'
    'Interpolation Factor  : 11                          '
    'Decimation Factor     : 2                           '
    'Filter Length         : 5                           '
    'Stable                : Yes                         '
    'Linear Phase          : No                          '
    '                                                    '
    'Arithmetic            : double                      '
    'Output Rate Tolerance : 1.000000 %                  '
    'Adjusted Output Rate  : 44000.000000                '


ans =

  struct with fields:

                 NumCoefficients: 25
                       NumStates: 4
    MultiplicationsPerInputSample: 121
          AdditionsPerInputSample: 99
```

Cascade of Farrow and FIR Polyphase Structures

We now try to design a hybrid solution that would take advantage of the two types of filters that we have previously seen. Polyphase filters are particularly well adapted for interpolation or decimation by an integer factor and for fractional rate conversions when the interpolation and the decimation factors are low. Farrow filters can efficiently implement arbitrary (including irrational) rate change factors. First, we interpolate the original 8 kHz signal by 4 using a cascade of FIR halfband filters.

```
intSampRateConv = dsp.SampleRateConverter('Bandwidth',6e3, ...
    'InputSampleRate',8e3,'OutputSampleRate',32e3, ...
    'StopbandAttenuation',50);
info(intSampRateConv)
```

```
ans =

    'Overall Interpolation Factor    : 4
     Overall Decimation Factor       : 1
     Number of Filters               : 1
     Multiplications per Input Sample: 34.000000
     Number of Coefficients          : 34
     Filters:
        Filter 1:
        dsp.FIRInterpolator  - Interpolation Factor: 4
     '
```

Then, we interpolate the intermediate 32 kHz signal by 44.1/32 = 1.378125 to get the desired 44.1 kHz final sampling frequency. We use a cubic Lagrange polynomial-based filter for this purpose.

```
farrowSampRateConv = dsp.FarrowRateConverter('InputSampleRate',32e3, ...
    'OutputSampleRate',44.1e3,'PolynomialOrder',3);
```

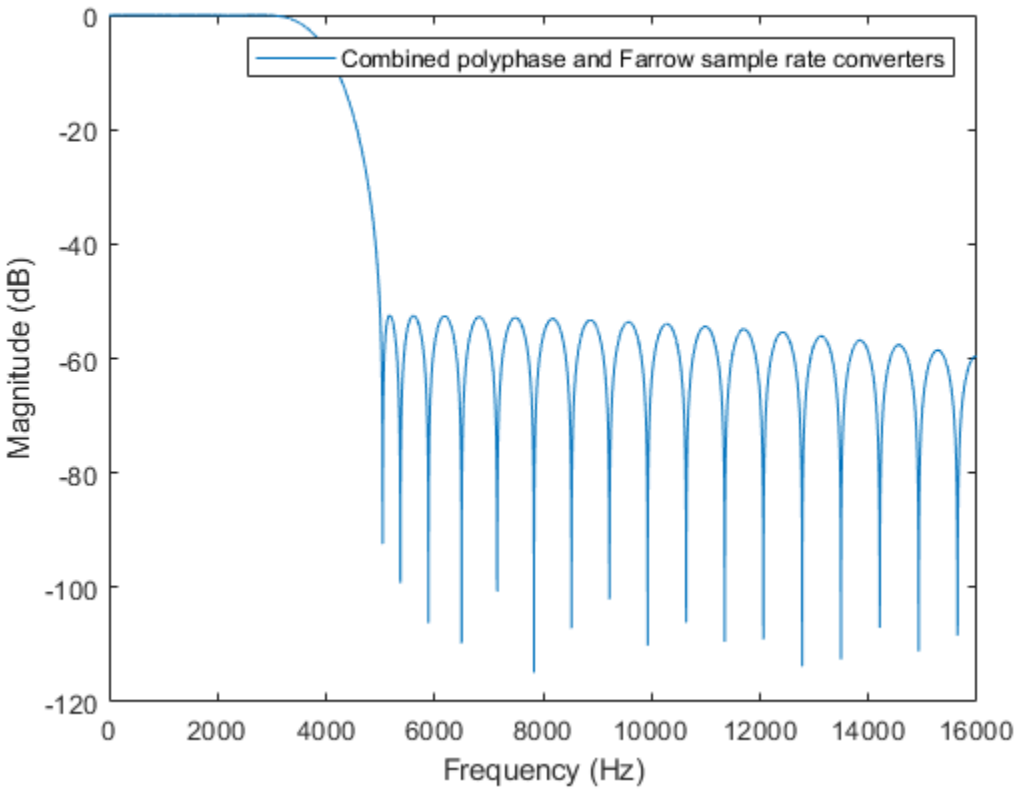
The overall filter is simply obtained by cascading the two filters.

```
cost(intSampRateConv)
cost(farrowSampRateConv)
```

```
ans =

  struct with fields:

                 NumCoefficients: 34
                       NumStates: 11
    MultiplicationsPerInputSample: 34
          AdditionsPerInputSample: 31


ans =

  struct with fields:

                 NumCoefficients: 16
                       NumStates: 3
    MultiplicationsPerInputSample: 16.5375
          AdditionsPerInputSample: 15.1594
```

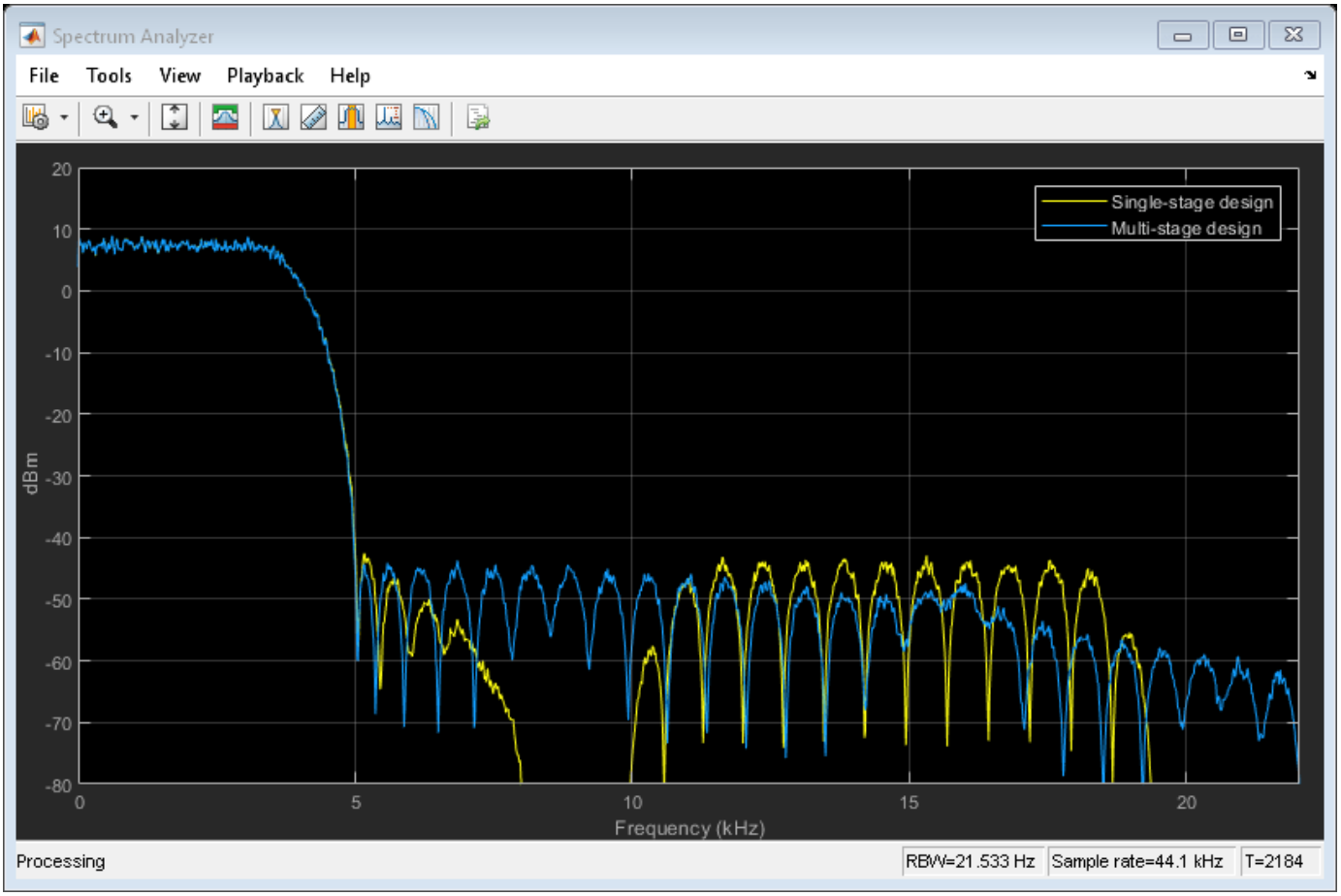
The number of coefficients of this hybrid design is relatively low (36) and the number of multiplications per input sample is also relatively low: 28 + 16*4 = 92. The combined frequency response of these two designs is superior to that of farrowSampRateConv_3rd or farrowSampRateConv_4th.

```
[Hsrc,f]     = freqz(intSampRateConv);
Fsfar        = 32e3*441;
Hfsrc        = freqz(farrowSampRateConv,f,Fsfar);
Hhybrid      = Hsrc.*Hfsrc;
Hhybrid_norm = Hhybrid/norm(Hhybrid,inf); % Normalize magnitude to 0 dB
plot(f,20*log10(abs(Hhybrid_norm)));
xlabel('Frequency (Hz)')
ylabel('Magnitude (dB)')
legend('Combined polyphase and Farrow sample rate converters', ...
       'Location','NorthEast')
```



We now overlay the frequency responses of the single-stage and the multistage designs. Clearly the responses are very comparable.

```
scope = dsp.SpectrumAnalyzer('SpectralAverages',50, ...
    'SampleRate',44.1e3,'PlotAsTwoSidedSpectrum',false, ...
    'YLimits',[-80 20],'ShowLegend',true, ...
    'ChannelNames',{'Single-stage design','Multi-stage design'});
tic,
while toc < 20
    % Run for 20 seconds
    x = randn(8000,1);

    % Convert rate using multistage FIR filters
    y1 = sampRateConv(x);

    % Convert rate using cascade of multistage FIR and Farrow filter
    ytemp  = intSampRateConv(x);
    y2     = farrowSampRateConv(ytemp);

    % Compare the output from both approaches
    scope([y1,y2])
end
```


Spectrum Analyzer
File
Tools
View
Playback
Help
Single-stage design
Multi-stage design
dBm
20
10
0
-10
-20
-30
-40
-50
-60
-70
-80
0
5
10
15
20
Frequency (kHz)
Processing
RBW=21.533 Hz
Sample rate=44.1 kHz
T=2184

Reconstruction Through Two-Channel Filter Banks

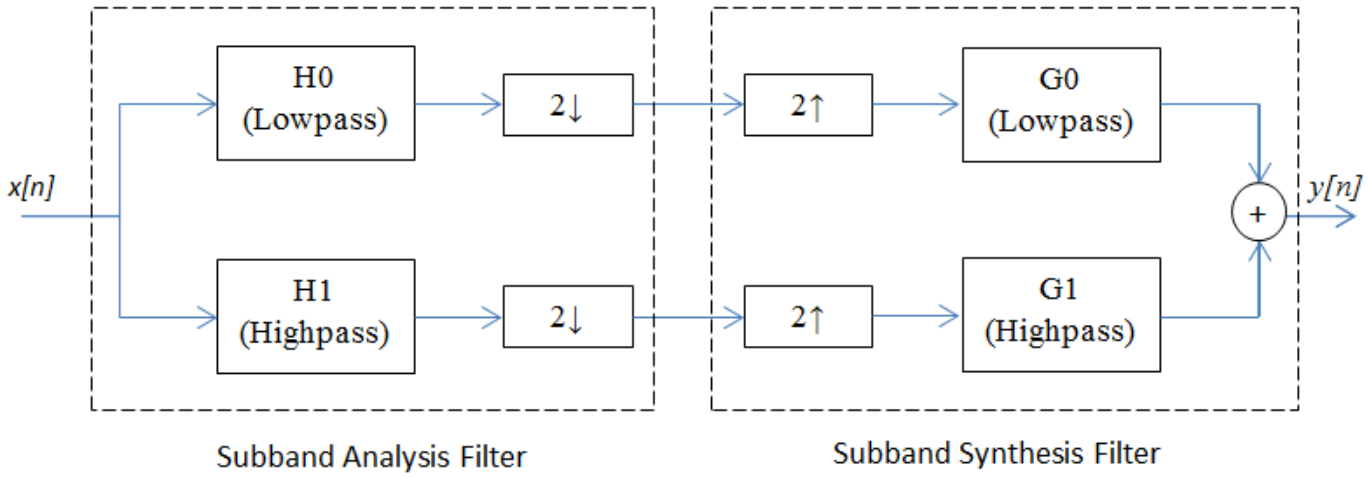
This example shows how to design perfect reconstruction two-channel filter banks, also known as the Quadrature Mirror Filter (QMF) banks. The QMF banks use power complementary filters.

There are several digital signal processing applications in which signals are first split into low frequency and high frequency subbands, and later these subbands are combined to reconstruct the original signal. One such application is subband coding (SBC).

First simulate the perfect reconstruction process by filtering a signal made up of scaled and time shifted impulses. Then plot the input, output, and error signals along with the magnitude spectrum of the transfer function of the complete system. The effectiveness of the perfect reconstruction is shown through the filter bank described in this example.

Perfect Reconstruction

A system is said to achieve perfect reconstruction when the output of the system is equal to the input signal or a delayed version of the same. Below is a block diagram of a perfect reconstruction process which uses ideal filters to separate the signal into its low frequency and high frequency components, and to recover the signal completely. The perfect reconstruction process requires four filters, two lowpass filters (H_0 and G_0) and two highpass filters (H_1 and G_1). In addition, the process requires a downsampler and an upsampler between the two lowpass and the two highpass filters. Since the analysis filters have unit gain, the synthesis filters compensate for the preceeding upsamplers by having a gain of 2.



Perfect Reconstruction Two-Channel Filter Bank

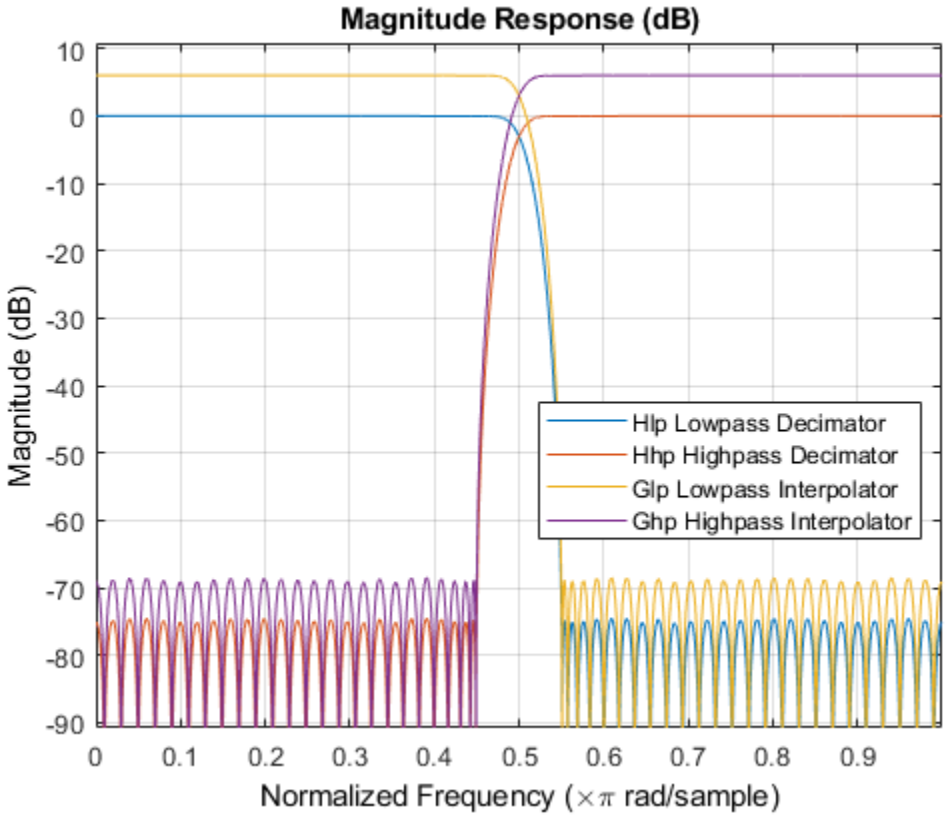
The DSP System Toolbox™ provides a specialized function called `firpr2chfb` to design the four filters required to implement an FIR perfect reconstruction two-channel filter bank. The `firpr2chfb` function designs the four FIR filters for the analysis (H_0 and H_1) and synthesis (G_0 and G_1) sections of a two-channel perfect reconstruction filter bank. The design corresponds to orthogonal filter banks known as power-symmetric filter banks, which are required in order to achieve the perfect reconstruction.

Design a filter bank with filters of order 99, passband edges of the lowpass and highpass filters at 0.45 and 0.55, respectively:

```
N = 99;
[LPAnalysis, HPAnalysis, LPSynthsis, HPSynthesis] = firpr2chfb(N, 0.45);
```

The magnitude response of these filters is plotted below:

```
fvt = fvtool(LPAnalysis,1, HPAnalysis,1, LPSynthsis,1, HPSynthesis,1);
fvt.Color = [1,1,1];
legend(fvt,'Hlp Lowpass Decimator','Hhp Highpass Decimator',...
    'Glp Lowpass Interpolator','Ghp Highpass Interpolator');
```



Note that the analysis path consists of a filter followed by a downsampler, which is a decimator, and the synthesis path consists of an upsampler followed by a filter, which is an interpolator. The DSP System Toolbox provides `dsp.SubbandAnalysisFilterr` and `dsp.SubbandSynthesisFilter` objects to implement the analysis portion and the synthesis portion of the filter bank, respectively.

```
% Analysis section
analysisFilter = dsp.SubbandAnalysisFilter(LPAnalysis, HPAnalysis);
% Synthesis section
synthFilter = dsp.SubbandSynthesisFilter(LPSynthsis, HPSynthesis);
```

For the sake of an example, let $p[n]$ denote the signal

$$p[n] = \delta[n] + \delta[n-1] + \delta[n-2]$$

and let the signal $x[n]$ be defined by

$$x[n] = p[n] + 2p[n-8] + 3p[n-16] + 4p[n-24] + 3p[n-32] + 2p[n-40] + p[n-48]$$

Note: Since MATLAB® uses one-based indexing, *delta*[*n*] = 1 when *n* = 1.

```
x = zeros(50,1);
x(1:3)   = 1;
x(8:10)  = 2;
x(16:18) = 3;
x(24:26) = 4;
x(32:34) = 3;
x(40:42) = 2;
x(48:50) = 1;
sigsource = dsp.SignalSource('SignalEndAction', 'Cyclic repetition',...
    'SamplesPerFrame', 50);
sigsource.Signal = x;
```

To view the results of the simulation, we need four scopes:

1. To compare the input signal with the reconstructed output.
2. To measure the error between the input signal and the reconstructed output
3. To plot the magnitude response of the overall system.
4. To plot the phase response of the overall system.

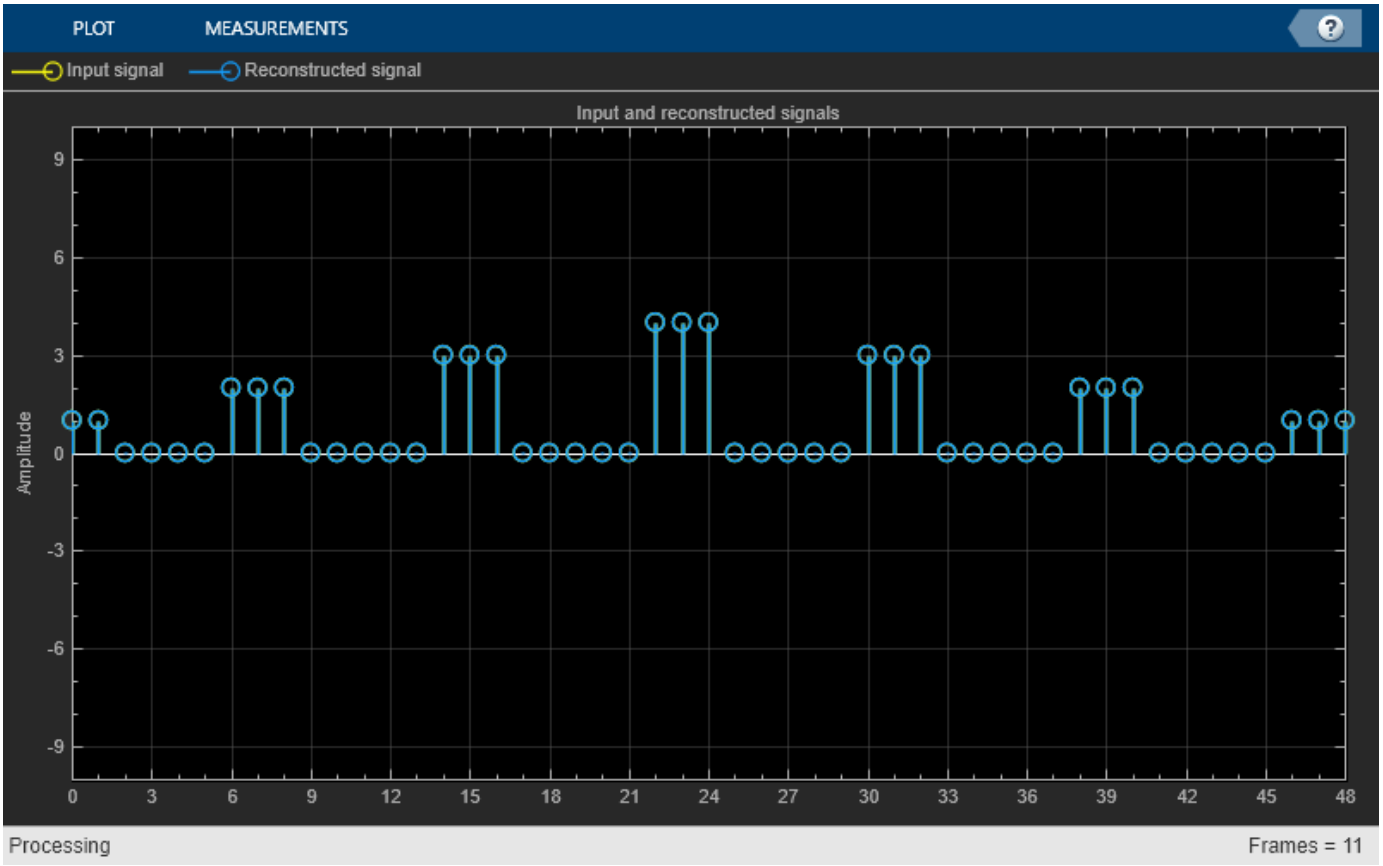
```
% Scope to compare input signal with reconstructed output
sigcompare = dsp.ArrayPlot('NumInputPorts', 2, 'ShowLegend', true,...
    'Title', 'Input and reconstructed signals',...
    'ChannelNames',{'Input signal','Reconstructed signal'});

% Scope to plot the RMS error between the input and reconstructed signals
errorPlot = timescope('Title', 'RMS Error', 'SampleRate', 1, ...
    'TimeUnits', 'seconds', 'YLimits', [-0.5 2],...
    'TimeSpanSource', 'property','TimeSpan',100,...
    'TimeSpanOverrunAction','scroll');

% To calculate the transfer function of the cascade of Analysis and
% Synthesis subband filters
tfestimate = dsp.TransferFunctionEstimator('FrequencyRange','centered',...
    'SpectralAverages', 50);
% Scope to plot the magnitude response of the estimated transfer function
magplot = dsp.ArrayPlot('PlotType','Line', ...
    'YLabel', 'Magnitude Response (dB)',...
    'Title','Magnitude response of the estimated transfer function',...
    'XOffset',-25, 'XLabel','Frequency (Hz)',...
    'YLimits',[-5 5]);
% Scope to plot the phase response of the estimated transfer function
phaseplot = dsp.ArrayPlot('PlotType','Line', ...
    'YLabel', 'Phase Response (degrees)',...
    'Title','Phase response of the estimated transfer function',...
    'XOffset',-25, 'XLabel','Frequency (Hz)',...
    'YLimits',[-180 180]);
```

Simulation of Perfect Reconstruction

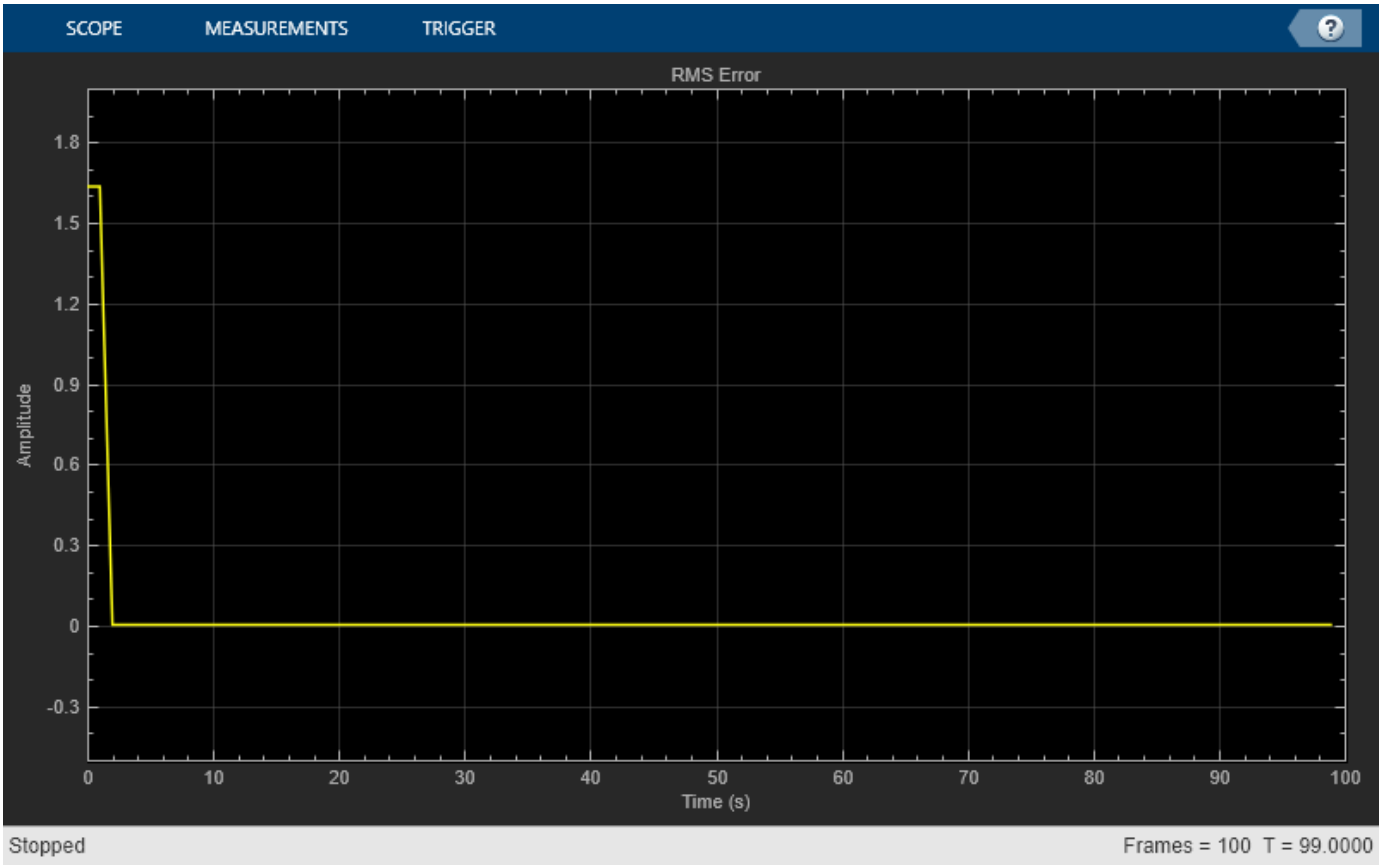
Pass the input signal through the subband filters and reconstruct the output. Plot the results on the scopes.

```
for i = 1:100
    % Use the same signal as input in each iteration.
    input = sigsource();
```

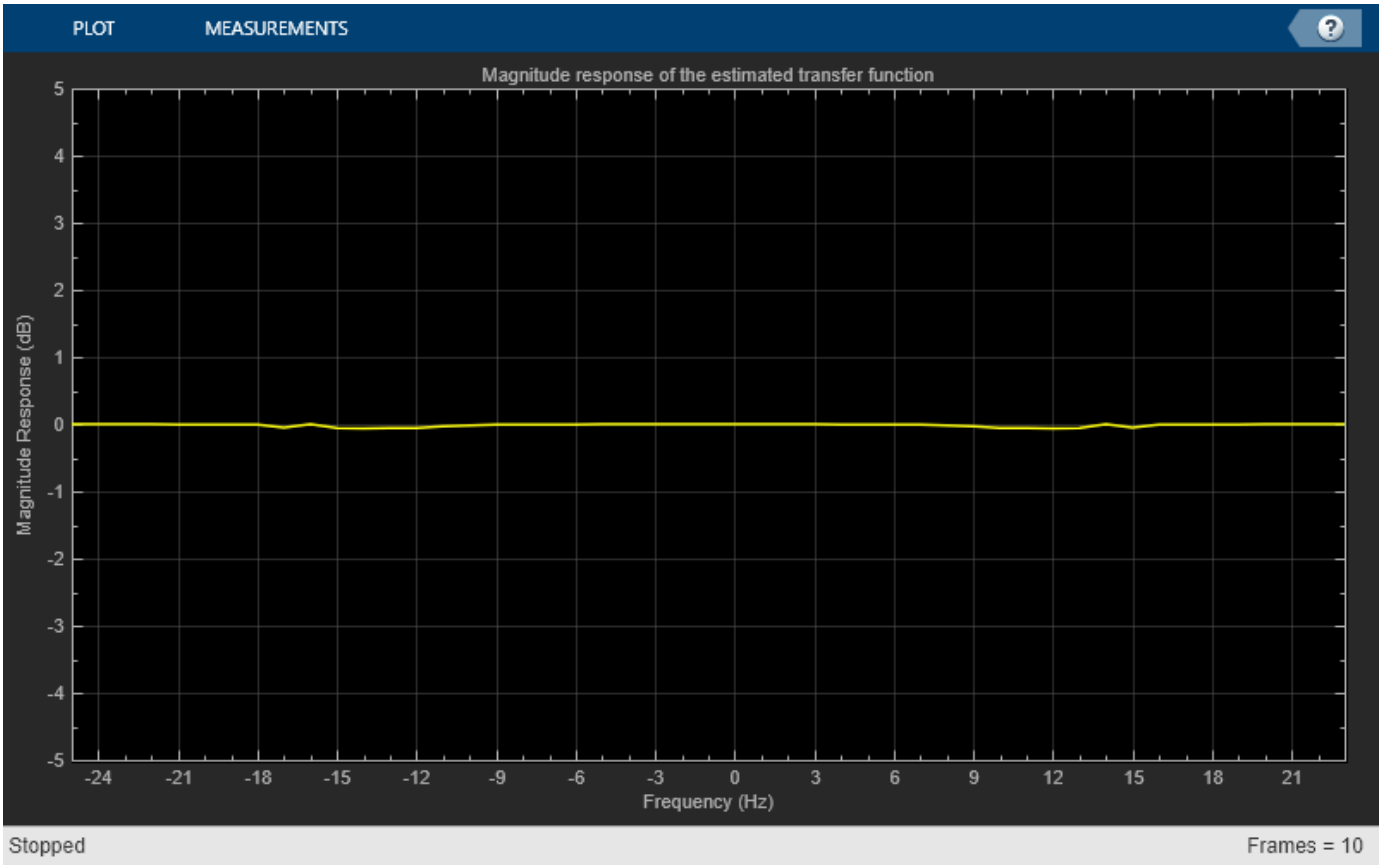
```
        % Analysis
        [hi, lo] = analysisFilter(input);
        % Synthesis
        reconstructed = synthFilter(hi, lo);

        % Compensate for the delay caused by the filters and compare the
        % signals. Since input signal is periodic, compare the current
        % frames of input and output signals.
        sigcompare(input(2:end), reconstructed(1:end-1));

        % Plot error between signals
        err = rms(input(2:end) - reconstructed(1:end-1));
        errorPlot(err);

        % Estimate transfer function of cascade
        Txy = tfestimate(input(2:end), reconstructed(1:end-1));
        magplot(20*log10(abs(Txy)));
        phaseplot(angle(Txy)*180/pi);
    end
```



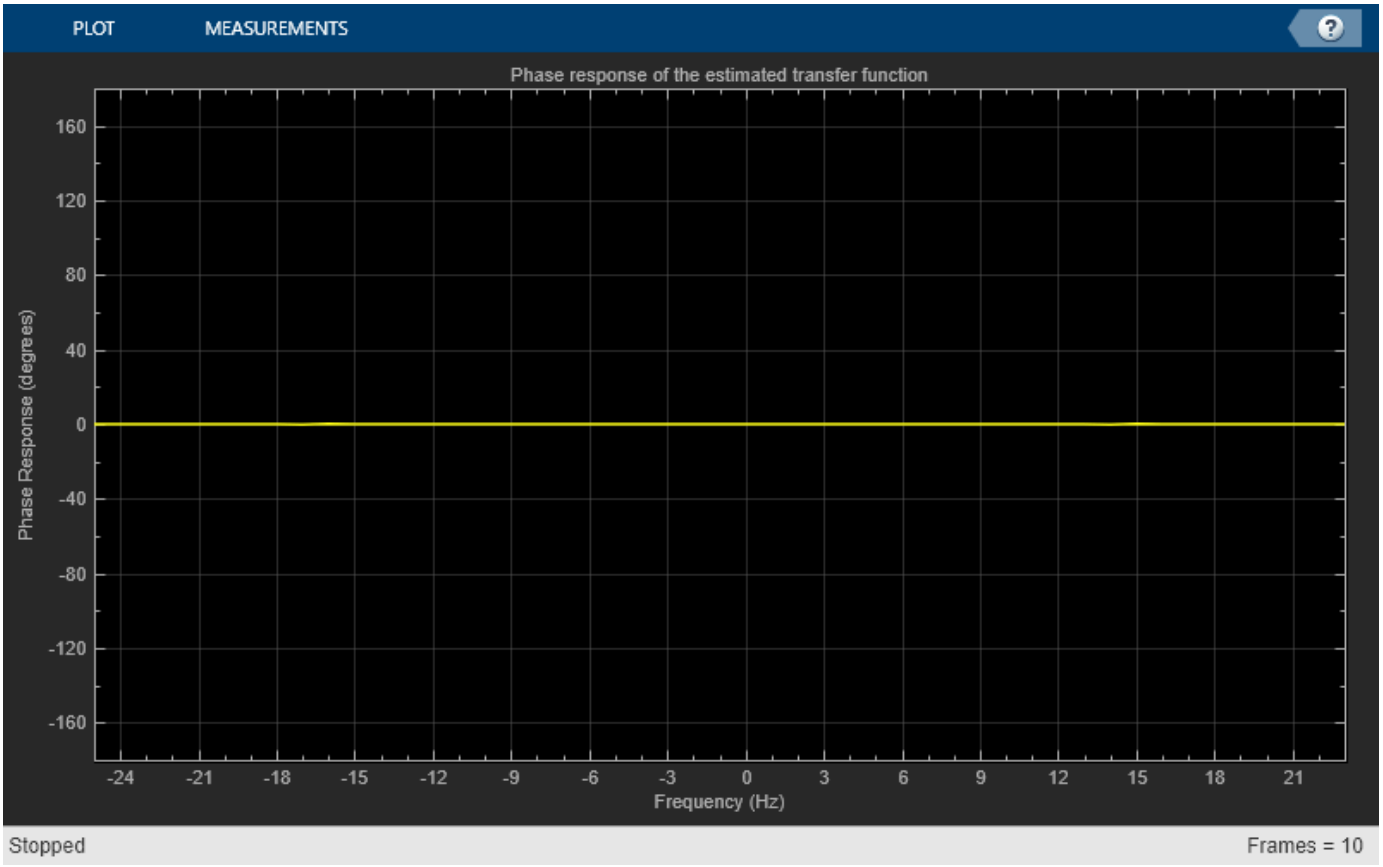

```
    release(errorPlot);
```




```
release(magplot);
```




```
release(phaseplot);
```



Perfect Reconstruction Output Analysis

The first two plots show that the two-channel filter bank perfectly reconstructs the original signal $x[n]$. The initial error in the second plot is due to delay in the filters. The third and the fourth plots show that the overall system has a magniude response of 0 dB and phase response of 0°, thereby preserving the frequency characteristics of the signal.

Adaptive Line Enhancer (ALE)

This example shows how to apply adaptive filters to signal separation using a structure called an adaptive line enhancer (ALE). In adaptive line enhancement, a measured signal x(n) contains two signals, an unknown signal of interest v(n), and a nearly-periodic noise signal eta(n).

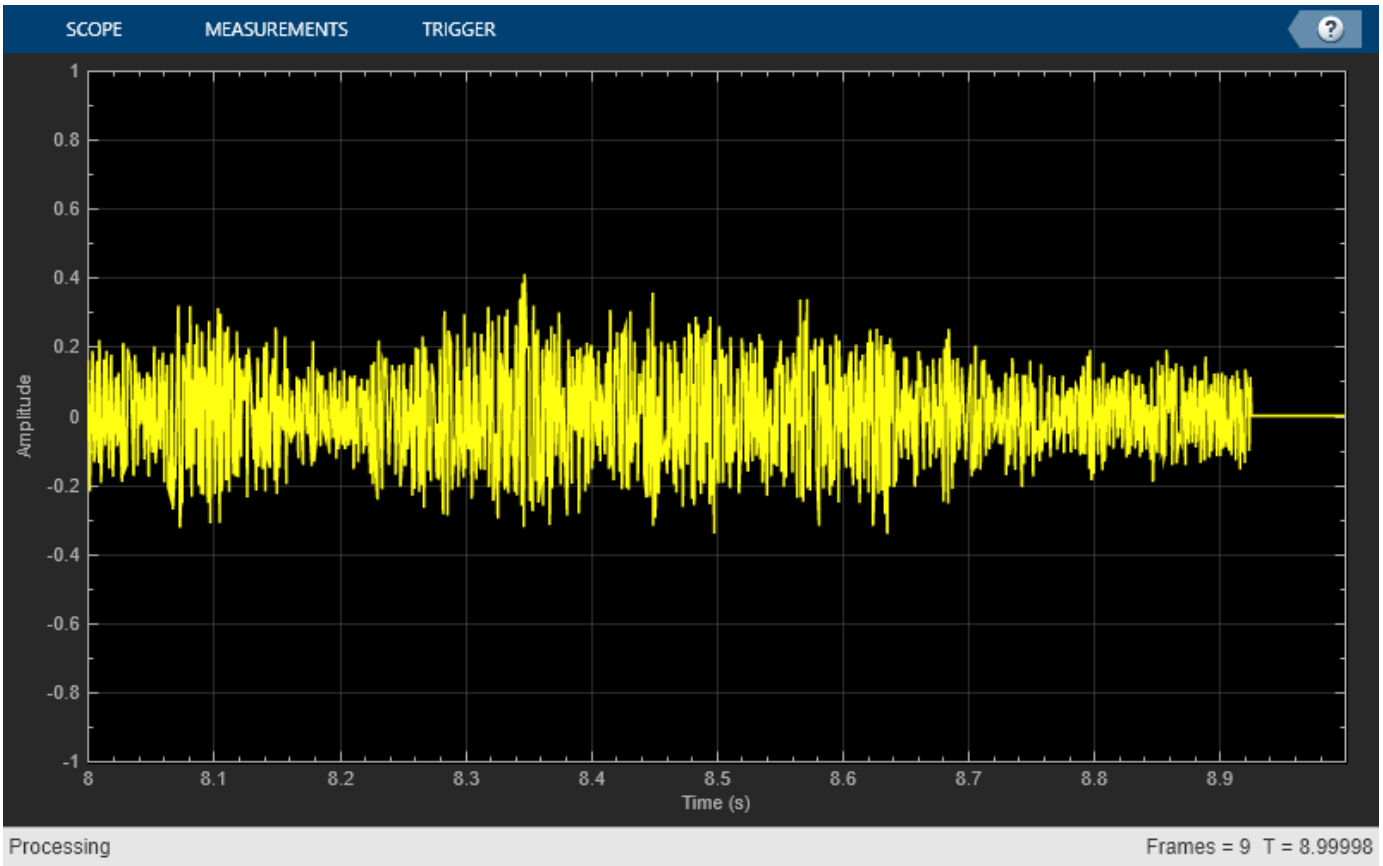
The goal is to remove the noise signal from the measured signal to obtain the signal of interest.

Author(s): Scott C. Douglas

Loading the Signal of Interest

We'll first load in a signal of interest, a short clip from Handel's Hallelujah chorus.

```
audioReader = dsp.AudioFileReader('handel.ogg','SamplesPerFrame',44100);
timeScope = timescope('SampleRate',audioReader.SampleRate,...
    'YLimits',[-1,1],'TimeSpanSource','property','TimeSpan',1);

while ~isDone(audioReader)
    x = audioReader() / 2;
    timeScope(x);
end
```



Listening to the Sound Clip

You can listen to the signal of interest using the audio device writer.

```
release(audioReader);
audioWriter = audioDeviceWriter;
while ~isDone(audioReader)
    x = audioReader() / 2;
    audioWriter(x);
end
```

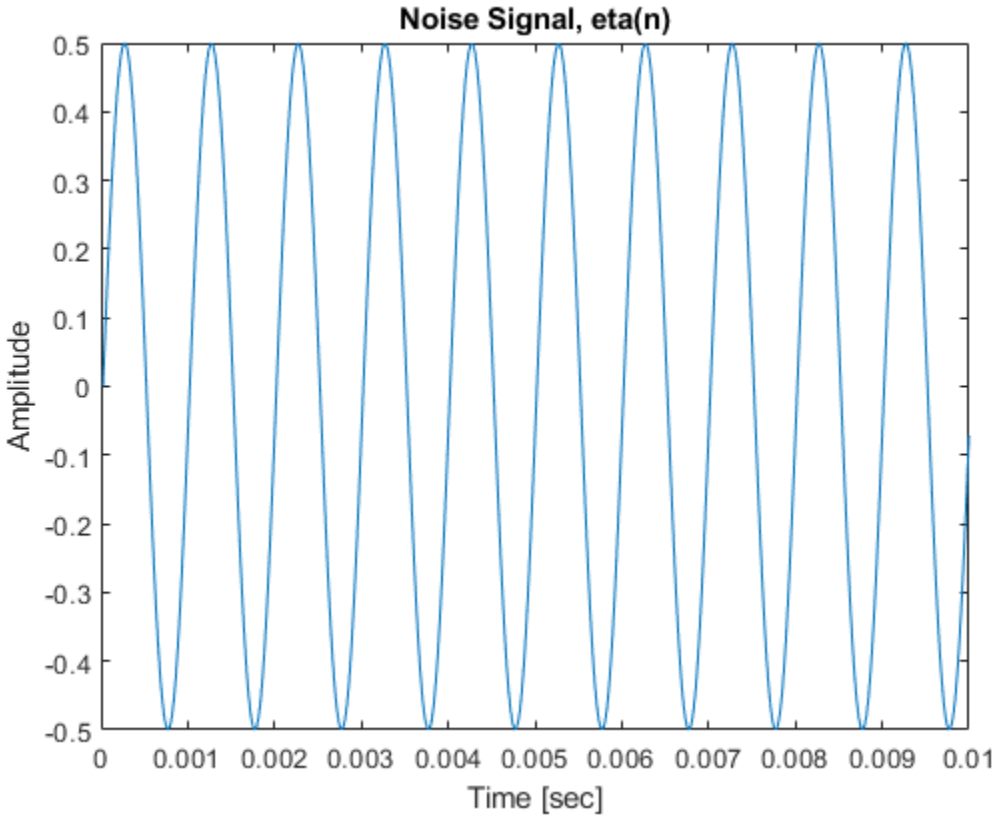
Generating the Noise Signal

Let's now make a periodic noise signal--a sinusoid with a frequency of 1000 Hz.

```
sine = dsp.SineWave('Amplitude',0.5,'Frequency',1000,...
    'SampleRate',audioReader.SampleRate,...
    'SamplesPerFrame',audioReader.SamplesPerFrame);
```

Now let's plot 10 msec of this sinusoid above. It shows 10 periods in 10 msec, just as it should.

```
eta = sine();
Fs  = sine.SampleRate;
plot(1/Fs:1/Fs:0.01,eta(1:floor(0.01*Fs)));
xlabel('Time [sec]');
ylabel('Amplitude');
title('Noise Signal, eta(n)');
```



Listening to the Noise

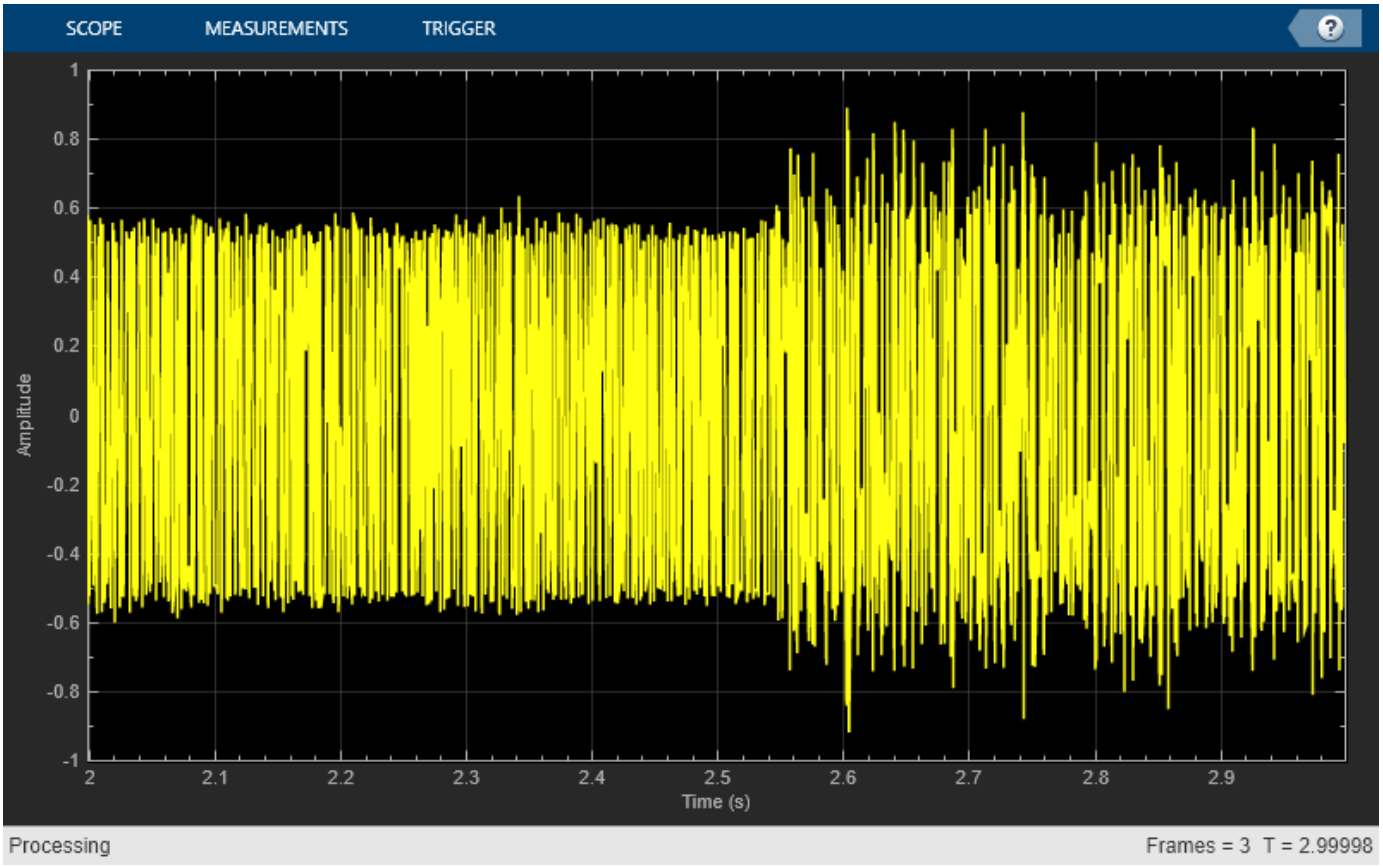
The periodic noise is a pure tone. The following code plays one second (one frame of 44100 samples) of the noise signal.

```
eta = sine();
release(audioWriter);
audioWriter(eta);
```

Measured Signal

The signal that we actually measure is the sum of these two signals, and we call this signal s(n). A plot of s(n) shows that the envelope of the music signal is largely obscured. Listening to a 3-second clip from the measured signal, the noise is clearly prominent...

```
release(audioReader);
release(timeScope);
release(audioWriter);
count = 1;
while count < 4
    s = (audioReader() / 2) + sine();
    timeScope(s);
    audioWriter(s);
    count = count + 1;
end
```



Adaptive Filter Configuration

An adaptive line enhancer (ALE) is based on the straightforward concept of linear prediction. A nearly-periodic signal can be perfectly predicted using linear combinations of its past samples, whereas a non-periodic signal cannot. So, a delayed version of the measured signal s(n-D) is used as the reference input signal x(n) to the adaptive filter, and the desired response signal d(n) is made

equal to s(n). The parameters to choose in such a system are the signal delay D and the filter length L used in the adaptive linear estimate. The amount of delay depends on the amount of correlation in the signal of interest. Since we don't have this signal (if we did, we wouldn't need the ALE!), we shall just pick a value of D=100 and vary it later. Such a choice suggests that samples of the Hallelujah Chorus are uncorrelated if they are more than about 12 msec apart. Also, we'll choose a value of L=32 for the adaptive filter, although this too could be changed.

```
D = 100;
delay = dsp.Delay(D);
```

Finally, we shall be using some block adaptive algorithms that require the lengths of the vectors for x(n) and d(n) to be integer multiples of the block length. We'll choose a block length of N=49 with which to begin.

Block LMS

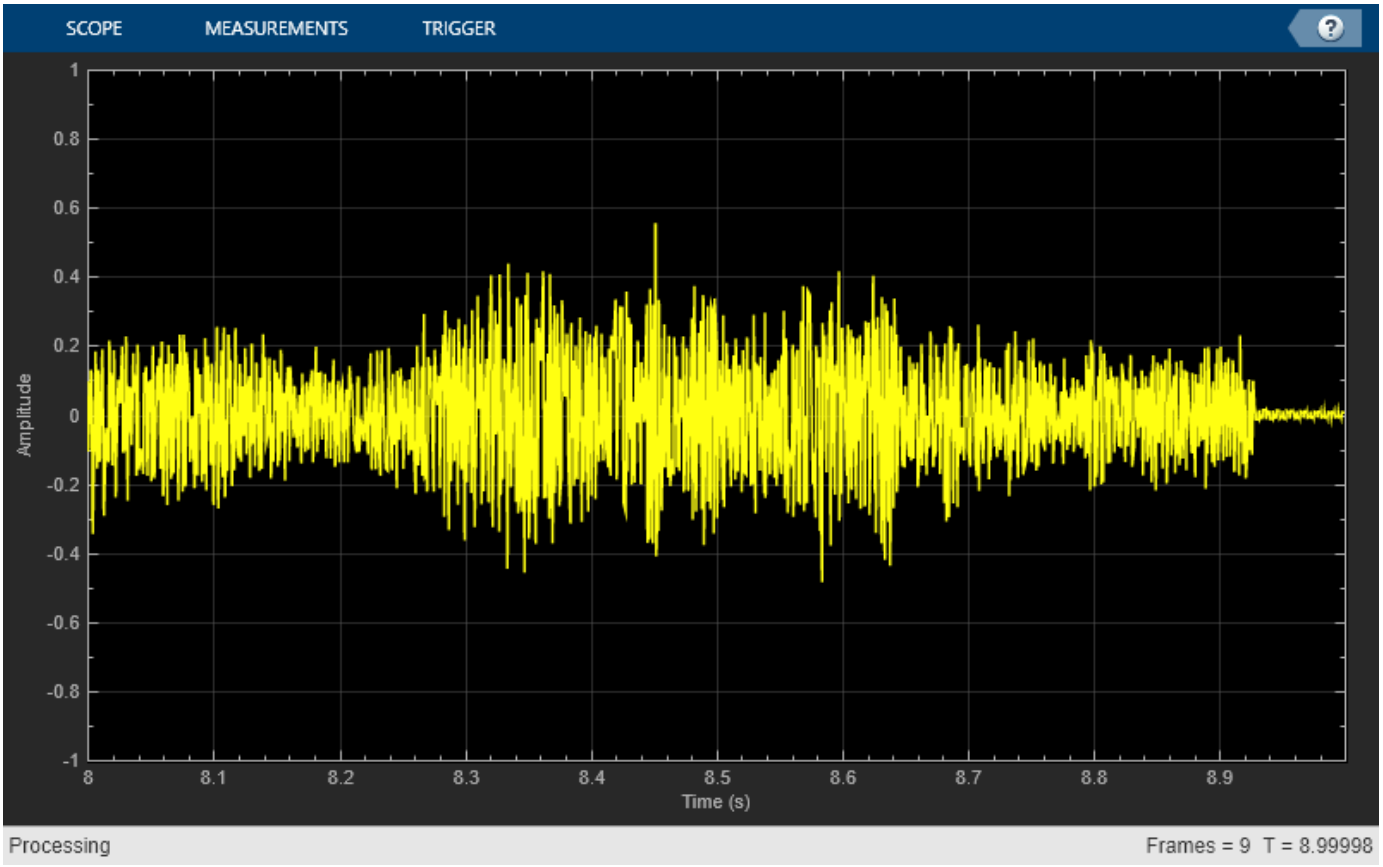
The first algorithm we shall explore is the Block LMS algorithm. This algorithm is similar to the well-known least-mean-square (LMS) algorithm, except that it employs block coefficient updates instead of sample-by-sample coefficient updates. The Block LMS algorithm needs a filter length, a block length N, and a step size value mu. How do we pick mu? Let's start with a value of mu = 0.0001 and refine it shortly.

```
L  = 32;
N  = 49;
mu = 0.0001;
blockLMSFilter = ...
    dsp.BlockLMSFilter('Length',L,'StepSize',mu,'BlockSize',N);
```

Running the Filter

The output signal y(n) should largely contain the periodic sinusoid, whereas the error signal e(n) should contain the musical information, if we've done everything right. Since we have the original music signal v(n), we can plot e(n) vs. v(n) on the same plot shown above along with the residual signal e(n)-v(n). It looks like the system is converged after about 5 seconds of adaptation with this step size. The real proof, however, is obtained by listening;

```
release(audioReader);
release(timeScope);
release(audioWriter);
while ~isDone(audioReader)
    x = audioReader() / 2;
    s = x + sine();
    d = delay(s);
    [y,e] = blockLMSFilter(s,d);
    timeScope(e);
    audioWriter(e);
end
```

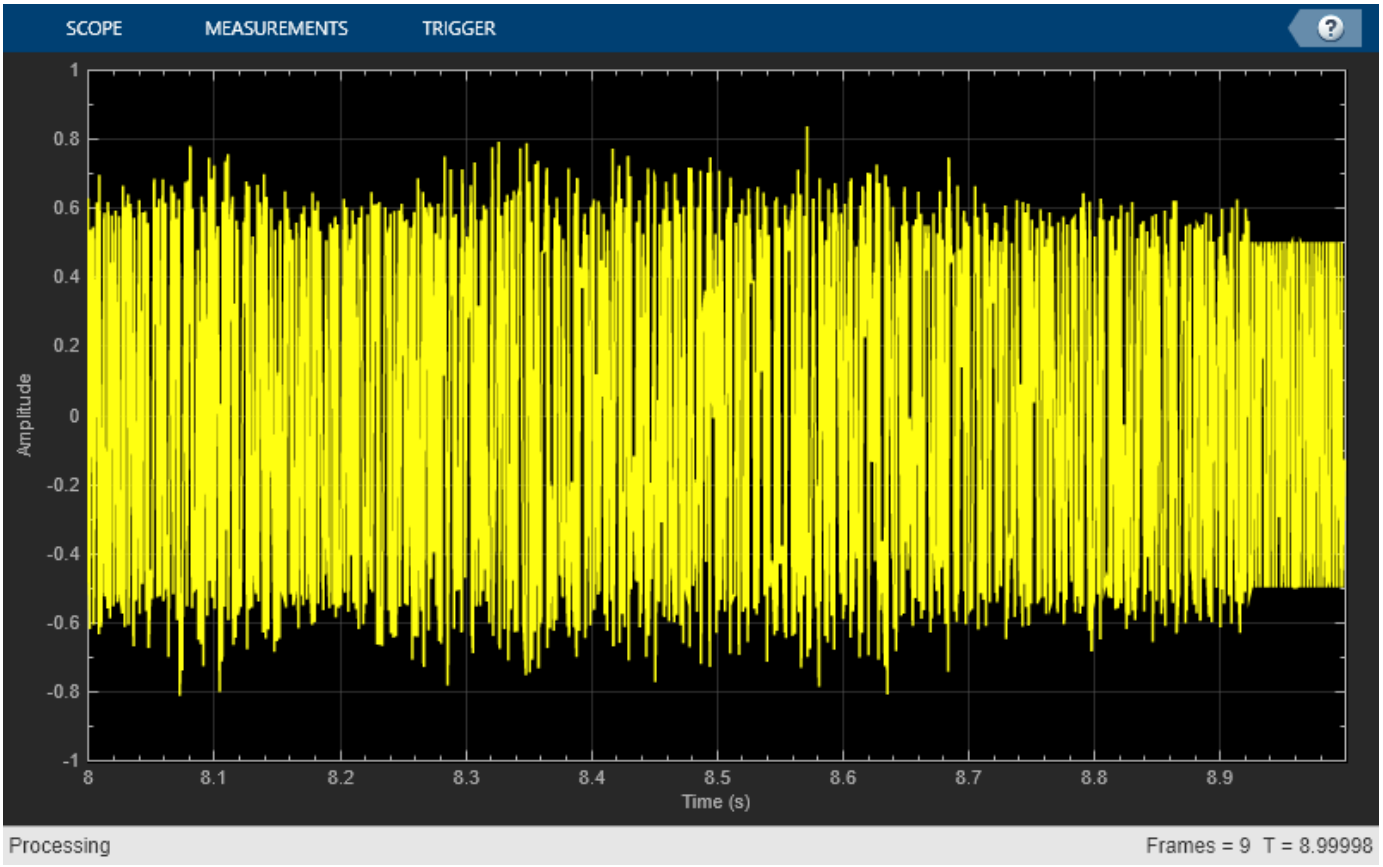


Notice how the sinusoidal noise decays away slowly. This behavior is due to the adaptation of the filter coefficients toward their optimum values.

FM Noise Source

Now, removing a pure sinusoid from a sinusoid plus music signal is not particularly challenging if the frequency of the offending sinusoid is known. A simple two-pole, two-zero notch filter can perform this task. So, let's make the problem a bit harder by adding an FM-modulated sinusoidal signal as our noise source.

```
eta = 0.5 * sin(2*pi*1000/Fs*(0:396899)' + 10*sin(2*pi/Fs*(0:396899)'));
signalSource = dsp.SignalSource(eta,...
    'SamplesPerFrame',audioReader.SamplesPerFrame,...
    'SignalEndAction','Cyclic repetition');
release(audioReader);
release(timeScope);
release(audioWriter);
while ~isDone(audioReader)
    x = audioReader() / 2;
    s = x + signalSource();
    timeScope(s);
    audioWriter(s);
end
```



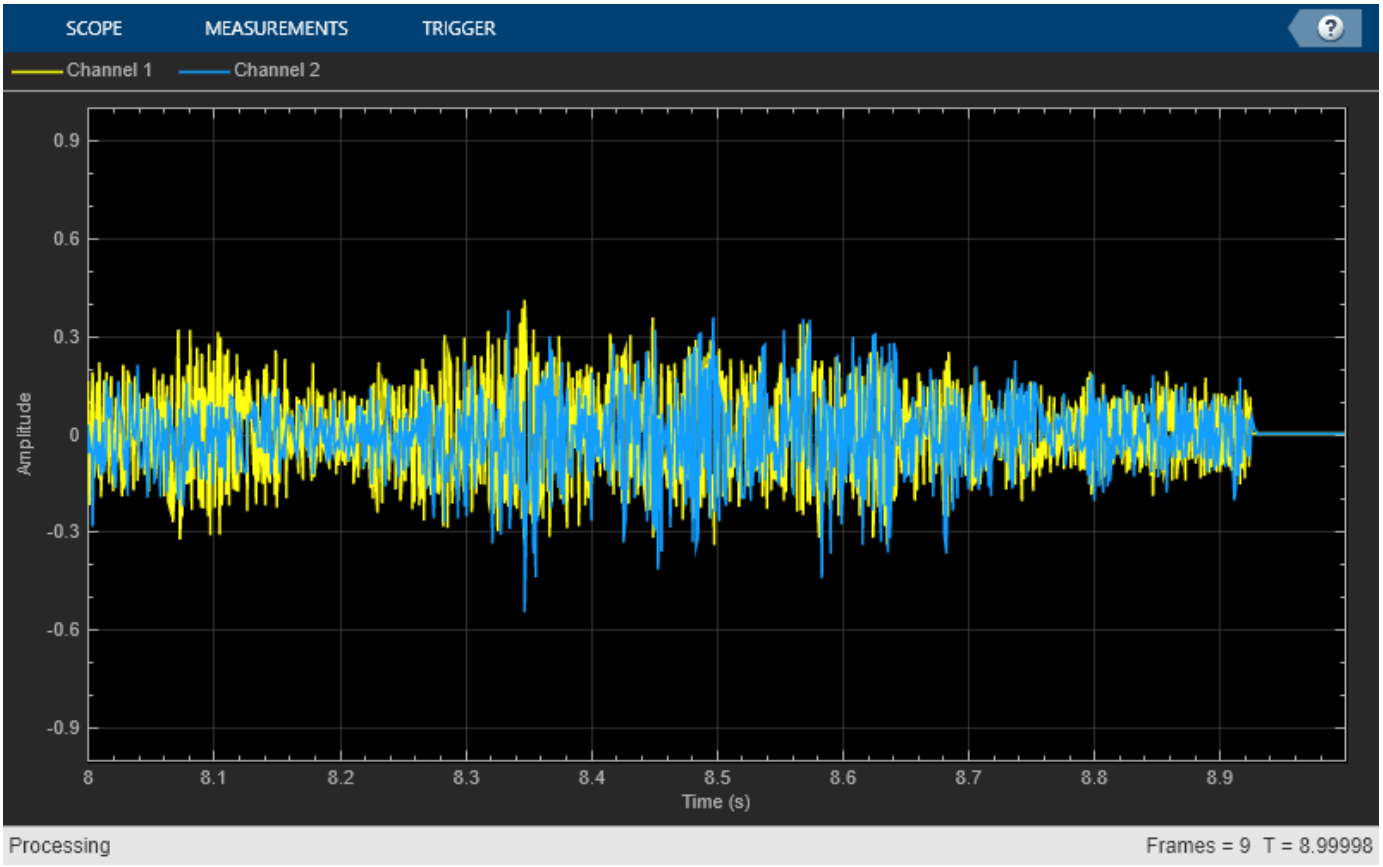
The "warble" in the signal is clearly audible. A fixed-coefficient notch filter won't remove the FM-modulated sinusoid. Let's see if the Block LMS-based ALE can. We'll increase the step size value to mu=0.005 to help the ALE track the variations in the noise signal.

```
mu = 0.005;
release(blockLMSFilter);
blockLMSFilter.StepSize = mu;
```

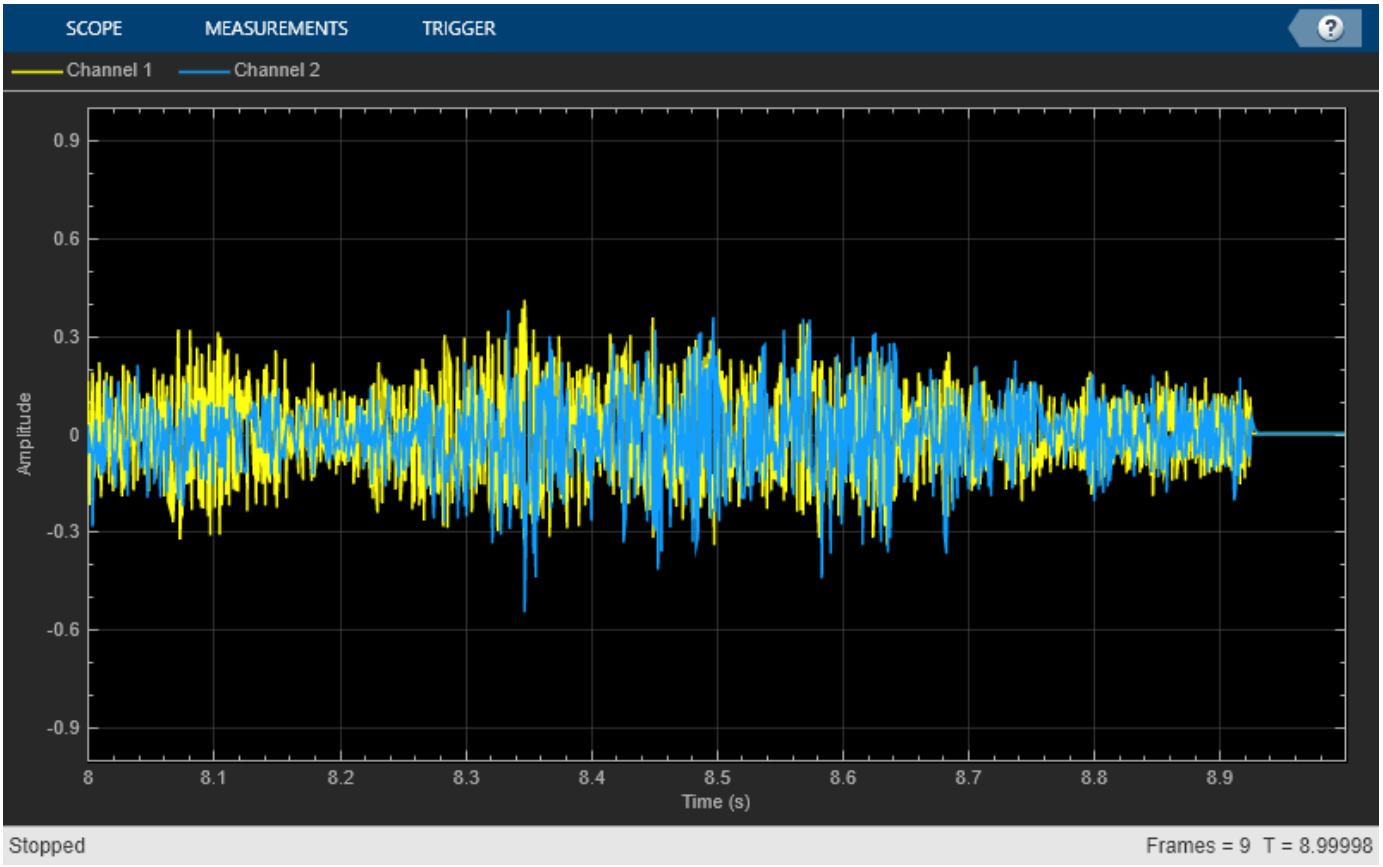
Running the Adaptive Filter

We now filter the noisy music signal with the adaptive filter and compare the error to the noiseless music signal.

```
release(audioReader);
release(timeScope);
release(audioWriter);
while ~isDone(audioReader)
    x = audioReader() / 2;
    s = x + signalSource();
    d = delay(s);
    [y,e] = blockLMSFilter(s,d);
    timeScope([x,e]);
    audioWriter(e);
end
```




```
release(audioReader);
release(timeScope);
release(audioWriter);
```


SCOPE
MEASUREMENTS
TRIGGER
Channel 1
Channel 2
0.9
0.6
0.3
0
-0.3
-0.6
-0.9
Amplitude
8
8.1
8.2
8.3
8.4
8.5
8.6
8.7
8.8
8.9
Time (s)
Stopped
Frames = 9 T = 8.99998

Adaptive Noise Canceling (ANC) Applied to Fetal Electrocardiography

This example shows how to apply adaptive filters to noise removal using adaptive noise canceling. The example uses a user interface (UI) which can be launched by typing the command `adaptiveNoiseCancellationExampleApp`. For more details, see 'Example Architecture' below.

Introduction

In adaptive noise canceling, a measured signal d(n) contains two signals: - an unknown signal of interest v(n) - an interference signal u(n) The goal is to remove the interference signal from the measured signal by using a reference signal x(n) that is highly correlated with the interference signal. The example considered here is an application of adaptive filters to fetal electrocardiography, in which a maternal heartbeat signal is adaptively removed from a fetal heartbeat sensor signal. This example is adapted from Widrow, et al, "Adaptive noise canceling: Principles and applications," Proc. IEEE®, vol. 63, no. 12, pp. 1692-1716, December 1975.

Creating the Maternal Heartbeat Signal

In this example, we shall simulate the shapes of the electrocardiogram for both the mother and fetus. We use a 4000 Hz sampling rate. The heart rate for this signal is approximately 89 beats per minute, and the peak voltage of the signal is 3.5 millivolts.

Creating the Fetal Heartbeat Signal

The heart of a fetus beats noticeably faster than that of its mother, with rates ranging from 120 to 160 beats per minute. The amplitude of the fetal electrocardiogram is also much weaker than that of the maternal electrocardiogram. The example creates an electrocardiogram signal corresponding to a heart rate of 139 beats per minute and a peak voltage of 0.25 millivolts for simulating fetal heartbeat.

The Measured Maternal Electrocardiogram

The maternal electrocardiogram signal is obtained from the chest of the mother. The goal of the adaptive noise canceller in this task is to adaptively remove the maternal heartbeat signal from the fetal electrocardiogram signal. The canceller needs a reference signal generated from a maternal electrocardiogram to perform this task. Just like the fetal electrocardiogram signal, the maternal electrocardiogram signal will contain some additive broadband noise.

The Measured Fetal Electrocardiogram

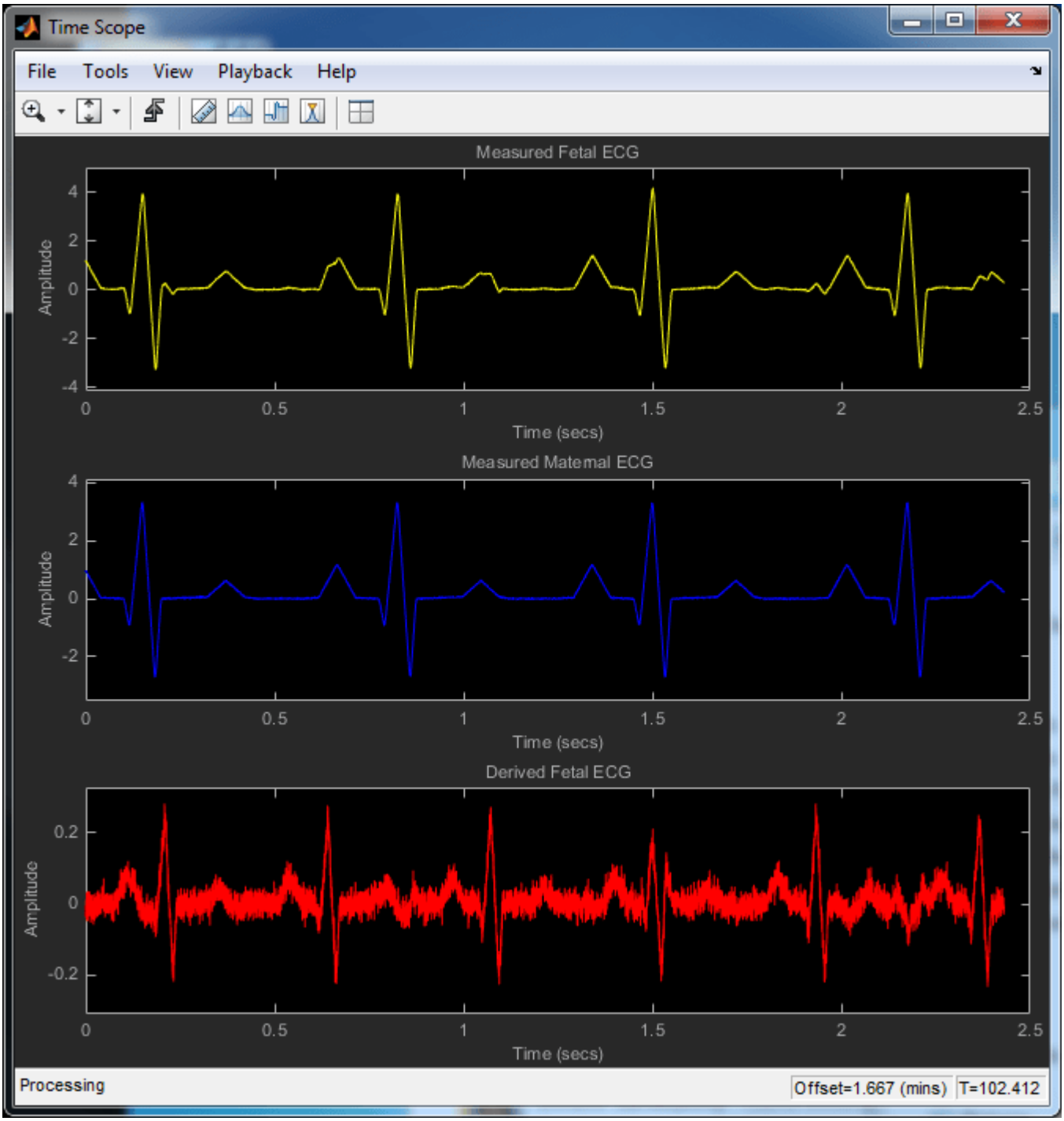
The measured fetal electrocardiogram signal from the abdomen of the mother is usually dominated by the maternal heartbeat signal that propagates from the chest cavity to the abdomen. We shall describe this propagation path as a linear FIR filter with 10 randomized coefficients. In addition, we shall add a small amount of uncorrelated Gaussian noise to simulate any broadband noise sources within the measurement.

Applying the Adaptive Noise Canceller

The adaptive noise canceller can use most any adaptive procedure to perform its task. For simplicity, we shall use the least-mean-square (LMS) adaptive filter with 15 coefficients and a step size of 0.00007. With these settings, the adaptive noise canceller converges reasonably well after a few seconds of adaptation--certainly a reasonable period to wait given this particular diagnostic application.

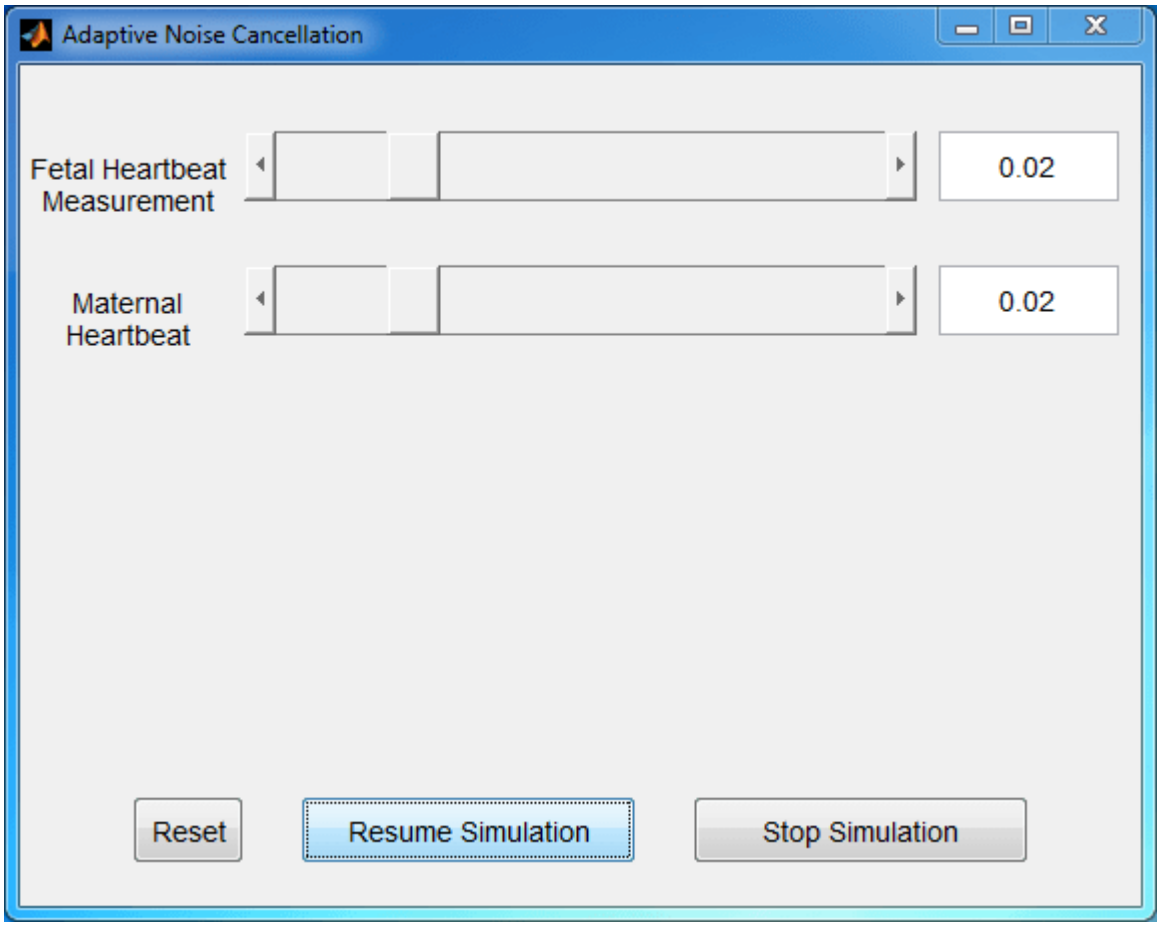
Recovering the Fetal Heartbeat Signal

The output signal y(n) of the adaptive filter contains the estimated maternal heartbeat signal, which is not the ultimate signal of interest. What remains in the error signal e(n) after the system has converged is an estimate of the fetal heartbeat signal along with residual measurement noise. Using the error signal, you can estimate the heart rate of the fetus.



Example Architecture

The command adaptiveNoiseCancellationExampleApp launches a user interface designed to interact with the simulation. It also launches a time scope to view the measured fetal heartbeat as well as the measured maternal heartbeat and the extracted fetal heartbeat.



Using a Generated MEX File

Using MATLAB Coder, you can generate a MEX file for the main processing algorithm by executing the command HelperANCCodeGeneration. You can use the generated MEX file by executing the command adaptiveNoiseCancellationExampleApp(true).

Adaptive Noise Cancellation Using RLS Adaptive Filtering

This example shows how to use an RLS filter to extract useful information from a noisy signal. The information bearing signal is a sine wave that is corrupted by additive white gaussian noise.

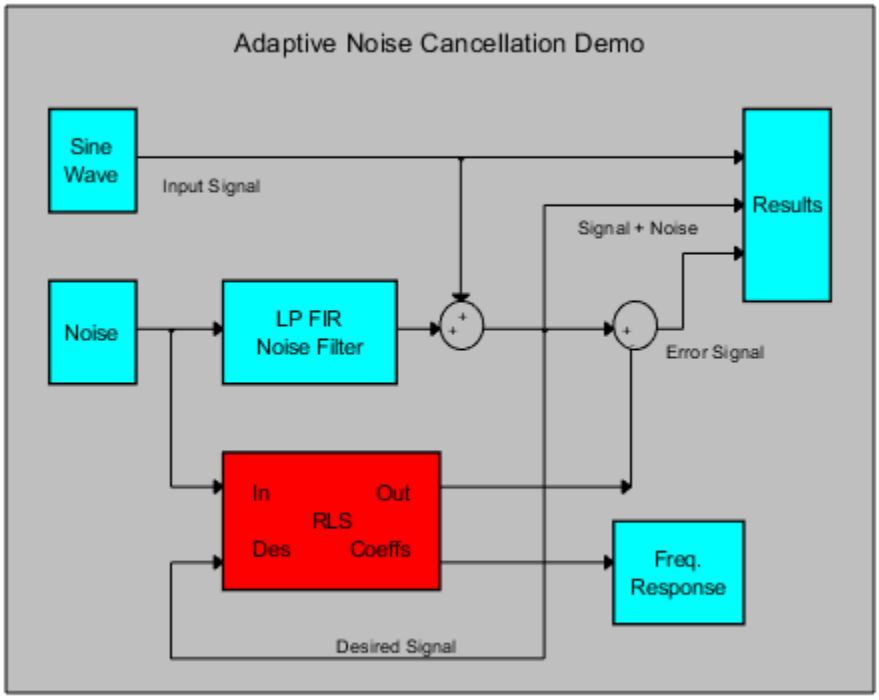
The adaptive noise cancellation system assumes the use of two microphones. A primary microphone picks up the noisy input signal, while a secondary microphone receives noise that is uncorrelated to the information bearing signal, but is correlated to the noise picked up by the primary microphone.

Note: This example is equivalent to the Simulink® model 'rlsdemo' provided.

Reference: S.Haykin, "Adaptive Filter Theory", 3rd Edition, Prentice Hall, N.J., 1996.

The model illustrates the ability of the Adaptive RLS filter to extract useful information from a noisy signal.

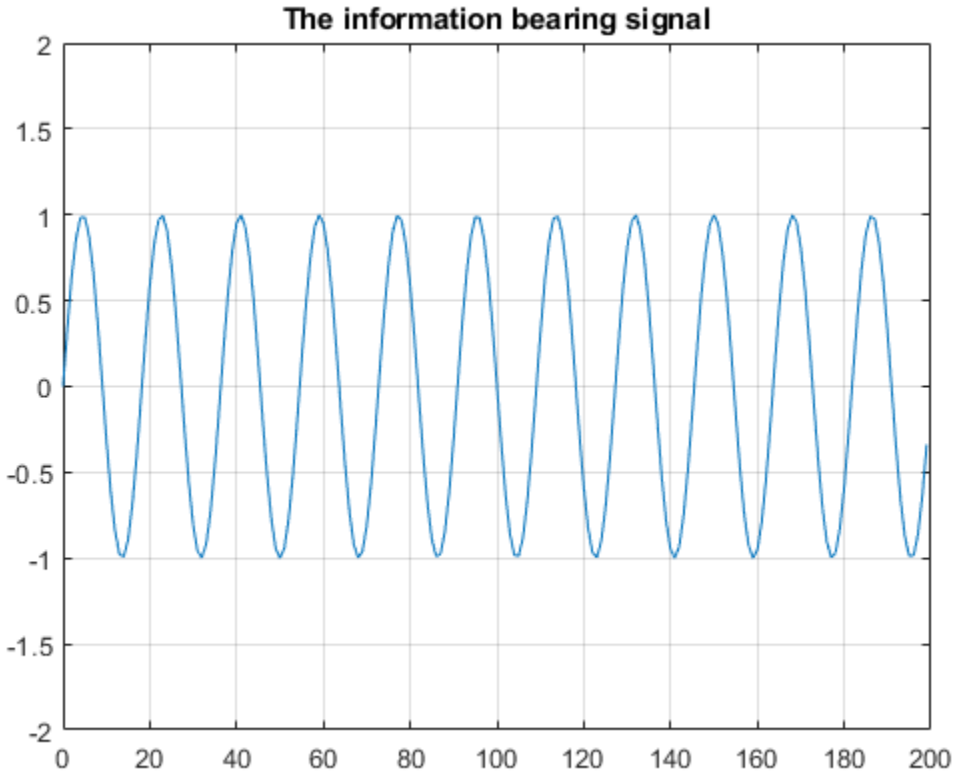
```
priv_drawrlsdemo
axis off
```



The information bearing signal is a sine wave of 0.055 cycles/sample.

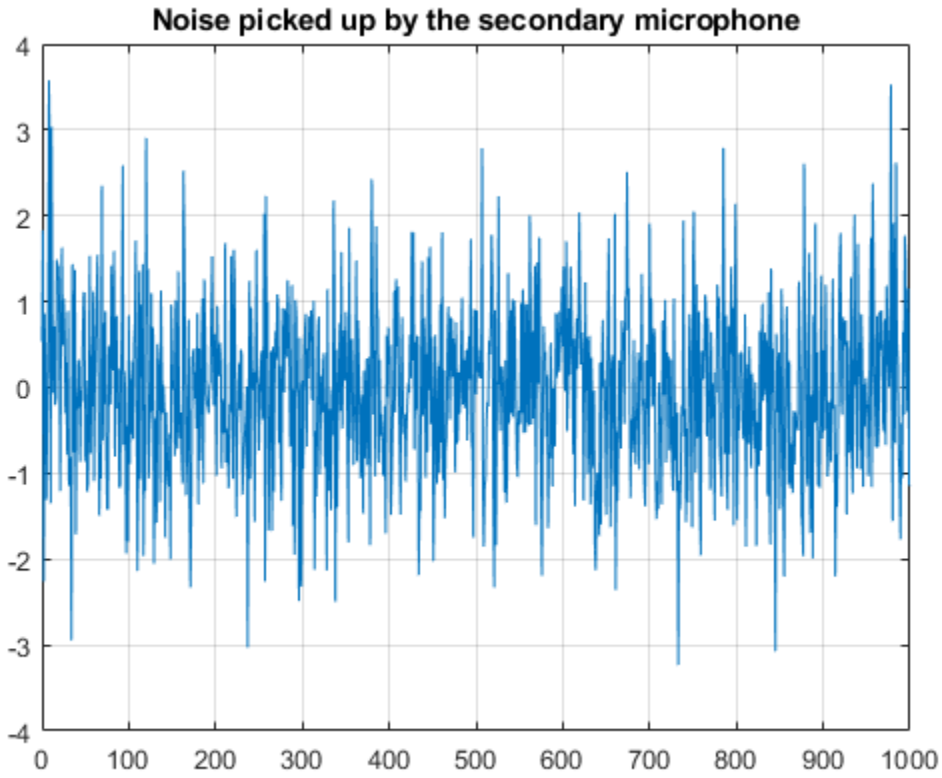
```
signal = sin(2*pi*0.055*(0:1000-1)');
signalSource = dsp.SignalSource(signal,'SamplesPerFrame',100,...
    'SignalEndAction','Cyclic repetition');

plot(0:199,signal(1:200));
```

```
grid; axis([0 200 -2 2]);
title('The information bearing signal');
```



The noise picked up by the secondary microphone is the input for the RLS adaptive filter. The noise that corrupts the sine wave is a lowpass filtered version of (correlated to) this noise. The sum of the filtered noise and the information bearing signal is the desired signal for the adaptive filter.

```
nvar  = 1.0;                  % Noise variance
noise = randn(1000,1)*nvar;   % White noise
noiseSource = dsp.SignalSource(noise,'SamplesPerFrame',100,...
    'SignalEndAction','Cyclic repetition');

plot(0:999,noise);
title('Noise picked up by the secondary microphone');
grid; axis([0 1000 -4 4]);
```



The noise corrupting the information bearing signal is a filtered version of 'noise'. Initialize the filter that operates on the noise.

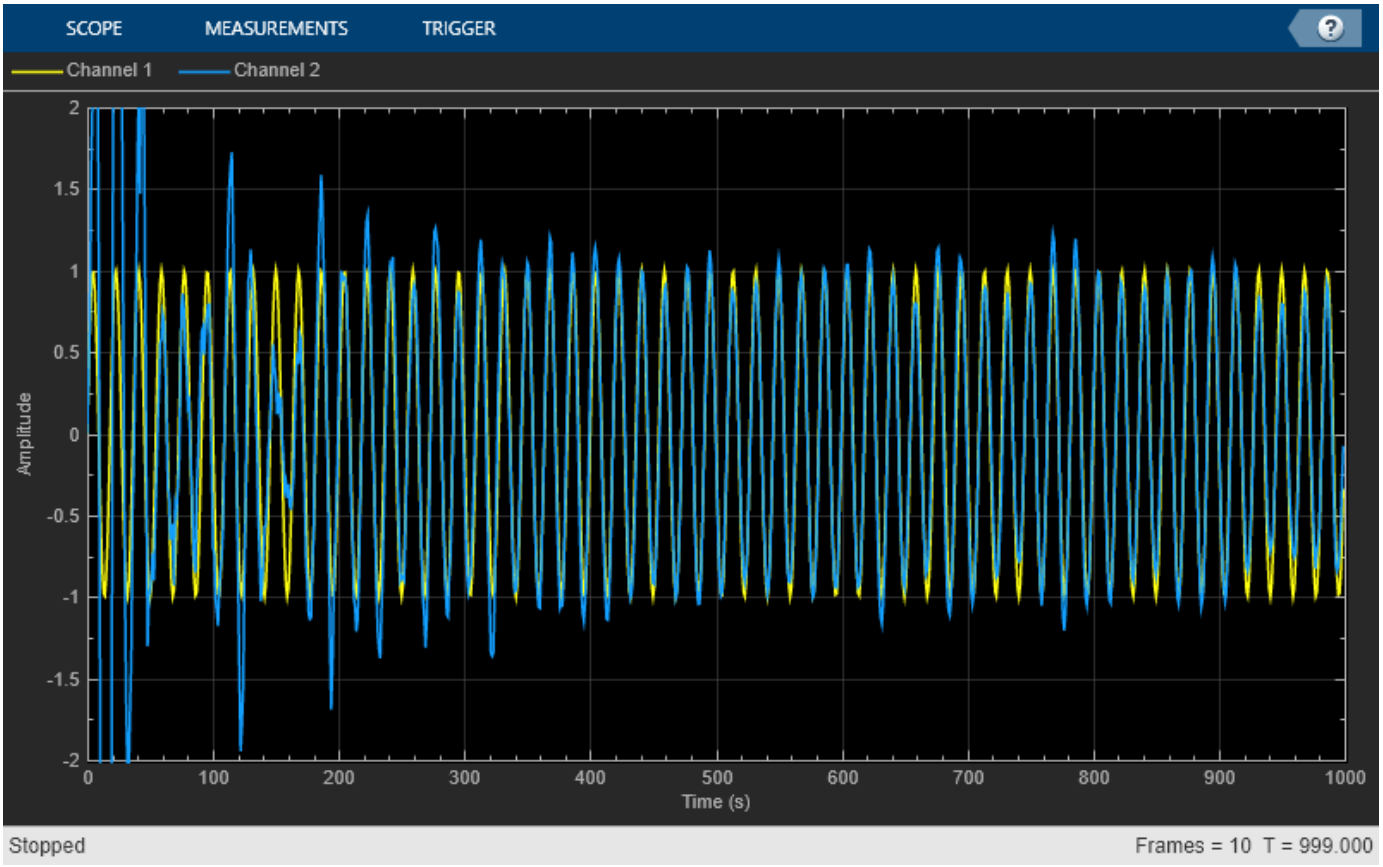
```
lp = dsp.FIRFilter('Numerator',fir1(31,0.5));% Low pass FIR filter
```

Set and initialize RLS adaptive filter parameters and values:

```
M      = 32;                % Filter order
delta  = 0.1;               % Initial input covariance estimate
P0     = (1/delta)*eye(M,M); % Initial setting for the P matrix
rlsfilt = dsp.RLSFilter(M,'InitialInverseCovariance',P0);
```

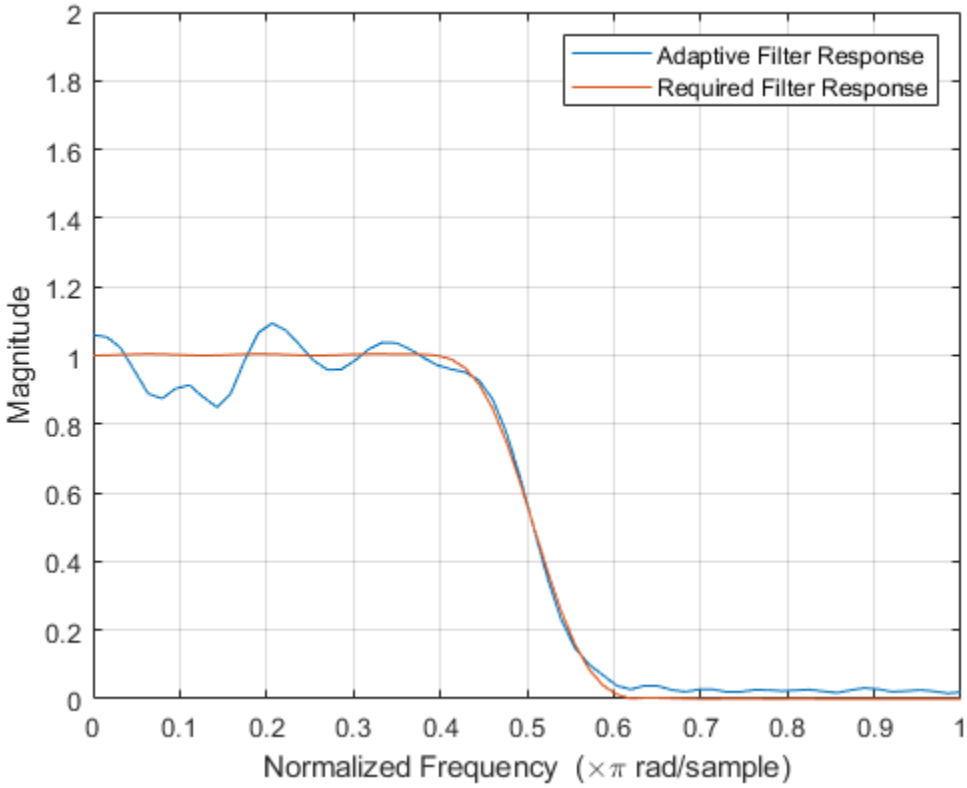
Running the RLS adaptive filter for 1000 iterations. As the adaptive filter converges, the filtered noise should be completely subtracted from the "signal + noise". Also the error, 'e', should contain only the original signal.

```
scope = timescope('TimeSpanSource','property','TimeSpan',1000, ...
                  'YLimits',[-2,2]);
for k = 1:10
    n = noiseSource(); % Noise
    s = signalSource();
    d = lp(n) + s;
    [y,e]  = rlsfilt(n,d);
    scope([s,e]);
end
release(scope);
```



The plot shows the convergence of the adaptive filter response to the response of the FIR filter.

```
H  = abs(freqz(rlsfilt.Coefficients,1,64));
H1 = abs(freqz(lp.Numerator,1,64));

wf = linspace(0,1,64);

plot(wf,H,wf,H1);
xlabel('Normalized Frequency  (\times\pi rad/sample)');
ylabel('Magnitude');
legend('Adaptive Filter Response','Required Filter Response');
grid;
axis([0 1 0 2]);
```


Adaptive Filter Response
Required Filter Response
Magnitude
Normalized Frequency (×π rad/sample)
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
0.1
0.3
0.5
0.7
0.9

System Identification Using RLS Adaptive Filtering

This example shows how to use a recursive least-squares (RLS) filter to identify an unknown system modeled with a lowpass FIR filter. The dynamic filter visualizer is used to compare the frequency response of the unknown and estimated systems. This example allows you to dynamically tune key simulation parameters using a user interface (UI). The example also shows you how to use MATLAB Coder to generate code for the algorithm and accelerate the speed of its execution.

Required MathWorks™ products:

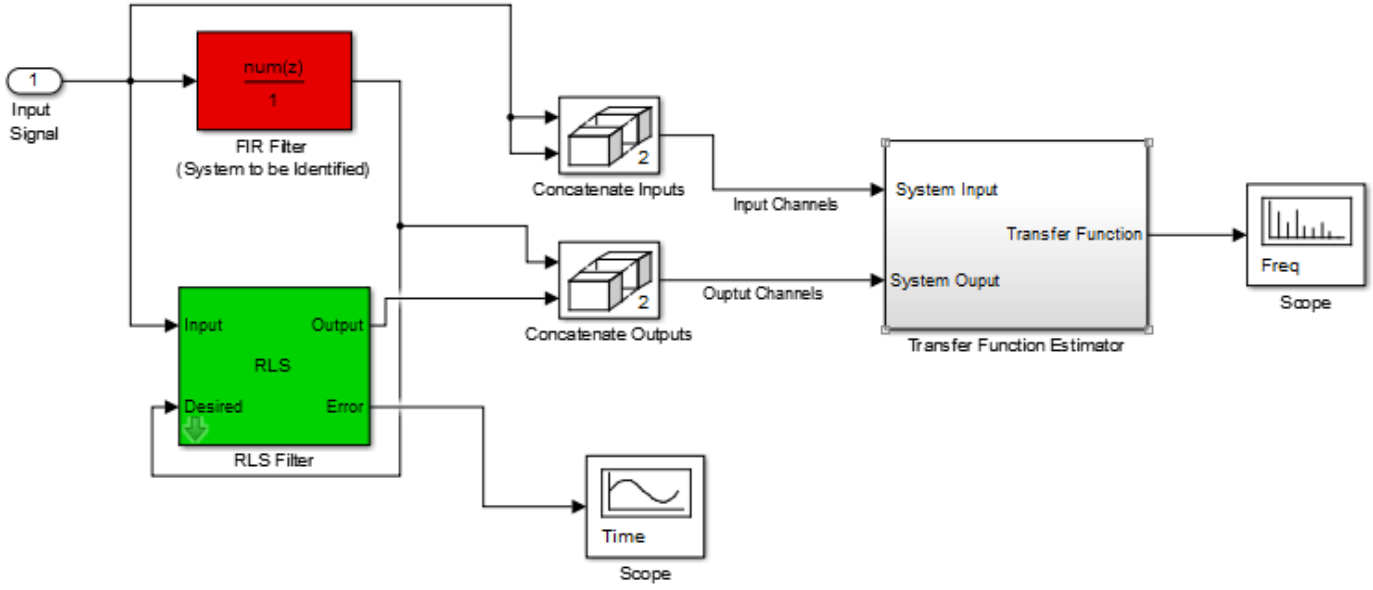
- DSP System Toolbox™

Optional MathWorks™ products:

- MATLAB Coder™ for generating C code from the MATLAB simulation
- Simulink™ for executing the Simulink version of the example

Introduction

Adaptive system identification is one of the main applications of adaptive filtering. This example showcases system identification using an RLS filter. The example's workflow is depicted below:



The unknown system is modeled by a lowpass FIR filter. The same input is fed to the FIR and RLS filters. The desired signal is the output of the unidentified system. The estimated weights of the RLS filter therefore converges to the coefficients of the FIR filter. The coefficients of the RLS filter and FIR filter are used by the dynamic filter visualizer to visualize the desired and estimated frequency response. The learning curve of the RLS filter (the plot of the mean square error (MSE) of the filter versus time) is also visualized.

Tunable FIR Filter

The lowpass FIR filter used in this example is modeled using a dsp.VariableBandwidthFIRFilter System object. This object allows you to tune the filter's cutoff frequency while preserving the FIR

structure. Tuning is achieved by multiplying each filter coefficient by a factor proportional to the current and desired cutoff frequencies. For more information on this object, type `dsp.VariableBandwidthFIRFilter`.

MATLAB Simulation

HelperRLSFilterSystemIdentificationSim is the function containing the algorithm's implementation. It instantiates, initializes and steps through the objects forming the algorithm.

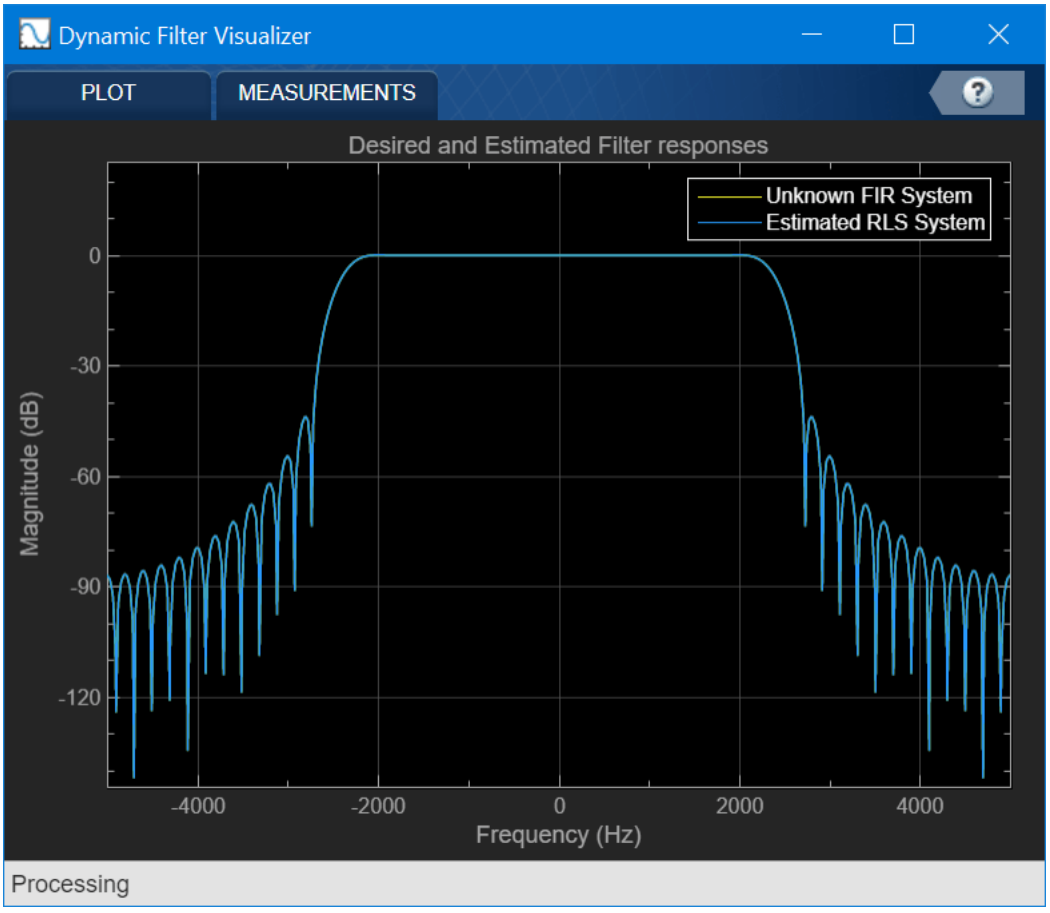
The function RLSFilterSystemIDExampleApp wraps around HelperRLSFilterSystemIdentificationSim and iteratively calls it, providing continuous adapting to the unidentified FIR system. Using `dsp.DynamicFilterVisualizer` the application also plots the following:

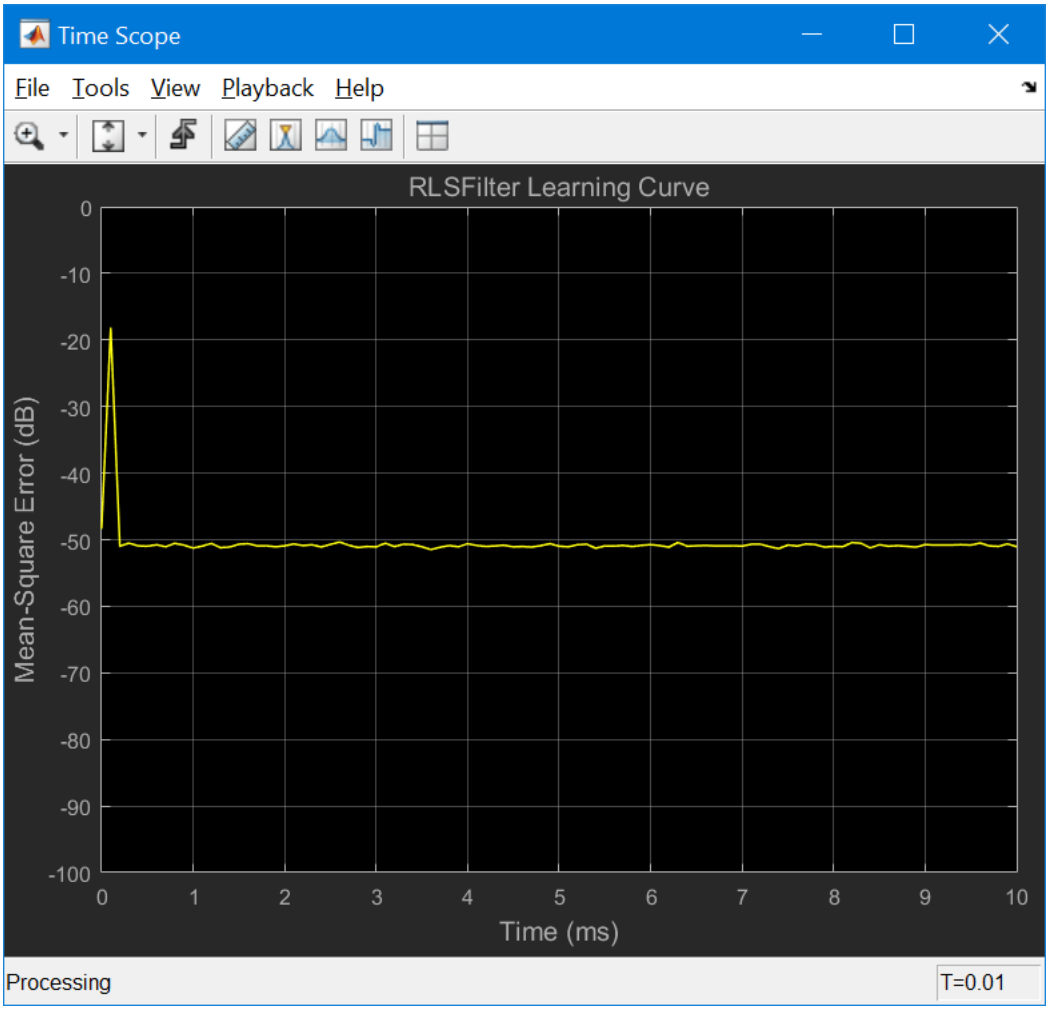
1. The desired versus estimated frequency transfer functions.
2. The learning curve of the RLS filter.

Plotting occurs when the 'plotResults' input to the function is 'true'.

Execute RLSFilterSystemIDExampleApp to run the simulation and plot the results on scopes. Note that the simulation runs for as long as the user does not explicitly stop it.

The plots below are the output of running the above simulation for 100 time-steps:

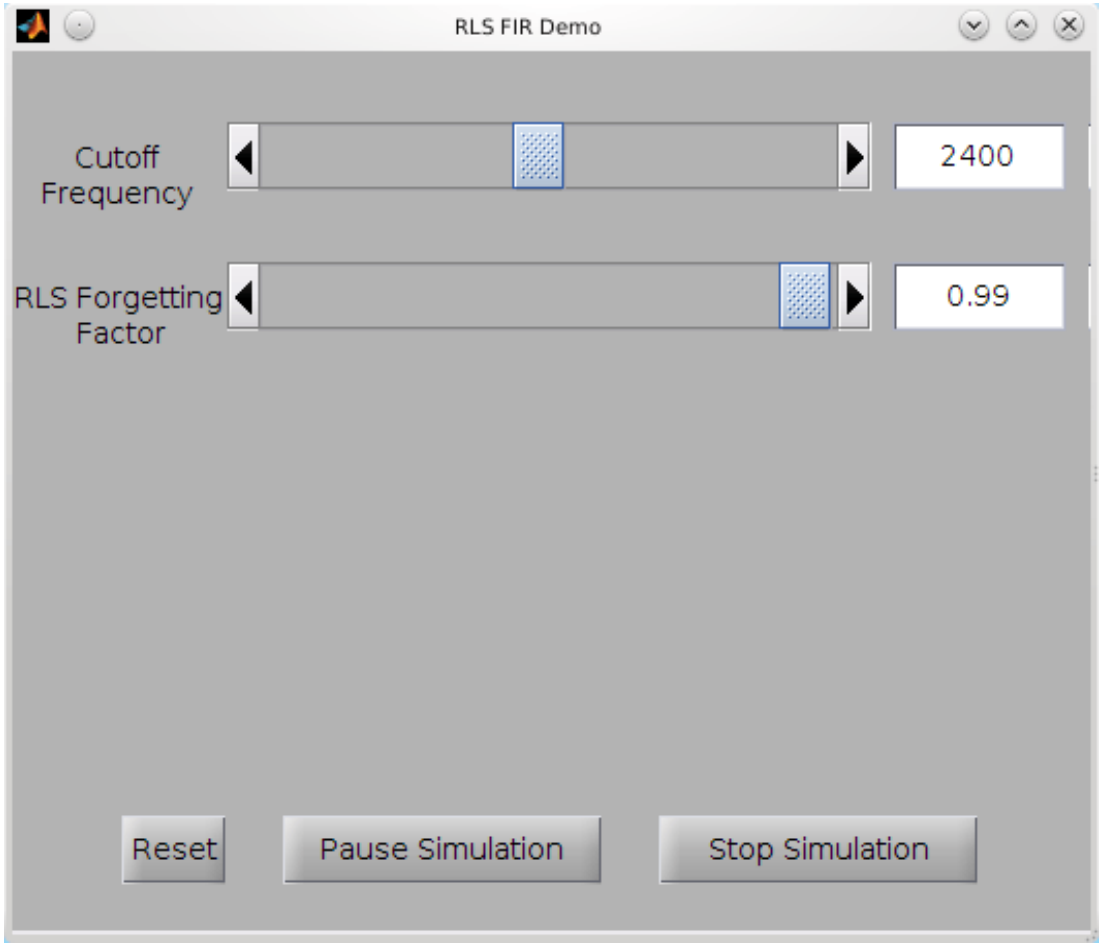




The fast convergence of the RLS filter towards the FIR filter can be seen through the above plots.

RLSFilterSystemIDExampleApp launches a User Interface (UI) designed to interact with the simulation. The UI allows you to tune parameters and the results are reflected in the simulation instantly. For example, moving the slider for the 'Cutoff Frequency' to the right while the simulation is running, increases the FIR filter's cutoff frequency. Similarly, moving the slider for the 'RLS Forgetting Factor' tunes the forgetting factor of the RLS filter. The plots reflects your changes as you tune these parameters. For more information on the UI, please refer to HelperCreateParamTuningUI.

There are also two buttons on the UI - the 'Reset' button resets the states of the RLS and FIR filters to their initial values, and 'Stop simulation' ends the simulation. If you tune the RLS filter's forgetting factor to a value that is too low, you will notice that the RLS filter fails to converge to the desired solution, as expected. You can restore convergence by first increasing the forgetting factor to an acceptable value, and then clicking the 'Reset' button. Use the UI to control either the simulation or, optionally, a MEX-file (or standalone executable) generated from the simulation code as detailed below. If you have a MIDI controller, it is possible to synchronize it with the UI. You can do this by choosing a MIDI control in the dialog that is opened when you right-click on the sliders or buttons and select "Synchronize" from the context menu. The chosen MIDI control then works in accordance with the slider/button so that operating one control is tracked by the other.



Generating the MEX-File

MATLAB Coder can be used to generate C code for the function HelperRLSFilterSystemIdentificationSim as well. In order to generate a MEX-file for your platform, execute the following:

```
currDir = pwd;  % Store the current directory address
mexDir   = [tempdir 'RLSFilterSystemIdentificationExampleMEXDir']; % Name of
% temporary directory
if ~exist(mexDir,'dir')
    mkdir(mexDir);       % Create temporary directory
end
cd(mexDir);          % Change directory

ParamStruct = HelperRLSCodeGeneration();
```

```
Code generation successful: To view the report, open('codegen\mex\HelperRLSFilterSystemIdentifica
```

By calling the wrapper function RLSFilterSystemIDExampleApp with `'true'` as an argument, the generated MEX-file `HelperRLSFilterSystemIdentificationSimMEX` can be used instead of `HelperRLSFilterSystemIdentificationSim` for the simulation. In this scenario, the UI is still running inside the MATLAB environment, but the main processing algorithm is being performed by a MEX-file. Performance is improved in this mode without compromising the ability to tune parameters.

Click here to call `RLSFilterSystemIDExampleApp` with `'true'` as argument to use the MEX-file for simulation. Again, the simulation runs till the user explicitly stops it from the UI.

Simulation Versus MEX Speed Comparison

Creating MEX-Files often helps achieve faster run-times for simulations. In order to measure the performance improvement, let's first time the execution of the algorithm in MATLAB without any plotting:

```
clear HelperRLSFilterSystemIdentificationSim
disp('Running the MATLAB code...')
tic
nTimeSteps = 100;
for ind = 1:nTimeSteps
     HelperRLSFilterSystemIdentificationSim(ParamStruct);
end
tMATLAB = toc;
```

```
Running the MATLAB code...
```

Now let's time the run of the corresponding MEX-file and display the results:

```
clear HelperRLSFilterSystemIdentificationSim
disp('Running the MEX-File...')
tic
for ind = 1:nTimeSteps
    HelperRLSFilterSystemIdentificationSimMEX(ParamStruct);
end
tMEX = toc;

disp('RESULTS:')
disp(['Time taken to run the MATLAB System object: ', num2str(tMATLAB),...
     ' seconds']);
disp(['Time taken to run the MEX-File: ', num2str(tMEX), ' seconds']);
disp(['Speed-up by a factor of ', num2str(tMATLAB/tMEX),...
    ' is achieved by creating the MEX-File']);
```

```
Running the MEX-File...
RESULTS:
Time taken to run the MATLAB System object: 7.5625 seconds
Time taken to run the MEX-File: 0.98745 seconds
Speed-up by a factor of 7.6586 is achieved by creating the MEX-File
```

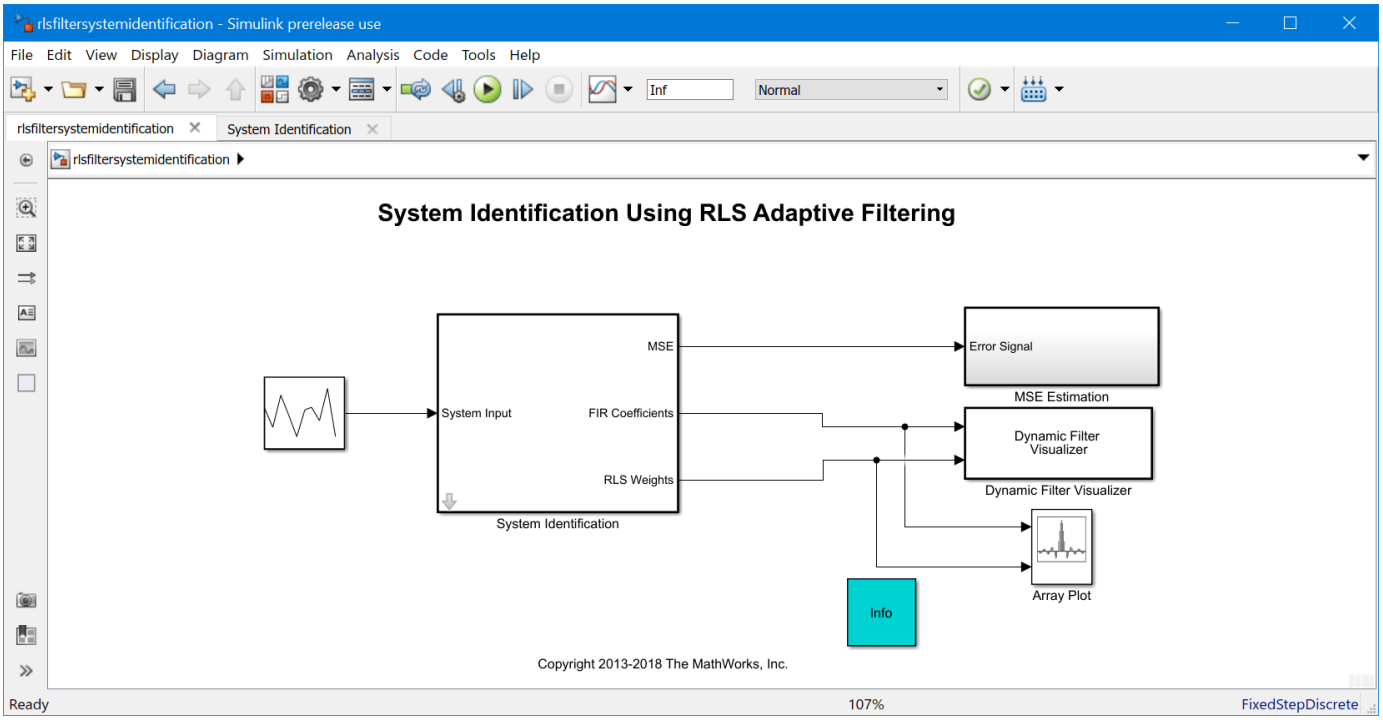
Clean up Generated Files

The temporary directory previously created can be deleted through:

```
cd(currDir);
clear HelperRLSFilterSystemIdentificationSimMEX;
rmdir(mexDir, 's');
```

Simulink Version

rlsfiltersystemidentification is a Simulink model that implements the RLS System identification example highlighted in the previous sections.



In this model, the lowpass FIR filter is modeled using the Variable Bandwidth FIR Filter block. Magnitude response visualization is performed using `dsp.DynamicFilterVisualizer`.

Double-click the System Identification subsystem to launch the mask designed to interact with the Simulink model. You can tune the cutoff frequency of the FIR filter and the forgetting factor of the RLS filter.

The model generates code when it is simulated. Therefore, it must be executed from a folder with write permissions.

Acoustic Noise Cancellation (LMS)

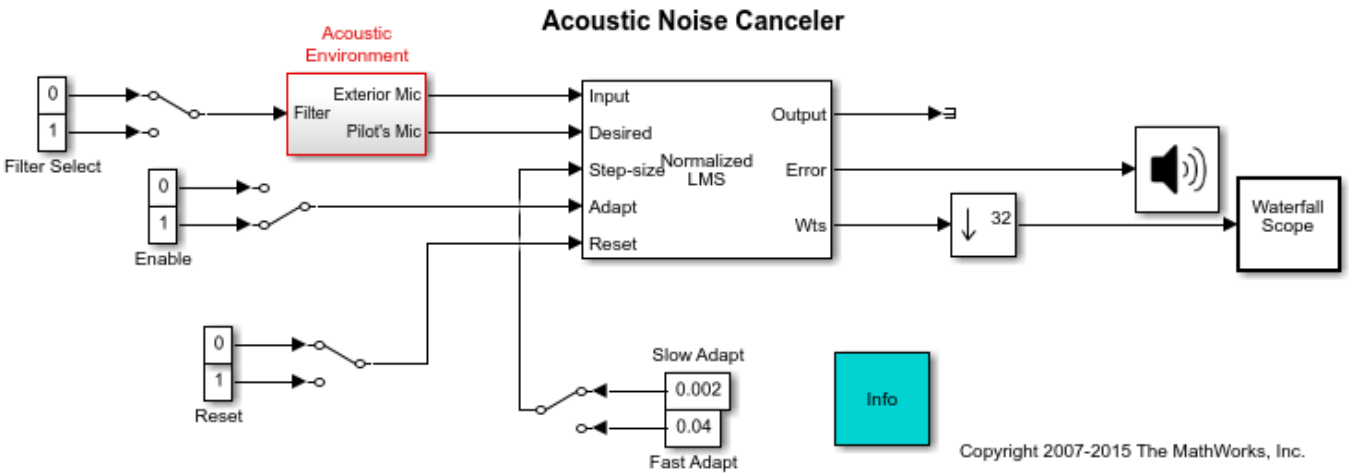
This example shows how to use the Least Mean Square (LMS) algorithm to subtract noise from an input signal. The LMS adaptive filter uses the reference signal on the `Input` port and the desired signal on the `Desired` port to automatically match the filter response. As it converges to the correct filter model, the filtered noise is subtracted and the error signal should contain only the original signal.

Exploring the Example

In the model, the signal output at the upper port of the Acoustic Environment subsystem is white noise. The signal output at the lower port is composed of colored noise and a signal from a .wav file. This example model uses an adaptive filter to remove the noise from the signal output at the lower port. When you run the simulation, you hear both noise and a person playing the drums. Over time, the adaptive filter in the model filters out the noise so you only hear the drums.

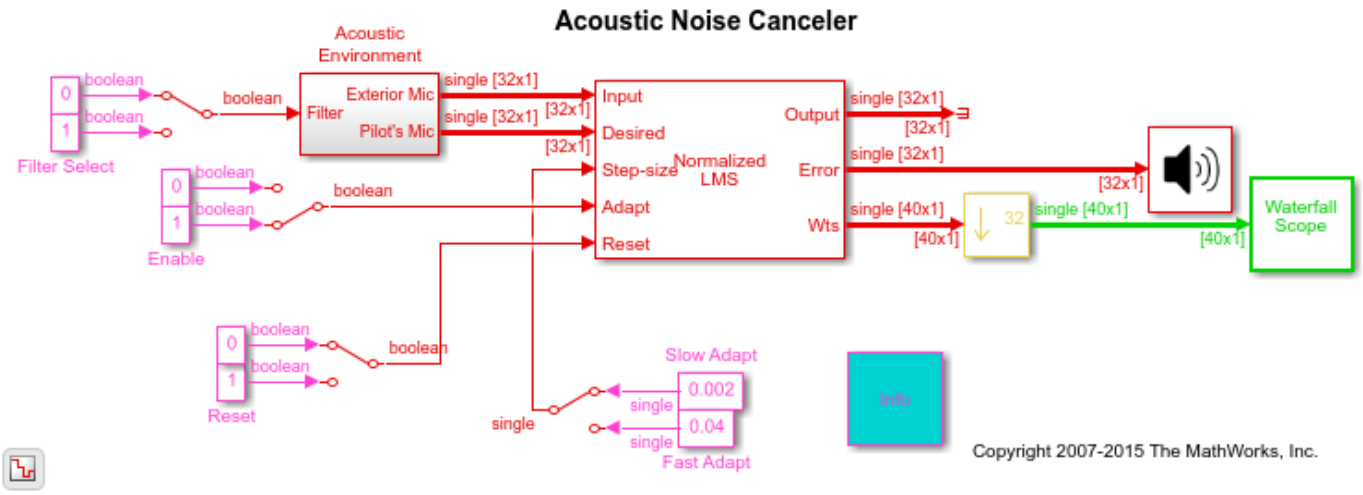
Acoustic Noise Canceler Model

The all-platform floating-point version of the model is shown below.



Utilizing Your Audio Device

By running this model, we can listen to the audio signal in real time (while running the simulation). The stop time is set to infinity. This allows us to interact with the model while it is running. For example, we can change the filter or alternate from slow adaptation to fast adaptation (and vice versa), and get a sense of the real-time audio processing behavior under these conditions.



Color Codes of the Blocks

Notice the colors of the blocks in the model. These are sample time colors that indicate how fast a block executes. Here, the fastest discrete sample time (e.g., the 8 kHz audio signal processing portion) is red, and the second fastest discrete sample time is green. You can see that the color changes from red to green after down-sampling by 32 (in the Downsample block before the Waterfall Scope block). Further information on displaying sample time colors can be found in the Simulink® documentation.

Waterfall Scope

The Waterfall window displays the behavior of the adaptive filter's filter coefficients. It displays multiple vectors of data at one time. These vectors represent the values of the filter's coefficients of a normalized LMS adaptive filter, and are the input data at consecutive sample times. The data is displayed in a three-dimensional axis in the Waterfall window. By default, the x-axis represents amplitude, the y-axis represents samples, and the z-axis represents time. The Waterfall window has toolbar buttons that enable you to zoom in on displayed data, suspend data capture, freeze the scope's display, save the scope position, and export data to the workspace.

Acoustic Environment Subsystem

You can see the details of the Acoustic Environment subsystem by double clicking on that block. 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 the signal coming from a .wav-file to produce the signal sent to the Pilot's Mic output port.

References

Haykin, S., **Adaptive Filter Theory**, 3rd Ed., Prentice-Hall, 1996.

Available Example Versions

Floating-point version: dspanc

Fixed-point version: dspanc_fixpt

Adaptive Filter Convergence

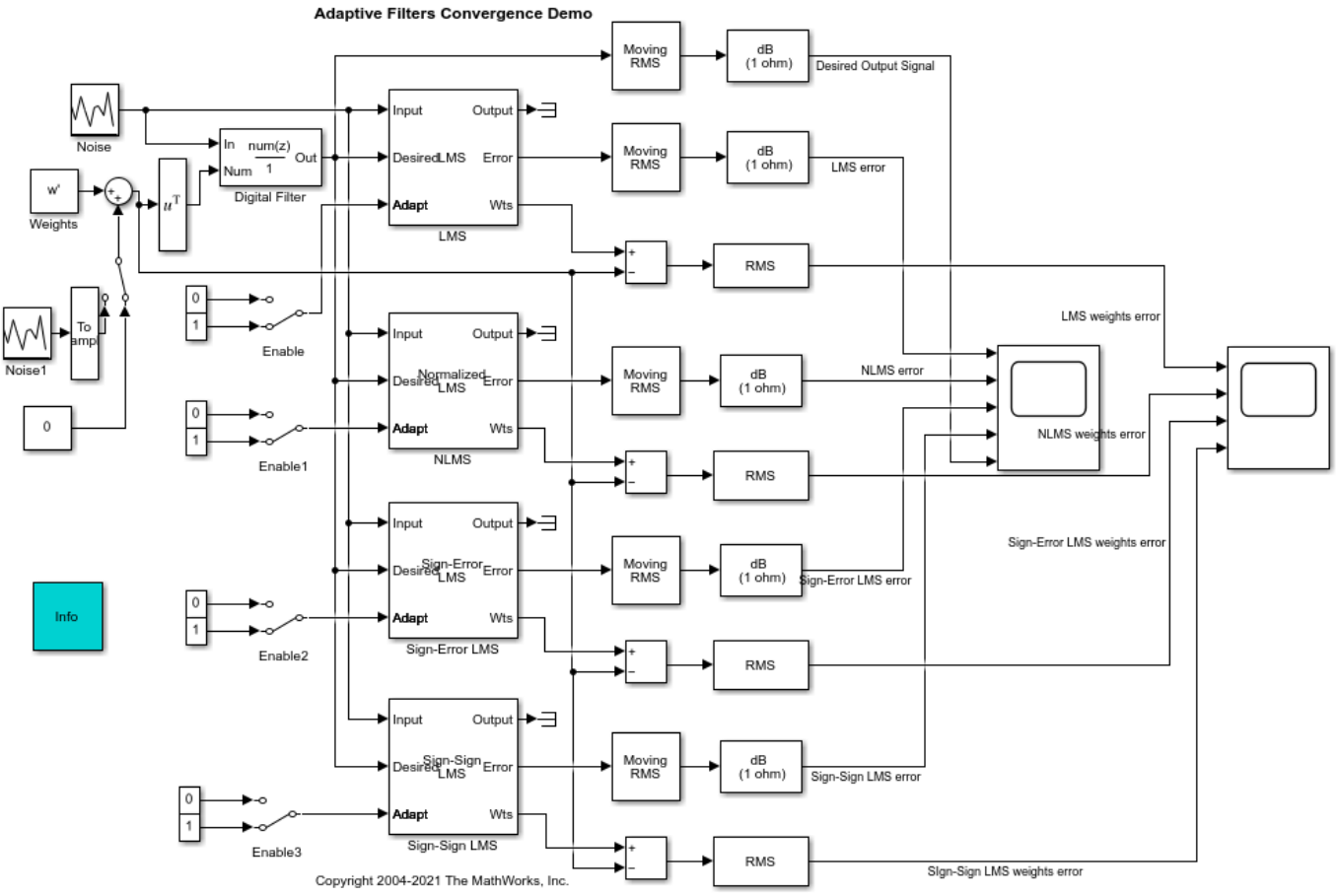
This example compares the rate of convergence for adaptive filters using different LMS algorithms.

Introduction

An adaptive filter adapts its filter coefficients so that its output matches the output of a desired unknown system. A major application of adaptive filters is filter identification, where the reference signal is filtered by an unknown convolution filter, as illustrated here. The same signal is passed through the adaptive filter and the unknown system, and the adaptive filter tries to adjust its coefficients such that the error between the two outputs is minimal. Adaptive filters typically have large error on their outputs initially and the error tends to go down with time, as the filter converges.

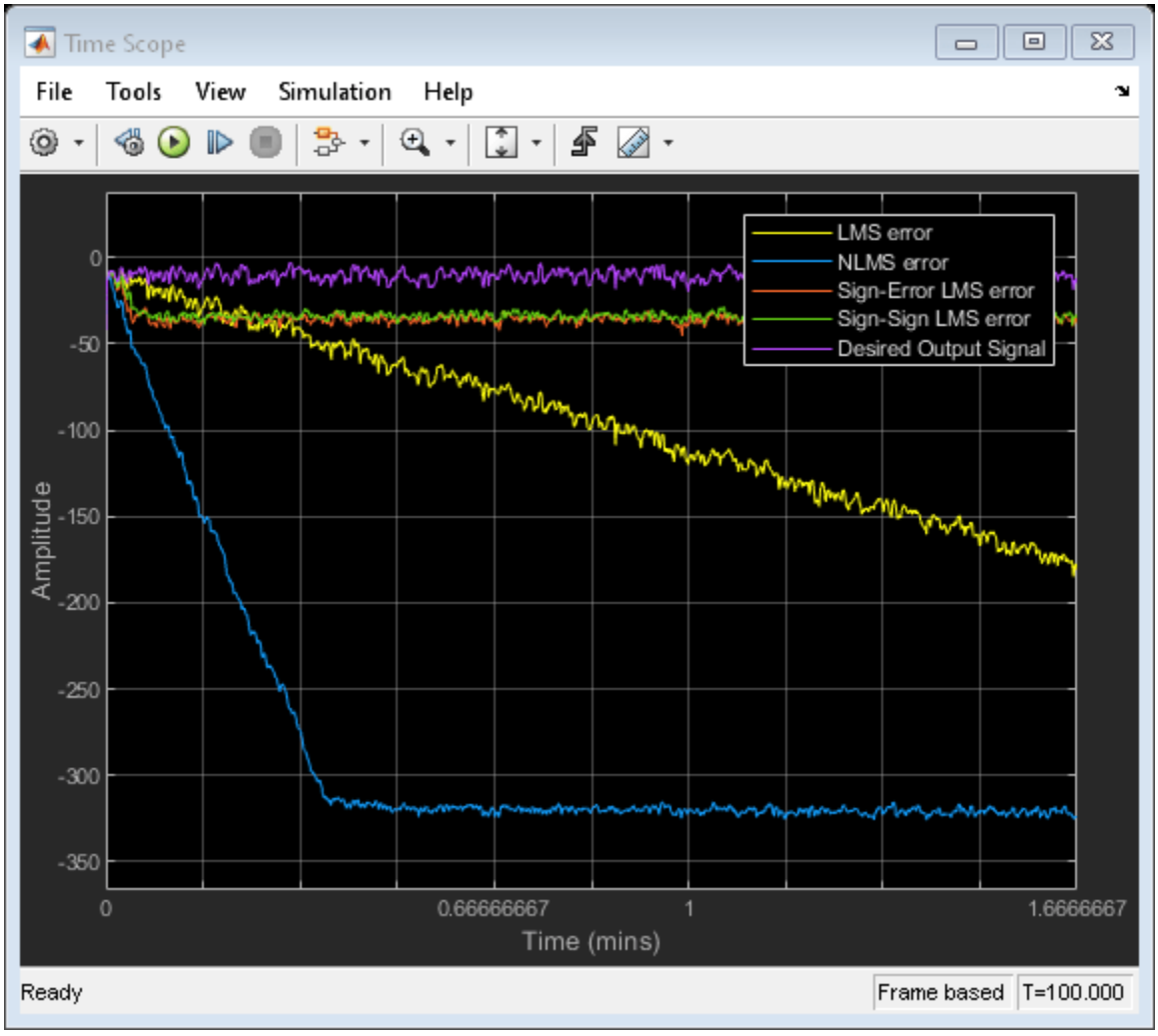
Adaptive filters using the Least Mean Squares (LMS) algorithm adjust their coefficients such that the error between the two outputs is minimal in mean square sense. This example considers 4 different variants of the the LMS algorithm and compares the rate of convergence for adaptive filters using these algorithms.

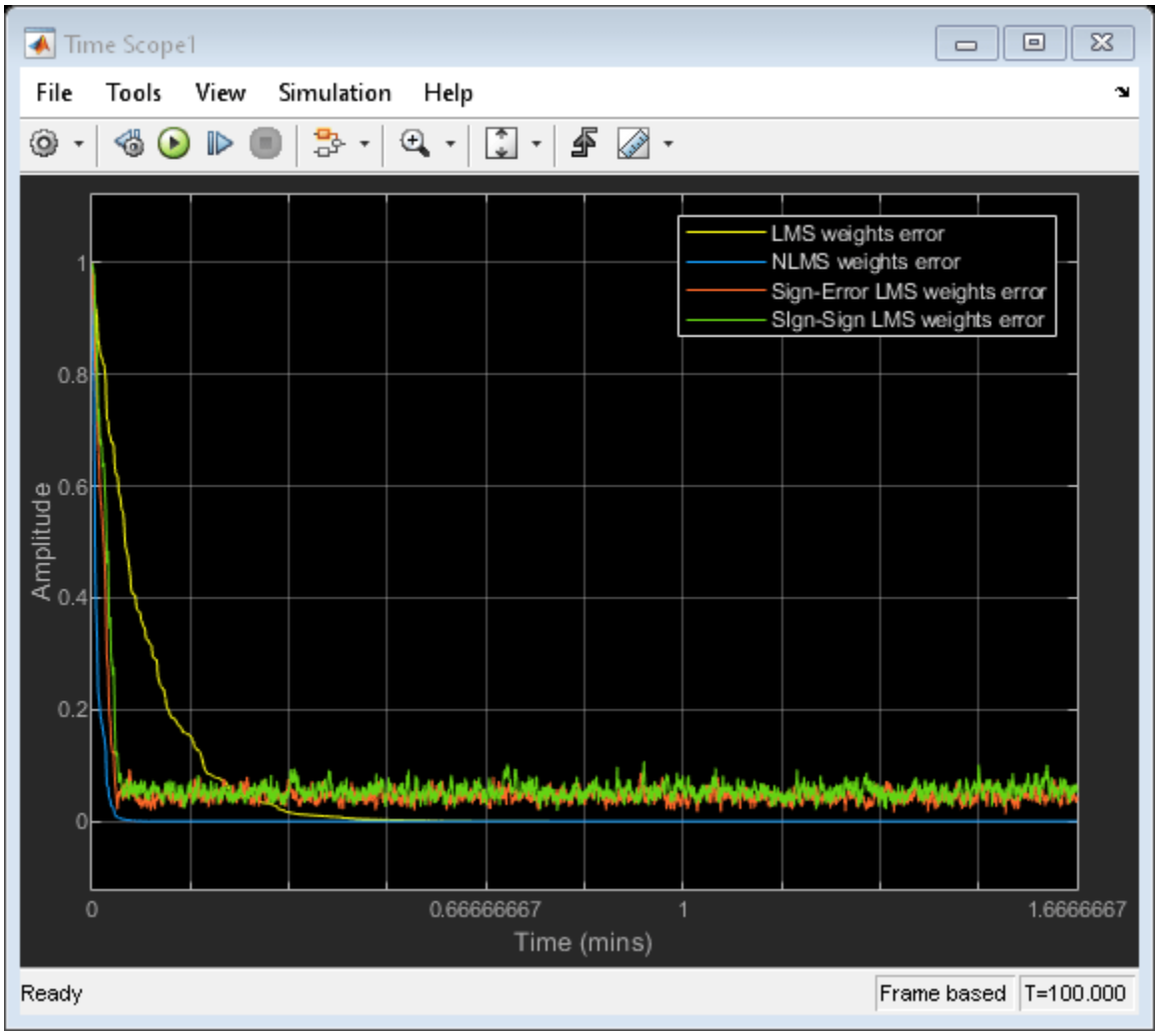
- LMS - Least Mean Square algorithm
- NLMS - Normalized LMS algorithm
- SELMS - Sign-Error LMS algorithm
- SSLMS - Sign-Sign LMS algorithm



Setup

Use a zero mean white Gaussian noise with a variance of 0.01 as the input signal to the filters. Choose an FIR filter of length 13 with all coefficients set to 1 as the desired unknown filter. You can optionally add a Gaussian noise of variance 1e-5 to these weights by toggling the Manual Switch. Set the switch towards the Constant block of value 0 for now, so that the weights do not contain any noise. Set the initial weights of the adaptive filters to 0. Use a step size of 0.2 for the LMS, NLMS and Sign-Error LMS blocks. For the Sign-Sign LMS algorithm, choose a more suitable step size of 0.02. Each of the adaptive filters can be enabled or disabled separately. Use a Moving RMS block to compute the energy of the desired output and the error signals. Visualize these signals in dB scale on a Time Scope. To examine the convergence of the adaptive filter weights, subtract the adaptive filter weights from the desired filter weights and plot the error using another Time Scope.



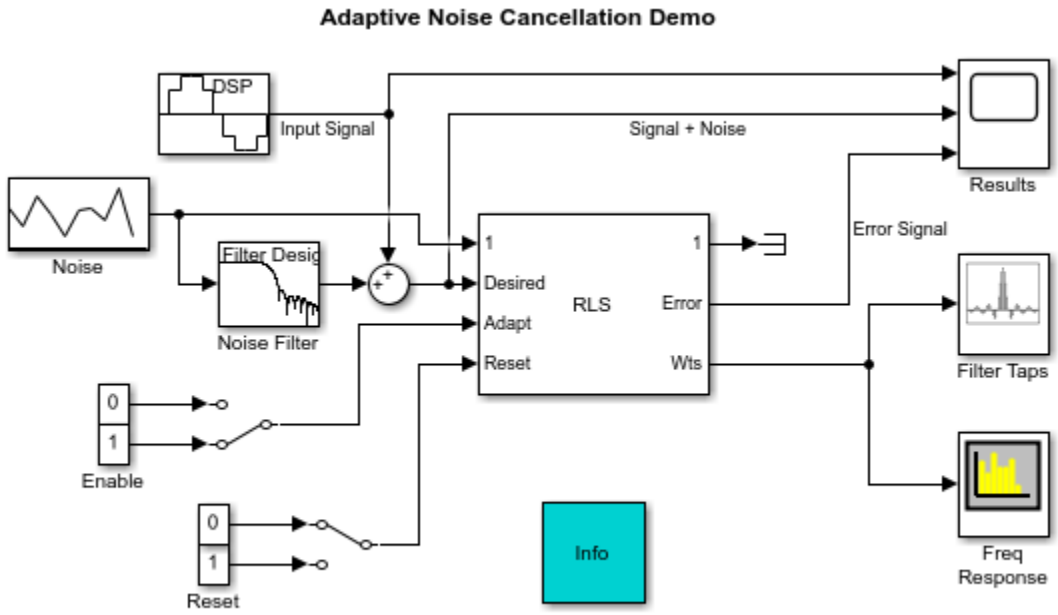


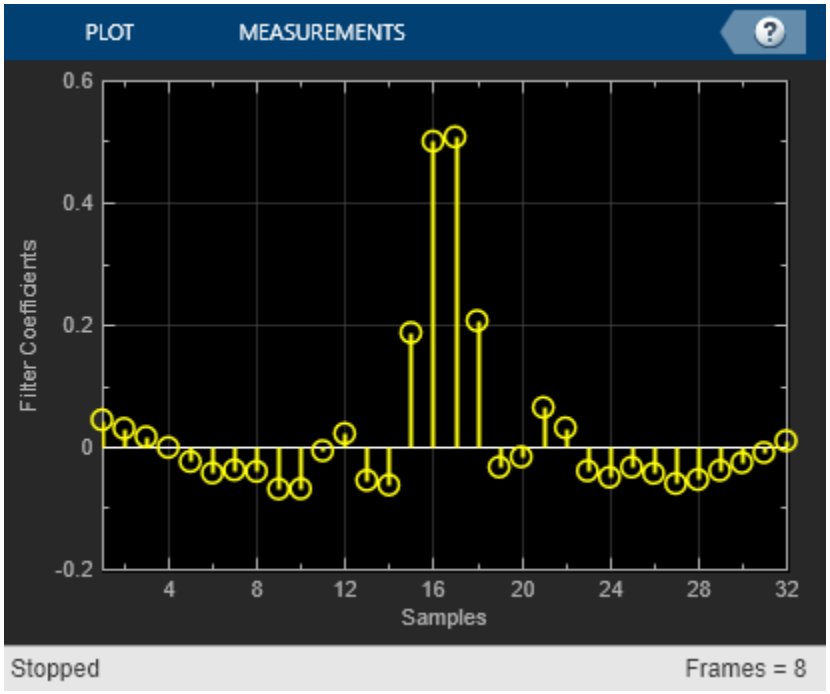
From the first plot, it can be seen that the error signal from the NLMS filter converges to zero much faster than the other variants, with a steady state error of more than 300 dB. The LMS filter is also able to match the output of the unknown filter closely with an SNR of more than 150 dB. From the second plot, the weights of the NLMS and LMS filters are also observed to converge to the desired filter weights with minimal final error between the weights.The Sign-Error LMS and Sign-Sign LMS filters appear to start converging faster than the LMS filter initially, but these filters have a larger steady state error in the filter output (SNR of approximately 20 dB) and filter weights.

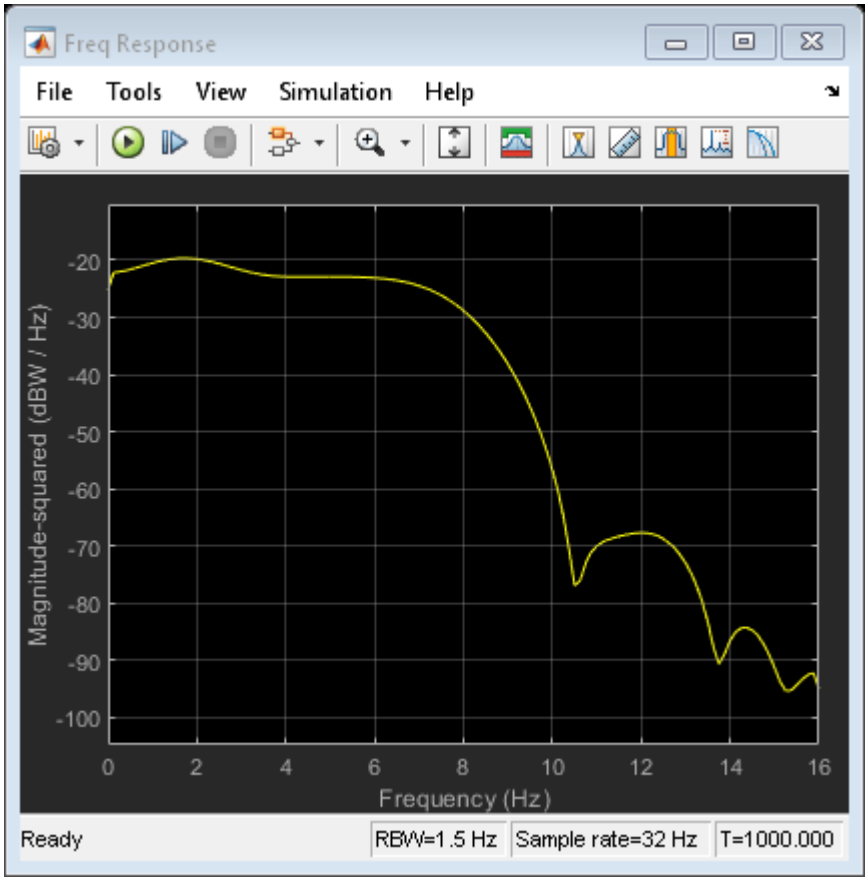
Noise Canceler (RLS)

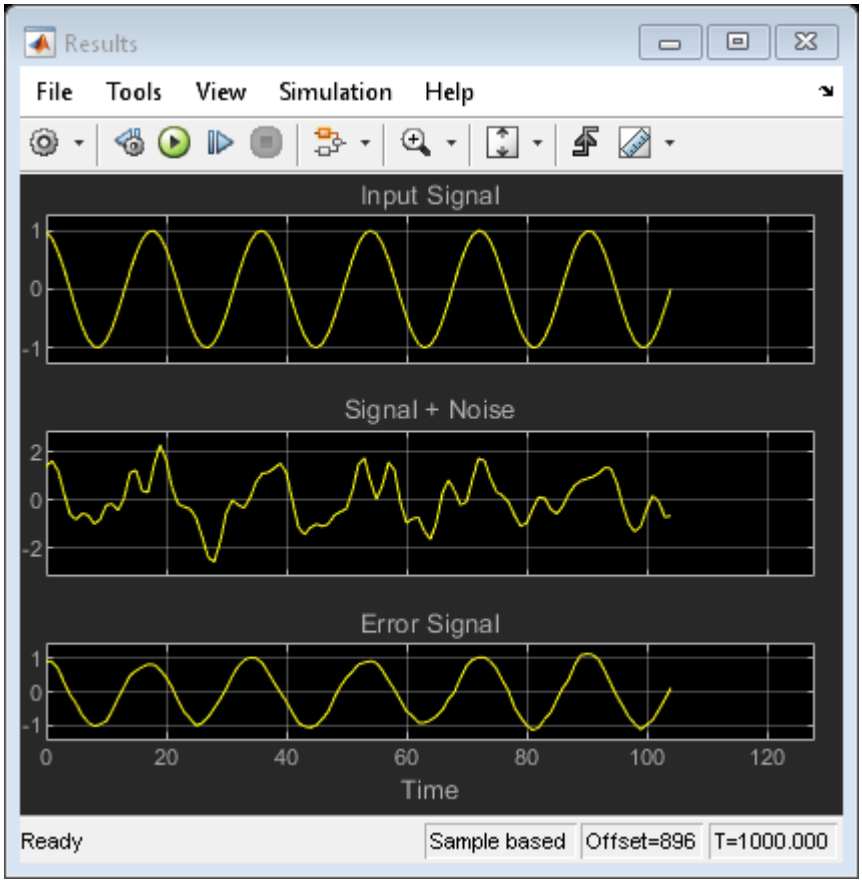
This example shows how to subtract noise from an input signal using the Recursive Least Squares (RLS) algorithm. The RLS adaptive filter uses the reference signal on the Input port and the desired signal on the Desired port to automatically match the filter response in the Noise Filter block. As it converges to the correct filter, the filtered noise should be completely subtracted from the "Signal +Noise" signal, and the "Error Signal" should contain only the original signal.

For details, see S. Haykin, **Adaptive Filter Theory**, 3rd Ed., Prentice-Hall 1996.




PLOT
MEASUREMENTS
0.6
0.4
0.2
0
-0.2
Filter Coefficients
4
8
12
16
20
24
28
32
Samples
Stopped
Frames = 8


Freq Response
File
Tools
View
Simulation
Help
-20
-30
-40
-50
-60
-70
-80
-90
-100
Magnitude-squared (dBW / Hz)
0
2
4
6
8
10
12
14
16
Frequency (Hz)
Ready
RBW=1.5 Hz
Sample rate=32 Hz
T=1000.000


Results
File
Tools
View
Simulation
Help
Input Signal
1
0
-1
Signal + Noise
2
0
-2
Error Signal
1
0
-1
0
20
40
60
80
100
120
Time
Ready
Sample based
Offset=896
T=1000.000

Time-Delay Channel Estimation Through Adaptive Filtering

This example shows how to adaptively estimate the time delay for a noisy input signal using the LMS adaptive FIR algorithm.

Assume a signal $s[n] = a[n] + w[n]$ where $w[n]$ is a white Gaussian process and $a[n]$ is deterministic. The signal is measured with an echo of M samples and attenuation α (both are unknown), resulting in the overall measurement:

$$x[n] = s[n] + \alpha s[n - M]$$

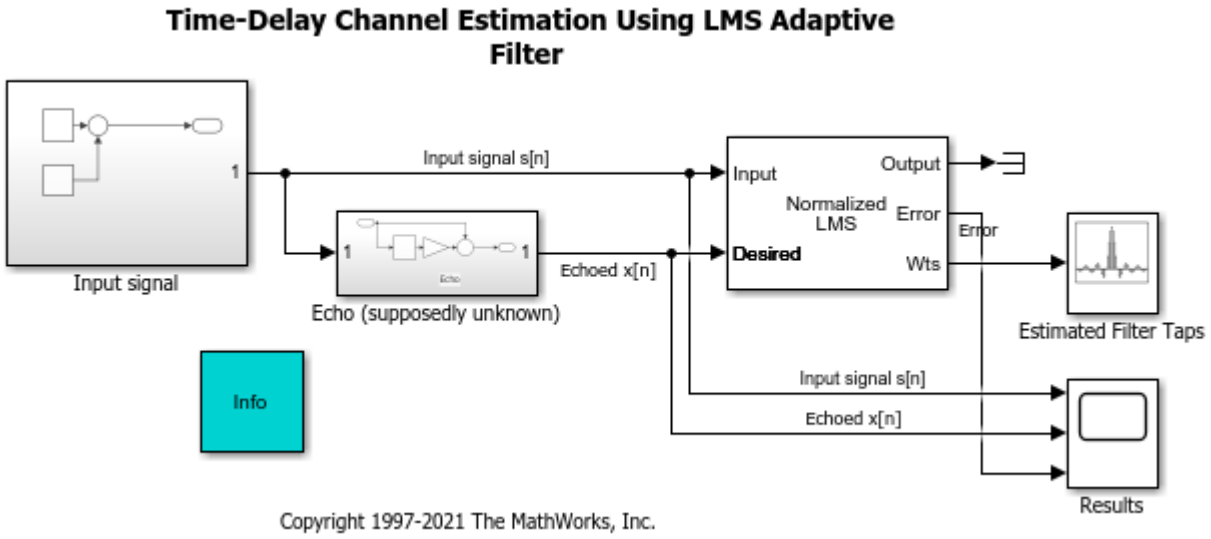
The goal is to estimate the delay M and the echo attenuation α. One can determine these parameters by solving the filter identification problem $x = h * s$ for h, combined with the prior $h[n] = \delta[n] + \alpha s[n - M]$. Provided that the filter h can be identified from the measurements signal x and the original signal s, one can derive α and M

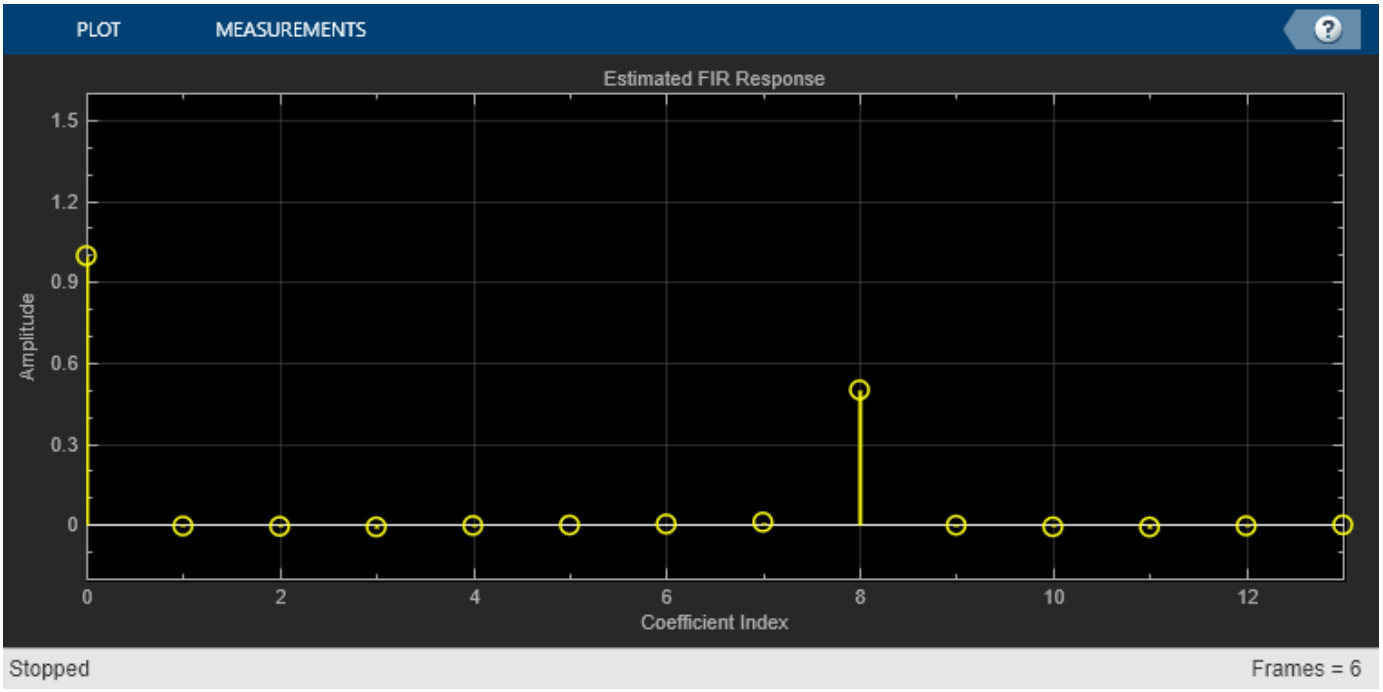
Such a filter identification problem can be posed in terms of adaptive LTI filtering. The reference signal is $d[n] = x[n]$, the input feed is $s[n]$, and the adaptive filter is w. Clearly, if the adaptation process concludes with $w \to h$ then the error signal $x - w * s = (h - w) * x$ vanishes.

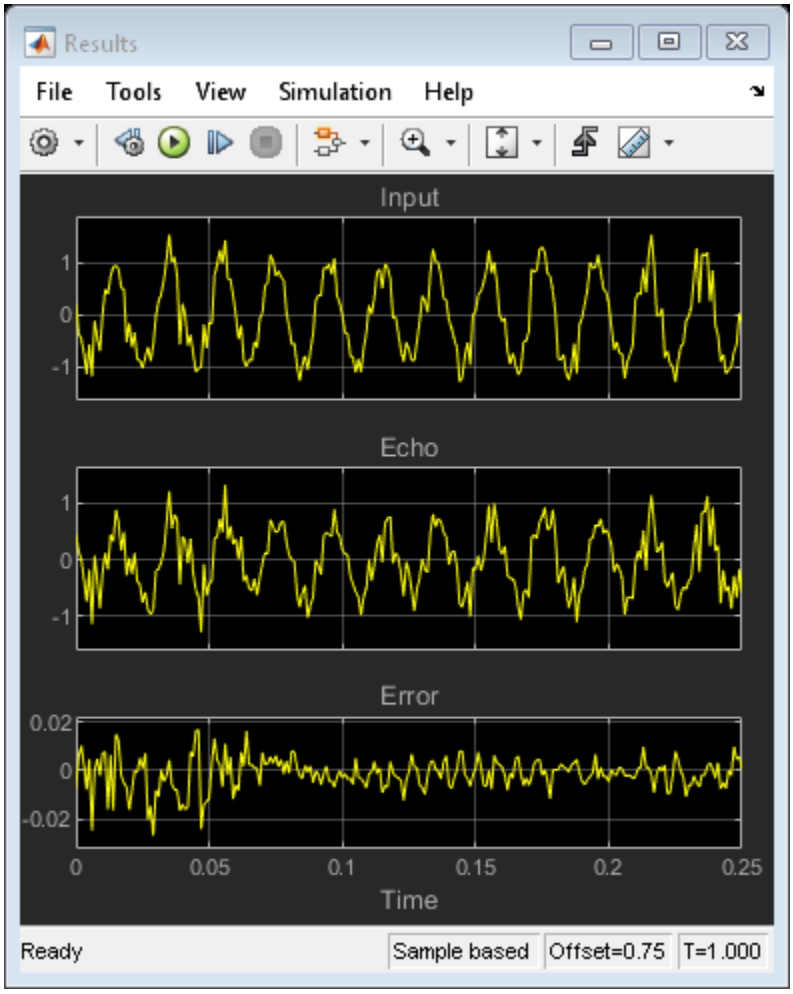
There are numerous adaptive filtering algorithms. For this paricular problem setup and signal model, the normalized LMS algorithm is suitable, and is available in the LMS Filter block.

Run the simulation. The peaks in the filter taps vector indicates the time-delay estimate. In this case $M = 8$ and $\alpha = \frac{1}{2}$.

For details, see S. Haykin, **Adaptive Filter Theory**, 3rd Ed., Prentice-Hall 1996.



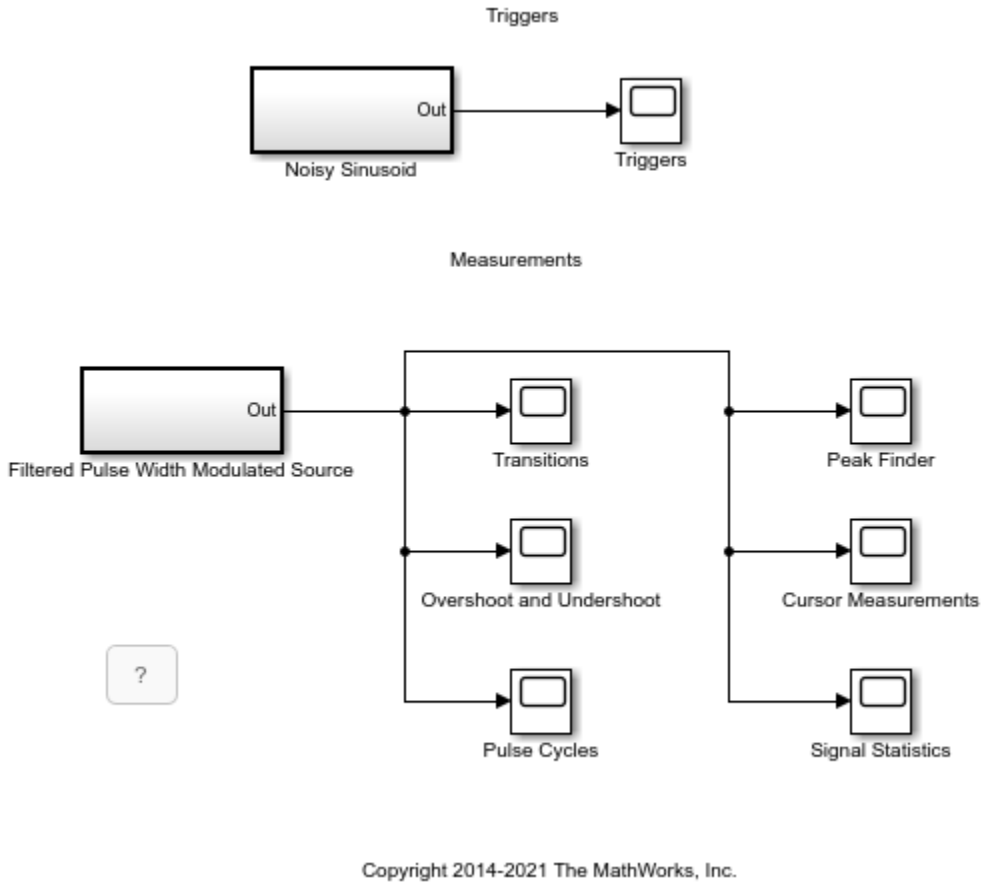

PLOT
MEASUREMENTS
Estimated FIR Response
1.5
1.2
0.9
0.6
0.3
0
Amplitude
0
2
4
6
8
10
12
Coefficient Index
Stopped
Frames = 6


Results
File
Tools
View
Simulation
Help
Input
1
0
-1
Echo
1
0
-1
Error
0.02
0
-0.02
0
0.05
0.1
0.15
0.2
0.25
Time
Ready
Sample based
Offset=0.75
T=1.000

Time Scope Measurements

This example shows how to measure performance characteristics of a pulse width modulated sinusoid. The example contains a model which you can modify to view the effects of parameter changes on rise time, fall time, overshoot, undershoot, pulse width, pulse period, and duty cycle measurements. The example also shows an example of a rising edge trigger and is set up to perform basic statistical operations (mean, median, RMS, maximum, minimum) and measure the frequency and period of the pulse period via cursors and peak finding.

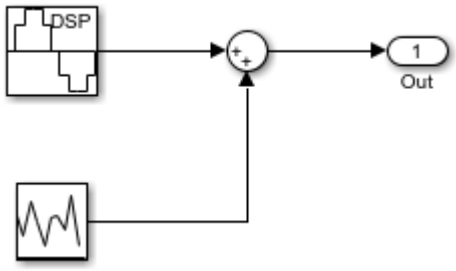
The example model contains several measurements and their corresponding setups.



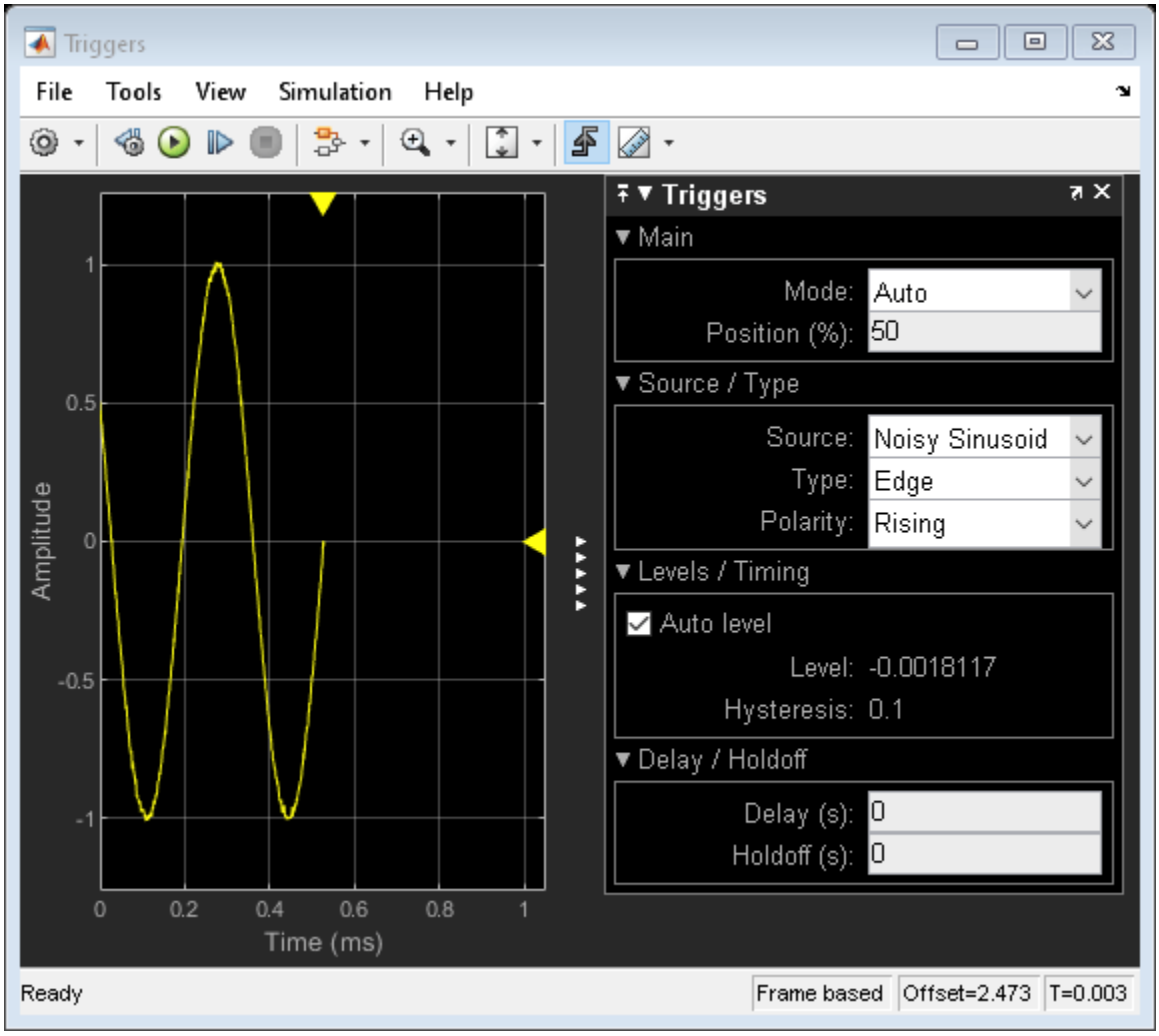
Copyright 2014-2021 The MathWorks, Inc.

Triggers

The first section shows how to use a trigger to stabilize a noisy sinusoid in the display. You can see how the sinusoid is constructed by double-clicking on the Noisy Sinusoid block.



The sinusoidal signal is fed into a Time Scope block with triggers enabled.



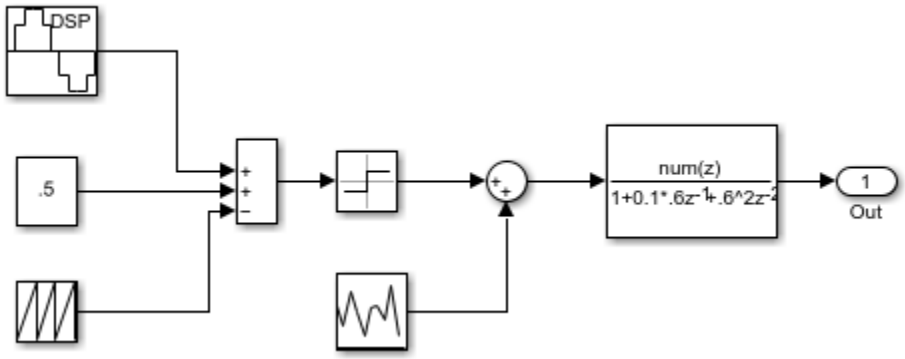
You can experiment with the trigger position by dragging the markers around the display. You can trigger upon rising or falling edges. This example includes 0.1 V of hysteresis to help stabilize the sinusoid in the presence of noise. The hysteresis ensures that the signal traverses at least 0.1 V below the trigger level before registering a positive-going transition.

If you close the triggers, you will see that the sinusoid no longer stays fixed in the screen. You can bring the triggers back by clicking on the trigger icon.

Measurements of a Pulse Width Modulated Source

In this example, a pulse width modulated source is connected to several time scopes that contain measurements.

You can view the source by clicking on it:



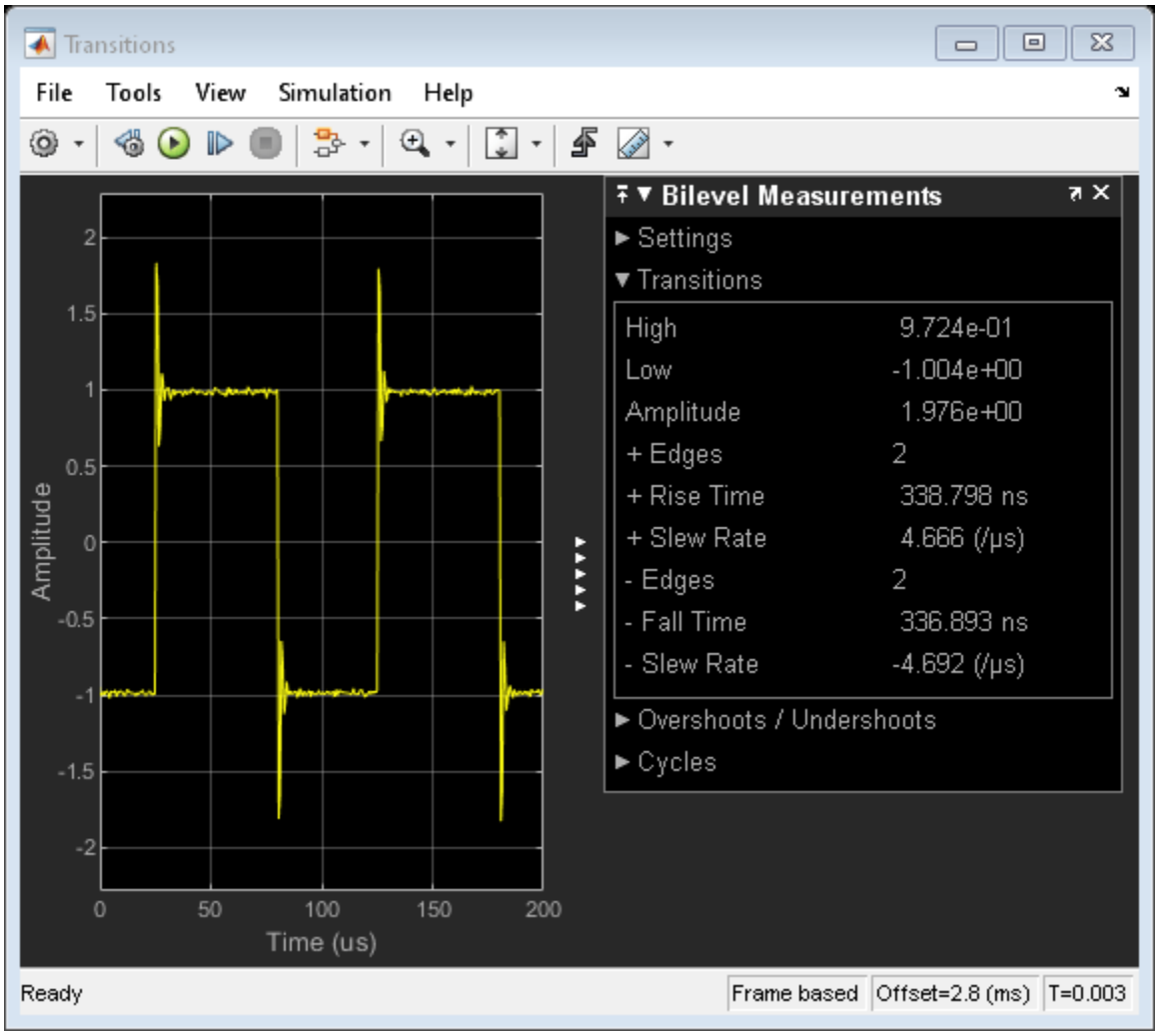
The model constructs sinusoidal pulse width modulation by applying a bias to the desired sine wave and subsequently subtracting a periodic sawtooth wave. The resulting waveform is then fed into a comparator to form the shape of the pulse. Noise is then added to the signal and then sent to a filter with an underdamped response.

You can modify the amount of additive noise on the input by clicking on the Random Source and modifying the variance of the Gaussian distribution.

You can similarly modify the response of the filter by changing its coefficients.

Transitions

You can view some basic information about the rising and falling transitions of the waveform by viewing the Transitions panel of the Bilevel Measurements dialog.



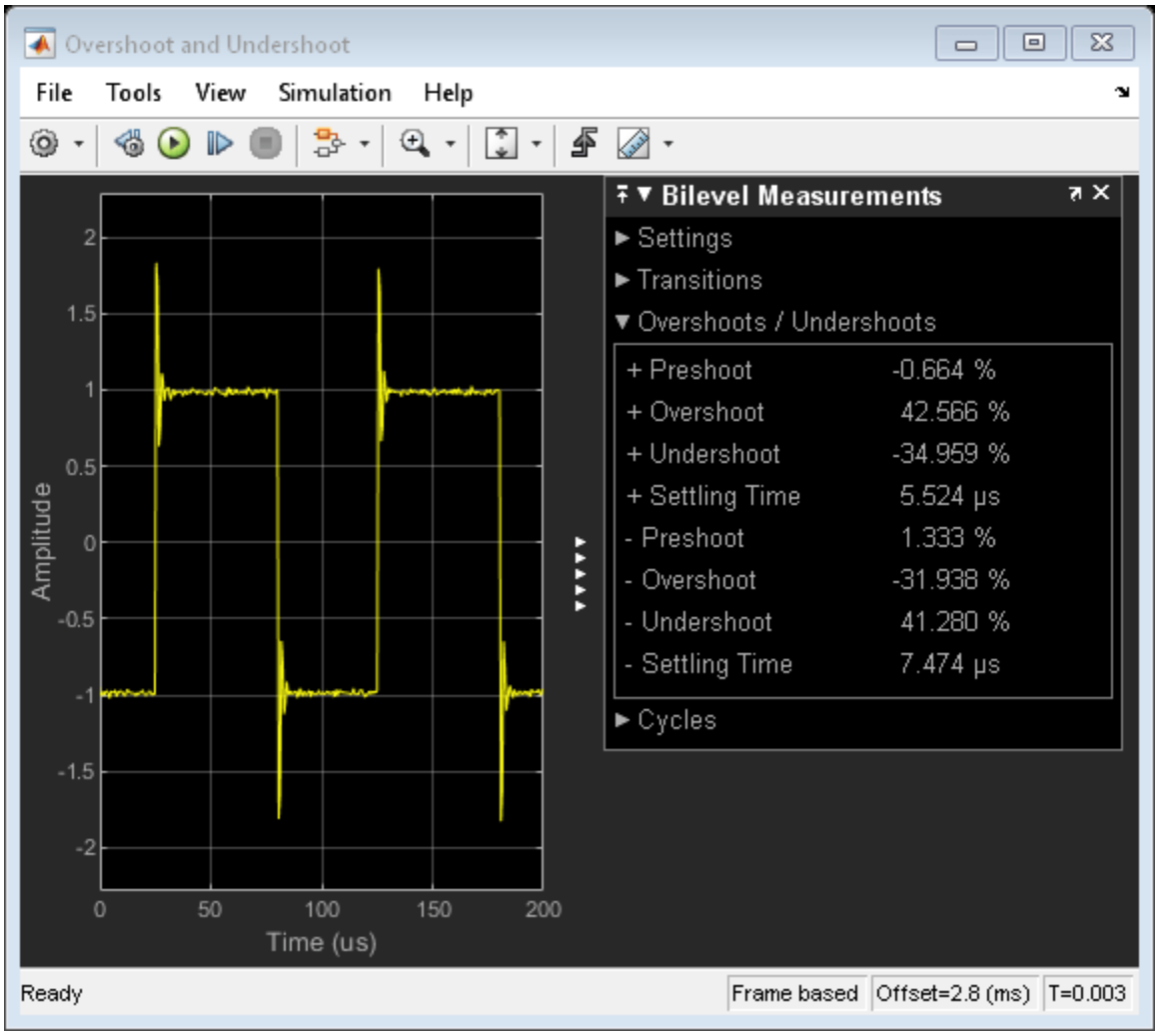
Viewing the results, you can see that the pulse has a high voltage level of +1 V and a low voltage level of -1 V.

The example above captures two rising (positive) edges and two falling (negative) edges with rise times and fall times of around 340 ns. If you zoom into just one edge of the waveform you can see the measurements for just that edge.

Note that the edges of the pulses are fairly steep, having a slew rate of about 4 V/us. An underdampened filter was used to achieve this rate. Changing the filter to be overdamped would decrease the rate at which the edge of each pulse could transition between pulse levels. The output of an underdampened filter exhibits significant ringing immediately after changing between low and high states. To quantify this ringing behavior, you can use the measurements in the Overshoots / Undershoots panel.

Overshoot and Undershoot

The Bilevel Measurements dialog also contains measurements that relate to an under-damped environment. You can view the transition aberrations by opening the Overshoots / Undershoots panel:

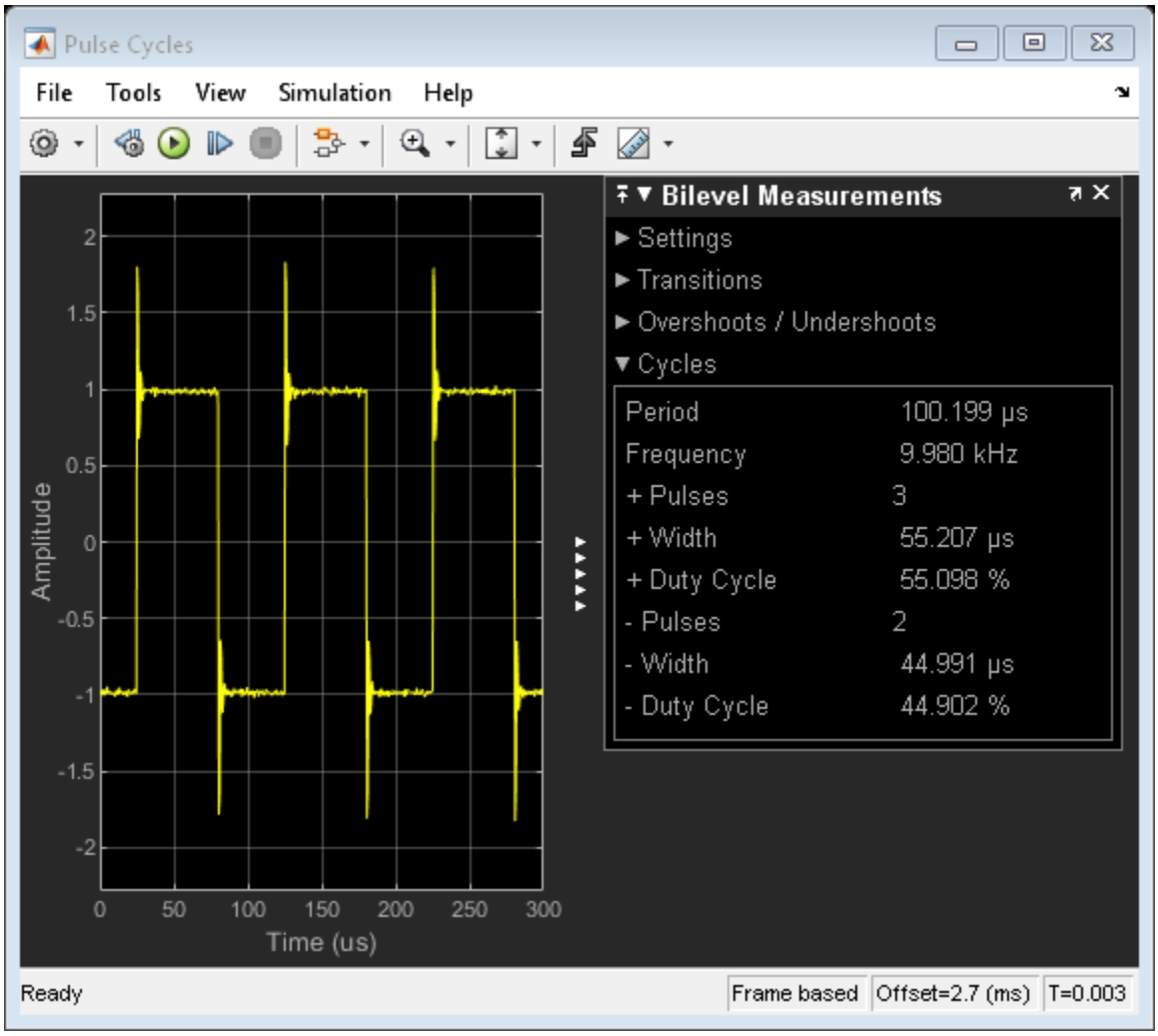


The average overshoot of the rising edges is about 42%. The undershoot is 34%. Large overshoots can sometimes damage logic devices which are designed to accept only a small voltage range. Large undershoots can cause devices to detect incorrect logic states. In this example the transitions settle on average within 7.3 microseconds.

You can reduce the amount of ringing by experimenting with the filter coefficients at the output of the modulated source.

Pulse Cycles

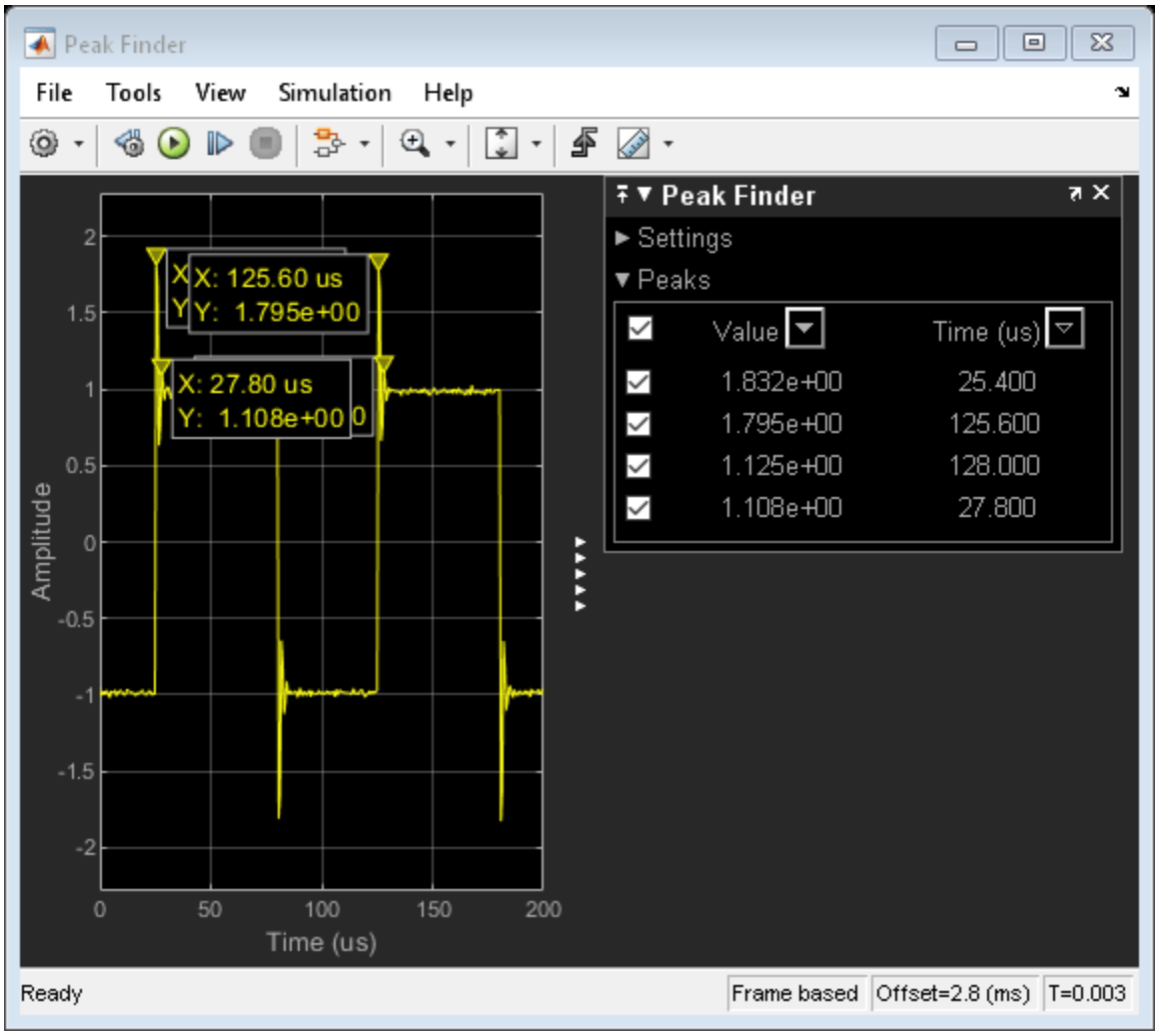
You can also view how the pulse width and duty cycle vary as functions of time by opening the Cycles panel in the Bilevel Measurements dialog:



This example shows three positive-polarity pulses but only two negative-polarity pulses. The pulse frequency is 10 kHz. You can observe the encoded sinusoid by watching how the duty cycle and pulse width change over time.

Peak Finder

Alternatively, you can measure the amplitudes and the times of significant peaks by invoking the Peak Finder dialog.

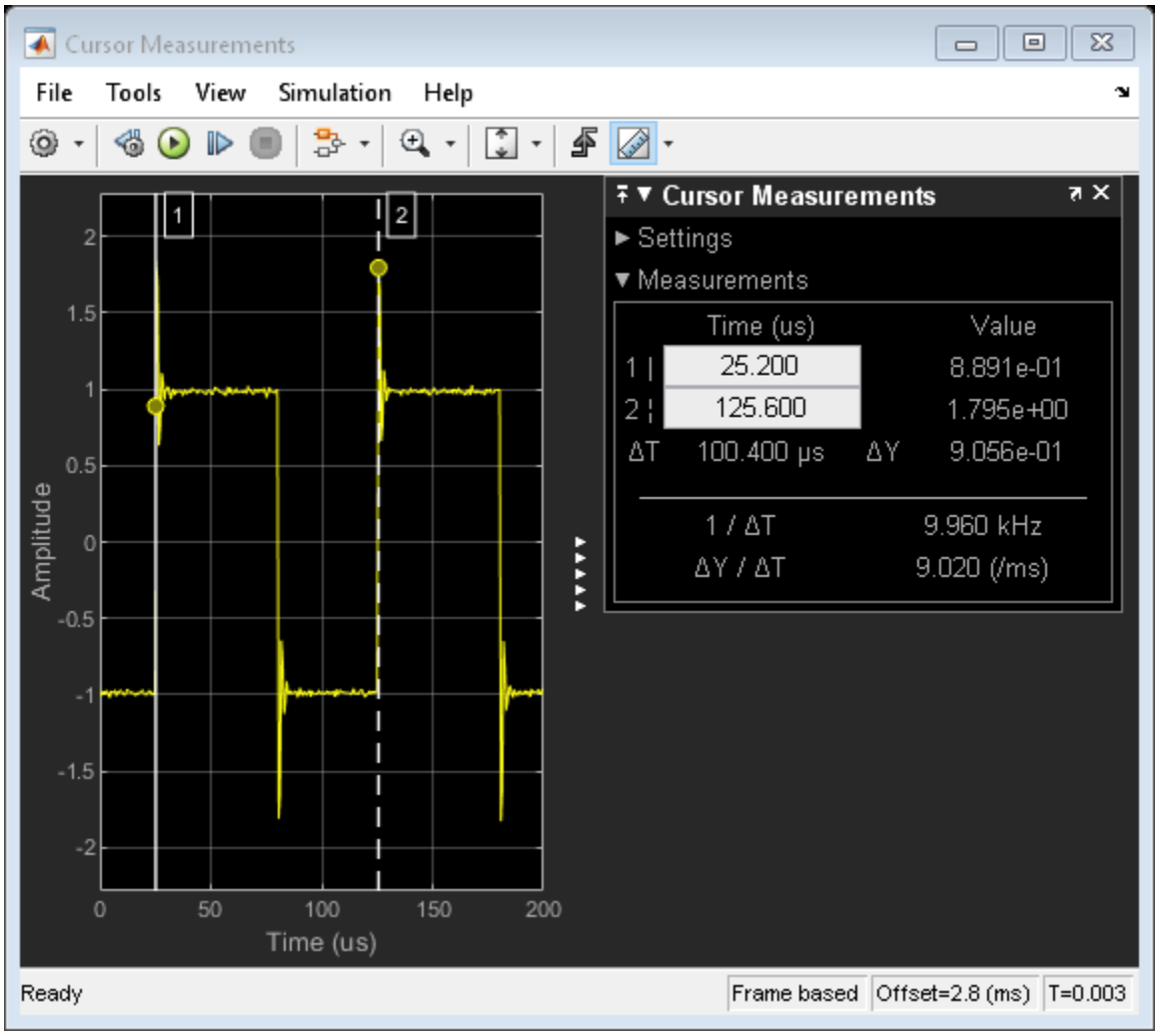


The voltage at the tip of each overshoot is about 1.8 V and the next largest ringing component of the first pulse is at 1.14 V.

Expand the settings panel to change the number of peaks shown. You can also filter based on height or distance between peaks. You can also change the text annotation shown in the display.

Cursor Measurements

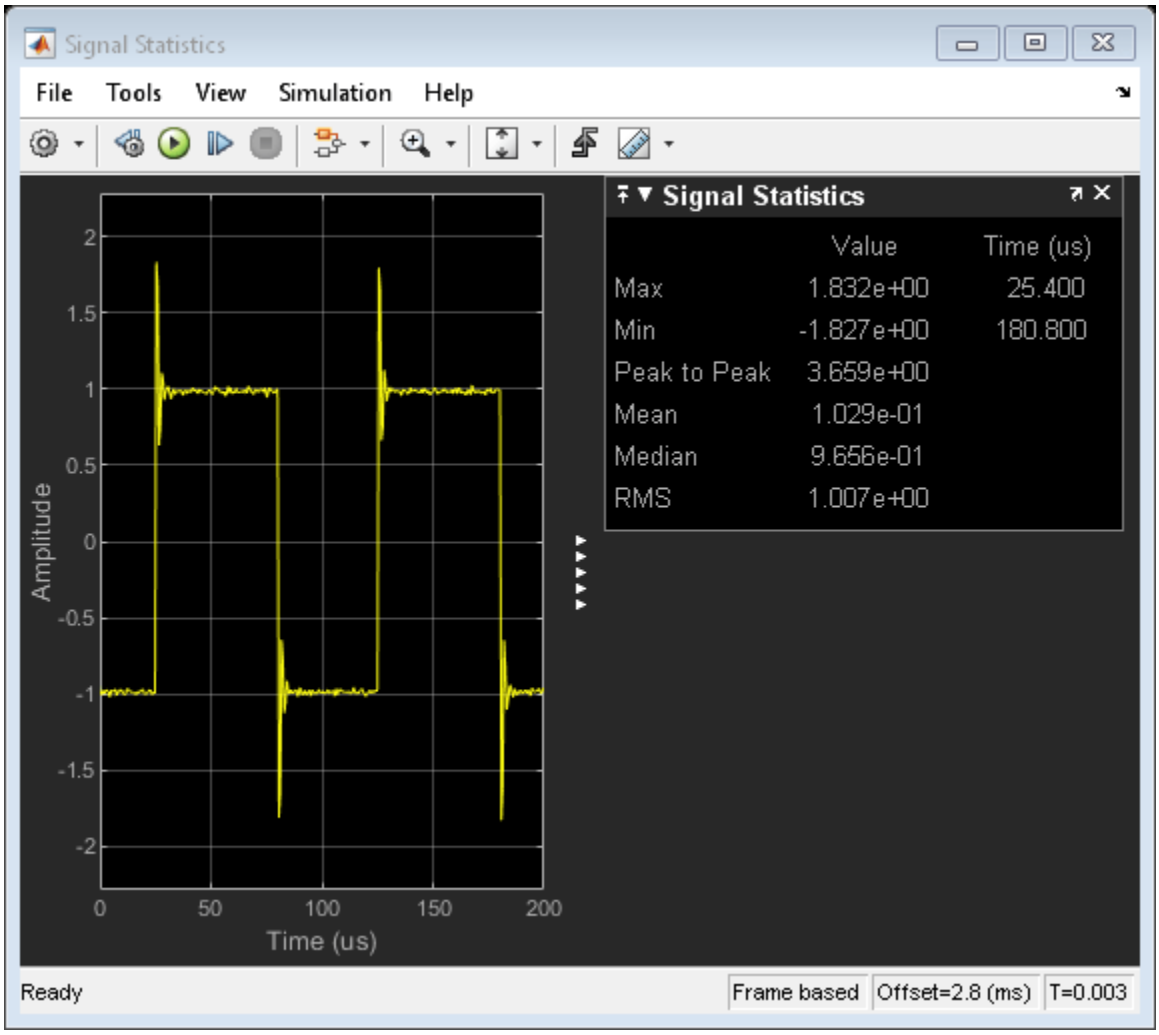
You can measure the relative distances between events of the waveform by using cursor measurements. Here the cursors are at the start of each pulse and confirm that the pulse period is 10 kHz.



Experiment with the settings to move the cursors anywhere on the screen or measure the locations of other signals. You can move the cursors with the arrow keys and also snap them to either the nearest data point or screen pixel.

Signal Statistics

You can view basic signal statistics of the captured wave with the Signals Statistics measurement dialog.



You can observe the minimum and maximum values of the displayed signal and other signal metrics, such as the peak-to-peak, mean, median, and RMS values.

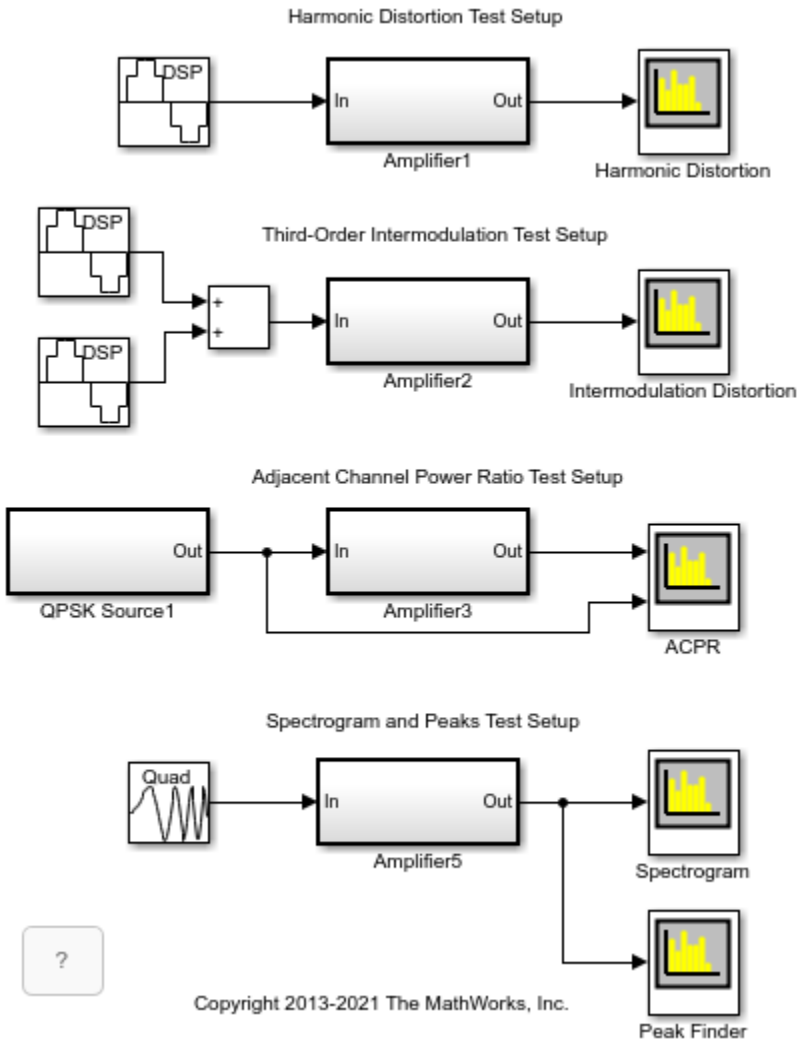
References

- IEEE Std. 181-2003 IEEE Standard on Transitions, Pulse, and Related Waveforms

Spectrum Analyzer Measurements

This example shows how to perform measurements using the Spectrum Analyzer block. The example contains a typical setup to perform harmonic distortion measurements (THD, SNR, SINAD, SFDR), third-order intermodulation distortion measurements (TOI), and adjacent channel power ratio measurements (ACPR). The example also shows how to view time-varying spectra by using a spectrogram and automatic peak detection.

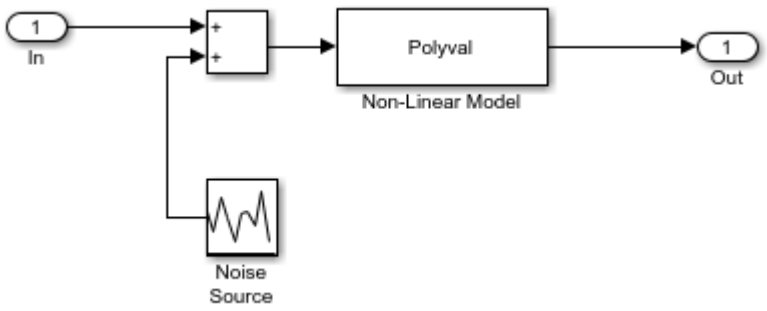
Several measurements and their corresponding setups are contained in the example model.



Exploring the Example

The model consists of five simple models of an amplifier, each of which is set up to perform specific measurements.

Open an amplifier model by double-clicking on an Amplifier block. The first amplifier model is shown below:



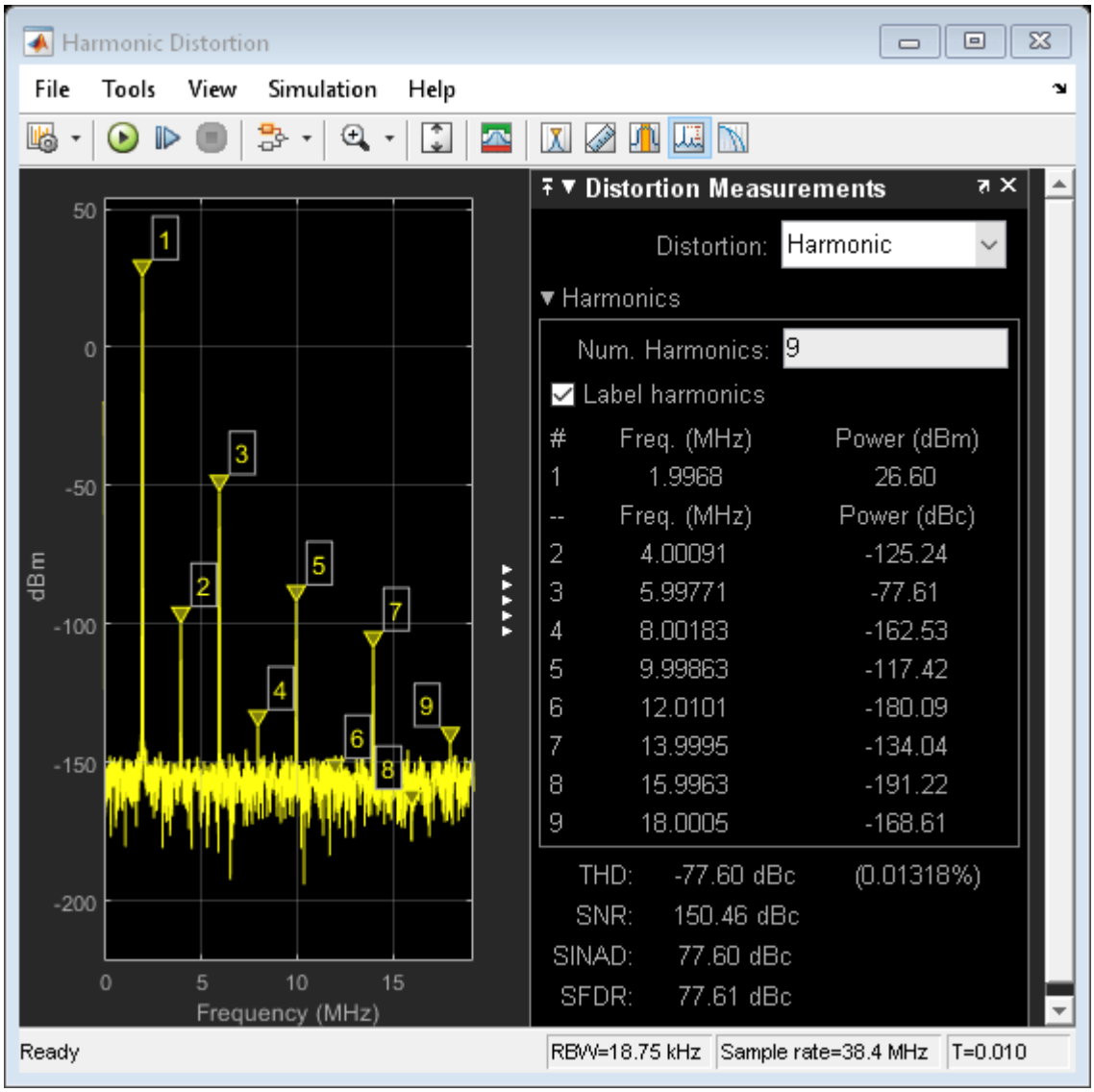
The input is first combined with a Gaussian noise source and then run through a high-order polynomial to model non-linear distortion.

You can modify the amount of additive noise on the input by clicking on the Noise Source and modifying the variance of the Gaussian distribution.

You can modify the parameters of the amplifier by changing the polynomial coefficients. The coefficients are arranged from highest-to-lowest order. If you edit the last coefficient you change the DC voltage offset of the amplifier. If you change the next-to-last coefficient, you change the voltage gain of the amplifier. If you change other coefficients, you can change the higher-order harmonics of the amplifier.

Harmonic Distortion

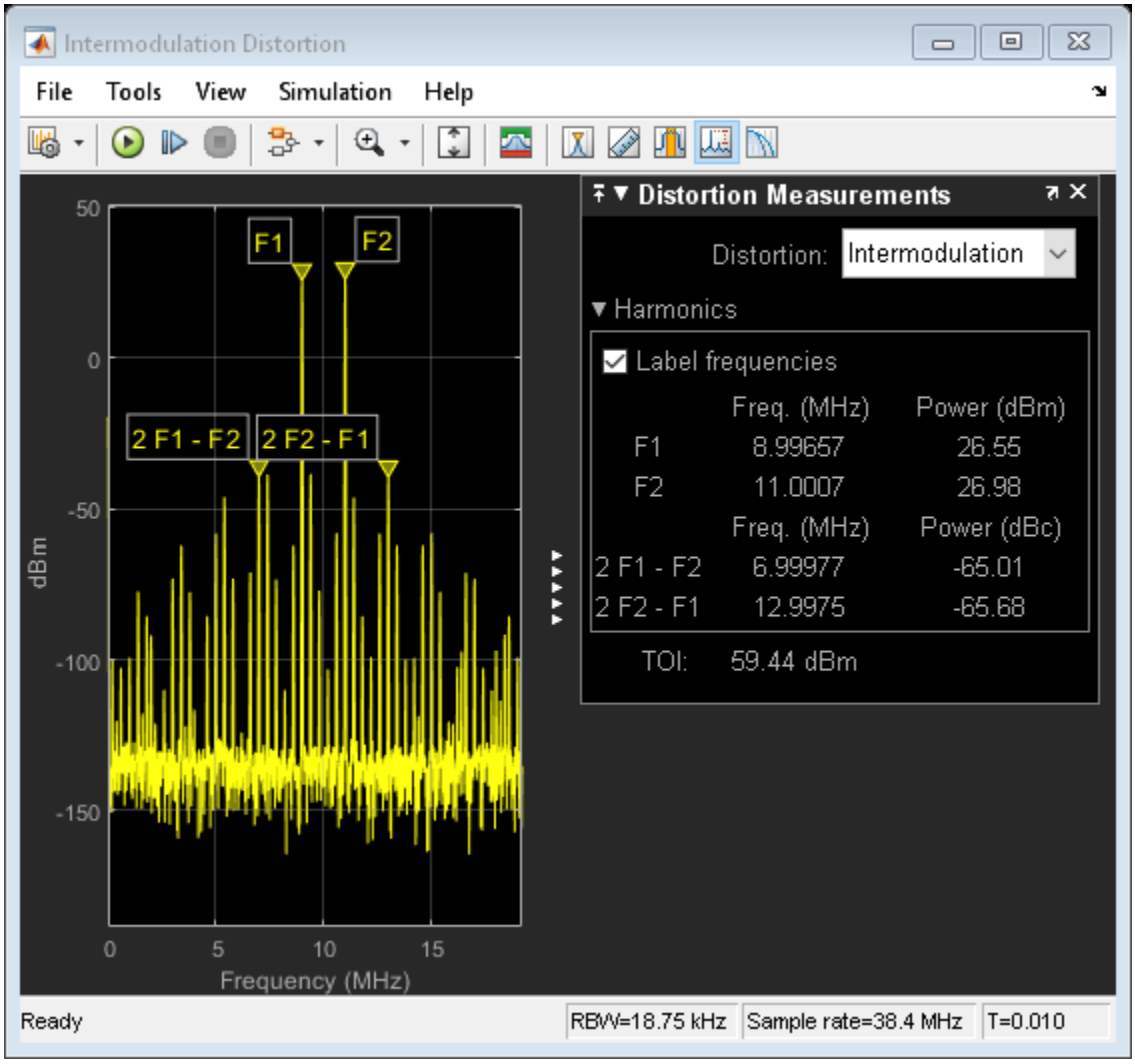
You can measure harmonic distortion by stimulating the amplifier with a sinusoidal input and viewing the harmonics in a spectrum analyzer. The harmonic distortion measurements can be invoked from the Measurements option in the Tools menu, or by clicking its corresponding icon in the toolbar (shown depressed in the figure, below).



Viewing the results in the Distortion Measurement panel you see the amplitudes of the fundamental and the harmonics as well as their SNR, SINAD, THD and SFDR values, which are referenced with respect to the fundamental output power.

Third Order Intermodulation Distortion

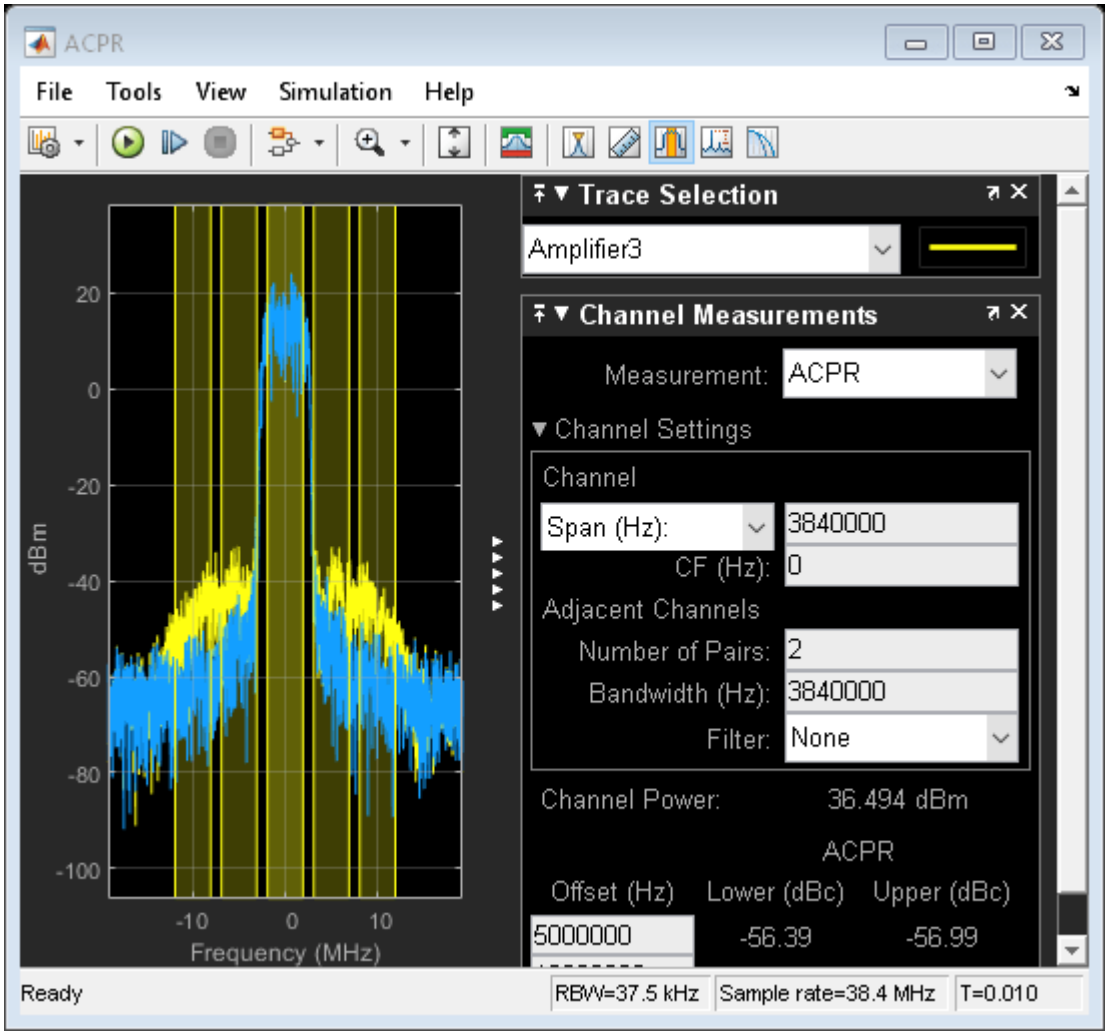
Amplifiers typically have significant odd-order harmonics. If you stimulate the amplifier with two closely spaced sinusoids of equal amplitude, you can produce intermodulation products at the output. Typically the distortion products decay away from the fundamental tones, the largest of which correspond to the third-order sum and difference frequencies of the input waveform. You can measure output third-order intermodulation (TOI) distortion by selecting intermodulation distortion measurements from the drop-down menu in the distortion measurement panel.



Also in the Distortion Measurement panel you see the intermodulation products highlighted and the output TOI displayed. Adjust the polynomial coefficients in the amplifier to change the harmonics shown in the signal.

ACPR

If you stimulate an amplifier that is broadcasting a communications channel, you may see spectral growth leaking into the bandwidth of neighboring channels due to intermodulation distortion. You can measure how much power leaks into these adjacent channels by measuring the adjacent channel power ratio (ACPR). You can see the measurements both before and after the amplifier by toggling the measurement input in the Trace Selection Dialog. ACPR measurements can be selected from the drop-down menu in the Channel Measurements dialog. This dialog can be invoked from the Measurements option in the Tools menu, or by clicking its corresponding icon in the toolbar (shown depressed in the figure, below).

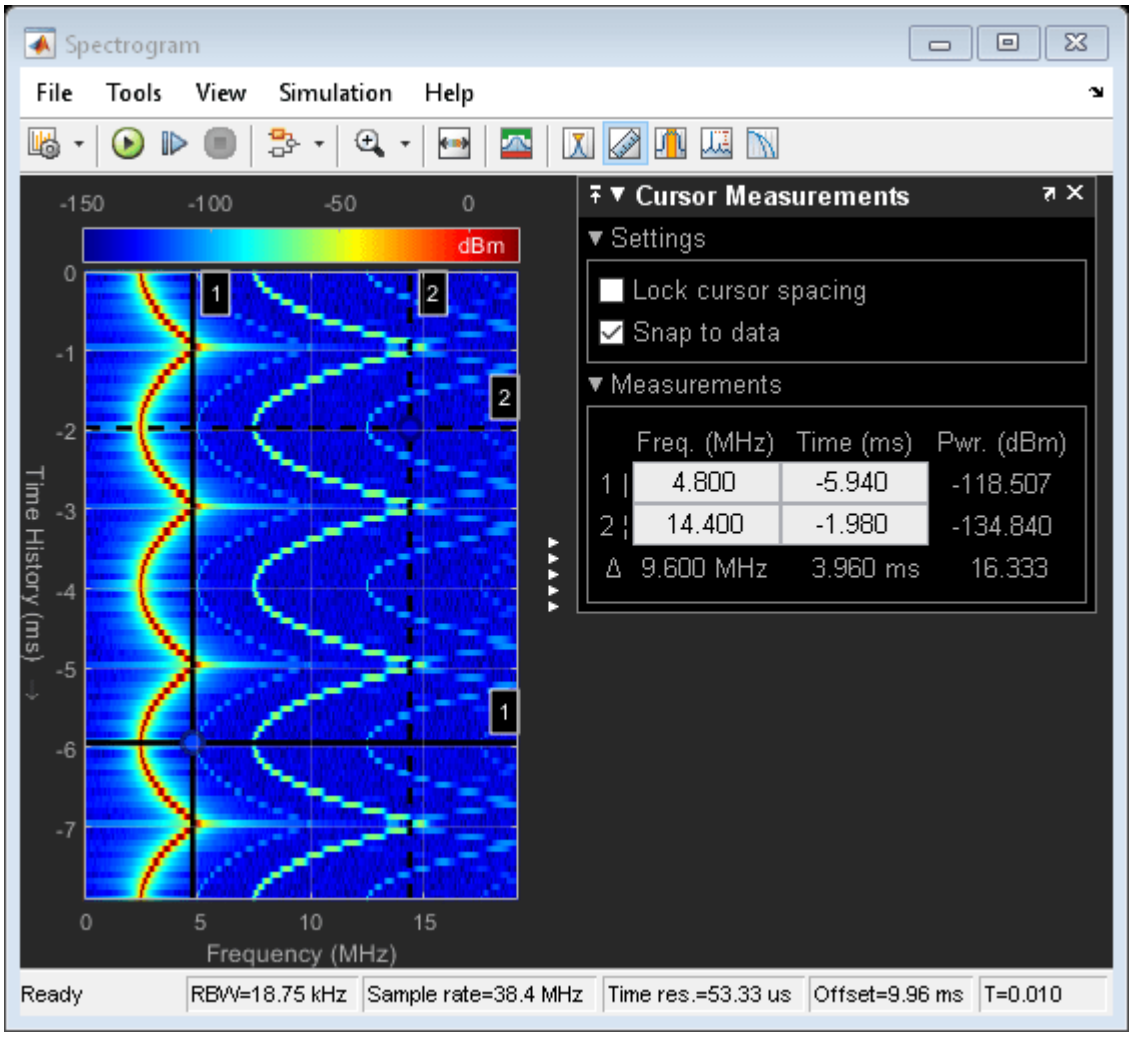


Adjust the polynomial coefficients in the amplifier to obtain different amounts of spreading of the central power due to intermodulation distortion. You can observe the ACPR readings at the specified offset frequencies.

In the above example you can see approximately 0.5 dB of compression between the input source (blue trace) and the output of Amplifier4 (yellow trace). The peak-to-average power ratio (PAPR) for the input channel is 3.3 dB whereas the PAPR for the output channel is 2.8 dB. This loss of dynamic range suggests that there is too much input power applied to the amplifier.

Spectrogram

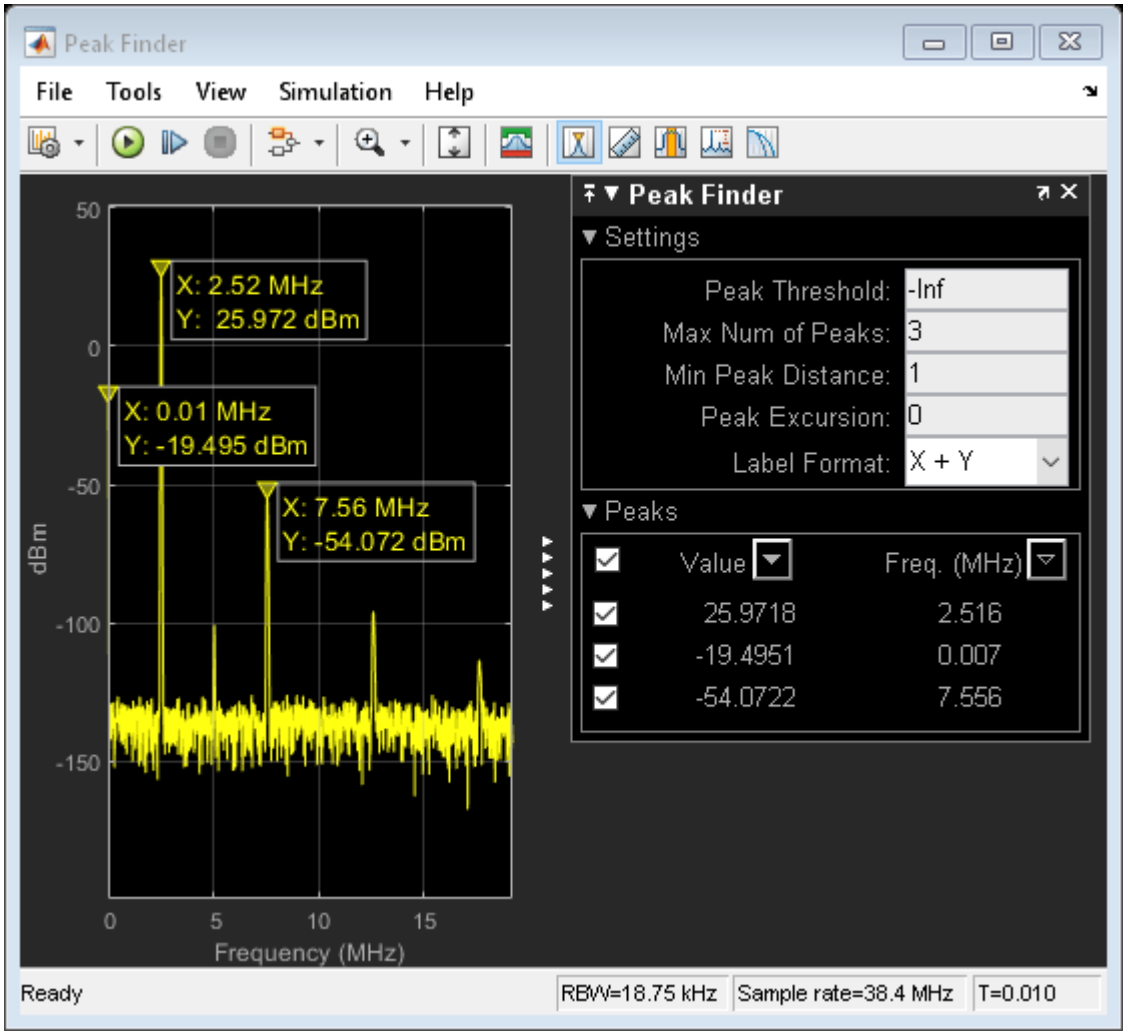
You can view time-varying spectral information by using the Spectrogram Mode of the spectrum analyzer. If you stimulate the amplifier with a chirp waveform you can observe how the harmonics behave as time progresses. Select "Spectrogram" from the "Type" dropdown menu in the Spectrum Settings dialog, which is invoked from the Spectrum Settings dialog in the View menu (not shown).



You can use cursors to make measurements of the period of the chirp and to confirm that the other spectral components are harmonically related. The Cursor Measurements dialog can be invoked from the Measurements option in the Tools menu, or by clicking its corresponding icon in the toolbar (shown depressed in the figure, above).

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. The Peak Finder dialog can be invoked from the Measurements option in the Tools menu, or by clicking its corresponding icon in the toolbar (shown depressed in the figure, above).



References

- IEEE Std. 1057-1994 IEEE Standard for Digitizing Waveform Recorders
- Allan W. Scott, Rex Frobenius, RF Measurements for Cellular Phones and Wireless Data Systems, John Wiley & Sons, Inc. 2008

Generate a Multithreaded MEX File from a MATLAB Function Using Unfolding

This example shows how to use the *dspunfold* function to generate a multithreaded MEX file from a MATLAB function.

NOTE : The following assumes that the current host computer has at least 2 physical CPU cores. The presented screenshots, speedup, and latency values were collected using a host computer with 8 physical CPU cores.

Required MathWorks™ products:

- DSP System Toolbox™
- MATLAB® Coder™

Introduction

`dspunfold` generates a multithreaded MEX file from a MATLAB function using unfolding technology. This MATLAB function can contain an algorithm which is stateless (has no states) or stateful (has states).

Using `dspunfold`

Consider the MATLAB function spectralAnalysisExample. The function performs the following algorithm:

1) Compute the one-sided spectrum estimate of the input

2) Compute the total harmonic distortion (THD), signal to noise ratio (SNR), signal to noise and Distortion ratio (SINAD) and the spurious free dynamic range (SFDR) of the spectrum

```
type spectralAnalysisExample
```

```
function [THD,SNR,SINAD,SFDR] = spectralAnalysisExample(x)
%

% Copyright 2015-2016 The MathWorks, Inc.

persistent powerSpectrum
if isempty(powerSpectrum)
  powerSpectrum  = dsp.SpectrumEstimator('FrequencyRange','onesided',...
                              'SampleRate',8000,...
                              'SpectralAverages',1);
end

% Get one-sided spectrum estimate
Pxx = powerSpectrum(x);

% Compute measurements
[amp, harmSum, totalNoise, maxSpur] = ...
    getHarmonicDistortion(...
    getFrequencyVector(powerSpectrum), Pxx, getRBW(powerSpectrum), 6);

THD   = 10*log10(harmSum/amp(1));
SNR   = 10*log10(amp(1)/totalNoise);
```

```
SINAD = 10*log10(amp(1)/(harmSum + totalNoise));
SFDR  = 10*log10(amp(1)/maxSpur);
```

To accelerate the algorithm, a common approach is to generate a MEX file using the *codegen* function. Below is an example of how to do so when using an input of 4096 doubles. The generated MEX file, dspunfoldDCTExample_mex, is single-threaded.

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

To generate a multithreaded MEX file, use the *dspunfold* function. The argument -s indicates that the algorithm in spectralAnalysisExample has no states.

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

```
State length: 0 frames, Repetition: 1, Output latency: 16 frames, Threads: 8
Analyzing: spectralAnalysisExample.m
Creating single-threaded MEX file: spectralAnalysisExample_st.mexmaci64
Creating multi-threaded MEX file: spectralAnalysisExample_mt.mexmaci64
Creating analyzer file: spectralAnalysisExample_analyzer.p
```

This will generate the following files:

- multithreaded MEX file, **spectralAnalysisExample_mt**
- single-threaded MEX file, **spectralAnalysisExample_st** (which is identical to the MEX file obtained using the *codegen* function)
- self-diagnostic analyzer function, **spectralAnalysisExample_analyzer**

To measure the speedup of the multithreaded MEX file relative to the single-threaded MEX file, see the example function dspunfoldBenchmarkSpectrumExample:

```
type dspunfoldBenchmarkSpectrumExample
```

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

% Copyright 2015 The MathWorks, Inc.

clear spectralAnalysisExample_st;  % for benchmark precision purpose
clear spectralAnalysisExample_mt;  % for benchmark precision purpose

numFrames = 1e5;
inputFrame = (1:4096)';

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

% exclude first run from timing measurements
spectralAnalysisExample_mt(inputFrame);
tic;  % measure execution time for the multi-threaded MEX
for frame = 1:numFrames
    spectralAnalysisExample_mt(inputFrame);
```

```
end
timeMultiThreaded = toc;
fprintf('Speedup = %.1fx\n',timeSingleThreaded/timeMultiThreaded);
```

dspunfoldBenchmarkSpectrumExample measures the execution time taken by spectralAnalysisExample_st and spectralAnalysisExample_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.

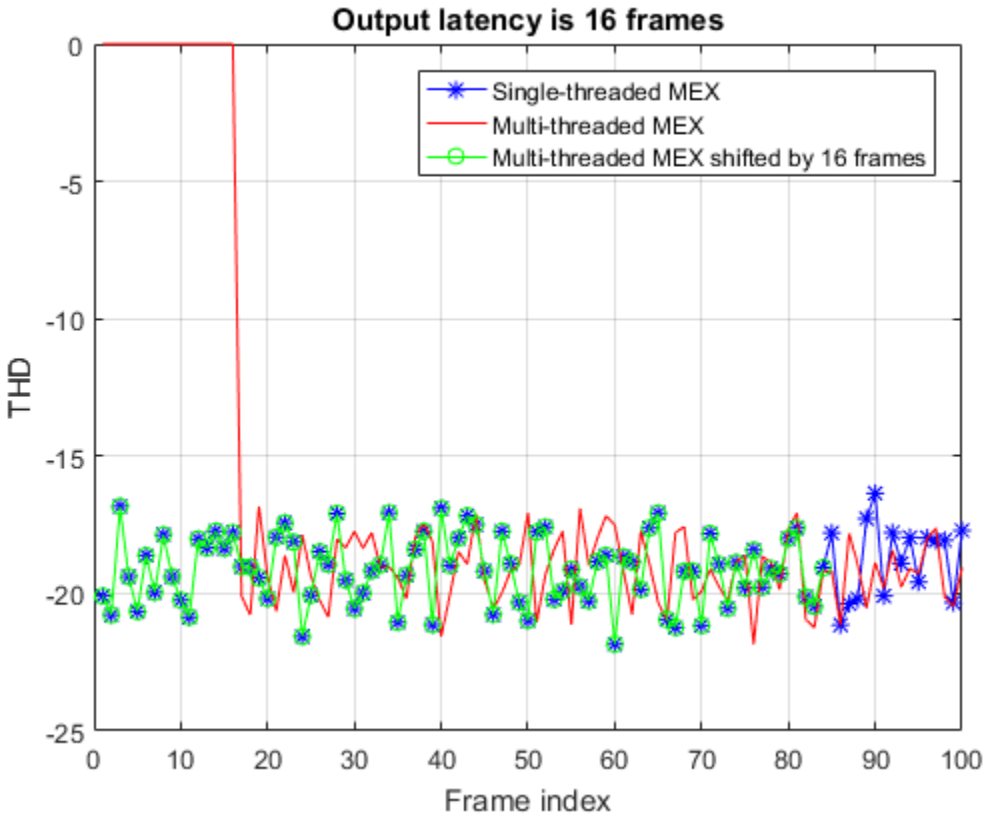
```
dspunfoldBenchmarkSpectrumExample;
```

```
Speedup = 2.0x
```

The speedup could be improved even more by increasing the Repetition value, which will be discussed later.

DSP unfolding 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 spectralAnalysisExample_mt' displays more information about the multithreaded MEX file, one of them being 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. Below is the plot generated by dspunfoldShowLatencySpectrumExample, which 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.

```
dspunfoldShowLatencySpectrumExample;
```



Verify Resulting Multithreaded MEX with 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 this example, the name of the analyzer is spectralAnalysisExample_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 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.

The example below executes the analyzer for the multithreaded MEX file, dspunfoldFIRExample_mt.

```
Fs = 8000;
NumFrames = 10;
t  = (1/Fs) * (0:4096*NumFrames-1); t = t.';
f = 100;
x  = sin(2*pi*f*t) + .01 * randn(size(t));
spectralAnalysisExample_analyzer(x)
```

```
Analyzing multi-threaded MEX file spectralAnalysisExample_mt.mexmaci64. For best results, please
Latency = 16 frames
Speedup = 2.3x

ans =

  struct with fields:

    Latency: 16
    Speedup: 2.3114
       Pass: 1
```

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 4096 doubles for its first input, while 4096*10 samples were provided to spectralAnalysisExample_analyzer. The analyzer interprets this input as 10 frames of 4096 samples. The analyzer alternates between these 10 input frames (in a circular fashion) while checking if the outputs of the multithreaded and single-threaded MEX files match.

NOTE : For the analyzer to correctly check for the numerical match between the multithreaded MEX and single-threaded MEX, it is recommended that you provide at least 2 frames with different values for each input.

Specifying State and Repetition Values

Let us modify the spectral measurement example by setting the spectral average length of the spectrum estimate to 4 instead of 1. The spectrum estimate is now a running average of the current estimate and the three previous estimate. This algorithm has a state length of 3 frames. The MATLAB function spectralAnalysisWithStatesExample contains the modified algorithm:

```
type spectralAnalysisWithStatesExample
```

```
function [THD,SNR,SINAD,SFDR] = spectralAnalysisWithStatesExample(x)
%

% Copyright 2015-2016 The MathWorks, Inc.
```

```
persistent powerSpectrum
if isempty(powerSpectrum)
  powerSpectrum  = dsp.SpectrumEstimator('FrequencyRange','onesided',...
                              'SampleRate',8000,...
                              'SpectralAverages',4);
end

% Get one-sided spectrum estimate
Pxx = powerSpectrum(x);

% Compute measurements
[amp, harmSum, totalNoise, maxSpur] = ...
    getHarmonicDistortion(...
    getFrequencyVector(powerSpectrum), Pxx, getRBW(powerSpectrum), 6);

THD   = 10*log10(harmSum/amp(1));
SNR   = 10*log10(amp(1)/totalNoise);
SINAD = 10*log10(amp(1)/(harmSum + totalNoise));
SFDR  = 10*log10(amp(1)/maxSpur);
```

To build the multithreaded MEX file, we have to provide the state length corresponding to the two FIR filters. Specifying -s 3 when invoking dspunfold indicates that the state length does not exceed 3 frames.

The speedup can be increased even more by increasing the repetition (-r) provided when invoking *dspunfold*. The default repetition value is 1. Increasing this value makes the multithreaded MEX buffer more frames internally, before it starts processing them, increasing the efficiency of the multithreading, but at the cost of a higher output latency. Also note that the maximum state length allowed is (threads-1)*Repetition*FrameSize frames. If the specified state length exceeds that value, dspunfold falls back a single-threaded MEX. If latency may be tolerated by the application, increasing the value of repetition allows generating a multithreaded MEX with a longer state.

The command below generates a multithreaded MEX function using a repetition value of 5 and a state length of 3 frames:

```
dspunfold spectralAnalysisWithStatesExample -args {(1:4096)'} -s 3 -r 5

State length: 3 frames, Repetition: 5, Output latency: 80 frames, Threads: 8
Analyzing: spectralAnalysisWithStatesExample.m
Creating single-threaded MEX file: spectralAnalysisWithStatesExample_st.mexmaci64
Creating multi-threaded MEX file: spectralAnalysisWithStatesExample_mt.mexmaci64
Creating analyzer file: spectralAnalysisWithStatesExample_analyzer.p
```

The analyzer may be used to validate the numerical results of the multithreaded MEX and provide speed-up and latency information:

```
L = 4096;
NumFrames = 10;
sine  = dsp.SineWave('SamplesPerFrame',L * NumFrames,'SampleRate',8000);
x     = sine() + 0.01 * randn(L * NumFrames, 1);
spectralAnalysisWithStatesExample_analyzer(x)

Analyzing multi-threaded MEX file spectralAnalysisWithStatesExample_mt.mexmaci64. For best result
Latency = 80 frames
Speedup = 2.4x
```

```
ans =

  struct with fields:

    Latency: 80
    Speedup: 2.4061
       Pass: 1
```

Simulation Example

The function dspunfoldNoisySineExample demonstrates the usage of the multithreaded MEX to estimate spectral characteristics of a noisy sine wave. The measurements are plotted on a time scope. Performance of the multithreaded MEX is compared to the MATLAB simulation and the single-threaded MEX performance. The gains of the multithreaded MEX are still apparent even with the overhead brought by the plotting and input signal generation of the testbench.

```
dspunfoldNoisySineExample
```

```
MATLAB Sim/Single-threaded MEX speedup: 2.3
MATLAB Sim/Multi-threaded MEX speedup: 3.1
```

References

DSP unfolding on Wikipedia : Unfolding (DSP implementation)

Generate Standalone Executable And Interact With It Using UDP

This example shows how to generate a standalone executable for streaming statistics using MATLAB Coder™ and tune the generated executable using a user interface (UI) that is running in MATLAB (TM).

Introduction

Most algorithms in DSP System Toolbox™ support C code generation using MATLAB Coder.

One of the options of MATLAB Coder is to generate a standalone executable that can be run outside of the MATLAB environment by launching the executable directly from a terminal or command prompt.

For algorithms that are tunable, it is desirable to interact with the algorithm at run-time using a UI. One way to achieve this is by sending/receiving information via UDP.

This example uses UDP to exchange between MATLAB and a generated standalone executable at run-time. The variance, bias, and exponential weighting values are sent from MATLAB to the executable. The actual random signal along with mean, RMS, and variance estimates are sent from the standalone executable back to MATLAB for plotting.

Example Architecture

The architecture of the example consists of two primary sections:

1. `streamingStatsCodegenExampleApp`: A MATLAB function that creates the user interface (UI) to change the variance, bias, and exponential weighting values. This function also plots the data received from the standalone executable.
2. `HelperStreamingStatsEXEProcessing`: This is the function from which the standalone executable is generated. This function generates a random signal of a given bias and variance and computes mean, RMS, and variance estimates of such a signal. The noise signal along with the statistics that are computed are sent over UDP for plotting (or any further processing). Anytime during the simulation, it can also respond to the changes in the sliders of the MATLAB UI.

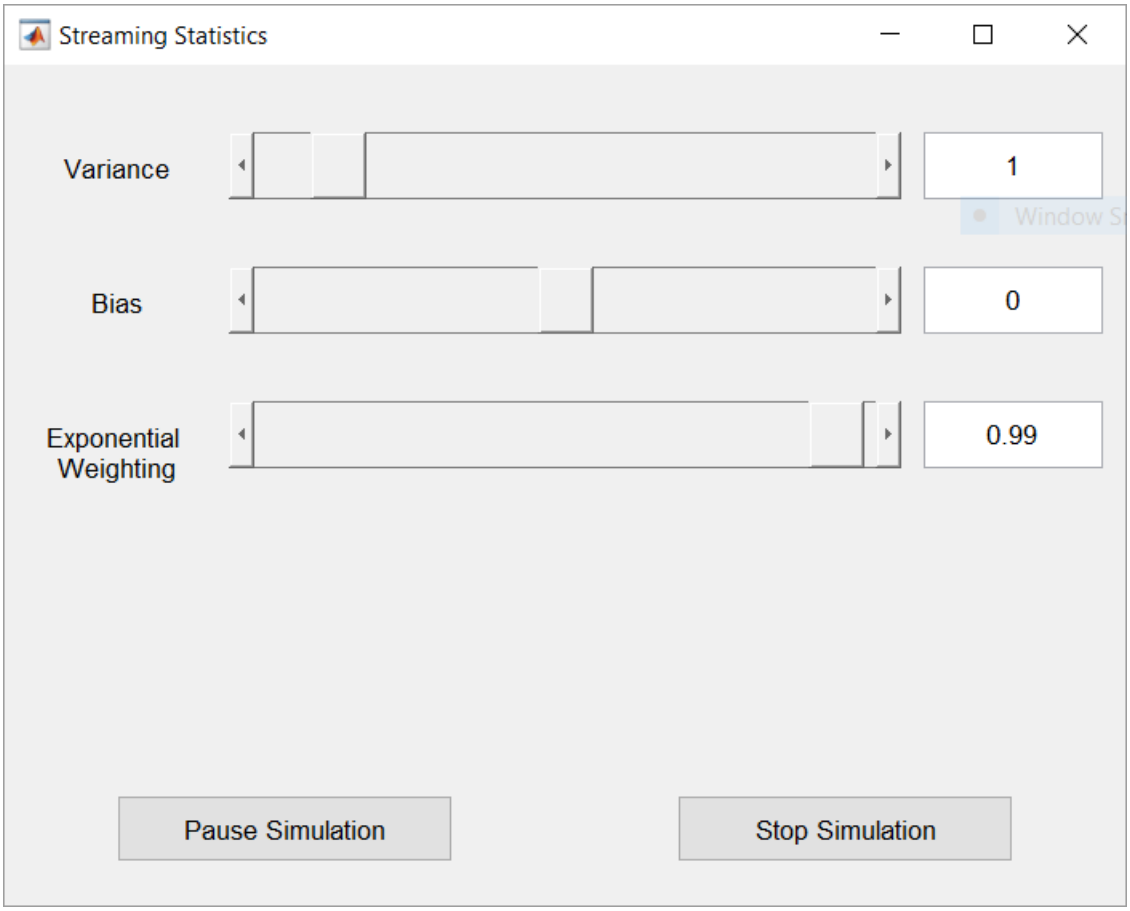
Generating Code and Building an Executable File

You can use MATLAB Coder to generate readable and standalone C-code from the streaming statistics algorithm code. Because UDP is used, there are additional dependencies for the generated code and executable file. These are available in the /bin directory of your MATLAB installation.

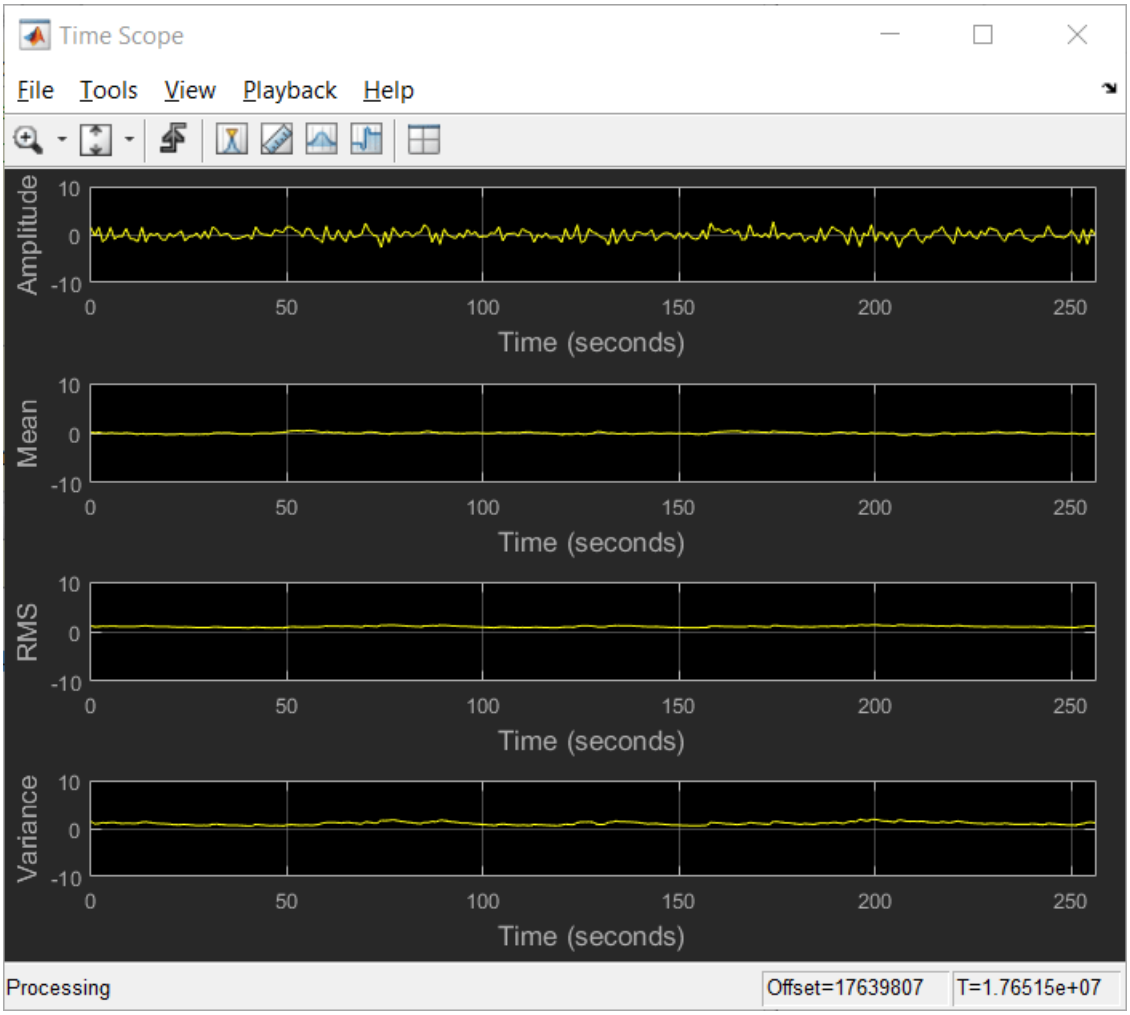
Running the script `HelperStreamingStatsGenerateEXE` will invoke MATLAB Coder to automatically generate C-code and a standalone executable from the algorithm code present in `HelperStreamingStatsEXEProcessing`.

Running the Example

Once you have generated the executable, run the function `streamingStatsCodegenExampleApp` to launch the executable and a user interface (UI) designed to interact with the simulation. The UI allows you to tune parameters and the results are reflected in the simulation instantly. For example, moving the slider for the 'Variance' while the simulation is running, will affect the noise signal along with the RMS and variance estimates that are plotted.



There are also two buttons on the UI - the 'Pause Simulation' button will hold the simulation until you press on it again. The simulation may be terminated by clicking on the 'Stop simulation' button.



Threading

The standalone executable is executed as a separate process. This means that the graphics can run in parallel with the statistics computation. This can be an attractive approach for high performance computations involving graphics.

Manually Invoking the Executable

In lieu of using the `system` command to launch the executable from within MATLAB, the executable can be launched manually from a terminal or command prompt. Because this executable includes UDP calls, it is necessary that the dlls be on the path for proper behavior. See “How To Run a Generated Executable Outside MATLAB” on page 20-27 for more information.

Code Generation for Parametric Audio Equalizer

This example shows how to model an algorithm specification for a three band parametric equalizer which will be used for code generation.

Required MathWorks™ products:

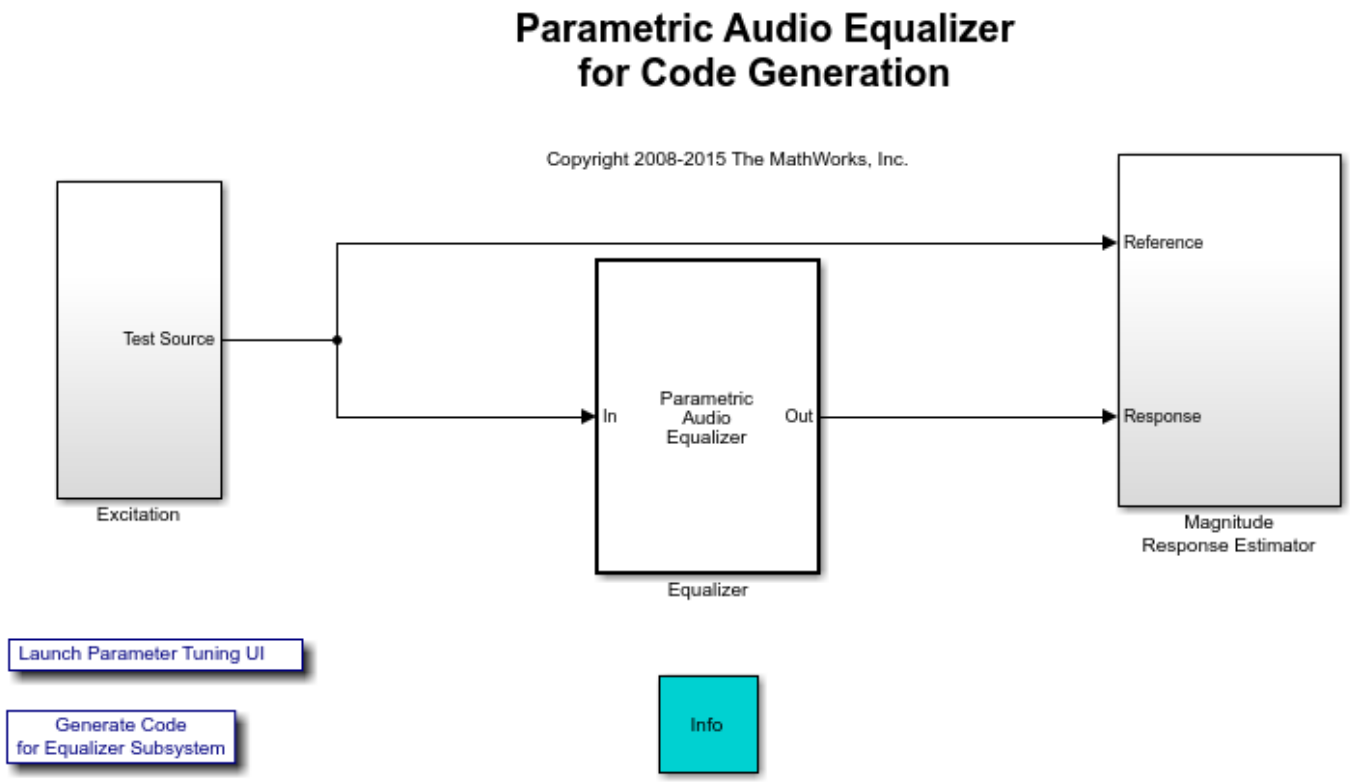
- MATLAB®
- Signal Processing Toolbox™
- DSP System Toolbox™
- Simulink®
- MATLAB® Coder™
- Simulink® Coder™
- Embedded Coder®

Introduction

Parametric equalizers are often used to adjust the frequency response of an audio system. For example, a parametric equalizer can be used to compensate for physical speakers which have peaks and dips at different frequencies.

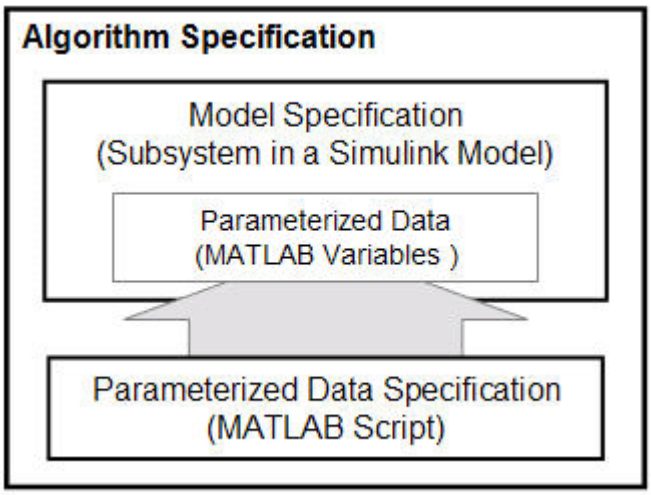
The parametric equalizer algorithm in this example provides three second-order (biquadratic) filters whose coefficients can be adjusted to achieve a desired frequency response. A user interface is used in simulation to dynamically adjust filter coefficients and explore behavior. For code generation, the coefficient variables are named and placed in files such that they could be accessed by other software components that dynamically change the coefficients while running on the target processor.

The following sections will describe how the parametric equalizer algorithm is specified, how the behavior can be explored through simulation, and how the code can be generated and customized.



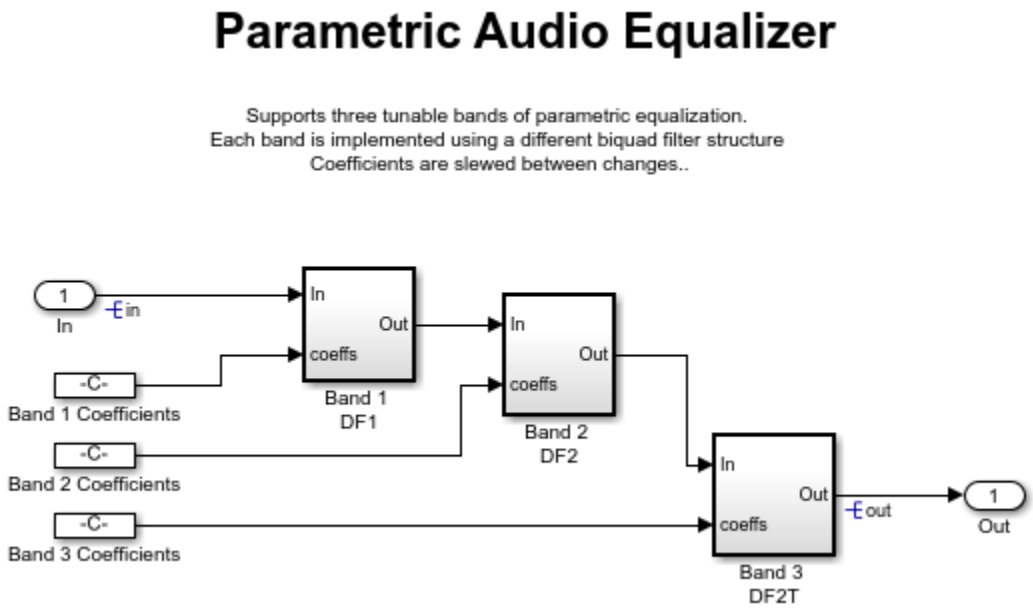
Specify Algorithm

The parametric equalizer algorithm is specified in two parts: a model specification and a parameterized data specification. The model specification is a Simulink subsystem that specifies the signal flow of the algorithm. The model specification also accesses parameterized data that exists in the MATLAB workspace. The parameterized data specification is a MATLAB script that creates the data that is accessed by the Simulink model.



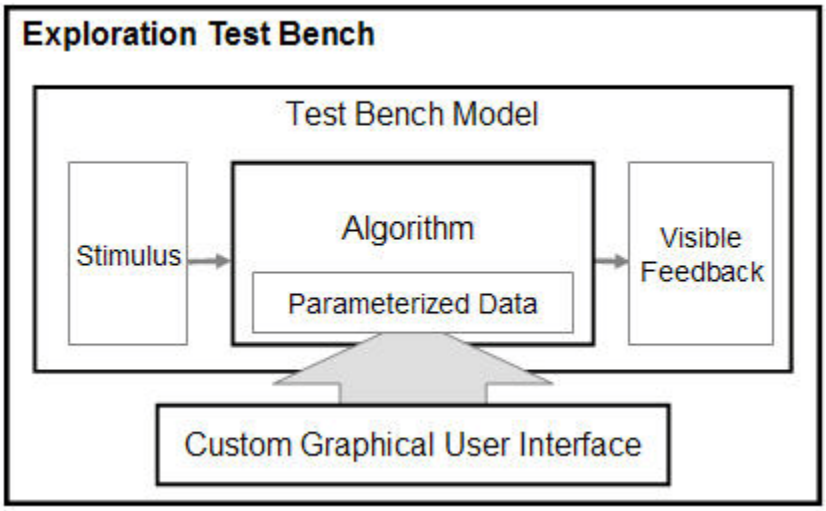
For this example, the model specification is the Equalizer subsystem of the Simulink model dspparameqcodegen. In this subsystem, the input is passed through three cascaded bands of equalization. Coefficient changes within each band are smoothed through a leaky integrator before being passed into a Biquad Filter block. Each Biquad Filter block is configured to use a different filter structure. Different filter structures are selected to show the differences in code generation later in this example.

For this example, the parameterized data specification is the MATLAB script dspparameq_data.m. This MATLAB script specifies the initial filter coefficients as well as code generation attributes. When you open the model dspparameqcodegen, the model's `PreLoadFcn` callback is configured to run the dspparameq_data.m script that creates the parameter data in the MATLAB workspace.

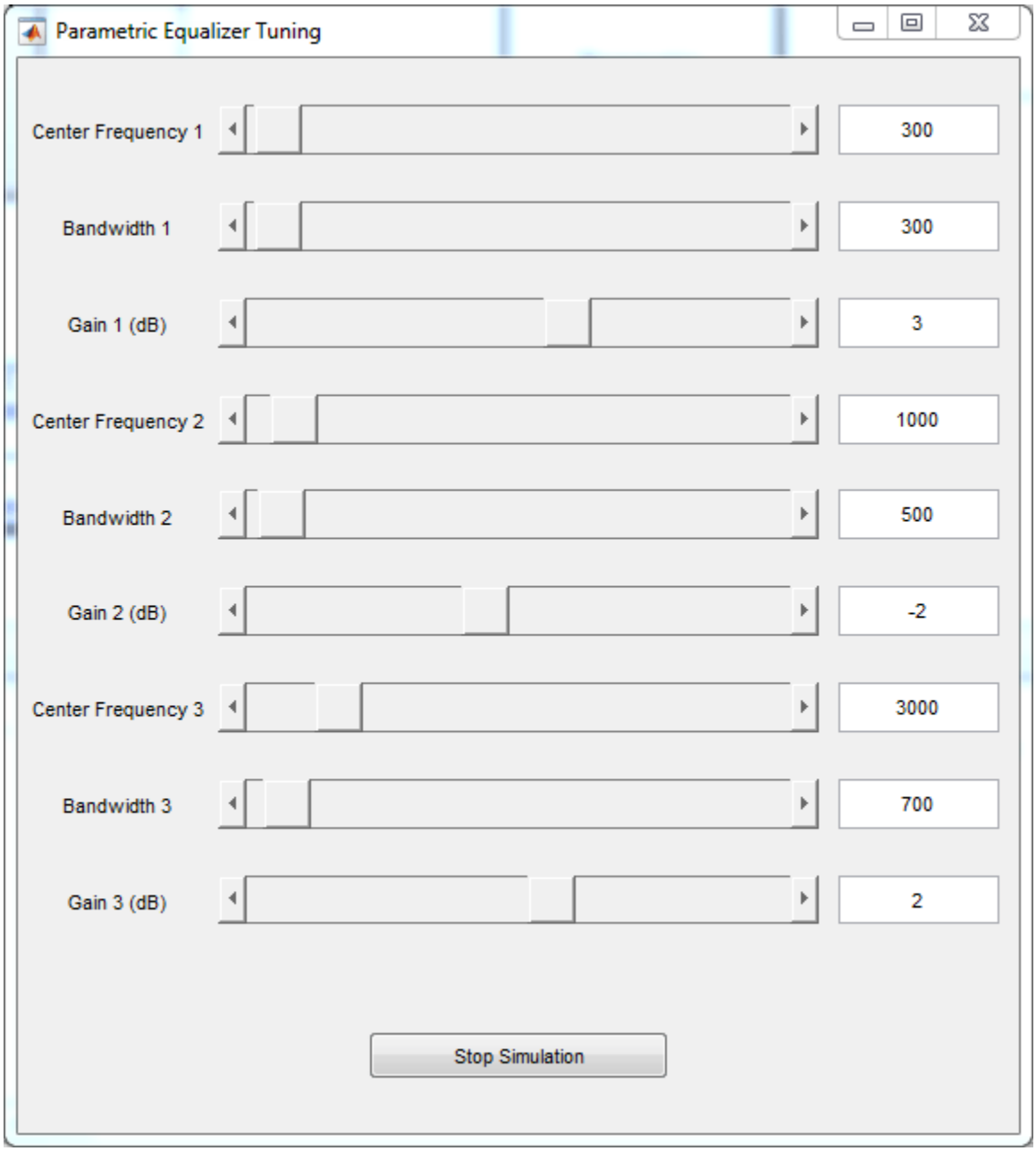


Explore Behavior Through Simulation

You can use a simulation test bench to explore the behavior of the algorithm. In this example, the test bench consists of the simulation model, dspparameqcodegen, as a well as custom user interface (UI) programmed in MATLAB.



This UI can be launched by clicking the 'Launch Parameter Tuning UI' link. The UI enables dynamic adjustment of coefficient parameter data in the MATLAB workspace during the simulation.



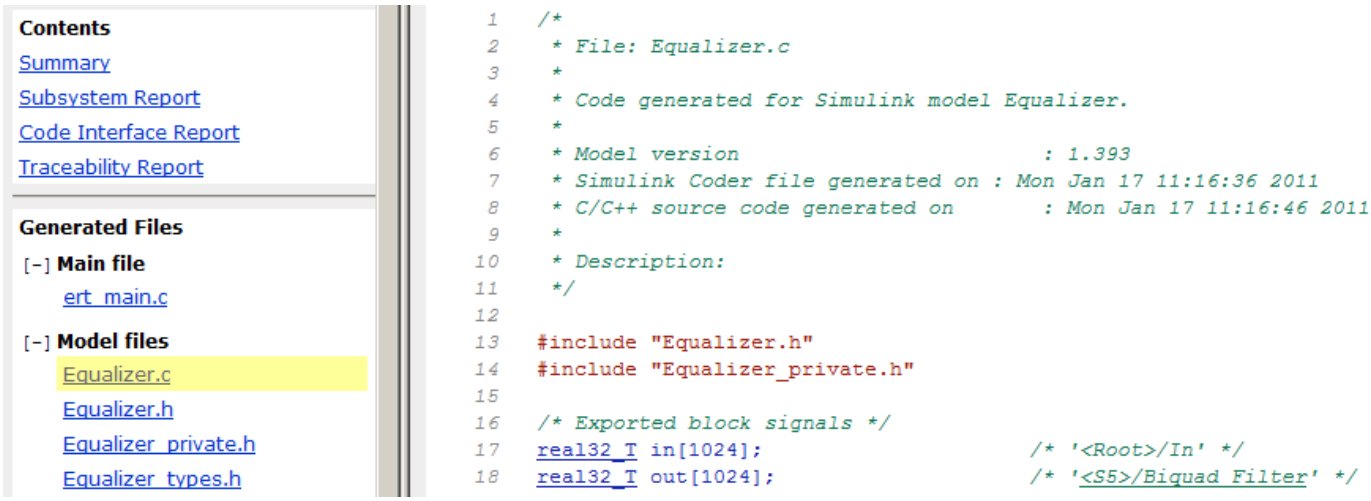
Generate C Code for the Equalizer Subsystem

Once you achieve the desired simulation behavior, you can generate C code for the Equalizer subsystem based on the algorithm specification. This model is configured to show some common code generation customizations accessible from Embedded Coder product. These customizations ease the code review and integration process. The following sections show some of the code customizations for this model and provide references to documentation that describe these customizations in more detail.

To generate C code, right-click on the Equalizer subsystem, select C/C++ Code > Build This Subsystem, then click the Build button when prompted for tunable parameters. You can also generate code by clicking the following hyperlink: Generate Code for the Equalizer Subsystem.

Code Generation Report with Links to and from the Model

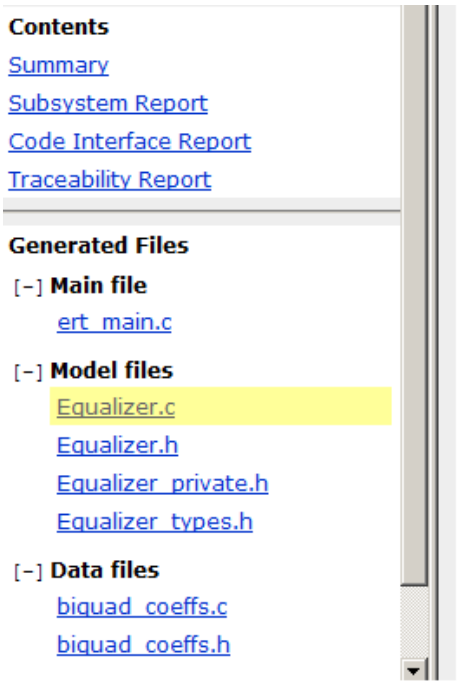
The model is configured to generate an HTML report that can be used to navigate the generated source and header files. The report also enables bidirectional linking between the generated code and the model. For example, each Biquad Filter block is configured to implement a different filter structure. You can trace from the block to the associated code by right clicking on any of the Biquad Filter blocks and then selecting C/C++ Code > Navigate To C/C++ Code.



For more information on traceability between the model and code see “Trace Simulink Model Elements in Generated Code” (Embedded Coder).

Calling the Generated Code

You can integrate the generated code into an application by making calls to the model initialization and model step functions. An example `ert_main.c` file is generated that shows how to call the generated code. Note that the example `main()` calls `Equalizer_initialize()` to initialize states. The example `rt_OneStep()` shows how a periodic mechanism such as an interrupt would call `Equalizer_step()` from the file `Equalizer.c`.




```
195  /* Model step function */
196  void Equalizer_step(void)
197  {
198    /* Outputs for atomic SubSystem: '<Root>/Equalizer' */
199
200    /* Outputs for atomic SubSystem: '<S1>/Band 1   DF1' */
201    Equalizer_Band1DF1();
202
203    /* end of Outputs for SubSystem: '<S1>/Band 1   DF1' */
204
205    /* Outputs for atomic SubSystem: '<S1>/Band 2   DF2' */
206    Equalizer_Band2DF2();
207
208    /* end of Outputs for SubSystem: '<S1>/Band 2   DF2' */
209
210    /* Outputs for atomic SubSystem: '<S1>/Band 3   DF2T' */
211    Equalizer_Band3DF2T();
212
213    /* end of Outputs for SubSystem: '<S1>/Band 3   DF2T' */
214
215    /* end of Outputs for SubSystem: '<Root>/Equalizer' */
216  }
217
```

For more information about how to integrate generated code into another application see “Deploy Generated Standalone Executable Programs To Target Hardware” (Embedded Coder).

Input and Output Data Interface

The parameterized data specification file, dspparameq_data.m, creates `in` and `out` signal data objects in the MATLAB workspace. These data objects are associated with signal lines in the model and are used to specify descriptions and storage classes of the corresponding variables in the generated code. For example, the signals `in` and `out` are declared as a global variable in `Equalizer.c`. To run the model step function, an application writes data to `in`, calls the `Equalizer_step()` function, and then reads the results from `out`.

For more information on Data Objects see “Create Data Objects for Code Generation with Data Object Wizard” (Embedded Coder).

Text Annotations in Code Comments

You can insert design documentation entered as text in the model into the comments of the generated code. The `Equalizer` subsystem contains annotation text with the keyword `S:Description`. The code generator identifies that the text starts with this keyword and inserts the text following the keyword as comments into the generated code.

For more information on inserting annotation text into code comments see “Add Global Comments in the Generated Code” (Embedded Coder).

Function Partitioning

To ease navigation of the generated code, each subsystem for the equalizer bands is configured to be atomic and create its own function. You can see the calling order in the `Equalizer_step()` function.

For more information on customizing function naming and placement see “About Nonvirtual Subsystem Code Generation” (Embedded Coder).

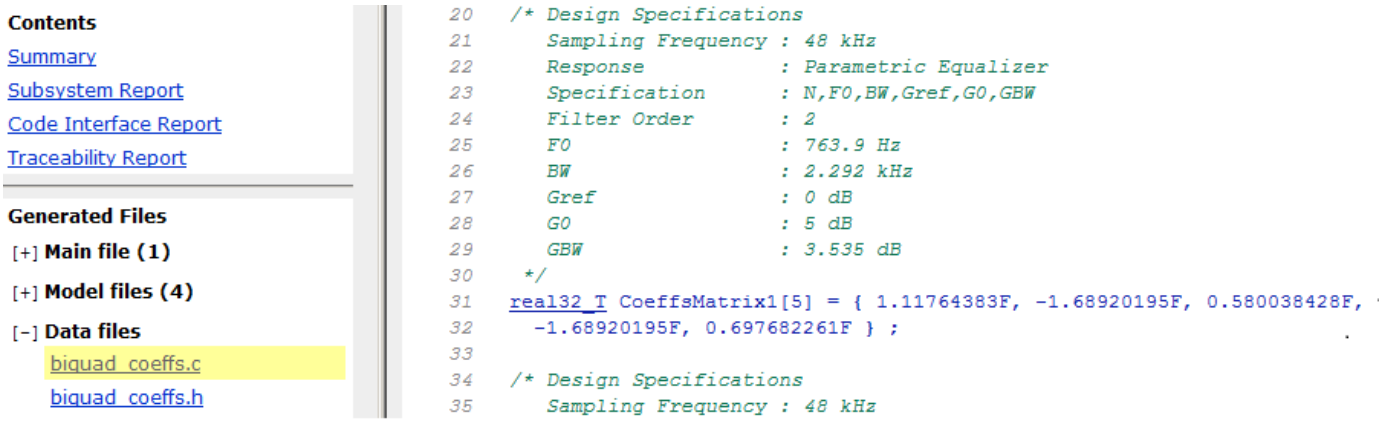
Coefficient File Placement

The parameterized data specification file, dspparameq_data.m creates parameter data objects for the coefficients in the MATLAB workspace. These data objects are configured to define and declare coefficient variables in separate files `biquad_coeffs.c` and `biquad_coeffs.h` respectively. Partitioning coefficients into separate files enables other software components to access this data. For example, in a deployed application, you could schedule another software component to modify these variables at runtime before they are used by `Equalizer_step()`.

For more information about file placement of Data Objects see “Control Placement of Global Data Definitions and Declarations in Generated Files” (Embedded Coder).

Filter Design Parameters in Coefficient Variable Comments

When coefficients are calculated (in the parameterized data file or by the graphical user interface), the filter design parameters are stored in the `Description` field of the coefficient parameter data objects. The model is configured to insert the design parameters as comments in the generated code. This enables reviewers of the code to easily identify which design parameters were used to design the filters.



For more information on customizing the comments of Data Objects in the generated code see “Add Custom Comments for Variables in the Generated Code” (Embedded Coder).

Package Generated Files

The generated files referenced by the HTML report exist in the `Equalizer_ert_rtw` directory. In addition to the files in this directory, other files in the MATLAB application install directory may be required for integration into a project. To ease porting the generated code to other environments, this model is configured to use the PackNGo feature, which packages up all of the required files into the zip file `Equalizer.zip`. Note that the zip file contains all of the required files, but might also contain additional files that may not be required.

For more information on packaging files for integration into other environments, see “Relocate Code to Another Development Environment” (Embedded Coder).

Generate DSP Applications with MATLAB Compiler

This example shows how to use the MATLAB Compiler™ to create a standalone application from a MATLAB function that uses System objects from DSP System Toolbox™.

Introduction

In this example, you start with the function RLSFilterSystemIDCompilerExampleApp that uses RLS filter for system identification. You generate an executable application from this function using MATLAB Compiler and then run the application. The advantage of generating such standalone applications is that they can be run even on systems that do not have MATLAB installed. These only need an installation of MATLAB Runtime.

System Identification Algorithm

Recursive Least-Squares (RLS) filters are adaptive filters that can be used to identify an unknown system. `RLSFilterSystemIDCompilerExampleApp` uses RLS filters to identify a system that has a variable cutoff frequency. The system is a lowpass FIR filter implemented using `dsp.VariableBandwidthFIRFilter`. The RLS filter is implemented using `dsp.RLSFilter`.

For more information on the algorithm and setup, follow the example: “System Identification Using RLS Adaptive Filtering” on page 4-264.

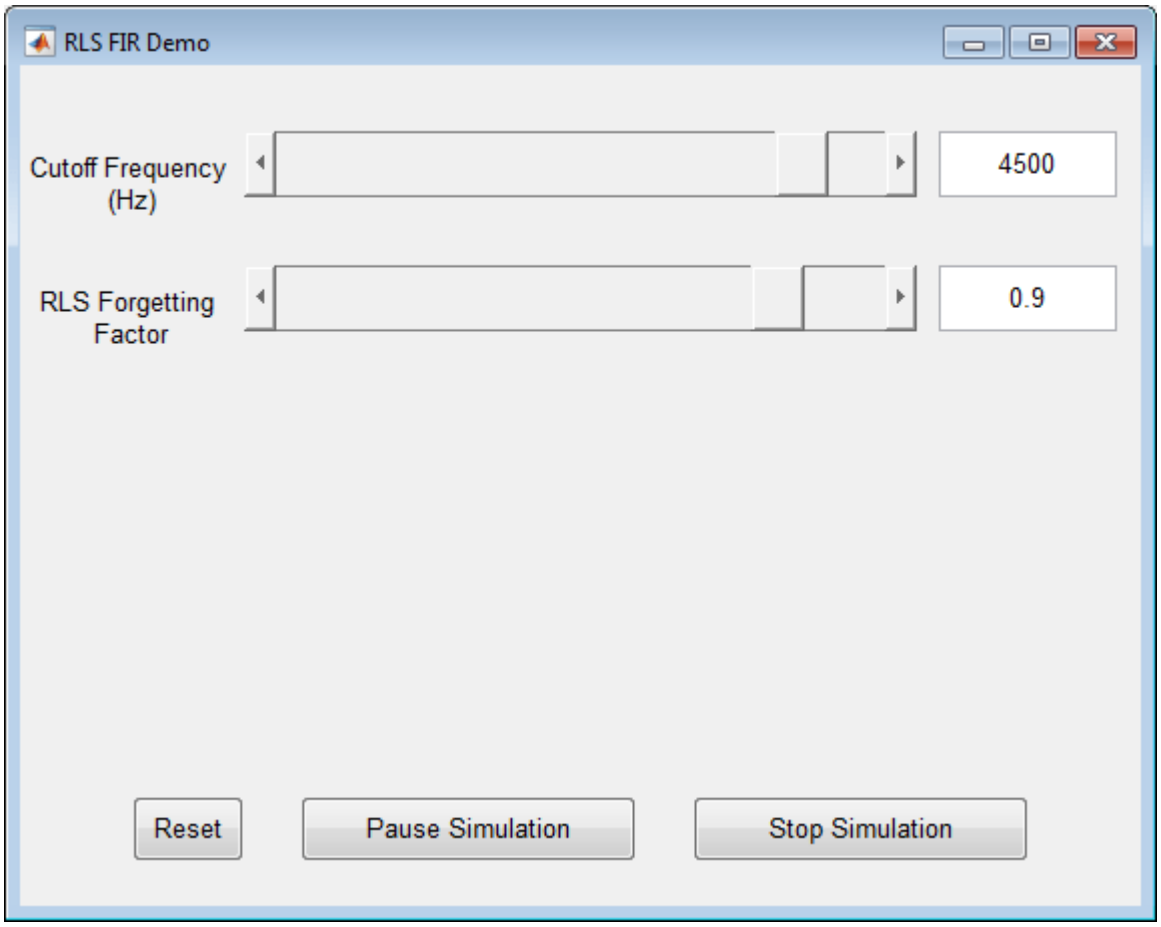
MATLAB Simulation

To verify the behavior of `RLSFilterSystemIDCompilerExampleApp`, run the function in MATLAB. It takes an optional input which is number of iteration steps. The default value is 300 iterations.

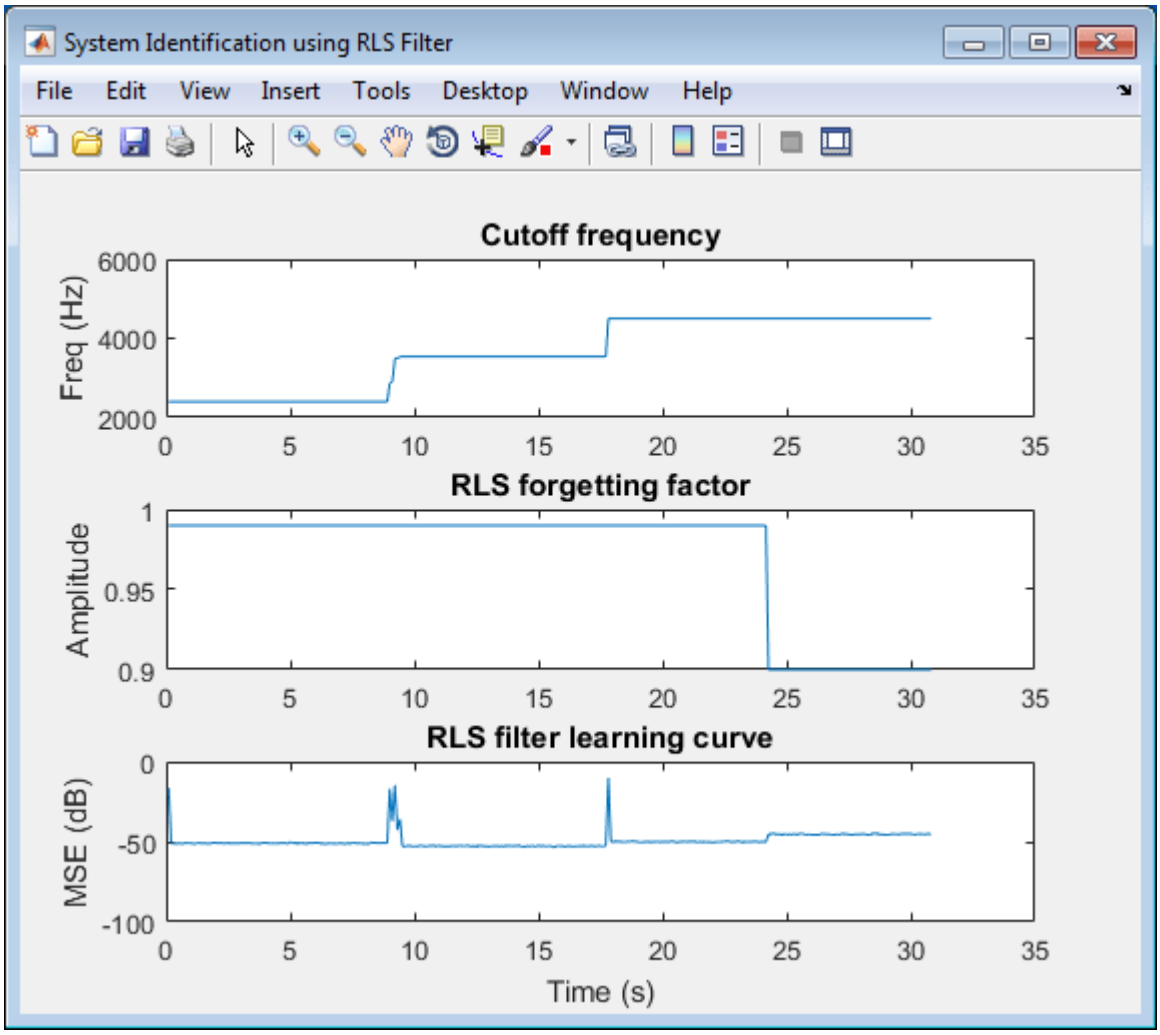
```
RLSFilterSystemIDCompilerExampleApp;
```

A user interface (UI) comes up which has two parameters that you can control:

1. Cutoff Frequency (Hz) - Cutoff frequency of the lowpass filter to be identified, specified as a scalar in the range [0, 5000] Hz.
2. RLS Forgetting Factor - Forgetting factor for the RLS filter used for system identification, specified as a scalar in the range [0, 1].



When the simulation finishes or when you click on `Stop Simulation` button, you will see a plot of the changes you made to these parameters and how it affected the mean-squared error (MSE) of the RLS filter.



Create a Temporary Directory for Compilation

Once you are satisfied with the function's simulation in MATLAB, you can compile the function. Before compiling, create a temporary directory in which you have write permissions. Copy the main MATLAB function and the associated helper files into this temporary directory.

```
compilerDir = fullfile(tempdir,'compilerDir'); % Name of temporary directory
if ~exist(compilerDir,'dir')
    mkdir(compilerDir); % Create temporary directory
end
curDir = cd(compilerDir);
copyfile(which('RLSFilterSystemIDCompilerExampleApp'));
copyfile(which('HelperRLSFilterSystemIdentificationSim'));
copyfile(which('HelperCreateParamTuningUI'));
copyfile(which('HelperUnpackUIData'));
```

Compile the MATLAB Function into a Standalone Application

In the temporary directory you just created, run `mcc` (MATLAB Compiler) command on the MATLAB function `RLSFilterSystemIDCompilerExampleApp`. `mcc` invokes the MATLAB Compiler which compiles the MATLAB function into a standalone executable that is saved in the current directory.

Use the `mcc` (MATLAB Compiler) function from MATLAB Compiler to compile `RLSFilterSystemIDCompilerExampleApp` into a standalone application. Specify the '-m' option to generate a standalone application, '-N' option to include only the directories in the path specified through the '-p' option.

```
mcc('-mN', 'RLSFilterSystemIDCompilerExampleApp', ...
    '-p', fullfile(matlabroot,'toolbox','dsp'));
```

```
DEMO Compiler license.
  The generated application will expire 30 days from today,
  on Mon Aug 10 18:05:54 2020.
```

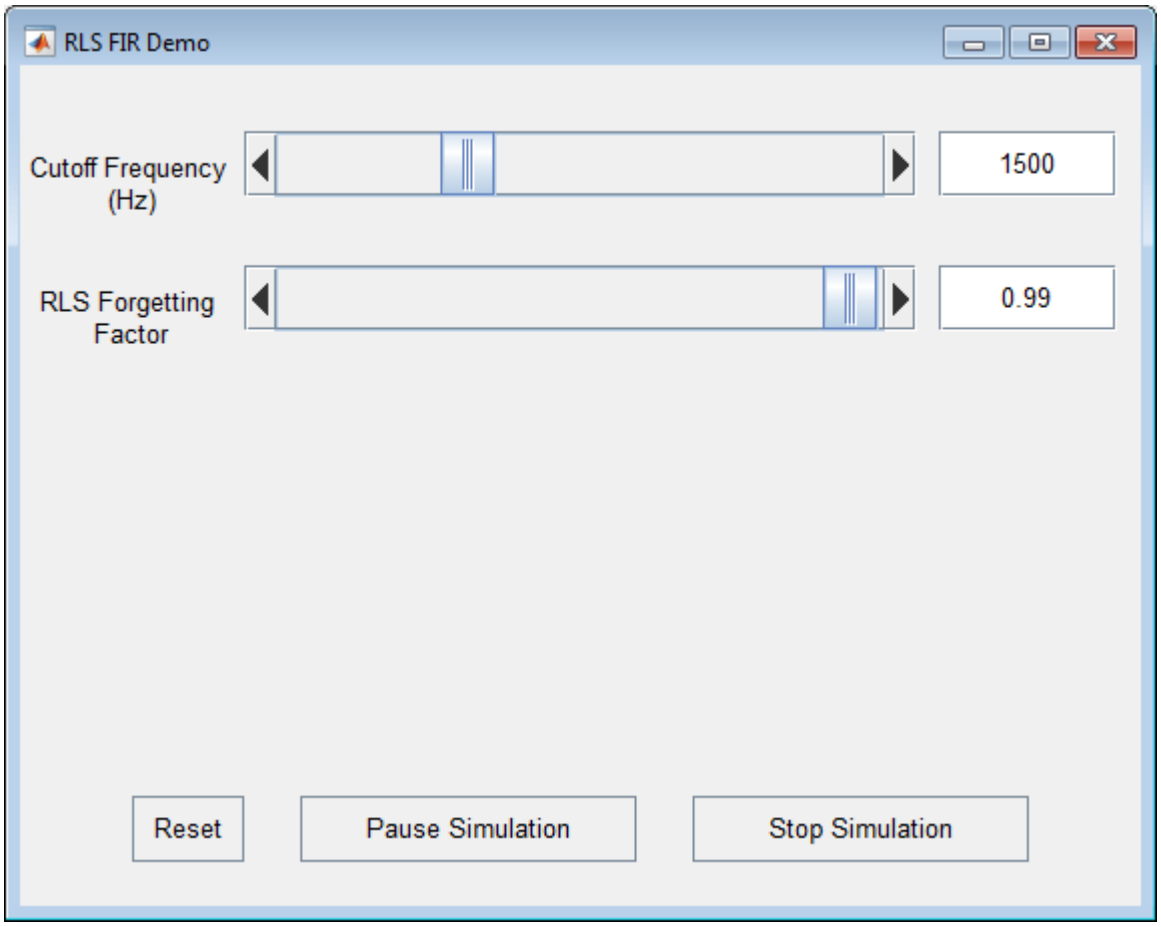
This step takes a few minutes to complete.

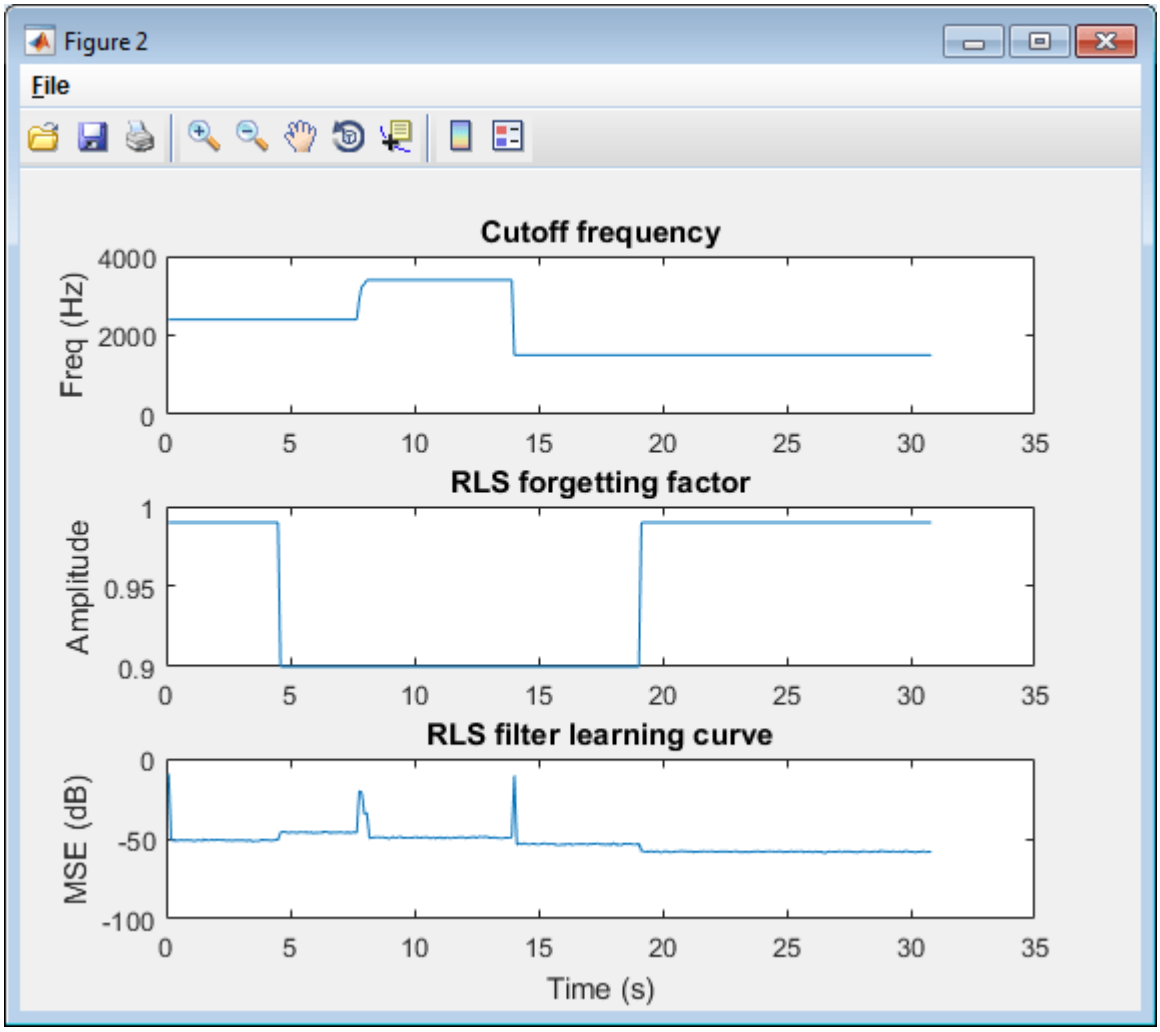
Run the Deployed Application

Use the `system` command to run the generated standalone application. Note that running the standalone application using the system command uses the current MATLAB environment and any library files needed from this installation of MATLAB. To deploy this application on a machine which does not have MATLAB installed, refer to “MATLAB Runtime” (MATLAB Compiler).

```
if ismac
    status = system(fullfile('RLSFilterSystemIDCompilerExampleApp.app', ...
        'Contents', 'MacOS', 'RLSFilterSystemIDCompilerExampleApp'));
else
    status = system(fullfile(pwd, 'RLSFilterSystemIDCompilerExampleApp'));
end
```

```
dyld: Library not loaded: @rpath/libmwlaunchermain.dylib
  Referenced from: /private/var/folders/dc/482hqt1x0_g33cty0z3_8v20000_8y/T/compilerDir/RLSFilte
  Reason: image not found
RLSFilterSystemIDCompilerExampleApp.app/Contents/MacOS/RLSFilterSystemIDCompilerExampleApp: Abor
```


RLS FIR Demo
Cutoff Frequency (Hz)
1500
RLS Forgetting Factor
0.99
Reset
Pause Simulation
Stop Simulation



Similar to the MATLAB example “System Identification Using RLS Adaptive Filtering” on page 4-264, running this executable application also launches a UI. The UI allows you to tune parameters and the results are reflected in the simulation instantly. For example, move the slider for the 'Cutoff frequency (Hz)' to the left while the simulation is running. You will see a drop in the plot for cutoff frequency and a corresponding fluctuation in the MSE of RLS filter. You can use the buttons on the UI to pause or stop the simulation.

Clean up Generated Files

After generating and deploying the executable, you can clean up the temporary directory by running the following in the MATLAB command prompt:

```
cd(curDir);
rmdir(compilerDir,'s');
```

Optimized Fixed-Point FIR Filters

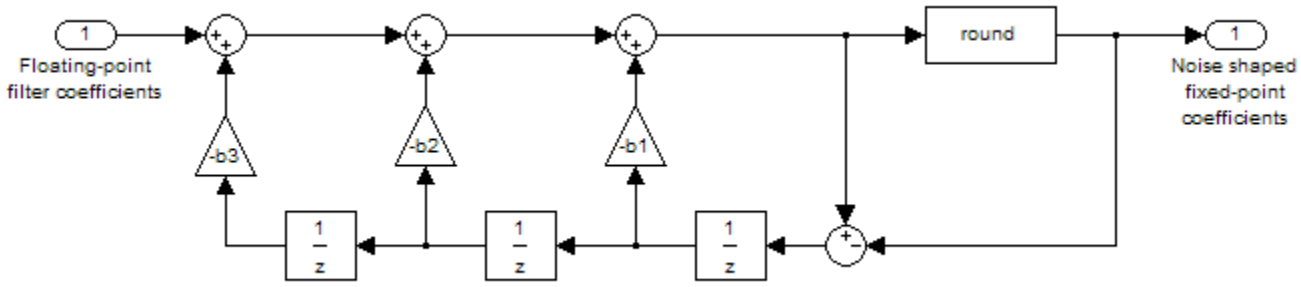
This example shows how to optimize fixed-point FIR filters. The optimization can refer to the characteristics of the filter response such as the stopband attenuation or the number of bits required to achieve a particular specification. This functionality is particularly useful for users targeting hardware that have a number of configurable coefficients of a specific wordlength and/or in cases typically found on ASICs and FPGAs where there is a large design space to explore. A hardware designer can usually trade off more coefficients for less bits or vice-versa to optimize for different ASICs or FPGAs.

This example illustrates various techniques based on the noise shaping procedure that yield optimized fixed-point FIR filter coefficients. The example shows how to:

- minimize coefficients wordlength,
- constrain coefficients wordlength,
- maximize stopband attenuation.

Theoretical Background

The noise shaping algorithm essentially moves the quantization noise out of a critical frequency band (usually the stopband) of a fixed-point FIR filter at the expense of increasing it in other bands. The block diagram below illustrates the process of noise shaping. Essentially, the filter coefficients are passed through a system that resembles a digital filter, but with a quantizer in the middle. The system is computing the quantization error for each coefficient, then passing the error through a simple IIR highpass filter defined by the b1, b2 and b3 coefficients. The 'round' block rounds the input to the nearest quantized value. After this, the quantized value is subtracted from the original floating point value. The values of the initial state in each delay block can be set to random noise between -LSB and +LSB.



The output of the system is the new, quantized and noise shaped filter coefficients. By repeating this procedure many times with different random initial states in the delay blocks, different filters can be produced.

Minimize Coefficients Wordlength

To begin with, we want to determine the minimum wordlength fixed-point FIR filter that meets a single-stage or multistage design specification. We take the example of a halfband filter with a normalized transition width of .08 and a stopband attenuation of 59 dB. A Kaiser window design yields 91 double-precision floating-point coefficients to meet the specifications.

```
TW = .08;   % Transition Width
Astop = 59; % Stopband Attenuation (dB)
```

```
f  = fdesign.halfband('TW,Ast',TW,Astop);
Hd = design(f,'kaiserwin');
```

To establish a baseline, we quantize the filter by setting its 'Arithmetic' property to 'fixed' and by iterating on the coefficients' wordlength until the minimum value that meets the specifications is found. Alternatively, we can use the minimizecoeffwl() to speed up the process. The baseline fixed-point filter contains 91 17-bit coefficients.

```
Hqbase = minimizecoeffwl(Hd,...
    'MatchRefFilter',true,'NoiseShaping',false, ...
    'Astoptol',0);       % 91 17-bit coefficients, Astop = 59.1 dB
```

The 17-bit wordlength is unappealing for many hardware targets. In certain situations we may be able to compromise by using only 16-bit coefficients. Notice however that the original specification is no longer strictly met since the maximum stopband attenuation of the filter is only 58.8 dB instead of the 59 dB desired.

```
Hq1 = copy(Hqbase);
Hq1.CoeffWordLength = 16; % 91 16-bit coefficients, Astop = 58.8 dB
m1 = measure(Hq1) %#ok
```

```
m1 =

Sample Rate      : N/A (normalized frequency)
Passband Edge    : 0.46
3-dB Point       : 0.49074
6-dB Point       : 0.5
Stopband Edge    : 0.54
Passband Ripple  : 0.017157 dB
Stopband Atten.  : 58.8741 dB
Transition Width : 0.08
```

Alternatively, we can set a tolerance to control the stopband error that is acceptable. For example, with a stopband tolerance of .15 dB we can save 3 bits and get a filter with 91 14-bit coefficients.

```
Hq2 = minimizecoeffwl(Hd,...
    'MatchRefFilter',true,'NoiseShaping',false, ...
    'Astoptol',.15);      % 91 14-bit coefficients, Astop = 58.8 dB
```

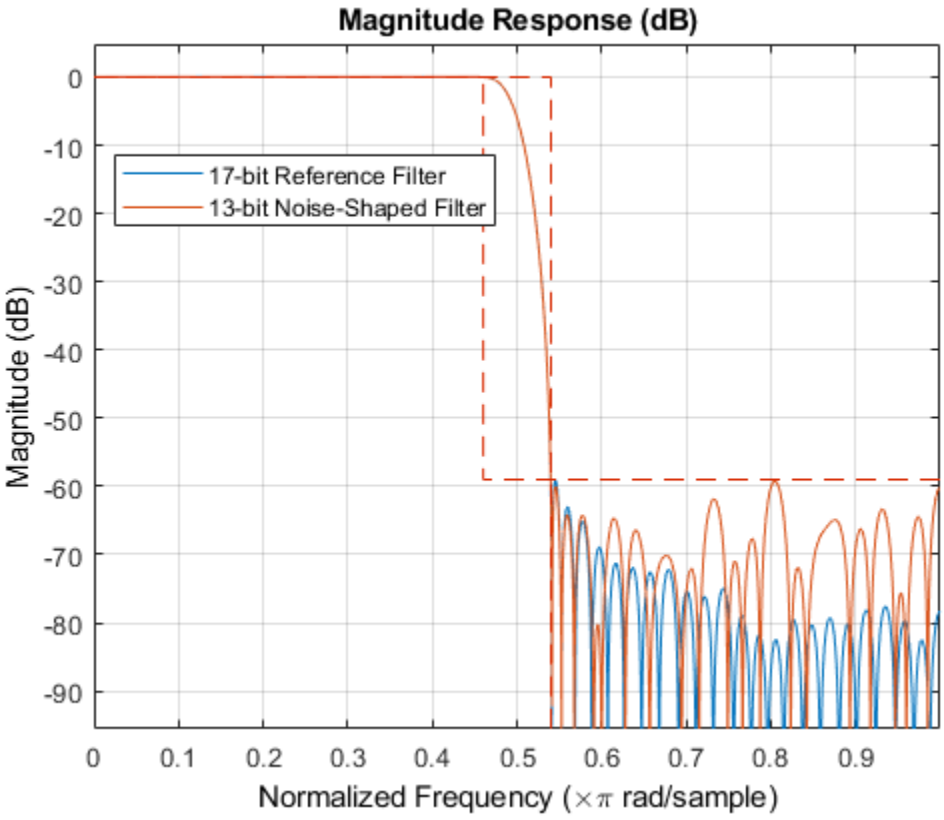
The saving in coefficients wordlength comes at the price of the fixed-point design no longer meeting the specifications. Tolerances can vary from one application to another but this strategy may have limited appeal in many situations. We can use another degree of freedom by relaxing the 'MatchRefFilter' constraint. By setting the 'MatchRefFilter' property to false, we no longer try to match the filter order (for minimum-order designs) or the filter transition width (for fixed order designs) of Hd. Allowing a re-design of the intermediate floating-point filter results in fixed-point filter that meets the specifications with 93 13-bit coefficients. Compared to the reference fixed-point designs, we saved 4 bits but ended up with 2 extra (1 non zero) coefficients.

```
Hq3 = minimizecoeffwl(Hd,...
    'MatchRefFilter',false,'NoiseShaping',false); % 93 13-bit coefficients
```

A better solution yet is to use noise shaping to maximize the stopband attenuation of the quantized filter. The noise shaping procedure is stochastic. You may want to experiment with the 'NTrials' option and/or to initialize RAND in order to reproduce the results below. Because 'MatchRefFilter' is false by default and 'NoiseShaping' is true, we can omit them. The optimized fixed-point filter meets

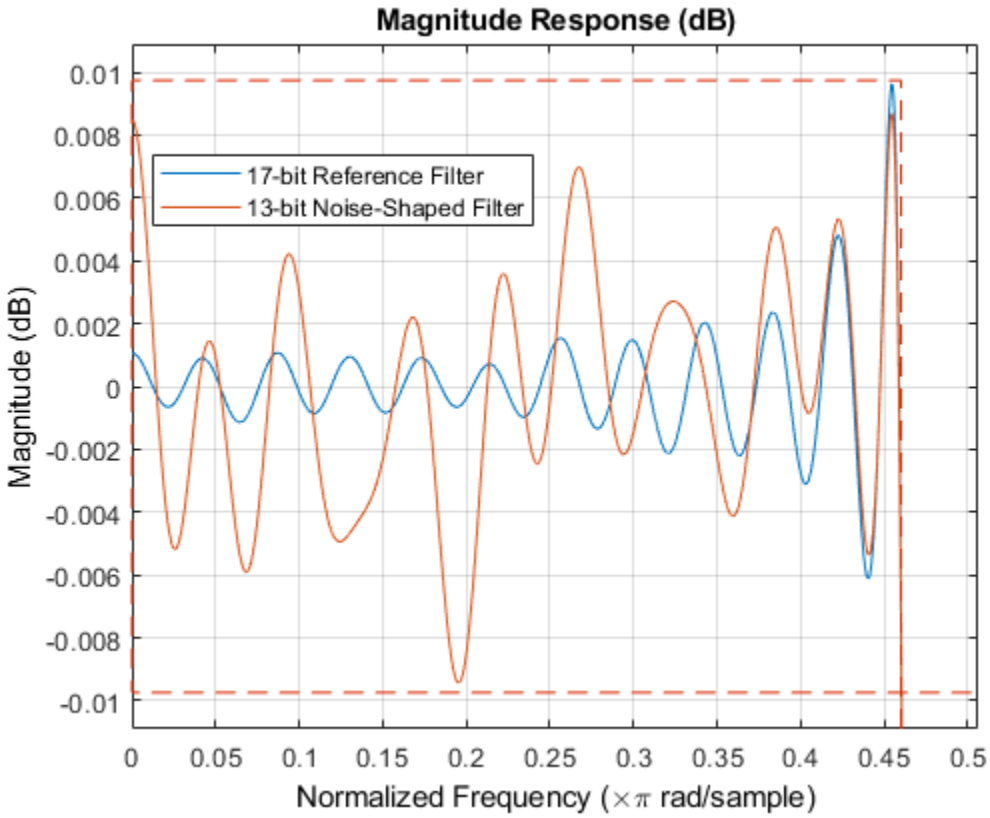
the specifications with 91 13-bit coefficients. This represents a saving of 4 bits over the reference fixed-point design with the same number of coefficients.

```
Hq4 = minimizecoeffwl(Hd,'Ntrials',10);  % 91 13-bit coefficients
hfvt = fvtool(Hqbase,Hq4,'ShowReference','off','Color','white');
legend(hfvt,'17-bit Reference Filter','13-bit Noise-Shaped Filter');
```



As a trade-off of the noise being shaped out of the stopband, the passband ripple of the noise-shaped filter increases slightly, which is usually not a problem. Also note that there is no simple relationship of passband ripple to frequency after the noise-shaping is applied.

```
axis([0 0.5060 -0.0109 0.0109])
```



Constrain Coefficients Wordlength

We have seen previously how we can trade-off more coefficients (or a larger transition width for designs with a fixed filter order) for a smaller coefficients wordlength by setting the 'MatchRefFilter' parameter of the minimizecoeffwl() method to 'false'. We now show how we can further control this trade-off in the case of a minimum-order design by taking the example of a multistage (3 stages) 8:1 decimator:

```
fm = fdesign.decimator(8,'lowpass','Fp,Fst,Ap,Ast',0.1,0.12,1,70);
Hm  = design(fm,'multistage','nstages',3);
```

Note: the following commands are computationally intensive and can take several minutes to run.

We first match the order of the floating-point design and obtain a noise-shaped fixed-point filter that meets the specifications with:

- 7 15-bit coefficients for the first stage,
- 10 13-bit coefficients for the second stage,
- 65 17-bit coefficients for the third stage.

```
Hmref = minimizecoeffwl(Hm,'MatchRefFilter',true);
```

By letting the filter order increase, we can reduce coefficients wordlengths to:

- 9 9-bit coefficients for the first stage,

- 10 12-bit coefficients for the second stage,
- 65 15-bit coefficients for the third stage.

```
Hq5 = minimizecoeffwl(Hm,'MatchRefFilter',false);
```

For better control of the final wordlengths, we can use the constraincoeffwl() method. For multistage designs, the wordlength for each stage can be constrained individually. For example, we constrain each stage to use 10, 12, and 14 bits respectively. The constrained design meets the specifications with:

- 8 10-bit coefficients for the first stage,
- 12 12-bit coefficients for the second stage,
- 68 14-bit coefficients for the third stage.

```
WL = [10 12 14];
Hqc = constraincoeffwl(Hm,WL);
```

Maximize Stopband Attenuation

When designing for shelf filtering engines (ASSPs) that have a number of configurable coefficients of a specific wordlength, it is desirable to maximize the stopband attenuation of a filter with a given order and a constrained wordlength. In the next example, we wish to obtain 69 dB of stopband attenuation with a 70th order halfband decimator while using 14 bits to represent the coefficients.

```
fh = fdesign.decimator(2,'halfband','N,Ast',70,69);
Hb1 = design(fh,'equiripple');
```

```
Warning: This design generates an MFILT object which will be removed in a future
release. To generate an equivalent System object, set the 'SystemObject'
parameter to true.
Syntax: Hs = design(D,...,'SystemObject',true)
```

If we simply quantize the filter with 14-bit coefficients, we get only 62.7 dB of attenuation.

```
Hb1.Arithmetic= 'fixed';
Hb1.CoeffWordLength = 14;
mb1 = measure(Hb1) %#ok
```

```
mb1 =

Sample Rate      : N/A (normalized frequency)
Passband Edge    : 0.44518
3-dB Point       : 0.48816
6-dB Point       : 0.5
Stopband Edge    : 0.55482
Passband Ripple  : 0.010552 dB
Stopband Atten.  : 62.7048 dB
Transition Width : 0.10963
```

By shaping the noise out of the stopband we can improve the attenuation by almost 1.5 dB to 64.18 dB but we still cannot meet the specifications.

```
Hbq1 = maximizestopband(Hb1,14);
mq1 = measure(Hbq1) %#ok
```

```
mq1 =

Sample Rate      : N/A (normalized frequency)
Passband Edge    : 0.44373
3-dB Point       : 0.48811
6-dB Point       : 0.5
Stopband Edge    : 0.55529
Passband Ripple  : 0.01064 dB
Stopband Atten.  : 64.1876 dB
Transition Width : 0.11156
```

The next step is to over design a floating-point filter with 80 dB of attenuation. We pay the price of the increased attenuation in the form of a larger transition width. The attenuation of the 14-bit non noise-shaped filter improved from 62.7 dB to 66.2 dB but is still not meeting the specifications.

```
fh.Astop = 80;
Hb2 = design(fh,'equiripple');
Hb2.Arithmetic= 'fixed';
Hb2.CoeffWordLength = 14;
mb2 = measure(Hb2) %#ok
```

```
Warning: This design generates an MFILT object which will be removed in a future
release. To generate an equivalent System object, set the 'SystemObject'
parameter to true.
Syntax: Hs = design(D,...,'SystemObject',true)

mb2 =

Sample Rate      : N/A (normalized frequency)
Passband Edge    : 0.43464
3-dB Point       : 0.48704
6-dB Point       : 0.5
Stopband Edge    : 0.56536
Passband Ripple  : 0.0076847 dB
Stopband Atten.  : 66.2266 dB
Transition Width : 0.13073
```

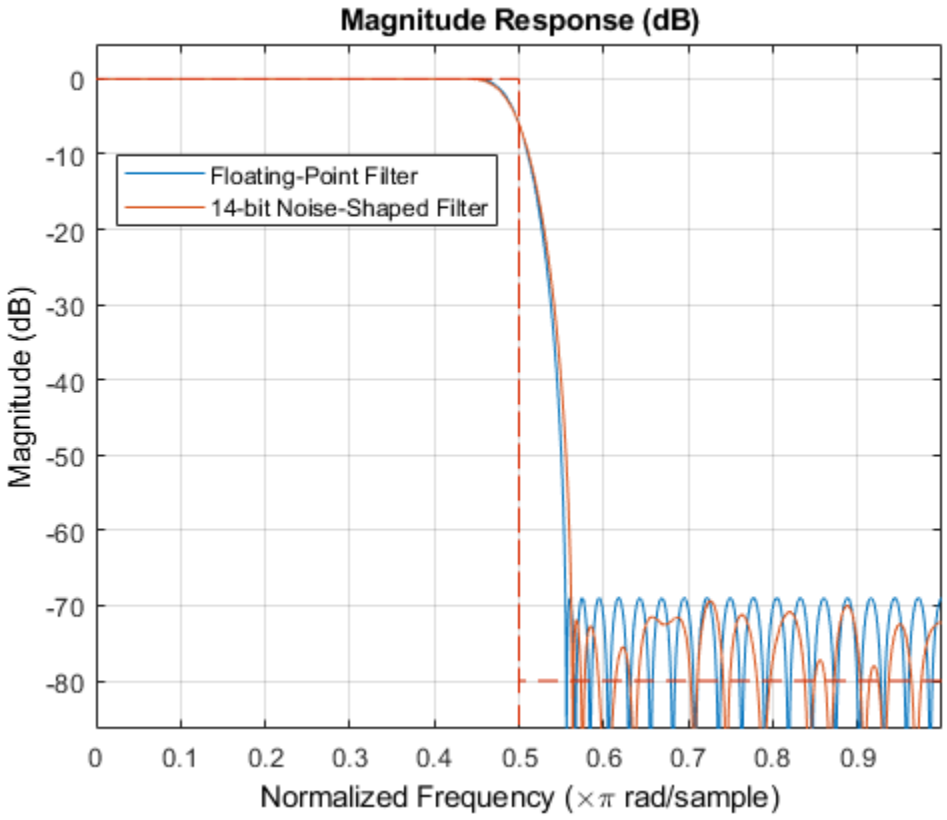
The noise shaping technique gives us a filter that finally meets the specifications by improving the stopband attenuation by more than 3 dB, from 66.2 dB to 69.4 dB.

```
Hbq2 = maximizestopband(Hb2,14);
mq2 = measure(Hbq2)  %#ok
```

```
mq2 =

Sample Rate      : N/A (normalized frequency)
Passband Edge    : 0.43584
3-dB Point       : 0.4871
6-dB Point       : 0.5
Stopband Edge    : 0.56287
Passband Ripple  : 0.0053253 dB
Stopband Atten.  : 69.4039 dB
Transition Width : 0.12703
```

The transition width of the fixed-point filter is increased compared to the floating-point design. This is the price to pay to get 69 dB of attenuation with only 14-bit coefficients as it would take 24-bit coefficients to match both the transition width and the stopband attenuation of the floating-point design.

```
close(hfvt);
hfvt = fvtool(reffilter(Hb1),Hbq2,'ShowReference','off','Color','white');
legend(hfvt,'Floating-Point Filter','14-bit Noise-Shaped Filter');
```




```
close(hfvt)
```

Summary

We have seen how the noise shaping technique can be used to minimize the coefficients wordlength of a single stage or multistage FIR fixed-point filter or how it can be used to maximize the stopband attenuation instead. We have also seen how bits can be traded for more coefficients in case of minimum order designs or for a larger transition width in case of designs with a fixed order.

References

Jens Jorgen Nielsen, Design of Linear-Phase Direct-Form FIR Digital Filters with Quantized Coefficients Using Error Spectrum Shaping Techniques, IEEE® Transactions on Acoustics, Speech, and Signal Processing, Vol. 37, No. 7, July 1989, pp. 1020--1026.

Alan V. Oppenheim and Ronald W. Schafer, Discrete-Time Signal Processing, 2nd edition, Prentice Hall, 1999, ISBN 0-13-754920-2.

Floating-Point to Fixed-Point Conversion of IIR Filters

This example shows how to use the Fixed-Point Converter App to convert an IIR filter from a floating-point to a fixed-point implementation. Second-order sections (also referred as biquadratic) structures work better when using fixed-point arithmetic than structures that implement the transfer function directly. We will show that the surest path to a successful "float-to-fixed" conversion consists of the following steps:

- Select a Second-Order Sections (SOS) structure, i.e. dsp.BiquadFilter
- Perform dynamic range analysis for each node of the filter, i.e. use a test bench approach with simulation minimum and simulation maximum instrumentation
- Compare alternative biquadratic scaling implementations and view quantization effects due to different choices using 'fvtool' and dsp.SpectrumAnalyzer for analysis and verification.

Introduction

An efficient way to implement an IIR filter is using a Second Order Section (SOS) Biquad filter structure. Suppose for example that we need to remove an interfering high frequency tone signal from a system. One way to achieve this is to use a lowpass filter design.

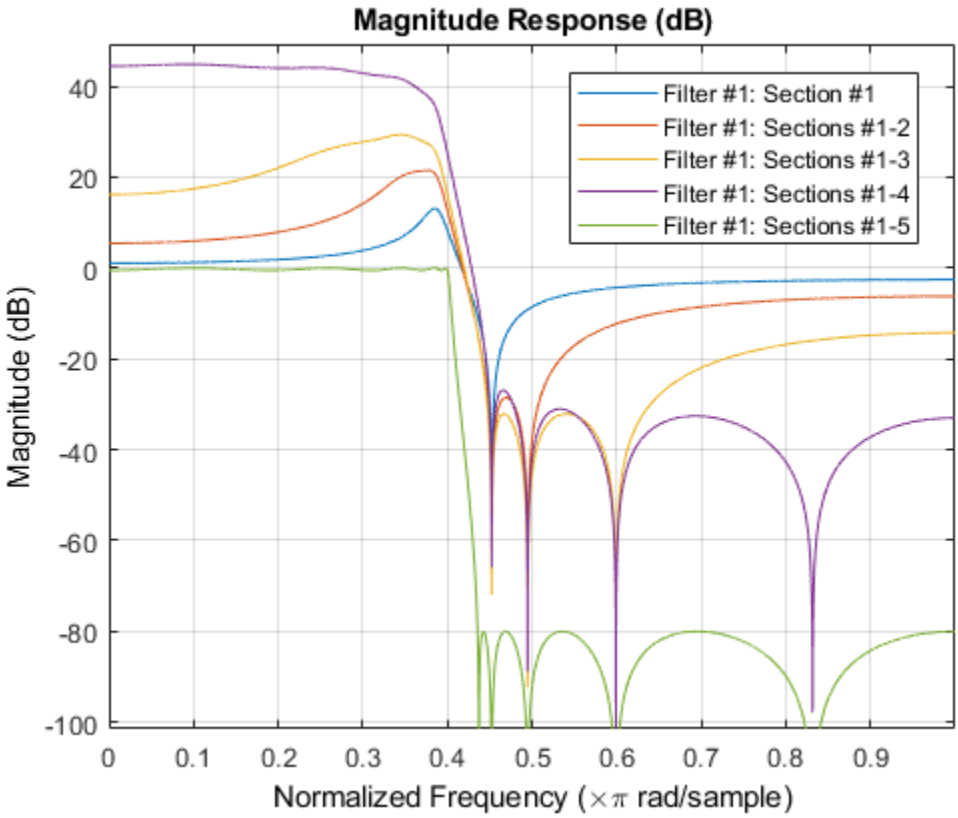
Design a Lowpass Elliptic Filter

Use a minimum-order lowpass Elliptic Direct-Form I design for the purpose of this example. The design specifications of the filter are:

- Passband frequency edge: 0.4*pi
- Stopband frequency edge: 0.45*pi
- Passband ripple: 0.5 dB
- Stopband attenuation: 80 dB

Visualize the cumulative filter response of all the second-order sections using the Filter Visualization Tool.

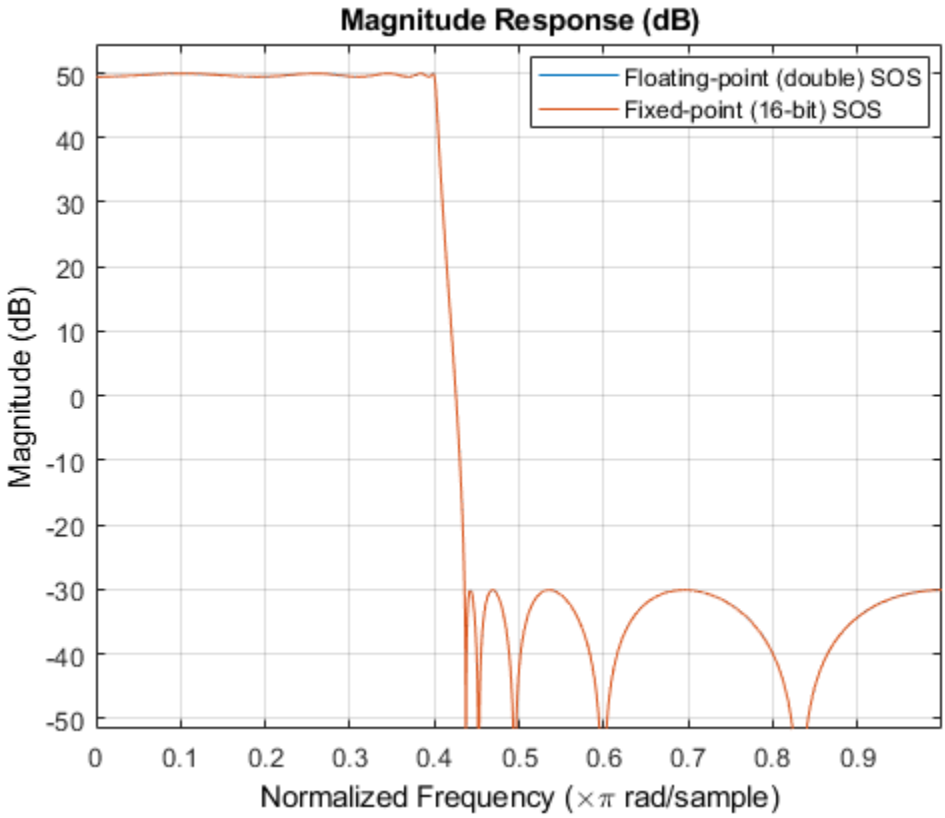
```
biquad = design(fdesign.lowpass('Fp,Fst,Ap,Ast',0.4,0.45,0.5,80), ...
    'ellip','FilterStructure','df1sos','SystemObject',true);
fvt = fvtool(biquad,'Legend','on');
fvt.SosviewSettings.View = 'Cumulative';
```



Obtain Filter Coefficients (SOS, B, A) for Float vs. Fixed Comparison

Note that the SOS filter coefficient values lead to a nearly identical filter response (whether they are double or 16-bit fixed-point values).

```
sosMatrix = biquad.SOSMatrix;
sclValues = biquad.ScaleValues;
fvt_comp = fvtool(sosMatrix,fi(sosMatrix,1,16));
legend(fvt_comp,'Floating-point (double) SOS','Fixed-point (16-bit) SOS');
b = repmat(sclValues(1:(end-1)),1,3) .* sosMatrix(:,(1:3));
a = sosMatrix(:,(5:6));
num = b'; % matrix of scaled numerator sections
den = a'; % matrix of denominator sections
```

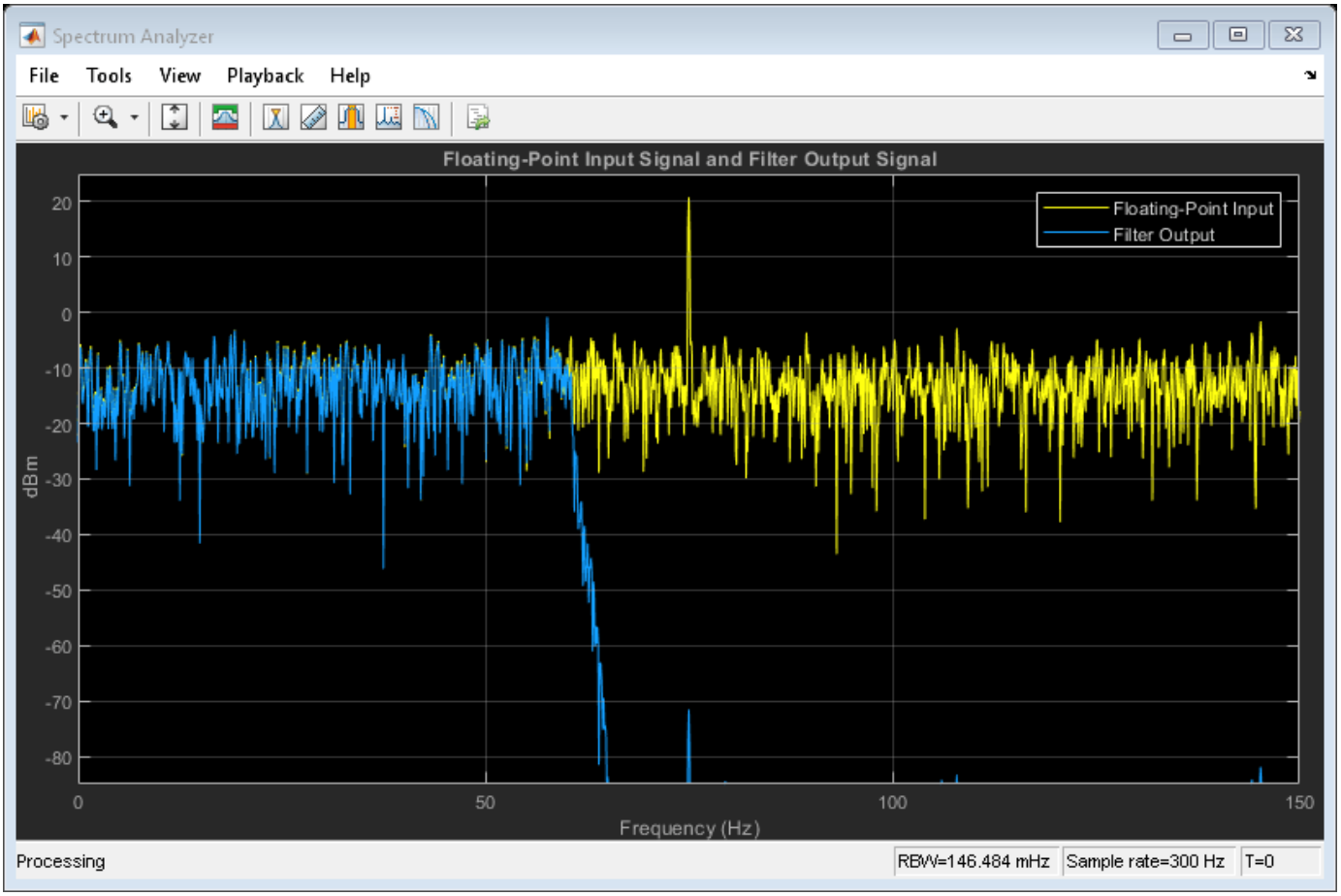



```
close(fvt);      % clean up
close(fvt_comp); % clean up
```

Default Floating-Point Operation

Verify the filter operation by streaming some data through it and viewing its input-output response. First try filtering a (floating-point) pseudorandom noise signal (sampled at 300 Hz) with an interfering additive high-frequency tone, to see if the tone is removed. This will also serve later as our reference signal for a test bench.

```
Wo          = 75/(300/2); % 75 Hz tone; system running at 300 Hz
inp_len     = 4000;       % Number of input samples (signal length)
inp_itf     = 0.5 .* sin((pi*Wo) .* (0:(inp_len-1))'); % tone interferer
scope = dsp.SpectrumAnalyzer('SampleRate',300, ...
    'PlotAsTwoSidedSpectrum',false, ...
    'ShowLegend',true,'YLimits',[-85 25], ...
    'Title','Floating-Point Input Signal and Filter Output Signal', ...
    'ChannelNames',{'Floating-Point Input','Filter Output'});
rng(12345); % seed the rng for repeatable results
biquadLPFiltFloat = dsp.BiquadFilter('SOSMatrixSource','Input port', ...
    'ScaleValuesInputPort',false);
for k = 1:10
    inp_sig = rand(inp_len,1) - 0.5; % random values in range (-0.5, 0.5)
    input = inp_sig + inp_itf; % combined input signal, range (-1.0, 1.0)
    out_1 = biquadLPFiltFloat(input,num,den); % filter
    scope([input,out_1]) % visualize input and filtered output
end
```

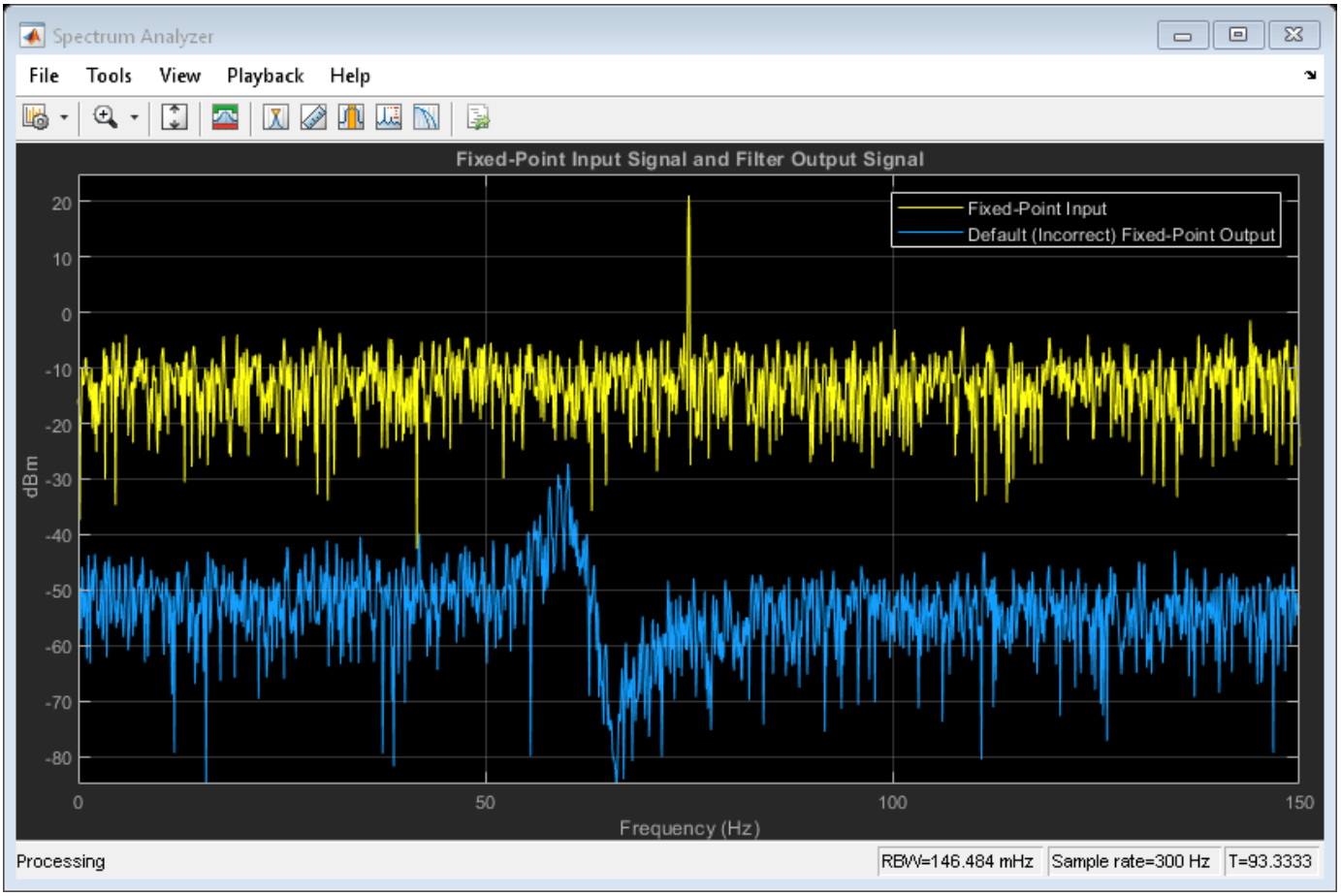



```
clear scope biquadLPFiltFloat; % clean up
```

Default Fixed-Point Operation

Now run some fixed-point data through the filters, using the object default settings. Note that the default fixed-point behavior leads to incorrect results.

```
scope = dsp.SpectrumAnalyzer('SampleRate',300, ...
    'PlotAsTwoSidedSpectrum',false, ...
    'ShowLegend',true,'YLimits',[-85 25], ...
    'Title','Fixed-Point Input Signal and Filter Output Signal', ...
    'ChannelNames', ...
    {'Fixed-Point Input','Default (Incorrect) Fixed-Point Output'});
rng(12345); % seed the rng for repeatable results
bqLPFiltFixpt = dsp.BiquadFilter('SOSMatrixSource','Input port', ...
    'ScaleValuesInputPort',false);
for k = 1:10
    inp_sig = rand(inp_len,1) - 0.5;  % random values in range (-0.5, 0.5)
    inputFi = fi(inp_sig + inp_itf, 1, 16, 15); % signal range (-1.0, 1.0)
    out_2 = bqLPFiltFixpt(inputFi,fi(num,1,16),fi(den,1,16));
    scope([inputFi,out_2]) % visualize
end
```




```
clear scope bqLPFiltFixpt; % clean up
```

Convert Floating-Point Biquad IIR Filter Function to Fixed-Point

Instead of relying on the default fixed-point settings of the object, convert the object to fixed-point using the Fixed-Point Converter App. This approach yields more visibility and control over individual fixed-point types within the filter implementation, and leads to more correct fixed-point operation.

To first prepare for using the Fixed-Point Converter App, create a function to convert, setting all filter data type choices to ''Custom'':

```
type myIIRLowpassBiquad;


function output = myIIRLowpassBiquad(inp,num,den)
%myIIRNotchBiquad Biquad lowpass filter implementation
%   Used as part of a MATLAB Fixed Point Converter App example.

%   Copyright 2016 The MathWorks, Inc.
persistent bqLPFilter;
if isempty(bqLPFilter)
    bqLPFilter = dsp.BiquadFilter(        ...
        'SOSMatrixSource', 'Input port',  ...
        'ScaleValuesInputPort', false,    ...
        'SectionInputDataType',           'Custom', ...
```

```
        'SectionOutputDataType',          'Custom', ...
        'NumeratorProductDataType',       'Custom', ...
        'DenominatorProductDataType',     'Custom', ...
        'NumeratorAccumulatorDataType',   'Custom', ...
        'DenominatorAccumulatorDataType', 'Custom', ...
        'StateDataType',                  'Custom', ...
        'OutputDataType',                 'Custom');
end
output = bqLPFilter(inp, num, den);
end
```

Create Test Bench Script

Create a test bench to simulate and collect instrumented simulation minimum and simulation maximum values for all of our data type controlled signal paths. This will allow the tool to later propose autoscaled fixed-point settings. Use parts of the code above as a test bench script, starting with floating-point inputs and verifying the test bench before simulating and collecting minimum and maximum data. Afterward, use the Fixed-Point Converter App to convert the floating-point function implementation to a fixed-point function implementation.

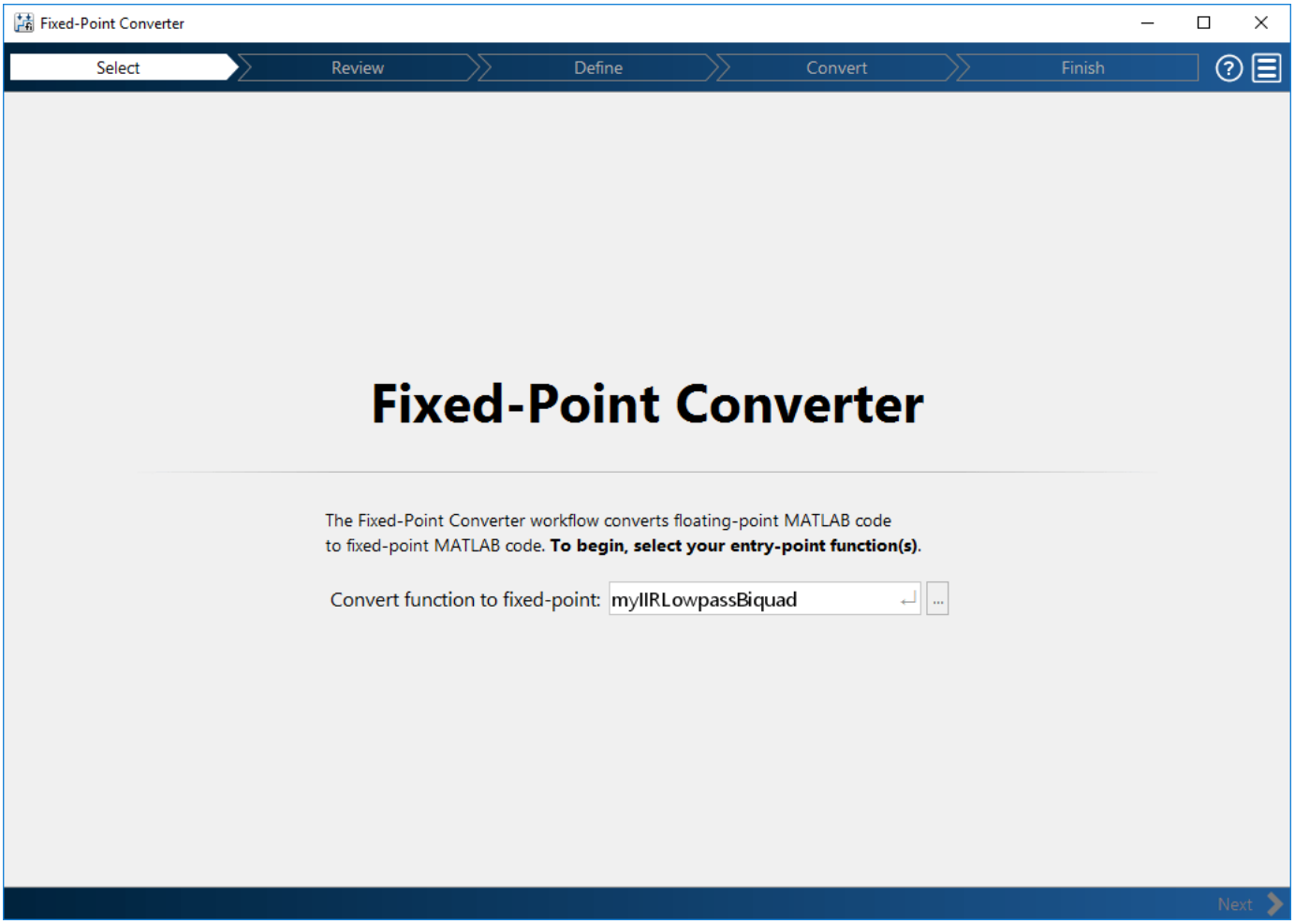
```
type myIIRLowpassBiquad_tb.m;
```

```
%% Test Bench for myIIRLowpassBiquad.m

%   Copyright 2016 The MathWorks, Inc.

%% Pre-designed filter (coefficients stored in MAT file):
% f = design(fdesign.lowpass('Fp,Fst,Ap,Ast',0.4,0.45,0.5,80), ...
%            'ellip', 'FilterStructure', 'df1sos','SystemObject',true);
% sosMatrix = f.SOSMatrix;
% sclValues = f.ScaleValues;
% b = repmat(sclValues(1:(end-1)),1,3) .* sosMatrix(:,(1:3));
% a = sosMatrix(:,(5:6));
% num = b';
% den = a';
% save('myIIRLowpassBiquadDesign.mat', 'b', 'a', 'num', 'den');
load('myIIRLowpassBiquadDesign.mat');

%% Interference signal, using values in range (-0.5, 0.5)
Wo      = 75/(300/2); % 75 Hz tone; system running at 300 Hz
inp_len = 4000; sinTvec = (0:(inp_len-1))';
inp_itf = 0.5 .* sin((pi*Wo) .* sinTvec);

%% Filtering and visualization
% Filter an input signal, including an interference
% tone, to see if the tone is successfully removed.
rng(12345); % seed the rng for repeatable results
scope = dsp.SpectrumAnalyzer('SampleRate',300,...
    'PlotAsTwoSidedSpectrum',false,'ShowLegend',true,'YLimits',[-125 25],...
    'Title','Input Signal and Filter Output Signal', ...
    'ChannelNames', {'Input', 'Filter Output'});

for k = 1:10
    inp_sig = rand(inp_len,1) - 0.5; % random values in range (-0.5, 0.5)
    inp = inp_sig + inp_itf;
    out_1 = myIIRLowpassBiquad(inp,num,den); % filter
```

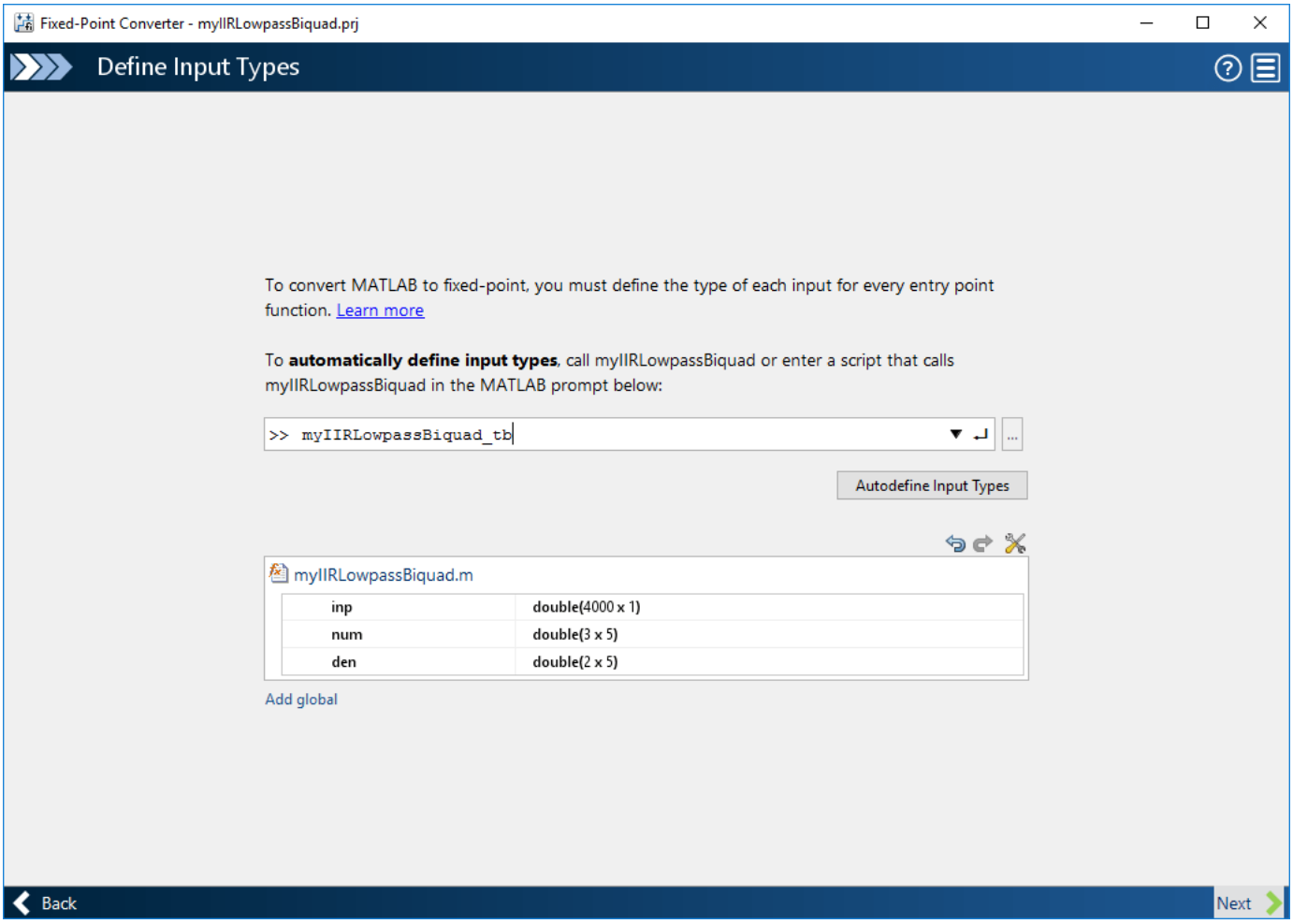
```
    scope([inp,out_1]); % visualize
end
```

Convert to Fixed-Point Using the Fixed-Point Converter App

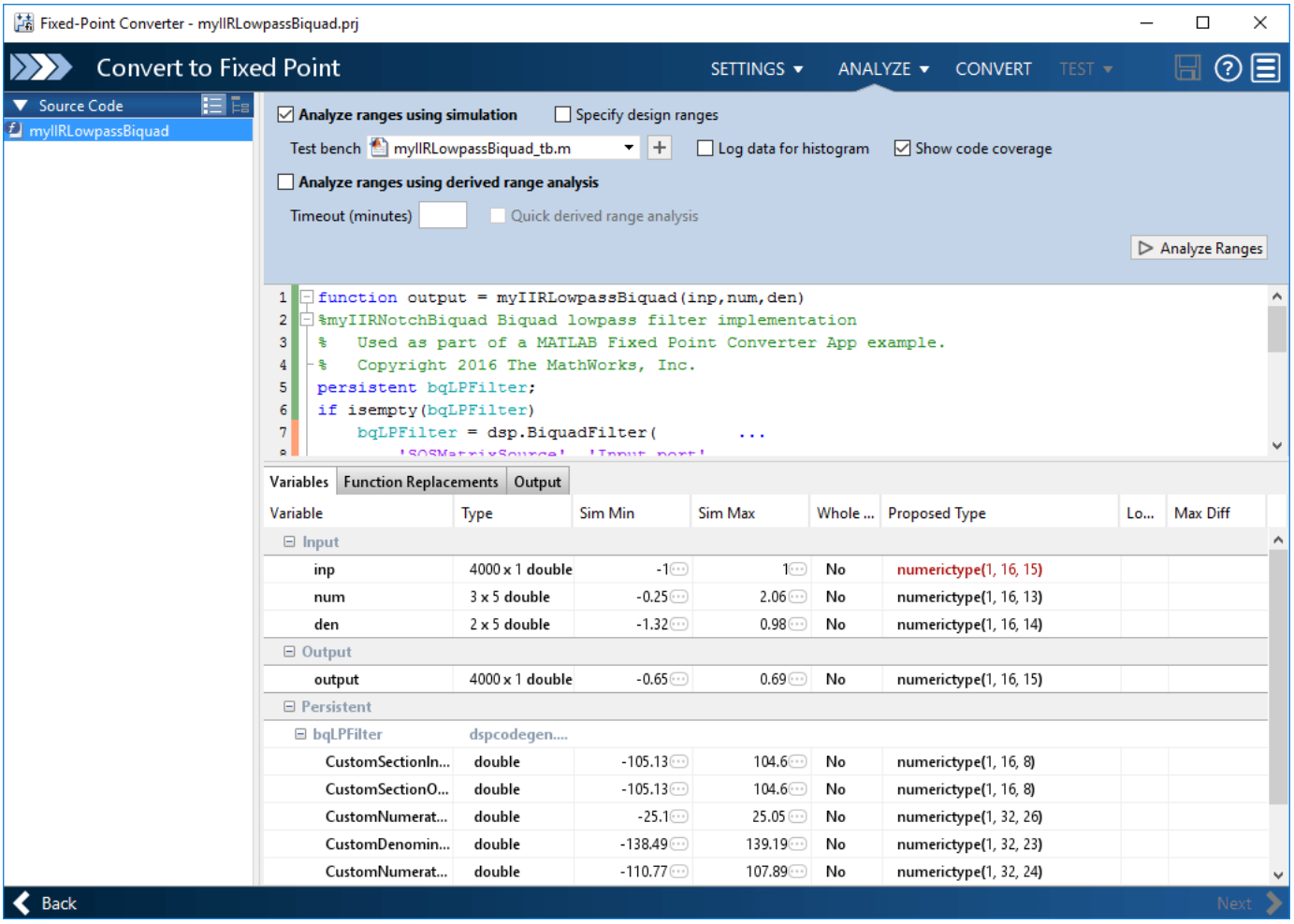
- Launch the Fixed-Point Converter App. There are two ways to launch the tool: via the MATLAB APPS menu or via the command 'fixedPointConverter'.
- Enter the function to convert in the Entry-Point Functions field.



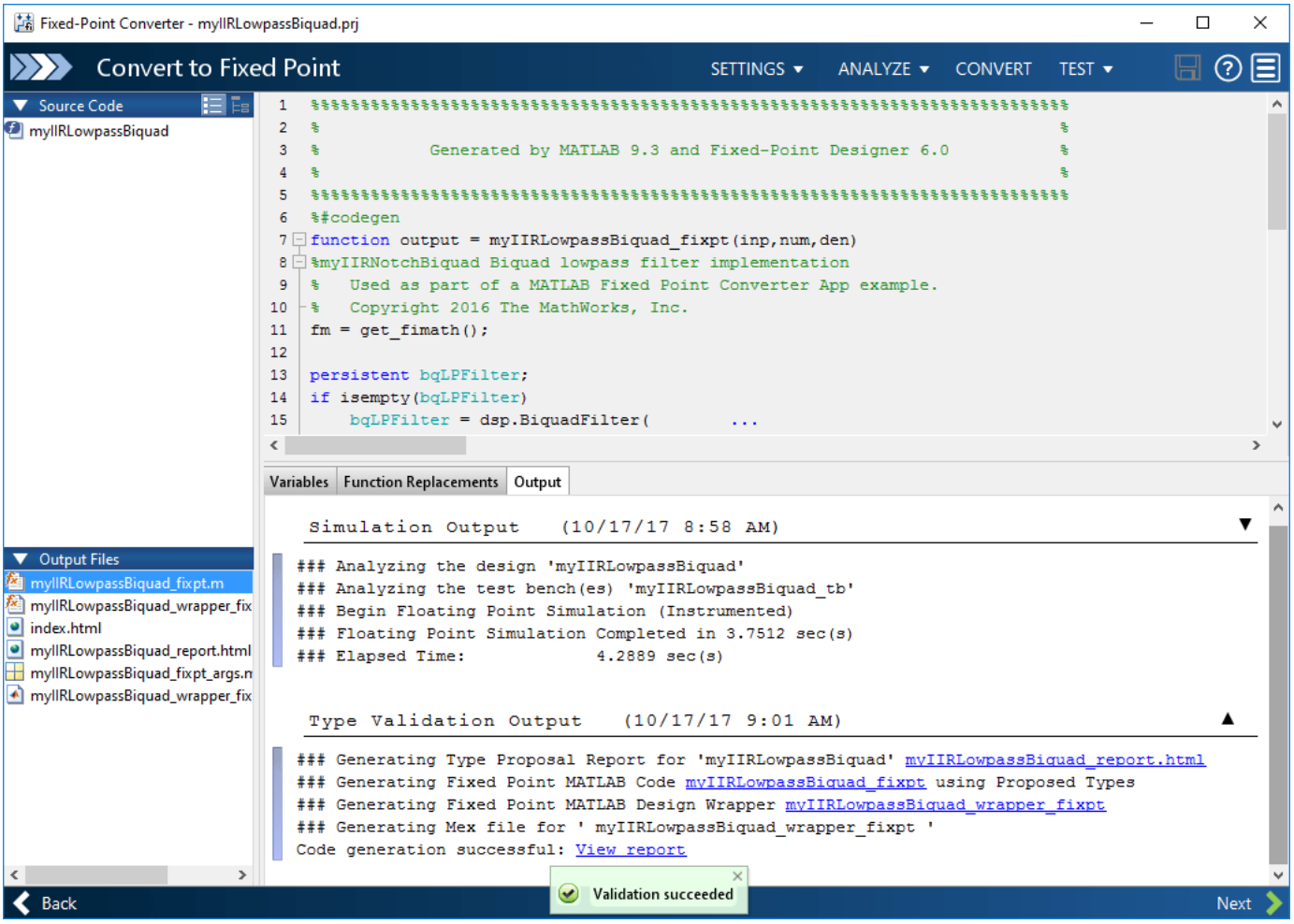
- Define inputs by entering the test bench script name.



- Click 'Analyze' to collect ranges by simulating the test bench.
- Observe the collected 'Sim Min', 'Sim Max', and 'Proposed Type' values.



- Make adjustments to the 'Proposed Type' fields as needed.
- Click 'Convert' to generate the fixed-point code and view the report.



Resulting Fixed-Point MATLAB Implementation

The generated fixed-point function implementation is as follows:

```
type myIIRLowpassBiquad_fixpt;
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                               %
%           Generated by MATLAB 9.3 and Fixed-Point Designer 6.0                %
%                                                                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%#codegen
function output = myIIRLowpassBiquad_fixpt(inp,num,den)
%myIIRNotchBiquad Biquad lowpass filter implementation
%   Used as part of a MATLAB Fixed Point Converter App example.
%   Copyright 2016 The MathWorks, Inc.
fm = get_fimath();

persistent bqLPFilter;
if isempty(bqLPFilter)
    bqLPFilter = dsp.BiquadFilter(        ...
        'SOSMatrixSource', 'Input port',  ...
```

```
        'ScaleValuesInputPort', false,    ...
        'SectionInputDataType',           'Custom', ...
        'SectionOutputDataType',          'Custom', ...
        'NumeratorProductDataType',       'Custom', ...
        'DenominatorProductDataType',     'Custom', ...
        'NumeratorAccumulatorDataType',   'Custom', ...
        'DenominatorAccumulatorDataType', 'Custom', ...
        'StateDataType',                  'Custom', ...
        'OutputDataType',                 'Custom' , 'CustomSectionInputDataType', numerictype([]
end
output = fi(bqLPFilter(inp, num, den), 1, 16, 15, fm);
end


function fm = get_fimath()
    fm = fimath('RoundingMethod', 'Floor',...
         'OverflowAction', 'Wrap',...
         'ProductMode','FullPrecision',...
         'MaxProductWordLength', 128,...
         'SumMode','FullPrecision',...
         'MaxSumWordLength', 128);
end
```

Automation of Fixed-Point Conversion Using the Command Line API

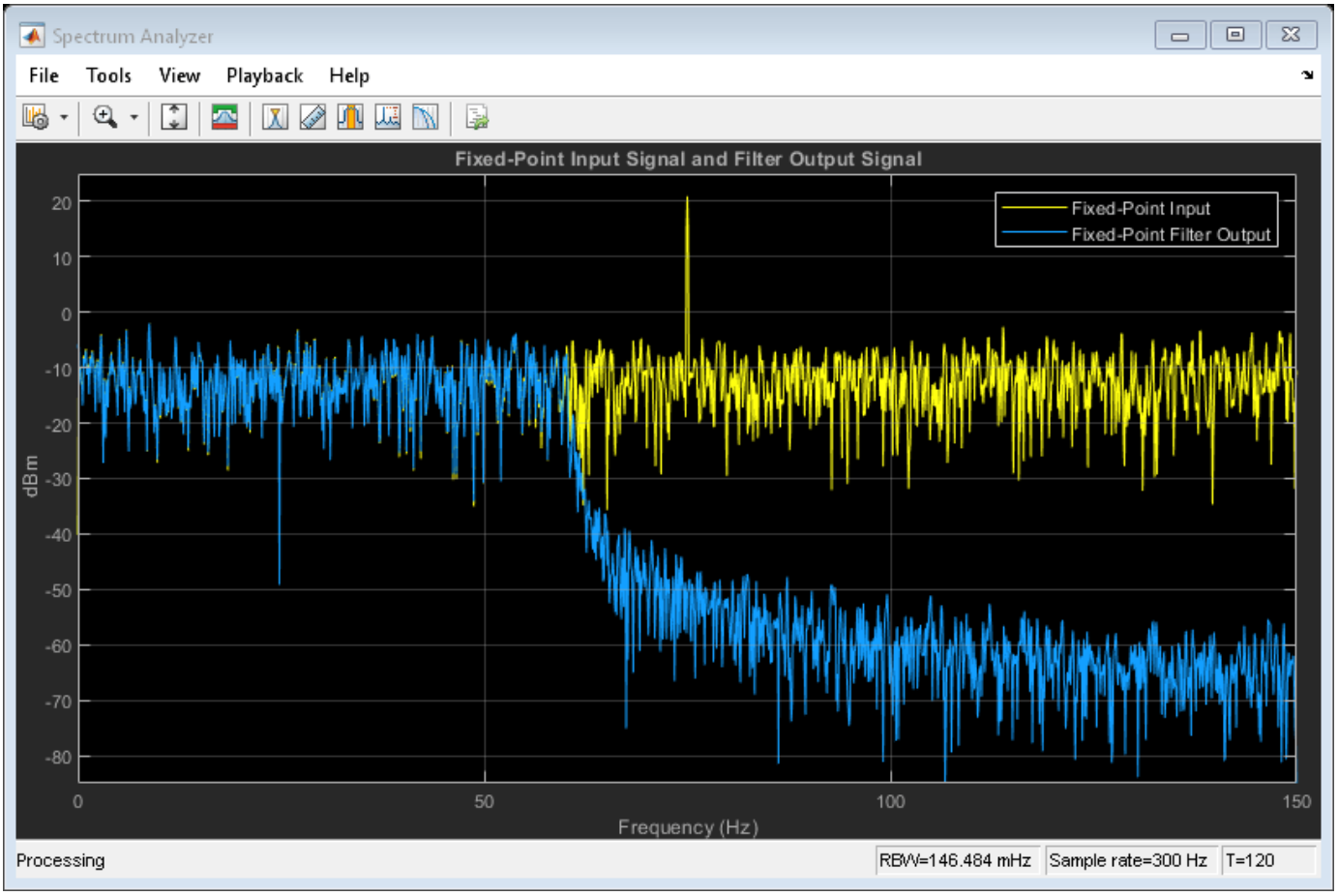
Alternatively the fixed-point conversion may be automated using the command line API:

```
type myIIRLowpassF2F_prj_script;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Script generated from project 'myIIRLowpassBiquad.prj' on 16-Oct-2014.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Create configuration object of class 'coder.FixPtConfig'.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
cfg = coder.config('fixpt');

cfg.TestBenchName = { sprintf('S:\\Work\\15aFeatureExamples\\biquad_notch_f2f\\myIIRLowpassBiquad
cfg.DefaultWordLength = 16;
cfg.LogIOForComparisonPlotting = true;
cfg.TestNumerics = true;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Define argument types for entry-point 'myIIRLowpassBiquad'.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
ARGS = cell(1,1);
ARGS{1} = cell(3,1);
ARGS{1}{1} = coder.typeof(0,[4000   1]);
ARGS{1}{2} = coder.typeof(0,[3 5]);
ARGS{1}{3} = coder.typeof(0,[2 5]);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Invoke MATLAB Coder.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
codegen -float2fixed cfg myIIRLowpassBiquad -args ARGS{1}
```

Test the Converted MATLAB Fixed-Point Implementation

Run the converted fixed-point function and view input-output results.

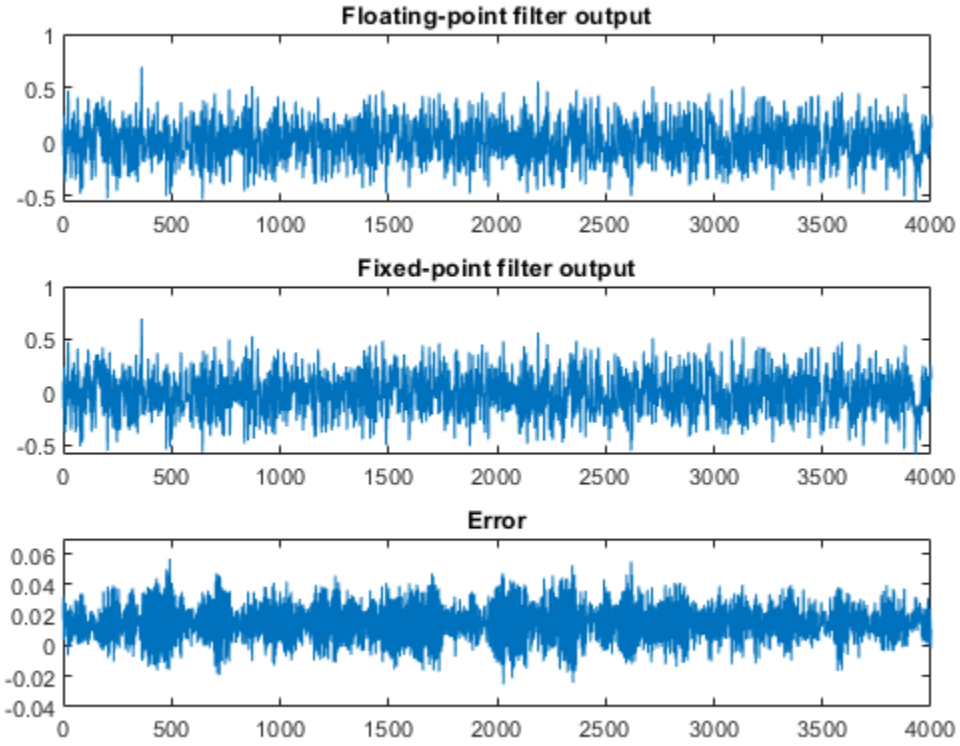
```
scope = dsp.SpectrumAnalyzer('SampleRate',300, ...
    'PlotAsTwoSidedSpectrum',false, ...
    'ShowLegend',true,'YLimits',[-85 25], ...
    'Title','Fixed-Point Input Signal and Filter Output Signal', ...
    'ChannelNames',{'Fixed-Point Input','Fixed-Point Filter Output'});
rng(12345); % seed the rng for repeatable results
for k = 1:10
    inp_sig = rand(inp_len,1) - 0.5;  % random values in range (-0.5, 0.5)
    inputFi = fi(inp_sig + inp_itf, 1, 16, 15); % signal range (-1.0, 1.0)
    out_3 = myIIRLowpassBiquad_fixpt(inputFi,fi(num,1,16),fi(den,1,16));
    scope([inputFi,out_3]) % visualize
end
```




```
clear scope; % clean up
```

The error between the floating-point and fixed-point outputs is shown on the next plot. The error seems rather high. The reason for these differences in output values is dominated by the choice of scaling and ordering of the second-order sections. In the next section, we illustrate a way to reduce this error earlier in the implementation.

```
fig = figure;
subplot(3,1,1);
plot(out_1);
title('Floating-point filter output');
subplot(3,1,2);
plot(out_3);
title('Fixed-point filter output');
subplot(3,1,3);
plot(out_1 - double(out_3));
axis([0 4000 -4e-2 7e-2]);
title('Error');
```

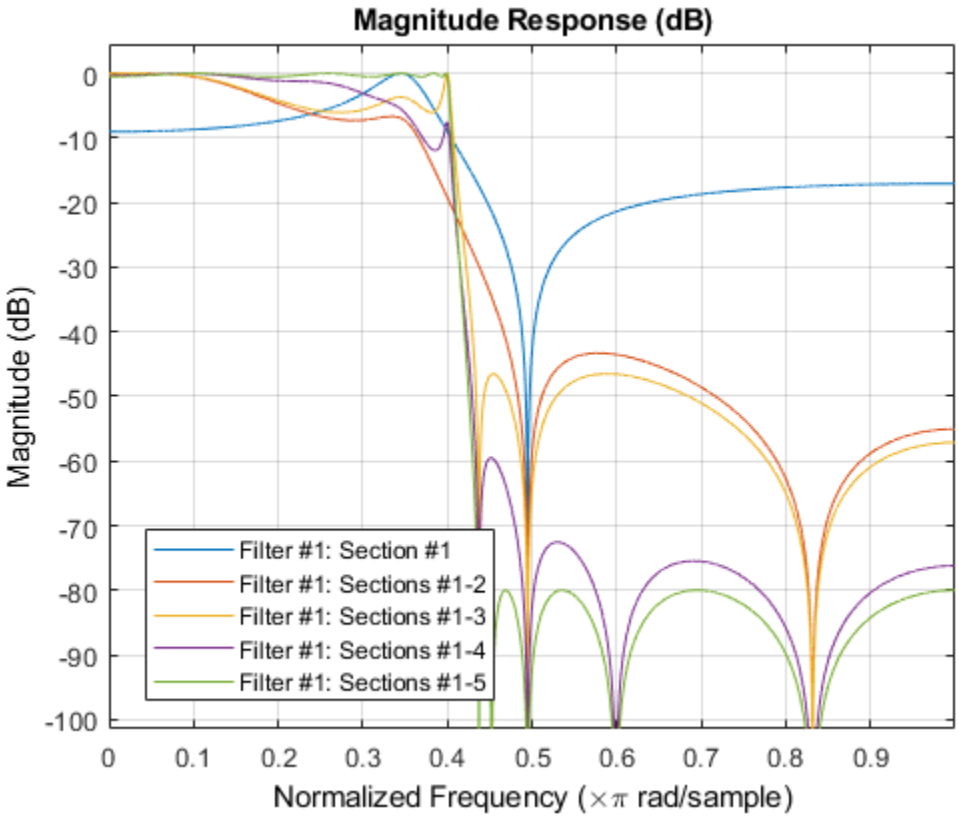



```
close(fig); % clean up
```

Re-designing the Elliptic Filter Using Infinity-Norm Scaling

Elliptic filter designs have the characteristic of being relatively well scaled when using the 'Linf' second-order section scaling (i.e., infinity norm). Using this approach often leads to smaller quantization errors.

```
biquad_Linf = design(fdesign.lowpass('Fp,Fst,Ap,Ast',0.4,0.45,0.5,80), ...
    'ellip','FilterStructure','df1sos', ...
    'SOSScaleNorm','Linf','SystemObject',true);
fvt_Linf = fvtool(biquad_Linf,'Legend','on');
fvt_Linf.SosviewSettings.View = 'Cumulative';
```

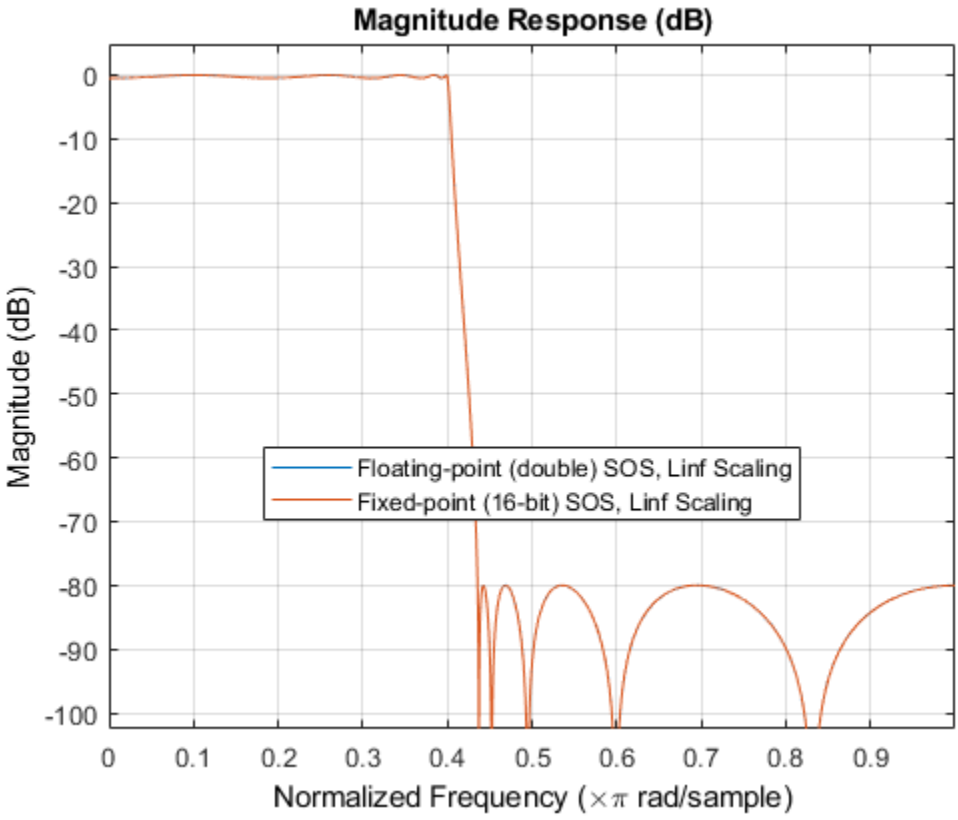


Notice that none of the cumulative internal frequency responses, measured from the input to the filter to the various states of each section, exceed 0 dB. Thus, this design is a good candidate for a fixed-point implementation.

Obtain Linf-norm Filter Coefficients for Float vs. Fixed Comparison

Note that the SOS filter coefficient values lead to a nearly identical filter response (whether they are double or 16-bit fixed-point values).

```
sosMtrLinf = biquad_Linf.SOSMatrix;
sclValLinf = biquad_Linf.ScaleValues;
fvt_comp_Linf = fvtool(sosMtrLinf,fi(sosMtrLinf,1,16));
legend(fvt_comp_Linf,'Floating-point (double) SOS, Linf Scaling', ...
    'Fixed-point (16-bit) SOS, Linf Scaling');
bLinf = repmat(sclValLinf(1:(end-1)),1,3) .* sosMtrLinf(:,(1:3));
aLinf = sosMtrLinf(:,(5:6));
numLinf = bLinf'; % matrix of scaled numerator sections
denLinf = aLinf'; % matrix of denominator sections
```

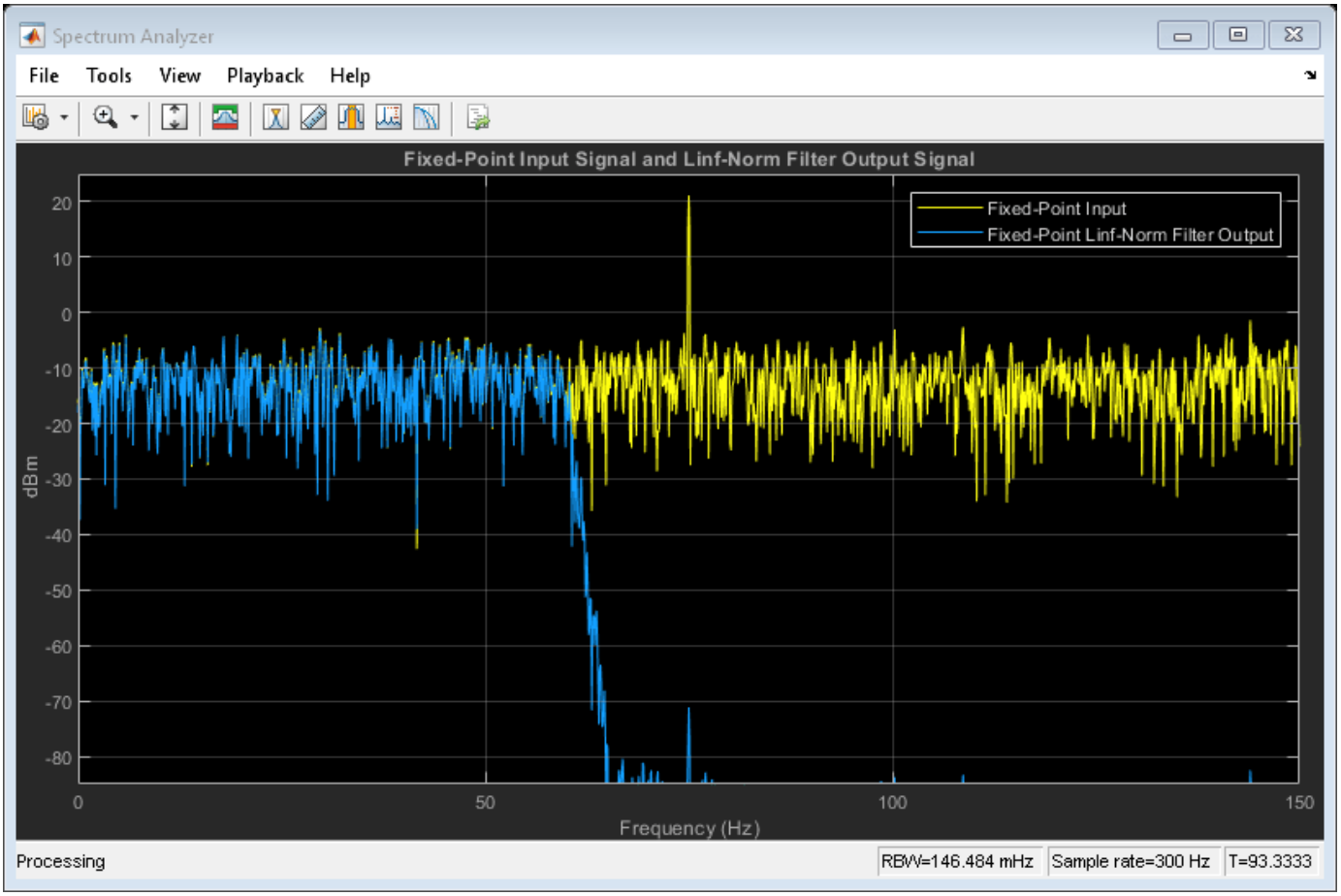



```
close(fvt_Linf);      % clean up
close(fvt_comp_Linf); % clean up
```

Test the Converted MATLAB Fixed-Point Linf-Norm Biquad Filter

After following the Fixed-Point Converter procedure again, as above, but using the Linf-norm scaled filter coefficient values, run the new converted fixed-point function and view input-output results.

```
scope = dsp.SpectrumAnalyzer('SampleRate',300, ...
    'PlotAsTwoSidedSpectrum',false,'ShowLegend',true, ...
    'YLimits',[-85 25],'Title', ...
    'Fixed-Point Input Signal and Linf-Norm Filter Output Signal', ...
    'ChannelNames', ...
    {'Fixed-Point Input','Fixed-Point Linf-Norm Filter Output'});
rng(12345); % seed the rng for repeatable results
for k = 1:10
    inp_sig = rand(inp_len,1) - 0.5;  % random values in range (-0.5, 0.5)
    inputFi = fi(inp_sig + inp_itf, 1, 16, 15); % signal range (-1.0, 1.0)
    out_4 = myIIRLinfBiquad_fixpt( ...
        inputFi,fi(numLinf,1,16),fi(denLinf,1,16));
    scope([inputFi,out_4]) % visualize
end
```

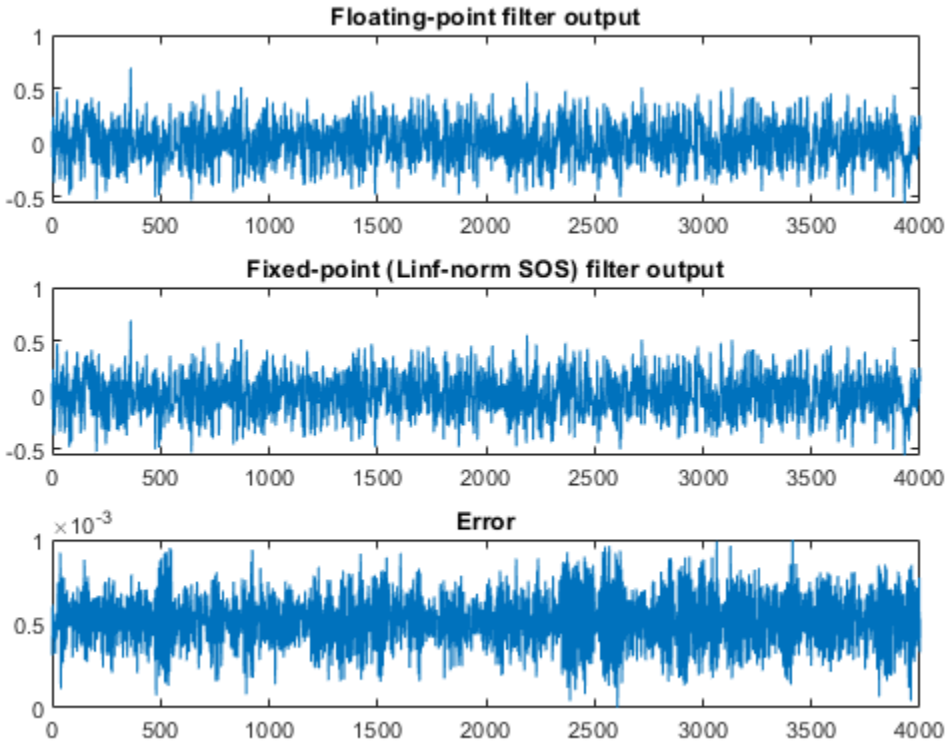



```
clear scope; % clean up
```

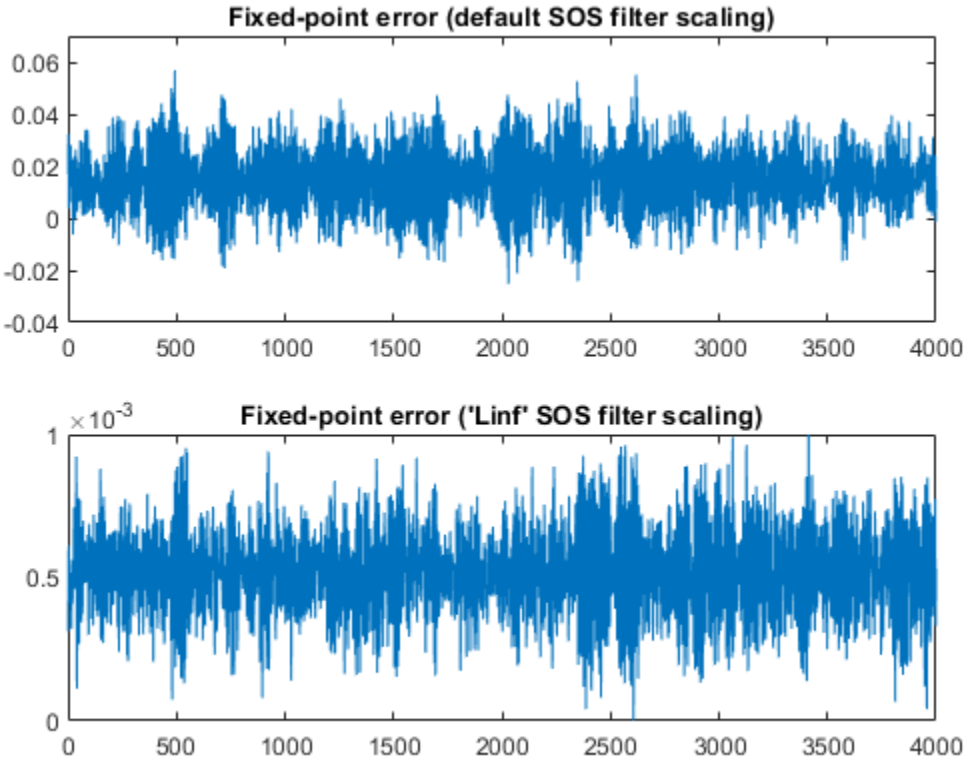
Reduced Fixed-Point Implementation Error Using Linf-norm SOS Scaling

Infinity-norm SOS scaling often produces outputs with lower error.

```
fig1 = figure;
subplot(3,1,1);
plot(out_1);
title('Floating-point filter output');
subplot(3,1,2);
plot(out_4);
title('Fixed-point (Linf-norm SOS) filter output');
subplot(3,1,3);
plot(out_1 - double(out_4));
axis([0 4000 0 1e-3]);
title('Error');
```




```
fig2 = figure;
subplot(2,1,1);
plot(out_1 - double(out_3));
axis([0 4000 -4e-2 7e-2]);
title('Fixed-point error (default SOS filter scaling)');
subplot(2,1,2);
plot(out_1 - double(out_4));
axis([0 4000 0 1e-3]);
title('Fixed-point error (''Linf'' SOS filter scaling)');
```




```
close(fig1); % clean up
close(fig2); % clean up
```

Summary

We have outlined a procedure to convert a floating-point IIR filter to a fixed-point implementation. The dsp.BiquadFilter objects of the DSP System Toolbox™ are equipped with Simulation Minimum and Maximum instrumentation capabilities that help the Fixed-Point Converter App automatically and dynamically scale internal filter signals. In addition, various 'fvtool' and dsp.SpectrumAnalyzer analyses provide tools to the user to perform verifications at each step of the process.

GSM Digital Down Converter in MATLAB

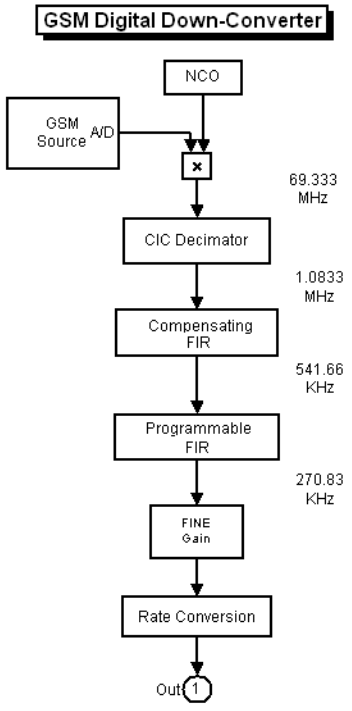
This example shows how to simulate steady-state behavior of a fixed-point digital down converter for GSM (Global System for Mobile) baseband conversions. The example uses signal processing System objects to emulate the operation of the TI Graychip 4016 Quad Digital Down Converter and requires a Fixed-Point Designer™ license.

Introduction

The Digital Down Converter (DDC) is an important component of a digital radio. It performs frequency translation to convert the high input sample rate down to a lower sample rate for efficient processing. In this example the DDC accepts a bandpass signal with a sample rate around 70 megasamples per seconds (MSPS) and performs the following operations:

- Digital mixing or down conversion of the input signal using a Numerically Controlled Oscillator (NCO) and a mixer.
- Narrowband low-pass filtering and decimation using a filter chain of Cascaded Integrator-Comb (CIC) and FIR filters.
- Gain adjustment and final resampling of the data stream.

The DDC produces a baseband signal with a sample rate of 270 kilosamples per seconds (KSPS) that is ready for demodulation. A block diagram of a typical DDC is shown below.




```
if ~isfixptinstalled
    error(message('dsp:dspDigitalDownConverter:noFixptTbx'));
end
```

Initialization

Create and configure a sine wave source System object to model the GSM source. You set the object's frequency to 69.1e6*5/24 MSPS because, after digital mixing, the object will have a baseband frequency of around 48 KSPS. Because the system you are modeling resamples the input by a factor of 4/(3*256), you need to set the object's frame size to be the least common multiplier of these factors.

```
Fs = 69.333e6;
FrameSize = 768;
sine = dsp.SineWave( ...
            'Frequency', 69.1e6*5/24, ...
            'SampleRate', Fs, ...
            'Method', 'Trigonometric function', ...
            'SamplesPerFrame', FrameSize);
```

Create and configure an NCO System object to mix and down convert the GSM signal. The TI Graychip requires the tuning frequency (PhaseIncrement property) to be a 32-bit data type with 32-bit fraction length. The phase offset needs to be a 16-bit data type with 16-bit fraction length. To reduce the amplitude quantization noise and spread the spurious frequencies across the available bandwidth, add a dither signal to the accumulator phase values. Typically, the number of dither bits (14) is the difference between the accumulator word length (32) and the table address word length (18).

```
nco = dsp.NCO( ...
            'PhaseIncrementSource', 'Property', ...
            'PhaseIncrement', int32((5/24) *2^32), ...
            'PhaseOffset', int16(0), ...
            'NumDitherBits', 14, ...
            'NumQuantizerAccumulatorBits', 18, ...
            'Waveform', 'Complex exponential', ...
            'SamplesPerFrame', FrameSize, ...
            'AccumulatorDataType', 'Custom', ...
            'CustomAccumulatorDataType', numerictype([],32), ...
            'OutputDataType', 'Custom', ...
            'CustomOutputDataType', numerictype([],20,18));
```

Create and configure a CIC decimator System object that decimates the mixer output by a factor of 64. CIC filters can achieve high decimation or interpolation rates without using any multipliers. This feature makes them very useful for digital systems operating at high rates.

```
M1 = 64;
cicdec = dsp.CICDecimator( ...
            'DecimationFactor', M1, ...
            'NumSections', 5, ...
            'FixedPointDataType', 'Minimum section word lengths', ...
            'OutputWordLength', 20);
```

Create and configure an FIR decimator System object to compensate for the passband droop caused by the CIC filter. This filter also decimates by a factor of 2.

```
gsmcoeffs; % Read the CFIR and PFIR coeffs
M2 = 2;
cfir = dsp.FIRDecimator(M2, cfir, ...
            'CoefficientsDataType', 'Custom', ...
            'CustomCoefficientsDataType', numerictype([],16), ...
            'FullPrecisionOverride', false,...
```

```
            'OutputDataType', 'Custom', ...
            'CustomOutputDataType', numerictype([],20,-12));
```

Create and configure an FIR decimator System object to reduce the sample rate by another factor of 2.

```
M3 = 2;
pfir = dsp.FIRDecimator(M3, pfir, ...
            'CoefficientsDataType', 'Custom', ...
            'CustomCoefficientsDataType', numerictype([],16), ...
            'FullPrecisionOverride',false, ...
            'OutputDataType', 'Custom', ...
            'CustomOutputDataType', numerictype([],20,-12));
```

Create and configure an FIR rate converter System object to resample the final output by a factor of 4/3.

```
firrc = dsp.FIRRateConverter(4, 3, fir1(31,0.25),...
    'CoefficientsDataType', 'Custom', ...
    'CustomCoefficientsDataType', numerictype([],12), ...
    'FullPrecisionOverride',false, ...
    'OutputDataType', 'Custom', ...
    'CustomOutputDataType', numerictype([],24,-12));
```

Create a fi object of specified numerictype to act as a data type conversion for the sine output.

```
gsmsig = fi(zeros(768,1),true,14,13);
```

Create a fi object of specified numerictype to store the fixed-point mixer output.

```
mixsig = fi(zeros(768,1),true,20,18);
```

Create and configure two Time Scope System objects to plot the real and imaginary parts of the FIR rate converter filter output.

```
timeScope1 = timescope(...
  'Name', 'Rate Converter Output: Real Signal', ...
  'SampleRate', Fs/256*4/3, ...
  'TimeSpanSource','property', ...
  'TimeSpan', 1.2e-3, ...
  'YLimits', [-2e8 2e8]);
pos = timeScope1.Position;
timeScope1.Position(1:2) = [pos(1)-0.8*pos(3) pos(2)+0.7*pos(4)];

timeScope2 = timescope(...
  'Name', 'Rate Converter Output: Imaginary Signal', ...
  'Position', [pos(1)-0.8*pos(3) pos(2)-0.7*pos(4) pos(3:4)], ...
  'SampleRate', Fs/256*4/3, ...
  'TimeSpanSource','property', ...
  'TimeSpan', 1.2e-3, ...
  'YLimits', [-2e8 2e8]);
```

Create and configure two Spectrum Analyzer System objects to plot the power spectrum of the NCO output and of the compensated CIC decimator output.

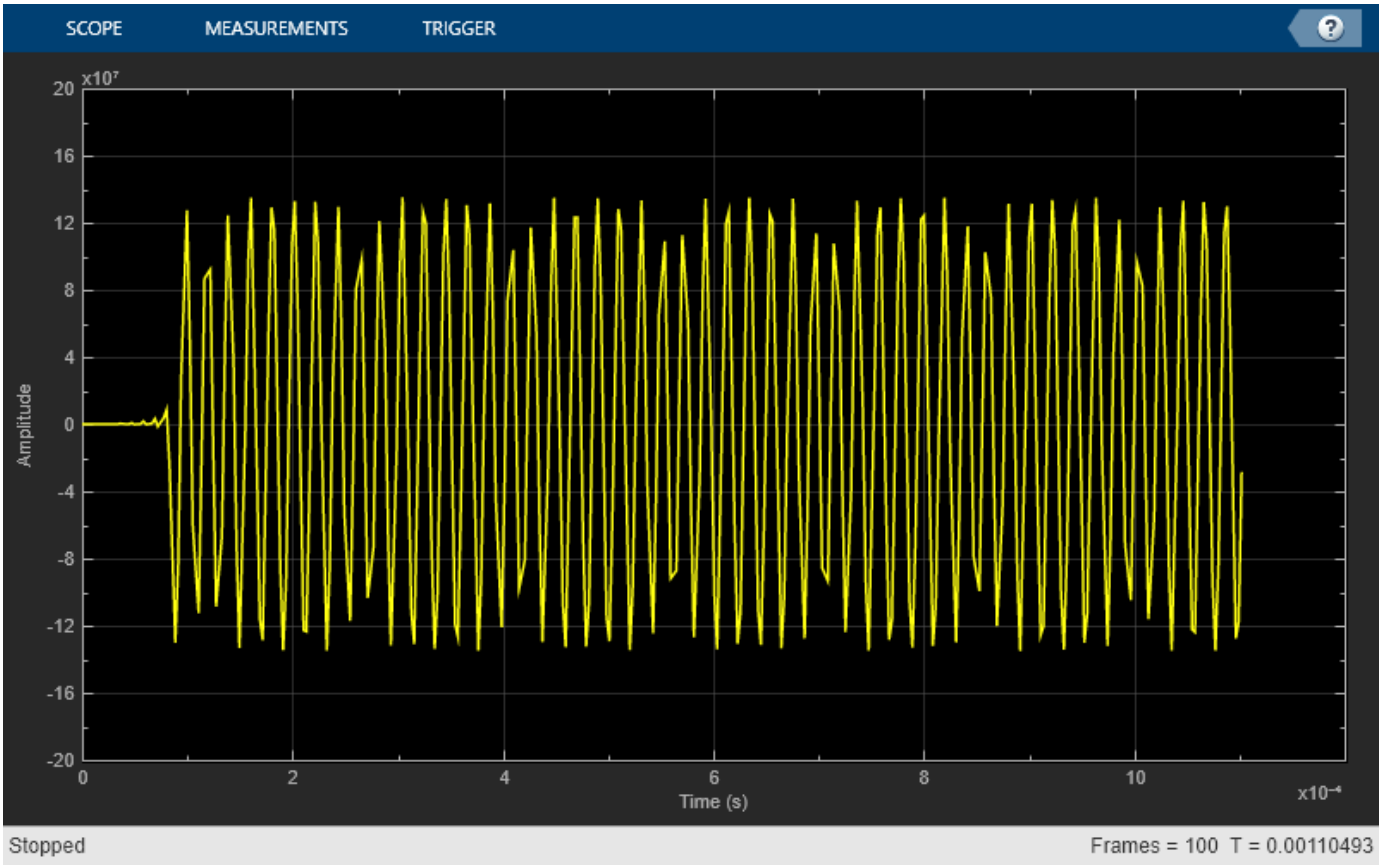
```
specScope1 = dsp.SpectrumAnalyzer(...
  'Name','DSPDDC: NCO Output',...
  'SampleRate',Fs,...
```

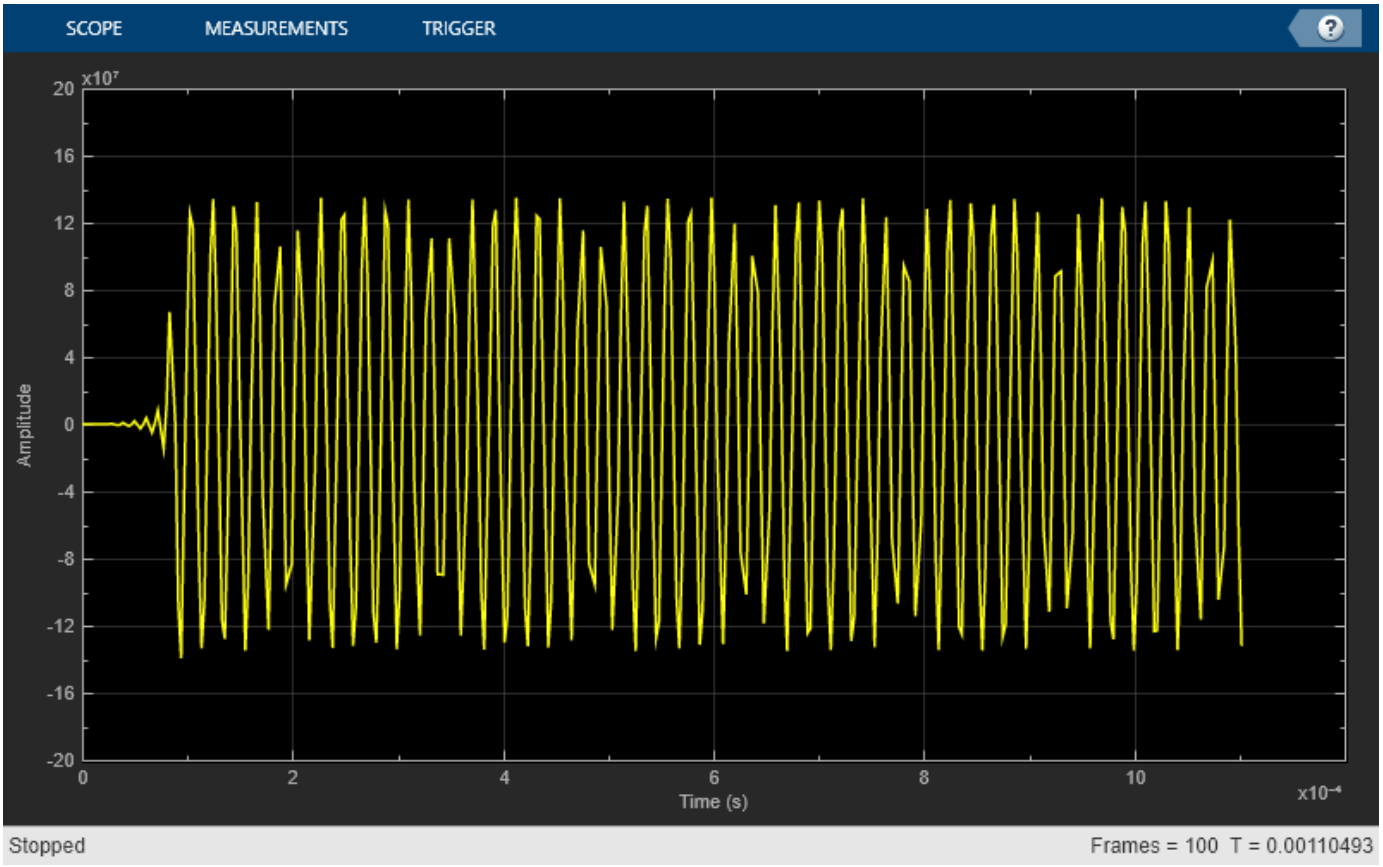
```
  'FrequencySpan','Start and stop frequencies',...
  'StartFrequency',0,'StopFrequency',Fs/2,...
  'RBWSource', 'Property', 'RBW', 4.2e3,...
  'SpectralAverages', 1,...
  'Title', 'Power spectrum of NCO output',...
  'Position',[pos(1)+.8*pos(3) pos(2)+0.7*pos(4) pos(3:4)]);

FsCICcomp = Fs/(M1*M2);
specScope2 = dsp.SpectrumAnalyzer(...
  'Name','DSPDDC: Compensated CIC Decimator Output',...
  'SampleRate',FsCICcomp,...
  'FrequencySpan','Start and stop frequencies',...
  'StartFrequency',0, 'StopFrequency',FsCICcomp/2,...
  'RBWSource', 'Property', 'RBW', 4.2e3,...
  'SpectralAverages', 1,...
  'Title', 'Power spectrum of compensated CIC decimator output',...
  'Position',[pos(1)+.8*pos(3) pos(2)-0.7*pos(4) pos(3:4)]);
```

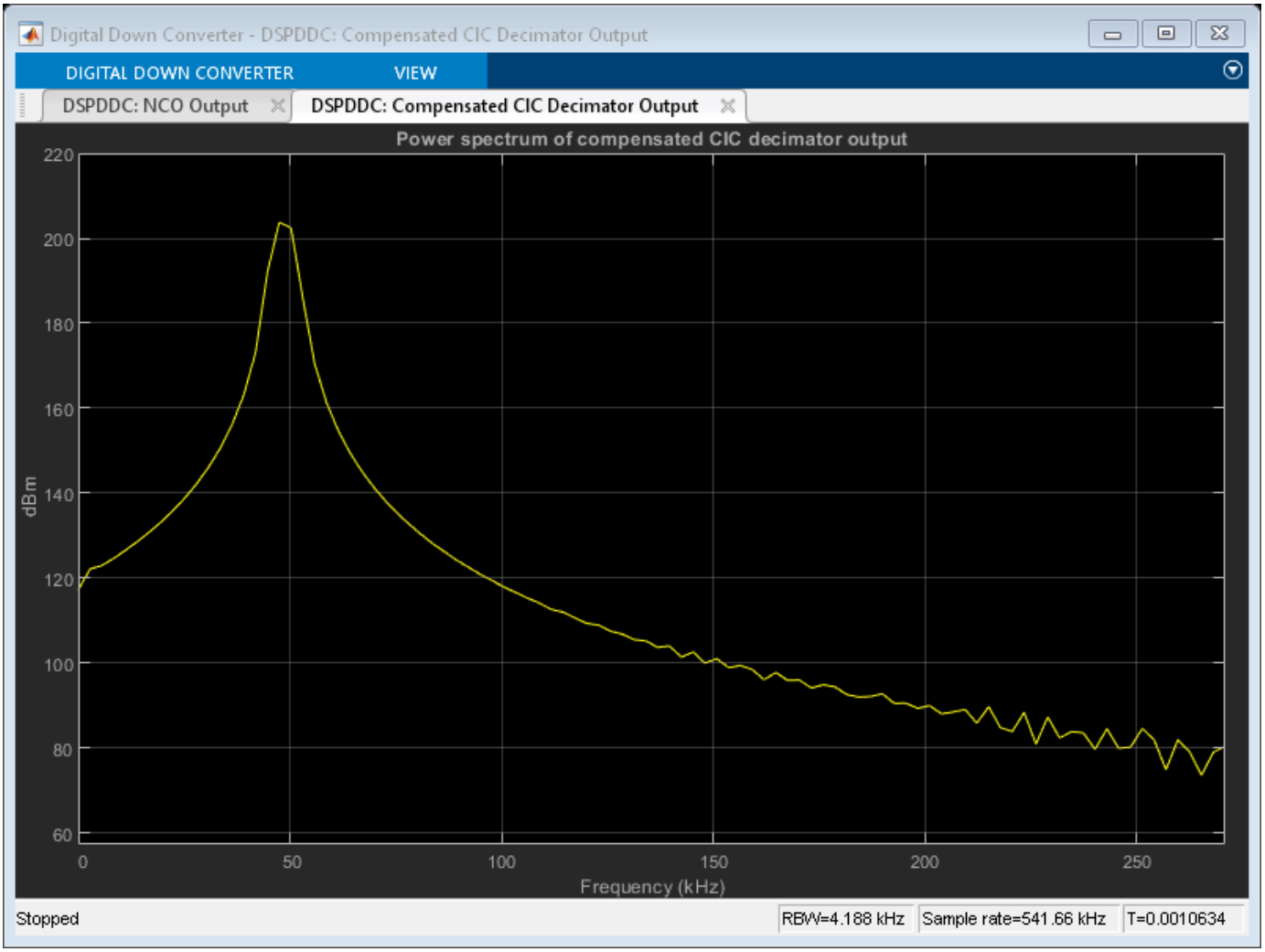
Processing Loop

In the processing loop, the mixer front-end digitally down converts the GSM signal to baseband. The CIC decimation and compensation filters downsample the signal by a factor of 128 and the programmable FIR filter decimates by another factor of 2 to achieve an overall decimation of 256. The resampling back-end performs additional application-specific filtering. Running the processing loop for 100 iterations is equivalent to processing around 1.1 ms of the resampled output.

```
% Create a container for the four scopes
scopesContainer = HelperCreateScopesContainer(...
                  {timeScope1,specScope1,timeScope2,specScope2},...
                   'Name','Digital Down Converter',...
                   'Layout',[2 2],...
                   'ExpandToolstrip',false);

for ii = 1:100
    gsmsig(:) = sine();         % GSM signal
    ncosig    = nco();          % NCO signal
    mixsig(:) = gsmsig.*ncosig; % Digital mixer

    % CIC filtering and compensation
    ycic = cfir(cicdec(mixsig));

    % Programmable FIR and sample-rate conversion
    yrcout = firrc(pfir(ycic));

    % Frequency and time-domain plots
    timeScope1(real(yrcout));
    timeScope2(imag(yrcout));
    specScope1(ncosig);
    specScope2(ycic);
end
release(timeScope1);
release(timeScope2);
release(specScope1);
release(specScope2);
```


SCOPE
MEASUREMENTS
TRIGGER
x10⁷
20
16
12
8
4
0
-4
-8
-12
-16
-20
Amplitude
0
2
4
6
8
10
Time (s)
x10⁻⁴
Stopped
Frames = 100 T = 0.00110493


SCOPE
MEASUREMENTS
TRIGGER
x10⁷
20
16
12
8
4
0
-4
-8
-12
-16
-20
Amplitude
0
2
4
6
8
10
Time (s)
x10⁻⁴
Stopped
Frames = 100 T = 0.00110493



Conclusion

In this example, you used DSP System Toolbox™ System objects to simulate the steady-state behavior of a fixed-point GSM digital down converter. The Time Scope and Spectrum Analyzer System objects enable you to analyze the various stages of a DDC.

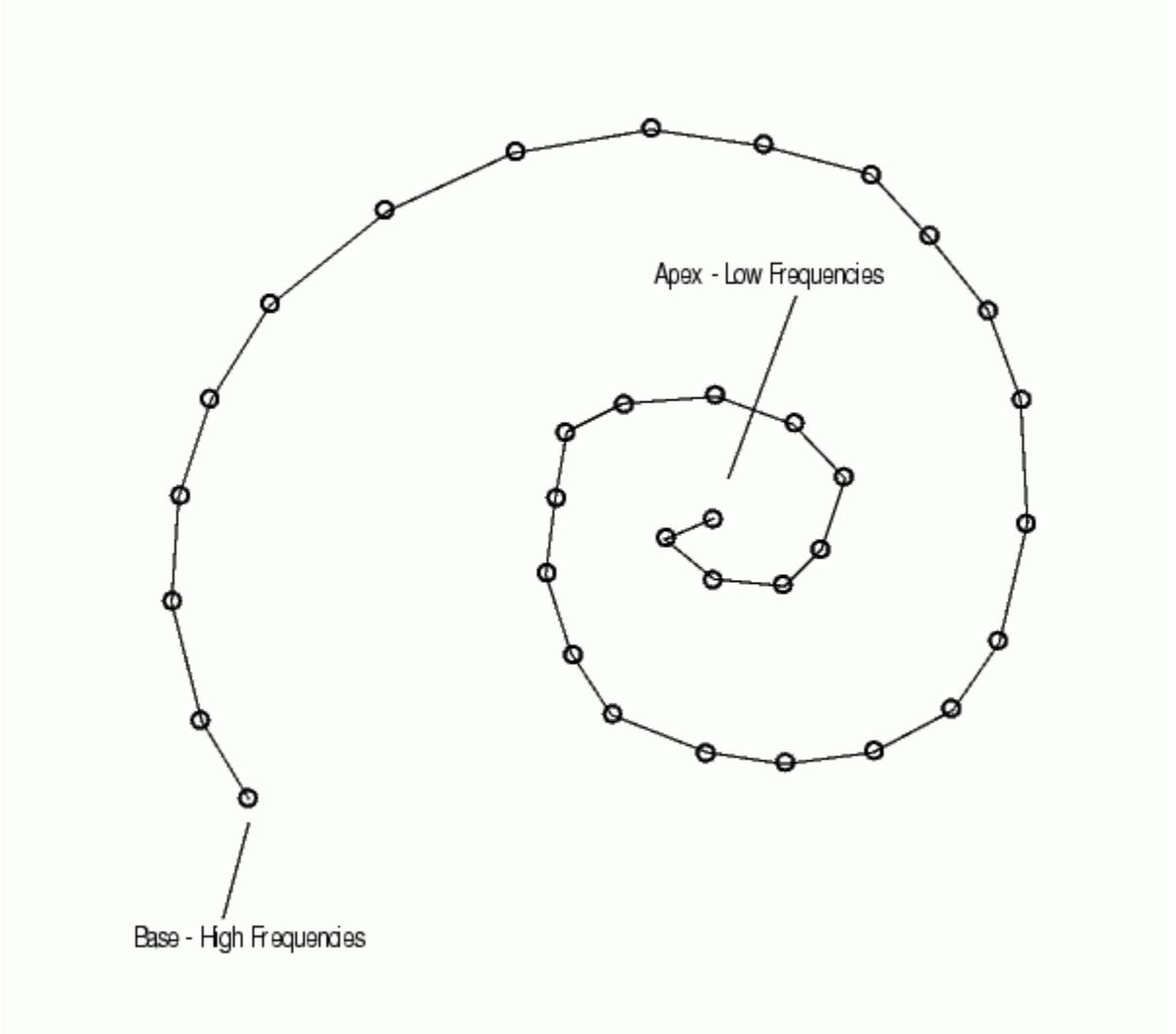
Cochlear Implant Speech Processor

This example shows how to simulate the design of a cochlear implant that can be placed in the inner ear of a profoundly deaf person to restore partial hearing. Signal processing is used in cochlear implant development to convert sound to electrical pulses. The pulses can bypass the damaged parts of a deaf person's ear and be transmitted to the brain to provide partial hearing.

This example highlights some of the choices made when designing cochlear implant speech processors that can be modeled using the DSP System Toolbox™. In particular, the benefits of using a cascaded multirate, multistage FIR filter bank instead of a parallel, single-rate, second-order-section IIR filter bank are shown.

Human Hearing

Converting sound into something the human brain can understand involves the inner, middle, and outer ear, hair cells, neurons, and the central nervous system. When a sound is made, the outer ear picks up acoustic waves, which are converted into mechanical vibrations by tiny bones in the middle ear. The vibrations move to the inner ear, where they travel through fluid in a snail-shaped structure called the cochlea. The fluid displaces different points along the basilar membrane of the cochlea. Displacements along the basilar membrane contain the frequency information of the acoustic signal. A schematic of the membrane is shown here (not drawn to scale).



Frequency Sensitivity in the Cochlea

Different frequencies cause the membrane to displace maximally at different positions. Low frequencies cause the membrane to be displaced near its apex, while high frequencies stimulate the membrane at its base. The amplitude of the displacement of the membrane at a particular point is proportional to the amplitude of the frequency that has excited it. When a sound is composed of many frequencies, the basilar membrane is displaced at multiple points. In this way the cochlea separates complex sounds into frequency components.

Each region of the basilar membrane is attached to hair cells that bend proportionally to the displacement of the membrane. The bending causes an electrochemical reaction that stimulates neurons to communicate the sound information to the brain through the central nervous system.

Alleviating Deafness with Cochlear Implants

Deafness is most often caused by degeneration or loss of hair cells in the inner ear, rather than a problem with the associated neurons. This means that if the neurons can be stimulated by a means other than hair cells, some hearing can be restored. A cochlear implant does just that. The implant electrically stimulates neurons directly to provide information about sound to the brain.

The problem of how to convert acoustic waves to electrical impulses is one that Signal Processing helps to solve. Multichannel cochlear implants have the following components in common:

- A microphone to pick up sound
- A signal processor to convert acoustic waves to electrical signals
- A transmitter
- A bank of electrodes that receive the electrical signals from the transmitter, and then stimulate auditory nerves.

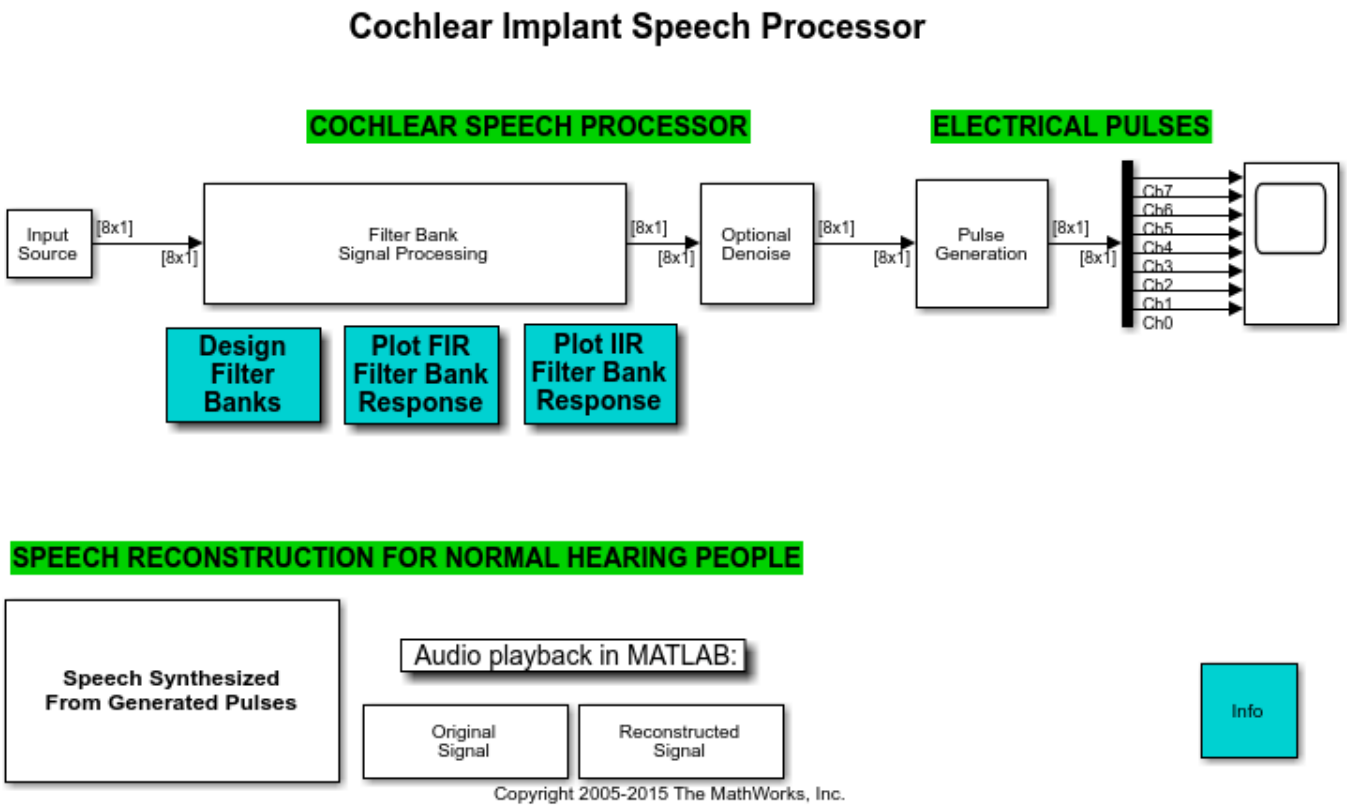
Just as the basilar membrane of the cochlea resolves a wave into its component frequencies, so does the signal processor in a cochlear implant divide an acoustic signal into component frequencies, that are each then transmitted to an electrode. The electrodes are surgically implanted into the cochlea of the deaf person in such a way that they each stimulate the appropriate regions in the cochlea for the frequency they are transmitting. Electrodes transmitting high-frequency (high-pitched) signals are placed near the base, while those transmitting low-frequency (low-pitched) signals are placed near the apex. Nerve fibers in the vicinity of the electrodes are stimulated and relay the information to the brain. Loud sounds produce high-amplitude electrical pulses that excite a greater number of nerve fibers, while quiet ones excite less. In this way, information both about the frequencies and amplitudes of the components making up a sound can be transmitted to the brain of a deaf person by a cochlear implant.

Exploring the Example

The block diagram at the top of the model represents a cochlear implant speech processor, from the microphone which picks up the sound (Input Source block) to the electrical pulses that are generated. The frequencies increase in pitch from Channel 0, which transmits the lowest frequency, to Channel 7, which transmits the highest.

To hear the original input signal, double-click the Original Signal block at the bottom of the model. To hear the output signal of the simulated cochlear implant, double-click the Reconstructed Signal block.

There are a number of changes you can make to the model to see how different variables affect the output of the cochlear implant speech processor. Remember that after you make a change, you must rerun the model to implement the changes before you listen to the reconstructed signal again.



Simultaneous Versus Interleaved Playback

Research has shown that about eight frequency channels are necessary for an implant to provide good auditory understanding for a cochlear implant user. Above eight channels, the reconstructed signal usually does not improve sufficiently to justify the rising complexity. Therefore, this example resolves the input signal into eight component frequencies, or electrical pulses.

The Speech Synthesized from Generated Pulses block at the bottom left of the model allows you to either play each electrical channel simultaneously or sequentially. Oftentimes cochlear implant users experience inferior results with simultaneous frequencies, because the electrical pulses interact with each other and cause interference. Emitting the pulses in an interleaved manner mitigates this problem for many people. You can toggle the **Synthesis mode** of the Speech Synthesized From Generated Pulses block to hear the difference between these two modes. Zoom in on the Time Scope block to observe that the pulses are interleaved.

Adjusting for Noisy Environments

Noise presents a significant challenge to cochlear implant users. Select the **Add noise** parameter in the Input Source block to simulate the effects of a noisy environment on the reconstructed signal. Observe that the signal becomes difficult to hear. The Denoise block in the model uses a Soft Threshold block to attempt to remove noise from the signal. When the **Denoise** parameter in the Denoise block is selected, you can listen to the reconstructed signal and observe that not all the noise is removed. There is no perfect solution to the noise problem, and the results afforded by any denoising technology must be weighed against its cost.

Signal Processing Strategy

The purpose of the Filter Bank Signal Processing block is to decompose the input speech signal into eight overlapping subbands. More information is contained in the lower frequencies of speech signals than in the higher frequencies. To get as much resolution as possible where the most information is contained, the subbands are spaced such that the lower-frequency bands are more narrow than the higher-frequency bands. In this example, the four low-frequency bands are equally spaced, while each of the four remaining high-frequency bands is twice the bandwidth of its lower-frequency neighbor. To examine the frequency contents of the eight filter banks, run the model using the `Chirp` **Source type** in the Input Source block.

Two filter bank implementations are illustrated in this example: a parallel, single-rate, second-order-section IIR filter bank and a cascaded, multirate, multistage FIR filter bank. Double click on the **Design Filter Banks** button to examine their design and frequency specifications.

Parallel Single-Rate SOS IIR Filter Bank: In this bank, the sixth-order IIR filters are implemented as second-order-sections (SOS). Notice that the DSP System Toolbox™ scale function is used to obtain optimal scaling gains, which is particularly essential for the fixed-point version of this example. The eight filters are running in parallel at the input signal rate. You can look at their frequency responses by double clicking the **Plot IIR Filter Bank Response** button.

Cascaded Multirate Multistage FIR Filter Bank: The design of this filter bank is based on the principles of an approach that combines downsampling and filtering at each filter stage. The overall filter response for each subband is obtained by cascading its components. Double click on the **Design Filter Banks** button to examine how design functions from the DSP System Toolbox are used in constructing these filter banks.

Since downsampling is applied at each filter stage, the later stages are running at a fraction of the input signal rate. For example, the last filter stages are running at one-eighth of the input signal rate. Consequently, this design is very suitable for implementations on the low-power DSPs with limited processing cycles that are used in cochlear implant speech processors. You can look at the frequency responses for this filter bank by double clicking on the **Plot FIR Filter Bank Response** button. Notice that this design produces sharper and flatter subband definition compared to the parallel single-rate SOS IIR filter bank. This is another benefit of a multirate, multistage filter design approach. For a related example see “Multistage Rate Conversion” on page 4-177.

Acknowledgments and References

Thanks to Professor Philip Loizou for his help in creating this example.

More information on Professor Loizou's cochlear implant research is available at:

- Loizou, Philip C., "Mimicking the Human Ear," **IEEE® Signal Processing Magazine**, Vol. 15, No. 5, pp. 101-130, 1998.

Available Example Versions

Floating-point version: dspcochlear

Fixed-point version: dspcochlear_fixpt

Three-Channel Wavelet Transmultiplexer

This example shows how to reconstruct three independent combined signals transmitted over a single communications link using a Wavelet Transmultiplexer (WTM). The example illustrates the perfect reconstruction property of the discrete wavelet transform (DWT).

Introduction

This WTM combines three source signals for transmission over a single link, then separates the three signals at the receiving end of the channel. The example demonstrates a three-channel transmultiplexer, but the method can be extended to an arbitrary number of channels.

The operation of a WTM is analogous to a frequency-domain multiplexer (FDM) in several respects. In an FDM, baseband input signals are filtered and modulated into adjacent frequency bands, summed together, then transmitted over a single link. On the receiving end, the transmitted signal is filtered to separate adjacent frequency channels, and the signals are demodulated back to baseband. The filters also must strongly attenuate the adjacent signal to provide a sharp transition from the filter passband to its stopband. This step limits the amount of crosstalk, or signal leakage, from one frequency band to the next. In addition, FDM often employs an unused frequency band between the three modulated frequency bands, known as a guard band, to relax the requirements on the FDM filters.

In a WTM, the filtering performed by the synthesis and analysis wavelet filters is analogous to the filtering steps in the FDM, and the interpolation in the synthesis stage is equivalent to frequency modulation. From a frequency domain perspective, the wavelet filters are fairly poor spectral filters as compared to the filters required by a FDM implementation, exhibiting slow transitions from passband to stopband, and providing significant distortion in their response. What makes the WTM special is that the analysis and synthesis filters together completely cancel the filter distortions and signal aliasing, producing perfect reconstruction of the input signals and thus perfect extraction of the multiplexed inputs. Ideal spectral efficiency can be achieved, since no guard band is required. Practical limitations in the implementation of channel filters create out-of-band leakage and distortion. In the conventional FDM approach, each channel within the same communications system requires its own filter and is susceptible to crosstalk from neighboring channels. With the WTM method, only a single bandpass filter is required for the entire communications channel, while channel-to-channel interference is eliminated.

Note that a noisy link can cause imperfect reconstruction of the input signals, and the effects of channel noise and other impairments in the recovered signals can differ in FDM and WTM. This can be modeled, for example, by adding a noise source to the data link.

Initialization

Creating and initializing your System objects before they are used in a processing loop is critical to get optimal performance.

```
% Initialize variables used in the example such as the standard deviation of
% the channel noise.
load dspwlets;    % load filter coefficients and input signal
NumTimes = 14;    % for-loop iterations
stdnoise = .2^.5; % standard deviation of channel noise
```

Create a sine wave System object to generate the Channel 1 signal.

```
sine = dsp.SineWave('Frequency',       fs/68, ...
                    'SampleRate',      fs, ...
                    'SamplesPerFrame', fs*2);
```

Create a random number generator stream for the channel noise.

```
strN = RandStream.create('mt19937ar','seed',1);
```

Create a chirp System object to generate the Channel 2 signal.

```
chirpSignal = dsp.Chirp( ...
    'Type', 'Swept cosine', ...
    'SweepDirection', 'Bidirectional', ...
    'InitialFrequency', fs/5000, ...
    'TargetFrequency', fs/50, ...
    'TargetTime', 1000, ...
    'SweepTime', 1000, ...
    'SampleRate', 1/ts, ...
    'SamplesPerFrame', fs);
```

Create and configure a dyadic analysis filter bank System object for subband decomposition of the signal.

```
dyadicAnalysis = dsp.DyadicAnalysisFilterBank( ...
    'CustomLowpassFilter', lod, ...
    'CustomHighpassFilter', hid, ...
    'NumLevels', 2 );
```

Create three System objects for inserting delays in each channel to compensate for the system delay introduced by the wavelet components.

```
delay1 = dsp.Delay(4);
delay2 = dsp.Delay(6);
delay3 = dsp.Delay(6);
```

Create and configure a dyadic synthesis filter bank System object for reconstructing the signal from different subbands of the signal.

```
dyadicSynthesis = dsp.DyadicSynthesisFilterBank( ...
    'CustomLowpassFilter',[0 lor], ...
    'CustomHighpassFilter',[0 hir], ...
    'NumLevels', 2 );
```

Create time scope System objects to plot the original, reconstructed and error signals.

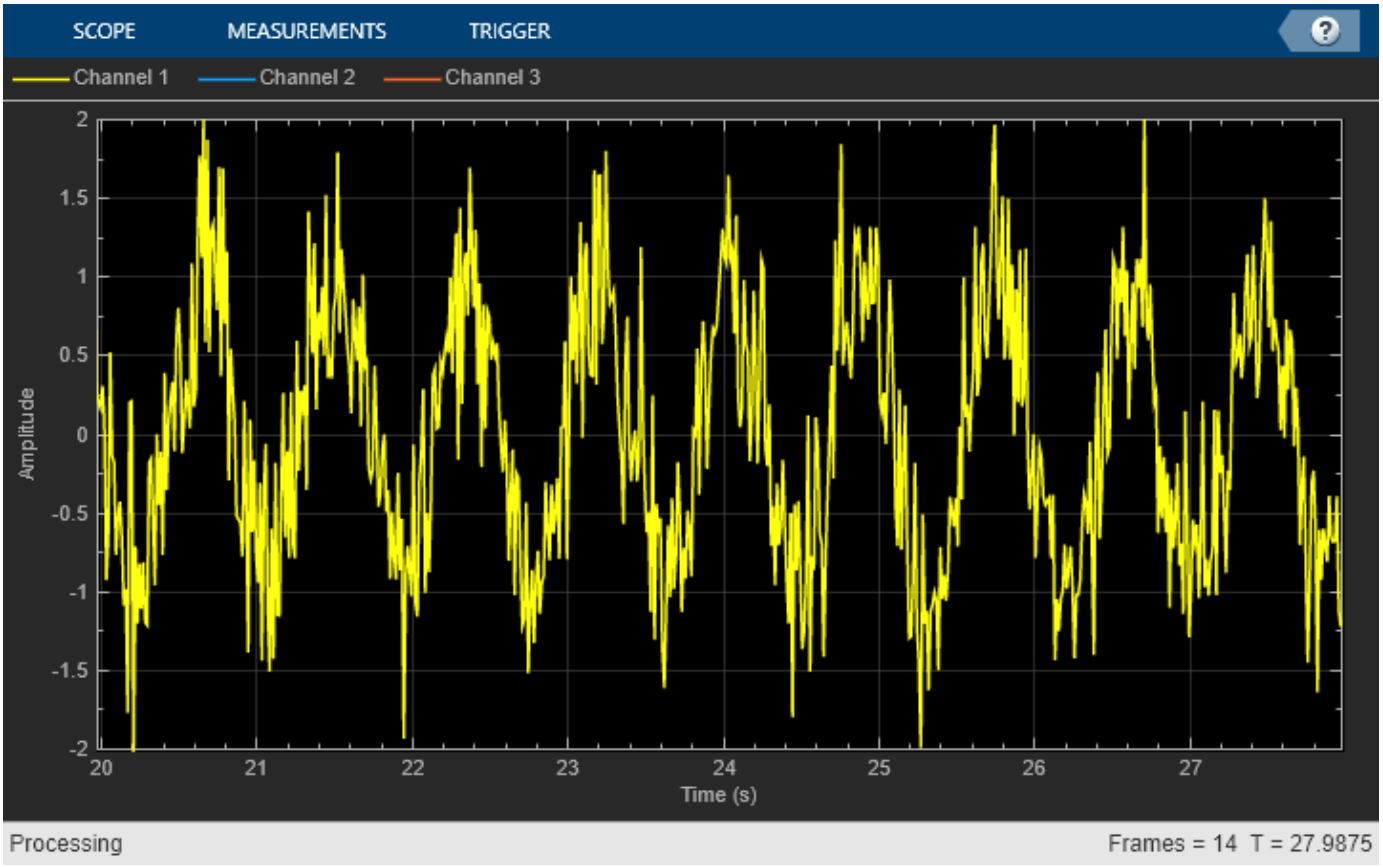
```
scope1 = timescope( ...
  'Name', 'Three Channel WTM: Original (delayed)', ...
  'SampleRate', fs, ...
  'TimeSpanSource','property', ...
  'TimeSpan', 8, ...
  'YLimits', [-2 2], ...
  'ShowLegend', true);
pos = scope1.Position;
pos(3:4) = 0.9*pos(3:4);
scope1.Position = [pos(1)-1.1*pos(3) pos(2:4)];

scope2 = timescope(...
  'Name', 'Three Channel WTM: Reconstructed', ...
```

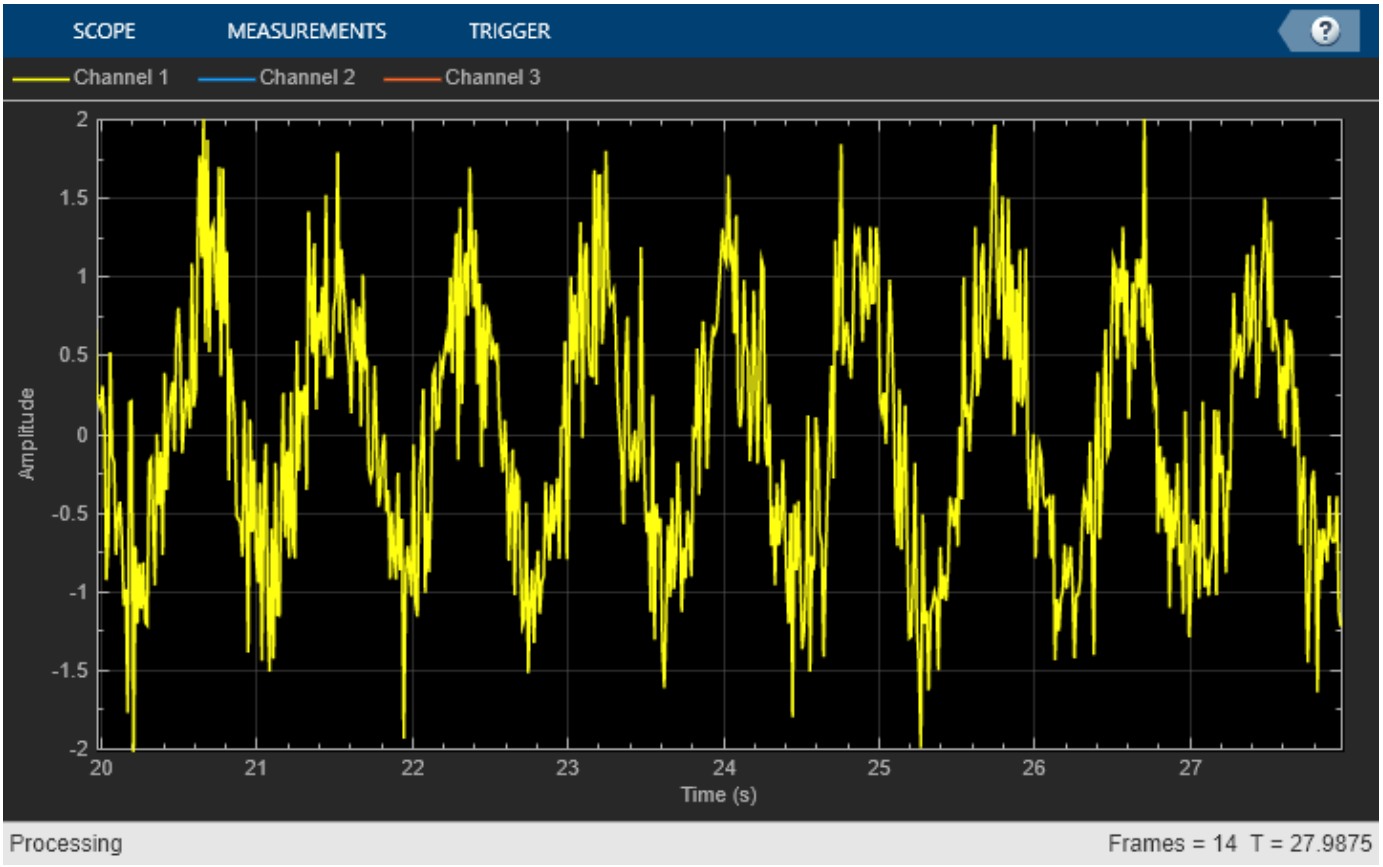
```
  'Position', pos, ...
  'SampleRate', fs, ...
  'TimeSpanSource', 'property', ...
  'TimeSpan', 8, ...
  'YLimits', [-2 2], ...
  'ShowLegend', true);

scope3 = timescope(...
  'Name', 'Three Channel WTM: Error', ...
  'Position', [pos(1)+1.1*pos(3) pos(2:4)], ...
  'SampleRate', fs, ...
  'TimeSpanSource','property', ...
  'TimeSpan', 8, ...
  'YLimits', [-5e-11 5e-11], ...
  'ShowLegend', true);

% Create variable for Channel 3 signal.
Tx_Ch3 = [ones(35,1);zeros(45,1)];   % Generate the Channel 3 signal
```

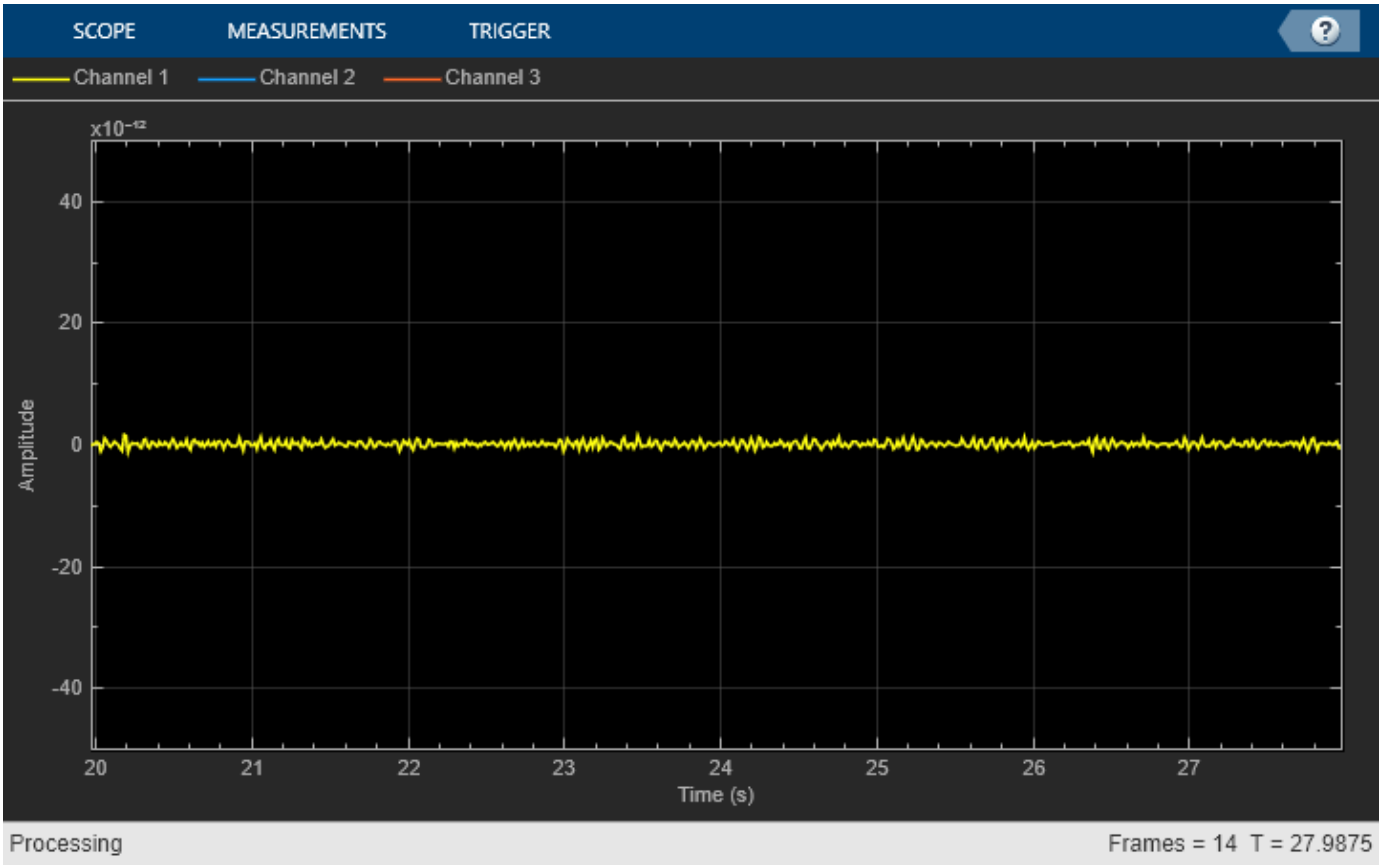
Stream Processing Loop

Create a processing loop to simulate the three channel transmultiplexer. This loop uses the System objects you instantiated above.

```
for ii = 1:NumTimes
    Tx_Ch1 = sine() + ...
        stdnoise*randn(strN,fs*2,1);      % Generate Channel 1 signal
    Tx_Ch1_delay = delay1(Tx_Ch1);

    Tx_Ch2 = chirpSignal();                % Generate Channel 2 signal
    Tx_Ch2_delay = delay2(Tx_Ch2);

    Tx_Ch3_delay = delay3(Tx_Ch3); % Delayed Channel 3 signal

    % Concatenate the three channel signals
    Tx = [Tx_Ch1; Tx_Ch2; Tx_Ch3];

    % Synthesis stage equivalent to frequency modulation.
    y = dyadicSynthesis(Tx);

    % Analysis stage
    Rx = dyadicAnalysis(y);

    % Separate out the three channels
    Rx_Ch1 = Rx(1:160);
    Rx_Ch2 = Rx(161:240);
    Rx_Ch3 = Rx(241:320);

    % Calculate the error between TX and RX signals
    err_Ch1 = Tx_Ch1_delay - Rx_Ch1;
    err_Ch2 = Tx_Ch2_delay - Rx_Ch2;
    err_Ch3 = Tx_Ch3_delay - Rx_Ch3;

    % Plot the results.
    scope1(Tx_Ch1_delay, Tx_Ch2_delay, Tx_Ch3_delay);
    scope2(Rx_Ch1, Rx_Ch2, Rx_Ch3);
    scope3(err_Ch1, err_Ch2, err_Ch3);
end
```


SCOPE
MEASUREMENTS
TRIGGER
Channel 1
Channel 2
Channel 3
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
Amplitude
20
21
22
23
24
25
26
27
Time (s)
Processing
Frames = 14 T = 27.9875


SCOPE
MEASUREMENTS
TRIGGER
Channel 1
Channel 2
Channel 3
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
Amplitude
20
21
22
23
24
25
26
27
Time (s)
Processing
Frames = 14 T = 27.9875



Summary

In this example you used the `DyadicAnalysisFilterBank` and `DyadicSynthesisFilterBank` System objects to implement a Wavelet Transmultiplexer. The perfect reconstruction property of the analysis and synthesis wavelet filters enables perfect extraction of multiplexed inputs.

Arbitrary Magnitude and Phase Filter Design

This example shows how to design filters with customized magnitude and phase specifications. Many filter design problems focus on the magnitude response only, while assuming a linear phase response through symmetry. In some cases, however, the desired filter needs to satisfy constraints on both magnitude and phase.

For example, custom magnitude and phase design specifications can be used for the equalization of magnitude and phase distortions found in data transmission systems (channel equalization) or in oversampled ADC (compensation for non-ideal hardware characteristics). Another application is the design of filters that have smaller group delays than linear phase filters and less distortion than minimum-phase filters for a given order.

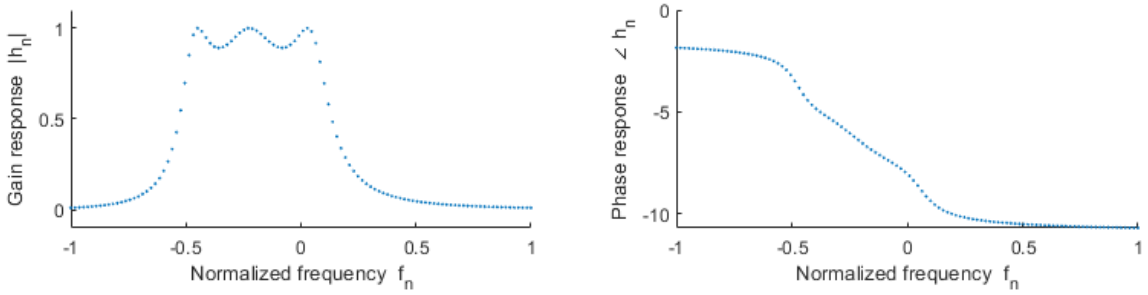
Frequency Response Specification and Filter Design

Filter responses are usually specified by frequency intervals (bands) along with the desired gain on each band. Custom magnitude and phase filter specifications are similar, but also include phase response, usually as a complex value encoding both gain and phase response. In most cases, the response specification is comprised of a frequencies vector $F = [f1, ..., fN]$ of N increasing frequencies, and a frequency responses vector $H = [h1, ..., hN]$ corresponding to the filter's complex response values. In DSP System Toolbox™, you can create a filter specification object with a desired frequency response using `fdesign.arbmagnphase`. Once a specification object has been created, you can design an FIR or an IIR filter using the `design` function. For more information about FIR and IIR design algorithms, see [1].

FIR Designs

In this first example, we compare several FIR design methods to model the magnitude and phase of a complex RF bandpass filter. First, load the desired filter specification: frequencies to the vector F, and the complex response values to the vector H. Plot the gain and phase frequency responses on the the left and the right graphs respectively.

```
load('gainAndPhase','F','H') % Load frequency response data
plotResponse(F, H) % A helper plotting function used in this demo
```



Create a specification object using `fdesign.arbmagnphase` with the `'N,F,H'` specification pattern. This specification accepts the desired filter order N, along with the frequency response vectors F and H. The `'N,F,H'` pattern defines the desired response on the entire Nyquist range (that is, a single-band specification with no relaxed "don't care" regions). In this example, the desired response data vectors F and H have 655 points, which is relatively dense across the frequency domain.

```
N = 100;
f = fdesign.arbmagnphase('N,F,H',N,F,H);
```

Determine which design methods can be used for this specification object using the `designmethods` function. In this case, the methods are: `equiripple`, `firls` (least squares), and `freqsamp` (frequency sampling).

```
designmethods(f,'fir')
```

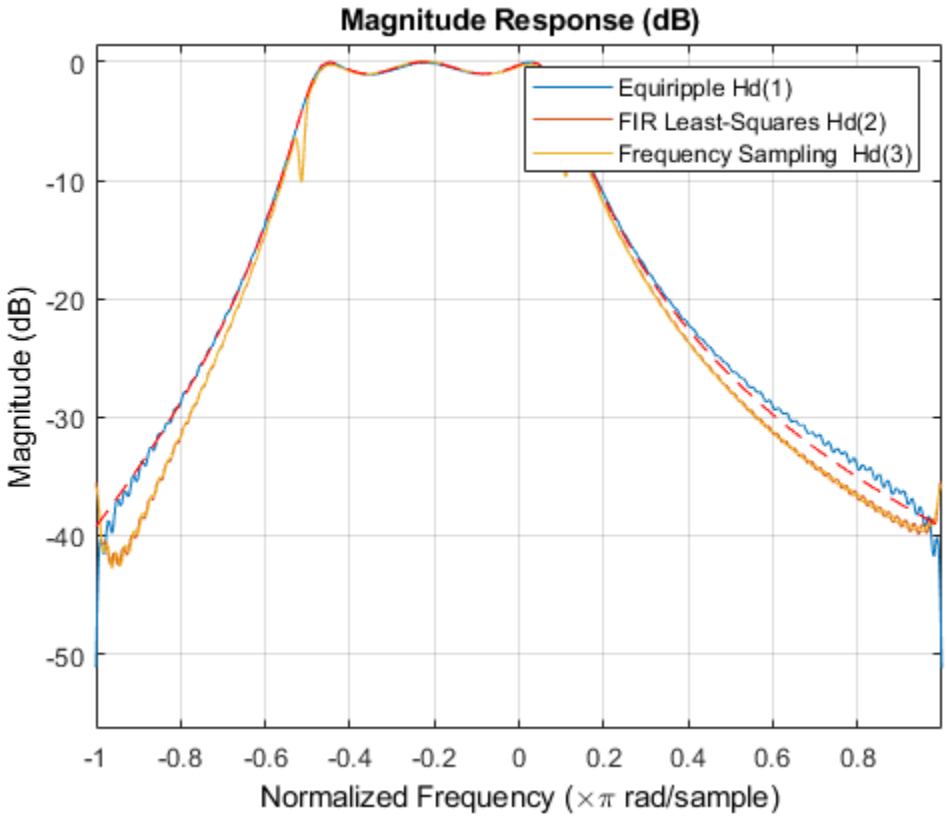
```
FIR Design Methods for class fdesign.arbmagnphase (N,F,H):


equiripple
firls
freqsamp
```

Design the filters with the `design` function using the desired method from the list above. You can also specify `'allfir'` to design using all available methods, in which case the function returns a cell array of System objects.

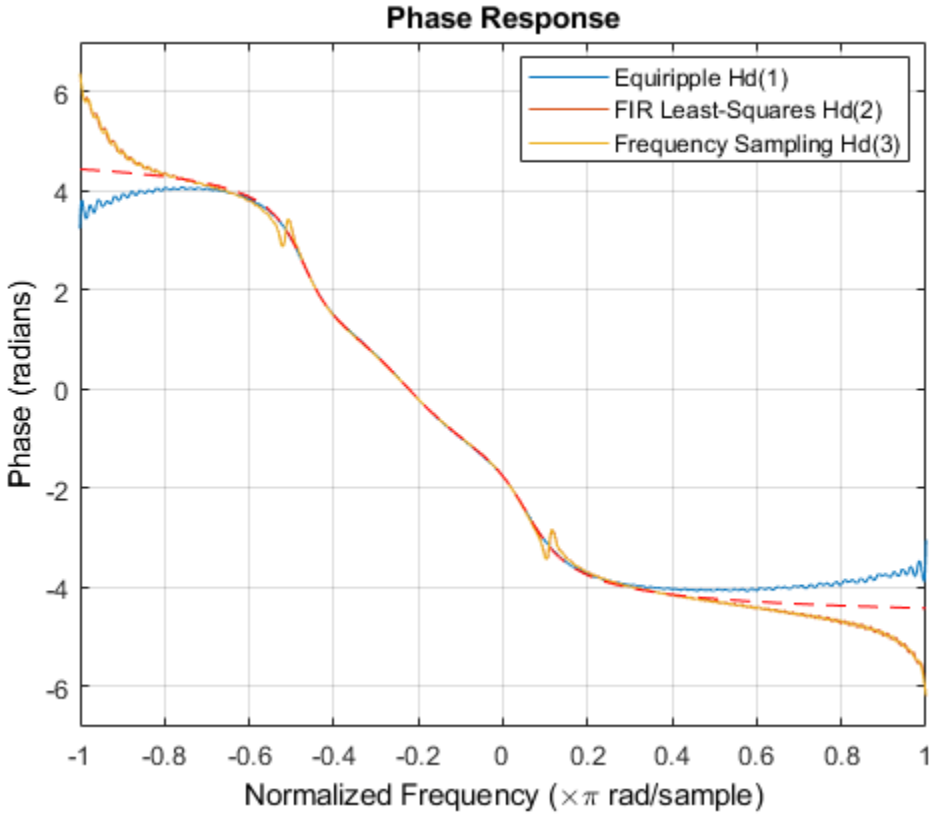
```
Hd = design(f,'allfir',SystemObject = true);
```

Plot the filters' frequency responses and the nominal response in a dashed line. The equiripple design *Hd*`(1)` appears to approximate very well on the passband, but slightly deviates in other regions. The least squares design *Hd*`(2)` is optimized for uniformly weighted quadratic norm (not favoring one region or another), and the frequency sampled FIR design *Hd*`(3)` appears to exhibit the worst approximation of all three.

```
hfvt = fvtool(Hd{:}, 'Color','w');
legend(hfvt,'Equiripple Hd(1)', 'FIR Least-Squares Hd(2)','Frequency Sampling  Hd(3)', ...
    'Location', 'NorthEast')
ax = hfvt.CurrentAxes;
ax.NextPlot = 'add';
plot(ax,F,20*log10(abs(H)),'r--')
```



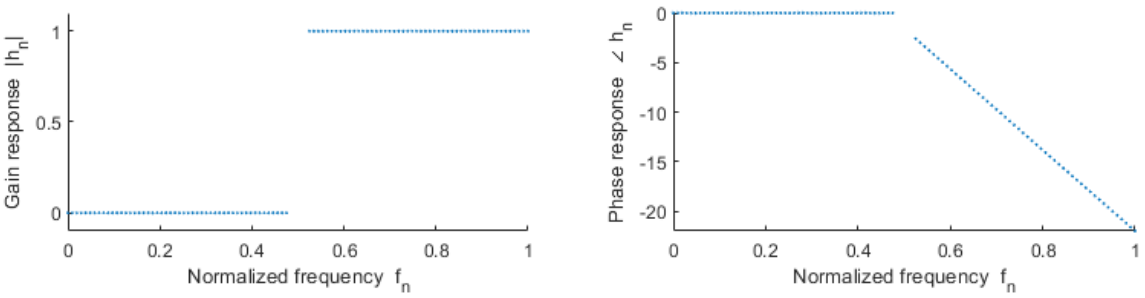

```
hfvt(2) = fvtool(Hd{:},'Analysis','phase','Color','white');
legend(hfvt(2),'Equiripple Hd(1)', 'FIR Least-Squares Hd(2)','Frequency Sampling Hd(3)')

ax = hfvt(2).CurrentAxes;
ax.NextPlot = 'add';
plot(ax,F,unwrap(angle(H))+2*pi,'r--')
```



IIR Designs

In the next part, we design an IIR filter. The desired filter is a halfband highpass filter with a linear phase on the passband. The specification is given by 100 points on the frequency domain as shown in the following figure.

```
F = [linspace(0,.475,50) linspace(.525,1,50)];
H = [zeros(1,50) exp(-1j*pi*13*F(51:100))];
plotResponse(F, H)
```



Create a specification object using a single-band design spec `'Nb, Na, F, H'`, which takes the desired IIR orders *`Na`* `= 10` (denominator order) and *`Nb`* `= 12` (numerator order) as inputs. There is only one design method available for this specification - the least squares IIR design (`iirls`).

```
Nb = 12;
Na = 10;
```

```
f = fdesign.arbmagnphase('Nb,Na,F,H',Nb,Na,F,H);
designmethods(f)
```

```
Design Methods for class fdesign.arbmagnphase (Nb,Na,F,H):


iirls
```

The `iirls` design method allows to specify different weights for different frequencies, giving more control over the approximation quality of each band. Design the filter with a weight of 1 on the stopband and a weight of 100 on the passband. The high weight given to the passband makes the approximation more accurate on this region.

```
W = 1*(F<=0.5) + 100*(F>0.5);
Hd = design(f,'iirls','Weights',W);
```

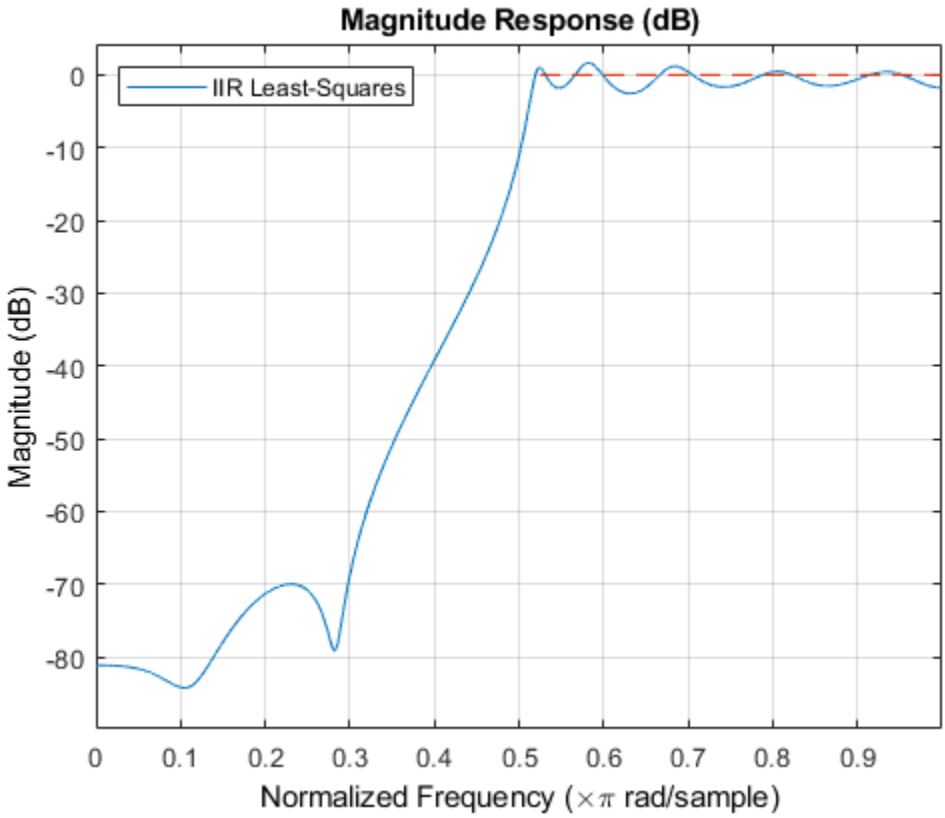
When using IIR design techniques, the stability of the filter is not guaranteed. Check the IIR stability using the `isstable` function. To do a more complete analysis, examine the poles and how close they are to the unit circle.

```
isstable(Hd)
```

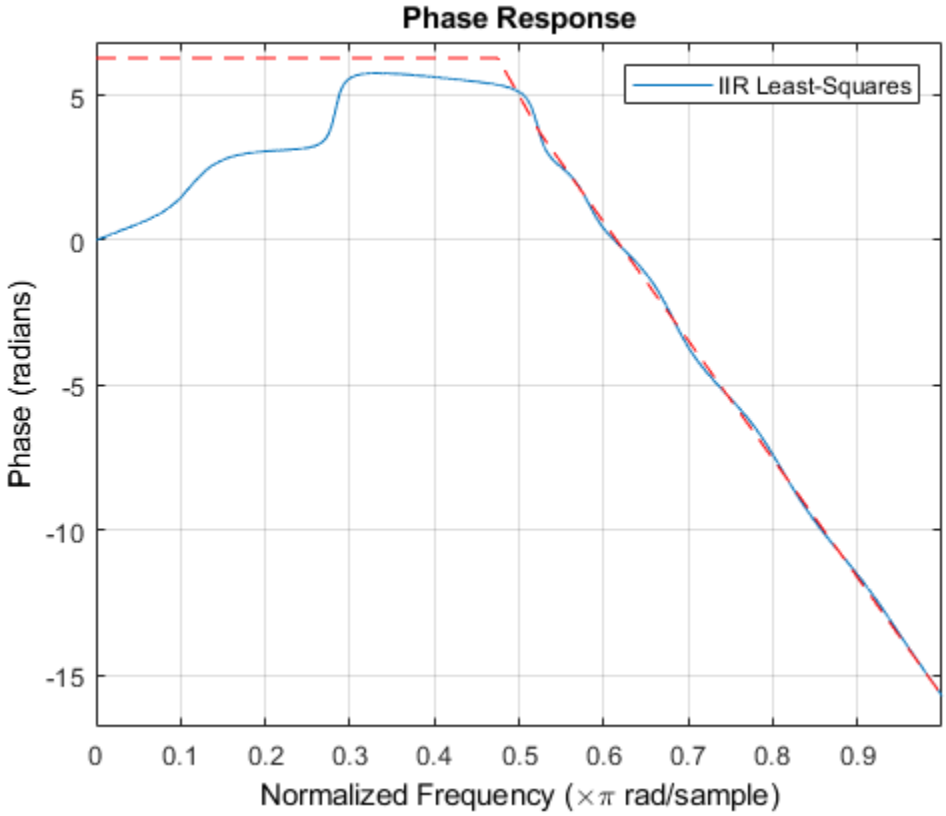
```
ans = logical
   1
```

Plot the IIR design response. Note that the approximation on the passband is better than on the stopband, and the phase response is less significant wherever the magnitude gain is small (low dB).

```
close(hfvt(1));
close(hfvt(2));
hfvt = fvtool(Hd, 'Color','w');
legend(hfvt,'IIR Least-Squares','Location', 'NorthWest')
```




```
hfvt(2) = fvtool(Hd,'Analysis','phase','Color','white');
legend(hfvt(2),'IIR Least-Squares','Location', 'NorthEast')
ax = hfvt(2).CurrentAxes;
ax.NextPlot = 'add';
plot(ax,F,unwrap(angle(H))+2*pi,'r--')
```



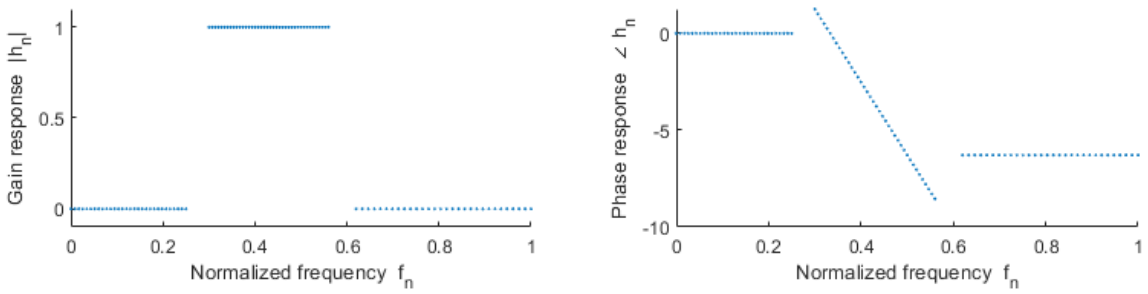
Bandpass FIR Design with a Low Group Delay

An interesting application of arbitrary magnitude and phase designs is the design of asymmetric FIR filters that sacrifice linear phase in favor of a shorter group delay. Such filters can still be designed to maintain a good approximation of a linear phase on the passband. Suppose we have three bands for the bandpass filter: a stopband on $F_1 = [0, 0.3)$ and on $F_3 = (0.6, \ 1]$, and a passband on $F_2 = [0.3, 0.6]$. On the passband, the desired frequency response is $H(\omega) = e^{-j\pi\omega \cdot \mathrm{gd}}$, which has a linear phase reponse with a group delay of *gd*.

```
F1 = linspace(0,.25,30);  % Lower stopband
F2 = linspace(.3,.56,40); % Passband
F3 = linspace(.62,1,30);  % Higher stopband

% Define the desired frequency response on the bands
gd = 12; % Desired Group Delay
H1 = zeros(size(F1));
H2 = exp(-1j*pi*F2*gd);
H3 = zeros(size(F3));
F = [F1 F2 F3];
H = [H1 H2 H3];
```

Plot the desired frequency response.

```
plotResponse(F, H)
```



Create a filter specification object using the `'N,B,F,H'` specification pattern. Here, *N* = `50` is the desired filter order, *B* = `3` stands for the number of bands, followed by *B* pairs of `F` and `H` vectors as before.

```
N = 50;  % Filter Order
B = 3;   % Number of bands
f = fdesign.arbmagnphase('N,B,F,H',N,B, F1,H1, F2,H2, F3,H3);
Hd_mnp = design(f,'equiripple');
```

This design does not have a linear phase, as can be seen by calling the `islinphase` function.

```
islinphase(Hd_mnp)
```

```
ans = logical
   0
```

Now, create a magnitude-only filter specification using `fdesign.arbmag`. The `'N,B,F,A'` specification pattern for this object is similar to the `'N,B,F,H'` specification of the `fdesign.argmagnphase` object. The difference between the two is that the complex filter response `H` in `'N,B,F,H'` is replaced with a magnitude only (nonnegative real) response `A` in `'N,B,F,A'`.

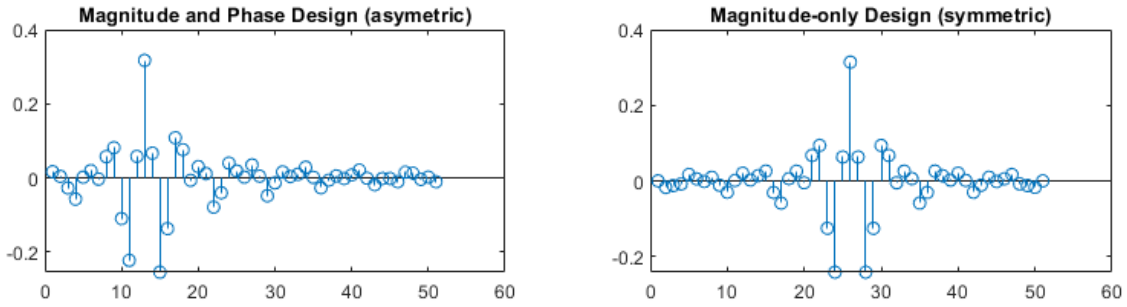
```
f_magonly = fdesign.arbmag('N,B,F,A',N,3,F1,abs(H1),F2,abs(H2),F3,abs(H3));
Hd_mo = design(f_magonly,'equiripple');
```

The magnitude-only specification yields symmetric design with a linear phase.

```
islinphase(Hd_mo)
```

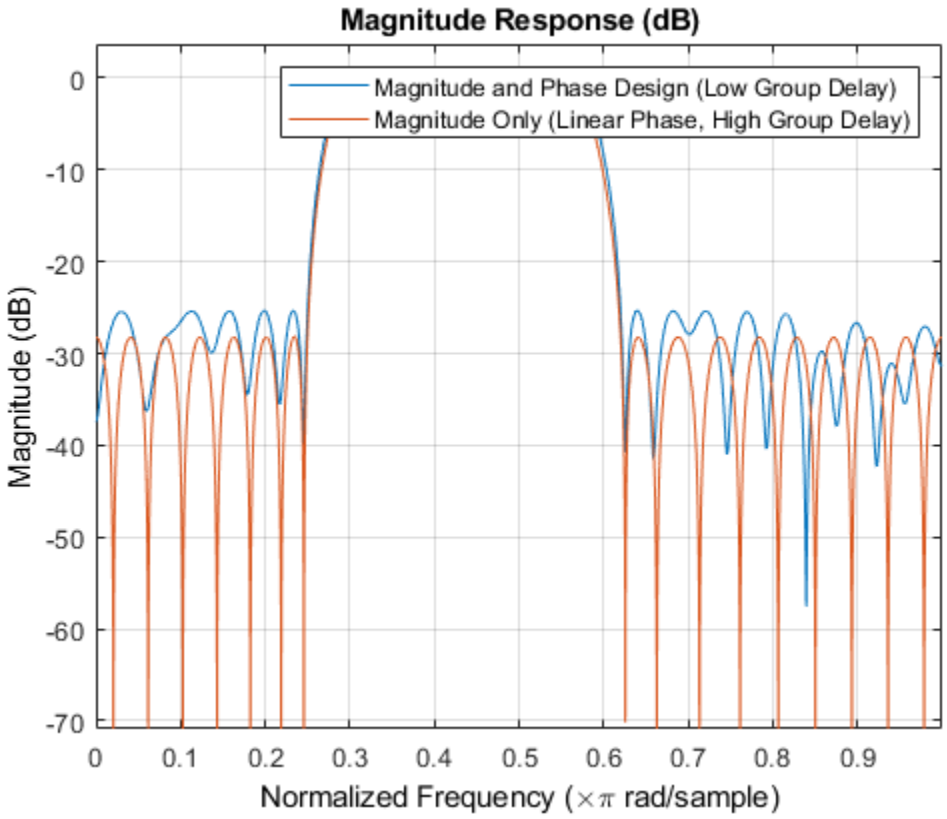
```
ans = logical
   1
```

```
subplot(1,2,1);
stem(Hd_mnp.Numerator)
title('Magnitude and Phase Design (asymetric)')
subplot(1,2,2);
stem(Hd_mo.Numerator)
title('Magnitude-only Design (symmetric)')
```



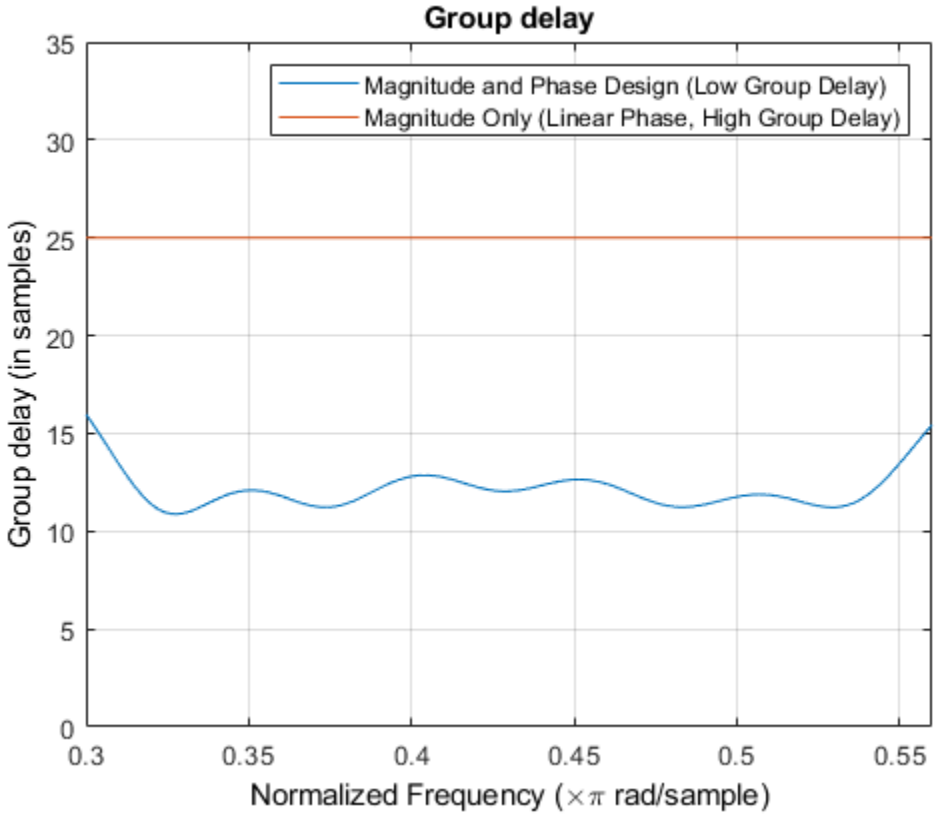
Compare the two designs. Note that they have a very similar bandpass magnitude response.

```
close(hfvt(1));
close(hfvt(2));
hfvt = fvtool(Hd_mnp,Hd_mo, 'Color', 'w');
legend(hfvt,'Magnitude and Phase Design (Low Group Delay)', 'Magnitude Only (Linear Phase, High G
```



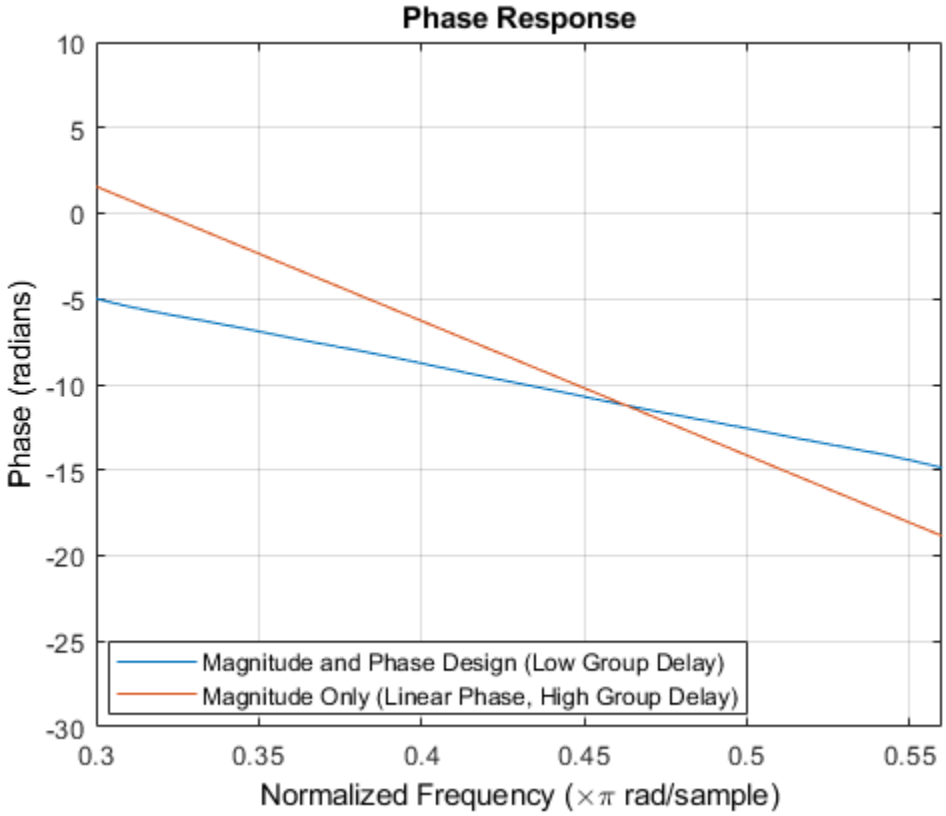
Plot the group delay. The arbitrary magnitude and phase design has a slightly varying group delay. However, the variation is small and on average is 12.5 samples. This group delay is half the group delay of the magnitude only design, which is 25 samples.

```
hfvt(2) = fvtool(Hd_mnp,Hd_mo,'Analysis','grpdelay','Color','w');
legend(hfvt(2),'Magnitude and Phase Design (Low Group Delay)', 'Magnitude Only (Linear Phase, Hig
axis([.3 .56 0 35])
```



The difference in group delays can also be seen in the phase response. The shallower slope indicates a smaller group delay.

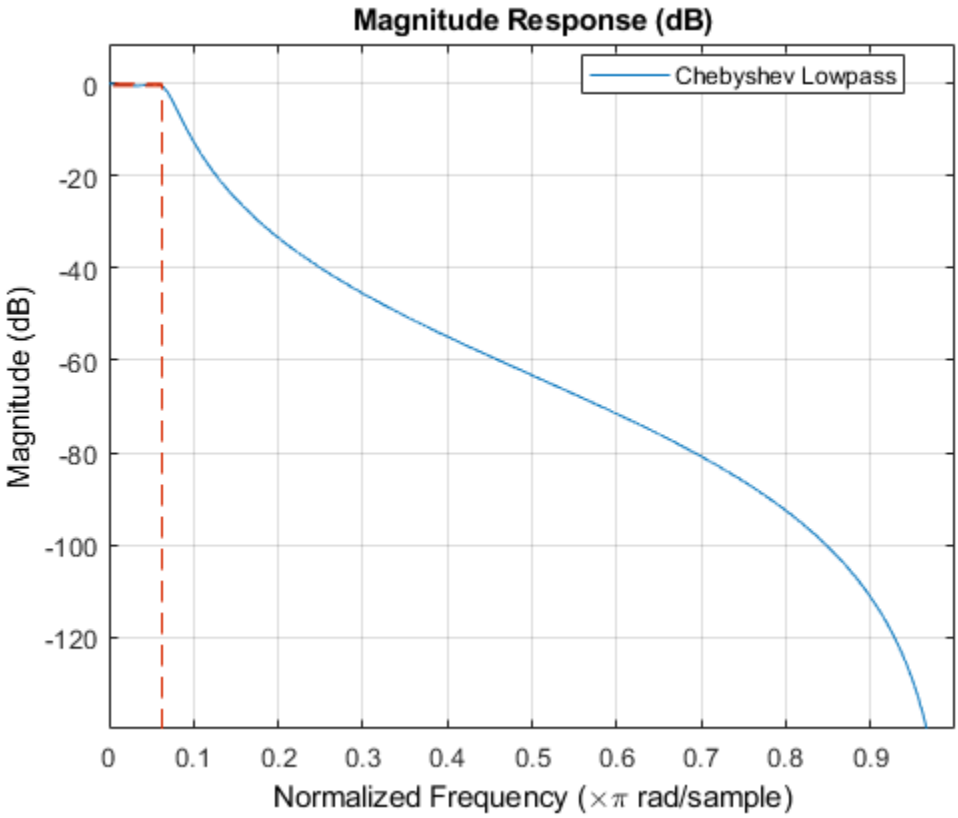
```
hfvt(2).Analysis = 'phase';
hfvt(2).Color = 'w';
axis([.3 .56 -30 10])
```



Passband Equalization of a Chebyshev Lowpass Filter

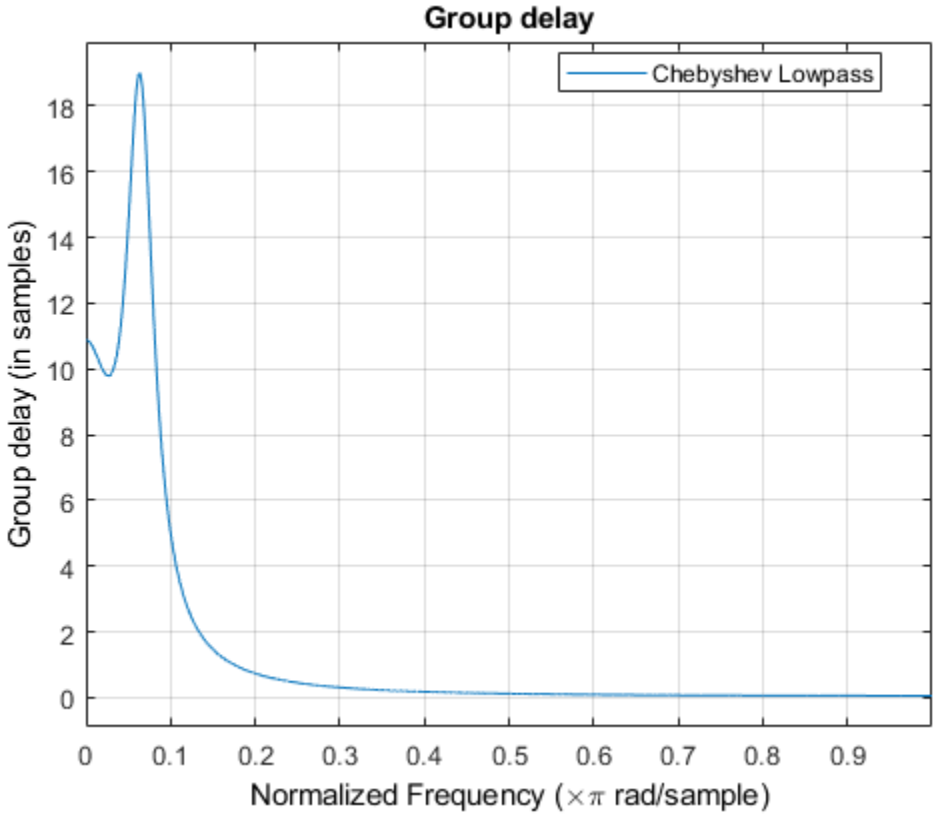
Another common application of arbitrary magnitue-phase designs is the equalization of nonlinear-phase responses of IIR filters. Consider a third order Chebyshev Type I lowpass filter with a normalized passband frequency of 1/16 and passband ripples of 0.5 dB.

```
Fp = 1/16;  % Passband frequency
Ap = .5;    % Passband ripples
f = fdesign.lowpass('N,Fp,Ap',3,Fp,Ap);
Hcheby = design(f,'cheby1');
close(hfvt(1));
close(hfvt(2));
hfvt = fvtool(Hcheby,'Color','w');
legend(hfvt, 'Chebyshev Lowpass');
```



Plot the group delay. There is a significant group delay distorition on the passband with group delays ranging from 10 to 20 samples.

```
hfvt(2) = fvtool(Hcheby,'Color','w','Analysis','grpdelay');
legend(hfvt(2), 'Chebyshev Lowpass');
```



To mitigate the distortion in the group delay, an FIR equalizer $H_{\mathrm{eq}}(\omega)$can be used after the IIR filter. Ideally, the combined filter is an ideal lowpass. The combined filter has the passband response $G(\omega) = H_{\mathrm{ch}}(\omega)H_{\mathrm{eq}}(\omega) = e^{-jg_d\omega}$, eliminating the magnitude ripples to a flat magnitude response and a constant group delay of g_d samples. The target group g_d is tied to the allotted FIR length for causal filter designs. In this example, $g_d = 35$ makes a reasonable choice.

To summarize, the equalizer design has two bands:

- On the passband, the desired frequency response of the equalizer should be $H_{\mathrm{eq}}(\omega) = e^{-jg_d\omega}/H_{\mathrm{ch}}(\omega)$.
- On the stopband, the desired response is $H_{\mathrm{eq}}(\omega) = 0$, consistent with H_{ch}.

This two-band design specification ensures that the FIR approximation of the equalizer focuses on the passband and the stopband only. The remaining parts of the frequency domain are considered don't-care regions.

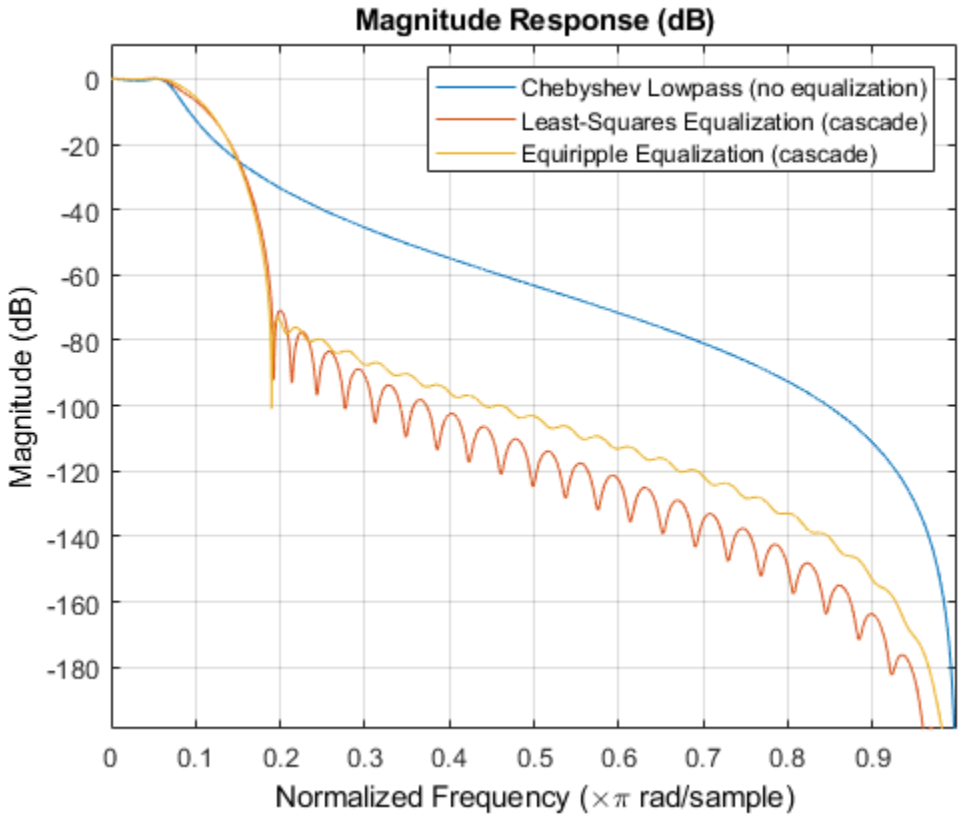
```
gd = 35;    % Passband Group Delay of the equalized filter (linear phase)
F1 = 0:5e-4:Fp; % Passband
D1 = exp(-1j*gd*pi*F1)./freqz(Hcheby,F1*pi);

Fst = 3/16; % Stopband
F2 = linspace(Fst,1,100);
D2 = zeros(1,length(F2));
```

There are several FIR design methods that can be used to implement this equalizer FIR specification. Compare the performance using two design methods: a least squares design, and an equiripple design.

```
feq = fdesign.arbmagnphase('N,B,F,H',51,2,F1,D1,F2,D2);
Heq_ls = design(feq,'firls');      % Least-Squares design
Heq_er = design(feq,'equiripple'); % Equiripple design

% Create the cascaded filters
Gls = cascade(Hcheby,Heq_ls);
Geq = cascade(Hcheby,Heq_er);
```

Plot the magnitude responses of the cascaded systems for both filters.

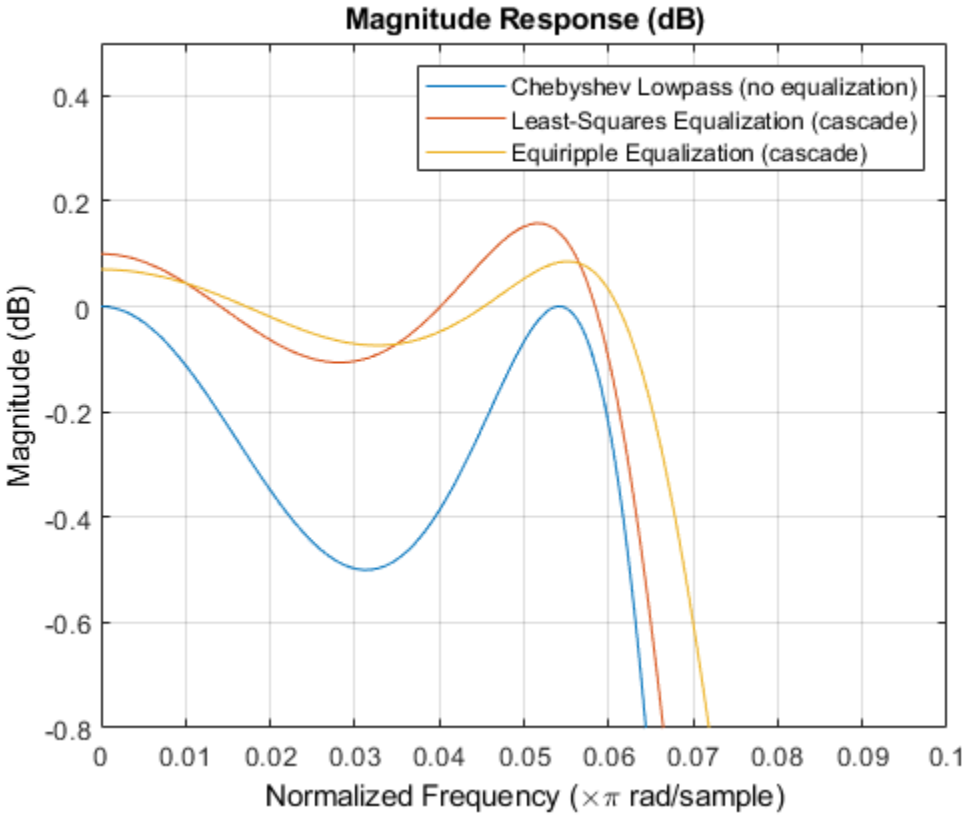
```
close(hfvt(1));
close(hfvt(2));
hfvt = fvtool(Hcheby,Gls, Geq, 'Color','w');
legend(hfvt,'Chebyshev Lowpass (no equalization)','Least-Squares Equalization (cascade)', ...
    'Equiripple Equalization (cascade)', 'Location', 'NorthEast')
```



Zoom in around the passband. The passband ripples are attenuated after equalization from 0.5 dB in the original filter to 0.27 dB with the least-squares designed equalizer, and to 0.16 dB with the equiripple designed equalizer.

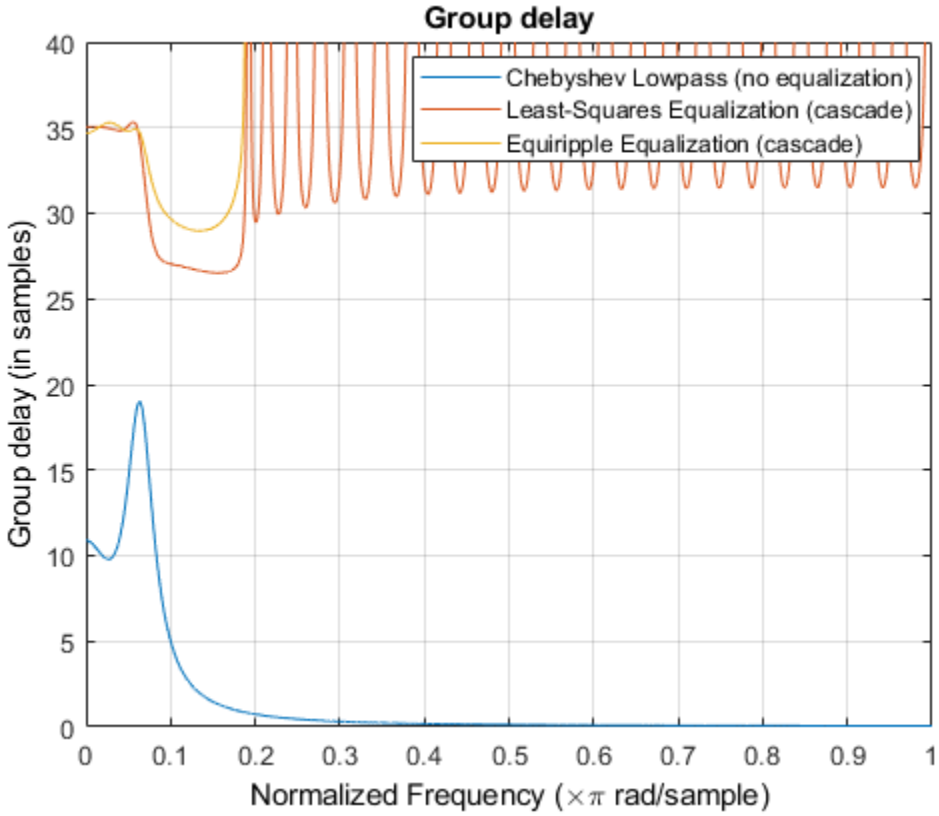
```
hfvt(2) = fvtool(Hcheby,Gls,Geq, 'Color','w');
legend(hfvt(2),'Chebyshev Lowpass (no equalization)', ...
    'Least-Squares Equalization (cascade)', ...
```

```
    'Equiripple Equalization (cascade)', 'Location', 'NorthEast')
axis([0 .1 -0.8 .5])
```



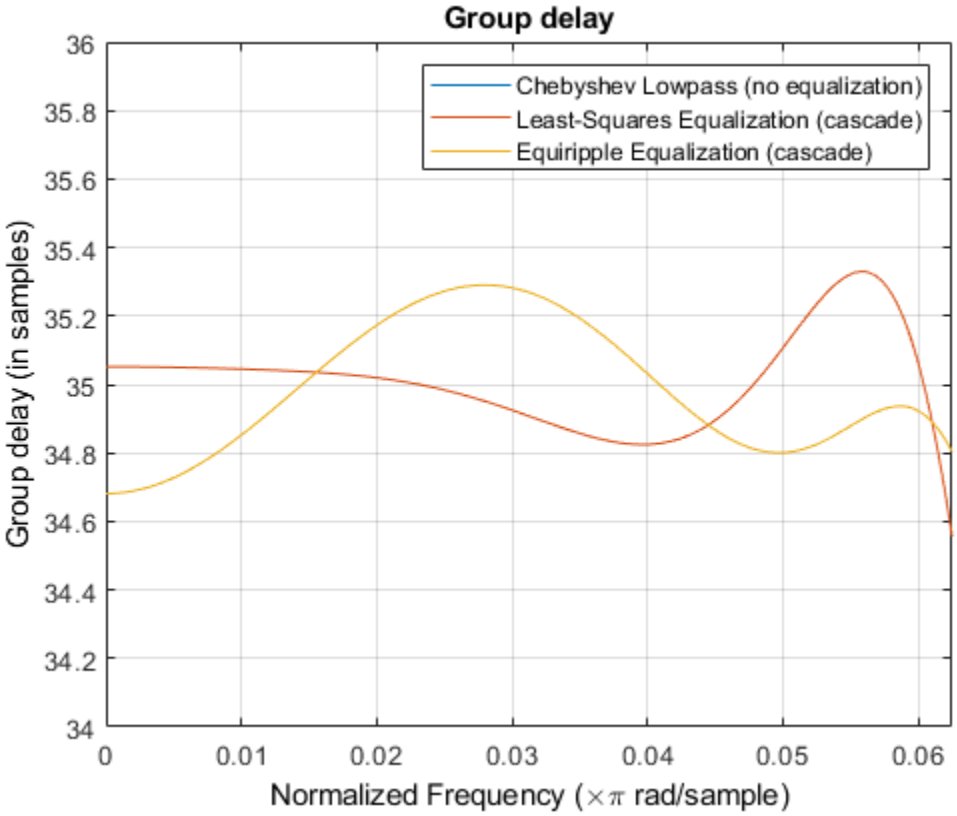
We now turn to the phase (and group delay) equalization. The combined group delay is nearly constant around 35 samples (the target group delay) on the passband. Outside the passband, the combined group delay is seemingly divergent, but this is insignificant as the gain of the filter vanishes on that region.

```
hfvt(2).Analysis = 'grpdelay';
axis([0 1 0 40])
```



Zoom in around the passband. The group delay in the passband is equalized from a peak-to-peak difference of 8.8 samples to 0.51 samples with the least-squares equalizer, and to 0.62 samples with the equiripple equalizer. Both equalizers perform equally well.

```
hfvt(3) = fvtool(Hcheby,Gls,Geq, 'Analysis', 'grpdelay','Color','w');
legend(hfvt(3),'Chebyshev Lowpass (no equalization)', ...
    'Least-Squares Equalization (cascade)', ...
    'Equiripple Equalization (cascade)', 'Location', 'NorthEast')
axis([0 Fp 34 36])
```




```
close(hfvt(1));
close(hfvt(2));
close(hfvt(3));
```

References

[1] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989.

G.729 Voice Activity Detection

This example shows how to implement the ITU-T G.729 Voice Activity Detector (VAD)

Introduction

Voice Activity Detection (VAD) is a critical problem in many speech/audio applications including speech coding, speech recognition or speech enhancement. For instance, the ITU-T G.729 standard uses VAD modules to reduce the transmission rate during silence periods of speech.

Algorithm

At the first stage, four parametric features are extracted from the input signal. These parameters are the full-band and low-band frame energies, the set of line spectral frequencies (LSF) and the frame zero crossing rate. If the frame number is less than 32, an initialization stage of the long-term averages takes place, and the voice activity decision is forced to 1 if the frame energy from the LPC analysis is above 21 dB. Otherwise, the voice activity decision is forced to 0. If the frame number is equal to 32, an initialization stage for the characteristic energies of the background noise occurs.

At the next stage, a set of difference parameters is calculated. This set is generated as a difference measure between the current frame parameters and running averages of the background noise characteristics. Four difference measures are calculated:

```
a) A spectral distortion

b) An energy difference

c) A low-band energy difference

d) A zero-crossing difference
```

The initial voice activity decision is made at the next stage, using multi-boundary decision regions in the space of the four difference measures. The active voice decision is given as the union of the decision regions and the non-active voice decision is its complementary logical decision. Energy considerations, together with neighboring past frames decisions, are used for decision smoothing. The running averages have to be updated only in the presence of background noise, and not in the presence of speech. An adaptive threshold is tested, and the update takes place only if the threshold criterion is met.

VAD Implementation

vadG729 is the function containing the algorithm's implementation.

Initialization

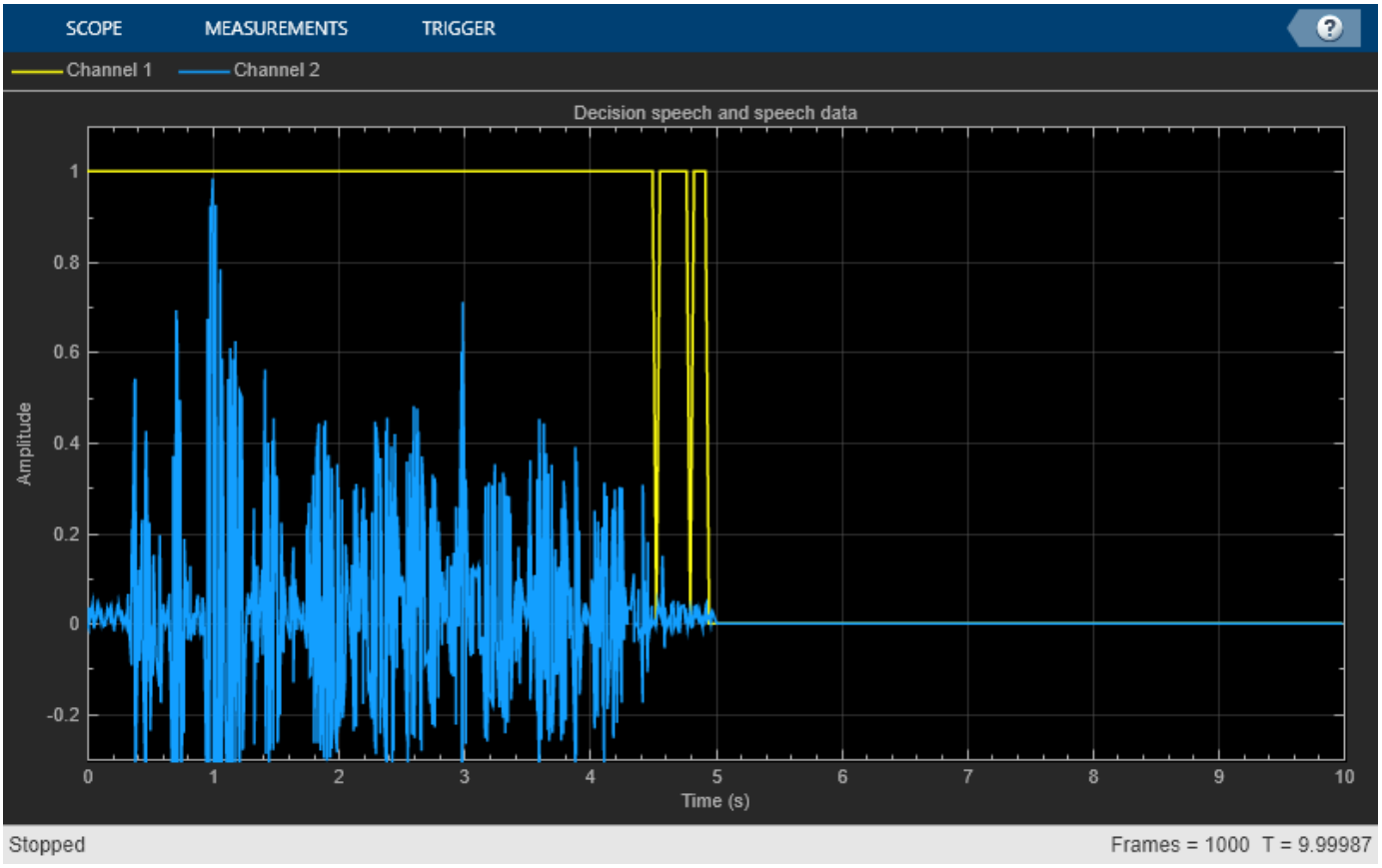
Set up an audio source. This example uses an audio file reader.

```
audioSource = dsp.AudioFileReader('SamplesPerFrame',80,...
                                  'Filename','speech_dft_8kHz.wav',...
                                  'OutputDataType', 'single');
% Note: You can use a microphone as a source instead by using an audio
% device reader (NOTE: audioDeviceReader requires an Audio Toolbox
% (TM) license)
% audioSource = audioDeviceReader('OutputDataType', 'single', ...
%                                 'NumChannels', 1, ...
%                                 'SamplesPerFrame', 80, ...
```

```
%                               'SampleRate', 8000);
% Create a time scope to visualize the VAD decision (channel 1) and the
% speech data (channel 2)
scope = timescope('SampleRate', [8000/80 8000], ...
                  'TimeSpanSource', 'property', ...
                  'TimeSpan', 10, ...
                  'YLimits', [-0.3 1.1], ...
                  'Title','Decision speech and speech data', ...
                  'TimeSpanOverrunAction','Scroll');
```

Stream Processing Loop

```
% Initialize VAD parameters
VAD_cst_param = vadInitCstParams;
clear vadG729
% Run for 10 seconds
numTSteps = 1000;
while(numTSteps)
  % Retrieve 10 ms of speech data from the audio recorder
  speech = audioSource();
  % Call the VAD algorithm
  decision = vadG729(speech, VAD_cst_param);
  % Plot speech frame and decision: 1 for speech, 0 for silence
  scope(decision, speech);
  numTSteps = numTSteps - 1;
end
release(scope);
```



Cleanup

Close the audio input device and release resources

```
release(audioSource);
```

Generating and Using the MEX-File

MATLAB Coder can be used to generate C code for the function vadG729. In order to generate a MEX-file, execute the following command.

```
codegen vadG729 -args {single(zeros(80,1)), coder.Constant(VAD_cst_param)}
```

```
Code generation successful.
```

Speed Comparison

Creating MEX-Files often helps achieve faster run-times for simulations. The following lines of code first measure the time taken by the MATLAB function and then measure the time for the run of the corresponding MEX-file. Note that the speedup factor may be different for different machines.

```
audioSource = dsp.AudioFileReader('speech_dft_8kHz.wav', ...
                                  'SamplesPerFrame', 80, ...
                                  'OutputDataType', 'single');
clear vadG729
VAD_cst_param = vadInitCstParams;
tic;
while ~isDone(audioSource)
  speech = audioSource();
  decision = vadG729(speech, VAD_cst_param);
end
t1 = toc;

reset(audioSource);

tic;
while ~isDone(audioSource)
  speech = audioSource();
  decision = vadG729_mex(speech, VAD_cst_param);
end
t2 = toc;

disp('RESULTS:')
disp(['Time taken to run the MATLAB code: ', num2str(t1), ' seconds']);
disp(['Time taken to run the MEX-File: ', num2str(t2), ' seconds']);
disp(['Speed-up by a factor of ', num2str(t1/t2),...
    ' is achieved by creating the MEX-File']);
```

```
RESULTS:
Time taken to run the MATLAB code: 0.64393 seconds
Time taken to run the MEX-File: 0.15446 seconds
Speed-up by a factor of 4.1689 is achieved by creating the MEX-File
```

Reference

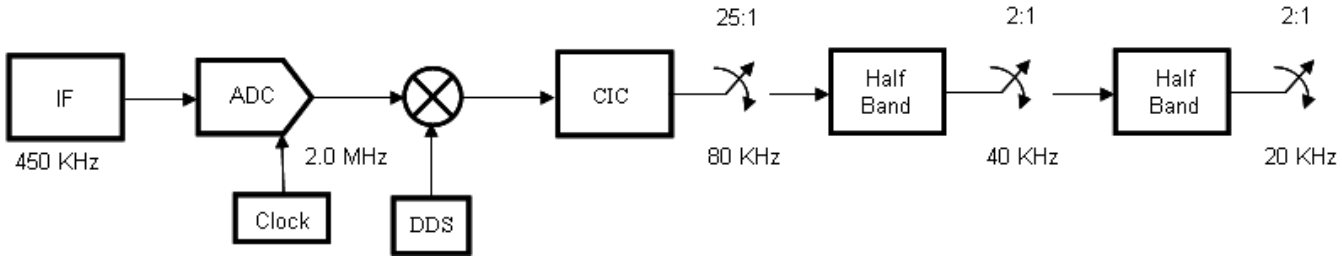
ITU-T Recommendation G.729 - Annex B: A silence compression scheme for G.729 optimized for terminals conforming to ITU-T Recommendation V.70

IF Subsampling with Complex Multirate Filters

This example shows how to use complex multirate filters in the implementation of Digital Down-Converters (DDC). The DDC is a key component of digital radios. It performs the frequency translation necessary to convert the high input sample rates typically found at the output of an analog-to-digital (A/D) converter down to lower sample rates for further and easier processing. In this example, we will see how an audio signal modulated with a 450 kHz carrier frequency can be brought down to a 20 kHz sampling frequency. After a brief review of the conventional DDC architecture, we will describe an alternative solution known as Intermediate Frequency (IF) subsampling and we will compare the respective implementation cost of these two solutions. This example requires a Fixed-Point Designer™ license.

Conventional Digital Down Converter

A conventional down conversion process starts with sampling the analog signal at a rate that satisfies the Nyquist criterion for the carrier. A possible option would be to sample the 450 kHz input signal at 2.0 MHz, then using a digital down converter to perform complex translation to baseband, filter and down sample by 25 with a Cascaded Integrator-Comb (CIC) filter, and then downsample by 4 with a pair of halfband filters. Such an implementation is shown below:



CIC Filter Design

The first filter of the conventional DDC is usually a CIC filter. CIC filters are efficient, multiplier-less structures which are used in high-decimation or interpolation systems. In our case it will bring the 2 MHz signal down to 2.0 MHz/25 = 80 kHz.

```
Fs_normDDC = 2e6;          % Sampling frequency
R          = 25;           % Decimation factor
Fpass      = 10e3;         % Passband Frequency
Astop      = 60;           % Aliasing Attenuation(dB)
D          = 1;            % Differential delay
dcic = fdesign.decimator(R,'cic',D,Fpass,Astop,Fs_normDDC);
cic = design(dcic,'SystemObject',true);
cicgain = dsp.FIRFilter('Numerator',1/gain(cic)); % Normalize gain of CIC
```

Compensation FIR Decimator Design

The second filter of the conventional DDC compensates for the passband droop caused by the CIC. Since the CIC has a sinc-like response, it can be compensated for the droop with a lowpass filter that has an inverse-sinc response in the passband.

```
Nsecs = cic.NumSections;       % Number of sections
Fpass  = 10e3;                 % Passband Frequency
Fstop  = 25e3;                 % Stopband Frequency
Apass  = 0.01;                 % Passband Ripple (dB)
```

```
Astop  = 80;                    % Stopband Attenuation (dB)
dcp = fdesign.decimator(2,'ciccomp', ...
    D,Nsecs,Fpass,Fstop,Apass,Astop,dcic.Fs_out);
cfir = design(dcp,'equiripple', ...
    'StopBandShape','linear','StopBandDecay',60,'SystemObject',true);
```

Halfband Filter Design

We finally use a 20th order halfband filter to bring the 40 kHz signal down to 20 kHz.

```
dhbfilter = fdesign.decimator(2,'halfband','N',20,dcp.Fs_out);
hbfilter = design(dhbfilter,'SystemObject',true);
```

The conventional DDC filter is obtained by cascading the three stages previously designed.

```
normDDCFilter = cascade(cicgain,cic,cfir,hbfilter);
```

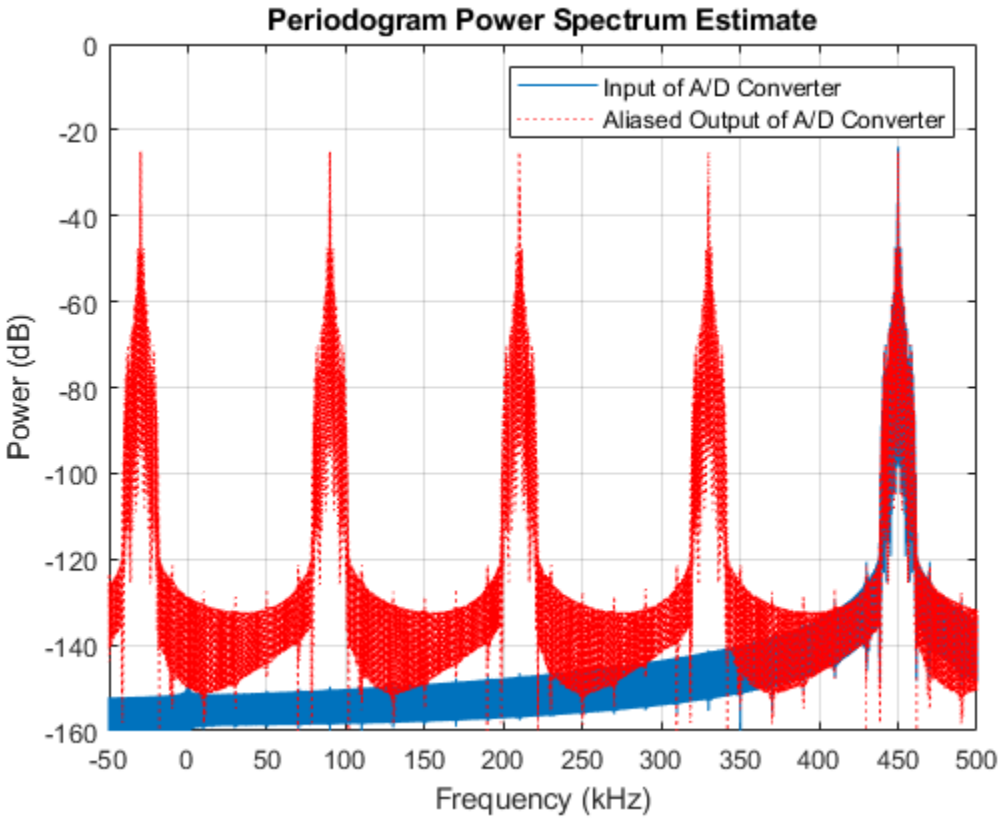
IF Subsampling

Since the carrier frequency is discarded as part of the signal extraction, there is no need to preserve it during the data-sampling process. The Nyquist criterion for the carrier can actually be violated as long as Nyquist criterion for the bandwidth of complex envelope is satisfied.

This narrowband interpretation of the Nyquist criterion leads to an alternate data collection process known as IF subsampling. In this process, the A/D converter's sample rate is selected to be less than the signal's center frequency to intentionally alias the center frequency. Since Nyquist criterion is being intentionally violated, the analog signal must be conditioned to prevent multiple frequency intervals from aliasing to the same frequency location as the desired signal component will alias.

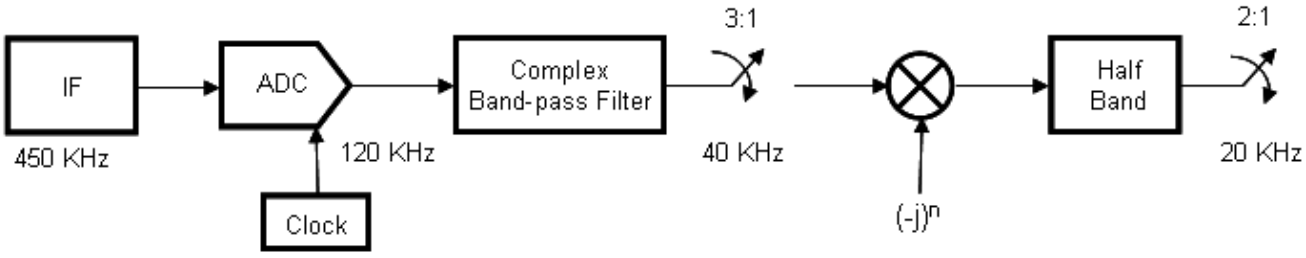
The variable y represents approximately 3 sec of an audio signal modulated with a 450 kHz carrier frequency. The discrete signal ys represents the output of a 120 kHz A/D converter.

```
[y,ys,Fs] = loadadcio;
Fs_altDDC = 1.2e5;   % Sampling frequency

[Hys,Fys] = periodogram(ys,[],[],Fs,'power','centered');
N = length(Fys);

figure('color','white')
periodogram(y,[],[],Fs,'power','centered');
clear y;
hold on;
plot((-(ceil(N/2*9)-1):floor(N/2*9))/N*Fs_altDDC/1000, ...
        repmat(10*log10(Hys),9,1),'r:');
axis([-50 500 -160 0])
legend('Input of A/D Converter','Aliased Output of A/D Converter', ...
    'Location','NorthEast');
```



The frequency band around 450 kHz aliased around -30 kHz. Aliasing to a quarter of the sampling frequency maximizes the separation between positive and negative frequency aliases. This permits maximum transition bandwidth for the analog bandpass filter and therefore minimizes its cost.

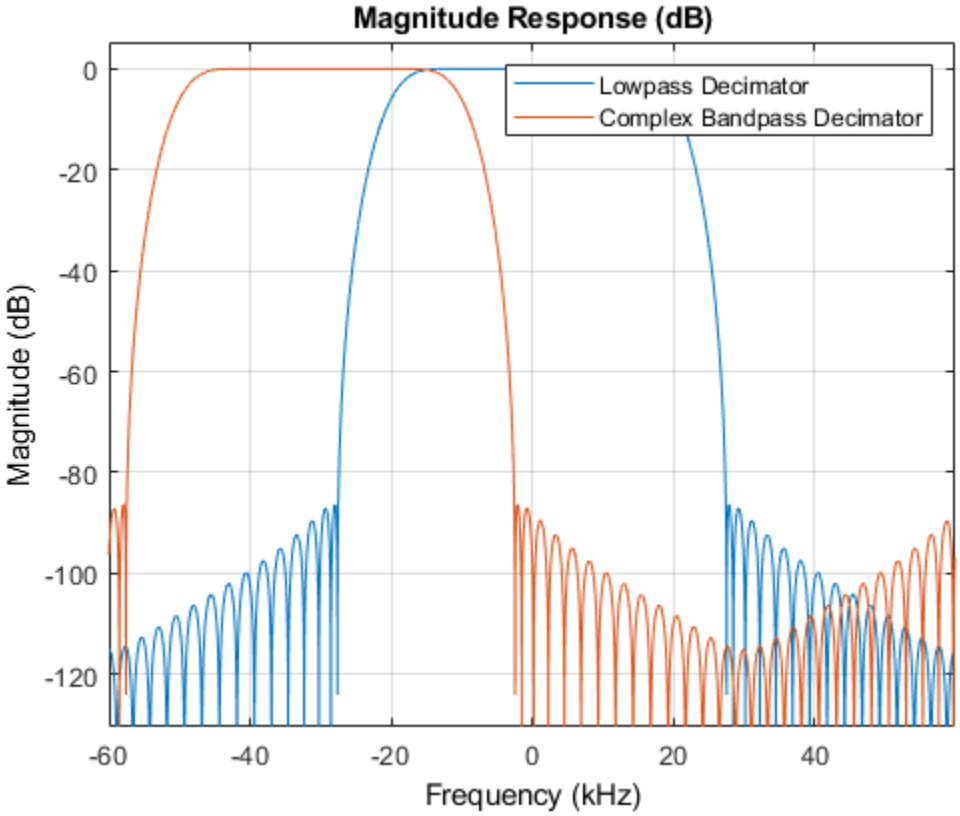
The choice of a 120 kHz sampling frequency also eases the subsequent task of down converting to 20 kHz which is accomplished by down sampling by a factor of 6. The down conversion can be achieved in two stages. First a 3-to-1 downsampling is performed by a complex bandpass filter followed by a 2-to-1 conversion with a halfband filter. The structure of this aliasing DDC is shown below.



Complex Bandpass Filter Design

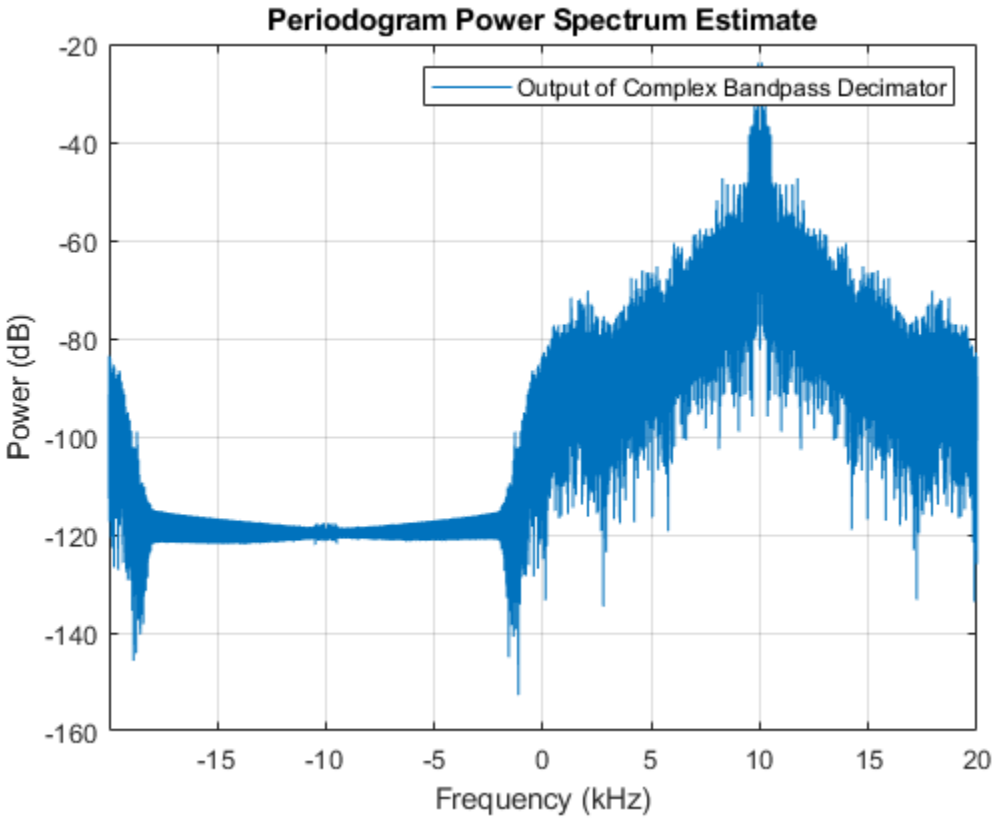
To obtain a complex bandpass filter, we translate a lowpass decimator prototype to a quarter sample rate by multiplying the filter coefficients with the heterodyne terms exp(-j*pi/2*n). Notice that while the coefficients of the lowpass filter are real, the coefficients of the translated filter are complex. The figure below depicts the magnitude responses of these filters.

```
M = 3;               % Decimation Factor
TW   = Fstop-Fpass;  % Transition Width (Hz)
designLowpass = fdesign.decimator(M,'nyquist',M,TW,Astop,Fs_altDDC);
lpfilter = design(designLowpass,'SystemObject',true); % Lowpass prototype
n = 0:length(lpfilter.Numerator)-1;
complexBPFilter = dsp.FIRDecimator(M,lpfilter.Numerator.*exp(-1i*pi/2*n));
fvt = fvtool(lpfilter,complexBPFilter,'Fs',Fs_altDDC,'Color','White');
legend(fvt,'Lowpass Decimator','Complex Bandpass Decimator', ...
    'Location','NorthEast')
```



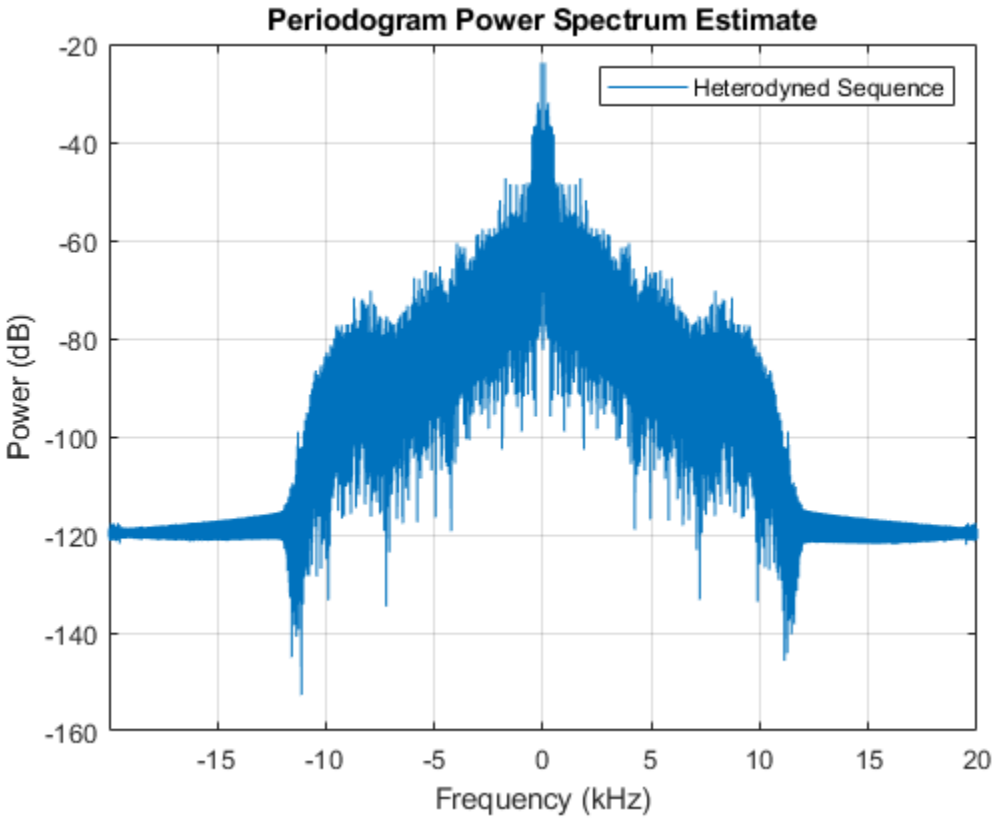
We now apply the complex bandpass decimator to the output of A/D converter. It can be shown that a signal at a quarter sample-rate will always alias to a multiple of the quarter sample-rate under decimation by any integer factor. In our example the -30 kHz centered signal will alias to 40/4 = 10 kHz.

```
ycbp = complexBPFilter(ys);
figure('color','white')
periodogram(ycbp,[],[],designLowpass.Fs_out,'power','centered');
legend('Output of Complex Bandpass Decimator')
```



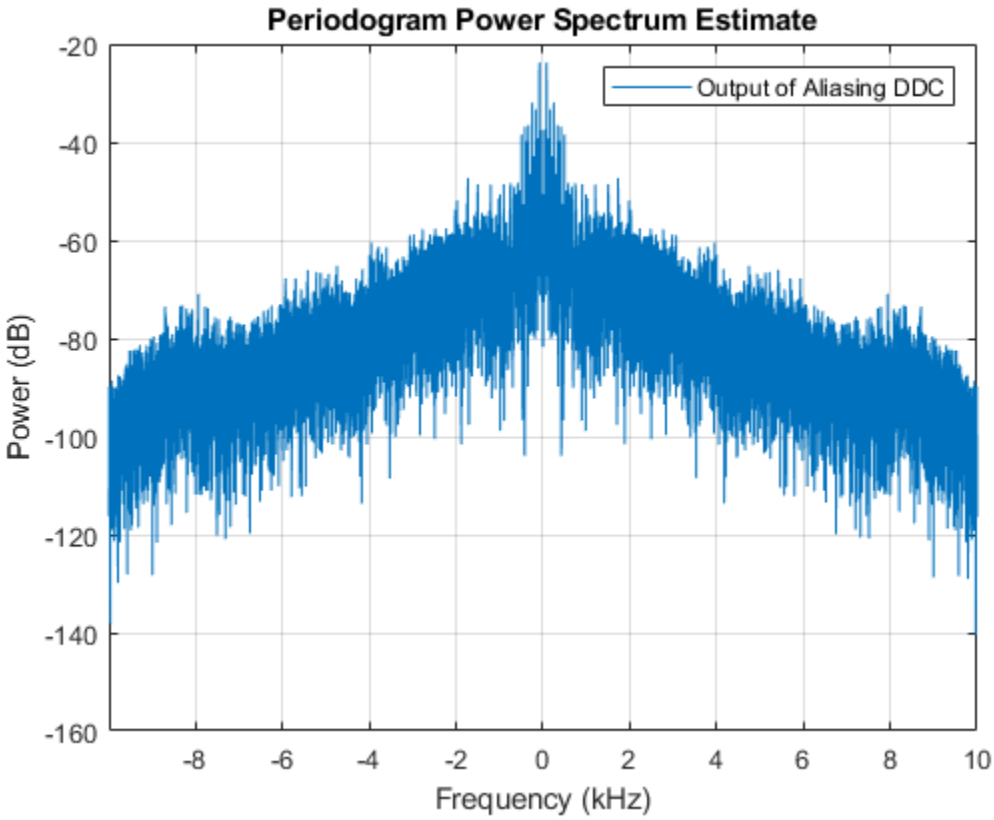
The output sequence ycbp is then heterodyned to zero.

```
yht = ycbp.*(-1i).^(0:length(ycbp)-1).';
figure('color','white')
periodogram(yht,[],[],designLowpass.Fs_out,'power','centered');
legend('Heterodyned Sequence')
```



Finally the heterodyned sequence is passed as input to the halfband filter and decimated by 2. We can reuse the same halfband filter as in the conventional DDC.

```
yf = hbfilter(yht);
figure('color','white')
periodogram(yf,[],[],dhbfilter.Fs_out,'power','centered');
legend('Output of Aliasing DDC')
```



Play the audio signal at the output of the "aliasing" DDC. (Copyright 2002 FingerBomb)

```
player = audioDeviceWriter('SampleRate',dhbfilter.Fs_out);
player(real(yf));
```

Implementation Cost Comparison

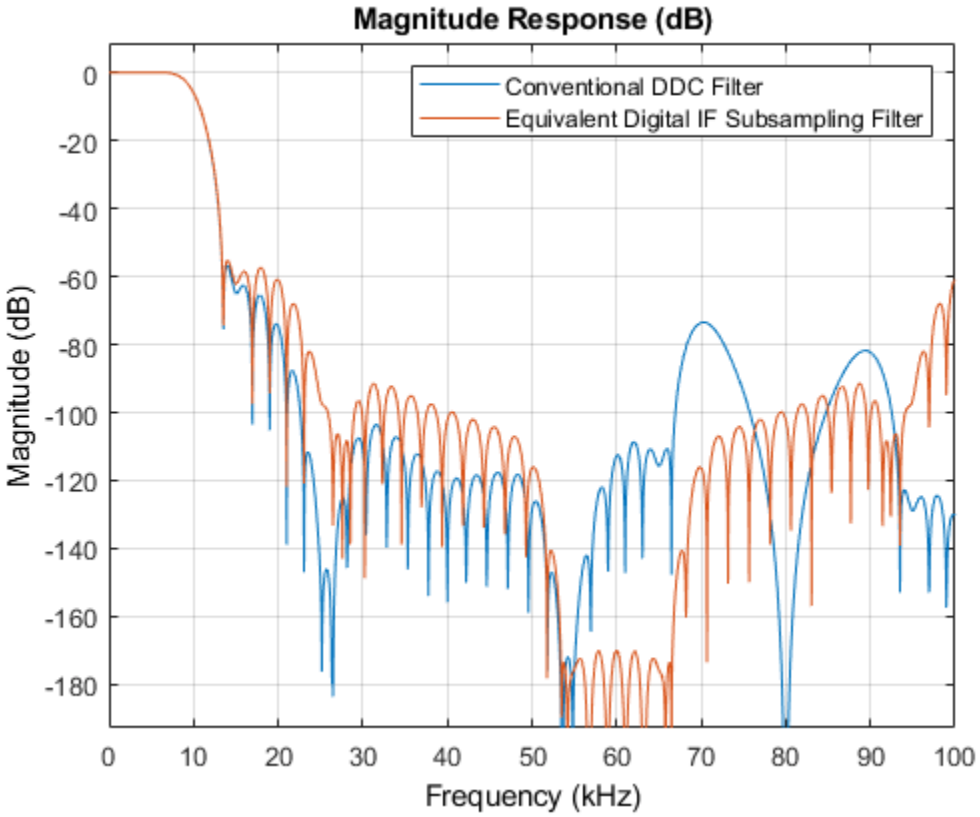
Before we proceed to the cost analysis let's verify that the magnitude responses of the filters in the two DDC solutions are comparable. We exclude both the complex translation to baseband in the conventional DDC case and the heterodyne in the IF subsampling case. Furthermore, we use the lowpass prototype decimator in the latter case since it has the same transition width, passband ripples, and stopband attenuation as the complex bandpass decimator.

```
altDDCFilter = cascade(lpfilter,hbfilter);
```

We verify that the filters used in both cases have very similar magnitude responses: less than 0.04 dB passband ripple, a 6dB cutoff frequency of 10 kHz and a 55 dB stopband attenuation at 13.4 kHz. It is therefore fair to proceed to the cost analysis.

```
set(fvt, ...
    'Filters',{normDDCFilter,altDDCFilter}, ...
    'FrequencyRange','Specify freq. vector', ...
    'FrequencyVector',linspace(0,100e3,2048), ...
    'Fs',[Fs_normDDC,Fs_altDDC], ...
    'ShowReference','off', ...
    'Color','White');
legend(fvt,'Conventional DDC Filter', ...
```

```
    'Equivalent Digital IF Subsampling Filter', ...
    'Location','NorthEast');
```



In the case of the conventional DDC, we must first take into account the cost of the baseband translation. We are assuming it is done with only one multiplier working at 2 MHz. We must then add the cost of the CIC and halfband filters. In the IF subsampling case, we must consider the cost of the heterodyne. We are assuming it is done with only one multiplier working at 40 kHz. We must then add the cost of the complex bandpass and halfband filters.

```
% Cost of CIC and halfband filters
c_normDDC = cost(normDDCFilter);
% Cost of complex bandpass and halfband filters
c_altDDC  = cost(cascade(complexBPFilter,hbfilter));
ddccostcomp(Fs_normDDC,c_normDDC,Fs_altDDC,c_altDDC)
```

```
ans =

    'Total Cost                     : Conventional DDC | IF subsampling
     ------------------------------------------------------------------
     Number of Coefficients         :               36 |              42
     Number of States               :               50 |              62
     Multiplications per microsecond :             5.18 |             1.5
     Additions per microsecond      :             9.44 |             1.4'
```

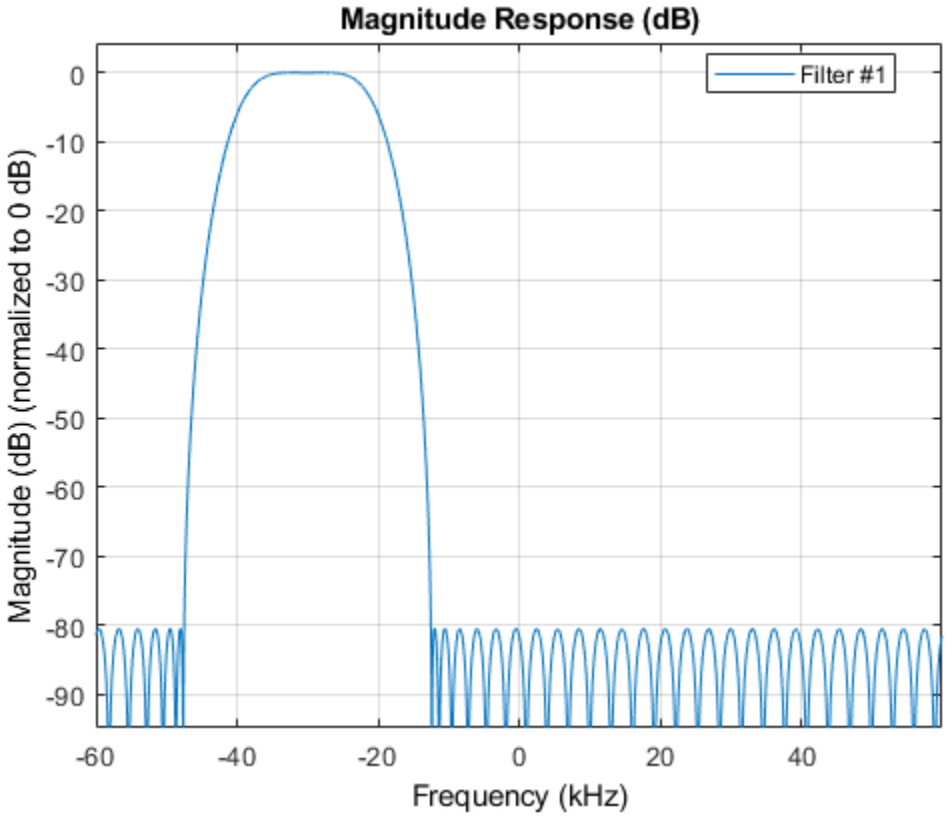
The number of multipliers, adders, and states required in the IF subsampling case is comparable to that of conventional DDC but the number of operations per second is significantly reduced since it

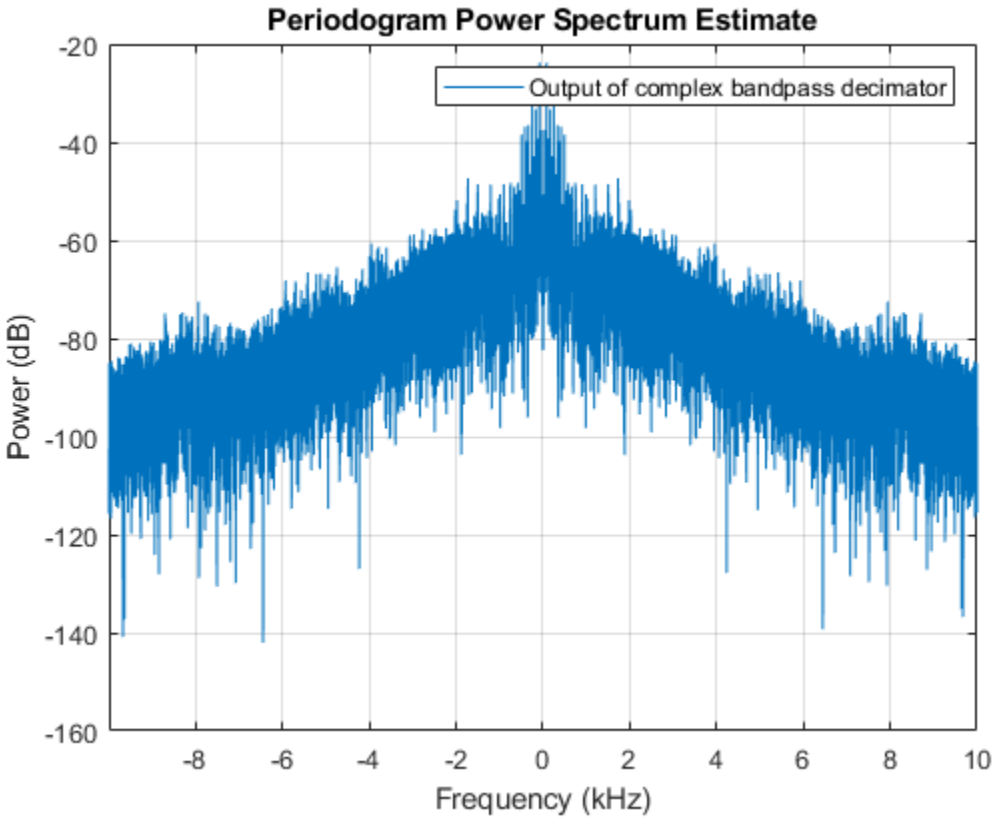
saves 71% of the number of multiplications per second and 85% of the number of additions per second.

Using dsp.ComplexBandpassDecimator

We can design the complex bandpass filter more easily by using the dsp.ComplexBandpassDecimator System object. The object designs the bandpass filter based on the specified decimation factor, center frequency, and sample rate. There is no need to translate lowpass coefficients to bandpass as we did in the design above: the object will do it for us. Moreover, the object will derive the frequency to which the filtered signal is aliased, and mix it back to zero Hz for us.

```
% Design a complex bandpass filter. Include the decimate-by-2 halfband
% filter into the design by specifying a decimation factor of 2*M:
bp = dsp.ComplexBandpassDecimator(M*2 , -30e3, Fs_altDDC,...
                                  'TransitionWidth',TW);
% Visualize the filter response
visualizeFilterStages(bp);
% Filter the output of the 120 kHz A/D converter
yf = bp(ys);
figure('color','white')
periodogram(yf,[],[],dhbfilter.Fs_out,'power','centered');
legend('Output of complex bandpass decimator')
player(real(yf));
```





Summary

This example showed how complex multirate filters can be used when designing IF subsampling-based digital down converters. The IF subsampling technique can be cost-efficient alternative to conventional DDCs in many applications. For more information on IF subsampling see Multirate Signal Processing for Communication Systems by fredric j harris, Prentice Hall, 2004.

Design and Analysis of a Digital Down Converter

This example shows how to use the digital down converter (DDC) System object to emulate the TI Graychip 4016 digital down converter in a simple manner. We base the example on a comparison with the “GSM Digital Down Converter in MATLAB” on page 4-346 example. We show how the DDC System object can be used to design and analyze the decimation filters and to quickly explore different design options that meet various passband and stopband frequency and attenuation specifications. This example requires a Fixed-Point Designer™ license.

```
if ~isfixptinstalled
    error(message('dsp:dspDigitalDownConverterDesign:noFixptTbx'));
end
```

Introduction

The “GSM Digital Down Converter in MATLAB” on page 4-346 example presents the steps required to emulate the TI Graychip 4016 digital down converter that brings a passband signal centered at 14.44 MHz to baseband, and down samples the signal by a factor of 256 to bring the input sample rate of 69.333 MHz down to 270.83 KHz. In that example you go through the following steps:

1) Design a numerically controlled oscillator to generate a mixer frequency of 14.44 MHz.

2) Load a pre-defined set of coefficients to generate a CIC decimator filter, a CIC compensator filter, and an FIR filter with a passband frequency of 80 KHz.

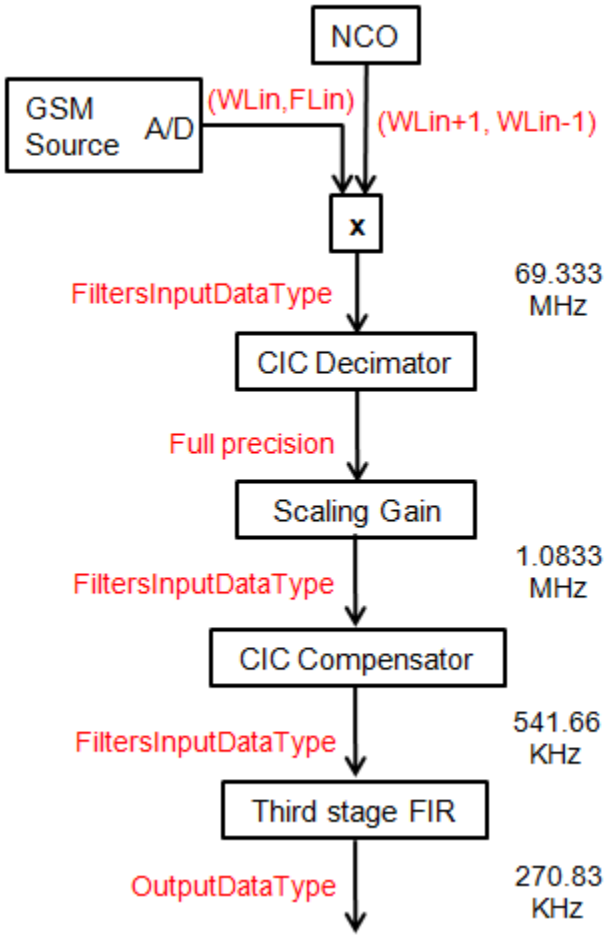
3) Frequency down convert a GSM signal (simulated as a complex exponential) and down sample the down converted output with the cascade of decimation filters.

4) Perform data casting to obtain the desired fixed-point data types across the different down converter sections.

The “GSM Digital Down Converter in MATLAB” on page 4-346 example also creates an FIR rate converter to resample the data at the output of the third filter stage. The DDC System object does not contain a rate converter; therefore, this example does not include implementing one.

This example shows how to use the DDC System object to design the set of decimation filters. It also shows how the DDC object achieves the down conversion process with fewer and simpler steps.

The following DDC System object block diagram contains the data types at each stage and the data rates for the example at hand. You control the input data type of the filters using the FiltersInputDataType and CustomFiltersInputDataType properties. You control the output data type using the OutputDataType and CustomOutputDataType properties.



Defining the DigitalDownConverter System Object

Create a DDC System object. Set the input sample rate of the object to 69.333 mega samples per second (MSPS), and the decimation factor to 256 to achieve an output sample rate of 270.83 KHz. The DDC object automatically factors the decimation value so that the CIC filter decimates by 64, the CIC compensator decimates by 2, and the third stage filter decimates by 2.

```
ddc = dsp.DigitalDownConverter('SampleRate',69.333e6, ...
    'DecimationFactor',256);
```

Decimation Filter Design

The “GSM Digital Down Converter in MATLAB” on page 4-346 example uses a predefined set of filter coefficients to generate two FIR decimators and a CIC decimator System object. Designing decimation filters so that their cascade response meets a given set of passband and stopband attenuation and frequency specifications can be a cumbersome process where you have to choose the correct combination of passband and stopband frequencies for each filter stage. Choosing stopband frequencies properly ensures lower order filter designs.

The DDC object automatically designs the decimation filters based on a set of passband and stopband attenuation and frequency specifications.

Minimum Order Filter Designs

The DDC object obtains minimum order decimation filter designs with the passband and stopband attenuation and frequency specifications you provide. Set the `MinimumOrderDesign` property to true to obtain minimum order filter designs.

```
ddc.MinimumOrderDesign = true;
```

The down converter processes a GSM signal with a double-sided bandwidth of 160 KHz. Set the Bandwidth property of the DDC object to 160 KHz so that the passband frequency of the decimation filter cascade equals 160e3/2 = 80 KHz.

Set the `StopbandFrequencySource` property to `'Auto'` so that the DDC object sets the cutoff frequency of the cascade response approximately at the output Nyquist rate, i.e. at 270.83e3/2 = 135.4 KHz, and the stopband frequency at 2Fc-Fpass = 2*135.4e3 - 160e3/2 = 190.8 KHz, where Fc is the cutoff frequency and Fpass is the passband frequency. When you set `StopbandFrequencySource` to `'Auto'`, the DDC object relaxes the stopband frequency as much as possible to obtain the lowest filter orders at the cost of allowing some aliasing energy in the transition band of the cascade response. This design tradeoff is convenient when your priority is to minimize filter orders.

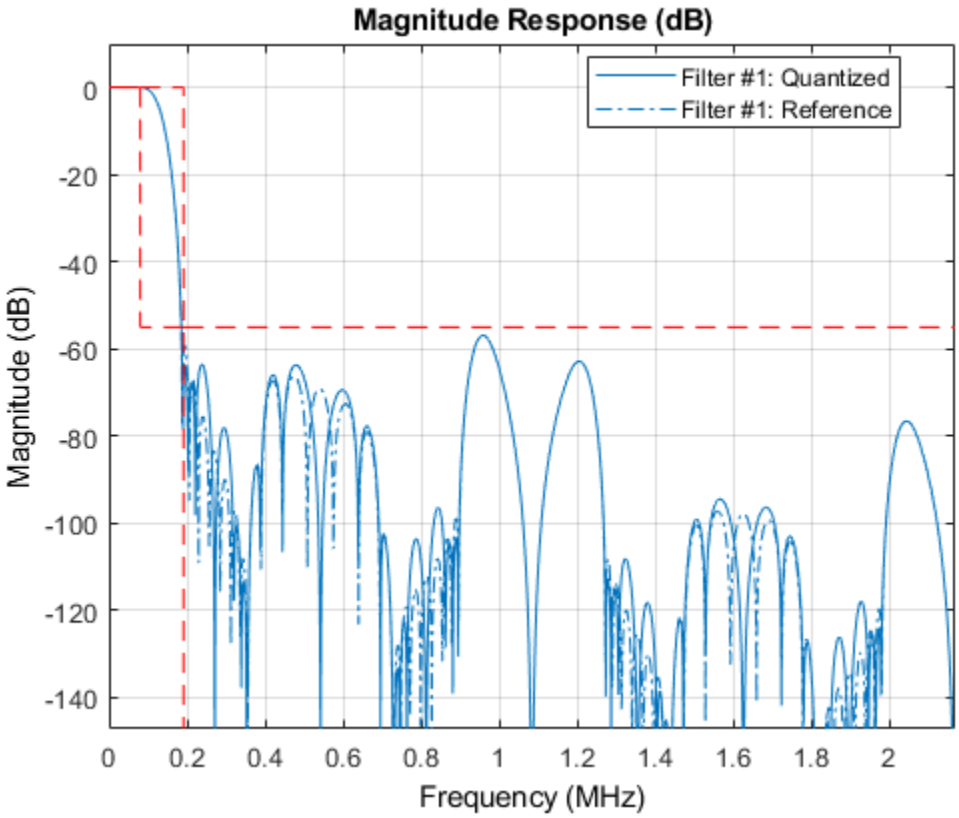
```
ddc.Bandwidth = 160e3; % Passband frequency equal to 80 KHz
ddc.StopbandFrequencySource = 'Auto'; % Allow aliasing in transition band
```

Finally, set a stopband attenuation of 55 dB and a passband ripple of 0.04 dB.

```
ddc.StopbandAttenuation = 55;
ddc.PassbandRipple = .04;
```

You can analyze the response of the cascade of decimation filters by calling the `fvtool` method of the DDC object. Specify a fixed-point arithmetic so that the DDC object quantizes the filter coefficients to an optimum number of bits that allow the cascade response to meet the stopband attenuation specifications.

```
fvt = fvtool(ddc,'Arithmetic','Fixed-point');
```



Get the designed filter orders and the coefficient word lengths for the CIC compensator and third stage FIR design.

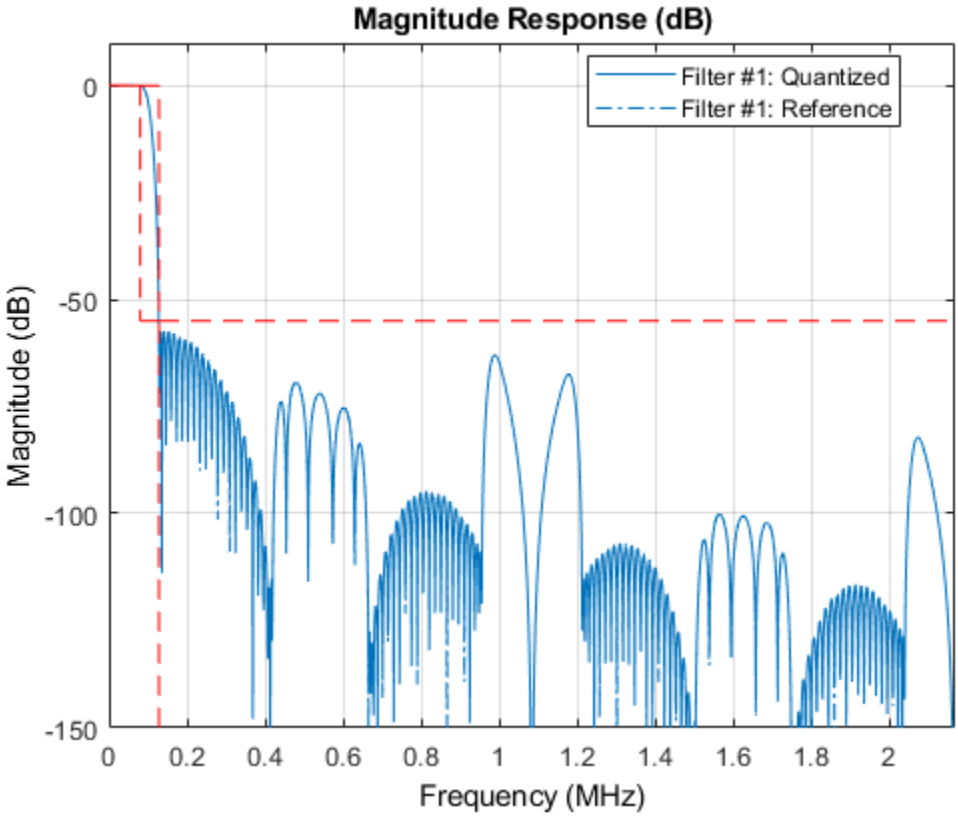
```
ddcFilters = getFilters(ddc,'Arithmetic','Fixed-point');
n = getFilterOrders(ddc);

CICCompensatorOrder = n.SecondFilterOrder %#ok
ThirdStageFIROrder = n.ThirdFilterOrder%#ok
CICCompensatorCoefficientsWordLength = ...
    ddcFilters.SecondFilterStage.CustomCoefficientsDataType.WordLength%#ok
ThirdStageFIRWordLength = ...
    ddcFilters.ThirdFilterStage.CustomCoefficientsDataType.WordLength%#ok
```

```
CICCompensatorOrder =

    12


ThirdStageFIROrder =

    18


CICCompensatorCoefficientsWordLength =

    11
```

```
ThirdStageFIRWordLength =

    11
```

If aliasing in the transition band is not acceptable, set the stopband frequency to an arbitrary value by setting the `StopbandFrequencySource` property to `'Property'`. Obtain a narrower transition band by setting the stopband frequency to 128 KHz at the expense of a larger third stage filter order.

```
ddc.StopbandFrequencySource = 'Property';
ddc.StopbandFrequency = 128e3;

close(fvt)
fvt = fvtool(ddc,'Arithmetic','fixed-point');
```

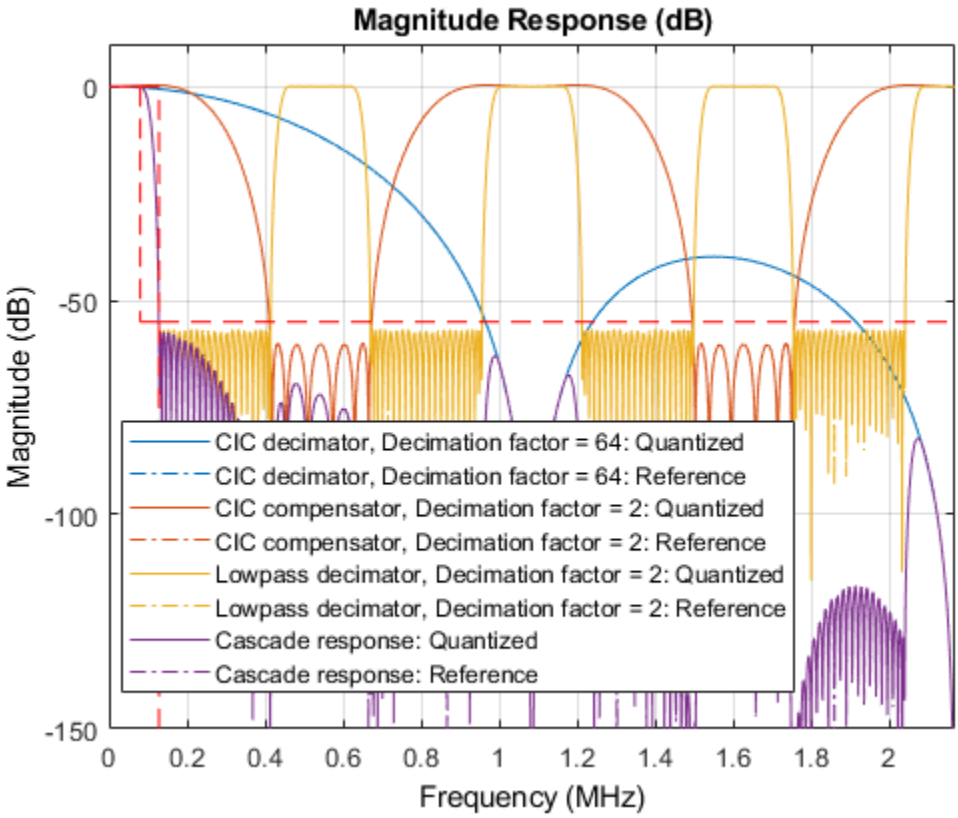



```
n= getFilterOrders(ddc);
CICCompensatorOrder = n.SecondFilterOrder%#ok
ThirdStageFIROrder = n.ThirdFilterOrder%#ok


CICCompensatorOrder =

    10


ThirdStageFIROrder =
```

```
34
```

Visualize the response of each individual filter stage and of the overall cascade using the visualizeFilterStages method of the DDC object.

```
close(fvt)
fvt = visualizeFilterStages(ddc,'Arithmetic','fixed-point');
```



Controlling the Filter Orders

There are cases when filter orders are the main design constraint. You use the DDC object to design decimation filters with a specified order by setting the `MinimumOrderDesign` property to false. You can still specify the required passband and stopband frequencies of the cascade response. Note however that the stopband attenuation and ripple are now controlled by the order of the filters and not by property values.

Oscillator Design

The DDC object designs a numerically controlled oscillator based on a small set of parameters. Set the `Oscillator` property to `'NCO'` to choose a numerically controlled oscillator. Use 32 accumulator bits, and 18 quantized accumulator bits. Set the center frequency to 14.44 MHz and choose 14 dither bits.

```
ddc.Oscillator = 'NCO';
ddc.CenterFrequency = 14.44e6;
ddc.NumAccumulatorBits = 32;
```

```
ddc.NumQuantizedAccumulatorBits = 18;
ddc.NumDitherBits = 14;
```

Fixed-Point Settings

You can set different properties on the DDC object to control the fixed-point data types along the down conversion path.

Cast the word and fraction lengths at the input of each filter to 20 and 19 bits respectively by setting the `CustomFiltersInputDataType` property to `numerictype([],20,19)`. Note that the DDC object scales the data at the output of the CIC decimator. The fact that this scaling is not done in the "GSM Digital Down Converter in MATLAB" on page 4-346 example explains the difference in the fraction length values chosen in each example.

Set the output data type to have a word length of 24 bits and a fraction length of 23 bits.

```
ddc.FiltersInputDataType = 'Custom';
ddc.CustomFiltersInputDataType = numerictype([],20,19);
ddc.OutputDataType = 'Custom';
ddc.CustomOutputDataType = numerictype([],24,23);
```

Processing Loop

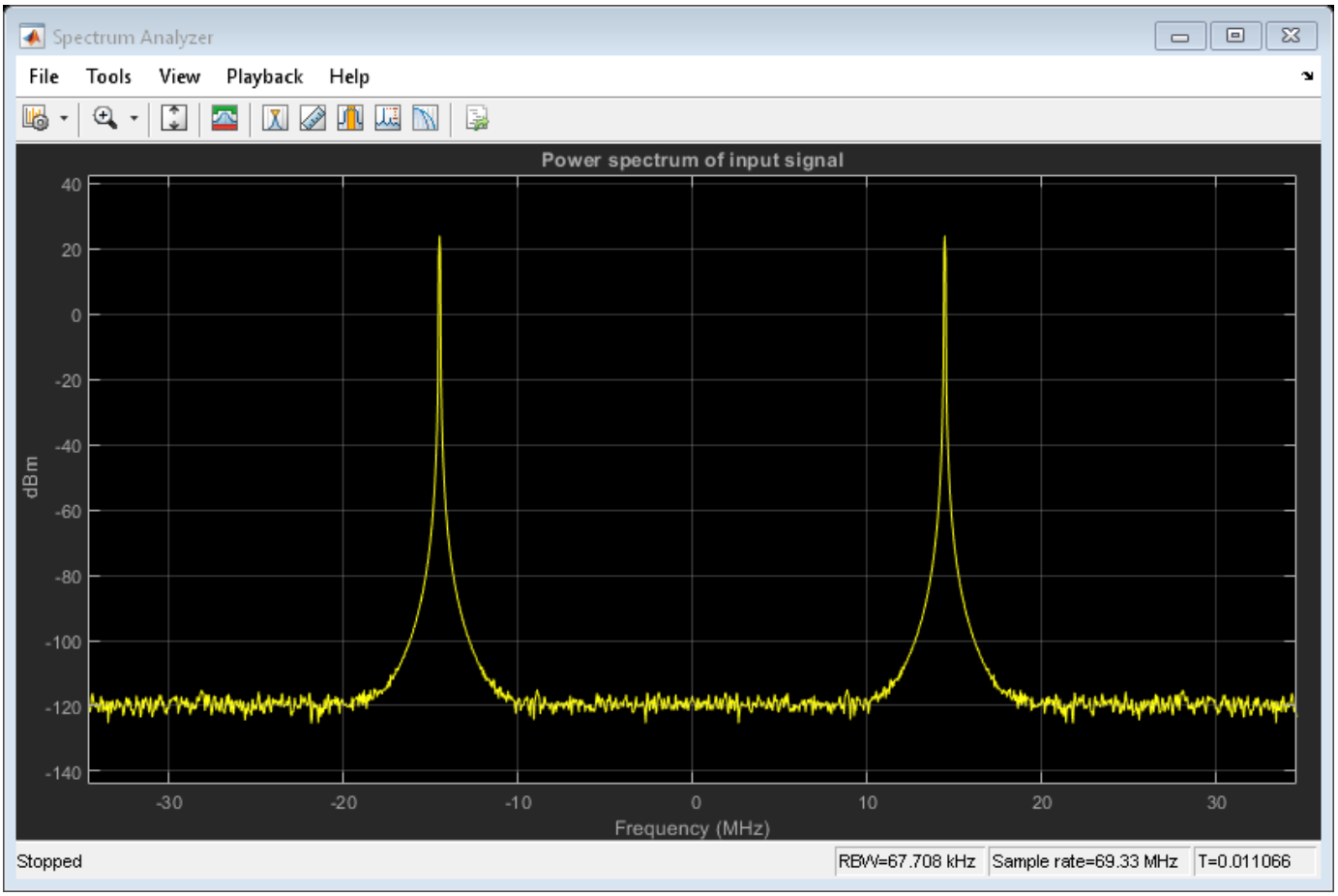
Initialize a sine wave generator to simulate a GSM source. Initialize a buffer to cast the input signal data type to 19 bits word length and 18 bits fraction length. Configure figures for plotting spectral estimates of signals.

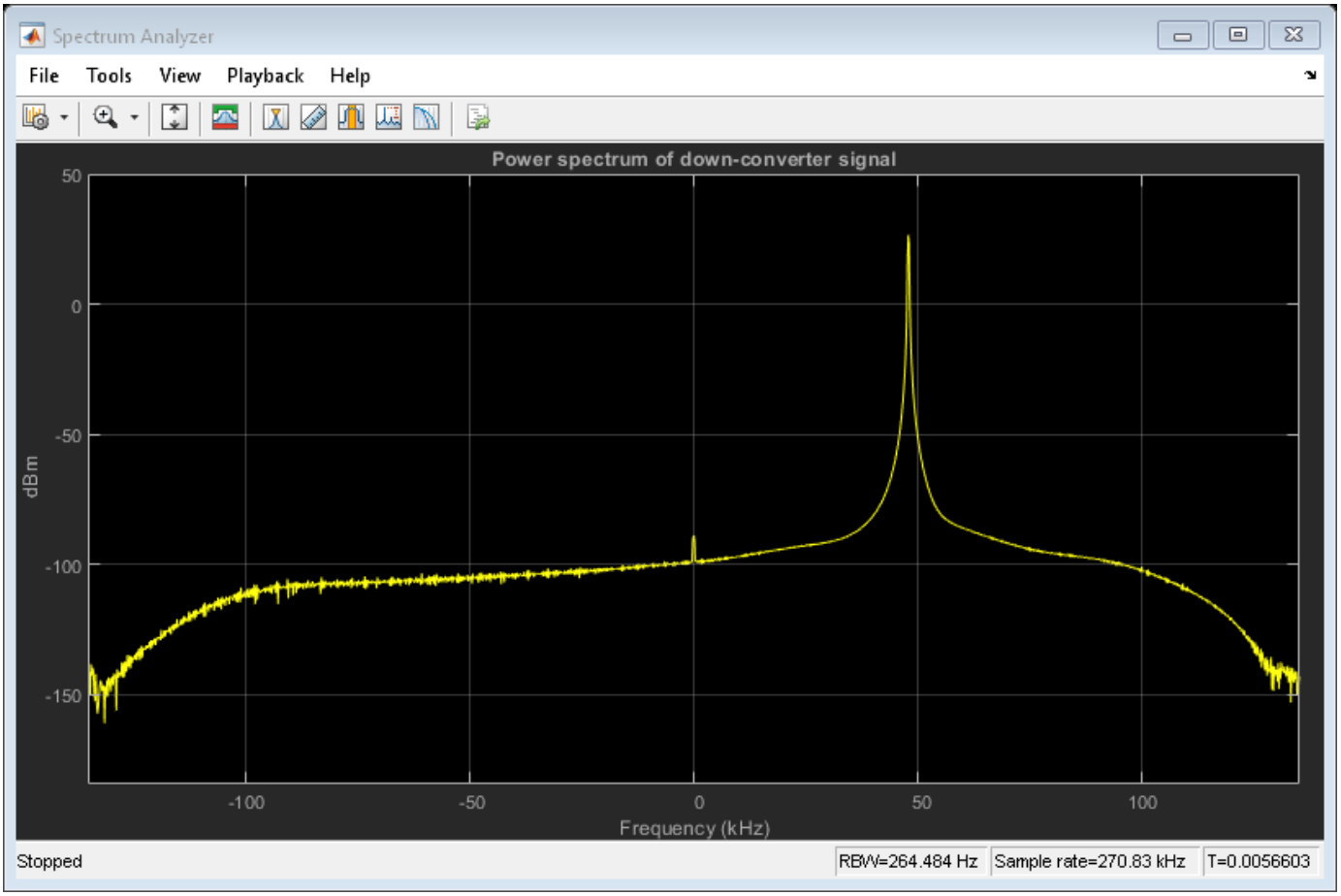
```
Fs = 69.333e6;
FrameSize = 768;
sine = dsp.SineWave('Frequency', 14.44e6+48e3, 'SampleRate', Fs, ...
    'PhaseOffset', 0, 'SamplesPerFrame', FrameSize);

gsmsig = fi(zeros(FrameSize,1),true,19,18);
inputSpectrum = dsp.SpectrumAnalyzer( ...
    'SampleRate',sine.SampleRate, ...
    'SpectralAverages',10, ...
    'Title','Power spectrum of input signal');
outputSpectrum = dsp.SpectrumAnalyzer( ...
    'SampleRate',sine.SampleRate/ddc.DecimationFactor, ...
    'SpectralAverages',10, ...
    'Title','Power spectrum of down-converter signal');
```

Main simulation loop

```
for ii = 1:1000
    % Create GSM signal with 19 bits of word length and 18 bits of fraction
    % length.
    gsmsig(:) = sine();

    % Down convert GSM signal
    downConvertedSig = ddc(gsmsig);

    % Frequency domain plots
    inputSpectrum(gsmsig);
    outputSpectrum(downConvertedSig);
end

% Release objects
```

```
release(sine);
release(ddc);
release(inputSpectrum);
release(outputSpectrum);
```





Notice the simplification of steps required to down convert a signal when compared to the “GSM Digital Down Converter in MATLAB” on page 4-346 example.

Conclusion

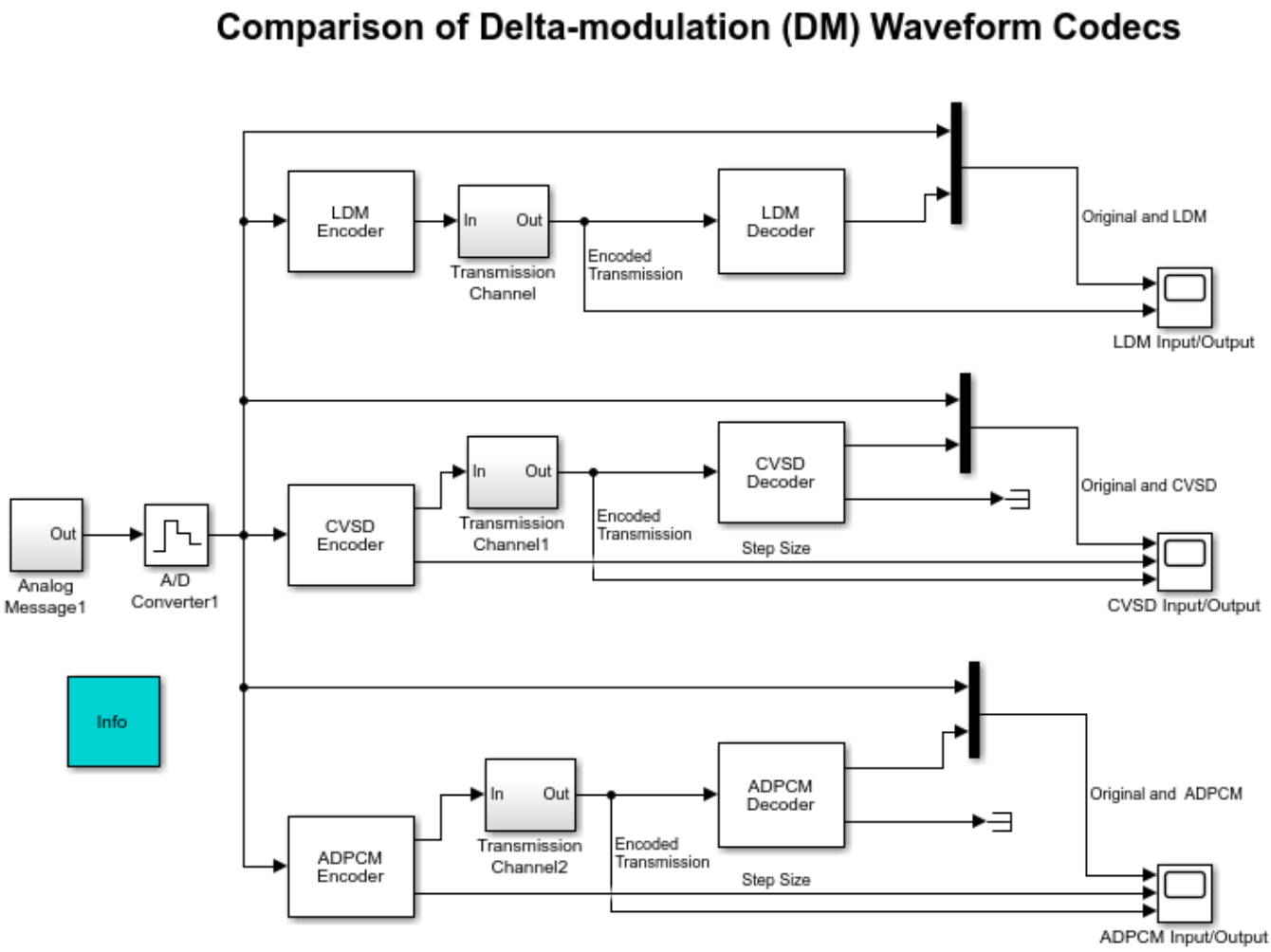
In this example, you compared the steps required to design a digital down converter as shown in the “GSM Digital Down Converter in MATLAB” on page 4-346 example with the steps required when using a DDC System object. The DDC object allows you to obtain down converter designs in one simple step. It provides tools to design decimation filters that meet passband frequency, passband ripple, stopband frequency, and stopband attenuation specifications. The DDC object also provides convenient tools to visualize and analyze the decimation filter responses.

Further Exploration

You can use the dsp.DigitalUpConverter System object to design a digital up converter system.

Comparison of LDM, CVSD, and ADPCM

This example shows how to compare three different delta-modulation (DM) waveform quantization, or coding, techniques.



What Are DM, LDM, CVSD, and ADPCM?

Delta-modulation (DM) is a differential waveform quantization or coding technique. A DM encoder uses the error between the original signal to be coded and the coded signal itself to create a differentially quantized data stream. This data stream, usually the computed error signal, is a lower-bit-rate signal that can be decoded by a matched decoder on the receiver side in order to achieve data compression, and therefore low data transmission rates.

Linear Delta-Modulation (LDM), Continuously Variable Slope Delta-Modulation (CVSD), and Adaptive Differential Pulse Code Modulation (ADPCM) are differential waveform coding techniques. Each employ two-level, or one-bit, encoders, and may be performed at many different sampling or data

rates. The encoded bit rate is usually directly proportional to the input signal sample rate. For example, in both LDM and CVSD, one bit per sample is used to compute the encoded data stream.

In LDM, a constant step-size is used to approximate the input signal with a single bit per signal sample. In the encoded bit stream, each 1 bit increases the amplitude by the step-size as compared to the previous decoded signal sample. Each 0 bit decreases the amplitude by the step-size. Using LDM, the encoder performance can suffer due to a condition known as "slope overload" when the input signal slope changes too rapidly for the encoder to track it accurately, for instance during high frequency content.

CVSD is LDM with the addition of an adaptive step-size. By adjusting or adapting the step-size to the changes in slope of the input signal, the encoder is able to represent low-frequency signals with greater accuracy without sacrificing as much performance due to slope overload at higher frequencies. When the slope of the input signal changes too quickly for the encoder to keep up with it, the step-size is increased. Conversely, when the input signal slope changes slowly, the step-size is decreased. A slope-overload detector and syllabic filter are used in conjunction with a pulse amplitude modulator (PAM) to accomplish the step-size adaptation.

CVSD is used in both commercial and military communications where "toll quality" or "communications quality" is required, yet low computation complexity and low memory requirements are desirable. Two examples of this technique are U.S. MIL-STD-188-113 (16 kbs and 32kbs CVSD) and U.S. Federal Standard 1023 (12 kbs CVSD). In addition, encoded CVSD data can be encrypted and made more secure, which is desirable for many wireless communications applications including speech and general-purpose audio coding.

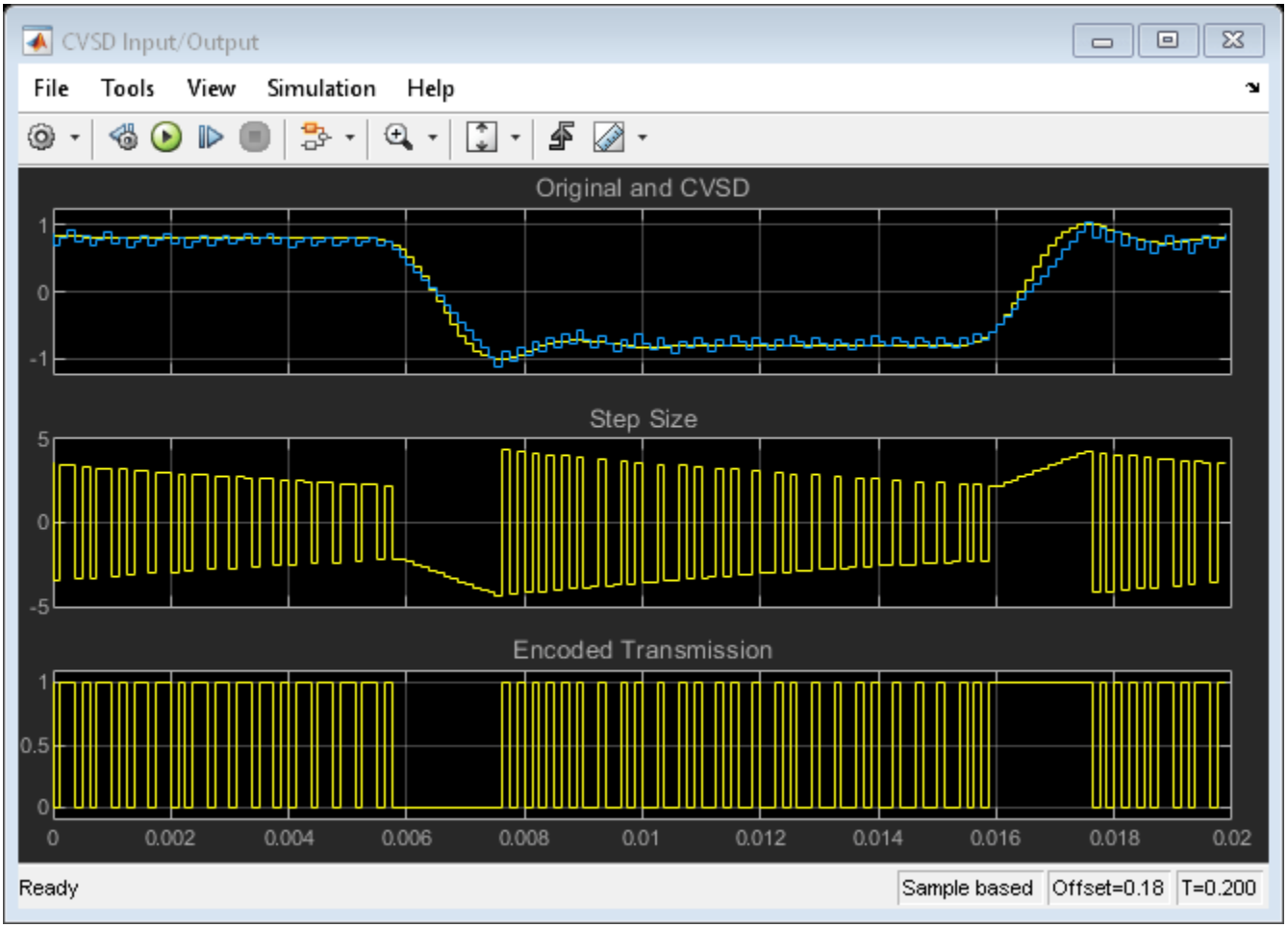
ADPCM is similar to CVSD, however it provides more accuracy and therefore preserved frequency bandwidth at the expense of additional computational requirements for the adaptation step-size calculations.

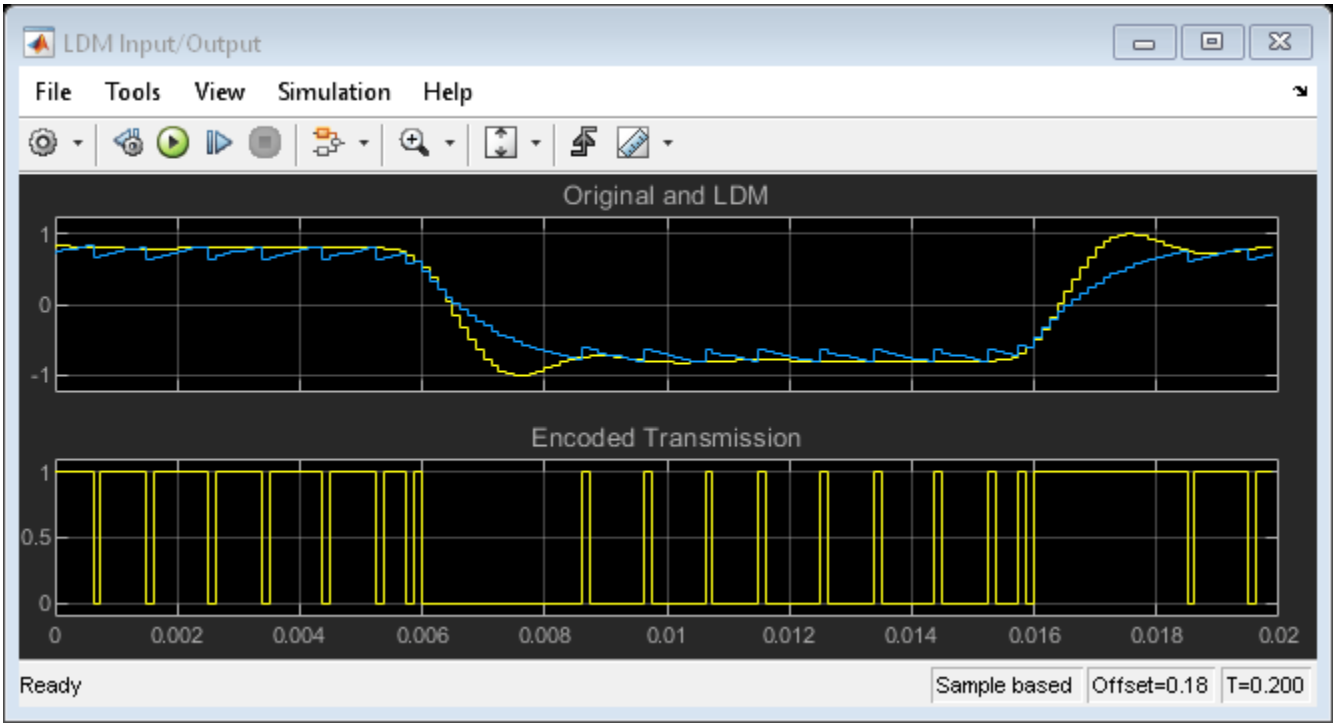
Comparison of Techniques

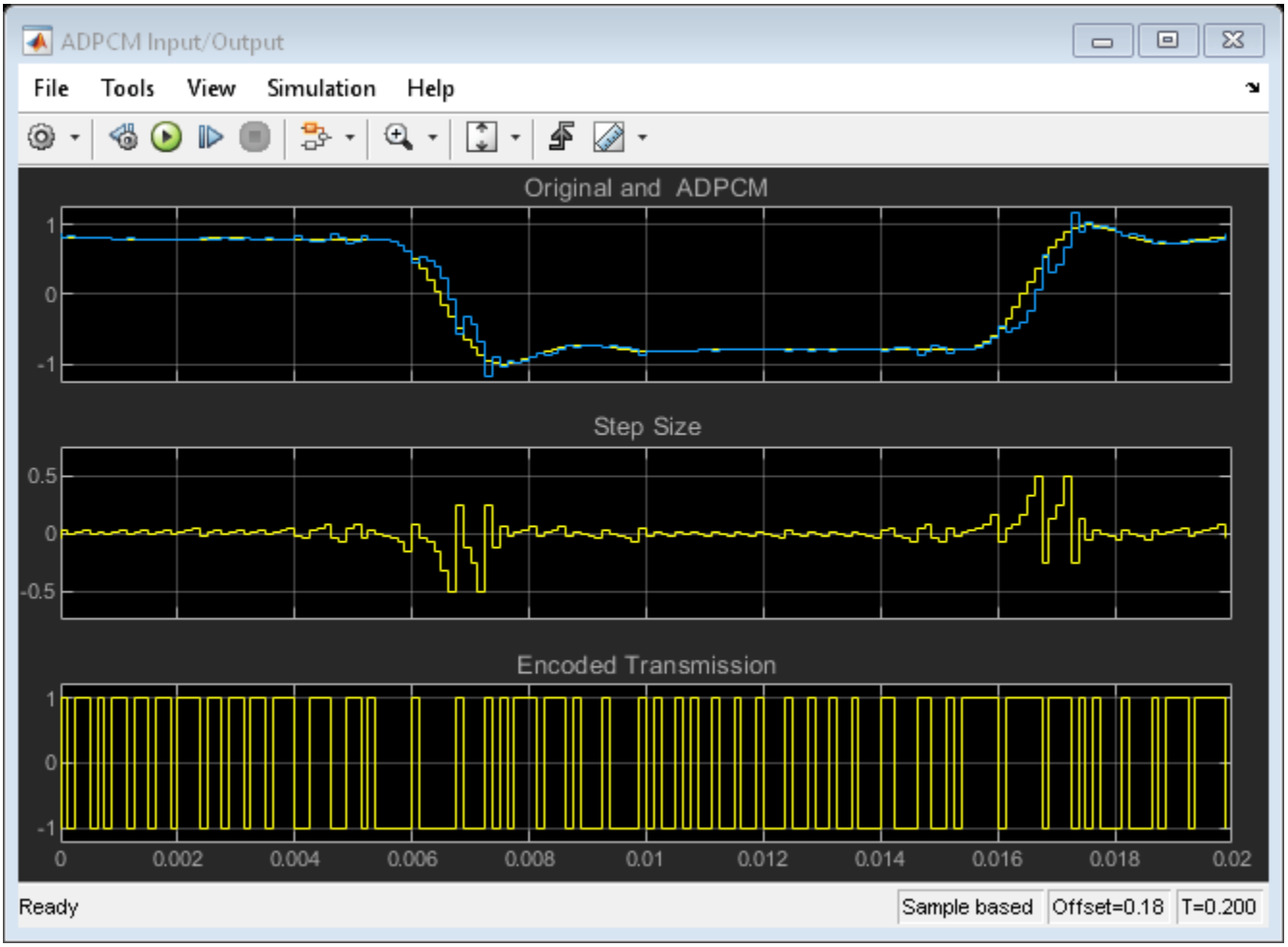
The model's scopes show the behavior of the three coding techniques. The first display in each scope shows the original signal in yellow and the recovered signal in magenta that has been encoded and then decoded. You can observe the response of each technique to both constant and rapidly changing signal regions.

Because LCM uses a constant step size, it exhibits slope overload while the signal is changing rapidly and granular noise while the signal is constant. Both CVSD and ADPCM mitigate these problems using a variable step size.

For both CVSD and ADPCM the variable step size is shown in the center display. For all three techniques the one-bit encoded transmission signal is shown in the lower display.


CVSD Input/Output
File
Tools
View
Simulation
Help
Original and CVSD
1
0
-1
Step Size
5
0
-5
Encoded Transmission
1
0.5
0
0
0.002
0.004
0.006
0.008
0.01
0.012
0.014
0.016
0.018
0.02
Ready
Sample based
Offset=0.18
T=0.200


LDM Input/Output
File
Tools
View
Simulation
Help
Original and LDM
1
0
-1
Encoded Transmission
1
0.5
0
0
0.002
0.004
0.006
0.008
0.01
0.012
0.014
0.016
0.018
0.02
Ready
Sample based
Offset=0.18
T=0.200



References

Proakis, J. G. **Digital Communications**. Third Ed. Sec. 3 ("Source Coding"). McGraw Hill. 1995.

Taylor, D. S. "Design of Continuously Variable Slope Delta Modulation Communication Systems." **Application Note AN1544/D**. Motorola®, Inc. 1996.

"Continuously Variable Slope Delta Modulation: A Tutorial." **Application Doc. #20830070.001**. MX-COM, Inc., Winston-Salem, North Carolina, 1997.

Conahan, S., Alva, C., and Norris, J. "Implementation of a Real-time 16 kbps CVSD Digital Voice Codec on a Fixed-point DSP." **International Conference on Signal Processing Applications and Technology (ICSPAT) Proceedings**, Vol. 1. 1998. pp. 282-286.

Digital Up and Down Conversion for Family Radio Service

This example shows how to use the digital up converter (DUC) and digital down converter (DDC) System objects to design a Family Radio Service (FRS) transmitter and receiver. These objects provide tools to design interpolation/decimation filters and simplify the steps required to implement the up/down conversion process. This example illustrates both MATLAB® and Simulink® implementations. The MATLAB version uses System objects for DUC and DDC, whereas the Simulink version uses blocks for DUC and DDC. In both versions, speech signal is used as an input, and the signal after transmission is played back.

FRS is an improved walkie talkie FM radio system authorized in the United States since 1996. This personal radio service uses channelized frequencies in the ultra high frequency (UHF) band. Devices operating in the FRS band must be authorized under Part 95 Subpart B "Family Radio Service" (Sections 95.191 through 95.194) of the FCC rules. The authorized bandwidth of FRS channels is 12.5 KHz and the center frequency separation between channels is 25 KHz.

Introduction

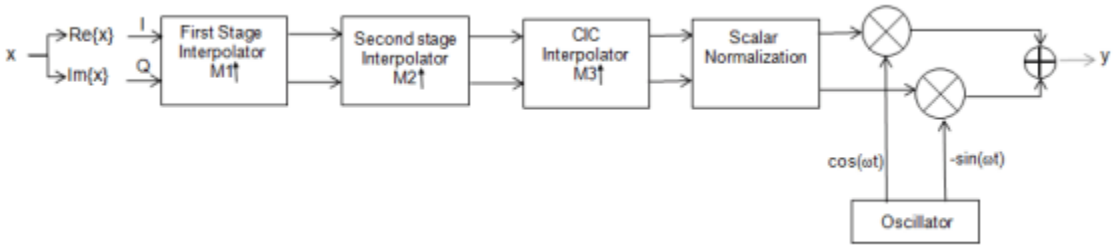
This example discusses the digital up conversion of a signal to be transmitted through an FRS channel, and the down conversion of the signal coming from the FRS radio transmitter.

The 8 KHz speech is first resampled to 50KHz. The DUC at the transmitter up converts the signal from 50 KHz to 2 MHz and shifts the signal to an IF frequency of 455 KHz.

The receiver has an analog front end that brings the received signal to an IF frequency of 455 KHz. The signal is then sampled at a rate of 2 MHz. The DDC at the receiver brings the signal back to baseband with a sample rate of 50 KHz. This is brought back to 8 KHz speech range.

Digital Up Converter Design

You design a digital up converter by creating a DUC System object. The DUC object consists of a cascade of three interpolation filters and an oscillator that up converts the interpolated signal to a specified passband frequency. A block diagram of the DUC object is shown next.



The DUC object provides options to define the interpolation filters. For instance, you can design the oscillator using either a sine wave generator or a numerically controlled oscillator. The following section showcases the different options available to design the interpolation filters for the FRS transmitter.

Designing Interpolation Filters

The DUC object implements the interpolation filter using three filter stages. When the DUC object designs the filters internally, the first stage consists of a halfband, or a lowpass filter, the second

stage consists of a CIC compensator, and the third stage consists of a CIC interpolation filter. The DUC object allows you to specify several characteristics that define the response of the cascade for the three filters, including passband and stopband frequencies, passband ripple, and stopband attenuation.

Minimum Order Filter Designs

By default (when the `MinimumOrderDesign` property is set to true) the DUC object obtains minimum order interpolation filter designs using the passband and stopband specifications you provide.

For this FRS example, you must up sample the transmitted signal from 50 KHz to 2 MHz. This yields an interpolation factor of 40. The DUC object automatically factors the interpolation value so that the first filter stage interpolates by 2, the second filter stage interpolates by 2, and the CIC filter interpolates by 10.

The FRS channel double sided bandwidth is 12.5 KHz. Set the `Bandwidth` property of the DUC object to 12.5 KHz so that the passband frequency of the cascade response of the interpolation filters is 12.5e3/2 = 6.25 KHz.

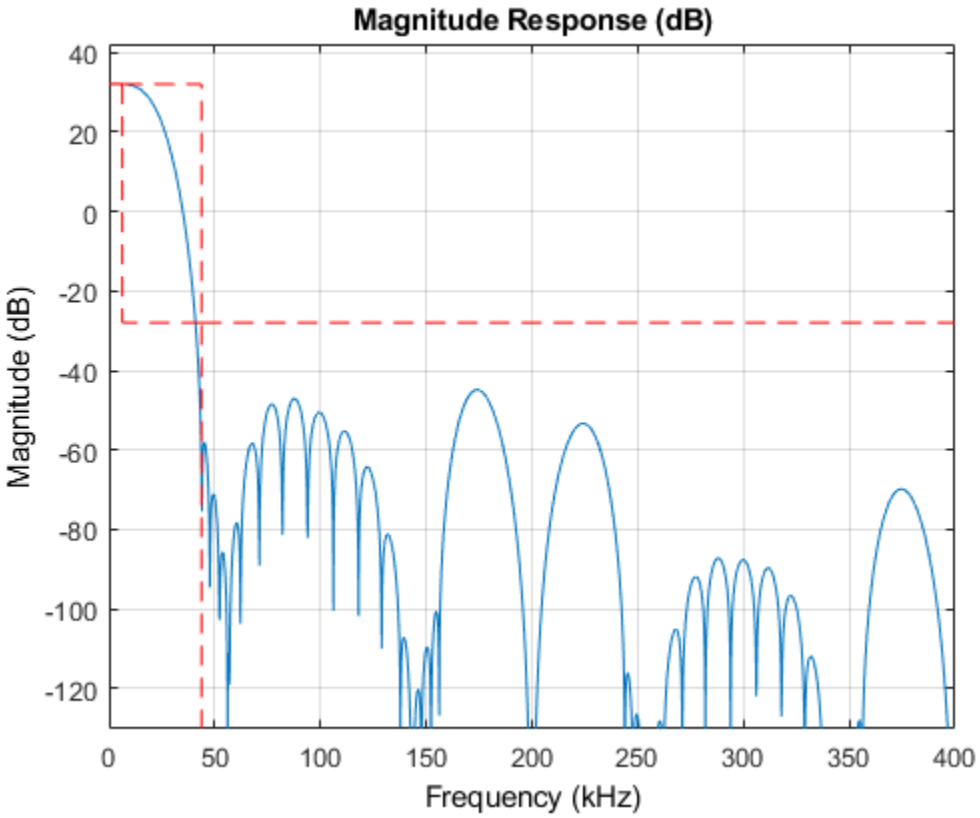
Set the passband ripple to a small value of 0.05 dB to avoid distortion of the FRS signal. Set the stopband attenuation to 60 dB.

By default (when the `StopbandFrequencySource` property is set to 'Auto') the DUC object sets the cutoff frequency of the cascade response approximately at the input Nyquist rate of 25 KHz. The object also sets the stopband frequency at 2Fc-Fpass = 2*25e3 - 12.5e3/2 = 43.75 KHz, where Fc is the cutoff frequency and Fpass is the passband frequency. In this scenario, the DUC object relaxes the stopband frequency as much as possible, obtaining the lowest filter orders at the cost of allowing some energy of interpolation replicas in the transition band of the cascade response. This design tradeoff is convenient when your priority is to minimize filter orders.

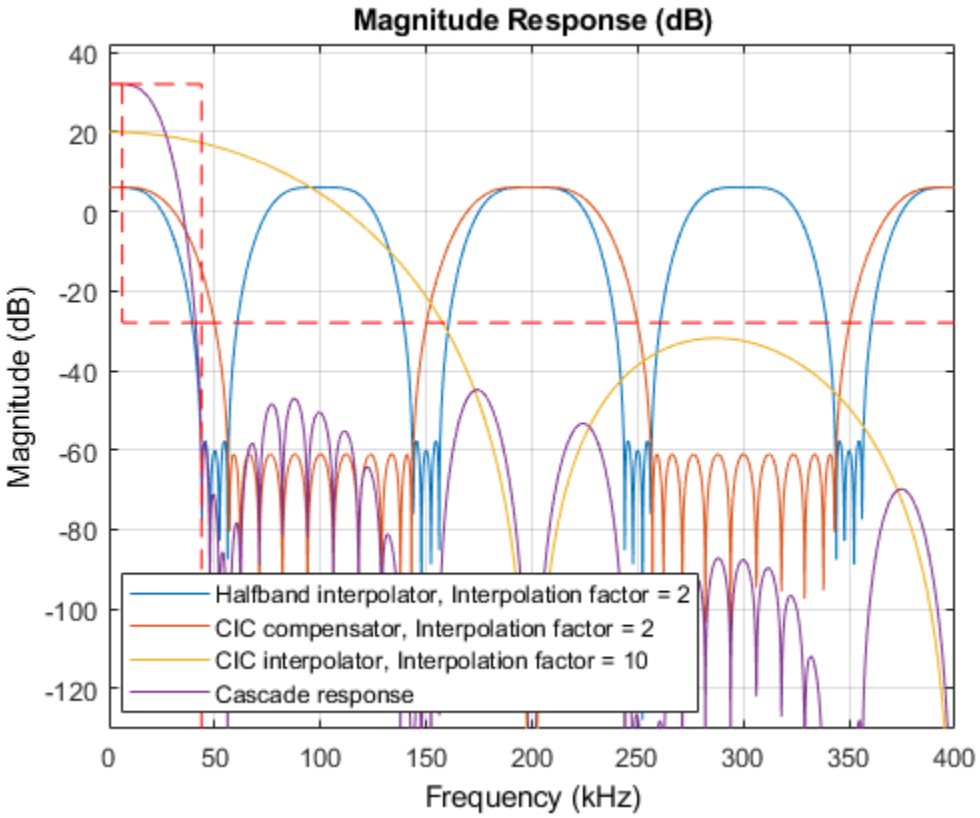
```
DUC = dsp.DigitalUpConverter(...
  'SampleRate',50e3,...
  'InterpolationFactor',40,...
  'Bandwidth',12.5e3,...
  'PassbandRipple',0.05,...
  'StopbandAttenuation',60);
```

Visualize the cascade response of the decimation filters using the `fvtool` or `visualizeFilterStages` methods of the DUC object. Specify the arithmetic as `'double'` so that the filter coefficients and operations are double-precision.

```
fvt = fvtool(DUC,'Arithmetic','double');
```




```
close(fvt)
fvt = visualizeFilterStages(DUC,'Arithmetic','double');
```



Get the orders of the designed filters using the `getFilterOrders` method.

```
s = getFilterOrders(DUC);
s.FirstFilterOrder
```

```
ans =

    10
```

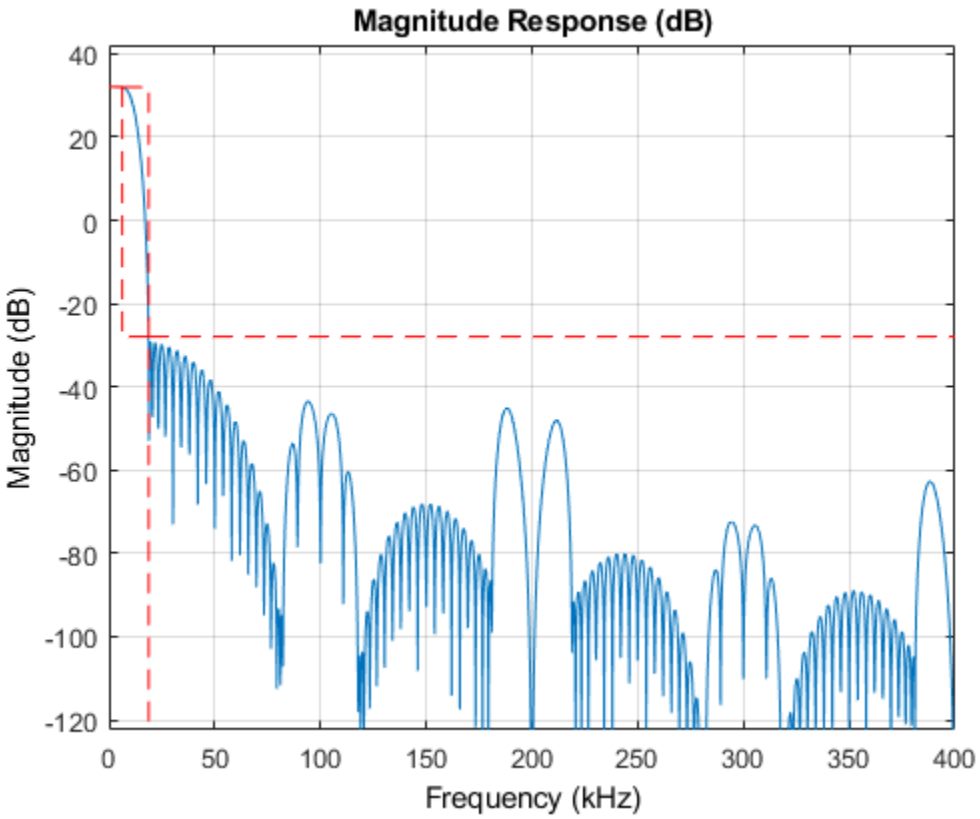
```
s.SecondFilterOrder
```

```
ans =

    12
```

The FRS channel separation is 25 KHz. Most commercial FRS radios offer 50 dB or higher adjacent channel rejection (ACR). Clearly, the cascade response of the decimation filters designed above does not achieve a 50 dB attenuation at 25 KHz. One possible solution to this problem is to filter the baseband FRS signal with a lowpass filter with the required transition width and stopband attenuation before passing the signal through the DUC object. Another solution is to set the DUC object so that it designs a cascade response with a narrower transition bandwidth that meets the required spectral mask. To design an overall filter response with a narrower transition bandwidth, set the `StopbandFrequencySource` property to `'Property'` and the `StopbandFrequency` property to a desired value.

Design the filters so that the cascade response has a stopband frequency at the edge of the passband of the adjacent FRS channel, i.e. at 25e3-12.5e3/2 = 18.75 KHz. Set the `StopbandAttenuation` property to 60 dB to achieve a 60 dB ACR.

```
DUC.StopbandFrequencySource = 'Property';
DUC.StopbandFrequency = 18.75e3;
DUC.StopbandAttenuation = 60;
close(fvt)
fvt = fvtool(DUC,'Arithmetic','double');
```



Get the filter orders

```
s = getFilterOrders(DUC);
s.FirstFilterOrder
```

```
ans =

    23
```

```
s.SecondFilterOrder
```

```
ans =

     7
```

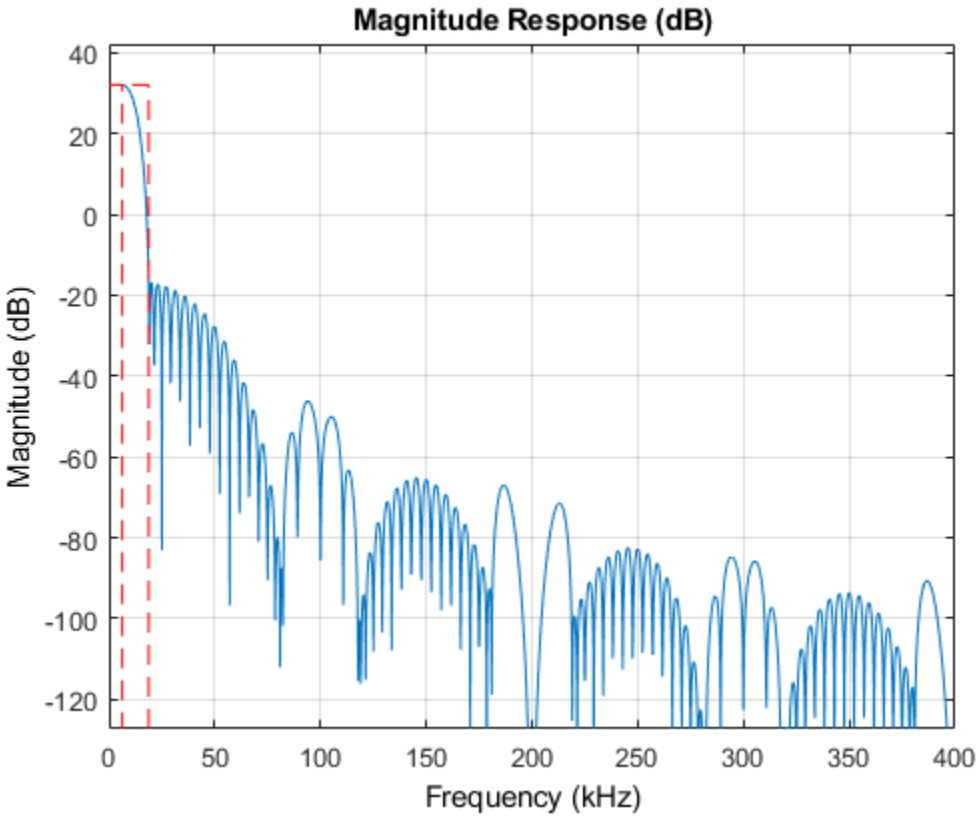
The new cascade response achieves 60 dB attenuation at 25 KHz, i.e., at the center of the adjacent FRS channel. The order of the first stage filter (lowpass interpolator) increases from 10 to 23. Note however that the order of the second stage filter (CIC compensator) decreases from 12 to 7. Because the first stage response has a narrower bandwidth, the second stage stopband can be relaxed even more to the edge of the left stopband of the first replica of the first stage filter. Since the second filter stage operates at a higher rate, this is a very convenient order reduction.

Controlling the Filter Orders

There are cases when filter orders are the main design constraint. Set the `MinimumOrderDesign` property to false to design interpolation filters with a specific order. In this configuration, you can still specify the required passband and stopband frequencies. The orders of the filters control the stopband attenuation and ripple of the cascade response.

To meet a constraint of a maximum of 20 coefficients in the first filter stage, set the `FirstFilterOrder` property to 20. Set the `SecondFilterOrder` property to 7, and the number of CIC sections to 4.

```
% Keep a copy of the minimum order design so that we can use it later on
% this example.
DUCMinOrder = clone(DUC);

% Specify the filter orders and visualize the cascade response.
DUC.MinimumOrderDesign = false;
DUC.FirstFilterOrder = 20;
DUC.SecondFilterOrder = 7;
DUC.NumCICSections = 4;

close(fvt)
fvt = fvtool(DUC,'Arithmetic','double');
```



The new design has lower stopband attenuation and larger passband ripple due to the reduced first filter order.

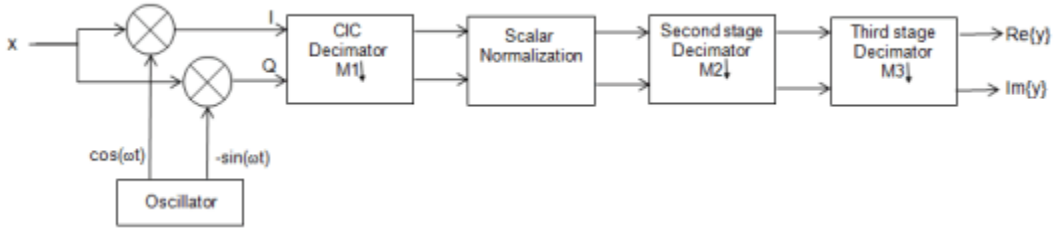
Oscillator Design

Use the `Oscillator` property to select the type of oscillator the object uses to perform the frequency up conversion. Set the property to `'Sine wave'` to obtain an oscillator signal from a sinusoidal computed using samples of the trigonometric function. Alternatively, set the property to `'NCO'` so the object designs a numerically controlled oscillator. Set the center frequency of the oscillator to the IF frequency of 455 KHz.

```
DUC.Oscillator = 'Sine wave';
DUC.CenterFrequency = 455e3;
```

Digital Down Converter Design

You design a digital down converter (DDC) by creating a DDC System object. The DDC System object consists of an oscillator that down converts an input signal from a specific passband frequency to 0 Hz. The object down samples the down converted signal using a cascade of three decimation filters. The following block diagram shows the DDC object.



As in the DUC case, the DDC object offers different options for defining decimation filters. For instance, you can design the oscillator using either a sine wave generator or a numerically controlled oscillator. Alternatively, you can provide an oscillator signal as an input.

Designing Decimation Filters

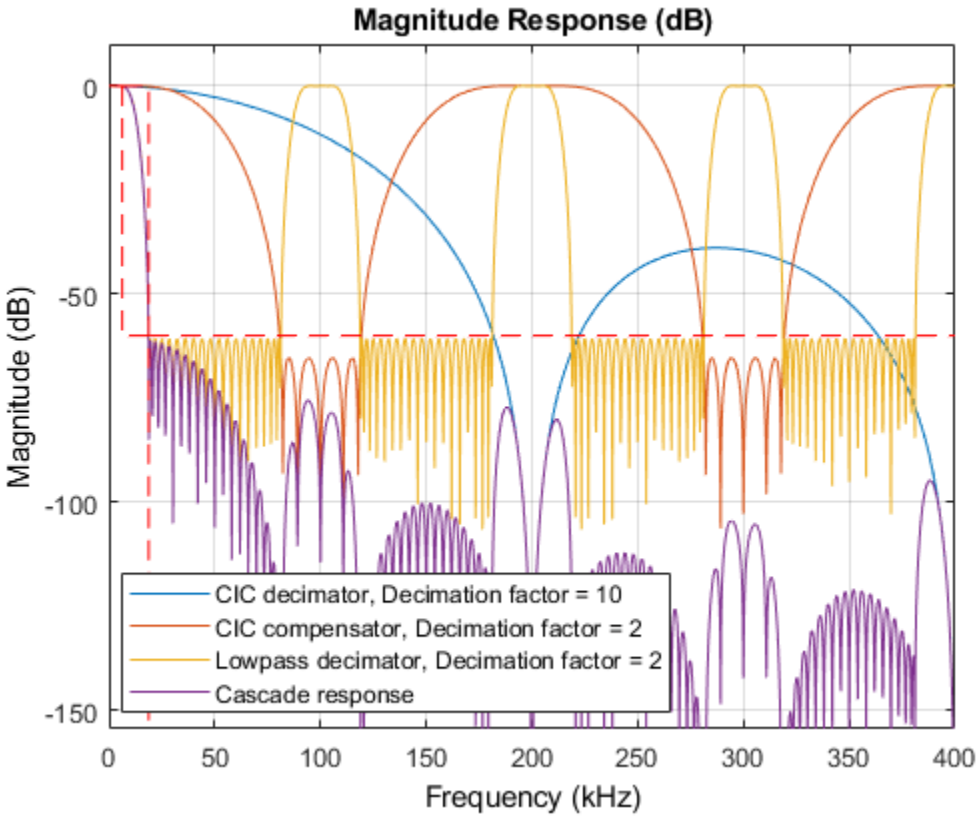
The DDC object implements decimation using three filter stages. When the object designs the filters internally, the first stage consists of a CIC decimator, the second stage consists of a CIC compensator, and the third stage consists of a halfband, or a lowpass, decimation filter. As in the DUC case, the DDC object allows you to specify characteristics that define the response of the cascade of the three filters, including passband and stopband frequencies, passband ripple, and stopband attenuation.

Design minimum order decimation filters to receive an FRS signal centered at an IF frequency of 455 KHz. Decimate the signal by 40 to downsample it from 2 MHz to 50 KHz. Set the `StopbandFrequencySource` property to `'Property'` and the stopband attenuation to 60 dB to design a cascade response that achieves an ACR of 60 dB.

```
DDCMinOrder = dsp.DigitalDownConverter(...
  'SampleRate',2e6,...
  'DecimationFactor',40,...
  'Bandwidth',12.5e3,...
  'PassbandRipple',0.05,...
  'StopbandAttenuation',60,...
  'StopbandFrequencySource', 'Property',...
  'StopbandFrequency',18.75e3,...
  'CenterFrequency',455e3);
```

Analyze the responses of the decimator filters and verify that the cascade response achieves an attenuation of 60 dB at 25 KHz. Note how the DDC relaxes the response of the second stage (CIC compensator) to the edge of the left stopband of the first alias of the third stage (lowpass decimator) to minimize order.

```
close(fvt)
fvt = visualizeFilterStages(DDCMinOrder,'Arithmetic','double');
```



Similar to the DUC case, you can define the filter orders by setting the `MinimumOrderDesign` property to false.

MATLAB Processing Loop

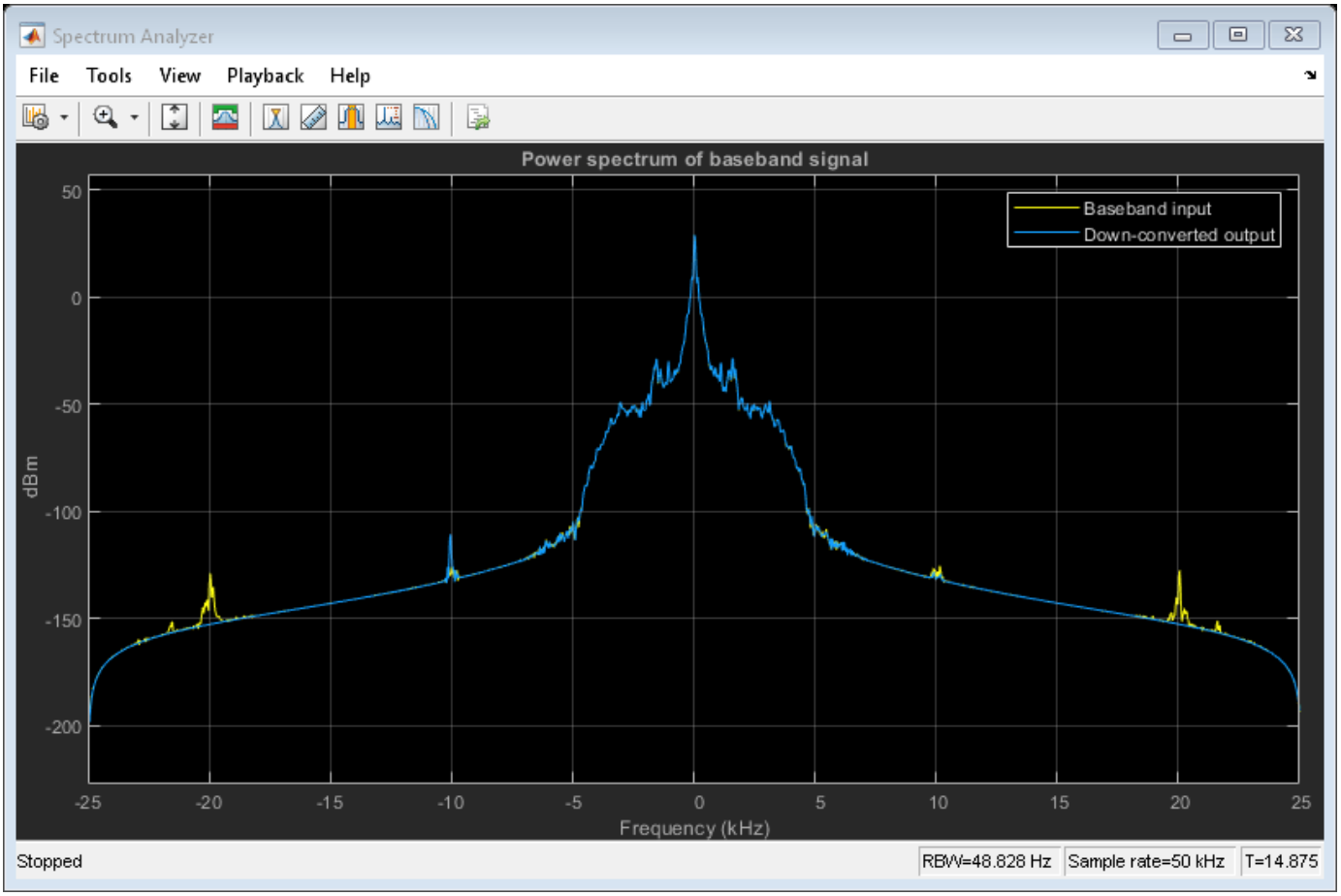
The FCC Part 95 specifies an FM modulation with maximum frequency deviation of 2.5 KHz and a maximum audio frequency of 3.125 KHz. Frequency-modulate the audio signal to obtain the FRS baseband signal (the signal does not include a squelch tone). Up convert and down convert the baseband FRS signal using the DUC and DDC objects that were designed in the previous sections using minimum order filters. Demodulate the signal and play it using audio player.

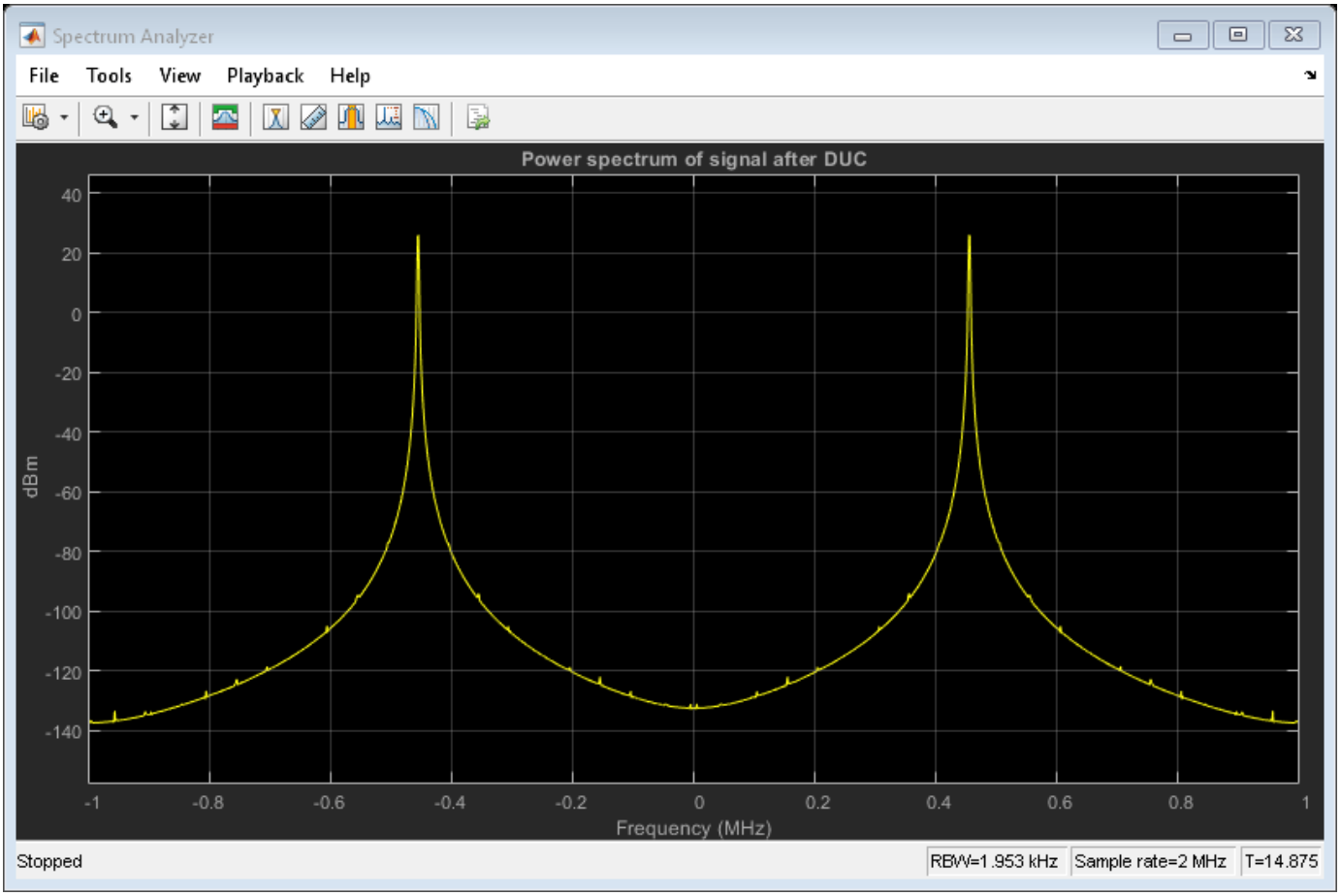
```
% Initialize simulation parameters
close(fvt)
Fs = 50e3;
frameLength = 1000;
maxAudioFrequency = 3.125e3; % Maximum allowed audio frequency for FRS radios
deltaF = 2.5e3; % Maximum allowed frequency deviation for FRS radios
freqSensitivityGain = deltaF*2*pi/Fs;    % K=FD/A*(2*pi*Ts)

ModulationFilter = dsp.IIRFilter('Numerator',1,'Denominator',[1, -1]);
DemodulationDelay = dsp.Delay(1);

audioReader = dsp.AudioFileReader('speech_dft_8kHz.wav', ...
    'PlayCount', 3, 'SamplesPerFrame', frameLength);

SRCTx = dsp.SampleRateConverter('InputSampleRate', audioReader.SampleRate, ...
    'OutputSampleRate', Fs, 'Bandwidth', 6.25e3);
```

```
SRCRx = dsp.SampleRateConverter('InputSampleRate', Fs, ...
    'OutputSampleRate', audioReader.SampleRate, 'Bandwidth', 6.25e3);

audioWriter = audioDeviceWriter('SampleRate', 8e3);

DUCMinOrder.CenterFrequency = 455e3;

basebandSignalSpectrum = dsp.SpectrumAnalyzer(...
        'SampleRate', Fs, 'ShowLegend', true, ...
        'ChannelNames', {'Baseband input', 'Down-converted output'}, ...
        'SpectralAverages', 10, 'Title', 'Power spectrum of baseband signal');

upConvertedSignalSpectrum = dsp.SpectrumAnalyzer(...
        'SampleRate', Fs*DUCMinOrder.InterpolationFactor, ...
        'SpectralAverages', 10, 'Title', 'Power spectrum of signal after DUC');
```

Stream data

```
while ~isDone(audioReader)
  % Input speech signal
  audioIn = audioReader();

  % Resample
  audioIn_200kHz = SRCTx(audioIn);

  % FM Modulation
  xFMBaseband = exp(1j * freqSensitivityGain * ModulationFilter(audioIn_200kHz));

  % Up conversion
  xUp = DUCMinOrder(xFMBaseband);
  upConvertedSignalSpectrum(xUp);

  % Down conversion
  xDown = DDCMinOrder(xUp);
  basebandSignalSpectrum([xFMBaseband, xDown]);

  % FM Demodulation
  audioOut_200kHz = angle(DemodulationDelay(xDown) .* conj(xDown));

  % Resample
  audioOut = SRCRx(audioOut_200kHz);

  % Play audio
  audioWriter(audioOut);
end

% Cleanup
release(audioReader);
release(SRCTx);
release(ModulationFilter);
release(DUCMinOrder);
release(DDCMinOrder);
release(DemodulationDelay);
release(SRCRx);
release(basebandSignalSpectrum);
release(upConvertedSignalSpectrum);
```

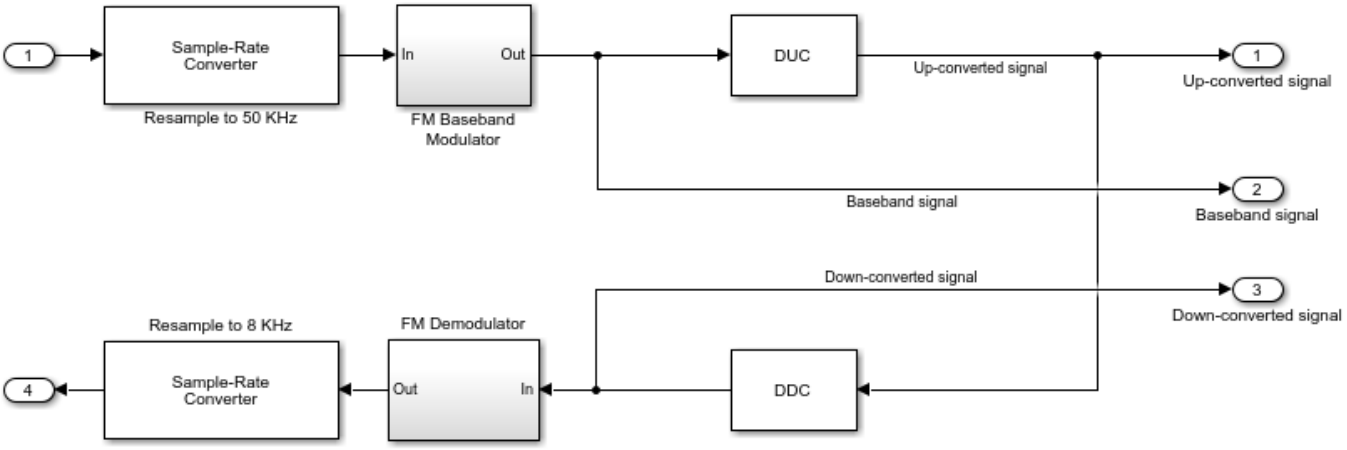

Spectrum Analyzer
File
Tools
View
Playback
Help
Power spectrum of baseband signal
Baseband input
Down-converted output
50
0
-50
-100
-150
-200
dBm
-25
-20
-15
-10
-5
0
5
10
15
20
25
Frequency (kHz)
Stopped
RBW=48.828 Hz
Sample rate=50 kHz
T=14.875

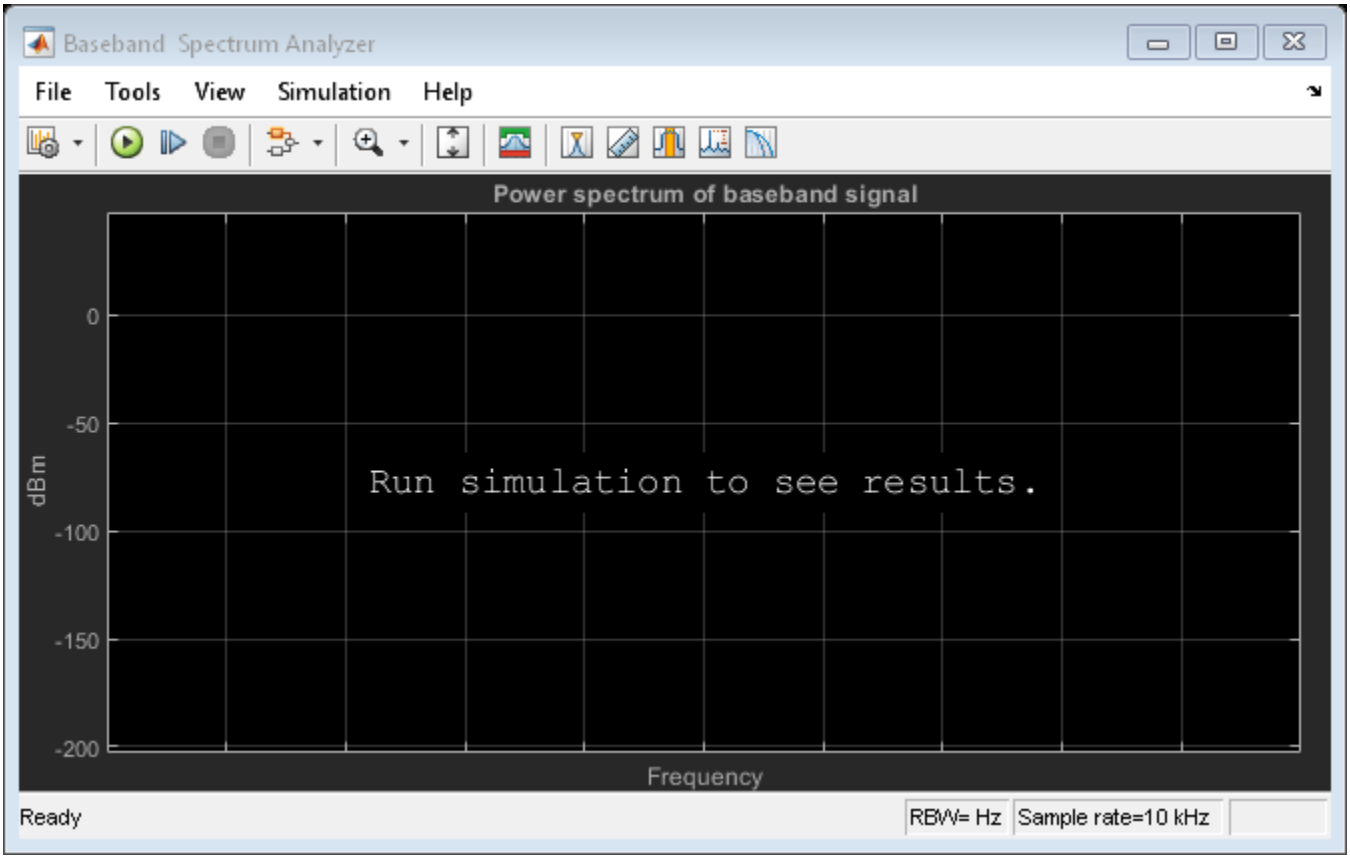


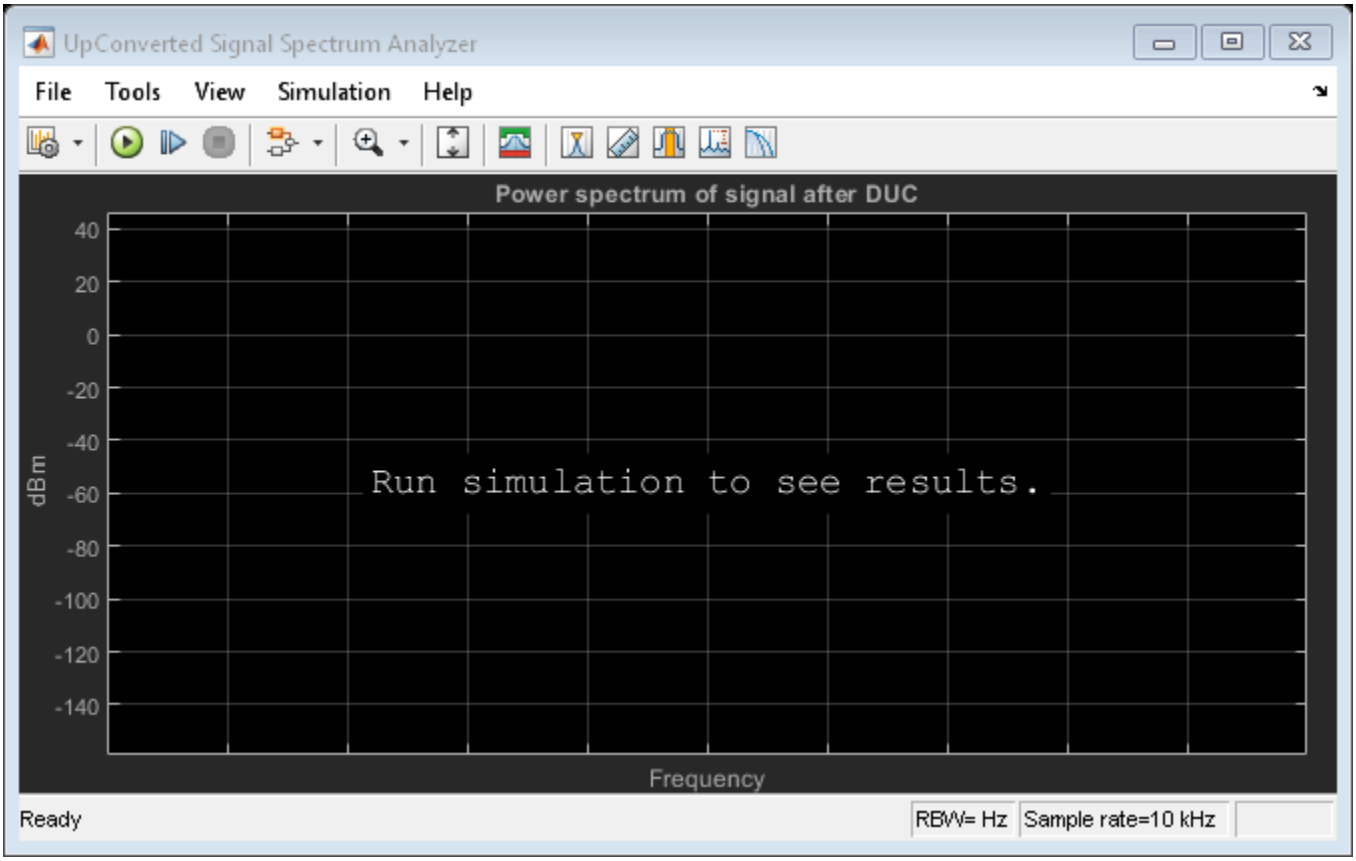
Simulink Version

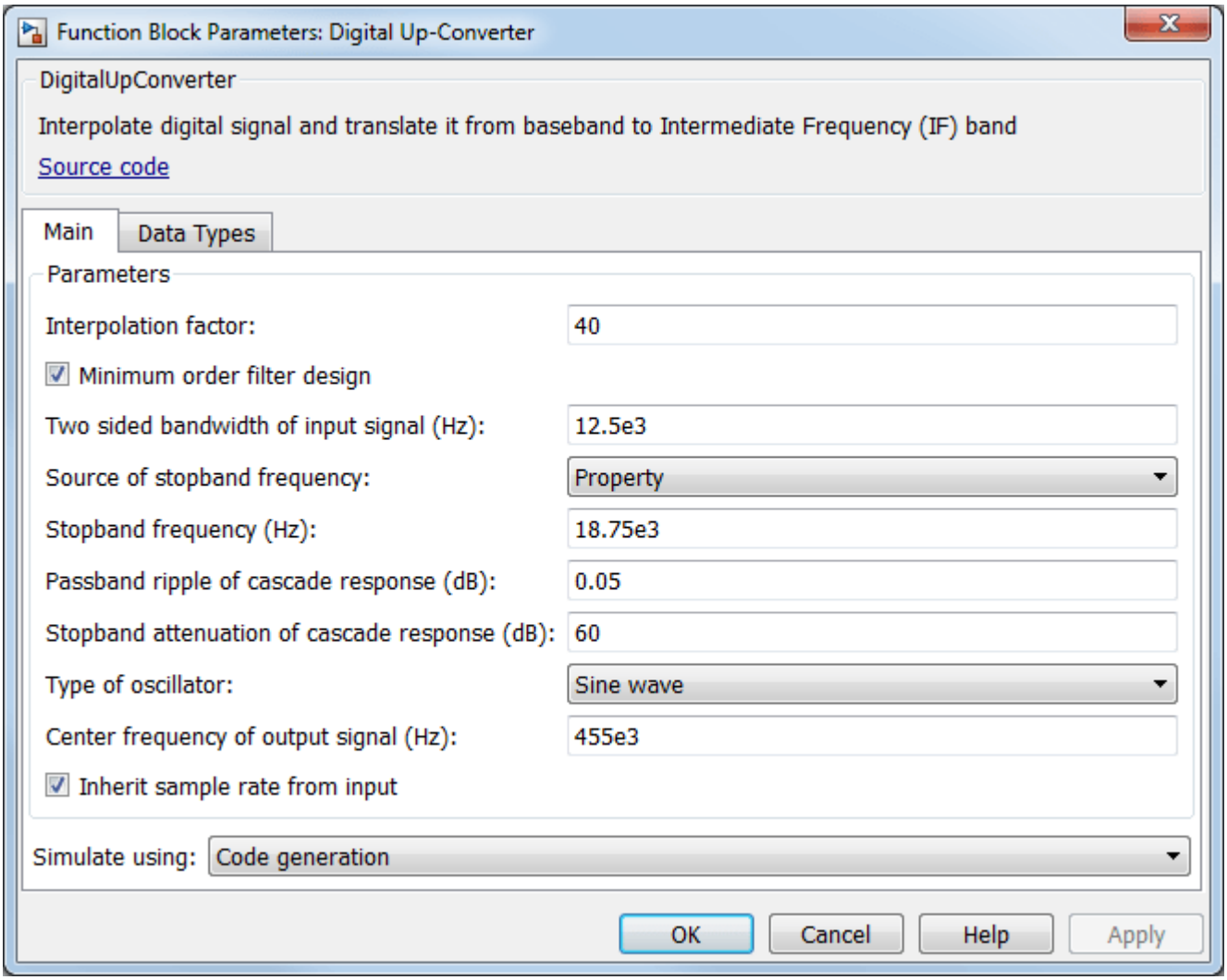
The setup for Family Radio Service shown above can be modeled in Simulink using the Digital Down-Converter and Digital Up-Converter blocks. This is implemented in the model familyRadioServiceExample.slx. While the simulation is running, you can listen to either the input speech signal or the signal after processing using a switch in the model.

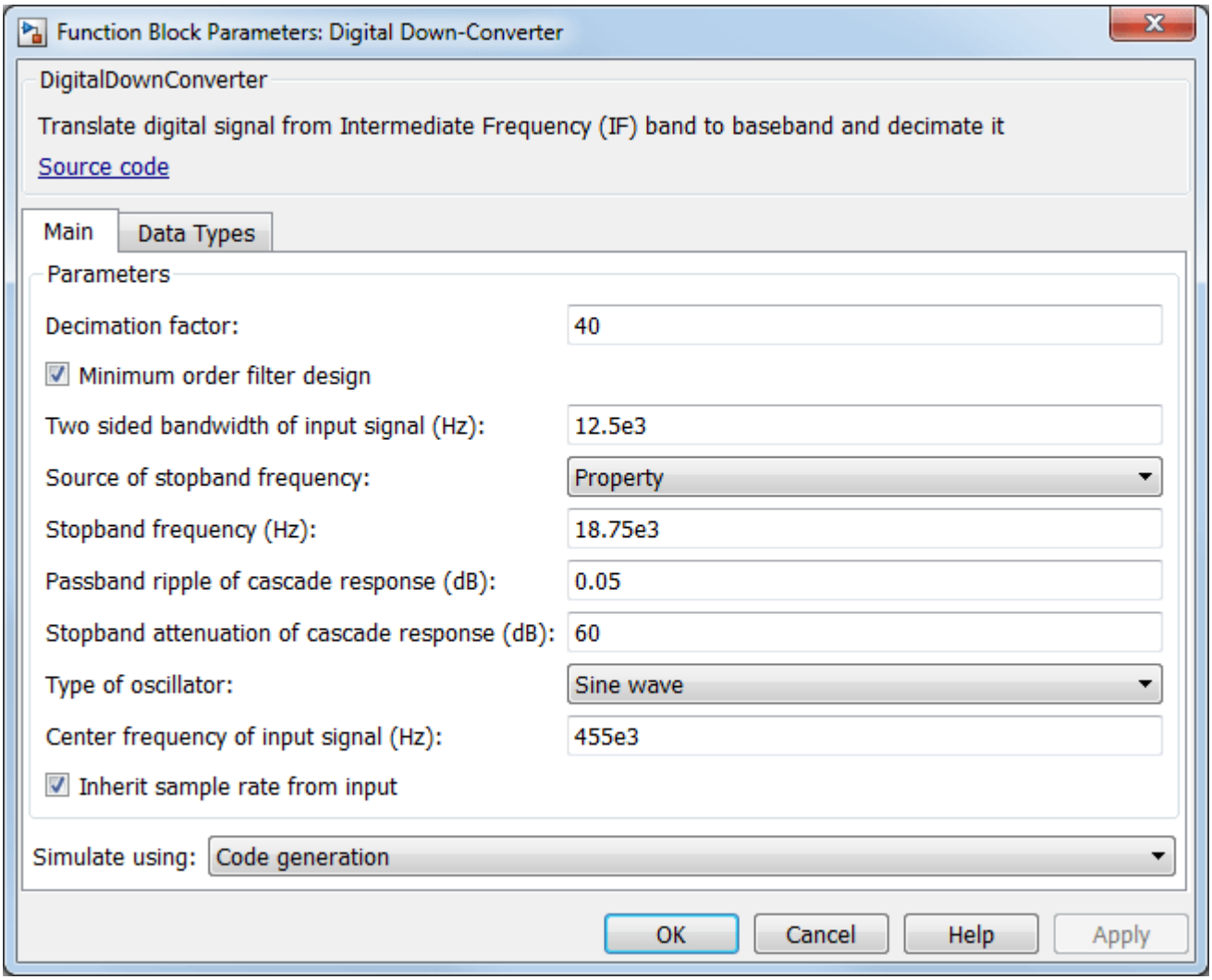
```
open_system('familyRadioServiceExample');
open_system('familyRadioServiceExample/Dataflow Subsystem');
```


1
Sample-Rate Converter
Resample to 50 KHz
In
Out
FM Baseband Modulator
DUC
Up-converted signal
1
Up-converted signal
Baseband signal
2
Baseband signal
Down-converted signal
3
Down-converted signal
Resample to 8 KHz
FM Demodulator
4
Sample-Rate Converter
Out
In
DDC


Baseband Spectrum Analyzer
File
Tools
View
Simulation
Help
Power spectrum of baseband signal
0
-50
-100
-150
-200
dBm
Run simulation to see results.
Frequency
Ready
RBW= Hz
Sample rate=10 kHz


UpConverted Signal Spectrum Analyzer
File
Tools
View
Simulation
Help
Power spectrum of signal after DUC
40
20
0
-20
-40
-60
-80
-100
-120
-140
dBm
Run simulation to see results.
Frequency
Ready
RBW= Hz
Sample rate=10 kHz


Function Block Parameters: Digital Up-Converter
DigitalUpConverter
Interpolate digital signal and translate it from baseband to Intermediate Frequency (IF) band
Source code
Main
Data Types
Parameters
Interpolation factor:
40
Minimum order filter design
Two sided bandwidth of input signal (Hz):
12.5e3
Source of stopband frequency:
Property
Stopband frequency (Hz):
18.75e3
Passband ripple of cascade response (dB):
0.05
Stopband attenuation of cascade response (dB):
60
Type of oscillator:
Sine wave
Center frequency of output signal (Hz):
455e3
Inherit sample rate from input
Simulate using:
Code generation
OK
Cancel
Help
Apply



Using Dataflow in Simulink

You can configure this example to use data-driven execution by setting the Domain parameter to dataflow for Dataflow Subsystem. With dataflow, blocks inside the domain execute based on the availability of data as opposed to Simulink's sample time. Simulink automatically partitions the system into concurrent threads. This autopartitioning accelerates simulation and increases data throughput. To learn more about dataflow and how to run this example using multiple threads, see “Multicore Execution using Dataflow Domain” on page 8-19.

Conclusions

In this example you designed a digital up and down converter for an FRS transmitter and receiver using DUC/DDC System objects. The example has explored the different options offered by the DUC/DDC objects to design interpolation and decimation filters. The example has also explored the filter analysis tools available in the DDC/DUC objects such as the `visualizeFilterStages`, `fvtool`, and `getFilterOrders` methods. A Simulink implementation for the configuration was also modeled.

The DUC/DDC objects designed in this example operate with double-precision filter coefficients and double-precision arithmetic. See the “Design and Analysis of a Digital Down Converter” on page 4-395 example if you are interested in designing a DUC or DDC that operates with fixed-point inputs.

```
close_system('familyRadioServiceExample', 0);
```

Parametric Audio Equalizer

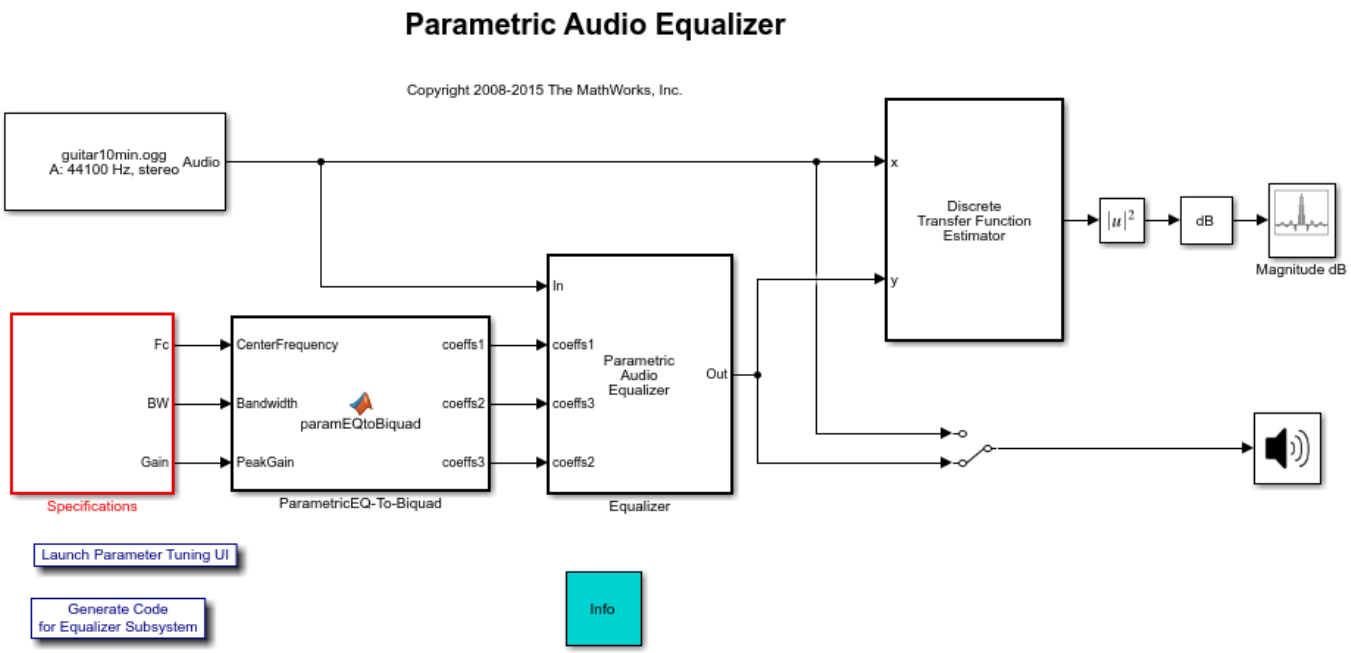
This example shows how to model an algorithm specification for a three band parametric equalizer

Introduction

Parametric equalizers are often used to adjust the frequency response of an audio system. For example, a parametric equalizer can be used to compensate for physical speakers which have peaks and dips at different frequencies.

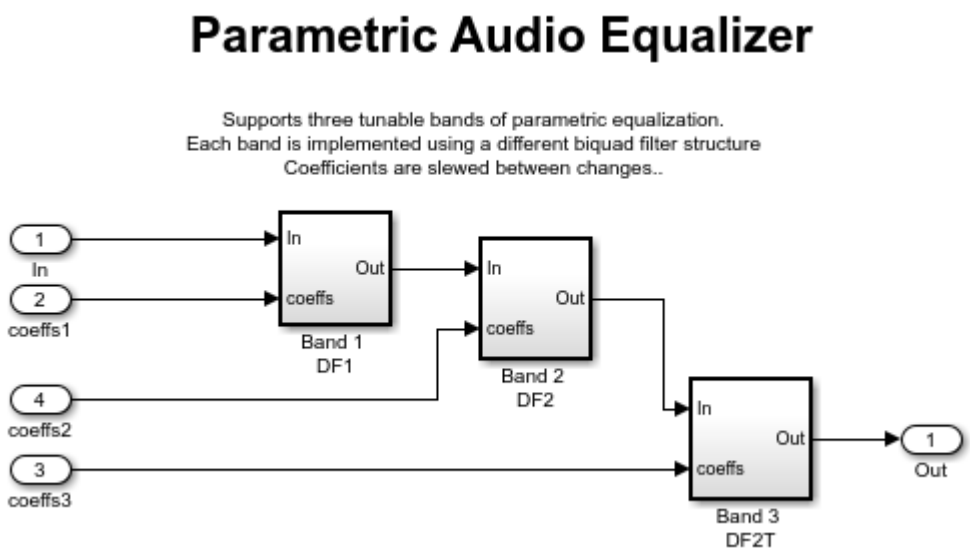
The parametric equalizer algorithm in this example provides three second-order (biquadratic) filters whose coefficients can be adjusted to achieve a desired frequency response. A user interface (UI) is used in simulation to dynamically adjust filter coefficients and explore behavior.

This example will describe how the parametric equalizer algorithm is specified and how the behavior can be explored through simulation



Parametric Equalizer

In this example, the equalizer is implemented in the Equalizer subsystem of the model. In this subsystem, the input is passed through three cascaded bands of equalization. Coefficient changes within each band are smoothed through a leaky integrator before being passed into a Biquad Filter block.

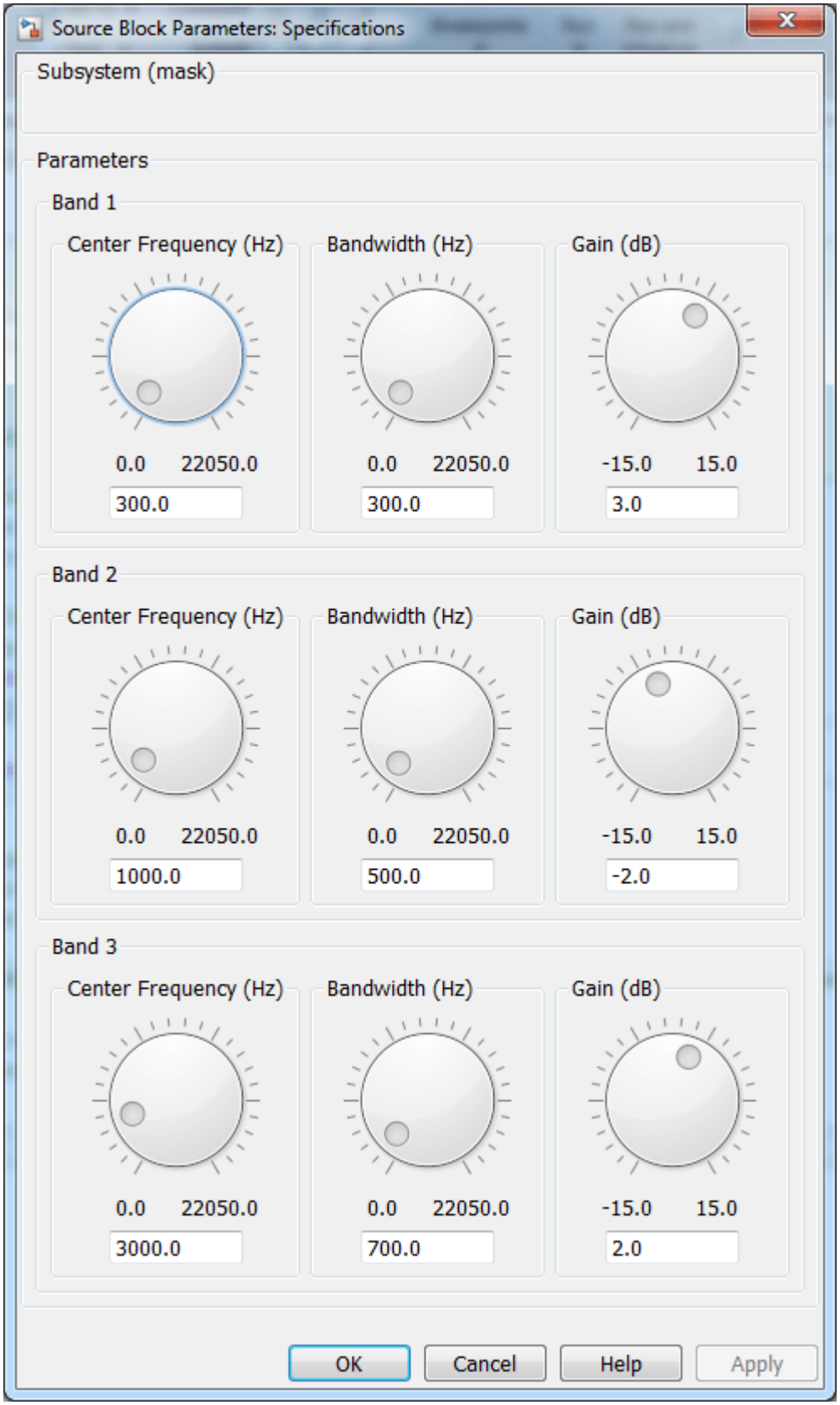


Equalizer Specifications

This example allows tuning of each equalizer band's center frequency, bandwidth, and peak (or dip) gain. The bandwidth is defined at the arithmetic mean between the base of the filter (1 in this example) and the peak power value. The specifications of the three bands are in the Specifications subsystem. These specifications are converted to Biquad coefficients using a MATLAB Function block. The coefficients of a particular band are recomputed whenever any of that band's specifications are modified.

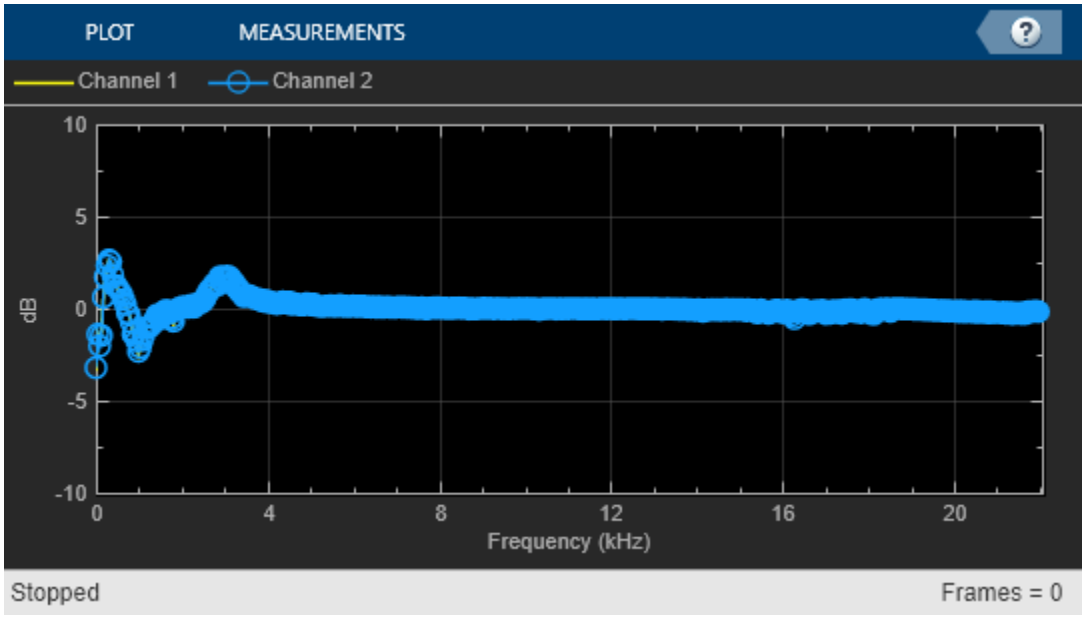
User Interface

A UI designed to interact with the simulation is provided with the model and can be launched by clicking the 'Launch Parameter Tuning UI' link. The UI allows you to tune the equalizer specifications and the results are reflected in the simulation instantly.


Source Block Parameters: Specifications
Subsystem (mask)
Parameters
Band 1
Center Frequency (Hz)
Bandwidth (Hz)
Gain (dB)
0.0 22050.0
300.0
0.0 22050.0
300.0
-15.0 15.0
3.0
Band 2
Center Frequency (Hz)
Bandwidth (Hz)
Gain (dB)
0.0 22050.0
1000.0
0.0 22050.0
500.0
-15.0 15.0
-2.0
Band 3
Center Frequency (Hz)
Bandwidth (Hz)
Gain (dB)
0.0 22050.0
3000.0
0.0 22050.0
700.0
-15.0 15.0
2.0
OK
Cancel
Help
Apply

Exploring the Simulation

When you simulate the model, you will visualize the equalizer's response on a scope. The response is computed using a Discrete Transfer Function Estimator block. The response will change as you tune the equalizer specifications. You can also listen to either the original or equalized audio by toggling the manual switch.



Generate C Code for the Equalizer Subsystem

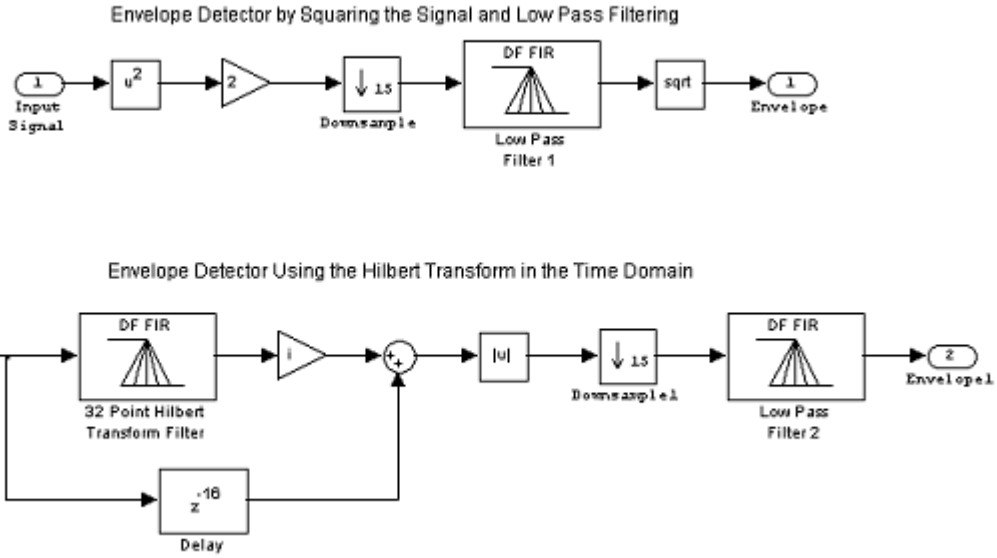
To learn how to generate C code for the equalizer subsystem based on the algorithm specifications, see the “Code Generation for Parametric Audio Equalizer” on page 4-306 example.

Envelope Detection

This example shows how to implement two common methods of envelope detection. One method uses squaring and lowpass filtering. The other uses the Hilbert transform. This example illustrates MATLAB® and Simulink® implementations.

Introduction

The signal's envelope is equivalent to its outline, and an envelope detector connects all the peaks in this signal. Envelope detection has numerous applications in the fields of signal processing and communications, one of which is amplitude modulation (AM) detection. The following block diagram shows the implementation of the envelope detection using the two methods.



Method 1: Squaring and Lowpass Filtering

This envelope detection method involves squaring the input signal and sending this signal through a lowpass filter. Squaring the signal effectively demodulates the input by using itself as its own carrier wave. This means that half the energy of the signal is pushed up to higher frequencies and half is shifted down toward DC. You then downsample this signal to reduce the sampling frequency. You can do downsampling if the signal does not have any high frequencies which could cause aliasing. Otherwise an FIR decimation should be used which applies a lowpass filter before downsampling the signal. After this, pass the signal through a minimum-phase, lowpass filter to eliminate the high frequency energy. Finally you are left with only the signal envelope.

To maintain the correct scale, you must perform two additional operations. First, you must amplify the signal by a factor of two. Since you are keeping only the lower half of the signal energy, this gain matches the final energy to its original energy. Second, you must take the square root of the signal to reverse the scaling distortion that resulted from squaring the signal.

This envelope detection method is easy to implement and can be done with a low-order filter, which minimizes the lag of the output.

Method 2: The Hilbert Transform

This envelope detection method involves creating the analytic signal of the input using the Hilbert transform. An analytic signal is a complex signal, where the real part is the original signal and the imaginary part is the Hilbert transform of the original signal.

Mathematically the envelope e(t) of a signal x(t) is defined as the magnitude of the analytic signal as shown by the following equation.

$$e(t) = \sqrt{x(t)^2 + \hat{x}(t)^2}$$

where

$$\hat{x}(t)$$

is the Hilbert transform of x(t).

You can find the Hilbert transform of the signal using a 32-point Parks-McClellan FIR filter. To form the analytic signal, you then multiply the Hilbert transform of the signal by sqrt(-1) (the imaginary unit) and add it to the time-delayed original signal. It is necessary to delay the input signal because the Hilbert transform, which is implemented by an FIR filter, will introduce a delay of half the filter length.

You find the envelope of the signal by taking the absolute value of the analytic signal. The envelope is a low frequency signal compared to the original signal. To reduce its sampling frequency, to eliminate ringing and to smooth the envelope, you downsample this signal and pass the result through a lowpass filter.

MATLAB Example: Initialization

Initialize required variables such as those for the frame size and file name. Creating and initializing your System objects before they are used in a processing loop is critical for getting optimal performance.

```
Fs = 22050;
numSamples = 10000;
DownsampleFactor = 15;
frameSize = 10*DownsampleFactor;
```

Create a sine wave System object and set its properties to generate two sine waves. One sine wave will act as the message signal and the other sine wave will be the carrier signal to produce Amplitude Modulation.

```
sine = dsp.SineWave([0.4 1],[10 200], ...
    'SamplesPerFrame',frameSize, ...
    'SampleRate',Fs);
```

Create a lowpass FIR filter for filtering the squared signal to detect its envelope.

```
lp1 = dsp.FIRFilter('Numerator',firpm(20,[0 0.03 0.1 1],[1 1 0 0]));
```

Create three digital filter System objects. The first implements the Hilbert transformer, the second compensates for the delay introduced by the Hilbert transformer, and the third is a lowpass filter for detecting the signal envelope.

```
N = 60; % Filter order
hilbertTransformer = dsp.FIRFilter( ...
```

```
        'Numerator',firpm(N,[0.01 .95],[1 1],'hilbert'));
delay = dsp.Delay('Length',N/2);
lp2 = dsp.FIRFilter('Numerator',firpm(20,[0 0.03 0.1 1],[1 1 0 0]));
```

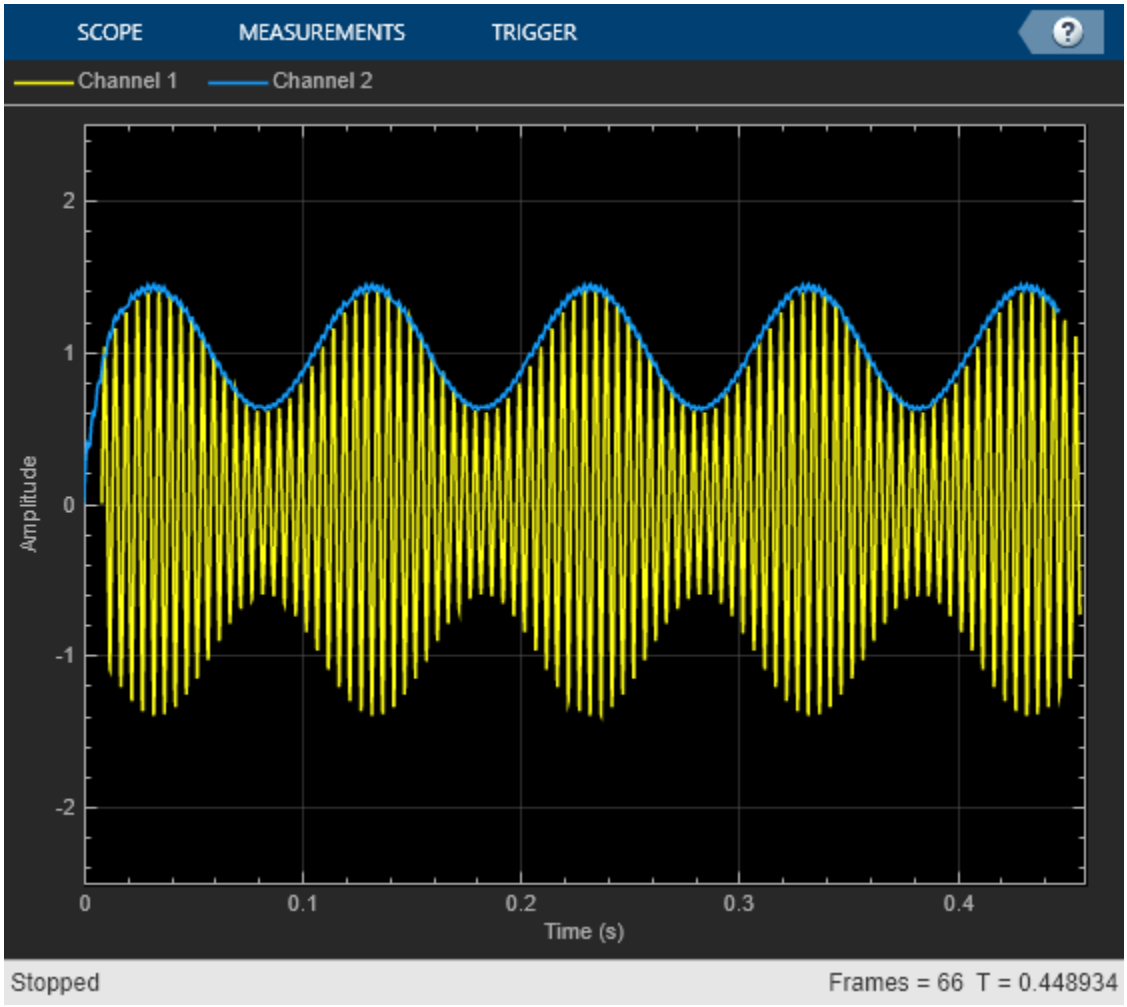
Create and configure two time scope System objects to plot the input signal and its envelope.

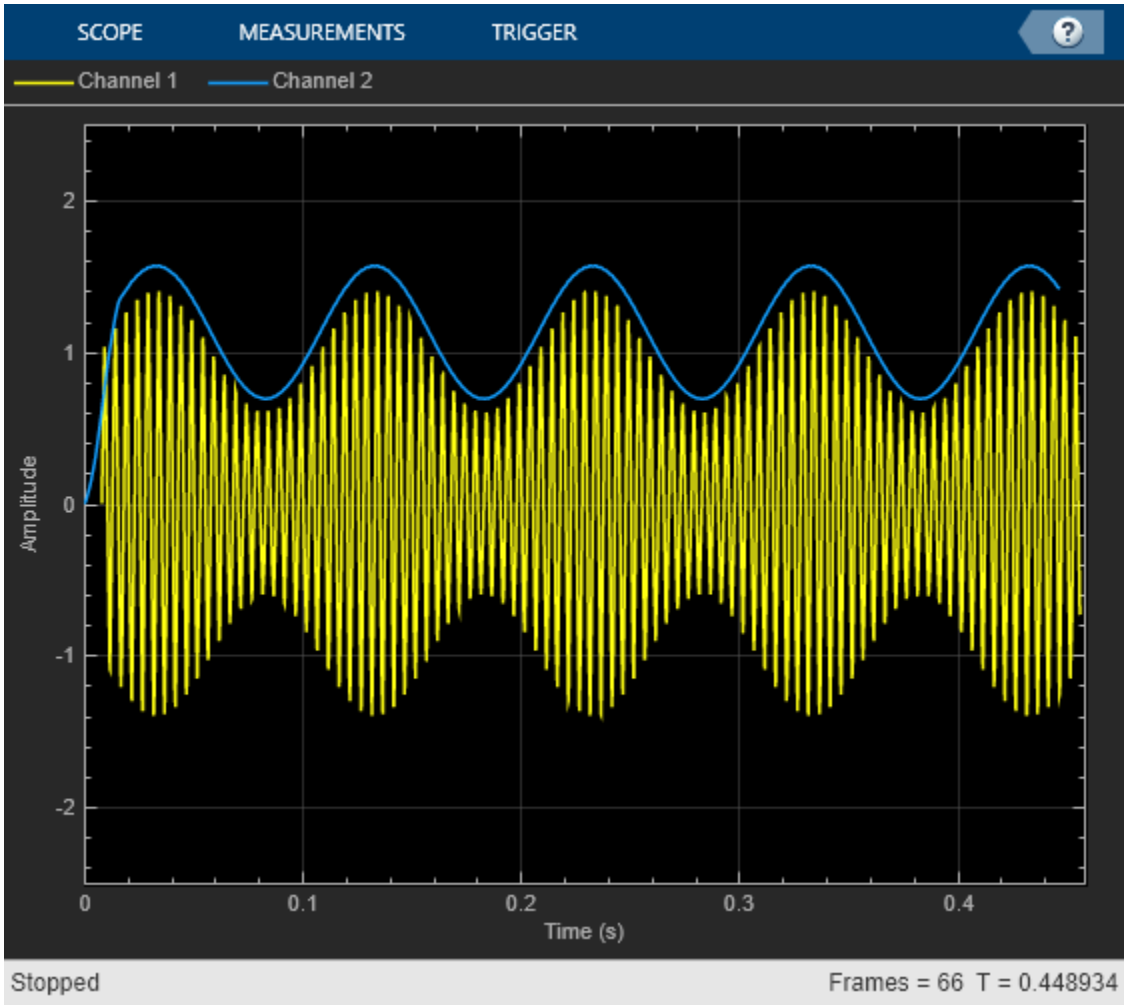
```
scope1 = timescope( ...
  'Name','Envelope detection using Amplitude Modulation', ...
  'SampleRate',[Fs,Fs/DownsampleFactor], ...
  'TimeDisplayOffset',[(N/2+frameSize)/Fs,0], ...
  'TimeSpanSource','property', ...
  'TimeSpan',0.45, ...
  'YLimits',[-2.5 2.5], ...
  'Position', [100 200 560 500]);
pos = scope1.Position;

scope2 = timescope( ...
  'Name','Envelope detection using Hilbert Transform', ...
  'Position',[pos(1) + pos(3), pos(2:4)], ...
  'SampleRate',[Fs,Fs/DownsampleFactor], ...
  'TimeDisplayOffset',[(N/2+frameSize)/Fs,0], ...
  'TimeSpanSource','Property', ...
  'TimeSpan',0.45, ...
  'YLimits',[-2.5 2.5]);
```

MATLAB Example: Stream Processing Loop

Create the processing loop to perform envelope detection on the input signal. This loop uses the System objects you instantiated.

```
for i = 1:numSamples/frameSize
    sig = sine();
    sig = (1 + sig(:,1)) .* sig(:, 2);      % Amplitude modulation

    % Envelope detector by squaring the signal and lowpass filtering
    sigsq = 2 * sig .* sig;
    sigenv1 = sqrt(lp1(downsample(sigsq,DownsampleFactor)));

    % Envelope detector using the Hilbert transform in the time domain
    sige = abs(complex(0, hilbertTransformer(sig)) + delay(sig));
    sigenv2 = lp2(downsample(sige,DownsampleFactor));

    % Plot the signals and envelopes
    scope1(sig,sigenv1);
    scope2(sig,sigenv2);
end
release(scope1);
release(scope2);
```


SCOPE
MEASUREMENTS
TRIGGER
Channel 1
Channel 2
2
1
0
-1
-2
Amplitude
0
0.1
0.2
0.3
0.4
Time (s)
Stopped
Frames = 66 T = 0.448934



MATLAB Example: Envelope Detector Results

In the plots, for the envelope detection method using Hilbert transform the envelope amplitude does not match the actual signal, because the Hilbert transform which was implemented using the FIR filter is not ideal. That is, the magnitude response is not one for all frequencies. The shape of the envelope still matches the actual signal's envelope.

Simulink Example: Method 1 (Squaring and Lowpass Filtering)

As above, Method 1 works by squaring the input signal and sending it through a lowpass filter.

In this Simulink example, a simple minimum-phase lowpass filter is used to remove the high frequency energy. In order to maintain the correct scaling, two more operations are included. The first is to place a gain of 2 on the signal. Since we are only keeping the lower half of the signal energy, this gain boosts the final energy to match its original energy. Finally, the square root of the signal is taken to reverse the scaling distortion from the input signal squaring operation.

This method is useful because it is very easy to implement and can be done with a low-order filter, minimizing the lag of the output.

Simulink Example: Method 2 (The Hilbert Transform)

As above, Method 2 works by creating the analytic signal of the input using a Hilbert transformer.

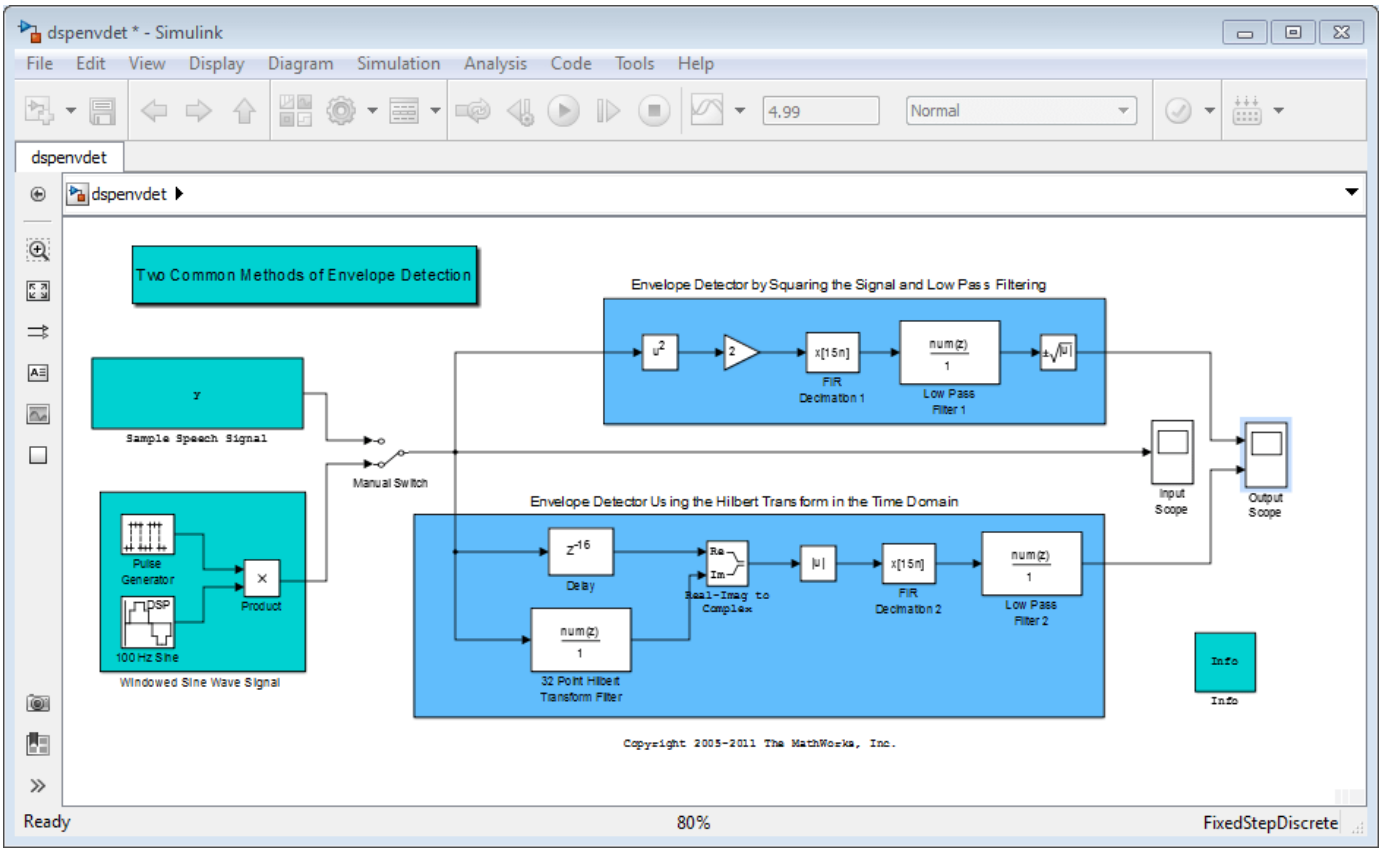
In this Simulink example, the Hilbert transform of the signal is found using a 32-point Parks-McClellan FIR filter. The Hilbert transform of the signal is then multiplied by i (the imaginary unit) and added to the original signal. The original signal is time-delayed before being added to the Hilbert transform to match the delay caused by the Hilbert transform, which is one-half the length of the Hilbert filter.

The envelope of the signal can be found by taking the absolute value of the analytic signal. In order to eliminate ringing and smooth the envelope, the result is subjected to a lowpass filter.

Note that the **Analytic Signal** block found in the DSP System Toolbox™ could also be used to implement this envelope detection design.

Simulink Example: Envelope Detector Model

The all-platform floating-point version of the Simulink model is shown below. When you run the model, you will see the original signal and the results of both envelope detectors.

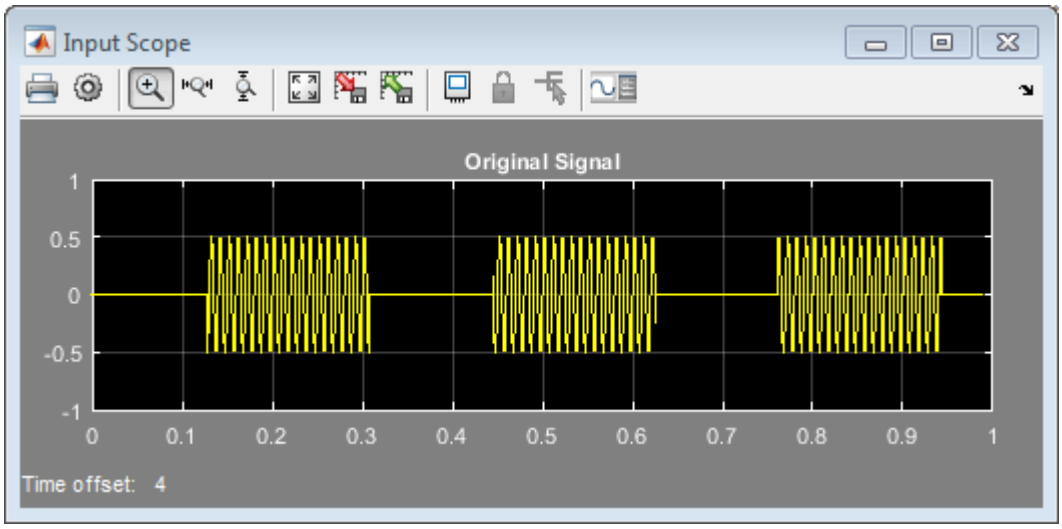


Simulink Example: Envelope Detector Results

This example shows the results of the two different envelope detectors for two different types of input signals. The input choices are a sample speech signal or a 100 Hz sine wave that turns on and off.

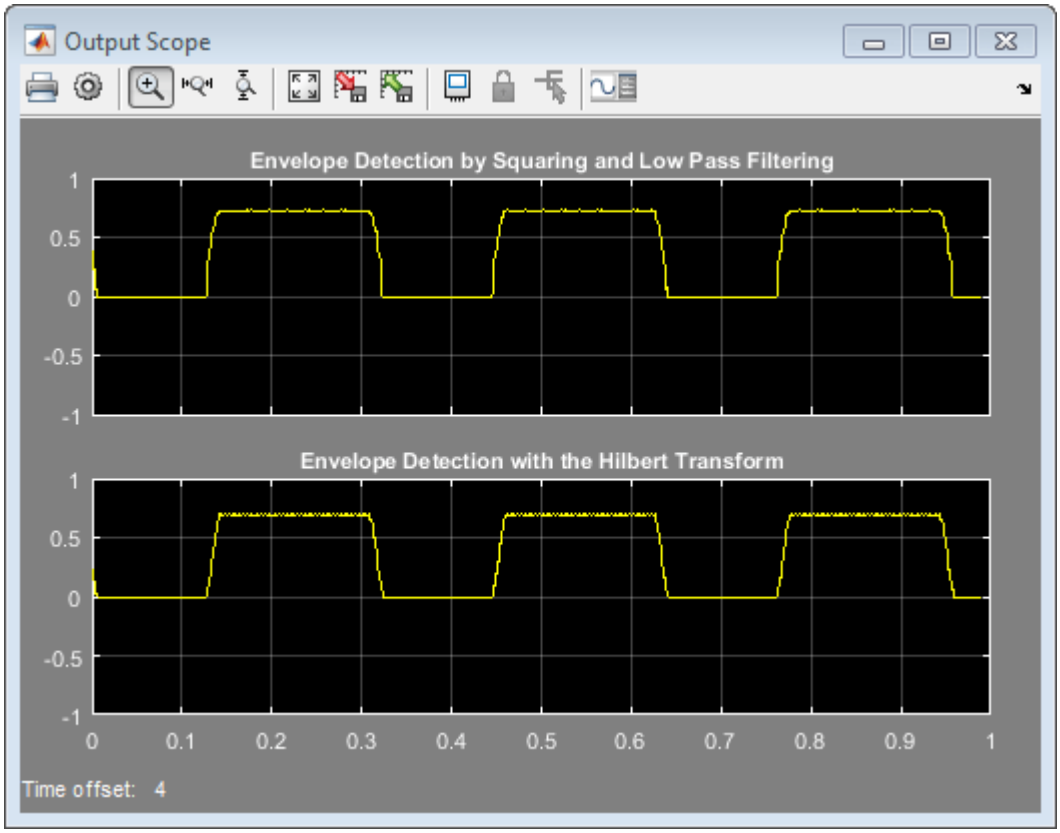
The model has a switchable input and two outputs which are routed to scopes for easy viewing. If a signal is not visible, double-click on the Scope block to open it.

The input scope plot shows the original signal. The signal lasts a total of 5 seconds, with 1 second of data being shown at a time.



The first output scope plot shows the output of the first envelope detector. This is the result of squaring the original signal and sending it through a low-pass filter. You can clearly see that the envelope was successfully extracted from the speech signal.

The second output scope plot shows the output of the second envelope detector, which employs a Hilbert transform. Though the output of this envelope detector looks very similar to the first method, you can see differences between them.



Simulink Model

All-platform floating-point version: dspenvdet

DTMF Generator and Receiver

This example shows how to model a dual-tone multifrequency (DTMF) generator and receiver. The model includes a bandpass filter bank receiver, a spectrum analyzer block showing a spectrum and spectrogram plot of the generated tones, a shift register to store the decoded digits, and a real-time soundcard audio on all platforms.

DTMF Generator

DTMF signaling uses two tones to represent each key on the touch pad. There are 12 distinct tones. When any key is pressed the tone of the column and the tone of the row are generated. As an example, pressing the '5' button generates the tones 770 Hz and 1336 Hz. In this example, use the number 10 to represent the '*' key and 11 to represent the '#' key.

The frequencies were chosen to avoid harmonics: no frequency is a multiple of another, the difference between any two frequencies does not equal any of the frequencies, and the sum of any two frequencies does not equal any of the frequencies.

The frequencies of the tones are as follows:

```
        1209 Hz    1336 Hz    1477 Hz

697 Hz    1          2          3
770 Hz    4          5          6
852 Hz    7          8          9
941 Hz    *          0          #
```

DTMF Receiver

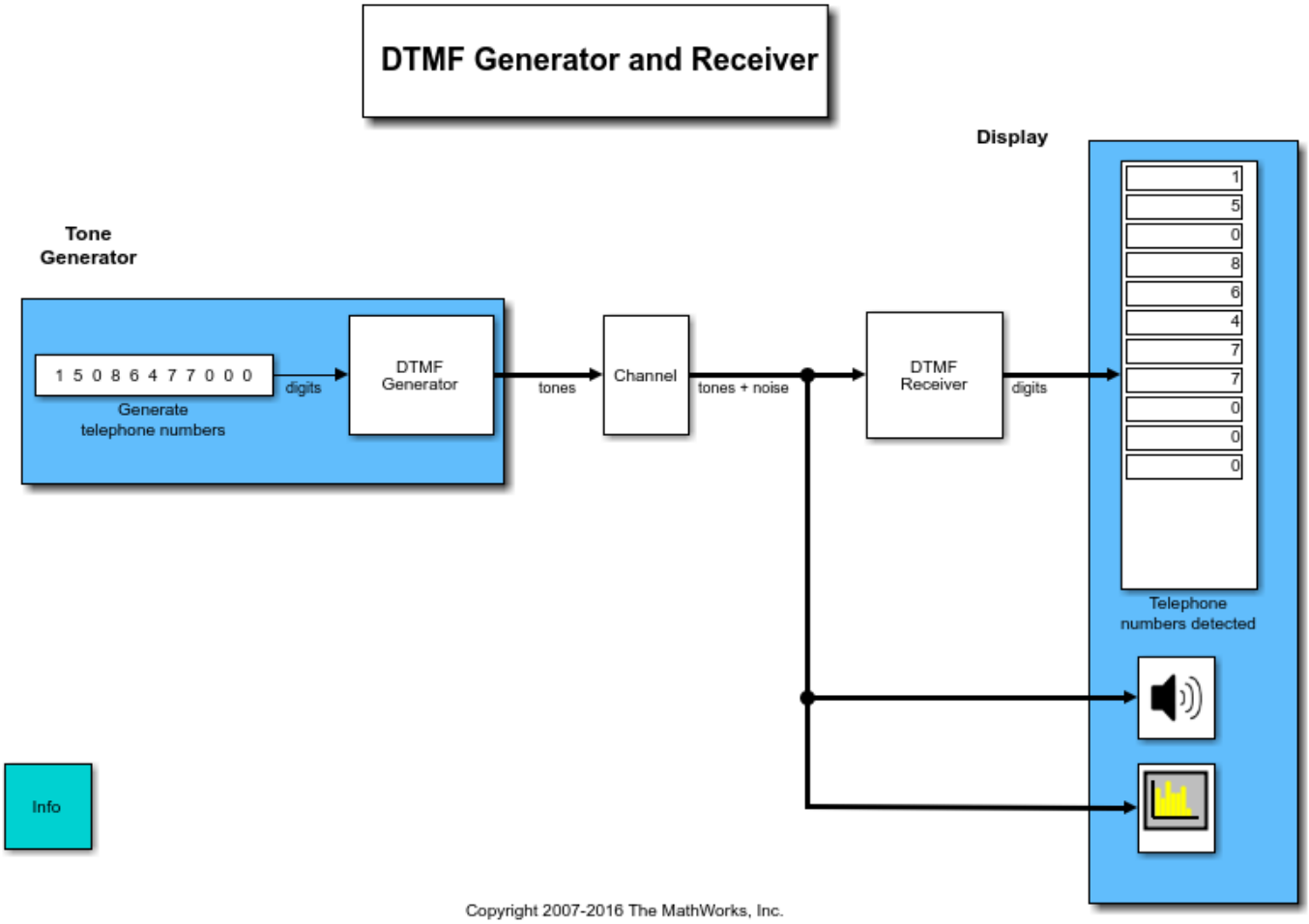
At the receiver the tone frequencies are detected and the number decoded. The DFT algorithm can be used to detect frequencies, but since there are only 7 frequency components (4 low frequencies and 3 high frequencies), a more efficient method is the Goertzel algorithm. This method detects the frequency components by passing the received signal through 7 bandpass filters. The filter bandwidths are adjustable as a percentage of the center frequency by adjusting the bandwidth parameter on the DTMF Receiver block mask.

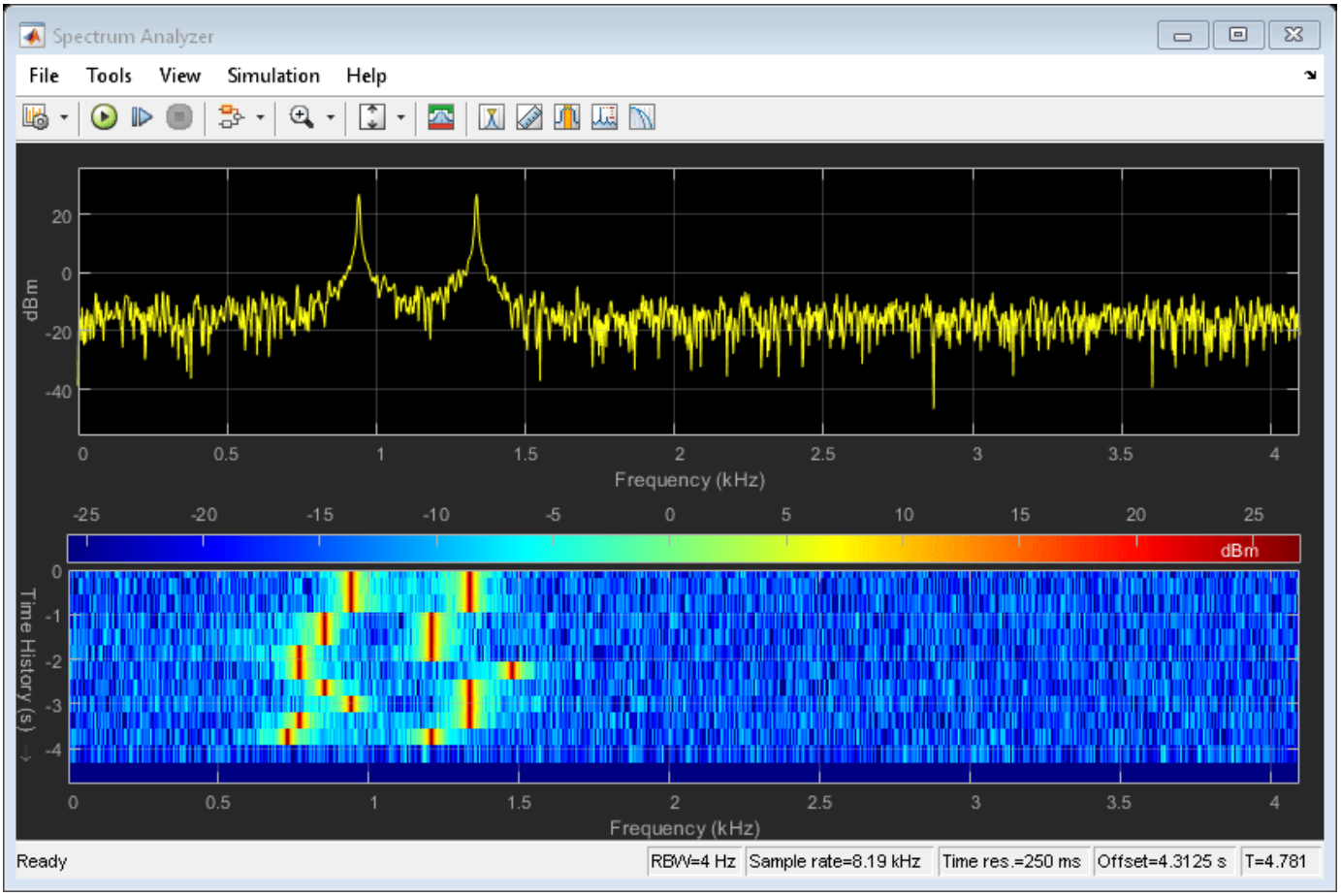
Running the Example

When you run the simulation, the spectrogram of the received tone will be constructed. If you use the version of the model designed for audio hardware, the received tone is played through the system soundcard. The detected dialed numbers will be shown on the numeric display scope. The following parameters can be adjusted:

- Frequency bias for each tone (from the DTMF Generator mask dialog)
- Channel noise power and signal gain (from the Channel mask dialog)
- Receiver bandpass filter frequency bandwidth (from the DTMF Receiver mask dialog)

Example Model




Spectrum Analyzer
File
Tools
View
Simulation
Help
dBm
20
0
-20
-40
0
0.5
1
1.5
2
2.5
3
3.5
4
Frequency (kHz)
-25
-20
-15
-10
-5
0
5
10
15
20
25
dBm
Time History (s)
0
-1
-2
-3
-4
0
0.5
1
1.5
2
2.5
3
3.5
4
Frequency (kHz)
Ready
RBW=4 Hz
Sample rate=8.19 kHz
Time res.=250 ms
Offset=4.3125 s
T=4.781

WWV Digital Receiver - Synchronization and Detection

This example shows an implementation of a digital receiver that synchronizes to the time code information broadcast by radio station WWV and decodes it to display time information. The example uses the Simulink®, DSP System Toolbox™, and Stateflow® products with the MATLAB® Function block to achieve a simple noncoherent digital receiver.

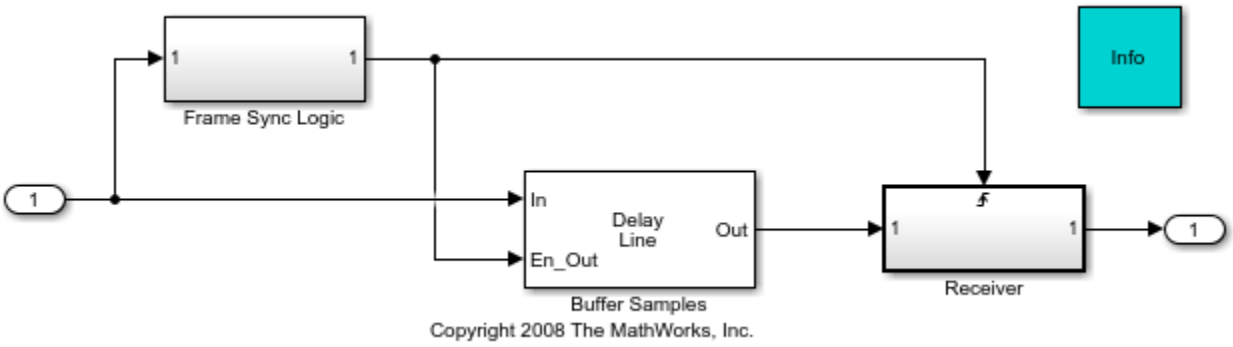
What Is WWV?

WWV is the call sign of a US government radio station run by the National Institute of Standards and Technology in Fort Collins, Colorado. WWV transmits frequency reference standards and time code information. The transmitted time code is referenced to a Cesium clock with a timing accuracy of 10 microseconds and a frequency accuracy of 1 part in 100 billion. The time code is transmitted using a 100-Hz audio signal with pulse-width modulation using the IRIG-B time code format.

You can find more information on WWV at Radio Station WWV.

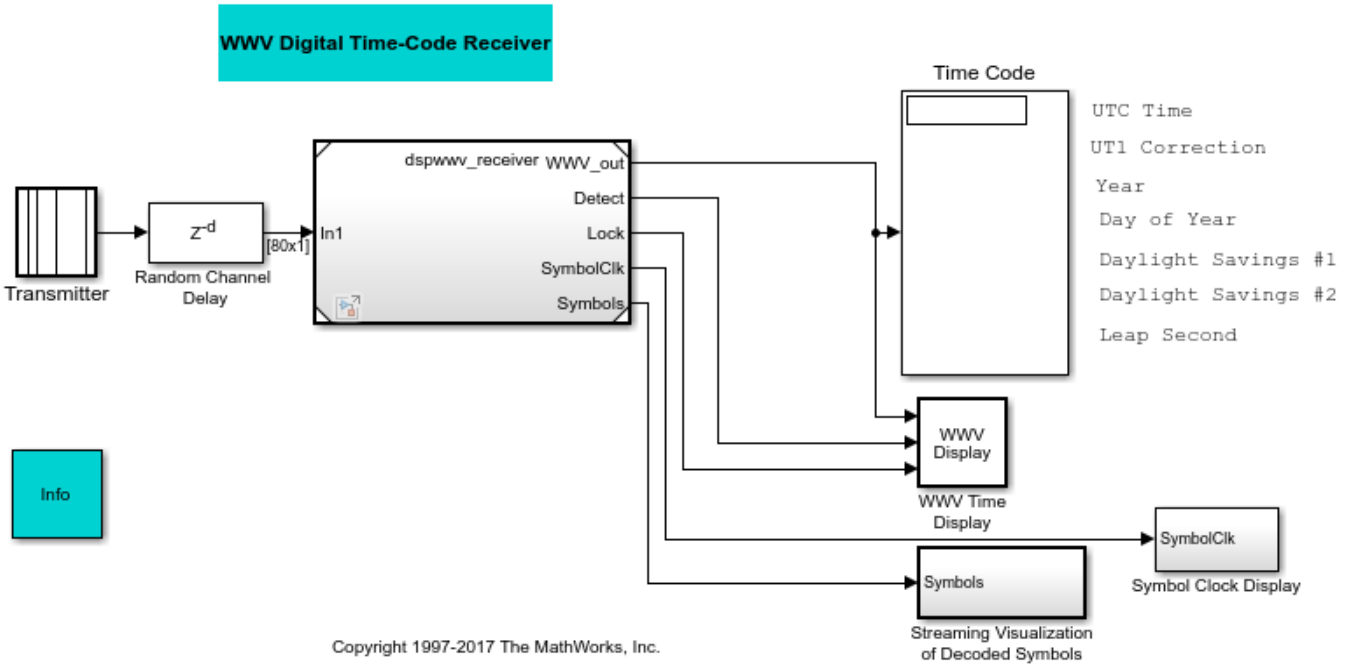
Introduction to Synchronization

Synchronization is a common problem in Communications applications. This example shows you one way of implementing a solution to this problem in Simulink. Consider the following simple model:



The Buffer Samples block maintains an internal circular buffer for efficient buffering of input samples. It uses a mode where a valid output frame is computed only when it receives a Boolean 'true' at the En_Out input port. The Frame Sync Logic subsystem outputs a Boolean 'true' when an appropriate frame, as expected by the Receiver, has been buffered. The same Boolean signal also acts as a trigger to the Receiver subsystem, which processes the valid frame. Due to this arrangement, the output sections of the Buffer Samples block and the Receiver subsystem only run when required. This arrangement is used in two places in this example, once for symbol synchronization and demodulation and then again for frame synchronization and decoding.

Exploring the Example



The example model consists of the following parts, which are described in the sections below:

- **Transmitter** - Generates and transmits a BCD time code
- **Random Channel Delay** - Adds random delay to the transmitted signal
- **Model** - References dspwwv_receiver model through a model reference block. This model consists of:

1. **Receiver** - Demodulates the received time code, synchronizes and locks in with the received signal, and detects the BCD symbols
2. **Decoder** - Decodes the BCD symbols

- **Display** - displays the corresponding time and date information

Note that `dspwwv` does not support code generation, but `dspwwv_receiver` does.

Transmitter

This subsystem generates a Binary Coded Decimal (BCD) time code on an 100-Hz tone. The sampling rate (Ts) used by the Simulink model is 8000 samples/sec. The time code broadcast by WWV provides UTC (Coordinated Universal Time) information serially at a speed of 1 bit per second. It requires 60 bits, or one minute, to send the entire time code. Various bits in each time code convey the following information:

- 24 hour time (UTC)
- UT1 time correction
- Year

- Day of year
- Daylight Saving indicators
- Leap seconds correction

Refer to the 'WWV Time Code Bits' and 'WWV Time Code Format' sections at the NIST website for more information on the time code. Depending on whether you select 'Current' or 'User-specified' for the **Display time** parameter on the transmitter subsystem mask, the subsystem generates the corresponding 60 BCD time code symbols. Each symbol is represented using Pulse Width Modulation (PWM) of an 100-Hz tone and is output from the Transmitter subsystem. One of the following possible symbols are transmitted each second:

1. MISS - No pulse is sent at the beginning of each frame, to indicate the start of a new frame
2. ZERO - A 170-ms pulse indicates a 0 bit
3. ONE - A 470-ms pulse indicates a 1 bit
4. MARKER - A 770-ms pulse is sent every 10 seconds for synchronization

The transmitted symbols are mapped to the following integer values in the Simulink model:

- 0 - MISS
- 1 - ZERO
- 2 - ONE
- 3 - MARKER

This transmitted tone is identical to the tone transmitted on the WWV subcarrier.

Random Channel Delay

This subsystem adds random delay to the transmitted signal. The receiver section synchronizes to the transmitted symbols and decodes the appropriate time code, even in the presence of an unknown delay.

Model - `dspwwv_receiver` Referenced Model

Double-click the Model block to open the `dspwwv_receiver` model. This model has all the components for appropriately demodulating, synchronizing, and detecting the transmitted signal. It consists of the following three subsystems:

R1 - Receiver

Double-click the Receiver Subsystem to view its component subsystems:

1) **Downconvert and Downsample** accepts as input the pulse width modulated signal. The subsystem demodulates the received signal by performing envelope detection, then performs lowpass filtering and downsamples by 80. Therefore, there are 100 samples for every transmitted symbol in the demodulated signal (`dm`). The output of this subsystem is a sequence of variable length square pulses.

2) **AGC (Automatic Gain Control)** estimates the amplitude of the `dm` signal, which is later used in thresholding the `dm` signal.

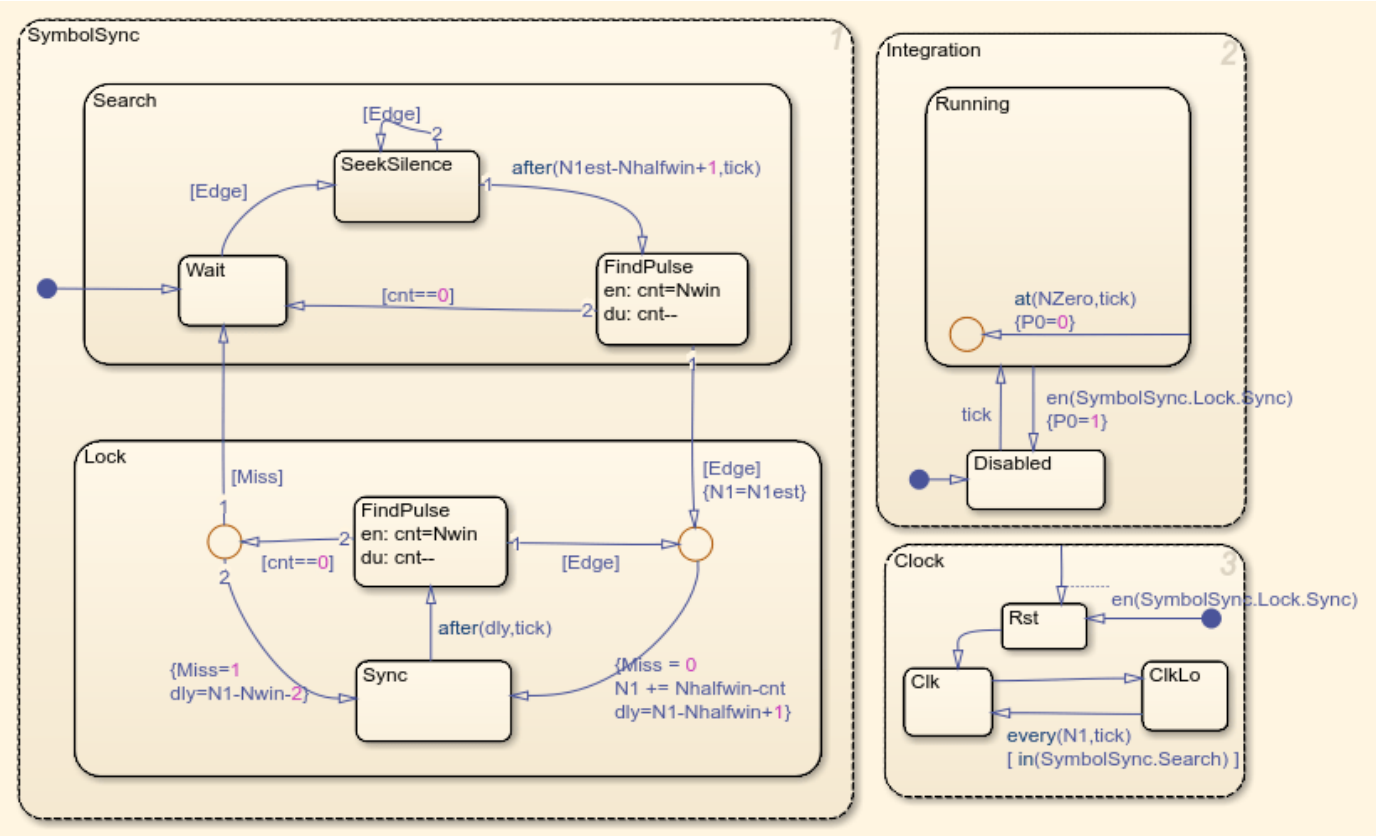
3) **Symbol Timing Recovery and Buffer for Demod** is used to achieve symbol synchronization and buffer the symbols for demodulation. It contains the following subsystems:

3.1) **Leading Edge Detector** takes in the demodulated signal `dm` and quantizes it into a Boolean signal. The `Detect` output signal is 'true' if the value of the `dm` signal is greater than the AGC value, otherwise it is 'false'. The subsystem also outputs the Boolean signal `Edge` that contains the rising edges of the `dm` signal.

3.2) **Symbol Sync** achieves symbol synchronization and creates a clock signal synchronized to the WWV signal. Note that the frame synchronization is done later on, in the Decoder section. Synchronization makes use of the Stateflow temporal logic feature. This Stateflow chart is composed of three parts:

1. SymbolSync - This chart is further divided into Sync State and Lock State charts
2. Clock Synchronization
3. Integration

Below are shown the **Symbol Timing Recovery and Buffer for Demod** subsystem and the **Symbol Sync** state chart.



3.2.1) **SymbolSync** performs symbol synchronization. The chart takes as input the rising edges (`Edge`) of the `dm` signal, which are approximately 100 samples apart.

The internal parameters of this chart are:

- *N1* - Actual number of samples between two edges
- *N1est* - Estimate of the number of samples between two edges (initial value 100)

- *Nwin* - Window in which to find another edge after N1est samples (default value 11 samples)
- *Nhalfwin* - Half of the window length (default value 6)

Sync State - To start synchronization, this chart looks for a rising edge, followed by a period of silence (no edges) for approximately 100 samples, and then looks for another rising edge in a window centered at that point. If the chart succeeds in doing this, the system claims to be synchronized and assumes that the rest of the symbols are valid symbols. Otherwise, the chart waits for such a pattern to occur again and keeps waiting until it succeeds:

1. The 'Sync' state chart waits for an edge and then seeks silence (no other edge) for at least *dly* = ' *N1est - Nhalfwin + 1* ' samples.
2. If this chart does not see silence in that duration (*dly* samples) and finds another edge, it treats the new edge as the reference edge and again seeks silence.
3. This chart repeats steps 1 and 2 until it succeeds in seeking silence for the next *dly* samples after the reference edge.
4. Once silence for *dly* samples is detected, this chart calculates how many samples (*cnt*) after *dly* samples it found another edge. If the next edge is found within the *Nwin* window, it transitions to the 'Lock' state to start receiving the subsequent symbols. If the next edge is not found within *Nwin* window samples, it discards the reference edge and starts searching for the reference edge again as described in steps 1-3.

Lock State - Once synchronized, this chart looks for the next symbol in a window centered at approximately every 100 samples and remains synchronized as long as it finds symbols. If the chart does not find any symbols for two consecutive times (approximately 200 samples), then it is no longer synchronized and tries to establish synchronization again as described above:

1. Once transitioned into the 'Lock' state, this chart assumes that the edges should now come in periodically (approximately every *N1est* samples).
2. The chart updates *N1* to *N1+Nhalfwin-cnt* and ignores the next *dly* = *N1-Nhalfwin+1* samples, and then searches for the next edge in a window of *Nwin* samples after that.
3. It keeps track of the number of samples in the window (*cnt*) after which it found the next edge. If it found an edge within the window, it again update *N1* as mentioned above.
4. Based on the new *cnt* value, it calculates the new *dly* and starts looking for the new edge as mentioned above.
5. The chart allows for not having found an edge within *Nwin* window once to account for the MISS symbol, but if that happens two consecutive times it gets out of the Lock state and starts symbol synchronization again by transitioning into the Sync state.

3.2.2) **Clock Synchronization** generates a clock signal when a new rising edge of the `dm` signal is received. This way the clock is synchronized with the occurrence of a new edge, rather than with the Simulink clock running periodically at a fixed rate.

3.2.3) **Integration** generates a template step function with a 17-sample width to represent a ZERO symbol when an edge is found, this is, whenever a clock is generated. This signal is used by the AGC subsystem.

3.3) **Symbol Buffer for Downstream Demod** buffers samples corresponding to a symbol when it receives a clock signal (computed above in 3.2.2).

4) **Symbol Demod and Frame Buffer** is triggered every time it receives a nonzero clock signal. It uses the Vector Quantization block to perform symbol demodulation by comparing the input 'Symbols'

buffer against the four possible symbol candidates (MISS, ZERO, ONE and MARKER). It outputs the symbol with the best match. The Delay Line block is used to buffer 60 consecutive symbols to create the 'WWV frame buffer.' The Frame Sync Logic subsystem preceding the Delay Line block looks for the occurrence of a consecutive MARKER and a MISS symbol, since this pattern indicates the start of a new WWV frame. The Delay Line block outputs a valid buffer only when this pattern in found. The subsequent IRIG-B decoder is also triggered at that instant.

R2 - IRIG-B Frame Decoder

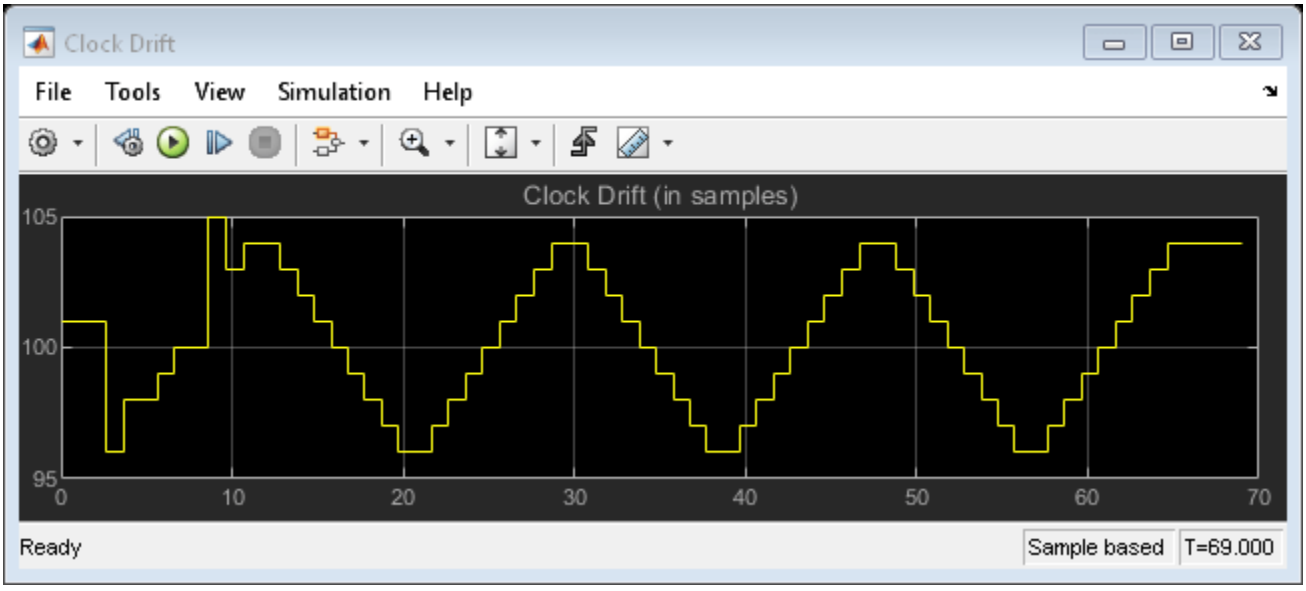
The IRIG-B Frame Decoder triggered subsystem consists of a MATLAB Function block that is used to decode the IRIG-B format symbol frames into individual elements of the time code. This subsystem is triggered only when a valid WWV frame is received.

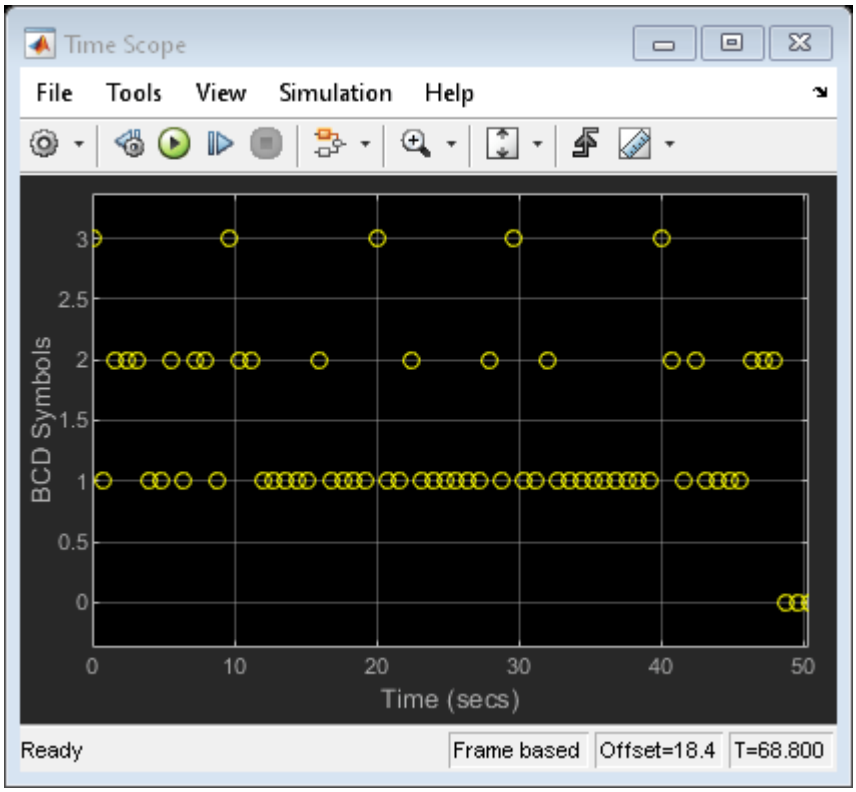
Display

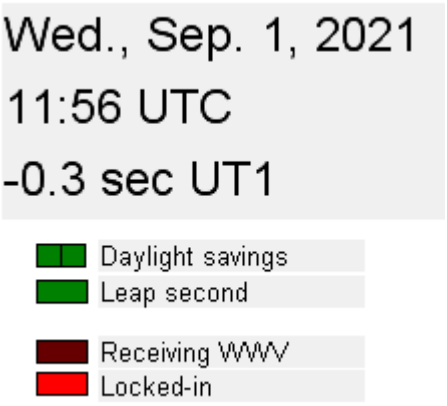
The transmitted symbols are displayed on the Decoded Symbols scope and the decoded time code information is displayed on the 'Time code' display and 'WWV time code' window. The boxes on the 'WWV time code' window represent LEDs that light up when the corresponding signal is true. The LED corresponding to Daylight Savings is split into two parts, where the first part is the 'Daylight savings indicator 1' and the second part is the 'Daylight savings indicator 2.' The Clock Drift plot indicates the number of samples between rising edges of successive symbols (`symbolClk`) as they are received. This plot varies between 95 and 105 samples.

Using the `dspwwv` Example Model

Simulate the model. You will see the clock drift, the corresponding BCD time-code symbols and the current time displayed (shown below in that order.)



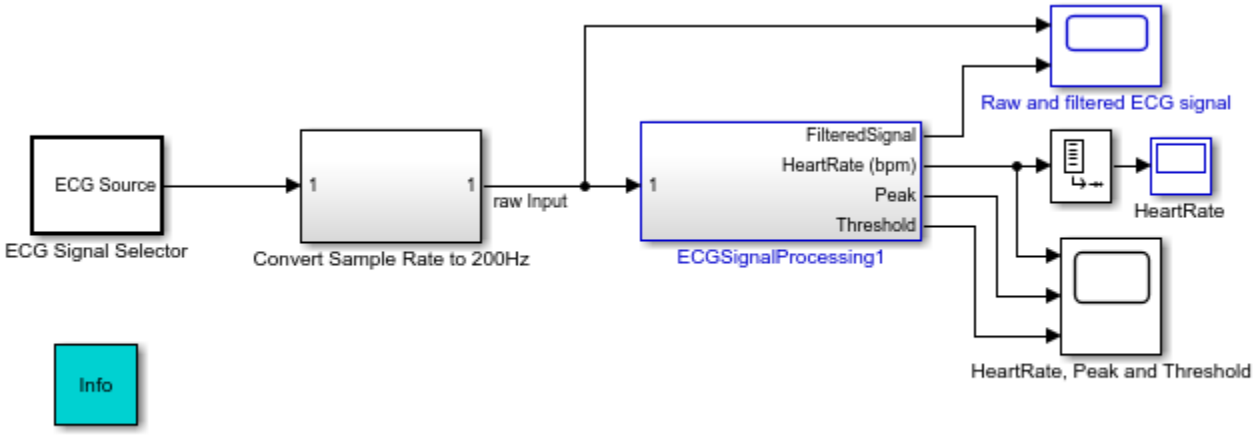




When the **Display time** parameter is set to 'Current', the model continues to display the current time, which is updated once every minute. You can change the **Display time** parameter of the Transmitted subsystem to 'User-defined' and specify any time you want to display.

Real-Time ECG QRS Detection

This example shows how to detect the QRS complex of electrocardiogram (ECG) signal in real-time. Model based design is used to assist in the development, testing and deployment of the algorithm.



Copyright 2015-2016 The MathWorks, Inc.

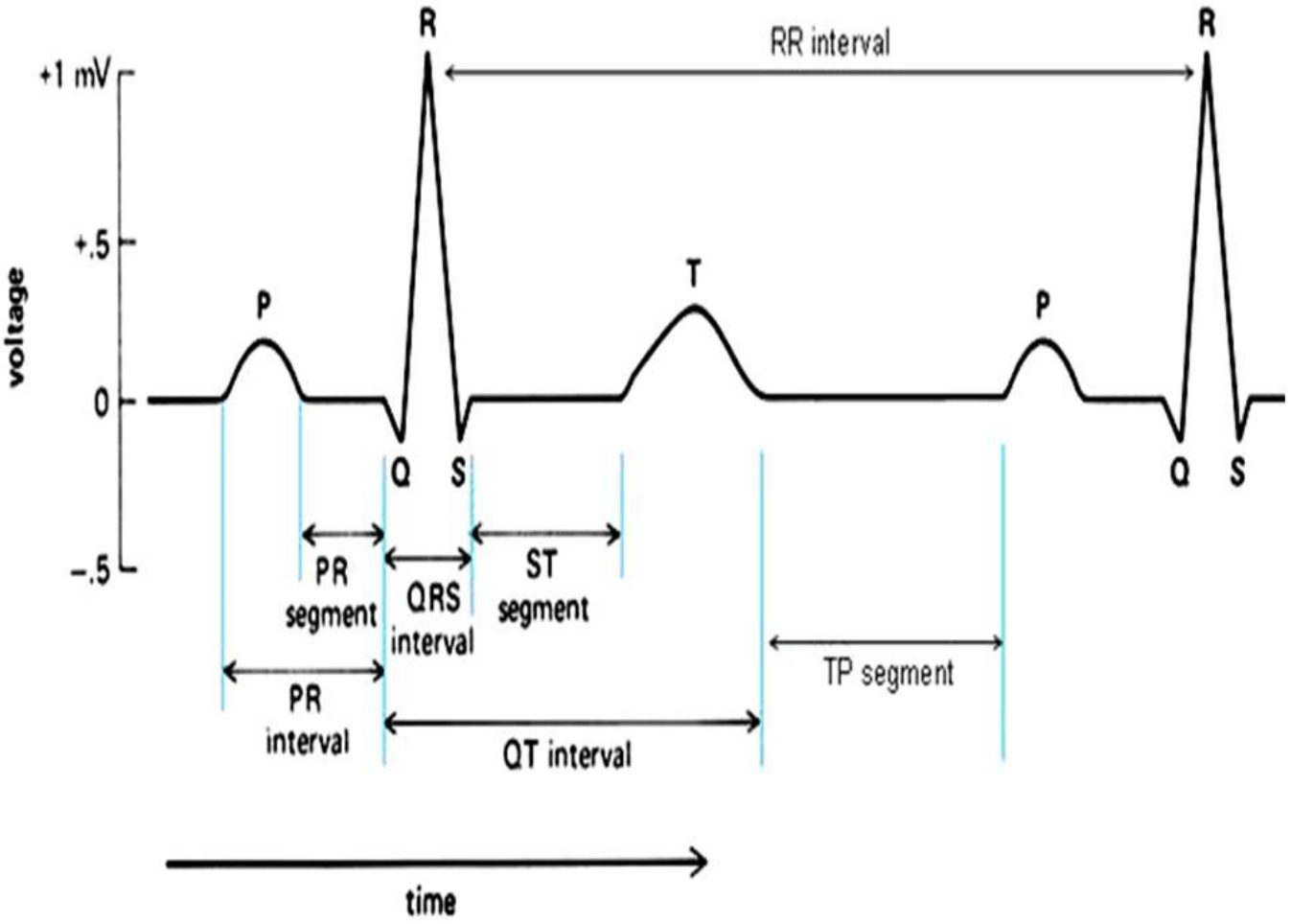
Introduction

The electrocardiogram (ECG) is a recording of body surface potentials generated by the electrical activity of the heart. Clinicians can evaluate an individual's cardiac condition and overall health from the ECG recording and perform further diagnosis.

A normal ECG waveform is illustrated in the following figure [1]. Because of the physiological variability of the QRS complex and various types of noise present in the real ECG signal, it is challenging to accurately detect the QRS complex.

The Noise sources that corrupt the raw ECG signals include:

- Baseline wander
- Power line interference (50 Hz or 60 Hz)
- Electromyographic (EMG) or muscle noise
- Artifacts due to electrode motion
- Electrode Contact Noise



ECG Signal Source

The ECG signals used in the development and testing of the biomedical signal processing algorithms are mainly from three sources: 1) Biomedical databases (e.g.,[2]) or other pre-recorded ECG data; 2) ECG simulator; 3) Real-time ECG data acquisition.

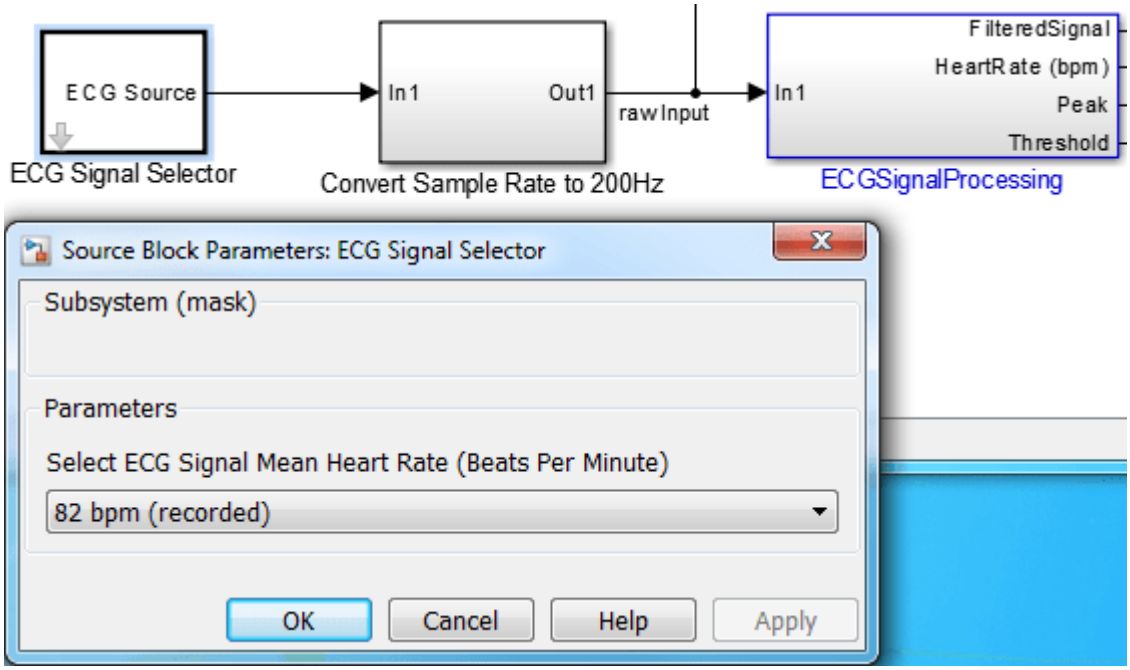
In this example, the following pre-recorded and simulated ECG signals are used. The signals all have sampling frequencies of 360 Hz.

- one set of recorded real ECG data sampled from a healthy volunteer with a mean heart rate of 82 beats per minute (bpm). This ECG data was pre-filtered and amplified by the analog front end before feeding it to the 12 bit ADC.
- four sets of synthesized ECG signals with different mean heart rates ranging from 45 bpm to 220 bpm. [3] is used to generate synthetic ECG signals in MATLAB.

Here are the settings for generating the synthesized ECG data:

- **Sampling frequency**: 360 Hz;
- **Additive uniformly distributed measurement noise**: 0.005 mV;

- **Standard deviation of heart rate** : 1 bpm.



ECG Signal Pre-processing and Filtering

A real-time QRS detection algorithm, which references [1, lab one], [4] and [5], is developed in Simulink with the assumption that the sampling frequency of the input ECG signal is always 200 Hz (or 200 samples/s). However, the recorded real ECG data may have different sampling frequencies ranging from 200 Hz to 1000 Hz, e.g., 360 Hz in this example. To bridge the different sampling frequencies, a sample rate converter block is used to convert the sample rate to 200 Hz. A buffer block is inserted to ensure the length of the input ECG signal is a multiple of the calculated decimation factor of the sample-rate converter block.

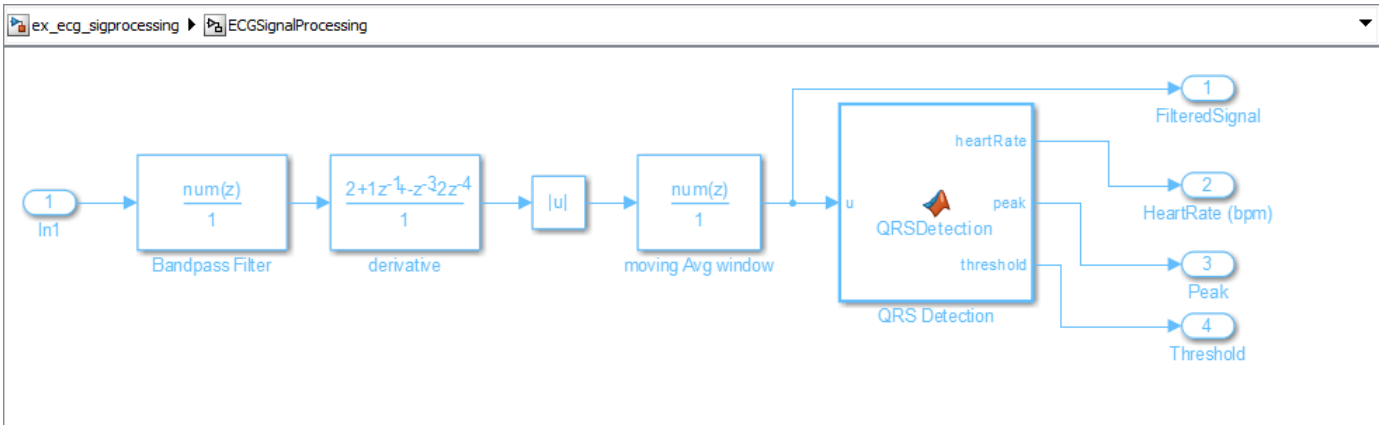
The ECG signal is filtered to generate a windowed estimate of the energy in the QRS frequency band. The filtering operation has these steps:

1. FIR Bandpass filter with a pass band from 5 to 26 Hz

2. Taking the derivative of the bandpass filtered signal

3. Taking the absolute value of the signal

4. Averaging the absolute value over an 80 ms window



Real-Time QRS Detection of ECG Signal

The QRS detection block detects peaks of the filtered ECG signal in real-time. The detection threshold is automatically adjusted based on the mean estimate of the average QRS peak and the average noise peak. The detected peak is classified as a QRS complex or as noise, depending on whether it is above the threshold.

The following QRS detection rules reference the PIC-based QRS detector implemented in [5].

Rule 1. Ignore all peaks that precede or follow larger peaks by less than 196 ms (306bpm).

Rule 2. If a peak occurs, check to see whether the raw signal contains both positive and negative slopes. If true, report a peak being found. Otherwise, the peak represents a baseline shift.

Rule 3. If the peak is larger than the detection threshold, classify it as a QRS complex. Otherwise classify it as noise.

Rule 4. If no QRS has been detected within 1.5 R-to-R intervals, but there is a peak that was larger than half the detection threshold, and that peak followed the preceding detection by at least 360ms, classify that peak as a QRS complex.

Simulate and Deploy

1. Open the example model.

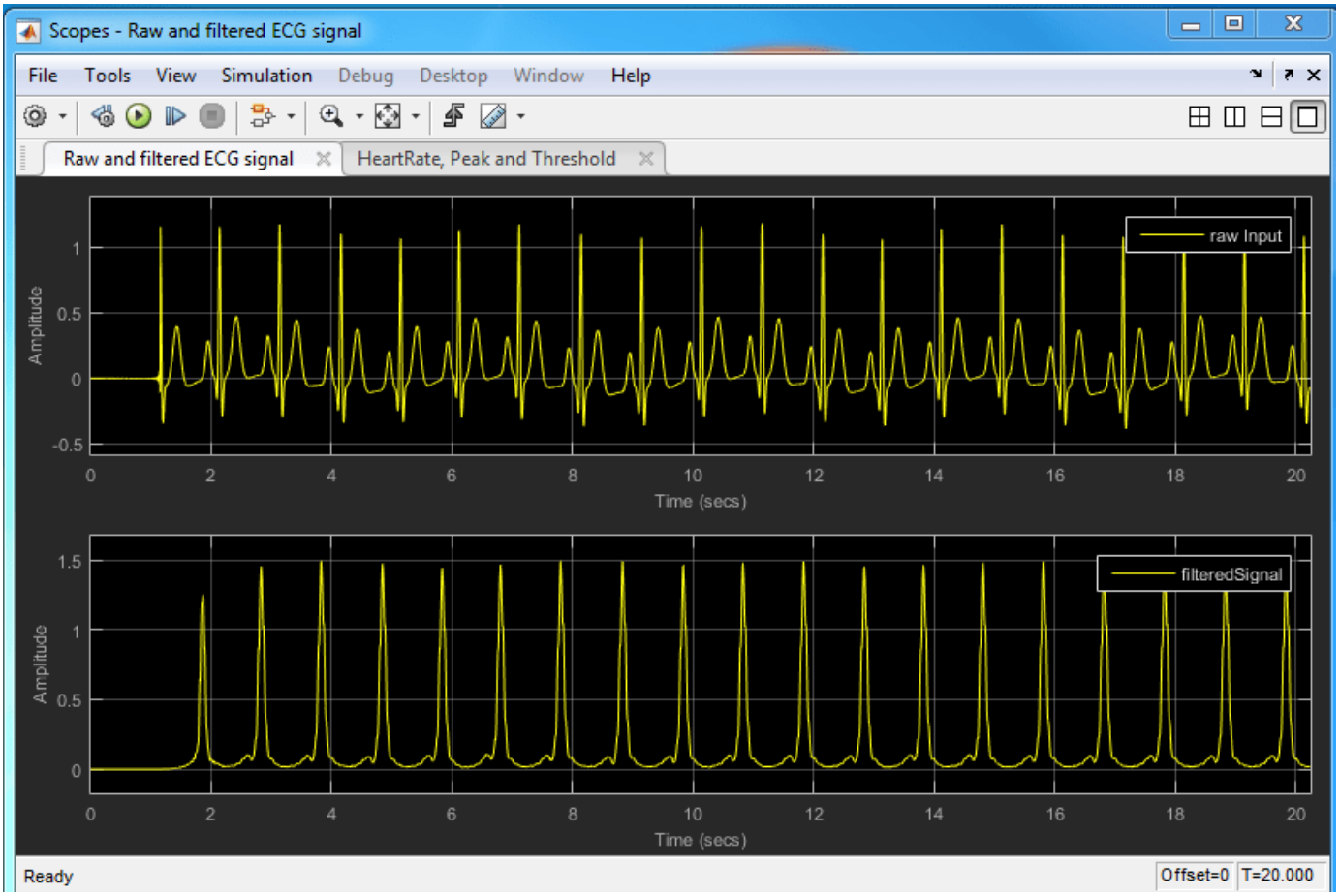
2. Change your current folder in MATLAB® to a writable folder.

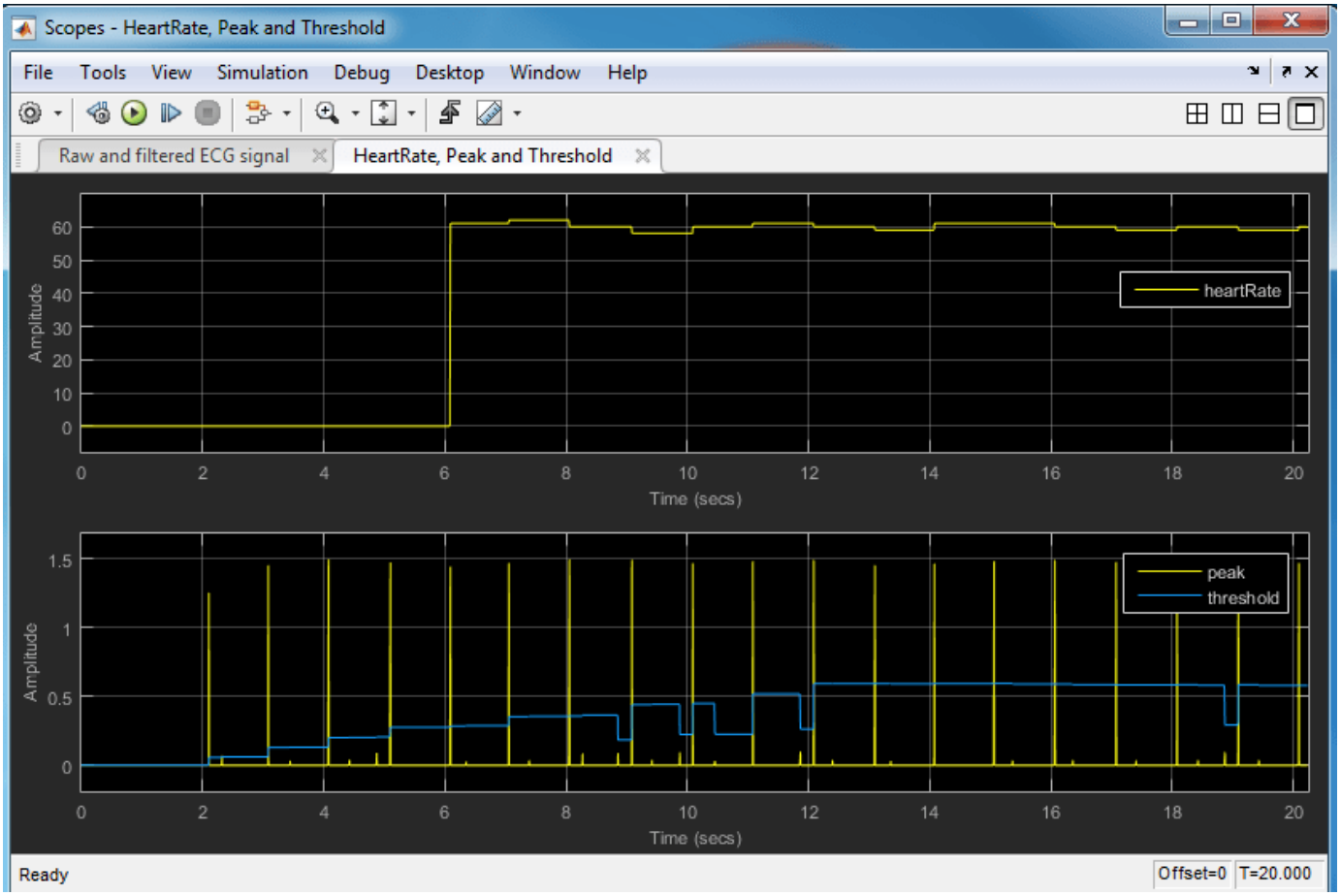
3. On the model tool strip, click **Run** to start the simulation. Observe the **HeartRate** display and the raw and filtered ECG signal in the scope, which also illustrates the updating of peaks, threshold and estimated mean heart rate.

4. Open the dialog of **ECG Signal Selector** block. Select the ECG signal mean heart rate in the drop down menu. Click **Apply** and observe the real-time detection results in the scopes and **HeartRate** display.

5. Click **Stop** to end simulation.

6. After selecting target hardware, you can generate code from the **ECGSignalProcessing** subsystem and deploy it to the target.


Scopes - Raw and filtered ECG signal
File Tools View Simulation Debug Desktop Window Help
Raw and filtered ECG signal
HeartRate, Peak and Threshold
raw Input
Amplitude
Time (secs)
filteredSignal
Amplitude
Time (secs)
Ready
Offset=0
T=20.000



References

[1] https://ocw.mit.edu/courses/health-sciences-and-technology/hst-582j-biomedical-signal-and-image-processing-spring-2007/index.htm

[2] https://www.physionet.org/physiobank/database/mitdb/

[3] https://www.physionet.org/physiotools/ecgsyn/

[4] J. Pan and W. Tompkins, A Real-Time QRS Detection Algorithm, IEEE Transactions on Biomedical Engineering, 32(3): 230-236, March 1985

[5] Patrick S. Hamilton, EP Limited: Open Source ECG Analysis Software, 2002

Internet Low Bitrate Codec (iLBC) for VoIP

This example implements the Internet Low Bitrate Codec (iLBC) and illustrates its use. iLBC is designed for encoding and decoding speech for transmission via VoIP (Voice Over Internet Protocol).

VoIP

Voice over Internet Protocol is the family of technologies that allows IP networks to be used for voice applications such as telephony and teleconferencing. Compression is normally required to reduce the bandwidth requirements of these applications. For efficiency, VoIP is often implemented using the lightweight but unreliable User Datagram Protocol (UDP). Packet loss correction is needed to maintain received voice quality over lossy networks.

Basic iLBC Design and Performance

iLBC is designed for compression of speech to be transmitted over the Internet. Thus, its algorithms are only meant to cover the narrow frequency range of 90-4000 Hz and it implements perceptual coding tuned to normal speech. All input signals to the iLBC encoder must be Pulse Code Modulated (PCM) speech signals sampled at exactly 8000 Hz with 16-bit samples ranging from -32768 to +32767.

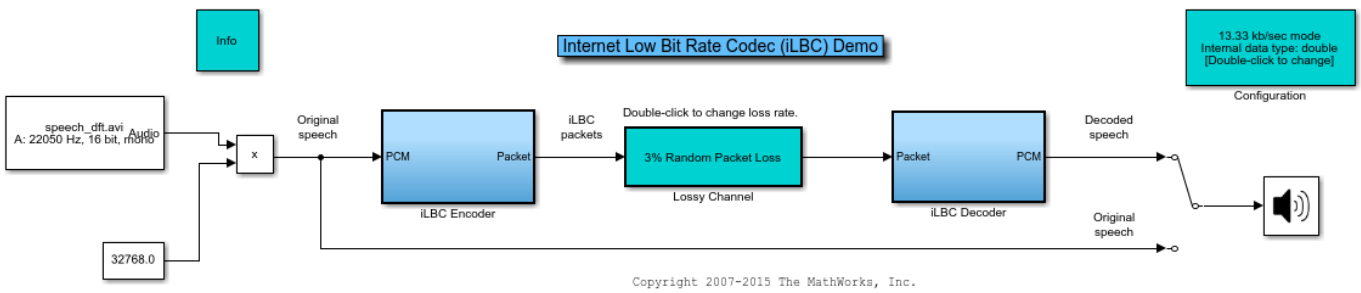
iLBC is defined for two different transmission rates, with a packet of data being encoded either after every 30ms or after every 20ms of speech. The advantage of encoding every 30ms is that the encoded data rate is lower: 13.33 kbit/sec as opposed to 15.20 kbit/sec for 20ms frames. However, encoding every 30ms leads to 50% more delay in the received speech, which can cause undesirable latency.

Since all inputs to iLBC must be 8000 Hz, 16-bit PCM speech, the input rate is (8000 Hz) * (16 bits) = 128 kbit/sec. Thus, iLBC compresses the speech to 10.4% and 11.9% of the original data-rate for 13.33 kbit/sec and 15.20 kbit/sec modes, respectively.

In addition to encoding to low data transmission rates, iLBC provides a framework for easily implementing Packet Loss Correction (PLC) systems. The codec is meant for real-time speech over the Internet, but the Internet is subject to random delays in routing information in real-time, which renders many packets useless to the iLBC decoder. The job of a PLC is to interpolate the speech for missing packets based on the packets before and immediately after the missing one. Though iLBC does not define a specific PLC algorithm, this example implements a simple PLC for illustration.

The iLBC Example Model

The model shown below reads in a speech signal and, after passing through iLBC, plays the output with the default audio device.



Using the iLBC Example Model

The top level of this example model consists of just a handful of simple blocks. The basic operation is to load a speech signal and pass it to the iLBC Encoder block to convert it to a stream of iLBC packets. Next, the packets are sent through a simulated lossy channel, which causes random packets to be set to all zeros. Finally, the packets are sent to the iLBC Decoder block to be converted back into a speech signal, which is then played. In addition, there is a manual switch that can be toggled as the model runs to compare the original speech signal with the decoded signal.

Double clicking on the configuration block in the upper right corner of the model brings up a dialog, where it is possible to change the data transmission rate to one of the two iLBC modes (13.33 kbit/sec or 15.20 kbit/sec). The decoder's transmission rate must be set to the same as the encoder, or else an error will occur. In addition, the user may specify whether to use double or single precision for all internal calculations in the encoder and decoder.

Double clicking on the Lossy Channel subsystem brings up a dialog that allows the percentage of lost packets to be set. The iLBC Decoder's Packet Loss Concealment algorithm is tuned to correct for 0-10% packet loss. Packet loss rates higher than 10% will be easily audible.

The iLBC encoder and decoder blocks are implemented as subsystems in this model. In order to accommodate a level of reuse, they also make use of a example library, which can be found at dspilbclib. This library contains four helper blocks used by the encoder and decoder. Feel free to open the library and look under the blocks to see how iLBC was implemented in Simulink®.

Filter Analysis, Design, and Implementation

- “Design a Filter in Fdesign — Process Overview” on page 5-2
- “Use Filter Designer with DSP System Toolbox Software” on page 5-9
- “Digital Frequency Transformations” on page 5-53
- “Digital Filter Design Block” on page 5-76
- “Filter Realization Wizard” on page 5-83
- “Digital Filter Implementations” on page 5-93
- “Removing High-Frequency Noise from an ECG Signal” on page 5-101
- “Minimax FIR Filter Design” on page 5-103

Design a Filter in Fdesign — Process Overview

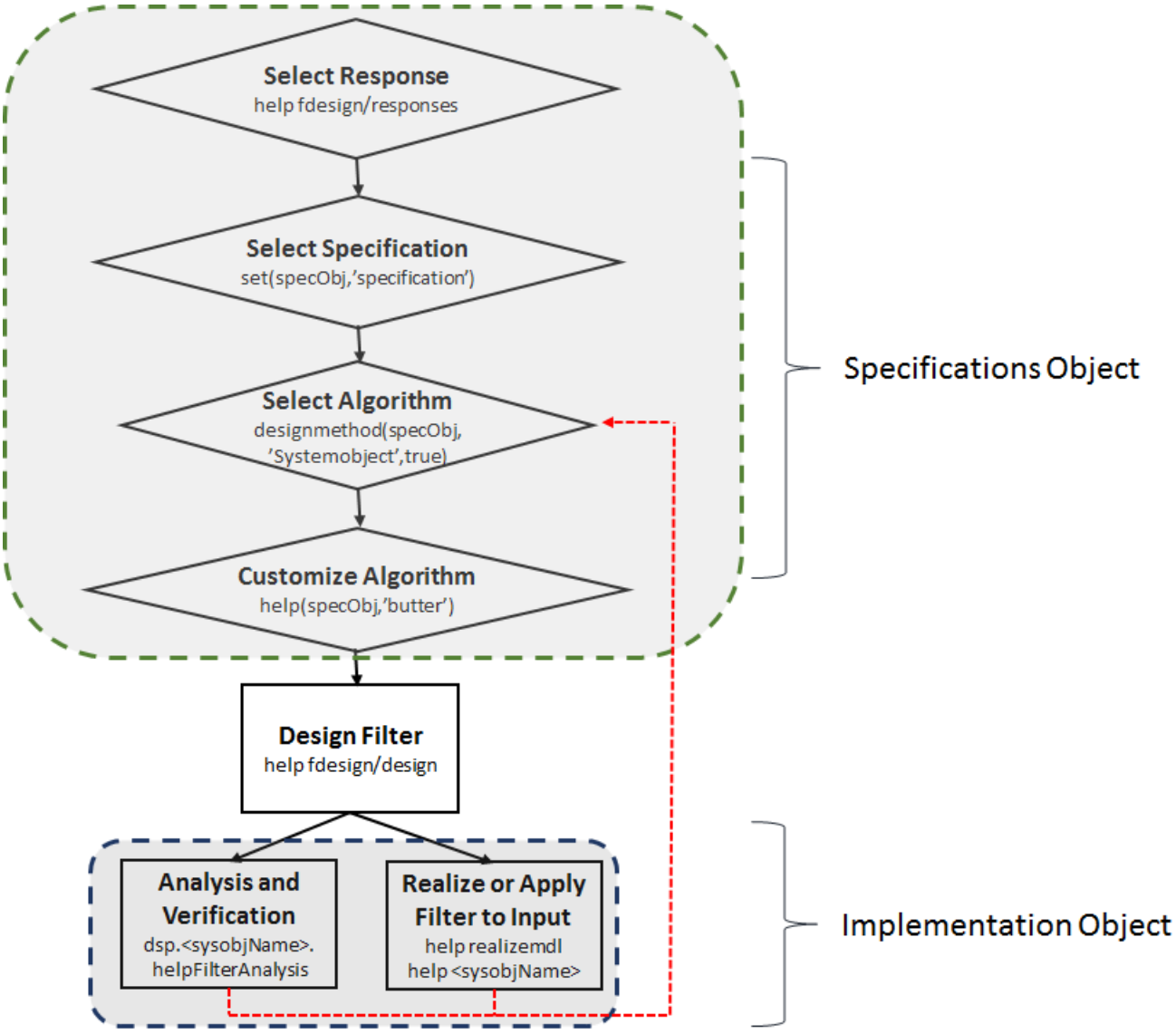
Process Flow Diagram and Filter Design Methodology

- “Exploring the Process Flow Diagram” on page 5-2
- “Select a Response” on page 5-4
- “Select a Specification” on page 5-4
- “Select an Algorithm” on page 5-5
- “Customize the Algorithm” on page 5-6
- “Design the Filter” on page 5-7
- “Design Analysis” on page 5-7
- “Realize or Apply the Filter to Input Data” on page 5-8

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 5-4, 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 5-5, but also on the specification selected in “Select a Specification” on page 5-4. 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 5-7, 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 5-5, 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 5-9
“Access the Quantization Features of Filter Designer” on page 5-11
“Quantize Filters in Filter Designer” on page 5-13
“Analyze Filters with a Noise-Based Method” on page 5-18
“Scale Second-Order Section Filters” on page 5-22
“Reorder the Sections of Second-Order Section Filters” on page 5-25
“View SOS Filter Sections” on page 5-28
“Import and Export Quantized Filters” on page 5-32
“Generate MATLAB Code” on page 5-35
“Import XILINX Coefficient (.COE) Files” on page 5-35
“Transform Filters Using Filter Designer” on page 5-36
“Design Multirate Filters in Filter Designer” on page 5-42
“Realize Filters as Simulink Subsystem Blocks” on page 5-50

Design Advanced Filters in Filter Designer

- “Overview of Filter Designer Features” on page 5-9
- “Use Filter Designer with DSP System Toolbox Software” on page 5-10
- “Design a Notch Filter” on page 5-10

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

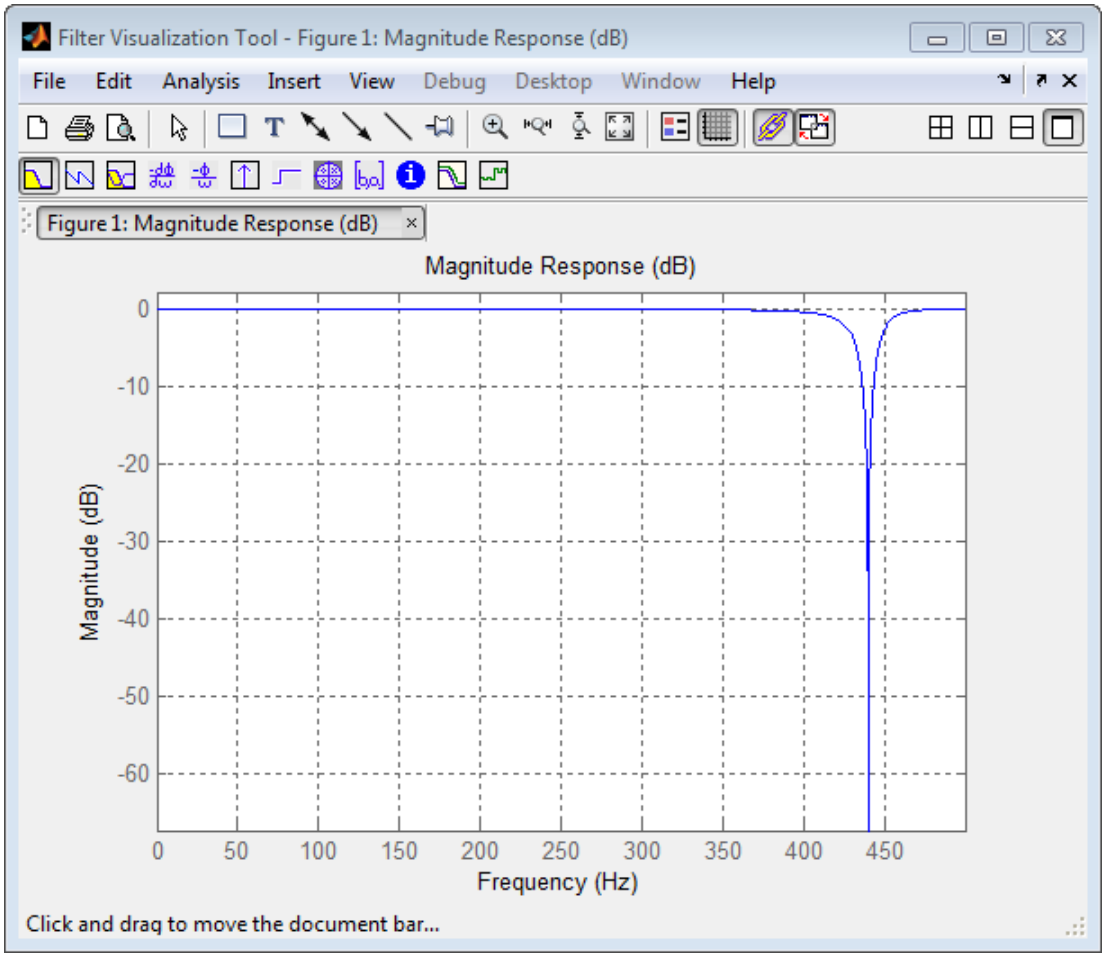
Here is how you design a notch filter that removes concert A (440 Hz) from an input musical signal spectrum.

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.

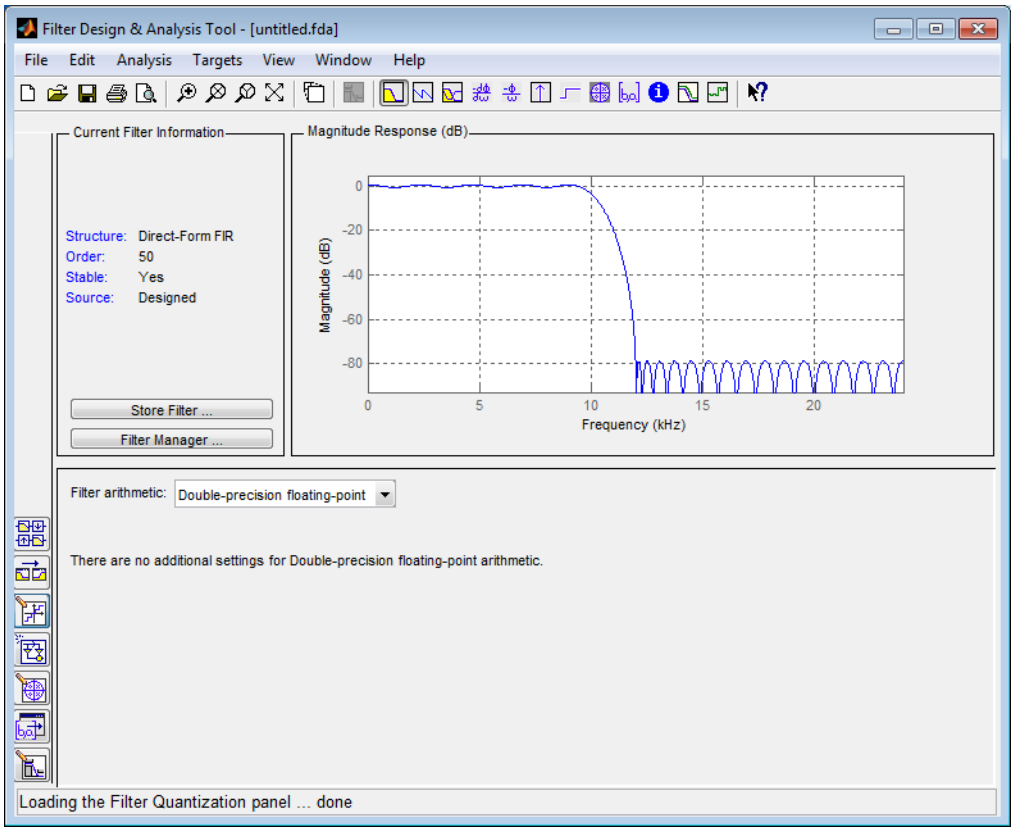
Access the Quantization Features of Filter Designer

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 5-13
- "Coefficients Options" on page 5-13
- "Input/Output Options" on page 5-14
- "Filter Internals Options" on page 5-15
- "Filter Internals Options for CIC Filters" on page 5-17

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 5-18
- “Compare the Estimated and Theoretical Magnitude Responses” on page 5-21
- “Select Quantized Filter Structures” on page 5-21
- “Convert the Structure of a Quantized Filter” on page 5-21
- “Convert Filters to Second-Order Sections Form” on page 5-22

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**.

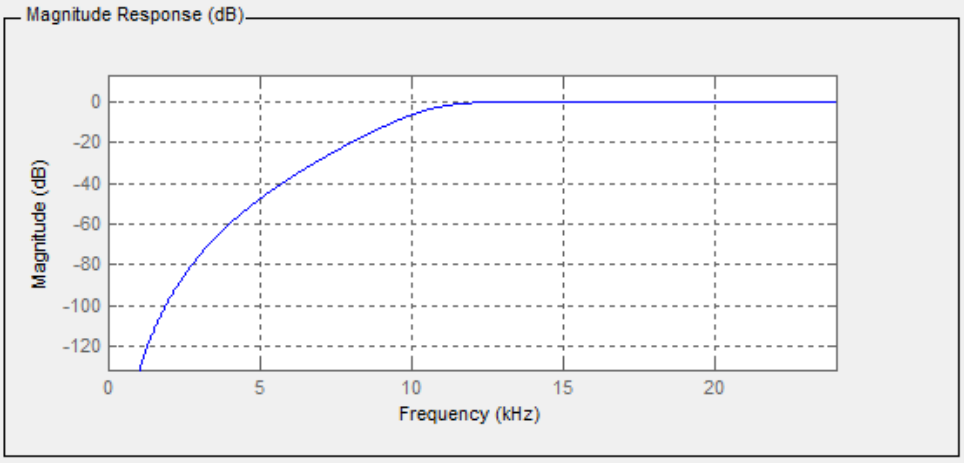
   In filter designer, the analysis area changes to display the magnitude response for your filter.
6. To generate the quantized version of your filter, using default quantizer settings, click on the side bar.

   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 5-19, 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 5-18.

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 24-14.

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

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 24-15. 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 Structure**. 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 of Second-Order Sections Dialog Box” on page 5-22
- “Scale an SOS Filter” on page 5-23

Use the Reordering and Scaling of 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 of 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.
Less Overflow — Highest 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.
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.

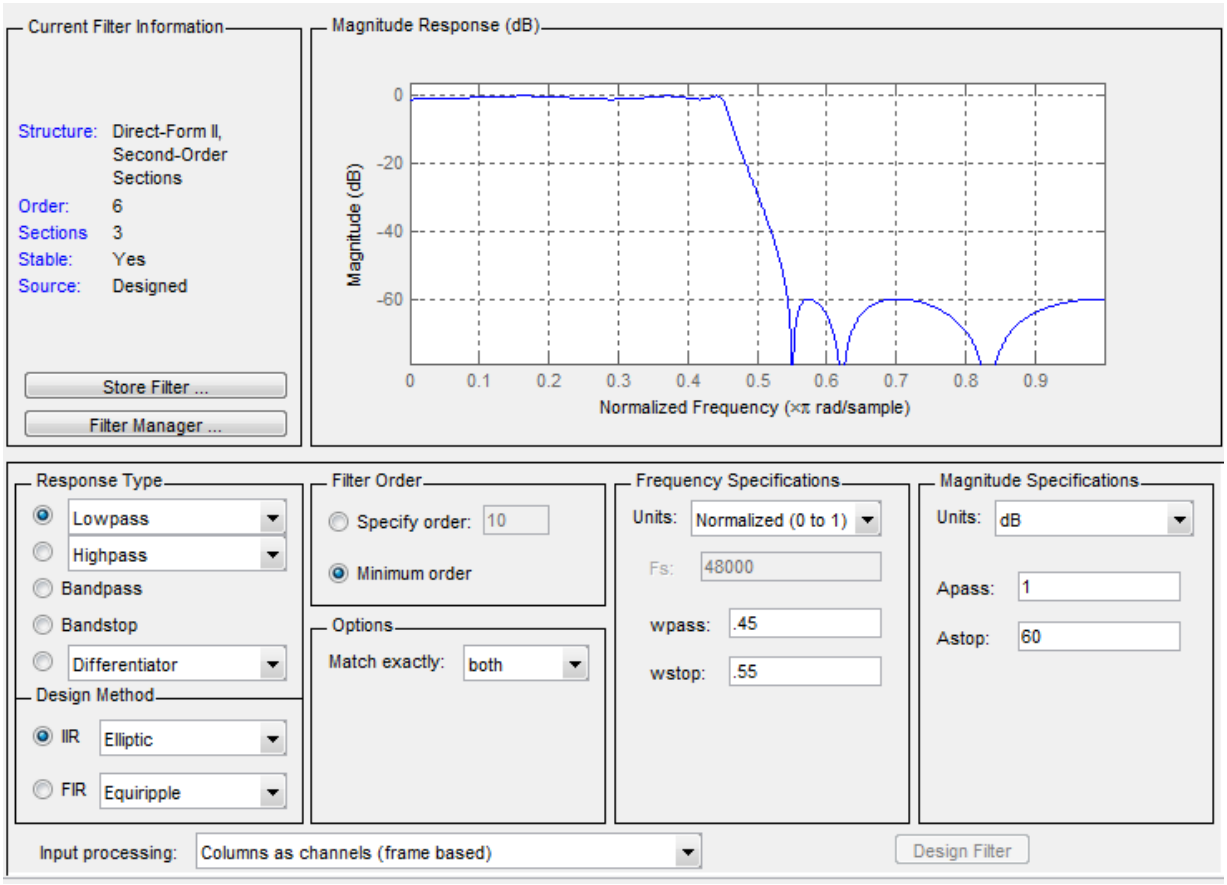
Scale an SOS Filter

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 **Less Overflow—Highest SNR** slider from **Less Overflow** to **Highest 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.

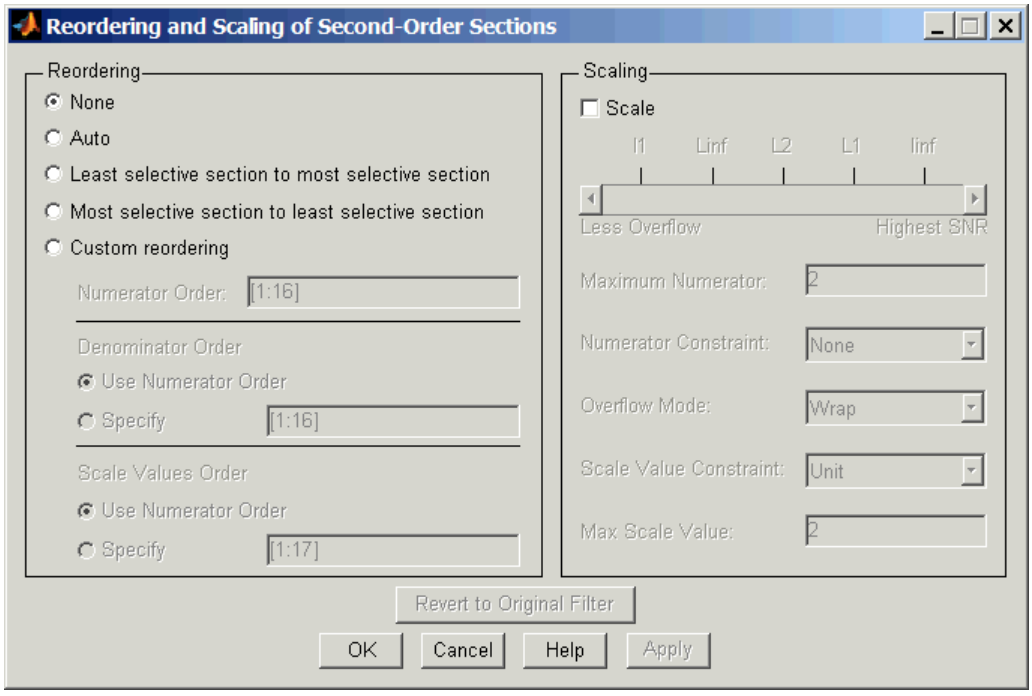
Reorder the Sections of Second-Order Section Filters

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 5-22 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.

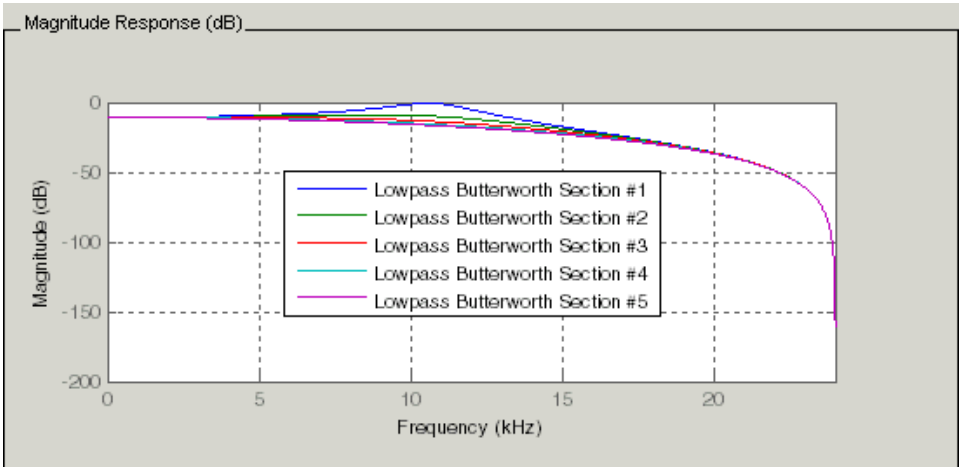
Now you are ready to reorder the sections of your filter. Note that filter designer performs the reordering on the current filter in the session.

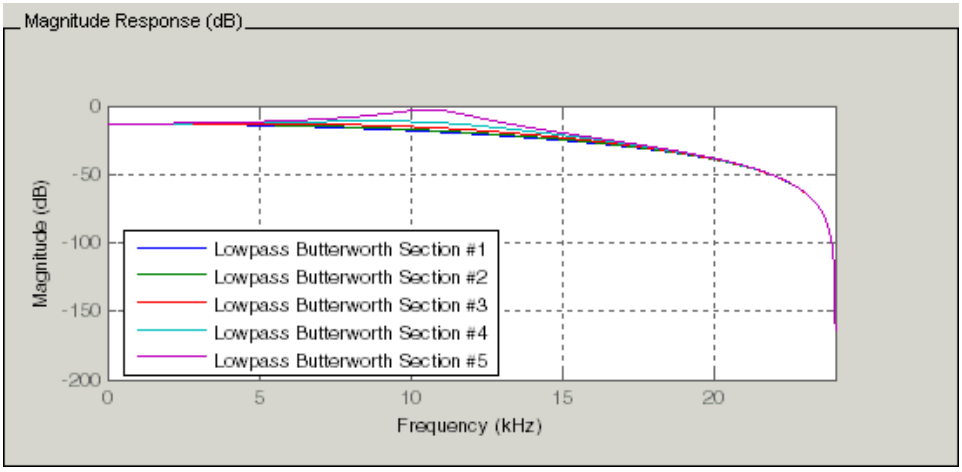
Use Least Selective to Most Selective Section Reordering

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

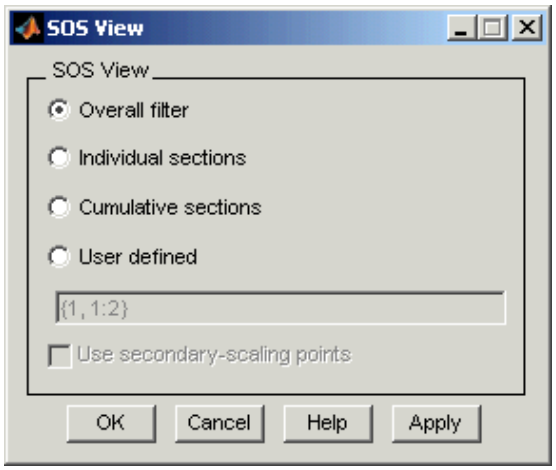
- “Using the SOS View Dialog Box” on page 5-28
- “View the Sections of SOS Filters” on page 5-30

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

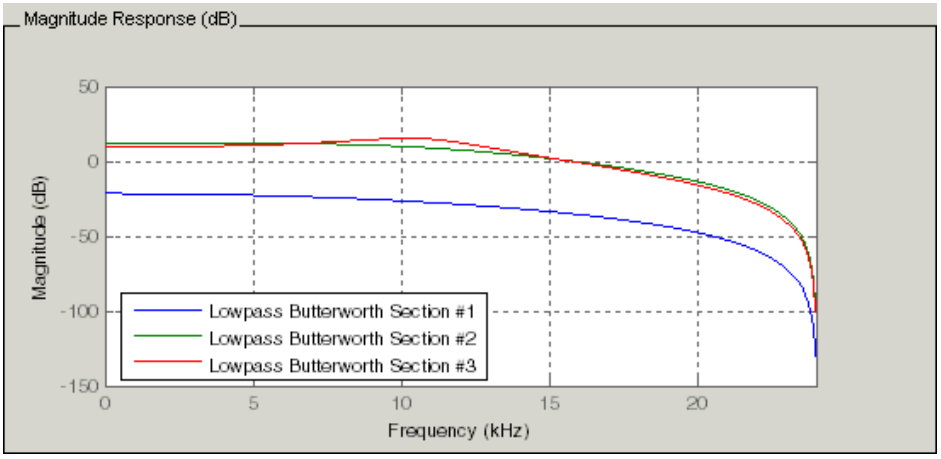
After you design or import an SOS filter in to filter designer, the SOS view option lets you see the per section performance of your filter. Enabling SOS View from the View menu in filter designer configures the tool to display the sections of SOS filters whenever the current filter is an SOS filter.

These next steps demonstrate using SOS View to see your filter sections displayed in filter designer.

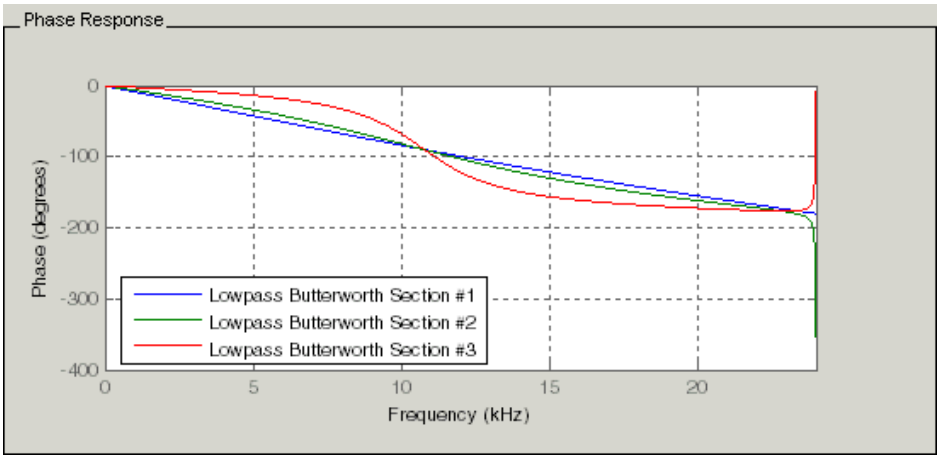
1. Launch filter designer.
2. Create a lowpass SOS filter using the Butterworth design method. Specify the filter order to be 6. Using a low order filter makes seeing the sections more clear.
3. Design your new filter by clicking **Design Filter**.

   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.

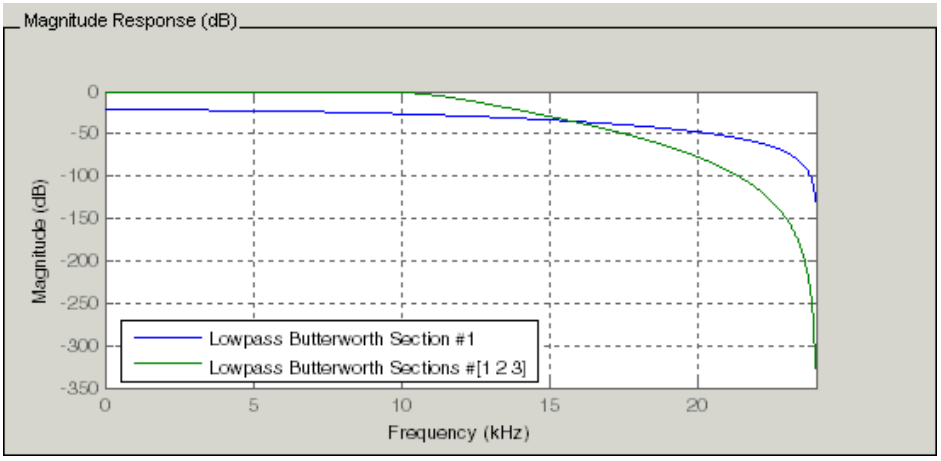


7 To define your own display of the sections, you use the **User defined** option and enter a vector of section indices to display. Now you see a display of the first section response, and the cumulative first, second, and third sections response:

- 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 5-32
- “Import Quantized Filters” on page 5-33
- “To Export Quantized Filters” on page 5-34

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 24-24, and “Exporting a Filter Design” on page 24-16.

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

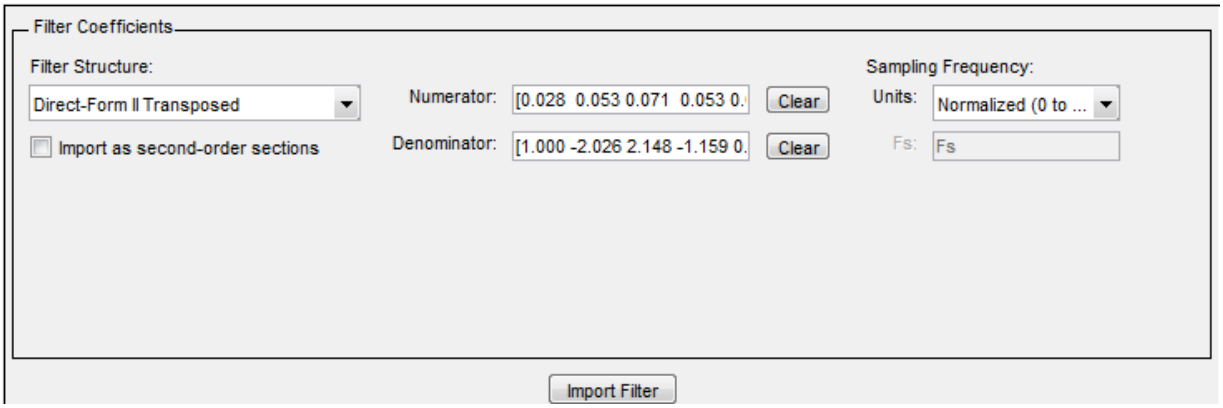
Import Quantized Filters

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 5-34
- Export to a text file on page 5-34
- Export to a MAT-file on page 5-35

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 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 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 5-36
- "Original Filter Type" on page 5-37
- "Frequency Point to Transform" on page 5-39
- "Transformed Filter Type" on page 5-40
- "Specify Desired Frequency Location" on page 5-40

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.

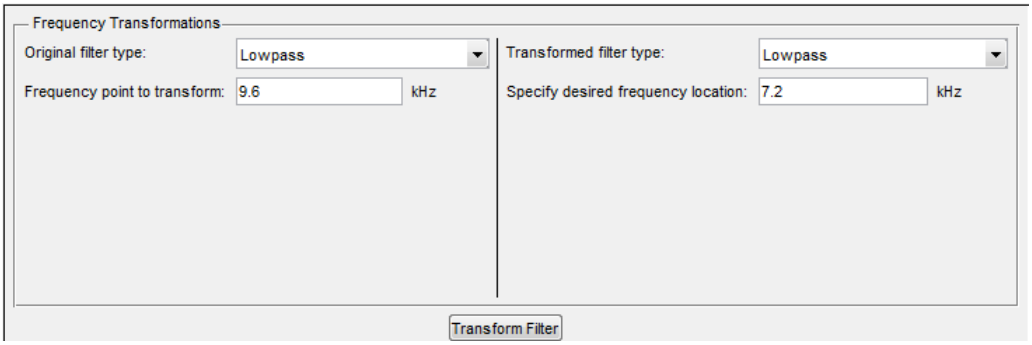
You can convert your FIR filters from:

- 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 5-58 and “Frequency Transformations for Complex Filters” on page 5-67.

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 5-53.

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 5-58 and “Frequency Transformations for Complex Filters” on page 5-67.

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 5-53.

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.

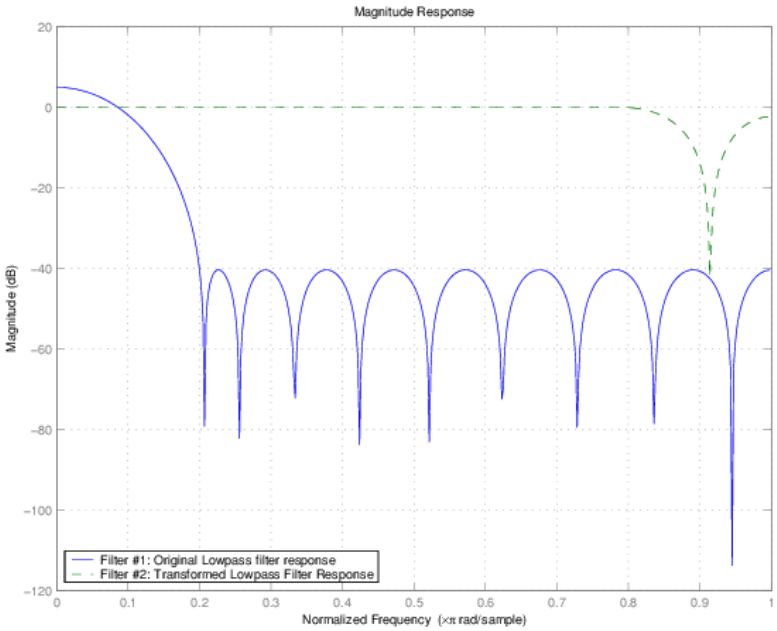
5 From the **Transformed filter type** list, select the type of filter you want to transform to.

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.

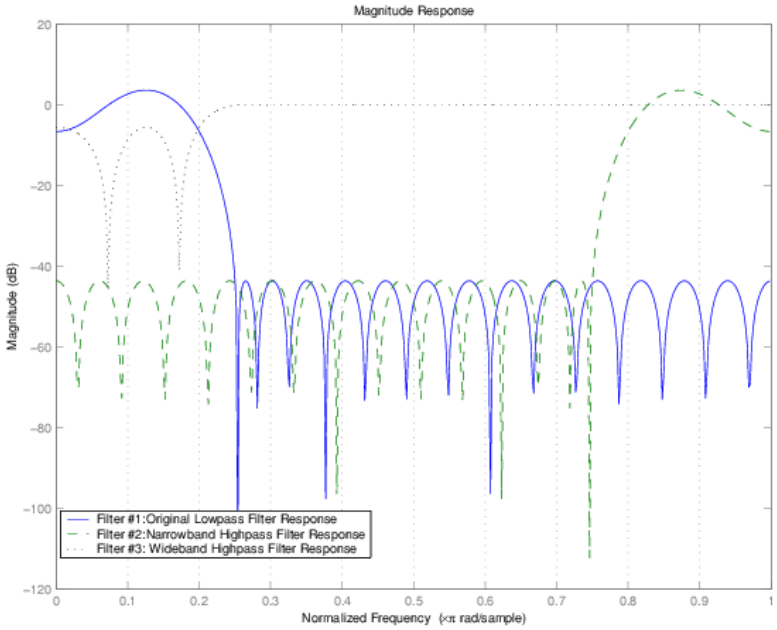
After you click **Transform Filter**, filter designer transforms your filter, displays the magnitude response of your new filter, and updates the **Current Filter Information** to show you that your filter has been transformed. In the filter information, the **Source** is **Transformed**.

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 5-42
- “Switch Filter Designer to Multirate Filter Design Mode” on page 5-42
- “Controls on the Multirate Design Panel” on page 5-43
- “Quantize Multirate Filters” on page 5-48
- “Export Individual Phase Coefficients of a Polyphase Filter to the Workspace” on page 5-50

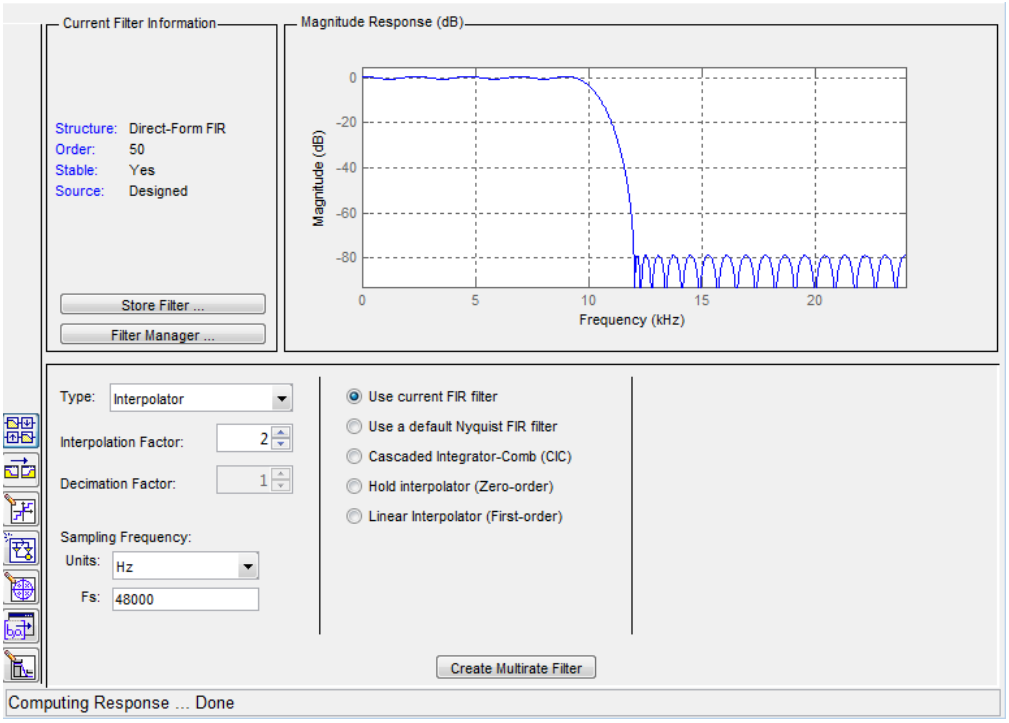
Introduction

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 a 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**.

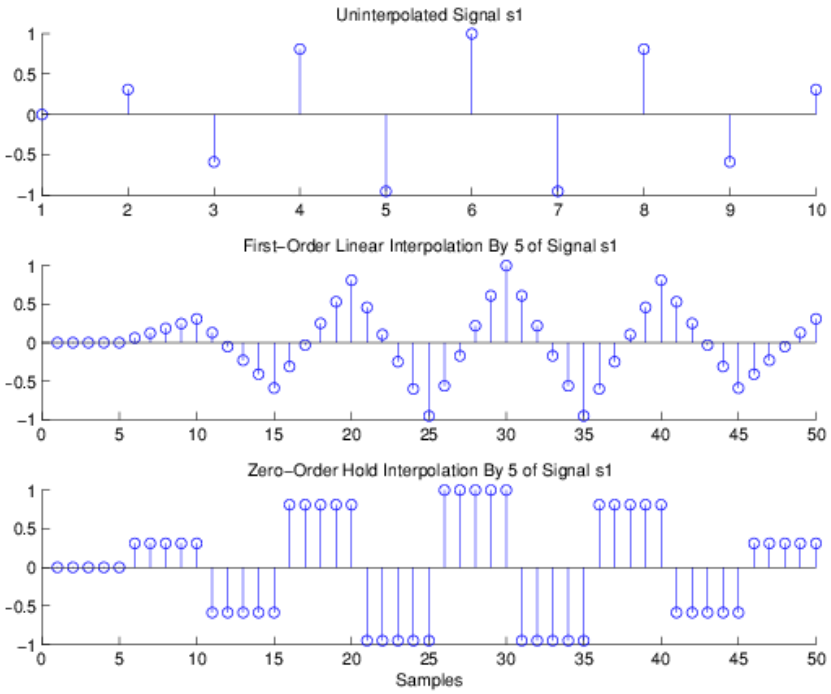
Design Your Filter

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 FIR 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

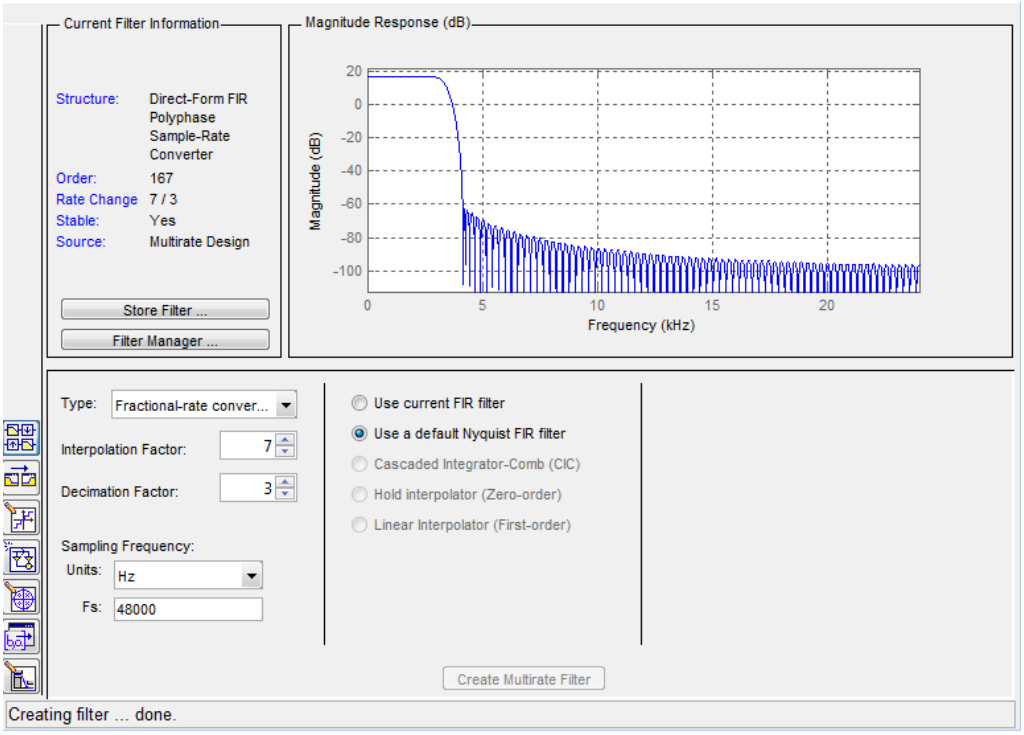
To introduce the process you use to design a multirate filter in filter designer, this example uses the options to design a fractional rate convertor which uses 7/3 as the fractional rate. Begin the design by creating a default lowpass FIR filter in filter designer. You do not have to begin with this 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 5-32.

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.

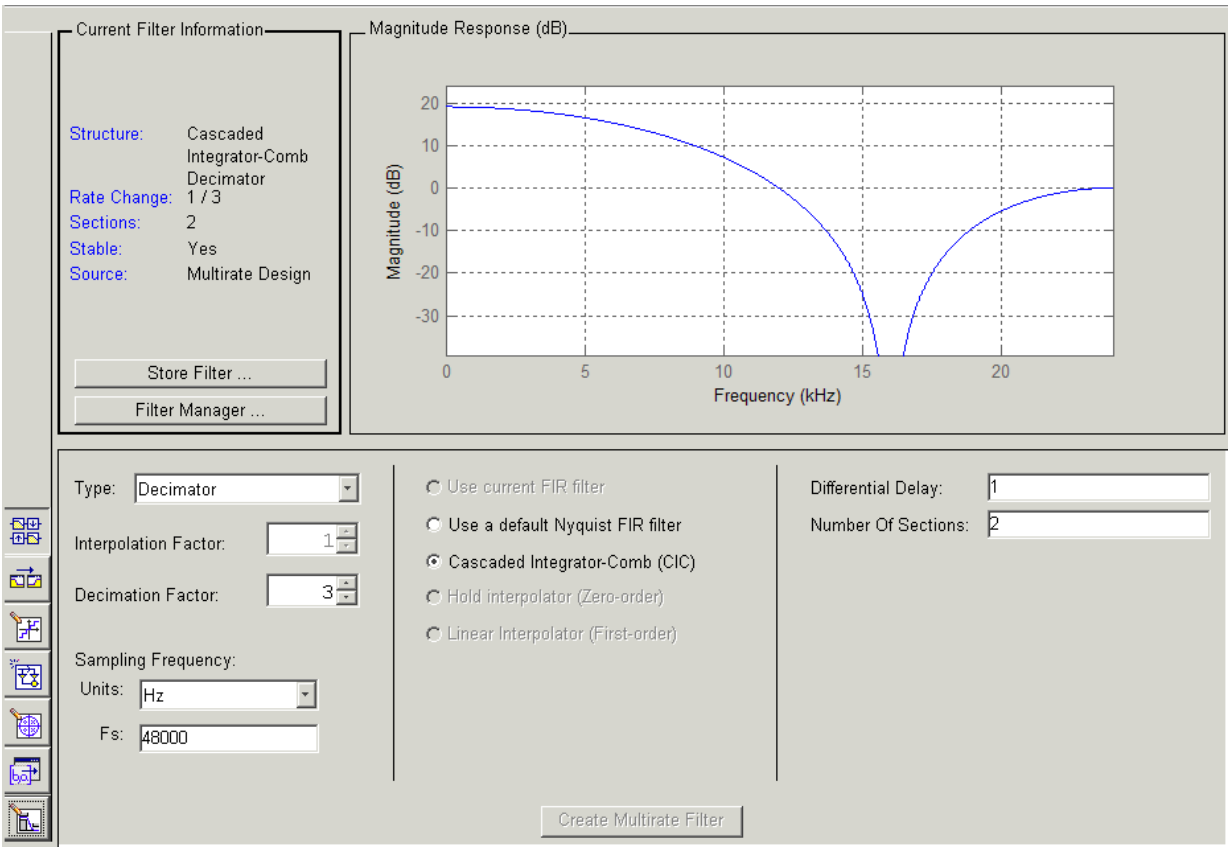
5 Set Differential Delay to 2. Generally, 1 or 2 are good values to use.

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 5-50
- “About the Realize Model Panel in Filter Designer” on page 5-51

Introduction

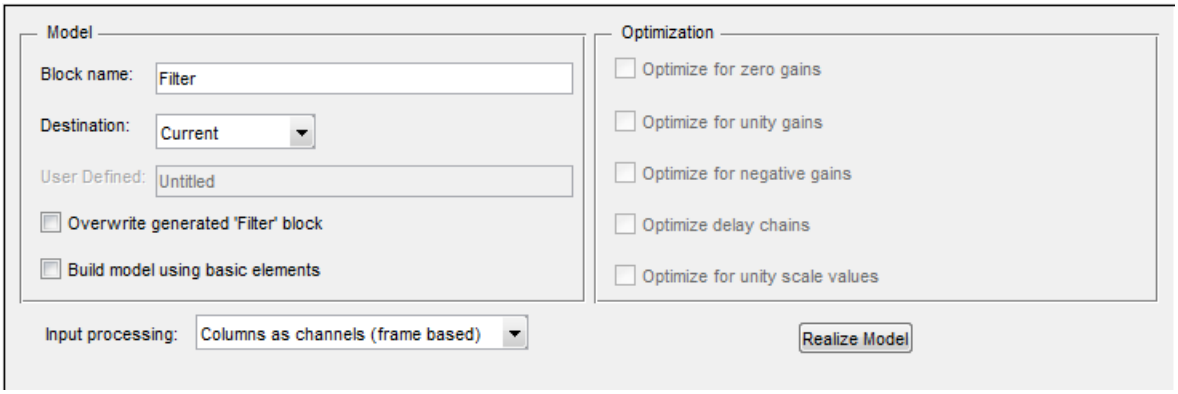
After you design or import a filter in filter designer, the realize model feature lets you create a Simulink subsystem block that implements your filter. The generated filter subsystem block uses either digital filter blocks from the DSP System Toolbox library, or the Delay, Gain, and Sum blocks in

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 25-2
- “Using Filter Designer” on page 24-2

Digital Frequency Transformations

In this section...
“Details and Methodology” on page 5-53
“Frequency Transformations for Real Filters” on page 5-58
“Frequency Transformations for Complex Filters” on page 5-67

Details and Methodology

- “Overview of Transformations” on page 5-53
- “Select Features Subject to Transformation” on page 5-55
- “Mapping from Prototype Filter to Target Filter” on page 5-57
- “Summary of Frequency Transformations” on page 5-58

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

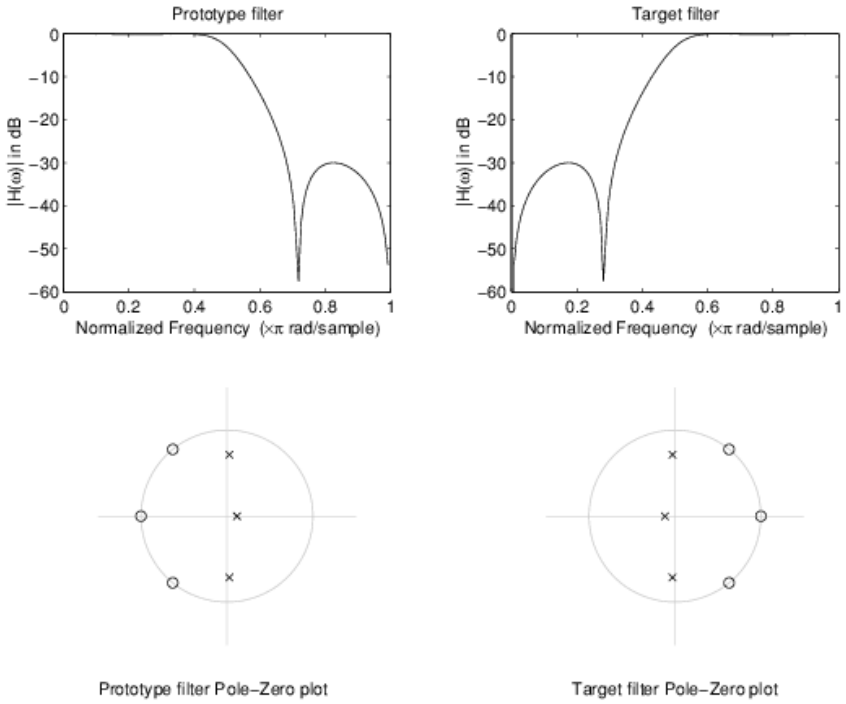
$H_A(z)$ — Transfer function of the allpass mapping filter

$H_T(z)$ — 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.



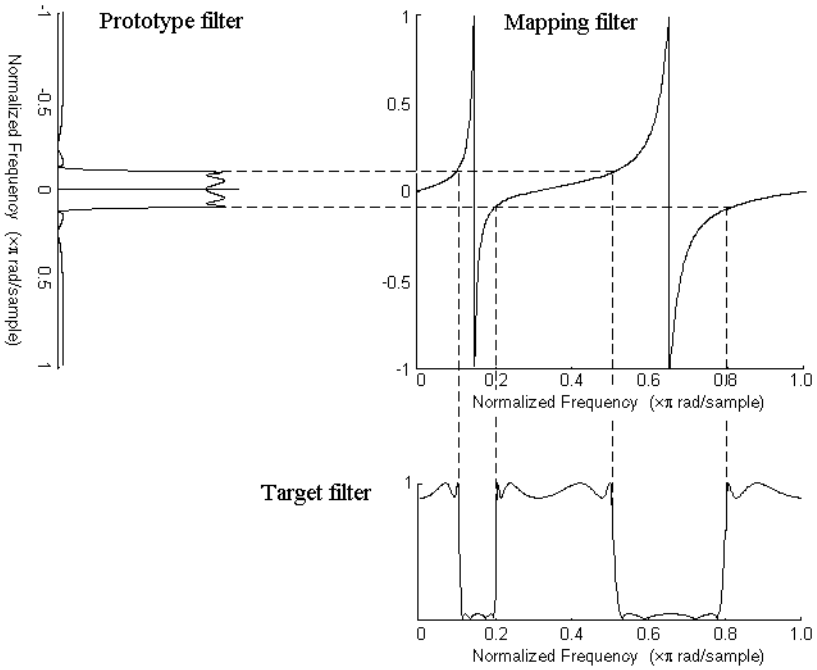
Example of a Simple Mirror Transformation

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

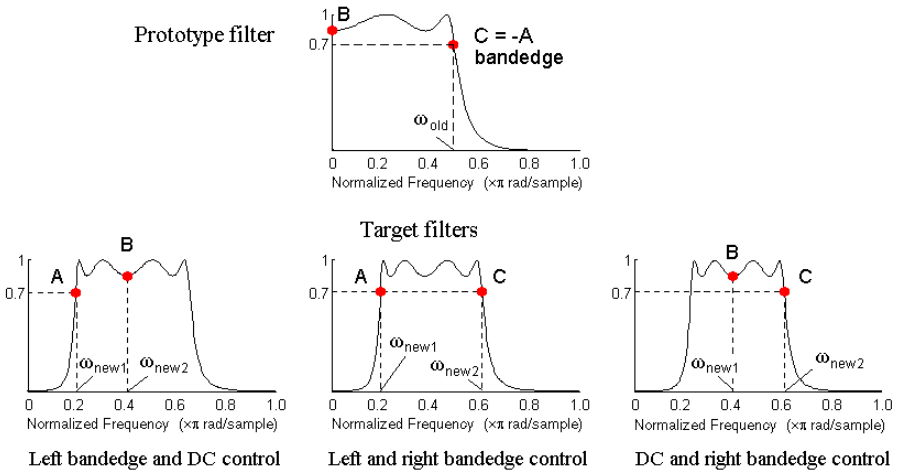
Most of the frequency transformations are based on the second-order allpass mapping filter:

$$H_A(z) = \pm \frac{1 + \alpha_1 z^{-1} + \alpha_2 z^{-2}}{\alpha_2 + \alpha_1 z^{-1} + z^{-2}}$$

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).

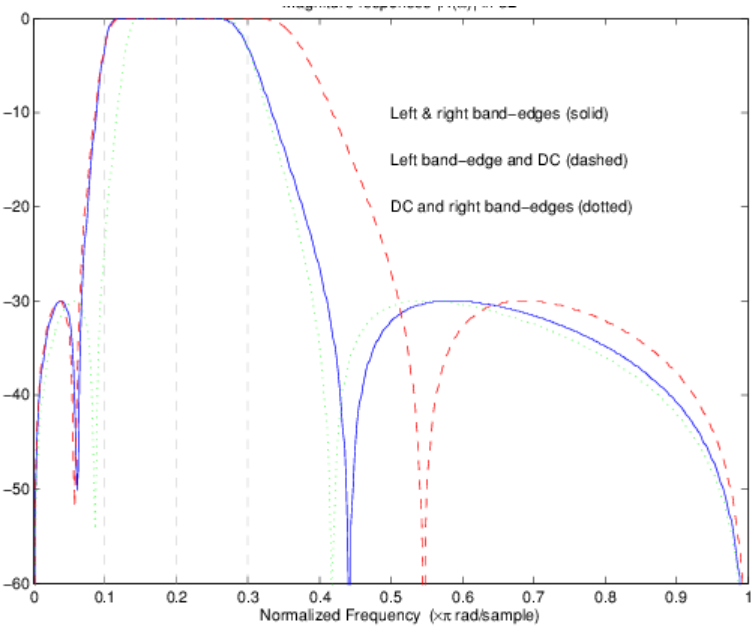
Select Features Subject to Transformation

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 5-53. 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 5-59
- “Real Frequency Shift” on page 5-59
- “Real Lowpass to Real Lowpass” on page 5-60
- “Real Lowpass to Real Highpass” on page 5-61
- “Real Lowpass to Real Bandpass” on page 5-62
- “Real Lowpass to Real Bandstop” on page 5-63
- “Real Lowpass to Real Multiband” on page 5-64
- “Real Lowpass to Real Multipoint” on page 5-66

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

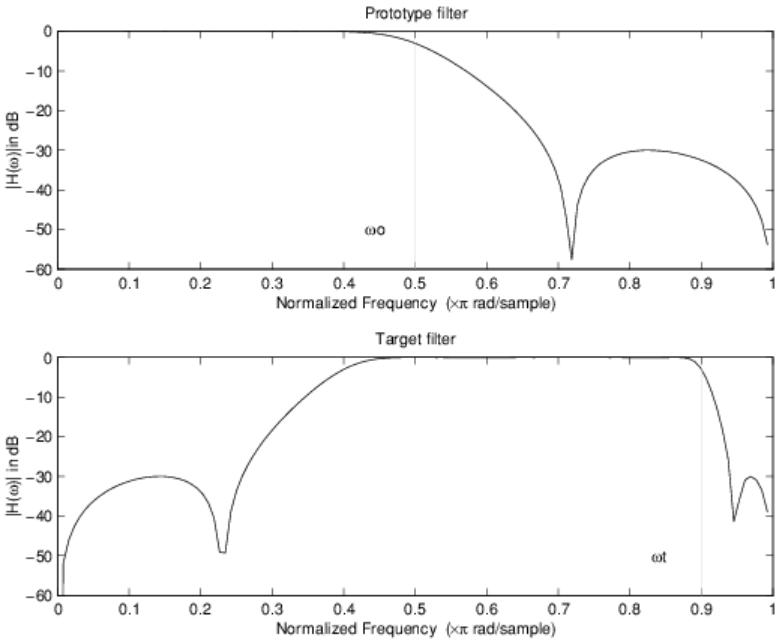
The following example shows how this transformation can be used to move the response of the prototype lowpass filter in either direction. Please note that because the target filter must also be real, the response of the target filter will inherently be disturbed at frequencies close to Nyquist and close 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

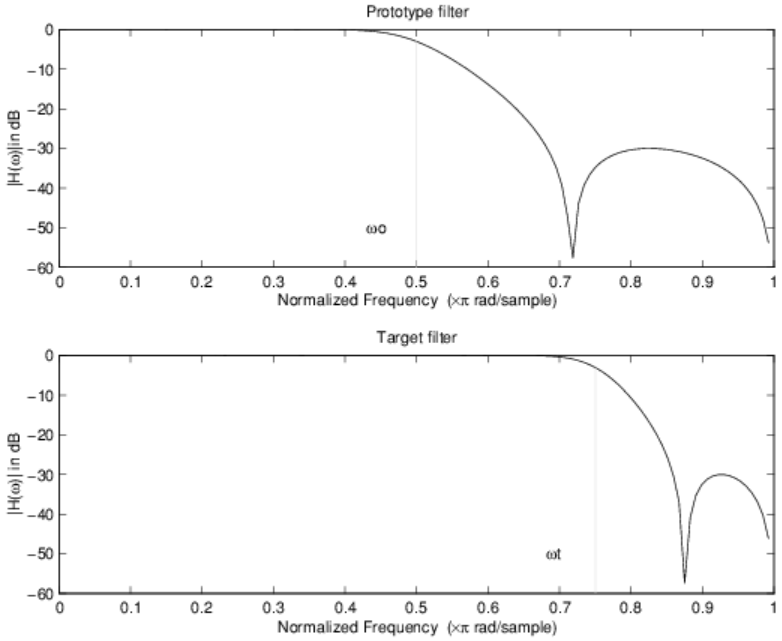
The example below shows how to modify the cutoff frequency of the prototype filter. The MATLAB code for this example is shown in the following figure.

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

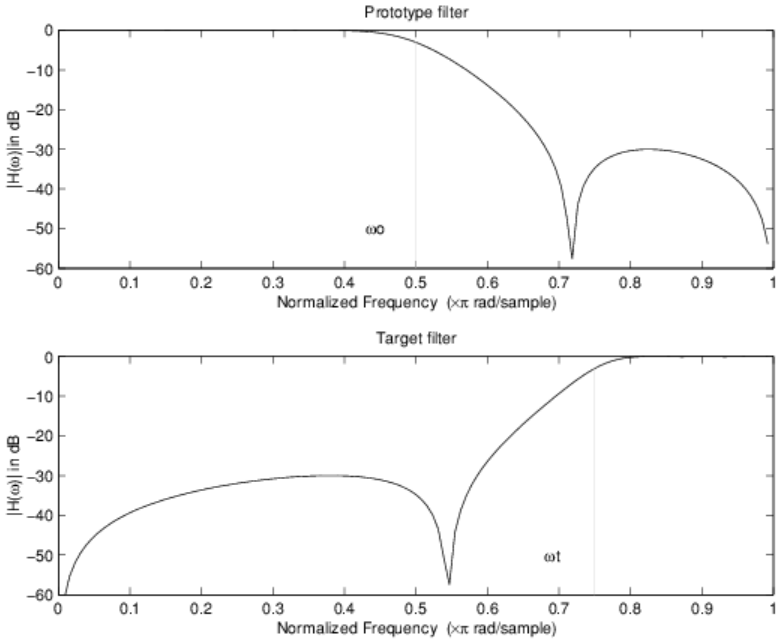
The example below shows how to convert the lowpass filter into a highpass filter with arbitrarily chosen cutoff frequency. In the MATLAB code below, the lowpass filter is converted into a highpass with cutoff frequency shifted from 0.5 to 0.75. Results are shown in the figure.

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

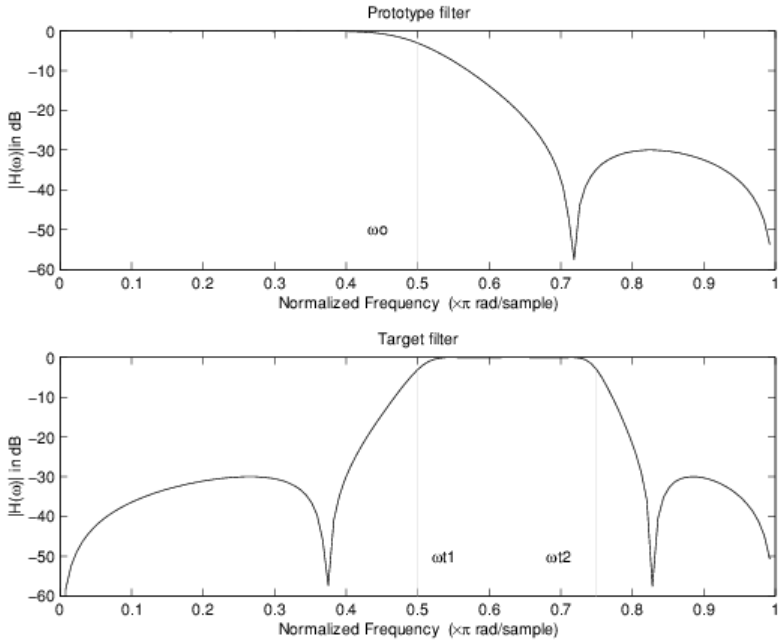
The example below shows how to move the response of the prototype lowpass filter in either direction. Please note that because the target filter must also be real, the response of the target filter will inherently be disturbed at frequencies close to Nyquist and close to DC. Here is the MATLAB code for 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

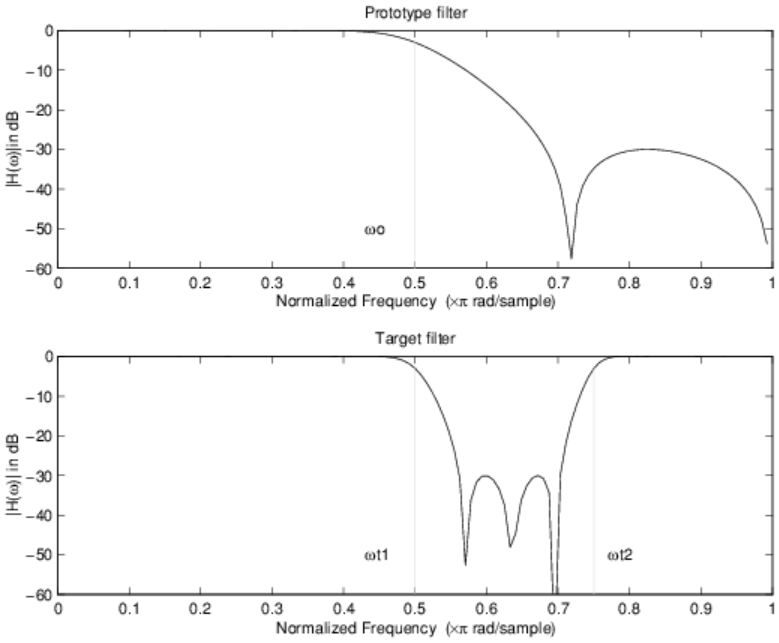
The following example shows how this transformation can be used to convert the prototype lowpass filter with cutoff frequency at 0.5 into a real bandstop filter with the same passband and stopband ripple structure and stopband positioned between 0.5 and 0.75. Here is the MATLAB code for generating the 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:

$$S = \begin{cases} 1 & Nyquist \\ -1 & DC \end{cases}$$

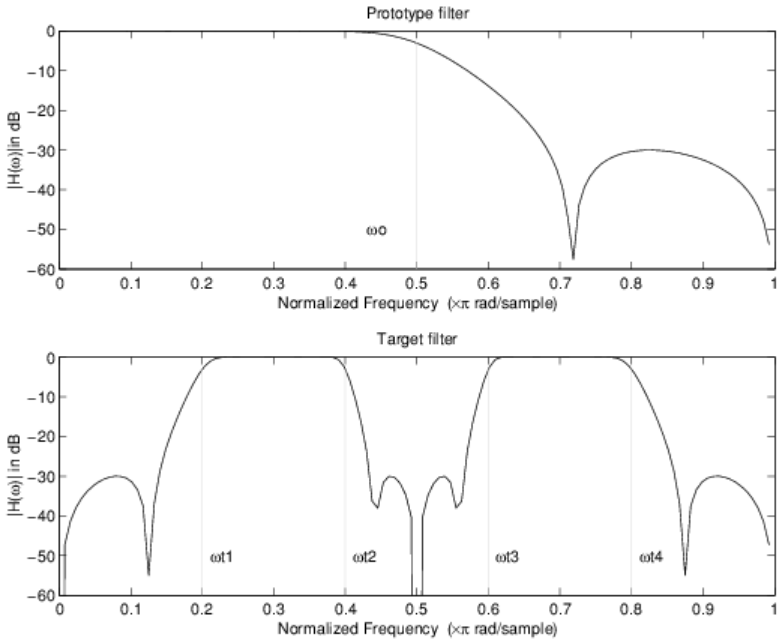
The example below shows how this transformation can be used to convert the prototype lowpass filter with cutoff frequency at 0.5 into a filter having a number of bands positioned at arbitrary edge frequencies 1/5, 2/5, 3/5 and 4/5. Parameter S was such that there is a passband at DC. Here is the MATLAB 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:

$$S = \begin{cases} 1 & Nyquist \\ -1 & DC \end{cases}$$

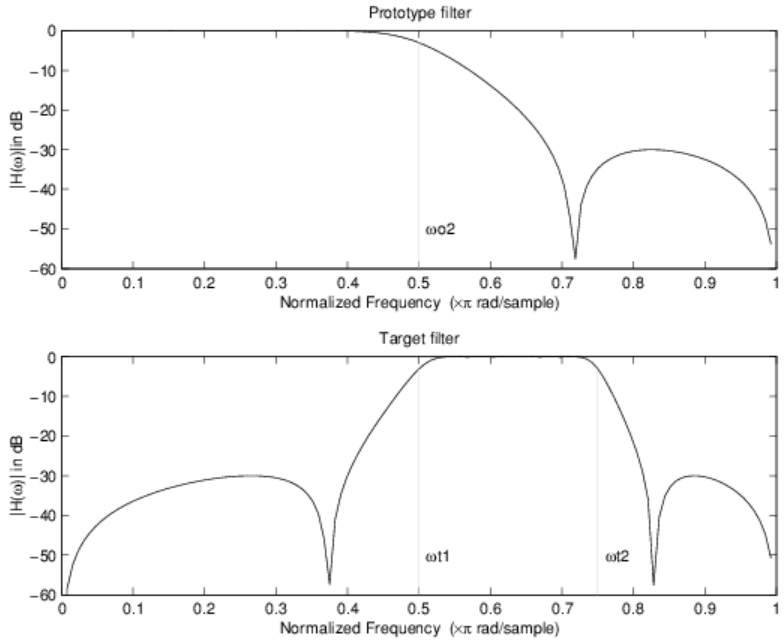
The example below shows how this transformation can be used to move features of the prototype lowpass filter originally at -0.5 and 0.5 to their new locations at 0.5 and 0.75, effectively changing a position of the filter passband. Here is the MATLAB code for generating the figure.

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 5-67
- “Complex Frequency Shift” on page 5-68
- “Real Lowpass to Complex Bandpass” on page 5-69
- “Real Lowpass to Complex Bandstop” on page 5-70
- “Real Lowpass to Complex Multiband” on page 5-71
- “Real Lowpass to Complex Multipoint” on page 5-73
- “Complex Bandpass to Complex Bandpass” on page 5-74

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

$$\alpha = \begin{cases} 1 & forward \\ -1 & inverse \end{cases}$$

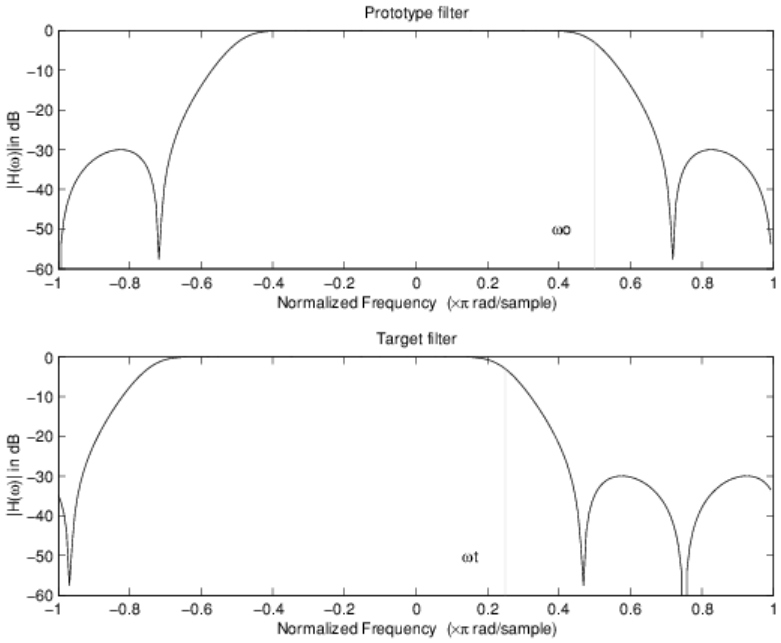
The example below shows how to apply this transformation to rotate the response of the prototype lowpass filter in either direction. Please note that because the transformation can be achieved by a simple phase shift operator, all features of the prototype filter will be moved by the same amount. Here 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

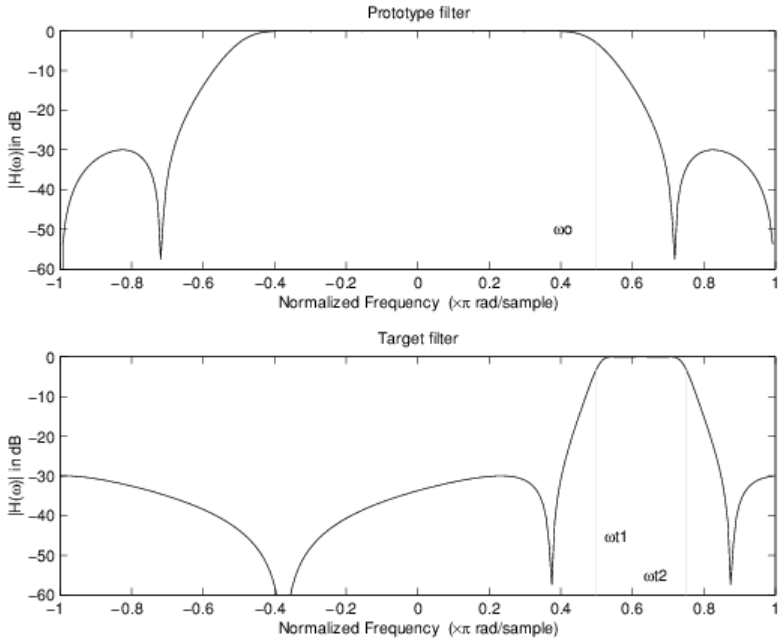
The following example shows the use of such a transformation for converting a real halfband lowpass filter into a complex bandpass filter with band edges at 0.5 and 0.75. Here is the MATLAB code for generating the example in the figure.

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

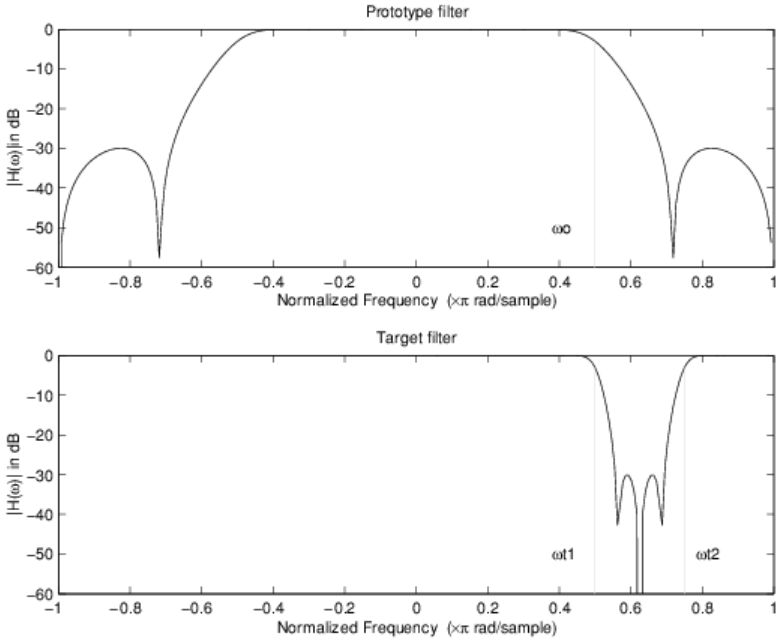
The example below shows the use of such a transformation for converting a real halfband lowpass filter into a complex bandstop filter with band edges at 0.5 and 0.75. Here is the MATLAB code for generating the example in the figure.

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

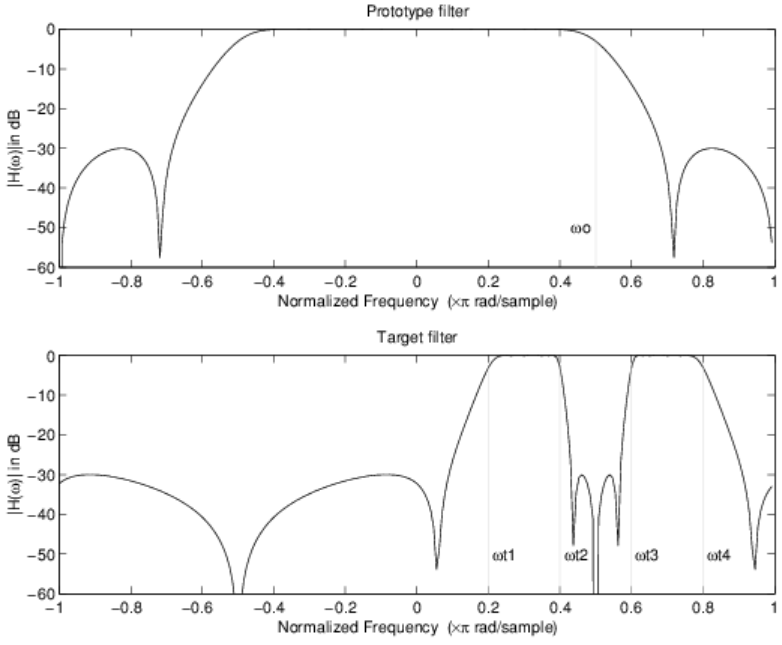
ω_{old} — Frequency location of the selected feature in the prototype filter

$\omega_{new,i}$ — 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:

$$S = e^{-j\pi\Delta C}$$

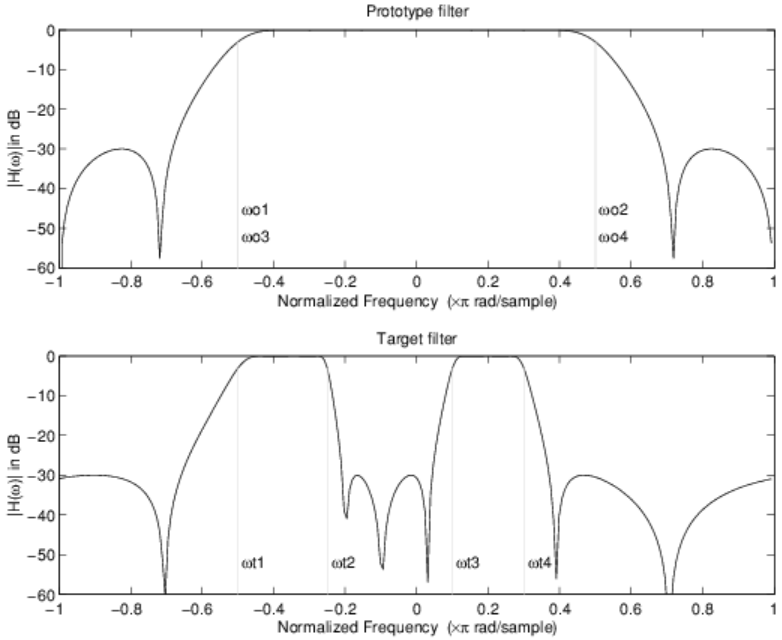
The following example shows how this transformation can be used to move one selected feature of the prototype lowpass filter originally at -0.5 to two new frequencies -0.5 and 0.1, and the second feature of the prototype filter from 0.5 to new locations at -0.25 and 0.3. This creates two nonsymmetric 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

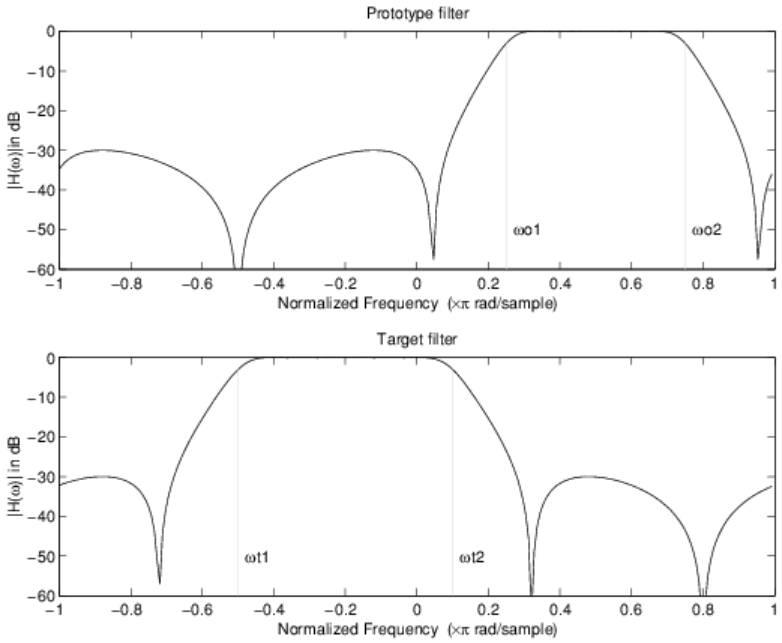
The following example shows how this transformation can be used to modify the position of the passband of the prototype filter, either real or complex. In the example below the prototype filter passband spanned from 0.5 to 0.75. It was converted to having a passband between -0.5 and 0.1. Here is the 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 5-76
“Select a Filter Design Block” on page 5-77
“Create a Lowpass Filter in Simulink” on page 5-78
“Create a Highpass Filter in Simulink” on page 5-78
“Filter High-Frequency Noise in Simulink” on page 5-79

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 24-22. To learn how to import and export your filter designs, see “Import and Export Quantized Filters” on page 5-32.

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 5-77.

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 5-83 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 5-78, 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 5-93.

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 5-78 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 5-78. 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 5-79, 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 5-93.

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 5-78 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.

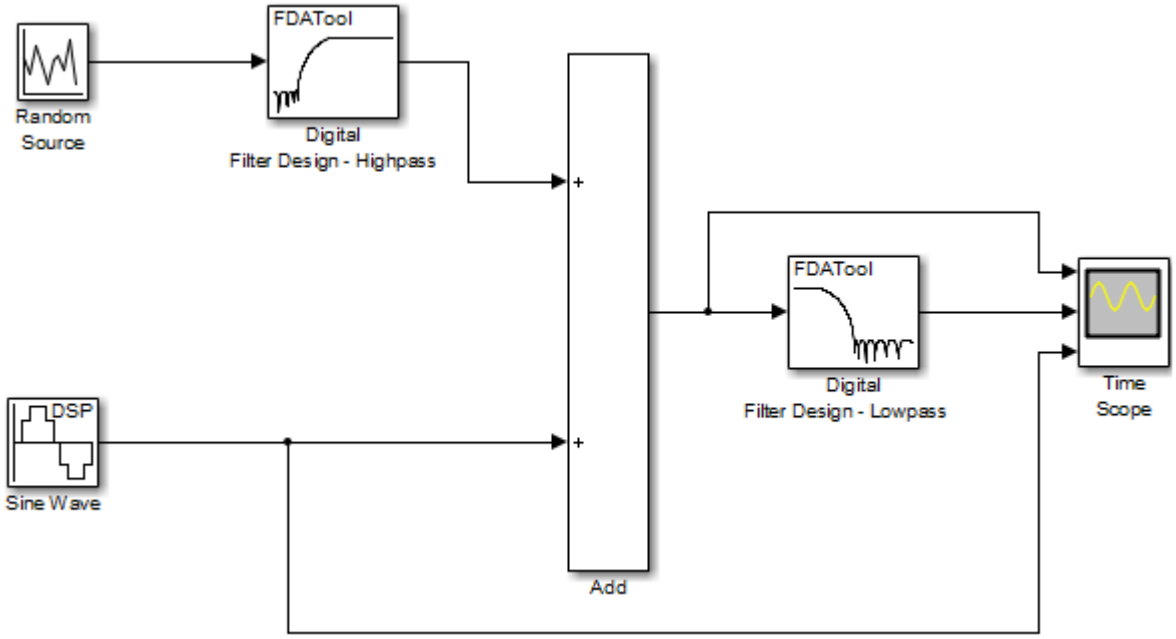
Block	Library	Quantity
Add	Simulink Math Operations library	1
Random Source	Sources	1
Sine Wave	Sources	1
Time Scope	Sinks	1

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:

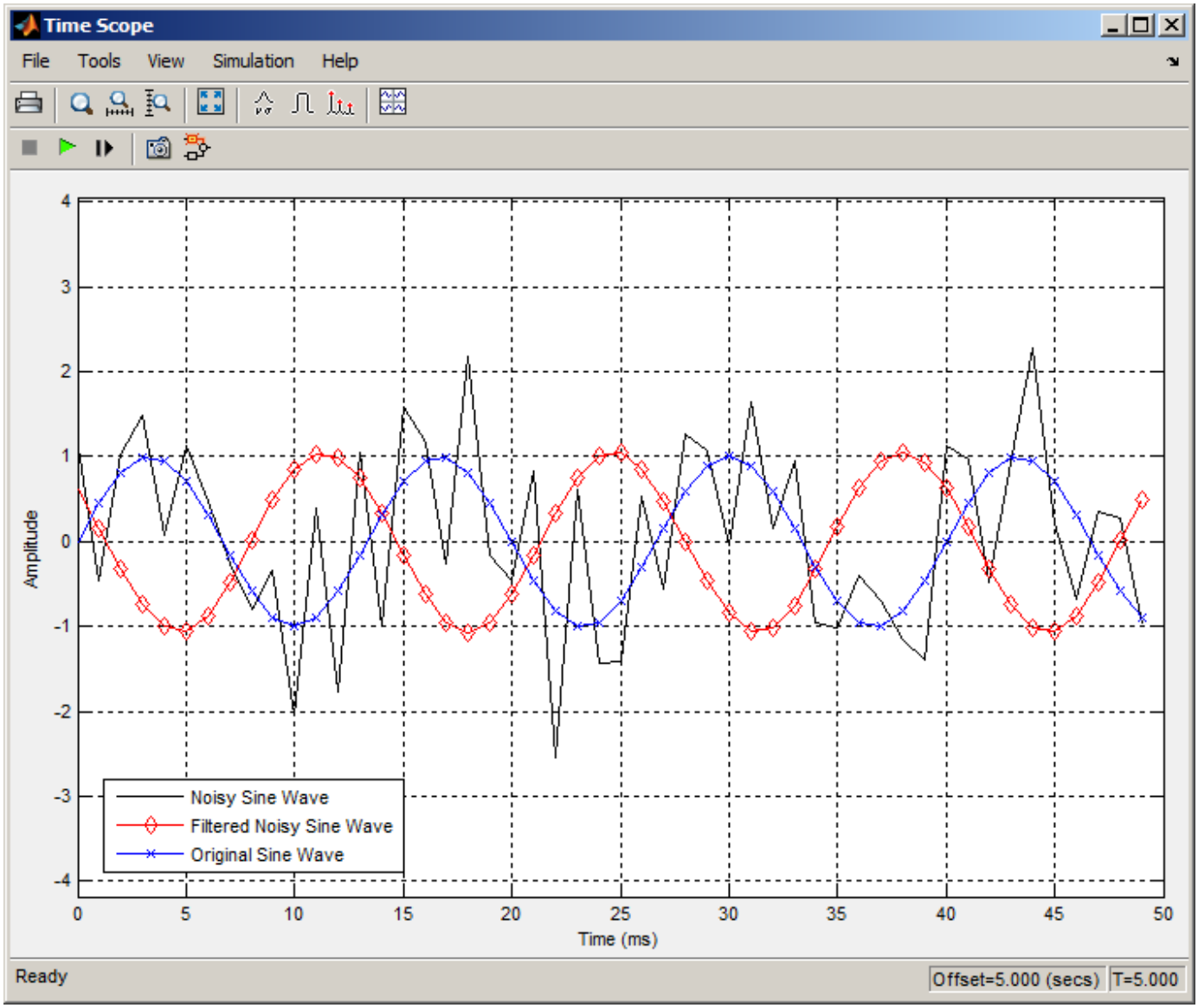
- Add = `Noisy Sine Wave`
- Digital Filter Design – Lowpass = `Filtered Noisy Sine Wave`
- Sine Wave = `Original Sine Wave`

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 5-83. To learn how to save your filter designs, see “Saving and Opening Filter Design Sessions” on page 24-22. To learn how to import and export your filter designs, see “Import and Export Quantized Filters” on page 5-32.

Filter Realization Wizard

In this section...
“Overview of the Filter Realization Wizard” on page 5-83
“Design and Implement a Fixed-Point Filter in Simulink” on page 5-83
“Set the Filter Structure and Number of Filter Sections” on page 5-90
“Optimize the Filter Structure” on page 5-90

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 5-77.

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 5-83
- “Part 2 — Create a Fixed-Point Filter with the Filter Realization Wizard” on page 5-84
- “Part 3 — Build a Model to Filter a Signal” on page 5-88
- “Part 4 — Examine Filtering Results” on page 5-89

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

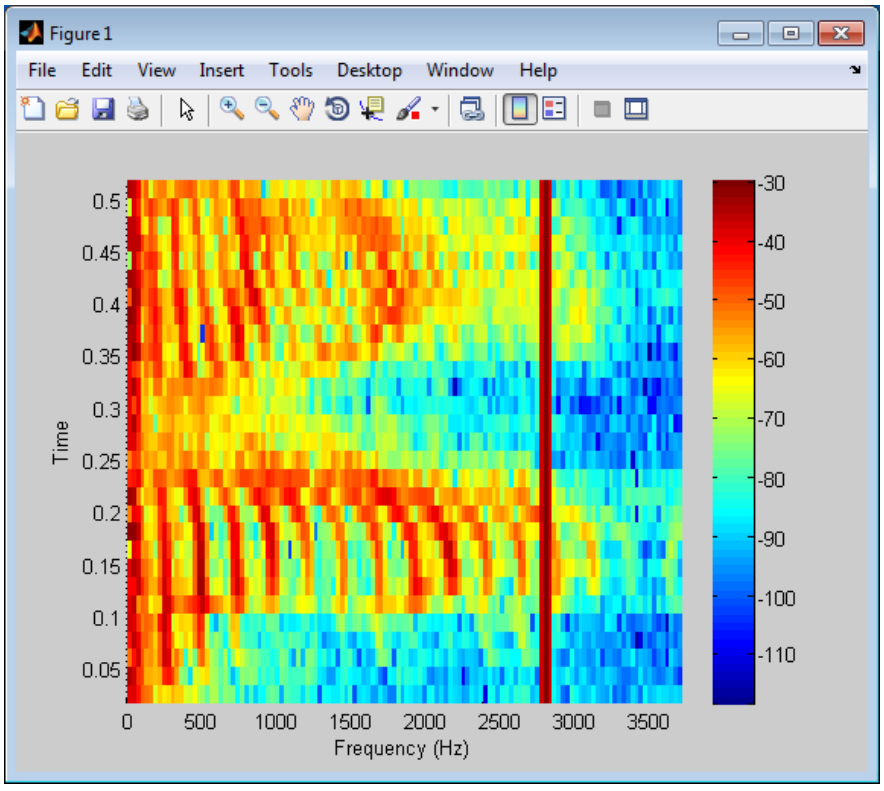
```
soundsc(u,Fs)
```

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.

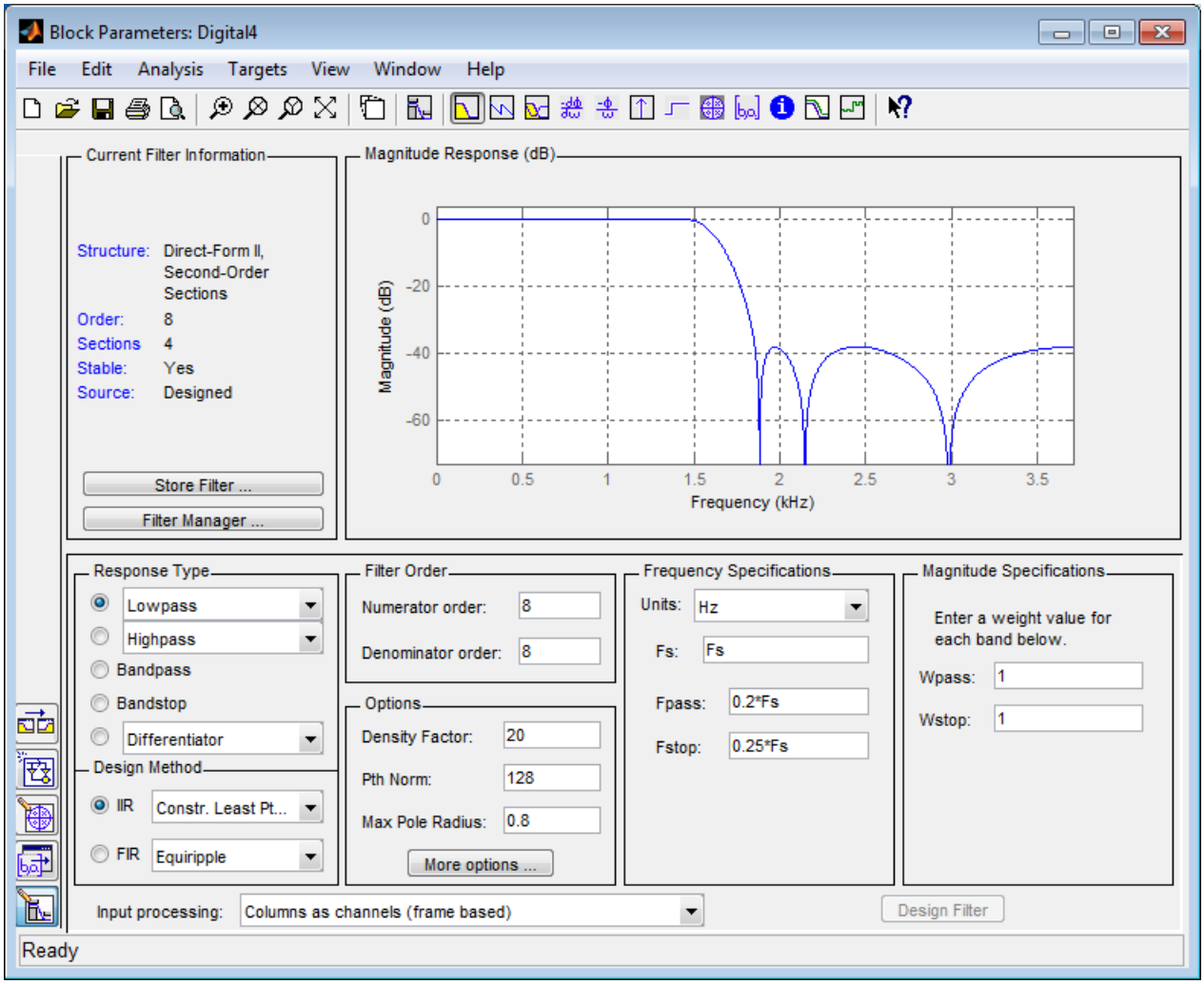
2. Double-click the Filter Realization Wizard block in your model. The **Realize Model** panel of the filter designer appears.

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.

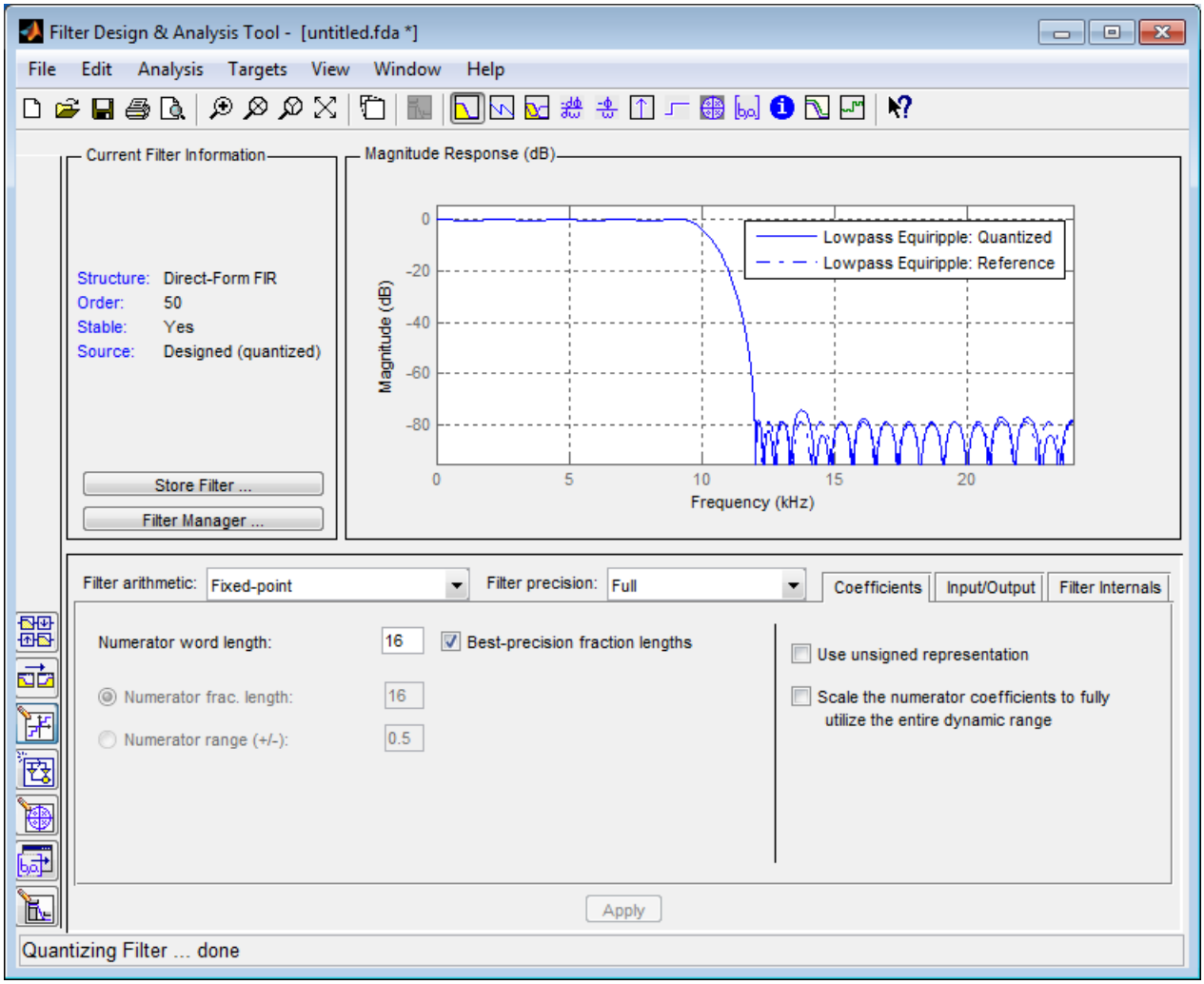


5 Click the **Set quantization parameters** button on the bottom left of filter designer ( ). This brings forward the **Set quantization parameters** panel of the tool.

6 Set the following fields in the **Set quantization parameters** panel:

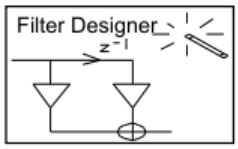
- Select `Fixed-point` for the **Filter arithmetic** parameter.
- Make sure the **Best precision fraction lengths** check box is selected on the **Coefficients** pane.

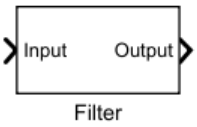
The **Set quantization parameters** panel should appear as follows.



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.

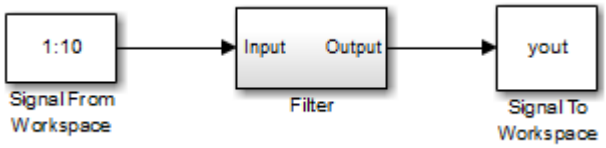
9 Double-click the `Filter` subsystem block in your model to view the filter implementation.

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.

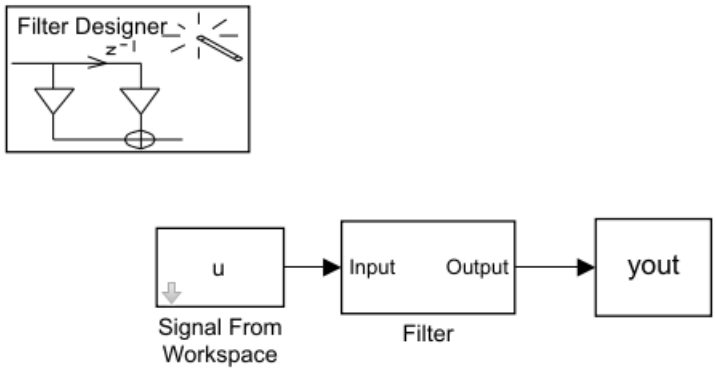
4 In the **Modeling** tab, select **Model Settings**. In the **Solver** pane of the dialog, set the following fields:

- **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

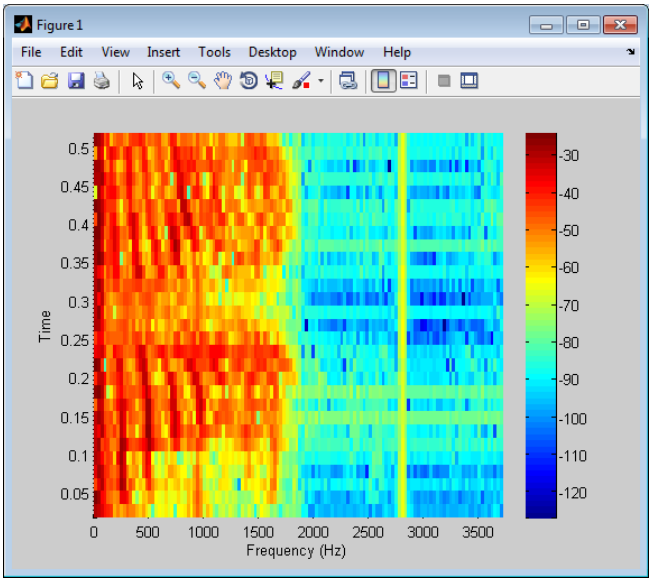
```
soundsc(yout,Fs)
```

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.

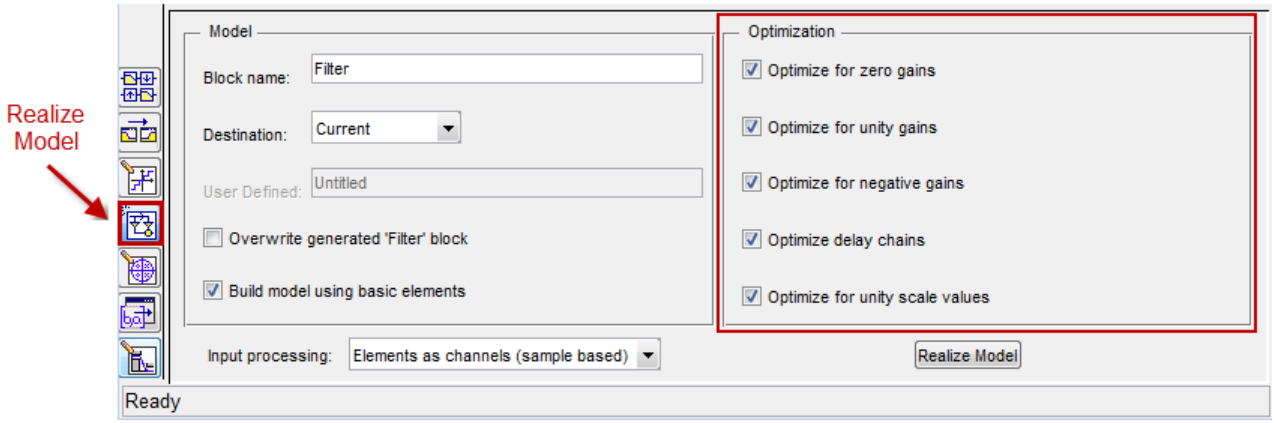
Change the filter structure and number of filter sections of your filter as follows:

- 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

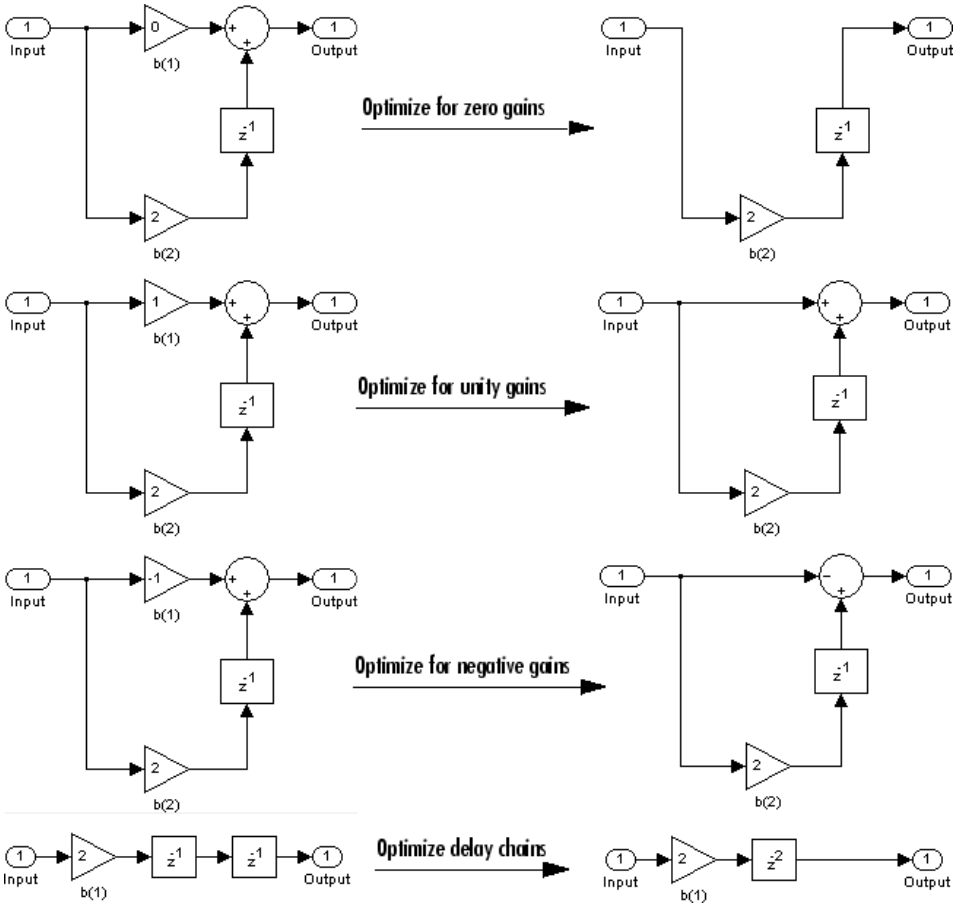
The Filter Realization Wizard can implement a digital filter using either digital filter blocks from the DSP System Toolbox library or by creating a subsystem (Simulink) block that implements the filter using Sum, Gain, and Delay blocks. The following procedure shows you how to optimize the filter implementation:

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 5-93
“Implement a Lowpass Filter in Simulink” on page 5-93
“Implement a Highpass Filter in Simulink” on page 5-94
“Filter High-Frequency Noise in Simulink” on page 5-95
“Specify Static Filters” on page 5-98
“Specify Time-Varying Filters” on page 5-99
“Specify the SOS Matrix (Biquadratic Filter Coefficients)” on page 5-99

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 5-94, 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 5-76.

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 5-93 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 5-95, 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 5-76.

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 5-94 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.

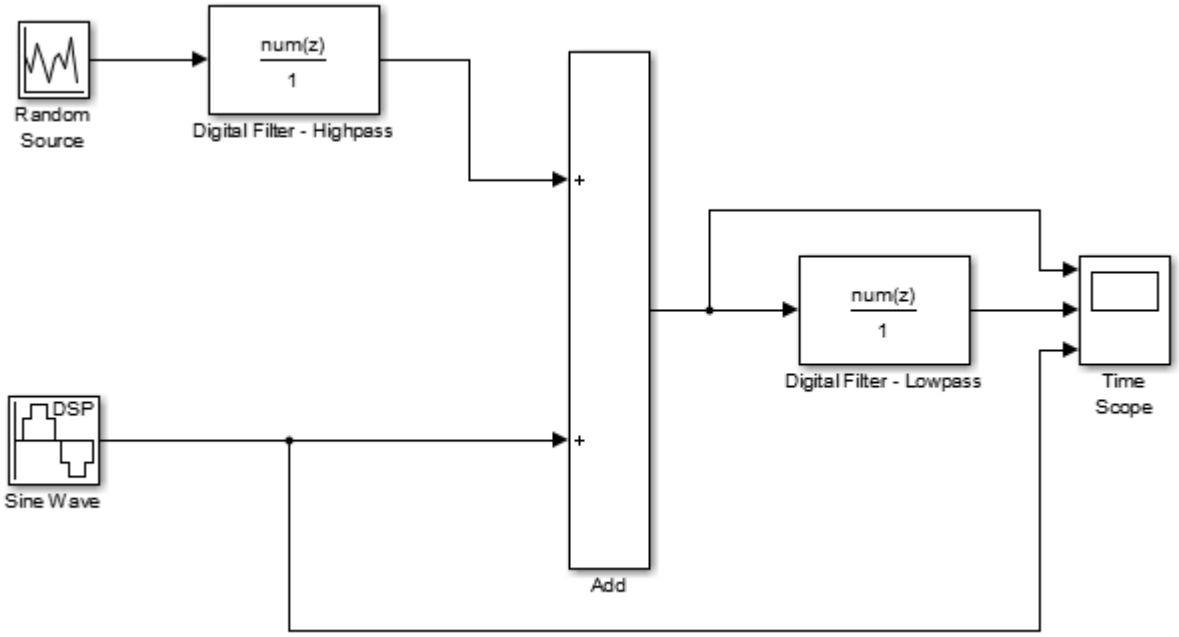
Block	Library	Quantity
Add	Simulink / Math Operations library	1
Random Source	Sources	1
Sine Wave	Sources	1
Time Scope	Sinks	1

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`

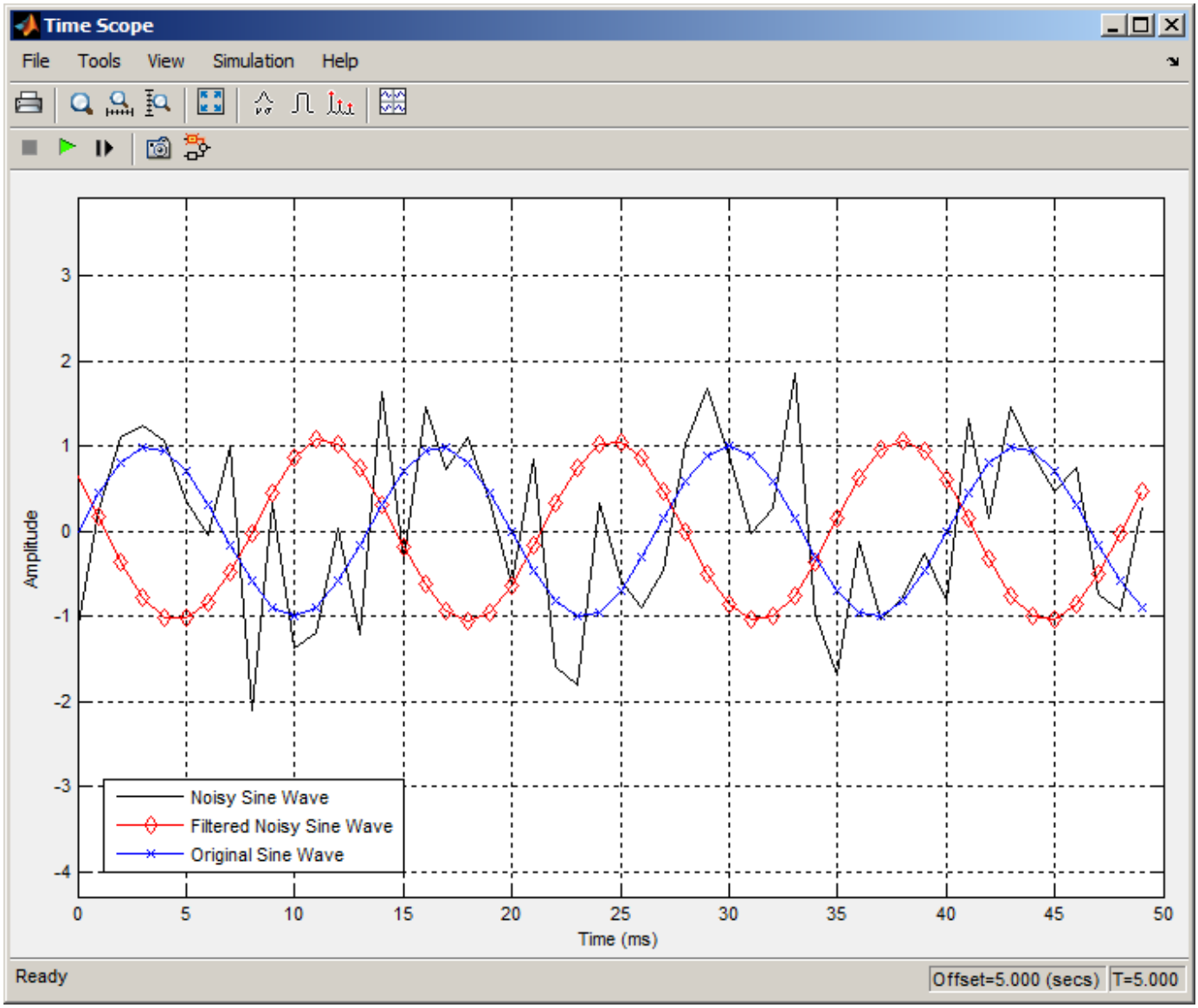
In the next step, you will set the color, style, and marker of each channel.

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 5-76.

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)

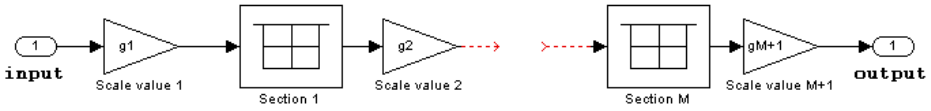
Use the Biquad Filter block to specify a static biquadratic IIR filter (also known as a second-order section or SOS filter). Set the following parameters:

- **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 = timescope('SampleRate',Fs,...
                    'TimeSpanSource','Property',...
                    'TimeSpan',1.5,...
                    'ShowGrid',true,...
                    'NumInputPorts',2,...
                    'LayoutDimensions',[2 1]);
TS.ActiveDisplay = 1;
TS.YLimits = [-1,1];
TS.Title = 'Noisy Signal';
TS.ActiveDisplay = 2;
TS.YLimits = [-1,1];
TS.Title = 'Filtered Signal';
```

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

`timescope` | `dsp.FIRFilter`

More About

- “Remove High-Frequency Noise from Gyroscope Data” on page 28-24

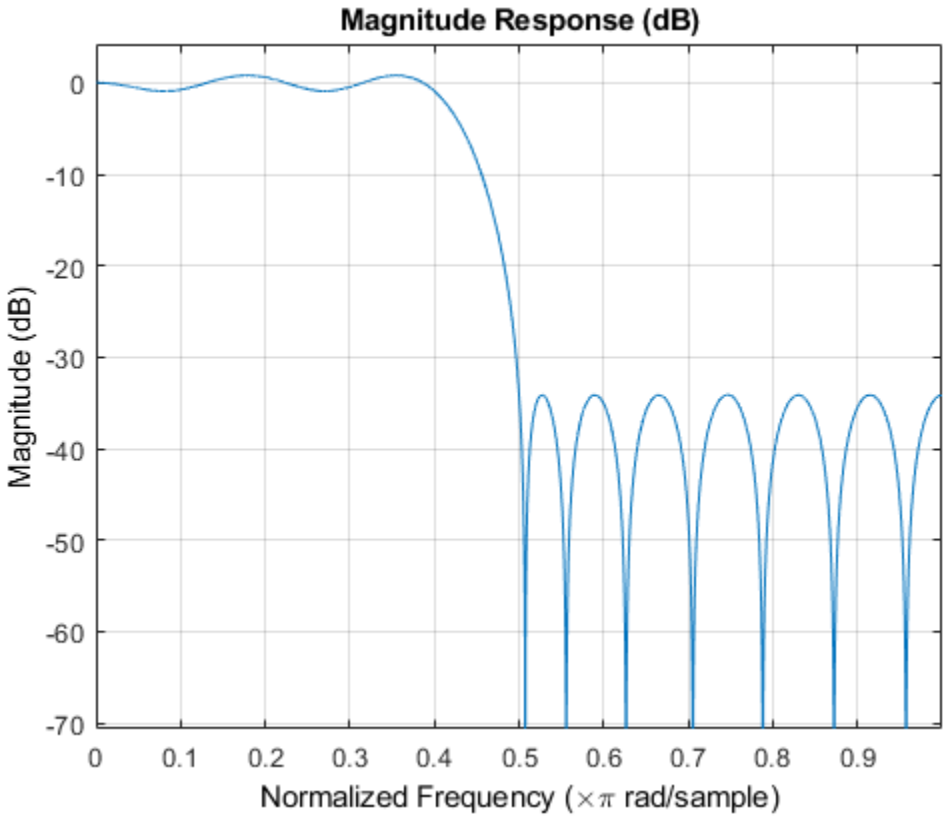
Minimax FIR Filter Design

This example shows how to use some of the key features of the generalized Remez FIR filter design function. This function provides all the functionality included in `firpm` plus many additional features showcased here.

Weighted-Chebyshev Design

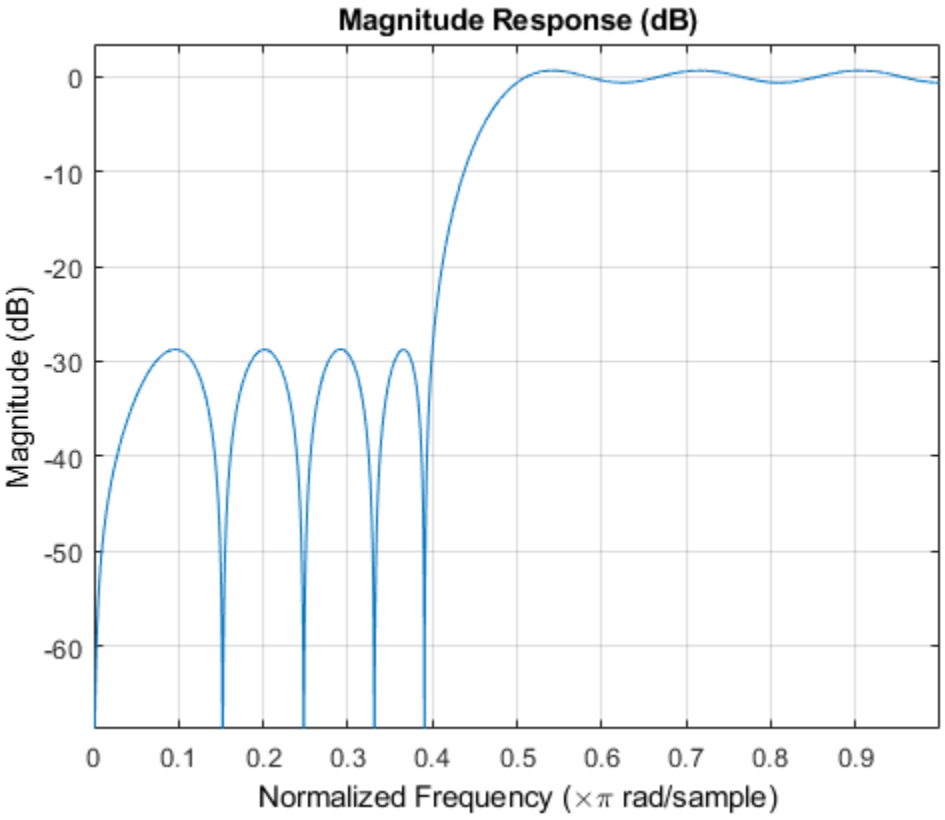
The following is an illustration of the weighted-Chebyshev design. This example shows the compatibility of `firgr` with `firpm`.

```
N = 22;             % Filter order
F = [0 0.4 0.5 1];  % Frequency vector
A = [1 1 0 0];      % Magnitude vector
W = [1 5];          % Weight vector
b = firgr(N,F,A,W);
fvtool(b,'Color','White');
```



The following is a weighted-Chebyshev design where a type 4 filter (odd-order, asymmetric) has been explicitly specified.

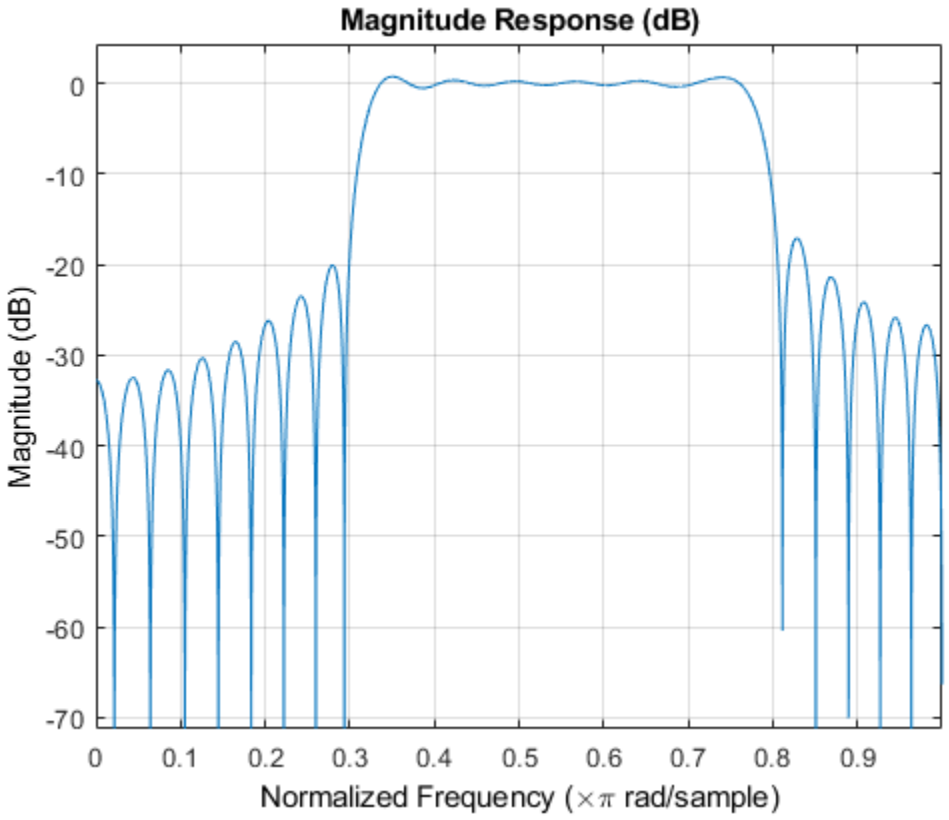
```
N = 21;             % Filter order
F = [0 0.4 0.5 1];  % Frequency vector
A = [0 0 1 1];      % Magnitude vector
W = [2 1];          % Weight vector
b = firgr(N,F,A,W,'4');
fvtool(b,'Color','White');
```



"Least-Squares-Like" Design

The following illustrates a "least-squares-like" design. A user-supplied frequency-response function (`taperedresp.m`) is used to perform the error weighting.

```
N = 53;                                    % Filter order
F = [0 0.3 0.33 0.77 0.8 1];               % Frequency vector
fresp = {@taperedresp, [0 0 1 1 0 0]};  % Frequency response function
W = [2 2 1];                               % Weight vector
b = firgr(N,F,fresp,W);
fvtool(b,'Color','White');
```

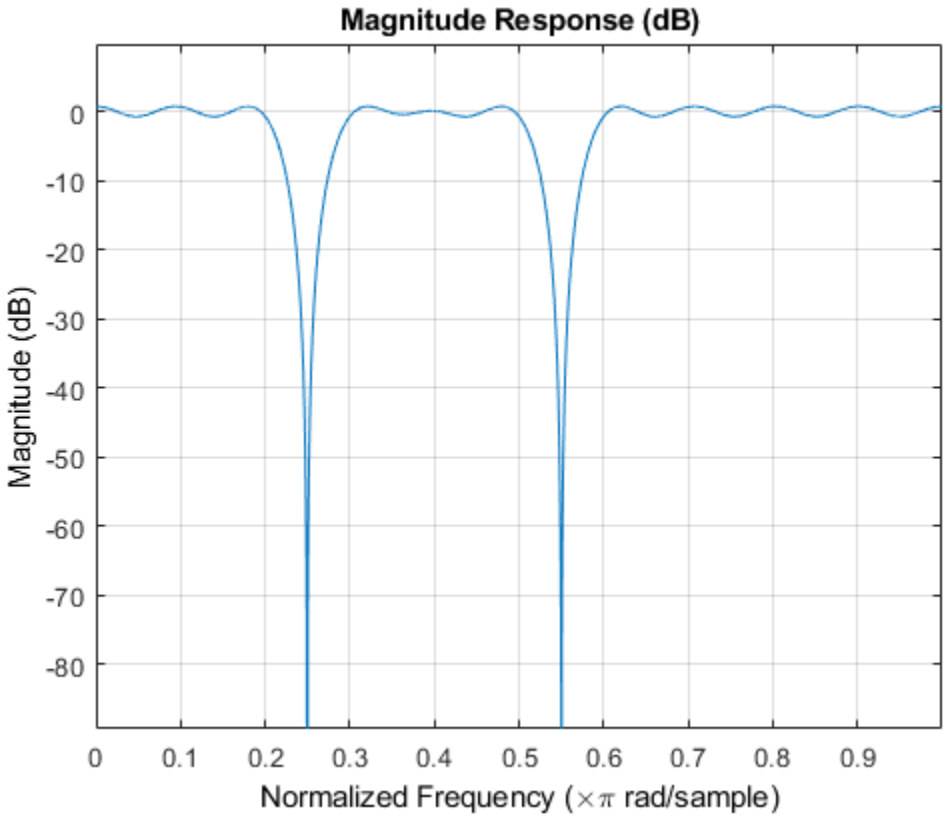


Filter Designed for Specific Single-Point Bands

This is an illustration of a filter designed for specified single-point bands. The frequency points f = 0.25 and f = 0.55 are single-band points. These points have a gain that approaches zero.

The other band edges are normal.

```
N = 42;                                 % Filter order
F = [0 0.2 0.25 0.3 0.5 0.55 0.6 1];  % Frequency vector
A = [1 1 0 1 1 0 1 1];                  % Magnitude vector
S = {'n' 'n' 's' 'n' 'n' 's' 'n' 'n'};
b = firgr(N,F,A,S);
fvtool(b,'Color','White');
```

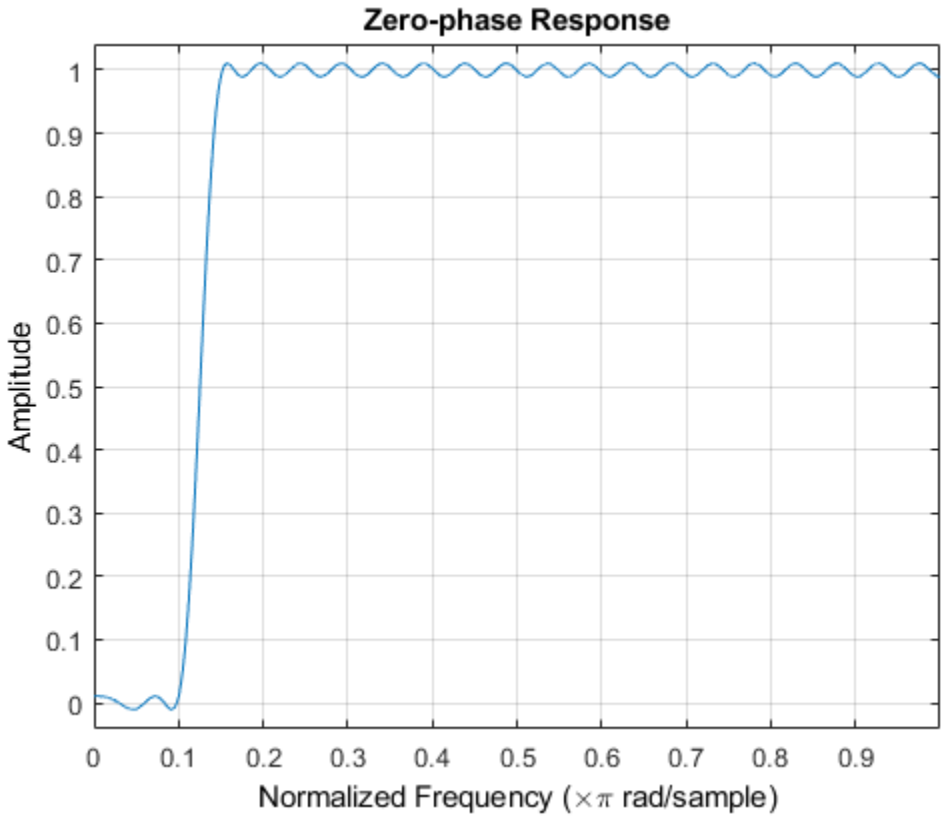


Filter Designed for Specific In-Band Value

Here is an illustration of a filter designed for an exactly specified in-band value. The value is forced to be exactly the specified value of 0.0 at f = 0.06.

This could be used for 60 Hz rejection (with Fs = 2 kHz). The band edge at 0.055 is indeterminate since it should abut the next band.

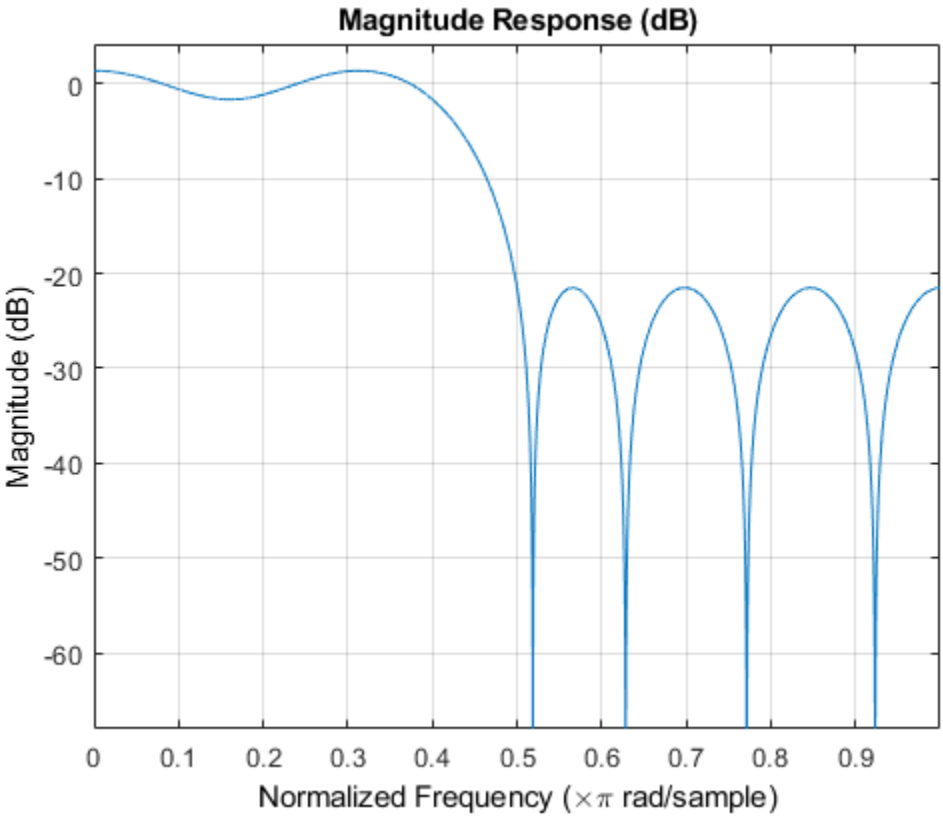
```
N = 82;                          % Filter order
F = [0 0.055 0.06 0.1 0.15 1];   % Frequency vector
A = [0 0 0 0 1 1];               % Magnitude vector
S = {'n' 'i' 'f' 'n' 'n' 'n'};
b = firgr(N,F,A,S);
fvtool(b,'Color','White', 'MagnitudeDisplay', 'Zero-phase');
```



Filter Design with Specific Multiple Independent Approximation Errors

Here is an example of designing a filter using multiple independent approximation errors. This technique is used to directly design extra-ripple and maximal ripple filters. One of the interesting properties that these filters have is a transition region width that is locally minimal. Further, these designs converge very quickly in general.

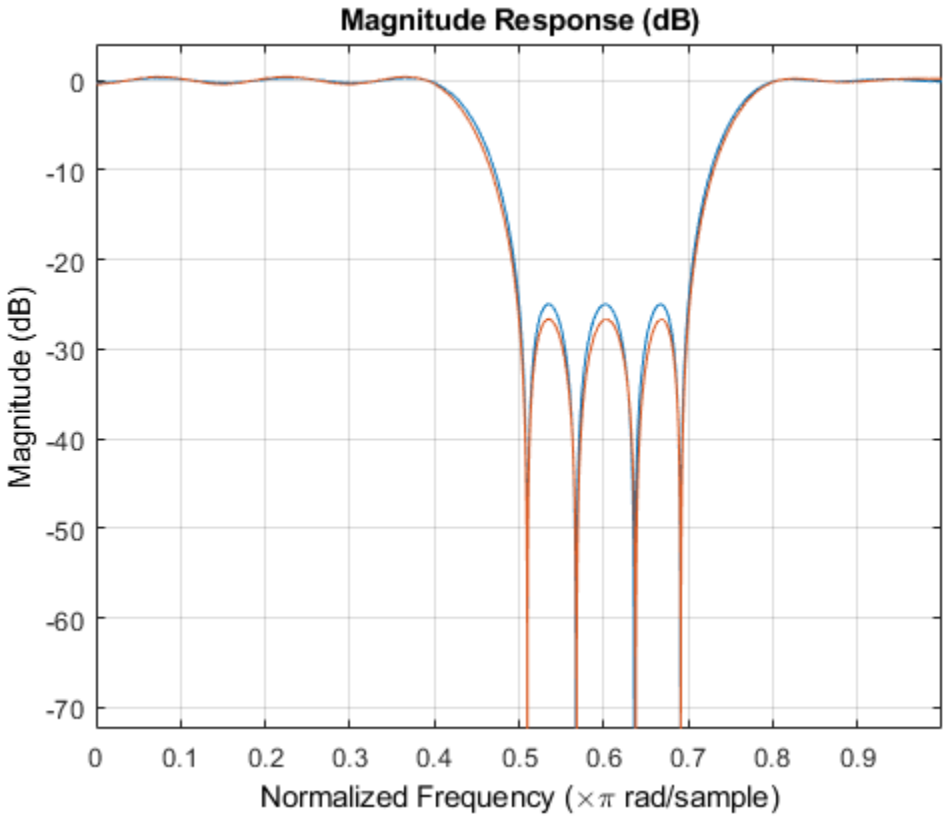
```
N = 12;              % Filter order
F = [0 0.4 0.5 1];   % Frequency vector
A = [1 1 0 0];       % Magnitude vector
W = [1 1];           % Weight vector
E = {'e1' 'e2'};     % Approximation errors
b = firgr(N,F,A,W,E);
fvtool(b,'Color','White');
```



Extra-Ripple Bandpass Filter

Here is an illustration of an extra-ripple bandpass filter having two independent approximation errors: one shared by the two passbands and the other for the stopband (in blue). For comparison, a standard weighted-Chebyshev design is also plotted (in green).

```
N = 28;                     % Filter order
F = [0 0.4 0.5 0.7 0.8 1];  % Frequency vector
A = [1 1 0 0 1 1];          % Magnitude vector
W = [1 1 2];                % Weight vector
E = {'e1','e2','e1'};       % Approximation errors
b1 = firgr(N,F,A,W,E);
b2 = firgr(N,F,A,W);
fvtool(b1,1,b2,1,'Color','White');
```

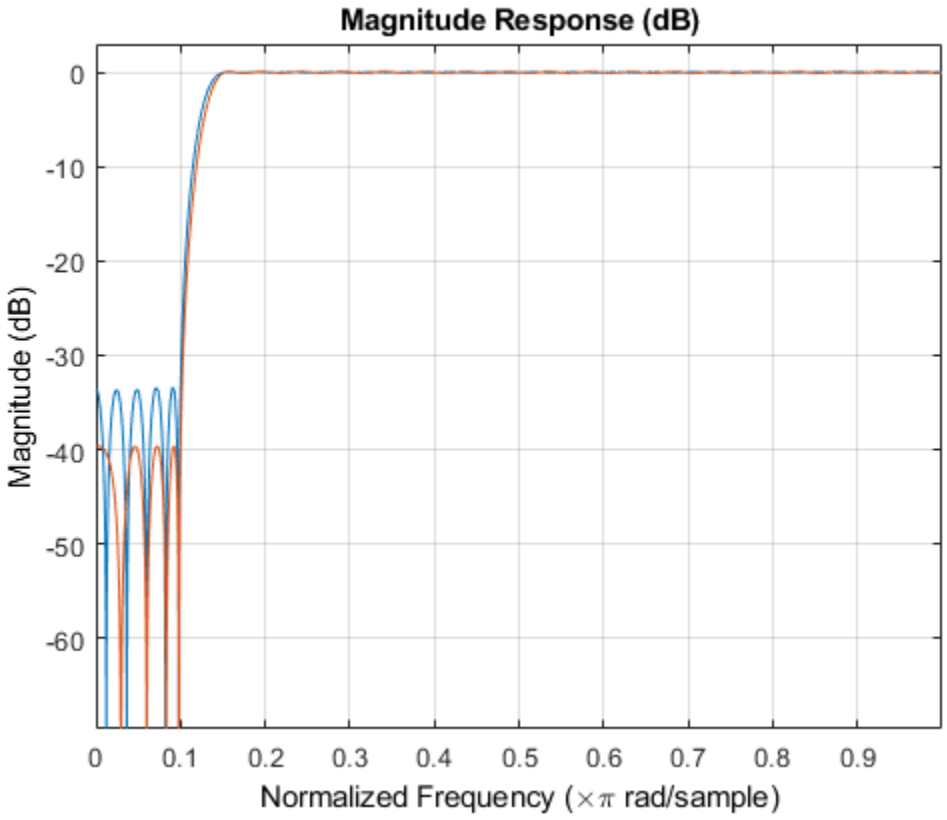


Designing an In-Band-Zero Filter Using Three Independent Errors

We'll now re-do our in-band-zero example using three independent errors.

Note: It is sometimes necessary to use independent approximation errors to get designs with forced in-band values to converge. This is because the approximating polynomial could otherwise be come very underdetermined. The former design is displayed in green.

```
N = 82;                          % Filter order
F = [0 0.055 0.06 0.1 0.15 1];   % Frequency vector
A = [0 0 0 0 1 1];               % Magnitude vector
S = {'n' 'i' 'f' 'n' 'n' 'n'};
W = [10 1 1];                    % Weight vector
E = {'e1' 'e2' 'e3'};            % Approximation errors
b1 = firgr(N,F,A,S,W,E);
b2 = firgr(N,F,A,S);
fvtool(b1,1,b2,1,'Color','White');
```



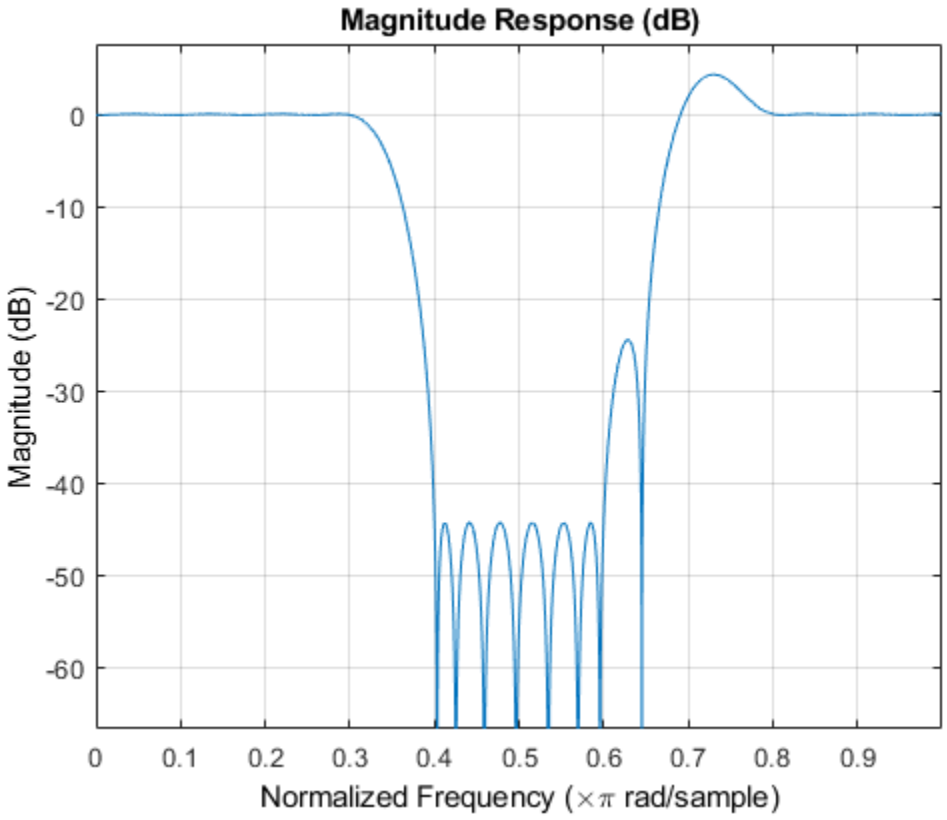
Checking for Transition-Region Anomalies

With the `'check'` option, one is made aware of possible transition region anomalies in the filter that is being designed. Here is an example of a filter with an anomaly. The `'check'` option warns one of this anomaly: One also get a results vector `res.edgeCheck`. Any zero-valued elements in this vector indicate the locations of probable anomalies. The "-1" entries are for edges that were not checked (there can't be an anomaly at f = 0 or f = 1).

```
N = 44;                     % Filter order
F = [0 0.3 0.4 0.6 0.8 1];  % Frequency vector
A = [1 1 0 0 1 1];          % Magnitude vector
b = firgr(N,F,A,'check');
```

```
Warning: Probable transition-region anomalies.  Verify with freqz.
```

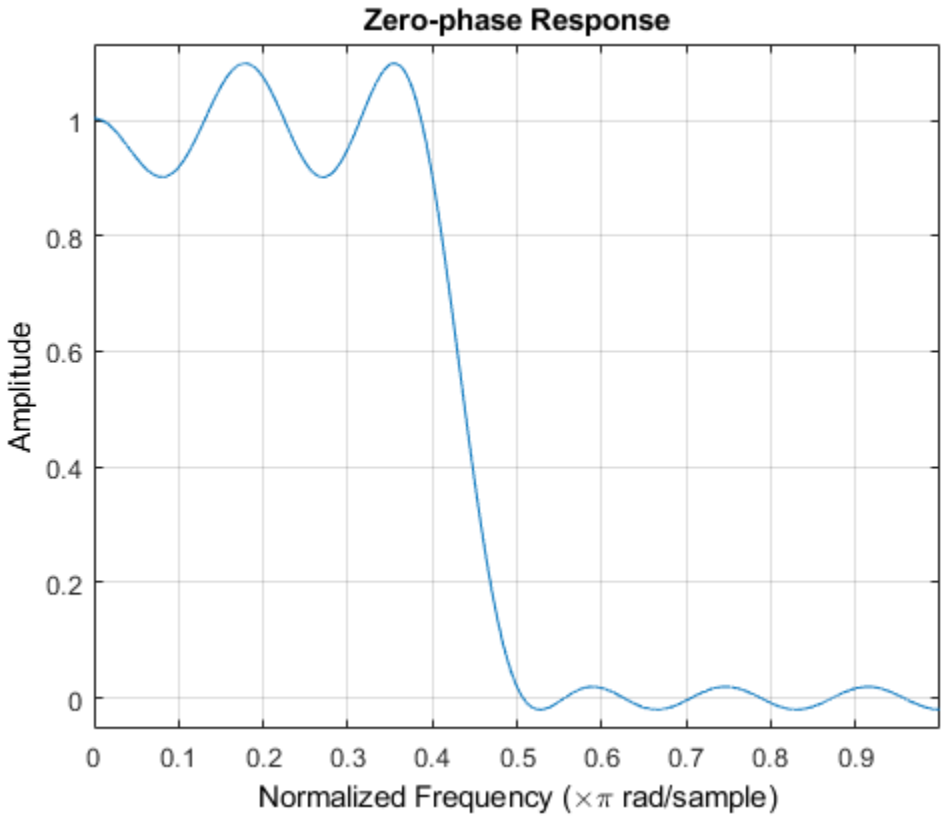
```
fvtool(b,'Color','White');
```



Determination of the Minimum Filter Order

The `firpm` algorithm repeatedly designs filters until the first iteration wherein the specifications are met. The specifications are met when all of the required constraints are met. By specifying `'minorder'`, `firpmord` is used to get an initial estimate. There is also `'mineven'` and `'minodd'` to get the minimum-order even-order or odd-order filter designs.

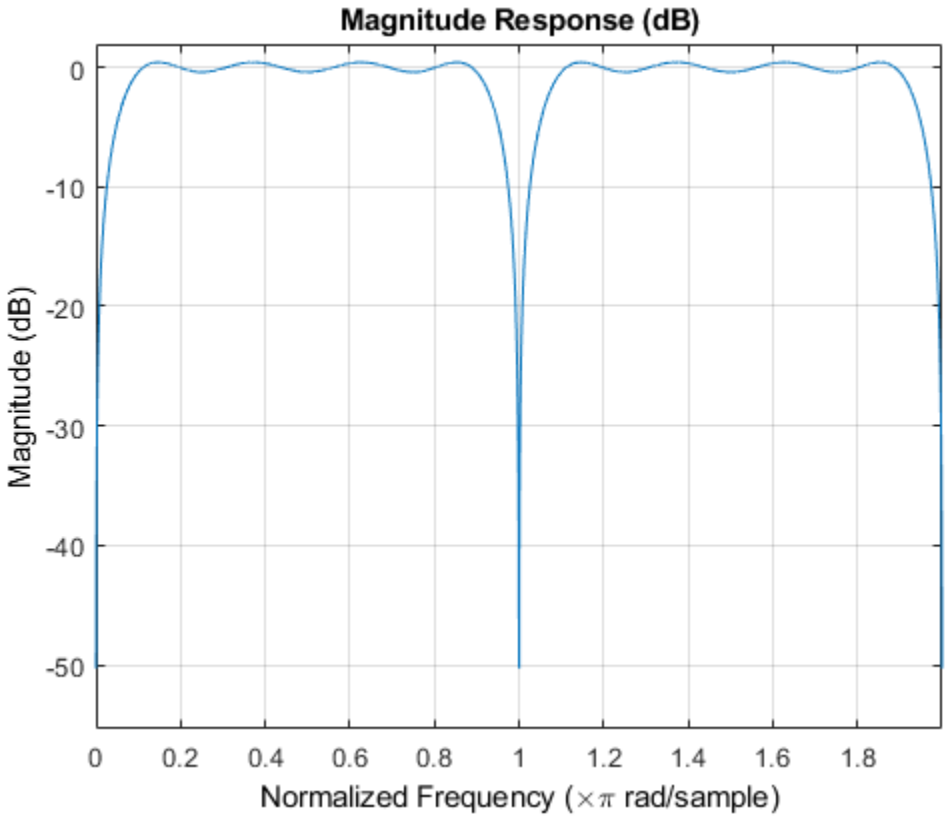
```
F = [0 0.4 0.5 1];  % Frequency vector
A = [1 1 0 0];      % Magnitude vector
R = [0.1 0.02];     % Deviation (ripple) vector
b = firgr('minorder',F,A,R);
fvtool(b,'Color','White', 'MagnitudeDisplay', 'Zero-phase');
```



Differentiators and Hilbert Transformers

While using the minimum-order feature, an initial estimate of the filter order can be made. If this is the case, then `firpmord` will not be used. This is necessary for filters that `firpmord` does not support, such as differentiators and Hilbert transformers as well as user-supplied frequency-response functions.

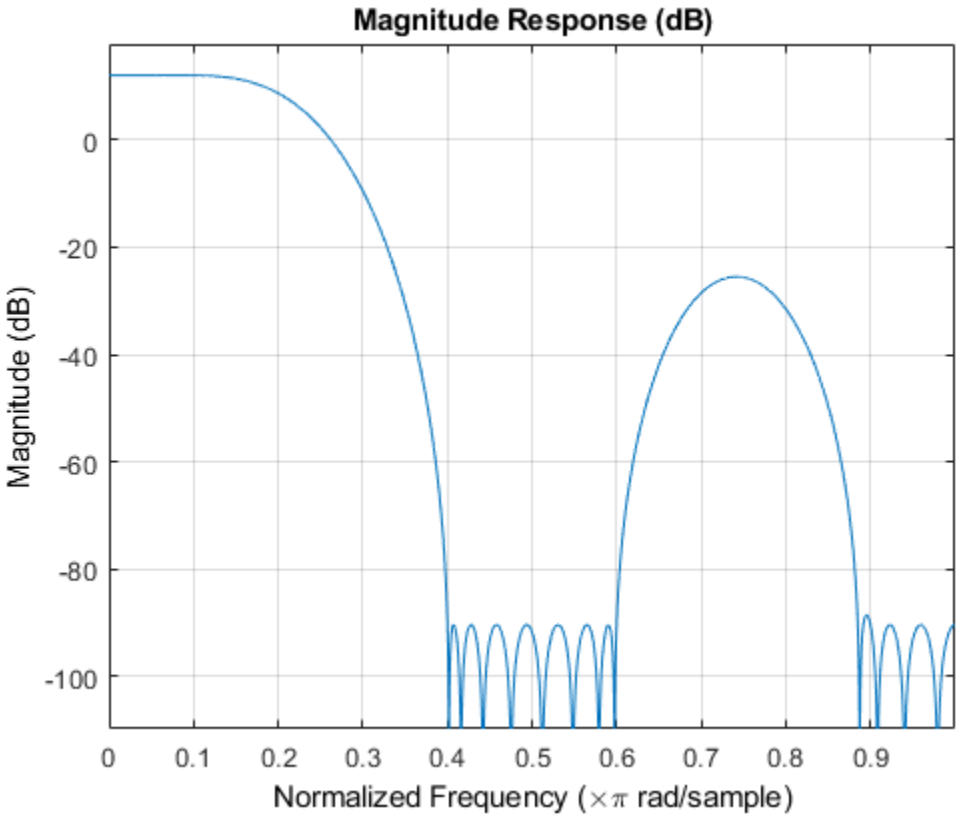
```
N = {'mineven',18}; % Minimum even-order, start order estimate at 18
F = [0.1 0.9];      % Frequency vector
A = [1 1];          % Magnitude vector
R = 0.1;            % Deviation (ripple)
b = firgr(N,F,A,R,'hilbert');
fvtool(b,'Color','White', 'FrequencyRange', '[0, 2pi)');
```



Design of an Interpolation Filter

This section illustrates the use of an interpolation filter for upsampling band-limited signals by an integer factor. Typically one would use `intfilt(r,l,alpha)` from the Signal Processing Toolbox™ to do this. However, `intfilt` does not give one as much flexibility in the design as does `firgr`.

```
N = 30;                       % Filter order
F = [0 0.1 0.4 0.6 0.9 1];    % Frequency vector
A = [4 4 0 0 0 0];            % Magnitude vector
W = [1 100 100];              % Weight vector
b = firgr(N,F,A,W);
fvtool(b,'Color','White');
```

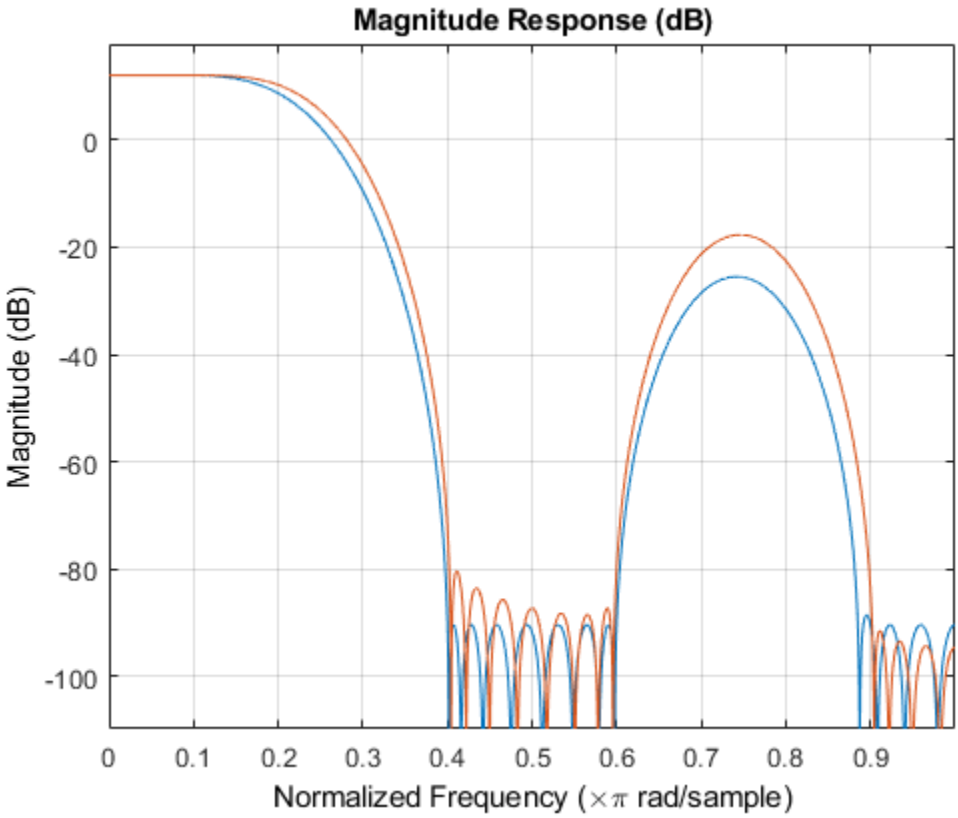


A Comparison Between `firpm` and `intfilt`

Here is a comparison made between a filter designed using `firpm` (blue) and a 30-th order filter designed using `intfilt` (green).

Notice that by using the weighting function in `firpm`, one can improve the minimum stopband attenuation by almost 20 dB.

```
b2 = intfilt(4, 4, 0.4);
fvtool(b,1,b2,1,'Color','White');
```



Notice that the equiripple attenuation throughout the second stopband is larger than the minimum stopband attenuation of the filter designed with `intfilt` by about 6 dB. Notice also that the passband ripple, although larger than that of the filter designed with `intfilt`, is still very small.

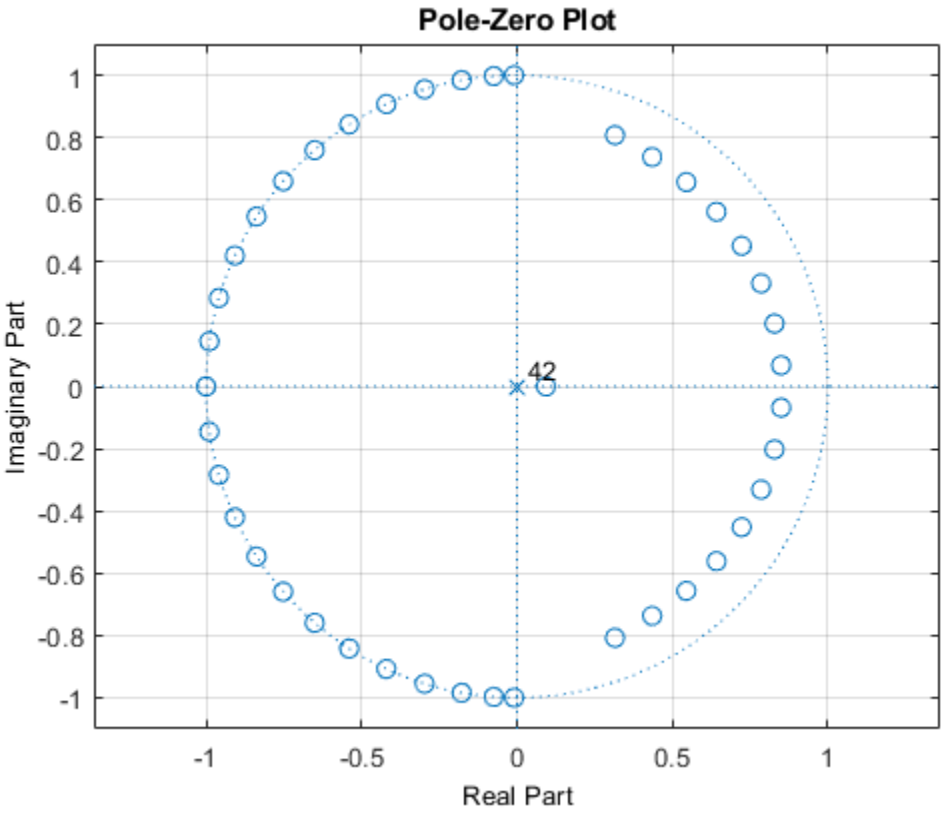
Design of a Minimum-Phase Lowpass Filter

Here is an illustration of a minimum-phase lowpass filter.

```
N = 42;              % Filter order
F = [0 0.4 0.5 1];   % Frequency vector
A = [1 1 0 0];       % Magnitude vector
W = [1 10];         % Weight-constraint vector
b = firgr(N,F,A,W, {64},'minphase');
hfvt = fvtool(b,'Color','White');
```

The pole/zero plot shows that there are no roots outside of the unit circle.

```
hfvt.Analysis = 'polezero';
```


Pole-Zero Plot
Imaginary Part
Real Part
42
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-1
-0.5
0
0.5
1

Adaptive Filters

Learn how to design and implement adaptive filters.

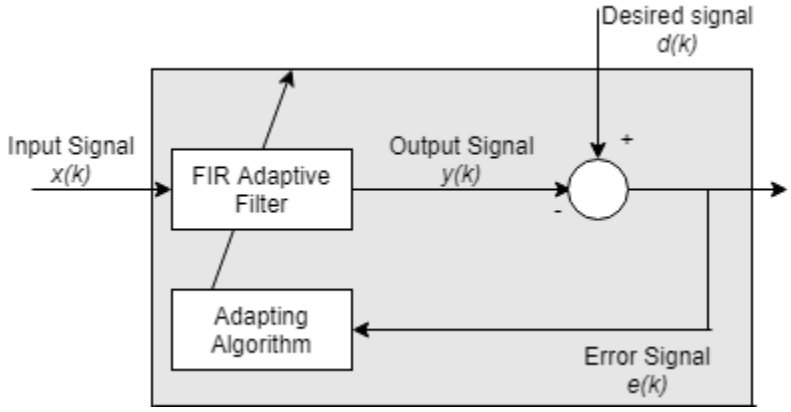
- “Overview of Adaptive Filters and Applications” on page 6-2
- “System Identification of FIR Filter Using LMS Algorithm” on page 6-9
- “System Identification of FIR Filter Using Normalized LMS Algorithm” on page 6-17
- “Compare Convergence Performance Between LMS Algorithm and Normalized LMS Algorithm” on page 6-20
- “Noise Cancellation Using Sign-Data LMS Algorithm” on page 6-22
- “Compare RLS and LMS Adaptive Filter Algorithms” on page 6-26
- “Inverse System Identification Using RLS Algorithm” on page 6-29
- “Signal Enhancement Using LMS and NLMS Algorithms” on page 6-34
- “Noise Cancellation in Simulink Using Normalized LMS Adaptive Filter” on page 6-43

Overview of Adaptive Filters and Applications

In this section...
“Adaptive Filters in DSP System Toolbox” on page 6-2
“Choosing an Adaptive Filter” on page 6-4
“Mean Squared Error Performance” on page 6-5
“Common Applications” on page 6-5

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

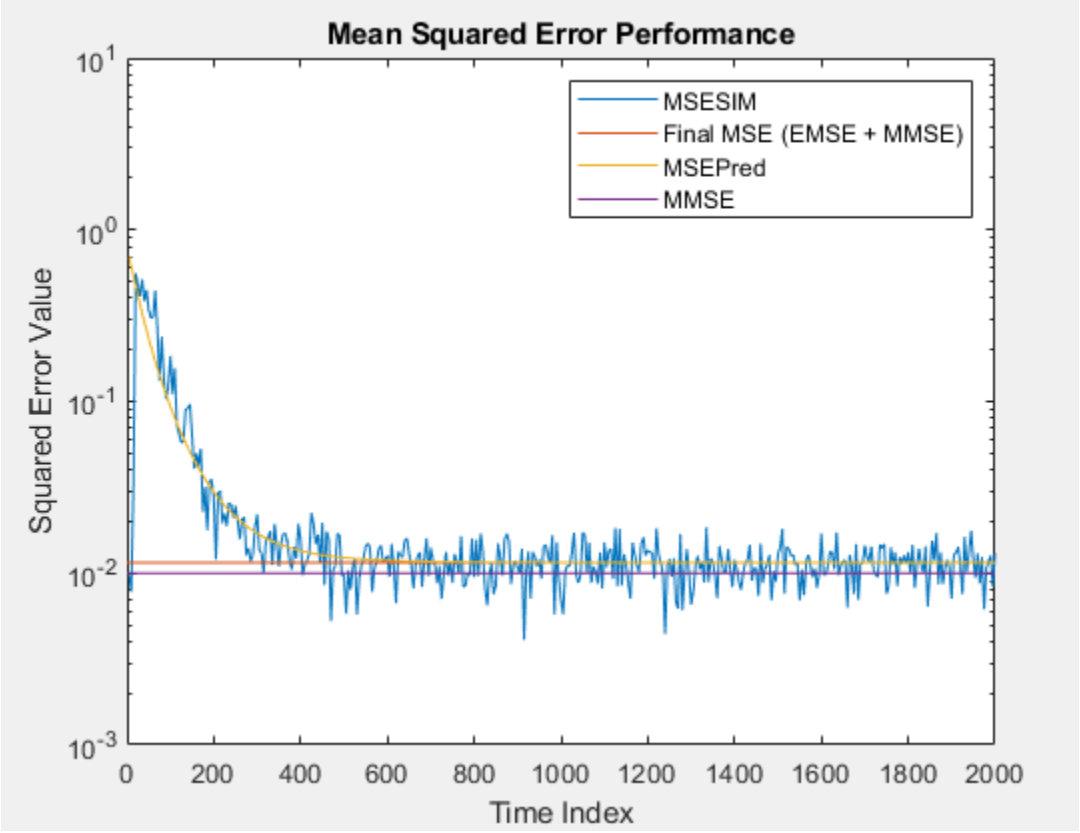
At steady state when the filter has adapted, the error between the filter output and the desired signal is minimal, not zero. This error is known as the steady state error. The speed with which the filter converges to the optimal state, known as the convergence speed, depends on multiple factors such 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 6-20. 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 6-26. 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.



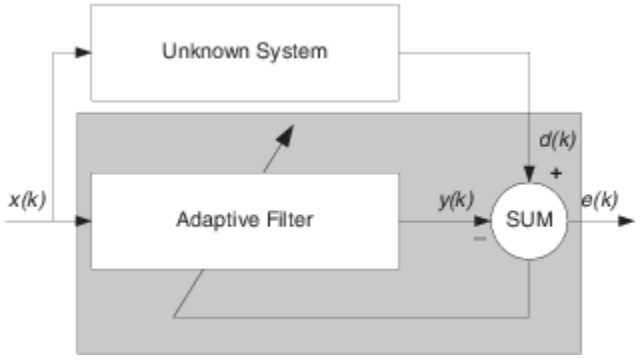
Common Applications

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.

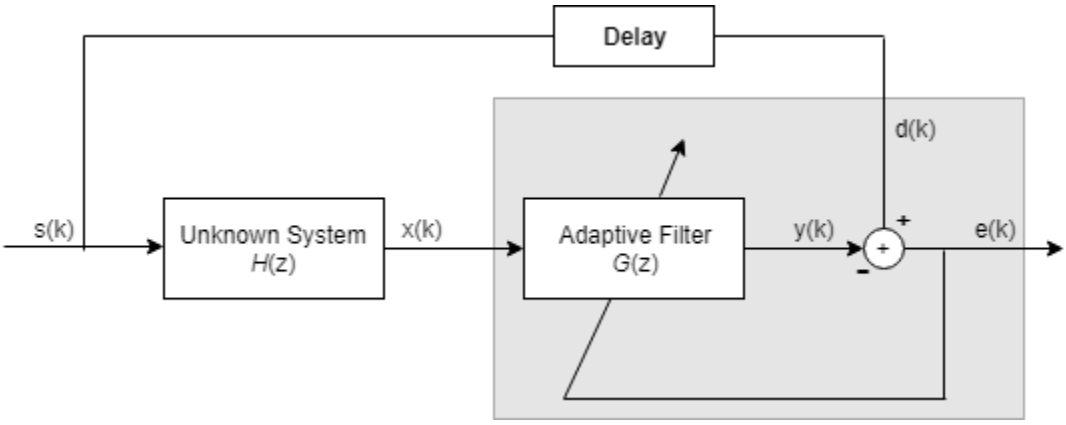
In the figure, the unknown system is placed in parallel with the adaptive filter. This layout represents just one of many possible structures. The shaded area contains the adaptive filter system.



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.



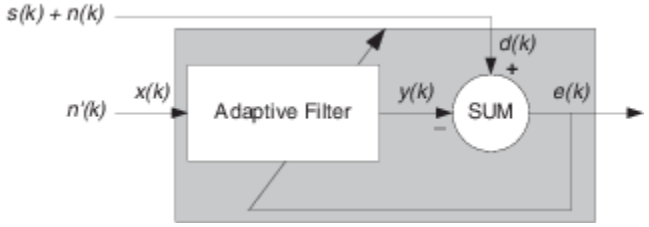
Including the delay to account for the delay caused by the unknown system prevents this condition.

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

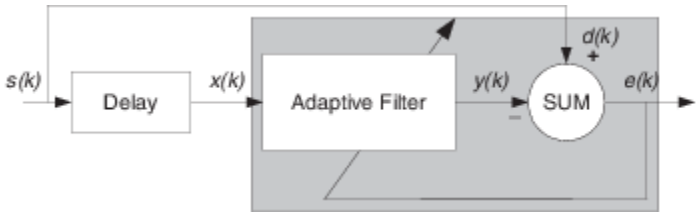
In noise cancellation, adaptive filters let you remove noise from a signal in real time. Here, the desired signal, the one to clean up, combines noise and desired information. To remove the noise, feed a signal *n'(k)* to the adaptive filter that is correlated to the noise to be removed from the desired 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 6-5. 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.

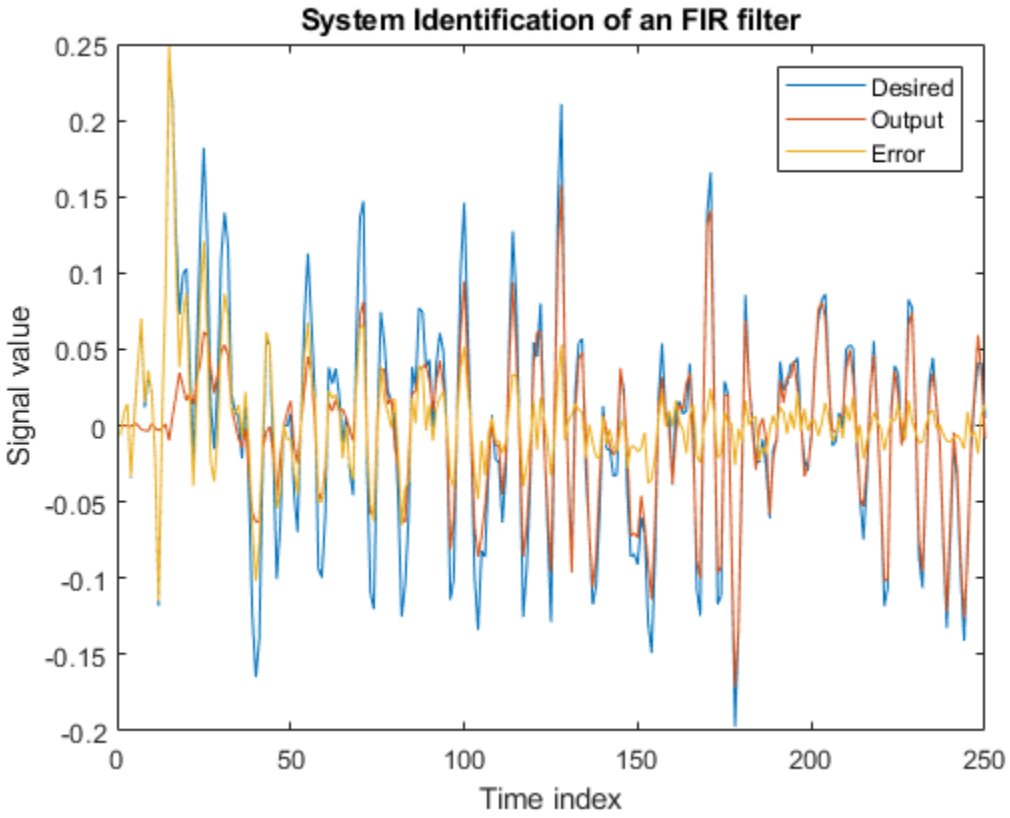
```
mu = 0.8;
lms = dsp.LMSFilter(13,'StepSize',mu)
```

```
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')
```

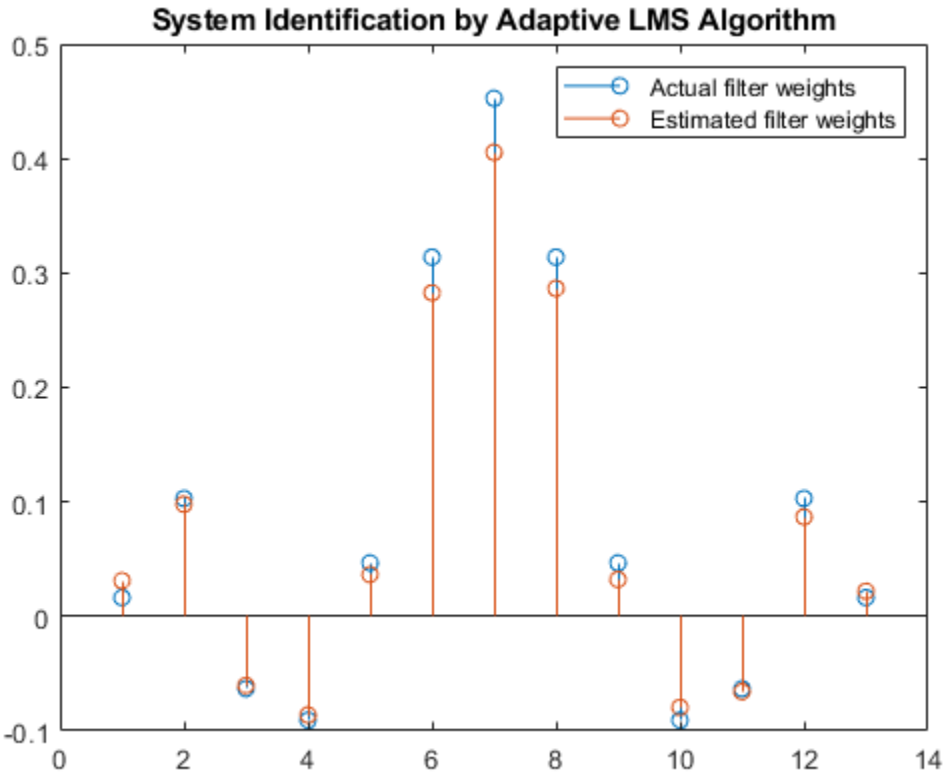


Compare the Weights

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.

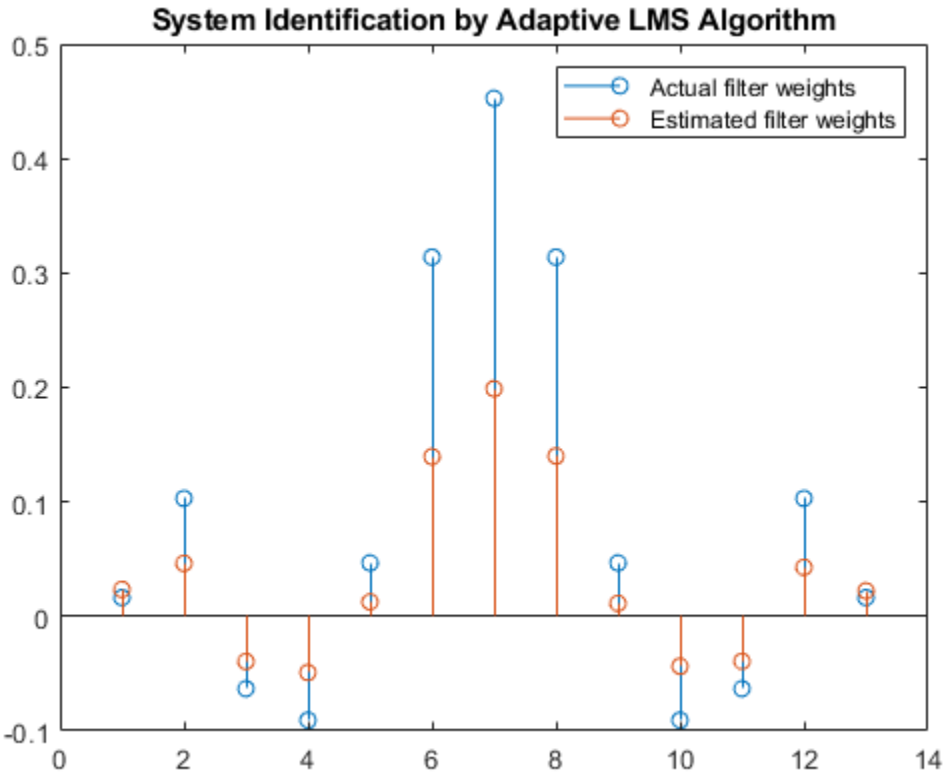
```
stem([(filt.Numerator).' w])
title('System Identification by Adaptive LMS Algorithm')
legend('Actual filter weights','Estimated filter weights',...
       'Location','NorthEast')
```



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.

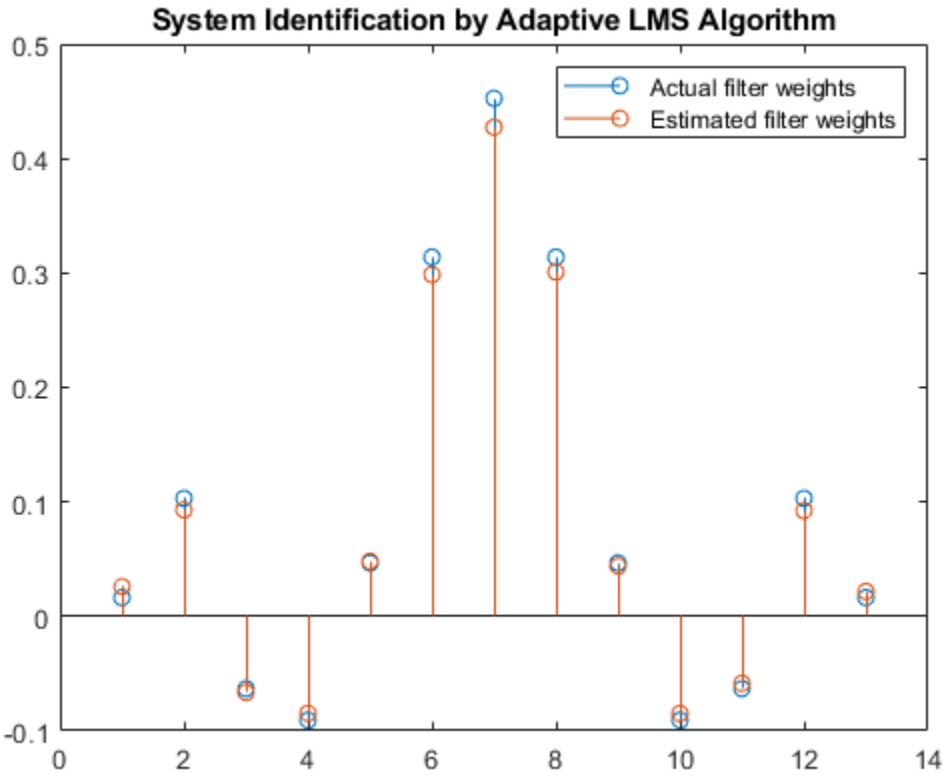
```
mu = 0.2;
lms = dsp.LMSFilter(13,'StepSize',mu);
[~,~,w] = lms(x,d);
stem([(filt.Numerator).' w])
title('System Identification by Adaptive LMS Algorithm')
legend('Actual filter weights','Estimated filter weights',...
       'Location','NorthEast')
```



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.

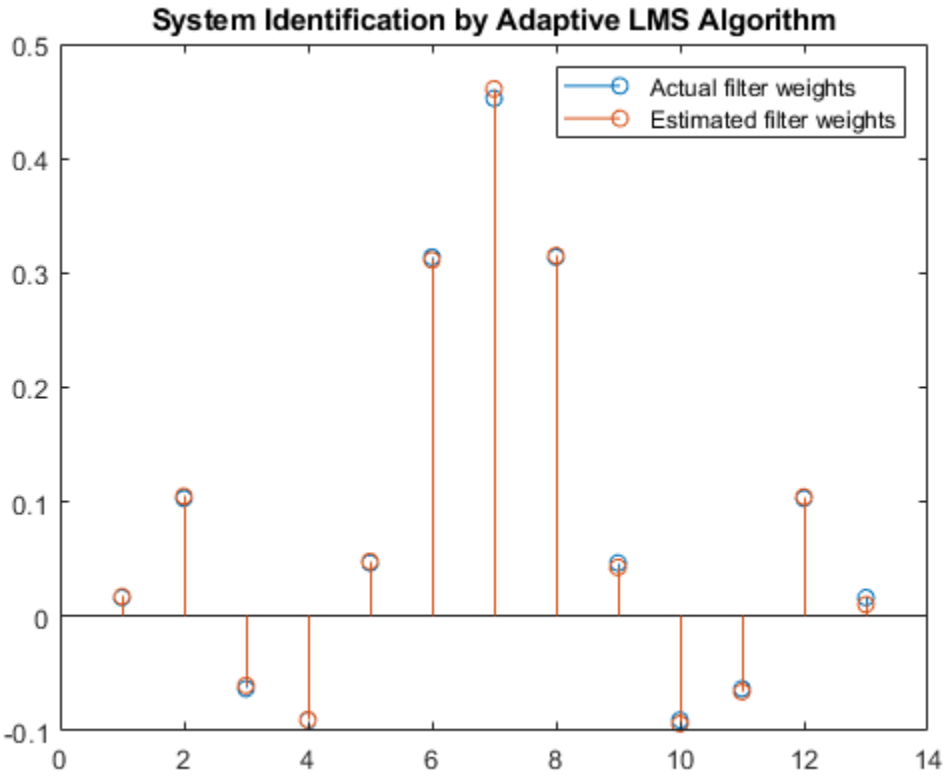
```
release(filt);
x = 0.1*randn(1000,1);
n = 0.01*randn(1000,1);
d = filt(x) + n;
[y,e,w] = lms(x,d);
stem([(filt.Numerator).' w])
title('System Identification by Adaptive LMS Algorithm')
legend('Actual filter weights','Estimated filter weights',...
       'Location','NorthEast')
```



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.

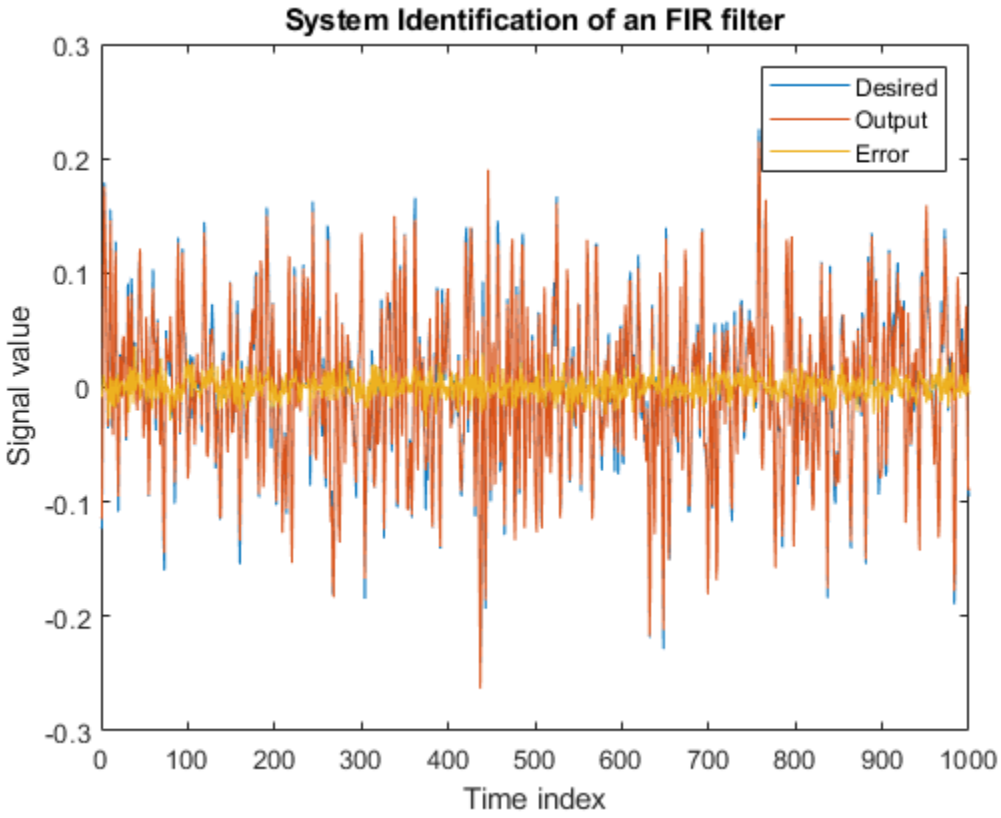
Compare the filter weights. The weights of the LMS filter match the weights of the FIR filter very closely, indicating a good convergence.

```
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 6-17
- "Compare Convergence Performance Between LMS Algorithm and Normalized LMS Algorithm" on page 6-20

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 6-9.

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.

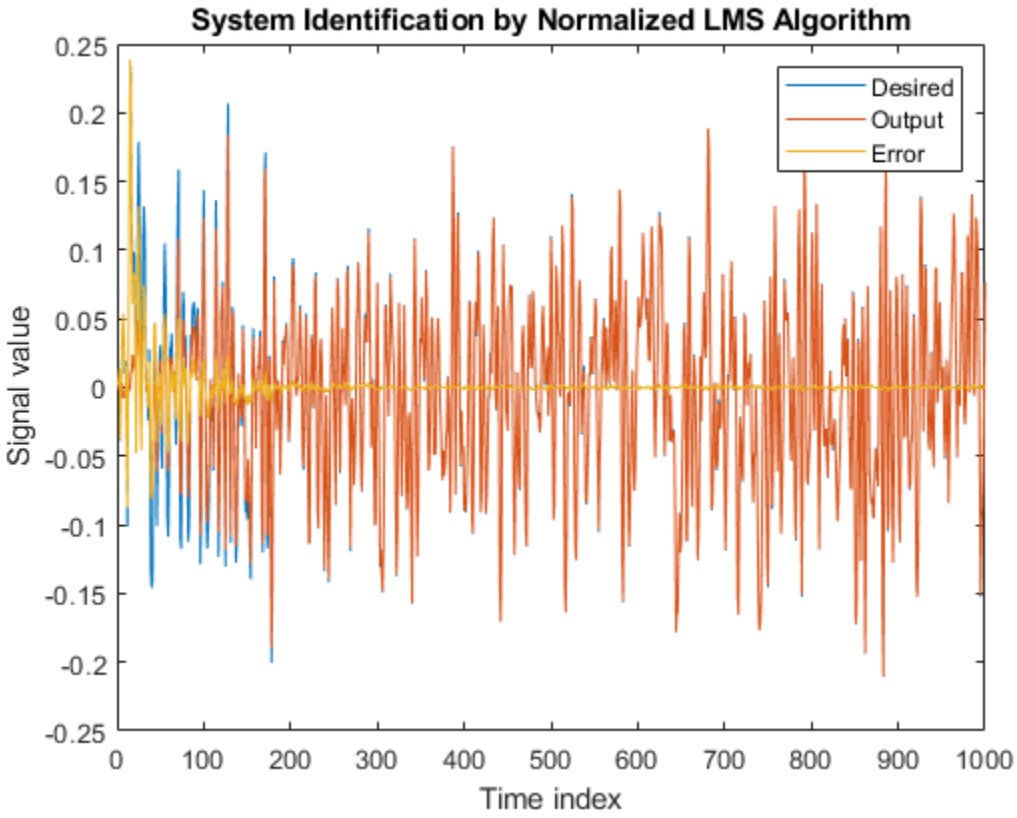
```
mu = 0.2;
lms = dsp.LMSFilter(13,'StepSize',mu,'Method',...
   'Normalized LMS');
```

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.

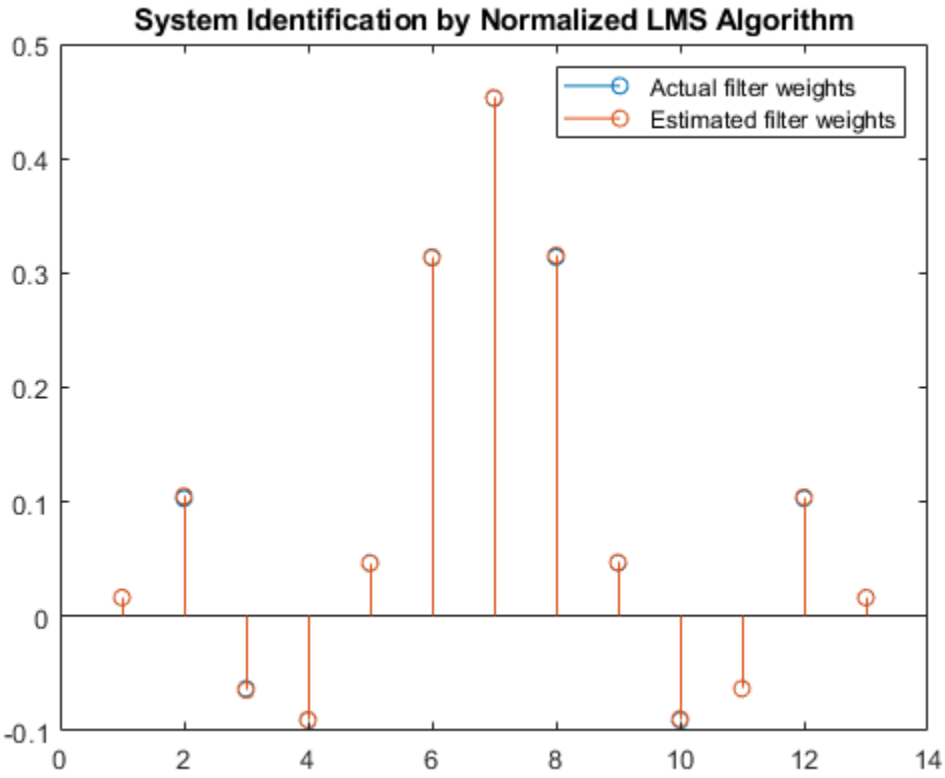
```
plot(1:1000, [d,y,e])
title('System Identification by Normalized LMS Algorithm')
legend('Desired','Output','Error')
xlabel('Time index')
ylabel('Signal value')
```



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 6-9
- “Compare Convergence Performance Between LMS Algorithm and Normalized LMS Algorithm” on page 6-20

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.

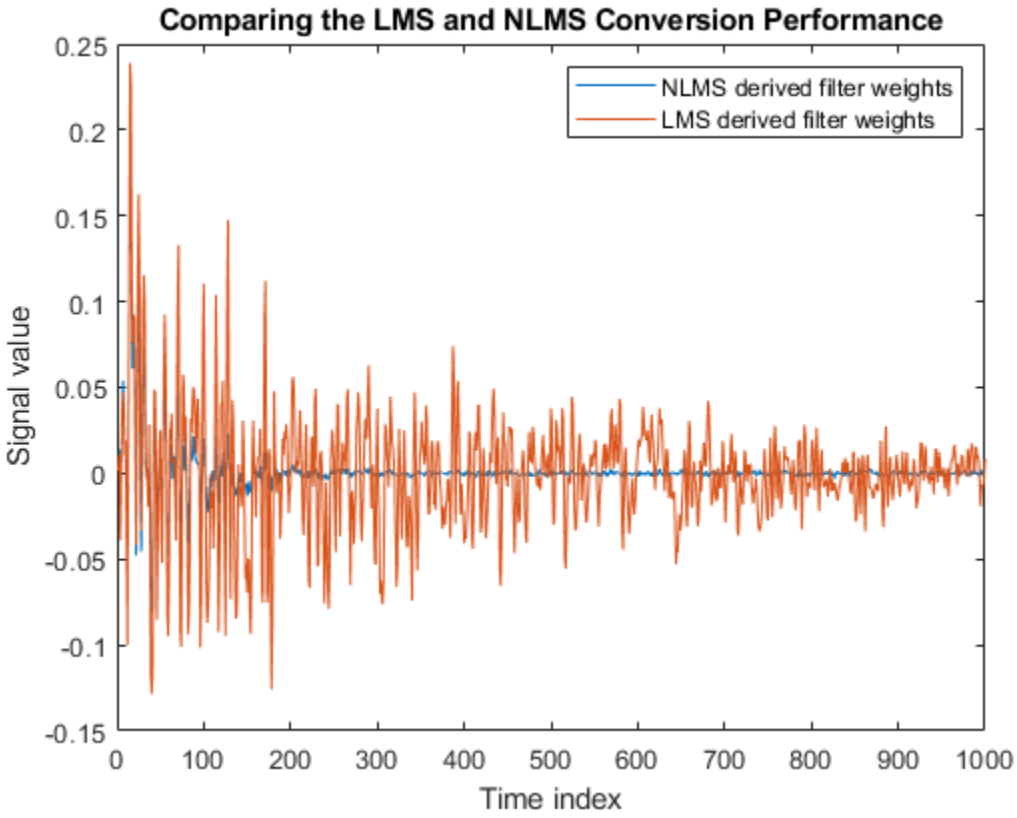
```
mu = 0.2;
lms_nonnormalized = dsp.LMSFilter(13,'StepSize',mu,...
    'Method','LMS');
lms_normalized = dsp.LMSFilter(13,'StepSize',mu,...
    'Method','Normalized LMS');
```

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 6-9
- “System Identification of FIR Filter Using Normalized LMS Algorithm” on page 6-17
- “Noise Cancellation Using Sign-Data LMS Algorithm” on page 6-22

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 6-7, 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 6-7, 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.

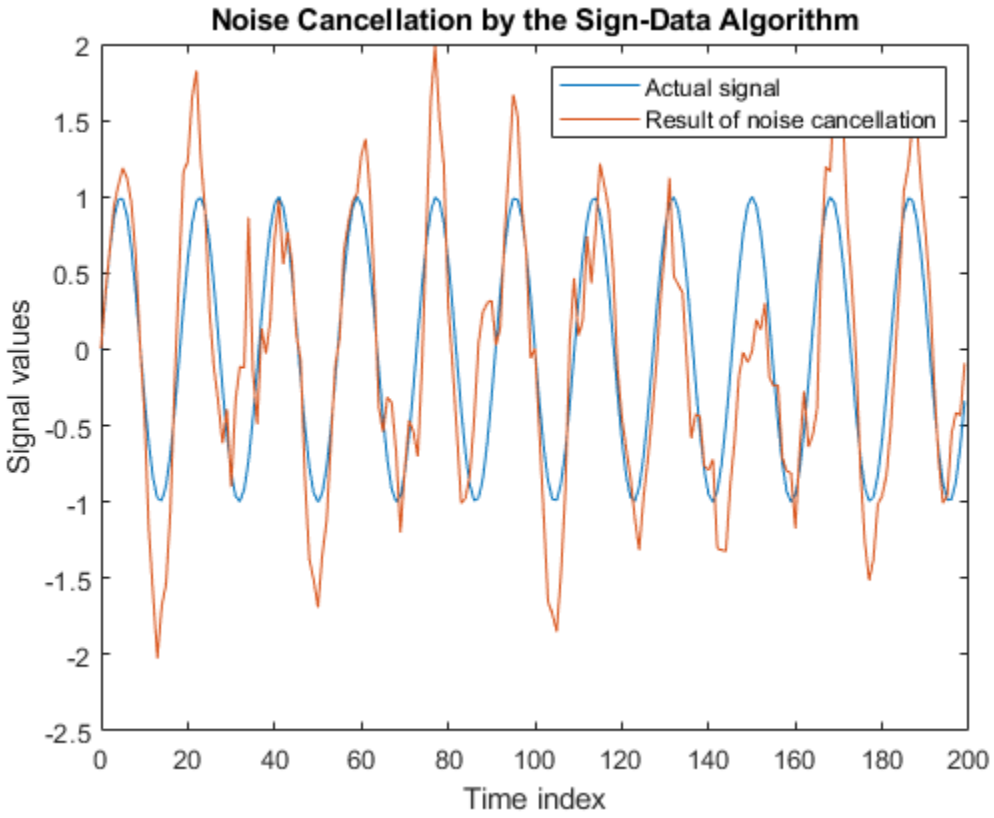
In “System Identification of FIR Filter Using LMS Algorithm”, you constructed a default filter that sets the filter coefficients to zeros. In most cases that approach does not work for the sign-data algorithm. The closer you set your initial filter coefficients to the expected values, the more likely 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 6-9

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 6-9
- “System Identification of FIR Filter Using Normalized LMS Algorithm” on page 6-17
- “Noise Cancellation Using Sign-Data LMS Algorithm” on page 6-22

- “Inverse System Identification Using RLS Algorithm” on page 6-29

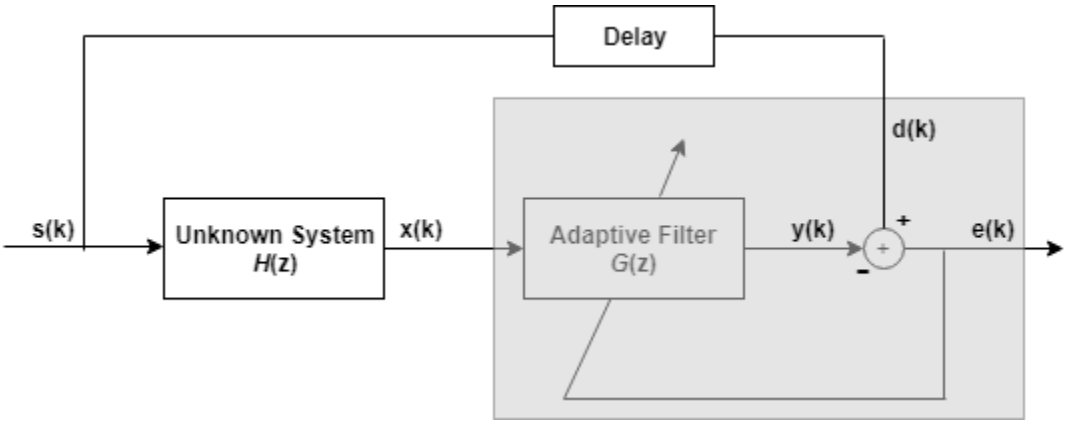
References

[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);
```

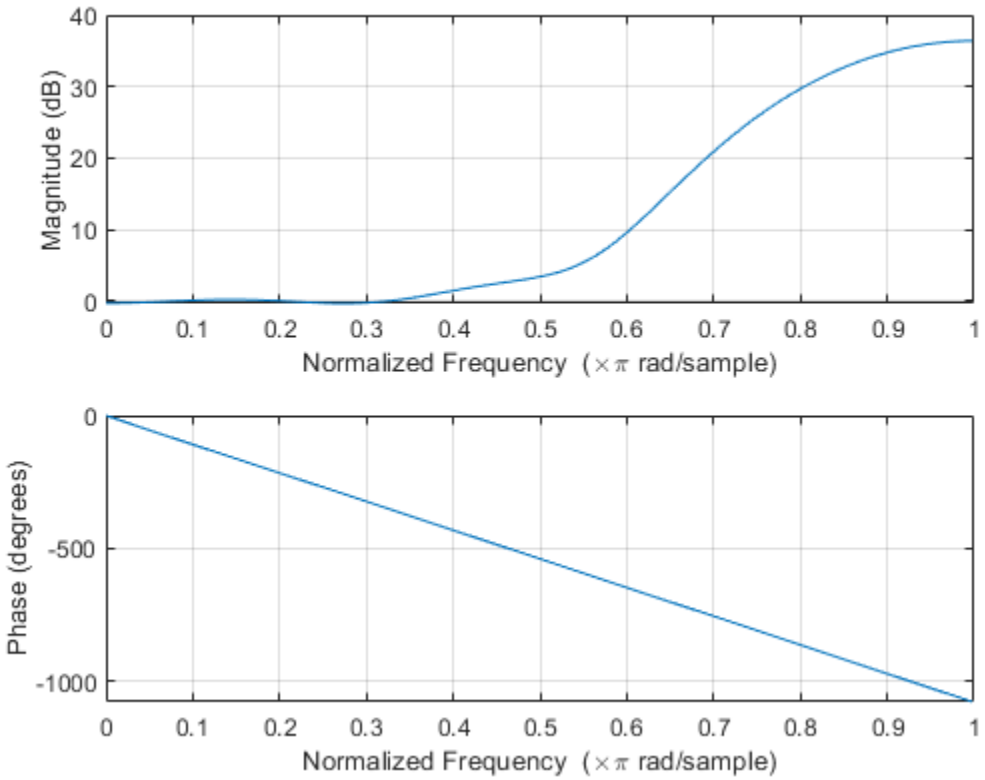
where `y` returns the filtered output and `e` contains the error signal as the filter adapts to find the inverse of the unknown system.

Obtain the estimated coefficients of the RLS filter.

```
b = rls.Coefficients;
```

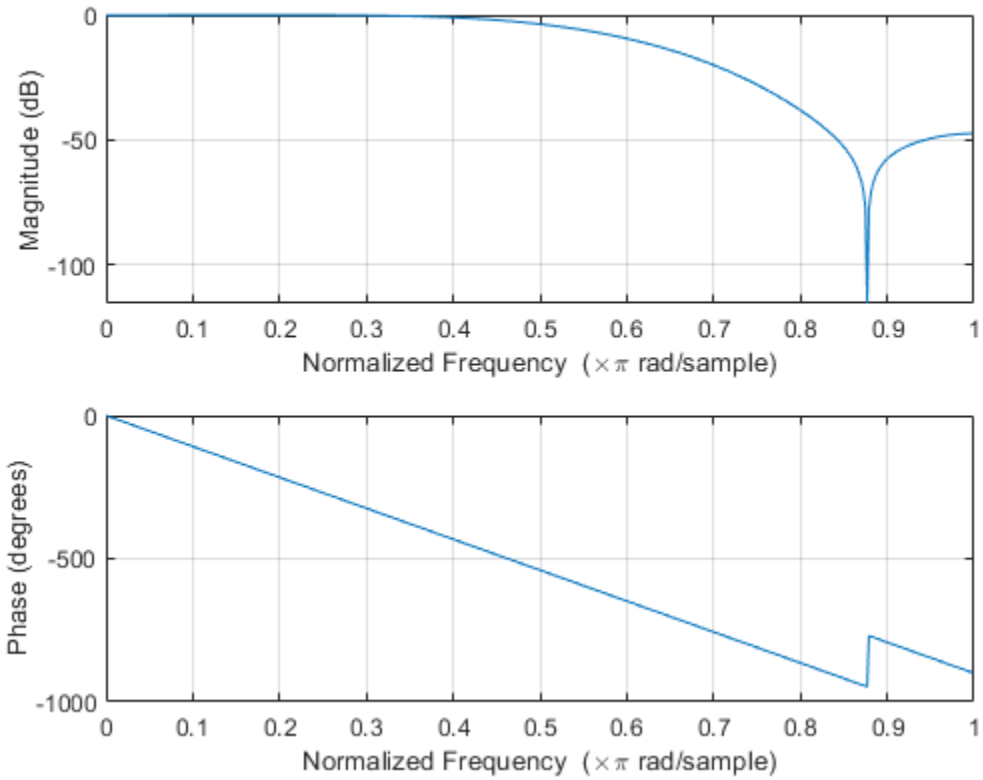
View the frequency response of the adapted RLS filter (inverse system, *G*(z)) using `freqz`. The inverse system looks like a highpass filter with linear phase.

```
freqz(b,1)
```



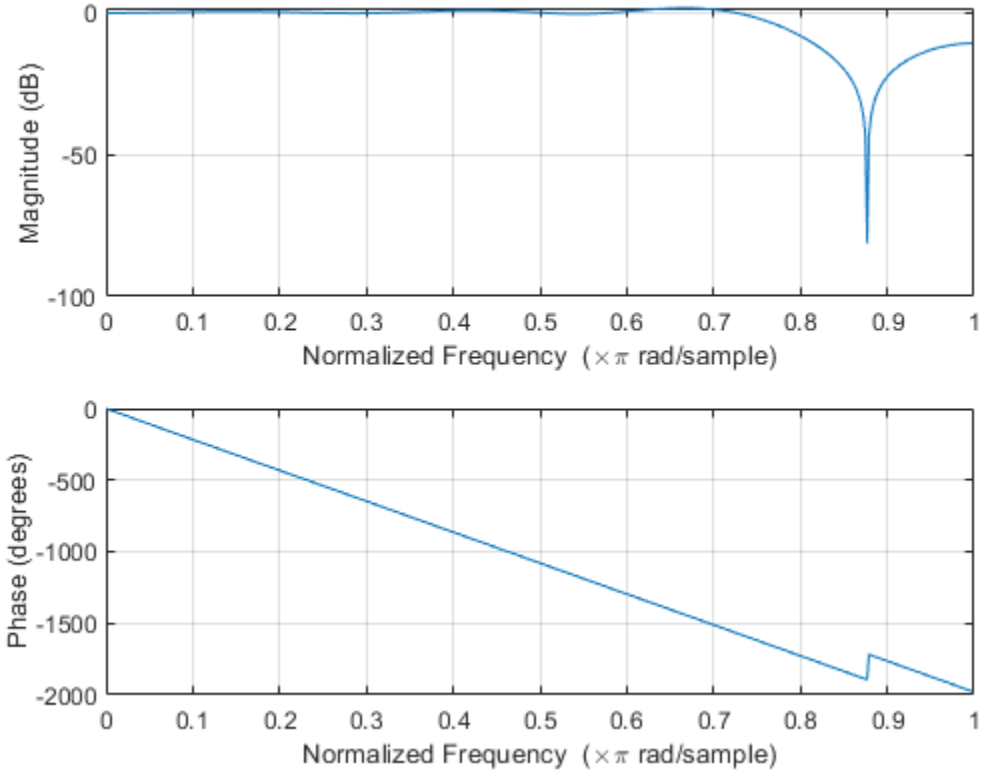
View the frequency response of the unknown system, $H(z)$. The response is that of a lowpass filter with a cutoff frequency of 0.55.

```
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 6-26
- “System Identification of FIR Filter Using LMS Algorithm” on page 6-9

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 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 per frame.

```
sine = dsp.SineWave('Frequency',375,'SampleRate',8000,'SamplesPerFrame',1000)
```

```
sine =
  dsp.SineWave with properties:

          Amplitude: 1
          Frequency: 375
        PhaseOffset: 0
      ComplexOutput: false
             Method: 'Trigonometric function'
         SampleRate: 8000
    SamplesPerFrame: 1000
     OutputDataType: 'double'
```

```
s = sine();
```

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(sine.SamplesPerFrame,1); % Random noise part.
ar = [1,1/2];          % Autoregression coefficients.
ARfilt = dsp.IIRFilter('Numerator',1,'Denominator',ar)
```

```
ARfilt =
  dsp.IIRFilter with properties:

            Structure: 'Direct form II transposed'
            Numerator: 1
          Denominator: [1 0.5000]
    InitialConditions: 0

  Show all properties
```

```
v1 = ARfilt(v);
```

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];
MAfilt = dsp.FIRFilter('Numerator',ma)
```

```
MAfilt =
  dsp.FIRFilter with properties:

            Structure: 'Direct form'
      NumeratorSource: 'Property'
            Numerator: [1 -0.8000 0.4000 -0.2000]
    InitialConditions: 0

  Show all properties
```

```
v2 = MAfilt(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')
```

```
lms =
  dsp.LMSFilter with properties:

                  Method: 'LMS'
                  Length: 7
          StepSizeSource: 'Property'
                StepSize: 0.1000
            LeakageFactor: 1
        InitialConditions: 0
           AdaptInputPort: false
    WeightsResetInputPort: false
            WeightsOutput: 'Last'

  Show all properties
```

```
nlms = dsp.LMSFilter(L,'Method','Normalized LMS')
```

```
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
```

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 = 0.2127

mumaxmselms = 0.1312

[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: 0.0044
           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.

```
[~,elms,wlms] = lms(v2,x);
[~,enlms,wnlms] = nlms(v2,x);
```

Compute the Optimal Solution

For comparison, compute the optimal FIR Wiener filter.

```
reset(MAfilt);
bw = firwiener(L-1,v2,x); % Optimal FIR Wiener filter
MAfilt = dsp.FIRFilter('Numerator',bw)

MAfilt =
  dsp.FIRFilter with properties:

            Structure: 'Direct form'
      NumeratorSource: 'Property'
            Numerator: [1.0001 0.3060 0.1050 0.0482 0.1360 0.0959 0.0477]
    InitialConditions: 0

  Show all properties


yw = MAfilt(v2); % Estimate of x using Wiener filter
ew = x - yw; % Estimate of actual sinusoid
```

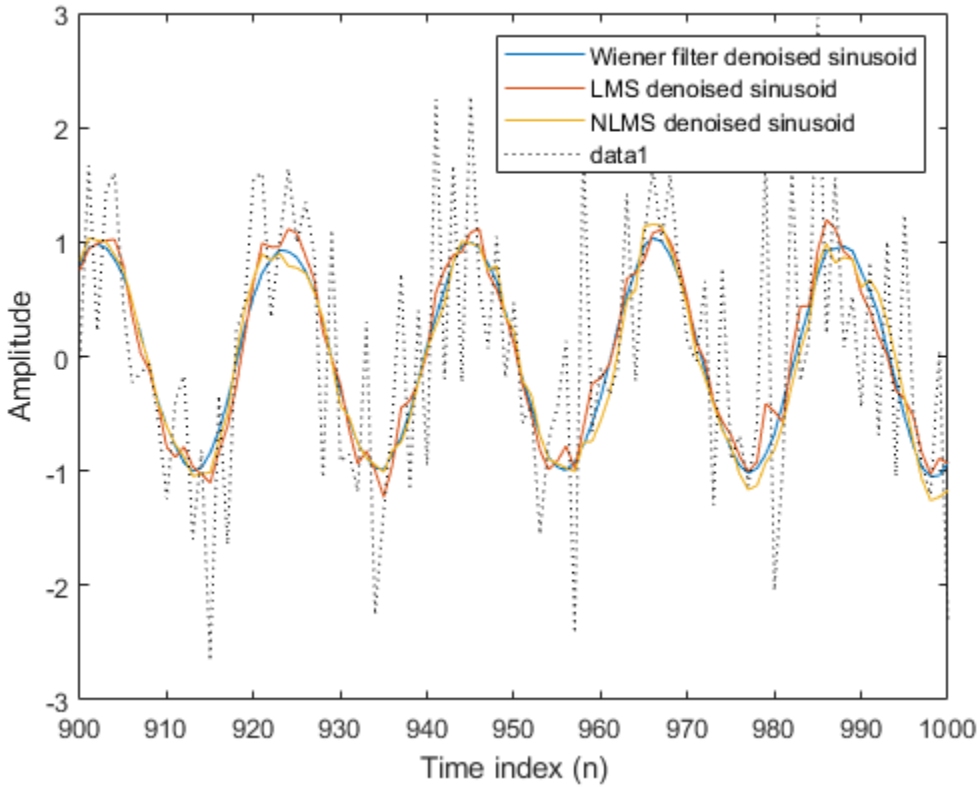
Plot the Results

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.

```
n = (1:1000)';
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

    1.0001    0.8644    0.9690
    0.3060    0.1198    0.2661
    0.1050   -0.0020    0.1226
    0.0482   -0.0046    0.1074
```

```
    0.1360    0.0680    0.2210
    0.0959    0.0214    0.1940
    0.0477    0.0292    0.1127
```

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.

```
reset(ARfilt)
reset(sine);
release(sine);
n = (1:5000)';
sine.SamplesPerFrame = 5000
```

```
sine =
  dsp.SineWave with properties:

          Amplitude: 1
          Frequency: 375
        PhaseOffset: 0
      ComplexOutput: false
             Method: 'Trigonometric function'
         SampleRate: 8000
    SamplesPerFrame: 5000
     OutputDataType: 'double'
```

```
s = sine();
nr = 25;
v = 0.8*randn(sine.SamplesPerFrame,nr);
ARfilt = dsp.IIRFilter('Numerator',1,'Denominator',ar)
```

```
ARfilt =
  dsp.IIRFilter with properties:

            Structure: 'Direct form II transposed'
            Numerator: 1
          Denominator: [1 0.5000]
    InitialConditions: 0

  Show all properties
```

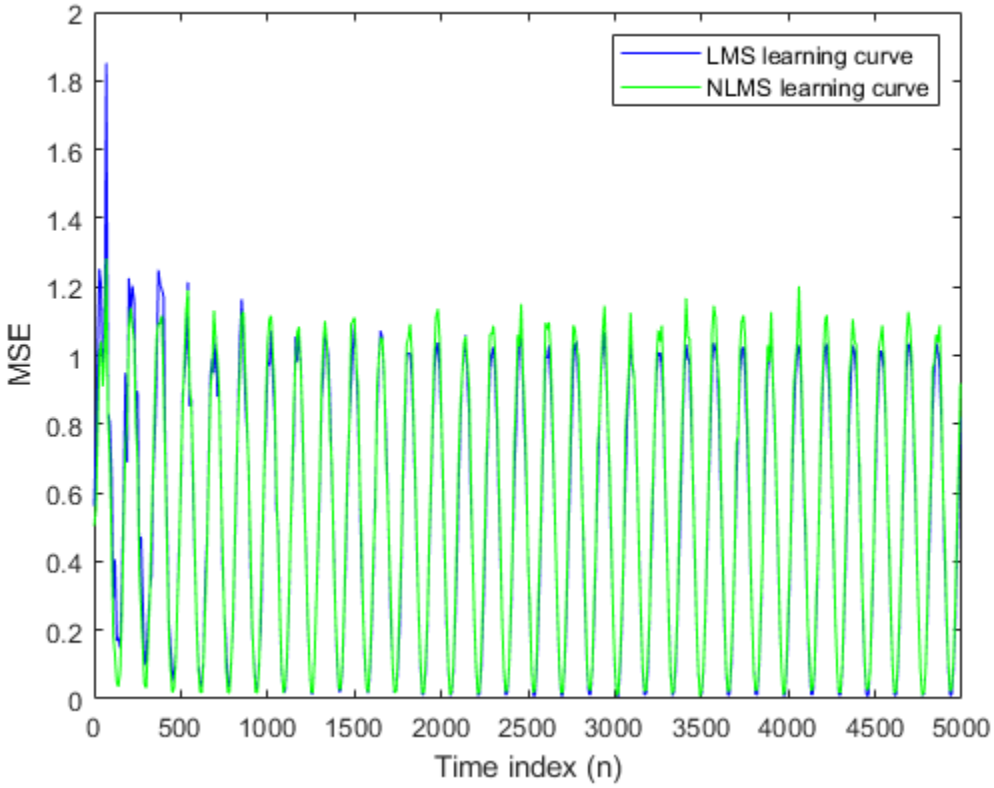
```
v1 = ARfilt(v);
x = repmat(s,1,nr) + v1;
reset(MAfilt);
MAfilt = dsp.FIRFilter('Numerator',ma)

MAfilt =
  dsp.FIRFilter with properties:

            Structure: 'Direct form'
      NumeratorSource: 'Property'
            Numerator: [1 -0.8000 0.4000 -0.2000]
    InitialConditions: 0

  Show all properties


v2 = MAfilt(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,'b',1:M:n(end),msenlms,'g')
legend('LMS learning curve','NLMS learning curve')
xlabel('Time index (n)')
ylabel('MSE')
```



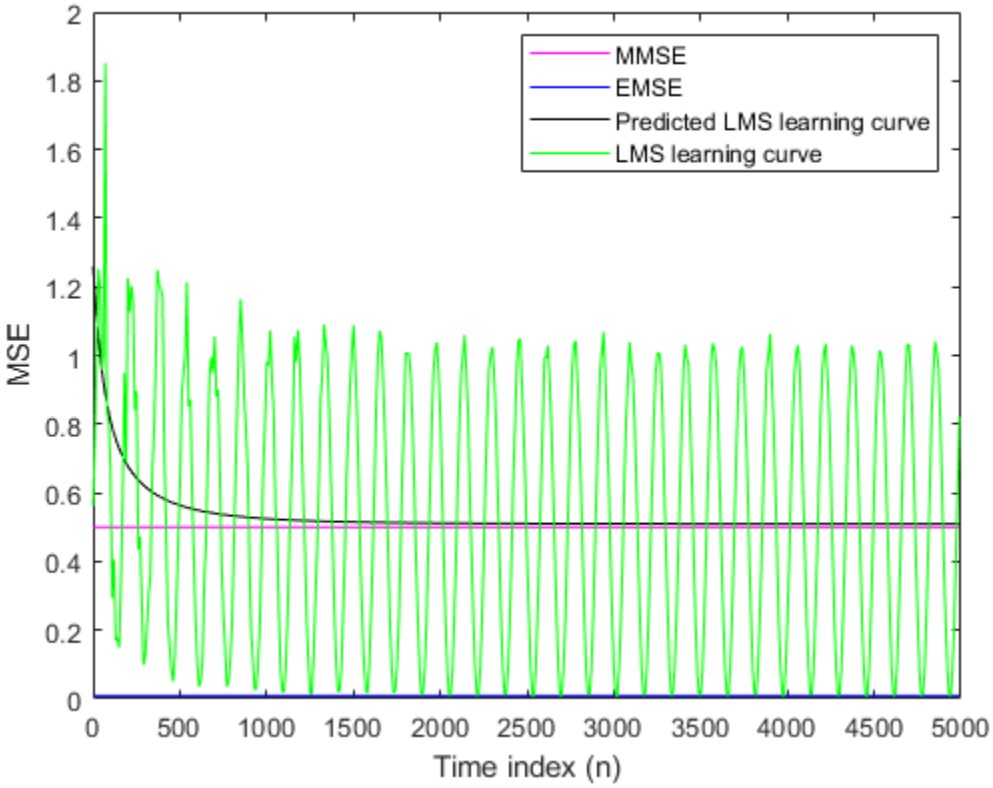
In this plot you see the calculated learning curves for the LMS and NLMS adaptive filters.

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
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
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 6-43
“LMS Filter Configuration for Adaptive Noise Cancellation” on page 6-44
“Modify Adaptive Filter Parameters During Model Simulation” on page 6-47

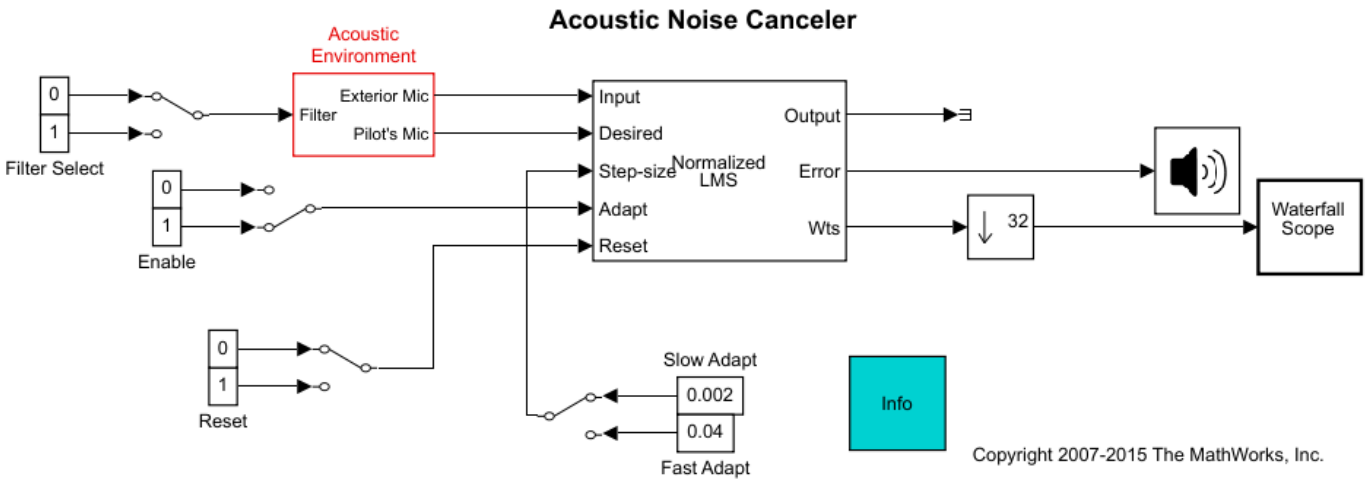
Create an Acoustic Environment in Simulink

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 6-43, 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 6-43 is not open on your desktop, you can open an equivalent model by typing

   ```
   ex_adapt1_audio
   ```

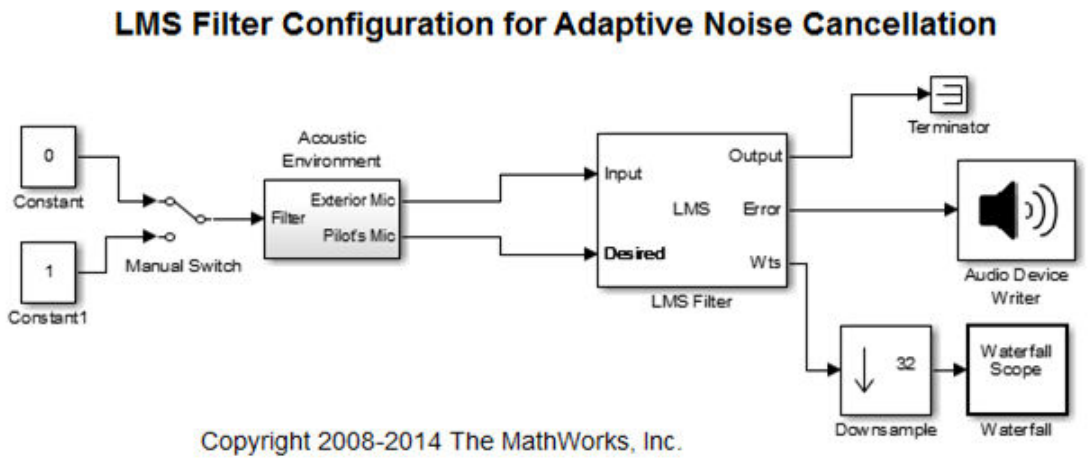
   at the MATLAB command prompt.
2. From the DSP System Toolbox Filtering library, and then from the Adaptive Filters library, click-and-drag an LMS Filter block into the model that contains the Acoustic Environment subsystem.
3. Double-click the LMS Filter block. Set the block parameters as follows, and 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.



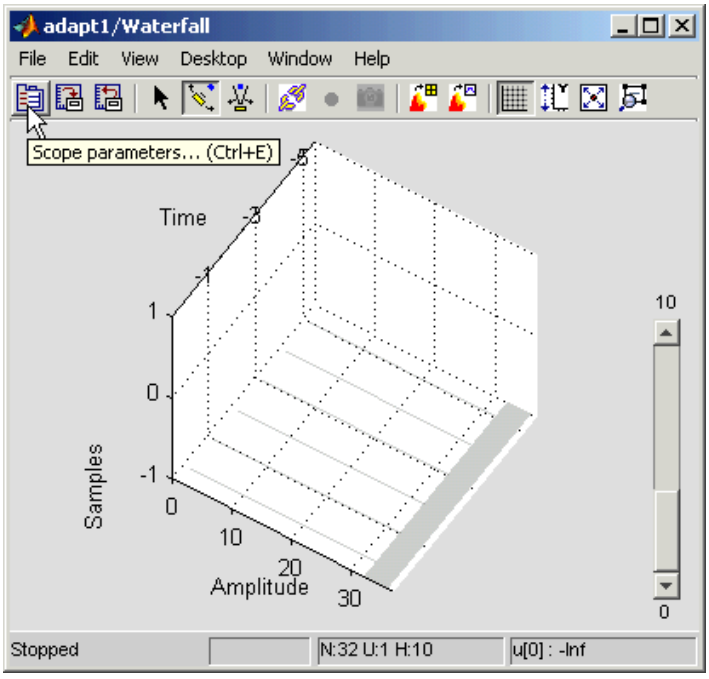
6 Double-click the Constant block. Set the **Constant value** parameter to `0` and then click **OK**.

7 Double-click the Downsample block. Set the **Downsample factor, K** parameter to 32. Click **OK**.

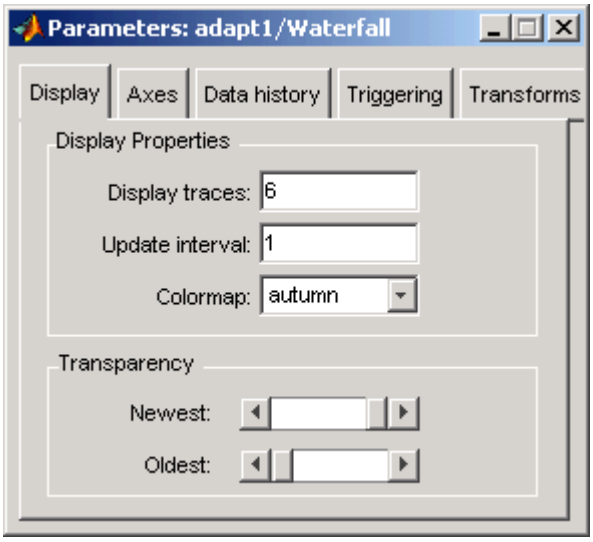
The filter weights are being updated so often that there is very little change from one update to the next. To see a more noticeable change, you need to downsample the output from the Wts port.

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`

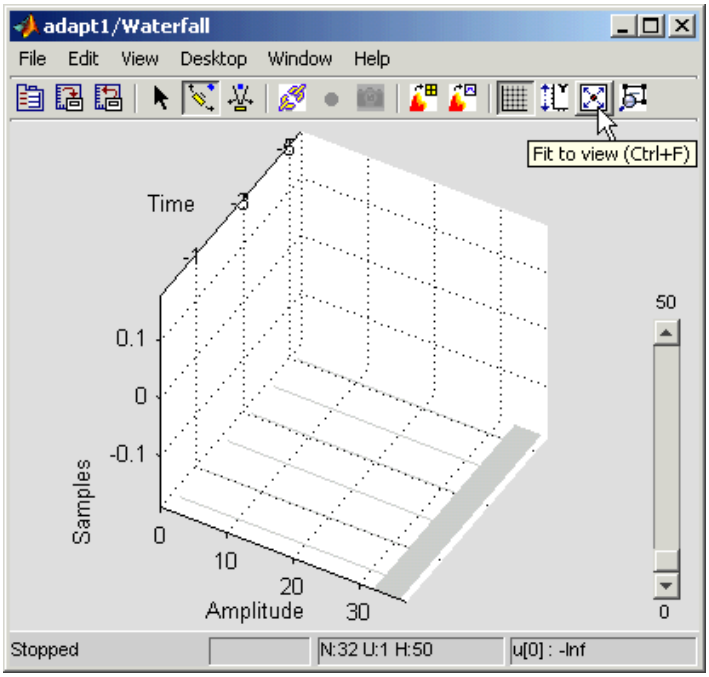
11 Click the **Data history** tab. Set the parameters as follows:

- **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 6-47, 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 6-44, 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 6-43 is not open on your desktop, you can open an equivalent model by typing

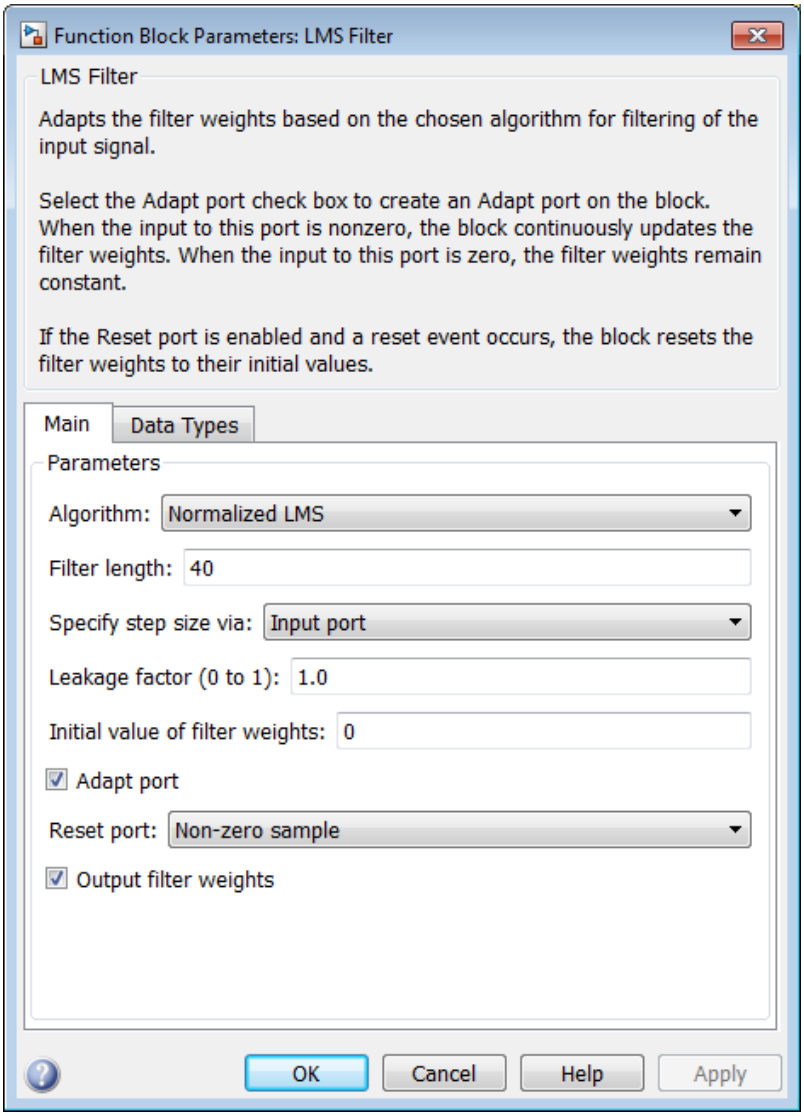
```
ex_adapt2_audio
```

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.

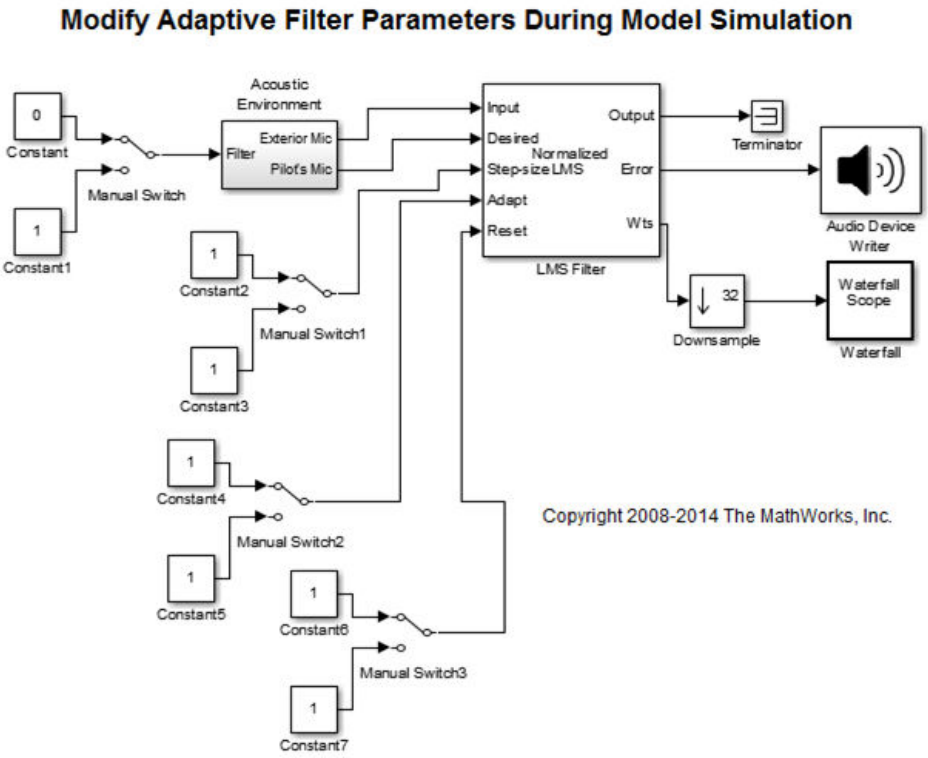


Step-size, Adapt, and Reset ports appear on the LMS Filter block.

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 Channel Estimation Through Adaptive Filtering” on page 4-278
- “Adaptive Noise Cancellation Using RLS Adaptive Filtering” on page 4-259

Multirate and Multistage Filters

Learn how to analyze, design, and implement multirate and multistage filters in MATLAB and Simulink.

- “Overview of Multirate Filters” on page 7-2
- “Overview of Multistage Filters” on page 7-11
- “Overview of Filter Banks” on page 7-14
- “Two-Channel Filter Bank Using Halfband Decimators and Halfband Interpolators” on page 7-19
- “Channelize and Synthesize Sine Wave in MATLAB” on page 7-26
- “Multilevel Filter Banks” on page 7-28

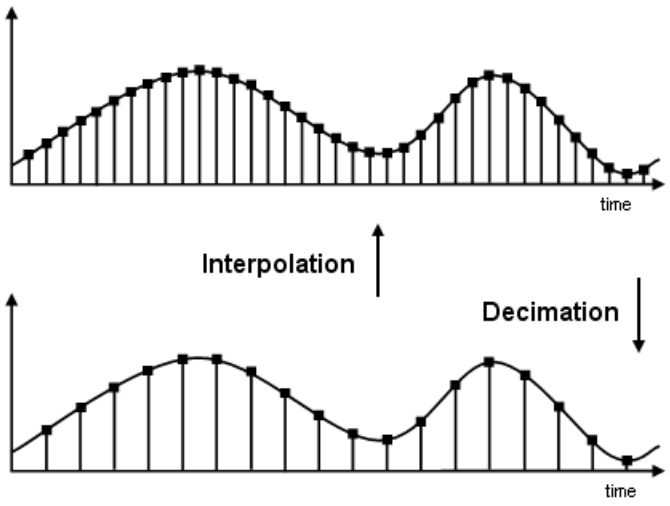
Overview of Multirate Filters

In this section...
“Decimation and Interpolation” on page 7-2
“Decimators” on page 7-3
“Interpolators” on page 7-4
“Sample Rate Converters” on page 7-6

Multirate filters are digital filters that change the sample rate of a digital signal without introducing aliasing or imaging in the rate-converted signal. These filters are categorized as decimators that reduce the sample rate, interpolators that increase the sample rate, and rate converters that do a combination of both. DSP System Toolbox offers several MATLAB System objects and Simulink blocks which implement decimators, interpolators, and rate converters. Advanced filter technologies such as channelizers, channel synthesizers, two-channel halfband filter banks, and multilevel filter banks use these filters as building components.

Decimation and Interpolation

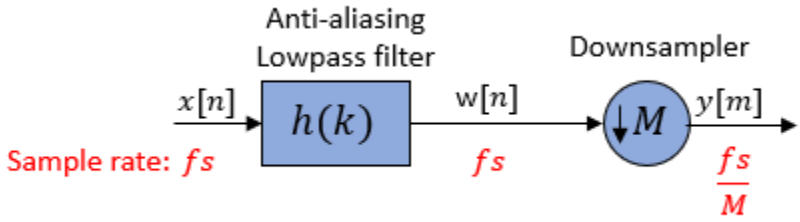
A filter that reduces the input rate is called a *decimator*. A filter that increases the input rate is called an *interpolator*. The process of decimation reduces the sample rate by compressing the data, retaining only the desired information. Interpolation, on the other hand, increases the sample rate of the signal. Interpolation is useful, for example, when you need to feed a signal to a system operating at a higher rate. 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 *M* is 2. The process of decimation and interpolation allow sample rate to be decreased or increased while minimizing undesirable effects of errors such aliasing and imaging.

Decimators

Conceptually, the decimator consists of an anti-aliasing lowpass filter followed by a downsampler. The role of the lowpass filter is to bandlimit the input signal before downsampling it, so that the Nyquist sampling theorem is satisfied.



The downsampler reduces the sample rate from *fs* to *fs*/*M*. To prevent aliasing at the lower rate of *fs*/*M*, the decimator uses a lowpass filter prior to the downsampler. The lowpass filter bandlimits the input signal to less than *fs*/2*M*. This bandlimiting operation makes sure that the Nyquist criterion for sampling is satisfied, and therefore guarantees near perfect reconstruction of the filtered signal *w*[*n*]. If the Nyquist criterion is not satisfied, aliasing occurs and the signal cannot be perfectly reconstructed.

Accordiing to the Nyquist theorem, for bandlimited signals, the sample rate 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 sample rate of 60 MHz, the highest frequency that can be handled by the system without aliasing is 60/2 = 30 MHz, which is greater than 10 MHz. You can safely set $M = 2$ in this case, since (60/2)/2 = 15 MHz, which is still greater than 10 MHz.

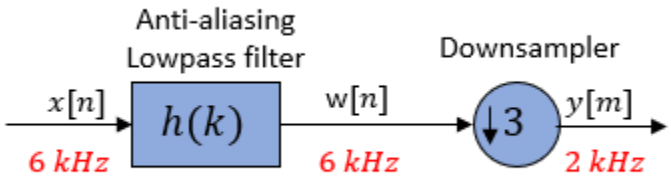
For the lowpass filter to be anti-aliasing, the filter must satisfy the following requirements:

- Stopband must contain the frequency range $fs/2M \le f \le fs/2$.
- Passband must be contained in the frequency range $0 \le f < fs/2M$.
- To not lose any information in the decimation process, the highest frequency of interest in the original signal *fp* must be less than *fs*/2*M*. This condition when not satisfied results in aliasing.

Effect on the Time and Frequency Domain

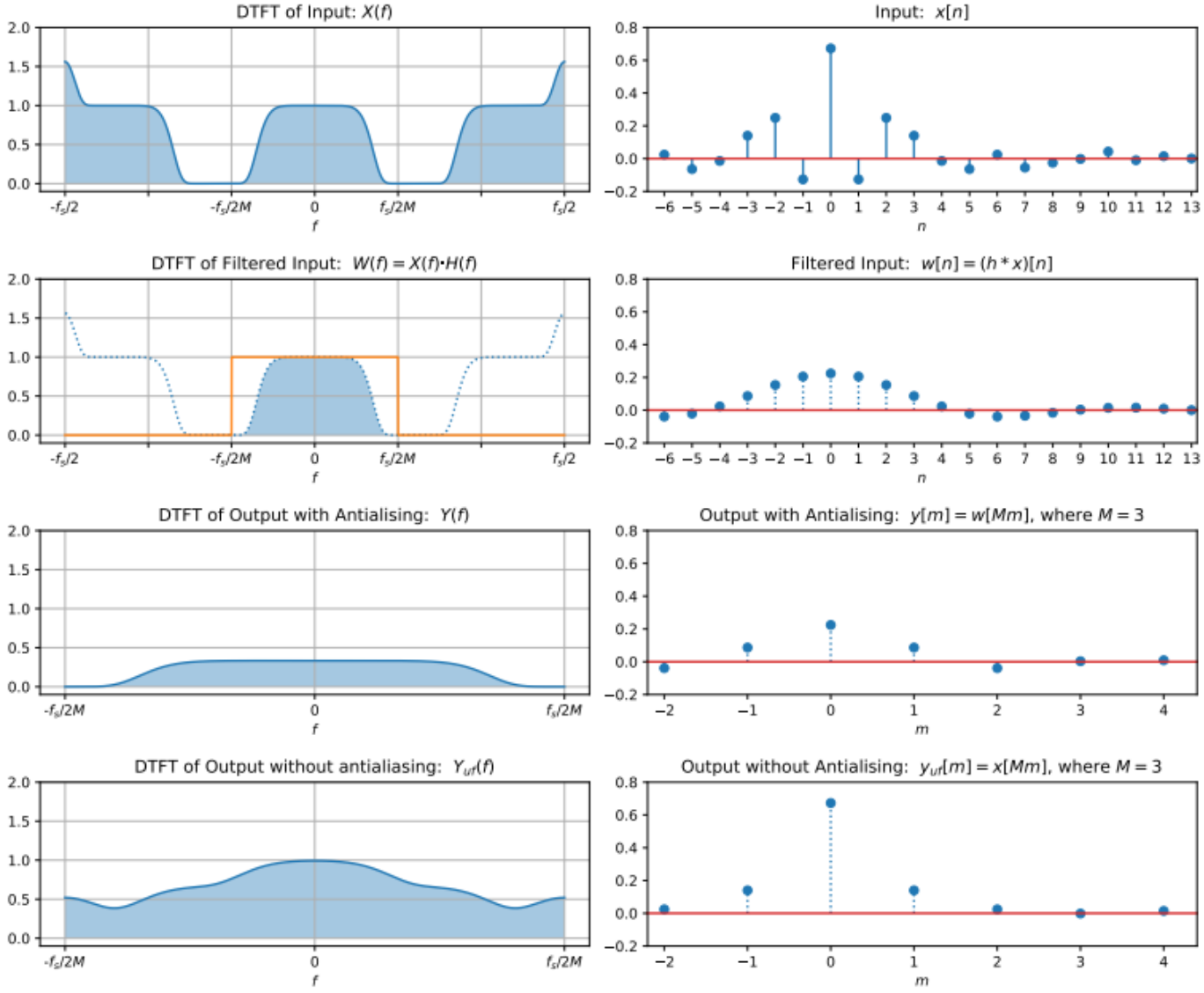
Consider a decimator with a downsample factor of 3.

Pass the input signal *x*[*n*] through an anti-aliasing lowpass filter to yield *w*[*n*]. The downsampler that follows reduces the filtered data by a factor of 3 by discarding 2 samples for every 3 samples to yield *y*[*m*].



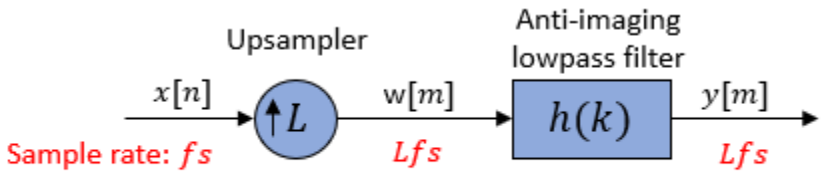
In the frequency domain, the input signal with a sample rate of 6 kHz first passes through the lowpass filter. The lowpass filter bandlimits the signal and removes image frequencies, which would otherwise cause aliasing. The downsampler that follows downsamples the filtered signal by a factor of 3 to a sample rate of 2 kHz.

The spectral components in the dashed lines are the frequencies that would have been aliased if the input signal $x[n]$ was not bandlimited by the lowpass filter. The spectrum clearly shows the role of lowpass filtering in avoiding aliasing.



Interpolators

Conceptually, an interpolator consists of an upsampler followed by an anti-imaging lowpass filter. The role of the lowpass filter is to remove the image frequencies caused by the upsampler.



The upsampler inserts $L - 1$ zero-valued samples to form the new signal $w[m]$ at a rate of Lfs. This rate increase creates image frequencies beyond the Nyquist interval. To remove these image frequencies, pass the $w[m]$ signal through an anti-imaging lowpass filter. The lowpass filter bandlimits the $w[m]$ signal to $fs/2$ or less. The highest valid frequency after you increase the rate to Lfs is $Lfs/2$. To eliminate spectral images, use the lowpass filter to bandlimit the upsampled signal to $Lfs/2L$ or $fs/2$.

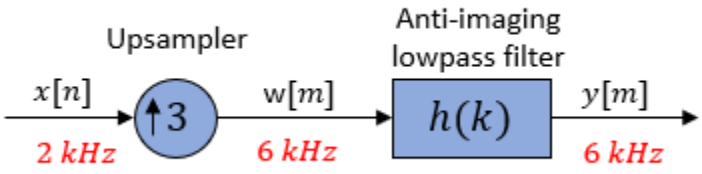
Based on these constraints, the lowpass filter must satisfy the following filter requirements:

- Stopband must contain the frequency range $fs/2 \le f \le Lfs/2$.
- Passband must be contained in the frequency range $0 \le f < fs/2$.
- A gain of L in the passband of the filter to compensate for the amplitude reduction by the interpolation process. Insertion of $L - 1$ zeros spreads the average energy of each signal sample over L output samples. This effectively attenuates each sample by a factor of L. To compensate for this attenuation, each sample of the output $y[m]$ needs to be multiplied by L.

Effect on the Time and Frequency Domain

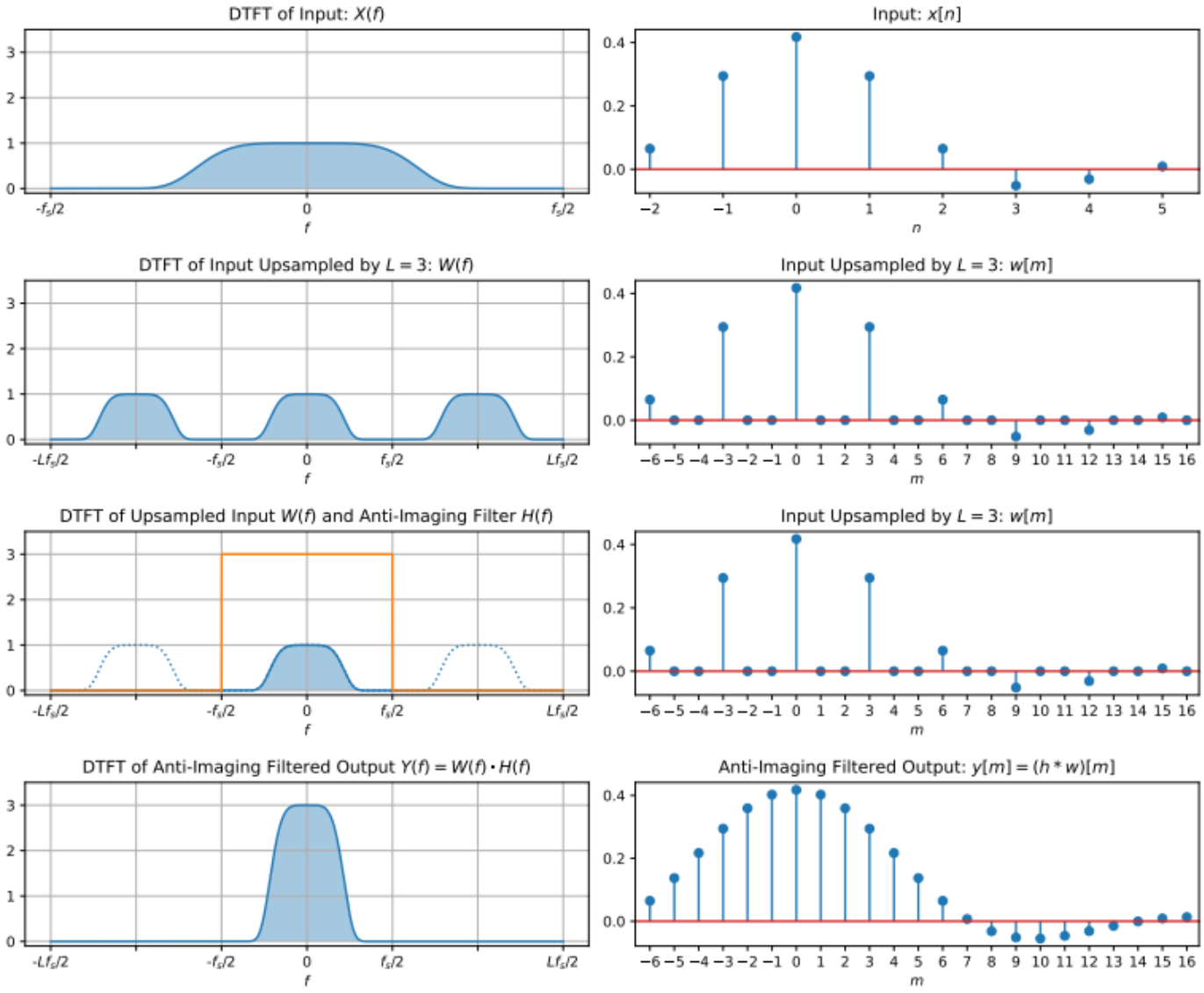
Consider an interpolator with an upsample factor of 3.

Pass the input signal through an upsampler. The upsampler inserts two zero-valued samples to form a new signal $w[m]$. Lowpass filter the signal to remove image frequencies created by the rate increase to yield the $y[m]$ signal.



In the frequency domain, pass the input signal with a sample rate of 2 kHz through an upsampler. The sample rate changes to 6 kHz. The lowpass filter that follows bandlimits the signal and removes image frequencies which would otherwise cause aliasing.

The spectral components in the dashed lines are the frequencies removed by the anti-imaging lowpass filter.

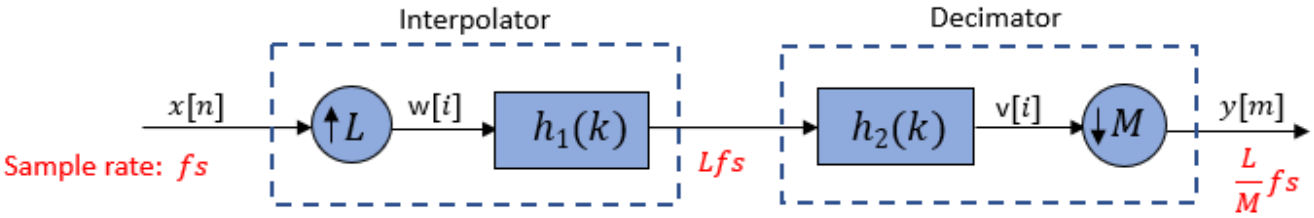


For more information about the effects of decimation and interpolation on a sampled signal, see “References” on page 7-9.

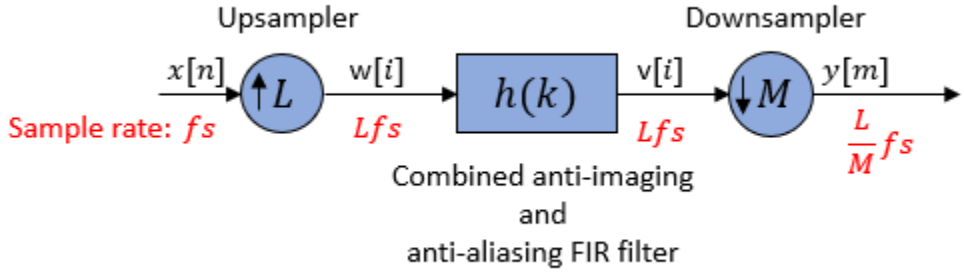
Sample Rate Converters

Sample rate converters change the sample rate of a signal by a noninteger factor, L/M. The sample rate change is achieved by first interpolating the data by L and then decimating by M. The interpolation precedes decimation because in that order the overall process is not as lossy.

Conceptually, a sample rate converter can be represented using the following diagram.



You can combine the two lowpass filters into a single filter since they are in cascade and have a common sample rate. The cascaded filter $h(k)$ must be both anti-aliasing and anti-imaging in order to effectively rate convert the signal without introducing significant aliasing and imaging errors. The combined filter has a cutoff frequency of `min`($fs/2$, $Lfs/2M$).



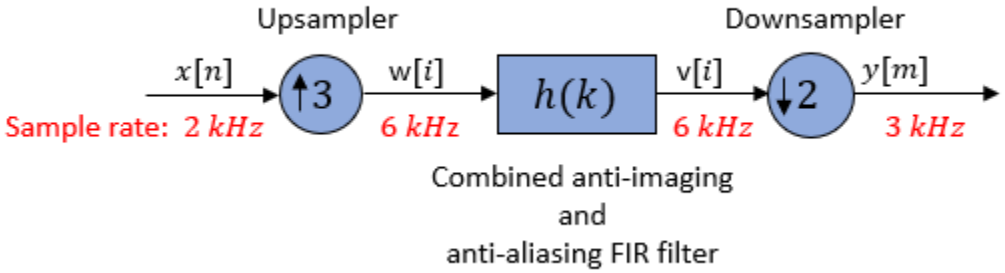
The rate converter resamples the signal according to the values of L and M:

- $L < M$ and L/M is not an integer -- Decimation with a noninteger conversion ratio.
- $L > M$ and L/M is not an integer -- Interpolation with a noninteger conversion ratio.
- $L/M = K/1$, where K is an integer -- Interpolation with an integer conversion ratio.
- $L/M = 1/K$ -- Decimation with an integer conversion ratio.

Effect on the Time and Frequency Domain

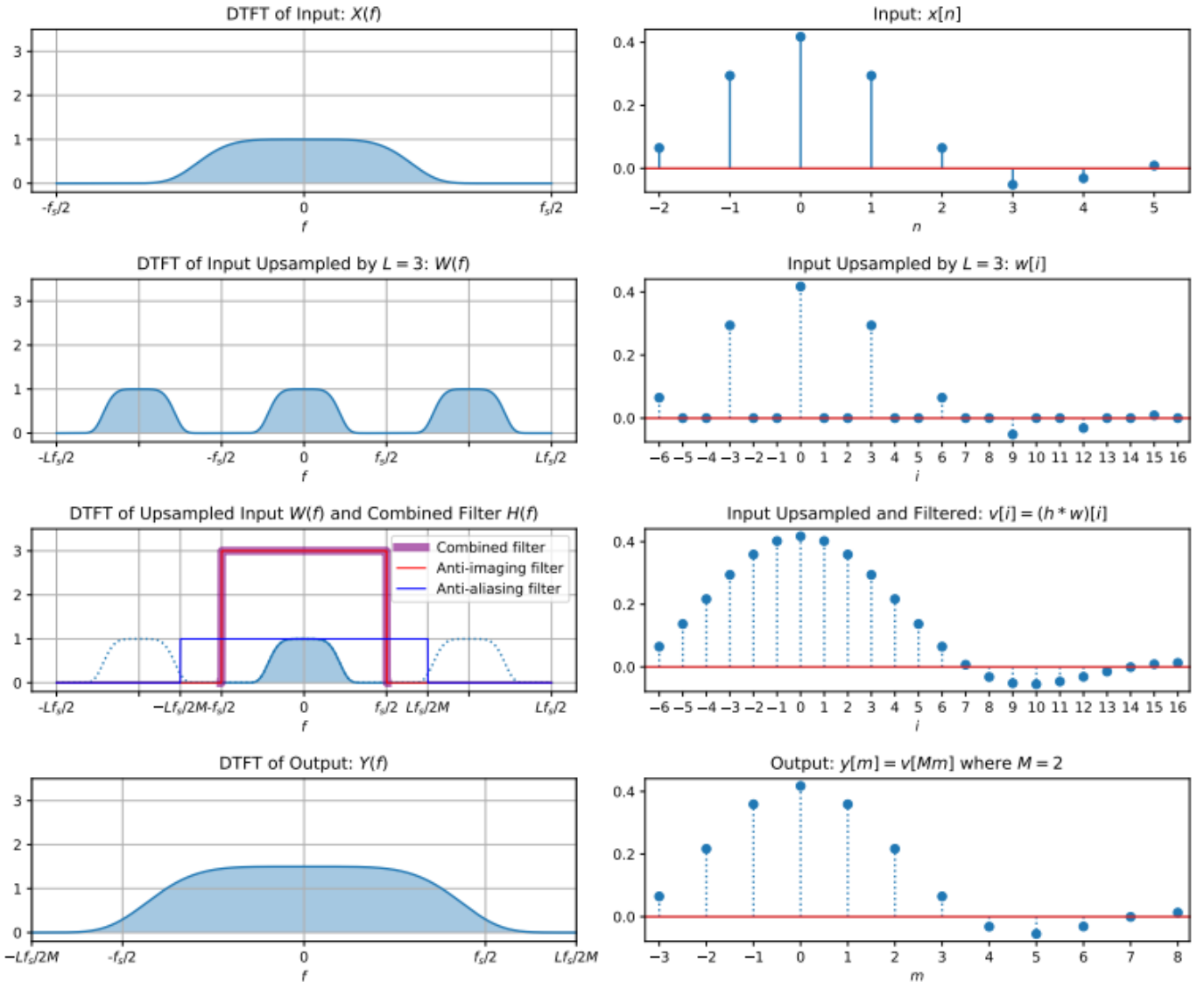
Consider a sample rate converter with a noninteger conversion factor 3/2.

Pass the input signal through an upsampler. The upsampler inserts two zero-valued samples for each sample of $x[n]$. Lowpass filter the signal to yield $v[i]$. Then pass the filtered data through a downsampler. The downsampler reduces the signal by a factor of 2 by retaining only one sample for every two samples of $v[i]$.



In the frequency domain, the input signal with a sample rate of 2 kHz is first increased by a factor of 3 to 6 kHz. The lowpass filter that follows bandlimits the signal and removes image frequencies which would otherwise cause aliasing. The filtered signal is then downsampled by a factor of 2 to 3 kHz.

The spectral components in the dashed lines are the frequencies which are removed by the lowpass filter.



When the sample rate conversion ratio is large, it is more efficient to implement the rate converter in two or more stages rather than in a single stage. For more details, see “Multistage Rate Conversion” on page 4-177.

References

[1] Fliege, Norbert. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. Wiley, 1994.

[2] Harris, Fred. *Multirate Signal Processing for Communication Systems*. Prentice Hall PTR, 2004.

[3] Hogenauer, E. “An Economical Class of Digital Filters for Decimation and Interpolation.” *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 29, no. 2, Apr. 1981, pp. 155–62.

[4] Lyons, Richard G. *Understanding Digital Signal Processing*. 2nd ed, Prentice Hall PIR, 2004.

[5] Mitra, S.K. *Digital Signal Processing*, McGraw-Hill, 1998.

[6] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Prentice Hall, 1996.

See Also

Related Examples

- “Multirate Filtering in MATLAB and Simulink” on page 1-36
- “Overview of Multistage Filters” on page 7-11
- “Overview of Filter Banks” on page 7-14

Overview of Multistage Filters

In this section...
“Multistage Decimator” on page 7-11
“Multistage Interpolator” on page 7-12
“Determine the number of stages and rate conversion factor for each stage” on page 7-12

Multistage filters are composed of several filter stages connected in series or parallel.

When you need to change the sample rate of a signal by a large factor, or implement a filter with a very narrow transition width, it is more efficient to implement the design in two or more stages rather than in one single stage. When the design is long (contains many coefficients) and costly (requires many multiplications and additions per input sample), the multistage approach is more efficient to implement compared to the single-stage approach.

Implementing a multirate filter with a large rate conversion factor using multiple stages allows for a gradual decrease or increase in the sample rate, allowing for a more relaxed set of requirements for the anti-aliasing or anti-imaging filter at each stage. Implementing a filter with a very narrow transition width in a single stage requires many coefficients and many multiplications and additions per input sample. When there are strict hardware requirements and it is impossible to implement long filters, the multistage approach acts as an efficient alternative. Though a multistage approach is efficient to implement, design advantages come at the cost of increased complexity.

Multistage Decimator

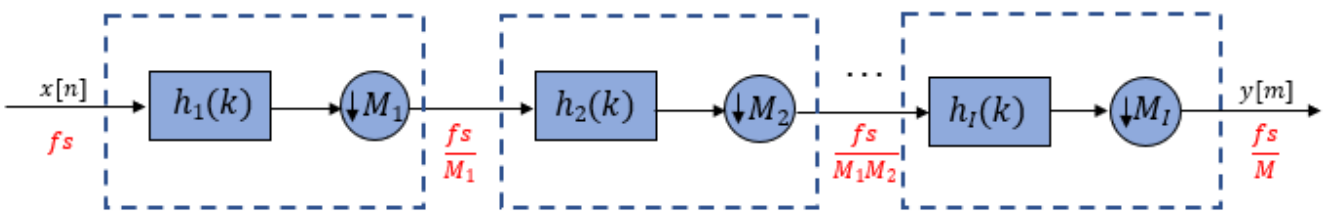
Consider an I-stage decimator. The overall decimation factor M is split into smaller factors with each factor being the decimation factor of the corresponding individual stage. The combined decimation of all the individual stages must equal the overall decimation. The combined response must meet or exceed the given design specifications.

The overall decimation factor M is expressed as the product of smaller factors:

$$M = M_1 M_2 \cdots M_I$$

where M_i is the decimation factor for stage i. Each stage is an independent decimator. The sample rate at the output of each i^{th} stage is:

$$f_i = \frac{f_{i-1}}{M_i}, \qquad i = 1, 2, \ldots, I$$



If $M \gg 1$, the multistage approach reduces computational and storage requirements significantly.

Multistage Interpolator

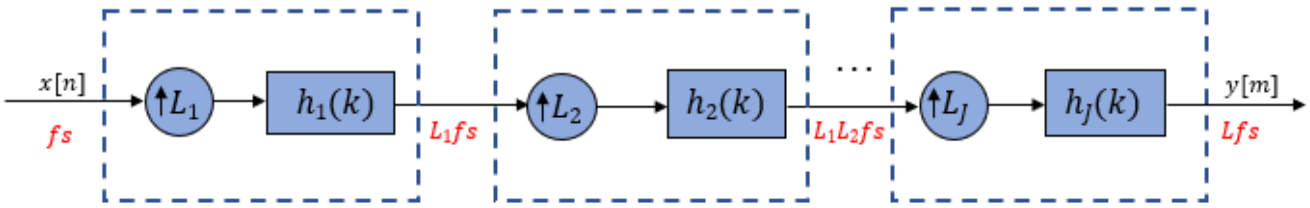
Consider a J-stage interpolator. The overall interpolation factor L is split into smaller factors with each factor being the interpolation factor of the corresponding individual stage. The filter in each interpolator eliminates the images introduced by the upsampling process in the corresponding interpolator. The combined interpolation of all the individual stages must equal the overall interpolation. The combined response must meet or exceed the given design specifications.

The overall interpolation factor L is expressed as the product of smaller factors:

$$L = L_1 L_2 \cdots L_J$$

where L_j is the interpolation factor for stage j. Each stage is an independent interpolator. The sample rate at the output of each j^{th} stage is:

$$f_j = L_j f_{j-1}, \qquad j = 1, 2, \ldots, J$$



If $L \gg 1$, the multistage approach reduces computational and storage requirements significantly.

Determine the number of stages and rate conversion factor for each stage

For a given rate conversion factor R, there is more than one possible configuration of filter stages. The number of stages and the rate conversion factor for each stage depends on the number of smaller factors R can be divided into. An optimal configuration is the sequence of filter stages leading to the least computational effort, with the computational effort measured by the number of multiplications per input sample, number of additions per input sample, and, in general, the total number of filter coefficients.

In the optimal configuration of multistage decimation filters, the shortest filter appears first and the longest filter (with the narrowest transition width) appears last. This sequence ensures that the filter with the longest length operates at the lowest sample rate, thereby reducing the cost of implementing the filter significantly.

Similarly, in the optimal configuration of multistage interpolation filters, the longest filter appears first and the shortest filter appears last. This sequence again ensures that the filter with the longest length operates at the lowest sample rate.

The `designMultistageDecimator` and `designMultistageInterpolator` functions in DSP System Toolbox automatically determine the optimal configuration, which includes determining the number of stages and the rate conversion factor for each stage. An optimal configuration leads to the least computational effort, and you can measure the cost of such an implementation using the `cost` function. For an example, see “Multistage Rate Conversion” on page 4-177.

See Also

Functions

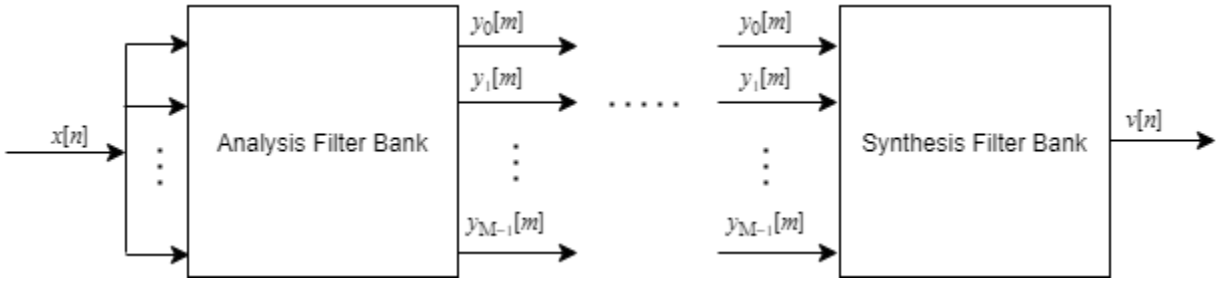
`cost` | `designMultistageDecimator` | `designMultistageInterpolator`

Related Examples

- “Multistage Rate Conversion” on page 4-177
- “Overview of Multirate Filters” on page 7-2
- “Overview of Filter Banks” on page 7-14

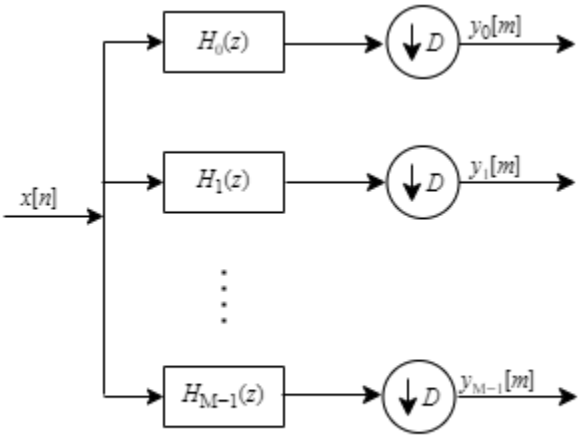
Overview of Filter Banks

A digital filter bank is an array of digital bandpass filters with either a common input or a common output. A filter bank can be an analysis filter bank with a series of analysis filters, or a synthesis filter bank with a series of synthesis filters. The analysis filter bank separates the input broadband signal $x[n]$ into multiple components, each carrying a subband of the original signal. The synthesis filter bank merges these subbands into a single broadband signal, a reconstructed version of the original input signal.



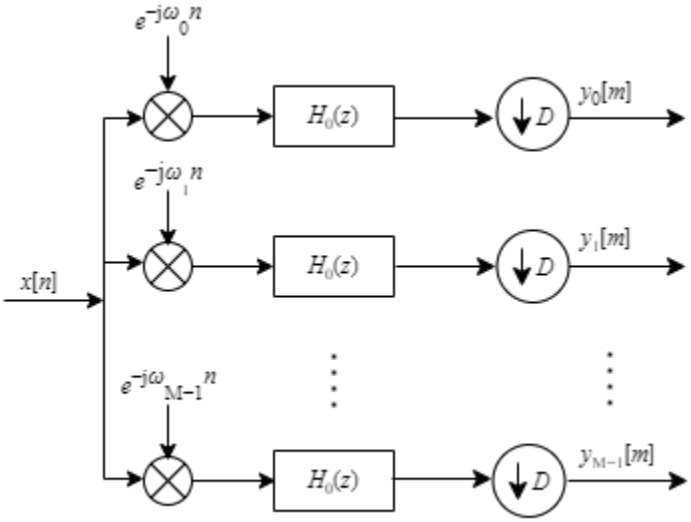
Analysis Filter Bank (Channelizer)

The generic analysis filter bank, also known as the channelizer, consists of a series of parallel bandpass filters that split an input broadband signal, $x[n]$, into a series of narrow subbands. Each bandpass filter retains a different portion of the input signal. After the bandwidth is reduced by one of the bandpass filters, the signal is downsampled to a lower sample rate commensurate with the new bandwidth.



Conceptual Diagram of Generic Analysis Filter Bank

The first branch in the filter bank contains a lowpass filter, $H_0(z)$, which acts as a prototype filter. The remaining filters $H_1(z)$ through $H_{M-1}(z)$ are modulated versions of this filter. These modulated versions can be rearranged in terms of a complex exponential (modulation factor) followed by the prototype lowpass filter $H_0(z)$.



Conceptual Diagram

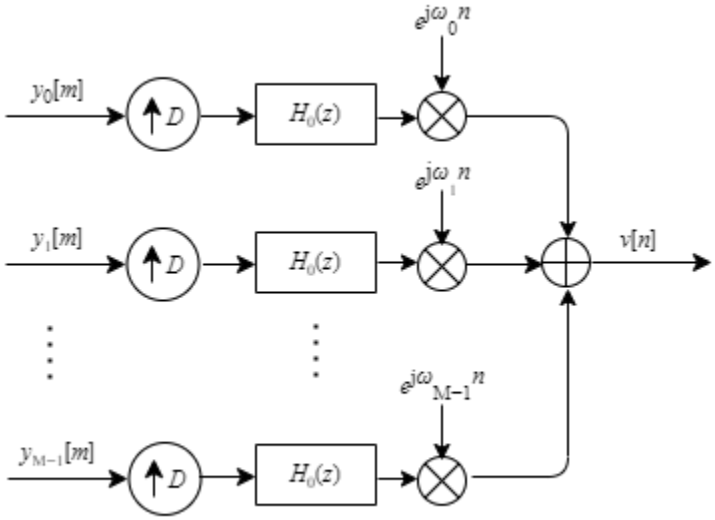
$y_1[m]$, $y_2[m]$, ..., $y_{M-1}[m]$ are narrow subband signals translated into baseband.

For more details on this structure, see Analysis Filter Bank.

This filter bank can be implemented efficiently using a polyphase structure. For more details on the polyphase structure and how it is implemented in the `dsp.Channelizer` object and the Channelizer block, see Channelizer Algorithm. The chief advantage of this polyphase implementation is that you can downsample the signal prior to filtering, thereby allowing you to filter the signal at a lower sample rate.

Synthesis Filter Bank (Channel synthesizer)

The synthesis filter bank, also known as the channel synthesizer, consists of a set of parallel bandpass filters that merge multiple input narrowband signals, $y_0[m]$, $y_1[m]$, $y_2[m]$, ... , $y_{M-1}[m]$ into a single broadband signal, $v[n]$. The input narrowband signals are in the baseband. Each narrowband signal is interpolated to a higher sample rate by using the upsampler, and then filtered by the lowpass filter. A complex exponential that follows the lowpass filter centers the baseband signal around w_k, where $w_k = 2\pi k/M$ and $k = 0, 1, 2, ..., M - 1$.



Conceptual Diagram

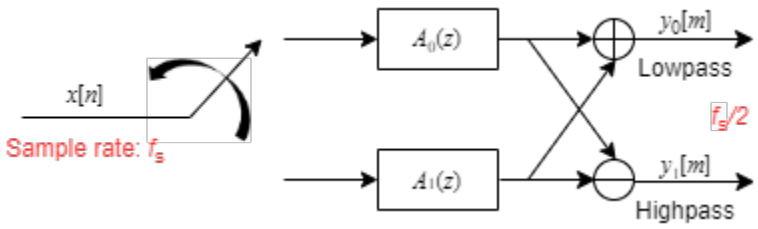
This filter bank is implemented efficiently using the polyphase structure described in Channel Synthesizer Algorithm. The `dsp.ChannelSynthesizer` object and the Channel Synthesizer block in DSP System Toolbox use this implementation.

Two-Channel (Halfband) Filter Bank

Two-channel filter bank is a special case of the generic *M*-channel filter bank, where the number of filter branches is two.

The DFT matrix of the analysis portion looks like the following matrix.

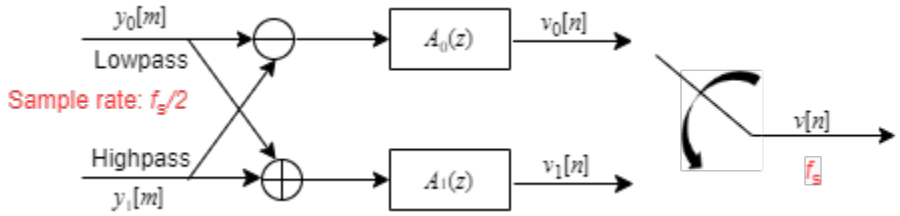
The first row adds the two polyphase branches to give the lowpass subband output. The second row subtracts the two polyphase branches to give the highpass subband output. The halfband decimator objects and blocks in DSP System Toolbox implement their algorithm as shown in this diagram. $A_0(z)$ and $A_1(z)$ are the allpass polyphase components. This structure is the analysis portion of the two-channel halfband filter bank. Due to the halfband nature of the filters, one of the branches in this polyphase structure becomes a pure delay component.



For more details on this structure and its derivation, see **Polyphase Implementation** under **Algorithms** on these reference pages.

Two-Channel Filter Bank	MATLAB	Simulink
Analysis portion using FIR halfband filter	`dsp.FIRHalfbandDecimator`	FIR Halfband Decimator
Analysis portion using IIR halfband filter	`dsp.IIRHalfbandDecimator`	IIR Halfband Decimator

Similarly, the halfband interpolator objects and blocks in DSP System Toolbox implement their algorithm as shown in this diagram. This structure is the synthesis portion of the two-channel halfband filter bank. Due to the halfband nature of the filters, one of the branches in this polyphase structure becomes a pure delay component.



For more details on this structure and its derivation, see **Polyphase Implementation** under **Algorithms** on these reference pages.

Two-Channel Filter Bank	MATLAB	Simulink
Synthesis portion using FIR halfband filter	`dsp.FIRHalfbandInterpolator`	FIR Halfband Interpolator
Synthesis portion using IIR halfband filter	`dsp.IIRHalfbandInterpolator`	IIR Halfband Interpolator

The other two-channel filter bank features that DSP System Toolbox offers let you specify the lowpass and highpass filter coefficients. These features can customize the partitioning of the broadband signal. For an example, see “Reconstruction Through Two-Channel Filter Banks” on page 4-241.

Two-Channel Filter Bank	MATLAB	Simulink
Analysis filter bank	`dsp.SubbandAnalysisFilter`	Two-Channel Analysis Subband Filter
Synthesis filter bank	`dsp.SubbandSynthesisFilter`	Two-Channel Synthesis Subband Filter

You can use the subband analysis and synthesis filter banks as basic units and create multilevel filter banks. For more details, see “Multilevel Filter Banks” on page 7-28.

See Also

Related Examples

- “Two-Channel Filter Bank Using Halfband Decimators and Halfband Interpolators” on page 7-19
- “Channelize and Synthesize Sine Wave in MATLAB” on page 7-26

- “Synthesize and Channelize Audio in Simulink” on page 15-2
- “Reconstruction Through Two-Channel Filter Banks” on page 4-241
- “Multilevel Filter Banks” on page 7-28

External Websites

- Filter Bank Method: A Better Approach to Spectral Analysis

Two-Channel Filter Bank Using Halfband Decimators and Halfband Interpolators

A two-channel filter bank is a special case of the generic *M*-channel filter bank, where there are two filter branches. Implement the analysis portion of the filter bank using halfband decimators. Implement the synthesis portion of the filter bank using halfband interpolators.

These features implement the halfband decimators and halfband interpolators in DSP System Toolbox™.

- `dsp.FIRHalfbandDecimator` and FIR Halfband Decimator: Implement the analysis portion of the filter bank by using the FIR halfband filter.
- `dsp.IIRHalfbandDecimator` and IIR Halfband Decimator: Implement the analysis portion of the filter bank by using the IIR halfband filter.
- `dsp.FIRHalfbandInterpolator` and FIR Halfband Interpolator: Implement the synthesis portion of the filter bank by using the FIR halfband filter.
- `dsp.IIRHalfbandInterpolator` and IIR Halfband Interpolator: Implement the synthesis portion of the filter bank by using the IIR halfband filter.

The two-channel analysis filter bank accepts a broadband signal as the input and splits this signal into lowpass and highpass subband signals. The two-channel synthesis filter bank accepts the lowpass and highpass subband signals as inputs and reconstructs the broadband signal using these two subbands.

Use either the FIR halfband or IIR halfband filters in the filter bank. Each filter has its own advantages and disadvantages and your choice should depend on the computational requirements of the application. IIR filters require fewer coefficients than FIR filters to execute similar filtering operations. IIR filters work faster and require less memory space. However, FIR filters are always stable. When the coefficients of an FIR filter are symmetric or anti-symmetric, the filter exhibits linear phase response.

Use an audio file input and compare the power spectrum of the filter bank output with that of the input signal. Then compare the cost of implementing both the FIR and IIR halfband filter banks.

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)`.

Note: The `audioDeviceWriter` System object™ is not supported in MATLAB Online.

FIR Halfband Two-Channel Filter Bank

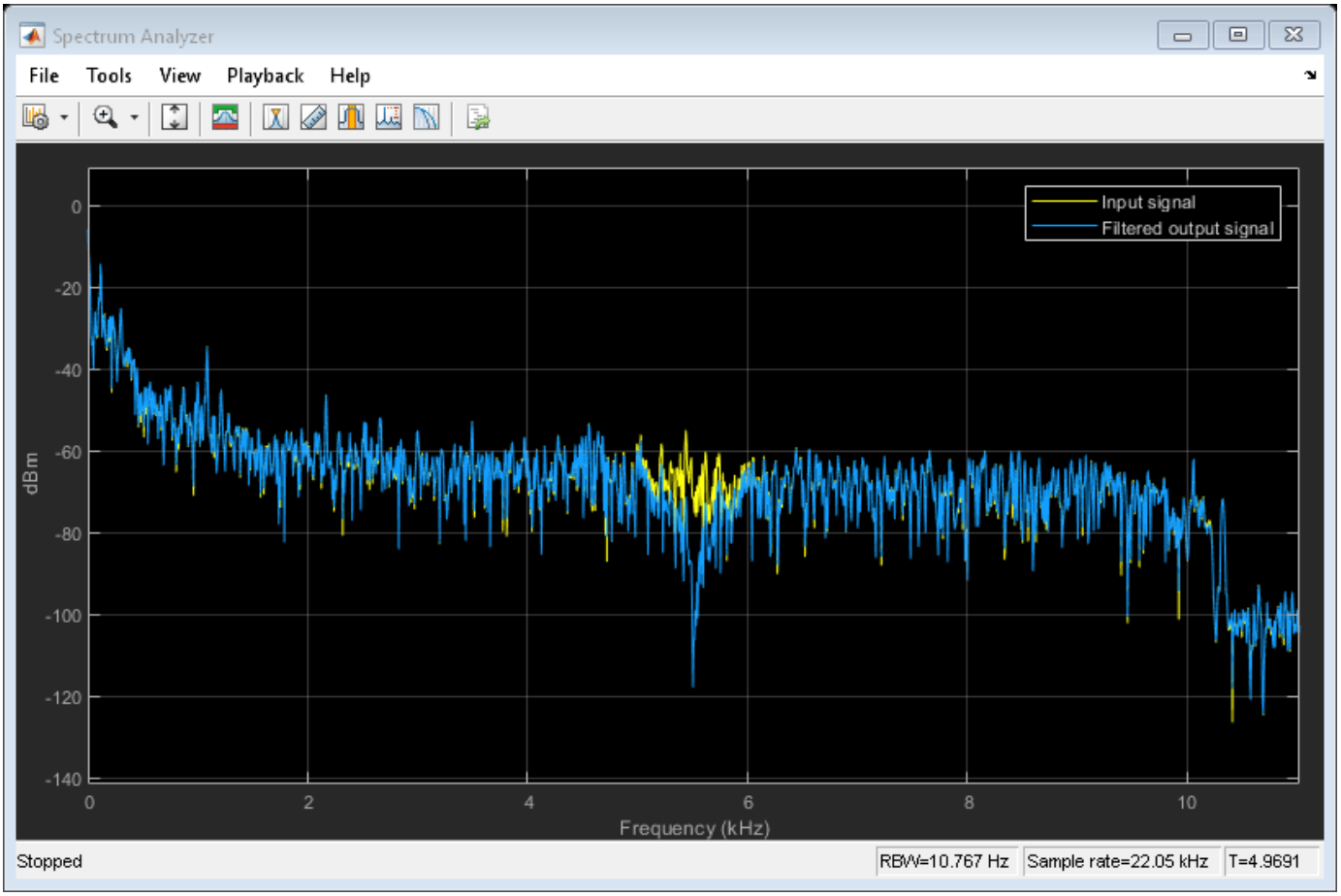
Set up the audio file reader and device writer. Create the FIR halfband decimator and interpolator to design a minimum order filter with a transition width of 2000 Hz and a stopband attenuation of 80 dB. Finally, set up the spectrum analyzer to display the power spectra of the filter bank input and output.

```
AF = dsp.AudioFileReader('speech_dft.mp3','SamplesPerFrame',1024);
AP = audioDeviceWriter('SampleRate',AF.SampleRate);

filterspec = 'Transition width and stopband attenuation';
TW = 2000;
Astop = 80;

firhalfbanddecim = dsp.FIRHalfbandDecimator(...
```

```
    'Specification',filterspec, ...
    'StopbandAttenuation',Astop,...
    'TransitionWidth',TW, ...
    'SampleRate',AF.SampleRate);

firhalfbandinterp = dsp.FIRHalfbandInterpolator(...
    'Specification',filterspec, ...
    'StopbandAttenuation',Astop,...
    'TransitionWidth',TW, ...
    'SampleRate',AF.SampleRate/2,...
    'FilterBankInputPort',true);

SpecAna = dsp.SpectrumAnalyzer('SampleRate',AF.SampleRate,...
    'PlotAsTwoSidedSpectrum',false,'ReducePlotRate',false,...
    'ShowLegend',true,...
    'ChannelNames',{'Input signal','Filtered output signal'});
```

Read the audio 1024 samples at a time. Filter the input to obtain the lowpass and highpass subband signals decimated by a factor of two. This is the analysis portion of the filter bank. Use the FIR halfband interpolator as the synthesis filter bank. Display the running power spectrum of the audio input and the output of the synthesis filter bank. Play the output using the audio device writer.

```
while ~isDone(AF)
    audioInput = AF();
    [xlo,xhigh] = firhalfbanddecim(audioInput);
    audioOutput = firhalfbandinterp(xlo,xhigh);
    spectrumInput = [audioInput audioOutput];
    SpecAna(spectrumInput);
    AP(audioOutput);
end

release(AF);
release(AP);
release(SpecAna);
```



IIR Halfband Two-Channel Filter Bank

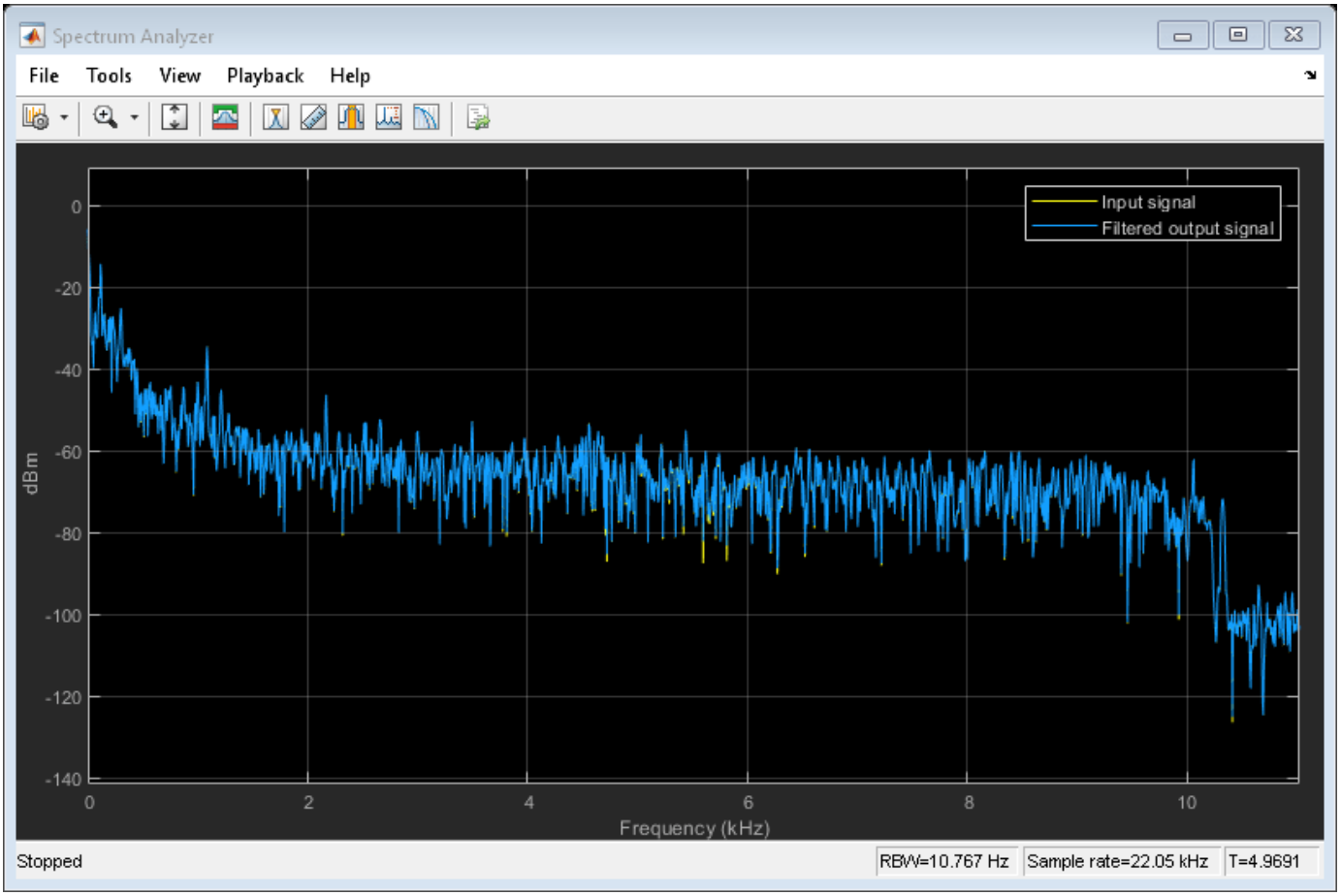
Set up the IIR halfband decimator and interpolator. The filters are of minimum order, have a transition width of 2000 Hz, and a stopband attenuation of 80 dB. Set up the spectrum analyzer to display the power spectra of the filter bank input and the output.

```
AF = dsp.AudioFileReader('speech_dft.mp3','SamplesPerFrame',1024);
AP = audioDeviceWriter('SampleRate',AF.SampleRate);

filterspec = 'Transition width and stopband attenuation';
TW = 2000;
Astop = 80;

iirhalfbanddecim = dsp.IIRHalfbandDecimator(...
    'Specification',filterspec,'StopbandAttenuation',Astop,...
    'TransitionWidth',TW,'SampleRate',AF.SampleRate);

iirhalfbandinterp = dsp.IIRHalfbandInterpolator(...
    'Specification',filterspec,'StopbandAttenuation',Astop,...
    'TransitionWidth',TW,'SampleRate',AF.SampleRate/2,...
    'FilterBankInputPort',true);

SpecAna = dsp.SpectrumAnalyzer('SampleRate',AF.SampleRate,...
    'PlotAsTwoSidedSpectrum',false,'ReducePlotRate',false,...
```

```
    'ShowLegend',true,...
    'ChannelNames',{'Input signal','Filtered output signal'});
```

Read the audio 1024 samples at a time. Filter the input signal to obtain the lowpass and highpass subband signals decimated by a factor of two. This is the analysis portion of the filter bank. Use the IIR halfband interpolator as the synthesis filter bank. Display the running power spectrum of the audio input and the output of the synthesis filter bank. Play the output using the audio device writer.

```
while ~isDone(AF)
    audioInput = AF();
    [xlo,xhigh] = iirhalfbanddecim(audioInput);
    audioOutput = iirhalfbandinterp(xlo,xhigh);
    spectrumInput = [audioInput audioOutput];
    SpecAna(spectrumInput);
    AP(audioOutput);
end

release(AF);
release(AP);
release(SpecAna);
```



Compare the Cost

Compare the cost of implementing the FIR and IIR halfband decimators using the `cost` function. You can see that IIR halfband decimator is computationally more efficient.

```
cost(firhalfbanddecim)
```

```
ans = struct with fields:
                  NumCoefficients: 27
                        NumStates: 50
    MultiplicationsPerInputSample: 13.5000
          AdditionsPerInputSample: 13
```

```
cost(iirhalfbanddecim)
```

```
ans = struct with fields:
                  NumCoefficients: 5
                        NumStates: 7
    MultiplicationsPerInputSample: 2.5000
          AdditionsPerInputSample: 5
```

Compare the cost of implementing the FIR and IIR halfband interpolators. Again, the computational cost of implementing the IIR halfband interpolator is much less compared to the computational cost of implementing the IIR halfband decimator.

```
cost(firhalfbandinterp)
```

```
ans = struct with fields:
                  NumCoefficients: 26
                        NumStates: 25
    MultiplicationsPerInputSample: 26
          AdditionsPerInputSample: 25
```

```
cost(iirhalfbandinterp)
```

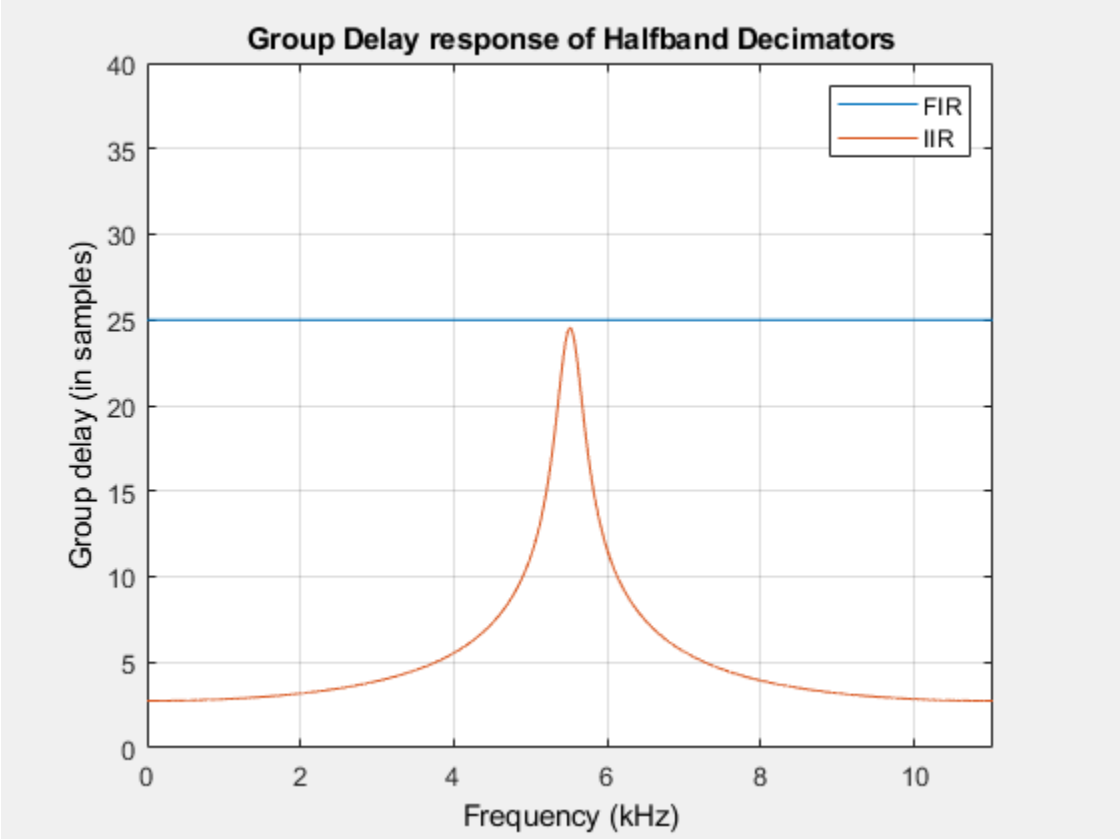
```
ans = struct with fields:
                  NumCoefficients: 5
                        NumStates: 7
    MultiplicationsPerInputSample: 5
          AdditionsPerInputSample: 10
```

Compare the Group Delay

You can also compare the group delay of the FIR and the IIR halfband filters.

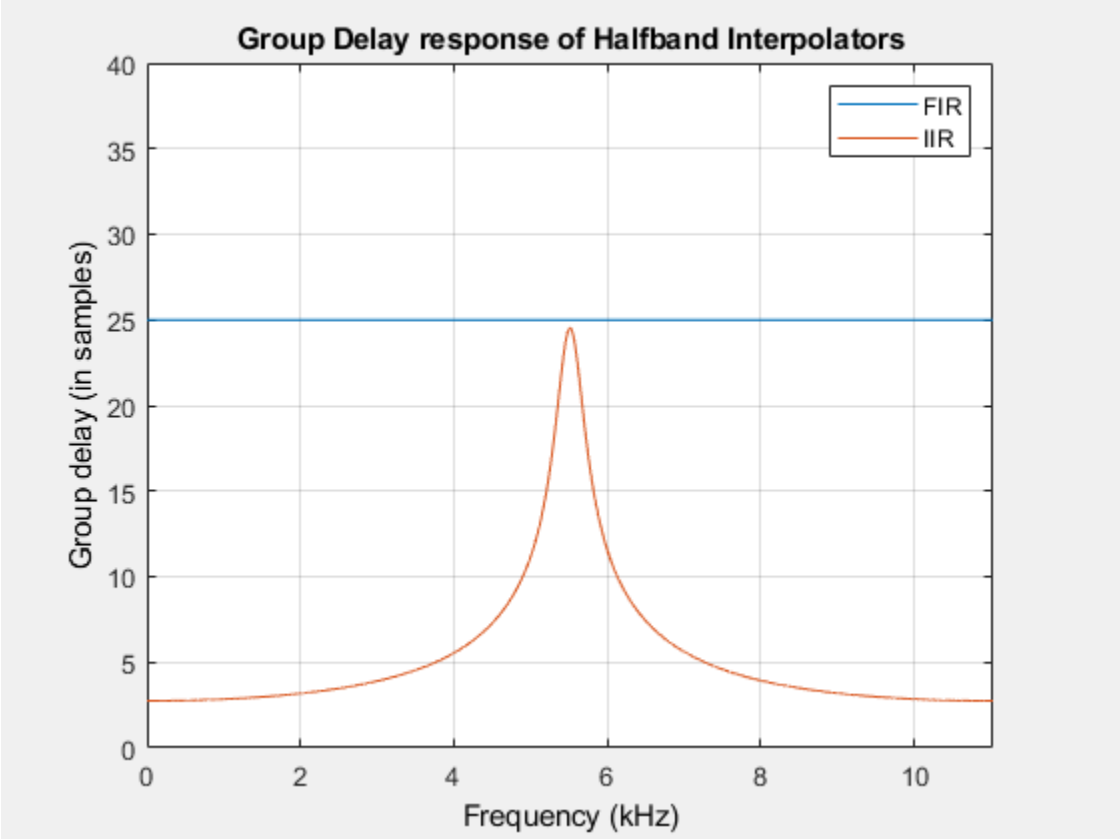
Use `fvtool` to display the group delay response of the FIR halfband and IIR halfband filters in the decimator. The FIR halfband filter has a linear phase response with a group delay of 25 samples. The IIR halfband filter has a nonlinear phase response, especially around the cutoff frequency of the filter. A constant group delay ensures that all the frequency components are delayed by the same amount.

```
h = fvtool(firhalfbanddecim,iirhalfbanddecim,'grpdelay');
legend('FIR','IIR')
title('Group Delay response of Halfband Decimators')
h.CurrentAxes.YLim = [0, 40];
```



Similarly in the interpolator, the FIR halfband filter has a linear phase response, while the IIR halfband filter has a nonlinear phase response.

```
h = fvtool(firhalfbandinterp,iirhalfbandinterp,'grpdelay');
legend('FIR','IIR')
title('Group Delay response of Halfband Interpolators')
h.CurrentAxes.YLim = [0, 40];
```



See Also

Objects

`dsp.AudioFileReader` | `audioDeviceWriter` | `dsp.FIRHalfbandDecimator` | `dsp.FIRHalfbandInterpolator` | `dsp.IIRHalfbandDecimator` | `dsp.IIRHalfbandInterpolator`

Related Examples

- “Overview of Filter Banks” on page 7-14
- “Channelize and Synthesize Sine Wave in MATLAB” on page 7-26
- “Synthesize and Channelize Audio in Simulink” on page 15-2
- “Multilevel Filter Banks” on page 7-28

Channelize and Synthesize Sine Wave in MATLAB

Channelize and synthesize a sine wave signal with multiple frequencies using an *M* -channel filter bank.

The *M* -channel filter bank contains an analysis filter bank section and a synthesis filter bank section. The `dsp.Channelizer` object implements the analysis filter bank section. The `dsp.ChannelSynthesizer` object implements the synthesis filter bank section. These objects use an efficient polyphase structure to implement the filter bank. For more details, see **Polyphase Implementation** under **Algorithms** on the object reference pages.

Initialization

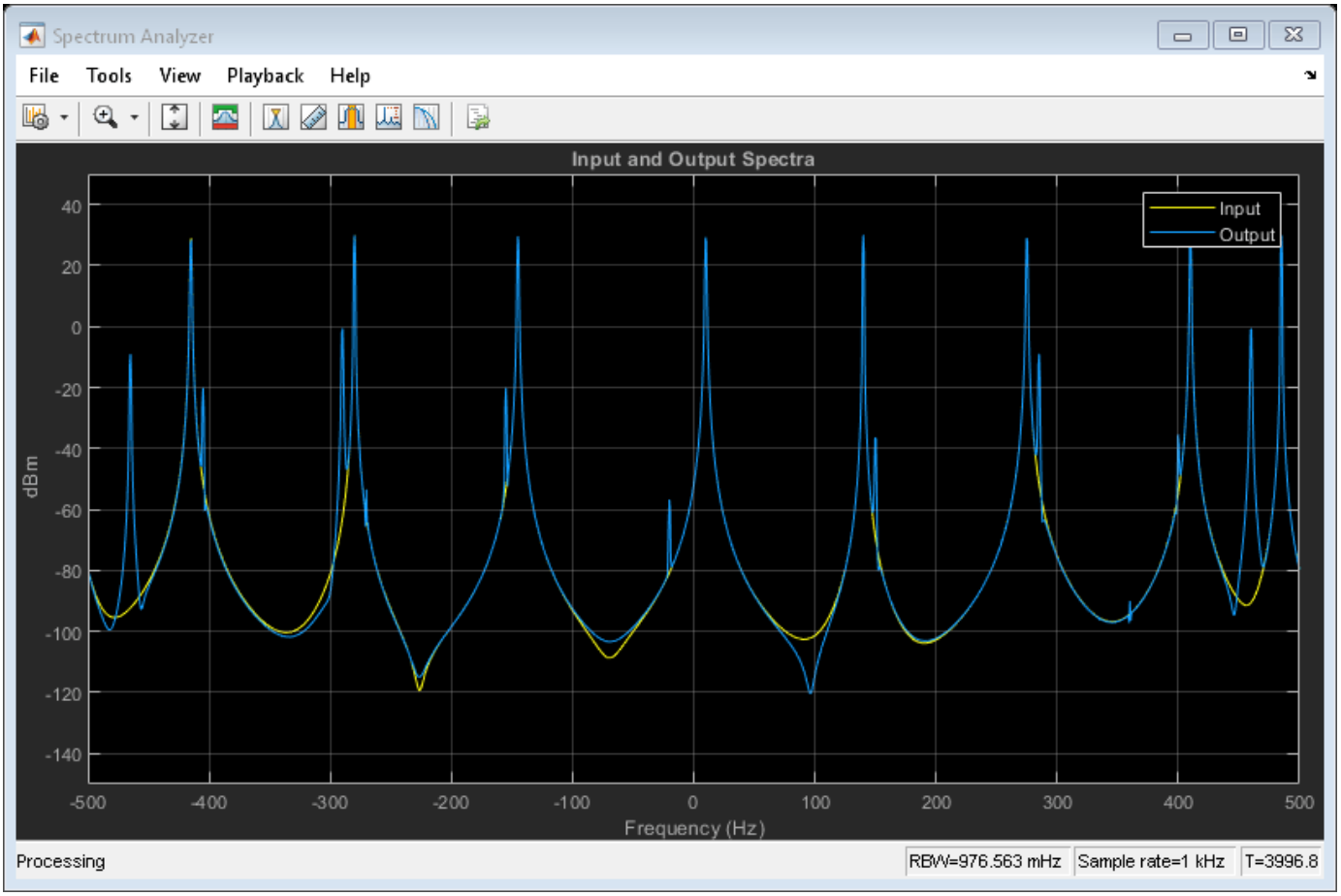
Initialize the `dsp.Channelizer` and `dsp.ChannelSynthesizer` System objects. Each object is set up with 8 frequency bands, 8 polyphase branches in each filter, 12 coefficients per polyphase branch, and a stopband attenuation of 140 dB. Use a sine wave with multiple frequencies as the input signal. View the input spectrum and the output spectrum using a spectrum analyzer.

```
offsets = [-40,-30,-20,10,15,25,35,-15];
sinewave = dsp.SineWave('ComplexOutput',true,'Frequency',...
    offsets+(-375:125:500),'SamplesPerFrame',800);

channelizer = dsp.Channelizer('StopbandAttenuation',140);
synthesizer = dsp.ChannelSynthesizer('StopbandAttenuation',140);
spectrumAnalyzer =  dsp.SpectrumAnalyzer('ShowLegend',true,...
    'SampleRate',sinewave.SampleRate,...
    'NumInputPorts',2,'ChannelNames',{'Input','Output'},...
    'Title','Input and Output Spectra');
```

Streaming

Use the channelizer to split the broadband input signal into multiple narrow bands. Then pass the multiple narrowband signals into the synthesizer, which merges these signals to form the broadband signal. Compare the spectra of the input and output signals. The input and output spectra match very closely.

```
for i = 1:5000
    x = sum(sinewave(),2);
    y = channelizer(x);
    v = synthesizer(y);
    spectrumAnalyzer(x,v)
end
```



See Also

`dsp.Channelizer` | `dsp.ChannelSynthesizer`

Related Examples

- “Overview of Filter Banks” on page 7-14
- “Two-Channel Filter Bank Using Halfband Decimators and Halfband Interpolators” on page 7-19
- “Synthesize and Channelize Audio in Simulink” on page 15-2
- “Multilevel Filter Banks” on page 7-28

Multilevel Filter Banks

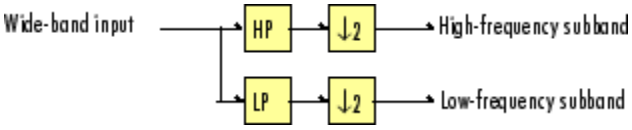
Multirate filters alter the sample rate of the input signal during the filtering process. Such filters are useful in both rate conversion and filter bank applications.

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

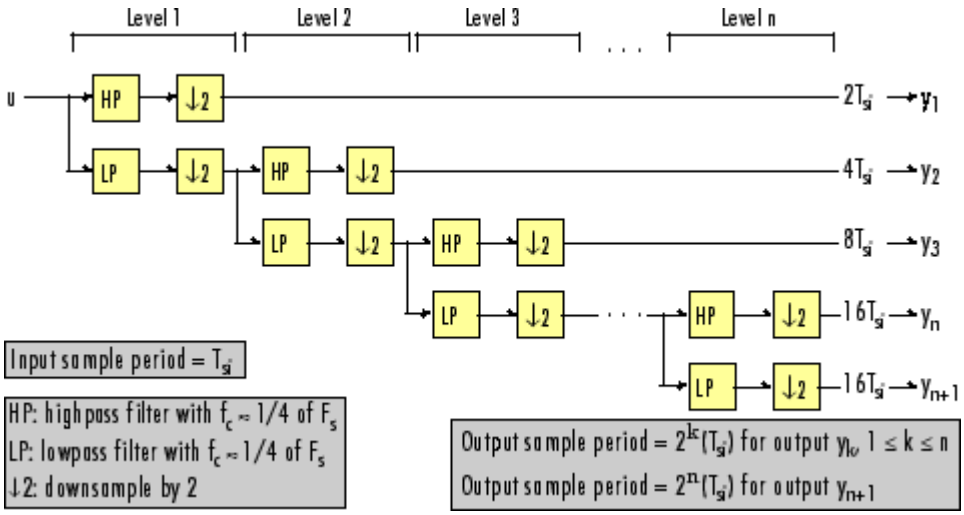
Dyadic analysis filter banks are constructed from the following basic unit. The unit can be cascaded to construct dyadic analysis filter banks with either a symmetric or asymmetric tree structure.



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.

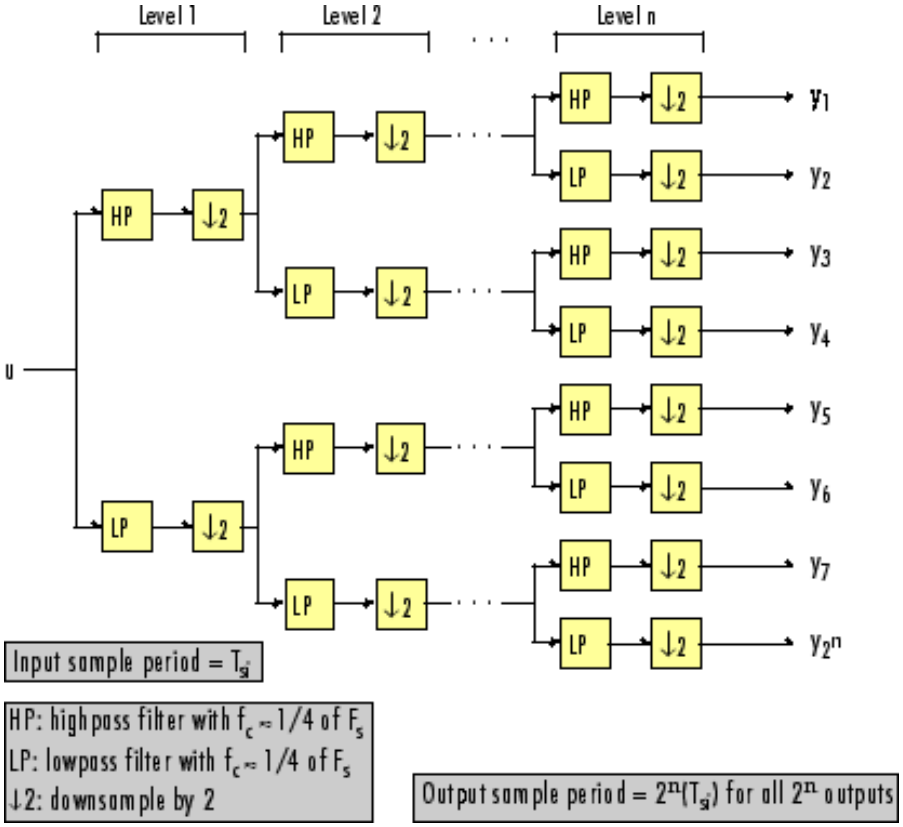
The unit decomposes its input into adjacent high-frequency and low-frequency subbands. Compared to the input, each subband has half the bandwidth (due to the half-band filters) and half the sample rate (due to the decimation by 2).

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

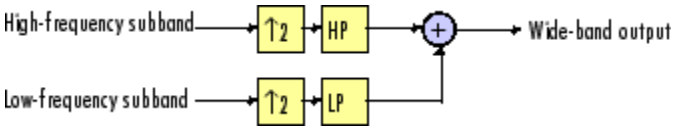
The following table summarizes the key characteristics of the symmetric and asymmetric dyadic analysis filter bank.

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

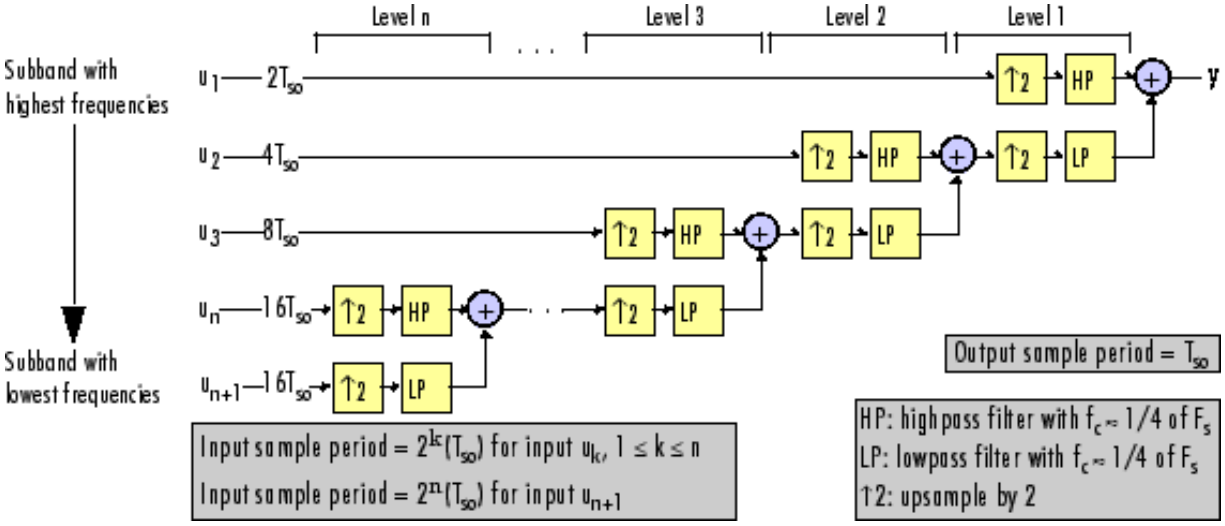
Dyadic synthesis filter banks are constructed from the following basic unit. The unit can be cascaded to construct dyadic synthesis filter banks with either a asymmetric or symmetric tree structure as illustrated in the figures titled 'n-Level Asymmetric Dyadic Synthesis Filter Bank' and 'n-Level Symmetric 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.

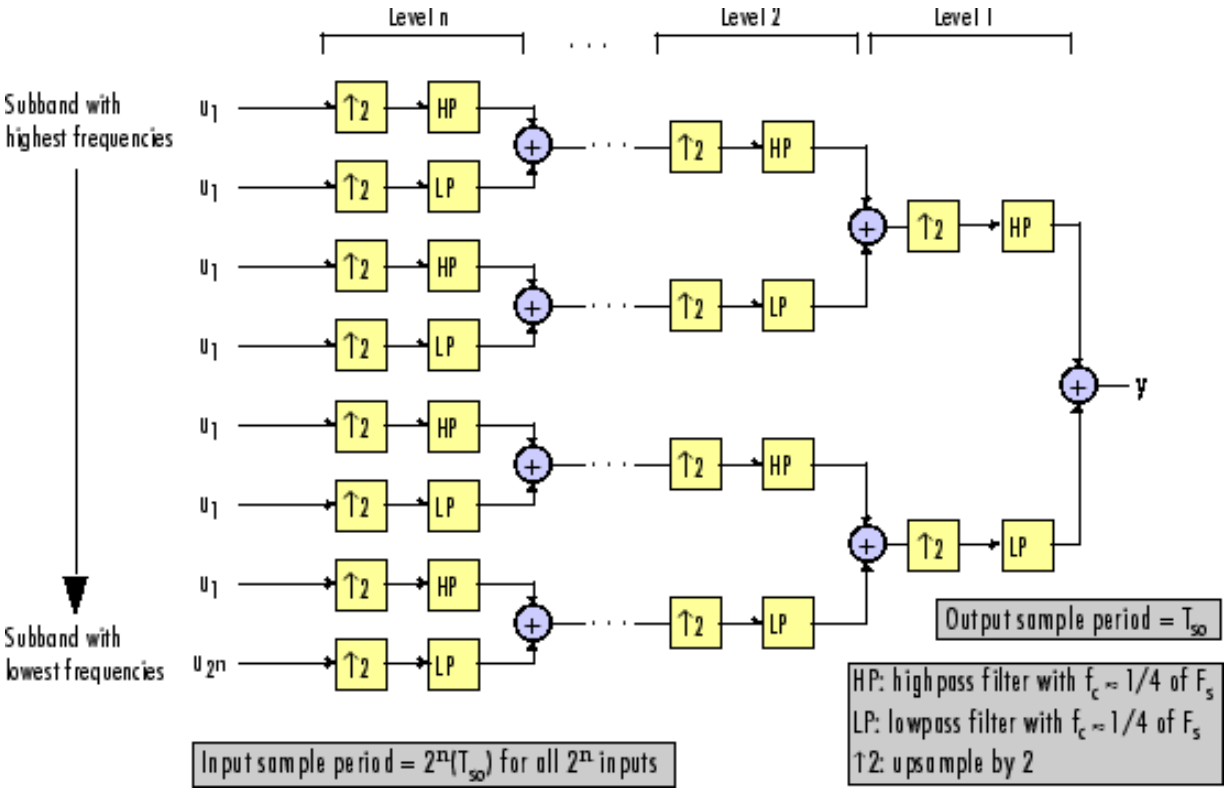
The unit takes in adjacent high-frequency and low-frequency subbands, and reconstructs them into a wide-band signal. Compared to each subband input, the output has twice the bandwidth and twice the sample rate.

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 *k*th 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 *k*th 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.

See Also

Objects
`dsp.SubbandAnalysisFilter` | `dsp.SubbandSynthesisFilter` | `dsp.DyadicAnalysisFilterBank` | `dsp.DyadicSynthesisFilterBank`

Blocks
Two-Channel Analysis Subband Filter | Two-Channel Synthesis Subband Filter | Dyadic Analysis Filter Bank | Dyadic Synthesis Filter Bank

Related Examples

- “Overview of Filter Banks” on page 7-14
- “Two-Channel Filter Bank Using Halfband Decimators and Halfband Interpolators” on page 7-19
- “Channelize and Synthesize Sine Wave in MATLAB” on page 7-26
- “Synthesize and Channelize Audio in Simulink” on page 15-2
- “Calculate Channel Latencies Required for Wavelet Reconstruction” on page 18-9

Dataflow

- “Dataflow Domain” on page 8-2
- “Model Multirate Signal Processing Systems Using Dataflow” on page 8-10
- “Multicore Simulation and Code Generation of Dataflow Domains” on page 8-12
- “Multicore Execution using Dataflow Domain” on page 8-19
- “Multicore Code Generation for Dataflow Domain” on page 8-27
- “Perform Multicore Analysis for Dataflow” on page 8-34
- “Multicore Analysis Using a Dataflow Domain” on page 8-41

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.

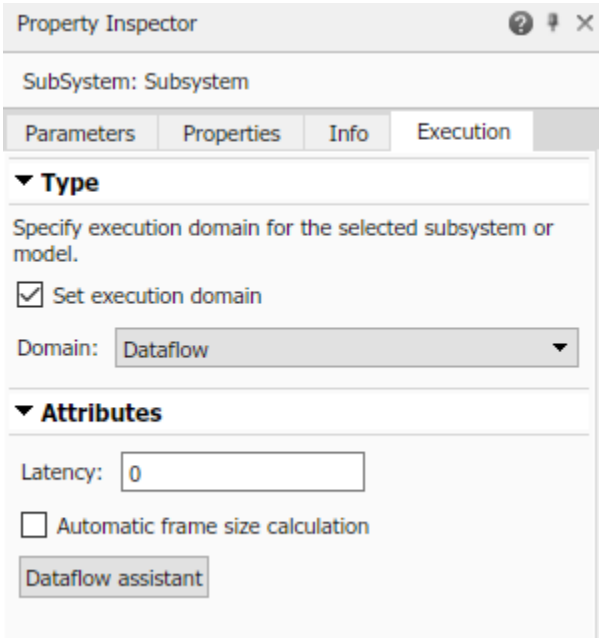
Some blocks and language features are not supported for multithreaded simulation. If a dataflow subsystem contains blocks or language features that do not support multithreaded simulation, Simulink issues a warning and the subsystem always simulates in a single thread.

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 8-8.

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.

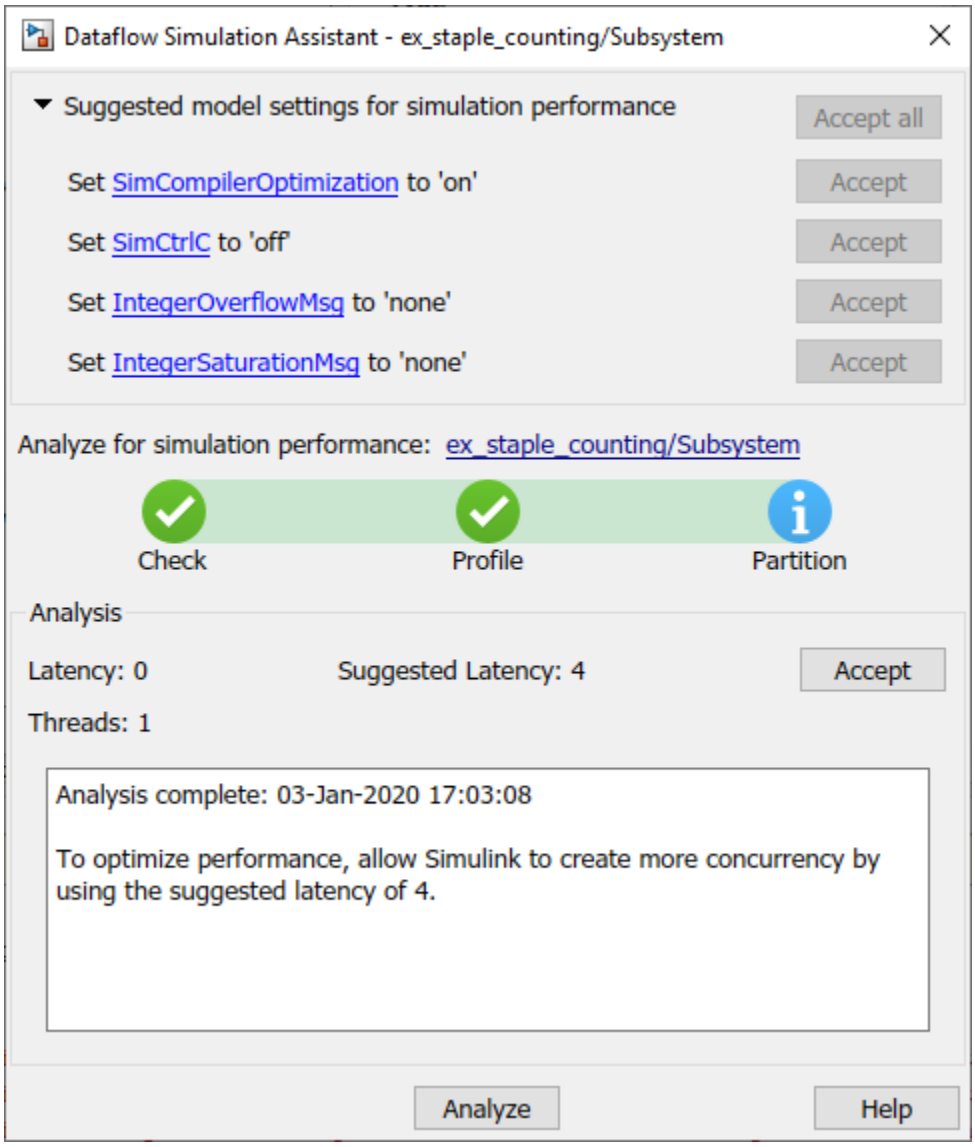
  ```
  set_param(gcs, 'IntegerSaturationMsg', 'none')
  ```

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.

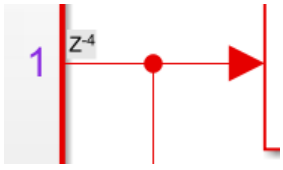
Click the **Analyze** button to analyze a dataflow domain for simulation performance and have the Dataflow Simulation Assistant suggest the optimal latency for your dataflow subsystem.

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.



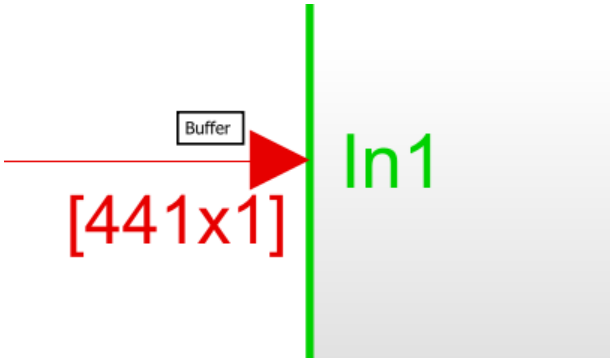
If you specify a latency greater than the latency suggested by the Dataflow Simulation Assistant, the additional delays are all inserted at the output of the subsystem. If you specify a latency value smaller than the value suggested by the Dataflow Simulation Assistant, the assistant warns that you are missing out on available concurrency.

If your model contains several dataflow subsystems, you can use the Performance Advisor to analyze and suggest a latency for each of the dataflow subsystems in a single step. To find the optimal latency settings for the Dataflow subsystems in your model, open the Performance Advisor. In the **Performance Advisor > Simulation > Checks that Require Simulation to Run** folder, run the **Check Dataflow Domain Settings** check.

For more information on types of parallelism in dataflow domains, see “Multicore Simulation and Code Generation of Dataflow Domains” on page 8-12.

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.

Not Supported	Description
Stateflow® charts	Stateflow charts are not supported inside dataflow subsystems.
SimEvents® blocks	SimEvents blocks are not supported inside dataflow subsystems.
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 8-12
- “Model Multirate Signal Processing Systems Using Dataflow” on page 8-10
- “Perform Multicore Analysis for Dataflow” on page 8-34
- “Multicore Execution using Dataflow Domain” on page 8-19

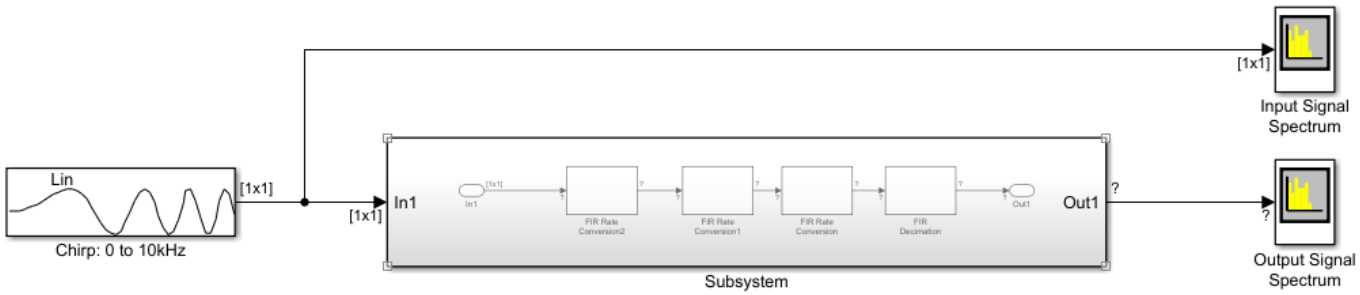
Model Multirate Signal Processing Systems Using Dataflow

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.

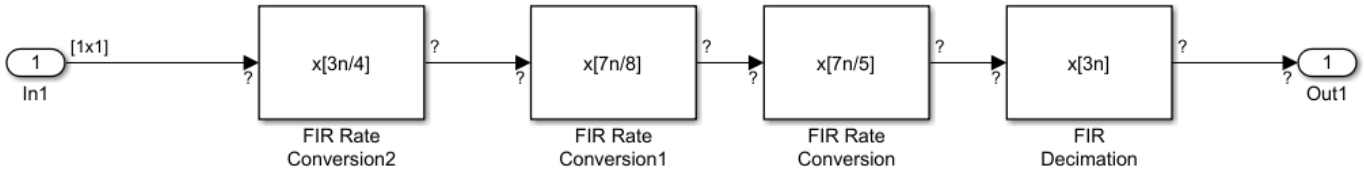
For more information on dataflow domains, see “Dataflow Domain” on page 8-2.

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.



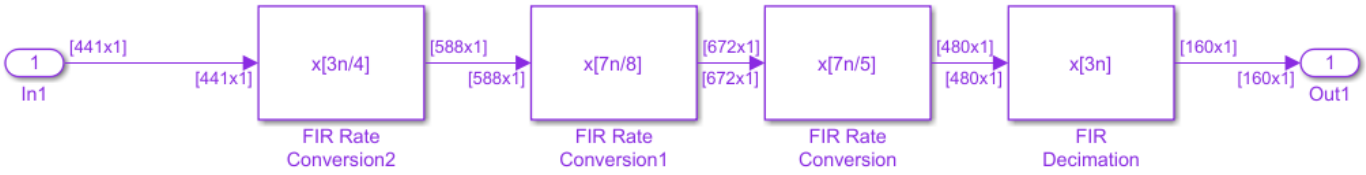
2 When you update the model diagram, Simulink generates an error due to a port dimension mismatch in the model. To fix this error, set the domain of the subsystem to dataflow.

3 If the Property Inspector is not visible, in the **Modeling** tab, under **Design**, select **Property Inspector**.

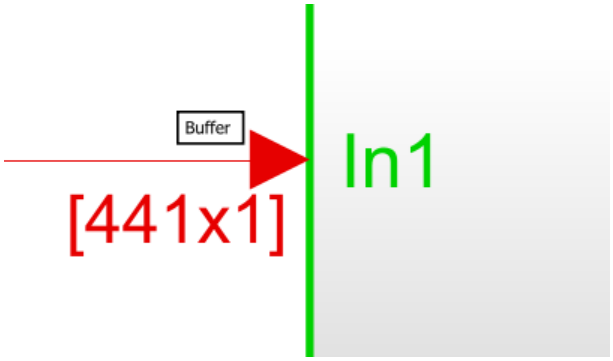
With the subsystem selected, in the **Execution** tab of the Property Inspector, select **Set execution domain**. Set the **Domain** to `Dataflow`.

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 where buffers are inserted.



See Also

Dataflow Subsystem

More About

- “Dataflow Domain” on page 8-2
- “Multicore Execution using Dataflow Domain” on page 8-19
- “Perform Multicore Analysis for Dataflow” on page 8-34

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.

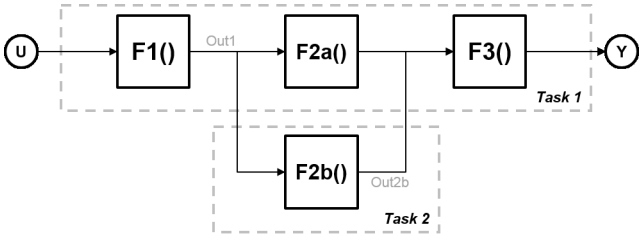
Types of Parallelism

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

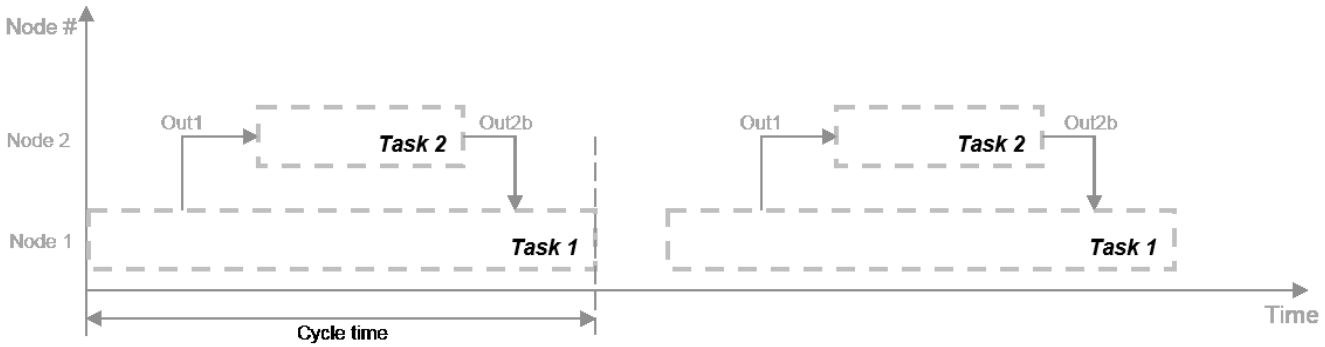
Task parallelism achieves parallelism by splitting up an application into multiple tasks. Task parallelism involves distributing tasks within an application across multiple processing nodes. Some tasks can have data dependency on others, so all tasks do not run at exactly the same time.

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

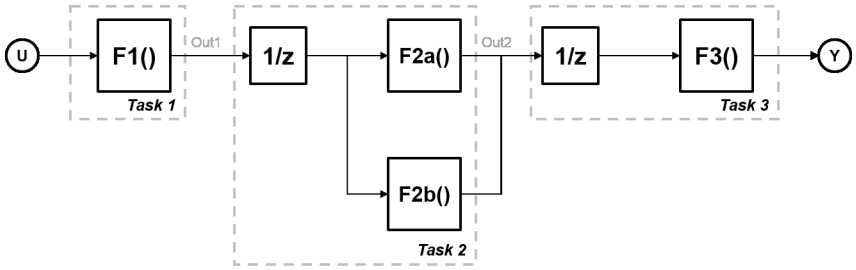
t = tF1 + max(tF2a, tF2b) + tF3.



Model Pipeline Execution (Pipelining)

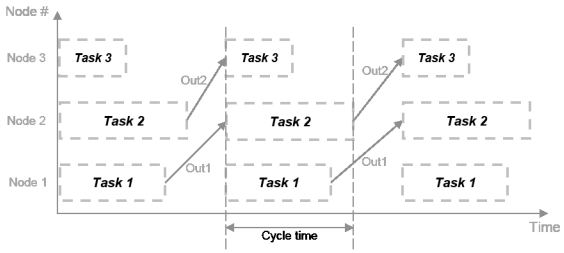
The software uses model pipeline execution, or pipelining, to work around the problem of task parallelism where threads do not run completely in parallel. This approach involves modifying the system to introduce delays between tasks where there is a data dependency.

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 8-2

1. To begin, open the model.

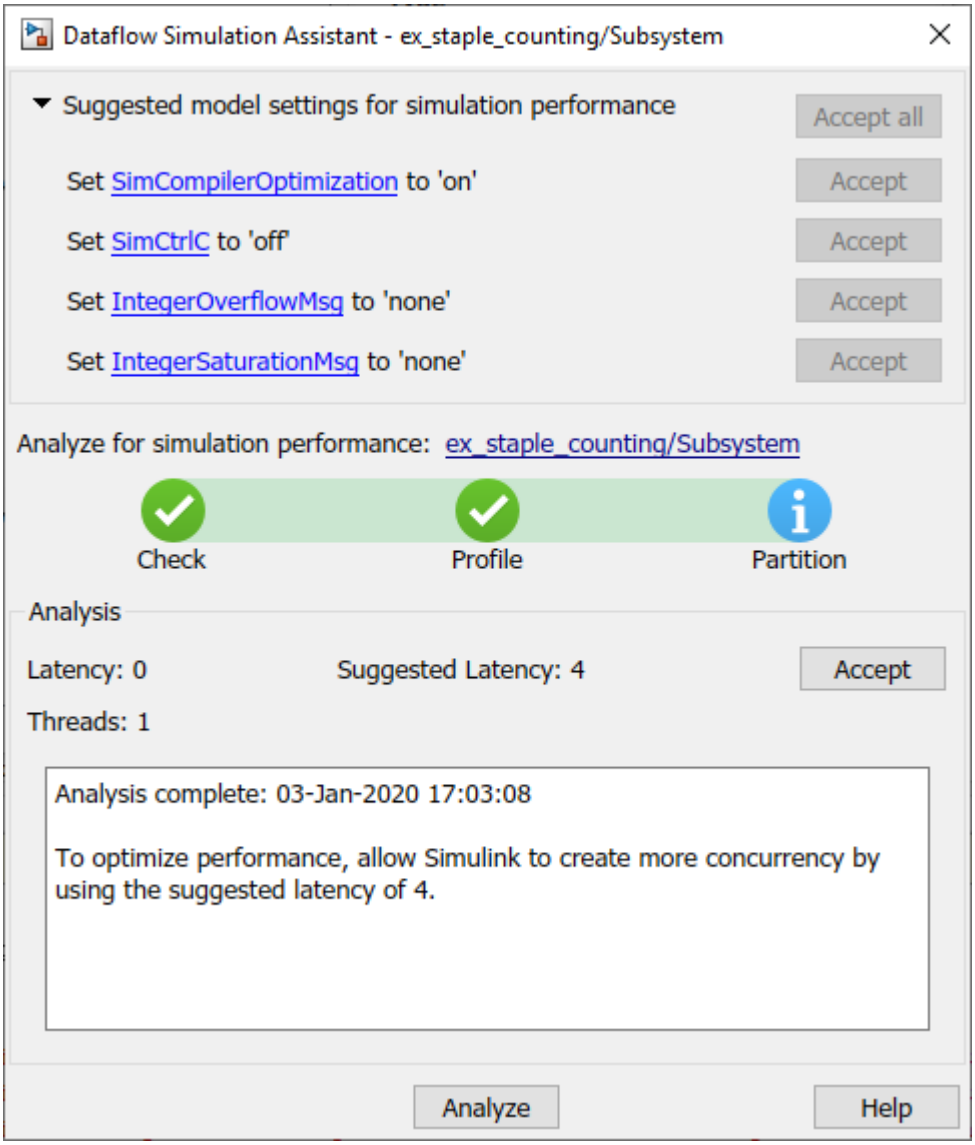
   ```
   addpath (fullfile(docroot, 'toolbox', 'dsp', 'examples'));
   ex_staple_counting
   ```
2. Simulate the model and observe the frame rate of the system in the Frame Rate Display block. This number indicates the number of frames per second that Simulink is able to process in a standard simulation.
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**.

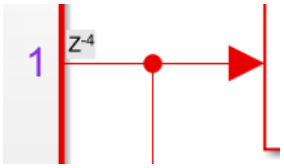
   With the subsystem selected, in the **Execution** tab of the Property Inspector, select **Set execution domain**. Set the **Domain** to `Dataflow`.
4. Sometimes, you can increase the available concurrency in your system by adding **Latency** to the system. To select an optimal latency value, use the 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 `slrealtime.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.

For multicore custom targets, use the `DataflowThreadingImplementation` parameter to select POSIX threads (Pthreads) or OpenMP threading implementation for multicore custom targets using dataflow domain. The parameter takes the values `off`, `OpenMP`, or `Pthreads`. By default, the parameter is set to `Off` which means that dataflow generates single core code (not multiple threads).

```
set_param(myModel, 'DataflowThreadingImplementation','off')
```

For information about customizing system target files, see “Customize System Target Files” (Embedded Coder).

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 8-2
- “Perform Multicore Analysis for Dataflow” on page 8-34
- “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.

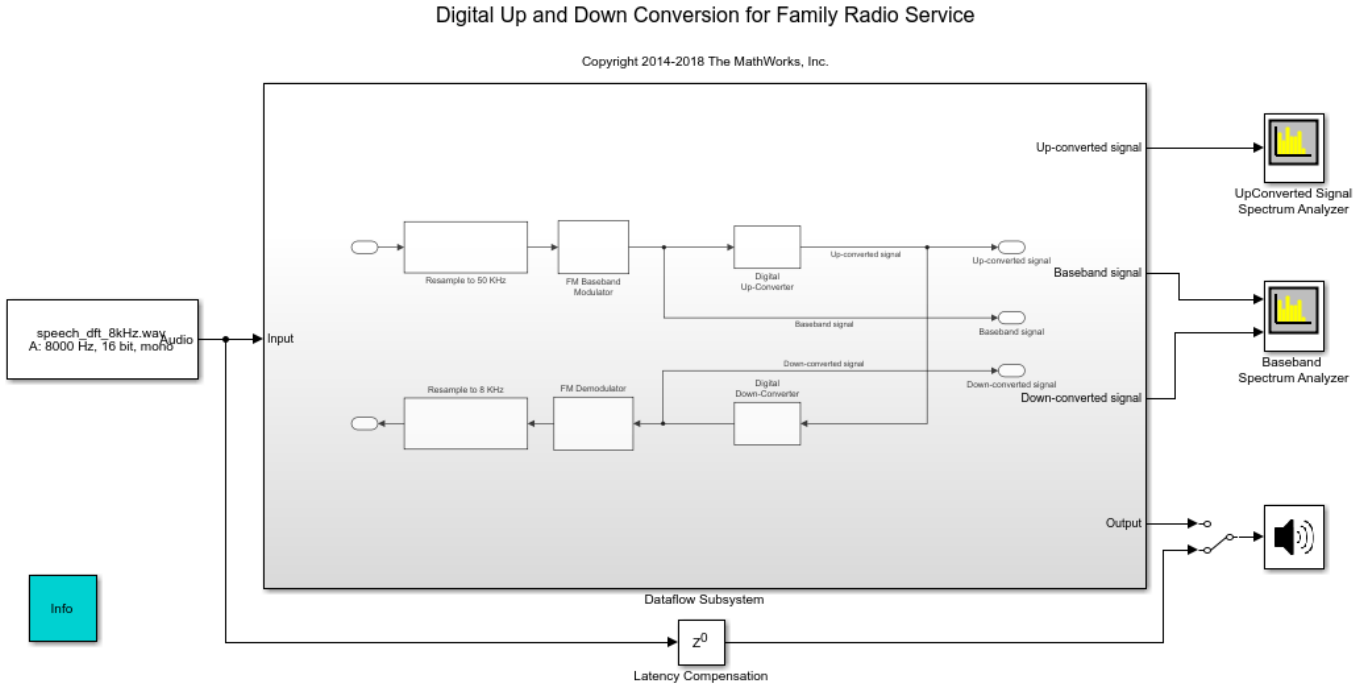
Introduction

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.



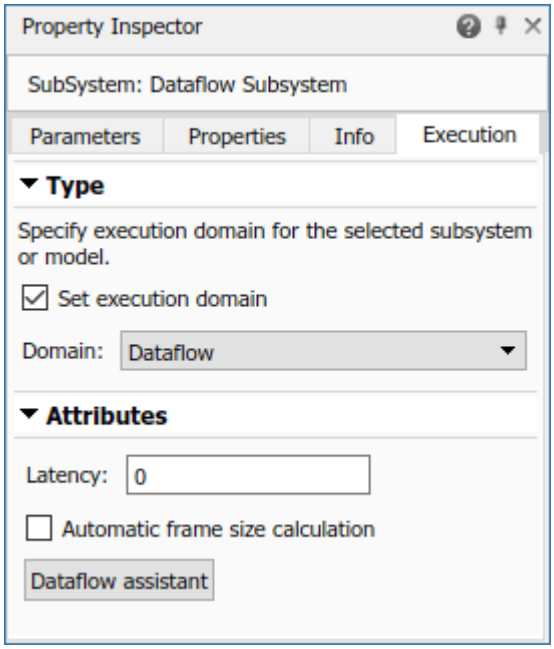
Simulate the model and measure the execution time. Execution time is measured using the output of the sim command which returns the simulation execution time of the model. To measure the time taken primarily for the Dataflow subsystem, comment out the Spectrum Analyzer blocks and Audio Device Writer block.

```
Simulation execution time for single-threaded model = 3.48s
```

Specify Dataflow Execution Domain

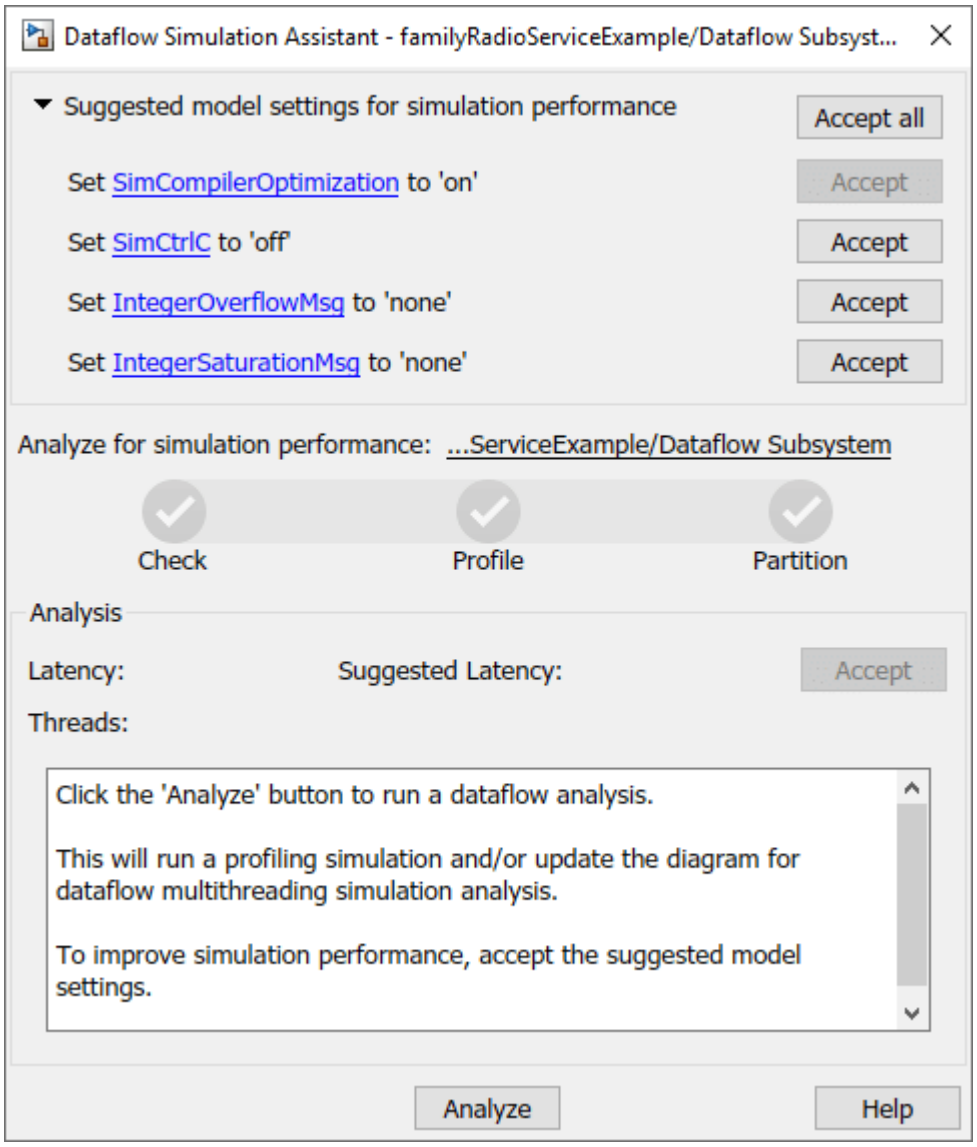
In Simulink, you specify dataflow as the execution domain for a subsystem by setting the Domain parameter to Dataflow using Property Inspector. You can view the Property Inspector for a subsystem, first by selecting the subsystem and then selecting View>Property Inspector. In the Property Inspector, 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.

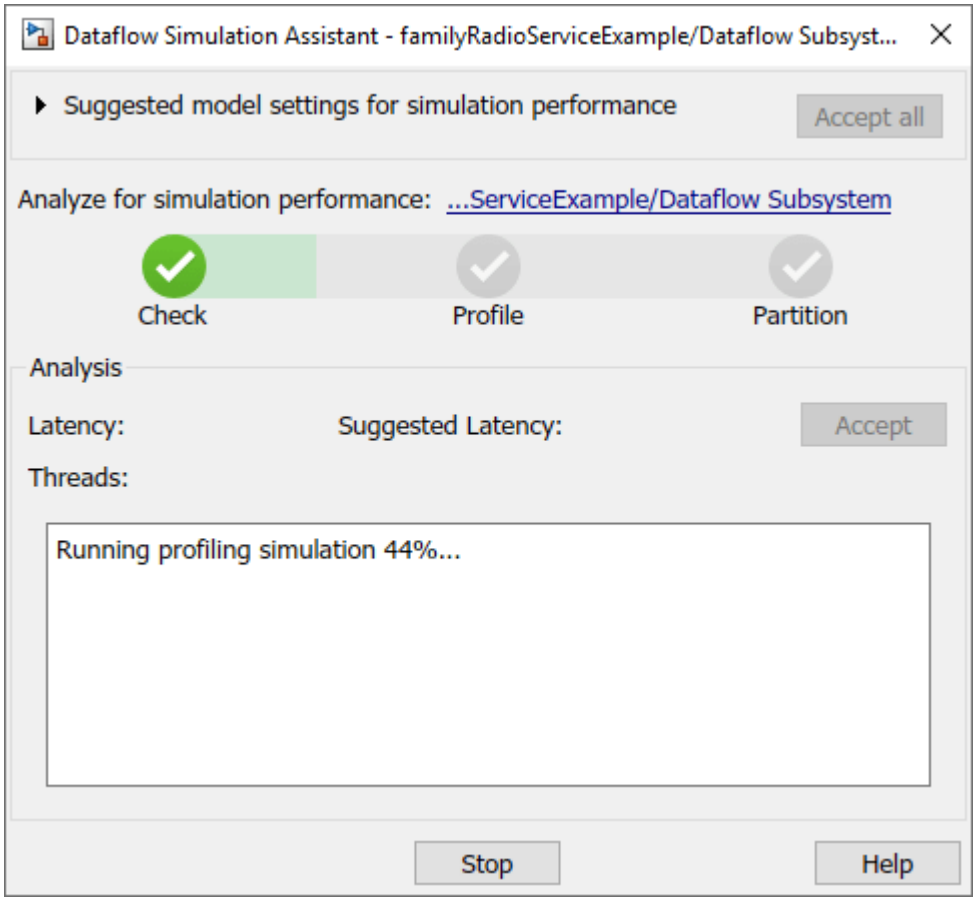


Multicore Simulation of Dataflow Domain

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.

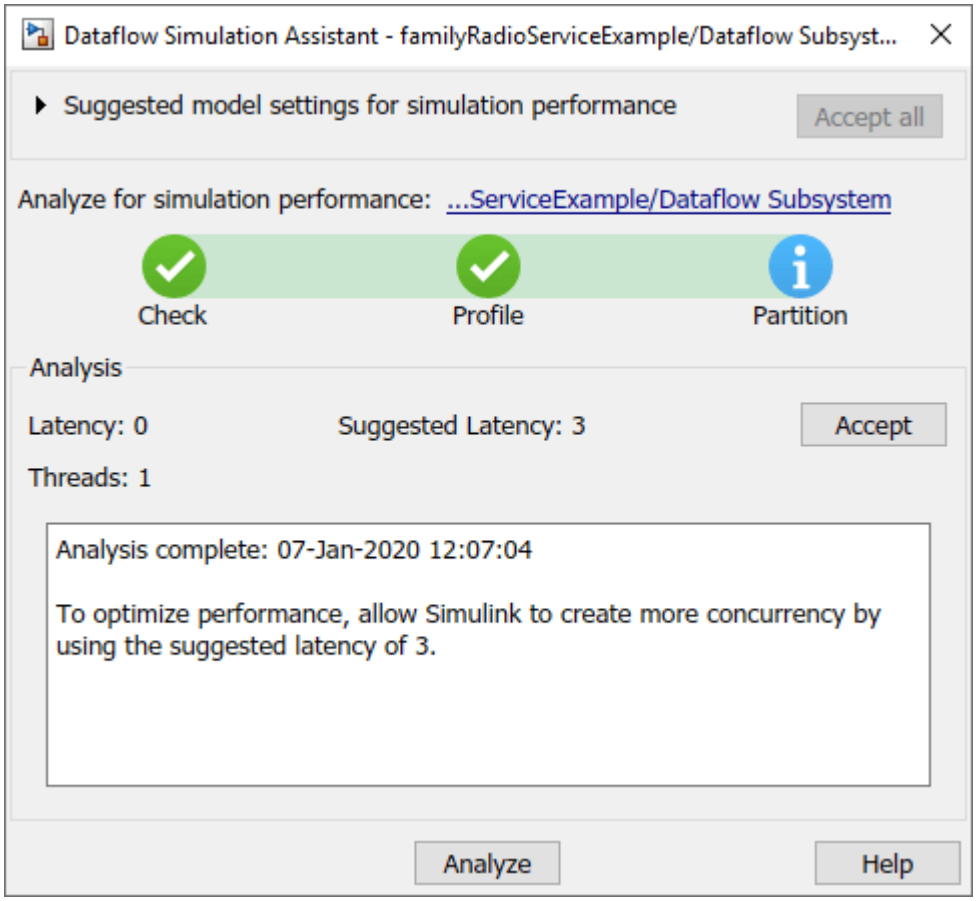


The Dataflow Simulation Assistant suggests changing model settings for optimal simulation performance. To accept the proposed model settings, next to **Suggested model settings for simulation performance**, click **Accept all**. Alternatively, you can expand the section to change the settings individually. 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.



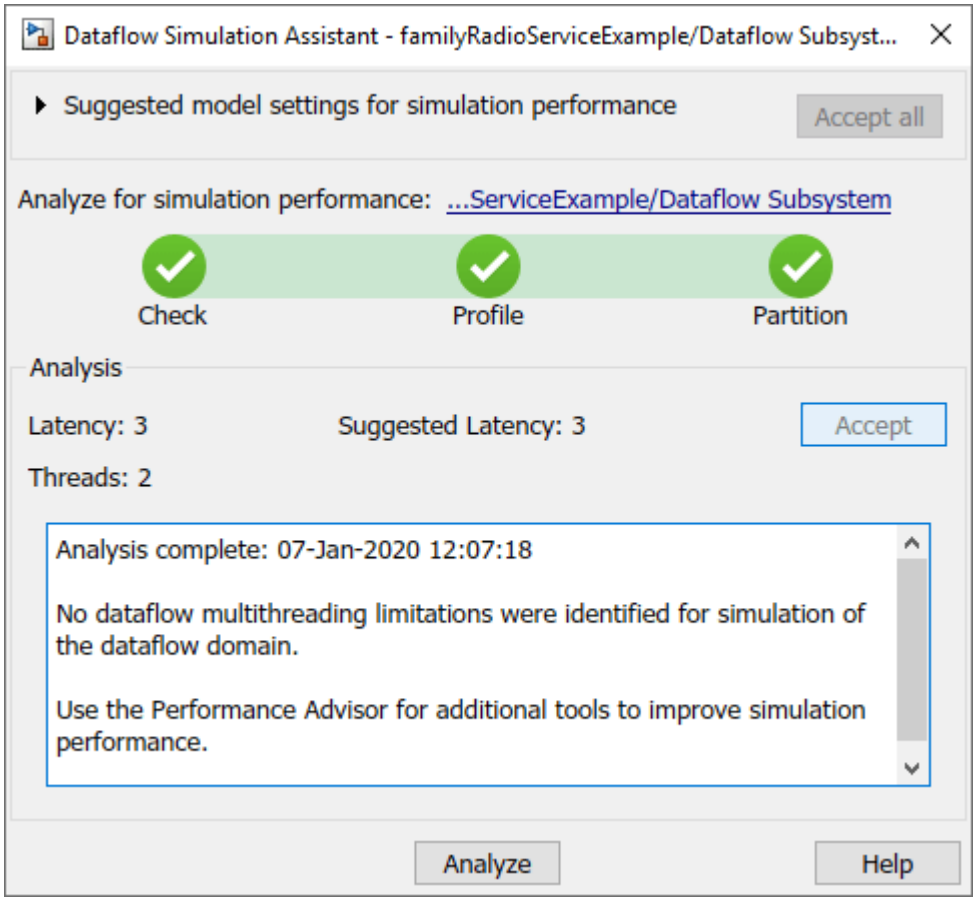
After analyzing the model, the assistant shows one thread because the data dependency between the blocks in the model prevents blocks from being executed concurrently. By pipelining the data dependent blocks, the Dataflow Subsystem can increase concurrency for higher data throughput. Dataflow Simulation 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

See Also

Dataflow Subsystem

More About

- “Perform Multicore Analysis for Dataflow” on page 8-34
- “Dataflow Domain” on page 8-2
- “Multicore Programming with Simulink” (Simulink)

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 “Install Support for Xilinx Zynq Platform” (Embedded Coder Support Package for Xilinx Zynq Platform).

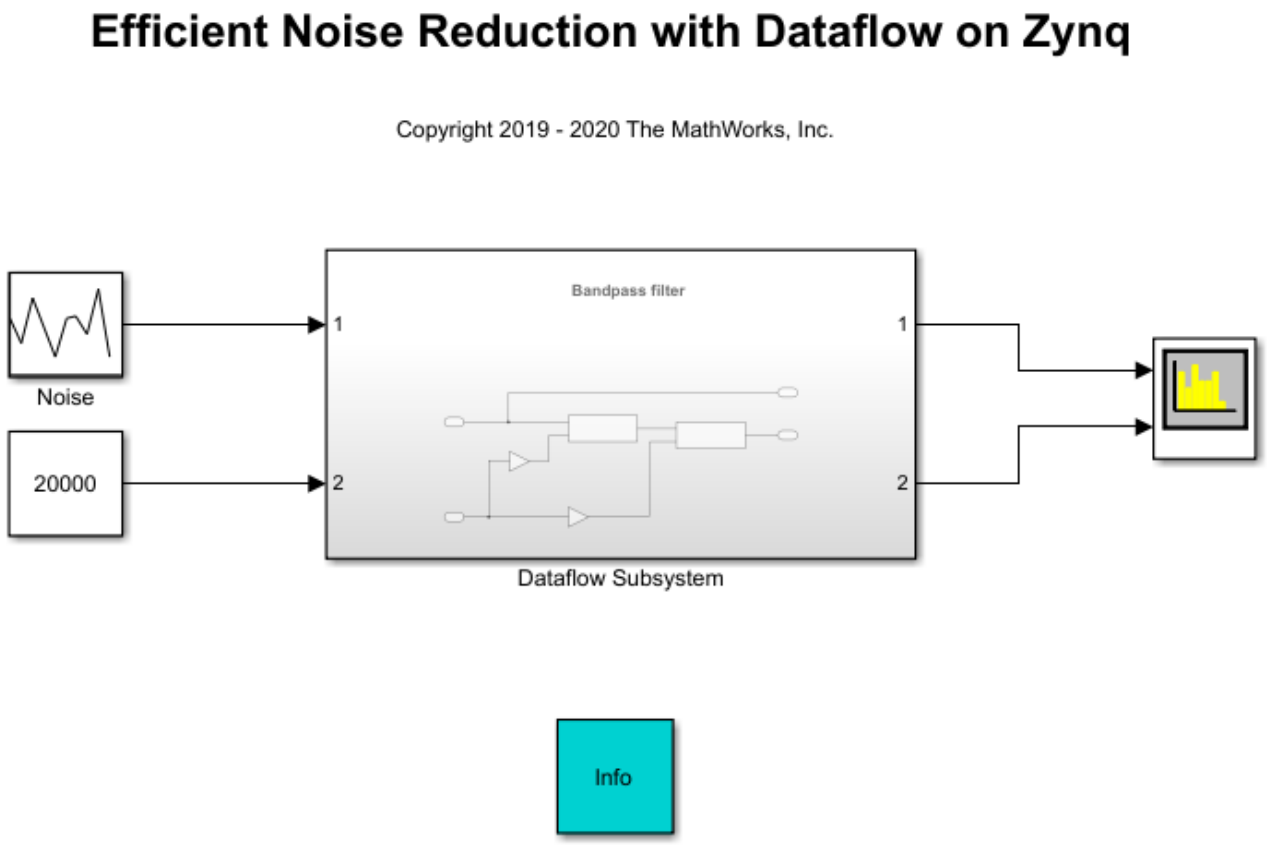
Introduction

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.

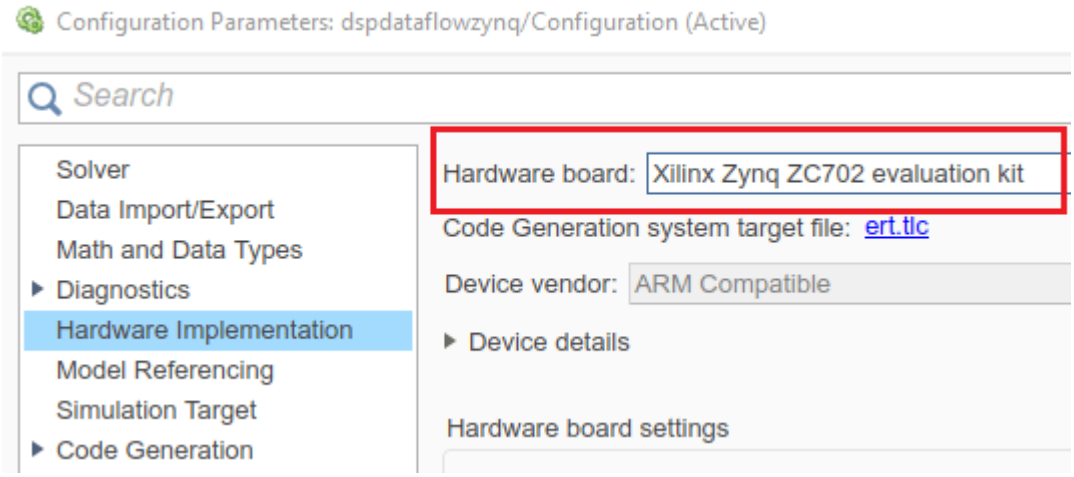
Noise Reduction System

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 dataflowzynq model.



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.



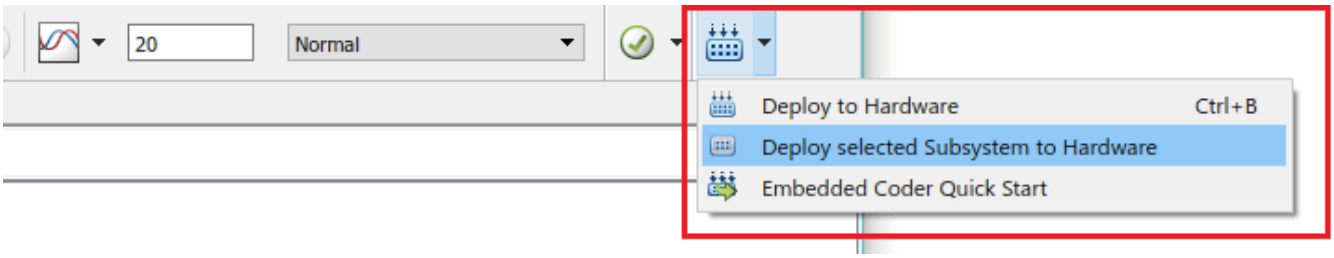
Configure PIL simulation and execution-time profiling

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 Subsystem Blocks” (Embedded Coder).

Enable profiling of function execution times for subsystems. For details on profiling with PIL, see “Code Execution Profiling with SIL and PIL” (Embedded Coder).

Generate code and simulate on target

Build the Dataflow Subsystem block. This step generates the PIL block from Dataflow Subsystem.



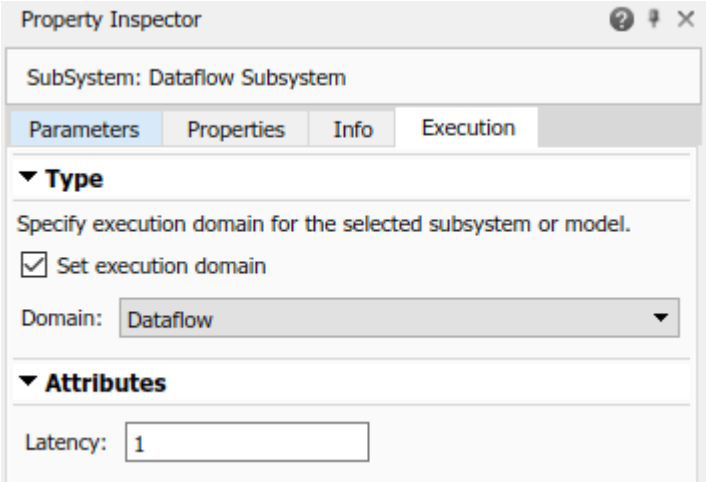
Replace Dataflow Subsystem block in the original model with the generated PIL block.

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.

Multi-Core 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 2020 The MathWorks, Inc.

See Also

Dataflow Subsystem

More About

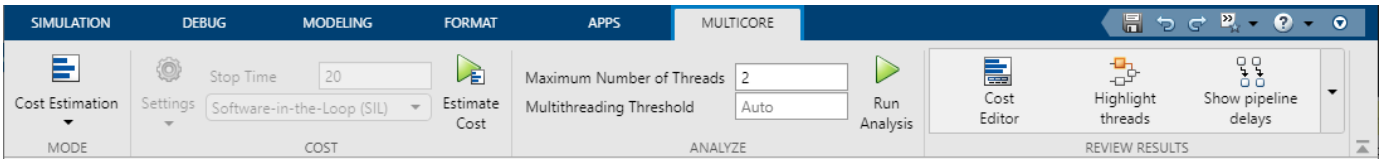
- “Dataflow Domain” on page 8-2
- “Perform Multicore Analysis for Dataflow” on page 8-34
- “Multicore Programming with Simulink” (Simulink)
- “Optimize and Deploy on a Multicore Target” (Simulink)

Perform Multicore Analysis for Dataflow

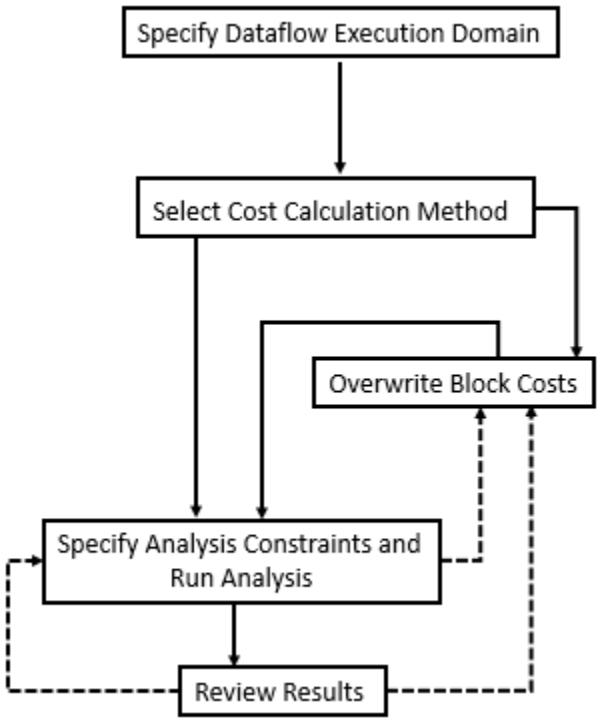
When a subsystem in a model is configured to use a dataflow execution domain, the **Multicore** tab is activated on the Simulink toolstrip. This tab consolidates multicore analysis techniques leveraged in dataflow into an incremental and iterative workflow.

Using the controls on the **Multicore** tab, you can:

- Estimate the relative cost of blocks using internal Simulink heuristics.
- Measure average execution times (cost) of blocks inside the dataflow subsystems by simulating the model with software-in-the-loop (SIL) or processor-in-the-loop (PIL) profiling . This functionality requires an Embedded Coder license.
- Manually override the block cost values.
- Provide analysis constraints, such as maximum number of threads and threading threshold.
- Run analysis to generate a block-to-threads allocation and visualize analysis results.

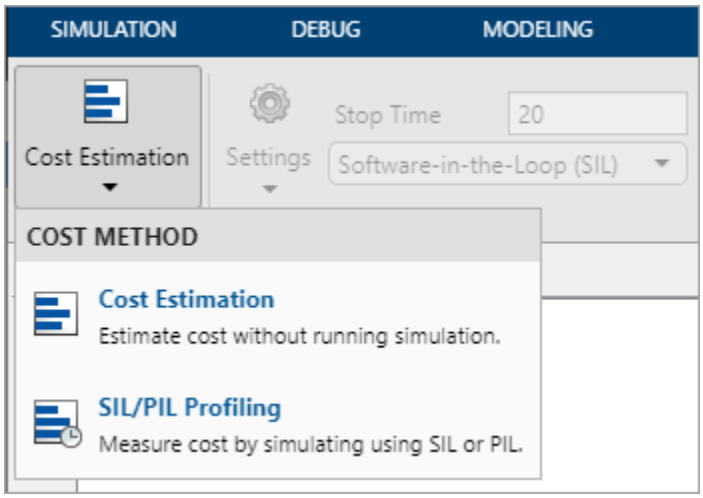


This chart illustrates the steps of multicore analysis. After you specify the dataflow execution domain for the subsystems in your model, you can select a cost calculation method, overwrite block costs, specify analysis constraints, run analysis, and review results.



Select the Cost Calculation Method

On the **Multicore** tab, in the **Mode** section, you can select the method of cost calculation as **Cost Estimation** or **SIL/PIL Profiling** . In both modes, the cost of individual blocks will be automatically determined and used in the multicore analysis for equal distribution of the computational load across multiple CPU cores.

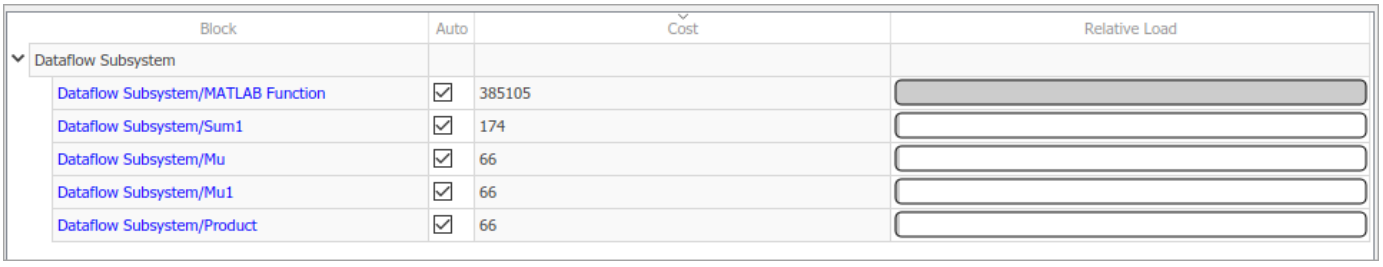


Cost Estimation

Use **Cost Estimation** for:

- Quick analysis without running the simulation or generating code.
- Preliminary analysis when the model is not fully implemented. In this case, you can modify the results of the estimation to match the anticipated cost values for the final implementation.

When you click **Estimate Cost**, the Cost Editor displays the estimated execution cost of each block in your model without simulating it.

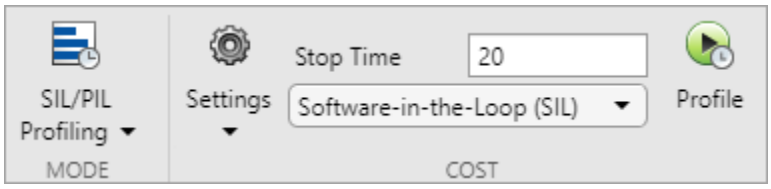


SIL/PIL Profiling

Use the software-in-the-loop (SIL) or processor-in-the-loop (PIL) profiling method (requires Embedded Coder license) to:

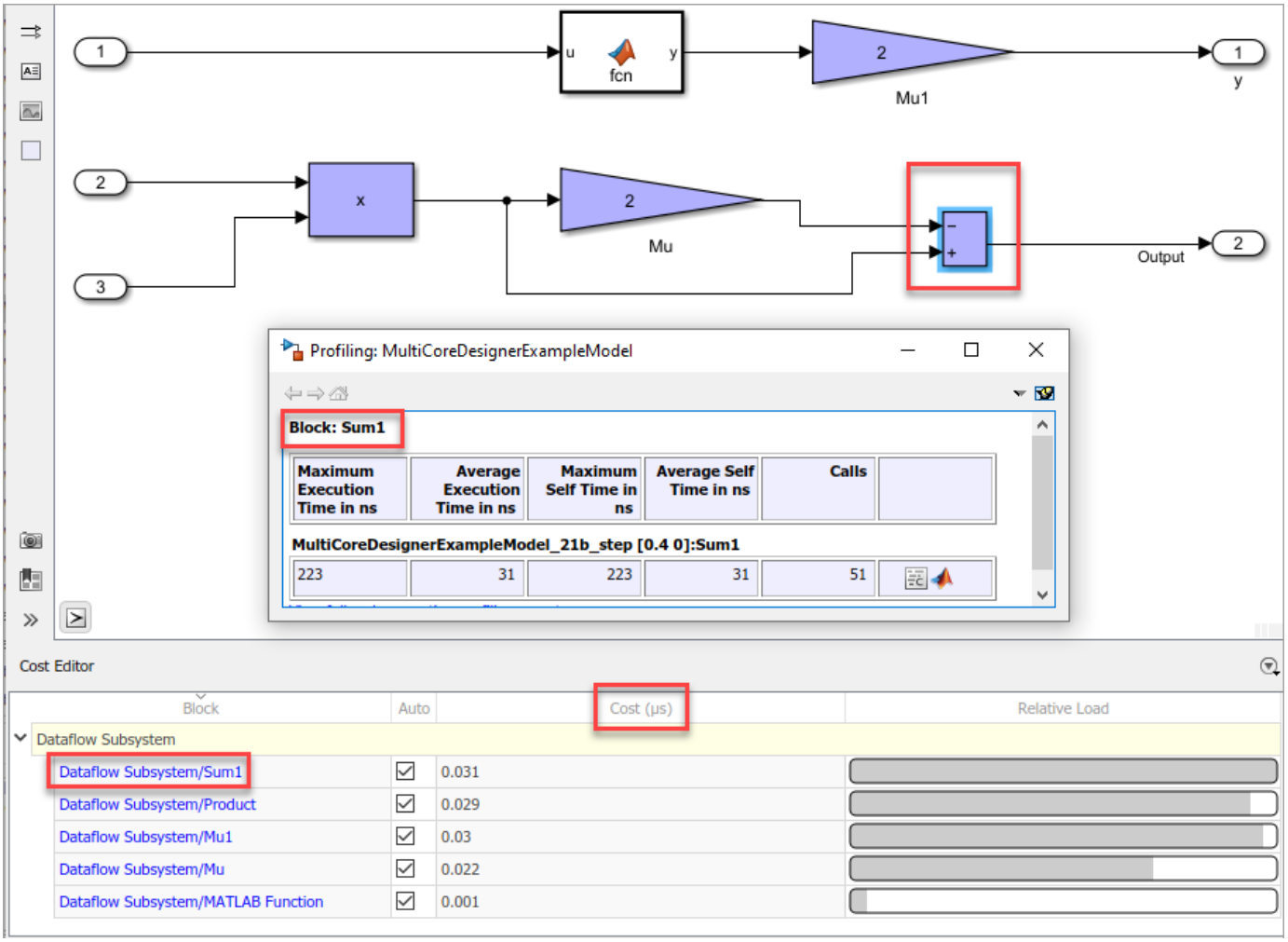
- Acquire accurate cost values measured on the host computer using the generated code. The generated code is the closest to the code that will be deployed on the hardware.
- Measure cost values on the actual target hardware in order to maximize the utilization of cores when the final code is deployed.

SIL/PIL profiling measures average execution times (cost) of blocks inside the dataflow subsystems by simulating the model with SIL/PIL.



- Use **Settings** to configure C/C++ code generation and hardware implementation settings.
- Use **Stop Time** to specify the time to measure the cost.
- Use the list to select the `Software-in-the-Loop (SIL)` or`Processor-in-the-Loop (PIL)` setting.
- Use **Profile** to measure the costs associated with blocks with the specified settings.

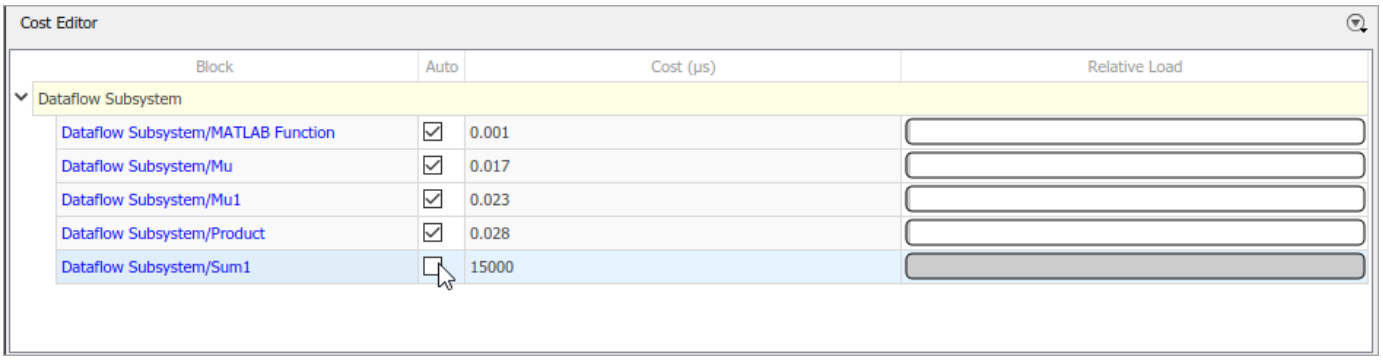
This example shows the highlighted block in the model and its cost. Observe that Cost Editor displays the units of the profiled cost values when you perform SIL/PIL profiling.



Manually Change Block Costs

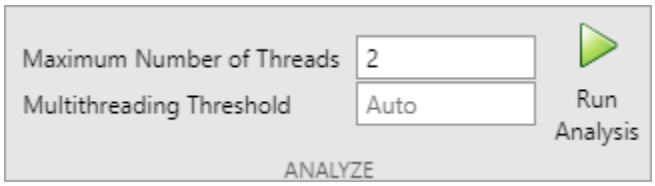
You can manually change the block cost values to understand their impact to the multicore behavior. To override block costs, clear the **Auto** column for the corresponding block and edit the value in the **Cost** column.

Overwriting block costs values allows you to perform analysis for custom costs.



Specify Analysis Constraints and Run Analysis

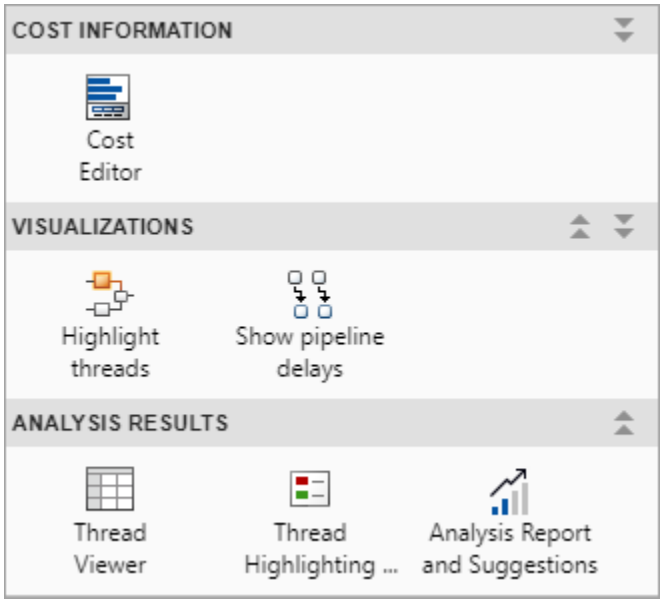
Next, set constraints and run multicore analysis. In the **Analyze** section:



- Use **Maximum Number of Threads** to specify the maximum number of threads produced by the analysis. By default, the tool automatically tries to determine the number of cores of the target processor from the hardware settings and uses that as maximum number of threads. If the tool is unable to determine the exact value, it will use the number of cores on the host platform as the maximum number of threads.
- Specify the **Multithreading Threshold** to set a minimum for the total cost (in microseconds) of the subsystem, for which the tool applies multithreading. If the total cost falls below the threshold, the tool will not partition the subsystem. By default, the tool uses a nominal value, 25 micro-seconds, as the threshold.
- Click **Run Analysis** to perform the analysis based on your configuration.

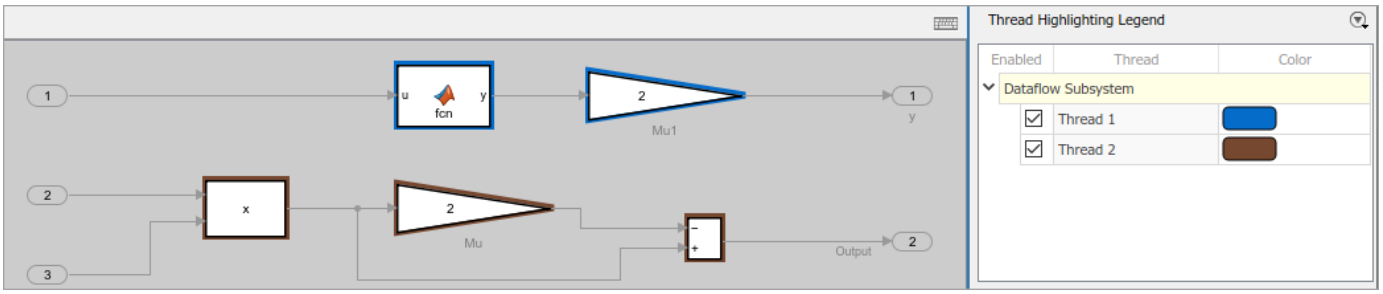
Review Results

Use the tools provided in the **Review Results** section to visualize and understand the multicore behavior of your model.

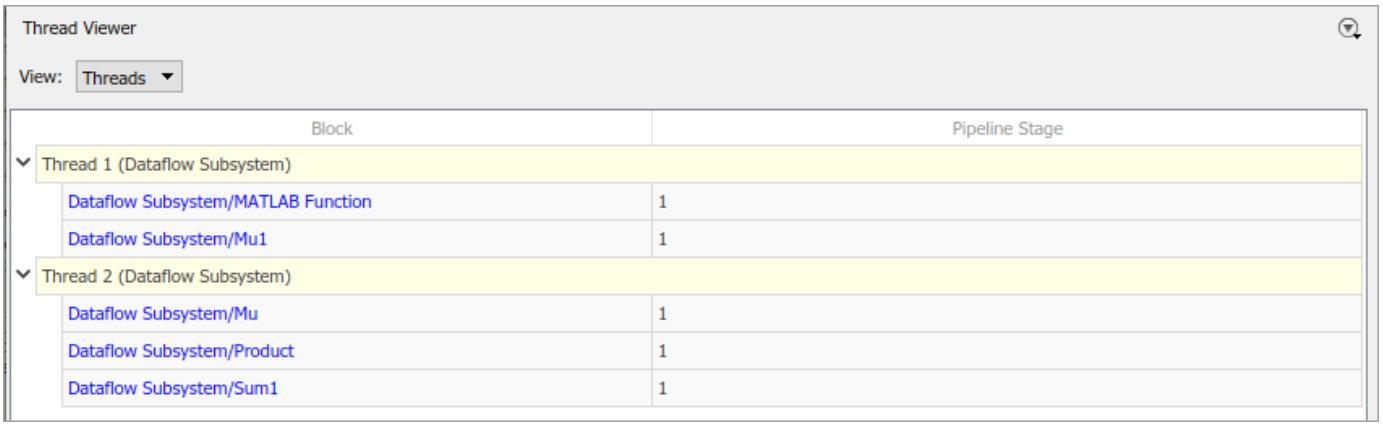


Highlight and View Threads

Select **Highlight threads** to highlight and visualize the threads and the assignment of blocks to the threads based on the block execution cost values.

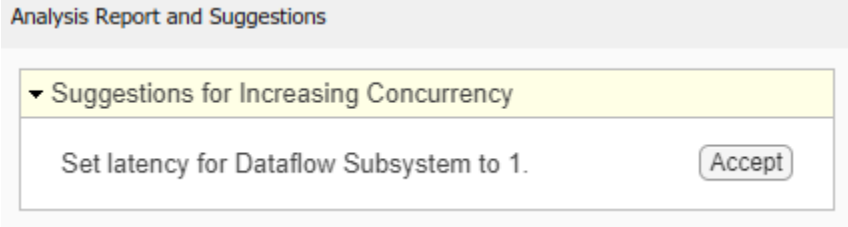


Select **Thread Viewer** to visualize the allocation of blocks to threads.

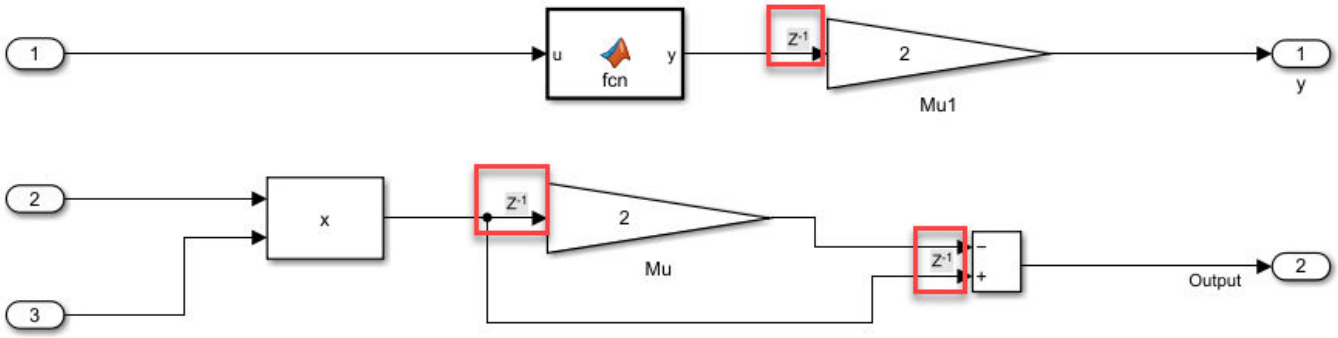


Analysis Report and Suggestions

Analyze the **Suggestions for Increasing Concurrency** section to see if there are suggested latencies for pipelining delays. By pipelining the data-dependent blocks, the Dataflow Subsystem block can increase concurrency for higher data throughput. For more information about pipelining delays, see “Multicore Simulation and Code Generation of Dataflow Domains” on page 8-12.



After accepting suggested latencies for pipelining delays, you can use **Show pipeline delays** to visualize the delays in your model.



Use the analysis report to investigate the relative weight of dataflow subsystems and the maximum theoretical speedup for the entire model. This speedup can be achieved as a result of the partitioning performed during the analysis. The amount of speedup is proportional to the relative weight of dataflow subsystems with respect to the entire model.

The analysis report displays total cost and number of threads values for each Dataflow Subsystem block.

The speedup is calculated using this formula, where n is the total number of Dataflow Subsystem blocks, `pctPar` is the percentage of the parallel execution of a subsystem, and `criticalPathCost` is the cost of the most costly thread in a subsystem.

$$Speedup \le \frac{1}{(1 - \sum_{i=0}^{n} pctPar_i) + \sum_{i=0}^{n} \frac{pctPar_i \times criticalPathCost_i}{totalCostInSubsystem_i}}$$

See Also

Dataflow Subsystem

More About

- “Multicore Analysis Using a Dataflow Domain” on page 8-41
- “Dataflow Domain” on page 8-2
- “Multicore Programming with Simulink” (Simulink)
- “Optimize and Deploy on a Multicore Target” (Simulink)

Multicore Analysis Using a Dataflow Domain

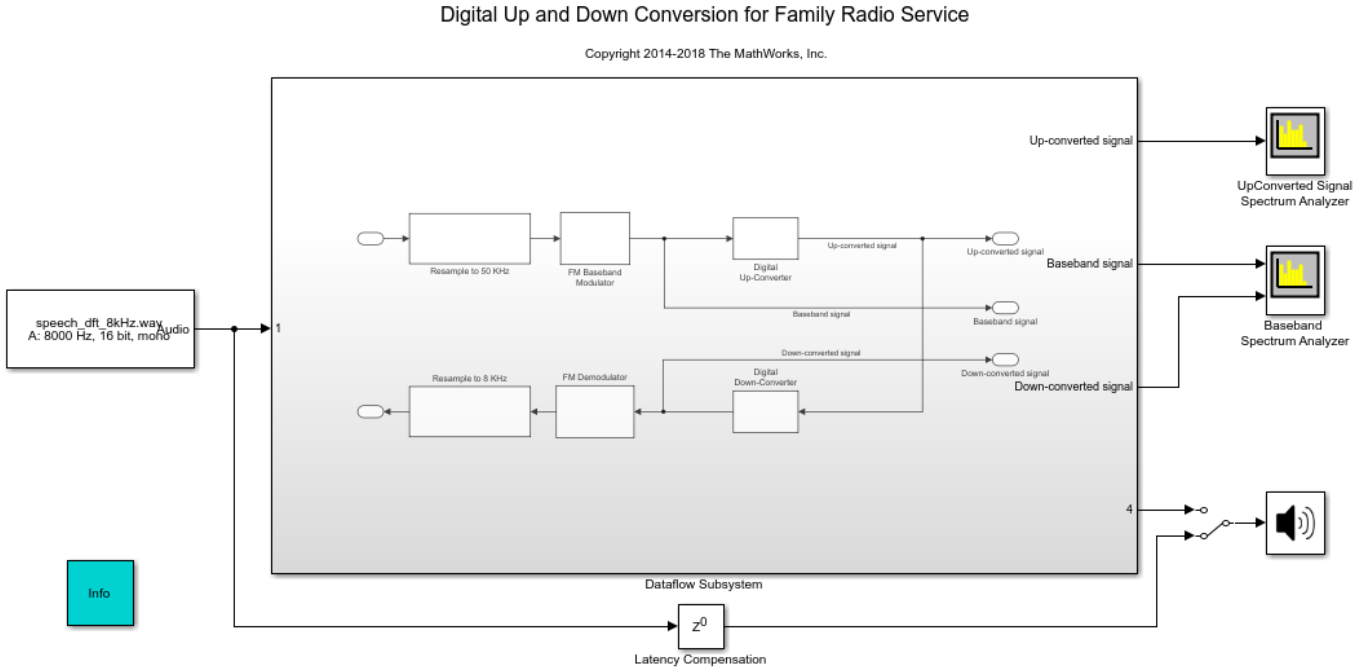
This example shows how to analyze the multicore execution behavior of a dataflow domain in Simulink.

Create a Family Radio Service System

The model in 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 DUC block converts a complex digital baseband signal to real passband signal. The DDC block converts the digitized real signal back to a baseband complex signal.

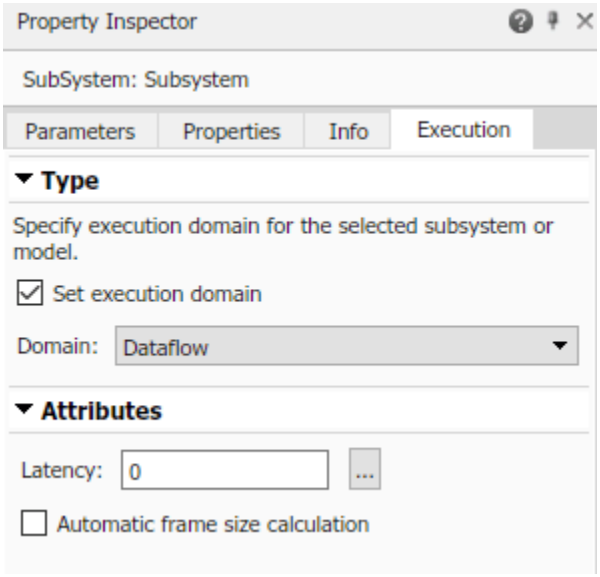
Open `familyRadioServiceMulticoreAnalysisExample` model.

```
model_name = 'familyRadioServiceMulticoreAnalysisExample';
open_system(model_name);
close_system([model_name '/UpConverted Signal Spectrum Analyzer'], 0);
close_system([model_name '/Baseband  Spectrum Analyzer'], 0);
```



Specify Dataflow Execution Domain

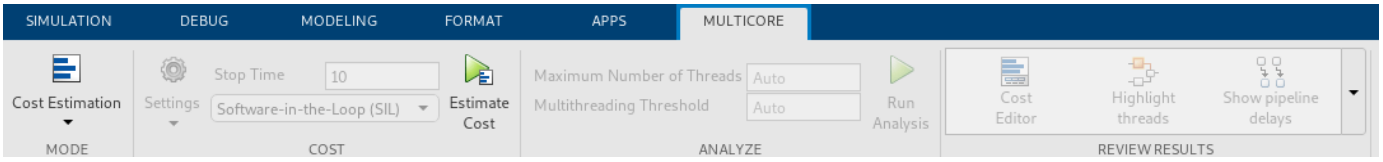
In Simulink®, to specify dataflow as the execution domain for a subsystem, use the Property Inspector to set the **Domain** parameter to `Dataflow`. To view the Property Inspector for a subsystem, select the subsystem, then select `View > Property Inspector`. Select **Set execution domain**, then click **Domain** and select `Dataflow`. You can also use a Dataflow Subsystem block from the Dataflow library of the DSP System toolbox to get a subsystem that is preconfigured with the dataflow execution domain.



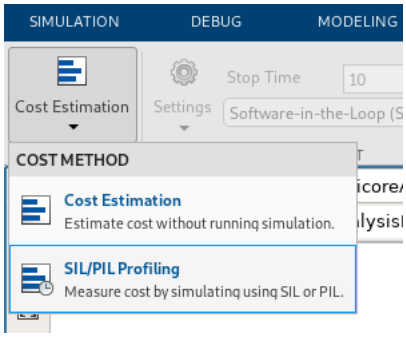

```
set_param([model_name,'/Dataflow Subsystem'],'SetDomainSpec','on');
set_param([model_name,'/Dataflow Subsystem'],'DomainSpecType','Dataflow');
set_param([model_name,'/Dataflow Subsystem'],'Latency','0');
set_param([model_name,'/Dataflow Subsystem'],'AutoFrameSizeCalculation','off');
```

Perform Multicore Analysis Using SIL Profiling

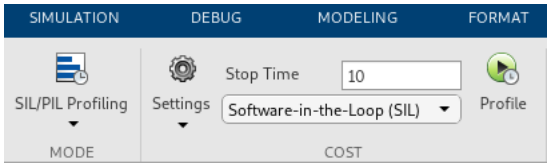
After specifying the dataflow execution domain, **Multicore** tab opens on the Simulink toolstrip.



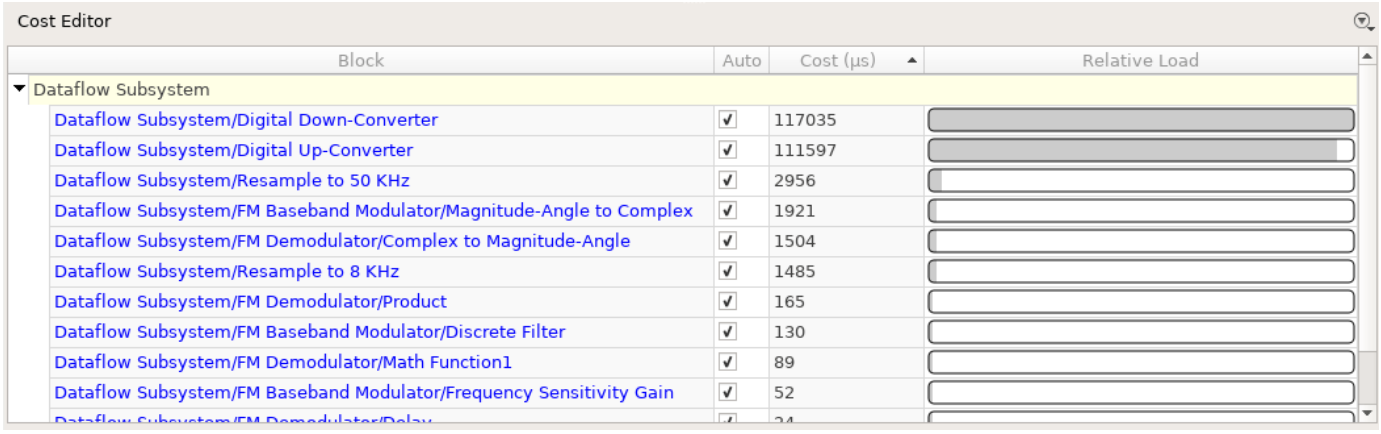
On the **Multicore** tab, click **Cost Estimation**. From the list, select **SIL/PIL Profiling**.



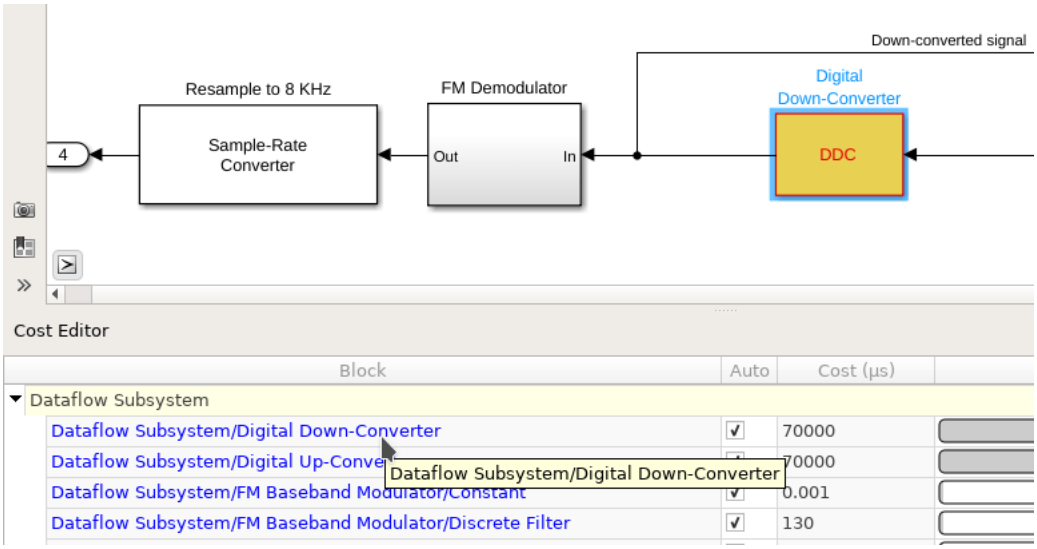
Click **Profile**.



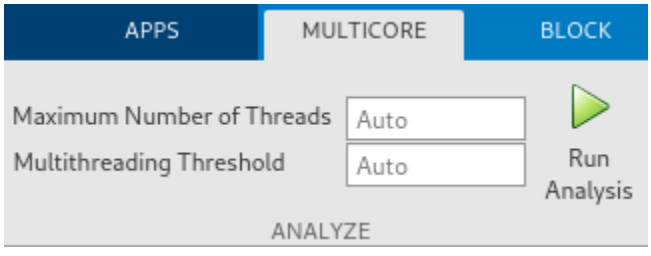
Once profiling finishes, the cost values display in the Cost Editor. Here, average execution time (cost) for each block is displayed in microseconds. The relative load of each block with respect to the most expensive block within the dataflow subsystem is indicated with bars of different length.



For example, the DDC block is the most expensive block in the table and has a cost of `117035` microseconds. Click on the block name in the **Block** column to highlight the corresponding block in the block diagram.

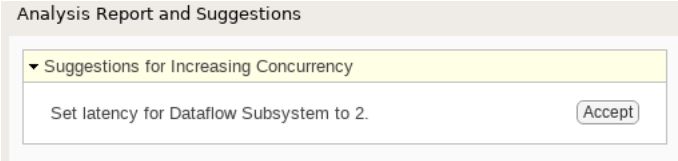


Click **Run Analysis**.



After analyzing the model, the Thread Highlighting Legend opens. The Thread Highlighting Legend shows one thread because the data dependency between blocks in the model prevents blocks from

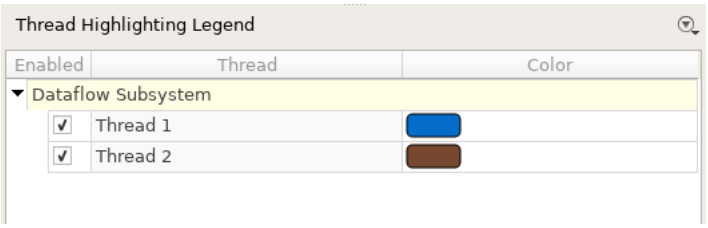
being executed concurrently. The Analysis Report and Suggestions pane that appears at the right of the canvas shows how to increase concurrency and obtain higher data throughput by pipelining the data-dependent blocks.



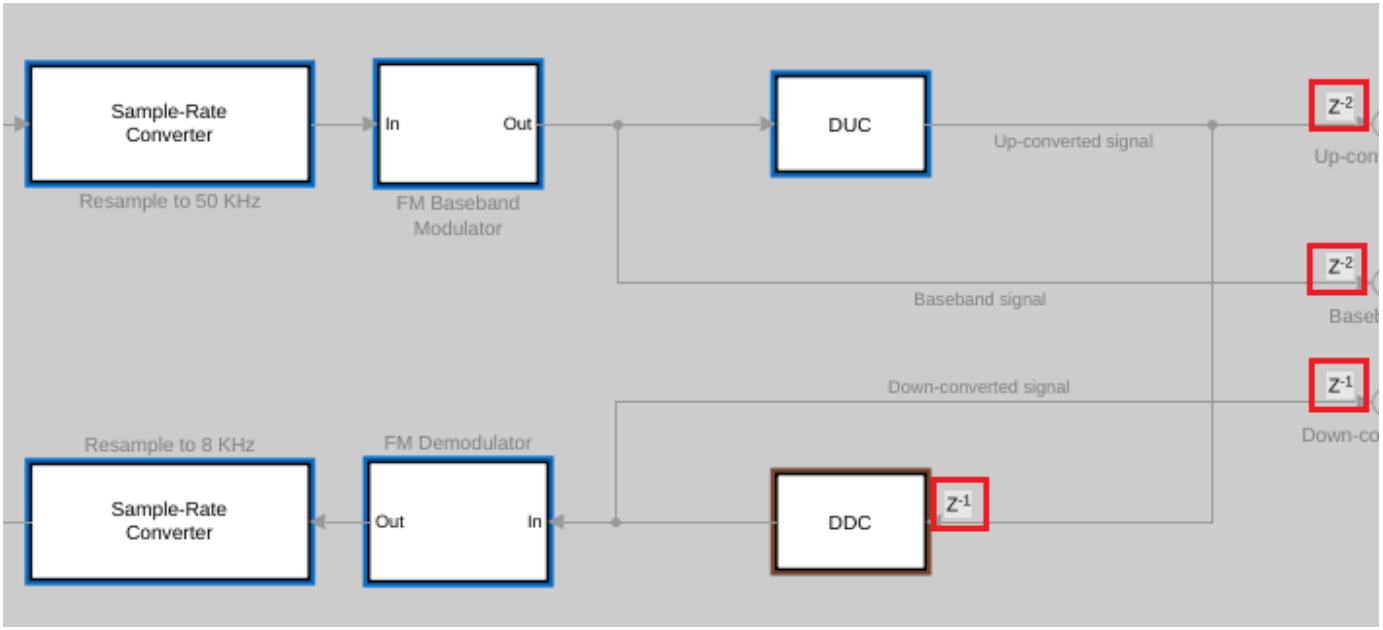
The Dataflow Subsystem specifies a latency value of `0`. The suggested latency for the system is `3`.

Click the **Accept** button to use the recommended latency for the Dataflow Subsystem and rerun the analysis. 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.

The Thread Highlighting Legend now shows two threads indicating that the blocks inside the dataflow subsystem can be executed in two parallel threads.



Inserted pipeline delays are shown in the canvas using Z^{-1} tags.

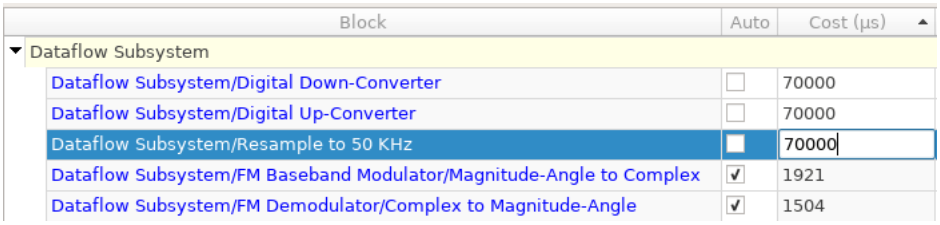


Check Analysis Report

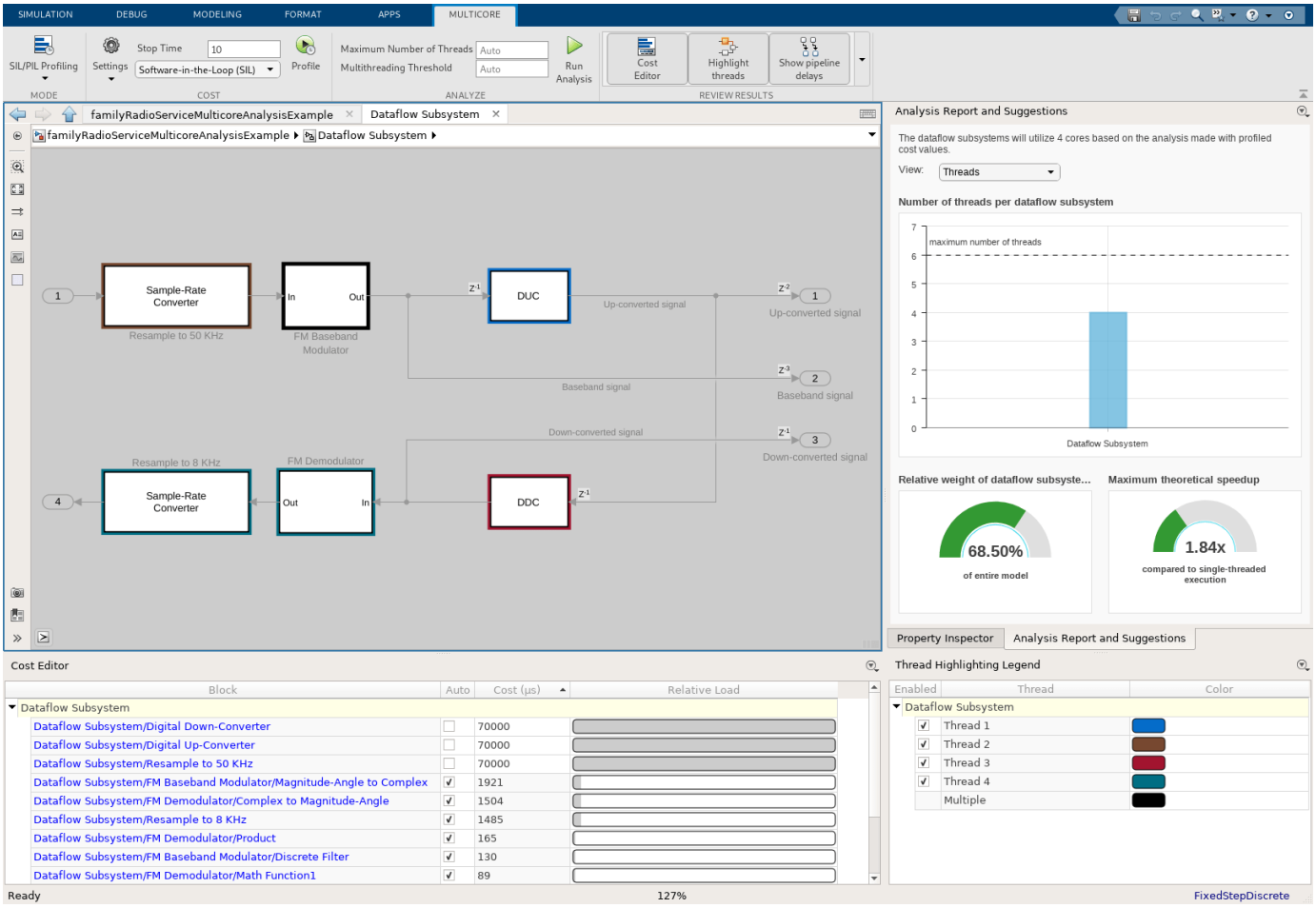
At the bottom of the Analysis Report and Suggestions pane, the maximum theoretical speedup is shown as `1.51x` for the model as a result of the partitioning performed during the analysis.

Manually Override Cost

To perform design space exploration, you can manually change the cost of the blocks. The first two blocks are relatively more expensive than other blocks in the subsystem, which should significantly influence how the blocks are mapped to threads. Divide the total sum of the cost for the first two blocks by three, then manually assign that number to the first three blocks by clearing checkboxes in the **Auto** column and editing the values in the **Cost** column.



Click **Run Analysis** to rerun the analysis, then accept the provided latency suggestion. The new result indicates that the subsystem is now partitioned into four threads with a corresponding theoretical speedup value of `1.84x`.



See Also

Dataflow Subsystem

More About

- “Perform Multicore Analysis for Dataflow” on page 8-34
- “Dataflow Domain” on page 8-2
- “Multicore Programming with Simulink” (Simulink)
- “Optimize and Deploy on a Multicore Target” (Simulink)

Simulink HDL Optimized Block Examples in DSP System Toolbox

- “Generate a Sine Wave” on page 9-2
- “Implement CIC Decimation Filter for HDL” on page 9-5
- “Fully Parallel Systolic FIR Filter Implementation” on page 9-8
- “Partly Serial Systolic FIR Filter Implementation” on page 9-12
- “Optimize Programmable FIR Filter Resources” on page 9-16
- “FIR Decimation for FPGA” on page 9-21
- “High Throughput Channelizer for FPGA” on page 9-23
- “Implement atan2 Function for HDL” on page 9-31
- “Downsample a Signal” on page 9-34
- “Control Data Rate Using the Ready and Request Ports” on page 9-37
- “Implement FFT for FPGA Using FFT HDL Optimized Block” on page 9-40
- “Automatic Delay Matching for the Latency of FFT HDL Optimized Block” on page 9-44

Generate a Sine Wave

This example shows how to use the NCO HDL Optimized block to generate a sine wave. The example generates a sine wave with these specifications.

```
F0 = 510;       % Desired output frequency (Hz)
Deltaf = 0.05;  % Frequency resolution (Hz)
SFDR = 90;      % Spurious free dynamic range (dB)
Ts = 1/8000;    % Sample period (s)
phOffd = pi/2;  % Desired phase offset (rad)
```

The frequency resolution of the sine wave depends on the size of the accumulator. Calculate the number of required accumulator bits to achieve the desired frequency resolution. `N` must be an integer value. Use this value to set the accumulator data type **Word length** parameter.

$$N = \text{ceil}(\text{log2}(\frac{1}{T_s \cdot \Delta f}))$$

```
N = ceil(log2(1/(Ts*Deltaf))) %#ok<*NOPTS>
```

```
N =

    18
```

Quantizing the output of the accumulator allows you to achieve better frequency resolution without increasing the lookup table size. Calculate the number of quantized accumulator bits from the spurious free dynamic range specification. Ensure that the **Quantize phase** parameter is selected, and then use this value to set the set the **Number of quantizer accumulator bits** parameter.

$$Q = \text{ceil}(\frac{SFDR - 12}{6})$$

```
Q = ceil((SFDR - 12)/6)
```

```
Q =

    13
```

Calculate the phase increment. Use this value to set the **Phase increment** parameter.

$$phIncr = \text{round}(F_0 \cdot 2^N T_s)$$

```
phIncr = round(F0*2^N*Ts)
```

```
phIncr =

       16712
```

Phase offset and dither are optionally added in the accumulator stage. Calculate the phase offset that the block adds from the desired phase offset of the output wave. Use this value to set the **Phase offset** parameter.

$$phOff = \frac{2^N \cdot PhOffd}{2\pi}$$

```
phOff = (2^N*phOffd)/(2*pi)
```

```
phOff =

       65536
```

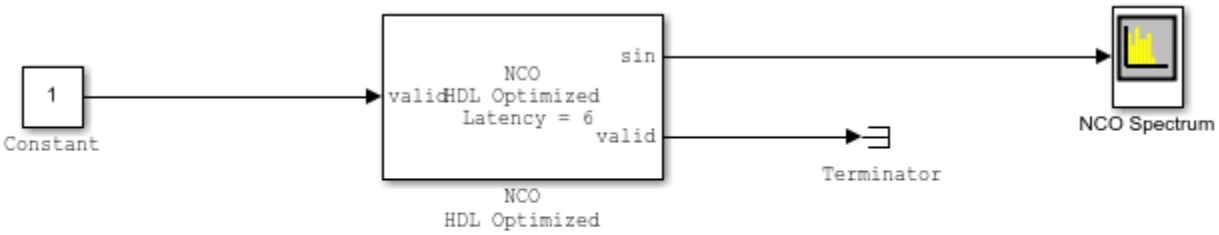
Select the number of dither bits. In general, a good choice for the number of dither bits is the accumulator word length minus the number of quantized accumulator bits. Use this value to set the **Number of dither bits** parameter.

```
ditherBits = N - Q
```

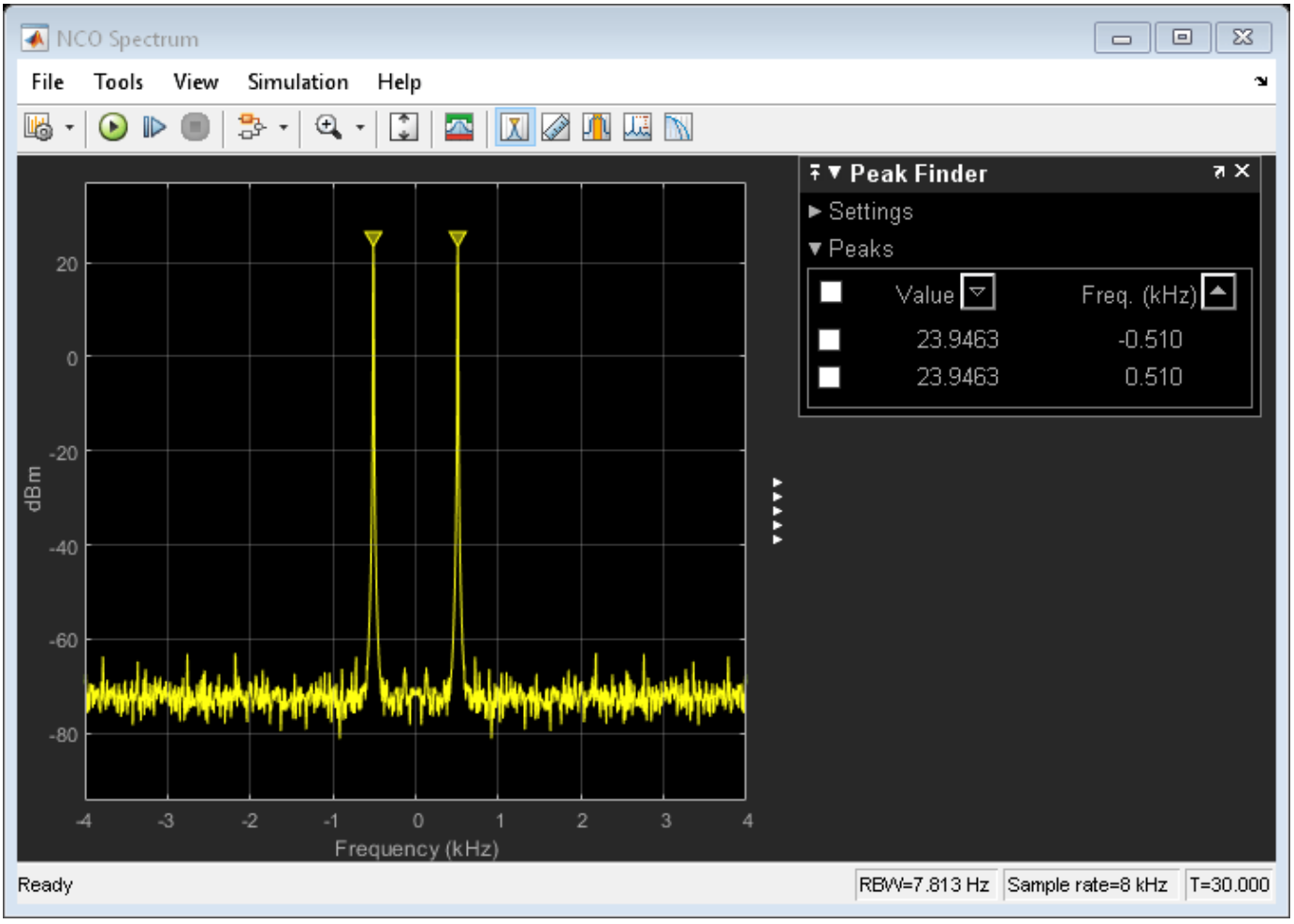
```
ditherBits =

     5
```

Open and simulate the model. The NCO HDL Optimized block in the model is configured with the variables calculated in this script. The output word length and fraction length depend on the constraints of your hardware; this example uses a word length of 16 and a fraction length of 14.

```
open_system('GenerateSineWaveHDL')
sim('GenerateSineWaveHDL')
```



Copyright 2016 The MathWorks, Inc.



The Peak Finder results in the Spectrum Analyzer show that the output waveform is generated at 510 Hz. 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.

Implement CIC Decimation Filter for HDL

This example shows how to use a CIC Decimation HDL Optimized block to filter and downsample data. This block supports scalar and vector inputs. In this example, two Simulink® models are provided to work with scalar and vector inputs separately. You can generate the HDL code from the subsystems in these Simulink models.

Set Up Input Data Parameters

Set up these workspace variables for the models to use. These variables configure the CIC Decimation HDL Optimized block inside of them. This block supports fixed and variable decimation rates for scalar inputs and only a fixed decimation rate for vector inputs. The example runs the `HDLCICDecimationModel.slx` model when you set the `scalar` value to `true` and runs the `HDLCICDecimationModelVectorSupport.slx` model when you set the `scalar` value to `false`. For scalar inputs, choose a range for the input `varRValue` values and set the decimation factor value, R, to the maximum expected decimation factor. For vector inputs, the input data must be a column vector of size 1 to 64. R must be an integer multiple of input frame size.

```
R = 8; % Decimation factor
M = 1; % Differential delay
N = 3; % Number of sections
scalar = false; % true for scalar; false for vector
if scalar
    varRValue = [4,R];
    vecSize = 1;
    modelname = 'HDLCICDecimationModel';
else
    varRValue = R;
    fac = (factor(R));
    vecSize = fac(randi(length(fac),1,1));
    modelname = 'HDLCICDecimationModelVectorSupport';
end

numFrames = length(varRValue);
dataSamples = cell(1,numFrames);
varRtemp = cell(1,numFrames);
cicFcnOutput = [];
WL = 0; % Word length
FL = 0; % Fraction length
```

Generate Reference Output from `dsp.CICDecimation` System Object™

Generate frames of random input samples and provide them as input to the `dsp.CICDecimation` System object. The output generated from this System object is used as a reference data for comparison. This System object does not support variable decimation rate, so you must create and release this object for any change in the decimation factor value.

```
for i = 1:numFrames
    framesize = varRValue(i)*randi([5 20],1,1);
    dataSamples{i} = fi(randn(vecSize,framesize),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
```

Convert Input to Stream of Samples and Import them to Simulink® Model

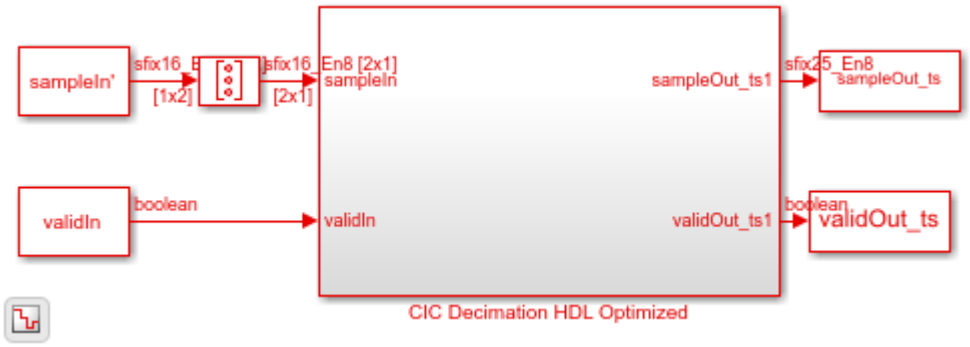
Generate a stream of samples by converting frames to samples. Provide those samples (`sampleIn`) and the valid signal (`validIn`) as inputs to the Simulink model. The latency of the block for scalar and vector inputs is calculated based on the type of input and the number of sections, `N`. For more information about latency, see the “Gain correction” parameter. To flush remaining data, run the model by inserting the required number of idle cycles after each frame using the `idlecyclesbetweenframes` value.

```
idlecyclesbetweensamples = 0;
idlecyclesbetweenframes = floor((vecSize-1)*(N/vecSize))+1 + N + (2+(vecSize+1)*N) + 9;

sampleIn = [];
validIn = [];
varRIn = [];
len = 0;

for ij = 1:numFrames

    dataInFrame =  dataSamples{ij};
    if scalar
        len = length(dataInFrame);
    else
        len = size(dataInFrame,2);
    end

    data = []; valid=[]; varR = [];
    for ii = 1:len
        data = [data dataInFrame(:,ii) ...
            zeros(vecSize,idlecyclesbetweensamples)];
        valid = [valid true(1,1) ...
            false(1,idlecyclesbetweensamples)];
        varR = [varR varRtemp{ij}(ii) ...
            zeros(1,idlecyclesbetweensamples)];
    end

    sampleIn = cast([sampleIn,data,zeros(vecSize,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 Simulink Model

Run the model. Running the model imports the input signal variables from the MATLAB® workspace to the CIC Decimation HDL Optimized block in the model.

```
open_system(modelname);
sim(modelname);
```



Compare Simulink Block Output with MATLAB System Object Output

Capture the output of the Simulink block. Compare that ouput with the output of 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 that differed between Simulink block output and MATLAB System
```

```
HDL CIC Decimation

Total number of samples that differed between Simulink block output and MATLAB System object outp
```

See Also

Blocks
CIC Decimation HDL Optimized

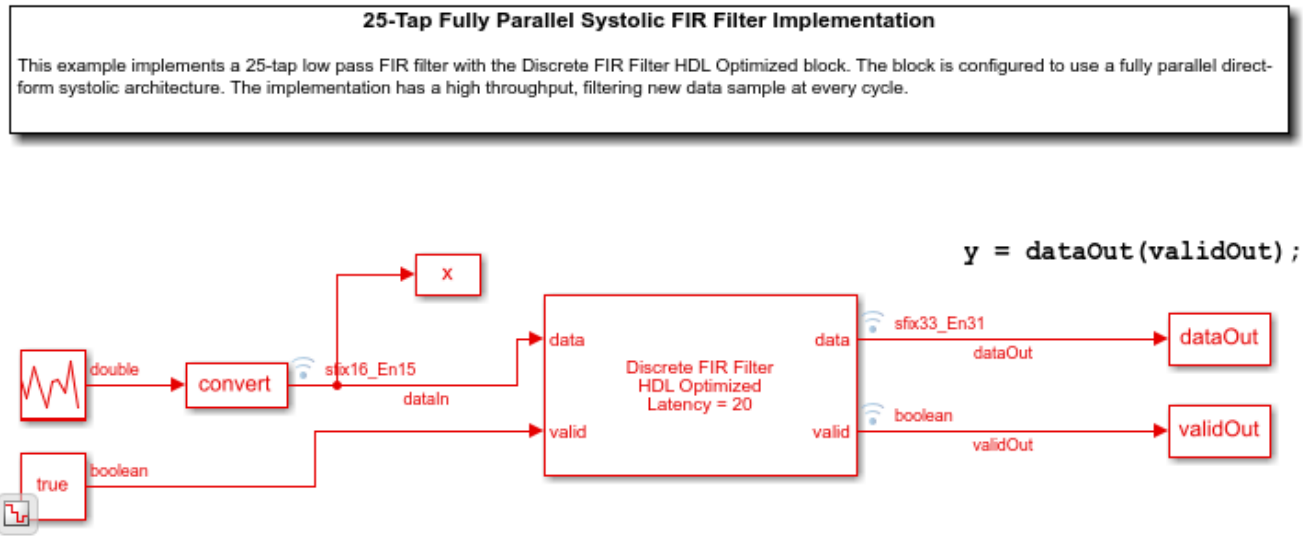
Fully Parallel Systolic FIR Filter Implementation

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.

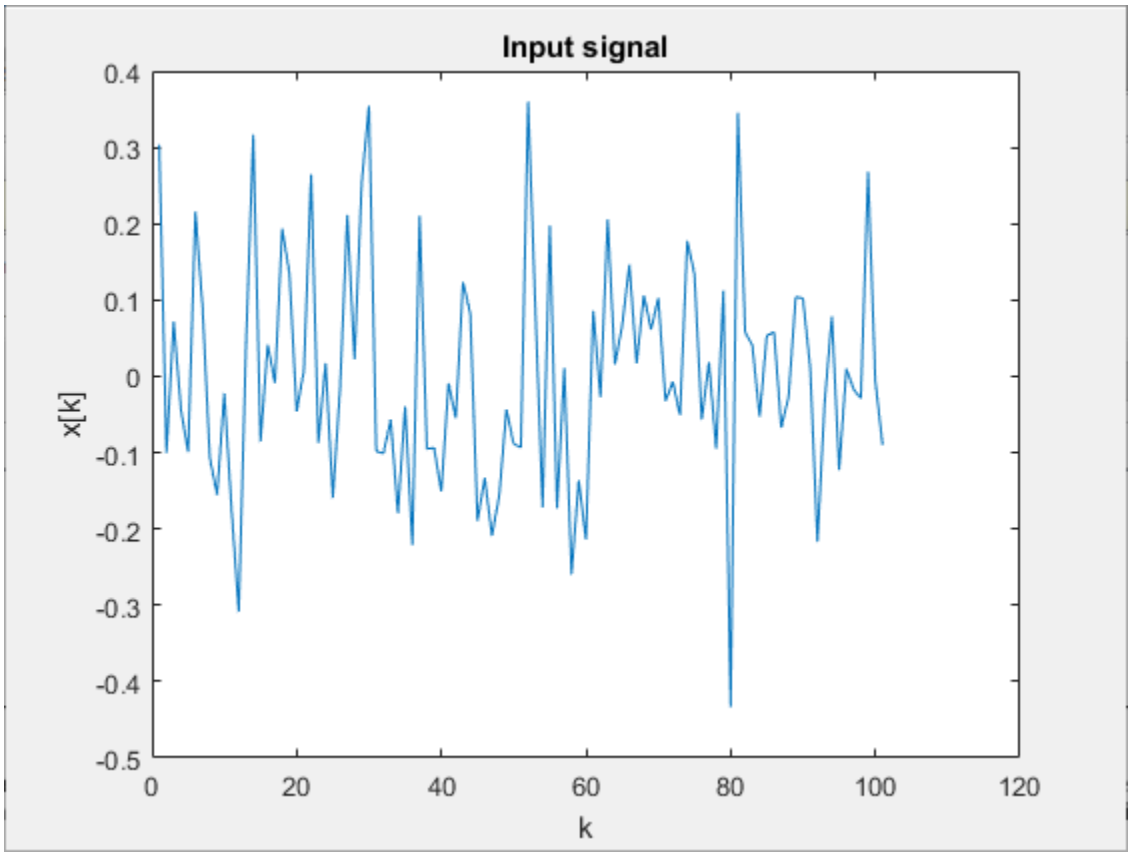
Open the Model

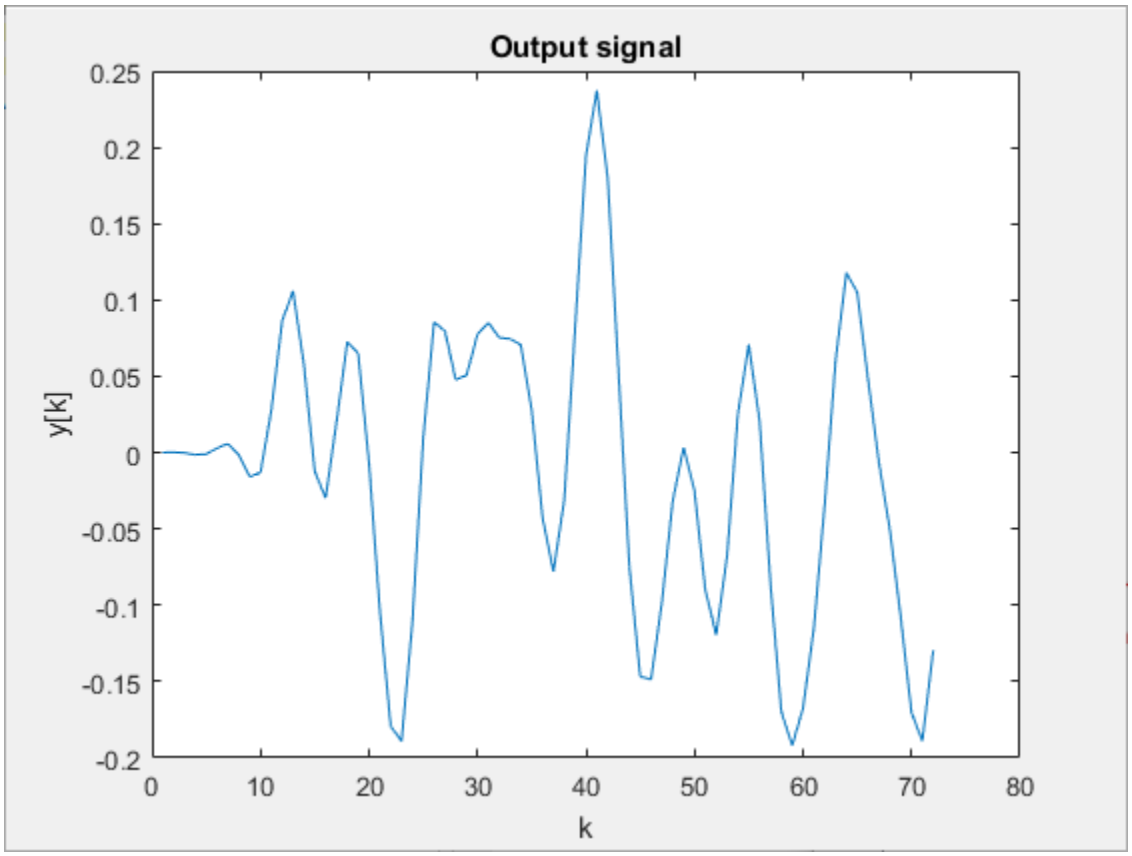
Open the model. Inspect the block parameters. The **Filter structure** is set to `Direct form systolic`.



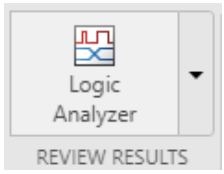
Run the Model and Inspect Results

Run the model. Observe the input and output signals in the generated plots.

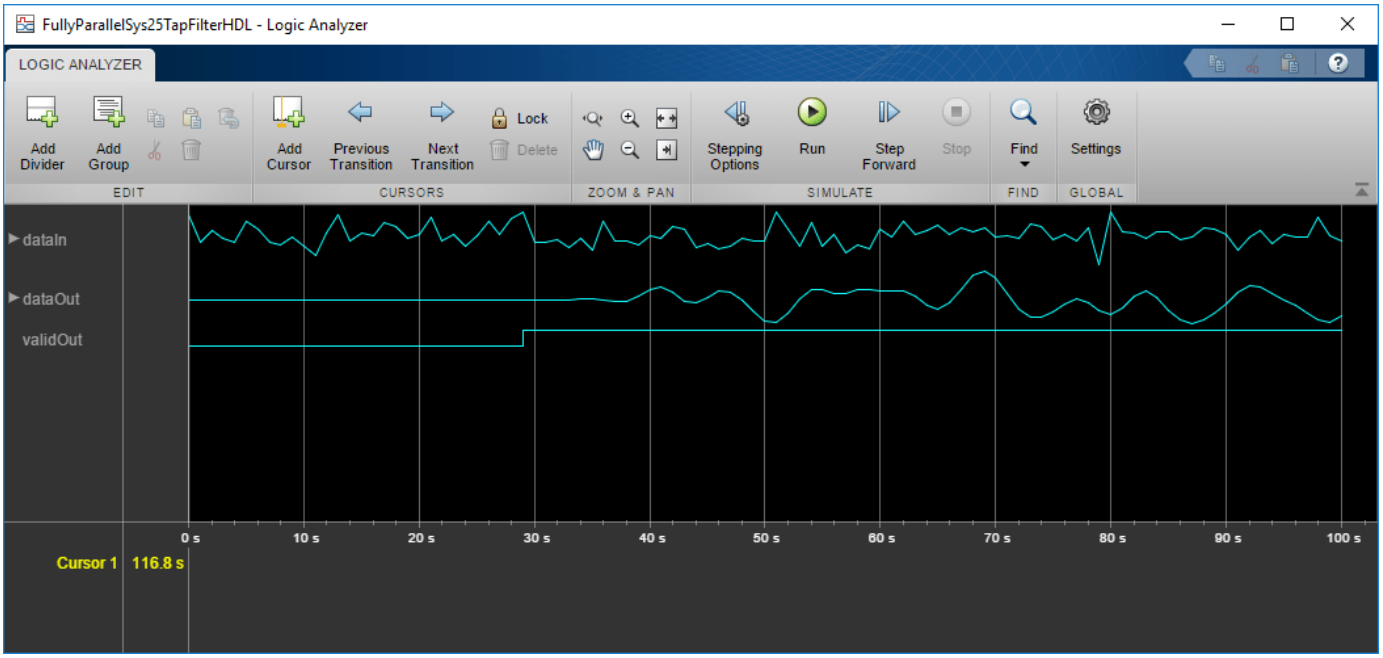

Input signal
x[k]
k
0.4
0.3
0.2
0.1
0
-0.1
-0.2
-0.3
-0.4
-0.5
0
20
40
60
80
100
120



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.

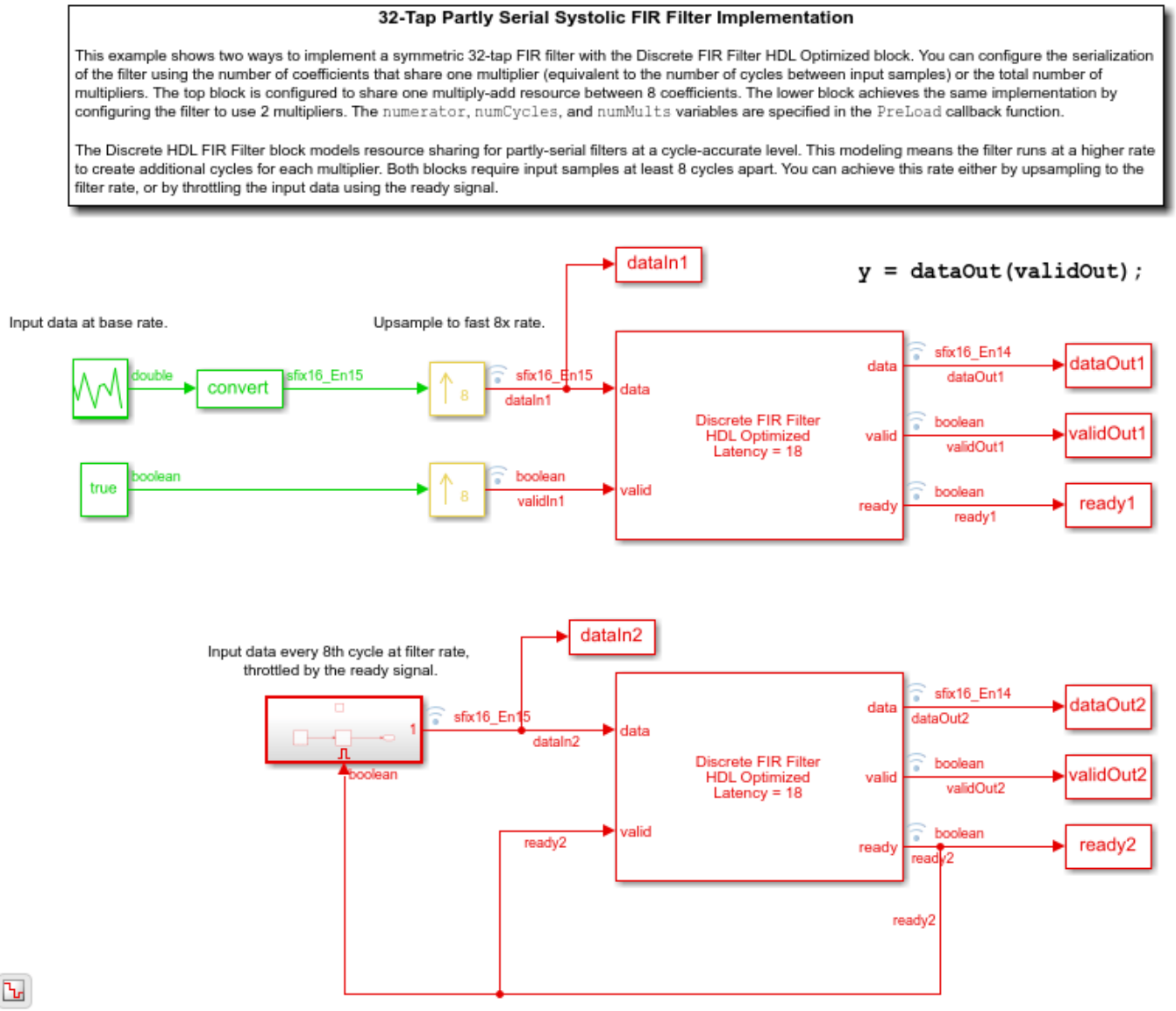
Both filter blocks in the example model implement an identical partly- serial 32-tap filter. The top block configures the serial filter by specifying the number of cycles, *N* , between input samples. This spacing allows each multiplier to be shared by *N* coefficients. The second block is configured 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

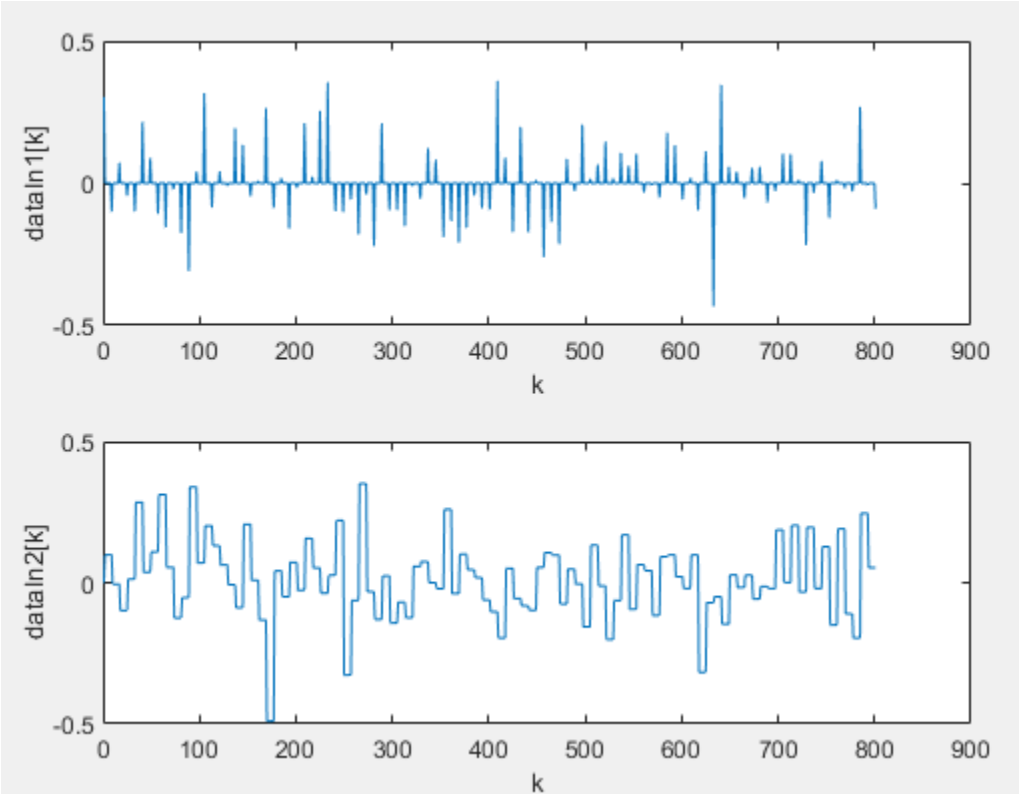
Open the model. Inspect the top block parameters. The **Filter structure** is set to `Partly serial systolic` and **Specify serialization factor as** is set to `Minimum number of cycles between valid input samples`. The **Number of cycles** is set (to 8) using a variable, *numCycles*. In the lower block, **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.

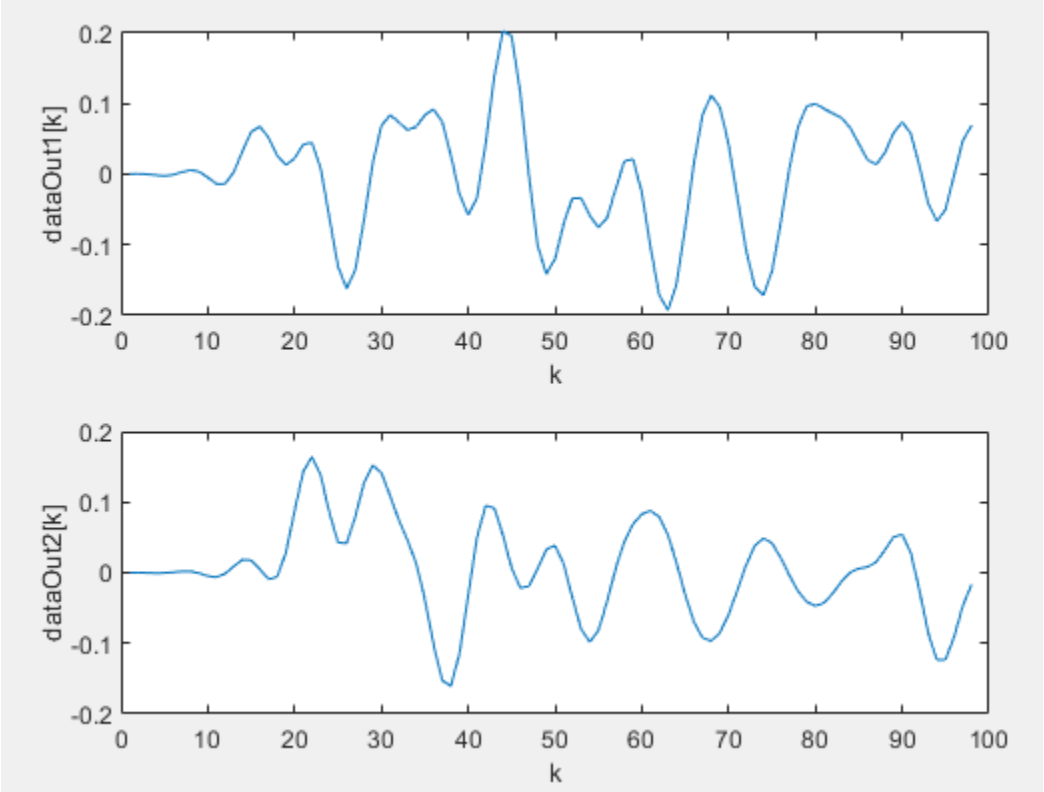
From the color coding, you can see the rate of both filter blocks is the same, while the rate of the generated input samples is different.



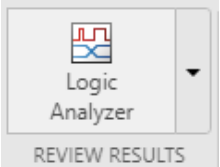
Run the Model and Inspect Results

Run the model. Observe the input and output signals in the generated plots. The code to generate the plots is in the `PostSimFcn` callback.

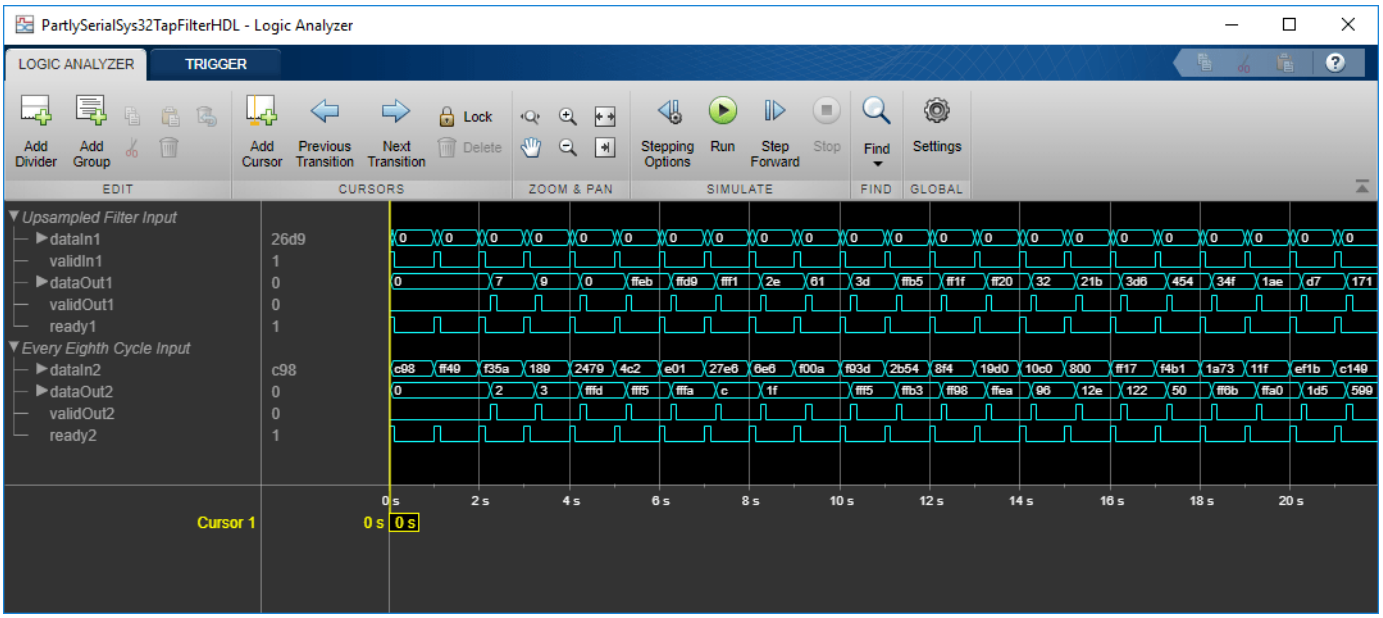

dataIn1[k]
0.5
0
-0.5
0
100
200
300
400
500
600
700
800
900
k
dataIn2[k]
0.5
0
-0.5
0
100
200
300
400
500
600
700
800
900
k


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

Optimize Programmable FIR Filter Resources

This example shows how to use programmable coefficients with the Discrete FIR Filter HDL Optimized block and how to optimize hardware resources for programmable filters.

The Discrete FIR Filter HDL Optimized block optimizes resource use when the filter coefficients are symmetric or antisymmetric or are zero-valued. To use these optimizations with programmable coefficients, all of the input coefficient vectors must have the same symmetry and zero-valued coefficient locations. Set the **Coefficients prototype** parameter to a representative coefficient vector. The block uses the prototype to optimize the filter by sharing multipliers for symmetric or antisymmetric coefficients, and removing multipliers for zero-valued coefficients.

If your coefficients are unknown or not expected to share symmetry or zero-valued locations, set **Coefficients prototype** to `[]`. In this case, the block does not optimize multipliers.

This example shows how to set a prototype and specify programmable coefficients for a symmetric filter and a filter with zero-valued coefficients. The example also explains how the block reduces the number of multipliers in the filter in these cases.

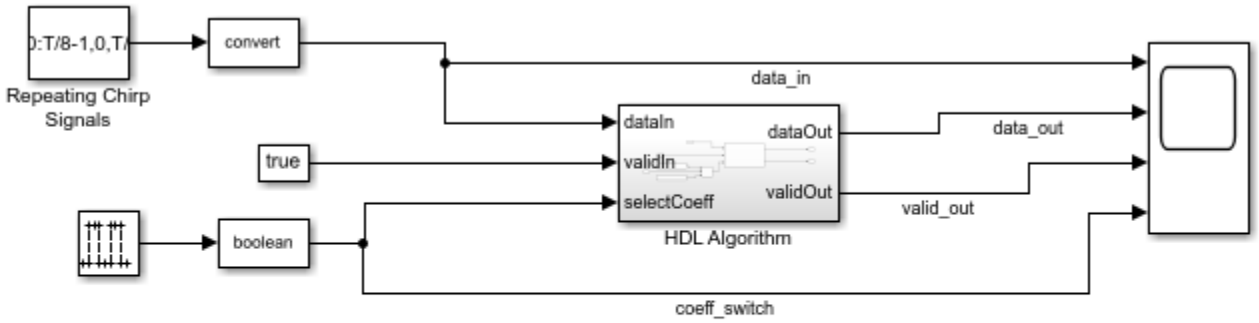
Symmetric Filter Coefficients

Design two FIR filters, one with a lowpass response and the other with the complementary highpass response. Both filters are odd-symmetric and have 43 taps.

```
Fpass = 0.45; % Passband frequency
Fstop = 0.55; % Stopband frequency
Apass = 1;    % Passband attenuation (dB)
Astop = 60;   % Stopband attenuation (dB)

f = fdesign.lowpass('Fp,Fst,Ap,Ast',Fpass,Fstop,Apass,Astop);
Hlp = design(f,'equiripple','FilterStructure','dffir'); % Lowpass
Hhp = firlp2hp(Hlp);                                    % Highpass

hpNumerator = Hlp.Numerator;  %#ok<NASGU>
lpNumerator = Hhp.Numerator;  %#ok<NASGU>
```

The example model shows a filter subsystem with a control signal to switch between two sets of coefficients. The HDL Algorithm subsystem includes a Discrete FIR Filter HDL Optimized block, and the two sets of coefficients defined by the workspace variables created above.

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

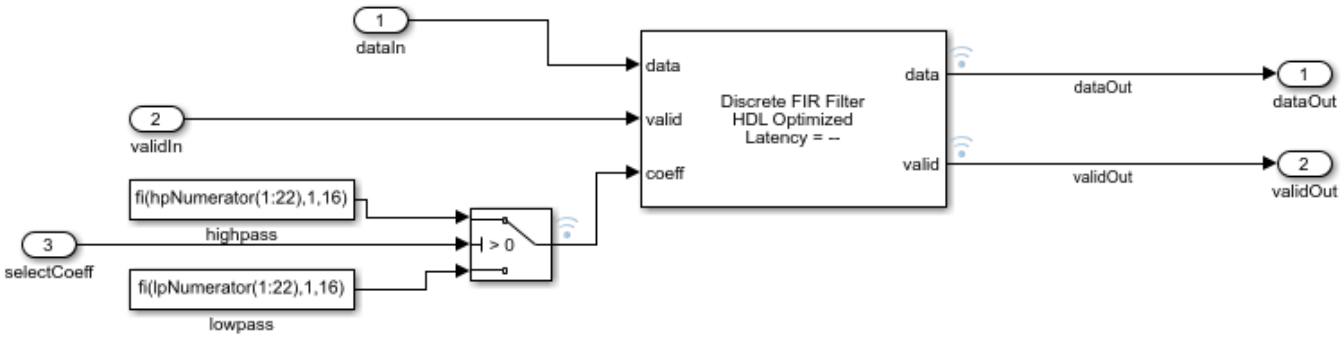


Copyright 2020 The MathWorks, Inc.

Because both sets of coefficients are symmetric in the same way, you can use the **Coefficient prototype** parameter of the Discrete FIR Filter HDL Optimized block to reduce the number of multipliers in the filter implementation. Set **Coefficient prototype** to either of the coefficient vectors. The example model sets the prototype to `hpNumerator`.

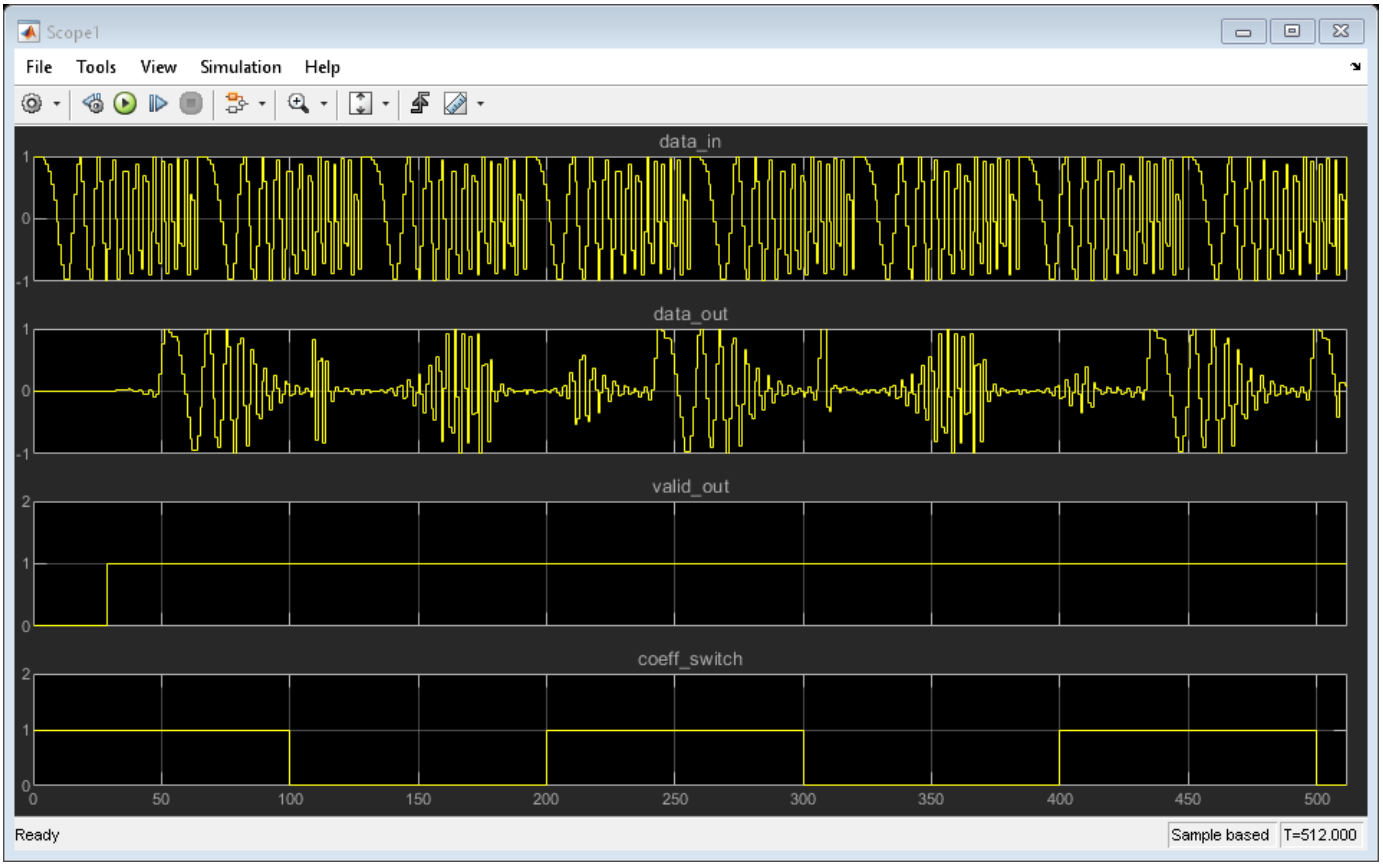
When you use the prototype for symmetric coefficients, you must provide only the unique coefficients to the **coeff** port. In this case, the filter has 43 odd-symmetric coefficients, so the input port expects the first half of the coefficients, that is, 22 values.

```
open_system('ProgFIRHDLOptim/HDL Algorithm');
```



The model switches coefficients every 100 cycles. The filtered output data shows the effect of the low and highpass coefficients.

```
T = 512;
sim(modelname);
```



The model is configured to enable the resource report from HDL code generation. This feature enables you to see the number of multipliers in the filter implementation. Because the block shares multipliers for symmetric coefficients, the filter implementation uses 22 multipliers rather than 43.

Multipliers	22
Adders/Subtractors	44
Registers	275
Total 1-Bit Registers	5117
RAMs	0
Multiplexers	266
I/O Bits	65
Static Shift operators	0
Dynamic Shift operators	0

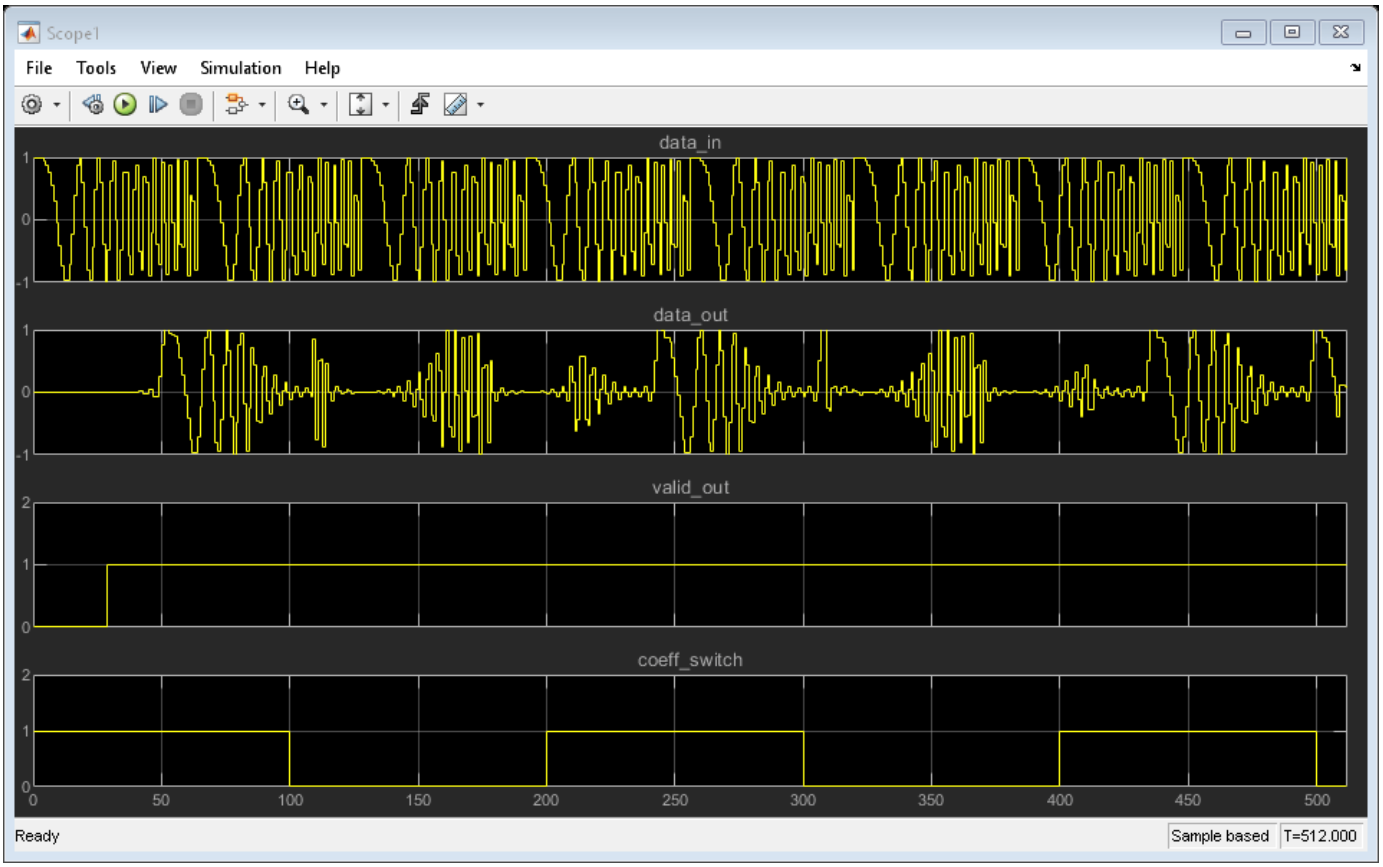
Zero-Valued Filter Coefficients

Design two halfband FIR filters, one with a lowpass response and the other with the complementary highpass response. Both filters have 43 symmetric taps, where every second tap is zero. Set the

Coefficient prototype parameter to either of the coefficient vectors. Changing the workspace value of `hpNumerator` updates the prototype in the block.

Similar to the previous case, specify 22 coefficients at the input port. Although no multipliers exist for the zero-valued coefficients, for the block to maintain the correct alignment of the coefficients, you must specify the zero-valued coefficients at the port.

```
N = 42;
f = fdesign.halfband('N,Ast',N,Astop);
Hlp = design(f,'equiripple','FilterStructure','dffir'); % Lowpass
Hhp = firlp2hp(Hlp);                                    % Highpass

hpNumerator = Hlp.Numerator;
lpNumerator = Hhp.Numerator;

sim(modelname);
```



The model is configured to enable the resource report from HDL code generation. This feature enables you to see the number of multipliers in the filter implementation. In this case, because the filter optimizes for symmetry and zero-valued coefficients, the implementation uses 12 multipliers rather than 43.

Multipliers	12
Adders/Subtractors	24
Registers	265
Total 1-Bit Registers	4117
RAMs	0
Multiplexers	186
I/O Bits	65
Static Shift operators	0
Dynamic Shift operators	0

See Also

Blocks
Discrete FIR Filter HDL Optimized

FIR Decimation for FPGA

This example shows how to decimate streaming samples using a hardware-friendly polyphase FIR filter. It also shows the differences between the `dsp.FIRDecimator` object and the hardware-friendly streaming interface of the FIR Decimation HDL Optimized block.

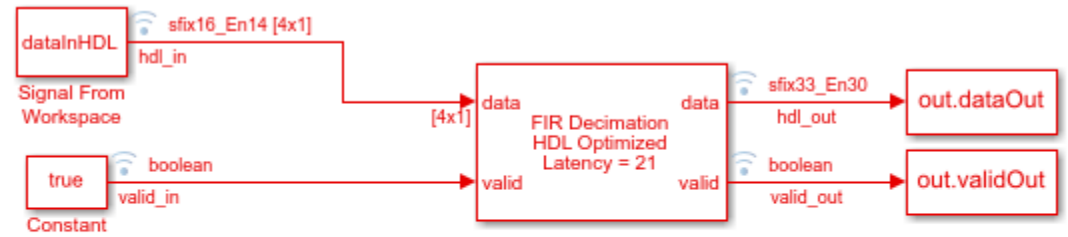
Generate an input sine wave in MATLAB®. The example model imports this signal from the MATLAB workspace. Choose a decimation factor and the input vector size for the HDL-optimized block.

```
T = 512;
waveGen = dsp.SineWave(0.9,100,0,'SamplesPerFrame',T);
dataIn = fi(waveGen()+0.05,1,16,14);

inputVecSize=4;
decimFactor=8;
```

The example model imports the input signal from the MATLAB workspace and applies it as vectors to the FIR Decimation HDL Optimized block. The model returns the output from the block to the MATLAB workspace for comparison.

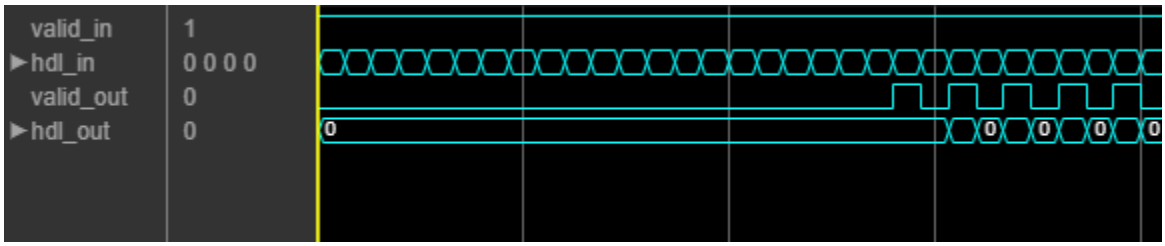
The FIR Decimation HDL Optimized block applies samples to the filter in a different phase than the `dsp.FIRDecimator object`. To match the numerical behavior, apply `decimFactor` - 1 zeros to the FIR Decimation HDL Optimized block before the start of the data samples.

```
dataInHDL = [zeros(decimFactor-1,1);dataIn];

modelName = 'FIRDecimHDL';
open_system(modelName);
set_param(modelName,'SimulationCommand','Update');
```



Copyright 2020 The MathWorks, Inc.

The FIR Decimation HDL Optimized block accepts a vector of `inputVecSize`-by-1 samples and returns scalar output. The block performs single-rate processing and indicates valid samples in the output signal by setting the output **valid** signal to 1. The Simulink sample time is `inputVecSize` for the whole model. The block sets the output **valid** signal to 1 every `decimFactor/inputVecSize` samples.




```
out = sim('FIRDecimHDL');
```

Create a `dsp.FIRDecimator` System object™ and generate reference data to compare against the HDL model output. This example uses the default filter coefficients, which are the same for the block and the object. The object can accept any input vector size that is a multiple of the decimation factor, so you can calculate the output with one call to the object. If you change the decimation factor to a value that is not a factor of `T`, you must also adjust the value of `T` (size of `dataIn`).

The block and the object are both configured to use full-precision internal data types. The object computes a different output data type in this mode. This difference means that the output might not match if the internal values saturate the data types.

```
decimRef = dsp.FIRDecimator(decimFactor);
refDataOut = decimRef(dataIn);
```

For the output of the FIR Decimation HDL Optimized block, select data samples where the output valid signal was `1`, and compare them with the samples returned from the `dsp.FIRDecimator` object.

```
hdlVec = out.dataOut(out.validOut);
refVec = refDataOut(1:size(hdlVec,1));
errVec = hdlVec - refVec;
maxErr = max(abs(errVec));
fprintf('\nFIR decimator versus reference: Maximum error out of %d values is %d\n',length(hdlVec
```

```
FIR decimator versus reference: Maximum error out of 54 values is 1
```

See Also

Blocks
FIR Decimation HDL Optimized

High Throughput Channelizer for FPGA

This example shows how to implement a high throughput (Gigasamples per second, GSPS) channelizer for hardware by using a polyphase filter bank.

High speed signal processing is a requirement for applications such as radar, broadband wireless and backhaul.

Modern ADCs are capable of sampling signals at sample rates up to several Gigasamples per second. But the clock speeds for the fastest FPGA fall short of this sample rate. FPGAs typically run at hundreds of MHz. An approach to perform GSPS processing on an FPGA is to move from scalar processing to vector processing and process multiple samples at the same time at a much lower clock rate. Many modern FPGAs support the JESD204B standard interface that accepts scalar input at GHz clock rate and produces a vector of samples at a lower clock rate.

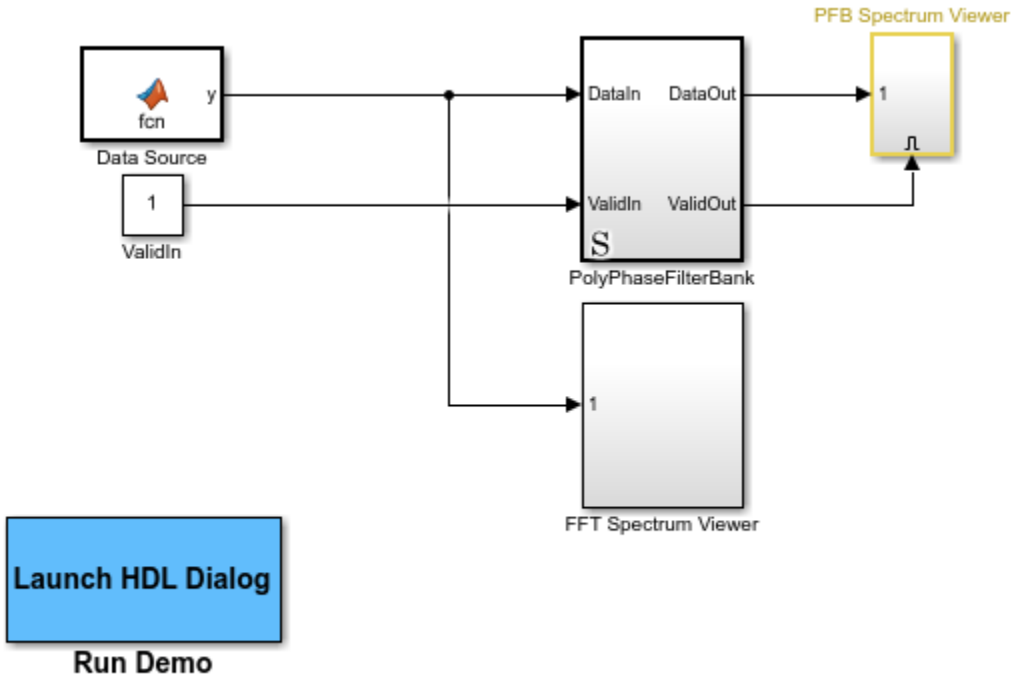
In this example we show how to design a signal processing application for GSPS throughput in Simulink. Input data is vectorized through a JESD204B interface and available at a lower clock rate in the FPGA. The model is a polyphase filter bank which consists of a filter and an FFT that processes 16 samples at a time. The polyphase filter bank technique is used to minimize the FFT's inaccuracy due to leakage and scalloping loss. See “High Resolution Spectral Analysis in MATLAB” on page 4-16 for more information about the polyphase filter bank.

The first part of the example implements a polyphase filter bank with a 4-tap filter.

The second part of the example uses the Channelizer HDL Optimized block configured for a 12-tap filter. The Channelizer HDL Optimized block uses the polyphase filter bank technique.

Polyphase Filter Bank

```
modelname = 'PolyphaseFilterBankHDLExample_4tap';
open_system(modelname);
```



Copyright 2016 The MathWorks, Inc.

The `InitFcn` callback (Model Properties > Callbacks > InitFcn) sets up the model. This model uses a 512-point FFT with a four tap filter for each band. Use the `dsp.Channelizer` System object™ to generate the coefficients. The polyphase method of the Channelizer object generates a 512-by-4 matrix. Each row represents the coefficients for each band. The coefficients are cast into fixed-point with the same word length as the input signal.

```
FFTLength  = 512;
h = dsp.Channelizer;
h.NumTapsPerBand = 4;
h.NumFrequencyBands = FFTLength;
h.StopbandAttenuation = 60;
coef =fi(polyphase(h),1,15,14,'RoundingMethod','Convergent');
```

The algorithm requires 512 filters (one filter for each band). For a vector input of 16 samples we can reuse 16 filters, 32 times.

```
InVect      = 16;
ReuseFactor = FFTLength/InVect;
```

To synthesize the filter to a higher clock rate, we pipeline the multiplier and the coefficient bank. These values are explained in the "Optimized Hardware Considerations" section.

```
Multiplication_PipeLine = 2;
CoefBank_PipeLine       = 1;
```

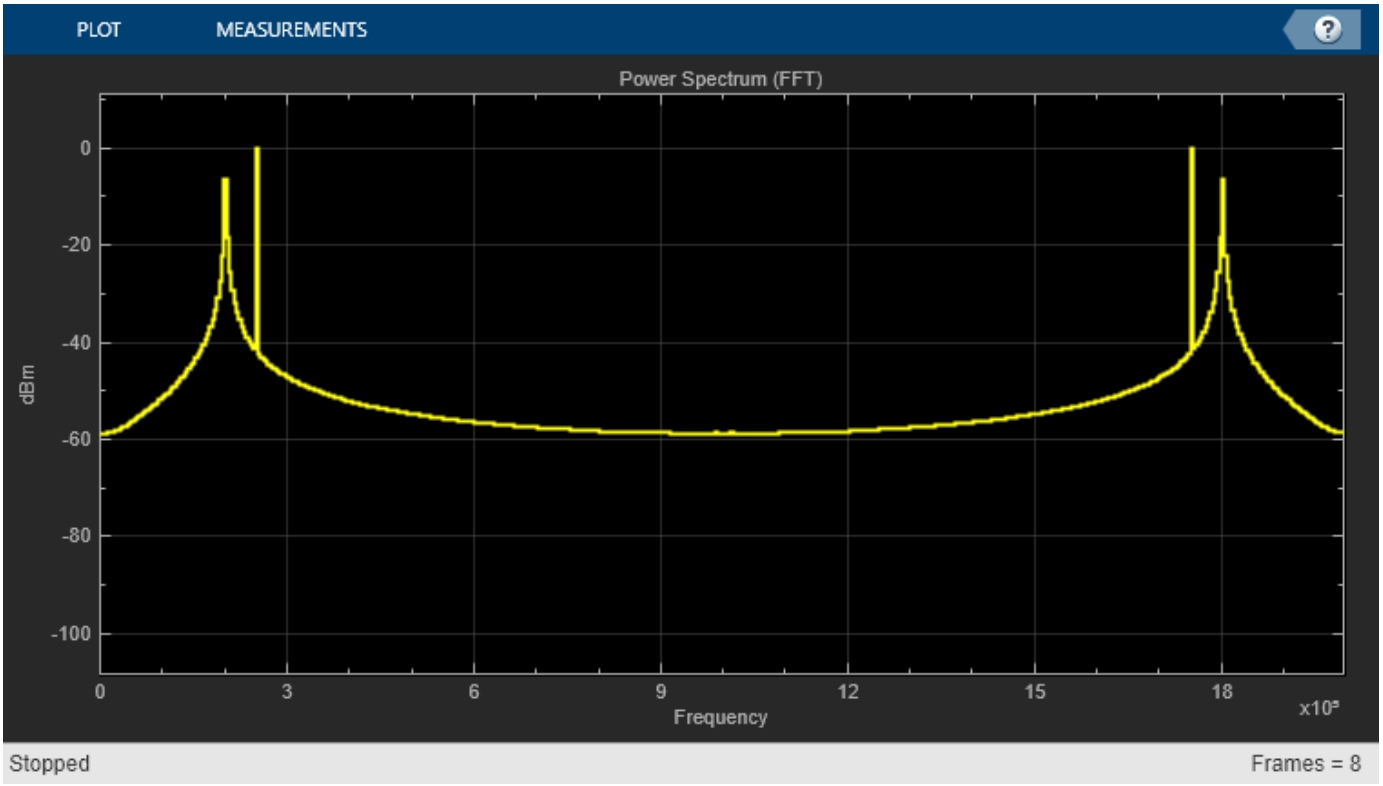
Data Source

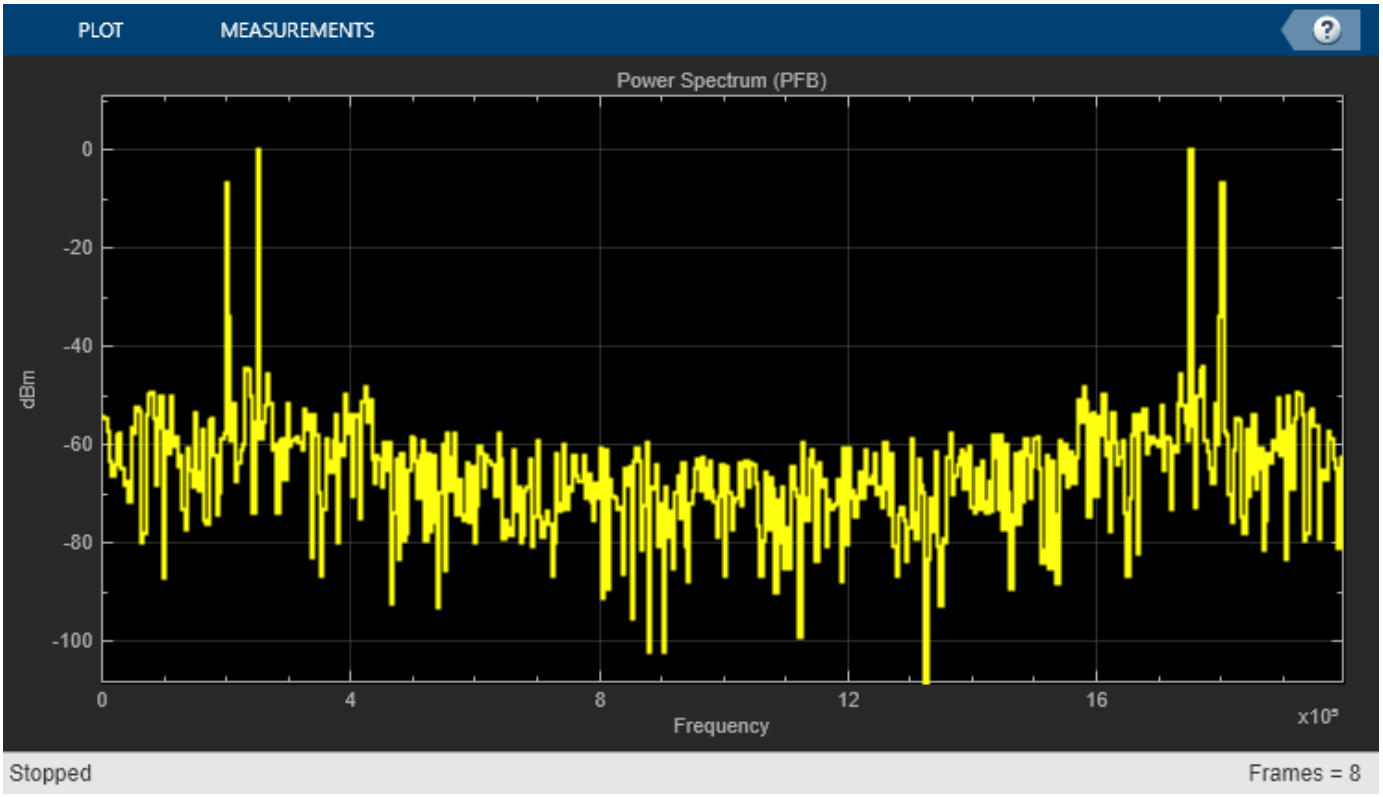
The input data consists of two sine waves, 200 KHz and 250 KHz.

Simulation Results

To visualize the spectrum result, open the spectrum viewers and run the model.

```
open_system('PolyphaseFilterBankHDLExample_4tap/FFT Spectrum Viewer/Power Spectrum viewer (FFT)')
open_system('PolyphaseFilterBankHDLExample_4tap/PFB Spectrum Viewer/Power Spectrum viewer (PFB)')
sim(modelname);
```





The polyphase filter bank Power Spectrum Viewer shows the improvement in the power spectrum and minimization of frequency leakage and scalloping compared with using only an FFT. By comparing the two spectrums, and zooming in between 100 KHz and 300 KHz, observe that the polyphase filter bank has fewer peaks over -40 dB than the classic FFT.

Optimized Hardware Considerations

- **Data type** : The data word length affects both the accuracy of the result and the resources used in the hardware. For this example we design the filter at full precision. With an input data type of `fixdt(1,15,13)`, the output is `fixdt(1,18,17)`. The absolute values of the filter coefficients are all smaller than 1 so the data doesn't grow after each multiplication, and we need to add one bit for each addition. To keep the accuracy in the FFT, we need to grow one bit for each stage. This makes the twiddle factor multiplication bigger at each stage. For many FPGAs it is desirable to keep multiplication size smaller than 18x18. Since a 512 point FFT has 9 stages, the input of the FFT cannot be more than 11 bits. By exploring the filter coefficients, we observe that the first 8 binary digits of the maximum coefficient are zero, and therefore we can cast the coefficients to `fixdt(1,7,14)` instead of `fixdt(1,15,14)`. Also we observe that the maximum value of the Datatype block output inside the polyphase filter bank has 7 leading zeros after the binary point, and therefore we cast the filter output to `fixdt(1,11,17)` instead. This keeps the multiplier size inside the FFT smaller than 18-by-18 and saves hardware resources.

- **Design for speed**:

1. *State control block*: The State Control (HDL Coder) block is used in Synchronous mode to generate hardware friendly code for the delay block with enable port.
2. *Minimize clock enable* : The model is set to generate HDL code with the Minimize Clock Enable option turned on (In Configuration Parameters choose > HDL Code Generation > Global settings

> Ports > Minimize clock enables). This option is supported when the model is single rate. Clock enable is a global signal which is not recommended for high speed designs.

3. *Usage of the DSP block in FPGA*: In order to map multipliers into a DSP block in the FPGA, the multipliers should be pipelined. In this example we pipeline the multipliers (2 delays before and 2 delays after) by setting Multiplication_PipeLine = 2; These pipeline registers should not have a reset. Set the reset type to none for each pipeline (right-click the Delay block and select HDL Code > HDL Block Properties > Reset Type = None).
4. *Usage of ROM in FPGA*: The Coefficient block inside the Coefficient Bank is a combinatorial block. In order to map this block into a ROM, add a register after the block. The delay length is set by `CoefBank_PipeLine`. Set the reset type for these delays to none (right-click the Delay block and select HDL Code > HDL Block Properties > Reset Type = None).

Generate HDL Code and Test Bench

You must have an HDL Coder™ license to generate HDL code for this example model. Use this command to generate HDL code. systemname = 'PolyphaseFilterBankHDLExample_4tap/PolyPhaseFilterBank'; makehdl(systemname);

Use this command to generate a test bench that compares the results of an HDL simulation against the Simulink simulation behavior. makehdltb(systemname);

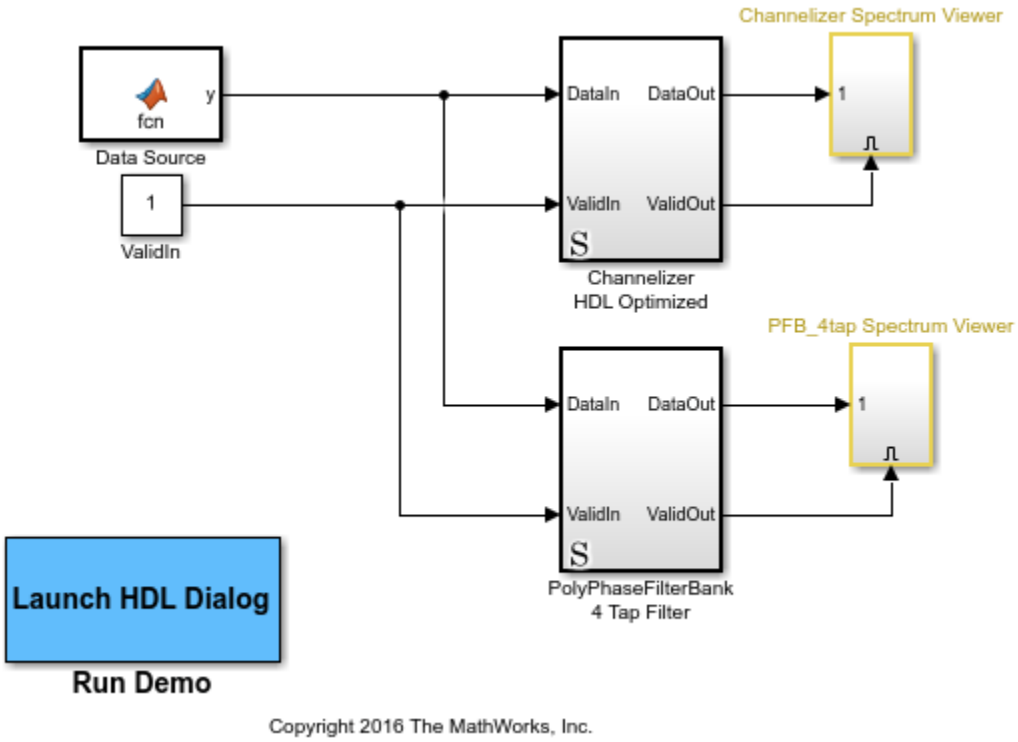
Synthesis Results

The design was synthesized for Xilinx Virtex 7 (xc7vx550t-ffg1158, speed grade 3) using ISE. The design achieves a clock frequency of 499.525 MHz (before place and route). At 16 samples per clock, this translates to 8 GSPS throughput. Note that this subsystem has a high number of I/O ports and it is not suitable as a standalone design targeted to the FPGA.

HDL Optimized Channelizer

To improve the frequency response, use a filter with more taps. The following model uses the Channelizer HDL Optimized block, configured with a 12-tap filter to improve the spectrum. Using a built-in Channelizer HDL Optimized block makes it easier to change design parameters.

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



The model uses workspace variables to configure the FFT and filter. In this case, the model uses a 512-point FFT and a 12-tap filter for each band. The number of coefficients for the channelizer is 512 frequency bands times 12 tap per frequency band. The `tf(h)` method generates all the coefficients.

```
InVect     = 16;
FFTLength  = 512;
h = dsp.Channelizer;
h.NumTapsPerBand = 12;
h.NumFrequencyBands = FFTLength;
h.StopbandAttenuation = 60;
coef12Tap = tf(h);
```

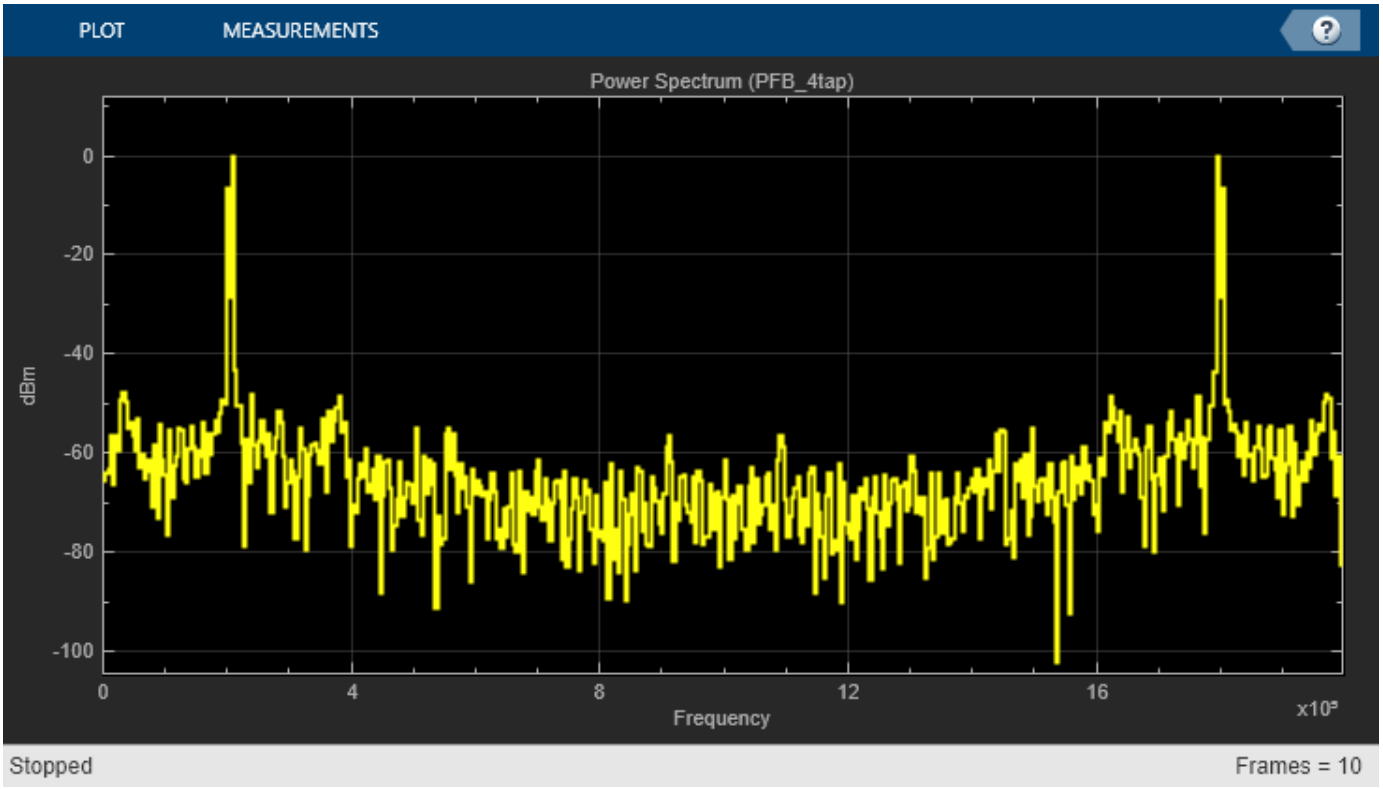
Data Source

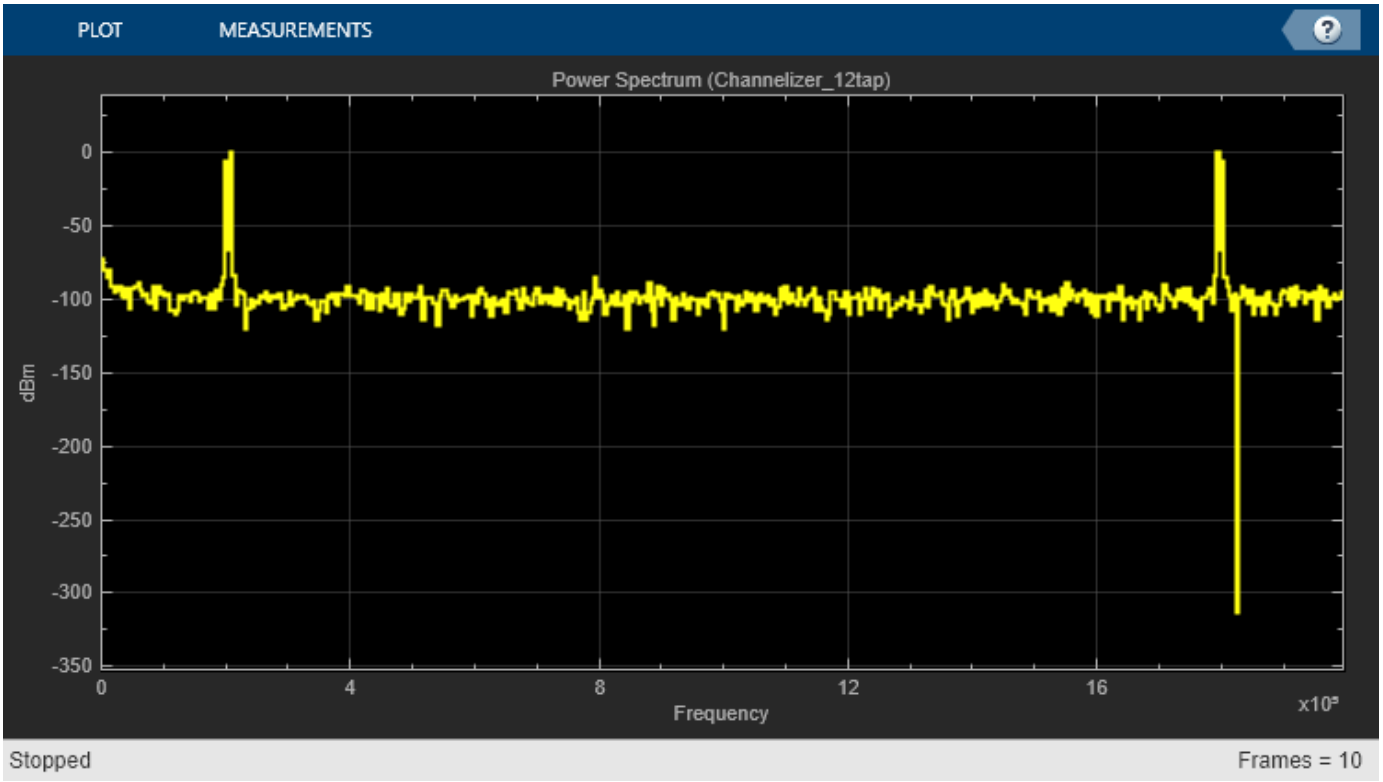
The input data consists of two sine waves, 200 KHz and 206.5 KHz. The frequencies are closer to each other than the first example to illustrate the difference between a channelizer and a 4-tap filter in spectrum resolution.

Simulation Results

To visualize the spectrum result, open the spectrum viewers and run the model.

```
open_system('PolyphaseFilterBankHDLExample_HDLChannelizer/PFB_4tap Spectrum Viewer/Power Spectrum
open_system('PolyphaseFilterBankHDLExample_HDLChannelizer/Channelizer Spectrum Viewer/Power Spec
sim(modelname);
```


PLOT
MEASUREMENTS
Power Spectrum (PFB_4tap)
0
-20
-40
-60
-80
-100
dBm
0
4
8
12
16
Frequency
x10⁸
Stopped
Frames = 10


PLOT
MEASUREMENTS
Power Spectrum (Channelizer_12tap)
0
-50
-100
-150
-200
-250
-300
-350
dBm
0
4
8
12
16
Frequency
x10⁸
Stopped
Frames = 10

The Power Spectrum Viewer for the Channelizer_12tap model shows the improvement in the power spectrum of the polyphase filter bank with 12-tap filter compared to the 4-tap filter in the previous model. Compare the spectrum results for the channelizer and 4-tap polyphase filter banks. Zoom in between 100 KHz and 300 KHz to observe that the channelizer detects only two peaks while the 4-tap polyphase filter bank detects more than 2 peaks. Two peaks is the expected result since the input signal has only two frequency components.

Implement atan2 Function for HDL

This example shows how to use the Complex to Magnitude-Angle HDL Optimized block to implement the atan2 function in hardware.

This example model compares the output of Complex to Magnitude-Angle HDL Optimized block with the `atan2` function implemented using Trigonometric Function block.

HDL Counter 1 and HDL Counter 2 blocks generate the real and imaginary parts, respectively, of the complex number.

Real-Imag to Complex block constructs the complex output from real and imaginary inputs.

Trigonometric Function block with function parameter set as `atan2` is used to generate the reference output angle. This block uses the CORDIC approximation to calculate the angle.

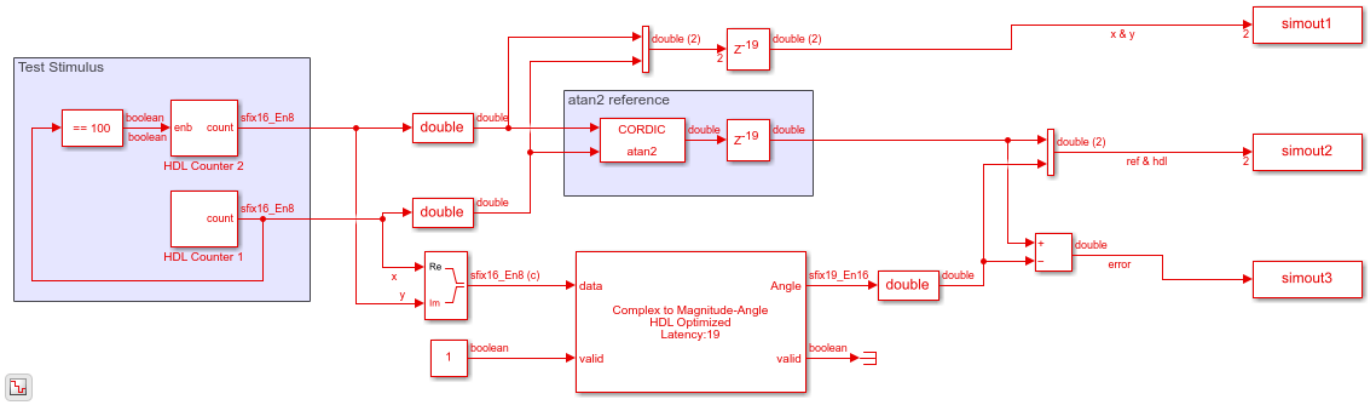
Complex to Magnitude-Angle block configured to return the angle in radians produces the angle of the complex input as an output. This block also models the latency of the hardware implementation.

To align the data for comparison, the reference data path includes a delay block with the same latency.

Complex to Magnitude-Angle block supports HDL code generation, if you add it to a subsystem.

Run the Simulink model

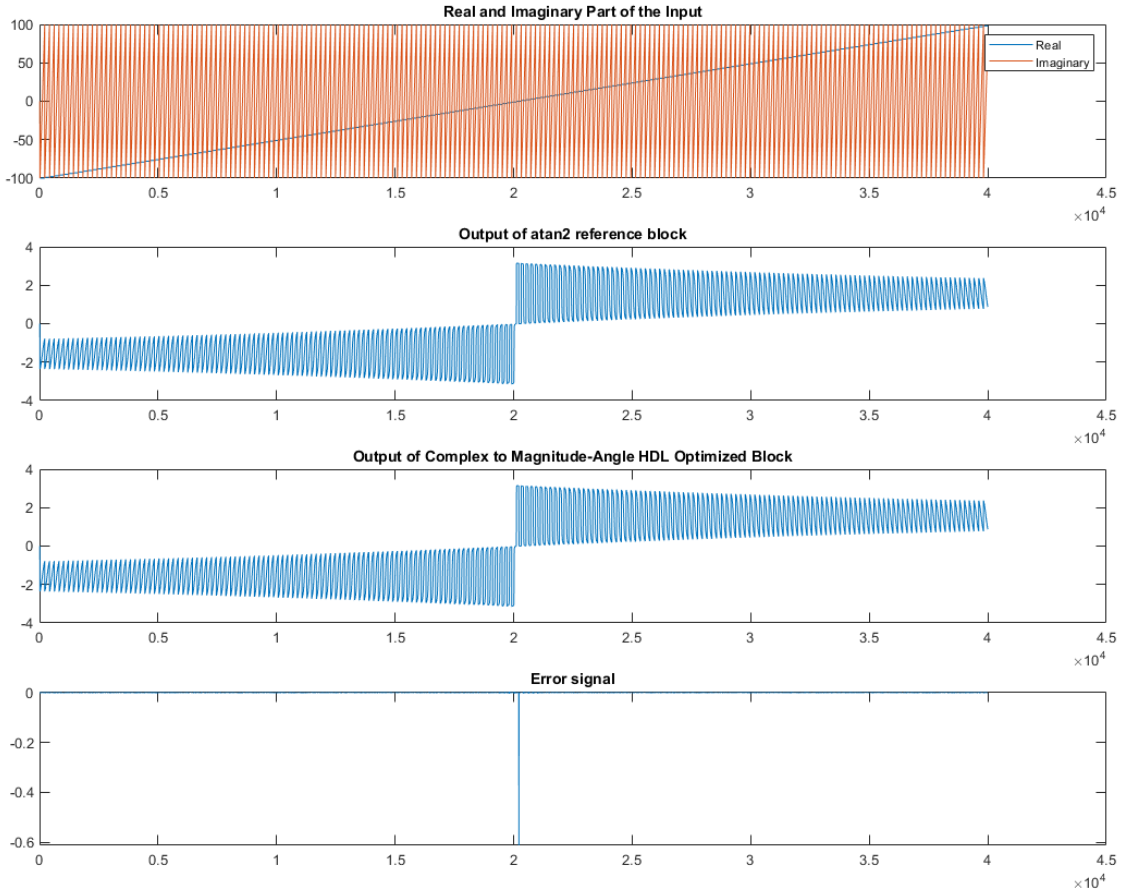
```
modelname = 'HDLatan2Example';
open_system(modelname);
set_param(modelname,'SampleTimeColors','on');
set_param(modelname,'SimulationCommand','Update');
set_param(modelname,'Open','on');
set(allchild(0),'Visible','off');
out = sim(modelname);
```



Compare the outputs of Complex to Magnitude-Angle HDL block against atan2 function block

```
figure('units','normalized','outerposition',[0 0 1 1])
subplot(4,1,1)
plot(simout1(:,1))
hold on;
```

```
plot(simout1(:,2))
hold off;
legend('Real','Imaginary')
title('Real and Imaginary Part of the Input')

subplot(4,1,2)
plot(simout2(:,1))
title('Output of atan2 reference block')

subplot(4,1,3)
plot(simout2(:,2))
title('Output of Complex to Magnitude-Angle HDL Optimized Block')

subplot(4,1,4)
plot(simout3)
title('Error signal')
```



See Also

Blocks
Complex to Magnitude-Angle HDL Optimized

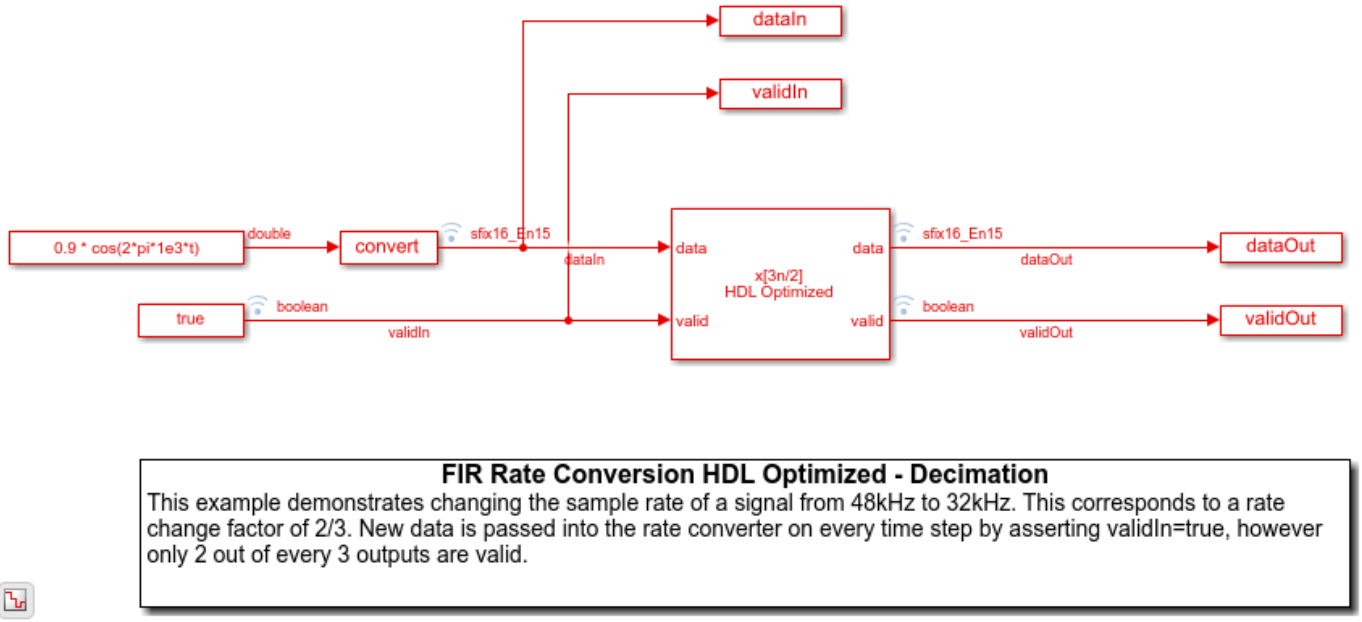
Functions
`atan2`

Downsample a Signal

Convert a signal from 48 kHz to 32 kHz using the FIR Rate Conversion HDL Optimized block.

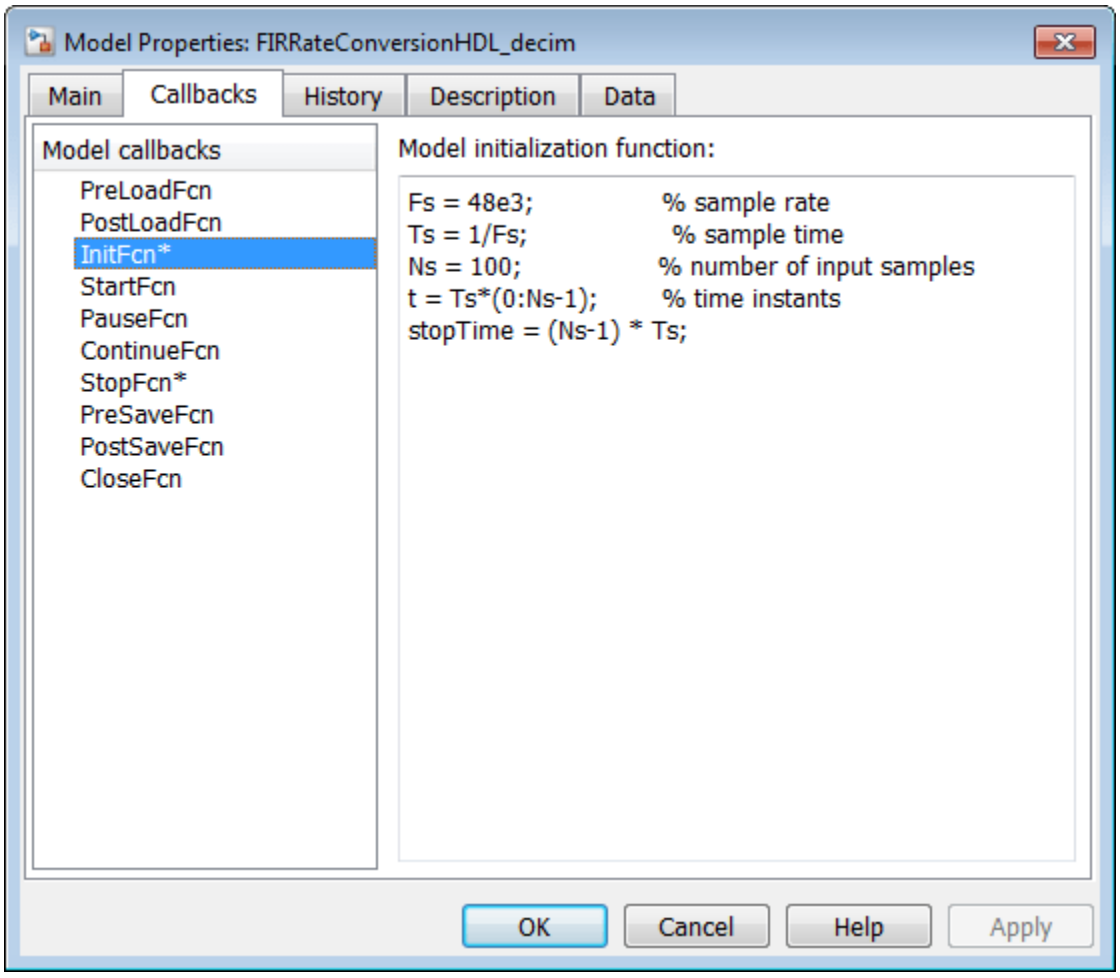
The source is a cosine input signal, sampled at 48kHz. The model passes a new data sample into the block on every time step by holding the input `valid` port `true`. After resampling, the output `valid` signal is `true` on only 2/3 of the time steps.

Open the Model



Configure the Model

Define the data rate parameters in the `InitFcn` callback.

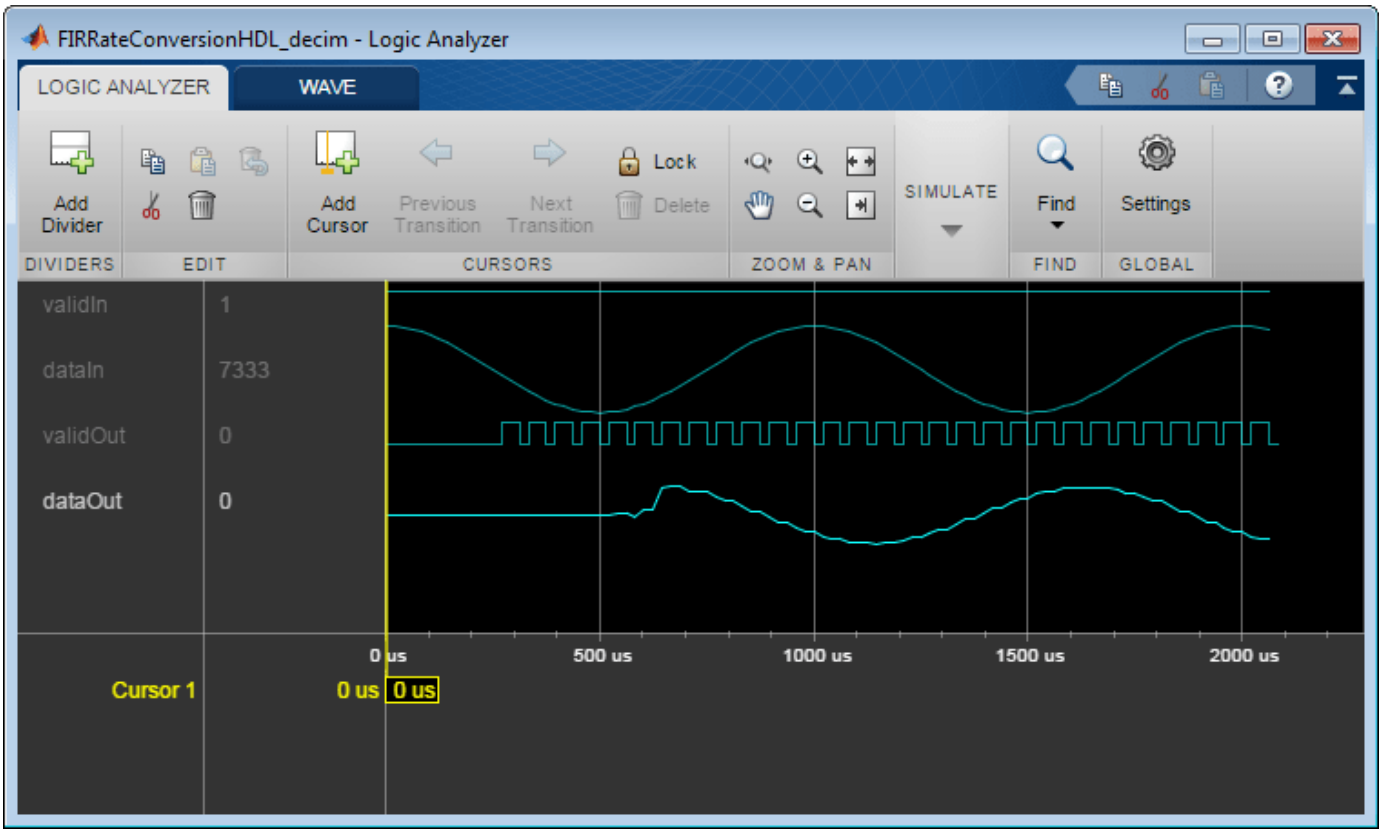


Configure the FIR Rate Conversion HDL Optimized block. Use the default interpolation factor of 2 and decimation factor of 3. Use the `firmpm` function to design an equiripple FIR filter. In the **Data Types** group, set the **Coefficients** data type to `fixdt(1,16,15)` to accommodate the filter you designed.

Run the Model and Display Results

Run the model. Use the **Logic Analyzer** to view the input and output signals of the block. The blue icon in the model indicates streamed signals. Launch the Logic Analyzer from the model's toolstrip.

In the Logic Analyzer, note the pattern of `validIn` and the resulting `validOut` signal.



Generate HDL Code

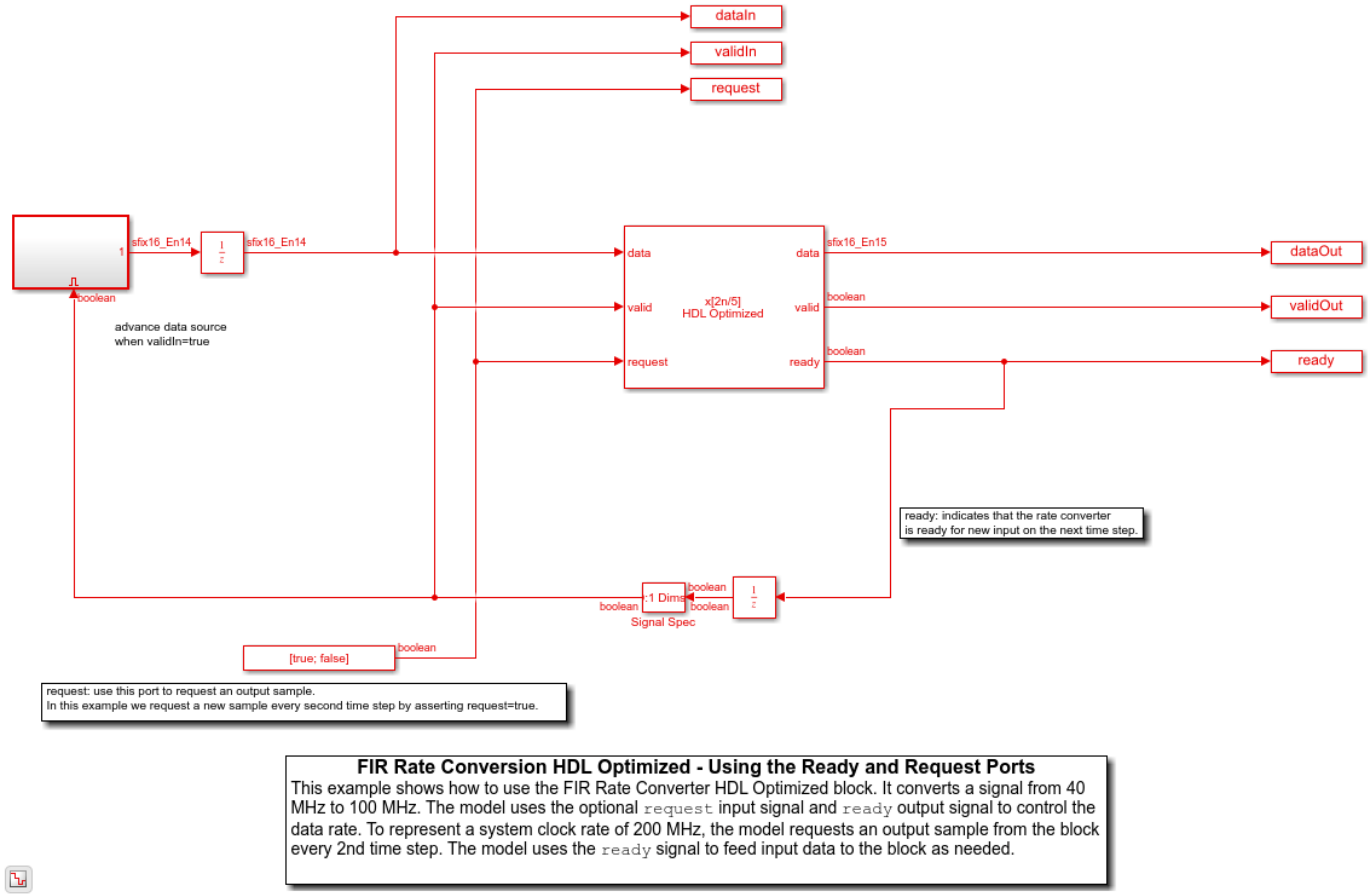
To generate HDL code from the FIR Rate Converter HDL Optimized block, right-click the block and select **Create Subsystem from Selection**. Then right-click the subsystem and select **HDL Code > Generate HDL Code for Subsystem**.

Control Data Rate Using the Ready and Request Ports

Convert a signal from 40 MHz to 100 MHz using the FIR Rate Converter HDL Optimized block. Uses the optional `request` input signal and `ready` output signal to control the data rate.

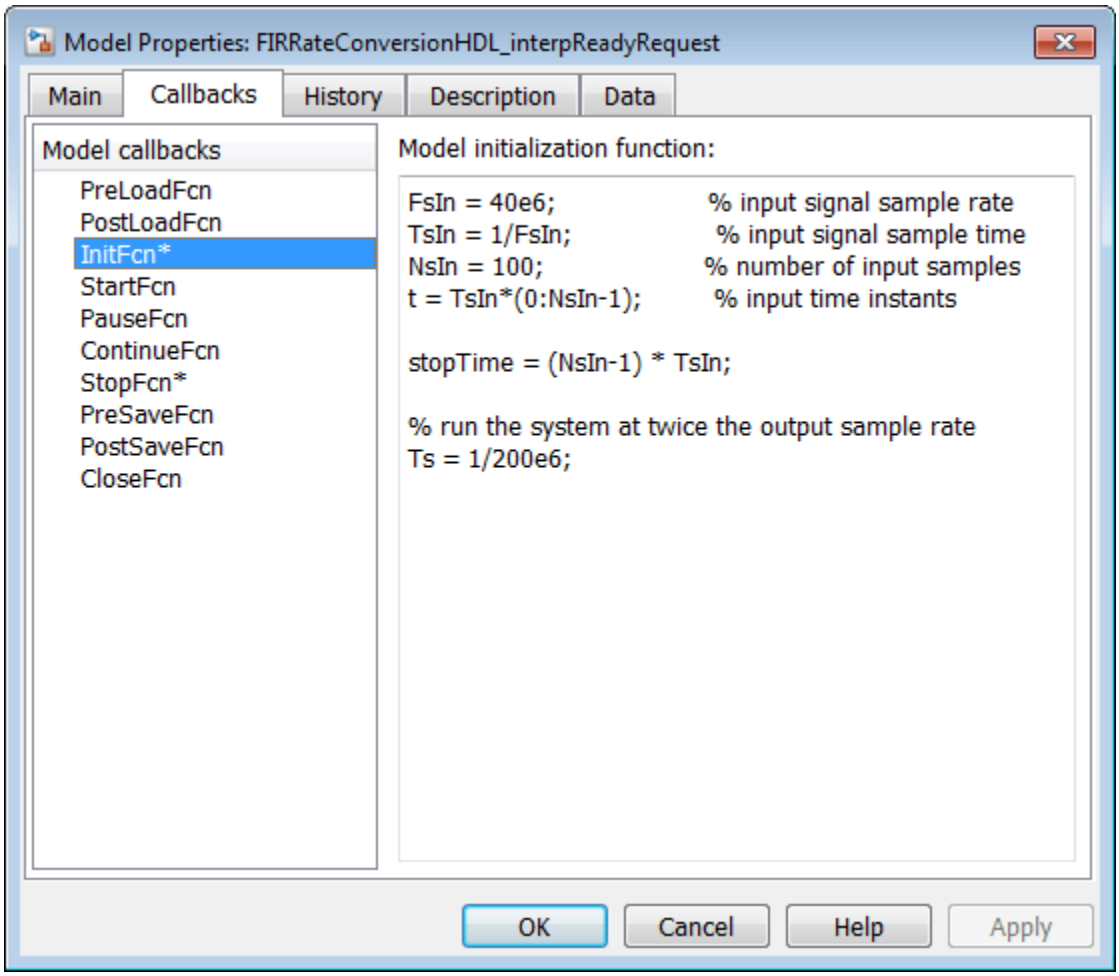
- To represent a system clock rate of 200MHz, the model connects a repeating true-false signal to the `request` port. This configuration generates output samples at 100 MHz, i.e. every second time step. Alternatively, you can connect this port to the `ready` port of a downstream block.
- When the block can accept a new input sample on the next time step, it sets the `ready` output signal to `true`. The model connects this signal to a waveform source that generates one sample at a time.

Open the Model



Configure the Model

Define the data rate parameters in the `InitFcn` callback.

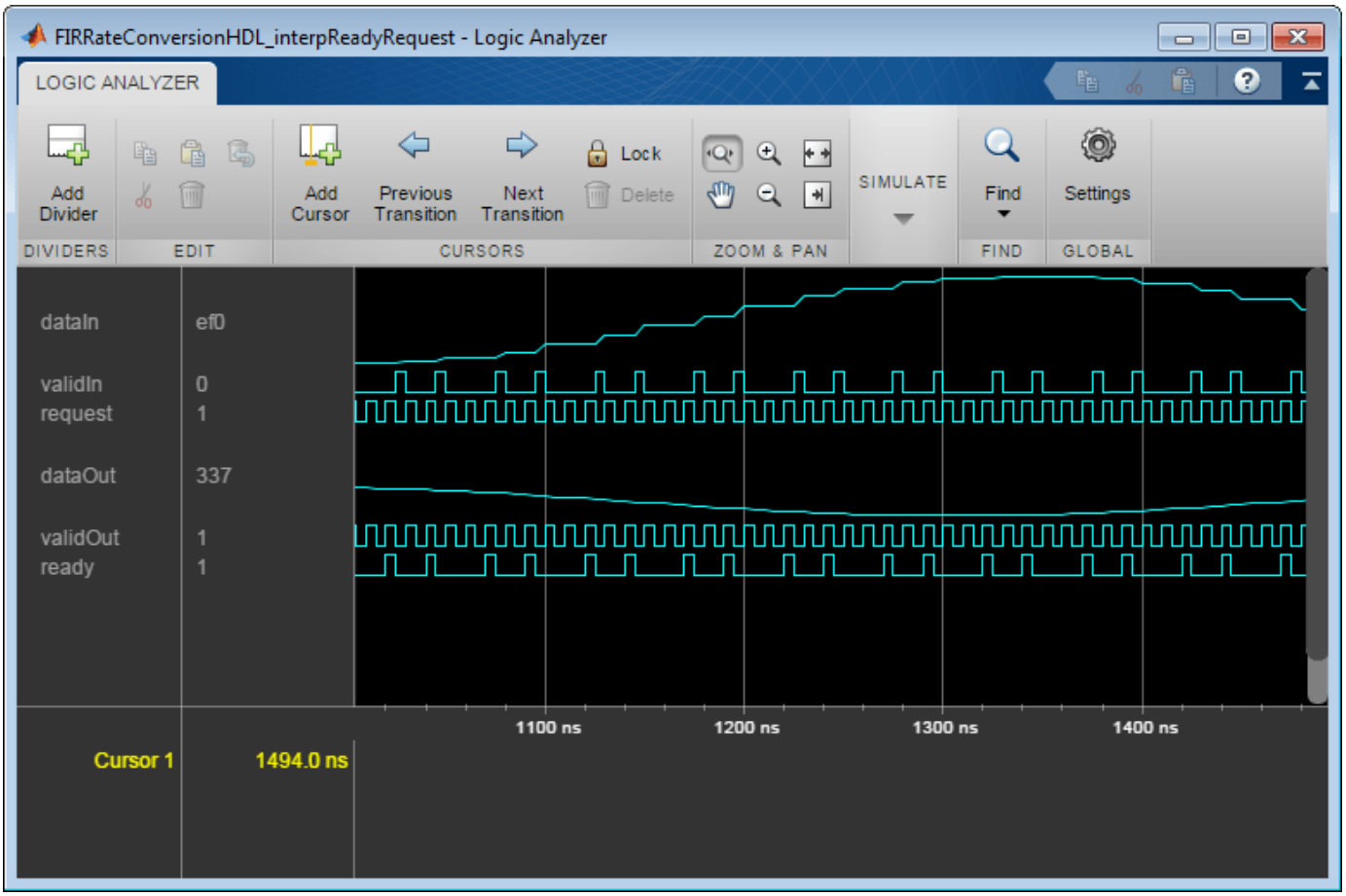


Configure the FIR Rate Conversion HDL Optimized block. Use an interpolation factor of 5 and a decimation factor of 2. Use the `firmpm` function to design an equiripple FIR filter. Select both check boxes to enable the `ready` and `request` ports. % In the **Data Types** group, set the **Coefficients** data type to `fixdt(1,16,15)` to accommodate your filter design.

Run the Model and Display Results

Run the model. Use the **Logic Analyzer** to view the input and output signals of the block. The blue icon in the model indicates streamed signals. Launch the Logic Analyzer from the model's toolstrip.

In the **Logic Analyzer**, note the pattern of `request` and the resulting `validOut` signal, and the pattern of `ready` and the resulting `validIn` signal.



Generate HDL Code

To generate HDL code from the FIR Rate Converter 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**.

Implement FFT for FPGA Using FFT HDL Optimized Block

This example shows how to use the FFT HDL Optimized block to implement a FFT for hardware.

The FFT and IFFT HDL Optimized blocks and system objects support simulation and HDL code generation for many applications. They provide two architectures optimized for different use cases:

- `Streaming Radix 2^2` - For high throughput applications. Achieves gigasamples per second (GSPS) when you use vector input.
- `Burst Radix 2` - For low area applications. Uses only one complex butterfly.

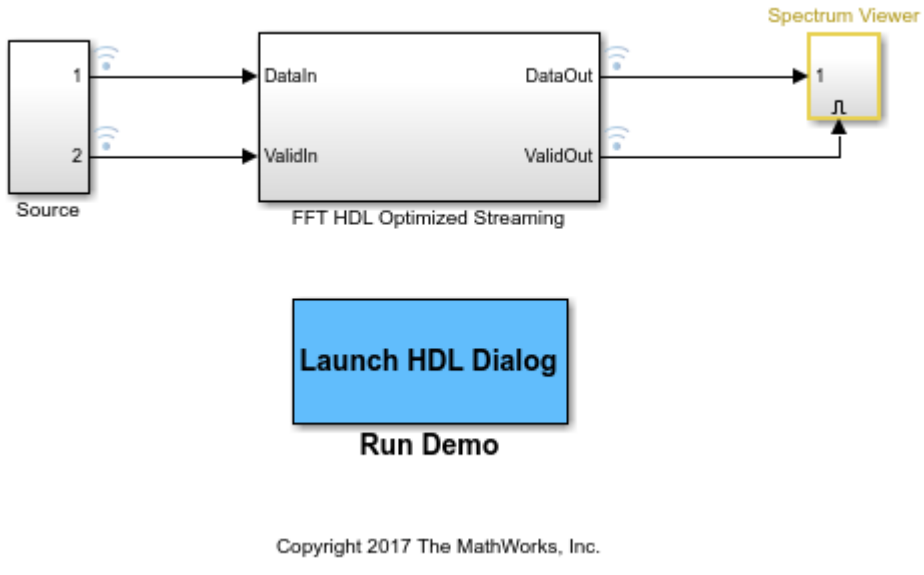
This example includes two models that show how to use the streaming and burst architectures of the FFT HDL Optimized block, respectively.

Streaming Radix 2^2 Architecture

Modern ADCs are capable of sampling signals at sample rates up to several gigasamples per second. However, clock speeds for the fastest FPGA fall short of this sample rate. FPGAs typically run at hundreds of MHz. One way to perform GSPS processing on an FPGA is to process multiple samples at the same time at a much lower clock rate. Many modern FPGAs support the JESD204B standard interface that accepts scalar input at GHz clock rate and produces a vector of samples at a lower clock rate. Therefore modern signal processing requires vector processing.

The `Streaming Radix 2^2` architecture is designed to support the high throughput use case. This example model uses an input vector size of 8, and calculates the FFT using the `Streaming Radix 2^2` architecture. For timing diagram, supported features, and FPGA resource usage, see the FFT HDL Optimized block reference page.

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



The InitFcn callback (Model Properties > Callbacks > InitFcn) sets parameters for the model. In this example, the parameters control the size of the FFT and the input data characteristics.

```
FFTLength  = 512;
```

The input data is two sine waves, 200 KHz and 250 KHz, each sampled at 1*2e6 Hz. The input vector size is 8 samples.

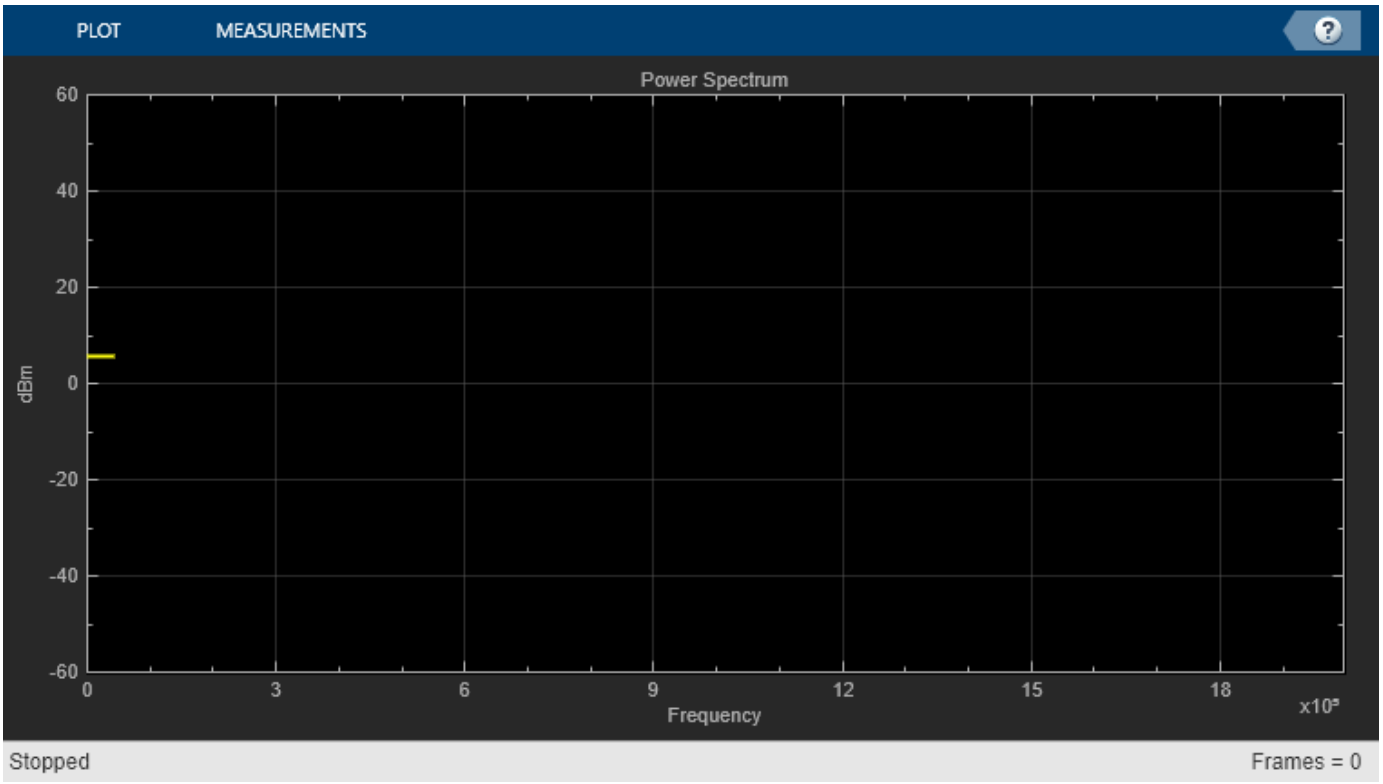
```
FrameSize    = 8;
Fs           = 1*2e6;
```

To demonstrate that data does not need to come continuously, this example applies valid input every other cycle.

```
ValidPattern = [1,0];
```

Open the Spectrum Viewer and run the example model.

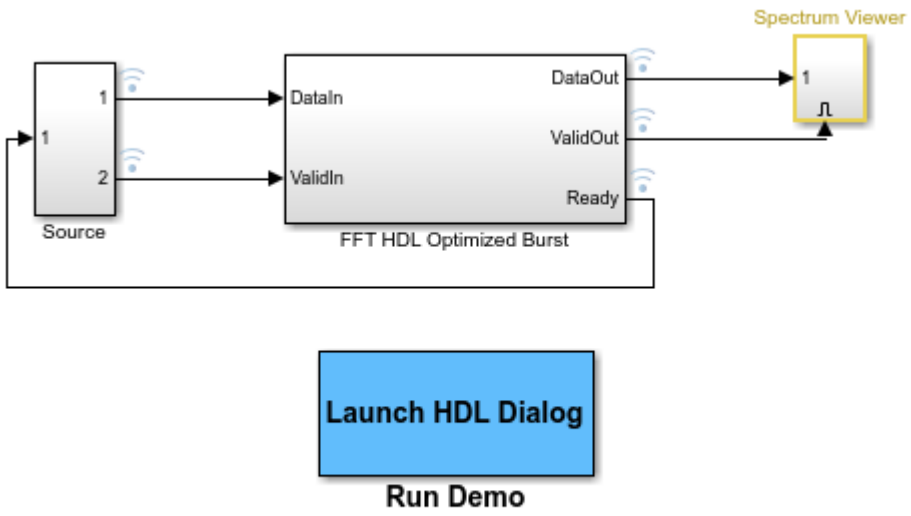
```
open_system('FFTHDLOptimizedExample_Streaming/Spectrum Viewer/Power Spectrum viewer');
sim(modelname);
```



Burst Radix 2 (Minimum Resource) Architecture

Use the `Burst Radix 2` architecture for applications with limited FPGA resources, especially when the FFT length is big. This architecture uses only one complex butterfly to calculate the FFT. The design accepts data while `ready` is asserted, and starts processing once the whole FFT frame is saved into the memory. While processing, the design cannot accept data, so `ready` is de-asserted. For timing diagram, supported features, and FPGA resource usage, see the FFT HDL Optimized block reference page.

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



The InitFcn callback (Model Properties > Callbacks > InitFcn) sets parameters for the model. In this example, the parameters control the size of the FFT and the input data characteristics.

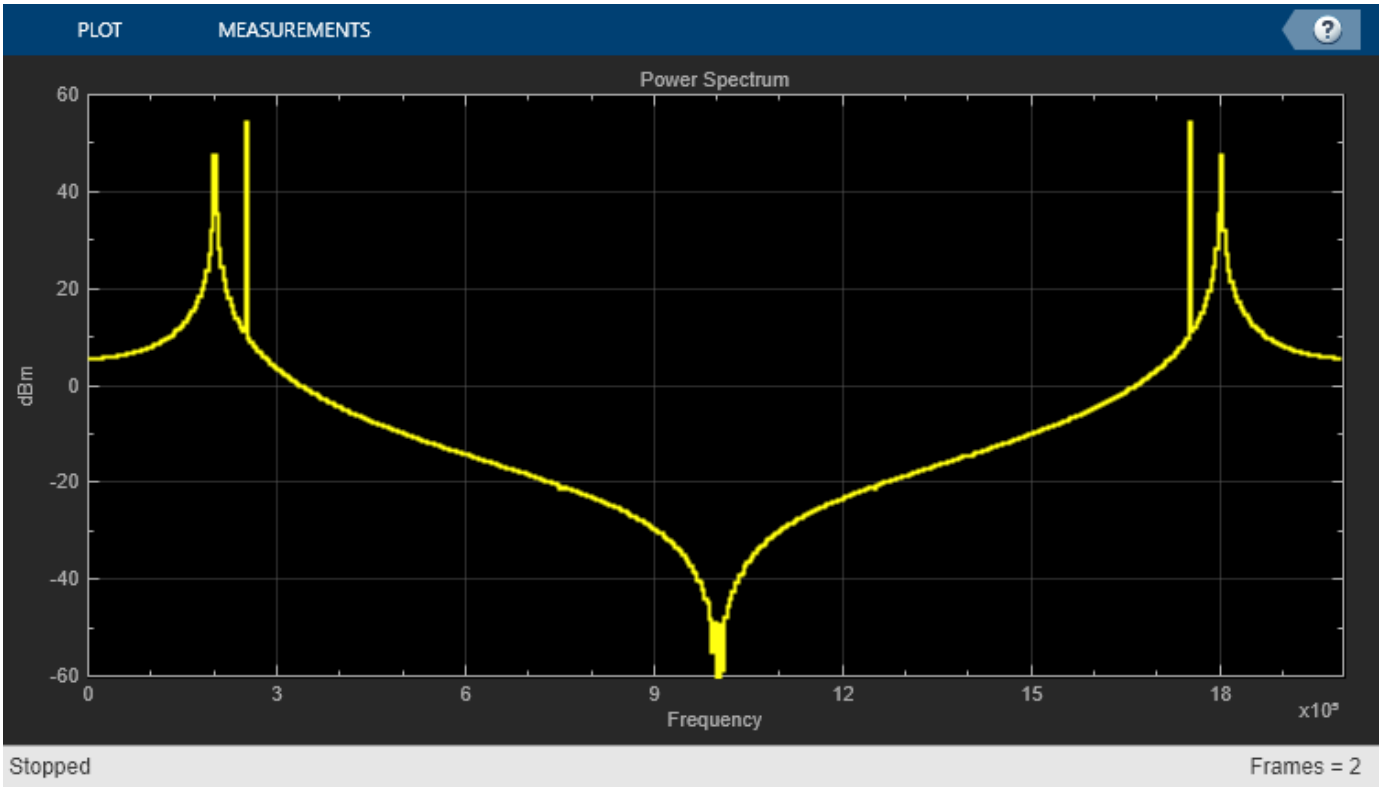
```
FFTLength  = 512;
```

The input data is two sine waves, 200 KHz and 250 KHz, each sampled at 1*2e6 Hz. Data is valid every cycle.

```
Fs           = 1*2e6;
ValidPattern = 1;
```

Open the Spectrum Viewer and run the example model.

```
open_system('FFTHDLOptimizedExample_Burst/Spectrum Viewer/Power Spectrum viewer');
sim(modelname);
```



Generate HDL Code and Test Bench

An HDL Coder™ license is required to generate HDL for this example.

Choose one of the models to generate HDL code and test bench:

```
systemname = 'FFTHDLOptimizedExample_Burst/FFT HDL Optimized Burst';
```

or

```
systemname = 'FFTHDLOptimizedExample_Streaming/FFT HDL Optimized Streaming';
```

Use this command to generate HDL code for either FFT mode. The generated can be used for any FPGA or ASIC target.

```
makehdl(systemname);
```

Use this command to generate a test bench that compares the results of an HDL simulation against the Simulink simulation behavior.

```
makehdltb(systemname);
```

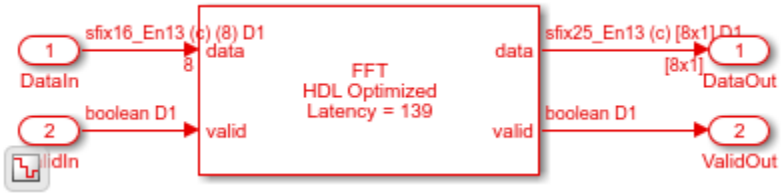
Automatic Delay Matching for the Latency of FFT HDL Optimized Block

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” on page 9-40 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 Multirate and Multistage Filters

- “FIR Decimation Using Single-Rate Processing” on page 10-2
- “FIR Decimation Using Multirate Frame-Based Processing” on page 10-3
- “Polyphase Implementation of FIR Decimation Block” on page 10-4
- “Two-Stage Multirate Narrow Lowpass Filter” on page 10-6
- “FIR Interpolation Using Single-Rate Processing” on page 10-8
- “FIR Interpolation Using Multirate Frame-Based Processing” on page 10-9
- “Polyphase Implementation of FIR Interpolation Block” on page 10-10

FIR Decimation Using Single-Rate Processing

In this example, the **FIR Decimation** block decimates the signal by decreasing the frame size, while keeping the same sample rate.

Open the model. The input to the **FIR Decimation** block is a single-channel input with a frame size of 64. In the block dialog box, set the **Decimation factor** parameter to 4 and the **Rate options** parameter to `Enforce single-rate processing`.

Run the Model. The block operates in the single-rate processing mode. Therefore, the output frame size of the block is $1/M$ times the input frame size of the block, where M is the decimation factor. With a decimation factor of 4, the output has a frame size of 16. As shown in the model, the input and the output of the FIR Decimation block have the same sample rate and different frame sizes.

Copyright 2006-2021 The MathWorks, Inc.

e: This model creates workspace variables called "mtlb", "Fs", and "dsp_examples_yout".

See Also

Blocks
Signal From Workspace | FIR Decimation | To Workspace | Probe

Related Examples

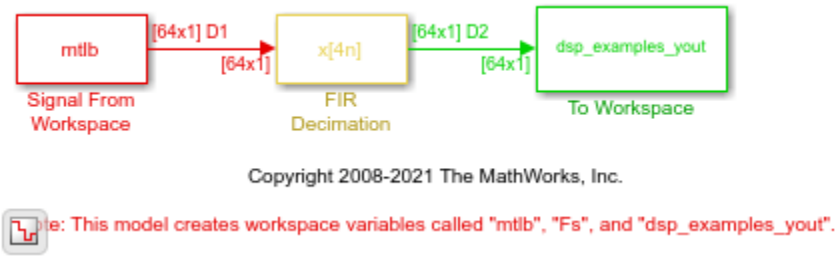
- “FIR Decimation Using Multirate Frame-Based Processing” on page 10-3
- “FIR Interpolation Using Single-Rate Processing” on page 10-8
- “Polyphase Implementation of FIR Decimation Block” on page 10-4
- “Two-Stage Multirate Narrow Lowpass Filter” on page 10-6

FIR Decimation Using Multirate Frame-Based Processing

In this example, the **FIR Decimation** block decimates the signal by decreasing the sample rate of the signal, while maintaining the frame size.

Open the model. The input to the **FIR Decimation** block is a single-channel input with a frame size of 64 and a frame period of 1 second. In the block dialog box, set the **Decimation factor** parameter to 4 and the **Rate options** parameter to `Allow multirate processing`.

Run the model. The block operates in the multirate frame-based processing mode. The input and output signals of the FIR Decimation block have the same frame size, but the sample rate of the output is $1/M$ times the sample rate of the input, where M is the decimation factor. With a decimation factor of 4, the sample rate of the output signal is 1/4 times the sample rate of the input. The color coding in the model shows the sample time of the signal at both the input and output ports of the **FIR Decimation** block.



See Also

Blocks

Signal From Workspace | FIR Decimation | To Workspace | Probe

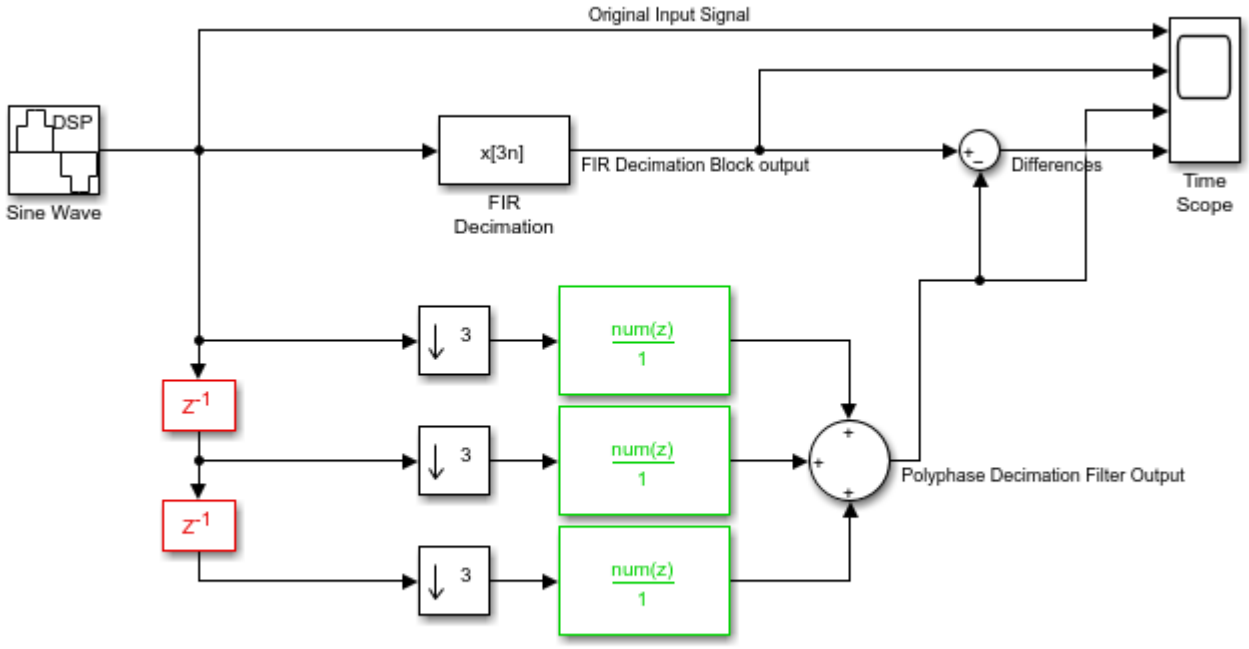
Related Examples

- “FIR Decimation Using Single-Rate Processing” on page 10-2
- “Polyphase Implementation of FIR Decimation Block” on page 10-4
- “Two-Stage Multirate Narrow Lowpass Filter” on page 10-6

Polyphase Implementation of FIR Decimation Block

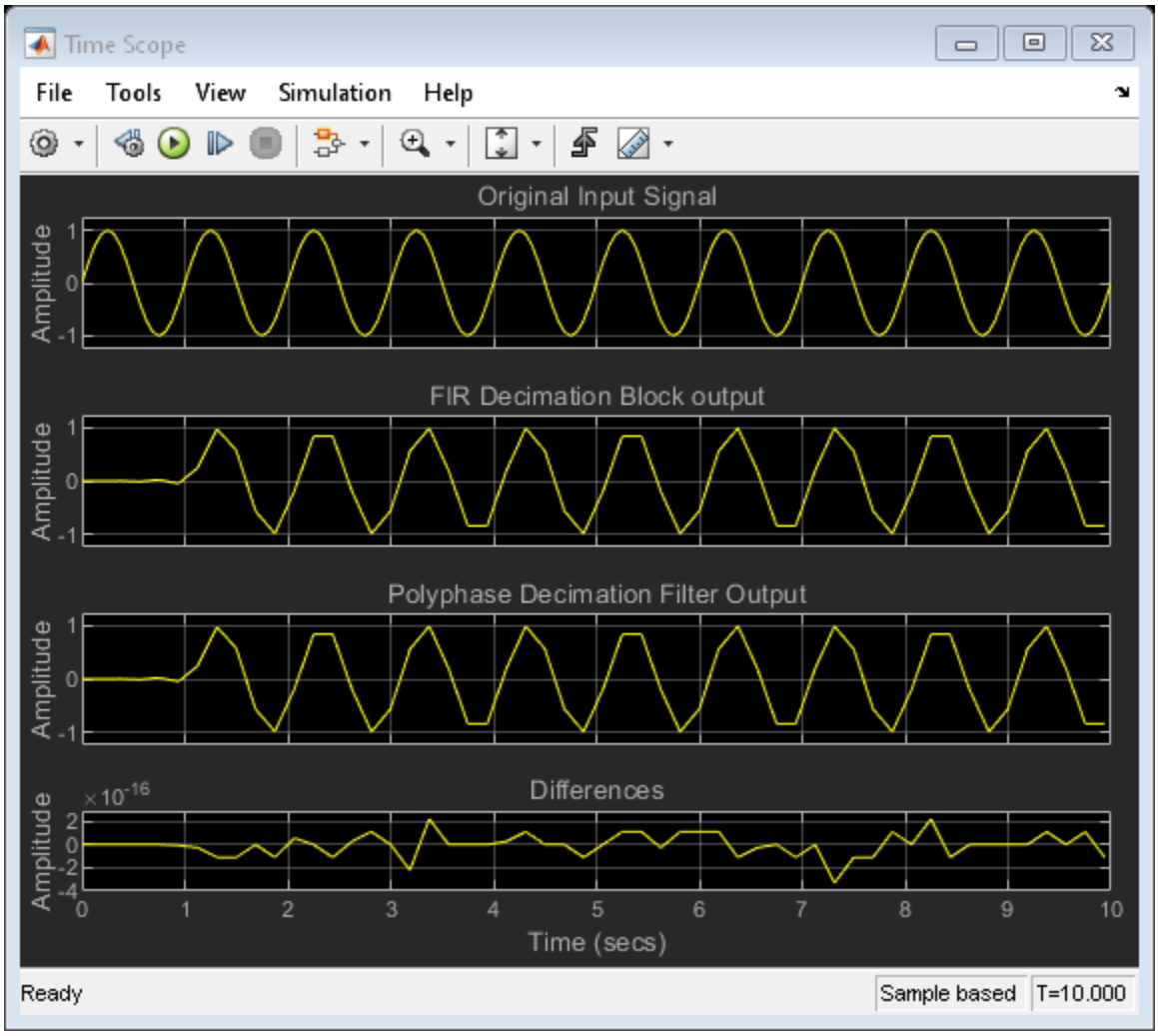
This example shows the underlying polyphase implementation of the **FIR Decimation** block using downsampling and FIR filtering as building blocks. The model compares the output of the **FIR Decimation** block with the output of a polyphase decimation structure. The **Time Scope** block compares the two outputs.

Open the model.



The model has two branches. The first branch contains an **FIR Decimation** block. The second branch contains a polyphase decimation structure. For more details on the structure, see the “Algorithms” section in the FIR Decimation block page.

Run the model and view the results on the scope. The first signal on the scope is the original sinusoidal signal. The second and third signals on the scope are the outputs of the **FIR Decimation** block and the polyphase structure, respectively. The last signal on the scope shows that the error between the two decimated outputs is very small, almost zero. This negligible error confirms the underlying polyphase implementation of the **FIR Decimation** block.



See Also

Blocks

Sine Wave | FIR Decimation | Time Scope | Discrete FIR Filter | Delay | Downsample

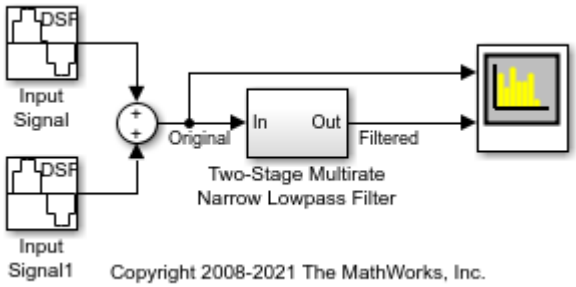
Related Examples

- “FIR Decimation Using Single-Rate Processing” on page 10-2
- “FIR Decimation Using Multirate Frame-Based Processing” on page 10-3
- “Two-Stage Multirate Narrow Lowpass Filter” on page 10-6

Two-Stage Multirate Narrow Lowpass Filter

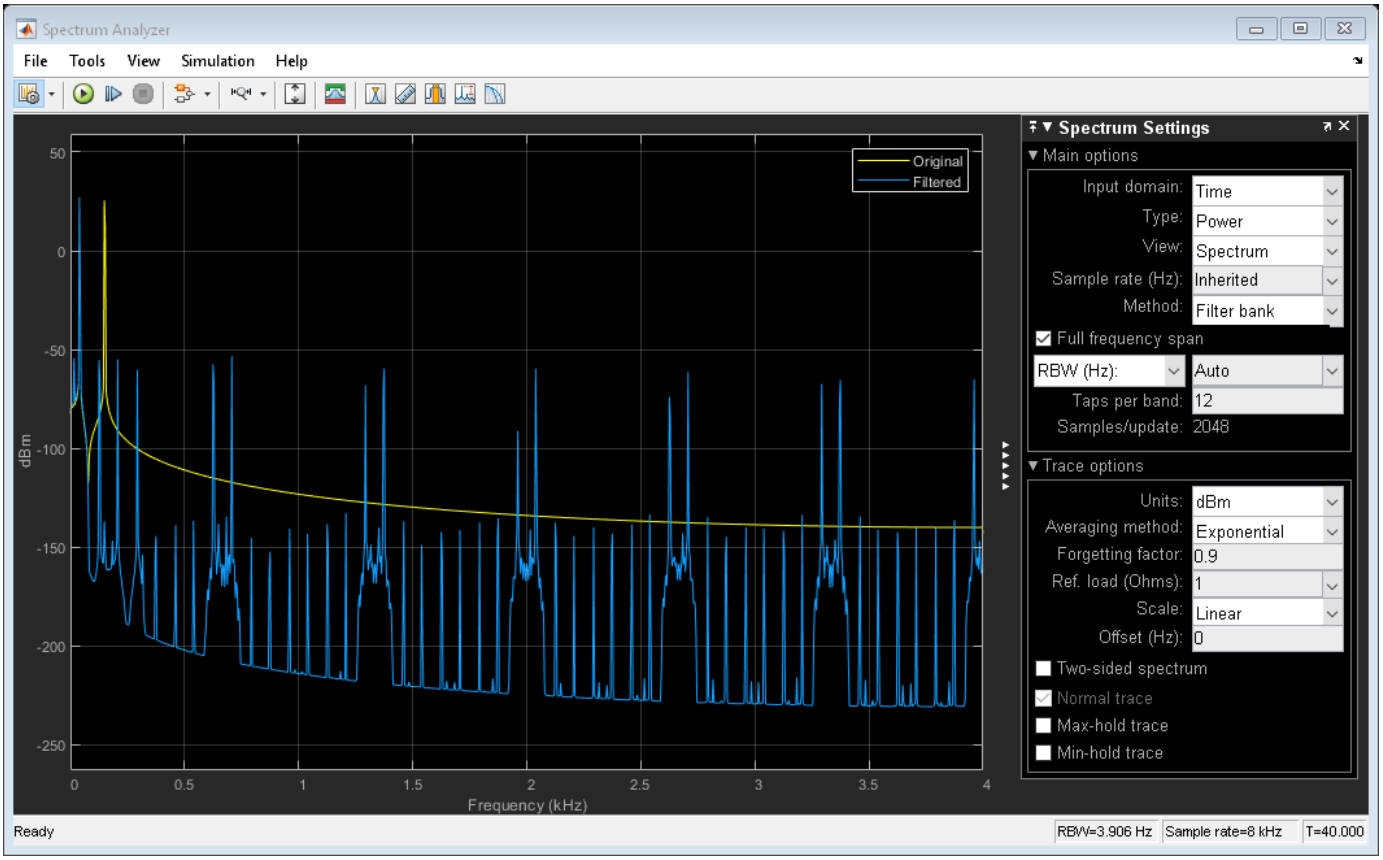
Implement a two-stage multirate narrow lowpass filter using the FIR Decimation and FIR Interpolation blocks.

Open the model.



The input signal contains a sum of two sine waves with frequencies 40 Hz and 100 Hz, respectively. The sample rate of the signal is 8000 Hz. The input sinusoidal signal is filtered using a two-stage multirate narrow lowpass filter. The filter is implemented as a subsystem block containing a two-stage decimation filter followed by a two-stage interpolation filter. The decimation filter contains two FIR Decimation blocks: first block with a decimation factor of 12 and second block with a decimation factor of 4. The interpolation filter similarly contains two FIR Interpolation blocks: first block with an interpolation factor of 4 and second block with an interpolation factor of 12. The overall filter you design is a narrow lowpass filter with a passband frequency of 75 Hz.

Run the model. Use a Spectrum Analyzer to compare the power spectra of the original and filtered signals. The block uses the `Filter Bank` method for spectral estimation. For more details, see “Spectrum Estimation — Filter Bank”. The 40 Hz frequency falls in the passband of the filter and is unattenuated. The 100 Hz frequency falls in the stopband of the filter and is therefore attenuated.



See Also

Blocks

Sine Wave | FIR Decimation | FIR Interpolation | Spectrum Analyzer

Related Examples

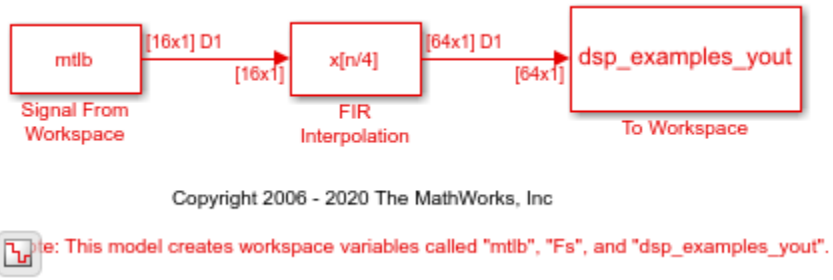
- “FIR Decimation Using Single-Rate Processing” on page 10-2
- “FIR Interpolation Using Single-Rate Processing” on page 10-8
- “FIR Decimation Using Multirate Frame-Based Processing” on page 10-3
- “FIR Interpolation Using Multirate Frame-Based Processing” on page 10-9
- “Polyphase Implementation of FIR Decimation Block” on page 10-4

FIR Interpolation Using Single-Rate Processing

In this example, the FIR Interpolation block interpolates the signal by increasing the frame size, while keeping the same sample rate.

Open the model. The input to the FIR Interpolation block is a single-channel input with a frame size of 16. In the block dialog box, set the **Interpolation factor** parameter to 4 and the **Rate options** parameter to `Enforce single-rate processing`.

Run the model. The block operates in the single-rate processing mode. Therefore, the output frame size of the block is L times the input frame size of the block, where L is the interpolation factor. With an interpolation factor of 4, the output has a frame size of 64. As shown in the model, the input and the output of the FIR Interpolation block have the same sample rate but have different frame sizes.



See Also

Blocks

FIR Interpolation | Signal From Workspace | To Workspace | Probe

Related Examples

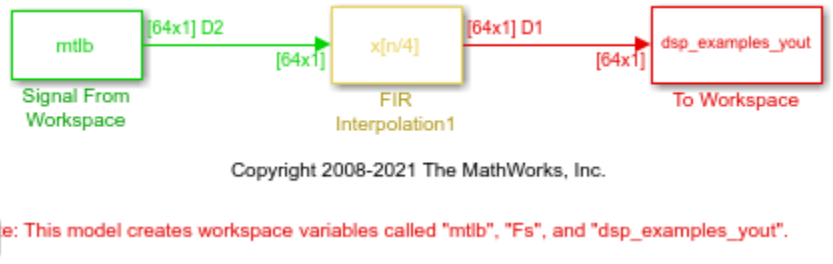
- “FIR Interpolation Using Multirate Frame-Based Processing” on page 10-9
- “Polyphase Implementation of FIR Interpolation Block” on page 10-10
- “Two-Stage Multirate Narrow Lowpass Filter” on page 10-6

FIR Interpolation Using Multirate Frame-Based Processing

In this example, the FIR Interpolation block interpolates the signal by increasing the sample rate of the signal, while maintaining the frame size.

Open the model. The input to the FIR Interpolation block is a single-channel input with a frame size of 64 and a frame period of 1 second. In the block dialog box, set the **Interpolation factor** parameter to 4 and the **Rate options** parameter to `Allow multirate processing`.

Run the model. The block operates in the multirate frame-based processing mode. The input and output signals of the FIR Interpolation block have the same frame size, but the sample rate of the output is L times the sample rate of the input, where L is the interpolation factor. With an interpolation factor of 4, the sample rate of the output signal is 4 times the sample rate of the input signal. The color coding in the model shows the sample time (inverse of sample rate) of the signal at both the input and output ports of the FIR Interpolation block.



See Also

Blocks
FIR Interpolation | Signal From Workspace | To Workspace | Probe

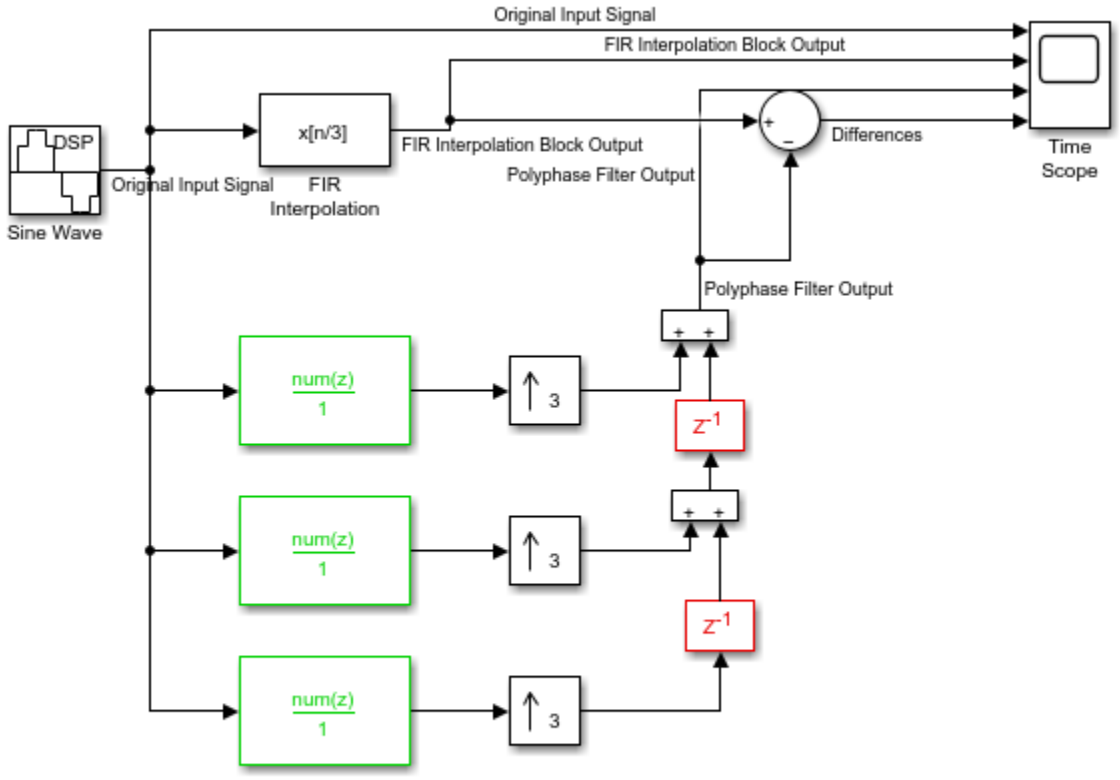
Related Examples

- “FIR Interpolation Using Single-Rate Processing” on page 10-8
- “Polyphase Implementation of FIR Interpolation Block” on page 10-10
- “Two-Stage Multirate Narrow Lowpass Filter” on page 10-6

Polyphase Implementation of FIR Interpolation Block

This example shows the underlying polyphase implementation of the FIR Interpolation block using upsampling and FIR filtering as building blocks. The model compares the output of the FIR Interpolation block with the output of a polyphase interpolation structure. The Time Scope block compares the two outputs.

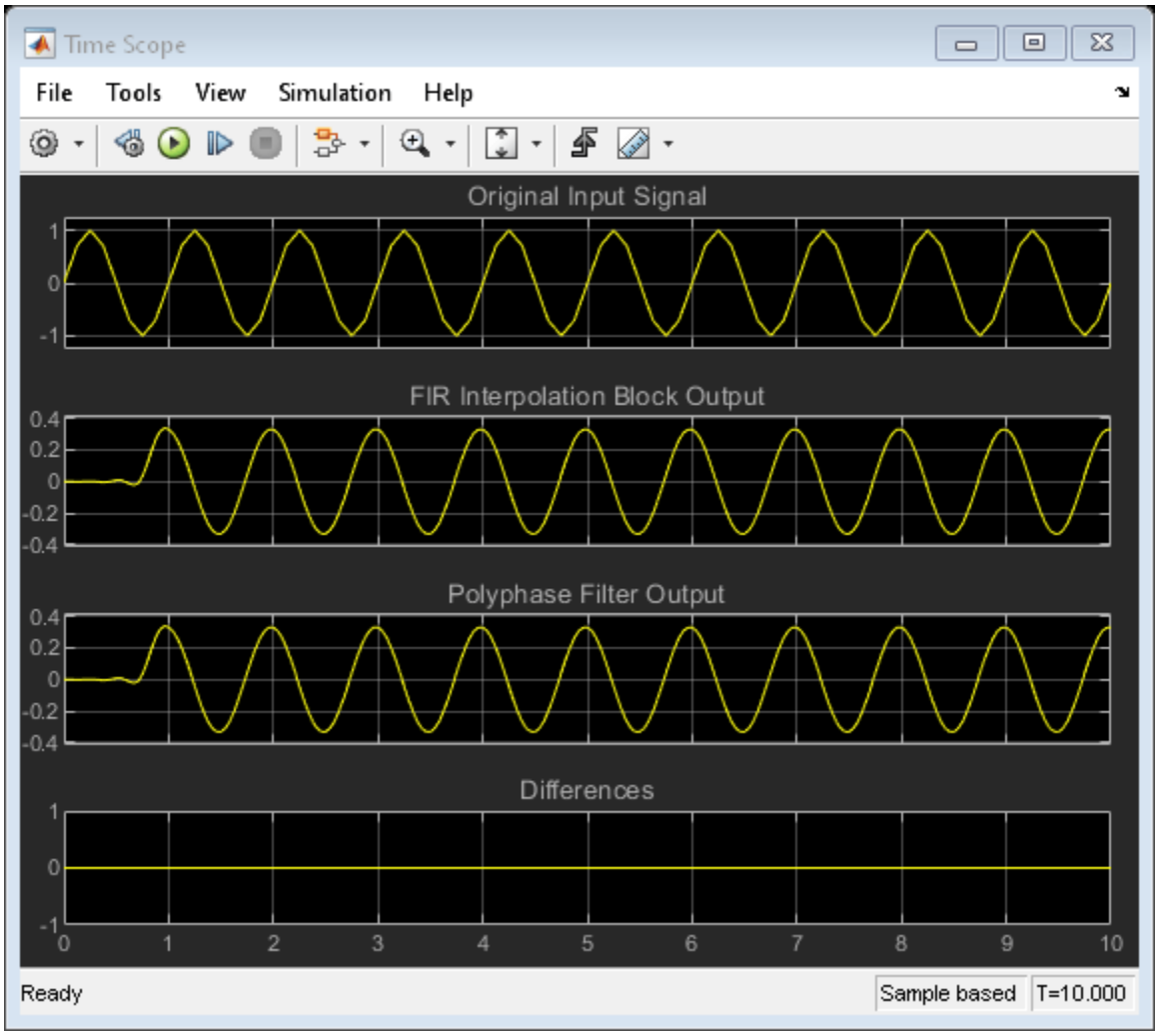
Open the model.



Copyright 2008-2021 The MathWorks, Inc.

The model has two branches. The first branch contains an FIR Interpolation block. The second branch contains a polyphase interpolation structure. For more details on the derivation of this structure, see the “Algorithms” section in the FIR Interpolation block page.

Run the model and view the results on the scope. The first signal on the scope is the original sinusoidal signal. The second and third signals on the scope are the outputs of the FIR Interpolation block and the polyphase structure, respectively. The last signal on the scope shows that the error between the two interpolated outputs is very small, almost zero. This negligible error confirms the underlying polyphase implementation of the FIR Interpolation block.



See Also

Blocks

Sine Wave | FIR Interpolation | Time Scope | Discrete FIR Filter | Delay | Upsample

Related Examples

- “FIR Interpolation Using Single-Rate Processing” on page 10-8
- “FIR Interpolation Using Multirate Frame-Based Processing” on page 10-9
- “Two-Stage Multirate Narrow Lowpass Filter” on page 10-6

Simulink Block Examples in Scopes and Data Logging Category

- “Obtain Measurement Data Programmatically for dsp.SpectrumAnalyzer System object” on page 11-2
- “Obtain Measurements Data Programmatically for Spectrum Analyzer Block” on page 11-5

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

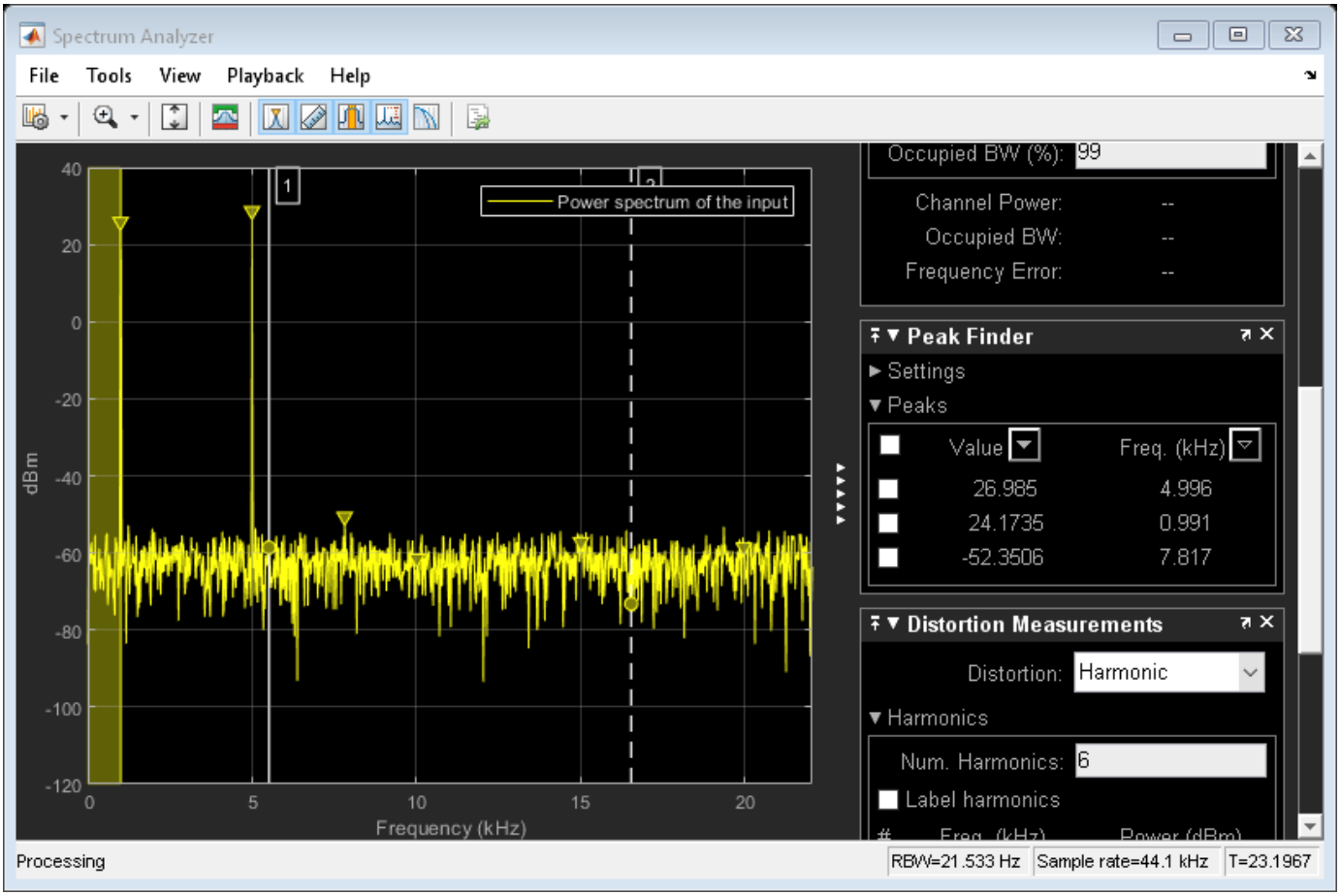
To obtain the measurements, set the `Enable` property of the measurements to `true`.

```
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.SpectrumAnalyzer` | `dsp.SineWave`

Obtain Measurements Data Programmatically for Spectrum Analyzer Block

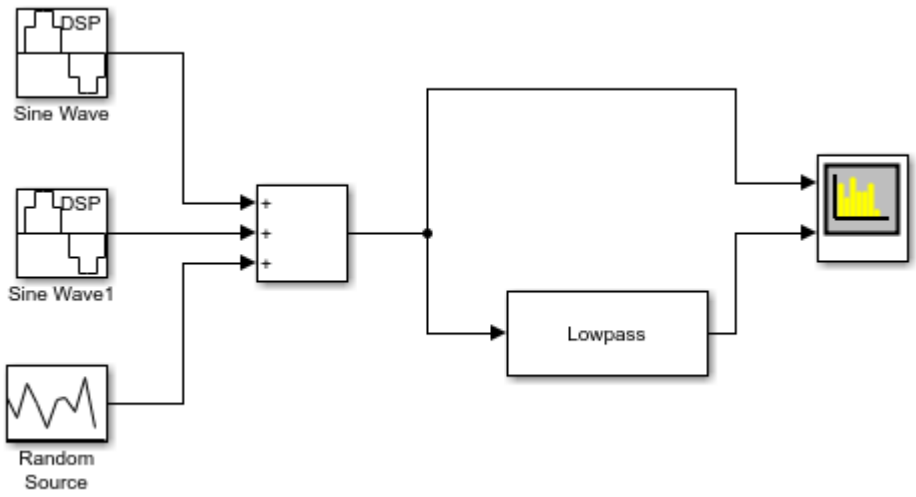
Compute and display the power spectrum of a noisy sinusoidal input signal using the Spectrum Analyzer block. Measure the peaks, cursor placements, adjacent channel power ratio, distortion, and CCDF values in the spectrum by enabling these block configuration properties:

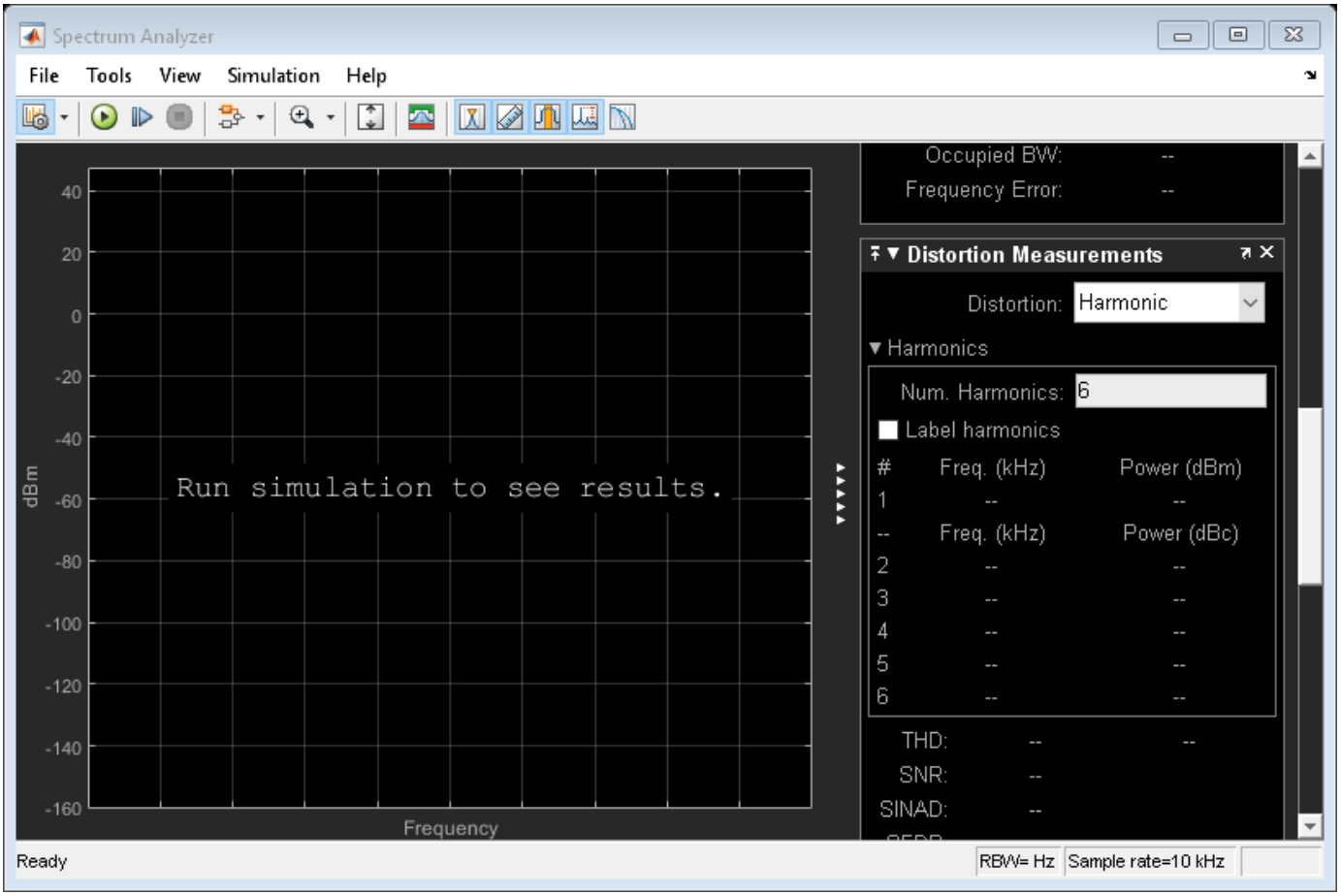
- PeakFinder
- CursorMeasurements
- ChannelMeasurements
- DistortionMeasurements
- CCDFMeasurements

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

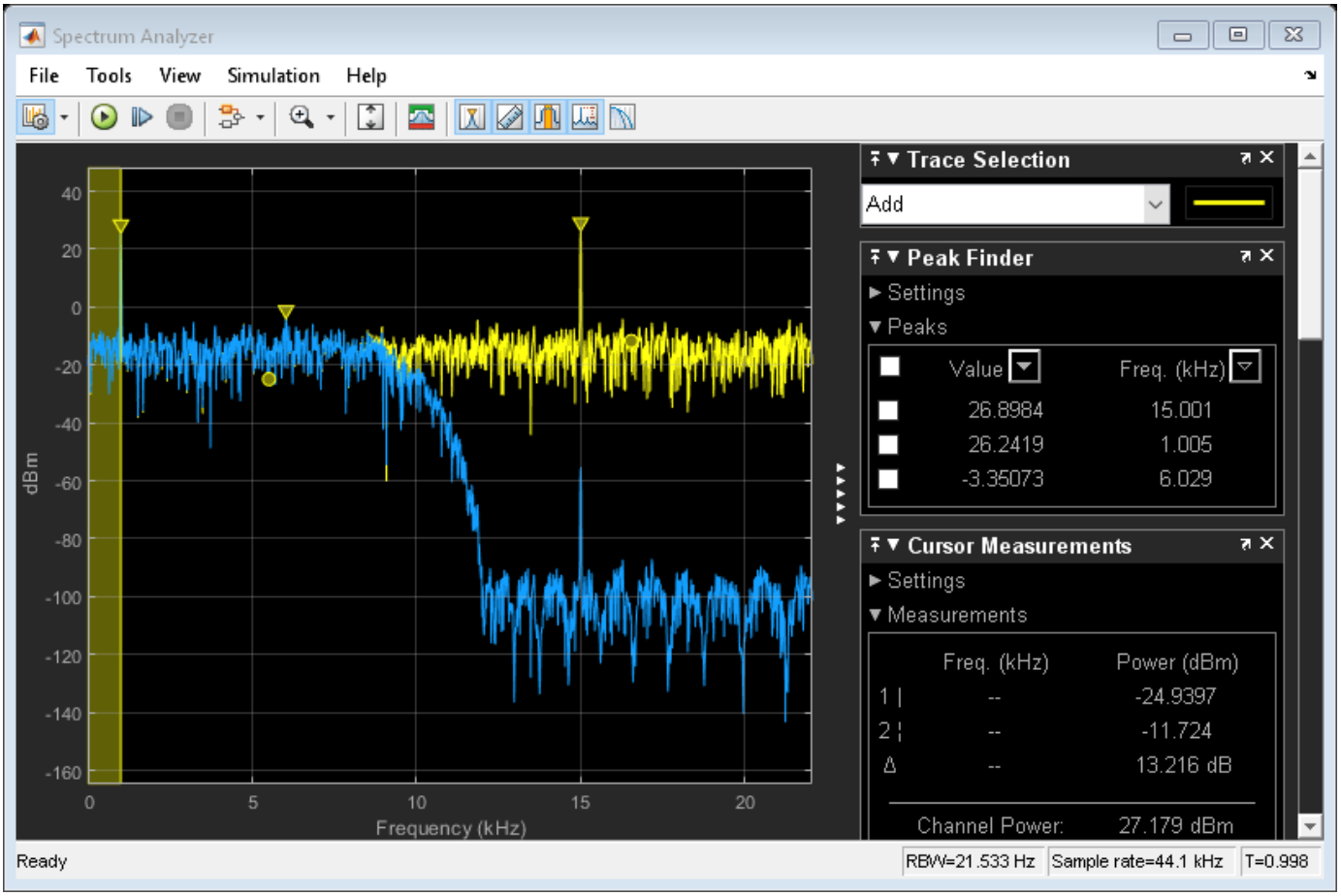
To obtain the measurements, set the `Enable` property of the measurements to `true`.

```
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    DistortionMeasu
    ______________    __________    __________________    ___________________    _______________

      {[0.9985]}      1x1 struct        1x1 struct            1x1 struct             1x1 struct
```

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 =

   26.8984
   26.2419
   -3.3507


frequencieskHz =

   15.0015
    1.0049
    6.0293
```

Save and Close the Model

```
save_system(model);
close_system(model);
```

See Also

Functions
`getMeasurementsData`

Objects
`SpectrumAnalyzerConfiguration`

Blocks
Spectrum Analyzer | Sine Wave | Lowpass Filter

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

- “Write and Read Binary Files in Simulink” on page 12-2
- “Write and Read Matrix Data from Binary Files in Simulink” on page 12-6
- “Write and Read Fixed-Point Data from Binary Files in Simulink” on page 12-8
- “Write and Read Character Data from Binary Files in Simulink” on page 12-10
- “Change the Endianness of the Data in Simulink” on page 12-11

Write and Read Binary Files in Simulink

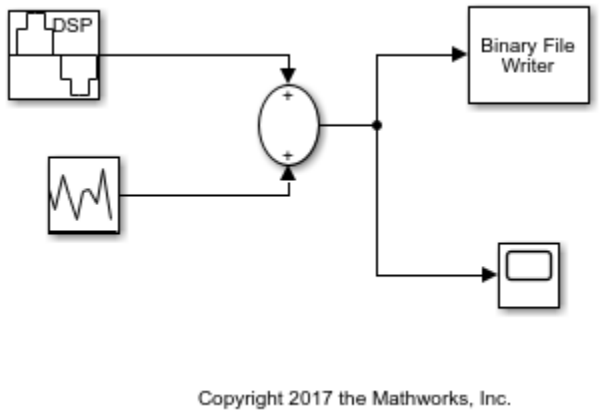
Create a binary file with a custom header using the Binary File Writer block. Write data to this file. Read the header and data using the Binary File Reader block.

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.

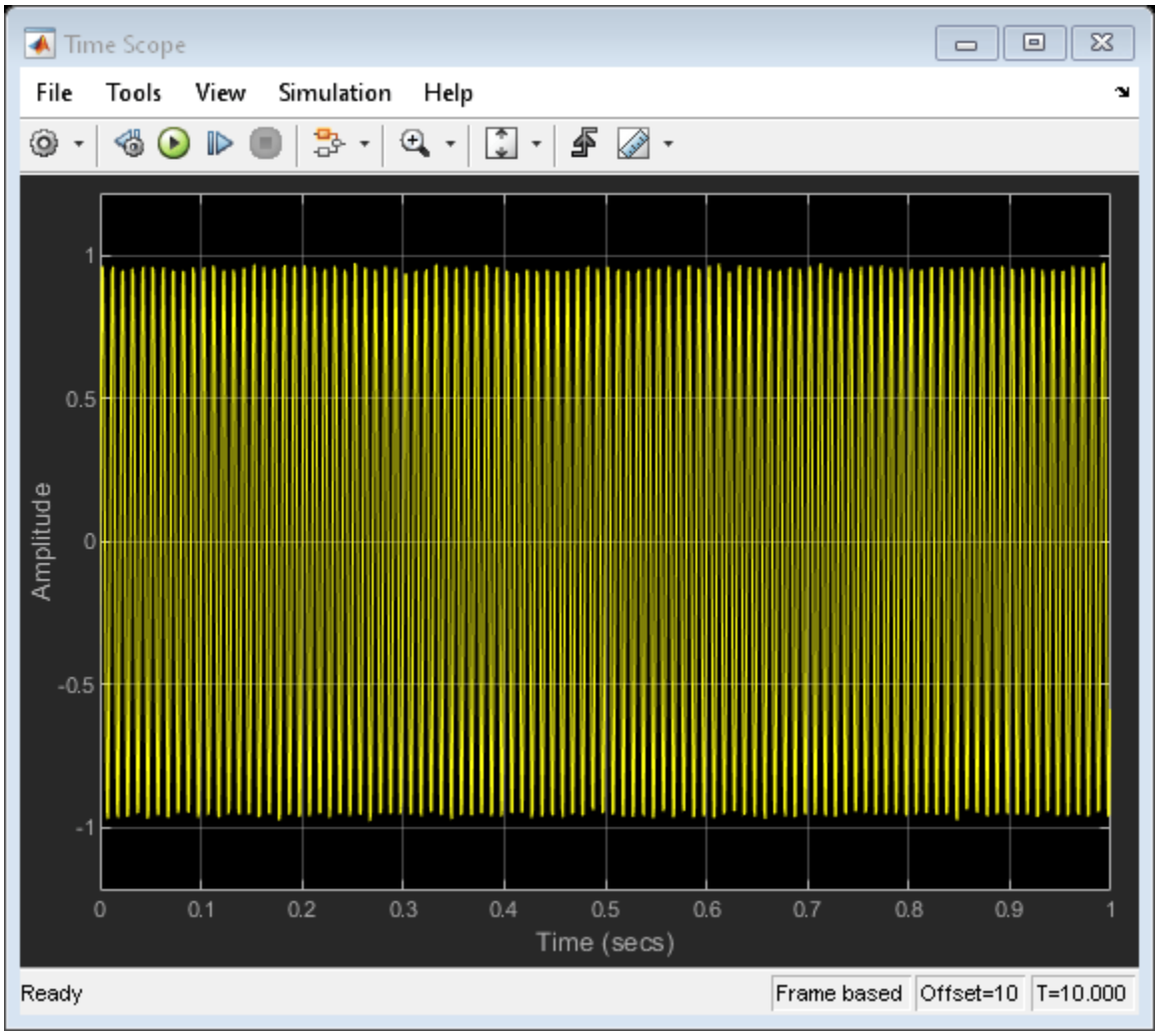
Open the model.

```
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)
```



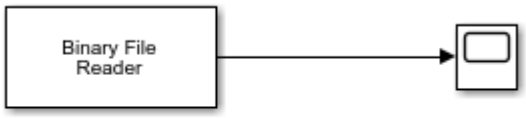
Read the Data

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.

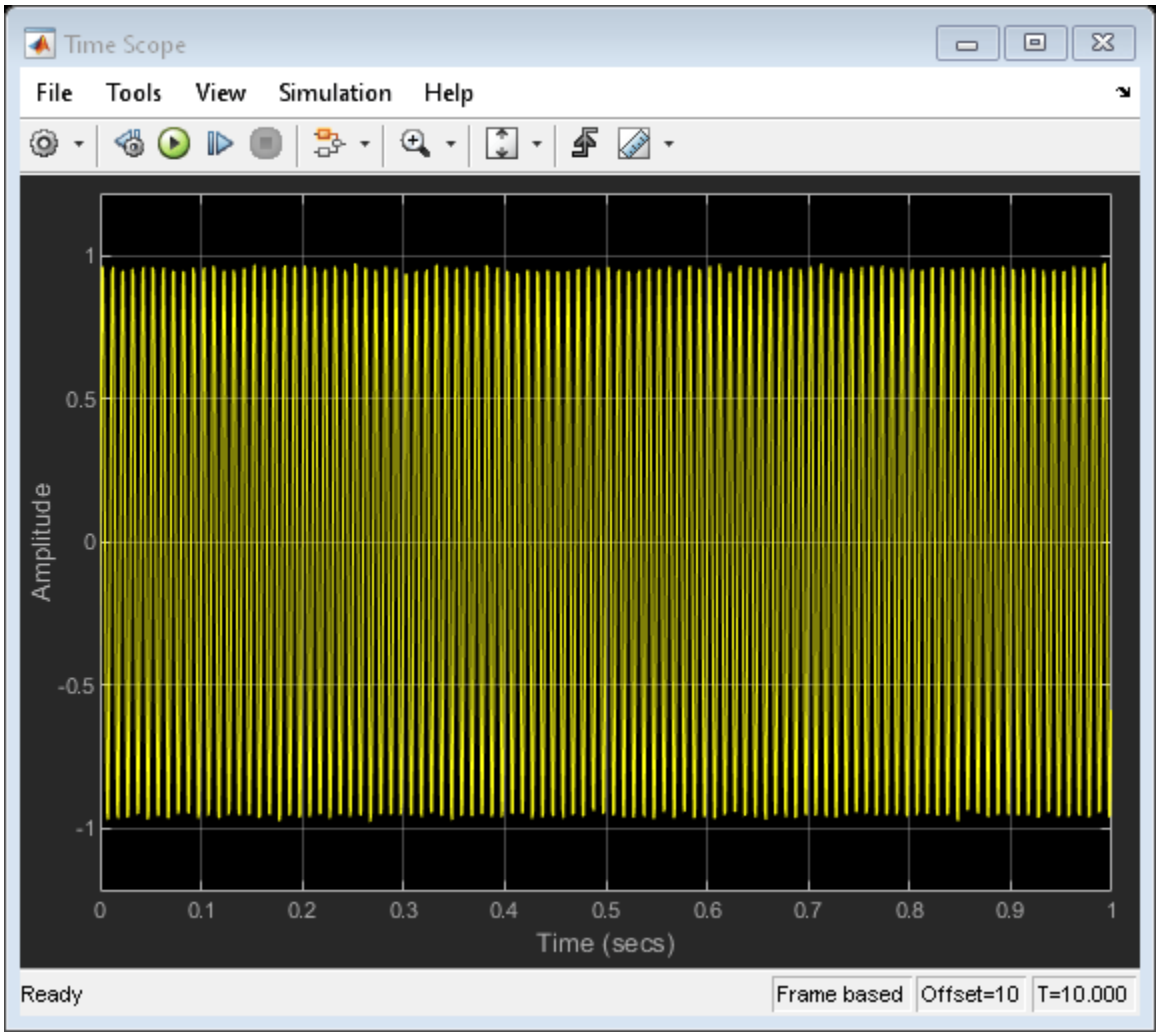
```
readModel = 'readData';
open_system(readModel)
```



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 Writer | Binary File Reader | Sine Wave | Random Source | Time Scope

Write and Read Matrix Data from Binary Files in Simulink

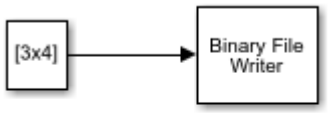
Use a Binary File Reader block to read real and complex matrix data from a binary file in a row-major format.

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`.

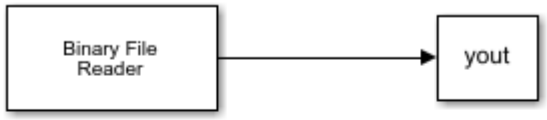
```
sim('writeMatrixData')
```

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

To Workspace | Binary File Writer | Binary File Reader

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.

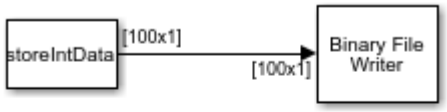
Write the Fixed-Point Data

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)
```

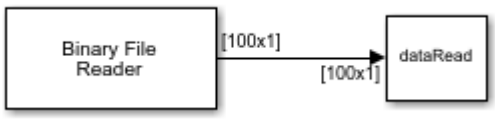


Copyright 2017 the Mathworks, Inc.

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

To Workspace | Binary File Writer | Binary File Reader | Signal From 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.

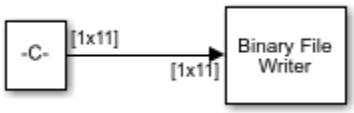
Write the Character Data

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`.

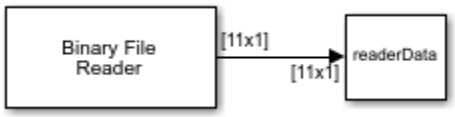
```
writeModel = 'writeCharData';
open_system(writeModel)
sim(writeModel)
```



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

To Workspace | Binary File Writer | Binary File Reader | Constant

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
`swapbytes` | `fopen` | `fwrite` | `fclose`

Blocks
Binary File Reader | Constant

Simulink Block Examples in Signal Generation and Operations Category

- “Delay Signal Using Multitap Fractional Delay” on page 13-2
- “Bidirectional Linear Sweep” on page 13-7
- “Unidirectional Linear Sweep” on page 13-10
- “When Sweep Time Is Greater than Target Time” on page 13-12
- “Sweep with Negative Frequencies” on page 13-14
- “Aliased Sweep” on page 13-17
- “Generate Discrete Impulse with Three Channels” on page 13-19
- “Generate Unit-Diagonal and Identity Matrices” on page 13-20
- “Generate Five-Phase Output from the Multiphase Clock Block” on page 13-21
- “Count Down Through Range of Numbers” on page 13-23
- “Import Two-Channel Signal From Workspace” on page 13-25
- “Import 3-D Array From Workspace” on page 13-26
- “Generate Sample-Based Sine Waves” on page 13-27
- “Generate Frame-Based Sine Waves” on page 13-28
- “Design an NCO Source Block” on page 13-29
- “Generate Constant Ramp Signal” on page 13-32
- “Averaged Power Spectrum of Pink Noise” on page 13-33
- “Downsample a Signal” on page 13-35
- “Sample and Hold a Signal” on page 13-38
- “Generate and Apply Hamming Window” on page 13-41
- “Convert Sample Rate of Speech Signal” on page 13-44
- “Unwrap Signal” on page 13-46
- “Convolution of Two Inputs” on page 13-48
- “Select Rows or Columns from Matrices” on page 13-50
- “Convert 2-D Matrix to 1-D Array” on page 13-51
- “Pad or Truncate Matrix with Constant Values” on page 13-52
- “Extract the Phase of Sine Wave” on page 13-54
- “Queues” on page 13-56

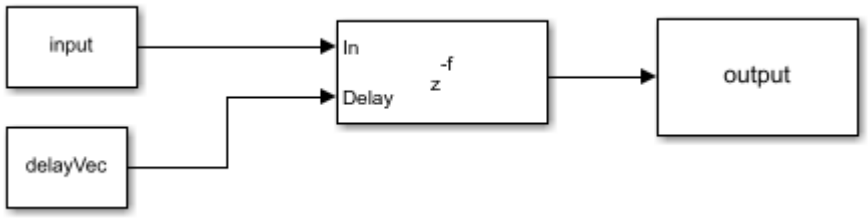
Delay Signal Using Multitap Fractional Delay

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

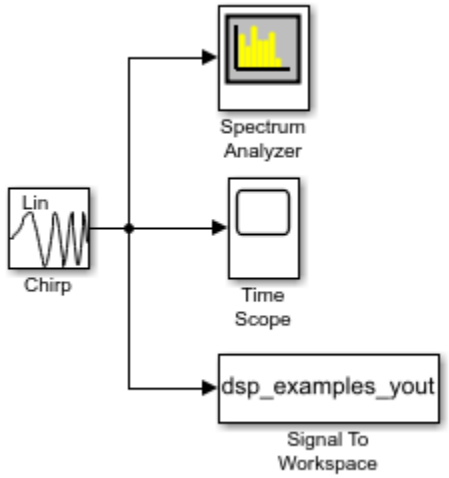
Blocks
Delay | Unit Delay | Variable Integer Delay

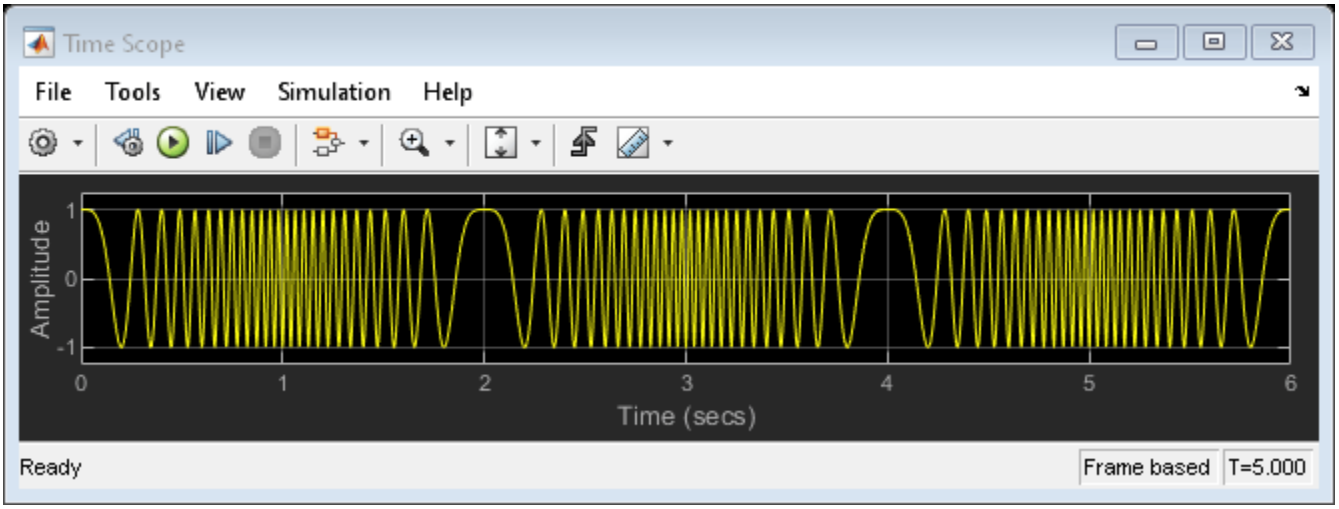
Bidirectional Linear Sweep

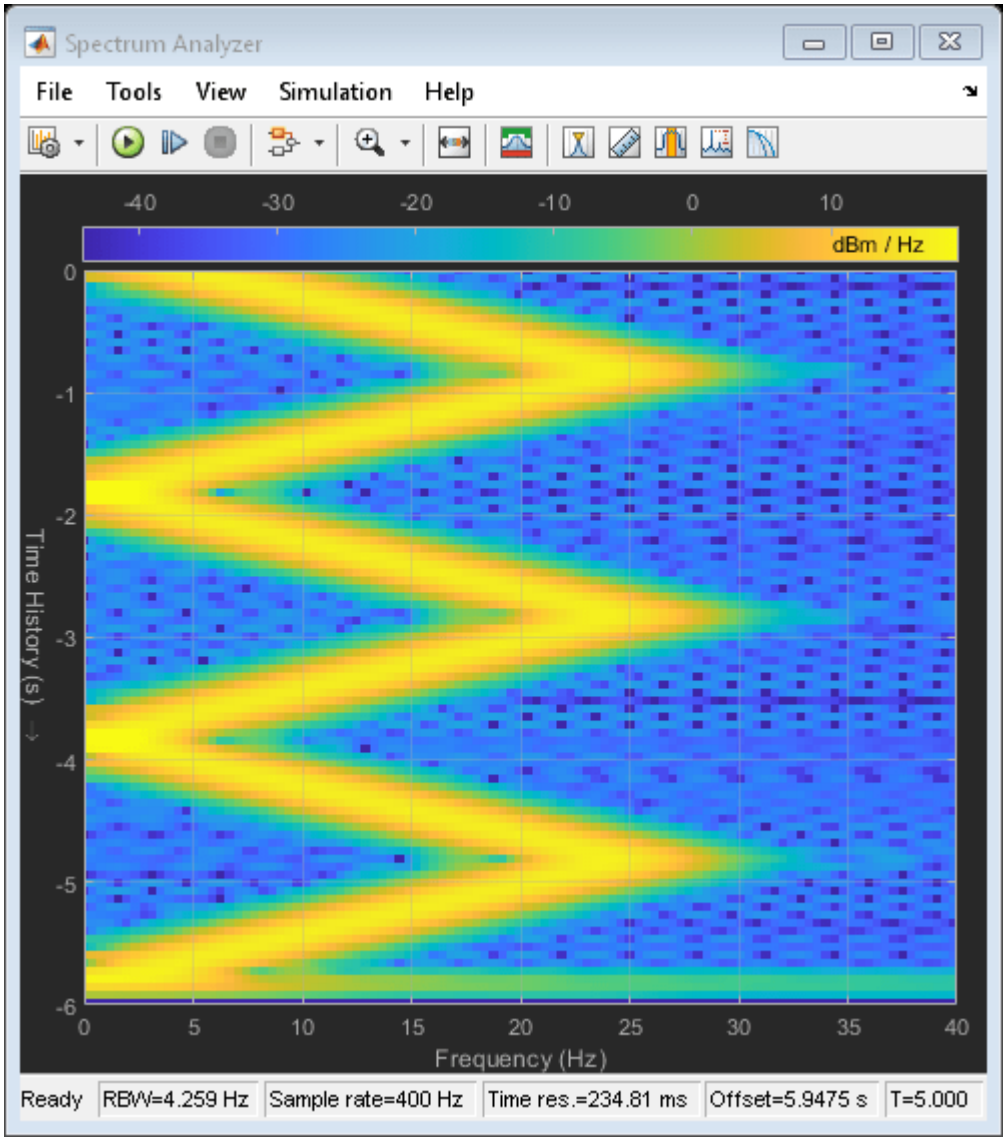
In this example, the Chirp block outputs a bidirectional, linearly swept chirp signal, which is displayed by the Time Scope and the Spectrum Analyzer. You can also export the signal to the MATLAB workspace by the Signal To Workspace block.

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.

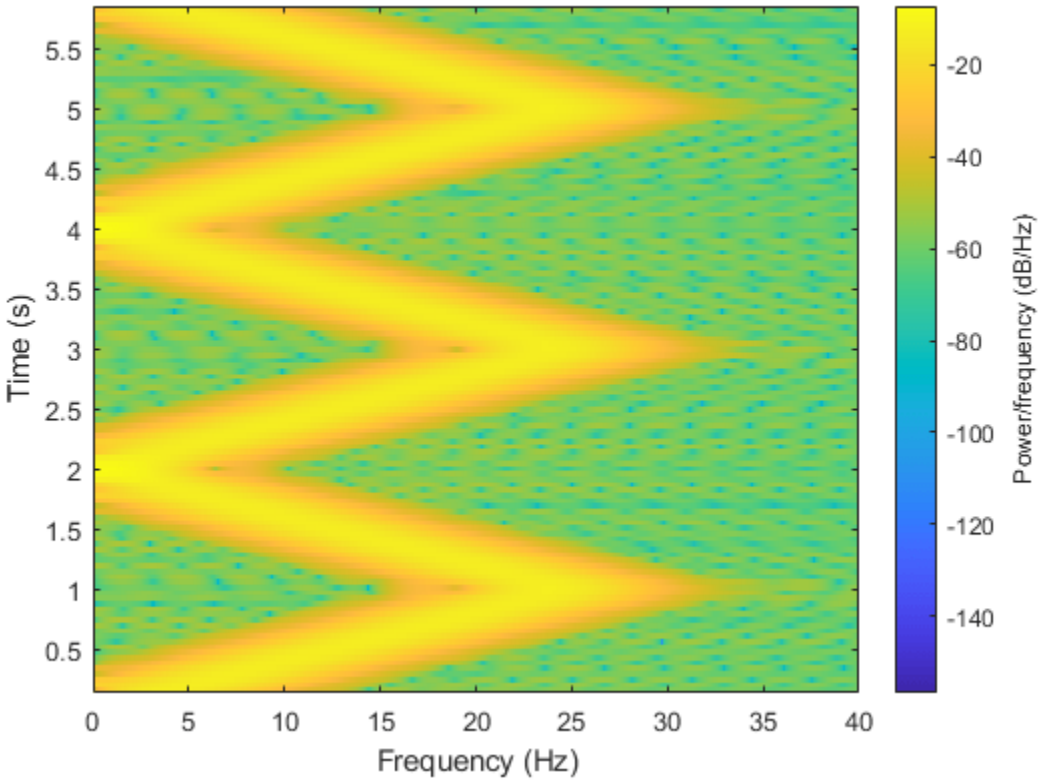






You can also view the spectrogram by saving the output to the workspace and using this command:

```
spectrogram(dsp_examples_yout,hamming(128),110,0:.01:40,400)
```



See Also

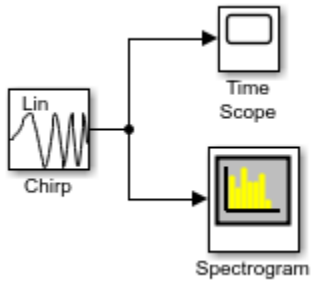
Blocks
Chirp | Signal To Workspace | Spectrum Analyzer | Time Scope

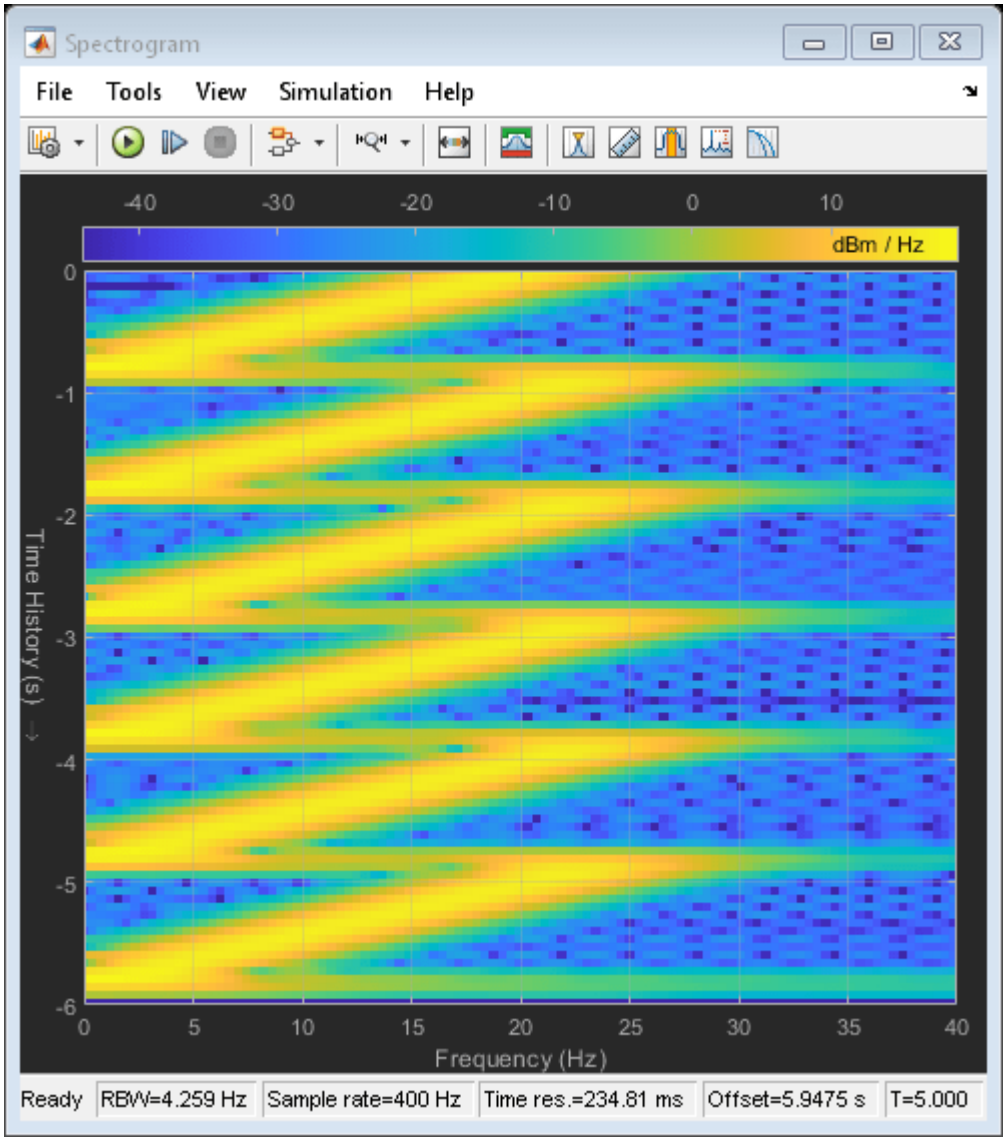
Unidirectional Linear Sweep

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

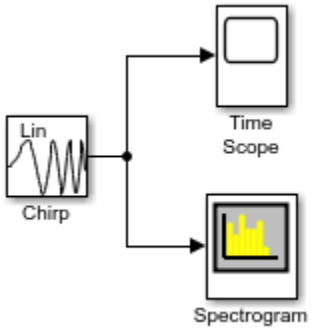
When Sweep Time Is Greater than Target Time

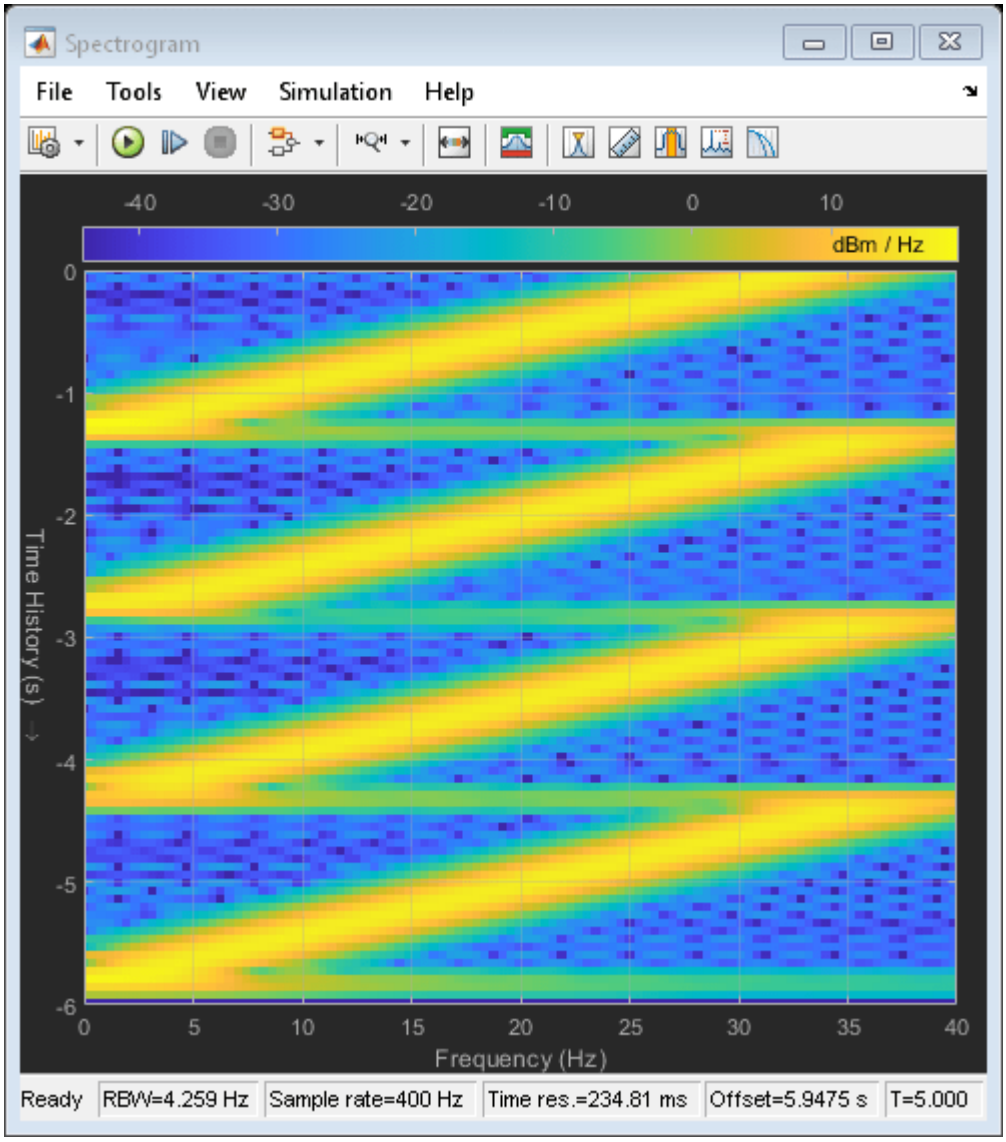
This example shows the unexpected behavior that might arise in the Chirp block when the **Sweep time** is greater than **Target time**. The Time Scope displays the signal output, and the Spectrum Analyzer displays the spectrogram in the frequency domain.

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.





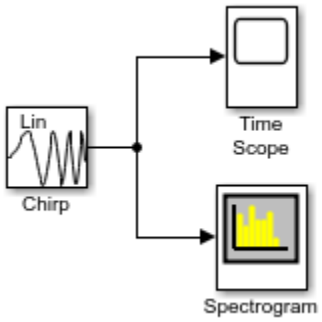
See Also

Blocks
Chirp | Spectrum Analyzer | Time Scope

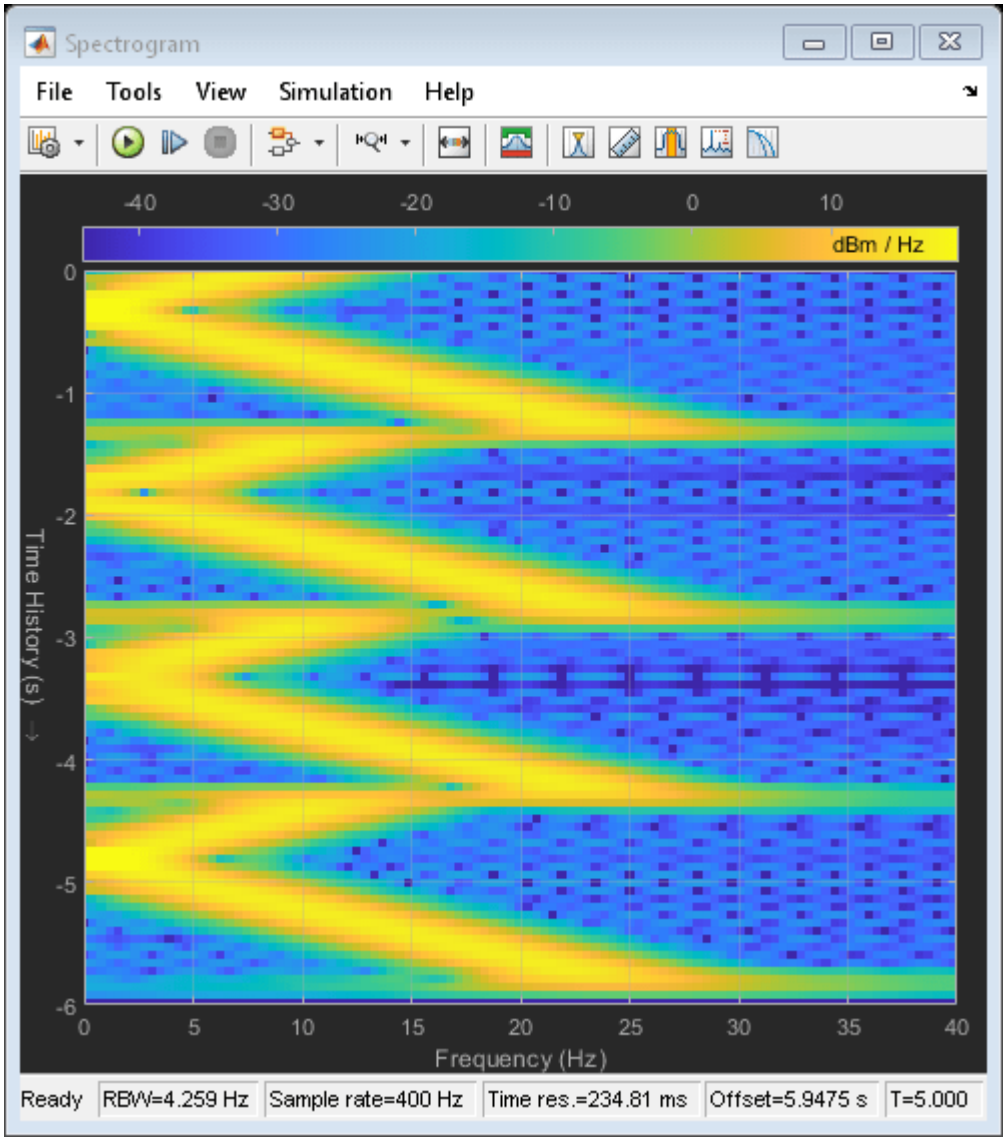
Sweep with Negative Frequencies

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=4.259 Hz
Sample rate=400 Hz
Time res.=234.81 ms
Offset=5.9475 s
T=5.000

See Also

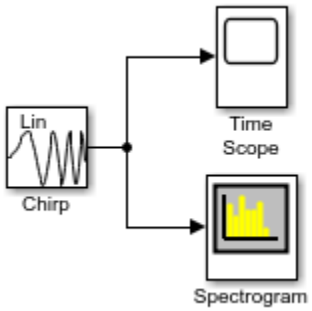
Blocks
Chirp | Spectrum Analyzer | Time Scope

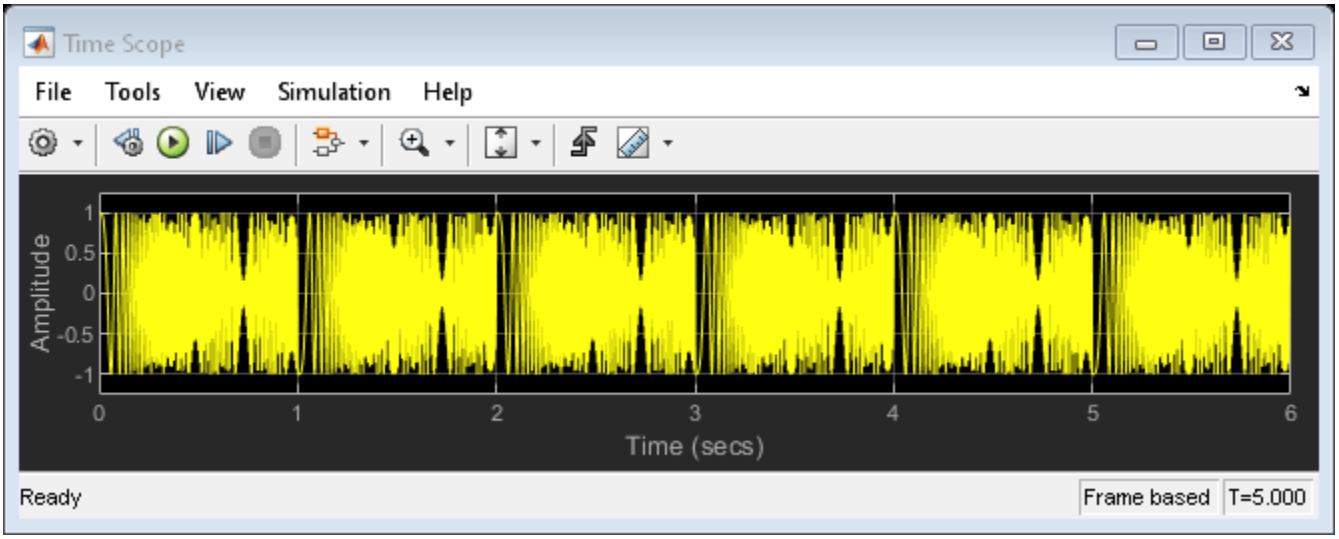
Aliased Sweep

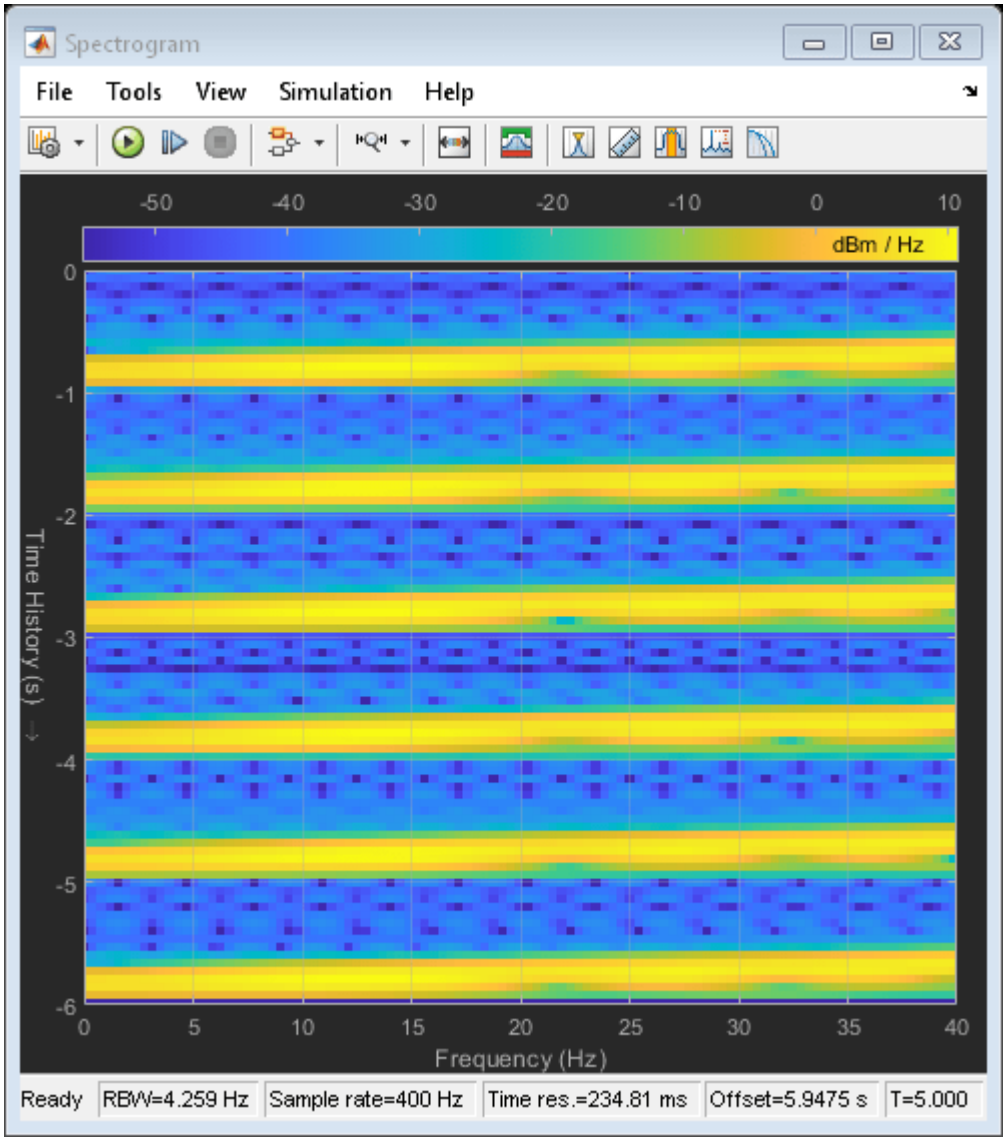
In this example, the Chirp block outputs a chirp signal containing negative frequencies. The Time Scope displays the signal output in the time domain, and the Spectrum Analyzer displays the spectrogram in the frequency domain.

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.







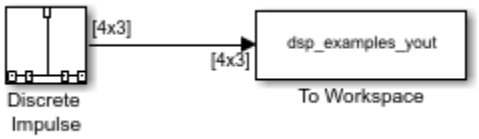
See Also

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
```

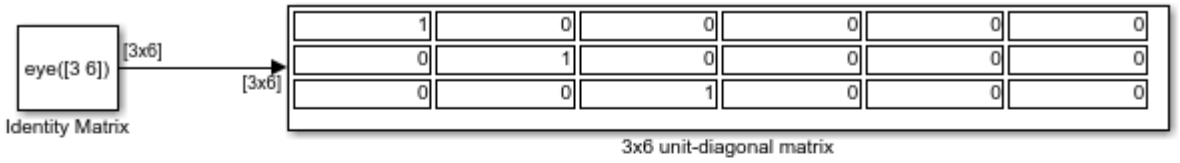
The block generates an impulse at sample 1 of channel 1 (first column), at sample 4 of channel 2 (second column), and at sample 6 of channel 3 (third column).

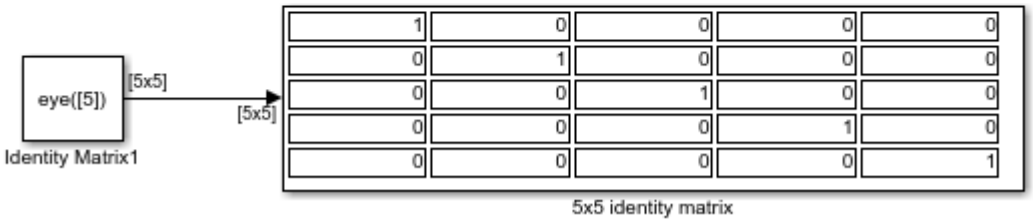
See Also

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.



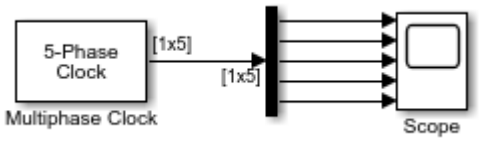


See Also

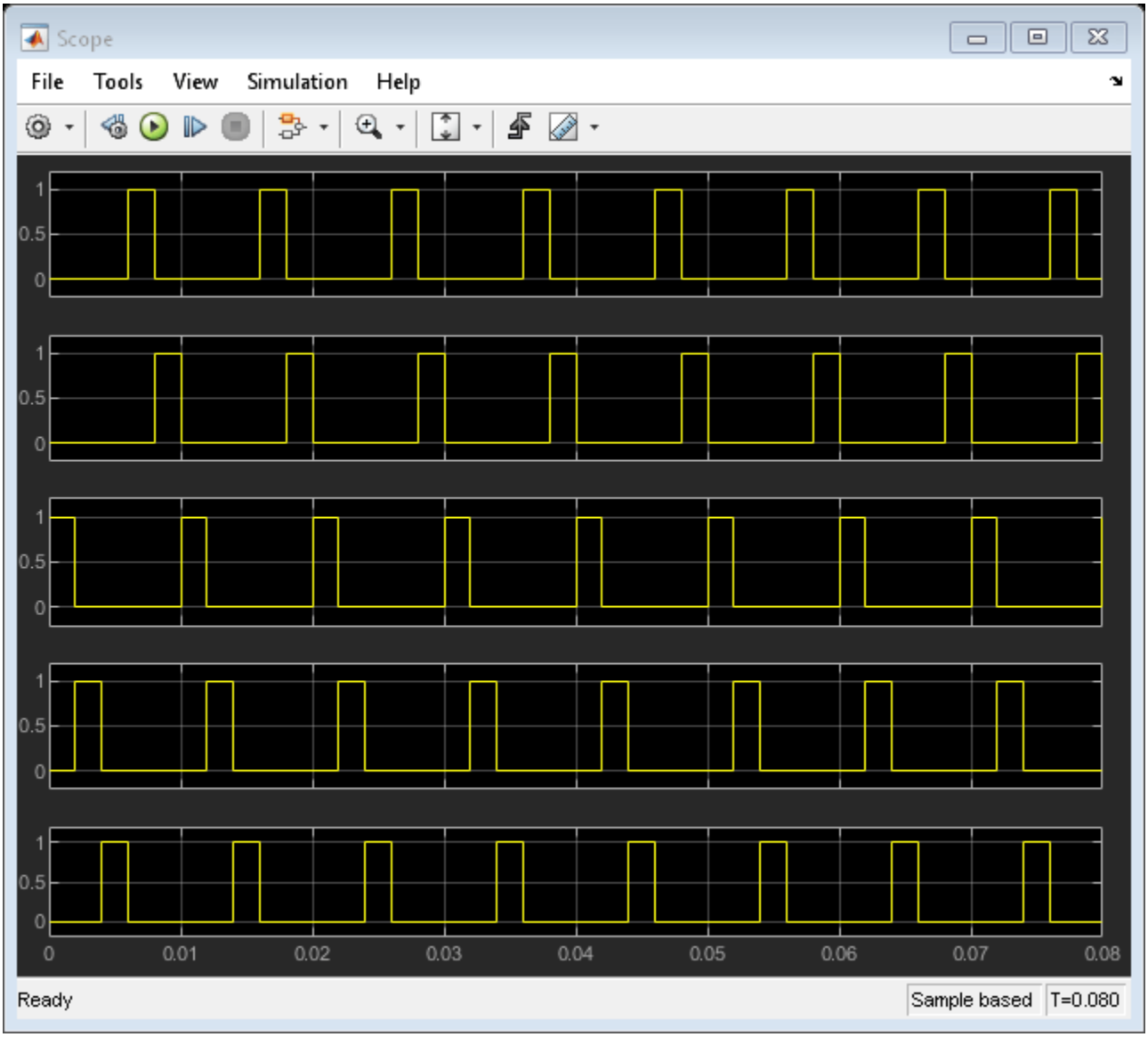
Blocks
Identity Matrix | Display

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.

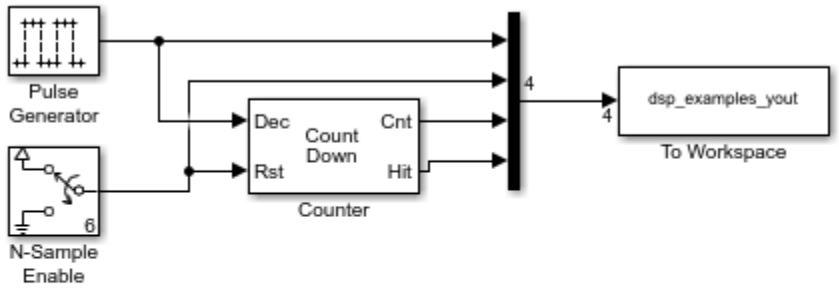


See Also

Blocks
Multiphase Clock | Scope

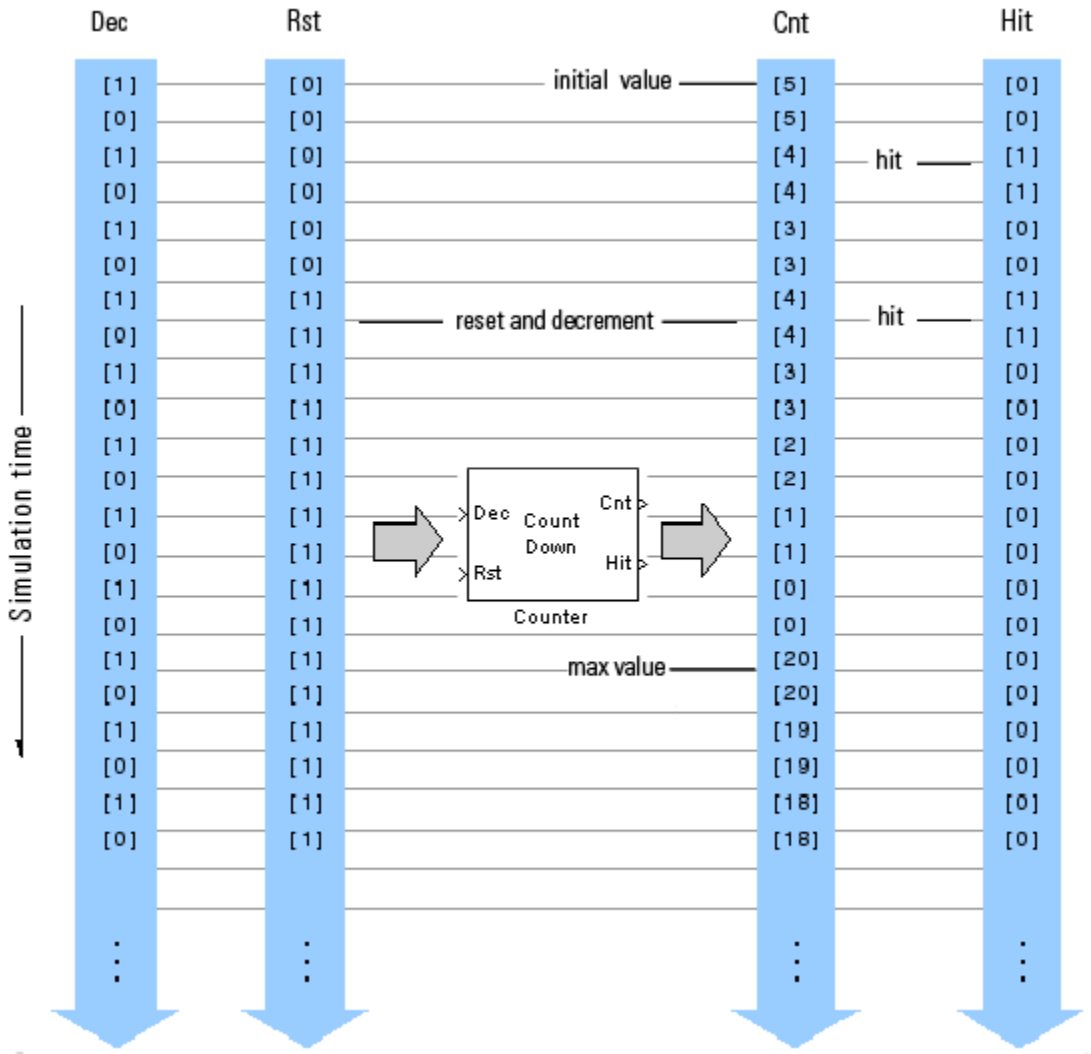
Count Down Through Range of Numbers

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 | To Workspace | Pulse Generator

Import Two-Channel Signal From Workspace

Import a two-channel signal, A, from the MATLAB® workspace into the Simulink® model using the **Signal From Workspace** block. Use the **To Workspace** block to write the imported data to the MATLAB workspace. In this model, the **To Workspace** block writes slightly modified data to the MATLAB workspace.

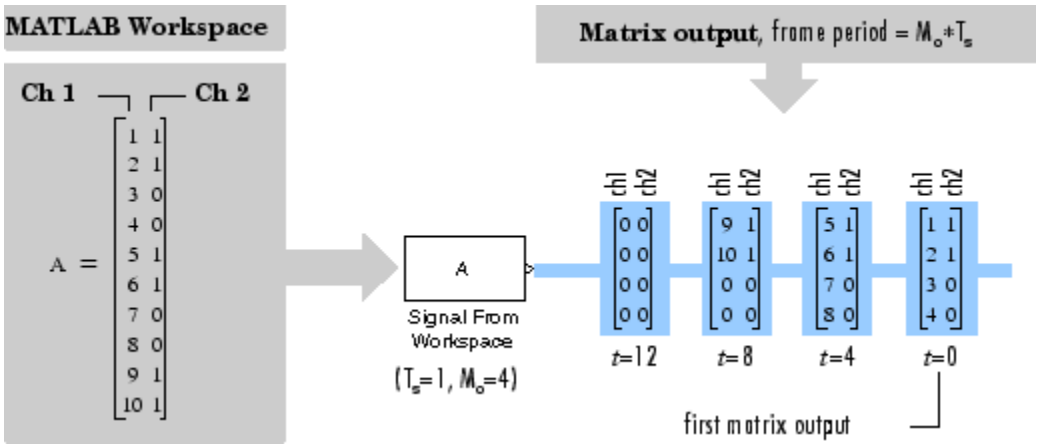
The parameters in the **Signal From Workspace** block are configured as follows:

- **Sample time** set to `1`: This parameter determines, *Ts* , the sample period of the output.
- **Samples per frame** set to `4`: Number of samples, *Mo* , to buffer into each output frame. The output frame period is *MoTs*.
- **Form output after final value by** set to `Setting to zero`: The block generates zero-valued outputs for the duration of the simulation after generating the last frame of the signal.

Open and simulate the model.

The **Signal From Workspace** block imports the signal from the variable A, which is a 10-by-2 matrix. When you simulate the model, the data output has a frame size of 4 and a frame period, *MoTs* , of 4 seconds. All outputs after the third frame (at `t = 8`) are zero because the **Form output after final data value by** parameter is set to `Setting to zero` in this model. The **To Workspace** block writes this modified output to the MATLAB workspace in the variable `dsp_ex_yout`.

This figure shows the signal in the input variable `A` and how this data is written to the output array `dsp_ex_yout`.



See Also

Blocks
Signal From Workspace | Signal To Workspace

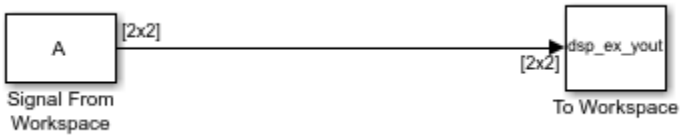
Import 3-D Array From Workspace

Import a 3-D array `A` from the MATLAB® workspace into the Simulink® model using the **Signal From Workspace** block. Use the **To Workspace** block to write the imported data to the MATLAB workspace. In this model, the **To Workspace** block slightly modifies data.

The parameters in the **Signal From Workspace** block are configured as follows:

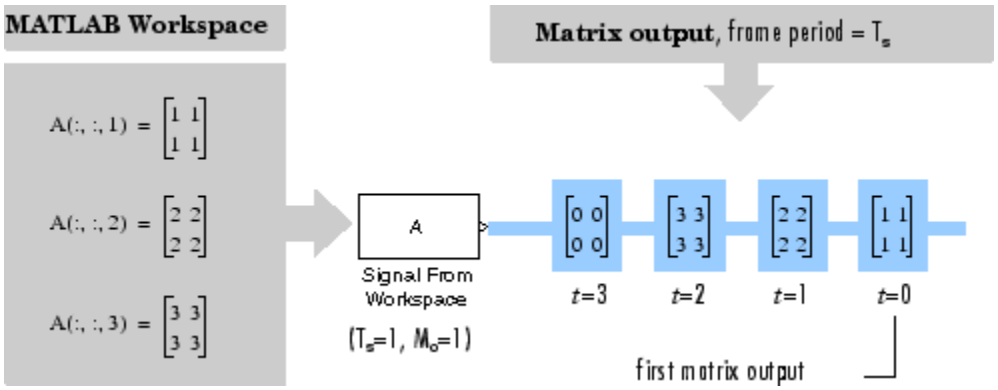
- **Sample time** set to `1`: This parameter determines, *Ts* , the sample period of the output.
- **Samples per frame** set to `1`: Number of samples, *Mo* , to buffer into each output frame. The output frame period is *MoTs*.
- **Form output after final data value by** set to `Setting to zero`: The block generates zero-valued outputs for the duration of the simulation after generating the last frame of the signal.

Open and simulate the model.



The **Signal From Workspace** block imports the signal from the variable `A`, which is a 2-by-2-by-3 array. When you simulate the model, each of the three pages (a 2-by-2 matrix) is output in sequence with period *Ts*. The third page is a matrix of zeros because the **Form output after final data value by parameter** is set to `Setting to zero` in this model. The **To Workspace** block writes the sequence of these matrices to the MATLAB workspace in the variable `dsp_ex_yout`.

This figure shows the signal in the input variable `A` and how this data is written to the output array `dsp_ex_yout`.

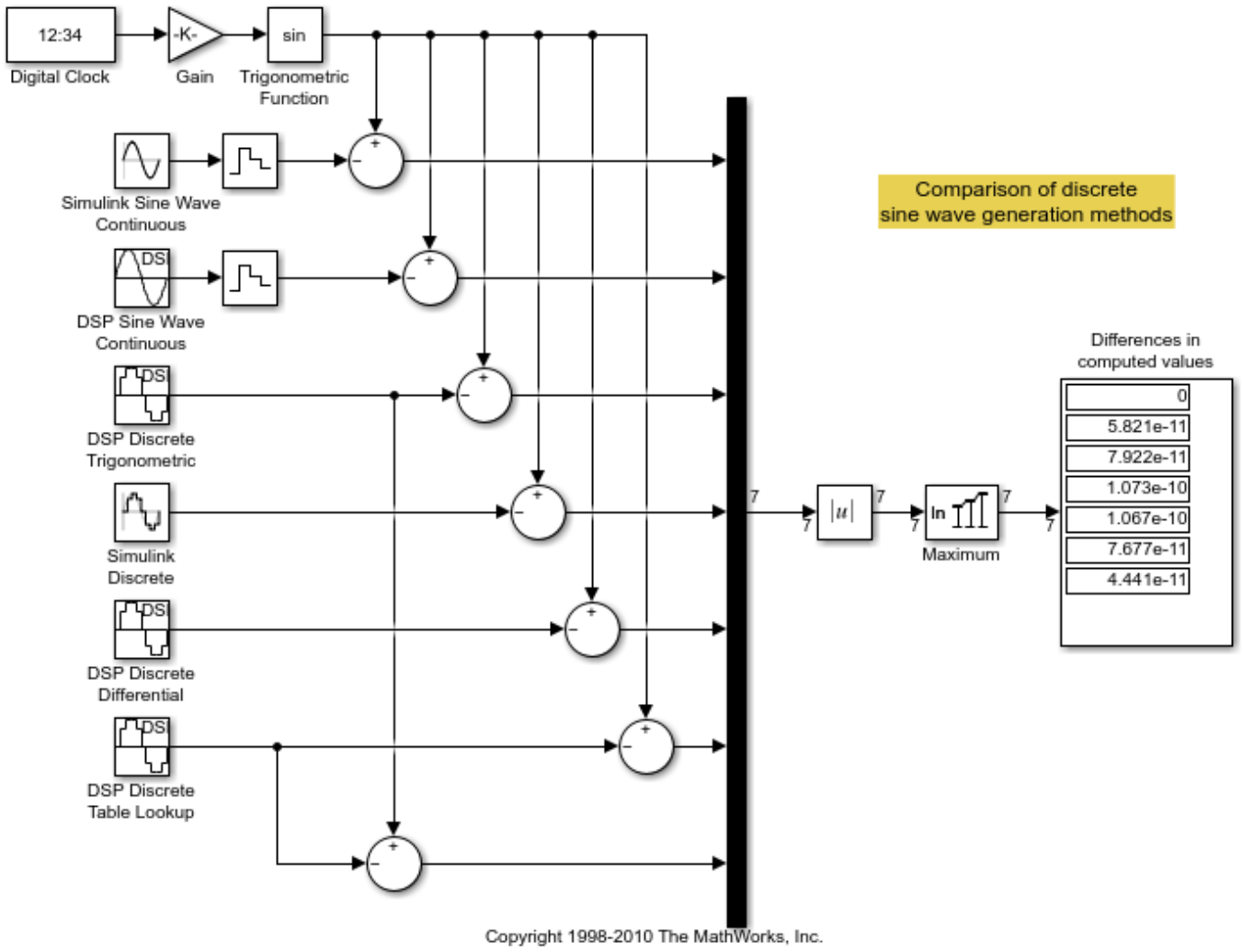


See Also

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.



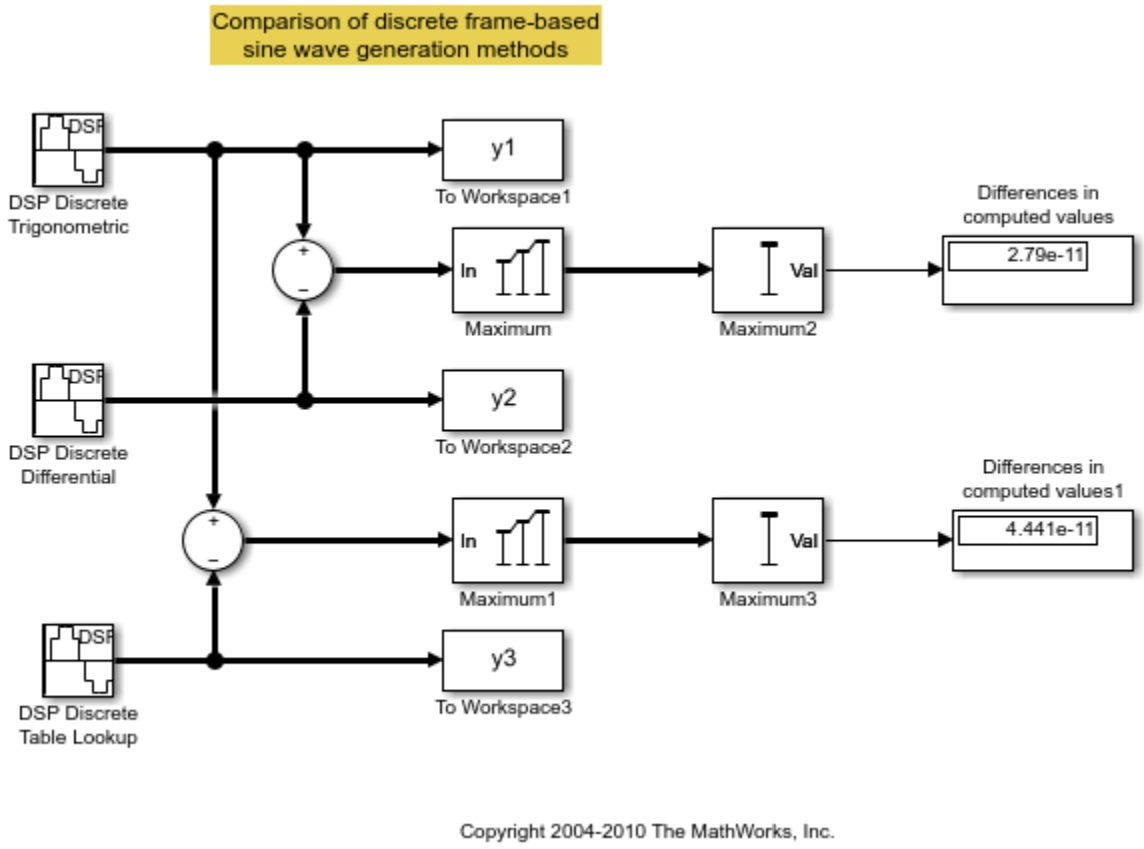
See Also

Blocks

Sine Wave | To Workspace | Display | Maximum | Add | Zero-Order Hold | Gain | Digital Clock | Trigonometric Function | Abs

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

Sine Wave | To Workspace | Display | Maximum | Add

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

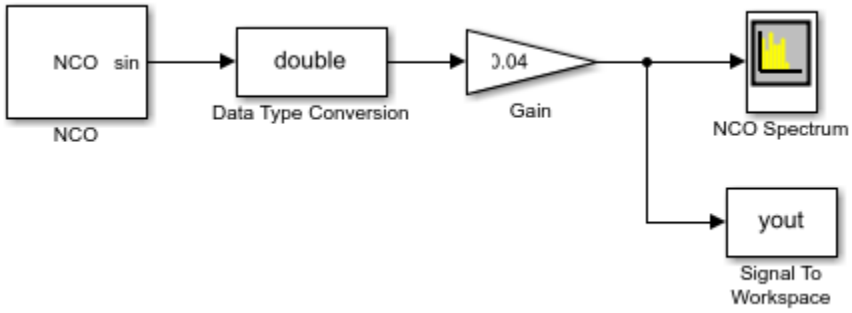
6. Calculate the phase offset, p_o, using the desired phase offset, $p_o desired$:

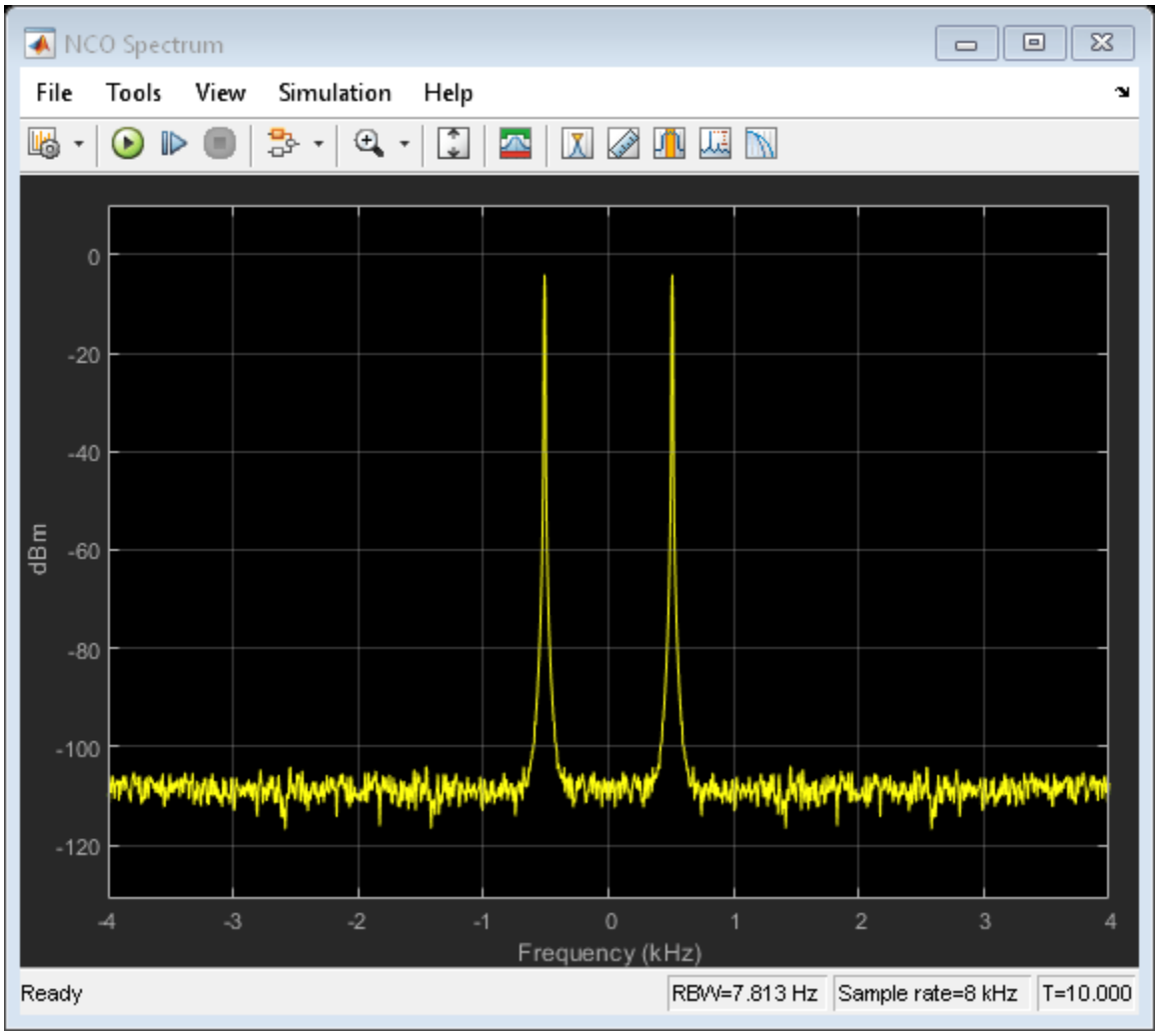
$$p_o = \frac{2^{accumulatorwordlength} \cdot p_o desired}{2\pi}$$

$$p_o = \frac{2^{18} \cdot \frac{\pi}{2}}{2\pi}$$

$$p_o = 65536$$

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.

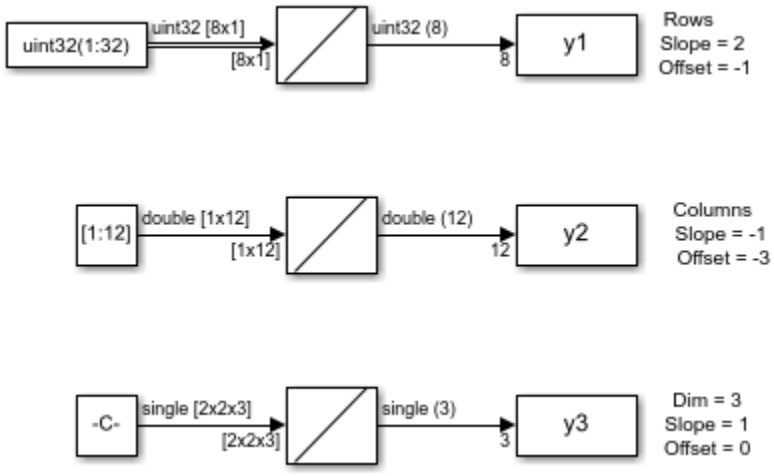
See Also

Blocks

NCO | Signal To Workspace | Spectrum Analyzer | Gain | Data Type Conversion

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

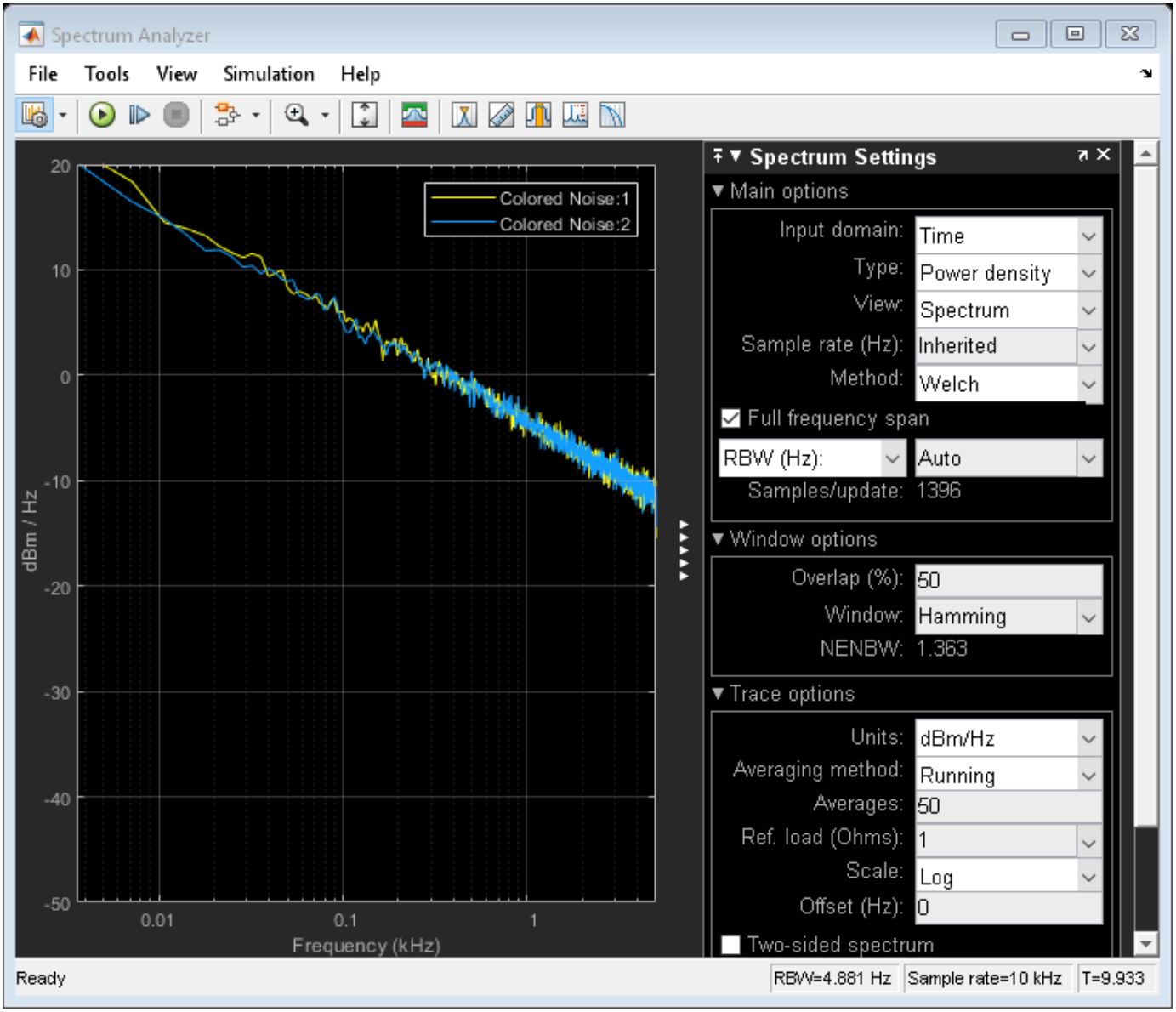
Blocks

Constant Ramp | To Workspace | Signal From Workspace | DSP Constant (Obsolete)

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.



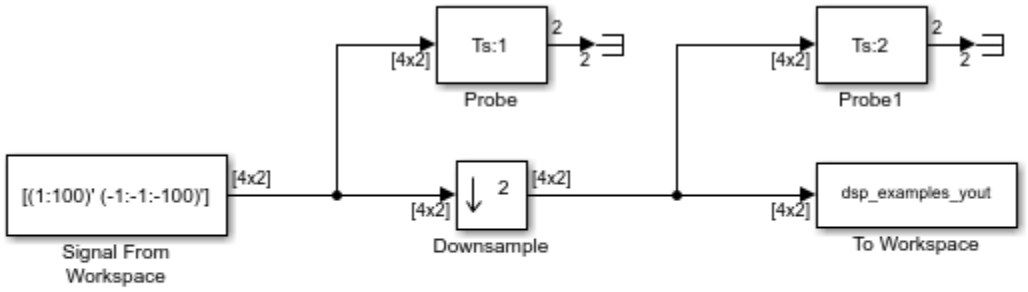
See Also

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 (Simulink) 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
Signal From Workspace | Downsample | Probe | To Workspace

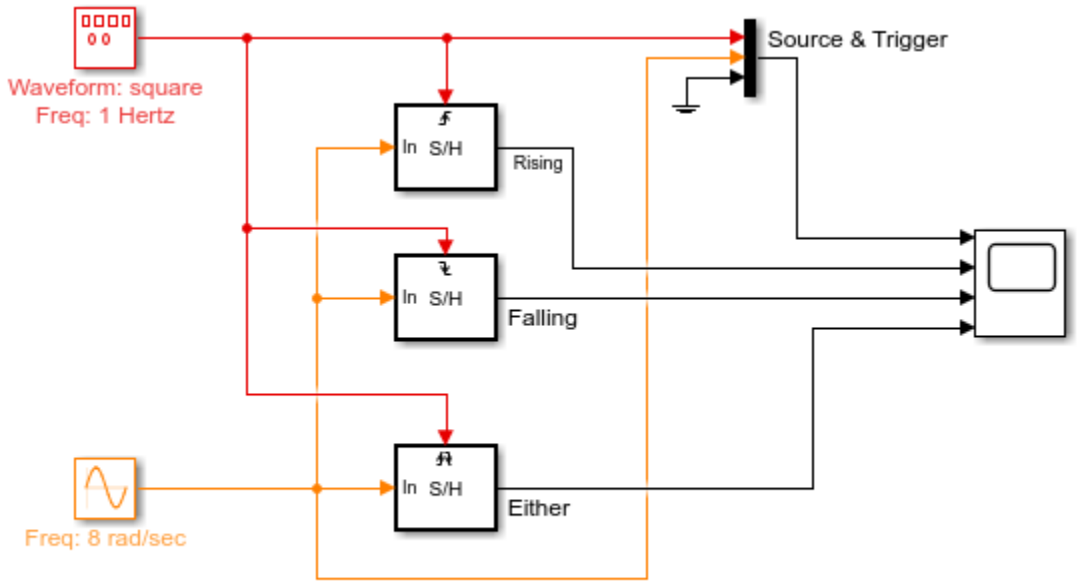
Sample and Hold a Signal

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

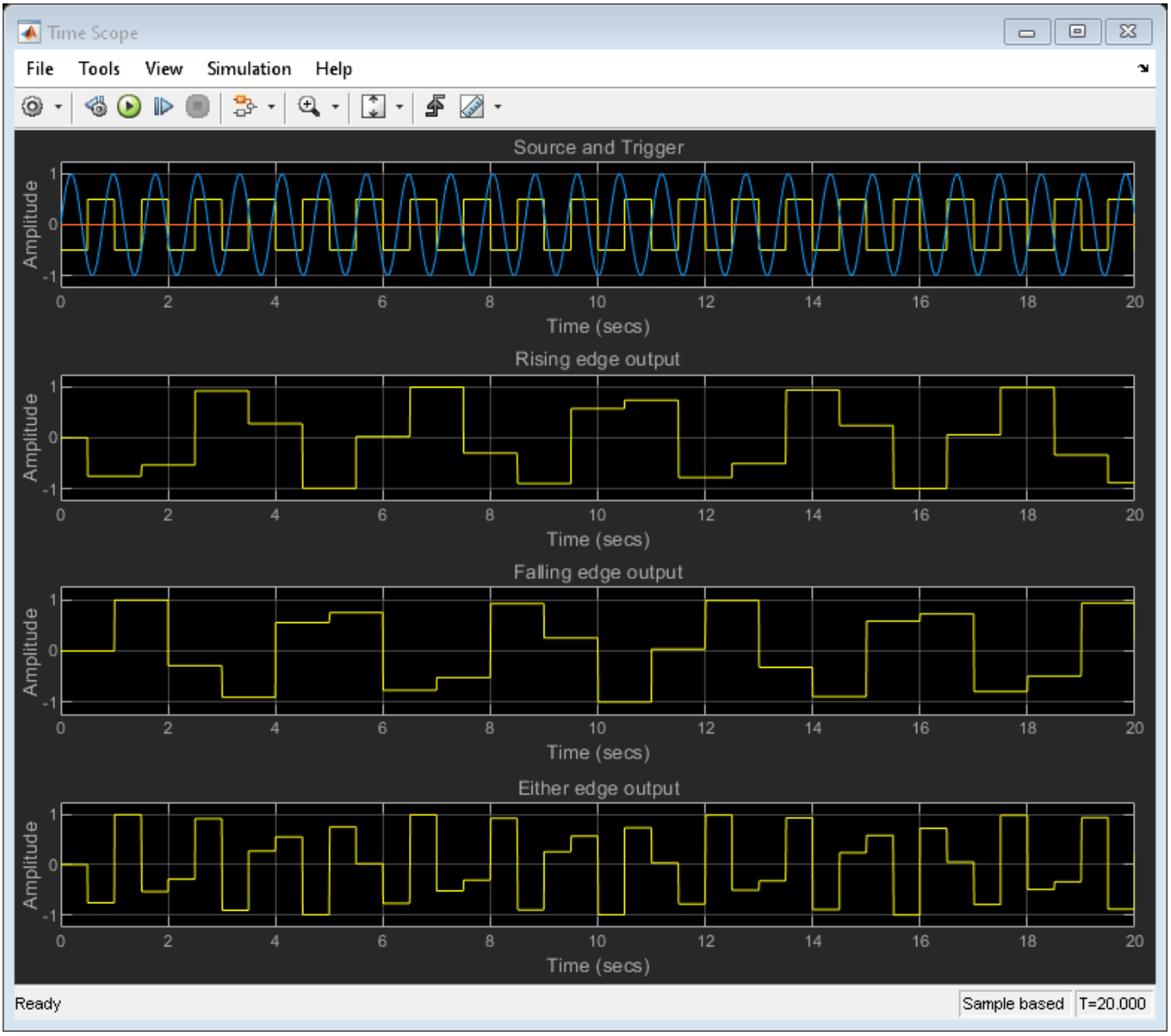
The model contains three **Sample and Hold** blocks which accept the three type of trigger events. The input signal is a continuous-time sine wave with an amplitude of 1 and a frequency of 8 rad/sec. The trigger signal is a square wave with an amplitude of 0.5 and a frequency of 1 Hz.



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

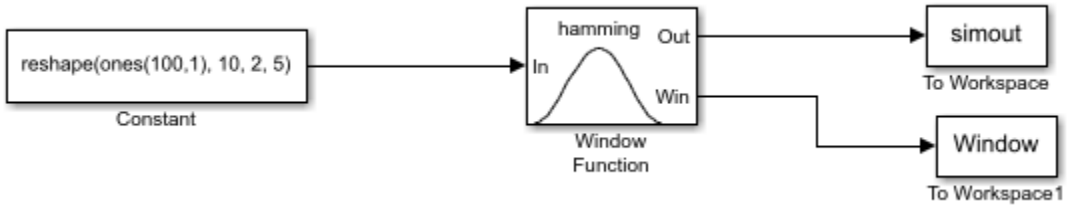
Blocks
Sine Wave | Sample and Hold | Time Scope | Mux | Ground | Signal Generator

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 multiplication, 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

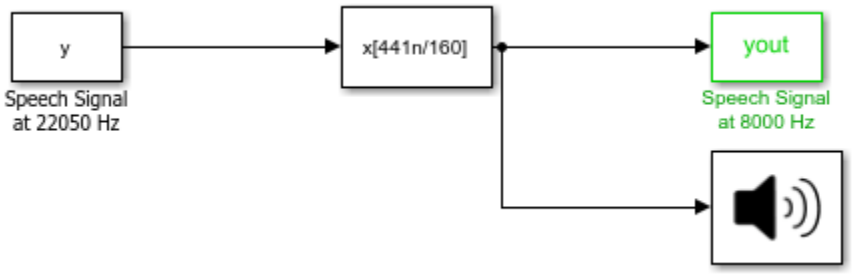
This example shows how to use the **FIR Rate Conversion** block to convert the sample rate of a speech signal. It compares the output/signal behavior in two scenarios -single rate processing and multirate processing. Convert the sample rate of a speech signal from 22.5 kHz to 8 kHz.

Enforce single-rate processing

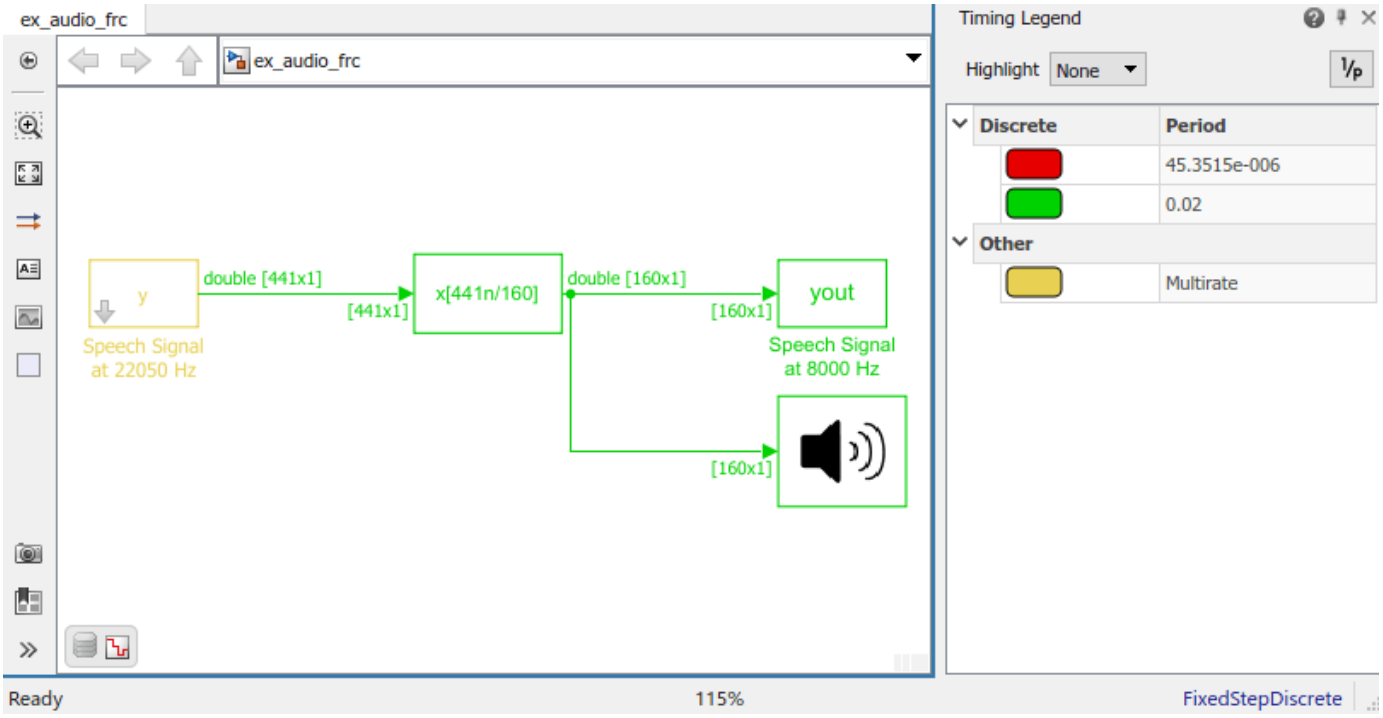
The output signal rate and the input signal rate in Simulink® are the same.

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.



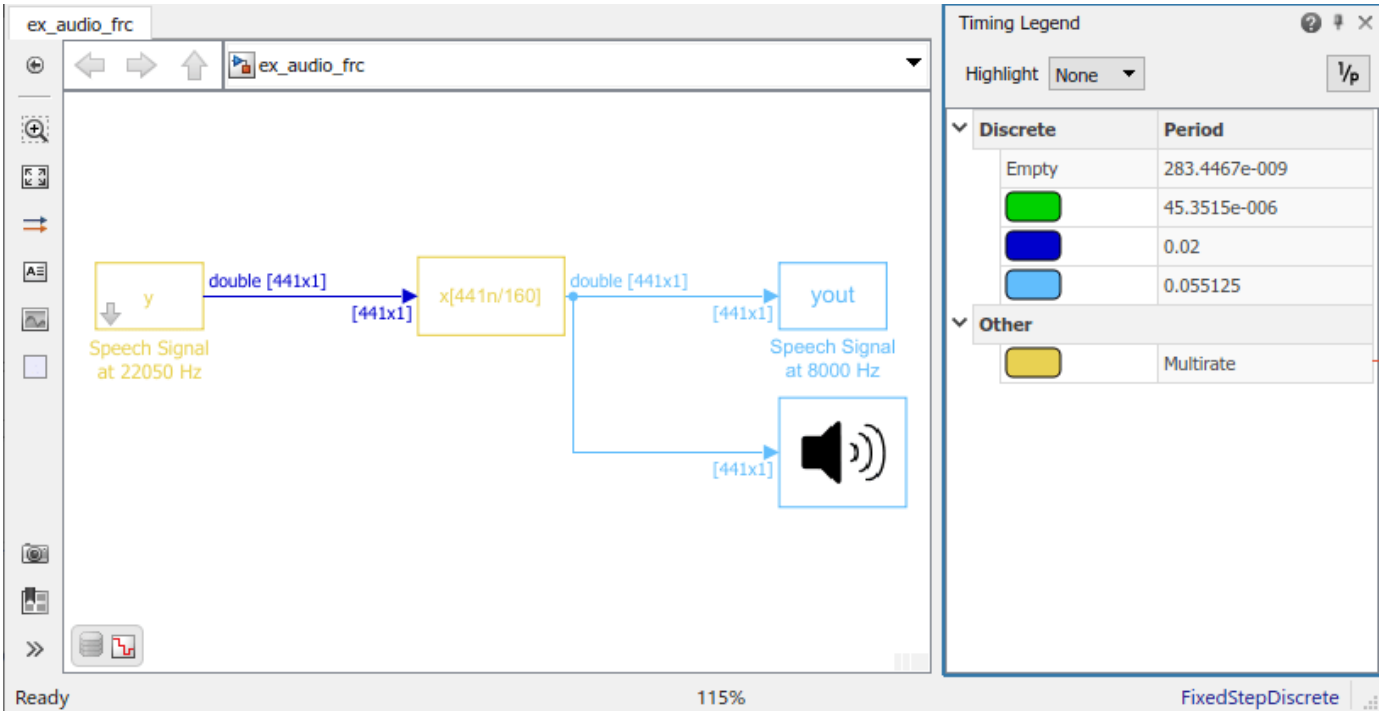
The output and the input signal rate are the same in Simulink. This is shown by the green color-coded signal lines at the input and output of the block.



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

Signal From Workspace | FIR Rate Conversion | Audio Device Writer | Signal To Workspace

Unwrap Signal

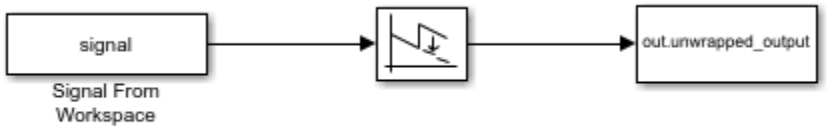
This example shows how to use the Unwrap block to unwrap a 3-by-2-by-3 array that has discontinuity.

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
Unwrap | From Workspace | To Workspace

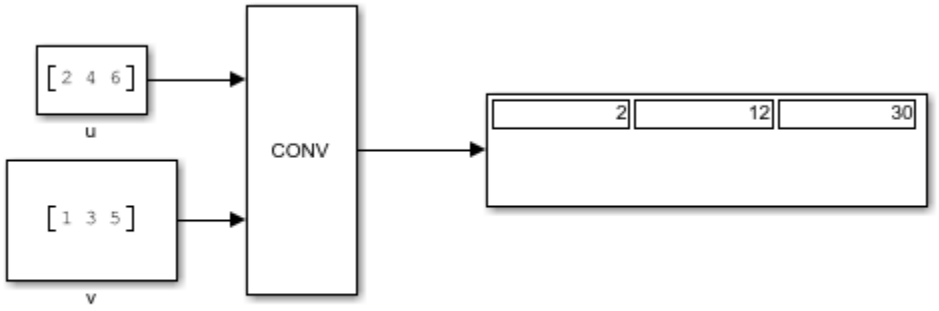
Convolution of Two Inputs

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);
```

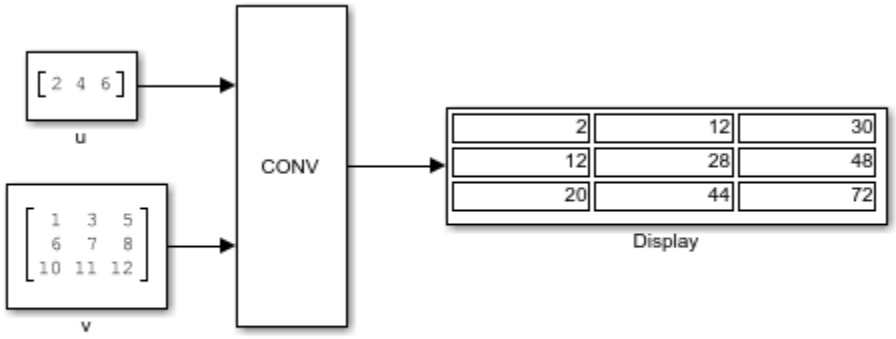


Copyright 2019 The MathWorks, Inc.

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.

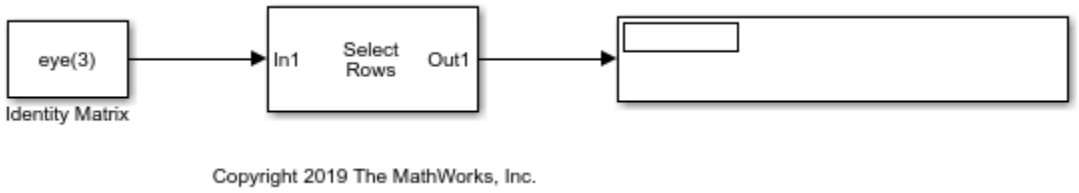
See Also

Blocks
Convolution | Display | Constant

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
```

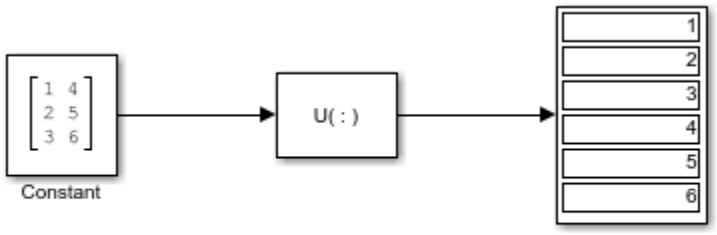
Run the model to verify this output.

See Also

Blocks
Variable Selector | Display | Identity Matrix

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.

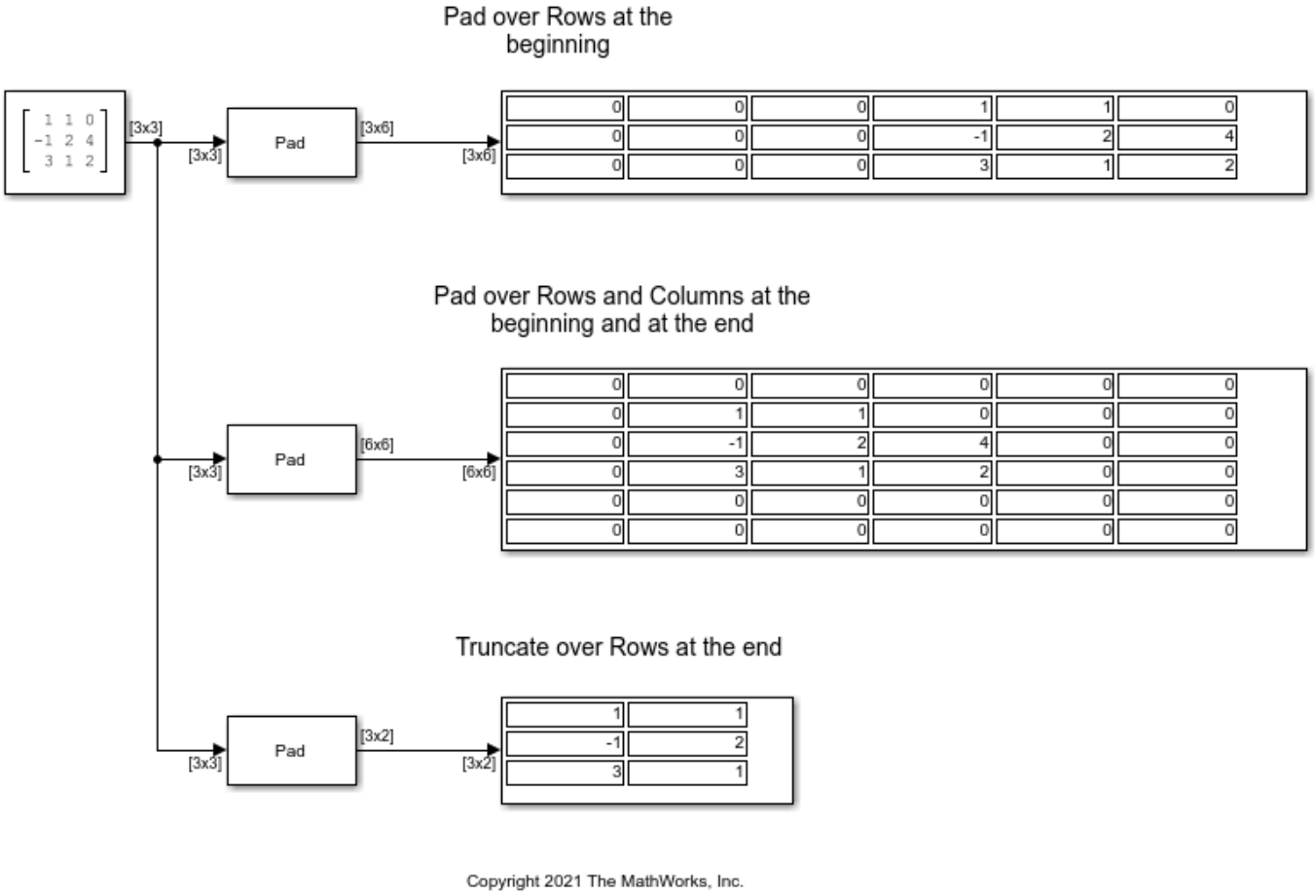
See Also

Blocks
Convert 2-D to 1-D | Display | Constant

Pad or Truncate Matrix with Constant Values

This example shows how to pad or truncate constant values from a matrix using the Pad block.

Open and run the model.



Copyright 2021 The MathWorks, Inc.

Pad over Rows at the Beginning

When you specify **Pad over** to `Rows` and **Pad signal at** to `Beginning`, the block pads the matrix rows with the value specified in **Pad value**. The number of values padded to each row is determined by the **Row size** parameter. In the first Pad block, the **Row size** parameter is set to 6.

Pad over Rows and Columns at the Beginning and at the End

When you specify **Pad over** to `Columns and rows` and **Pad signal at** to `Beginning and end`, the block pads the matrix rows and columns with the value specified in **Pad value**. The number of values padded to each row and column is determined by the **Row size** and the **Column size** parameters. In the second Pad block, the **Row size** and the **Column size** parameters are set to 6, respectively.

Truncate over Rows at the End

You can also truncate the matrix by specifying the size of a specific dimension in the output to be less than that of the input. In the third block, **Pad over** is set to to `Rows` and the output **Row size**

parameter is set to 2. With this setting, the block truncates the 3-by-3 input matrix over the Rows dimension and makes it a 3-by-2 matrix.

See Also

Blocks
Pad | Constant | Display

Extract the Phase of Sine Wave

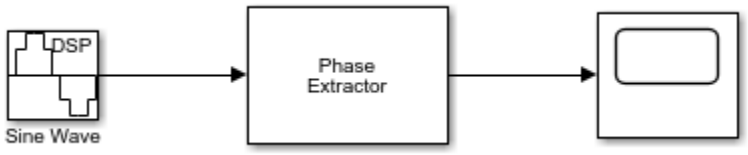
Extract the phase of a sine wave signal using the Phase Extractor block.

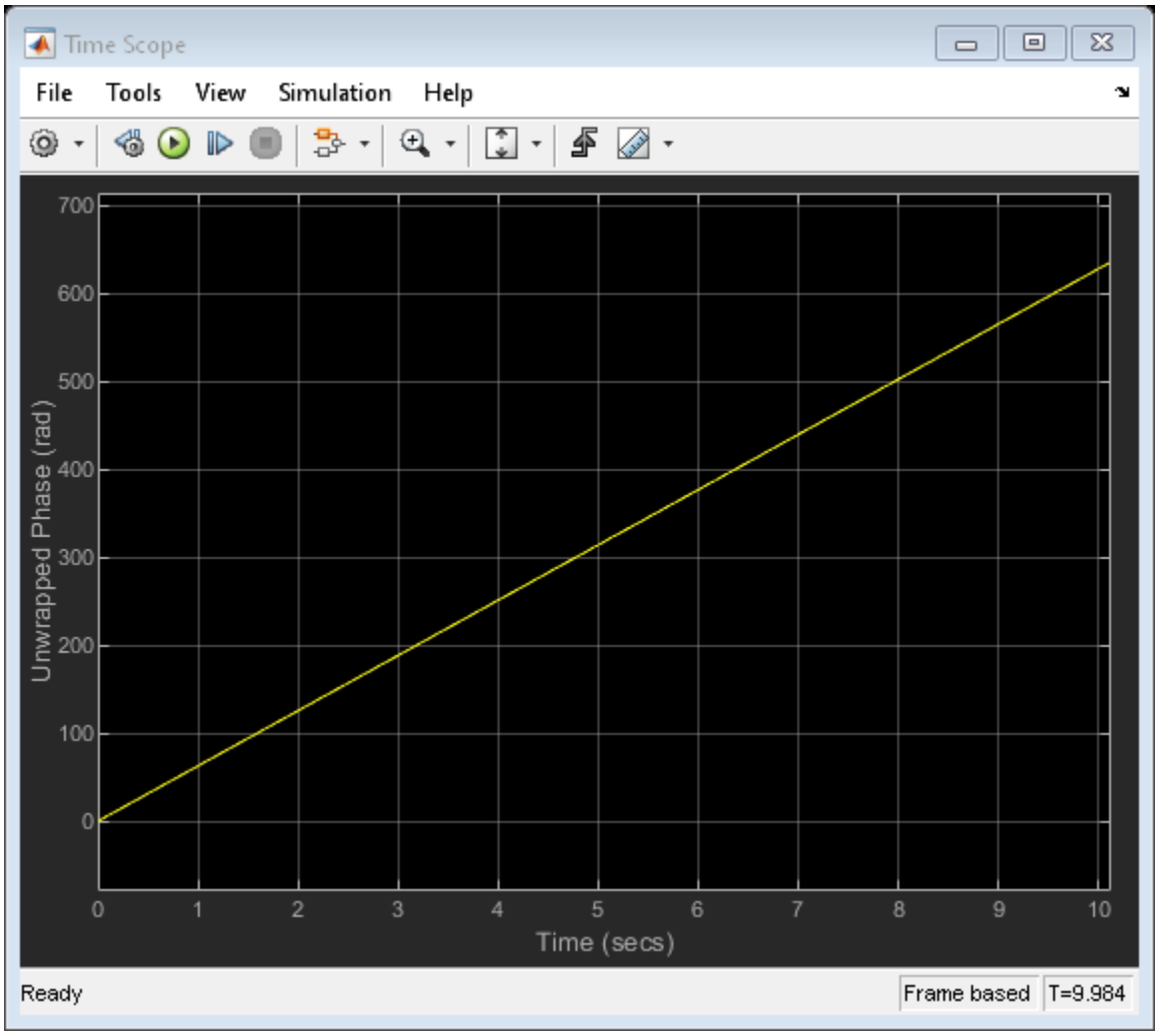
Genetate the input sine wave using the Sine Wave block. Set the Sine Wave block parameters to the following:

- **Frequency** set to 10 Hz
- **Sample mode** set to Discrete
- **Output complexity** set to Complex
- **Sample time** set to 1/1000
- **Sample per frame** set to 128

In the Phase Extractor block, do not select the **Unwrap phase only within the frame** parameter.

Open and run the model. The Time Scope block displays the extracted phase.



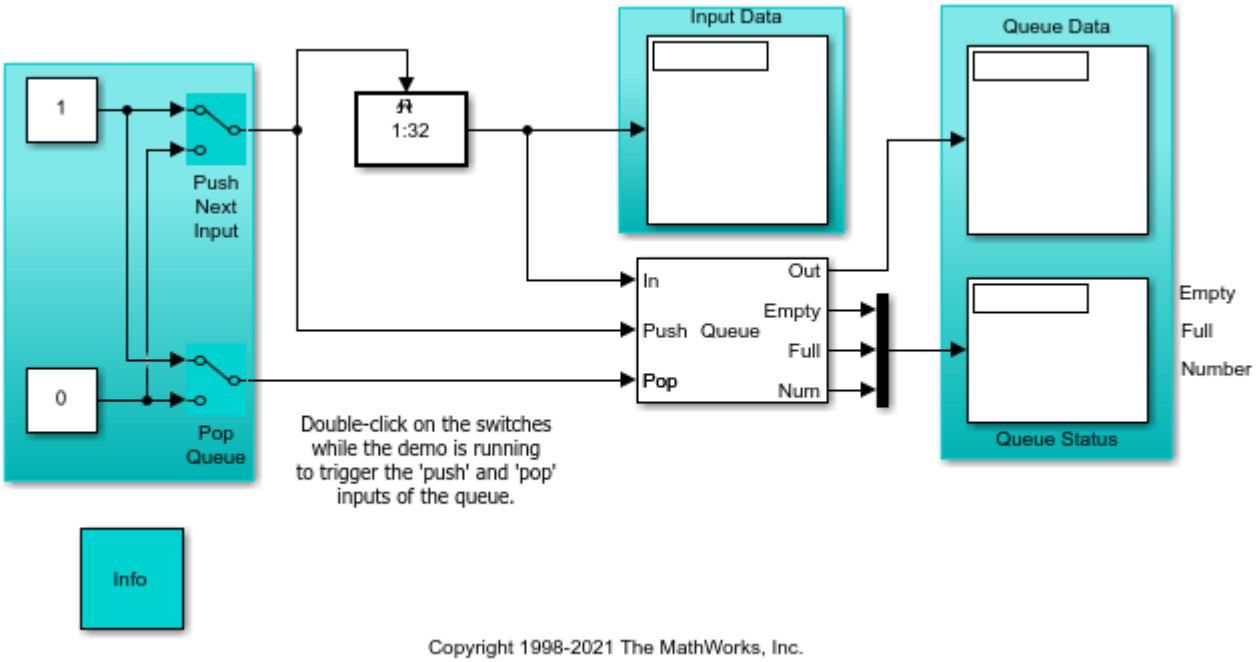


See Also

Sine Wave | Phase Extractor | Time Scope

Queues

This example shows how to push and pop elements from a queue using a Queue block with a system of selection switches.



Applications

Queues have many practical applications. They are used in modeling to study communication traffic over limited bandwidth channels and in any application where there is a limited resource serving an unknown number of clients. A simple example is people lining up in front of a teller at a bank.

Queues are used in messaging systems to provide reliable delivery. In multitasking systems, they are used to buffer requests for limited system resources.

Exploring the Example

While the model is running, toggle the 'Push Next Input' switch to update the signal from the Triggered Signal From Workspace block and to trigger the Push port of the Queue block. The signal is pushed into the block's FIFO register, and is shown on the 'Input Data' display. Next, toggle the 'Pop Queue' switch to trigger the Queue block's Pop port, which causes the block to output from its FIFO register. The output signal is shown on the 'Queue Data' display.

The 'Queue Status' display shows the state of the Queue's FIFO. The Queue block is configured to store a maximum of three signal samples. Try changing this value in the block's Register size

parameter and observe the behavior of the Queue block's Empty and Full states as signals are input and output from the FIFO.

See Also

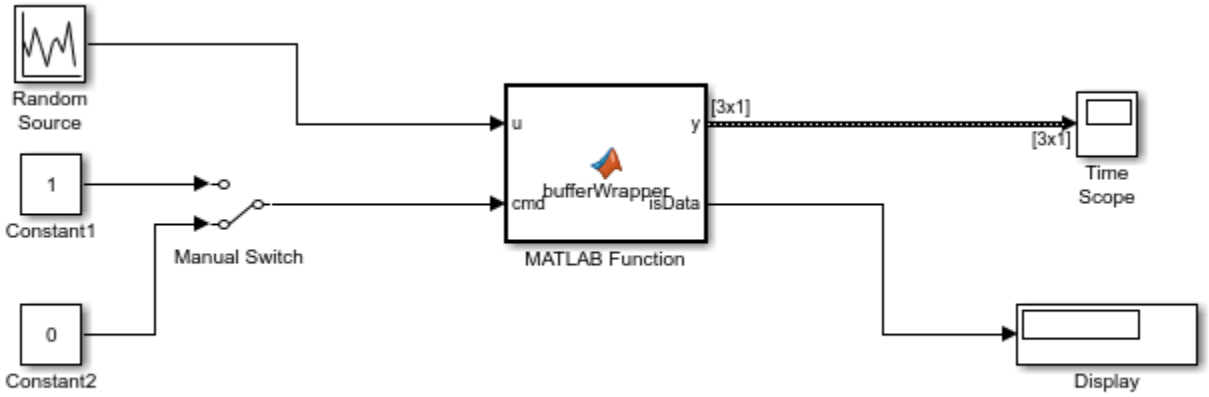
Blocks
Queue

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 14-2
- “Why Does the dsp.AsyncBuffer Object Error When You Call read Before write?” on page 14-7
- “Buffering Input with Overlap” on page 14-9

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.

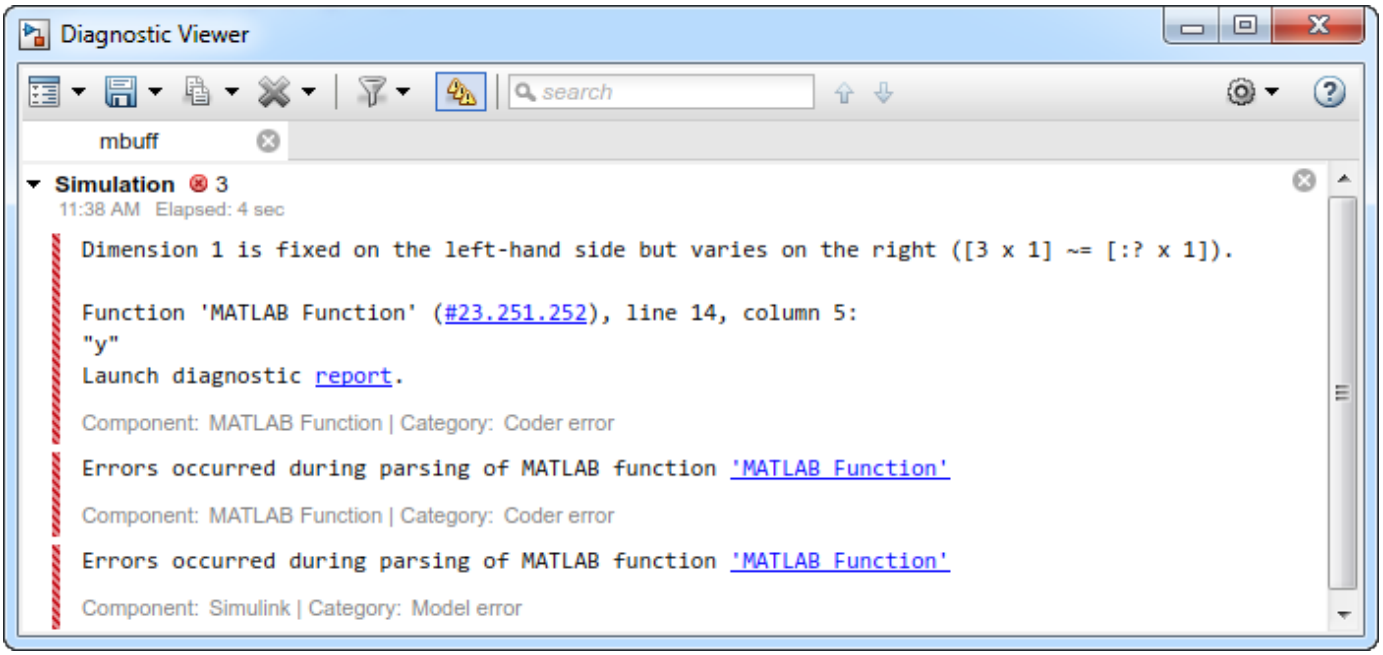
Here is the `bufferWrapper` function that contains the algorithm inside the MATLAB Function block. When input on the `cmd` port is `1`, the `dsp.AsyncBuffer` object writes the data input, `u`, to the buffer. When input on the `cmd` port is `0`, the object reads data from the buffer.

```
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`.

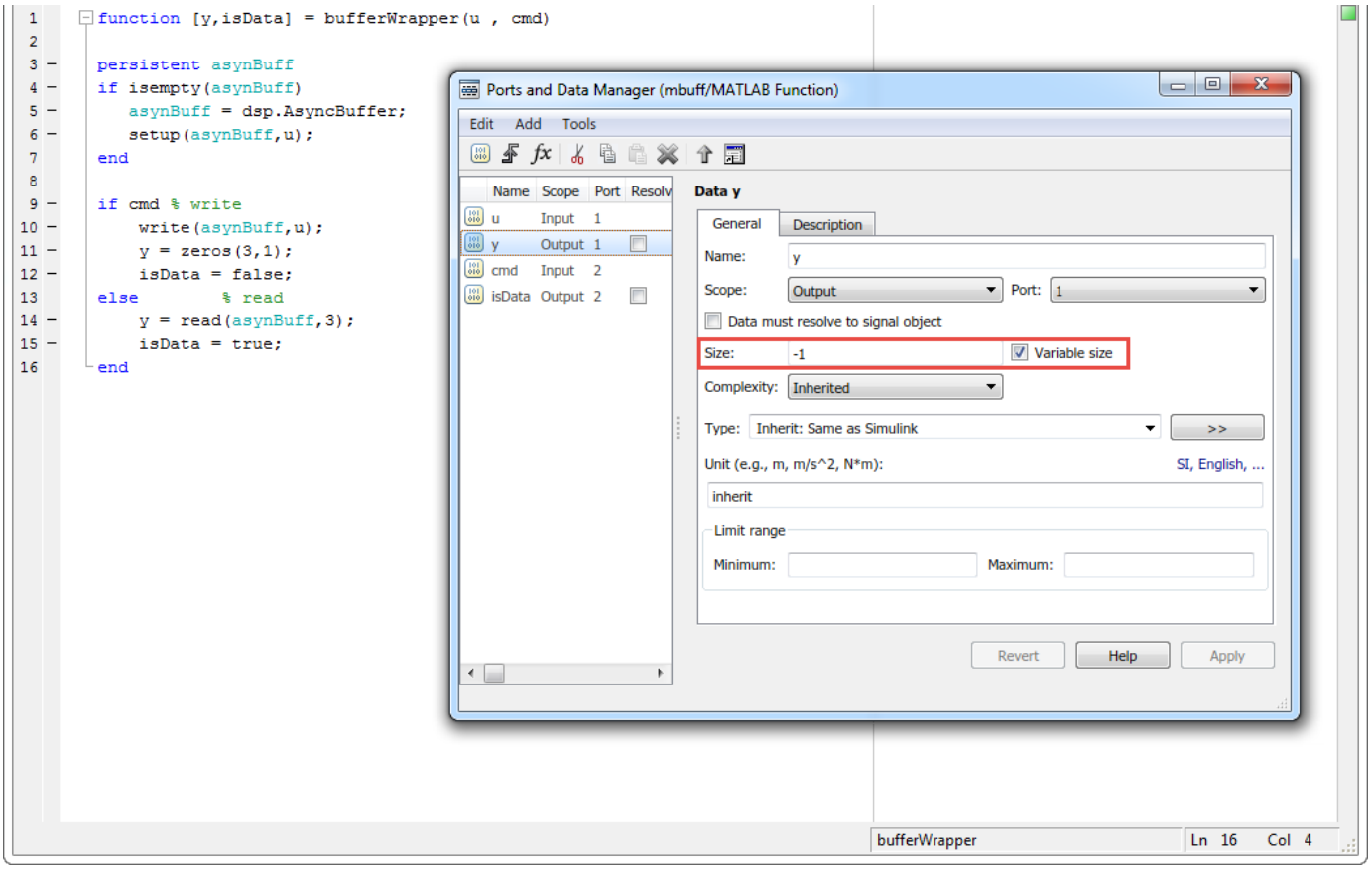
During the write operation, the first output, `y`, is `zeros(3,1)` and the second output, `isData`, is 0. During the read operation, `y` is the data in the buffer and `isData` is 1.

Run the model and the following error occurs.

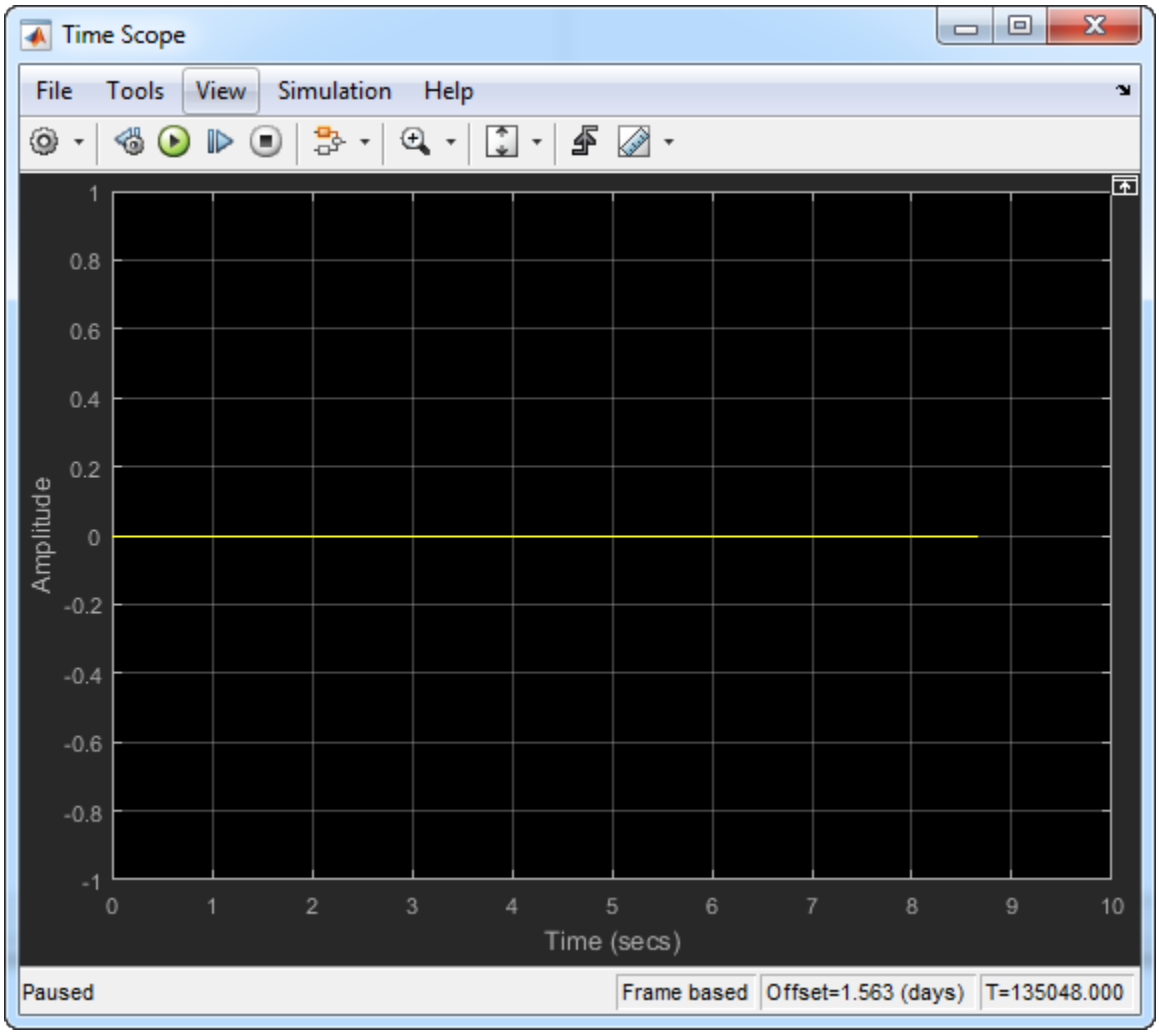


The output of `read(asyncBuff,3)` on line 14 is variable sized. The output is variable sized because the size of the signal output by the `read` function depends on the input arguments to `read`. To resolve this error, specify `y` as a variable-size signal.

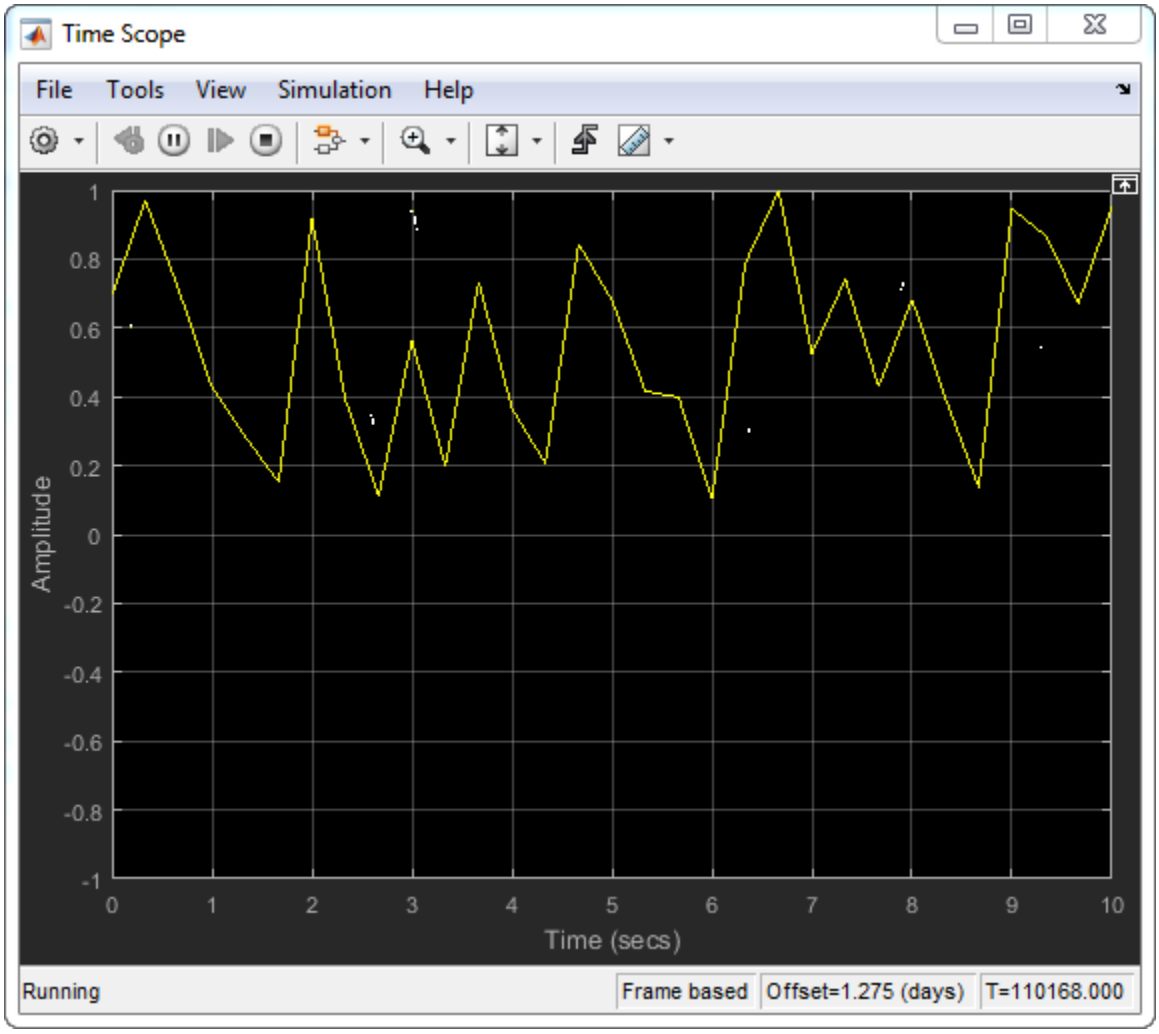
1. In the MATLAB Function block Editor, click **Edit Data** to open the Ports and Data Manager.
2. For the output `y`, select the **Variable size** check box.
3. Click **Apply**.



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 14-7
- “High Resolution Spectral Analysis in MATLAB” on page 4-16

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.
```

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 in MATLAB” on page 4-16

More About

- “Why Does Reading Data from the dsp.AsyncBuffer Object Give a Dimension Mismatch Error in the MATLAB Function Block?” on page 14-2

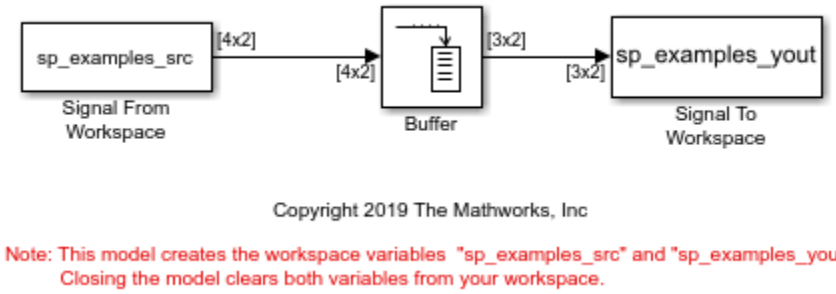
Buffering Input with Overlap

Buffering Two-Channel Input with Overlap

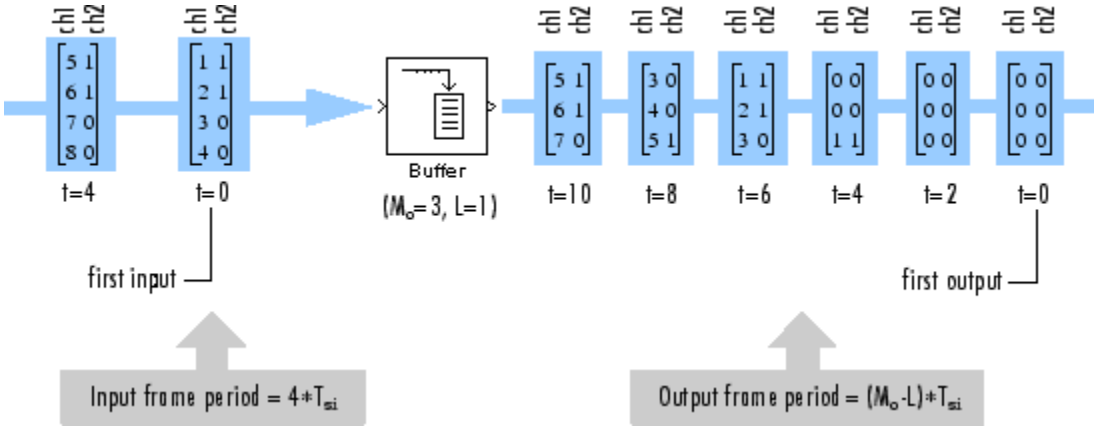
In the `ex_buffer_tut4` model, the Buffer block uses a one-sample overlap and rebuffers a signal with the frame size of 4 into a signal with the frame size of 3.

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.



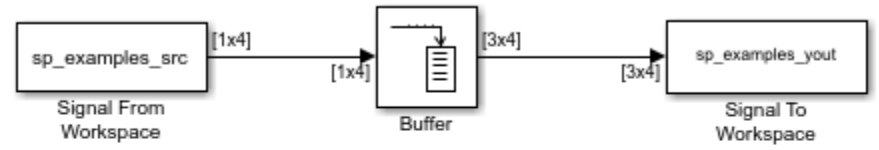
The output is delayed by eight samples. This latency occurs because of the parameter settings chosen in this model, and because the model is running in Simulink in the multitasking mode. The first eight output samples therefore adopt the value specified in the **Initial conditions** parameter, which 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.

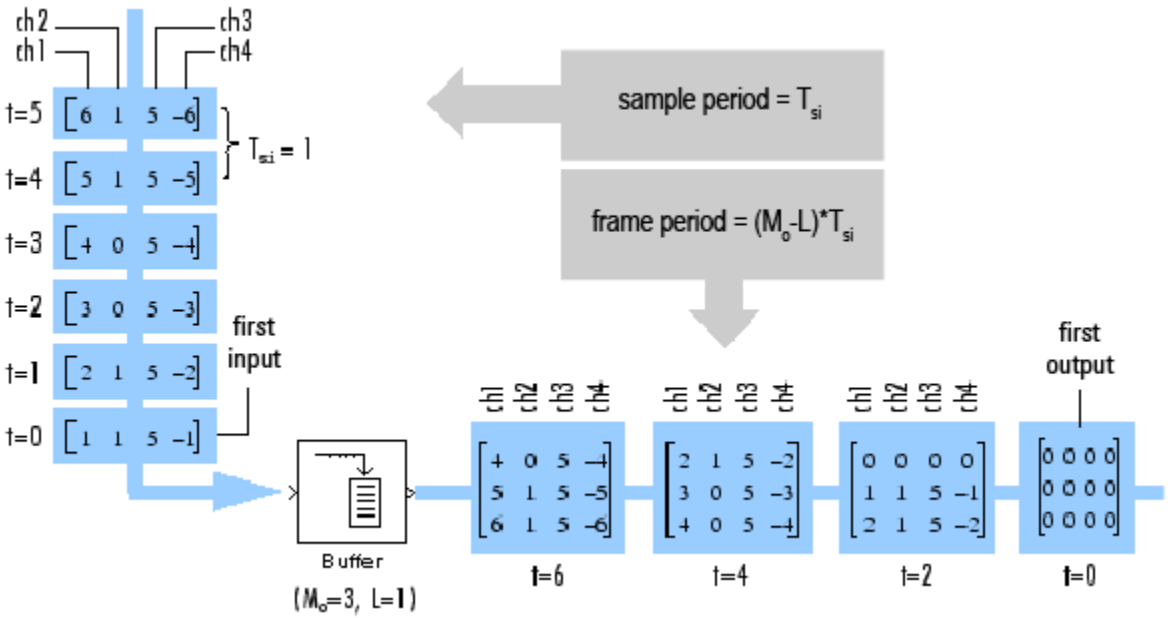
```
four_channel_model = "ex_buffer_tut3";
open_system(four_channel_model);
sim(four_channel_model);
```



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 in Simulink” on page 15-2
- “Filter input with Butterworth Filter in Simulink” on page 15-9
- “SSB Modulation” on page 15-10
- “Wavelet Reconstruction and Noise Reduction” on page 15-15

Synthesize and Channelize Audio in Simulink

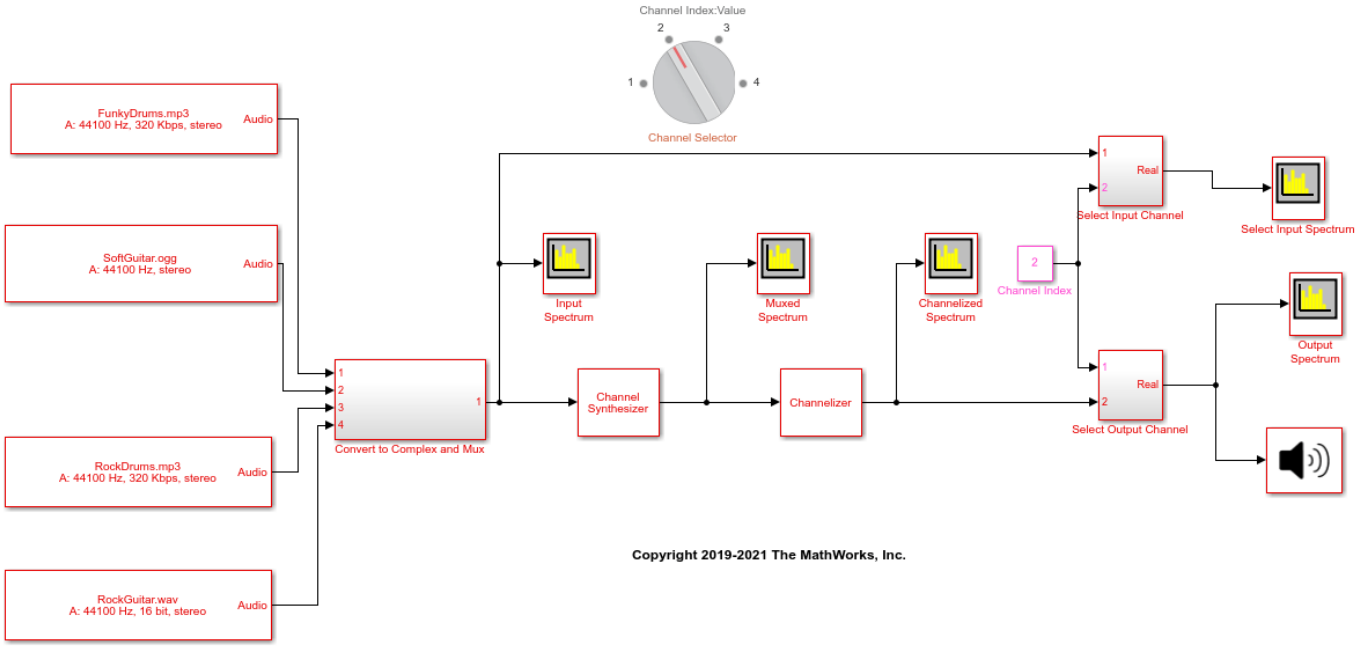
Synthesize a series of four stereo signals into a broadband signal by using the Channel Synthesizer block. At the receiving end of the model, split this broadband signal back into the individual narrowband signals by using the Channelizer block.

The inputs to the model are four stereo signals.

- `FunkyDrums.mp3`
- `SoftGuitar.ogg`
- `RockDrums.mp3`
- `RockGuitar.wav`

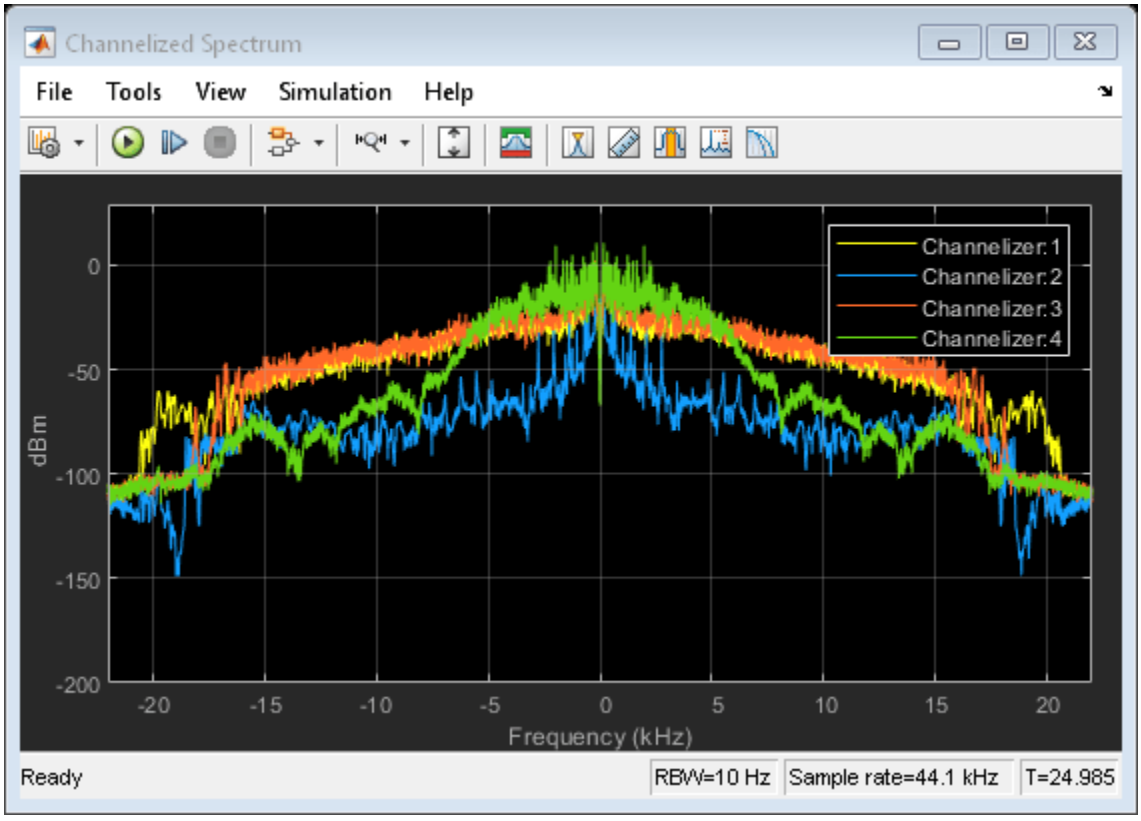
Each signal has a size of 1024-by-2 samples. The two columns (channels) represent the left channel and the right channel of the stereo signal. To store the stereo channels, the 'Convert to Complex and Mux' block converts each signal into complex signals. It then multiplexes the complex signals by a Matrix 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.

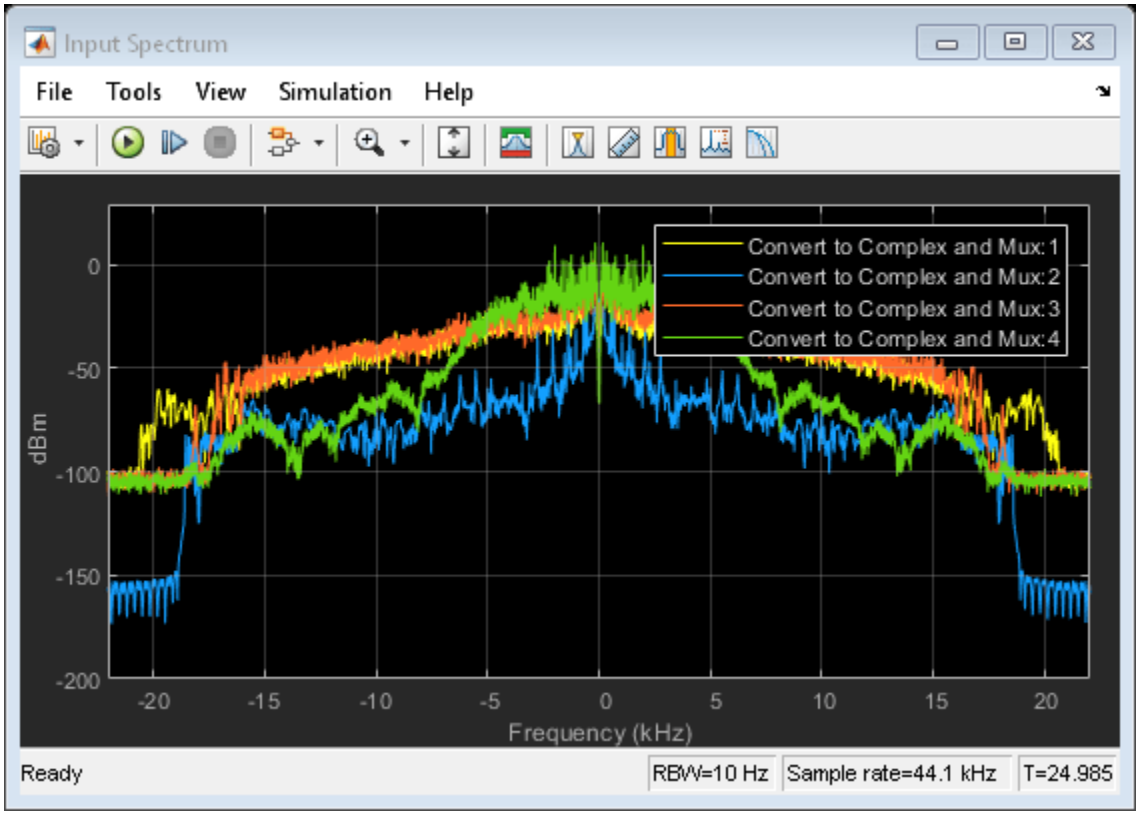
Open the model by clicking on the 'Open Model' button on the example in the MATLAB® help browser.

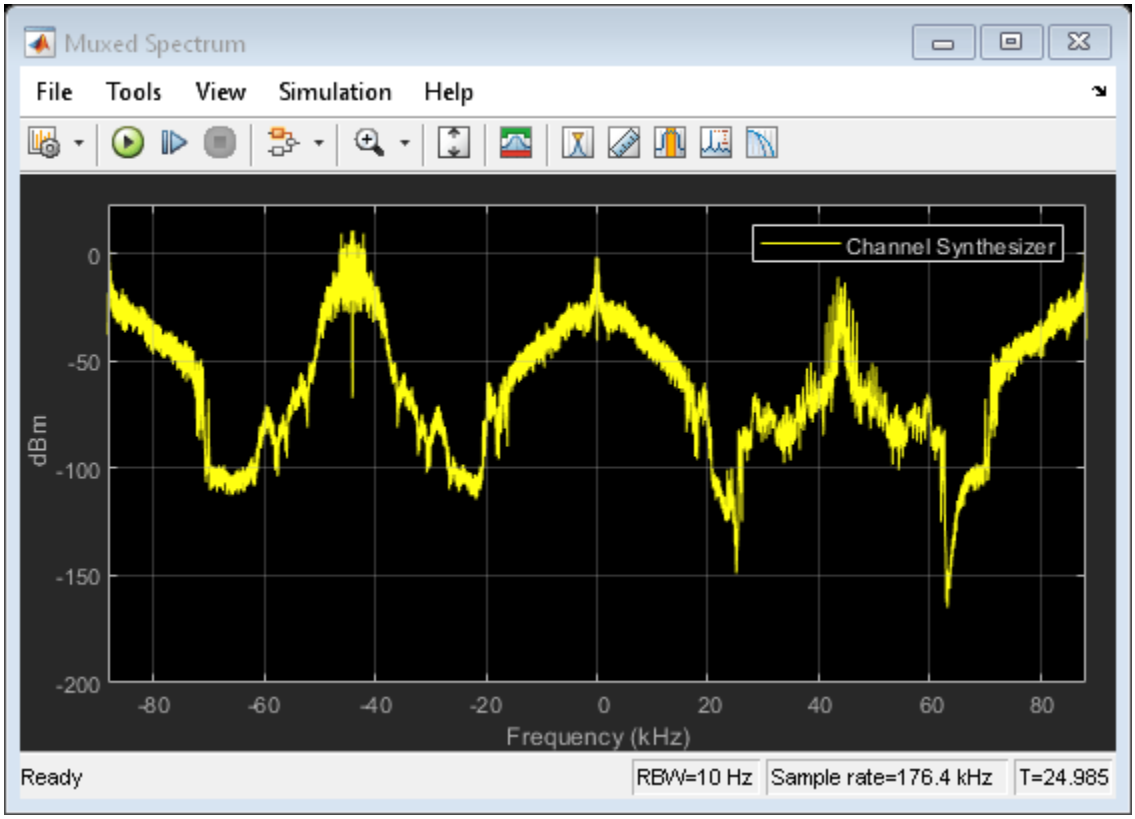


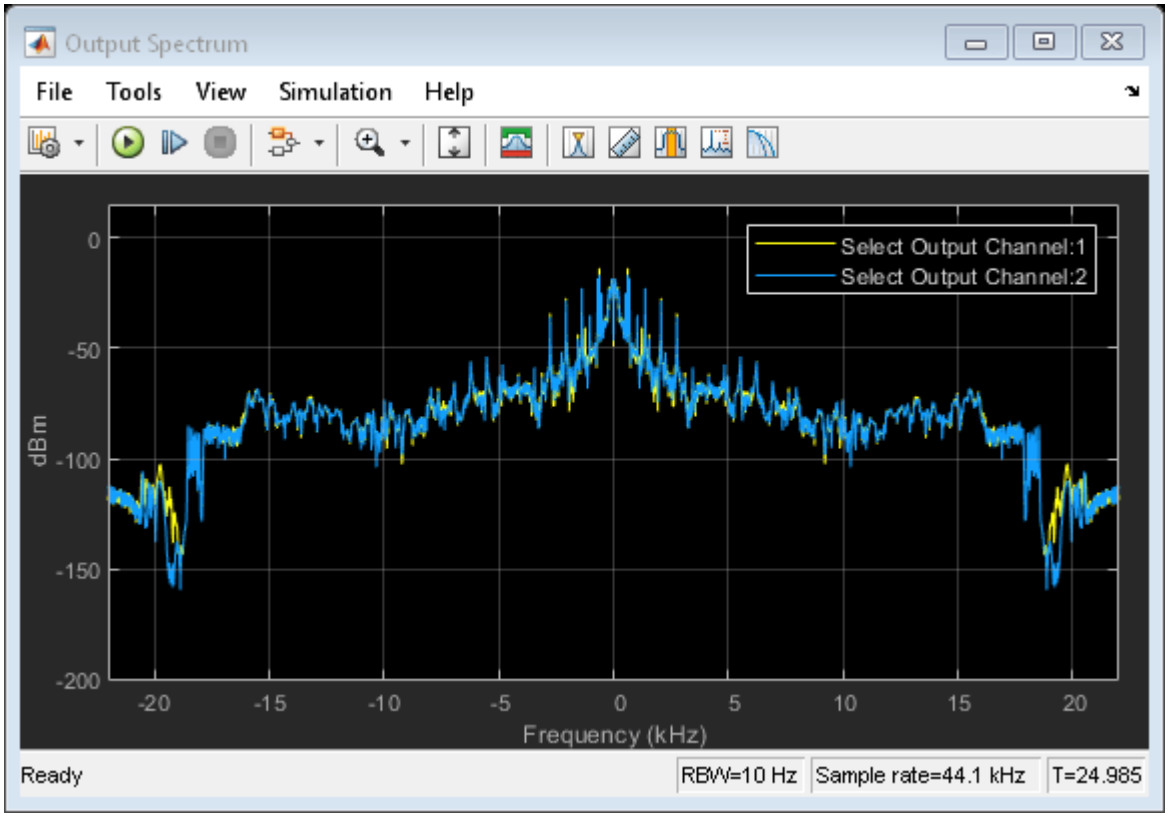
Select the audio signal you want to listen to and play this signal using the Audio Device Writer block.

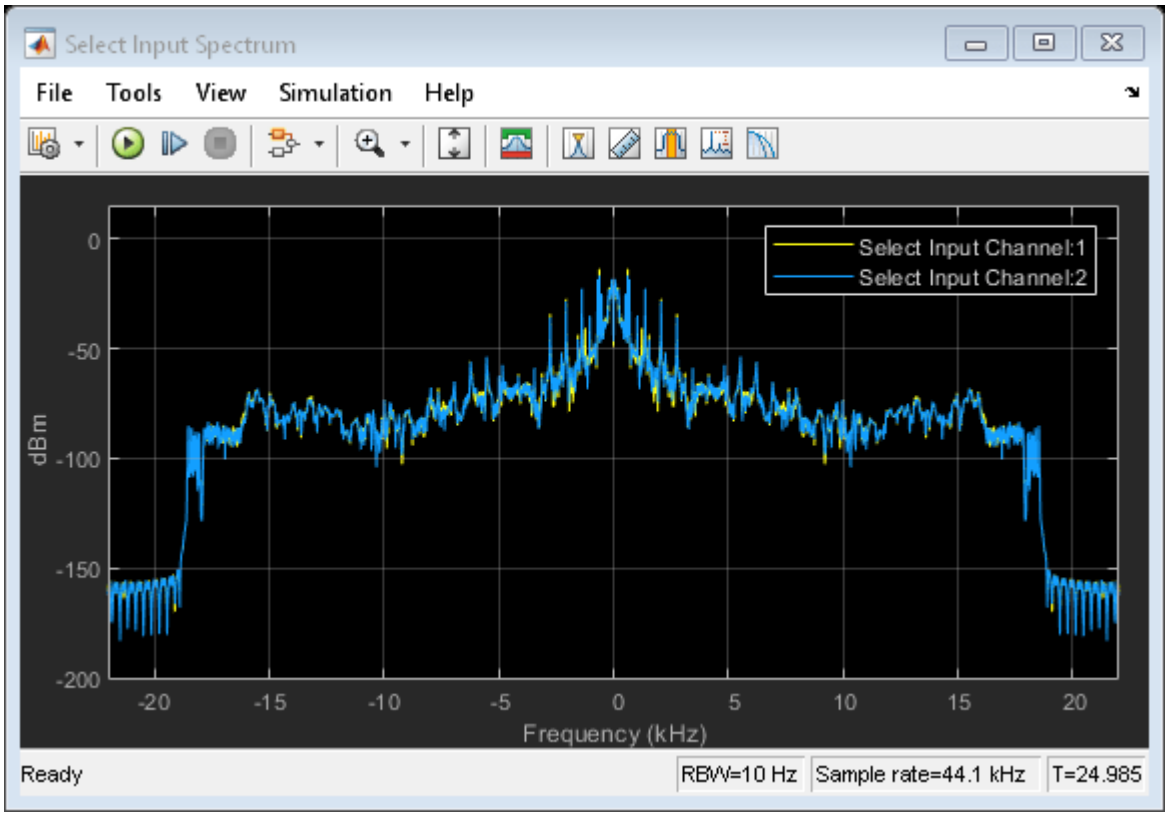
Run the model. View the spectra of the input, muxed, and output signals.


Channelized Spectrum
File Tools View Simulation Help
Channelizer: 1
Channelizer: 2
Channelizer: 3
Channelizer: 4
dBm
0
-50
-100
-150
-200
-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
-200
-20 -15 -10 -5 0 5 10 15 20
Frequency (kHz)
Ready
RBW=10 Hz
Sample rate=44.1 kHz
T=24.985

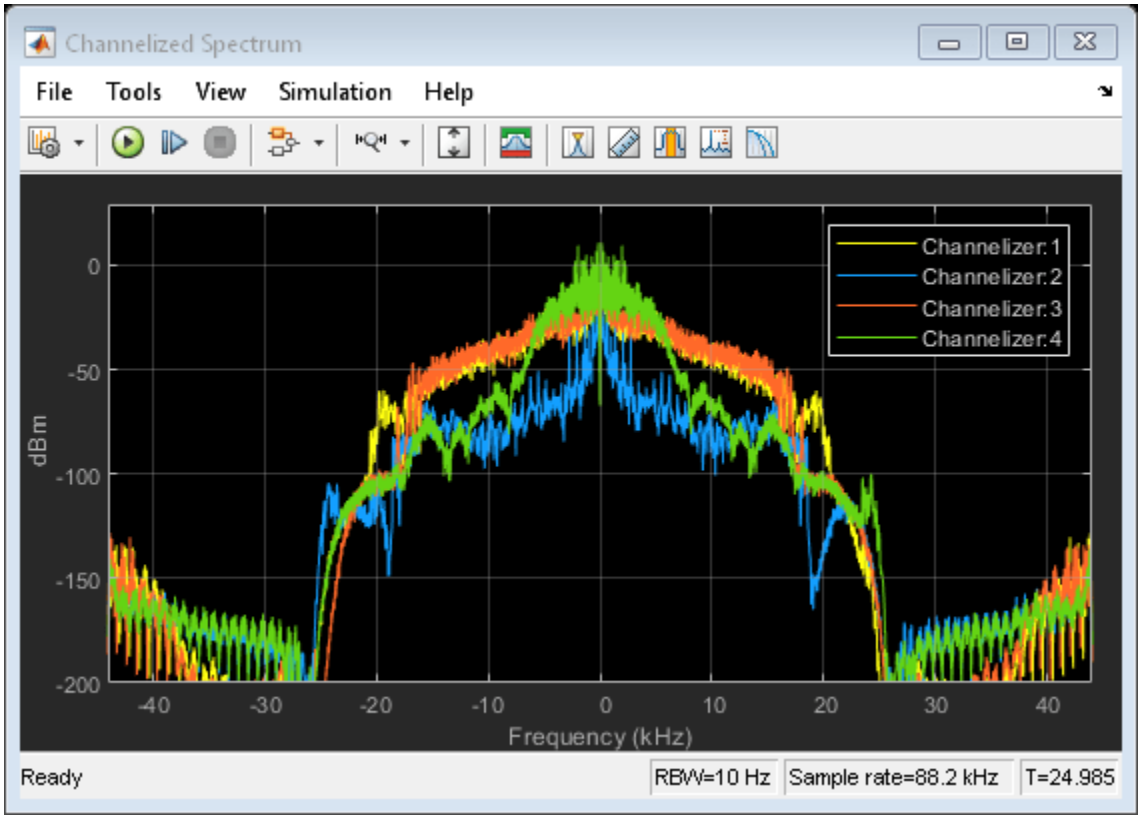

Muxed Spectrum
File Tools View Simulation Help
Channel Synthesizer
dBm
0
-50
-100
-150
-200
-80 -60 -40 -20 0 20 40 60 80
Frequency (kHz)
Ready
RBW=10 Hz
Sample rate=176.4 kHz
T=24.985

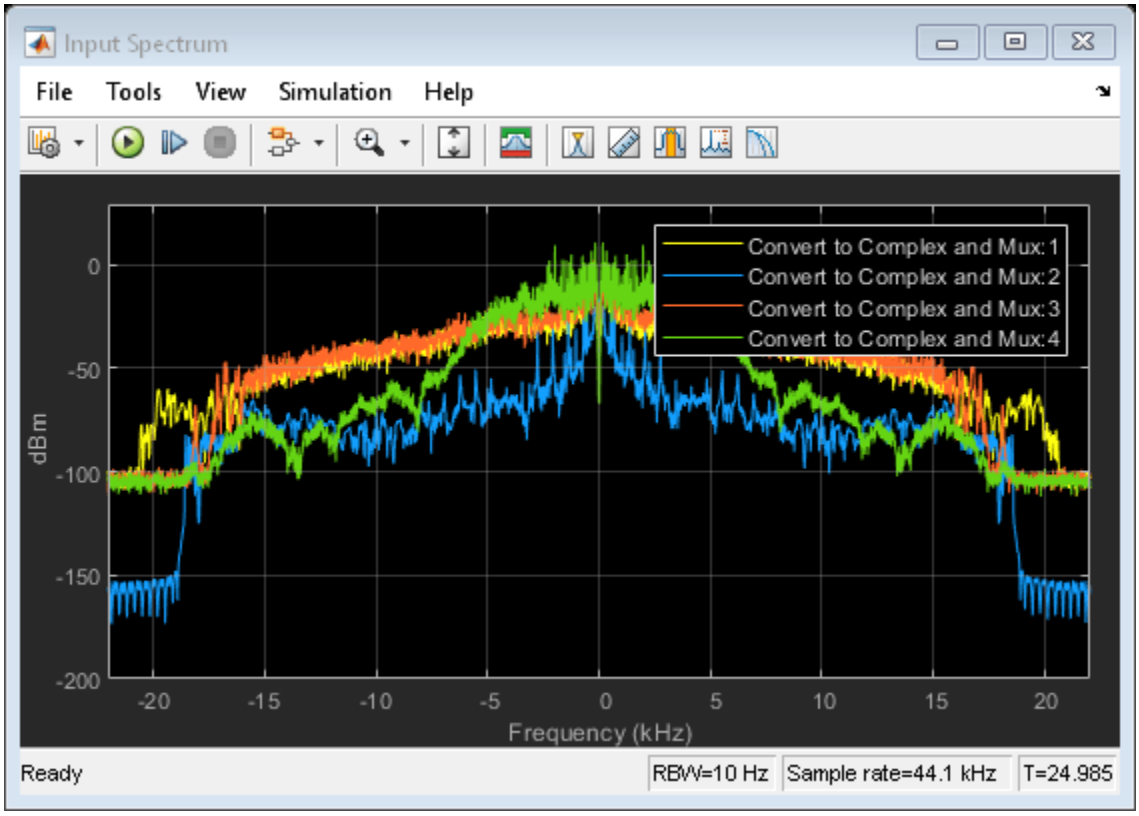

Output Spectrum
File Tools View Simulation Help
Select Output Channel:1
Select Output Channel:2
dBm
0
-50
-100
-150
-200
-20 -15 -10 -5 0 5 10 15 20
Frequency (kHz)
Ready
RBW=10 Hz
Sample rate=44.1 kHz
T=24.985

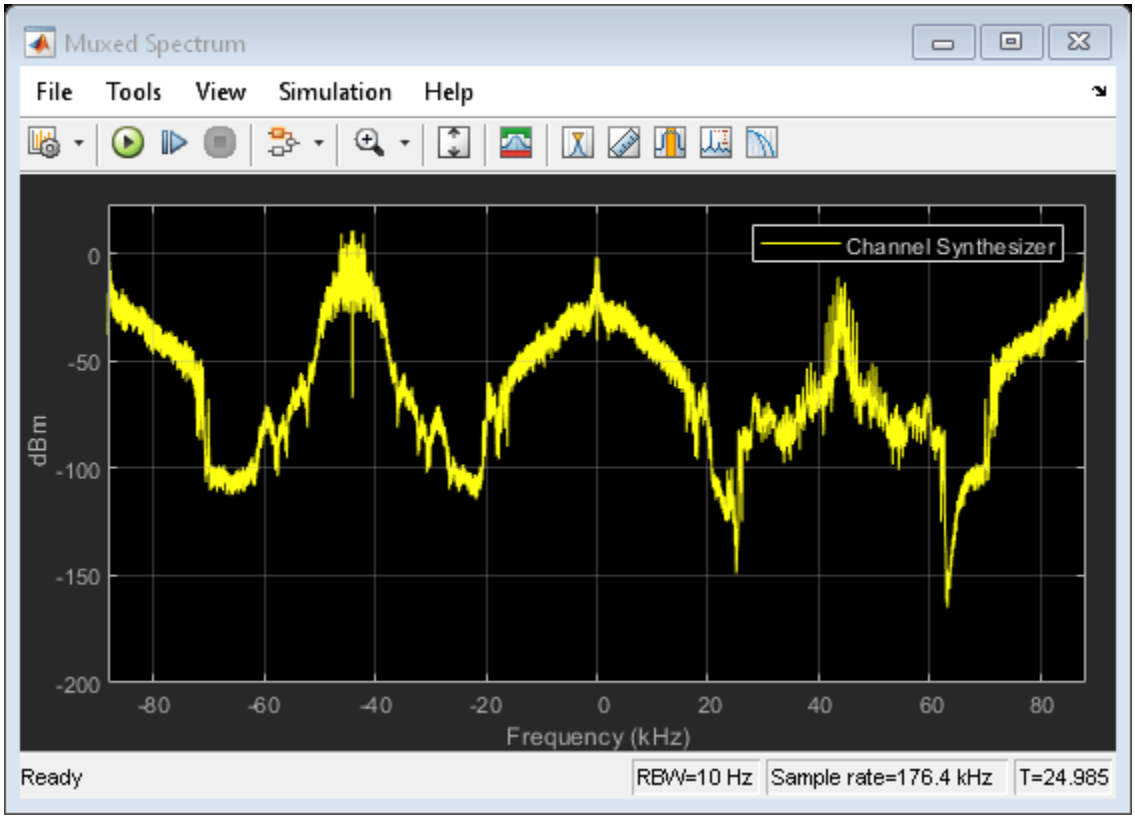


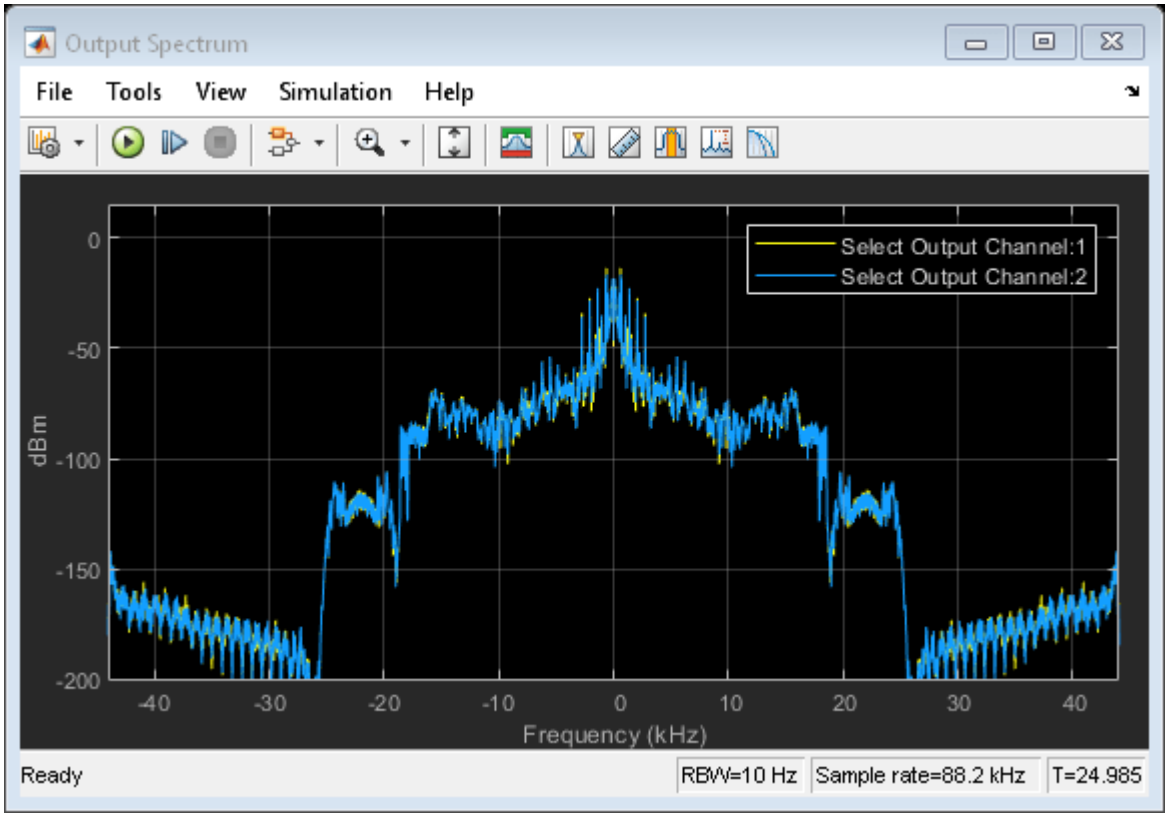
The Muxed Spectrum window shows the spectrum of the broadband signal. The Channelized Spectrum window shows the spectra of the four narrowband signals. The input and output spectra match for any selected signal.

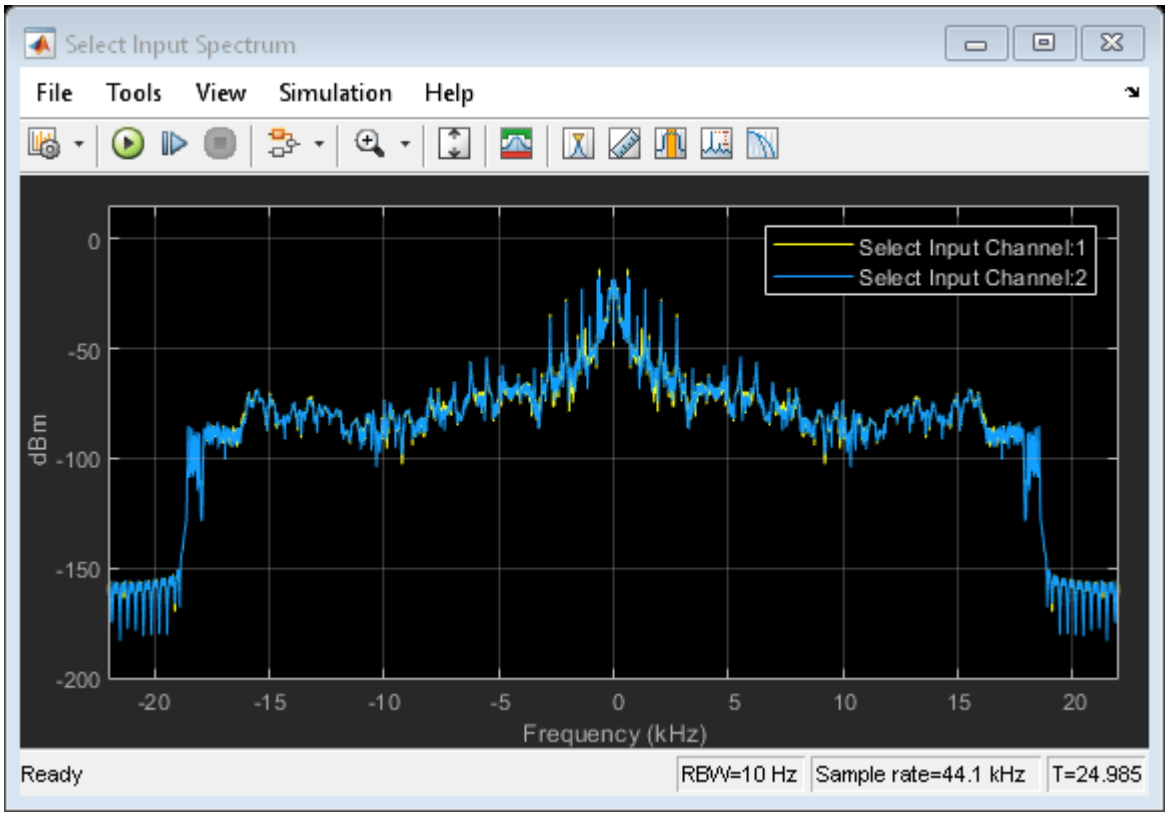
Increase the oversampling ratio of the Channelizer block to 2, which means the M/D ratio described in “Algorithm” is now 2. You can do this by decreasing the decimation factor to 2, so that M/D = 4/2, which equals 2.


Channelized Spectrum
File Tools View Simulation Help
Channelizer: 1
Channelizer: 2
Channelizer: 3
Channelizer: 4
dBm
0
-50
-100
-150
-200
-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
-200
-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
-200
-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

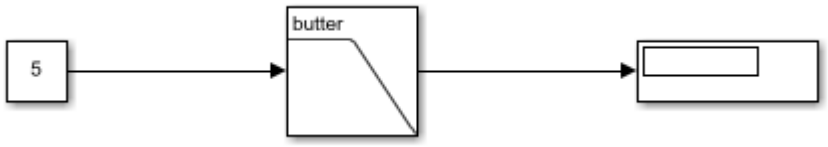
Channelizer | Channel Synthesizer | From Multimedia File | To Audio Device

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);
```



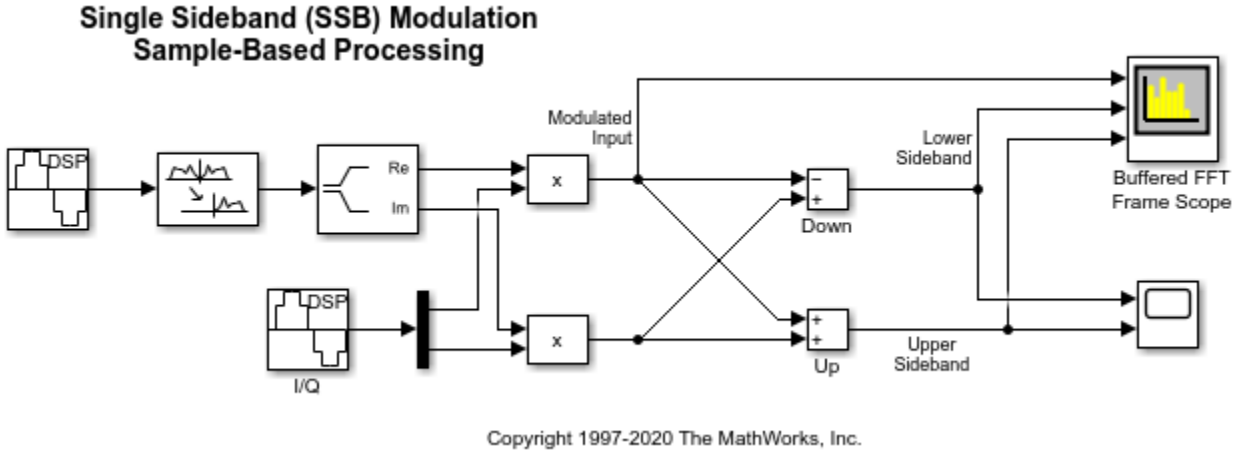
The **Analog Filter Design** block returns the filtered input as a scalar. Run the model.

```
sim(model);
```

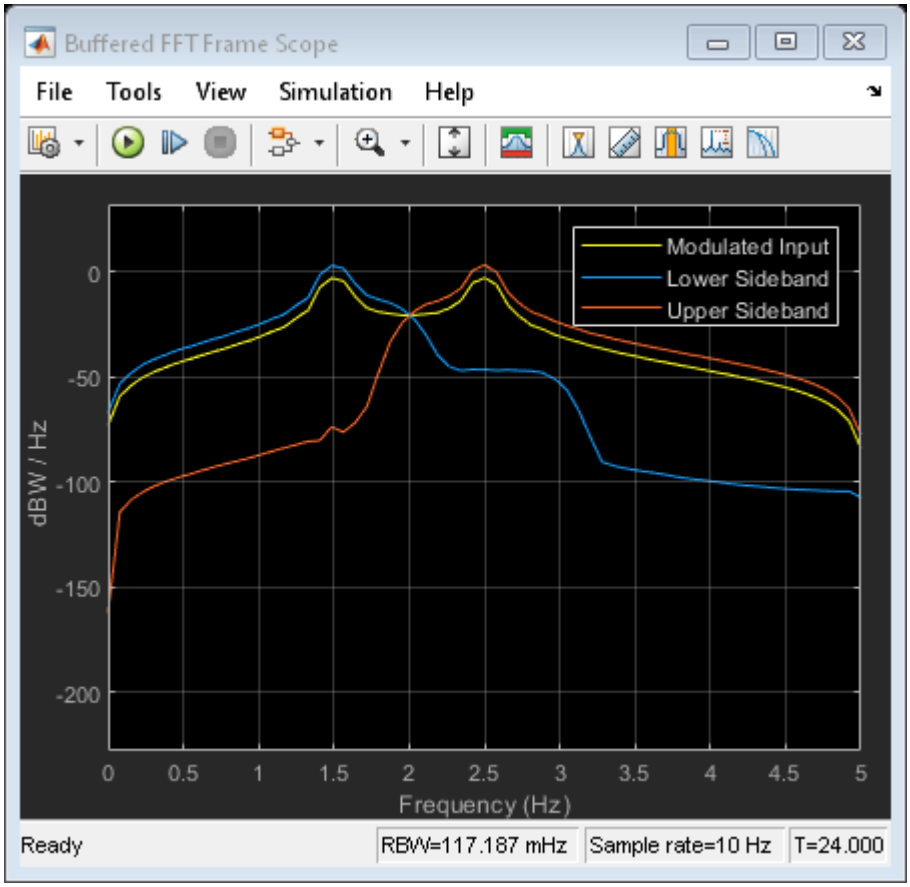
SSB Modulation

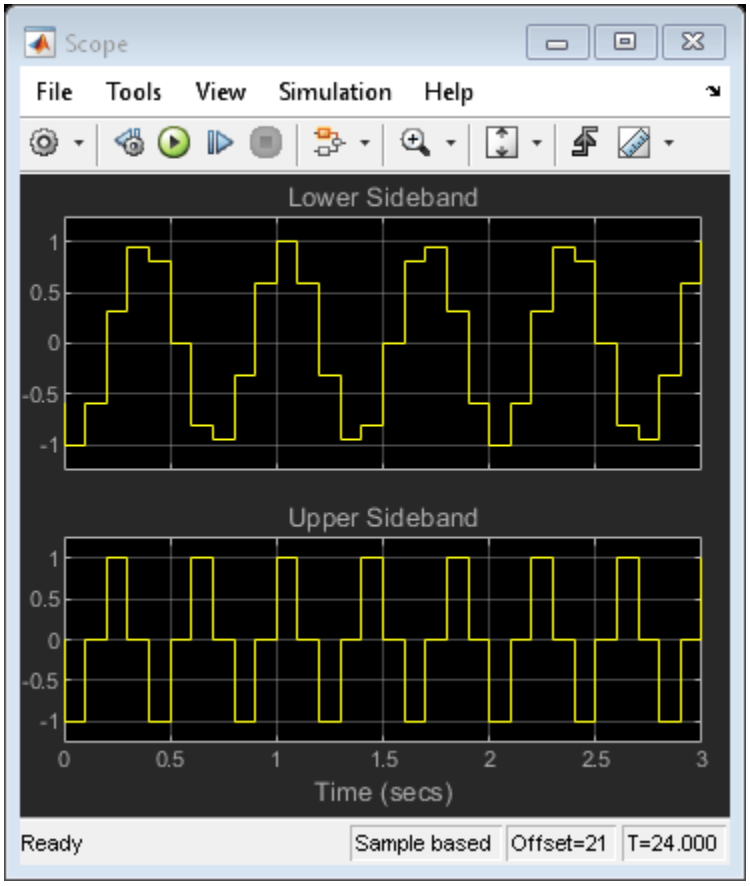
This example shows single sideband (SSB) modulation using sample-based and frame-based processing.

Open the `ssbMod` model. This model performs SSB modulation using sample-based processing.

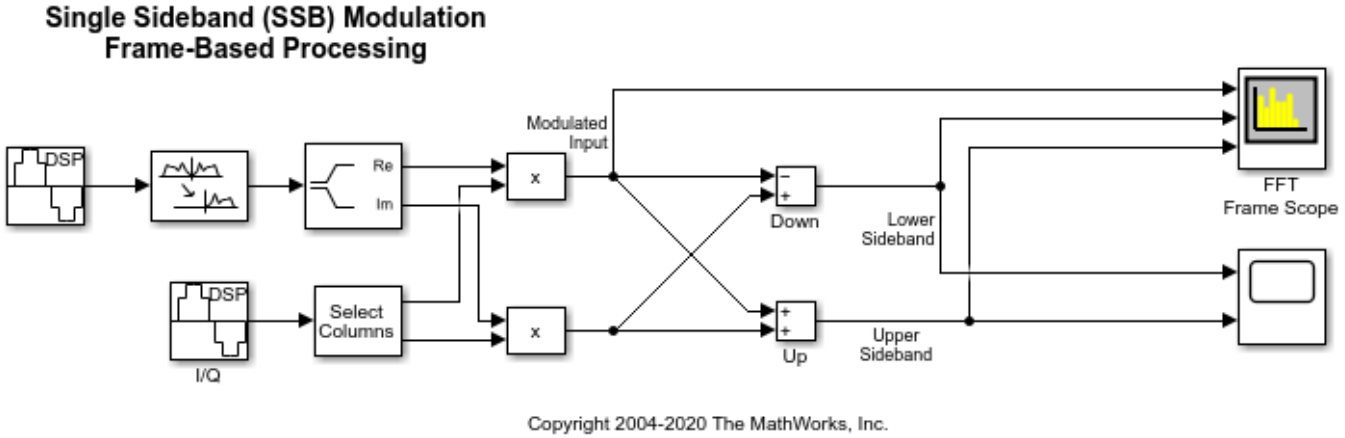


Simulate the model.

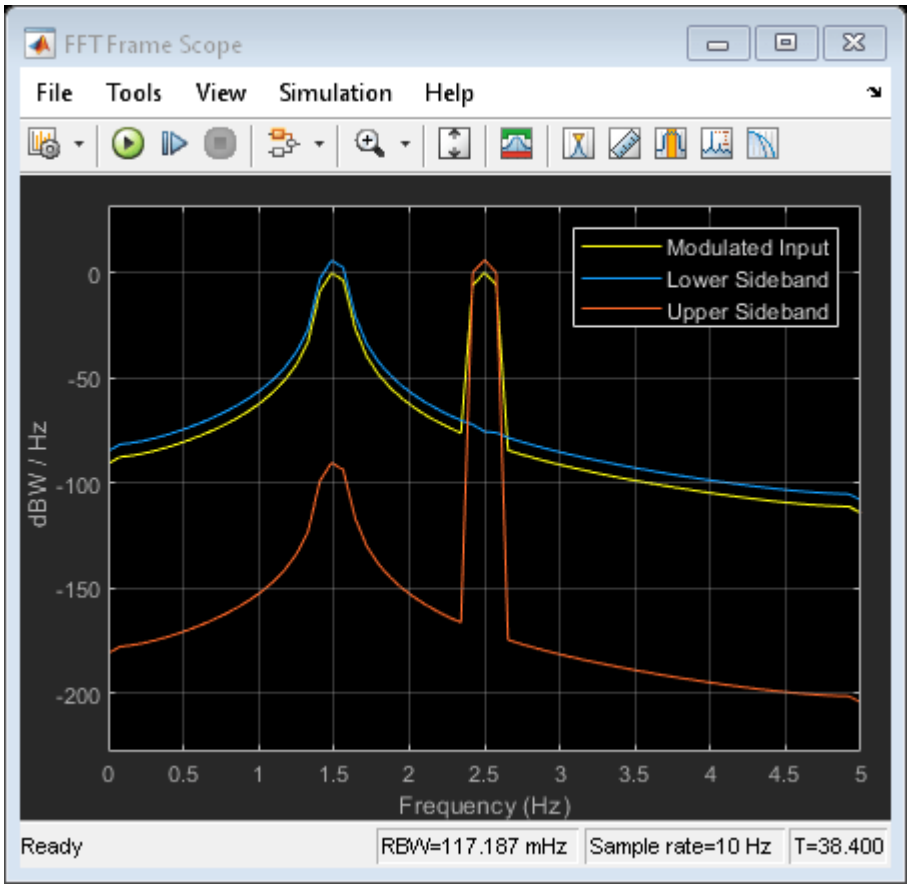

Buffered FFT Frame Scope
File
Tools
View
Simulation
Help
Modulated Input
Lower Sideband
Upper Sideband
0
-50
-100
-150
-200
dBW / Hz
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
Frequency (Hz)
Ready
RBW=117.187 mHz
Sample rate=10 Hz
T=24.000

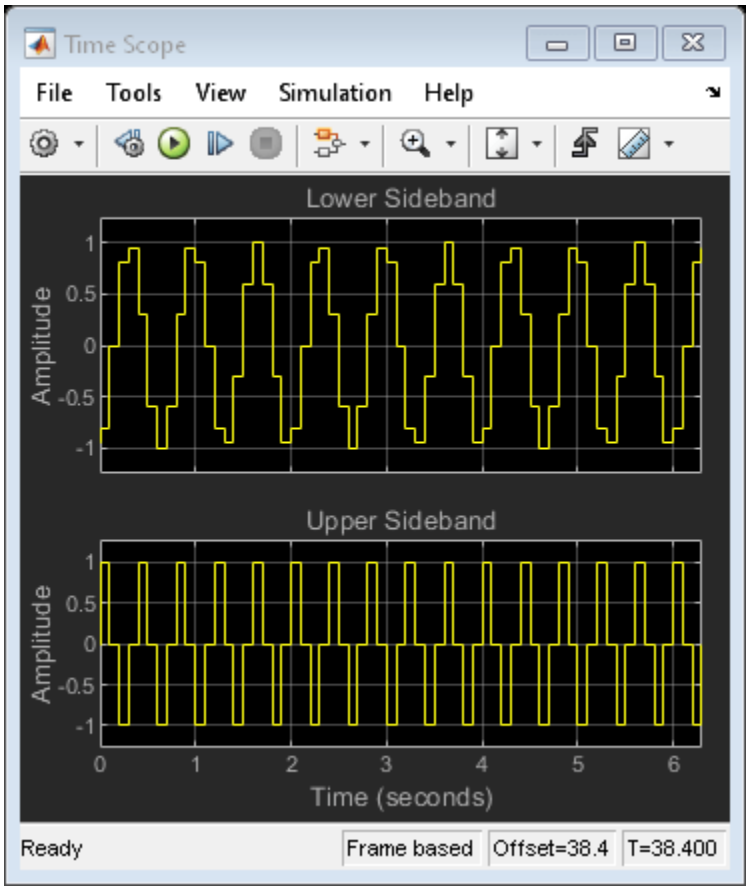


Open the `ssbMod_frame` model. This model performs SSB modulation using frame-based processing.



Simulate the model.


FFT Frame Scope
File
Tools
View
Simulation
Help
Modulated Input
Lower Sideband
Upper Sideband
dBW / Hz
0
-50
-100
-150
-200
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
Frequency (Hz)
Ready
RBW=117.187 mHz
Sample rate=10 Hz
T=38.400


Time Scope
File
Tools
View
Simulation
Help
Lower Sideband
Amplitude
1
0.5
0
-0.5
-1
Upper Sideband
Amplitude
1
0.5
0
-0.5
-1
0
1
2
3
4
5
6
Time (seconds)
Ready
Frame based
Offset=38.4
T=38.400

Wavelet Reconstruction and Noise Reduction

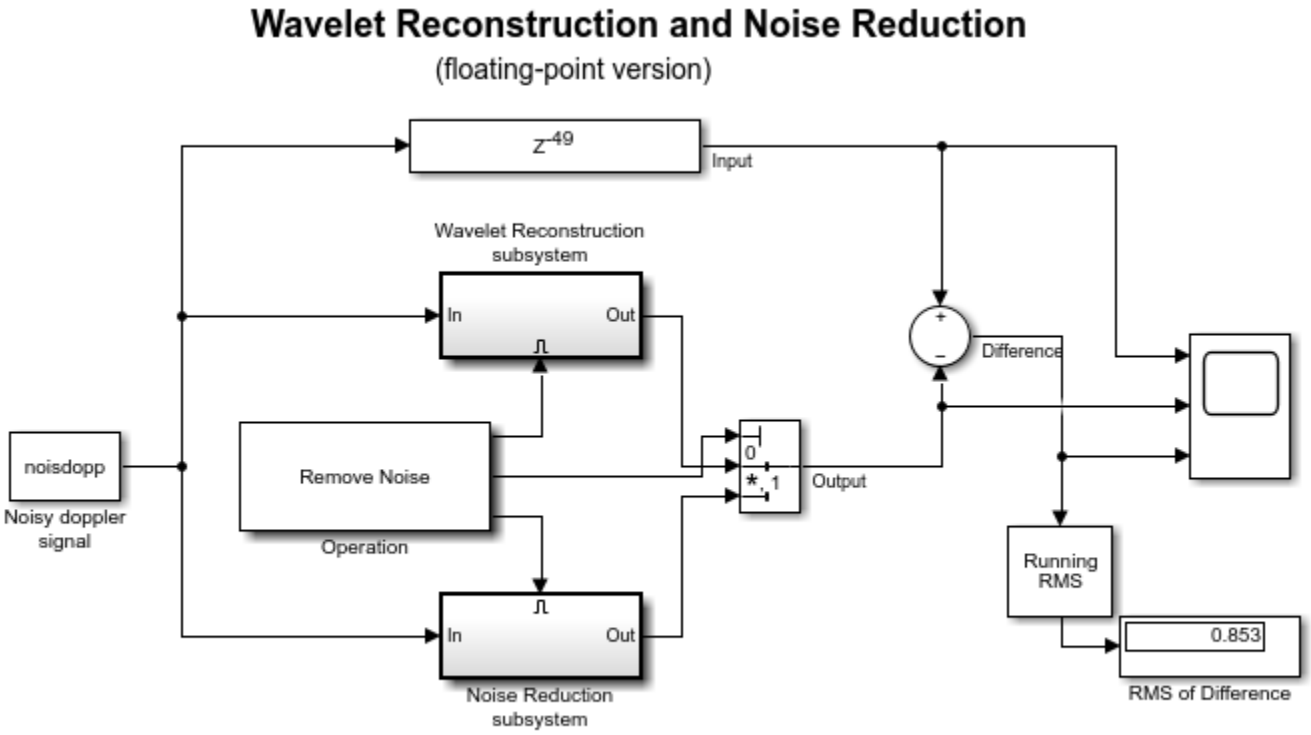
This example uses the Dyadic Analysis Filter Bank and Dyadic Synthesis Filter Bank blocks to show both the perfect reconstruction property of wavelets and an application for noise reduction.

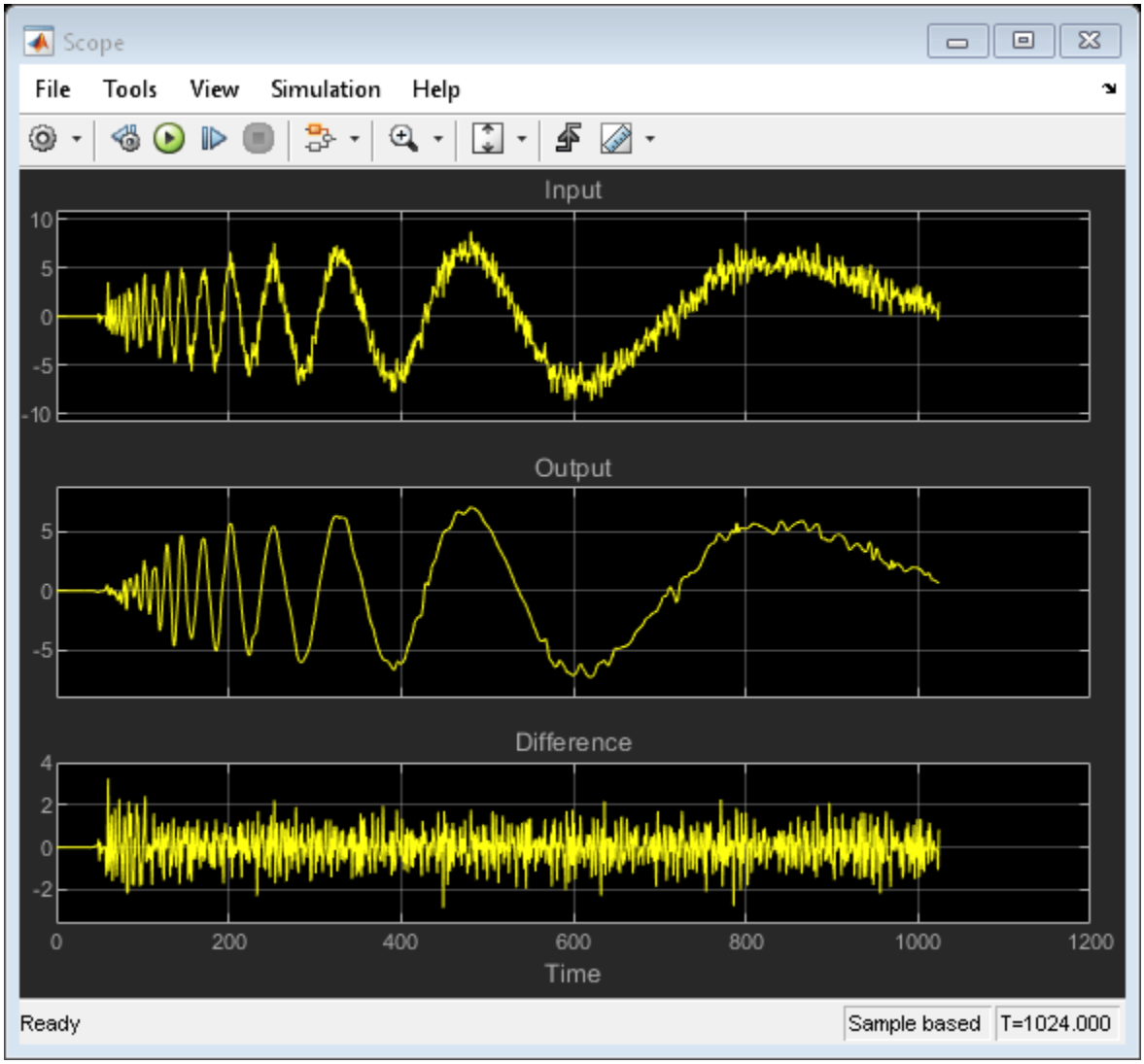
Open the Operation block dialog and select either `Remove noise` or `Perfect reconstruction`. The selection will enable the corresponding enabled subsystem.

Wavelet Reconstruction subsystem shows an Analysis Filter Bank followed by the Wavelet Reconstruction subsystem. The net effect of these two operations is perfect reconstruction of the input signal.

Opening the Noise Reduction subsystem shows the same wavelet blocks but with a soft threshold applied to the transformed signal bands. By attenuating the higher frequency bands, the high frequency noise is reduced. You can adjust the threshold levels to see the effects of attenuation on the denoising characteristics of the system.

Run the example to view the input and output signals and the difference between them. Note that for perfect reconstruction, the difference appears to be zero. However, due to numerical effects, there is a small difference that can be seen in the display of the running RMS display.





For floating-point sample-based version, open `dspwaveletModel.slx`. For floating-point frame-based version, open `dspwavelet_frameModel.slx`. For fixed-point sample-based version, open `dspwavelet_fixptModel.slx`. To find these models, open this example in MATLAB®, click on the **Open Script** button in the example. The example script opens and you can find all the models in the current working directory of MATLAB.

Simulink Block Examples in DSP System Toolbox

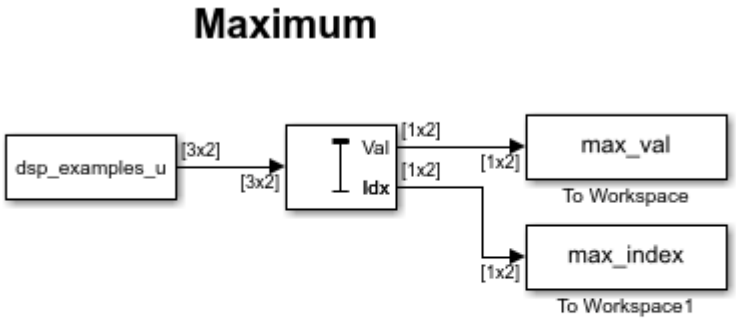
- “Compute the Maximum” on page 16-2
- “Compute the Running Maximum” on page 16-4
- “Compute the Minimum” on page 16-6
- “Compute the Running Minimum” on page 16-8
- “Compute the Mean” on page 16-10
- “Compute the Running Mean” on page 16-12
- “Compute the Histogram of Real and Complex Data” on page 16-14
- “Extract Submatrix from Input Signal” on page 16-19
- “Compute Difference of a Matrix” on page 16-21
- “Compute Maximum Column Sum of Matrix” on page 16-22
- “Extract Diagonal of Matrix” on page 16-23
- “Generate Diagonal Matrix from Vector Input” on page 16-24
- “Permute Matrix by Row or Column” on page 16-25
- “LDL Factorization of 3-by-3 Hermitian Positive Definite Matrix” on page 16-26
- “Compute Power Measurements of Voltage Signal in Simulink” on page 16-27
- “Compute Matrix Exponential” on page 16-30
- “Compute Moving RMS of Noisy Step Signal” on page 16-31
- “Compute RMS of Noisy Step Signal” on page 16-34

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

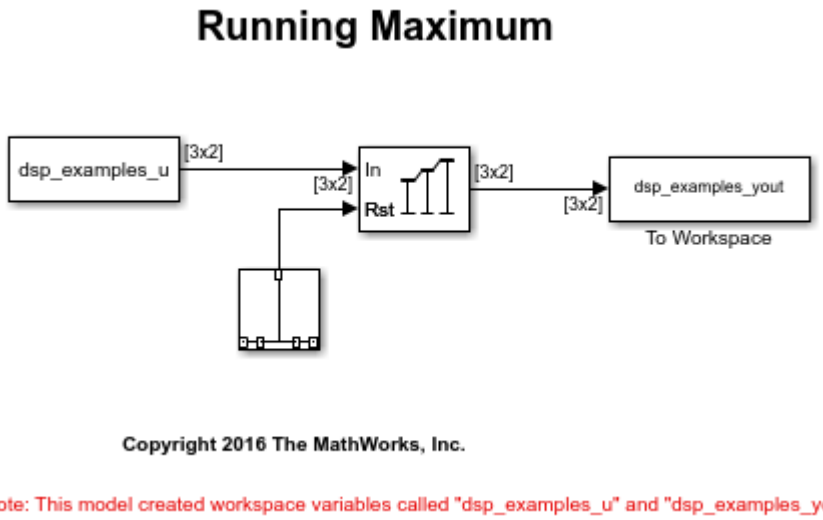
Blocks
Maximum | Signal From Workspace

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.

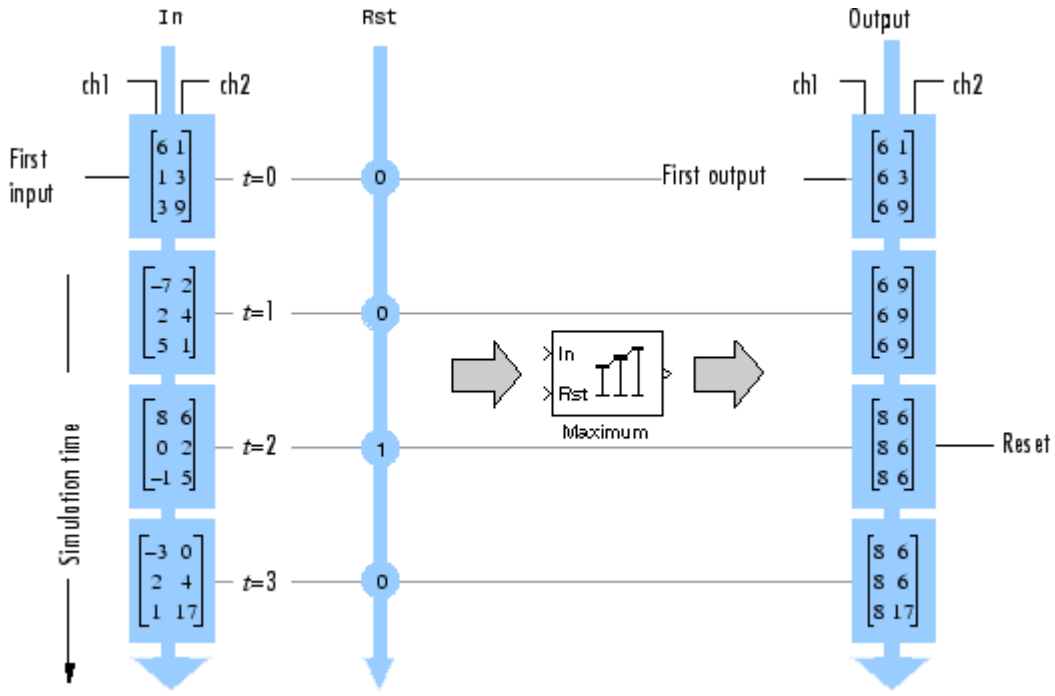
```
model = 'ex_runningmaximum_ref';
open_system(model)
```



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

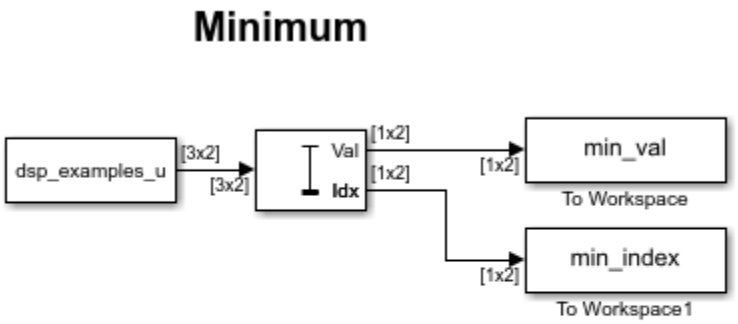
Maximum | Signal From Workspace | Discrete Impulse

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)
```



Note: This model created workspace variables called "dsp_examples_u", "min_val", and "min_index".

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

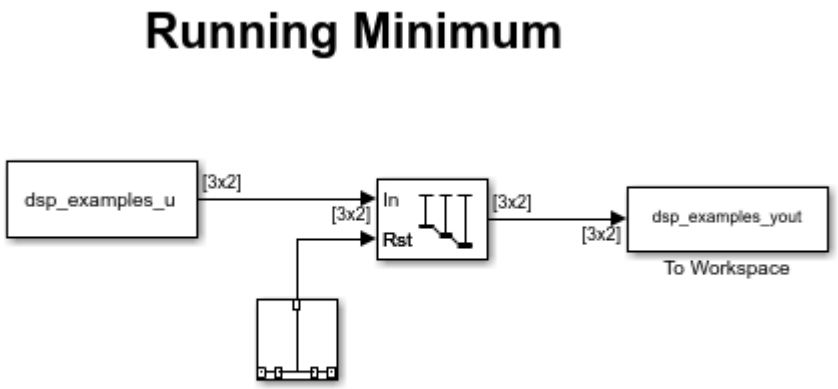
Blocks
Minimum | Signal From Workspace

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.

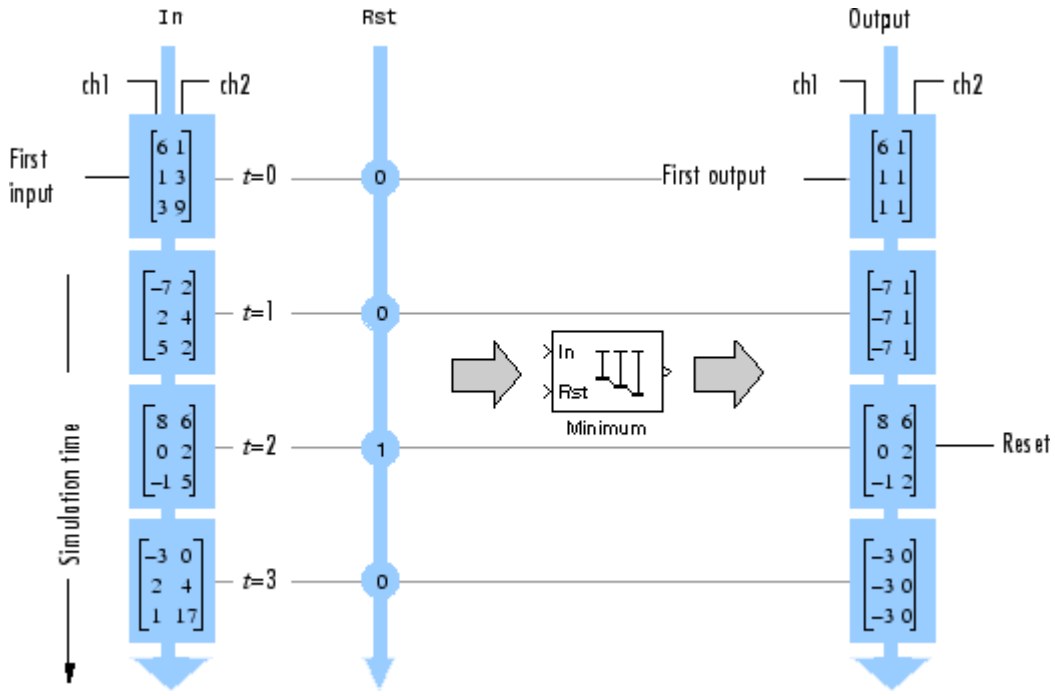
```
model = 'ex_runningminimum_ref';
open_system(model)
```



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

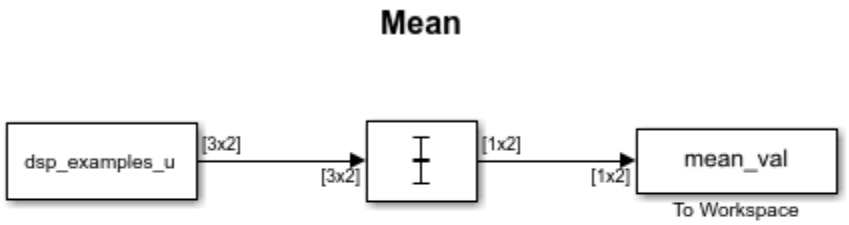
Minimum | Signal From Workspace | Discrete Impulse

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)
```



Copyright 2016 The MathWorks, Inc.

Note: This model created workspace variables called "dsp_examples_u" and "mean_val".

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

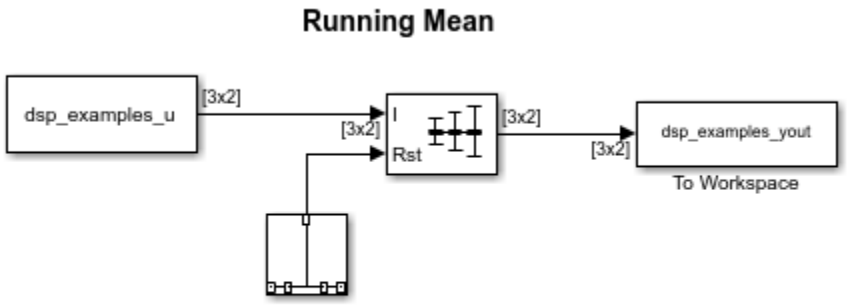
Blocks
Mean | Signal From Workspace | To Workspace

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)
```



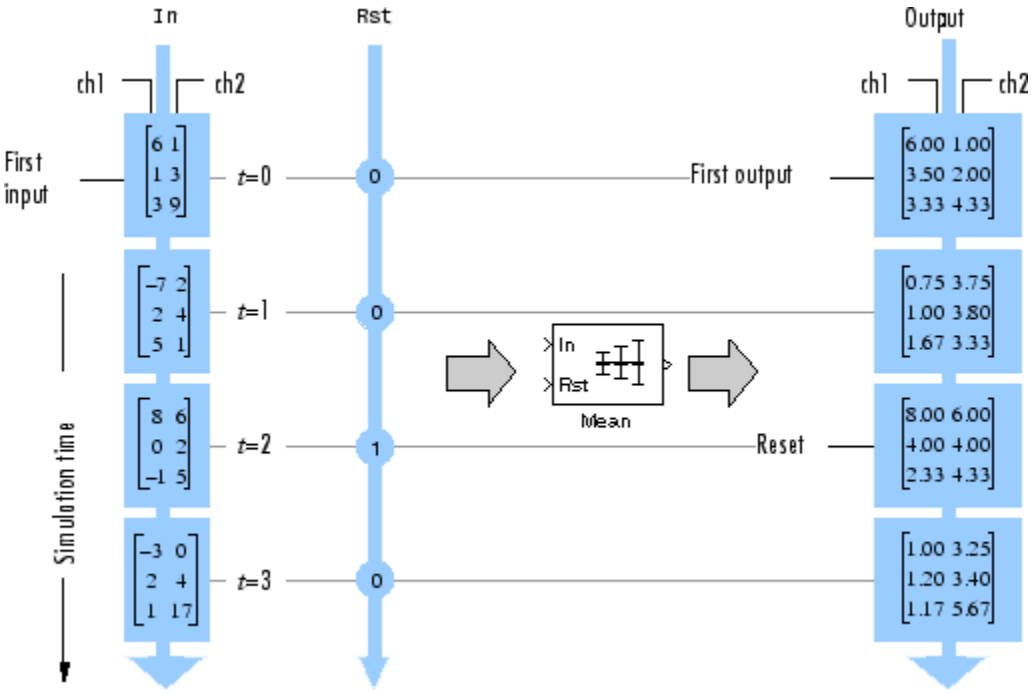
Copyright 2016 The MathWorks, Inc.

Note: This model created workspace variables called "dsp_examples_u" and "dsp_examples_yout".

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

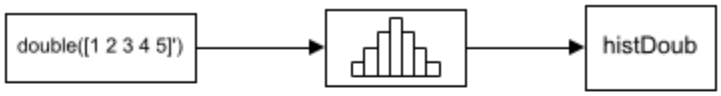
The bin boundaries created by the Histogram block are determined by the data type of the input. The following two models show the differences in the output of the Histogram block based on the data type of the input.

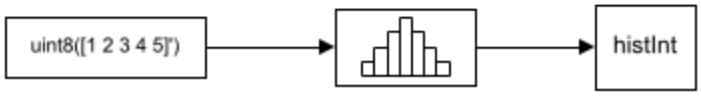
Real Input Data

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)
```



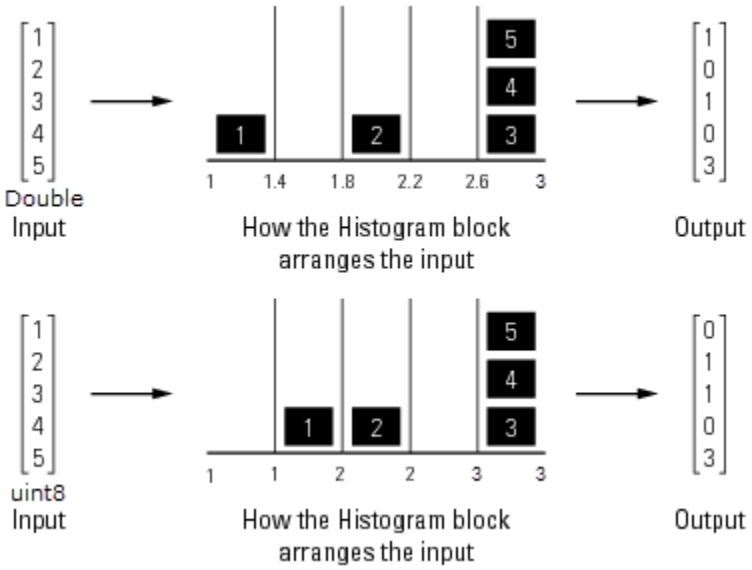


Run the model.

```
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.



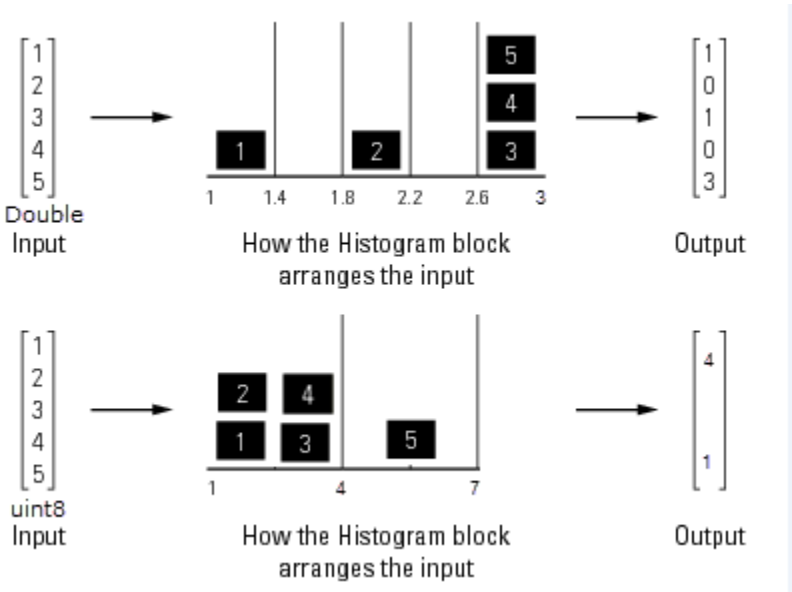
The output of the Histogram block differs based on the data type of the input. A warning occurs in the second histogram block, where the bin boundaries are uint8([1 1.4 1.8 2.2 2.6 3.0]) = [1 1 2 2 3 3]. The width of the first and third bins are 0, and the precision of the data is 1. The block expects 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)
```



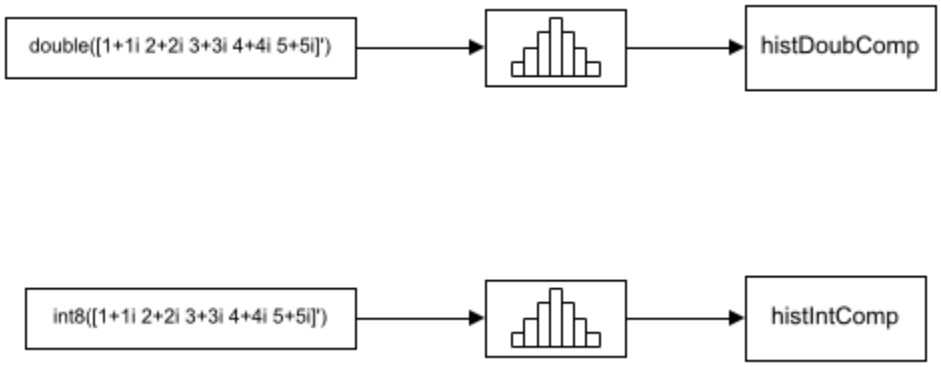
Complex Input Data

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)
```

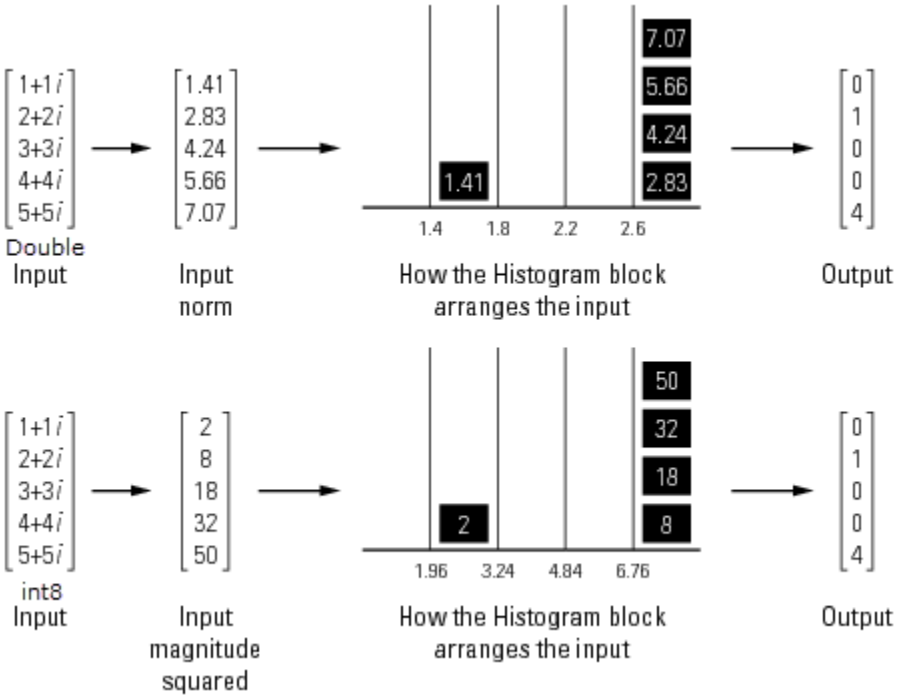


Run the model.

```
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.



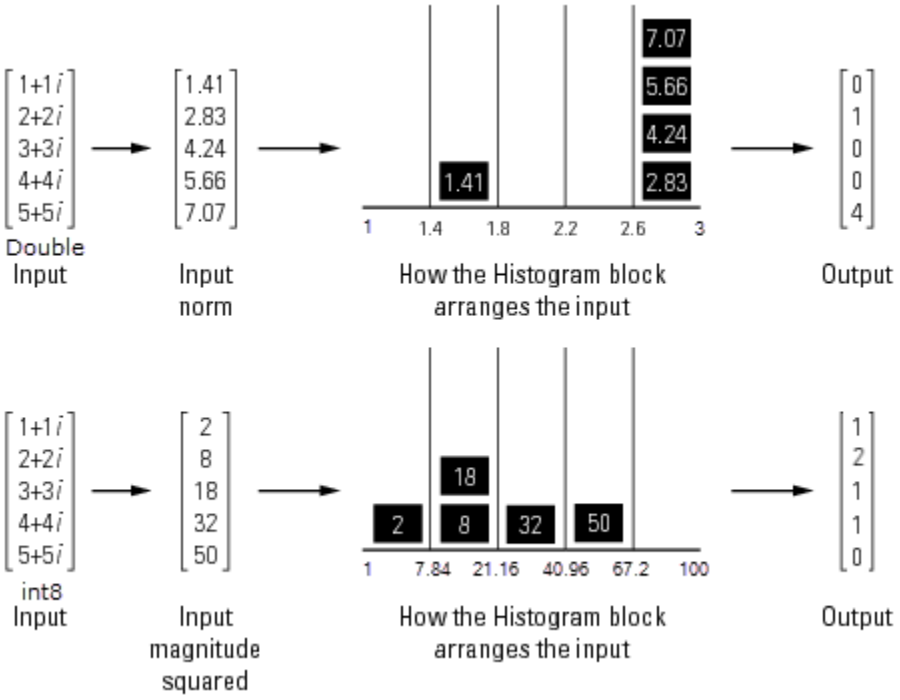
The top figure shows the histogram for the double-precision input, and the bottom figure shows the histogram for the integer input. The double-precision inputs are normalized for comparison, whereas the integer inputs are placed using their magnitude-squared value. A warning occurs in the second histogram 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.

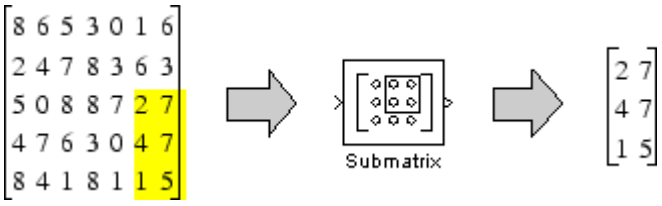
```
save_system(modelRealData);
save_system(modelComplexData)
close_system(modelRealData);
close_system(modelComplexData);
```

See Also

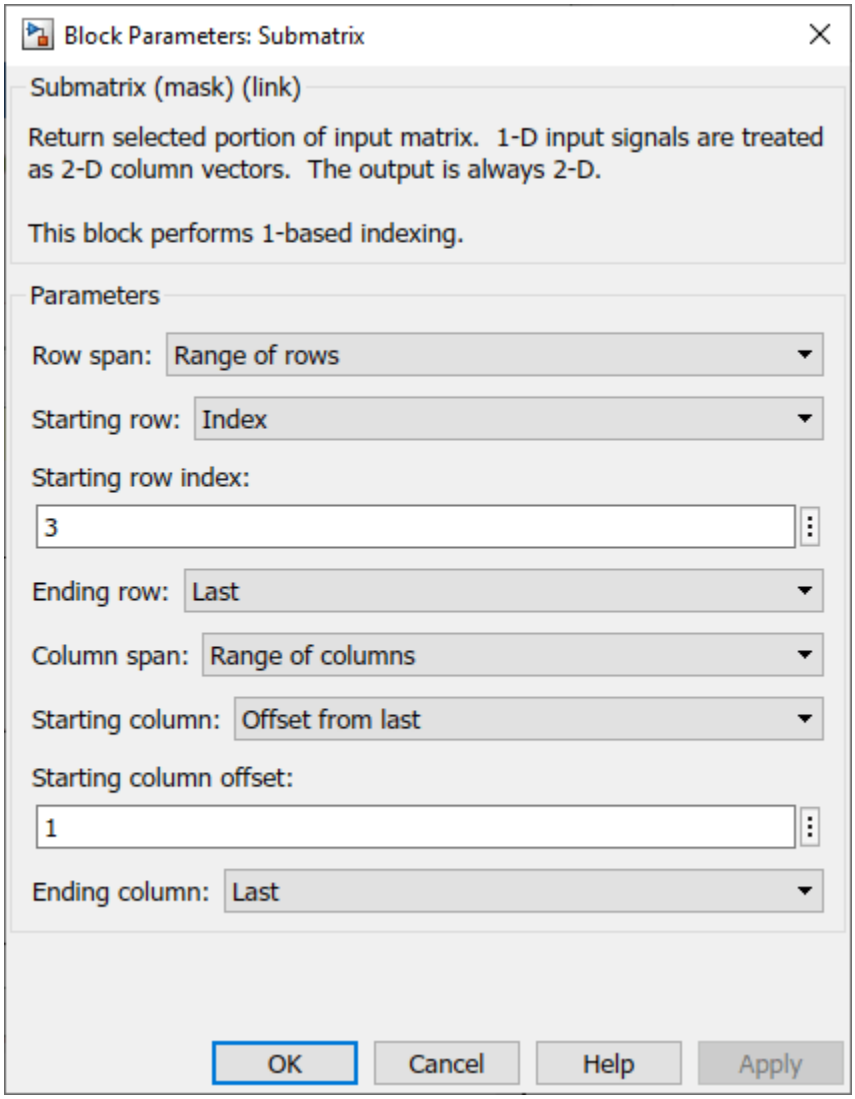
Blocks
Histogram

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)`.



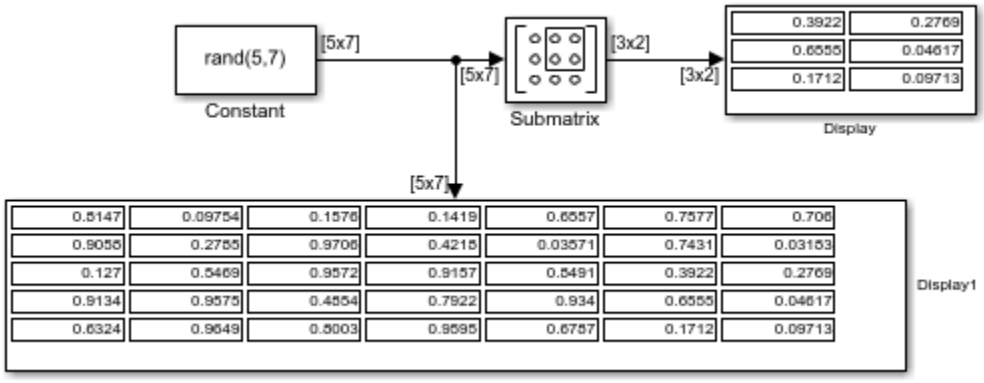
Here are the settings used for the `Submatrix` block in this example.



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

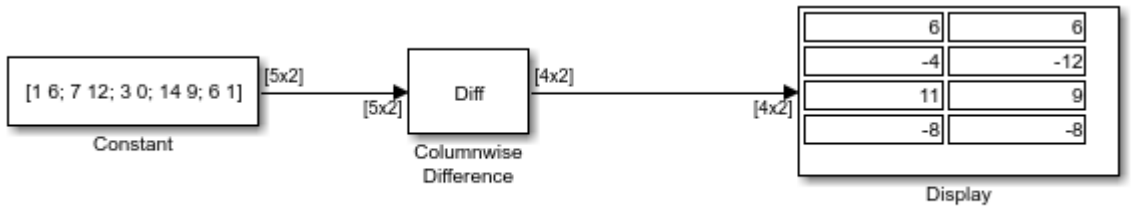
Open and simulate the model. You can see that the 3-by-2 submatrix from the lower-right corner of a 5-by-7 input matrix has been extracted.



Compute Difference of a Matrix

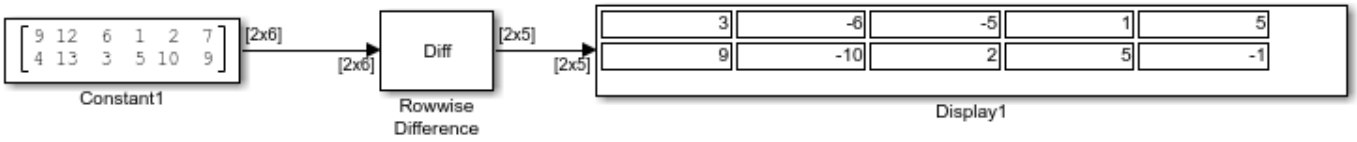
Open and run the model.

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`.



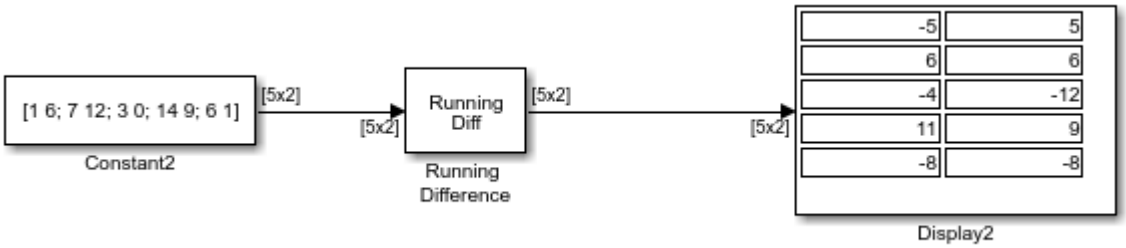
Copyright 2019 The MathWorks, Inc.

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.

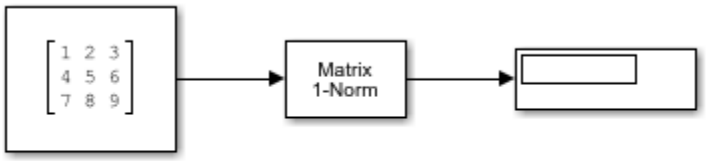
See Also

Blocks
Difference | Display | Constant

Compute Maximum Column Sum of Matrix

This example shows how to compute the highest column-sum of a 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.

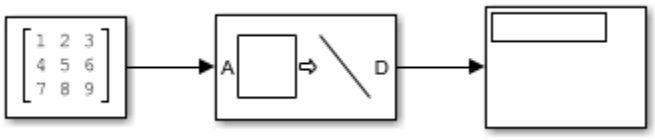
See Also

Blocks
Matrix 1-Norm | Display | Constant

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.

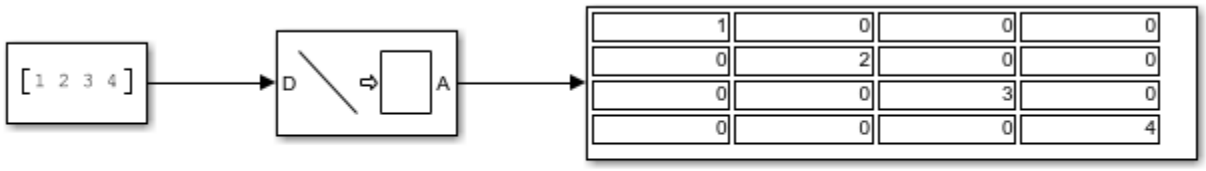
See Also

Blocks
Extract Diagonal | Display | Constant

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

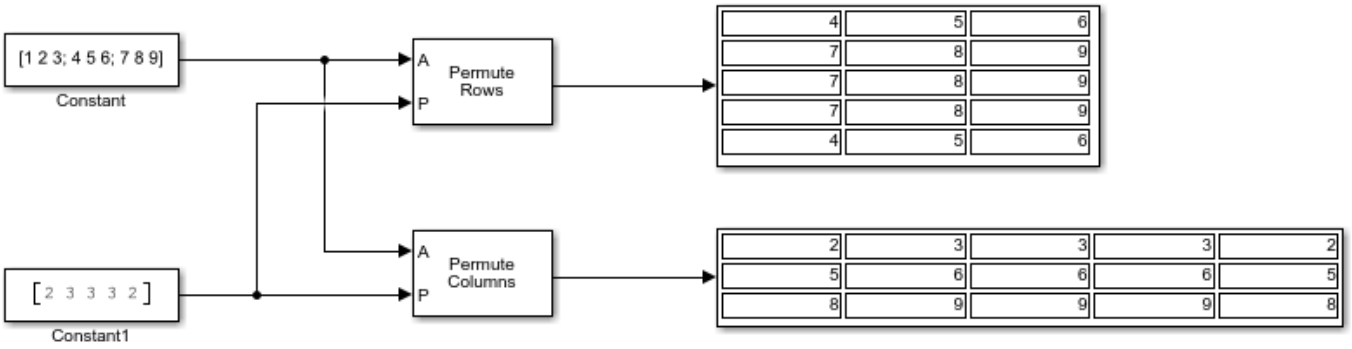
Blocks
Create Diagonal Matrix | Display | Constant

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
Permute Matrix | Display | Constant

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.

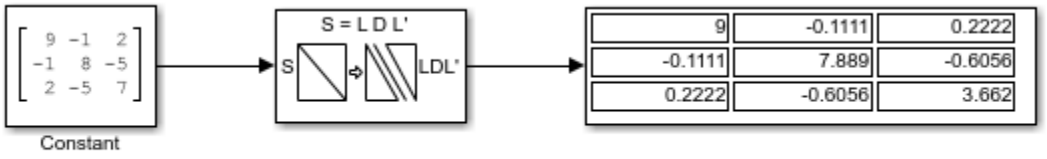
For the input in the model, the corresponding L and D values become:

```
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
LDL Factorization | Display | Constant

Compute Power Measurements of Voltage Signal in Simulink

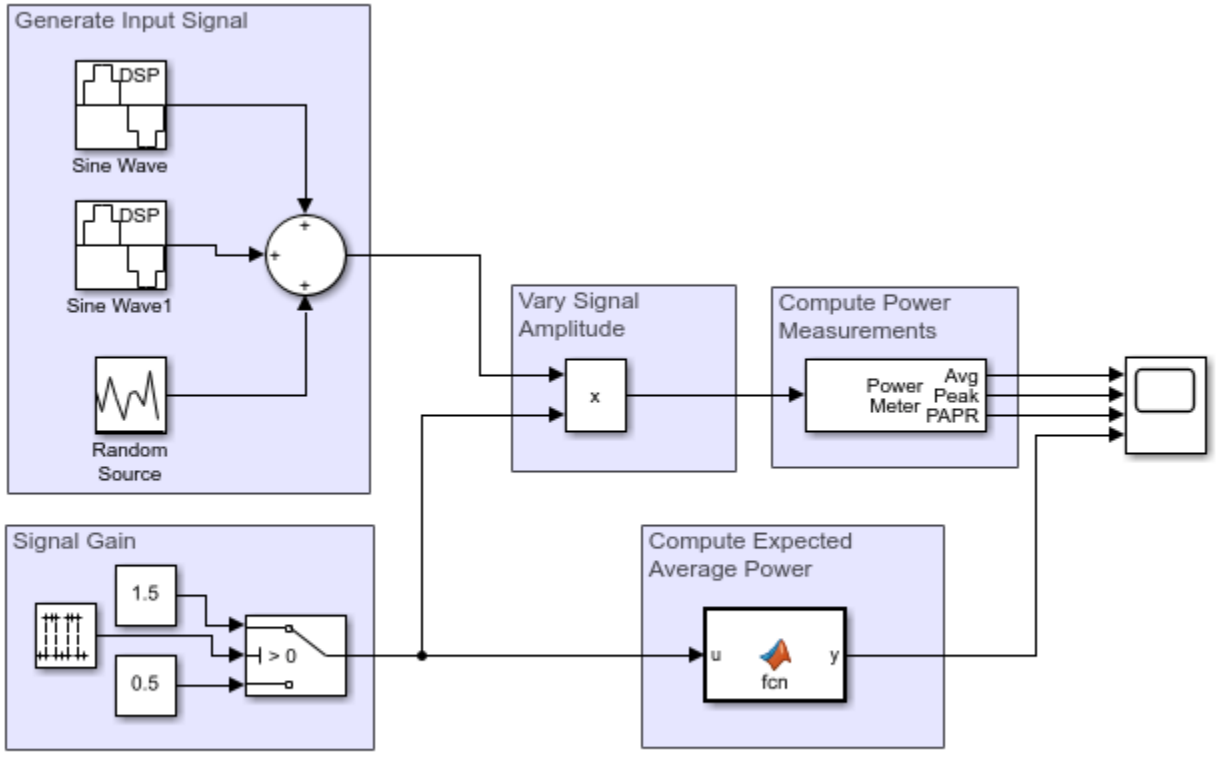
Compute the power measurements of a noisy sinusoidal signal using a power meter. These measurements include average power, peak power, and peak-to-average power ratio.

Assume the maximum voltage of the signal $Vmax$ to be 100 V. The instantaneous values of a sinusoidal waveform are given by the equation $Vi = Vmax * sin(2\pi ft)$, where Vi is the instantaneous value, $Vmax$ is the maximum voltage of the signal, and f is the frequency of the signal in Hz.

Open Model

The input signal is a sum of two sine waves with frequencies set to 1 kHz and 10 kHz, respectively. The frame length and the sampling frequency of the generated signal is 512 samples and 44.1 kHz, respectively. Add zero-mean white Gaussian noise that has a variance of 0.001 to the sinusoidal signal. Vary the amplitude of the sine waves.

To measure the power of this signal, use the Power Meter block. The **Measurement** parameter is set to `'All'`. This setting enables the block to measure the average power, peak power, and peak-to-average power ratio. The length of the sliding window in the power meter is set to 16 samples and the reference load is set to 50 ohms. The power is measured in `dBm` units. Visualize the power measurements using the Time Scope block.



Copyright 2020 The MathWorks, Inc.

Compute the Power Measurements

Run the model. The average power, peak power, and peak-to-average power ratio is plotted in the Time Scope window. For details on how the power meter block computes these measurements, see “Algorithms”.

Compare the measured value to the expected value of the average power.

The expected value of the average power *P* of the noisy sinusoidal signal is given by the following equation.

$$P = A_1^2/2R + A_2^2/2R + var(noise)$$

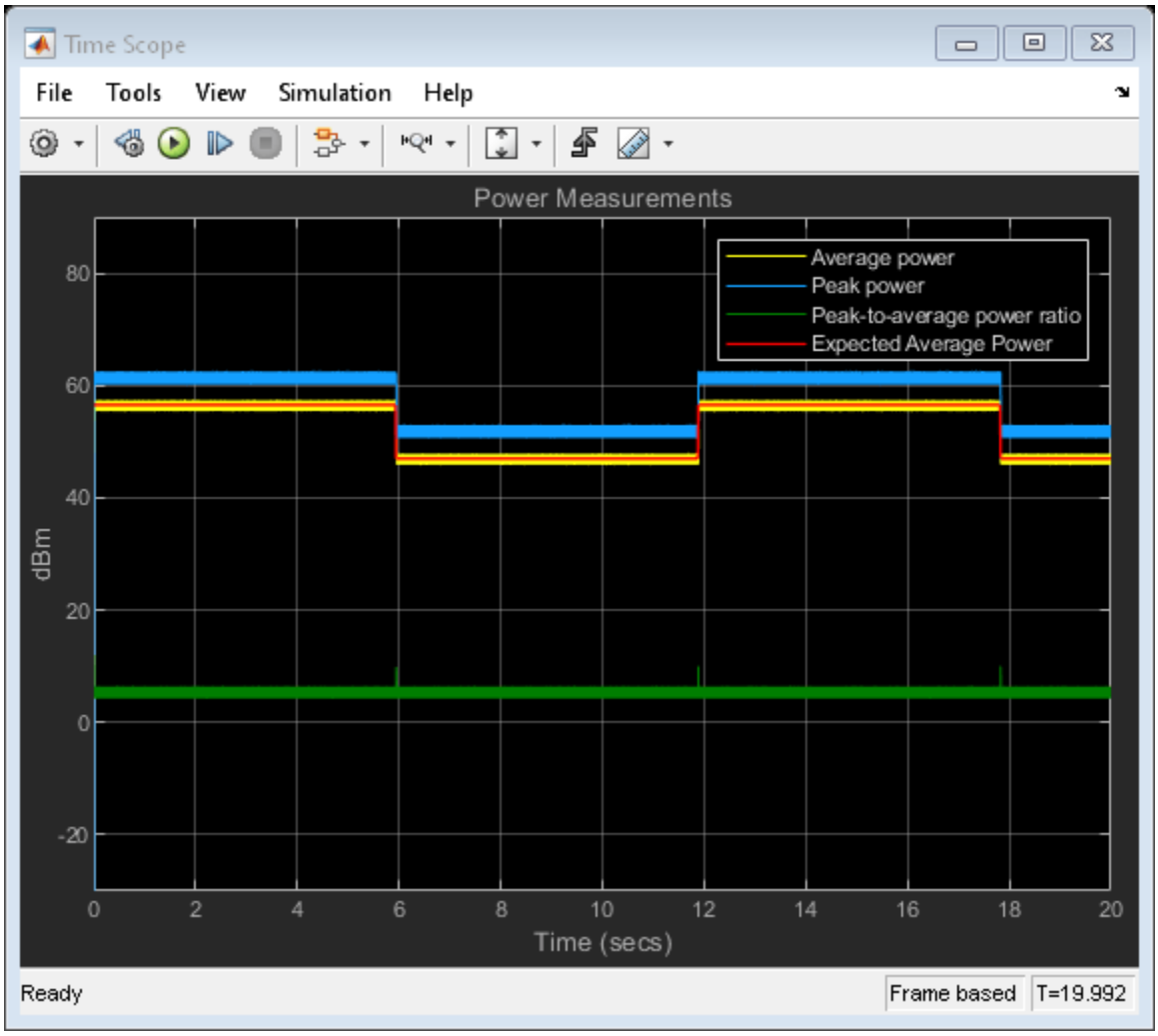
where,

- A_1 is the amplitude of the first sinusoidal signal.
- A_2 is the amplitude of the second sinusoidal signal.
- R is the reference load in ohms.

The expected power in `dBm` is computed using the following equation:

$$expPwr_{dBm} = 10log_{10}(P) + 30$$

Compare the expected value with the value computed by the object. All values are in `dBm`. These values match very closely. To verify, view the computed measurements in the Time Scope window.



See Also

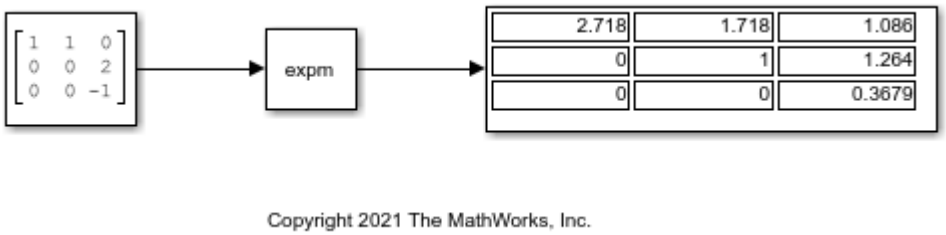
Blocks

Power Meter | Sine Wave | Random Source | Time Scope | MATLAB Function | Pulse Generator | Constant | Switch | Product

Compute Matrix Exponential

Compute the matrix exponential of an input matrix.

Open and run the model.



The input matrix is a square matrix. The Matrix Exponential block computes the matrix exponential of the square matrix using a scaling and squaring algorithm with a Pade approximation. To view the implementation, look under the mask of the Matrix Exponential block by either clicking the down arrow on the block mask or by right-clicking the block and clicking **Mask > Look Under Mask**.

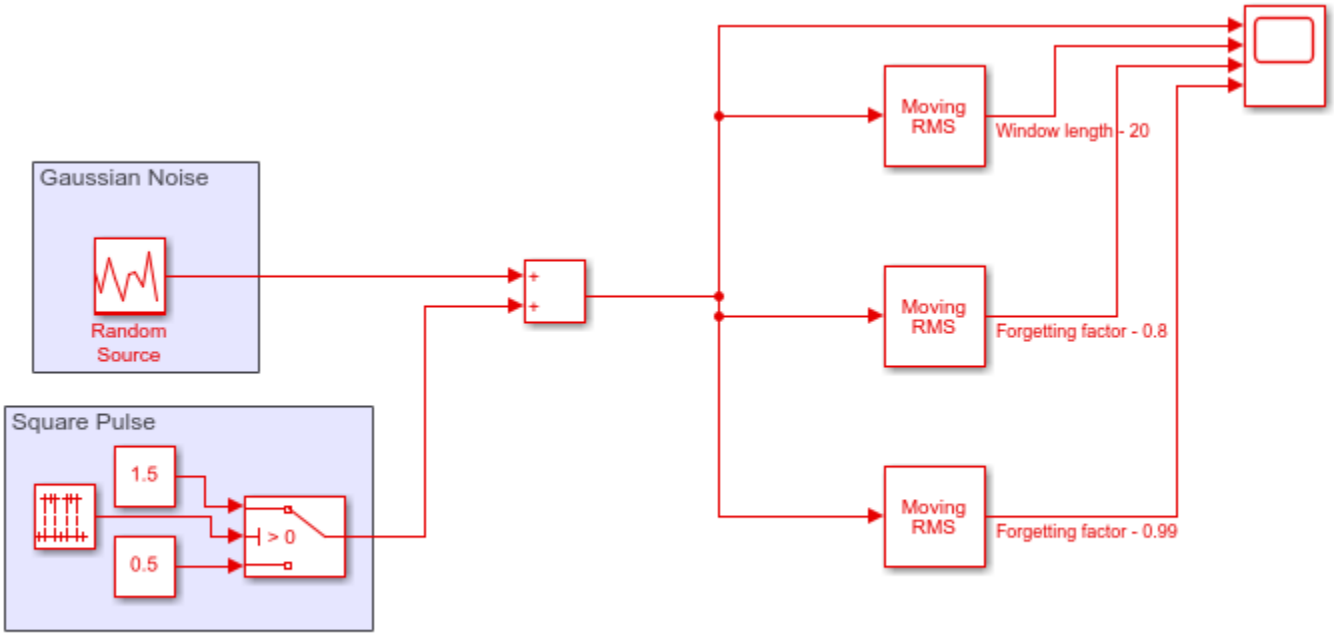
See Also

Constant | Matrix Exponential | Display

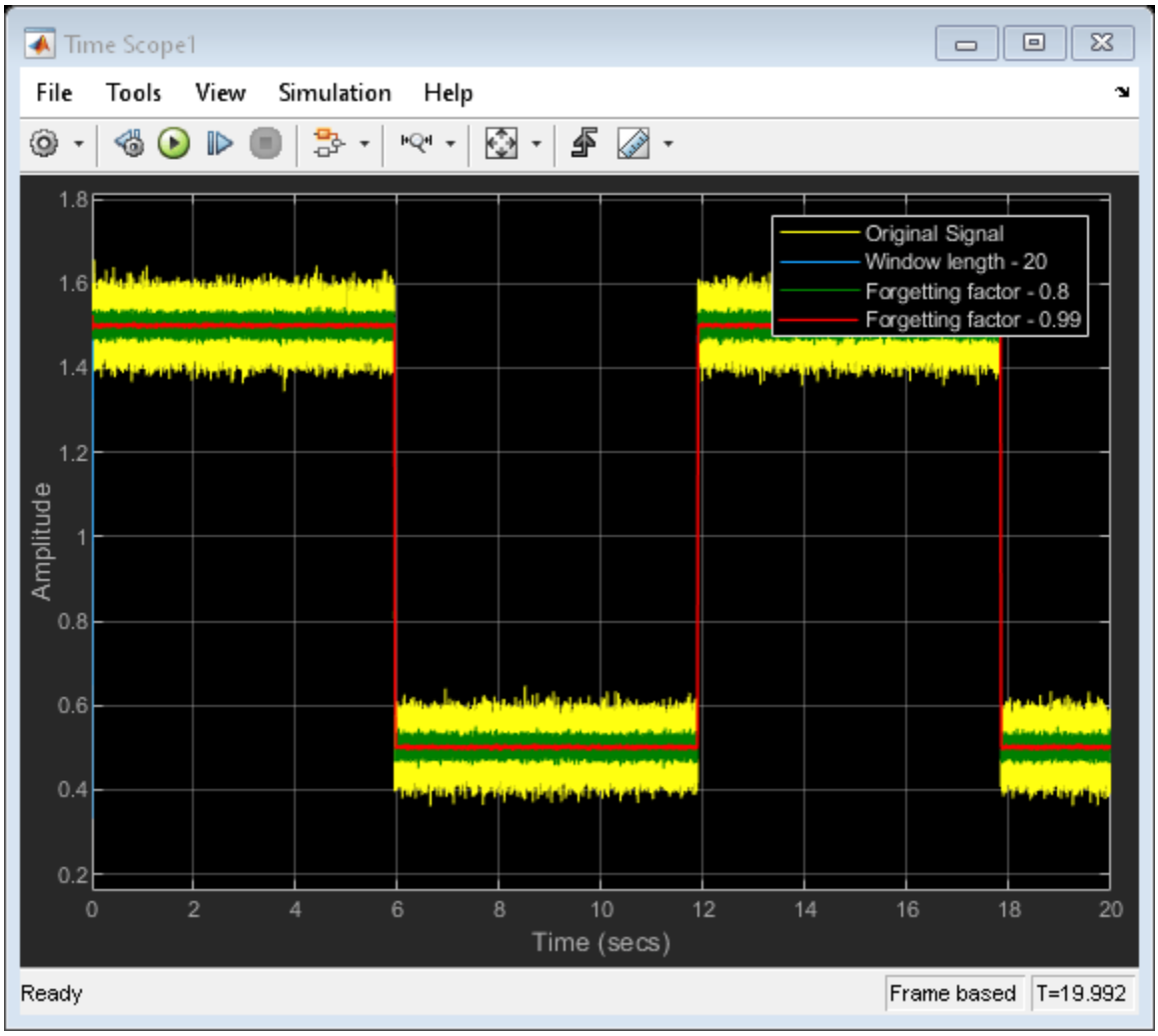
Compute Moving RMS of Noisy Step Signal

Compute the moving RMS of a noisy square wave signal using the Moving RMS block. Use the sliding window method with a window length of 20 samples and the exponential weighting method with a forgetting factor of 0.8 and 0.99. For more details on these two methods, see “Sliding Window Method and Exponential Weighting Method” on page 28-5.

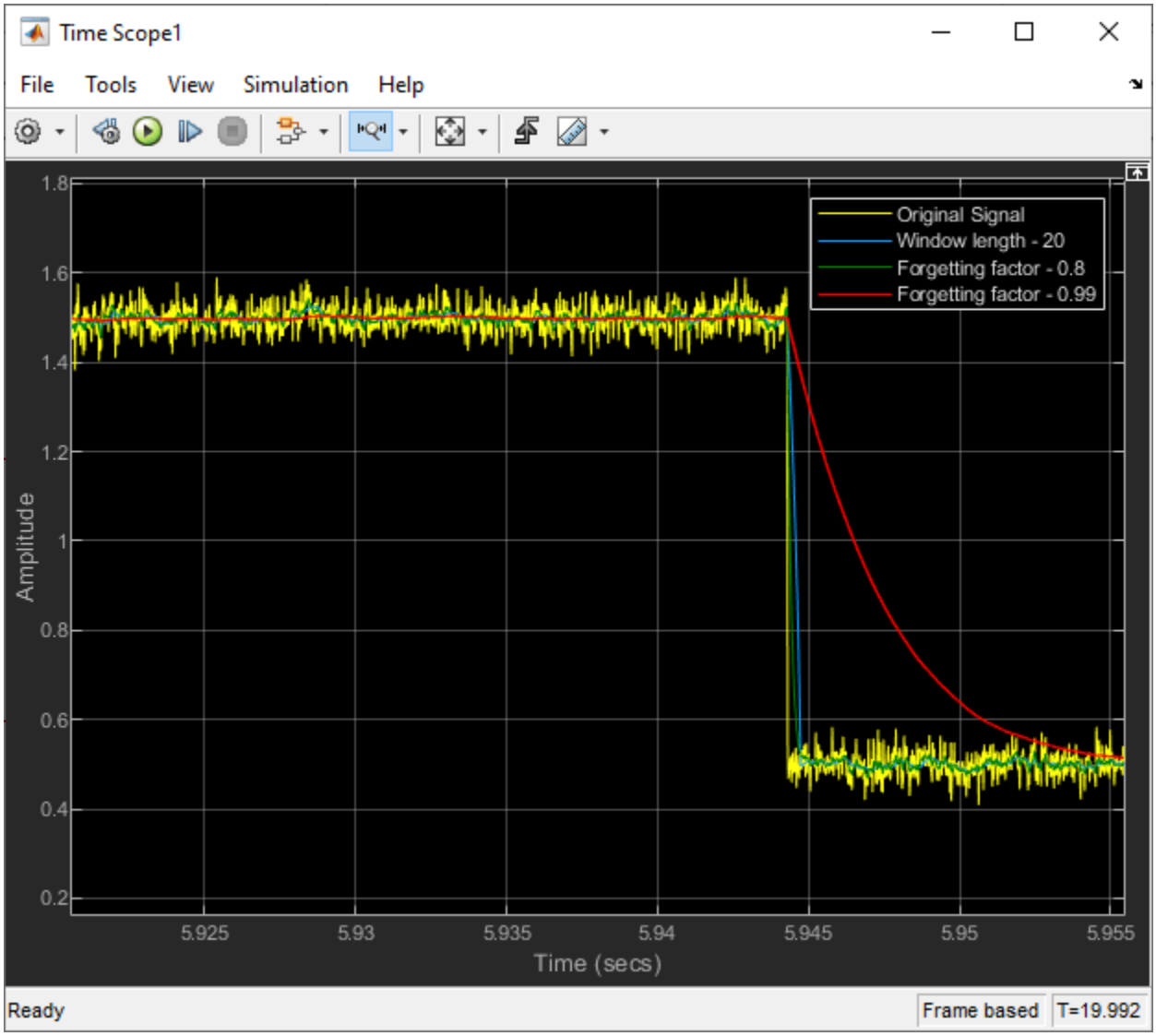
Generate a noisy square wave signal. Apply the sliding window method and the exponential weighting method on this signal. View the output in the Time Scope.



Copyright 2021 The MathWorks, Inc.



This example compares the performance of the sliding window method with the exponential weighting method. When the signal changes rapidly as in this example, use a smaller sliding window and a lower forgetting factor. When the forgetting factor is low, past data have lesser impact on the current average. This makes the transient sharper, but the signal in general contains more noise.



See Also

Blocks

Random Source | Pulse Generator | Constant | Switch | Moving RMS | Time Scope

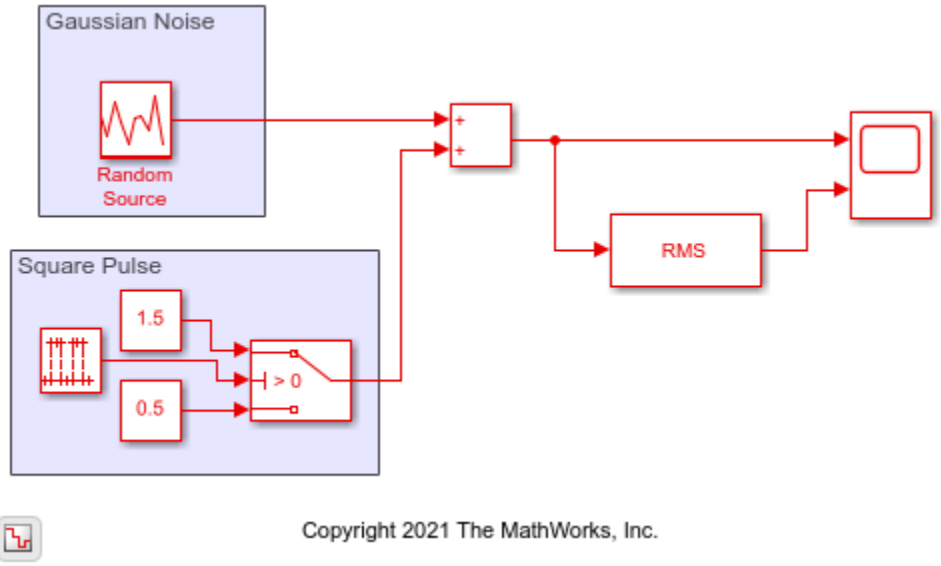
Related Examples

- “Sliding Window Method and Exponential Weighting Method” on page 28-5

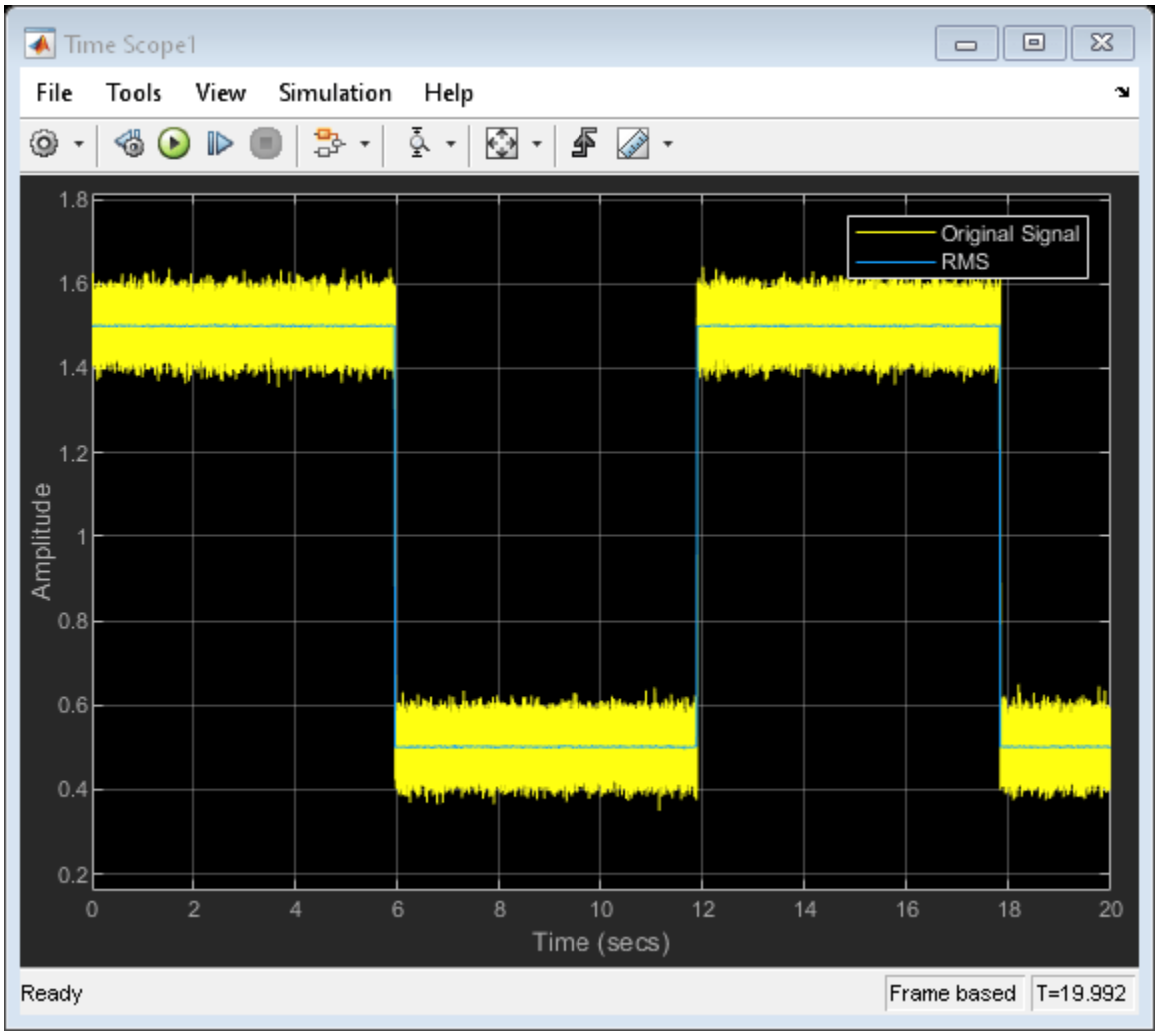
Compute RMS of Noisy Step Signal

Compute the RMS of a noisy square wave signal using the RMS block.

Generate a square pulse signal of 512 samples in width using the Pulse Generator block. Vary the amplitude of the signal between 1.5 and 0.5 using the Switch block. Add zero-mean white Gaussian noise with a variance of 0.001 to the square pulse. The frame length of the noise signal is 512 samples. Compute the RMS of this signal across each frame using the RMS block. View the output in the Time Scope.



Copyright 2021 The MathWorks, Inc.



See Also

Blocks

Random Source | Pulse Generator | Constant | Switch | RMS | Time Scope

Simulink Block Examples in Transforms and Spectral Analysis Category

- “Analyze a Subband of Input Frequencies Using Zoom FFT” on page 17-2
- “Group Delay Estimation in Simulink” on page 17-4
- “High Resolution Filter-Bank-Based Power Spectrum Estimation” on page 17-7
- “Estimate Data Series Using Forward Linear Predictor” on page 17-11
- “Continuous-Time Transfer Function Estimation” on page 17-13

Analyze a Subband of Input Frequencies Using Zoom FFT

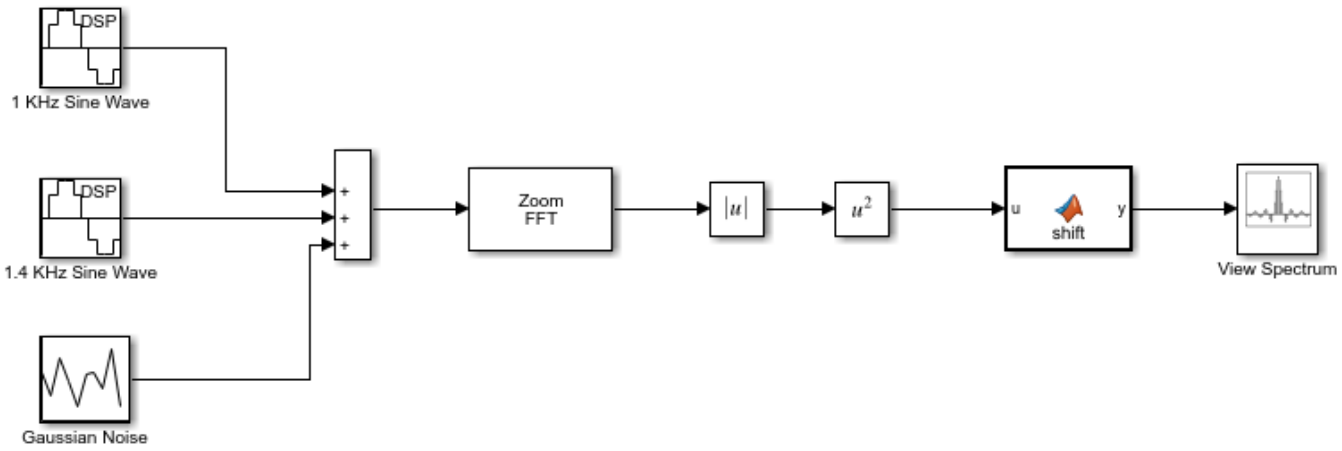
The `Zoom FFT` block implements zoom FFT based on the multirate multistage bandpass filter designed in “Complex Bandpass Filter Design” on page 4-190. If you specify the center frequency and the decimation factor, the `Zoom FFT` block designs and applies the filter to the input signal. Using zoom FFT, 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.

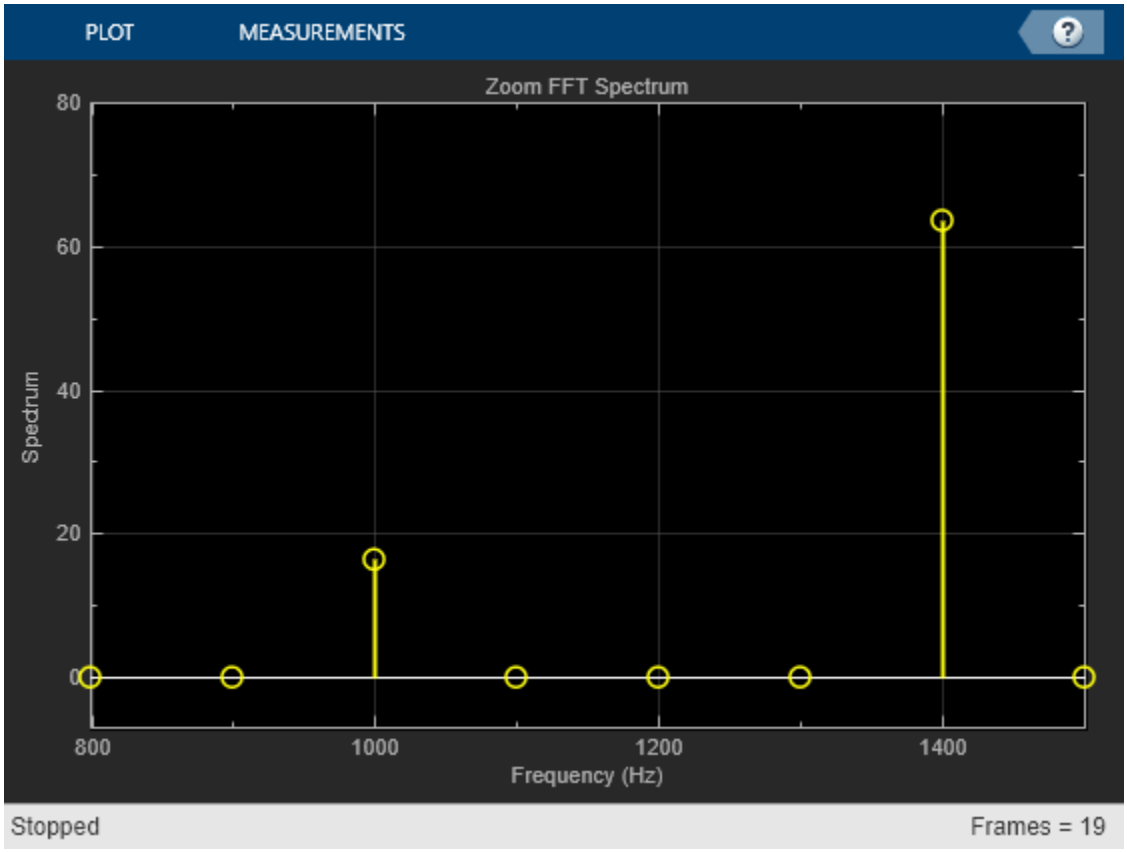
```
model = 'zoomfftEx';
open_system(model)
```



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
Sine Wave | Random Source | Array Plot | Math Function | Abs | Zoom FFT

Group Delay Estimation in Simulink

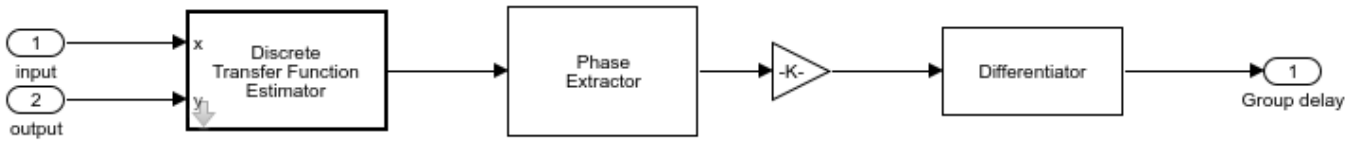
This example shows how to estimate the group delay of a filter in Simulink®.

To estimate the group delay of the filter, extract the phase response and compute its negative derivative with respect to frequency. Group delay is defined as $-d(\phi(f))/d(f)$.

The Example Model

The Simulink model GroupDelayEstimator estimates the group delay of the given filter using the following blocks:

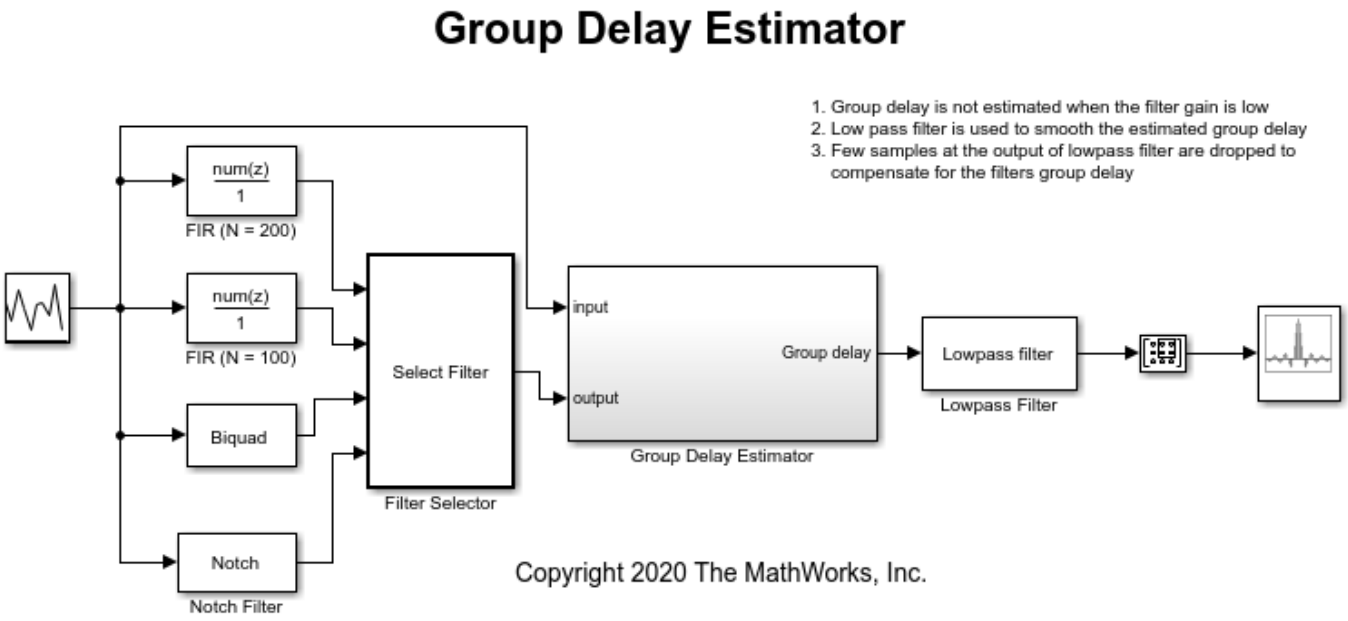
1. Discrete Transfer Function Estimator - Estimate the discrete transfer function of the filter using its input and output.
2. Phase Extractor - Extracts the phase response from the filter transfer function estimate.
3. Gain (Simulink) - Scales the phase response to denormalize frequency to 0 to half the sample-rate. In this case sample-rate is set to 44.1kHz. Negative of this value is used for estimating the group delay in number of samples.
4. Differentiator Filter - Takes the derivative of the phase with respect to frequency.
5. Array Plot - View the group delay of the filter in number of samples.



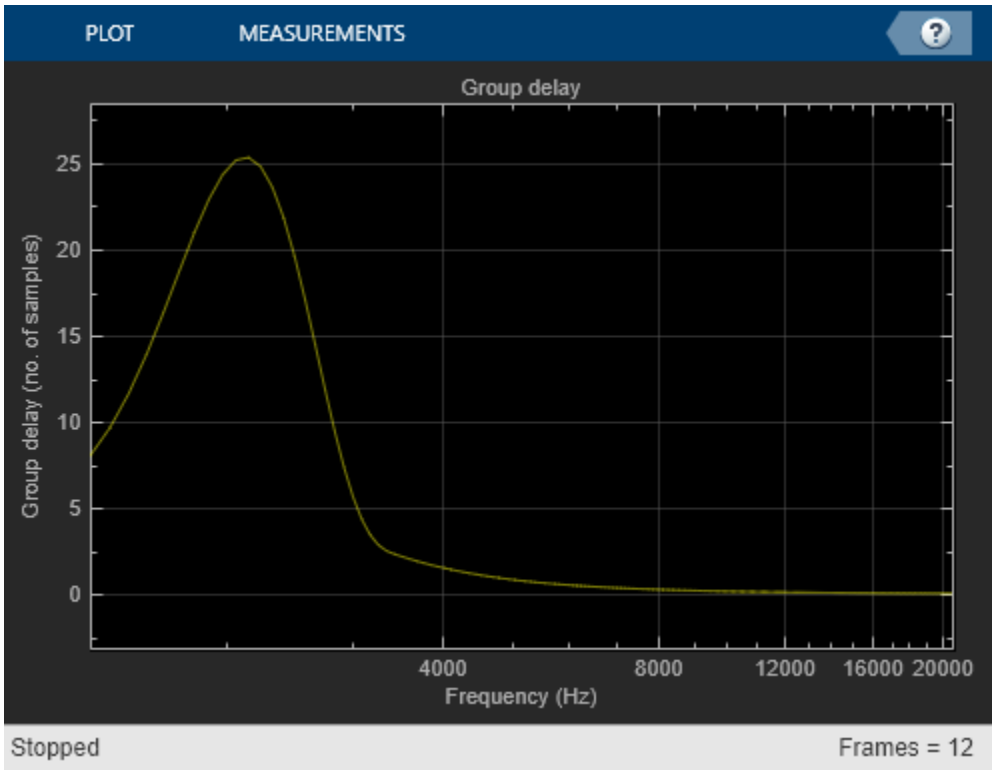
The **Filter Selector** block will allow you to choose from different filters. The group delay estimator output is noisy. To filter the noise the output of the estimator is passed through a low pass filter so that the estimated group delay can be smoothly visualized. This lowpass filter has a group delay which is equal to half the filter order. Hence initial few samples are dropped to compensate for this group delay.

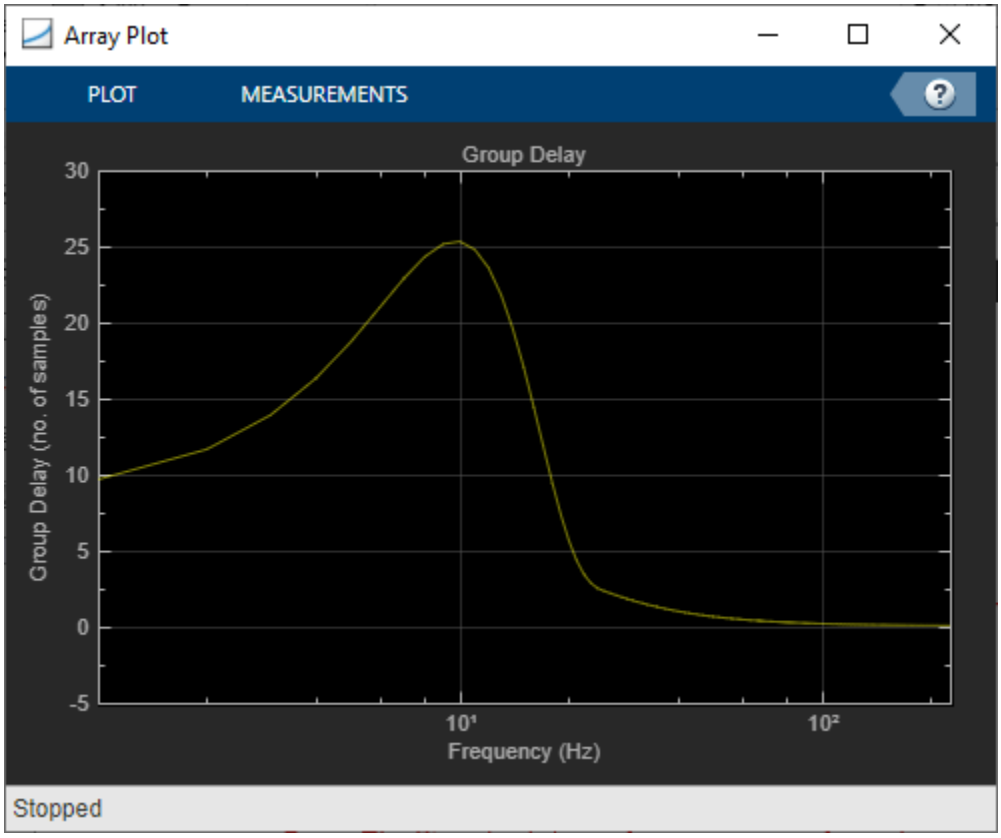
Exploring the Example

Open and run the model. You can see the group delay of the selected filter in number of samples in the **Array Plot** block. The theoretical value for a linear phase FIR Filter block is half the order of the filter. For Biquad filter and Notch filter the theoretical group delay can be visualized by opening the block mask and clicking the **View Filter Response** button. For Notch filter block, you can tune the notch frequency and see the group delay change accordingly.



The Lowpass filter block after the group delay estimator is used to smooth the estimate. Tune the cutoff frequency of this filter and notice the noise in the estimated group delay.





See Also

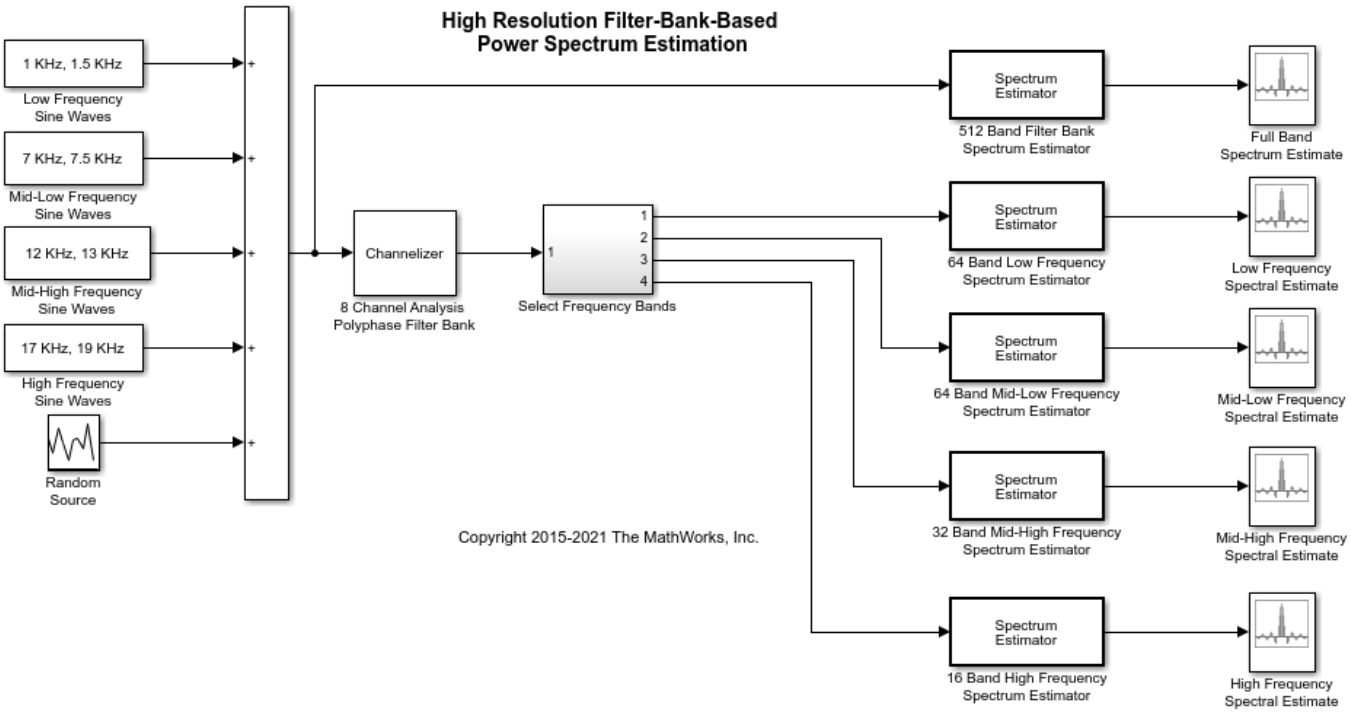
Blocks

Discrete Transfer Function Estimator | Phase Extractor | Gain | Differentiator Filter | Array Plot | Biquad Filter | Lowpass Filter

High Resolution Filter-Bank-Based Power Spectrum Estimation

This example shows how to perform high resolution spectral analysis by using an efficient polyphase filter bank sometimes referred to as a channelizer.

Open the Model



Exploring the Example

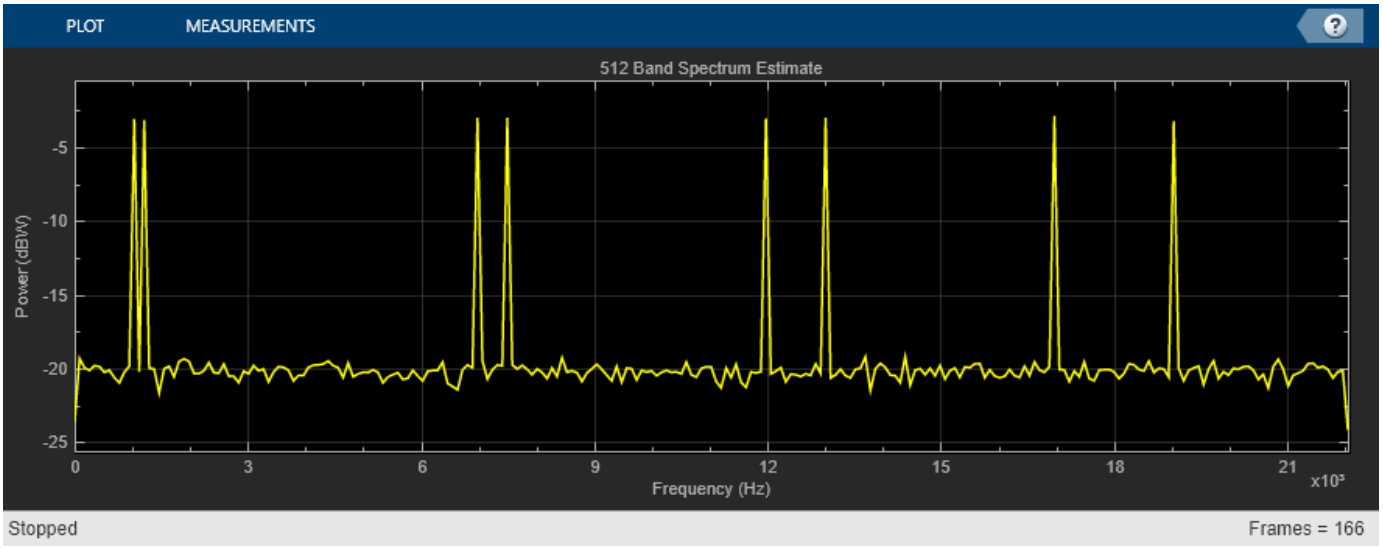
This example compares full band and subband spectral estimators. Both spectral estimators use polyphase filter bank (channelizer) implementations which provide good resolution and improved accuracy when compared to Welch-method-based estimators. See “High Resolution Spectral Analysis in MATLAB” on page 4-16 for a comparison between filter bank and Welch-based spectral estimators.

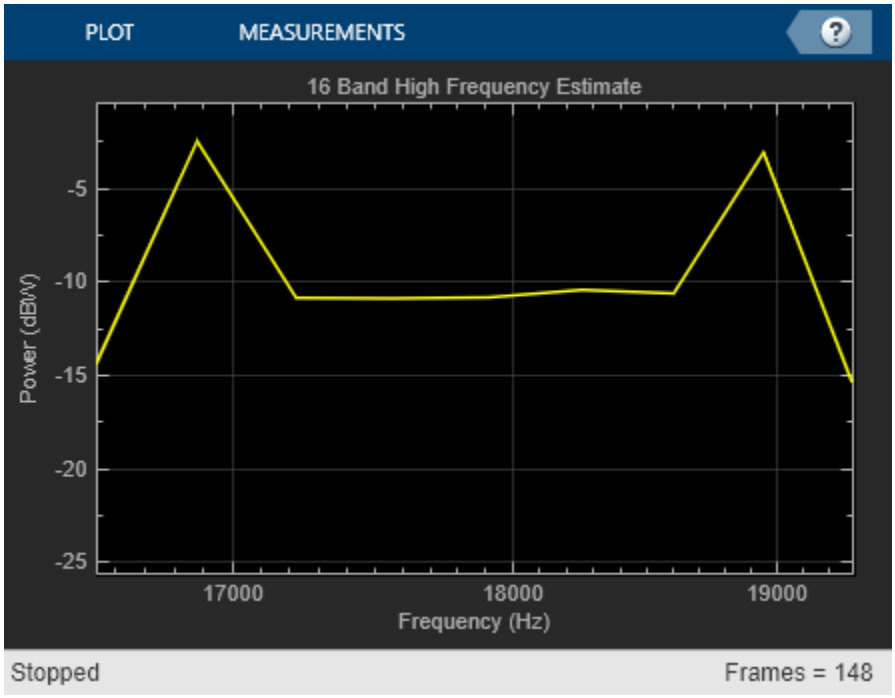
In this example, the full band estimator requires a 512-phase polyphase FIR filter and a 512-point FFT in order to compute the spectral estimate. The sinusoid frequencies in each subband are spaced further apart as the frequency increases. The idea is to setup a case in which higher frequency resolution is required at the low frequency band and lower resolution is required at higher frequency bands.

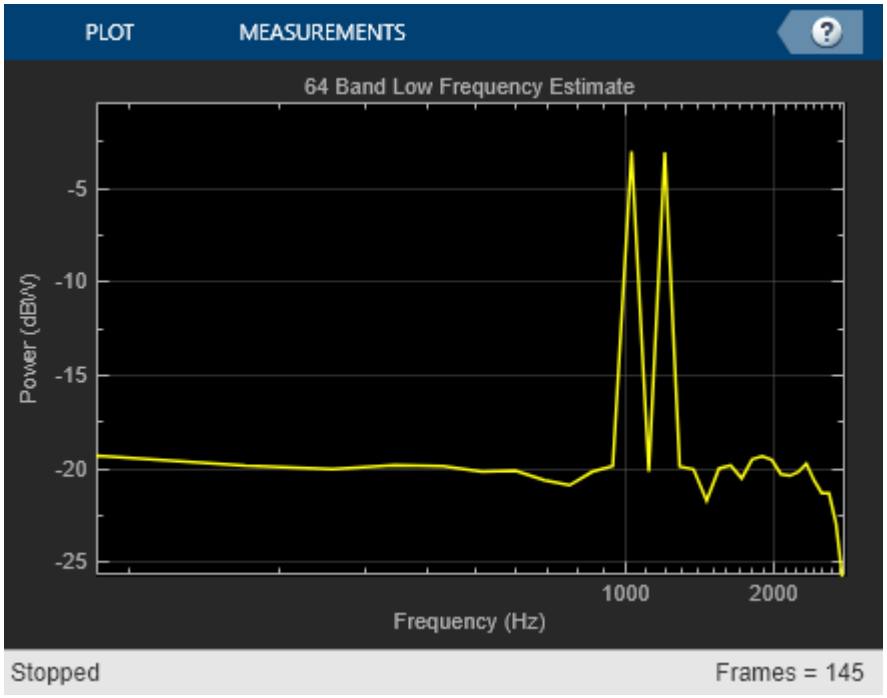
The sub-band approach is more efficient. It uses an 8-phase polyphase FIR filter and an 8-point FFT to divide the broadband signal into 8 sub-bands. Subsequently, a 64 band filter bank estimator (itself containing a 64-phase polyphase FIR filter and a 64-point FFT) is used with the low frequency sub-band in order to compute the spectral estimate with the same resolution as the full band estimator. The same implementation is used for the mid-low frequency band.

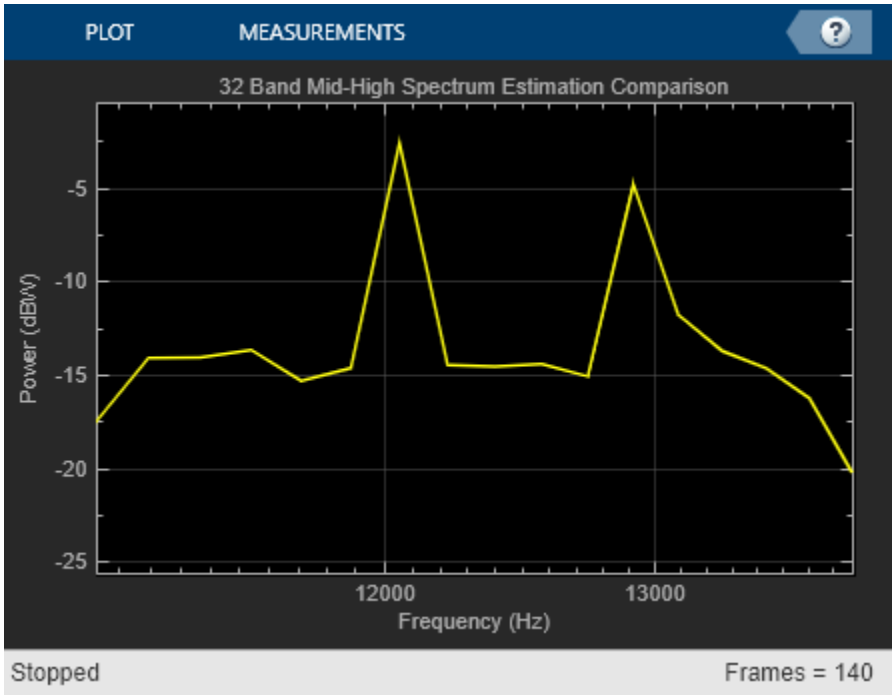
Simulate the Model

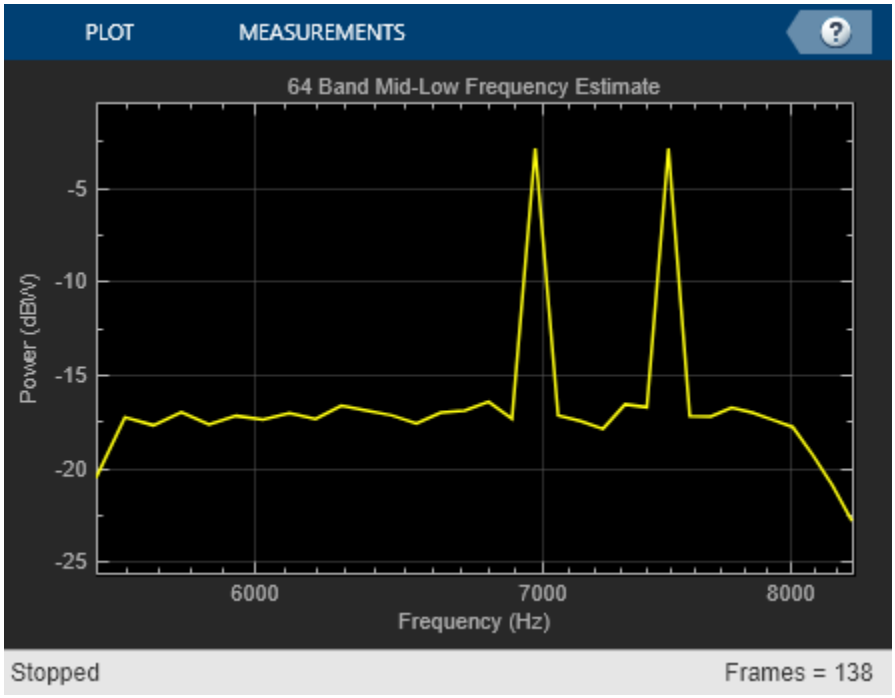
For the mid-high frequency band, the sinusoids are spaced further apart. Hence, a 32 band filter bank estimator is used. For the high-frequency band, we use a 16 band filter bank estimator.


PLOT
MEASUREMENTS
512 Band Spectrum Estimate
Power (dBW)
-5
-10
-15
-20
-25
0
3
6
9
12
15
18
21
x10³
Frequency (Hz)
Stopped
Frames = 166


PLOT
MEASUREMENTS
16 Band High Frequency Estimate
Power (dBW)
-5
-10
-15
-20
-25
17000
18000
19000
Frequency (Hz)
Stopped
Frames = 148


PLOT
MEASUREMENTS
64 Band Low Frequency Estimate
-5
-10
-15
-20
-25
Power (dB/W)
1000
2000
Frequency (Hz)
Stopped
Frames = 145


PLOT
MEASUREMENTS
32 Band Mid-High Spectrum Estimation Comparison
-5
-10
-15
-20
-25
Power (dB/W)
12000
13000
Frequency (Hz)
Stopped
Frames = 140



References

harris, f. j. **Multirate Signal Processing for Communications Systems**, Prentice Hall PTR, 2004.

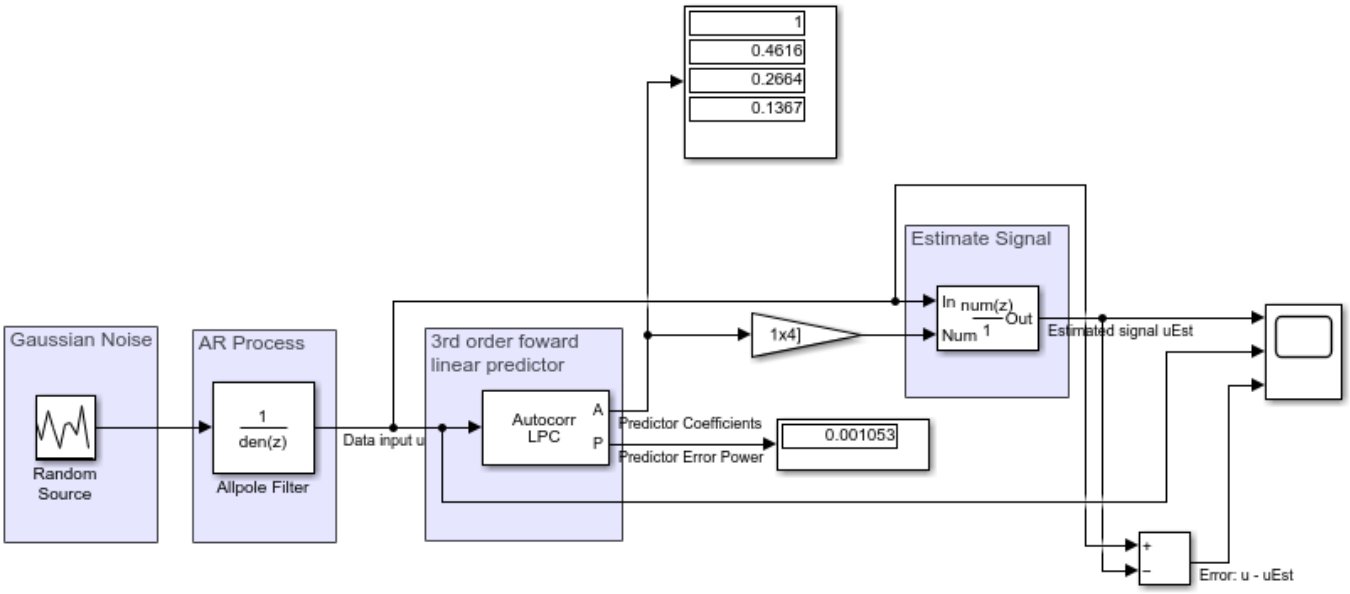
Estimate Data Series Using Forward Linear Predictor

Estimate a data series using a third-order forward linear predictor, which is modeled by the Autocorrelation LPC block. The block computes the predictor polynomial coefficients by minimizing the prediction error in the least squares sense. Estimate the future values of the signal using these predictor coefficients and past values of the signal.

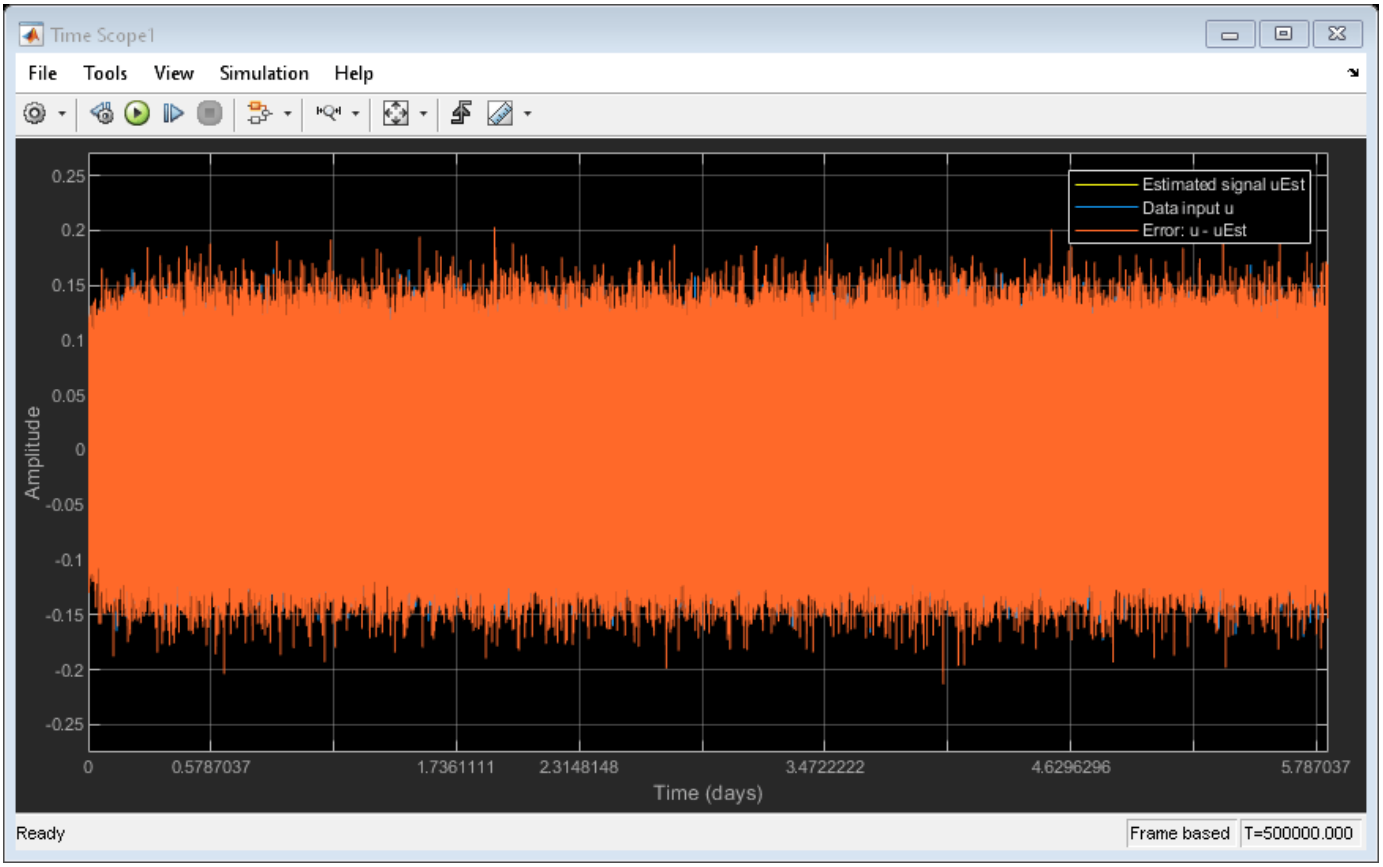
First, create the signal data *u* as the output of a fifth order autoregressive (AR) process, driven by normalized white Gaussian noise. The noise has zero mean and a variance of 0.001.

Pass the signal data *u* to the Autocorrelation LPC block. The block acts as a third order forward linear predictor and outputs the coefficients of the predictor polynomial. Using these coefficients, estimate the future values of the signal *u*. The prediction error power is output through the Port P of the Autocorrelation LPC block. The value of this power is very close to the value of the input white noise variance indicating that the predictor polynomial coefficients are optimal. To enable the port P, you must select the **Ouput prediction error power (P)** in the block dialog box.

Compare the original signal and the estimated signal.



Copyright 2021 The MathWorks, Inc.



See Also

Blocks

Random Source | Discrete FIR Filter | Autocorrelation LPC | Time Scope | Display

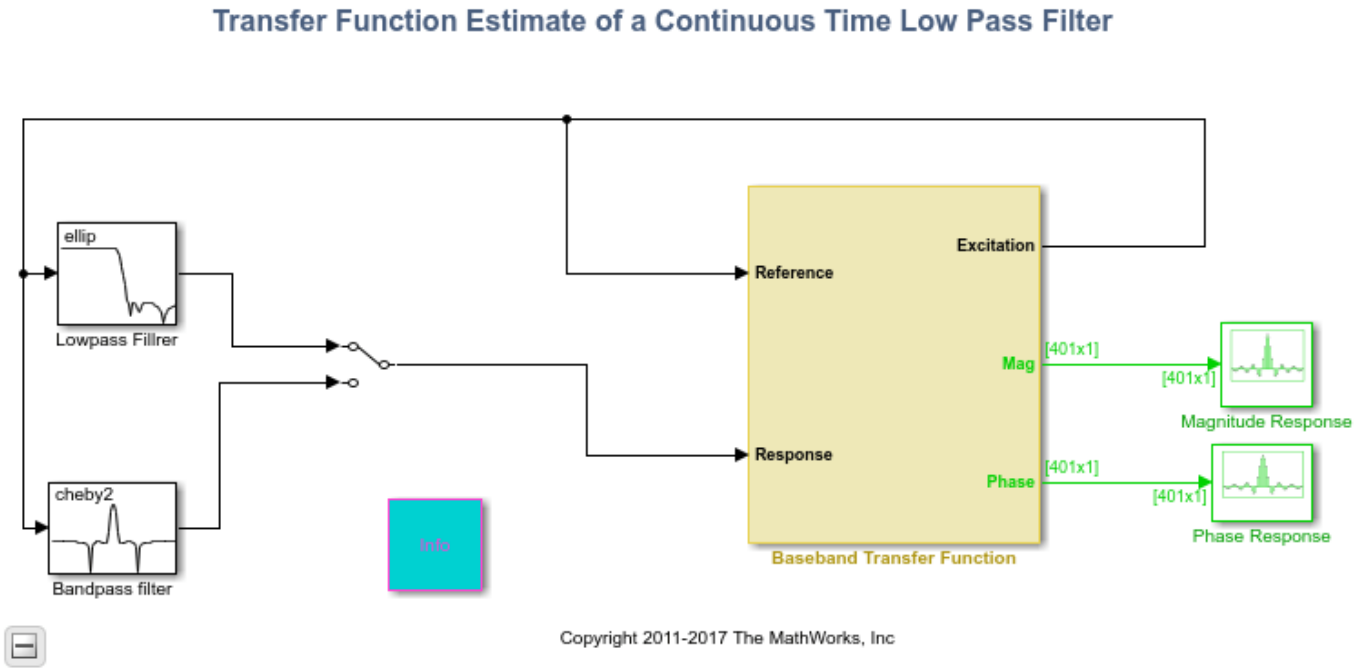
External Websites

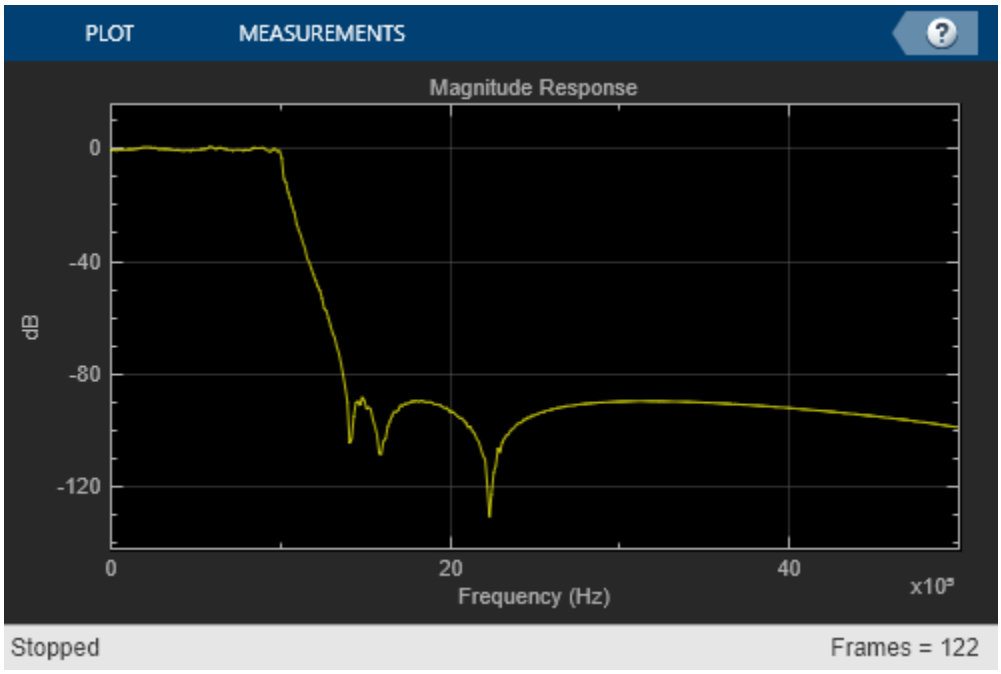
- https://authors.library.caltech.edu/25063/1/S00086ED1V01Y200712SPR003.pdf

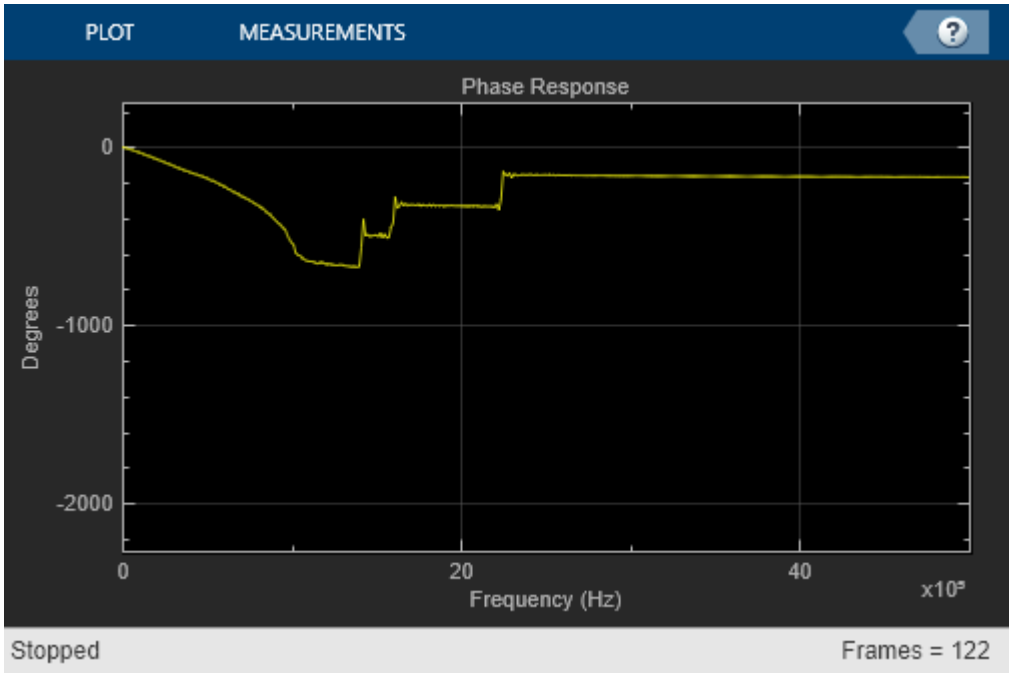
Continuous-Time Transfer Function Estimation

This example shows how to use the Discrete Transfer Function Estimator block to estimate the magnitude and phase response of a continuous-time analog filter.

Example Model







Exploring the Example

This example estimates the magnitude and phase response of two analog filters:

a. A lowpass, eighth-order elliptical filter with a passband edge frequency of 1 MHz.

b. A bandpass, eighth-order Chebyshev II filter with lower and upper stopband edge frequencies of 2 MHz and 3 MHz, respectively.

You can use the manual switch block to toggle between the two filters while the model is running.

You can specify the following on the dialog of the Baseband Transfer Function block: the alias-free signal bandwidth (BW), the FFT length used in transfer estimation, and the number of spectral averages used to smooth the estimate.

The excitation input is a random signal with uniform distribution. We feed the excitation through the filters under test. We pass both excitation and filtered signals through anti-alias analog filters with bandwidth BW Hz, and then we then transform them to discrete-time signals using Zero-Order Hold blocks with a sampling frequency of 2.56*BW Hz. The discrete excitation and output signals are fed to the Discrete Transfer Estimator block. The phase response is computed using a Phase Extractor block. We use array plot scopes to visualize the estimated filter magnitude and phase response.

Transforms, Estimation, and Spectral Analysis

Learn about transforms, estimation and spectral analysis.

- “Transform Time-Domain Data into Frequency Domain” on page 18-2
- “Transform Frequency-Domain Data into Time Domain” on page 18-5
- “Linear and Bit-Reversed Output Order” on page 18-7
- “Calculate Channel Latencies Required for Wavelet Reconstruction” on page 18-9
- “Estimate the Power Spectrum in MATLAB” on page 18-15
- “Estimate the Power Spectrum in Simulink” on page 18-28
- “Estimate the Transfer Function of an Unknown System” on page 18-44
- “View the Spectrogram Using Spectrum Analyzer” on page 18-52
- “Spectral Analysis” on page 18-61
- “Streaming Power Spectrum Estimation Using Welch's Method” on page 18-65

Transform Time-Domain Data into Frequency Domain

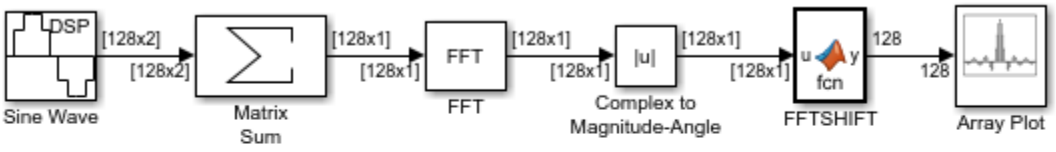
This example shows how to transform time-domain data into the frequency domain using the FFT block.

Note: To open the example and the associated models, you must have MATLAB® open. Click on the **Open Script** button while you have this page open on the MATLAB help browser.

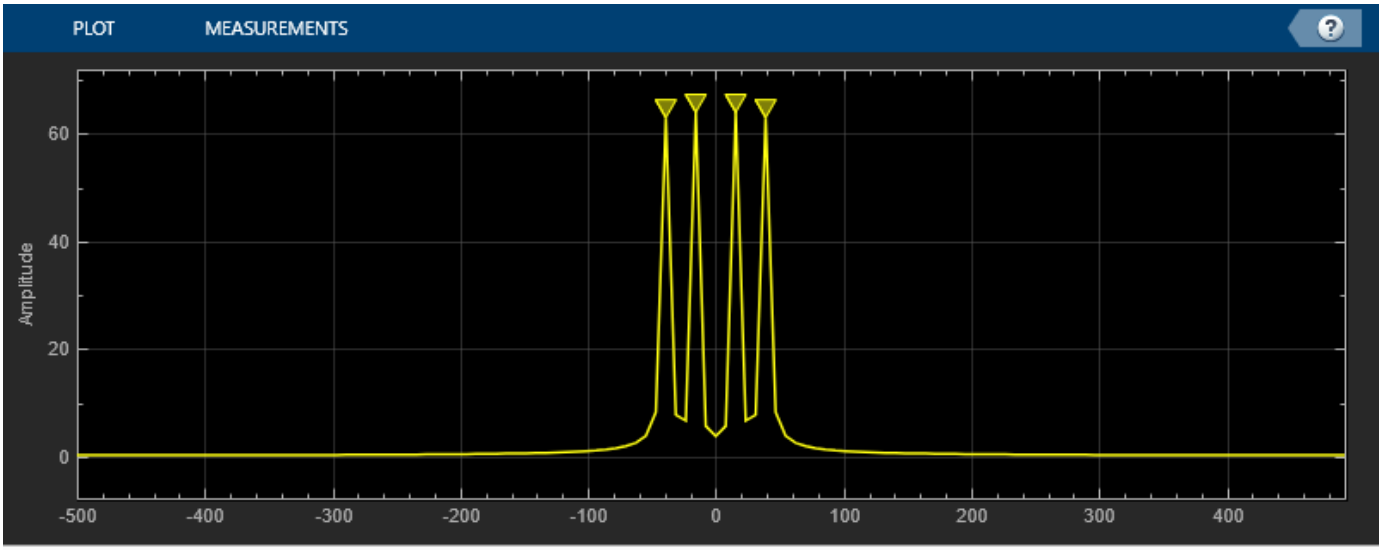
Use the Sine Wave block to generate two sinusoids, one at 15 Hz and the other at 40 Hz. Use the Matrix Sum block to add the sinusoids point-by-point to generate the compound sinusoid:

$$u = sin(2 * 15\pi t) + sin(2 * 40\pi t)$$

Transform this sinusoid into the frequency domain using an FFT block. See the `ex_fft_tut` model:



Copyright 2018 The MathWorks, Inc.



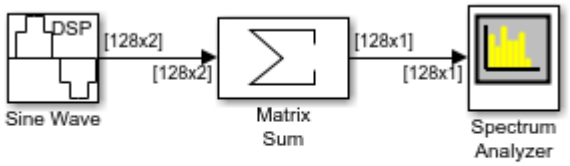
Peaks

Channel 1	P1	P2	P3	P4
X	1.5625e+1	-1.5625e+1	3.9063e+1	-3.9063e+1
Y	6.3983e+1	6.3983e+1	6.2981e+1	6.2981e+1

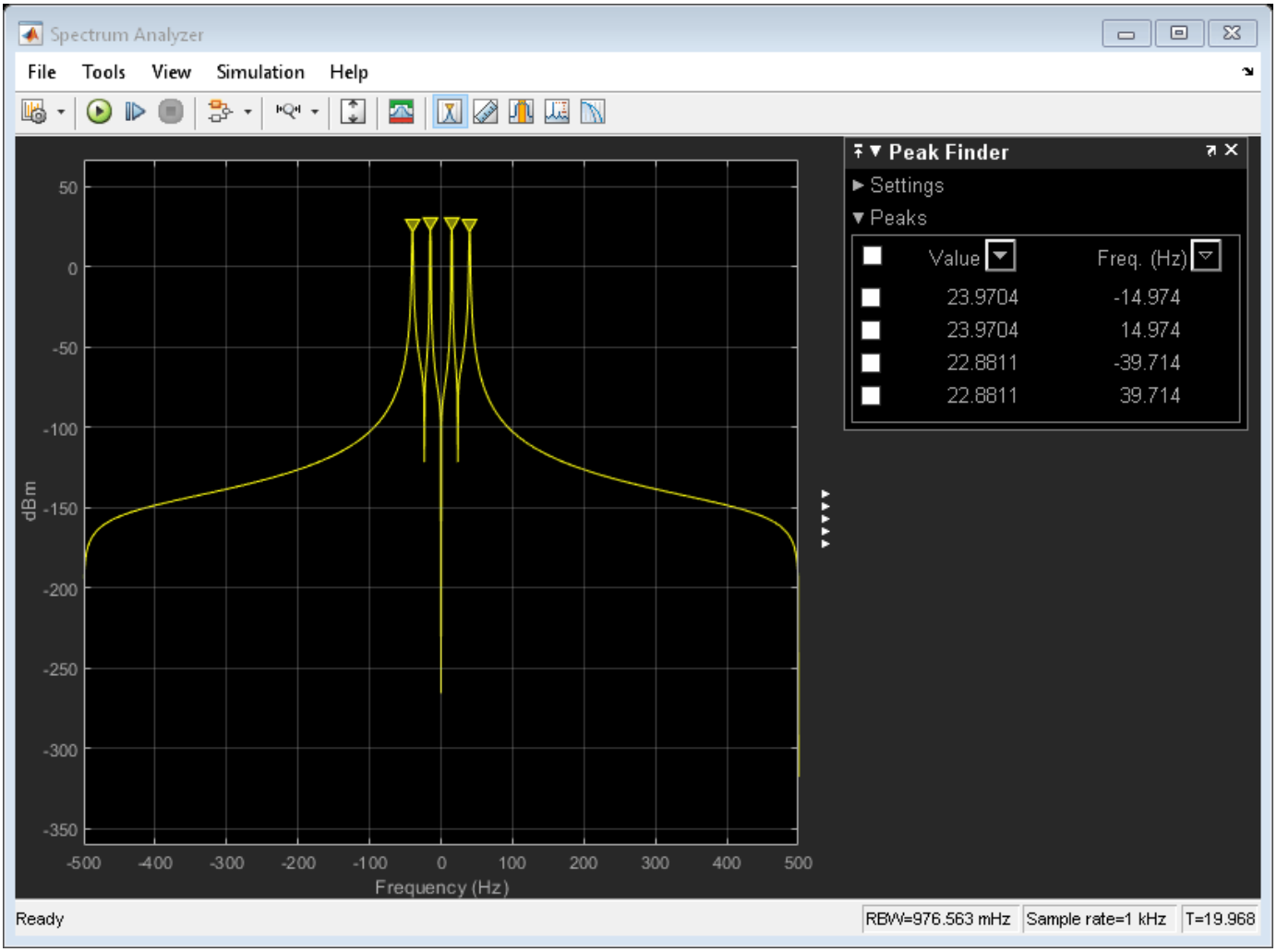
Stopped Frames = 0

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.

See Also

Functions
`fftshift`

Blocks
Sine Wave | Matrix Sum | FFT | Spectrum Analyzer | Array Plot | Complex to Magnitude-Angle

More About

- “Transform Frequency-Domain Data into Time Domain” on page 18-5

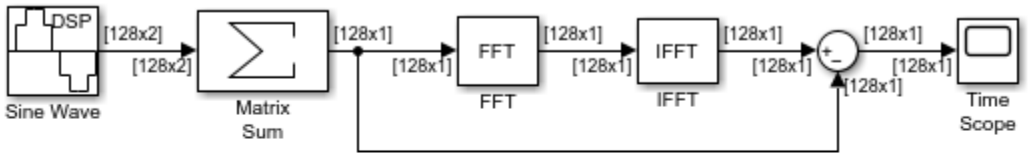
Transform Frequency-Domain Data into Time Domain

When you want to transform frequency-domain data into the time domain, use the IFFT block.

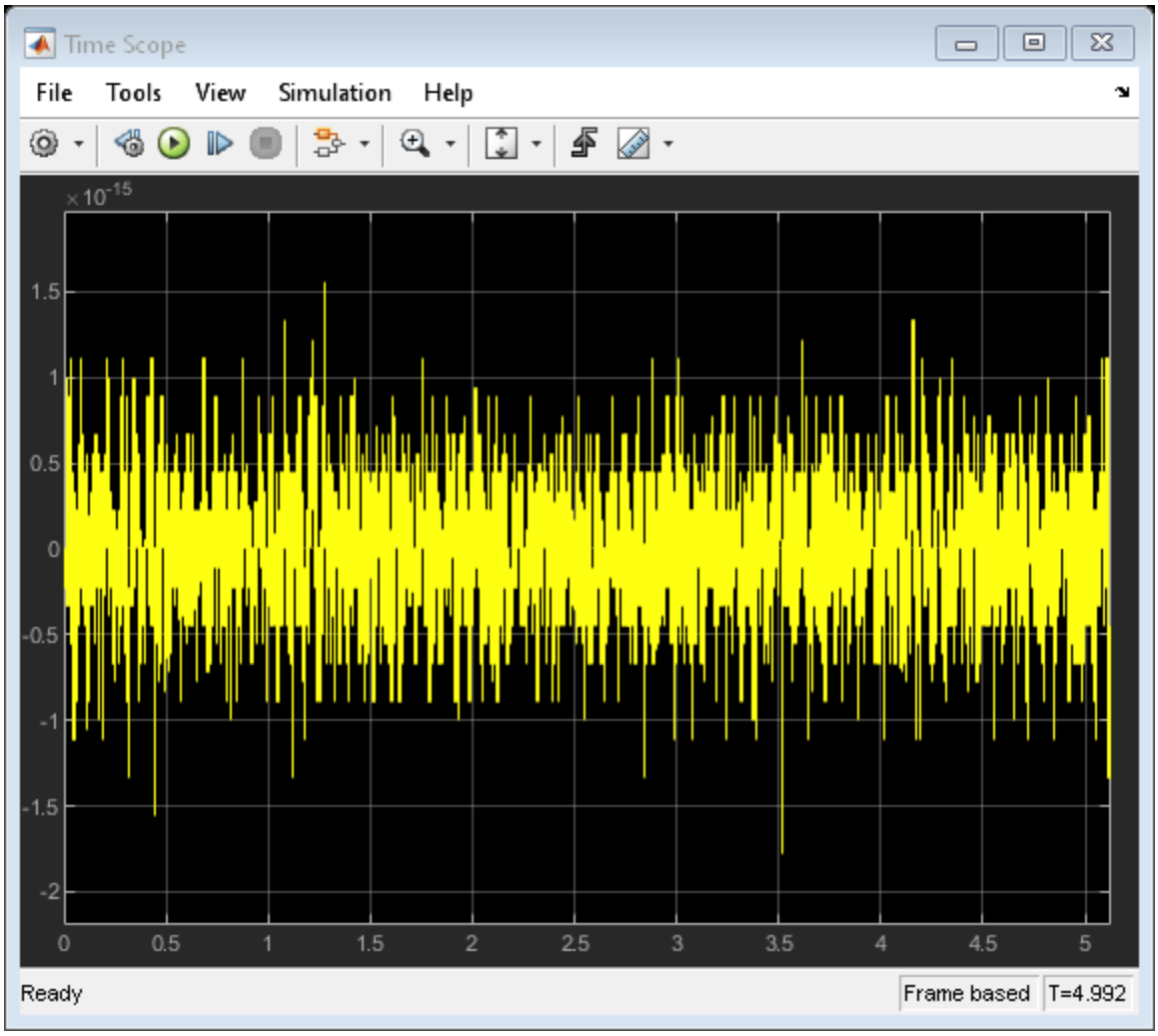
Use the Sine Wave block to generate two sinusoids, one at 15 Hz and the other at 40 Hz. Use a Matrix Sum block to add the sinusoids point-by-point to generate the compound sinusoid:

$$u = sin(30\pi t) + sin(80\pi t)$$

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}.

See Also

Blocks

Sine Wave | Matrix Sum | FFT | IFFT | Time Scope

More About

- “Transform Time-Domain Data into Frequency Domain” on page 18-2

Linear and Bit-Reversed Output Order

In this section...
“FFT and IFFT Blocks Data Order” on page 18-7 “Find the Bit-Reversed Order of Your Frequency Indices” on page 18-7

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.

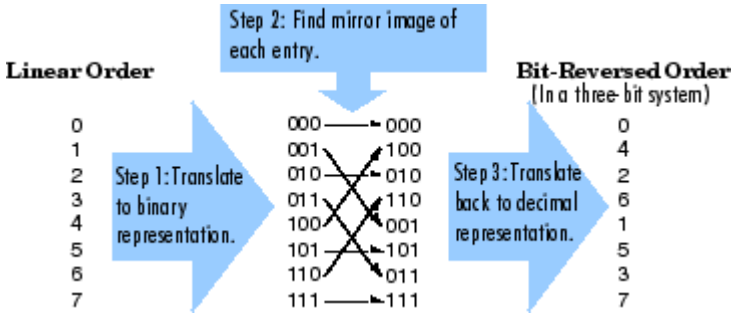
The input to the IFFT block can be in linear or bit-reversed order. Therefore, you do not have to alter the ordering of your data before transforming it back into the time domain. However, the IFFT block may run more quickly when the input is provided in bit-reversed order.

Find the Bit-Reversed Order of Your Frequency Indices

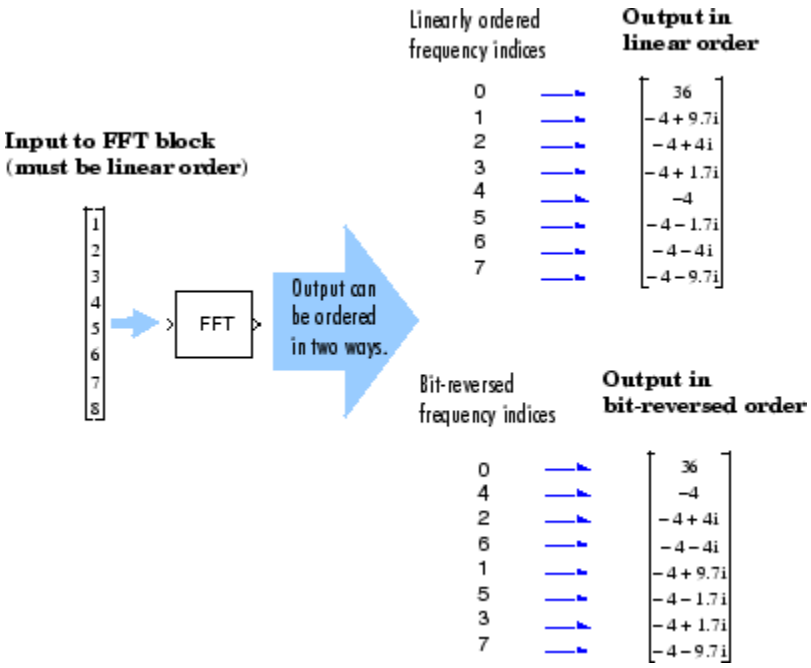
Two numbers are bit-reversed values of each other when the binary representation of one is the mirror image of the binary representation of the other. For example, in a three-bit system, one and four are bit-reversed values of each other, since the three-bit binary representation of one, 001, is the 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

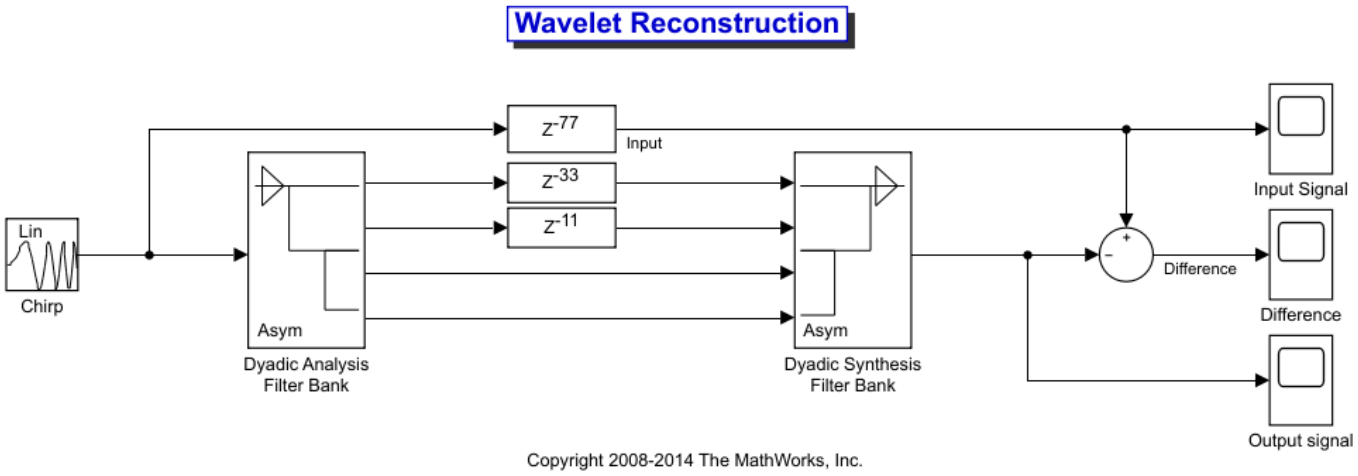
In this section...

“Analyze Your Model” on page 18-9
“Calculate the Group Delay of Your Filters” on page 18-10
“Reconstruct the Filter Bank System” on page 18-11
“Equalize the Delay on Each Filter Path” on page 18-12
“Update and Run the Model” on page 18-13
“References” on page 18-14

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:

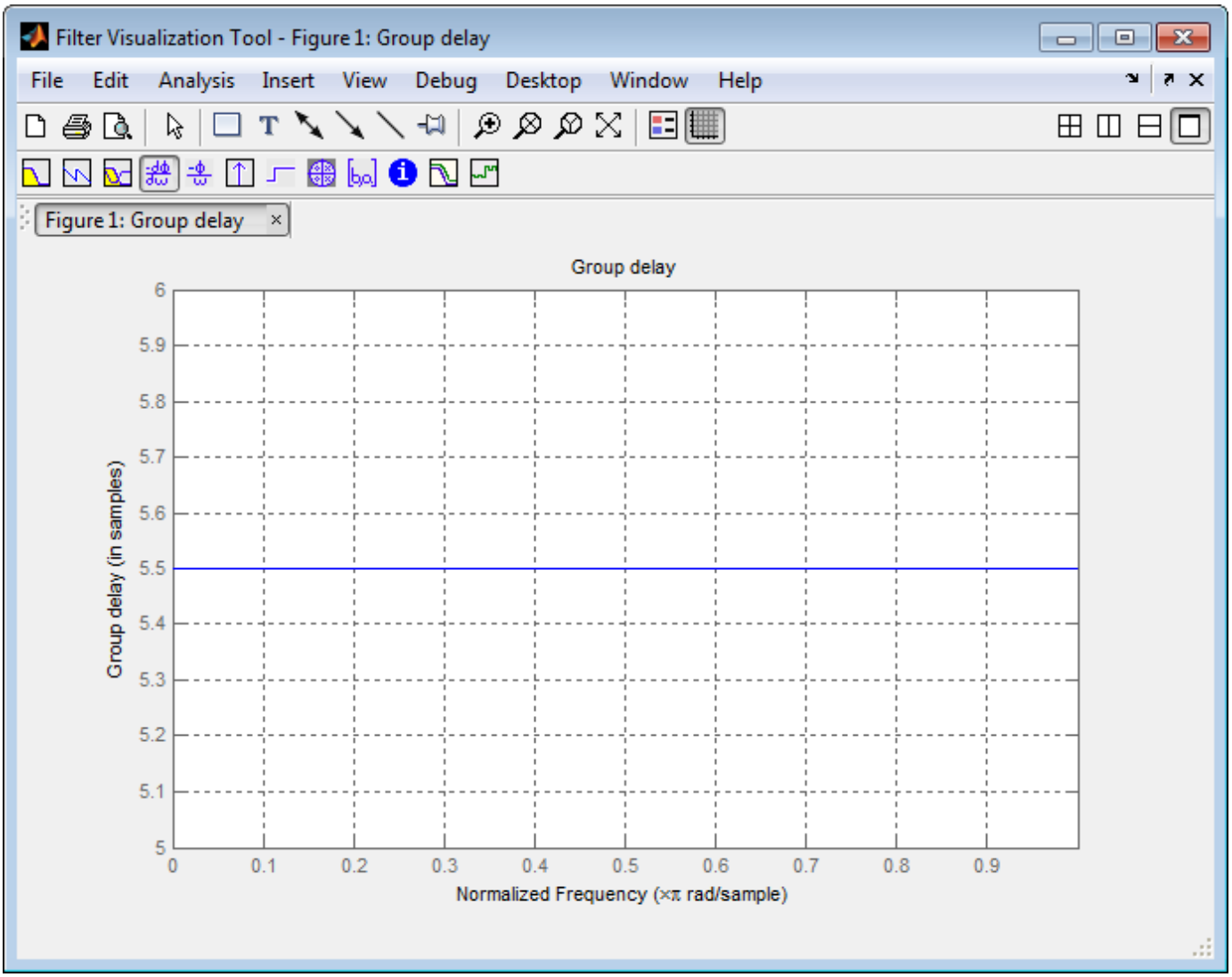
```
[Lo_D, Hi_D, Lo_R, Hi_R] = wfilters('bior3.5')
```

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.



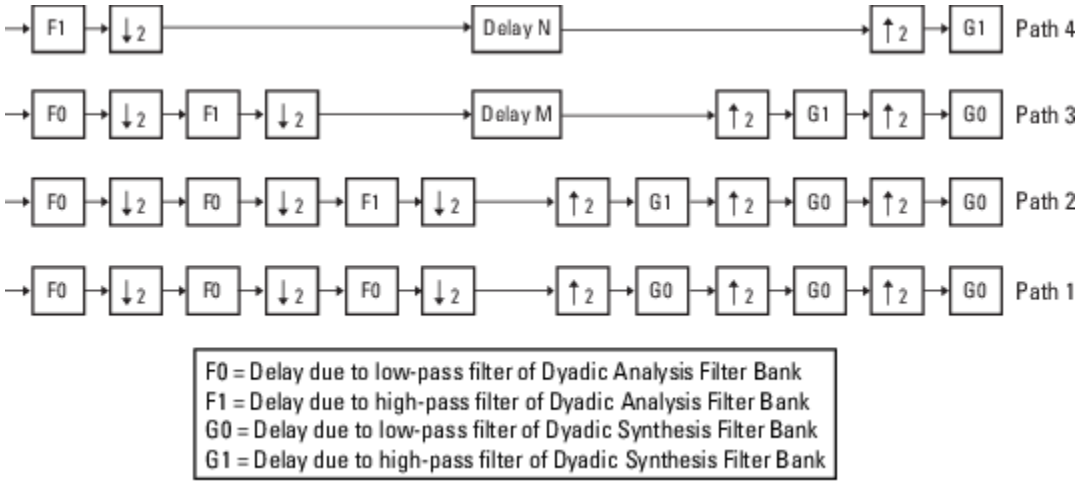
Note Repeat this procedure to analyze the group delay of each of the filters in your model. This section does not show the results for each filter in the `ex_wavelets` model because all wavelet filters in this particular example have the same group delay.

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 7-28
- “Dyadic Synthesis Filter Banks” on page 7-30

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.

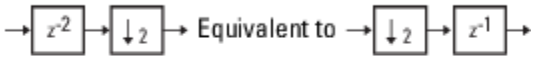


The extra delay values of M and N on paths 3 and 4 in the previous figure ensure that the total delay on each of the four filter paths is identical.

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

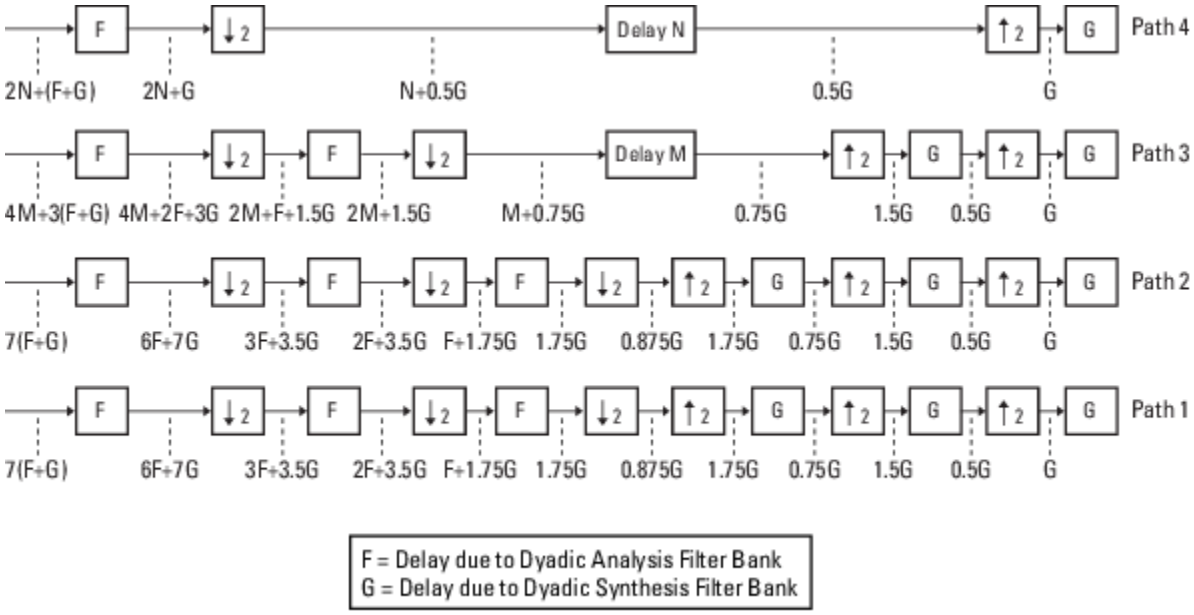


Second Noble Identity

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:

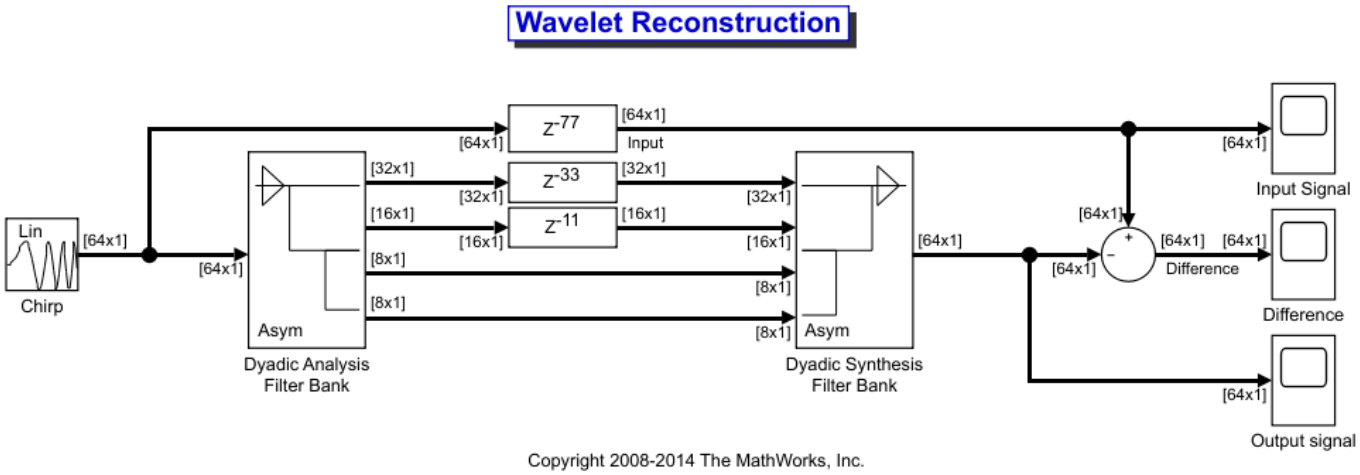
$$\text{Path 3} = \text{Path 1} \Rightarrow 4M + 3(F+G) = 7(F+G)$$
$$\Rightarrow M = F + G$$

$$\text{Path 4} = \text{Path 1} \Rightarrow 2N + (F+G) = 7(F+G)$$
$$\Rightarrow N = 3(F+G)$$

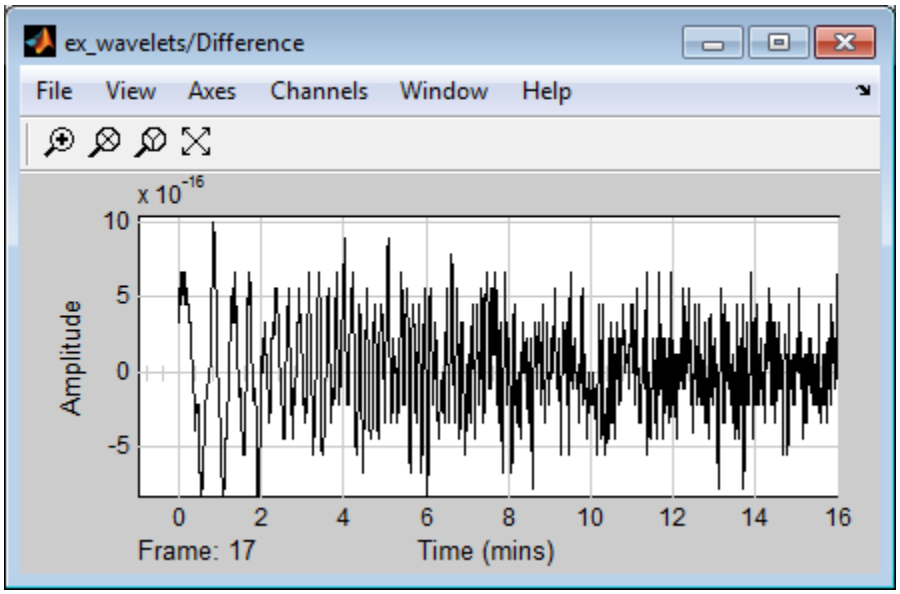
The `fvtool` analysis in step 1 found the group delay of each biorthogonal wavelet filter in this model to be 5.5 samples. Therefore, $F = 5.5$ and $G = 5.5$. By inserting these values into the two previous equations, you get $M = 11$ and $N = 33$. Because the total delay on each filter path must be 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 18-15
“Convert the Power Between Units” on page 18-22
“Estimate the Power Spectrum Using dsp.SpectrumEstimator” on page 18-24

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 18-61. 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.

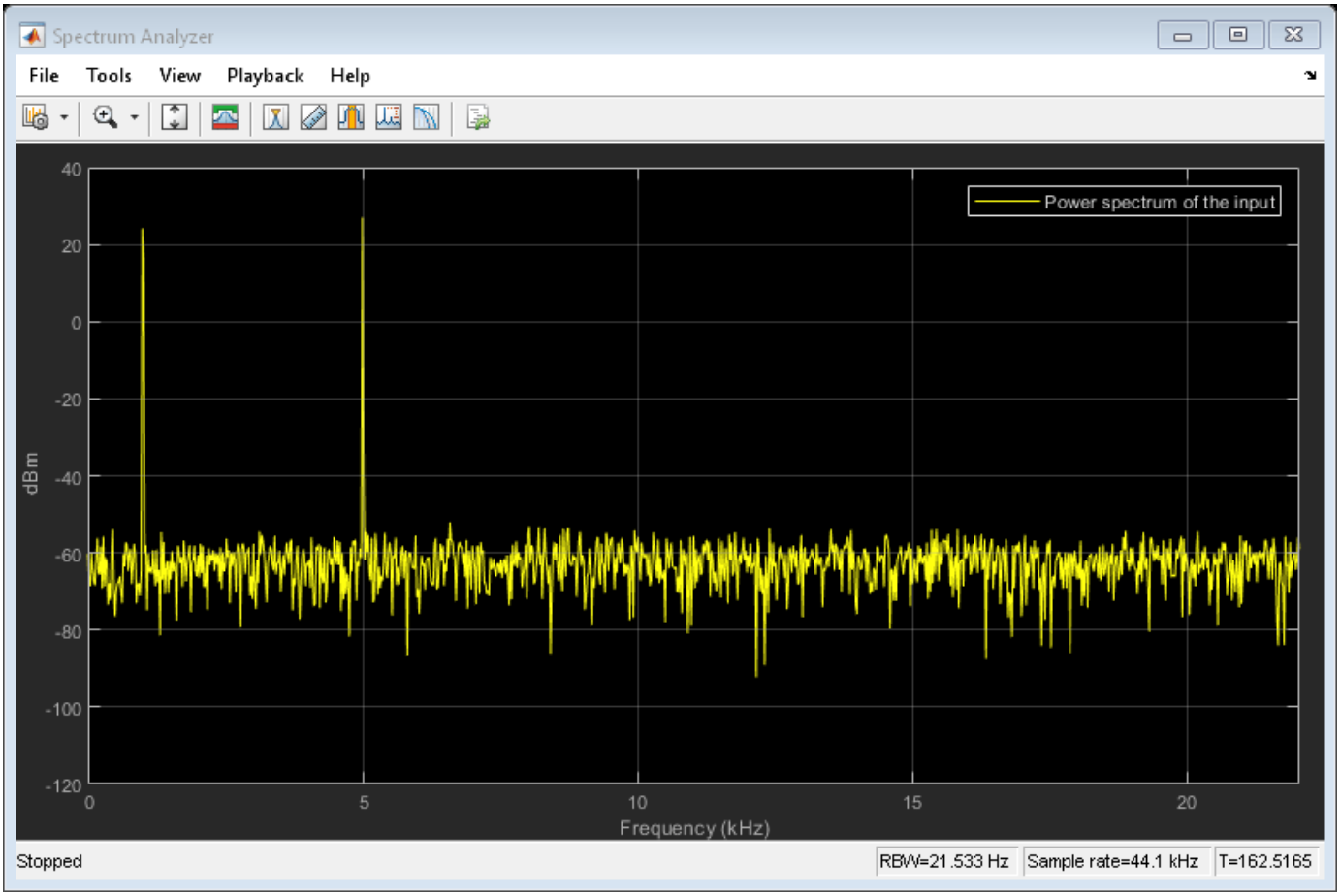
```
Fs = 44100;
Sineobject1 = dsp.SineWave('SamplesPerFrame',1024,'PhaseOffset',10,...
    'SampleRate',Fs,'Frequency',1000);
Sineobject2 = dsp.SineWave('SamplesPerFrame',1024,...
    'SampleRate',Fs,'Frequency',5000);
SA = dsp.SpectrumAnalyzer('SampleRate',Fs,'Method','Filter bank',...
    'SpectrumType','Power','PlotAsTwoSidedSpectrum',false,...
    '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

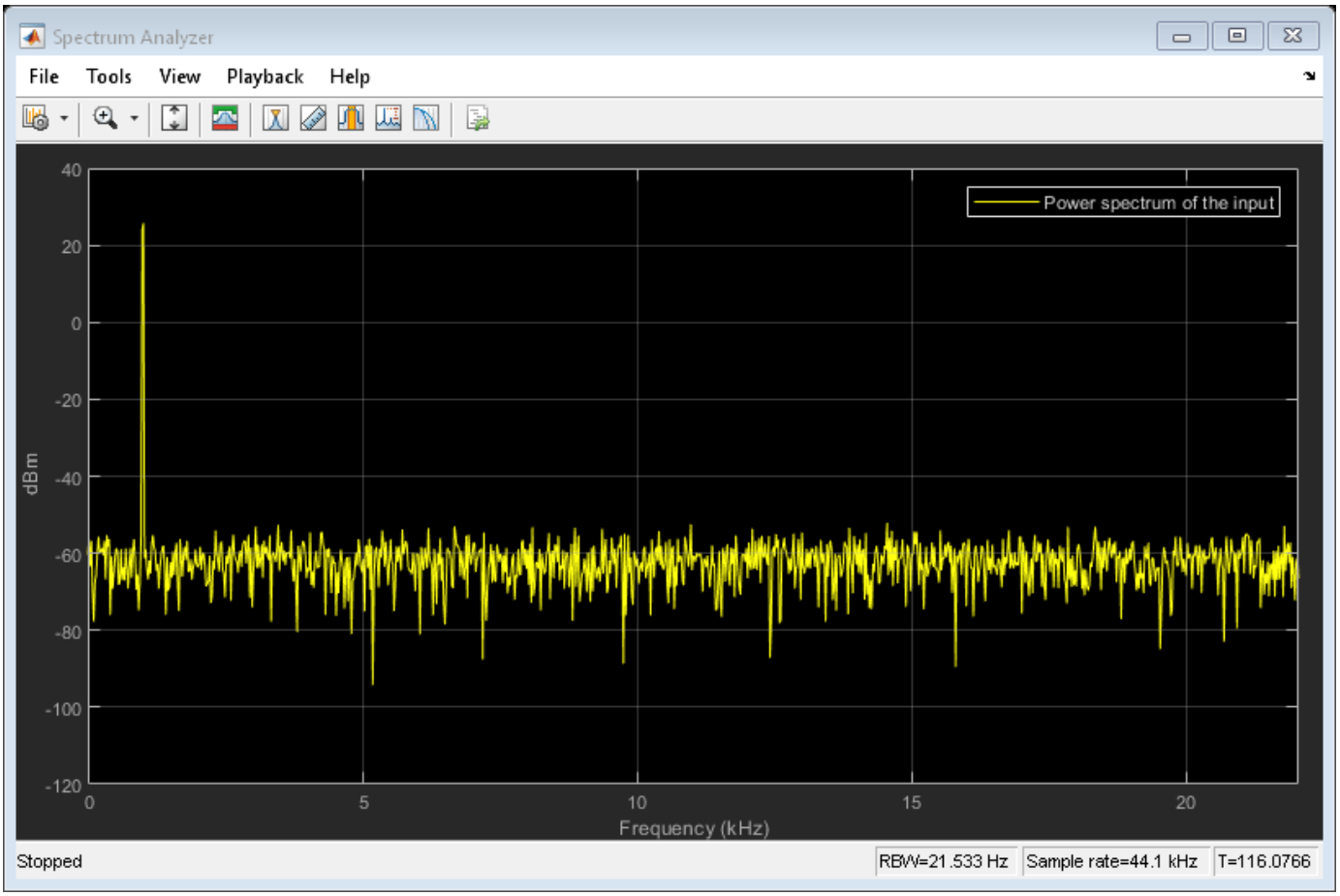
Using this value of RBW, the number of input samples required to compute one spectral update, $N_{samples}$ is given by the following equation: $N_{samples} = \frac{F_s}{RBW}$.

In this example, $N_{samples}$ is 44100/21.53 or 2048 samples.

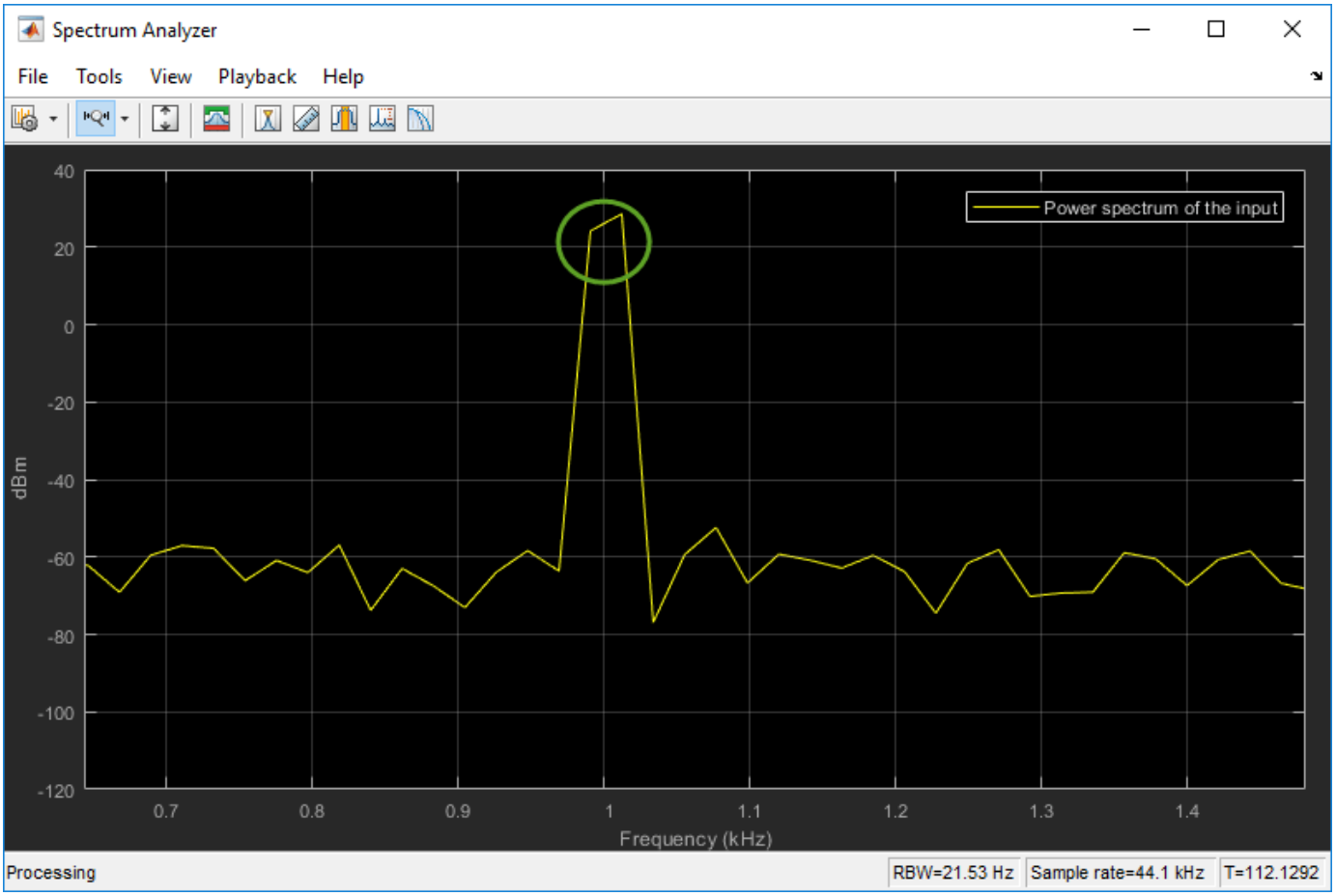
RBW calculated in the 'Auto' mode gives a good frequency resolution.

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);
```

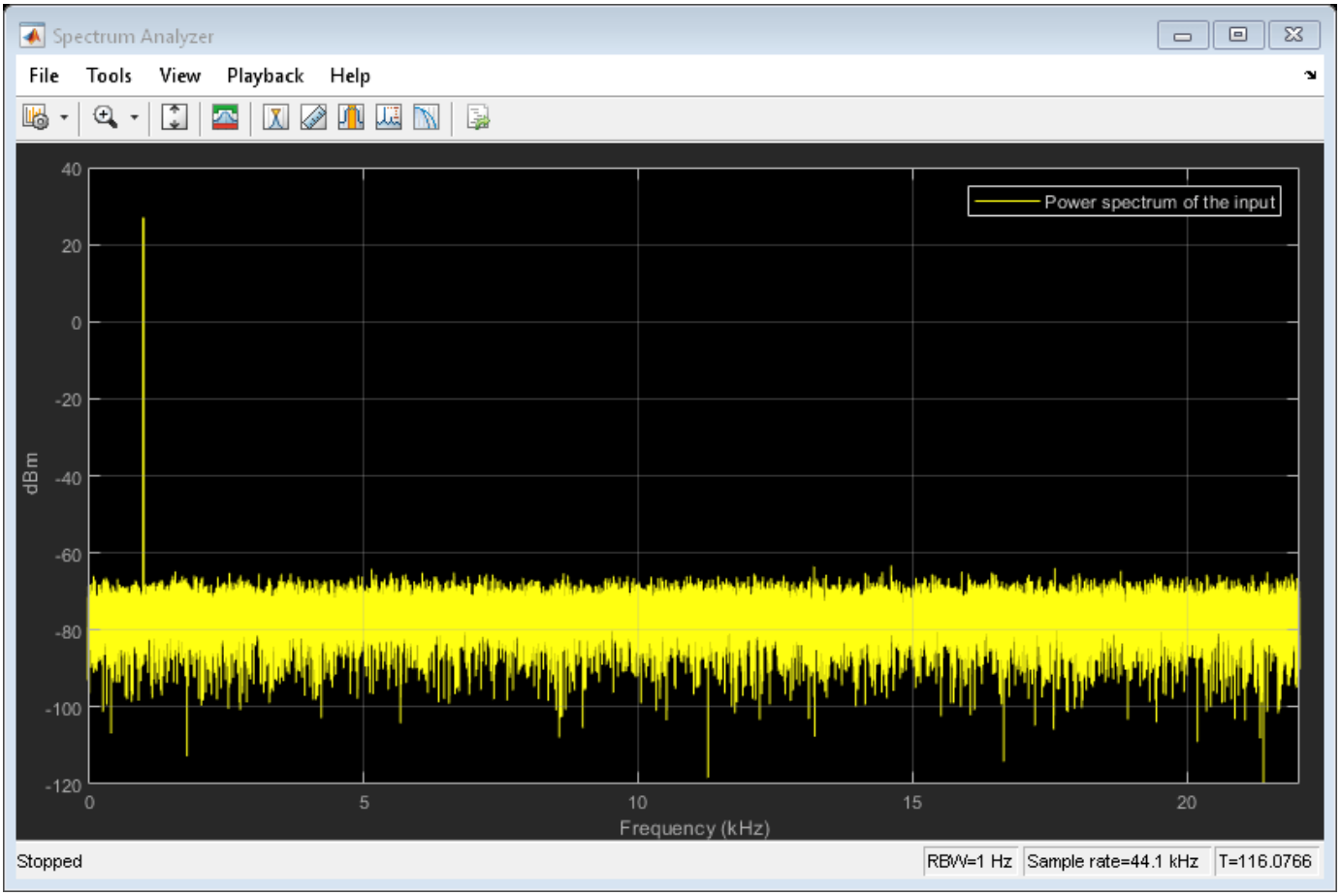


The peaks are not distinguishable.

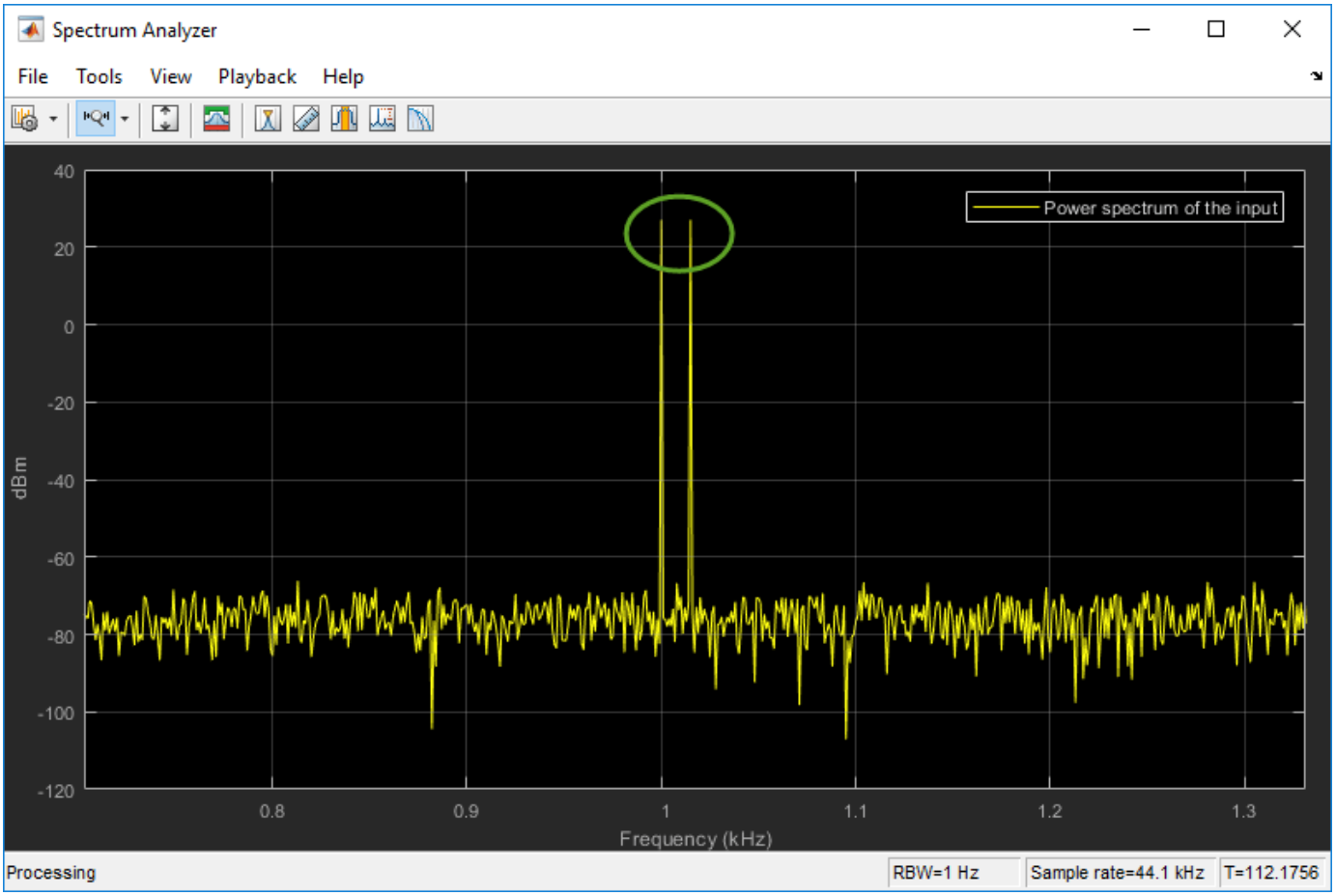


To increase the frequency resolution, decrease RBW to 1 Hz.

```
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);
```



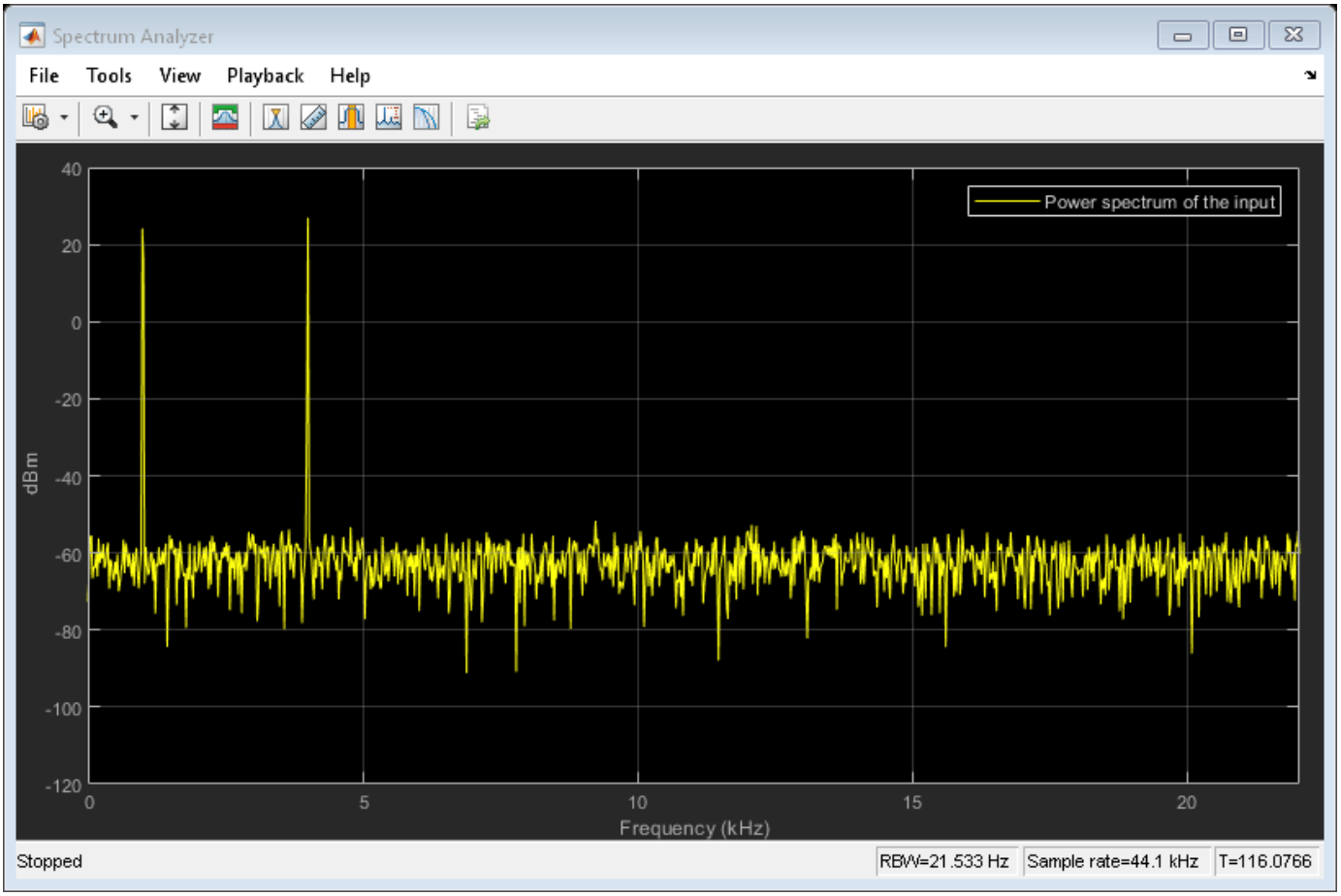
On zooming, the two peaks, which are 15 Hz apart, are now distinguishable.



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.

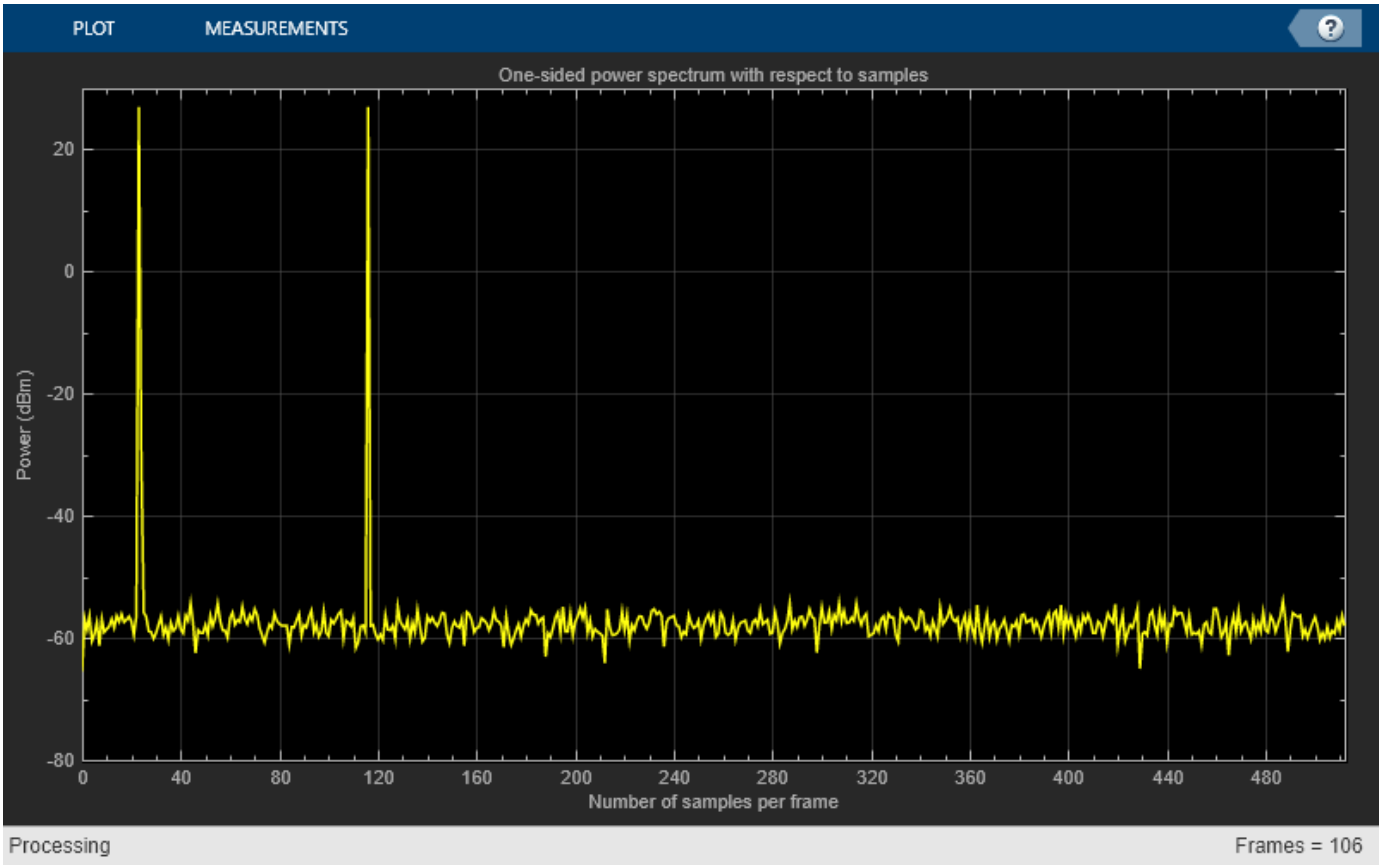
```
Fs = 44100;
Sineobject1 = dsp.SineWave('SamplesPerFrame',1024,'PhaseOffset',10,...
```

```
    '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.

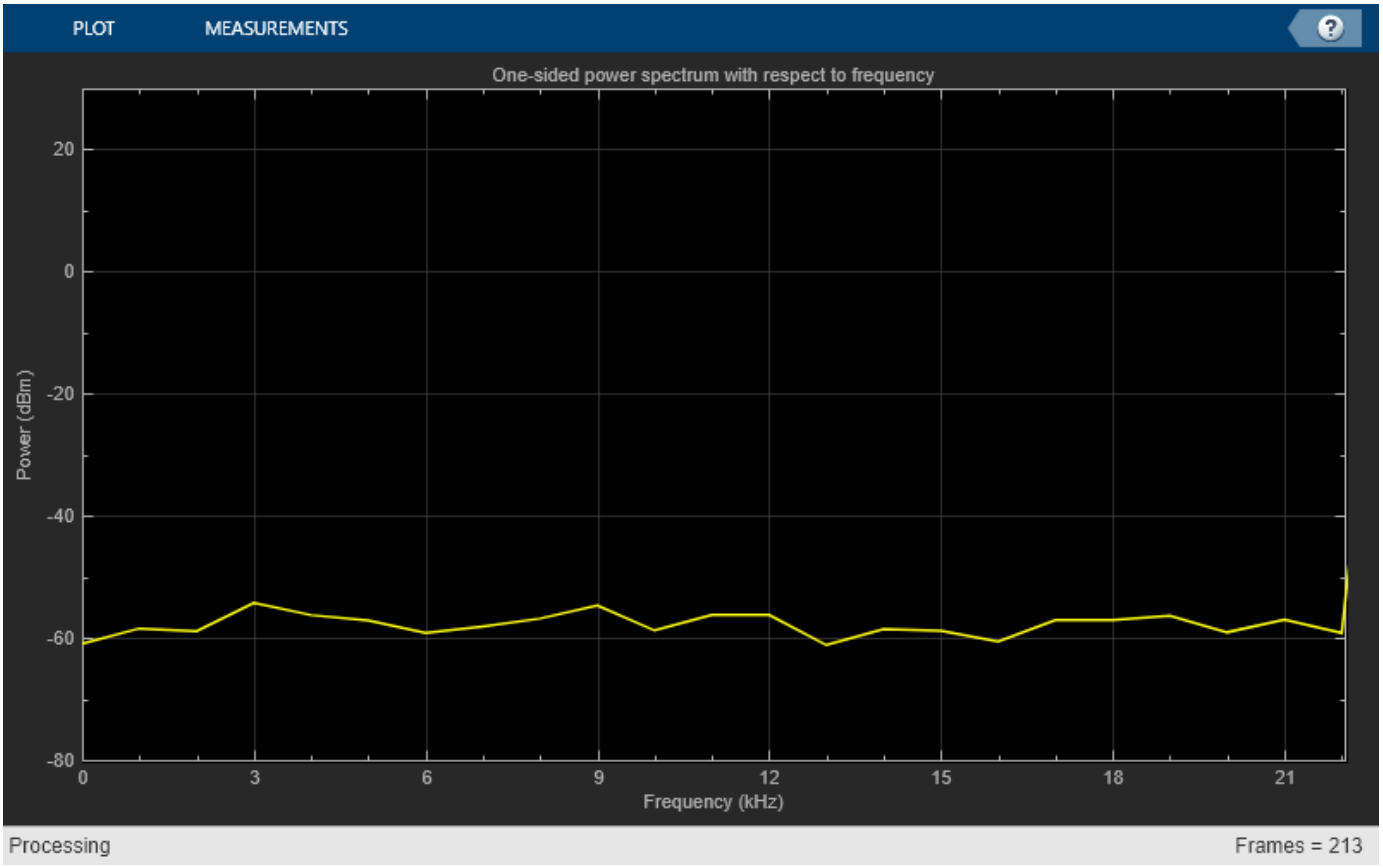
Convert *x*-axis to Represent Frequency

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 18-28
- “Estimate the Transfer Function of an Unknown System” on page 18-44
- “View the Spectrogram Using Spectrum Analyzer” on page 18-52
- “Spectral Analysis” on page 18-61

Estimate the Power Spectrum in Simulink

In this section...
“Estimate the Power Spectrum Using the Spectrum Analyzer” on page 18-28
“Convert the Power Between Units” on page 18-37
“Estimate Power Spectrum Using the Spectrum Estimator Block” on page 18-39

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 18-61. 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 `SpectrumAnalyzerConfiguration` 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.

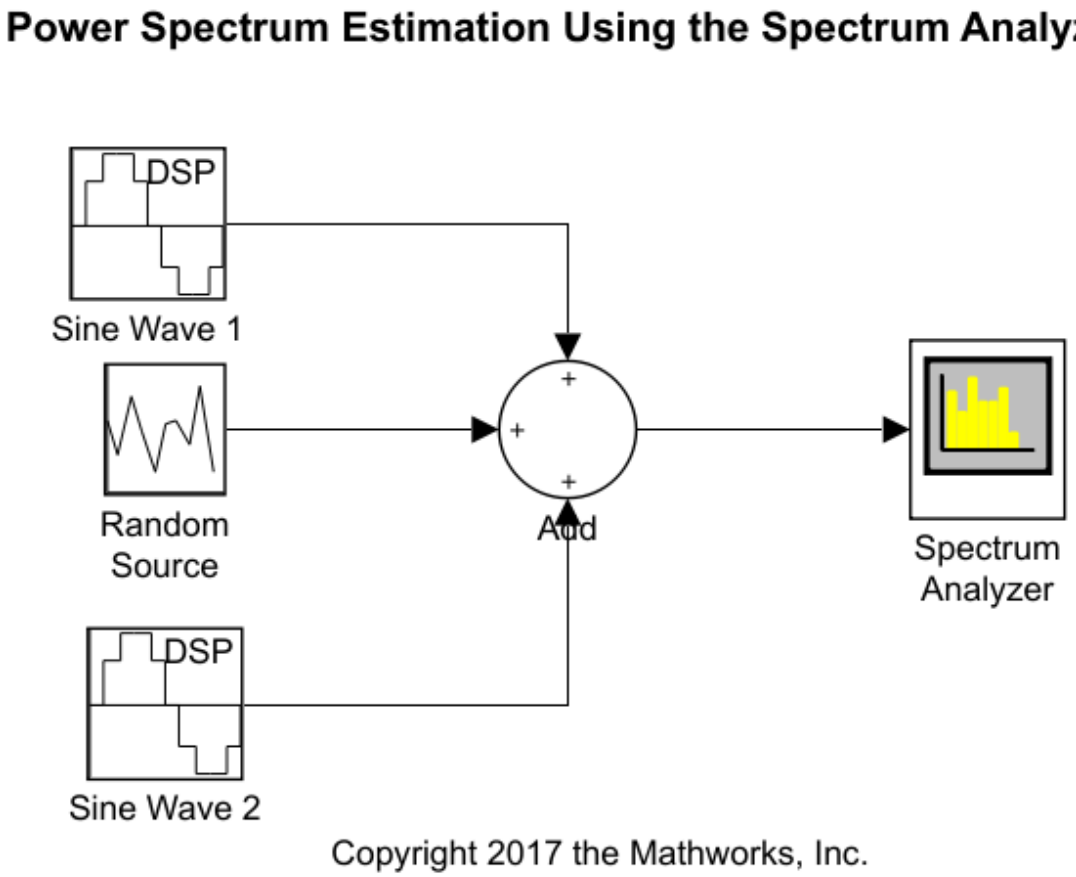
Estimate the Power Spectrum Using the Spectrum Analyzer

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.

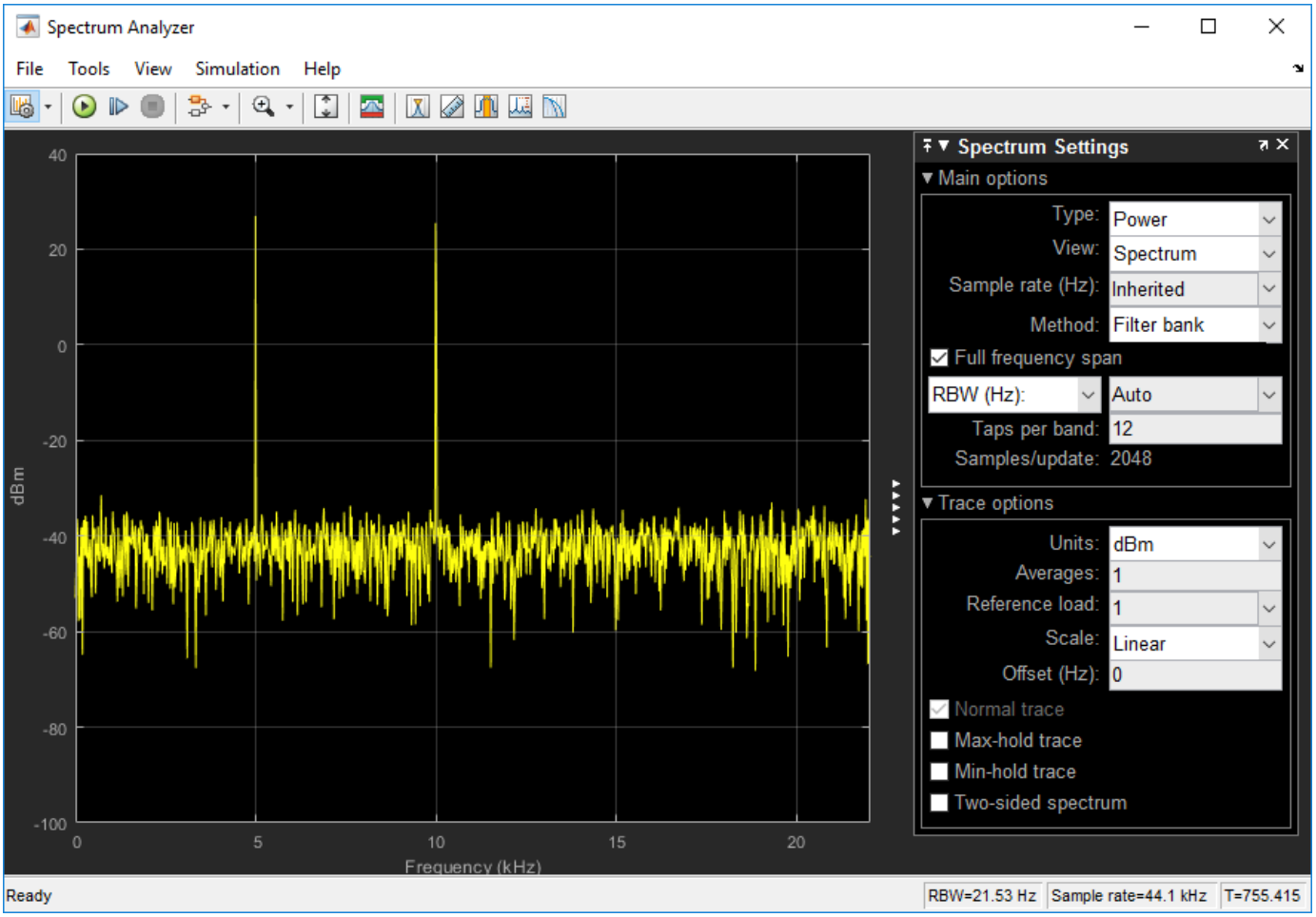


Here are the settings of the blocks in the model.

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.

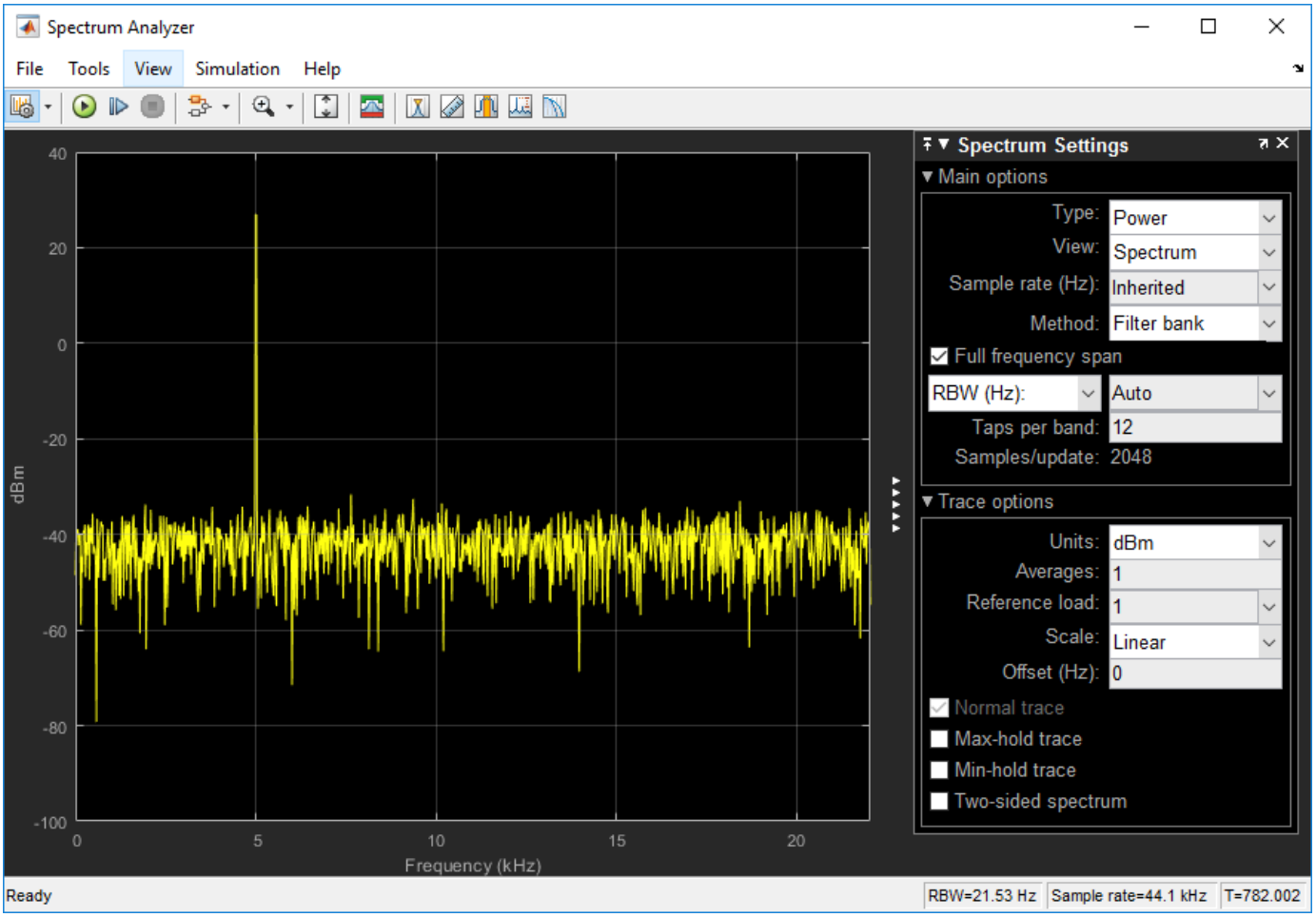


RBW, the resolution bandwidth is the minimum frequency bandwidth that can be resolved by the spectrum analyzer. By default, **RBW (Hz)** is set to `Auto`. In the `Auto` mode, RBW is the ratio of the frequency span to 1024. In a two-sided spectrum, this value is $F_s/1024$, while in a one-sided spectrum, 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.

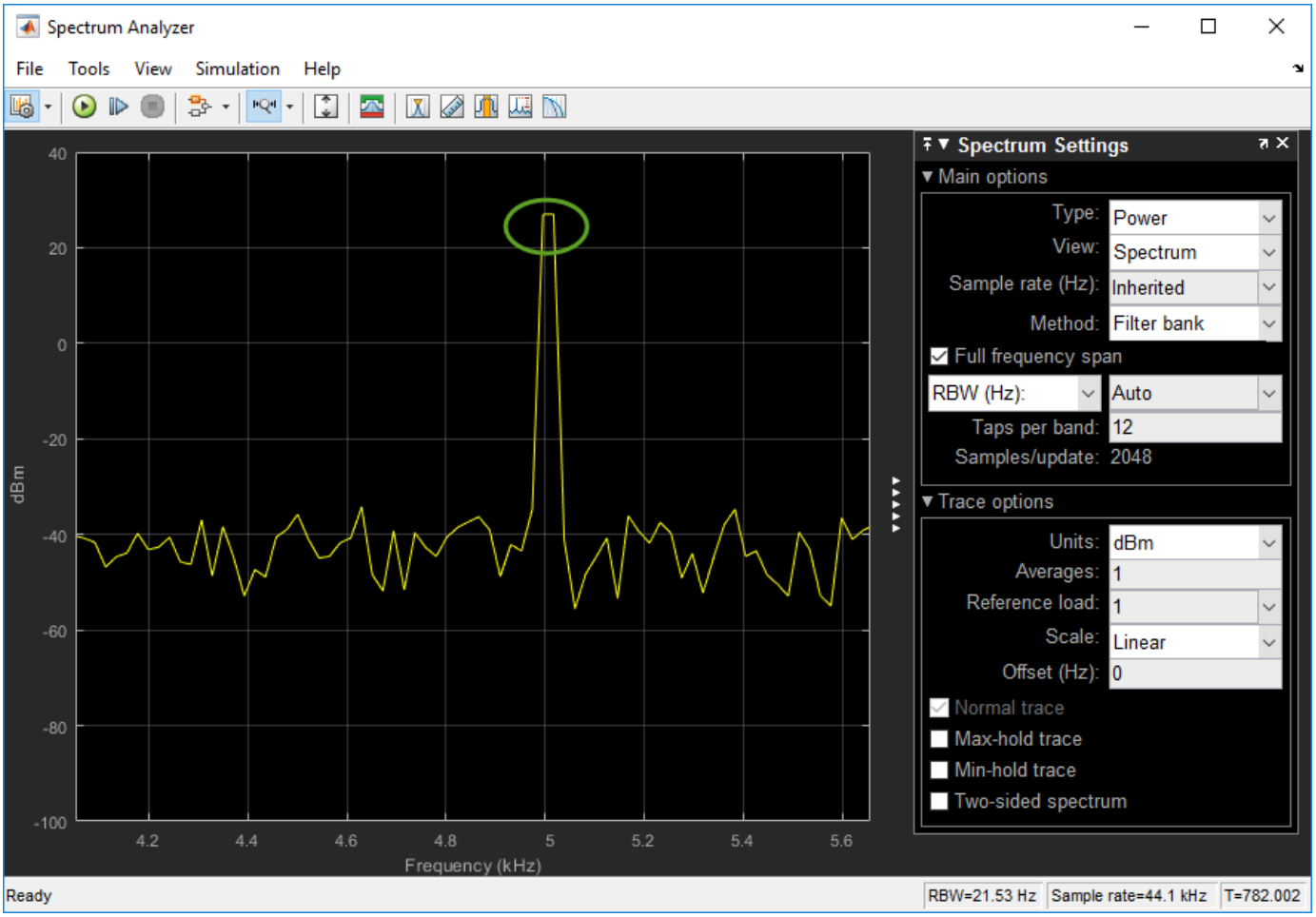
Using this value of RBW, the number of input samples used to compute one spectral update is given by $N_{samples}$ = Fs/RBW, which is 44100/21.53 or 2048.

RBW calculated in this mode gives a good frequency resolution.

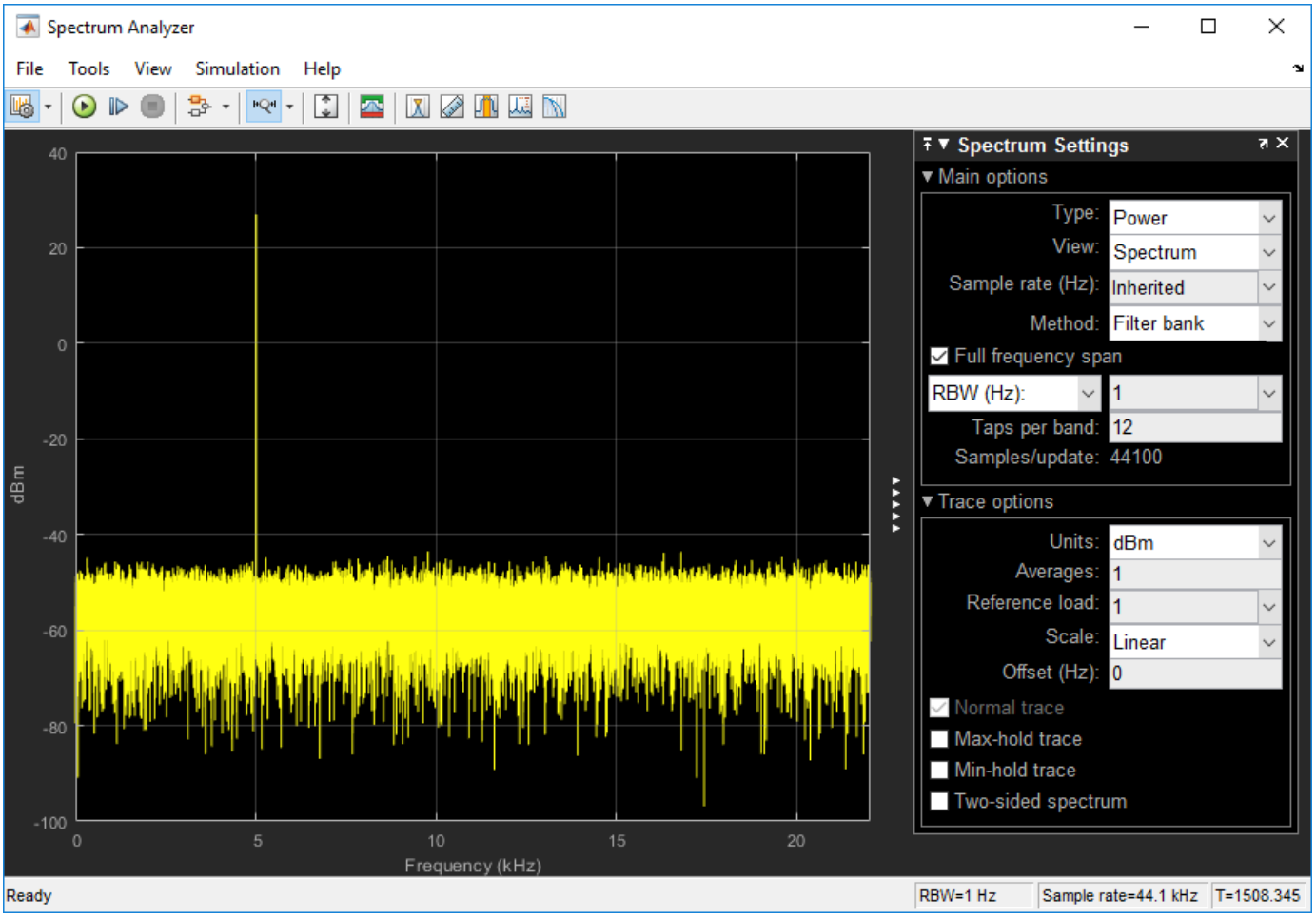
To distinguish between two frequencies in the display, the distance between the two frequencies must be at least RBW. In this example, the distance between the two peaks is 5000 Hz, which is greater than RBW. Hence, you can see the peaks distinctly. Change the frequency of the second sine wave from 10000 Hz to 5015 Hz. The difference between the two frequencies is less than *RBW*.



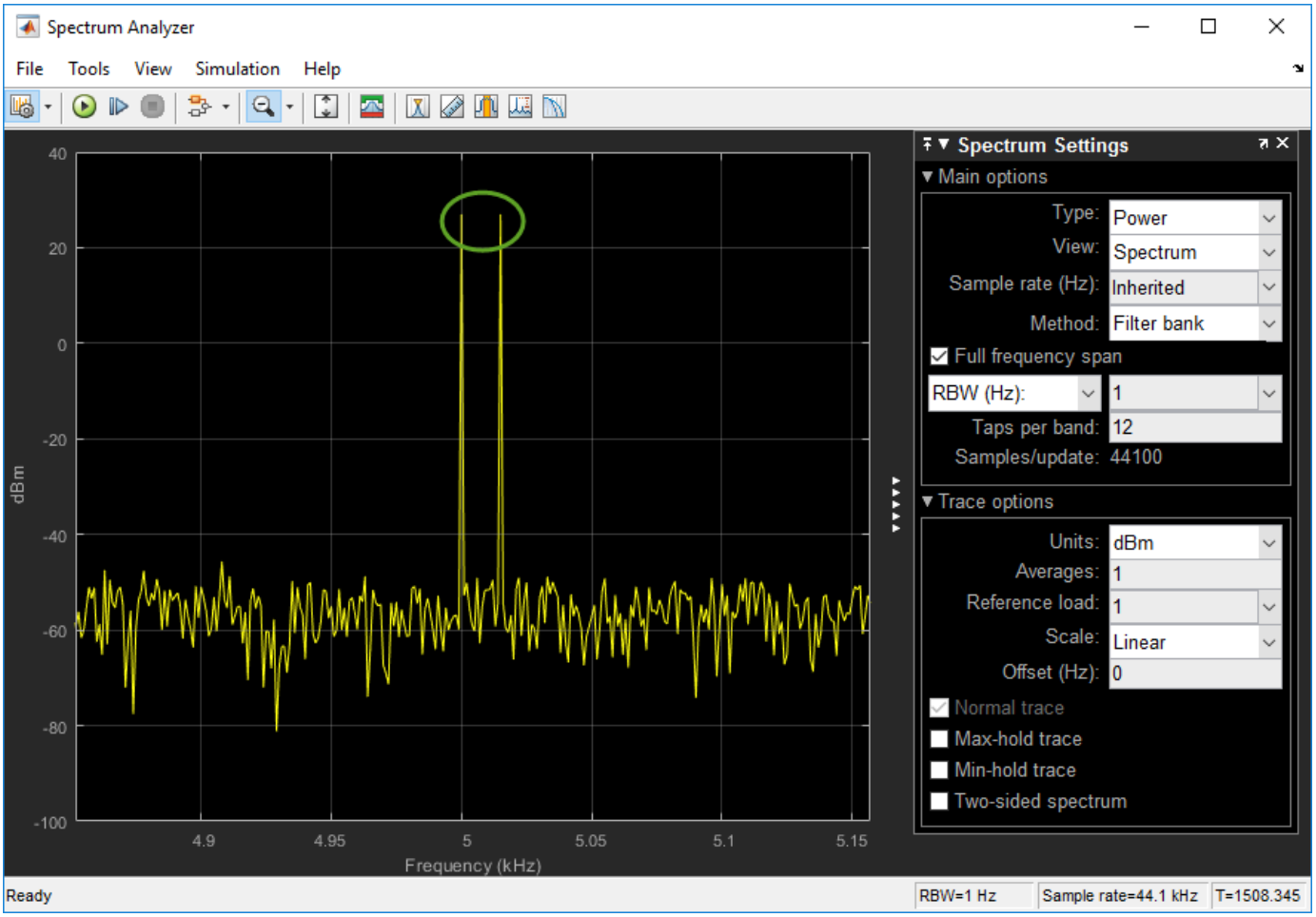
On zooming, you can see that the peaks are not distinguishable.



To increase the frequency resolution, decrease *RBW* to 1 Hz and run the simulation.



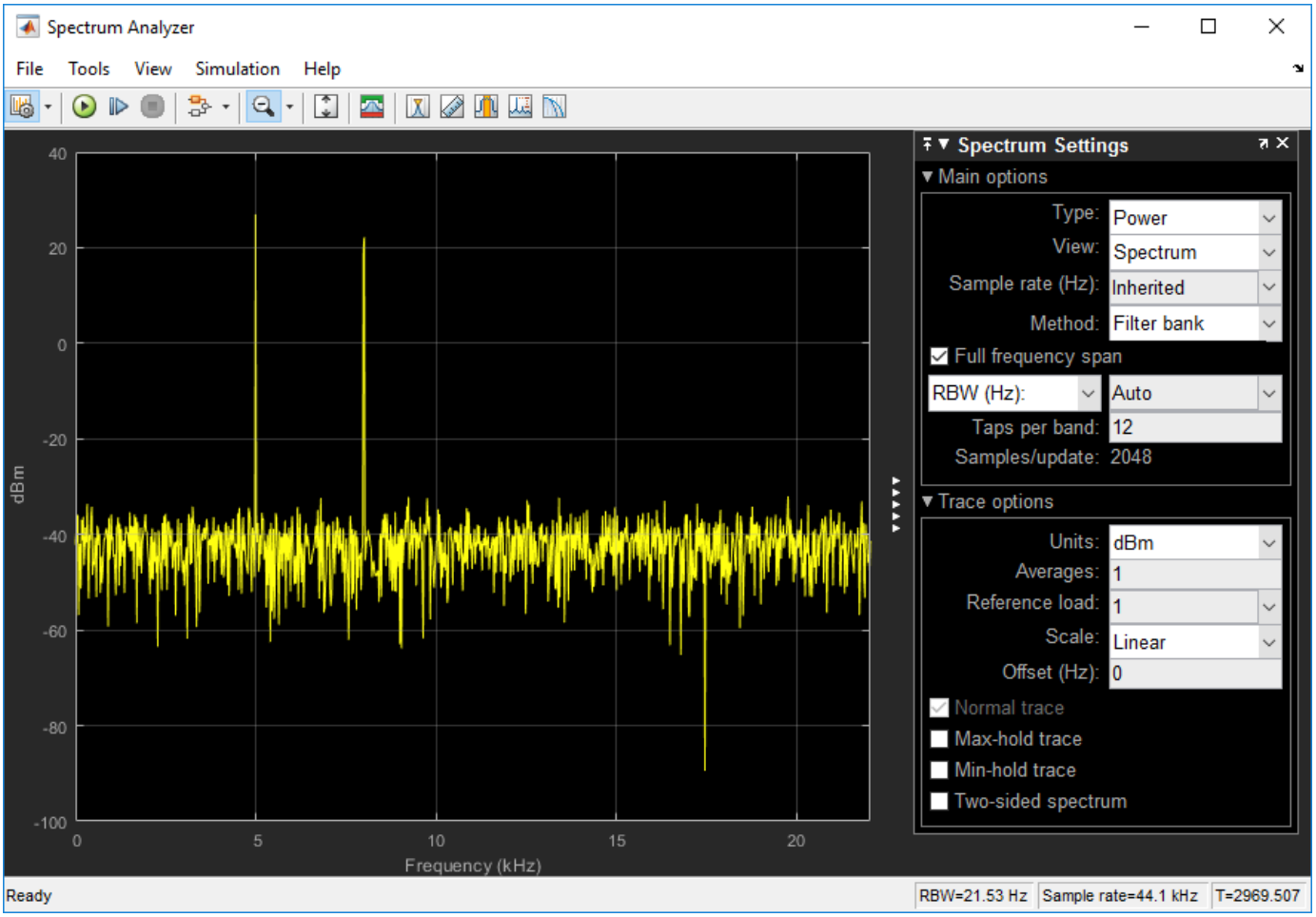
On zooming, the two peaks, which are 15 Hz apart, are now distinguishable



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

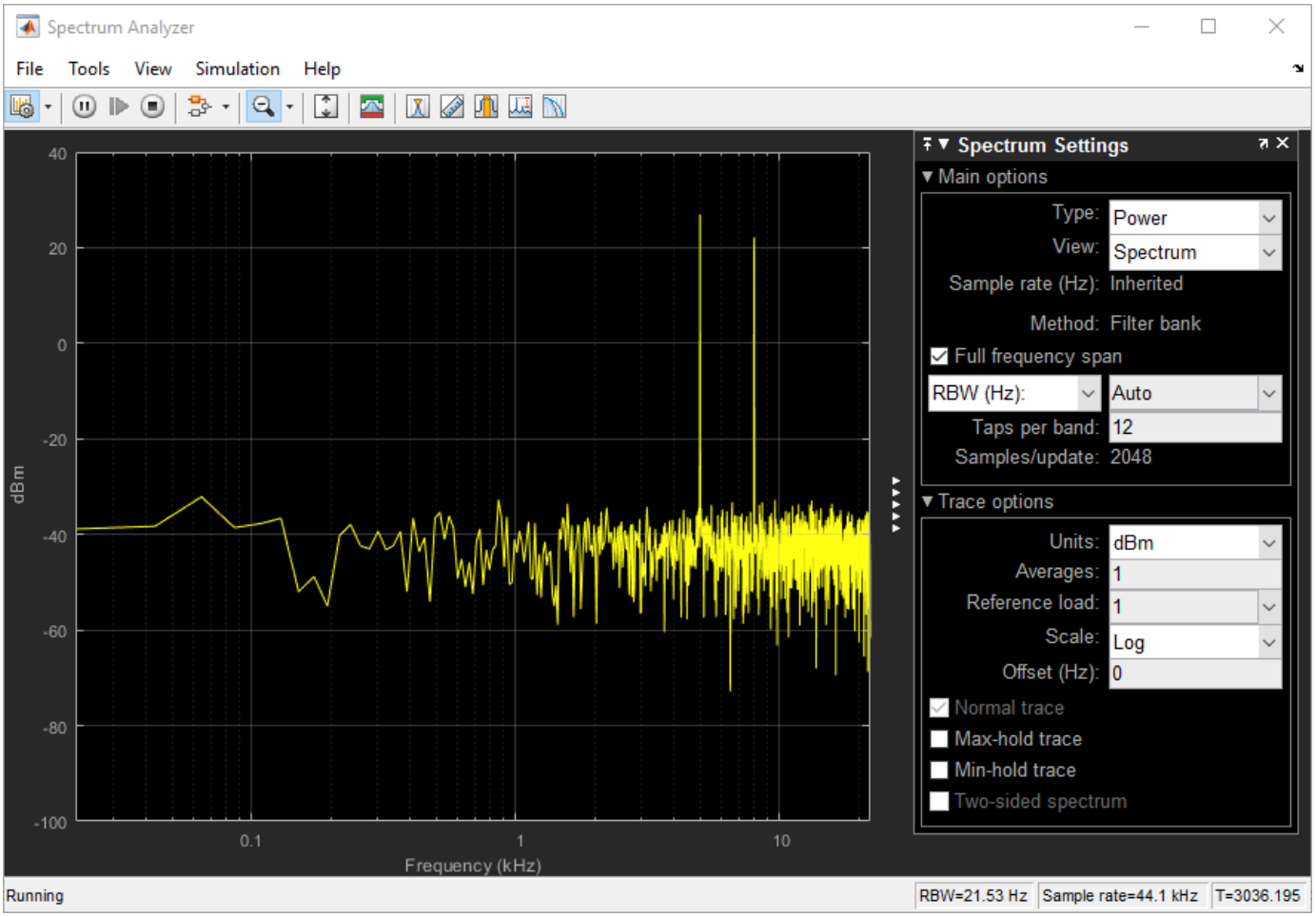
When you change the dynamics of the input signal during simulation, the power spectrum of the signal also changes in real time. While the simulation is running, change the **Frequency** of the Sine Wave 1 block to `8000` and click **Apply**. The second tone in the spectral analyzer output shifts to 8000 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

$$V_{rms} = 10^{26.9897/20}\sqrt{0.001}$$

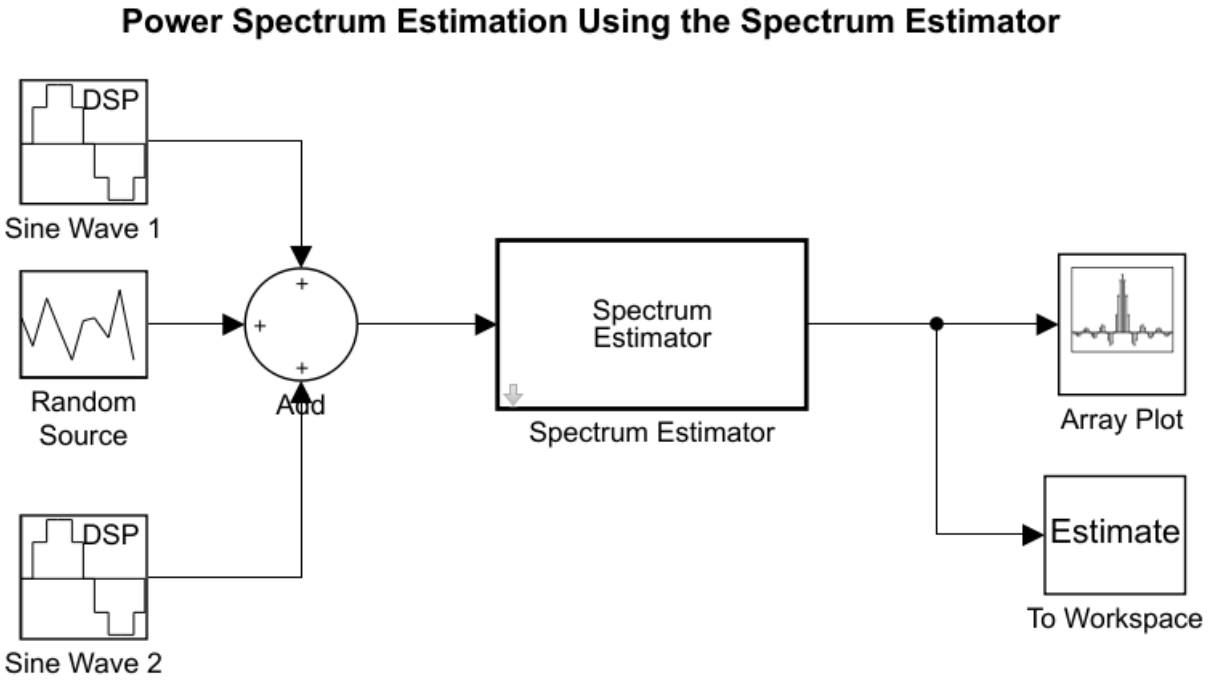
which equals 0.7071.

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.

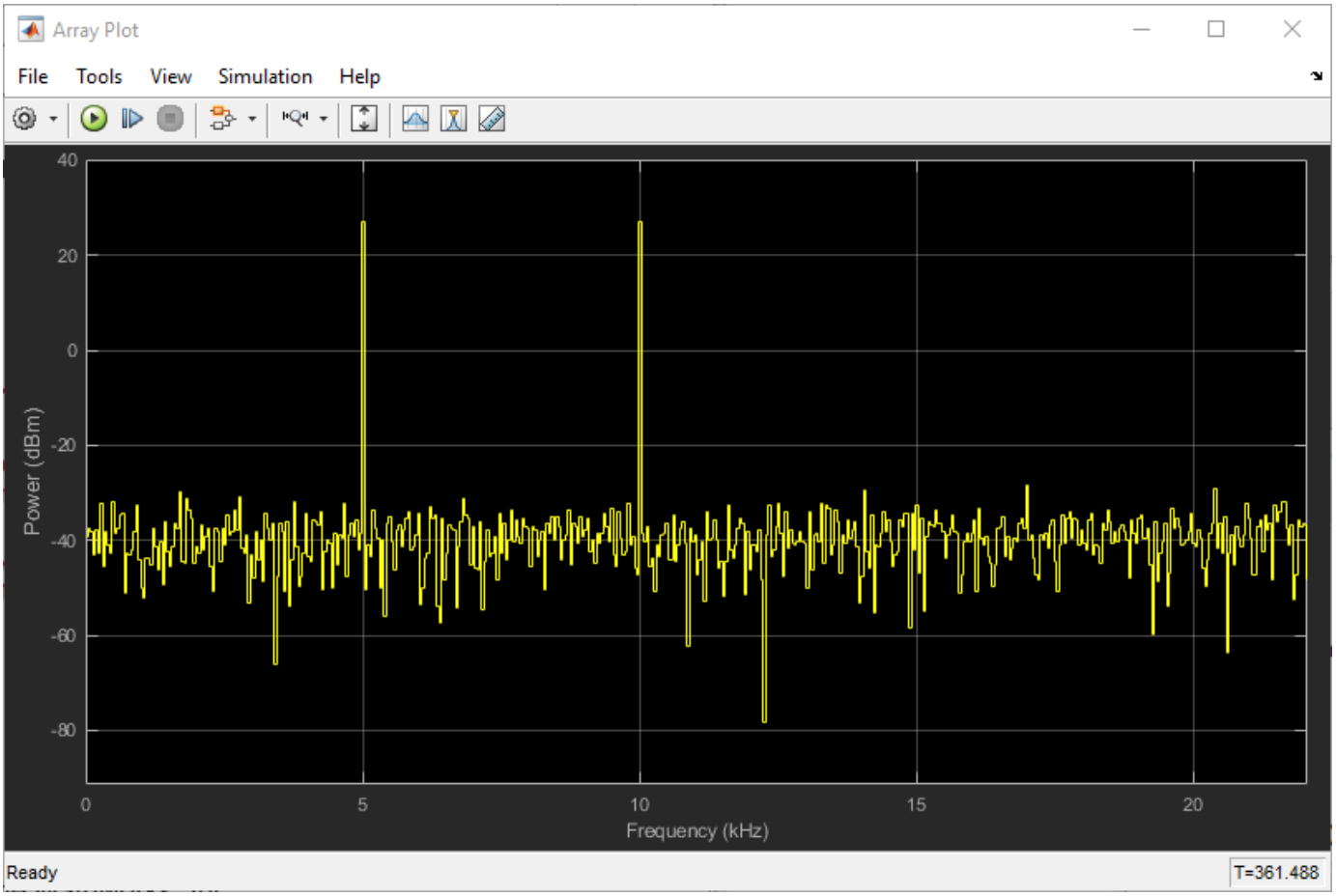


Replace the Spectrum Analyzer block in `ex_psd_sa` with the Spectrum Estimator block followed by an Array Plot block. To view the model, enter `ex_psd_estimatorblock` in the MATLAB command prompt. In addition, to access the spectral estimation data in MATLAB, connect the To Workspace block to the output 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 18-15
- “Estimate the Transfer Function of an Unknown System” on page 18-44
- “View the Spectrogram Using Spectrum Analyzer” on page 18-52
- “Spectral Analysis” on page 18-61

Estimate the Transfer Function of an Unknown System

In this section...
“Estimate the Transfer Function in MATLAB” on page 18-44
“Estimate the Transfer Function in Simulink” on page 18-48

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 18-61.

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);
```

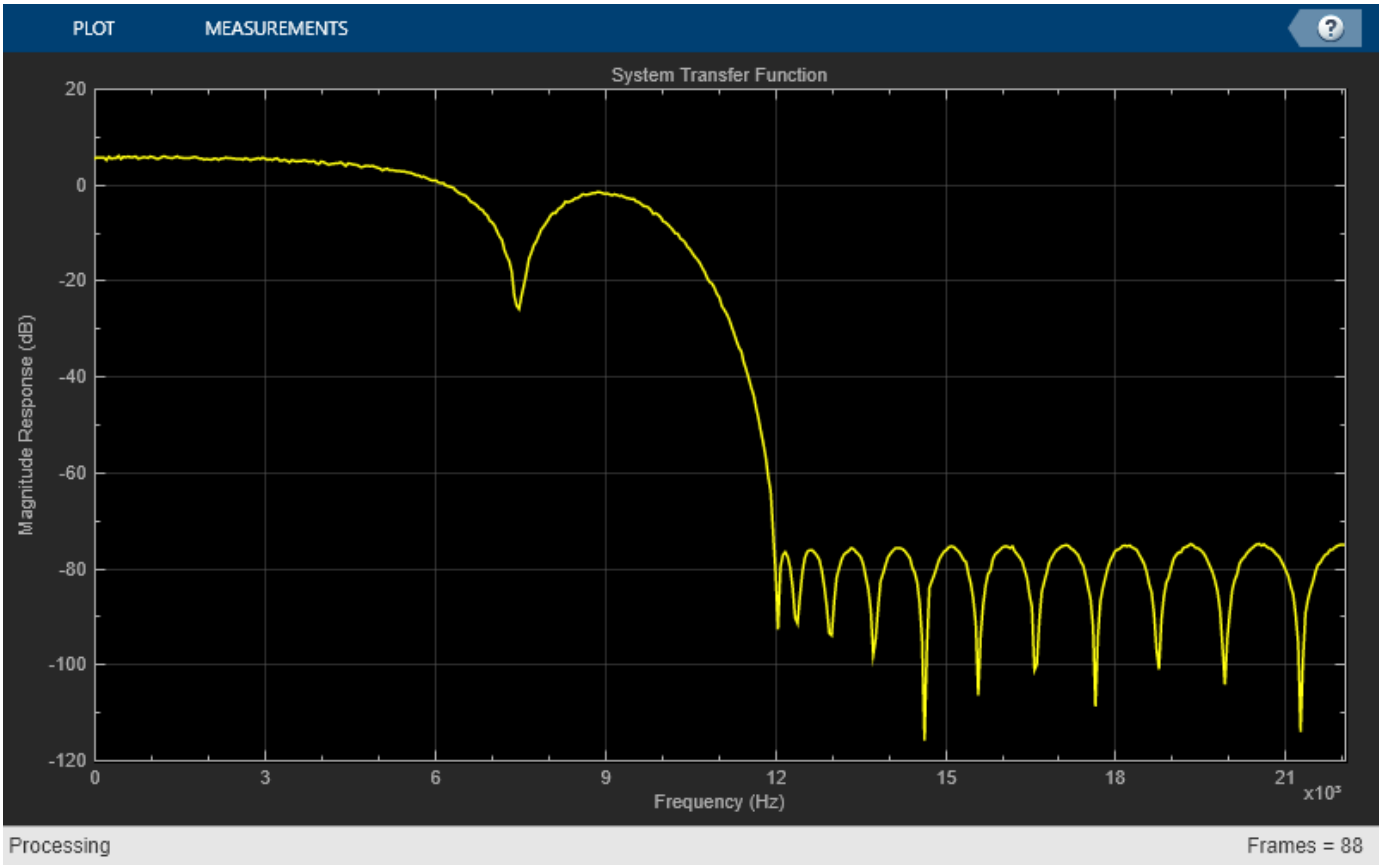
By default, the x-axis of the array plot is in samples. To convert this axis into frequency, set the 'SampleIncrement' property of the `dsp.ArrayPlot` object to Fs/1024. In this example, this value is 44100/1024, or 43.0664. For a two-sided spectrum, the `XOffset` property of the `dsp.ArrayPlot` object 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].

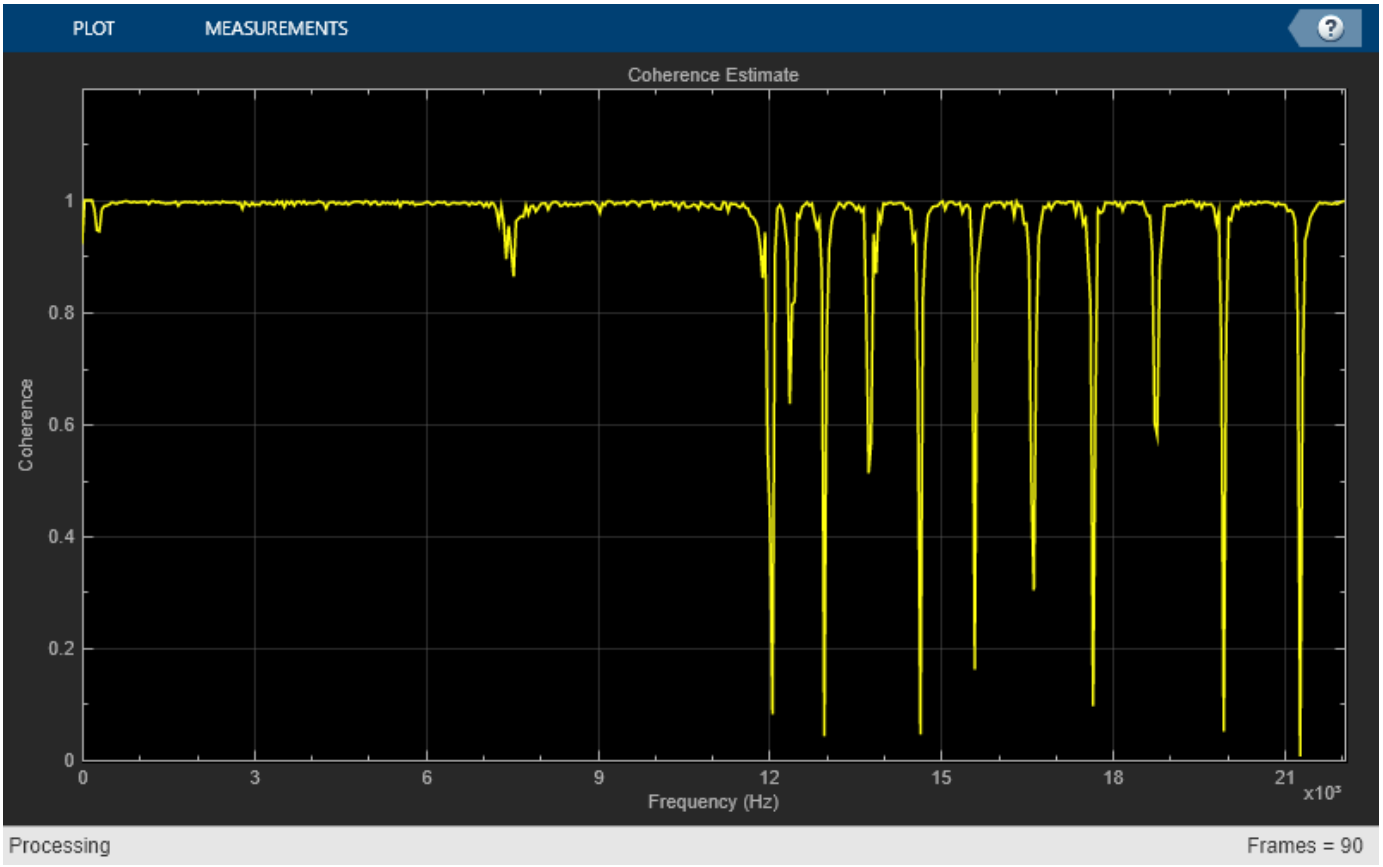
Estimate the Transfer Function

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.

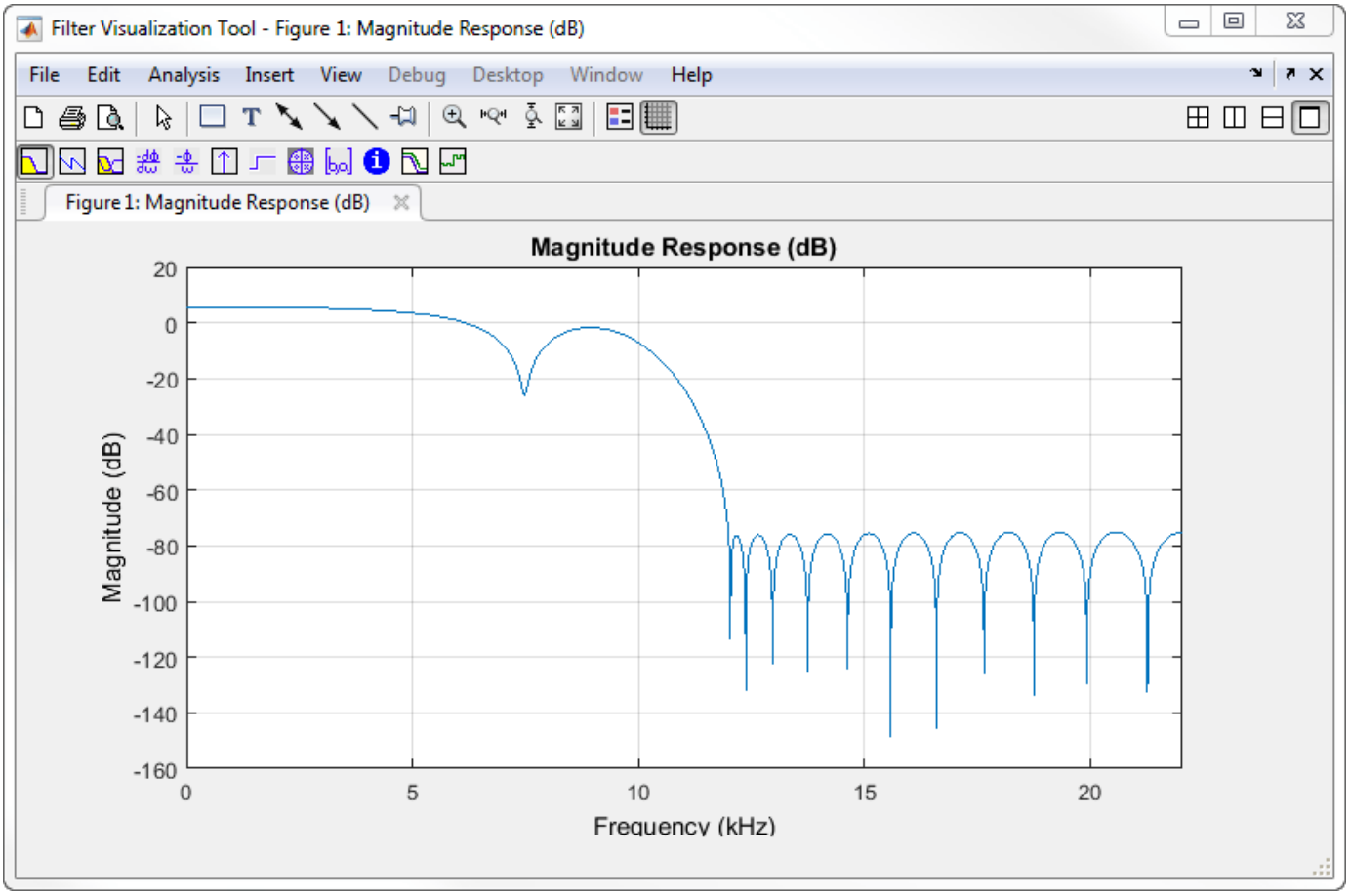
Magnitude Response of the Filter Using fvtool

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:

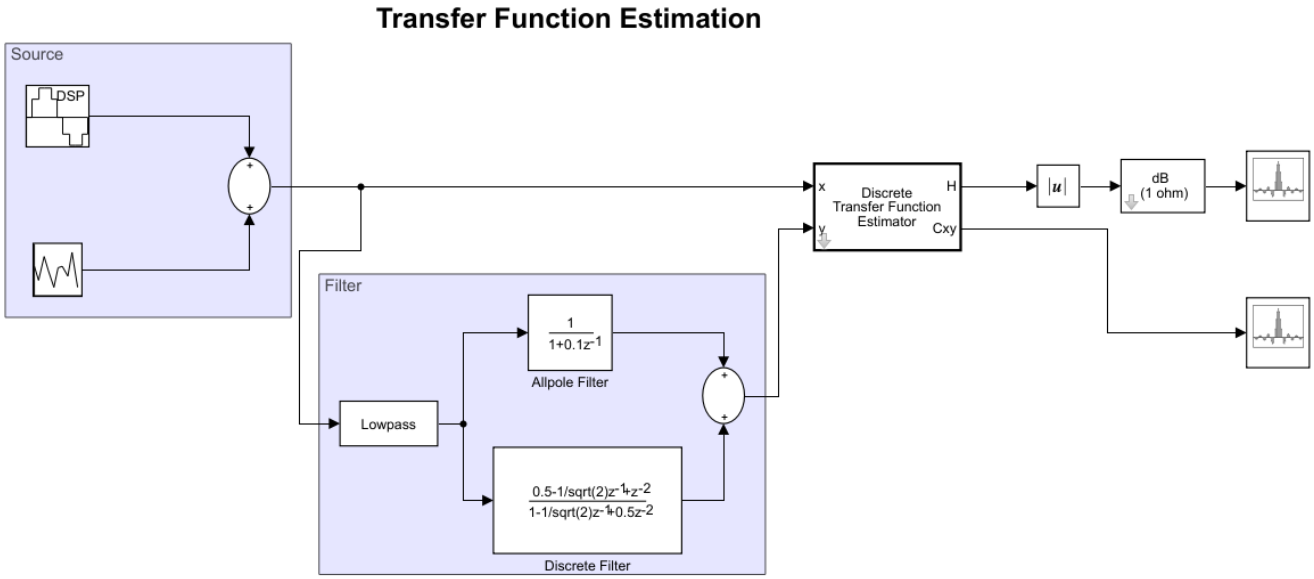


The magnitude response that fvtool shows matches the magnitude response that the `dsp.TransferFunctionEstimator` object estimates.

Estimate the Transfer Function in Simulink

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.



Copyright 2015 the Mathworks, Inc.

Open and Inspect the Model

To open the model, enter `ex_transfer_function_estimator` in the MATLAB command prompt.

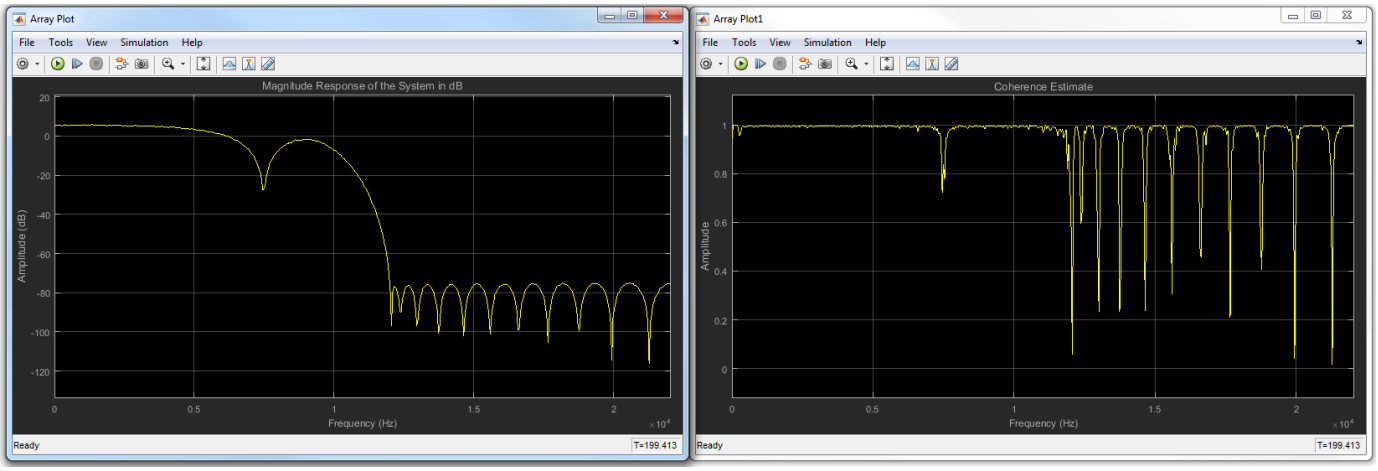
Here are the settings of the blocks in the model.

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 18-61
- “Estimate the Power Spectrum in MATLAB” on page 18-15
- “Estimate the Power Spectrum in Simulink” on page 18-28

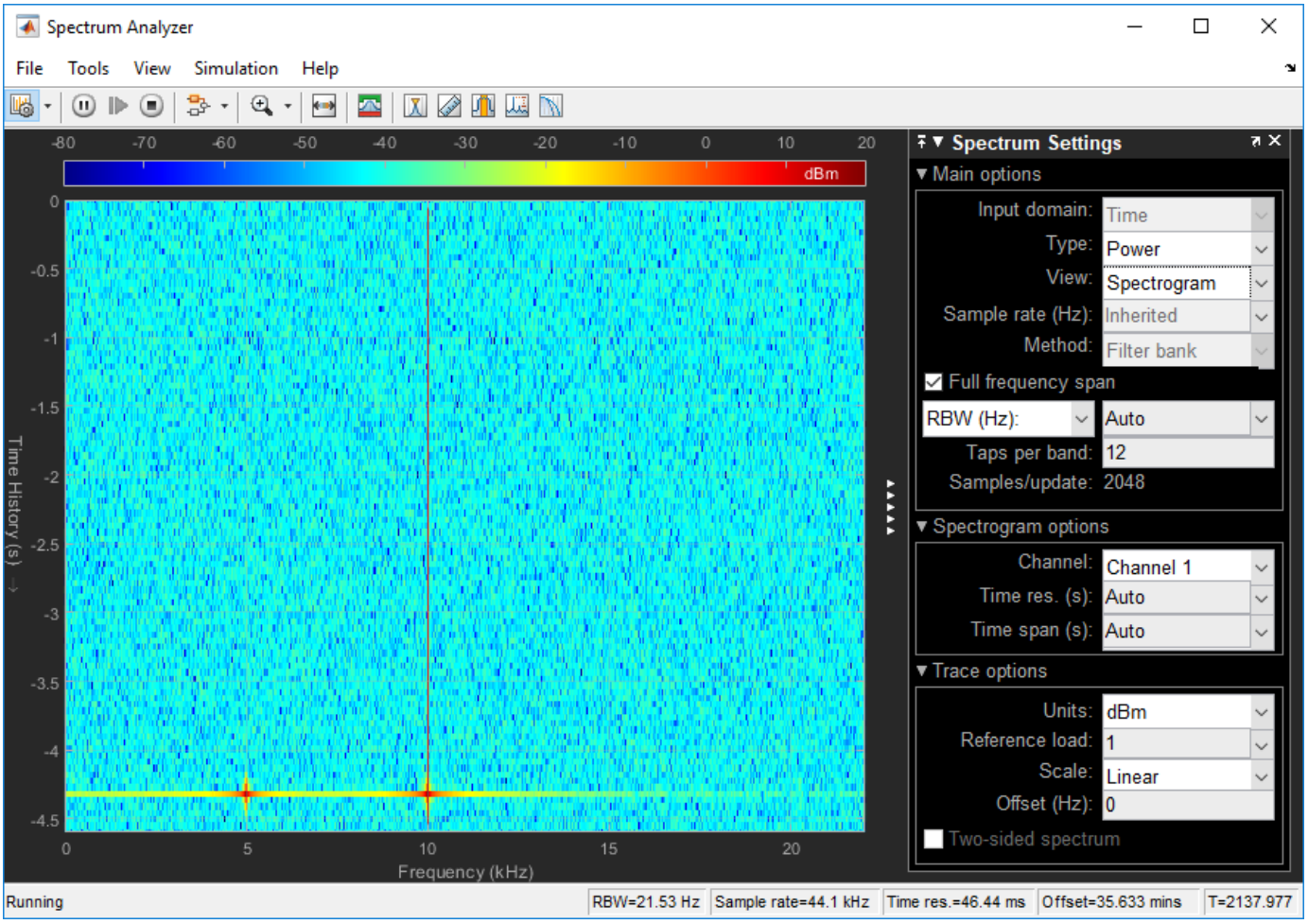
View the Spectrogram Using Spectrum Analyzer

In this section...
“Colormap” on page 18-53
“Display” on page 18-54
“Resolution Bandwidth (RBW)” on page 18-54
“Time Resolution” on page 18-57
“Convert the Power Between Units” on page 18-57
“Scale Color Limits” on page 18-59

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.

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 `SpectrumAnalyzerConfiguration` object and run the `getSpectrumData` function on this object.

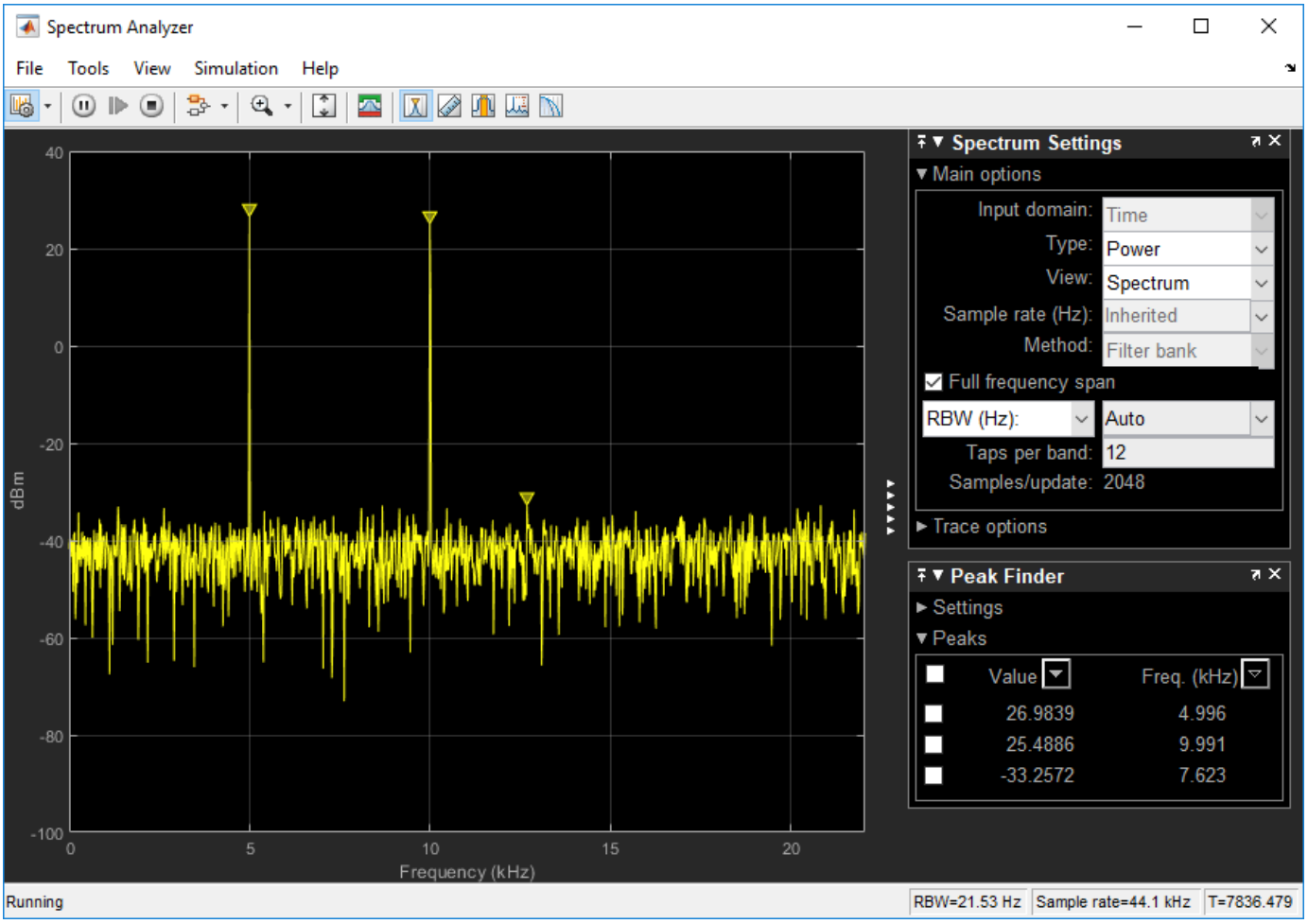


Colormap

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:

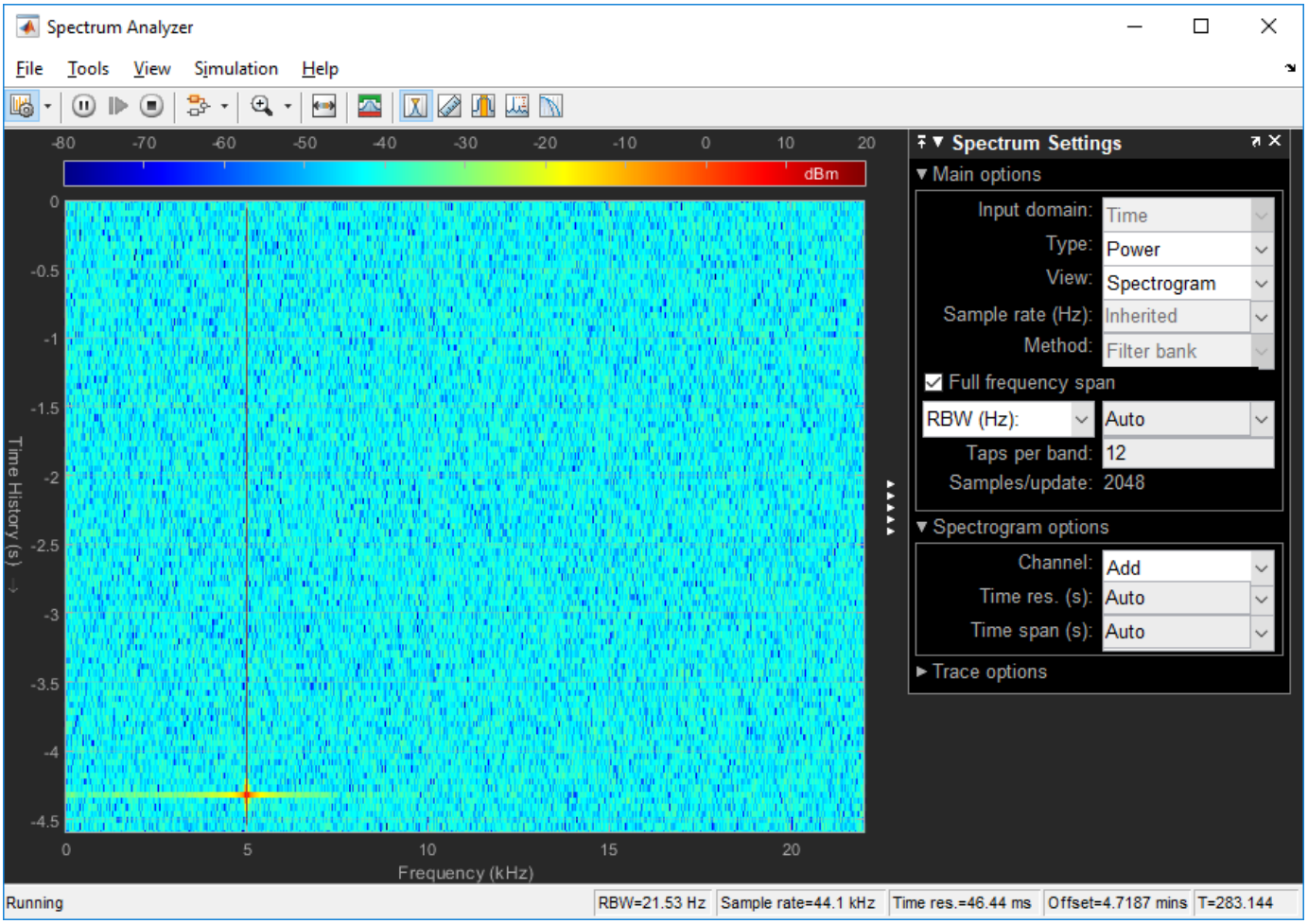
$$N_{samples} = \frac{\left(1 - \frac{O_p}{100}\right) \times NENBW \times F_s}{RBW}$$

O_p is the amount of overlap between the previous and current buffered data segments. *NENBW* is the equivalent noise bandwidth of the window.

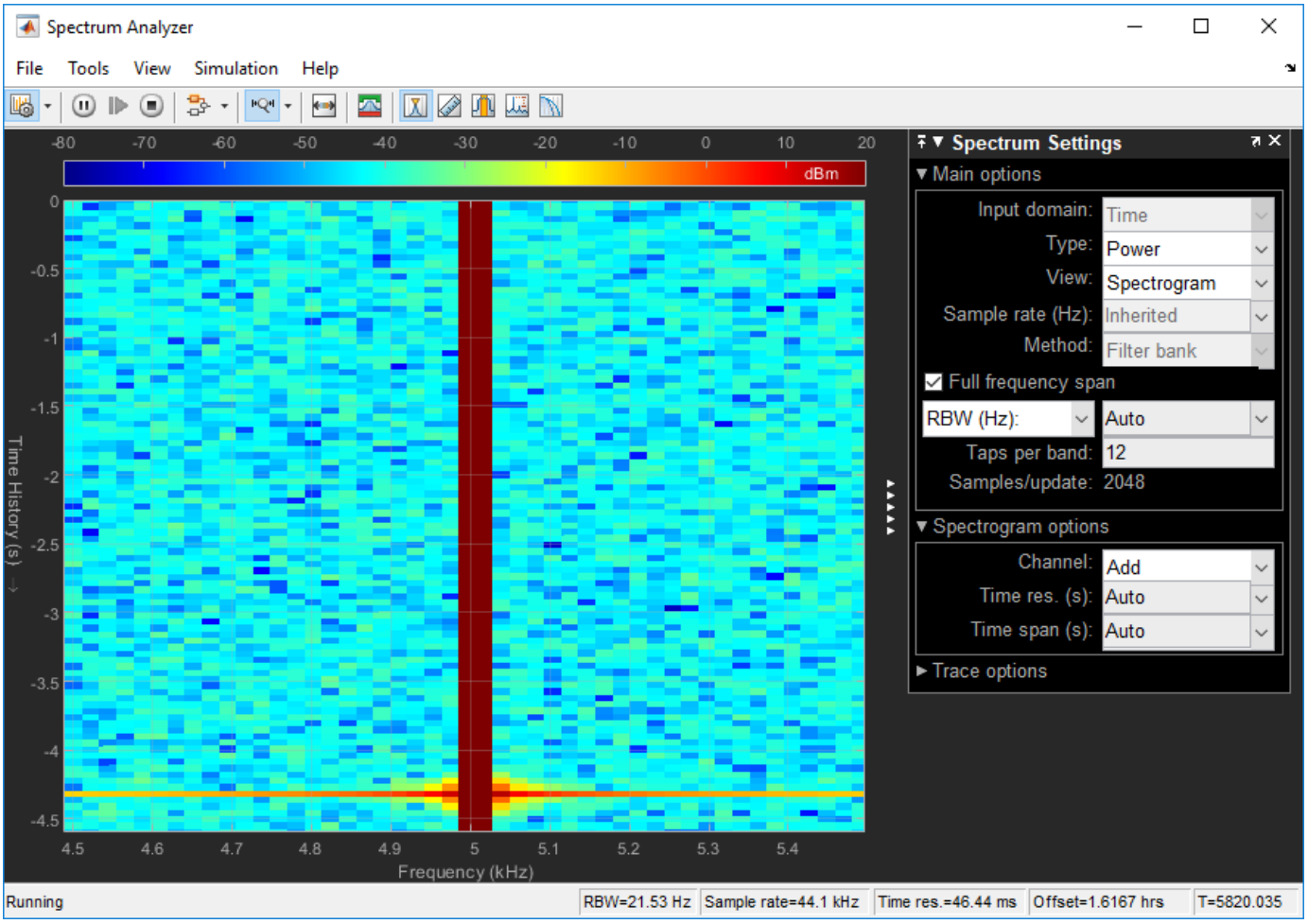
For more information on the details of the spectral estimation algorithm, see “Spectral Analysis” on page 18-61.

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.

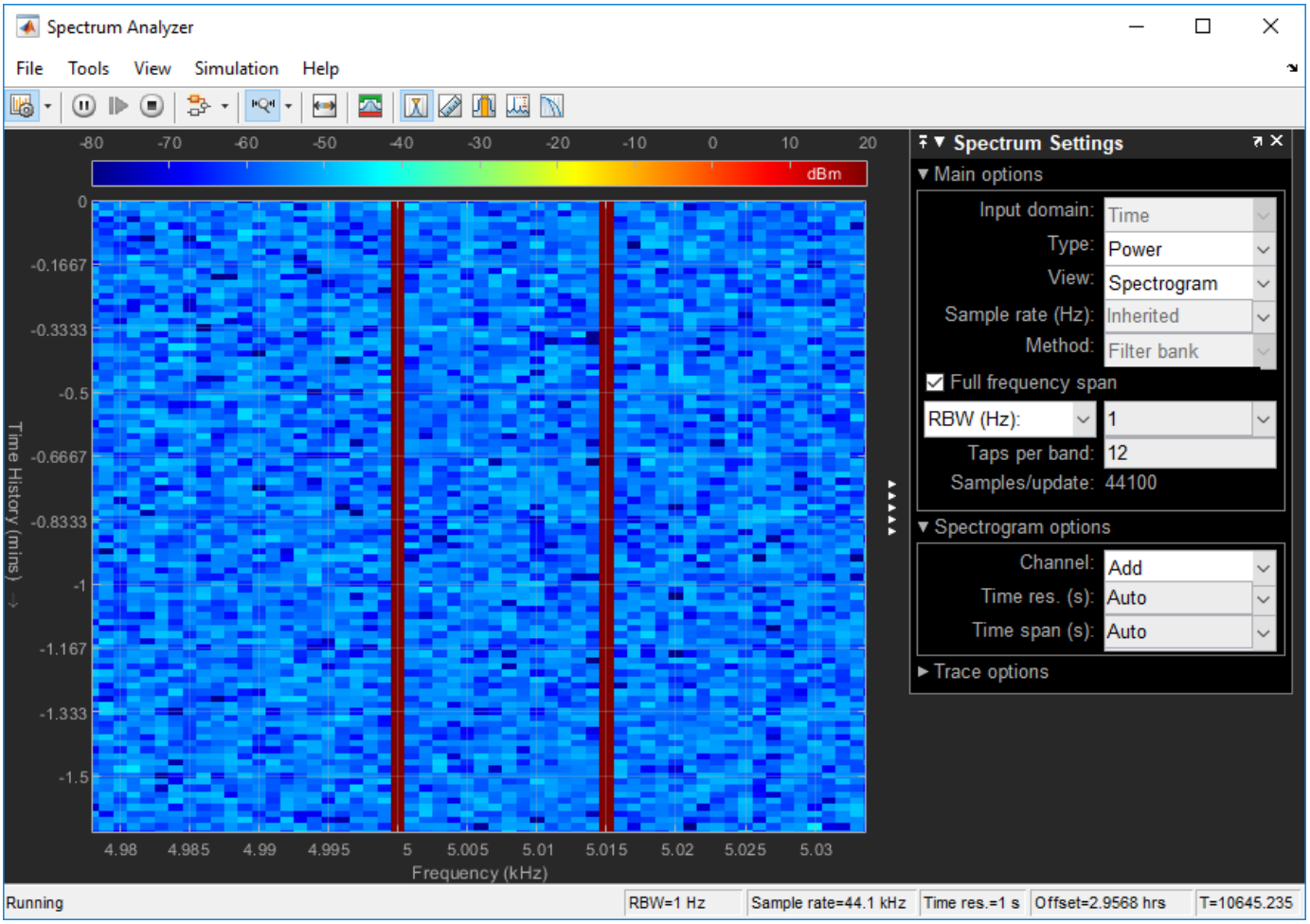
Change the frequency of the second sine wave from 10000 Hz to 5015 Hz. The difference between the two frequencies is 15 Hz, which is less than *RBW*.



On zooming, you can see that the peaks at 5000 Hz and 5015 Hz 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 18-15
- “Estimate the Power Spectrum in Simulink” on page 18-28

Spectral Analysis

In this section...
“Welch’s Algorithm of Averaging Modified Periodograms” on page 18-61
“Filter Bank” on page 18-64

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.

$$P_{xx}^{i}(f) = \frac{1}{MU}\left|\sum_{n=0}^{M-1} x_i(n)w(n)e^{-j2\pi fn}\right|^2, \qquad i = 0, 1, \ldots, L-1$$

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 18-15
- “Estimate the Power Spectrum in Simulink” on page 18-28
- “Estimate the Transfer Function of an Unknown System” on page 18-44
- “View the Spectrogram Using Spectrum Analyzer” on page 18-52

Streaming Power Spectrum Estimation Using Welch's Method

Compute the power spectrum estimate of a time-domain input signal using the Spectrum Estimator block. The block uses one of the following methods to compute the power spectrum estimate:

- Welch's method of averaged modified periodograms
- Filter bank method

This example uses the Welch's method of averaged modified periodograms. For an example that uses the filter bank-based spectrum estimation method, see “High Resolution Spectral Analysis in MATLAB” on page 4-16. The same example also shows the comparison between the filter bank estimator and the Welch-based spectral estimator. Generally, filter bank-based spectrum estimation yields better resolution with less spectral leakage, more accurate peaks and a more accurate noise floor.

For more details on algorithm of these two methods, see the 'Algorithms' section in the Spectrum Estimator block.

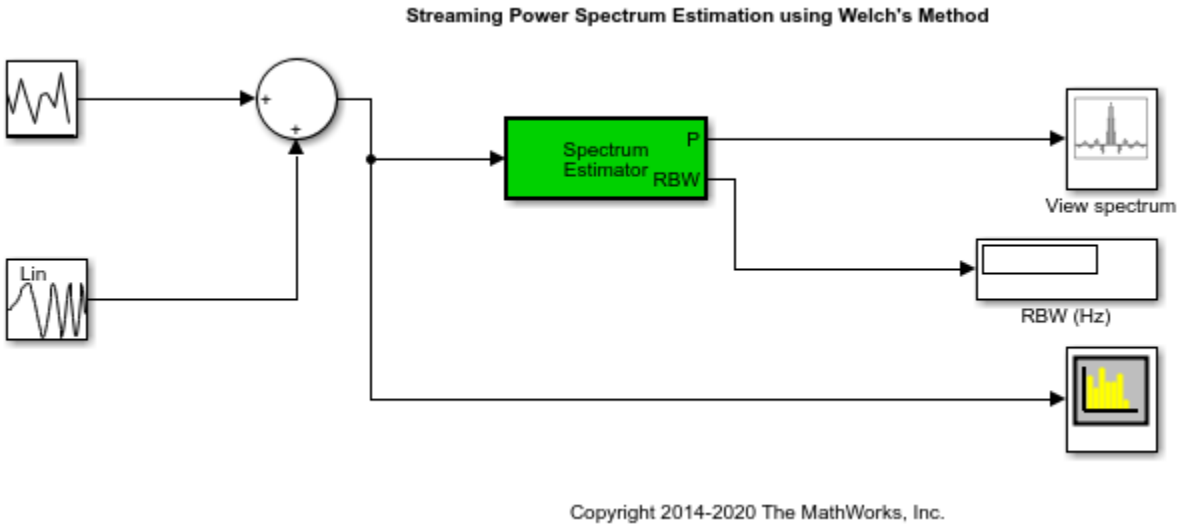
The Spectrum Estimator block is useful if you need direct access to the estimated spectrum (rather than just visualize it). The output power spectrum may be used as an input to other blocks in your model, or may be logged to the workspace for post-processing. To visualize the spectra, use the Spectrum Analyzer scope block.

Welch's Method of Averaged Modified Periodograms

In the Welch method, the input time-domain data is partitioned into data segments based on the selected window length and overlap percentage. This stage is implemented using a Buffer block. A window is applied to each segment, and then an averaged periodogram is computed based on the windowed sequences. This stage is implemented using a `dsp.SpectrumEstimator` System object™. The length of the data segments and the choice of the window determine the estimate's resolution bandwidth (RBW), which is the smallest positive frequency that can be resolved in the power spectral estimate.

Specifying Window Length

The `dspstreamingwelch` model shown below uses a Welch Spectrum Estimator block to estimate the spectrum of a noisy chirp signal sampled at 44100 Hz. The power spectrum estimate is displayed using an Array Plot scope. The peak value of the spectrum, as well as the frequency at which the peak occurs, are detected and displayed on the scope. The estimate's RBW is also displayed. Moreover, a Spectrum Analyzer scope block is included for comparison and validation purposes.



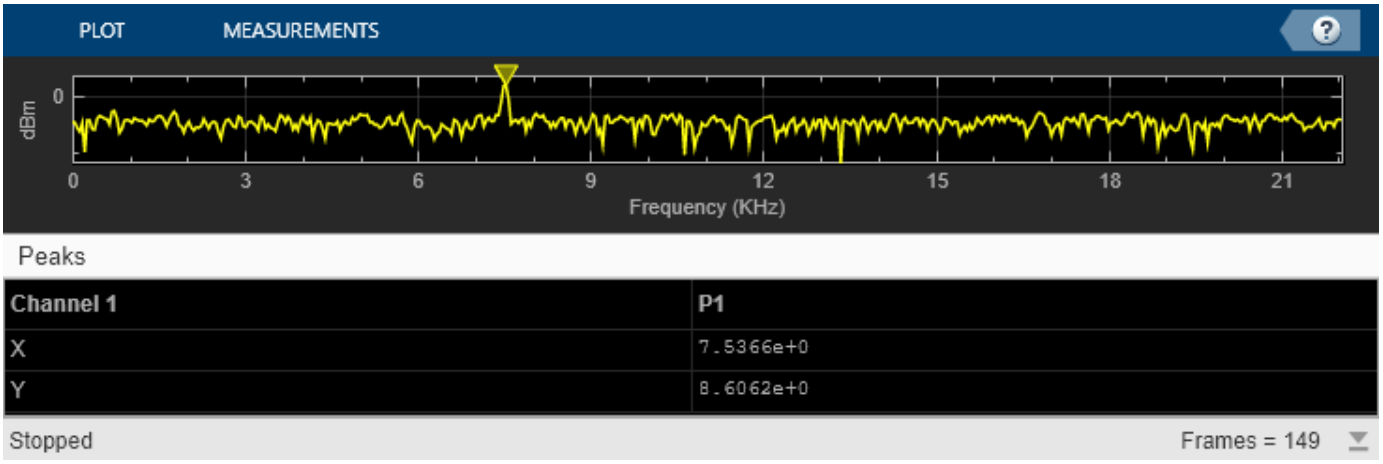
The block's frequency resolution method is set to `Window length`. The window length is set to 1024. The FFT length, *NFFT* , is equal to the window length. The data is windowed using a Chebyshev window with a sidelobe attenuation of 60 dB. The frequency range is one-sided. In this case, the length of the spectrum estimate is $NFFT/2+1=513$ and is computed over the interval [0 Hz,22050 Hz]. The **Sample increment** property of the Array Plot scope is accordingly set to $Fs/NFFT=44100/(1024*1000)$, where the increment is divided by 1000 to scale the frequency units to kHz. You can access the scope's **Sample increment** property by opening its Configuration properties window.

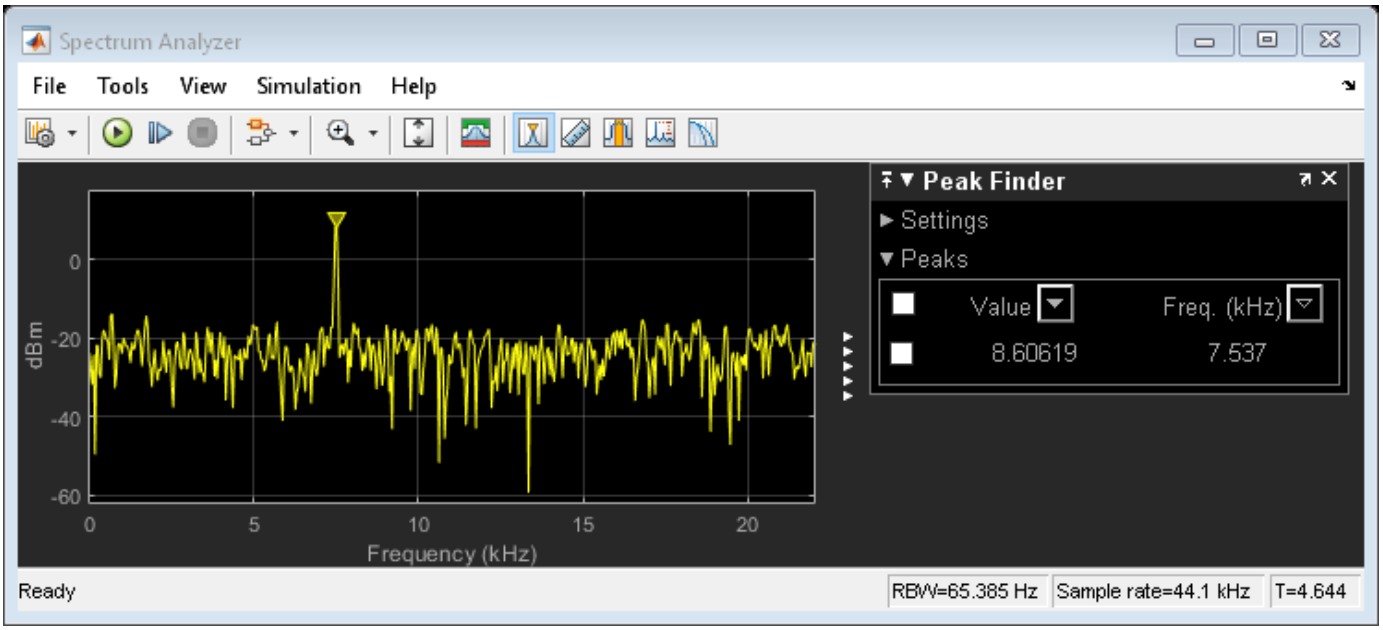
The resolution bandwidth is given by:

$$RBW=enbw(chebwin(N,SL))*Fs/N$$

where *N* is the window length, *enbw* is a function that computes the window's equivalent noise bandwidth, *SL* is the sidelobe attenuation of the selected Chebyshev window and *Fs* is the sample rate. In this case, RBW is equal to 65.38 Hz.

When you simulate the model, you can verify that the displayed RBW value is equal to the one shown on the lower bar of the Spectrum Analyzer scope. Moreover, the two blocks give the same peak measurements.



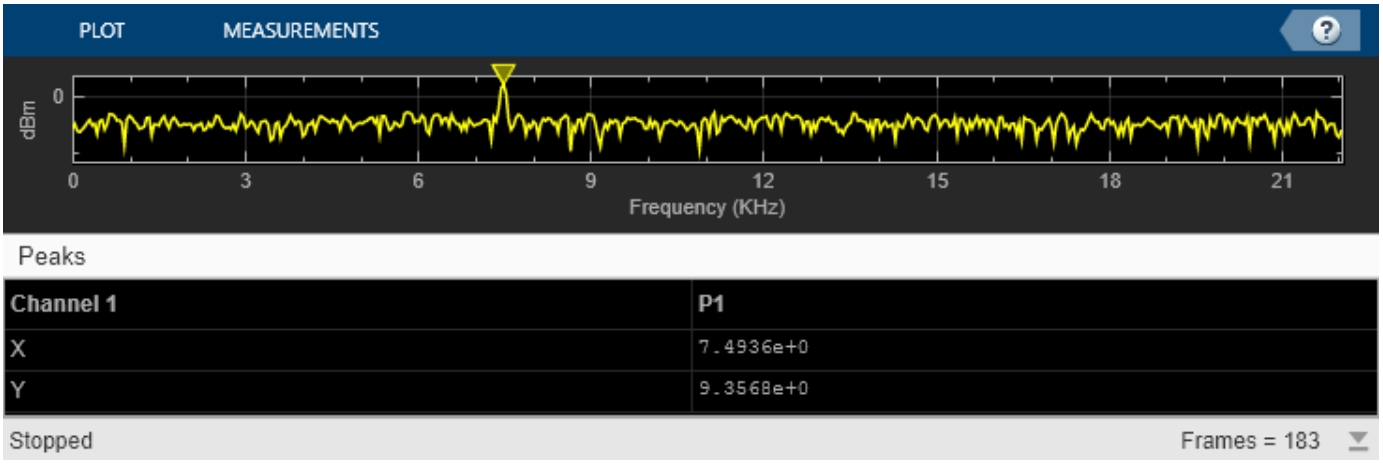


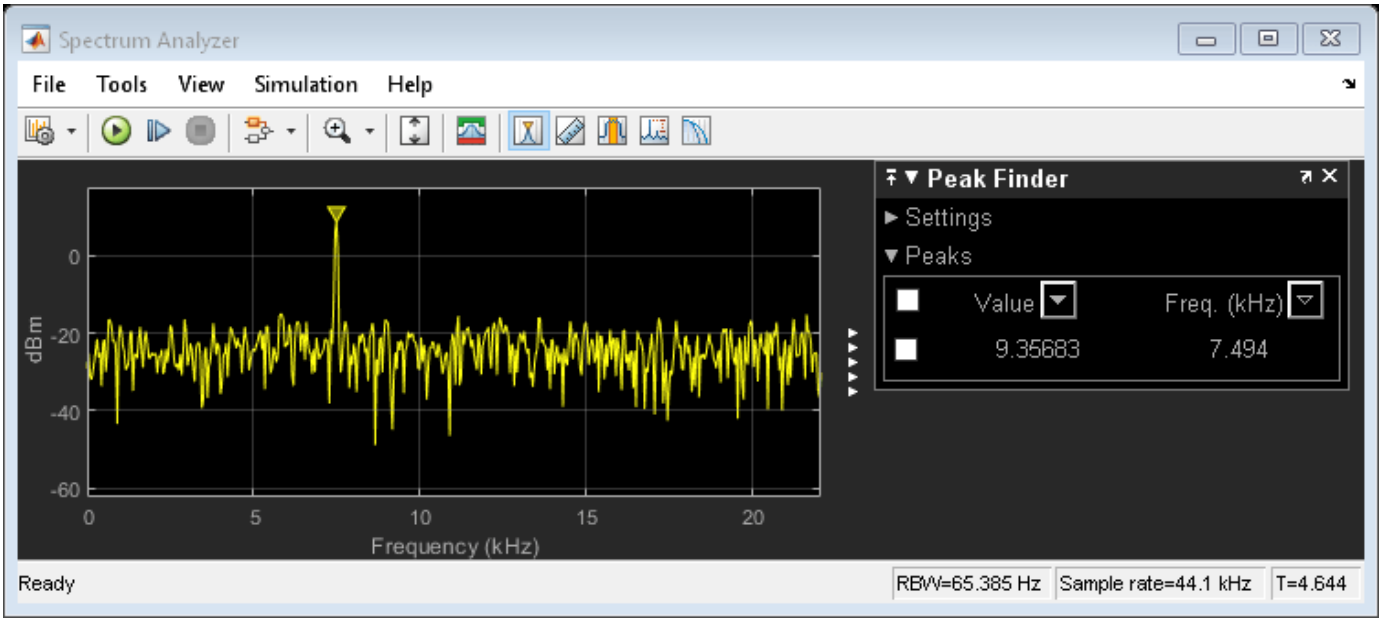
Specifying Non-Zero Overlap

The model in the previous section had zero-overlap. In the `dspstreamingwelch_overlap` model, we use a Welch Estimation block with an overlap of 50%. Since other model parameters are identical to the previous section, the RBW is unchanged and is equal to 65.38 Hz. With a window length of 1024 and an overlap percentage of 50%, 512 input samples are required to form a new data segment. Since the input data is of length 1024, each new data frame yields two new periodograms, and the block's output port runs at a rate twice as fast as the input port.

Note that the Welch estimate block does not have zero latency in this case. The first spectrum estimate output is based on the buffer's initial condition, which is equal to eps. In order to match the spectrum and measurements of the Spectrum Analyzer scope, we therefore insert a delay block at the input of the Spectrum Analyzer.

The results of the Spectrum Analyzer and Welch estimate block may be validated by simulating the model.





Specifying RBW

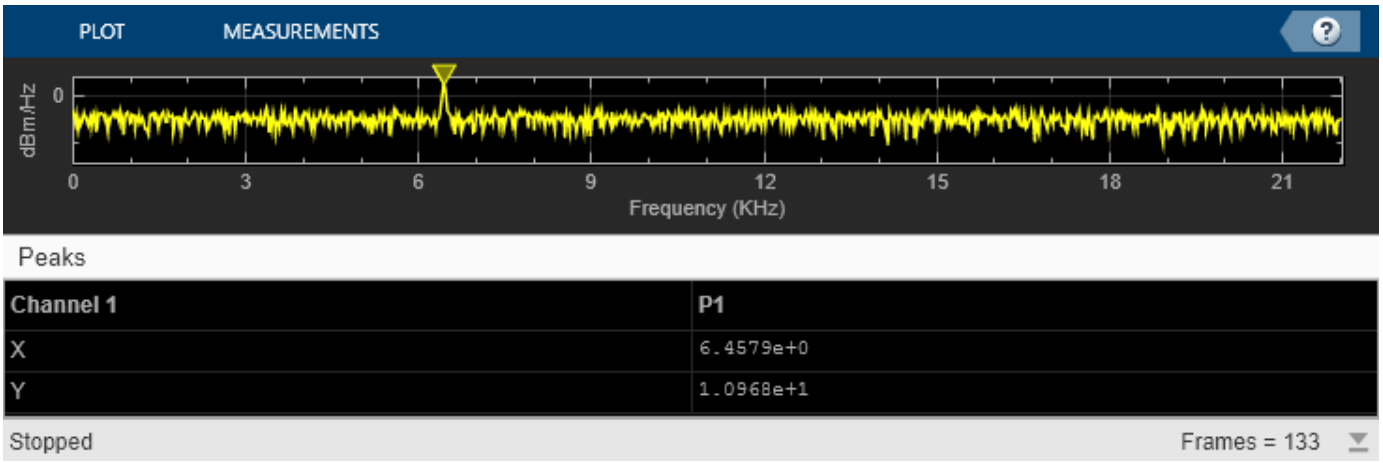
In the `dspstreamingwelch_rbw` model, the **Frequency Resolution Method** parameter is set to `RBW`. RBW source is Auto. In this mode, similar to the Spectrum Analyzer scope block, the resolution bandwidth is chosen such that there are 1024 RBW intervals over the specified Frequency Span. Since the span in this case is 22050 HZ, the RBW is 21.53 Hz.

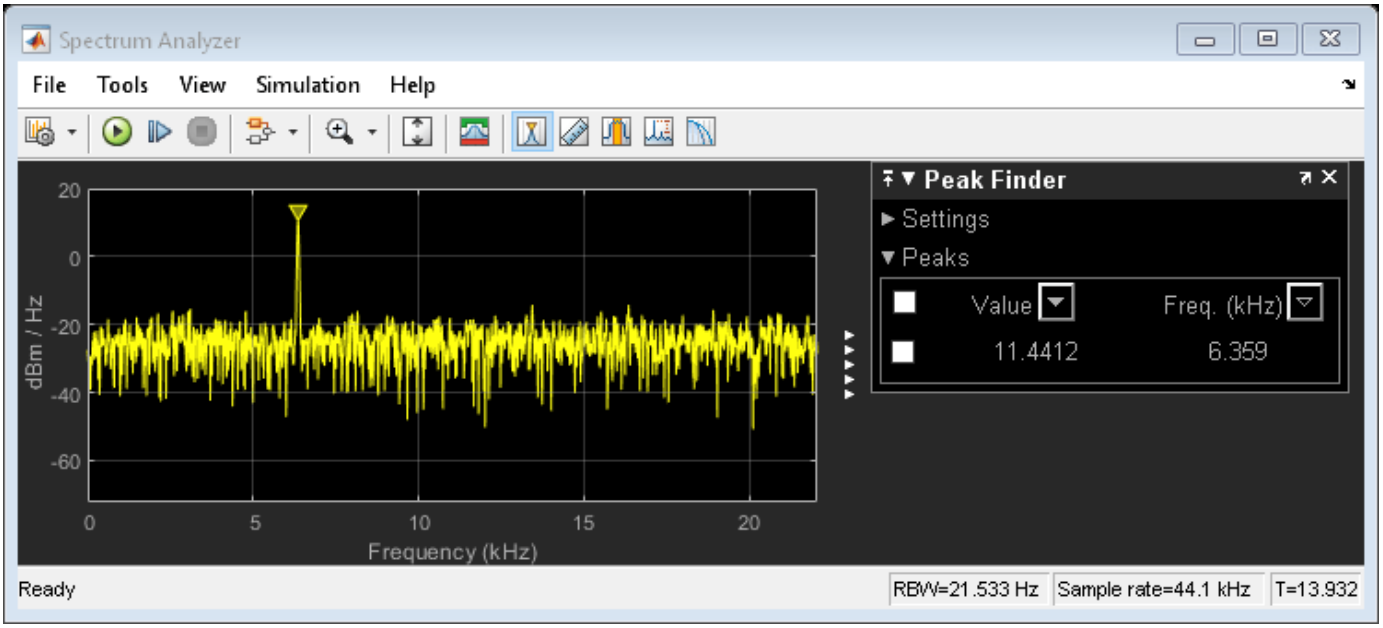
The window length used to buffer the data is iteratively computed to yield the desired RBW. The window length in this case is equal to 3073. To verify this value, we can compute the RBW that results from this window length:

$$RBW = enbw(hann(3073)) * 44100/3073 = 21.53Hz$$

Note that a Hann window is used in this model. In this case, the FFT length, NFFT, is odd and equal to 3073 (the window length). Since the frequency range is one-sided, the spectrum estimate is of length (NFFT + 1)/2 and is computed over the interval [0,44100/2). The **Sample increment** property of the Array Plot scope is set to $Fs/NFFT = 44100/(3073 * 1000)$ KHz.

Again, the results of the Spectrum Analyzer and Welch estimate block can be validated by simulating the model.





References

[1] Hayes, Monson H. *Statistical Digital Signal Processing and Modeling* Hoboken, NJ: John Wiley & Sons, 1996.

See Also

Blocks

Spectrum Estimator | Spectrum Analyzer | Array Plot | Random Source | Chirp

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 19-2
- “Fixed-Point Concepts and Terminology” on page 19-4
- “Arithmetic Operations” on page 19-8
- “System Objects in DSP System Toolbox that Support Fixed-Point Design” on page 19-15
- “Simulink Blocks in DSP System Toolbox that Support Fixed-Point Design” on page 19-19
- “System Objects Supported by Fixed-Point Converter App” on page 19-20
- “Convert dsp.FIRFilter Object to Fixed-Point Using the Fixed-Point Converter App” on page 19-21
- “Specify Fixed-Point Attributes for Blocks” on page 19-28
- “Quantizers” on page 19-43
- “Create an FIR Filter Using Integer Coefficients” on page 19-50
- “Fixed-Point Precision Rules for Avoiding Overflow in FIR Filters” on page 19-62

Fixed-Point Signal Processing

In this section...
“Fixed-Point Features” on page 19-2
“Benefits of Fixed-Point Hardware” on page 19-2
“Benefits of Fixed-Point Design with System Toolboxes Software” on page 19-2

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 19-4 “Scaling” on page 19-5 “Precision and Range” on page 19-6

Fixed-Point Data Types

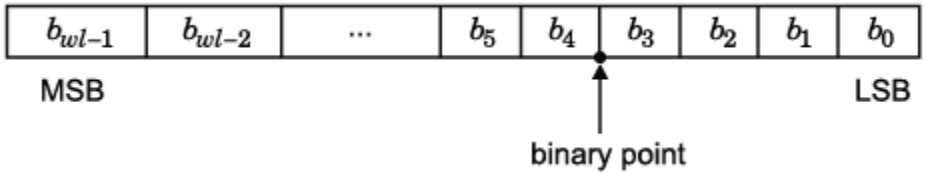
In digital hardware, numbers are stored in binary words. A binary word is a fixed-length sequence of bits (1's and 0's). The way hardware components or software functions interpret this sequence of 1's and 0's is defined by the data type.

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 19-8 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.

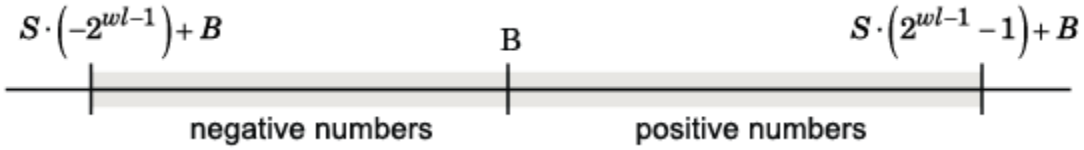
All System Toolbox blocks that support fixed-point data types support signals with binary-point only scaling. Many fixed-point blocks that do not perform arithmetic operations but merely rearrange data, such as Delay and Matrix Transpose, also support signals with [Slope Bias] scaling.

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.

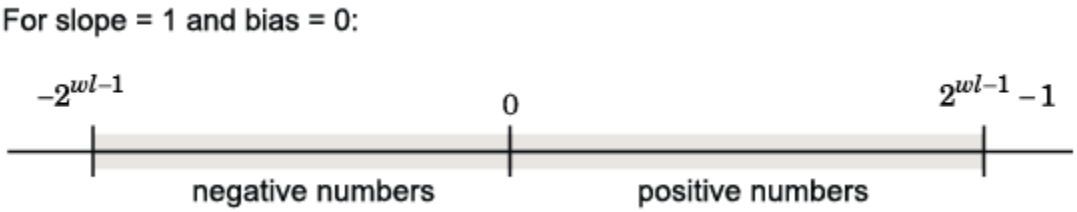
Range

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 19-8 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

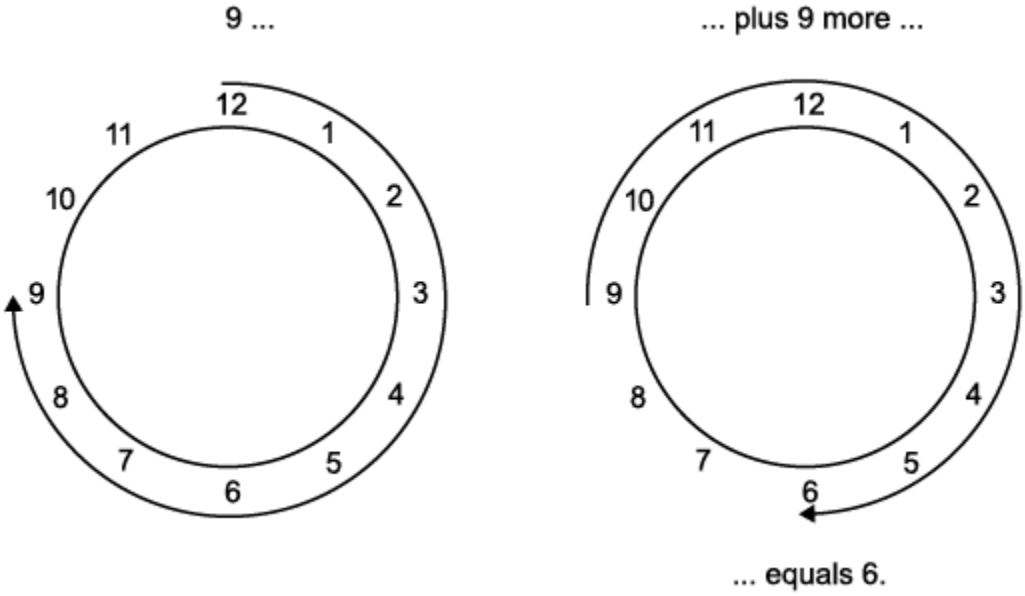
In this section...
“Modulo Arithmetic” on page 19-8
“Two's Complement” on page 19-8
“Addition and Subtraction” on page 19-9
“Multiplication” on page 19-10
“Casts” on page 19-12

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)
```

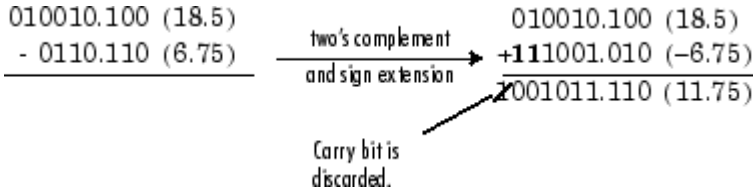
Addition and Subtraction

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):

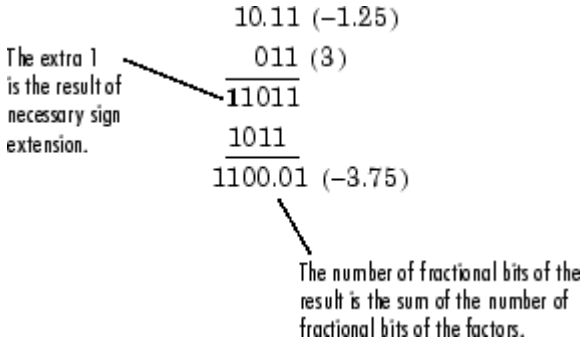


Most fixed-point DSP System Toolbox blocks that perform addition cast the adder inputs to an accumulator data type before performing the addition. Therefore, no further shifting is necessary during the addition to line up the binary points. See “Casts” on page 19-12 for more information.

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

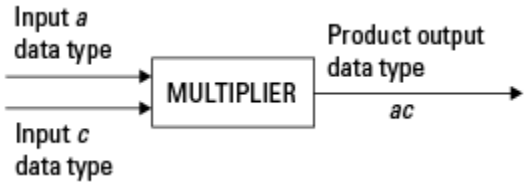
The following diagrams show the data types used for fixed-point multiplication in the System Toolbox software. The diagrams illustrate the differences between the data types used for real-real, complex-real, and complex-complex multiplication. See individual reference pages to determine whether a particular 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 19-12.

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.

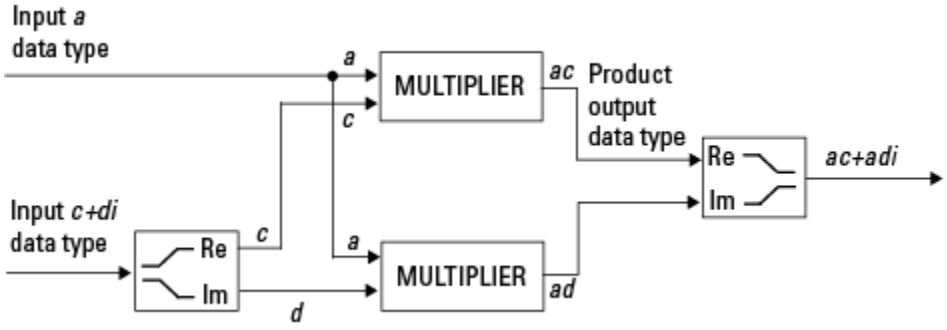
Real-Real Multiplication

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.



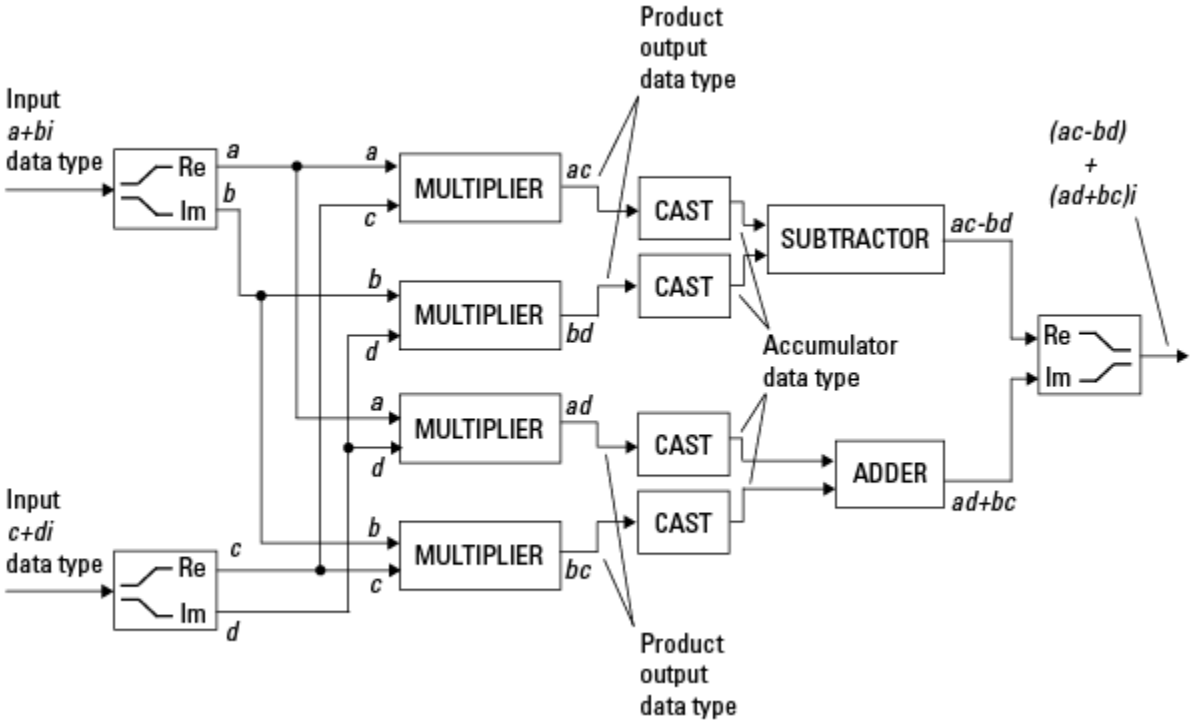
Real-Complex Multiplication

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 19-28. 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 19-28.

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.

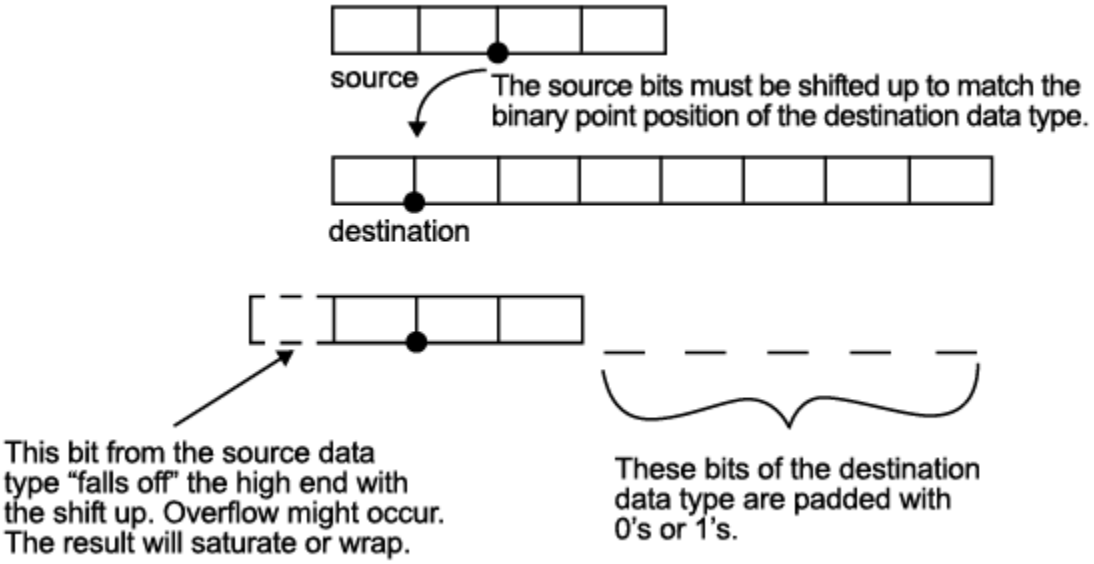
Note that although you cannot mix fixed-point and floating-point signals on the input and output ports of blocks, you can have fixed-point signals with different word and fraction lengths on the ports of blocks that support fixed-point signals.

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:



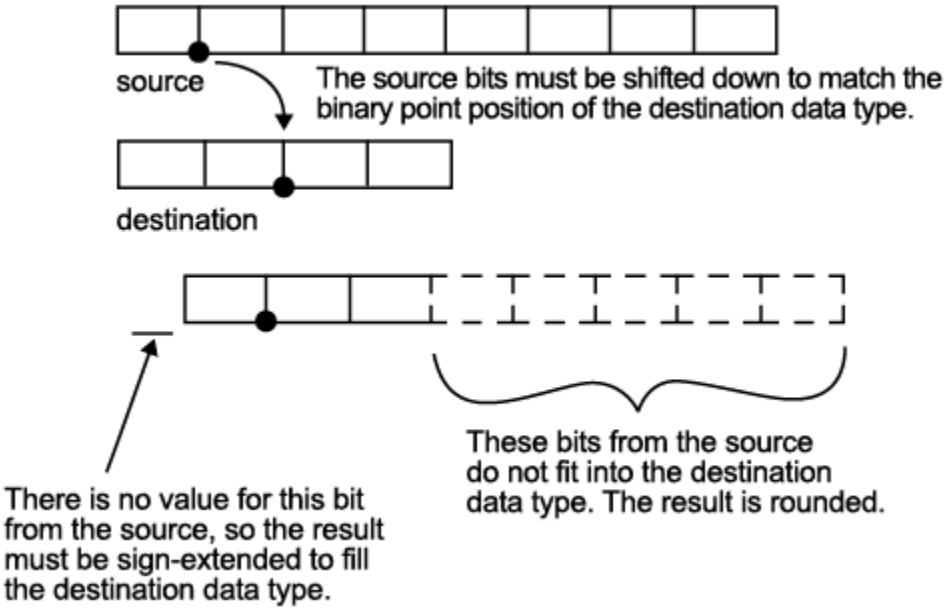
As the diagram shows, the source bits are shifted up so that the binary point matches the destination binary point position. The highest source bit does not fit, so overflow might occur and the result can saturate or wrap. The empty bits at the low end of the destination data type are padded with either 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 Design

In this section...
“Get Information About Fixed-Point System Objects” on page 19-15
“Set System Object Fixed-Point Properties” on page 19-17
“Full Precision for Fixed-Point System Objects” on page 19-18

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
`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
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
Signal Operations	
`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.ZeroCrossingDetector`	Zero crossing detector
Math Operations	
`dsp.HDLComplexToMagnitudeAngle`	Compute magnitude and phase angle of complex signal— optimized for HDL code generation

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”. 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

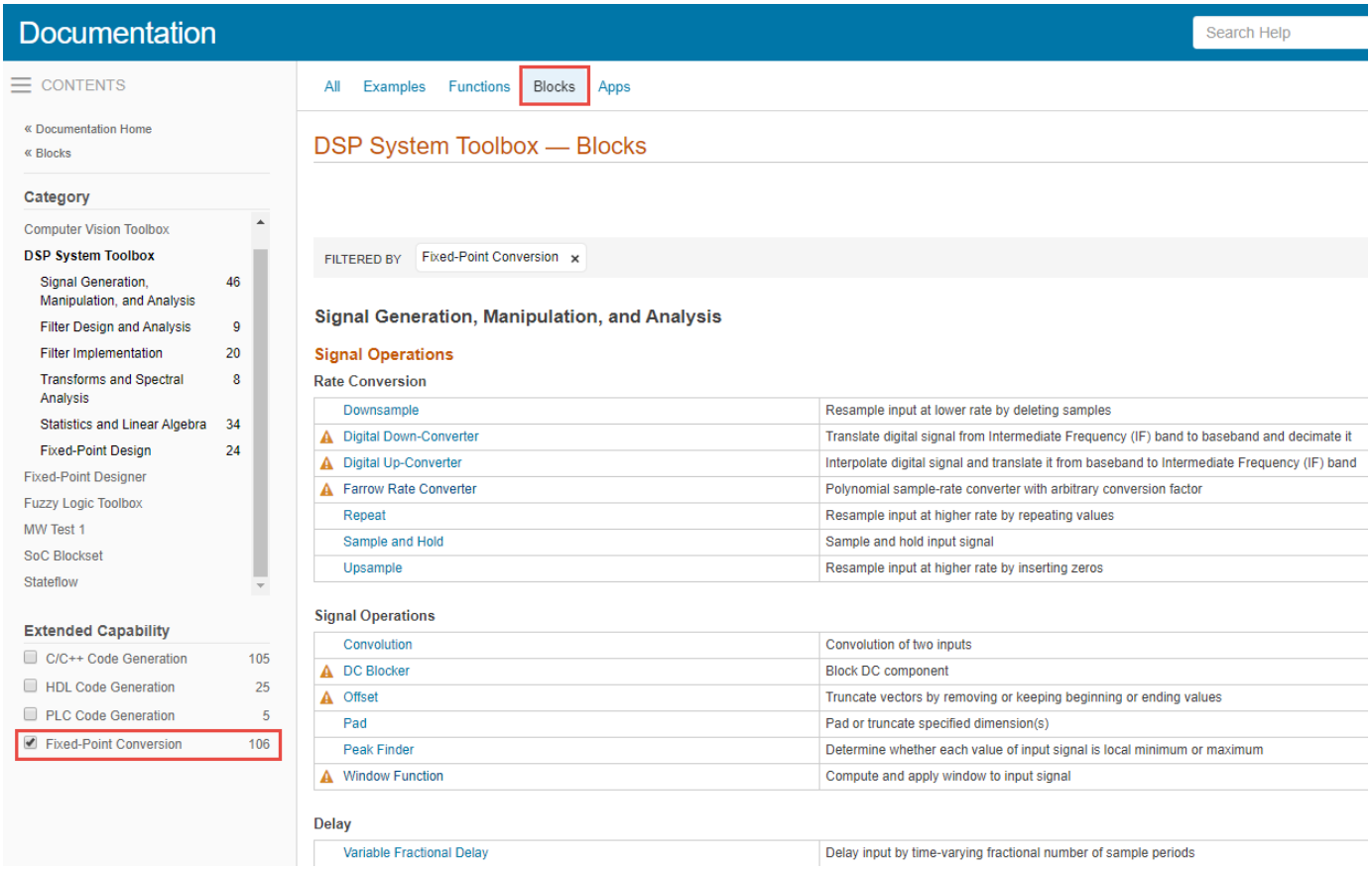
More About

- “Simulink Blocks in DSP System Toolbox that Support Fixed-Point Design” on page 19-19

Simulink Blocks in DSP System Toolbox that Support Fixed-Point Design

You can view a list of blocks that support fixed-point design in documentation by filtering the blocks reference list. Click **Blocks** right below the blue bar at the top of the Help window, then select the **Fixed-Point Conversion** check box at the bottom of the left column under **Extended Capability**. The blocks are listed in their respective categories. You can use the table of contents in the left column to navigate between the categories. Refer to the **Extended Capabilities > Fixed-Point Conversion** section of each block page for any notes and limitations.

To obtain this filtered list for DSP System Toolbox, click Blocks that Support Fixed-Point Design.



Alternatively, you can find this information in the Simulink block data type support table for the DSP System Toolbox. To access this table, type this command in the MATLAB Command Window.

```
showsignalblockdatatypetable
```

See Also

More About

- “System Objects in DSP System Toolbox that Support Fixed-Point Design” on page 19-15

System Objects Supported by Fixed-Point Converter App

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.BiquadFilter`
- `dsp.FIRDecimator`
- `dsp.FIRInterpolator`
- `dsp.FIRFilter` (Direct Form and Direct Form Transposed only)
- `dsp.FIRRateConverter`
- `dsp.VariableFractionalDelay`

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 19-21
- “Floating-Point to Fixed-Point Conversion of IIR Filters” on page 4-328

Convert dsp.FIRFilter Object to Fixed-Point Using the Fixed-Point Converter App

Convert 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],...
```

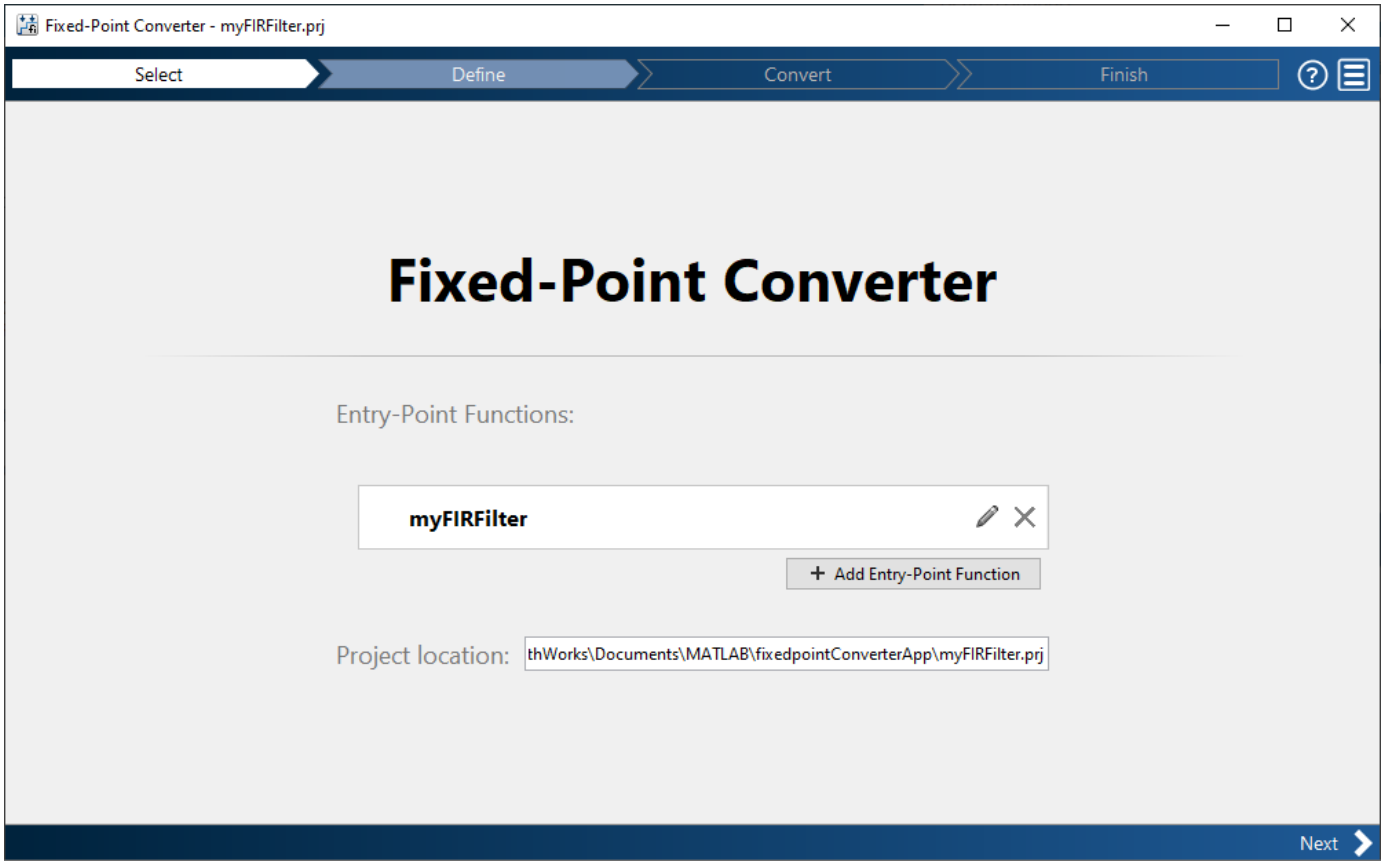
```
    'Title','Input Signal (Channel 1) Output Signal (Channel 2)');

% 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`.

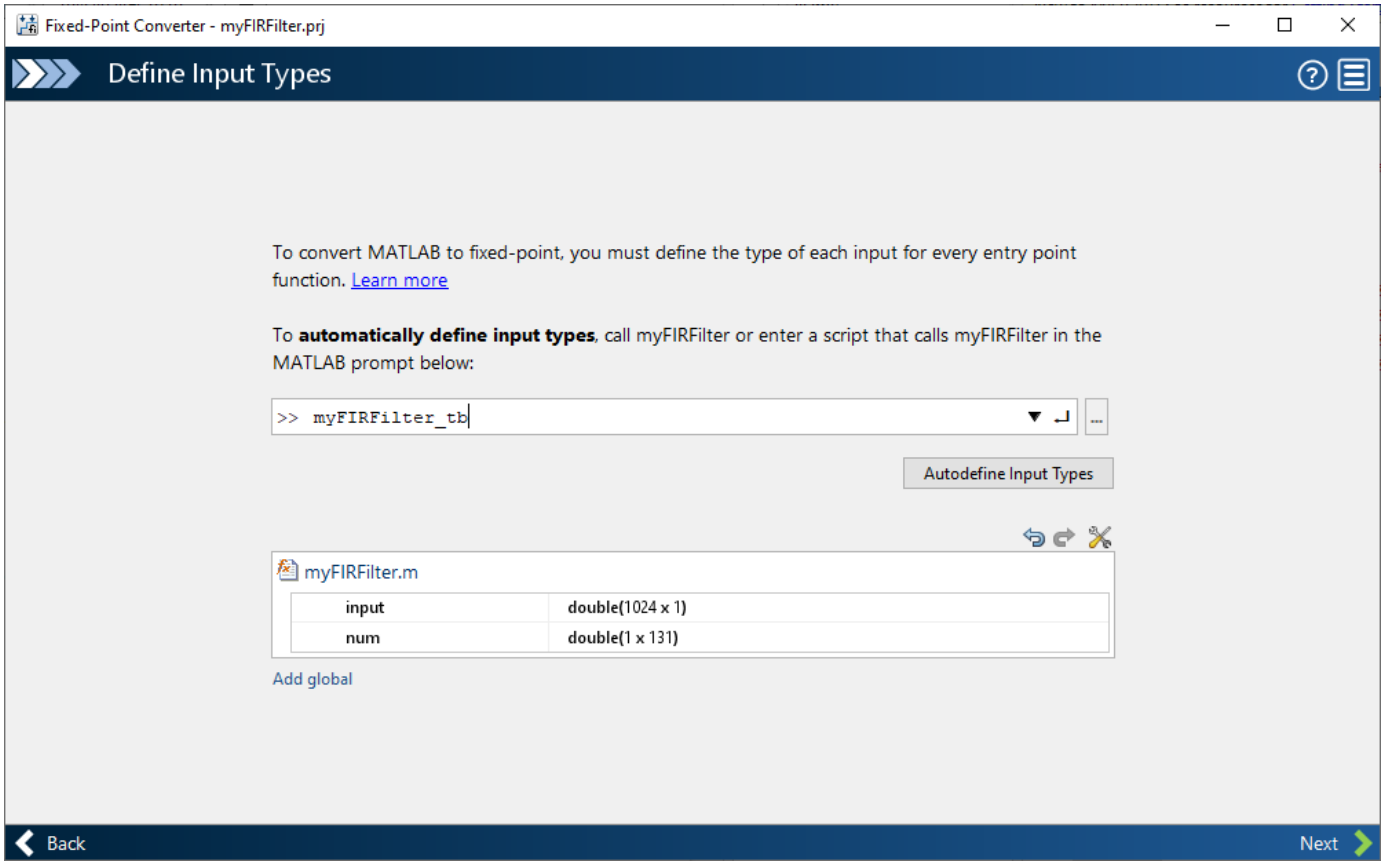


3. Click **Next** to go to the **Define Input Types** step.

   The app screens `myFIRFilter.m` for code violations and readiness issues. The app does not find issues in `myFIRFilter.m`.

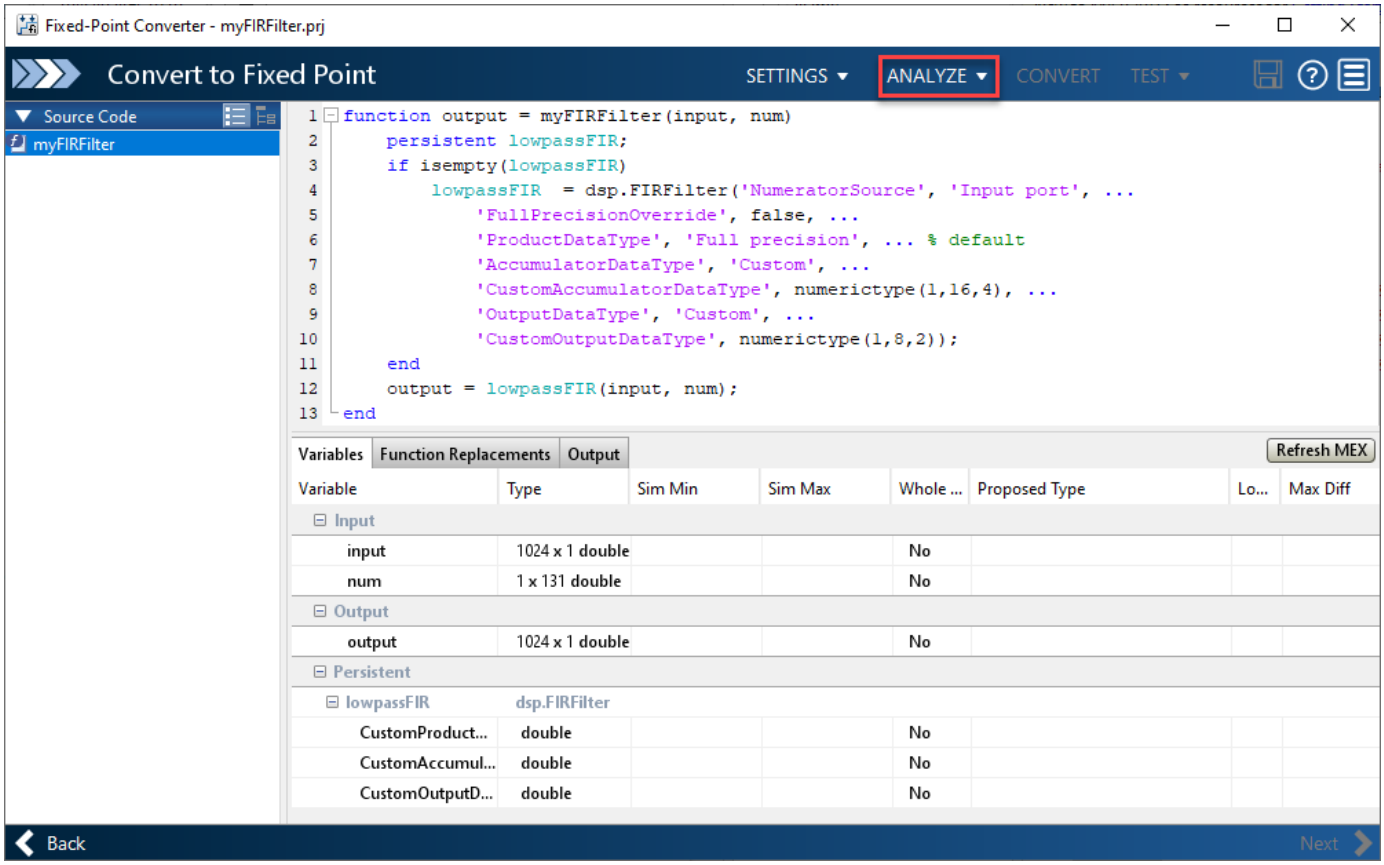
4 On the **Define Input Types** page, to add `myFIRFilter_tb` as a test file, browse to `myFIRFilter_tb.m`, and then click **Autodefine Input Types**.

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)`.

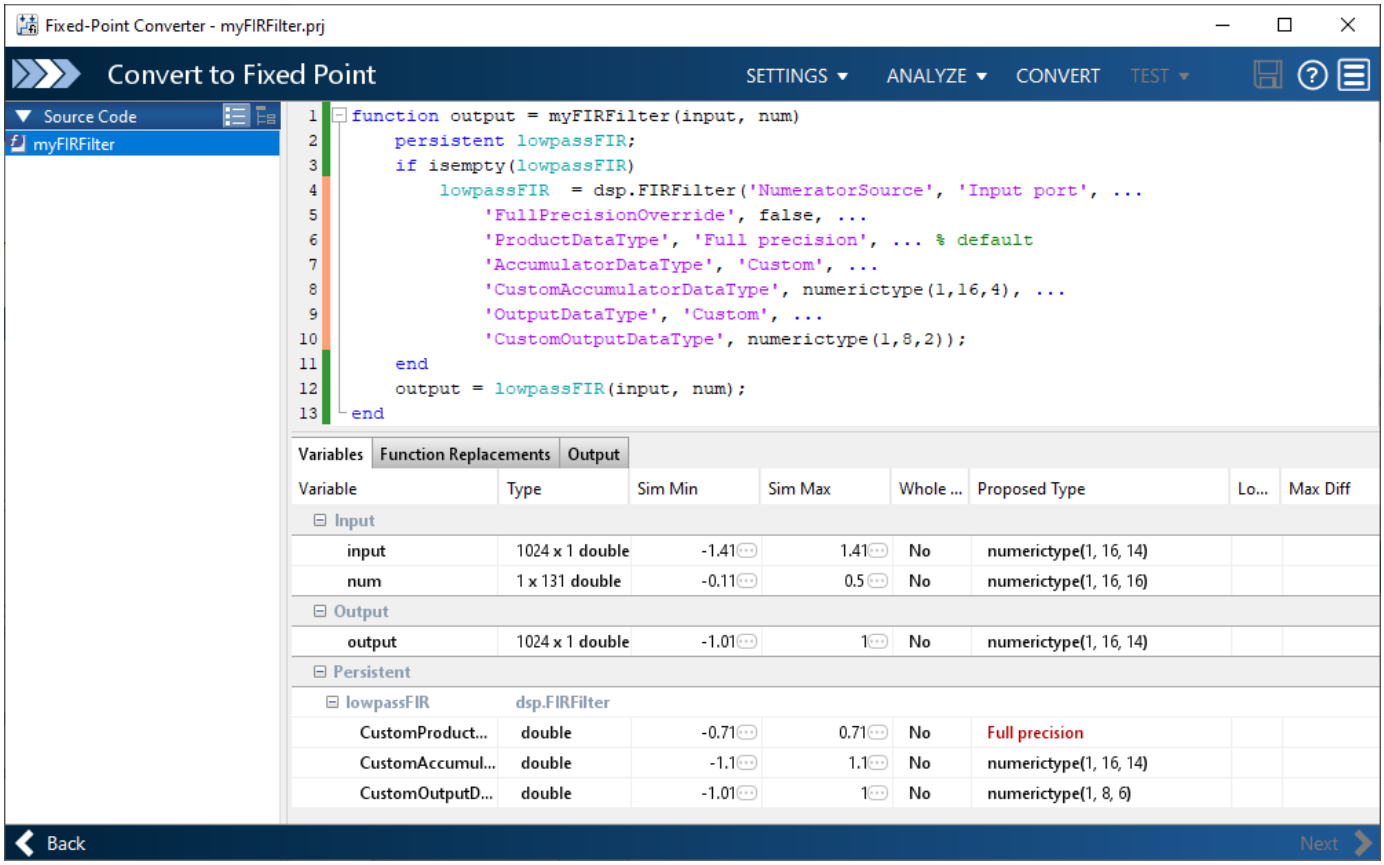


5 Click **Next** to go to the **Convert to Fixed Point** step.

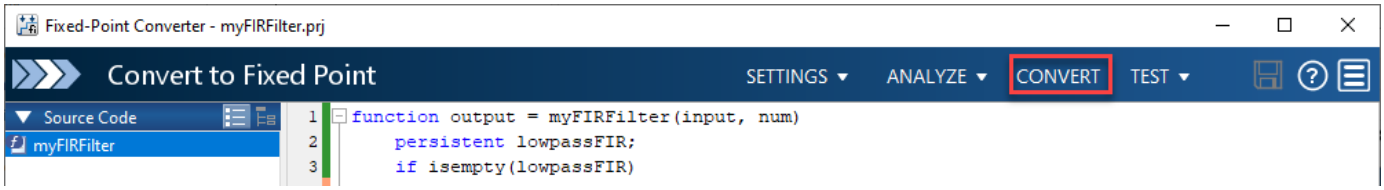
6 On the **Convert to Fixed Point** page, click **Analyze** to collect range information.



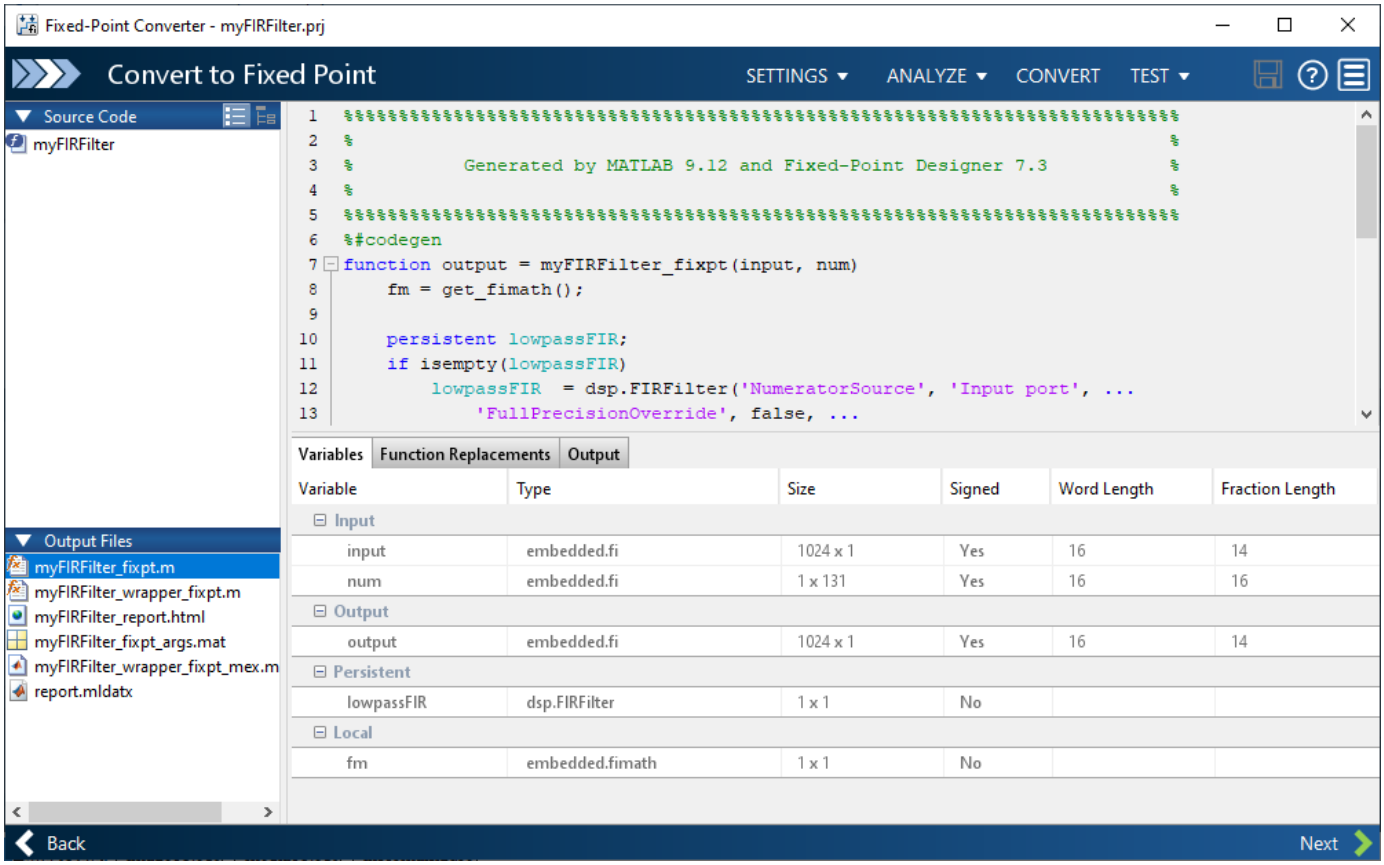
The **Variables** tab displays the collected range information and type proposals. Manually edit the data type proposals as needed.



7 Click **Convert** to apply the proposed data types to the function.



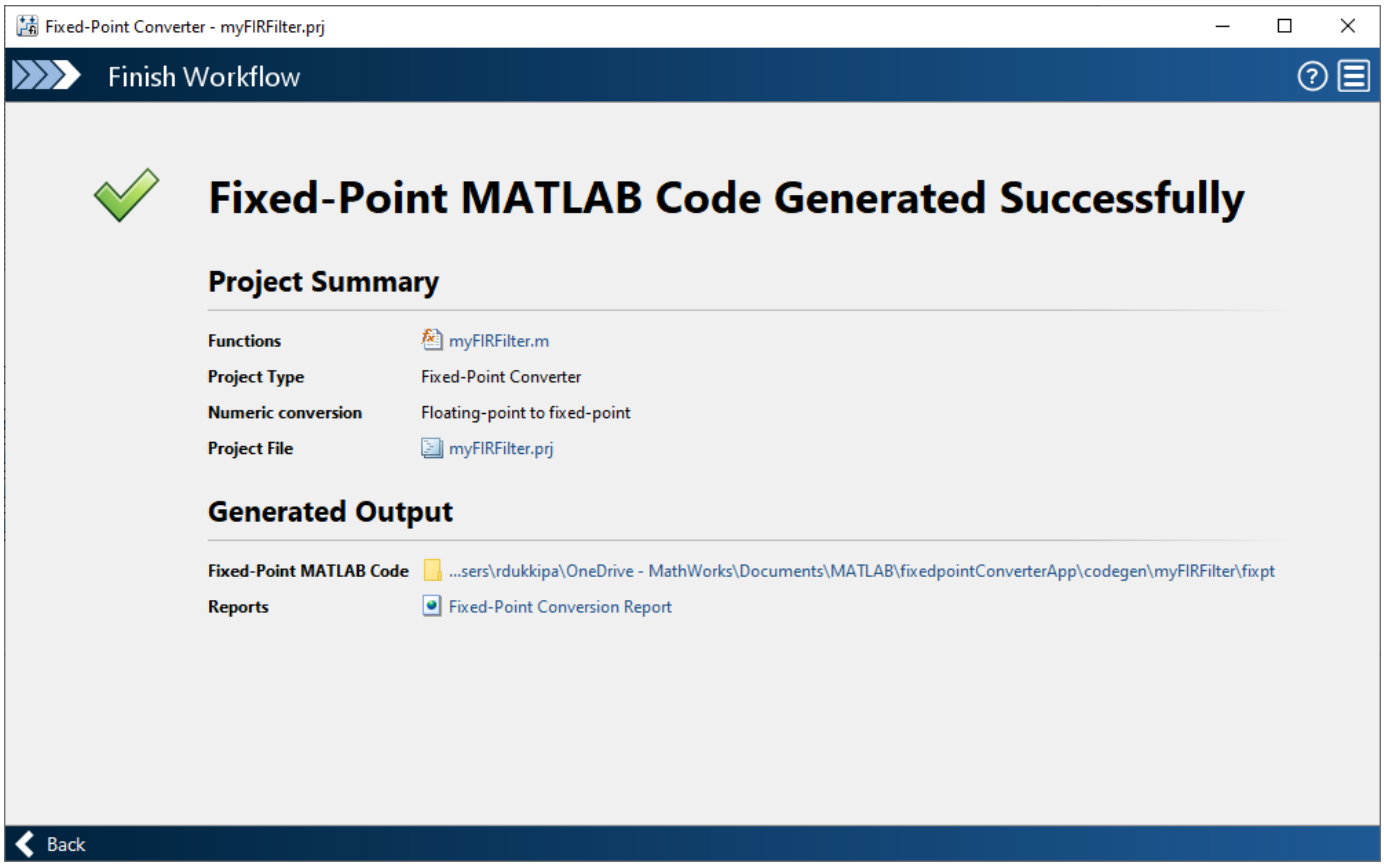
The Fixed-Point Converter app applies the proposed data types and generates a fixed-point function, `myFIRFilter_fixpt`.




```
%#codegen
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(1,16,4), ...
            'OutputDataType', 'Custom', ...
            'CustomOutputDataType', numerictype(1,8,2));
    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
```

8 Click **Next** to see the project summary details and links to the fixed-point MATLAB code and conversion report.



See Also

More About

- “System Objects Supported by Fixed-Point Converter App” on page 19-20
- “Floating-Point to Fixed-Point Conversion of IIR Filters” on page 4-328

Specify Fixed-Point Attributes for Blocks

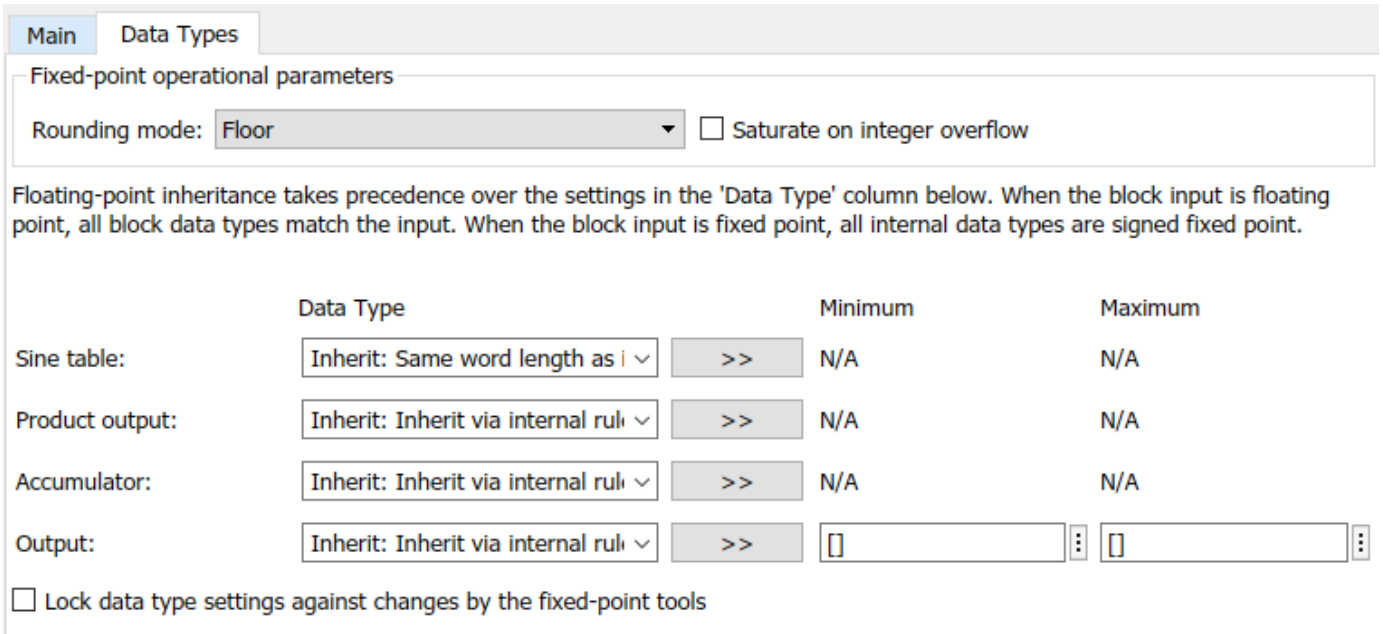
In this section...
“Fixed-Point Block Parameters” on page 19-28 “Specify System-Level Settings” on page 19-30 “Inherit via Internal Rule” on page 19-30 “Specify Data Types for Fixed-Point Blocks” on page 19-37

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 19-7 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 19-6 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 19-10 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 19-31
- “Internal Rule for Product Data Types” on page 19-31
- “Internal Rule for Output Data Types” on page 19-31
- “The Effect of the Hardware Implementation Pane on the Internal Rule” on page 19-31
- “Internal Rule Examples” on page 19-32

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 19-31 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 19-31 for more information.

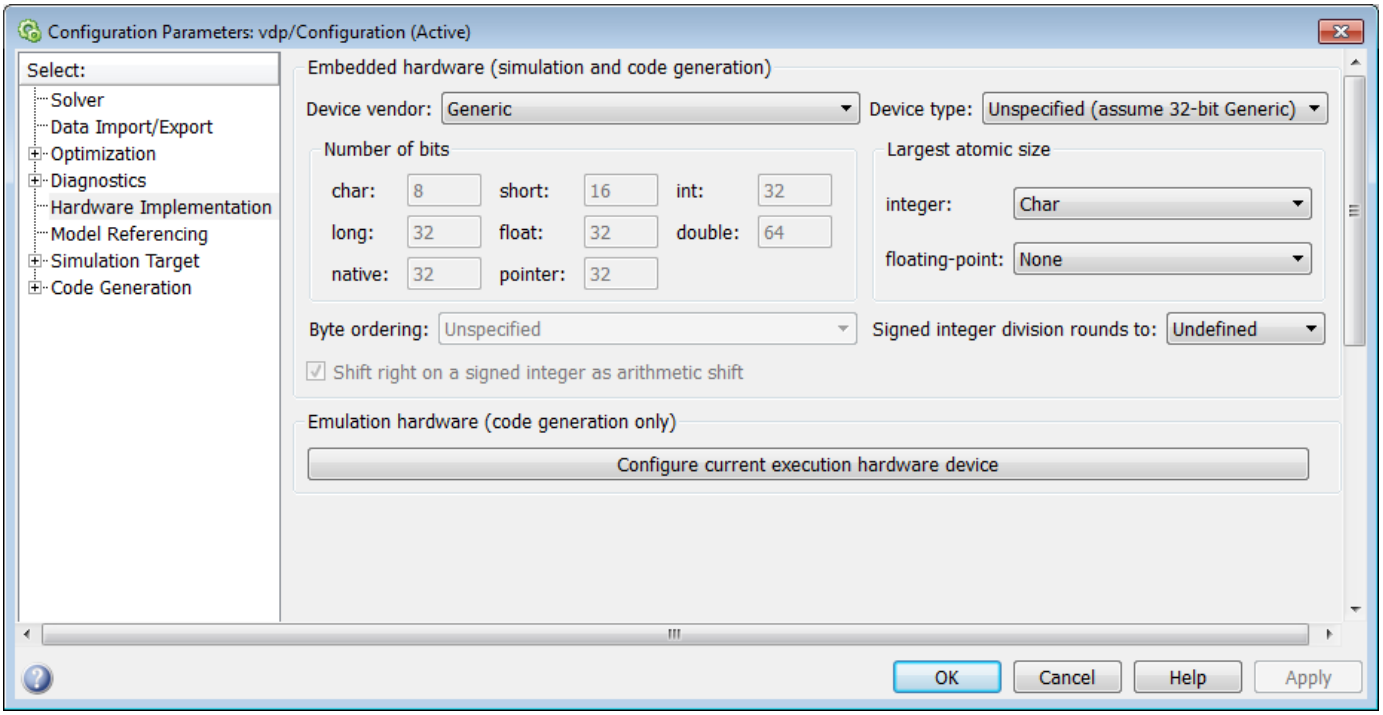
Internal Rule for Output Data Types

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 19-31.

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. To open this dialog box, click **Modeling** > **Model Settings** in the Simulink toolstrip.



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.

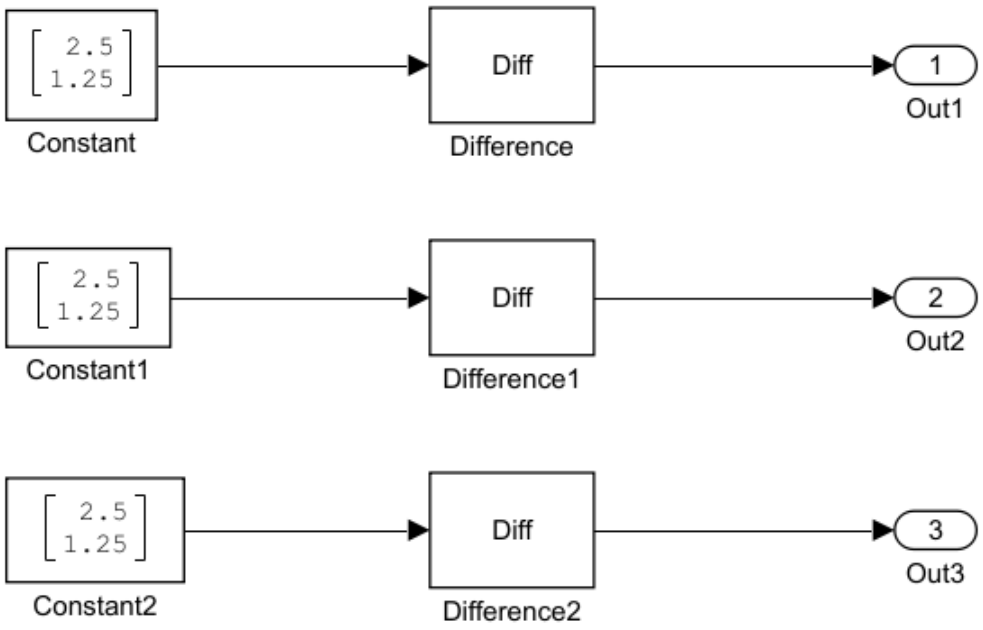
The largest word length allowed for Simulink and toolbox software on any target is 128 bits.

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 19-32 and product data types on page 19-35.

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:

$$WL_{idealaccumulator} = WL_{inputtoaccumulator} + \text{floor}(\log_2(numberofaccumulations)) + 1$$
$$WL_{idealaccumulator} = 9 + \text{floor}(\log_2(1)) + 1$$
$$WL_{idealaccumulator} = 9 + 0 + 1 = 10$$

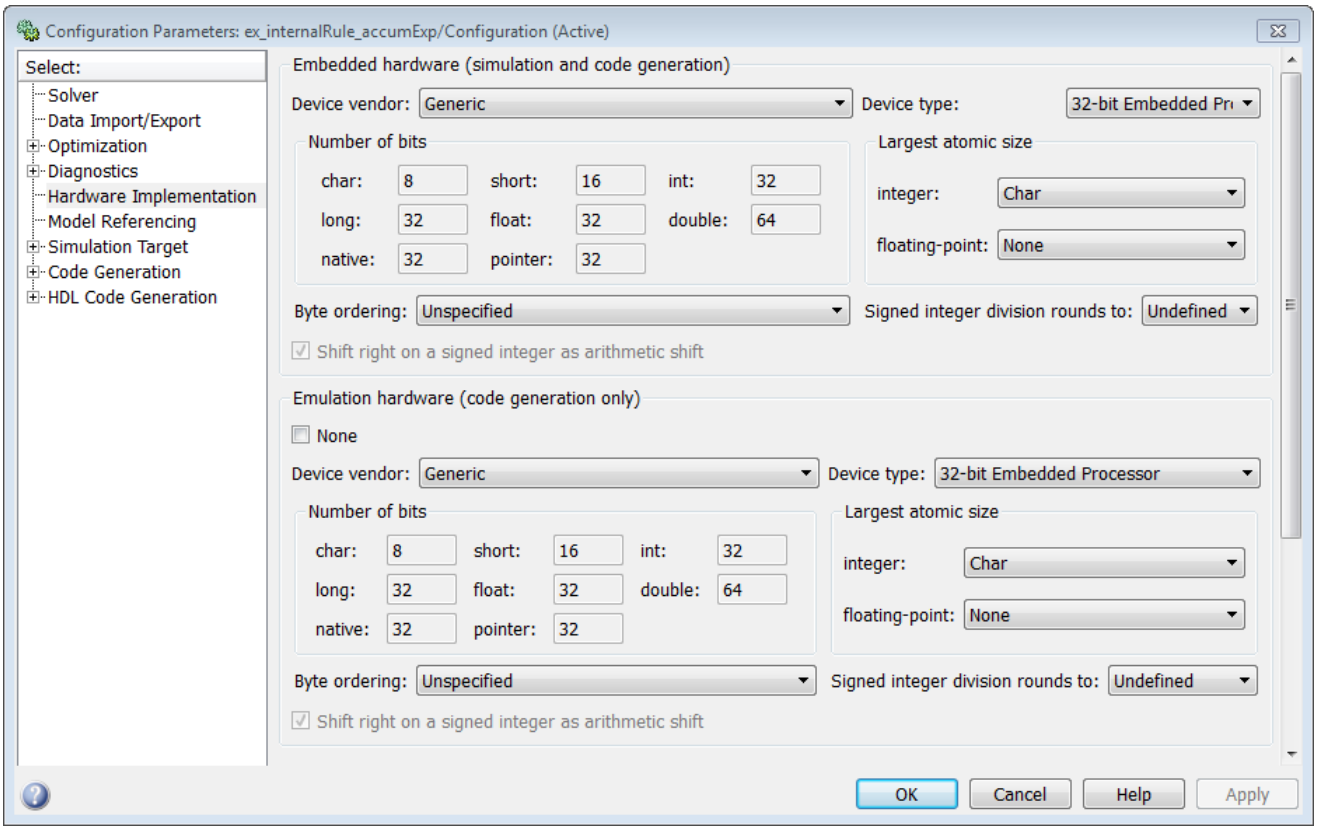
$$WL_{idealaccumulator1} = WL_{inputtoaccumulator1} + \text{floor}(\log_2(numberofaccumulations)) + 1$$
$$WL_{idealaccumulator1} = 16 + \text{floor}(\log_2(1)) + 1$$
$$WL_{idealaccumulator1} = 16 + 0 + 1 = 17$$

$$WL_{idealaccumulator2} = WL_{inputtoaccumulator2} + \text{floor}(\log_2(numberofaccumulations)) + 1$$
$$WL_{idealaccumulator2} = 127 + \text{floor}(\log_2(1)) + 1$$
$$WL_{idealaccumulator2} = 127 + 0 + 1 = 128$$

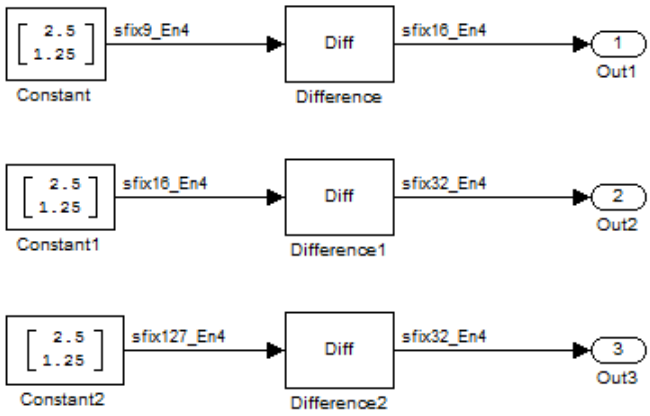
Calculate the full-precision fraction length, which is the same for each Matrix Sum block in this example:

$$FL_{idealaccumulator} = FL_{inputtoaccumulator}$$
$$FL_{idealaccumulator} = 4$$

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.

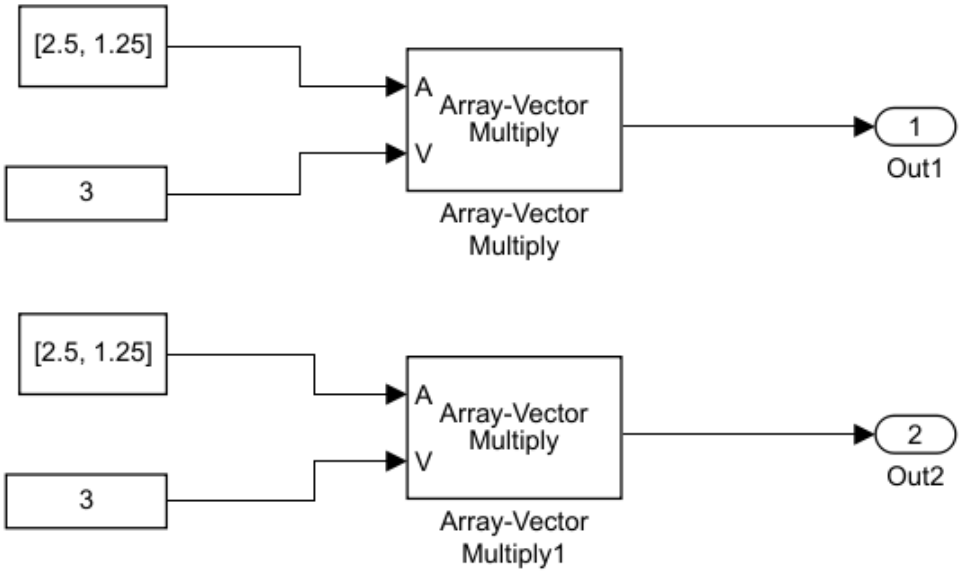


As you can see in the dialog box, this device has 8-, 16-, and 32-bit word lengths available. Therefore, the ideal word lengths of 10, 17, and 128 bits calculated by the internal rule cannot be used. Instead, the internal rule uses the next largest available word length in each case You can see this 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:

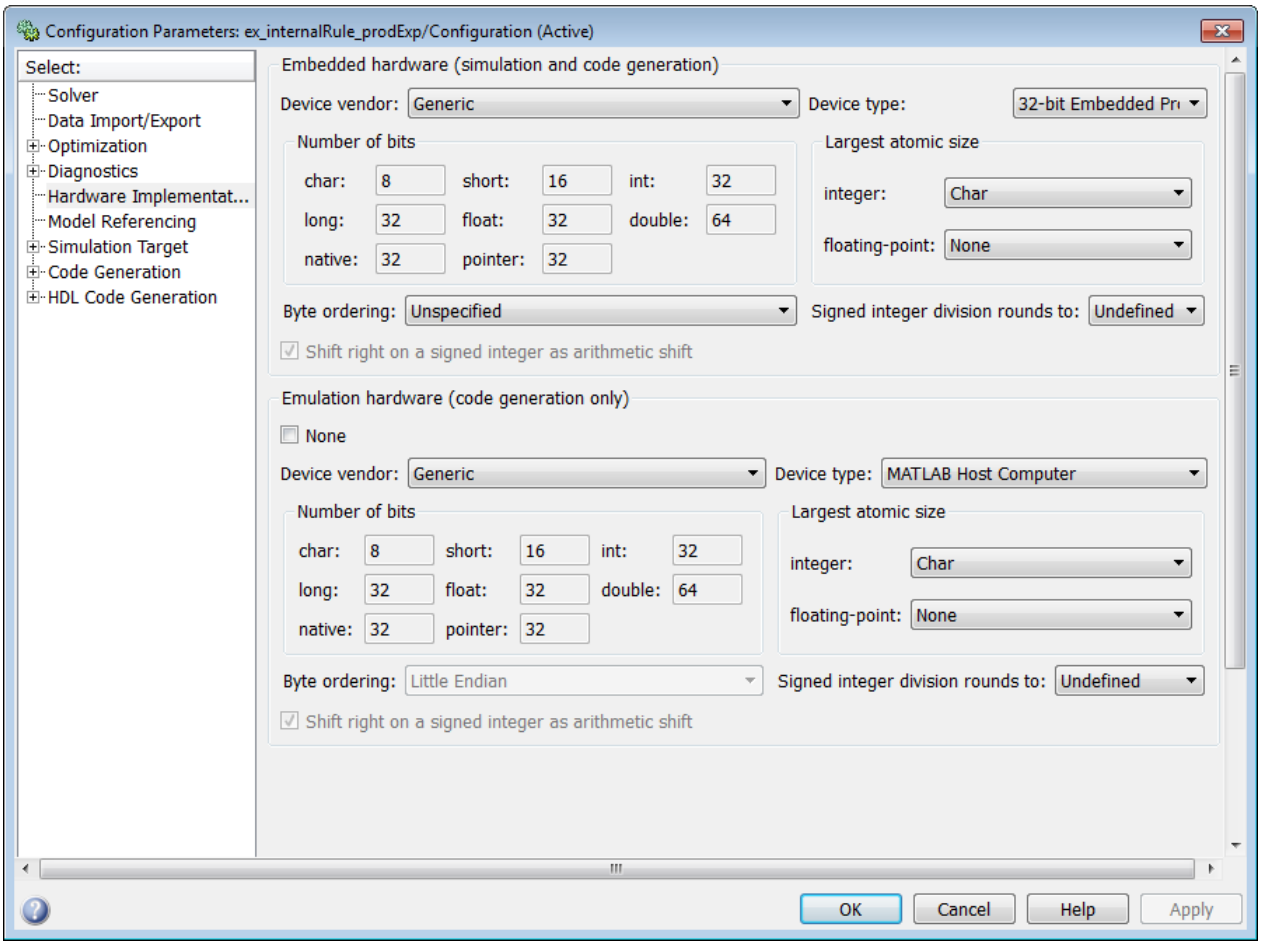
$$WL_{idealproduct} = WL_{inputa} + WL_{inputb}$$
$$WL_{idealproduct} = 7 + 5 = 12$$

$$WL_{idealproduct1} = WL_{inputa} + WL_{inputb}$$
$$WL_{idealproduct1} = 16 + 15 = 31$$

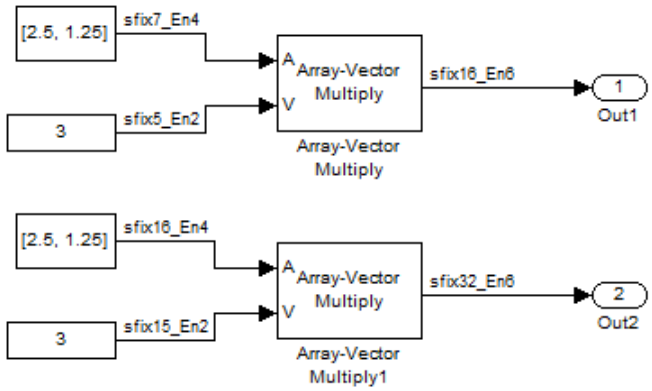
Calculate the full-precision fraction length, which is the same for each Array-Vector Multiply block in this example:

$$FL_{idealaccumulator} = FL_{inputtoaccumulator}$$
$$FL_{idealaccumulator} = 4$$

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.



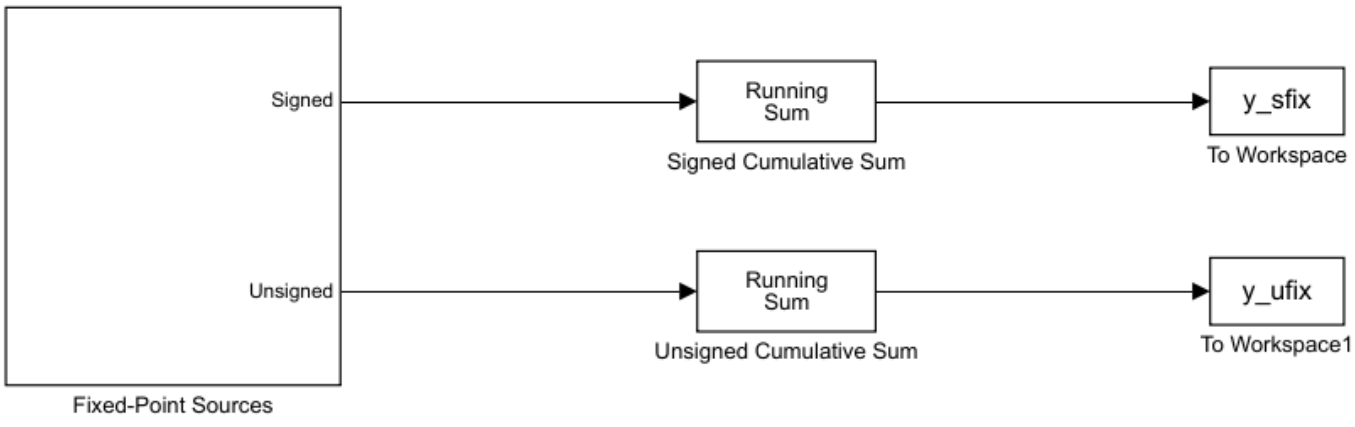
Specify Data Types for Fixed-Point Blocks

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 19-37
- "Use Data Type Override to Find a Floating-Point Benchmark" on page 19-41
- "Use the Fixed-Point Tool to Propose Fraction Lengths" on page 19-41
- "Examine the Results and Accept the Proposed Scaling" on page 19-42

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:

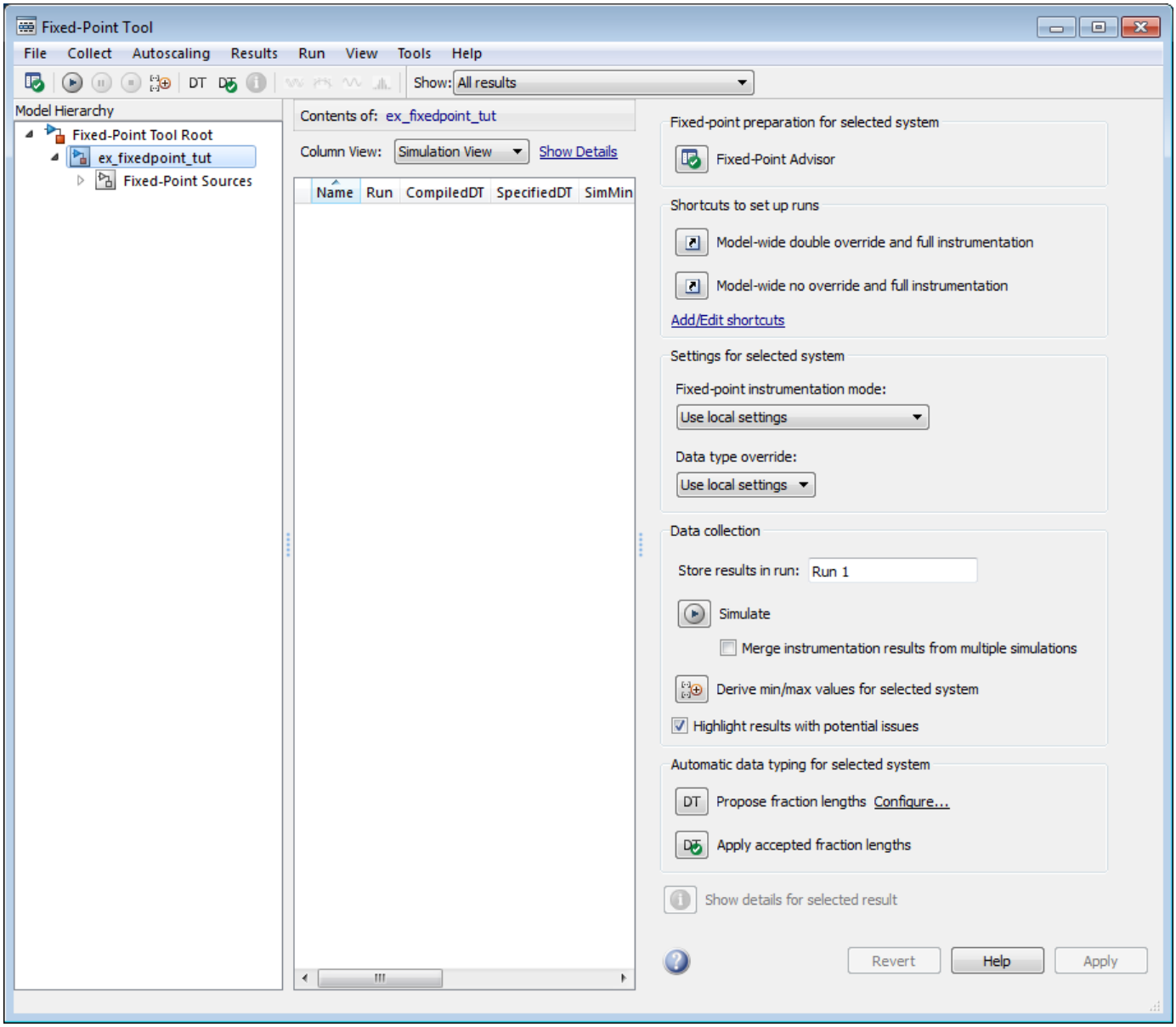
- The Signed source has a word length of 16 bits and a fraction length of 15 bits.
- The Unsigned source has a word length of 16 bits and a fraction length of 16 bits.

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:

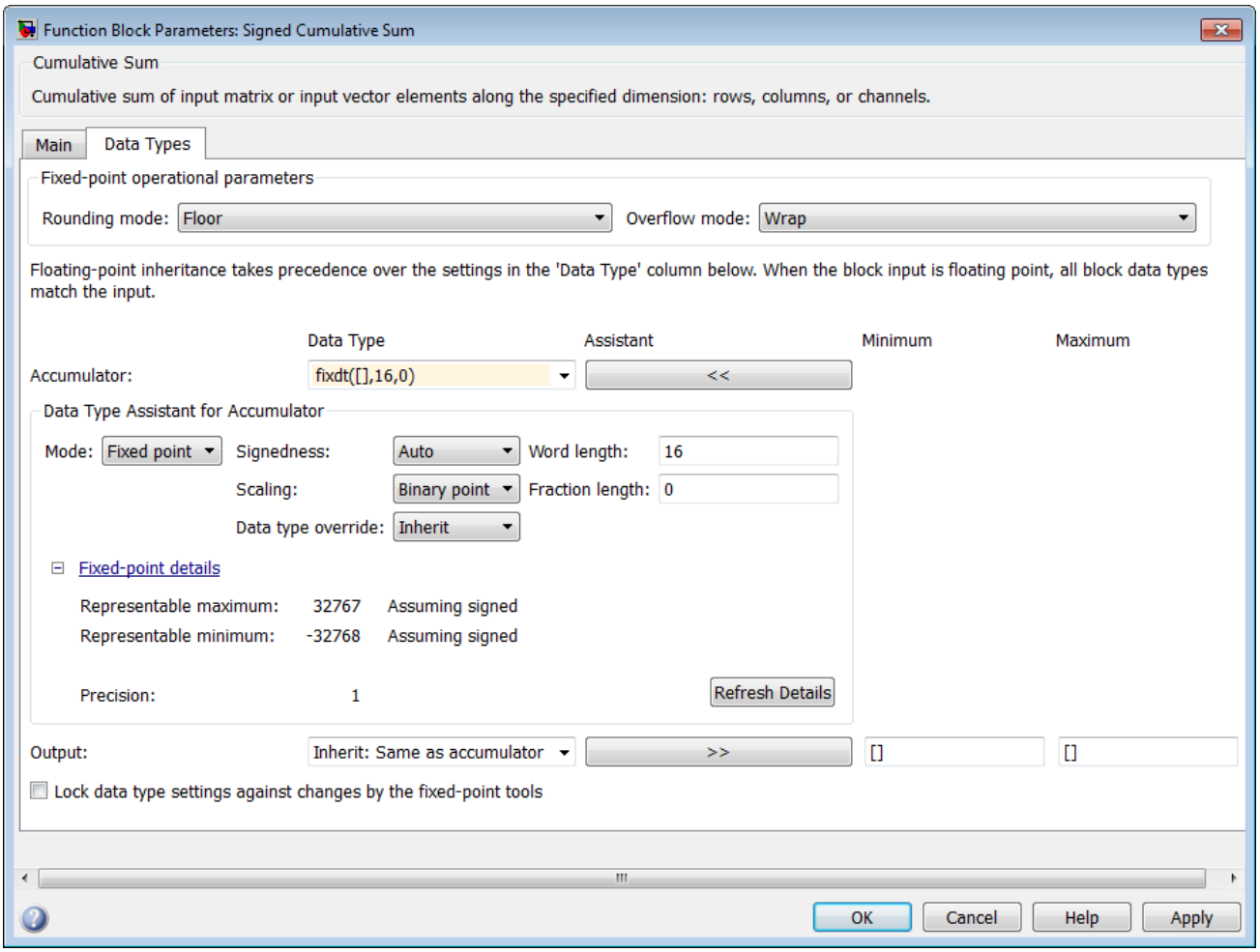
- **Name** — Provides the name of each signal in the following format: `Subsystem Name/Block Name: Signal Name`.
- **SimDT** — The simulation data type of each logged signal.
- **SpecifiedDT** — The data type specified on the block dialog for each signal.
- **SimMin** — The smallest representable value 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.

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 19-43 “Vector Quantizers” on page 19-46

Scalar Quantizers

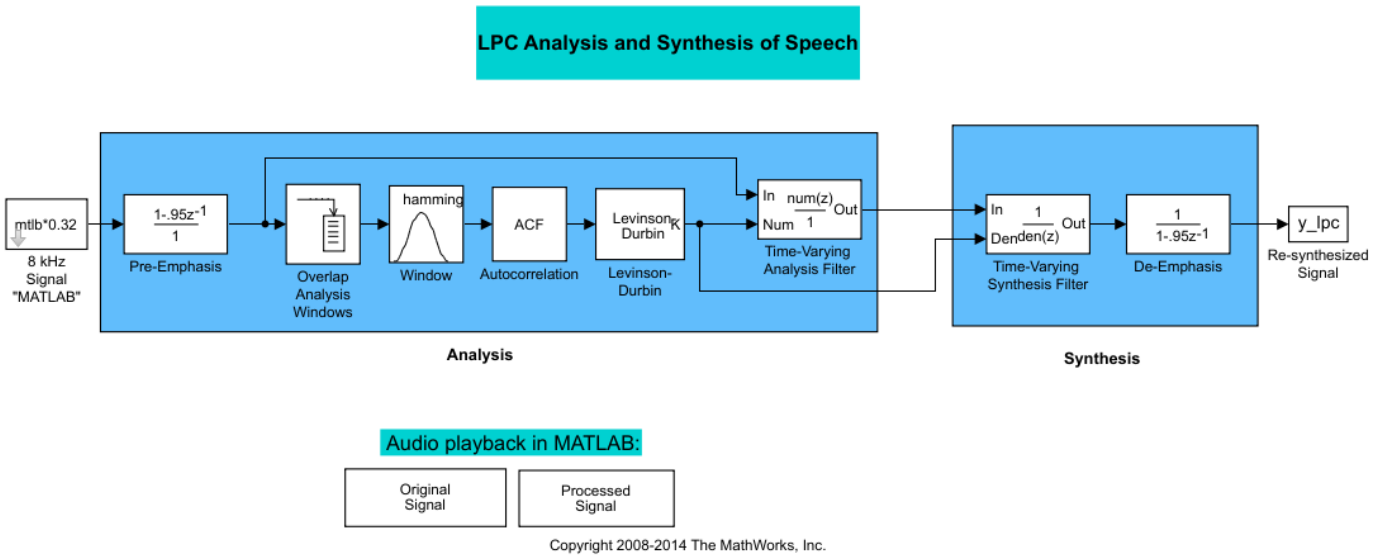
- “Analysis and Synthesis of Speech” on page 19-43
- “Identify Your Residual Signal and Reflection Coefficients” on page 19-44
- “Add a Scalar Quantizer” on page 19-45

Analysis and Synthesis of Speech

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.

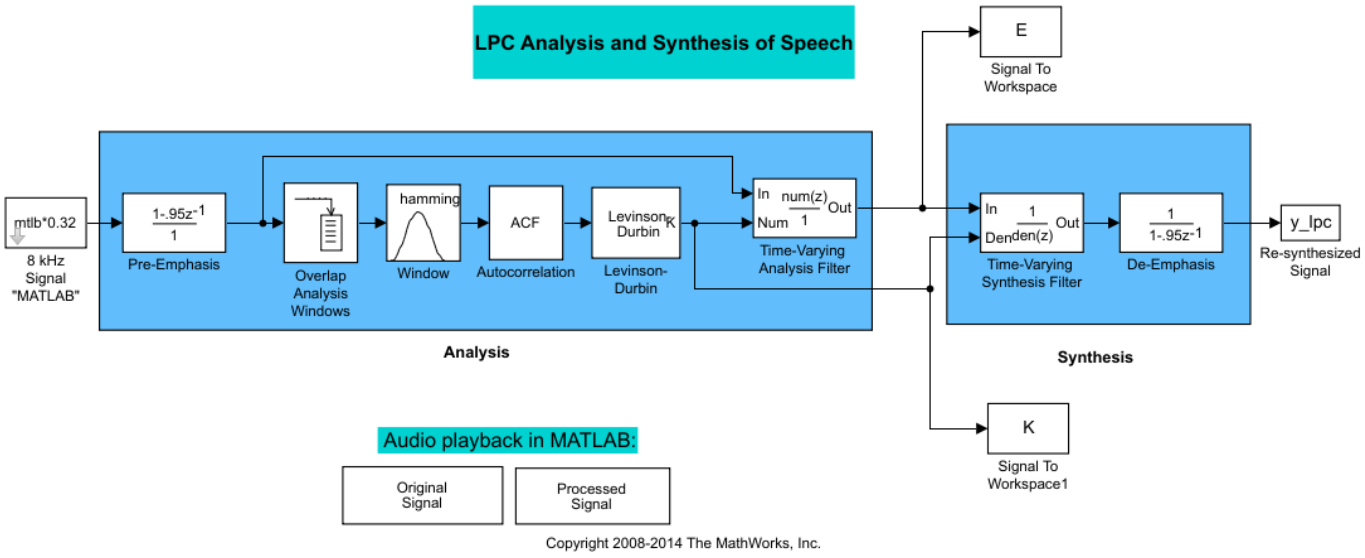
To better approximate a real-world speech analysis and synthesis system, quantize the residual signal and reflection coefficients before they are transmitted.

Identify Your Residual Signal and Reflection Coefficients

In the previous topic, “Analysis and Synthesis of Speech” on page 19-43, 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.

Add a Scalar Quantizer

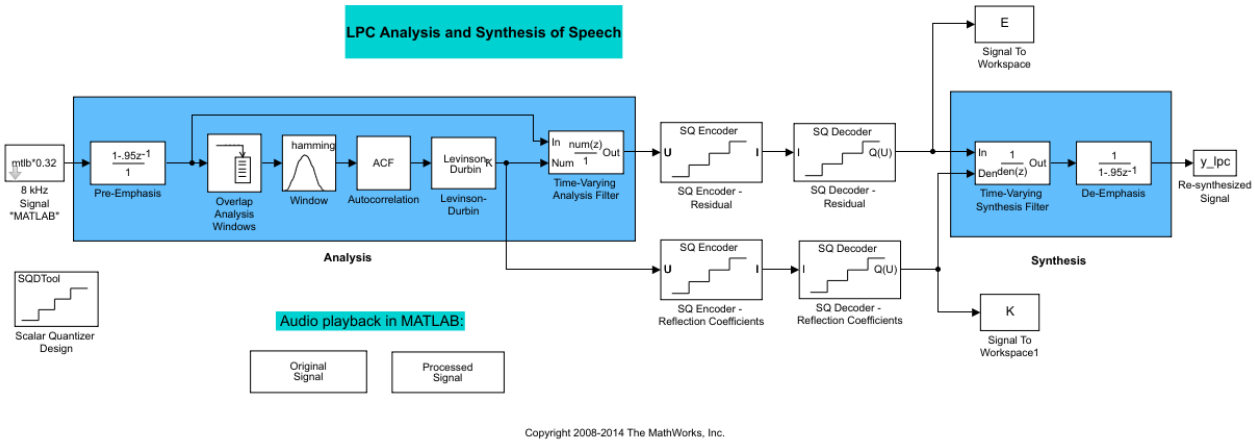
In this topic, you add scalar quantizer encoders and decoders to quantize the residual signal, *E*, and the reflection coefficients, *K*:

1 If the model you created in “Identify Your Residual Signal and Reflection Coefficients” on page 19-44 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 Encoder and Scalar Quantizer Decoder blocks to the model for each signal you want to quantize. Quantize the residual signal, *E*, and the reflection coefficients, *K*

4 Save the model as `ex_sq_example3`. Your model should look similar to the following figure.



5 Run your model.

6 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 19-46.

Vector Quantizers

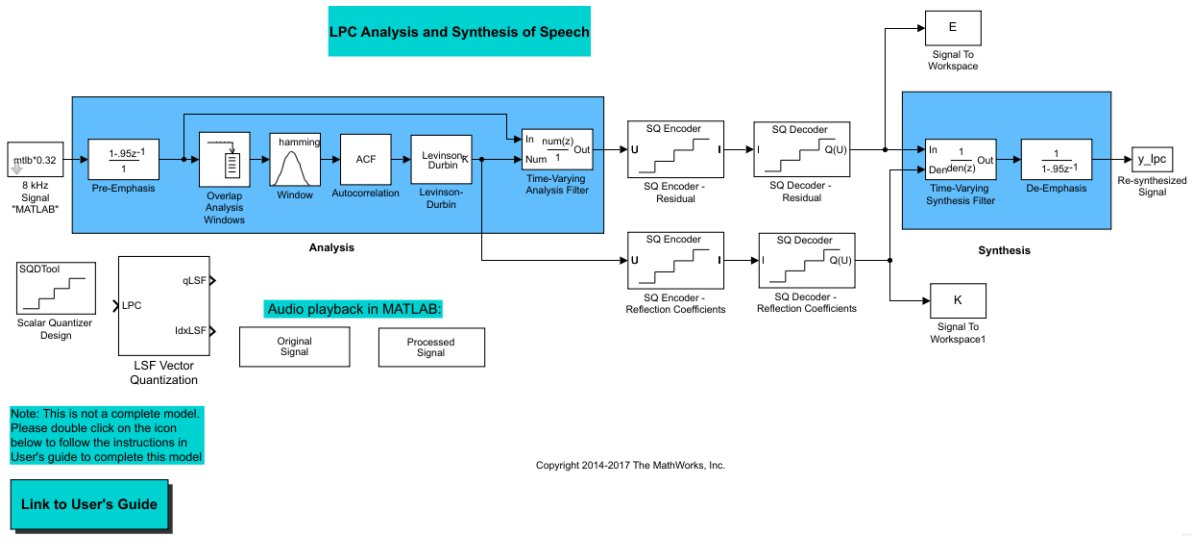
- “Build Your Vector Quantizer Model” on page 19-46
- “Configure and Run Your Model” on page 19-47

Build Your Vector Quantizer Model

In the previous section, you quantized 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 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.



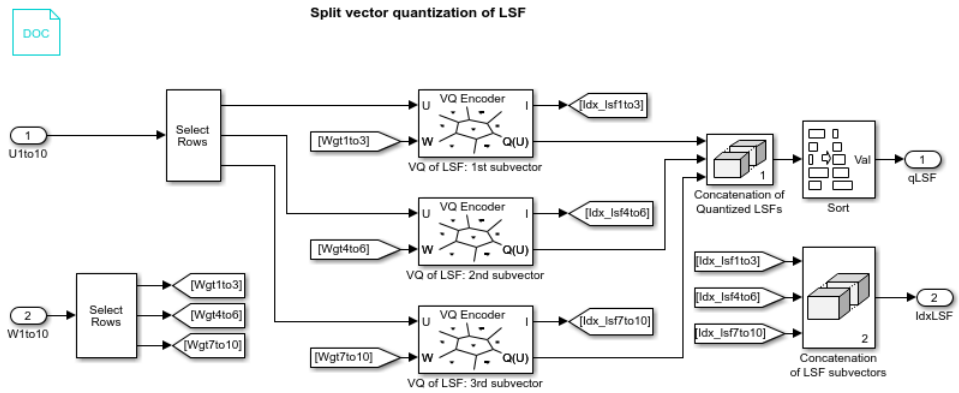
You have modified your model to include a subsystem capable of vector quantization. In the next topic, you reset your model parameters to quantize your reflection coefficients using the split vector quantization method.

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.

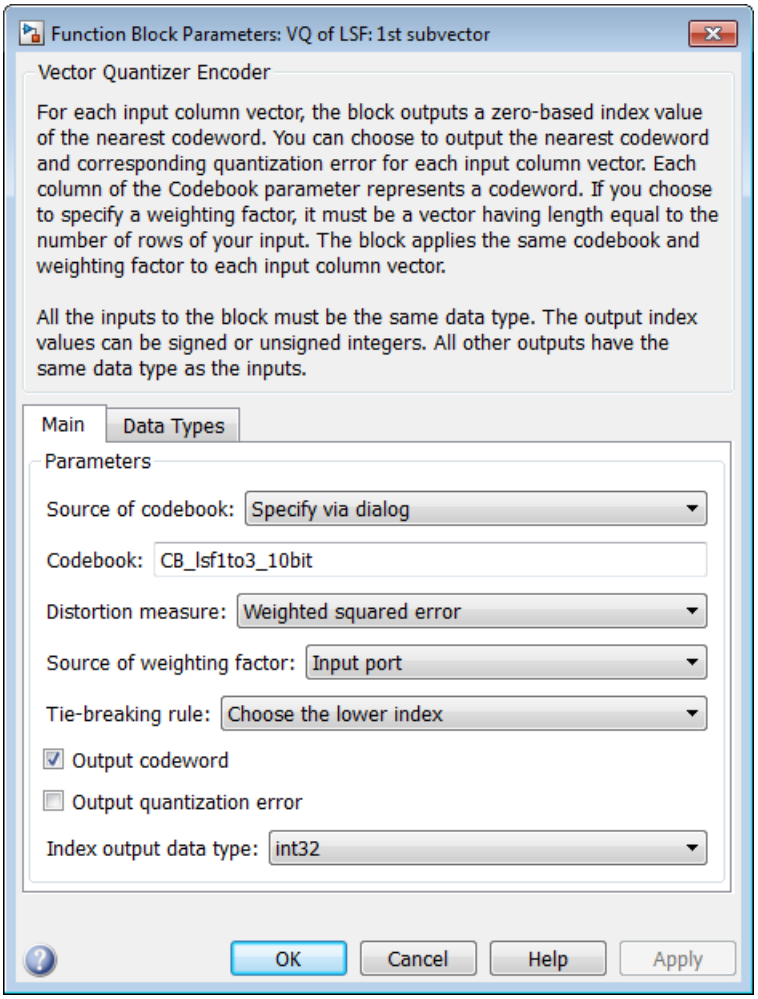
1. If the model you created in “Build Your Vector Quantizer Model” on page 19-46 is not open on your desktop, you can open an equivalent model by typing ex_vq_example2 at the MATLAB command prompt.
2. Double-click the LSF Vector Quantization subsystem, and then double-click the LSF Split VQ subsystem.

   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**.

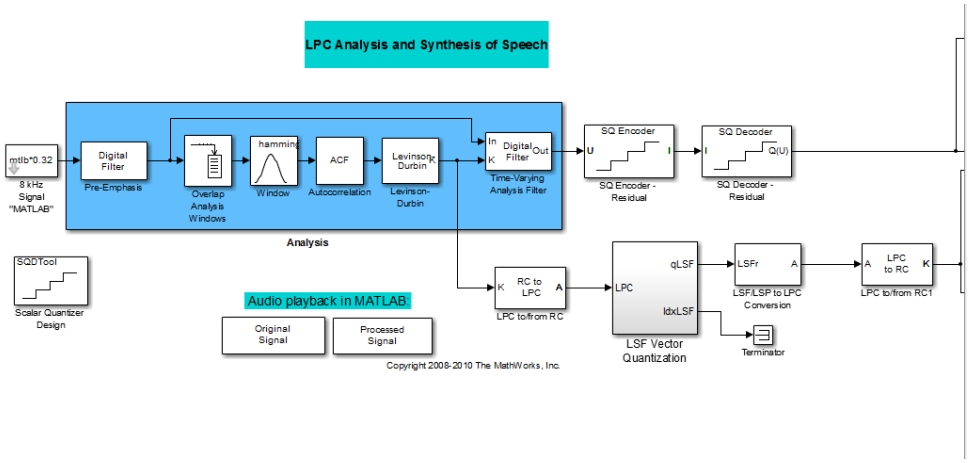
This parameter controls the number of linear polynomial coefficients (LPCs) that are input to the split vector quantization method.

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 19-50
“Build the FIR Filter” on page 19-50
“Set the Filter Parameters to Work with Integers” on page 19-51
“Create a Test Signal for the Filter” on page 19-52
“Filter the Test Signal” on page 19-52
“Truncate the Output WordLength” on page 19-54
“Scale the Output” on page 19-56
“Configure Filter Parameters to Work with Integers Using the set2int Method” on page 19-59

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:

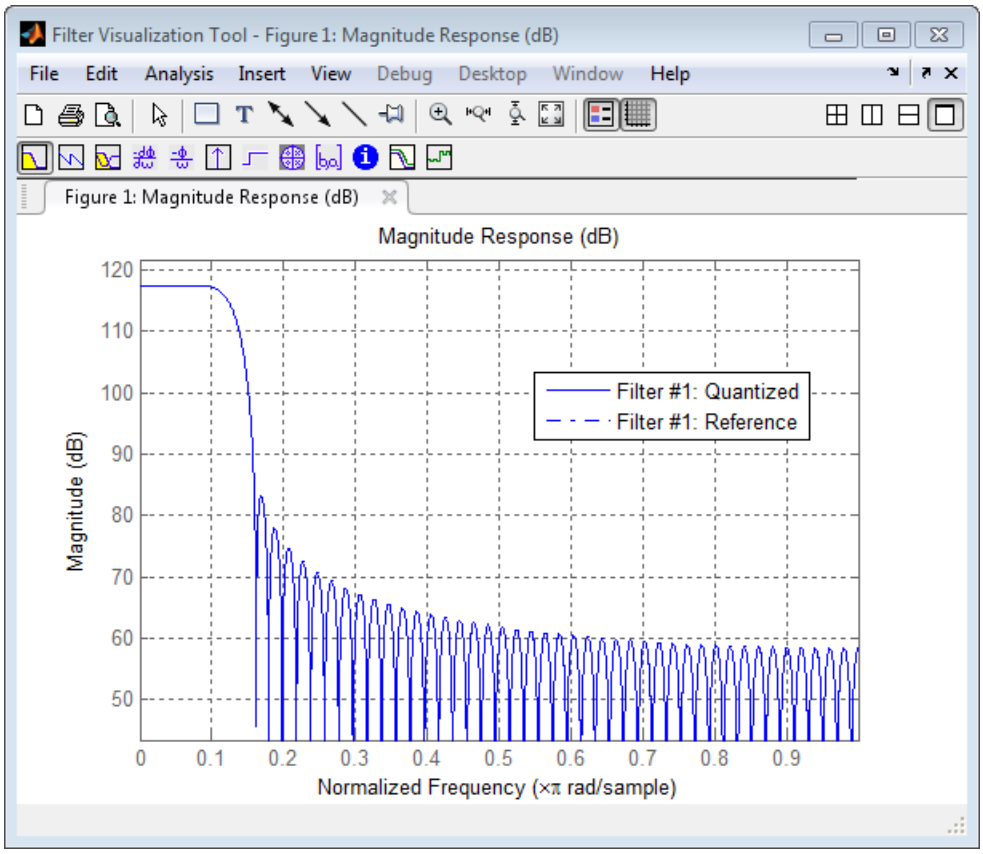
```
h.Arithmetic = 'fixed';
h.CoeffWordLength = 18;
```

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:

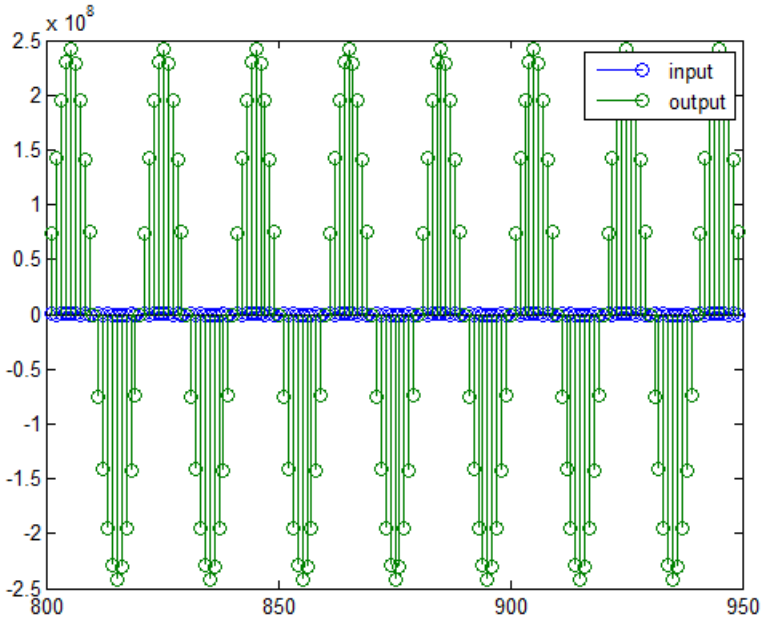
```
ysc = filter(h, xsc);
```

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
```

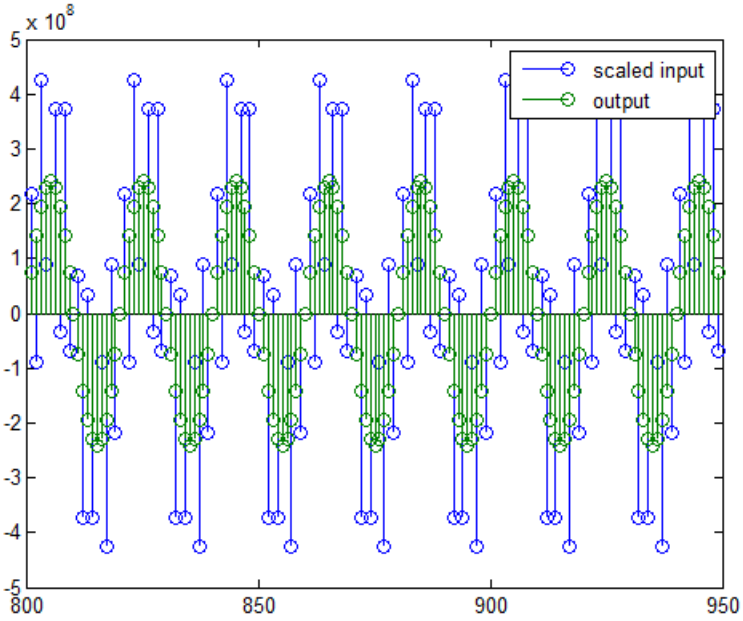
Now you can examine the output compared to the input. This example is plotting only the last few samples to minimize the effect of transients:

```
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:

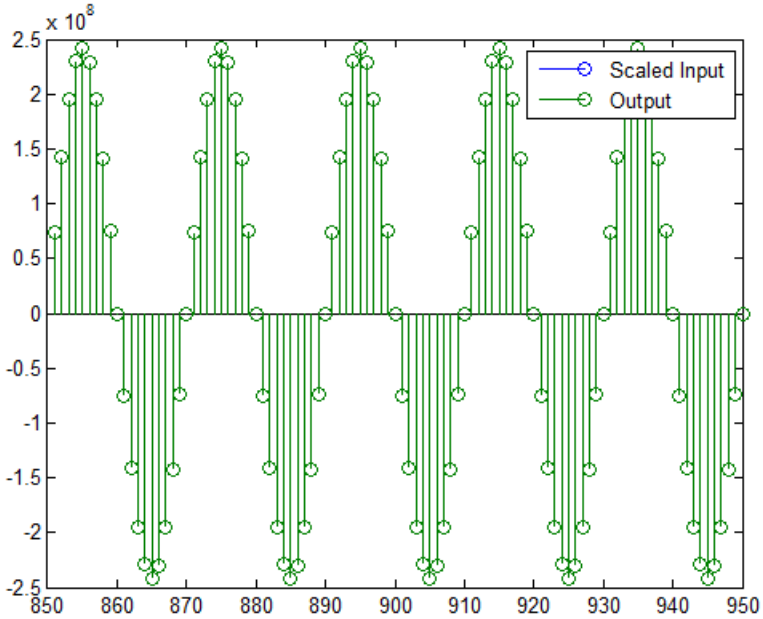
```
norm(double(yout) - double(yout2))

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:

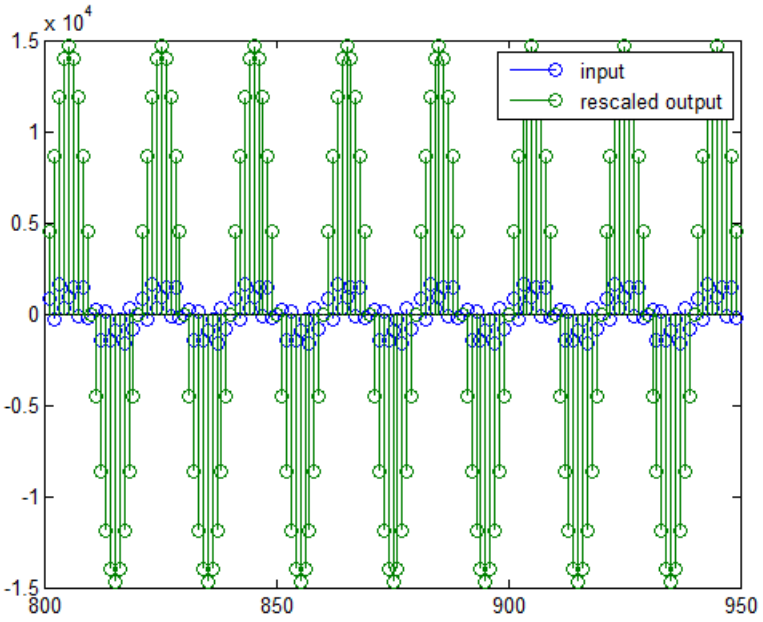
```
stem(n(yidx), [double(yout(yidx)'), double(ysc(yidx)')]);
axis([850 950 -2.5e8 2.5e8]);
legend('Scaled Input', 'Output');
set(gcf, 'color', 'white');
```



Scale the Output

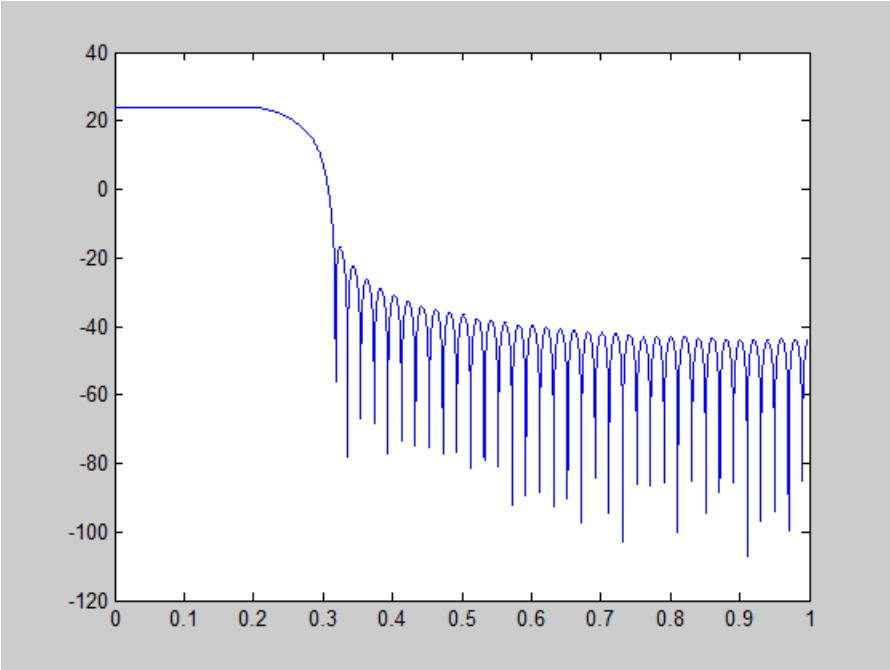
Because the filter in this example has such a large gain, the output is at a different scale than the input. This scaling is purely theoretical however, and you can scale the data however you like. In this case, you have 16 bits for the output, but you can attach whatever scaling you choose. It would 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:

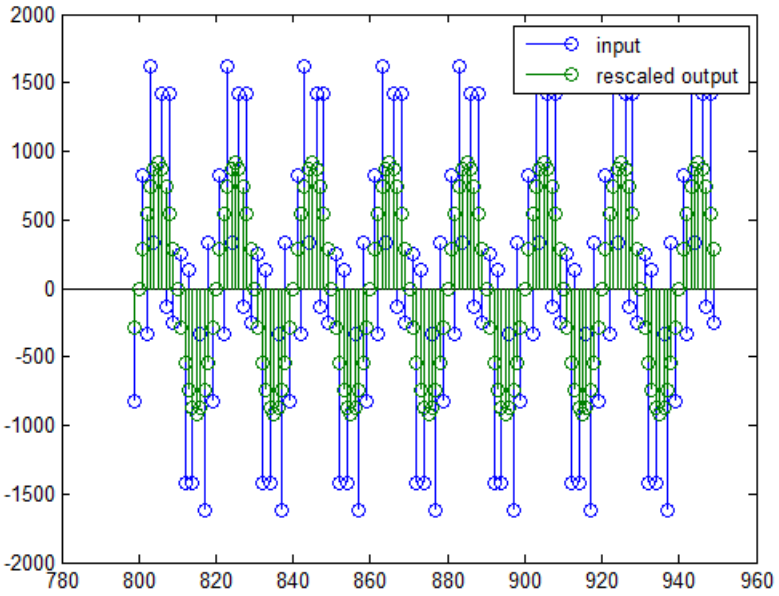
```
[H,w] = freqz(h);
plot(w/pi, 20*log10(2^(-14)*abs(H)));
```



This plot shows that the passband gain is still above 0 dB.

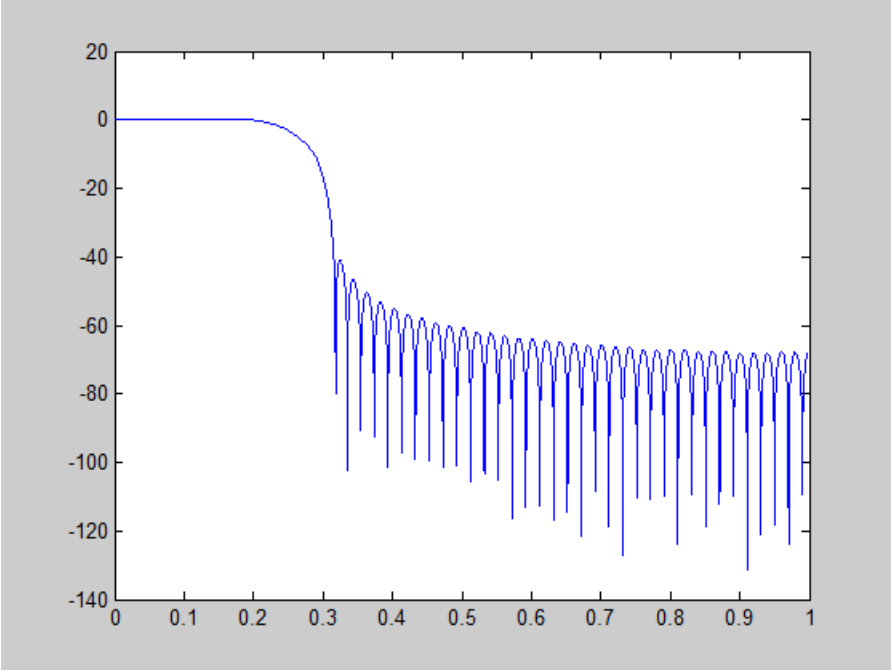
To put the input and output on the same scale, the MSBs must be weighted equally. The input MSB has a weight of 2^11, whereas the scaled output MSB has a weight of 2^(29–14) = 2^15. You need to give the output MSB a weight of 2^11 as follows:

```
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.

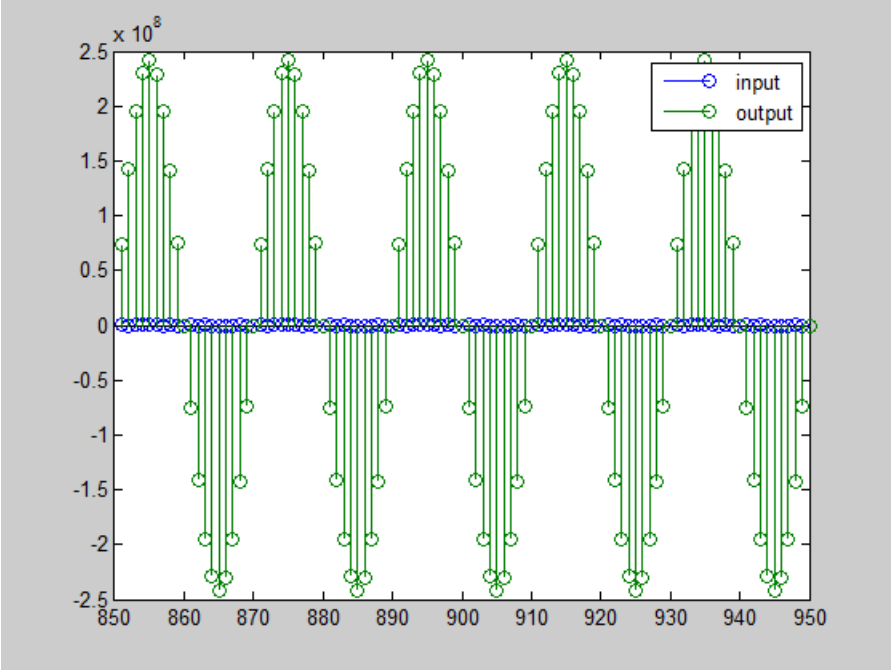
Reinterpret the Output

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
```

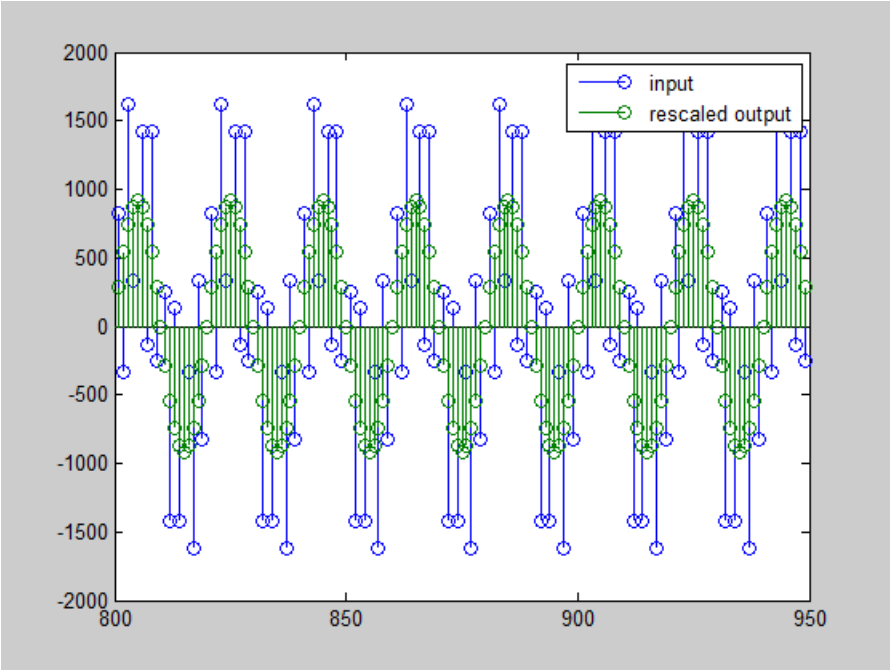
You can then truncate the output to only 16 bits:

```
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 19-62
“Fixed-Point Precision Rules” on page 19-64
“Polyphase Interpolators and Decimators” on page 19-65

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 $[Y_{min}, Y_{max}]$.

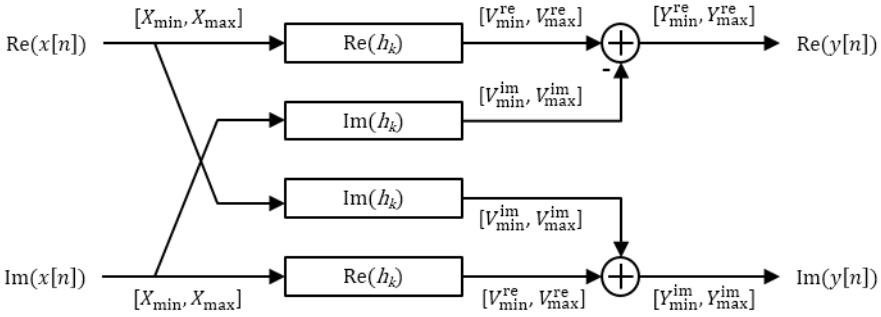
Complex Filter Convolution Equations

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:

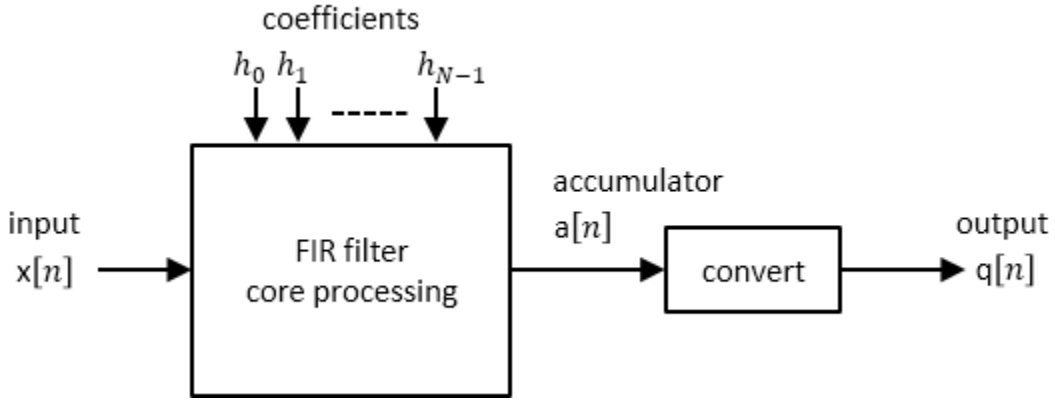
$$Y_{\min}^{re} = V_{\min}^{re} - V_{\max}^{im}$$
$$Y_{\max}^{re} = V_{\max}^{re} - V_{\min}^{im}$$
$$Y_{\min}^{im} = V_{\min}^{re} + V_{\min}^{im}$$
$$Y_{\max}^{im} = V_{\max}^{re} + V_{\max}^{im}$$

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 19-4
- “System Objects Supported by Fixed-Point Converter App” on page 19-20
- “Floating-Point to Fixed-Point Conversion of IIR Filters” on page 4-328

C Code Generation

Learn how to generate code for signal processing applications.

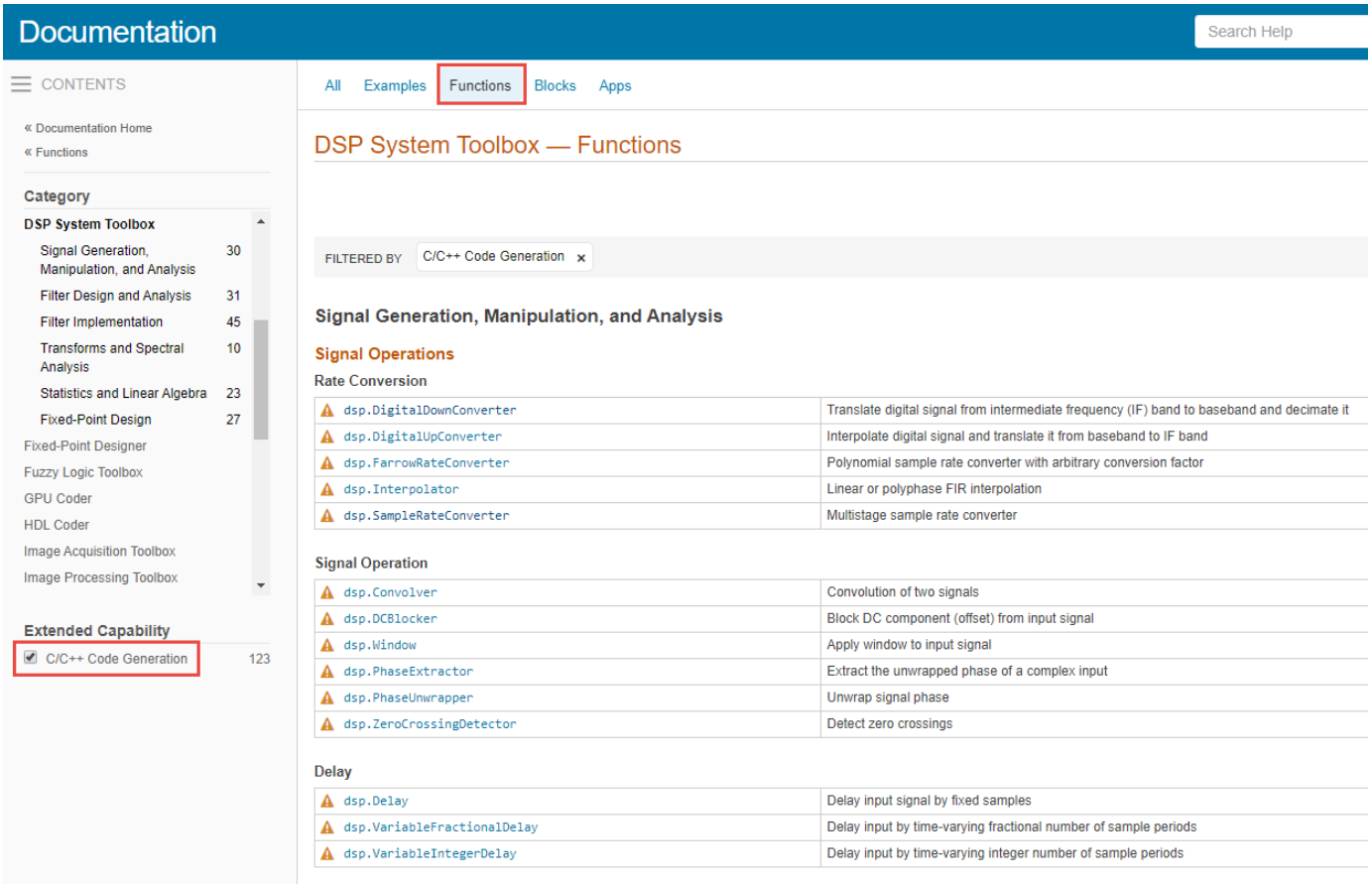
- “Functions and System Objects in DSP System Toolbox that Support C Code Generation” on page 20-2
- “Simulink Blocks in DSP System Toolbox that Support C Code Generation” on page 20-4
- “Understanding C Code Generation in DSP System Toolbox” on page 20-6
- “Generate C Code from MATLAB Code” on page 20-10
- “Relocate Code Generated from MATLAB Code to Another Development Environment” on page 20-17
- “Generate C Code from Simulink Model” on page 20-19
- “Relocate Code Generated from a Simulink Model to Another Development Environment” on page 20-24
- “How To Run a Generated Executable Outside MATLAB” on page 20-27
- “Use Generated Code to Accelerate an Application Deployed with MATLAB Compiler” on page 20-30
- “How Is dspunfold Different from parfor?” on page 20-41
- “Workflow for Generating a Multithreaded MEX File using dspunfold” on page 20-43
- “Why Does the Analyzer Choose the Wrong State Length?” on page 20-47
- “Why Does the Analyzer Choose a Zero State Length?” on page 20-49
- “Array Plot with Android Devices” on page 20-50
- “System objects in DSP System Toolbox that Support SIMD Code Generation” on page 20-55
- “Generate High Performance SIMD Code on Intel from MATLAB Algorithms in DSP System Toolbox” on page 20-57
- “Simulink Blocks in DSP System Toolbox that Support SIMD Code Generation” on page 20-59
- “Generate High Performance SIMD Code on Intel from Simulink Blocks in DSP System Toolbox” on page 20-64
- “In-Place Memory Optimization” on page 20-67

Functions and System Objects in DSP System Toolbox that Support 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 an example, see “Generate C Code from MATLAB Code” on page 20-10. For usage rules and limitations while generating code from System objects, see “System Objects in MATLAB Code Generation” (MATLAB Coder).

You can view functions and System objects that are supported for C/C++ code generation in documentation by filtering the functions reference list. Click **Functions** right below the blue bar at the top of the Help window, then select the **C/C++ Code Generation** check box at the bottom of the left column under **Extended Capability**. The functions and objects are listed in their respective categories. You can use the table of contents in the left column to navigate between the categories. Refer to the **Extended Capabilities > C/C++ Code Generation** section of each function page for any usage notes and limitations.

To obtain this filtered list for the DSP System Toolbox, click Functions that Support C/C++ Code Generation.



See Also

More About

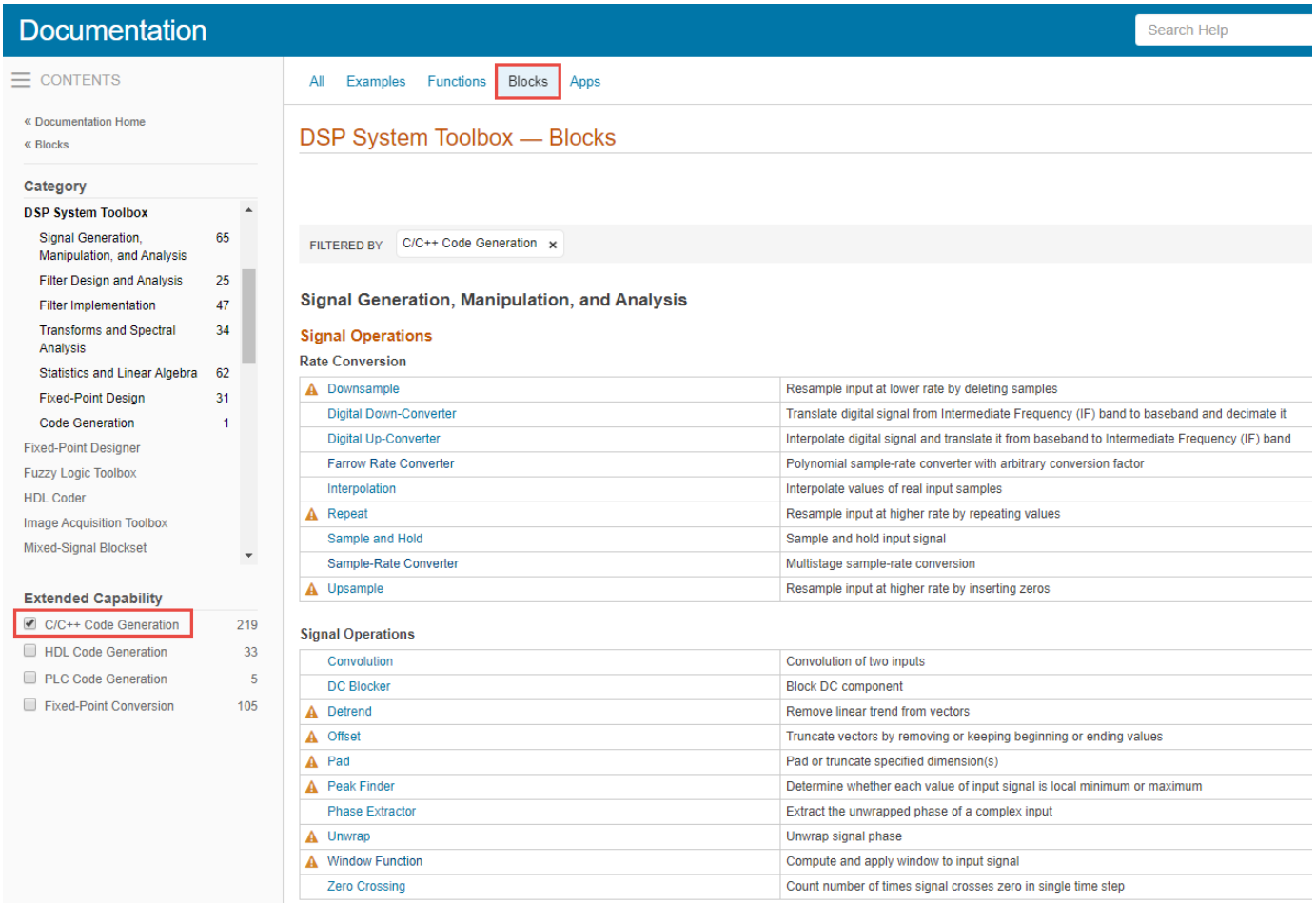
- “Simulink Blocks in DSP System Toolbox that Support C Code Generation” on page 20-4

Simulink Blocks in DSP System Toolbox that Support C Code Generation

If you have a Simulink Coder license, you can generate C and C++ code from certain Simulink blocks in DSP System Toolbox. For an example, see “Generate C Code from Simulink Model” on page 20-19.

You can view blocks that are supported for C/C++ code generation in documentation by filtering the blocks reference list. Click **Blocks** right below the blue bar at the top of the Help window, then select the **C/C++ Code Generation** check box at the bottom of the left column under **Extended Capability**. The blocks are listed in their respective categories. You can use the table of contents in the left column to navigate between the categories. Refer to the **Extended Capabilities > C/C++ Code Generation** section of each block page for any usage notes and limitations.

To obtain this filtered list for the DSP System Toolbox, click Blocks that Support C/C++ Code Generation.



See Also

More About

- “Functions and System Objects in DSP System Toolbox that Support C Code Generation” on page 20-2

Understanding C Code Generation in DSP System Toolbox

In this section...
“Generate C and C++ code from MATLAB code” on page 20-6
“Generate C and C++ Code from a Simulink Model” on page 20-6
“Shared Library Dependencies” on page 20-7
“Generate C Code for ARM Cortex-M and ARM Cortex-A Processors” on page 20-8
“Generate Code for Mobile Devices” on page 20-8

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 in DSP System Toolbox that Support C Code Generation” on page 20-2. 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 20-10. 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, see “Simulink Blocks in DSP System Toolbox that Support C Code Generation” on page 20-4. Alternatively, you can find this data in 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 20-19.

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 20-17 and “Relocate Code Generated from a Simulink Model to Another Development Environment” on page 20-24.

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 20-27.

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 20-50.

See Also

Functions

`codegen` | `getNonBuildFiles`

More About

- "Generate C Code from MATLAB Code" on page 20-10
- "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 20-19
- "Generate C Code for a Model" (Simulink Coder)
- "Relocate Code Generated from MATLAB Code to Another Development Environment" on page 20-17
- "Relocate Code Generated from a Simulink Model to Another Development Environment" on page 20-24

- “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 20-50

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”. 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
```

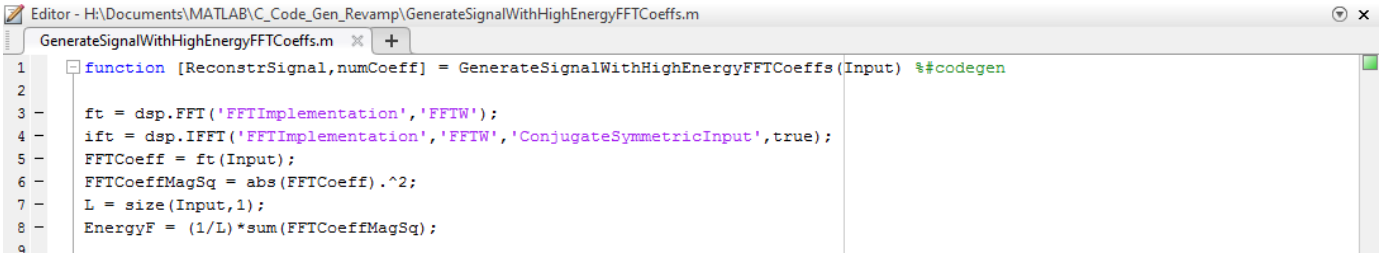
Make Code Suitable for Code Generation

Before you generate code, you must prepare your MATLAB code for code generation.

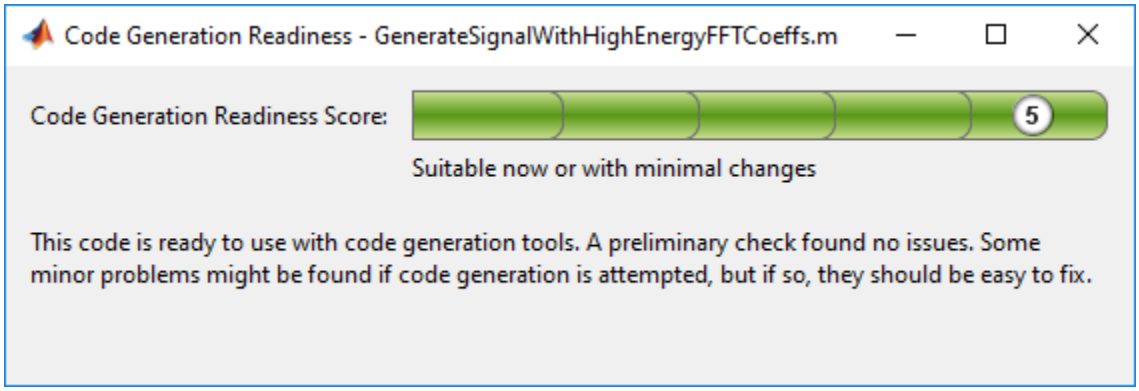
Check Issues at Design Time

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 1-53.

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.

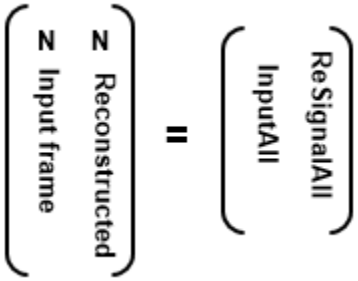
Generate a Standalone Executable

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 “RTW.BuildInfo 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 20-17.

See Also

Functions

`codegen`

More About

- “Relocate Code Generated from MATLAB Code to Another Development Environment” on page 20-17
- “Generate C Code from Simulink Model” on page 20-19
- “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 20-10 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 "RTW.BuildInfo 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 20-10 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 20-10 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 20-27.

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 20-10
- “Relocate Code Generated from a Simulink Model to Another Development Environment” on page 20-24
- “RTW.BuildInfo Methods” (MATLAB Coder)
- “Package Code for Other Development Environments” (MATLAB Coder)

Generate C Code from Simulink Model

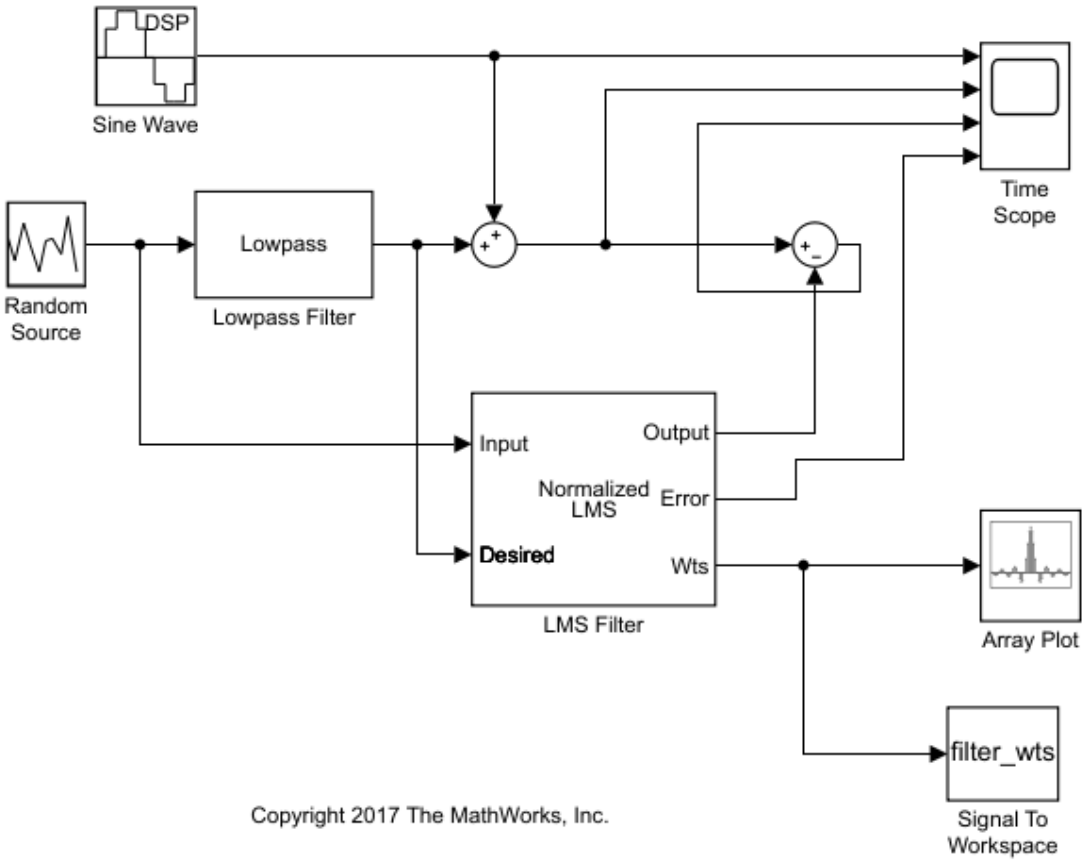
Simulink Coder generates standalone C and C++ code from Simulink models for deployment in a wide variety of applications. For a list of DSP System Toolbox features supported by Simulink Coder, see Blocks Supported for C Code Generation.

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')
```



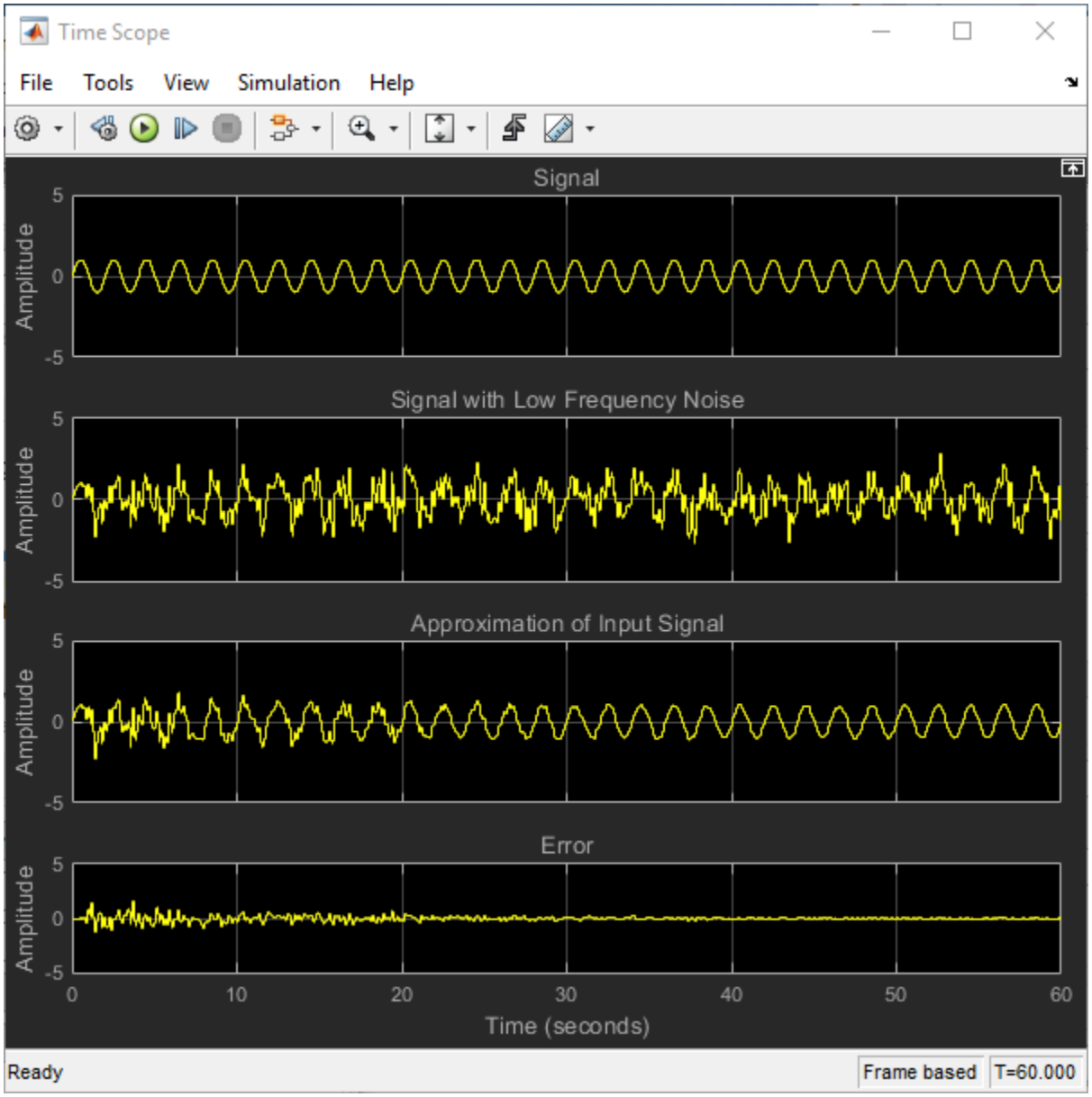
You can alternatively create the model using the **DSP System** template. For more information, see “Configure the Simulink Environment for Signal Processing Models”.

Configure Model for Code Generation

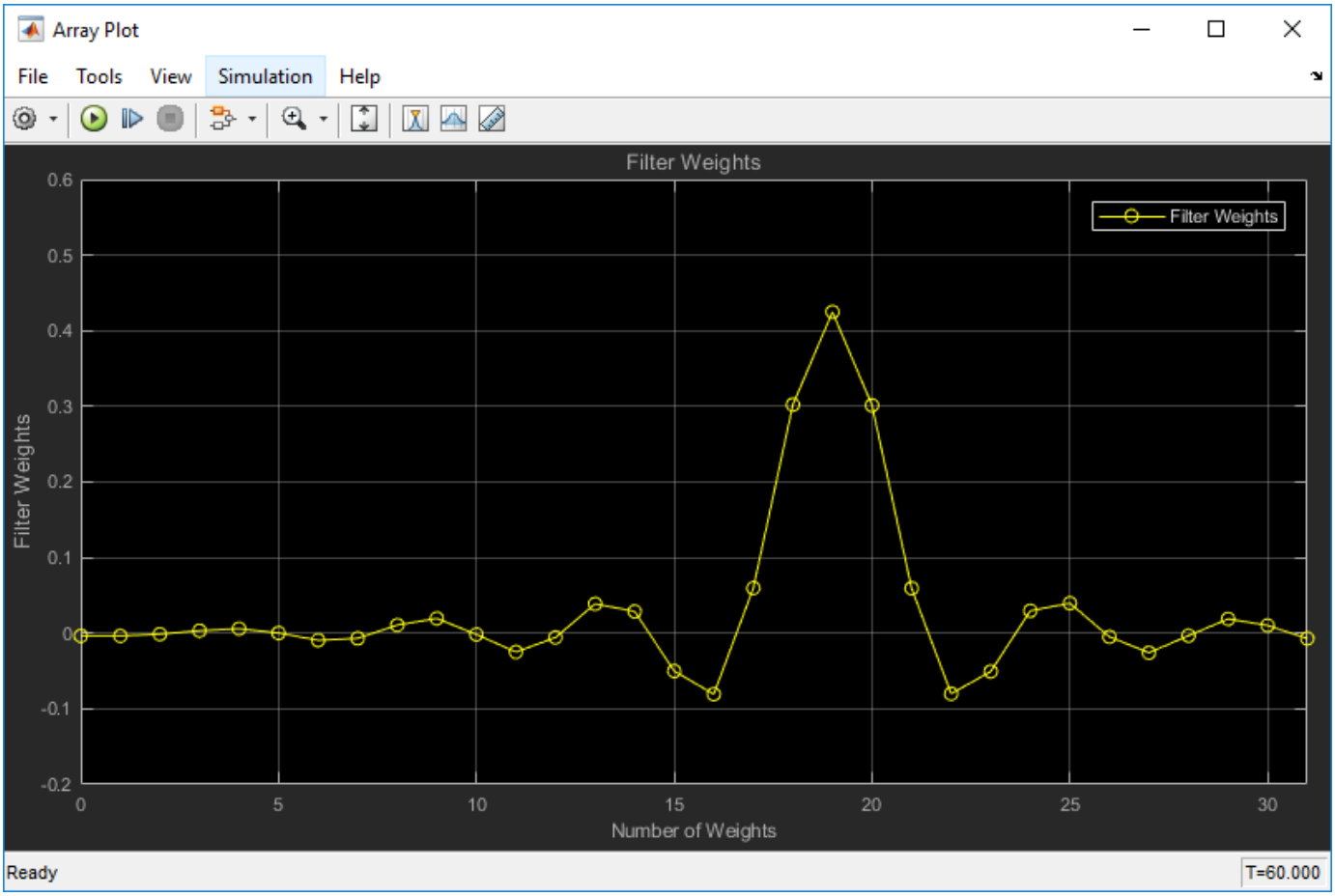
Prepare the model for code generation by specifying code generation settings in the **Configuration Parameters** dialog box. Choose the appropriate solver and code generation target, and check the model configuration for execution efficiency. For more details on each of these steps, see “Generate C Code 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 20-24.

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 20-24
- “Relocate Code to Another Development Environment” (Simulink Coder)
- “Generate C Code from MATLAB Code” on page 20-10
- “How To Run a Generated Executable Outside MATLAB” on page 20-27

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 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 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

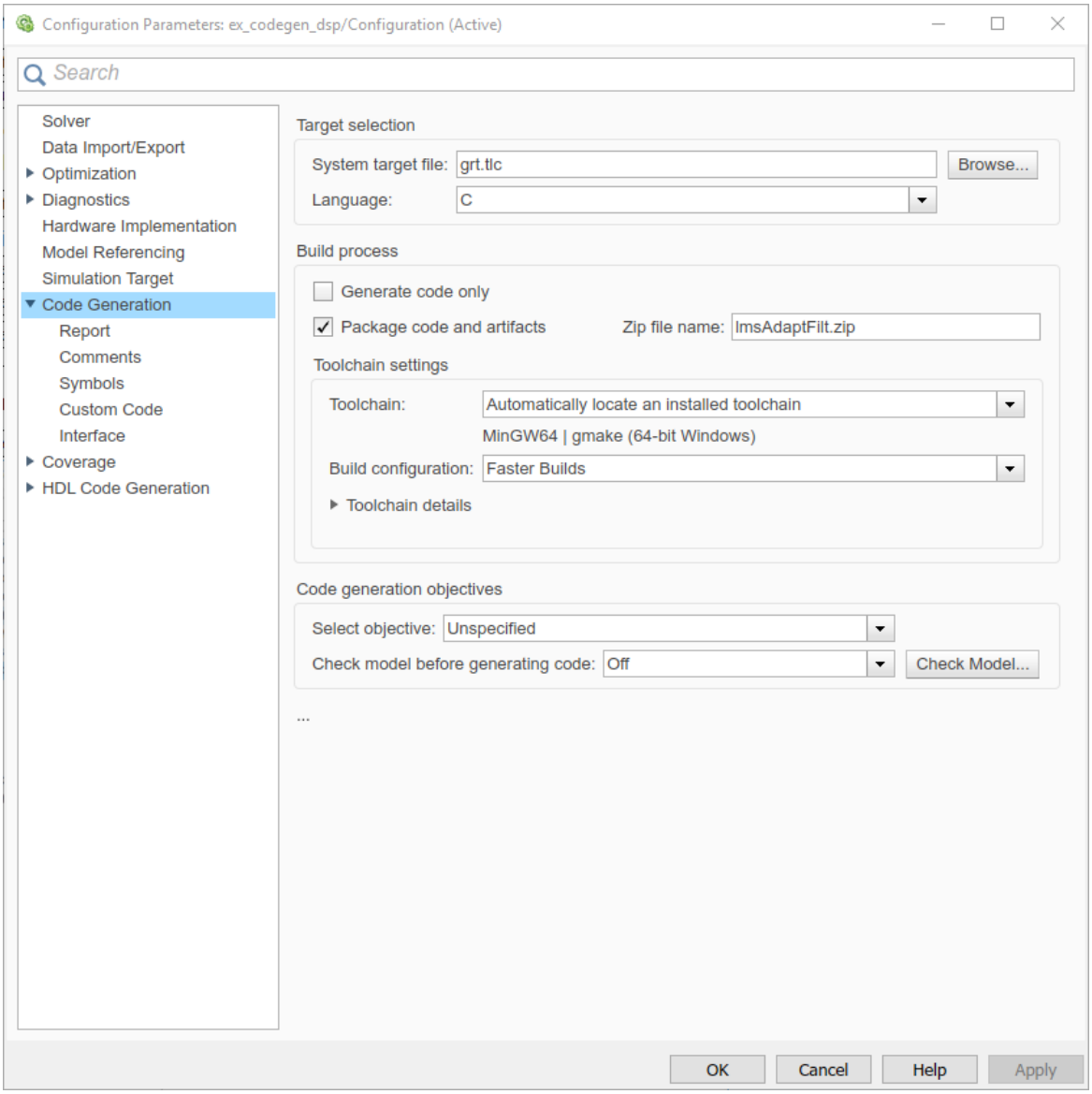
This example shows how to package the executable generated from the `ex_codegen_dsp` model in the “Generate C Code from Simulink Model” on page 20-19 example using the user interface. You can also generate and package a static library file or a dynamic library file.

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 20-27, 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 20-19
- “Relocate Code to Another Development Environment” (Simulink Coder)
- “Relocate Code Generated from MATLAB Code to Another Development Environment” on page 20-17

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 20-17. 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 20-24.

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 20-6
- “MATLAB Programming for Code Generation” (MATLAB Coder)
- “Relocate Code Generated from MATLAB Code to Another Development Environment” on page 20-17
- “Relocate Code Generated from a Simulink Model to Another Development Environment” on page 20-24

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. The algorithm 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 of the simulation on scopes.

For details about this application, see “System Identification Using RLS Adaptive Filtering” on page 4-264 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

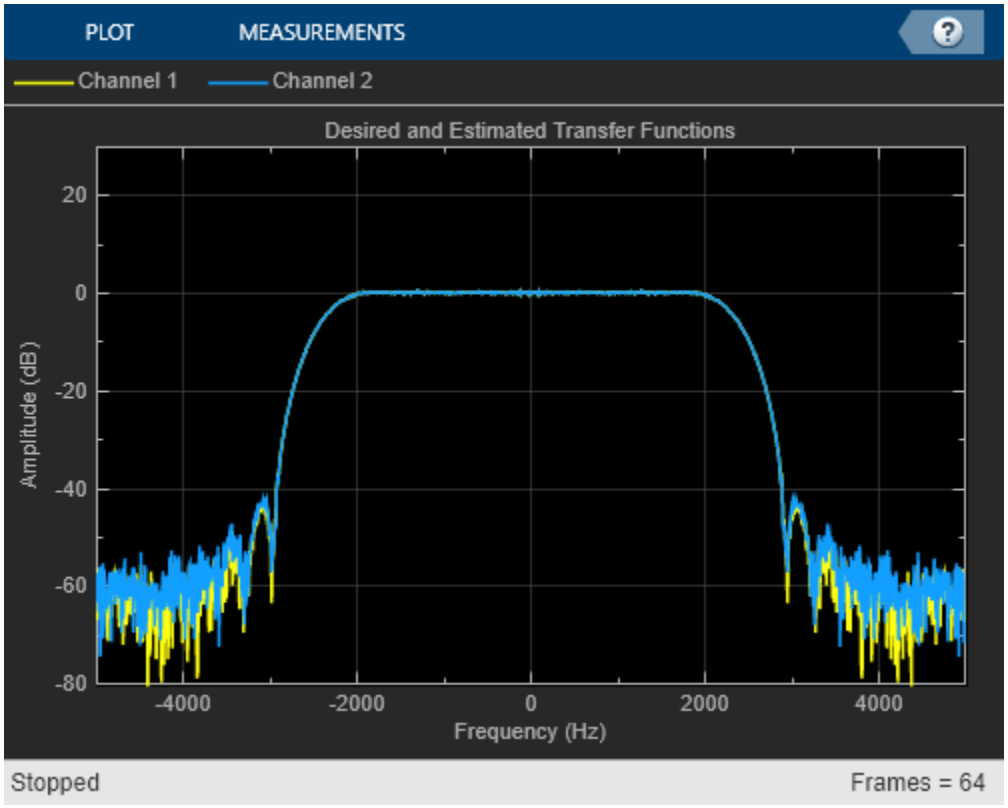
```
function scopeHandles = myRLSFilterSystemIDApp(numTSteps)
% myRLSFilterSystemIDApp initialize and execute RLS Filter
% system identification example. Then, display results using
% scopes. The function returns the handles to the scope and UI objects.
%
% Input:
%   numTSteps - number of time steps
% 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 = timescope('SampleRate',1e4,...
    'Position',[8 184 520 420],...
    'TimeSpanSource','property','TimeSpan',0.01,...
    '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
```

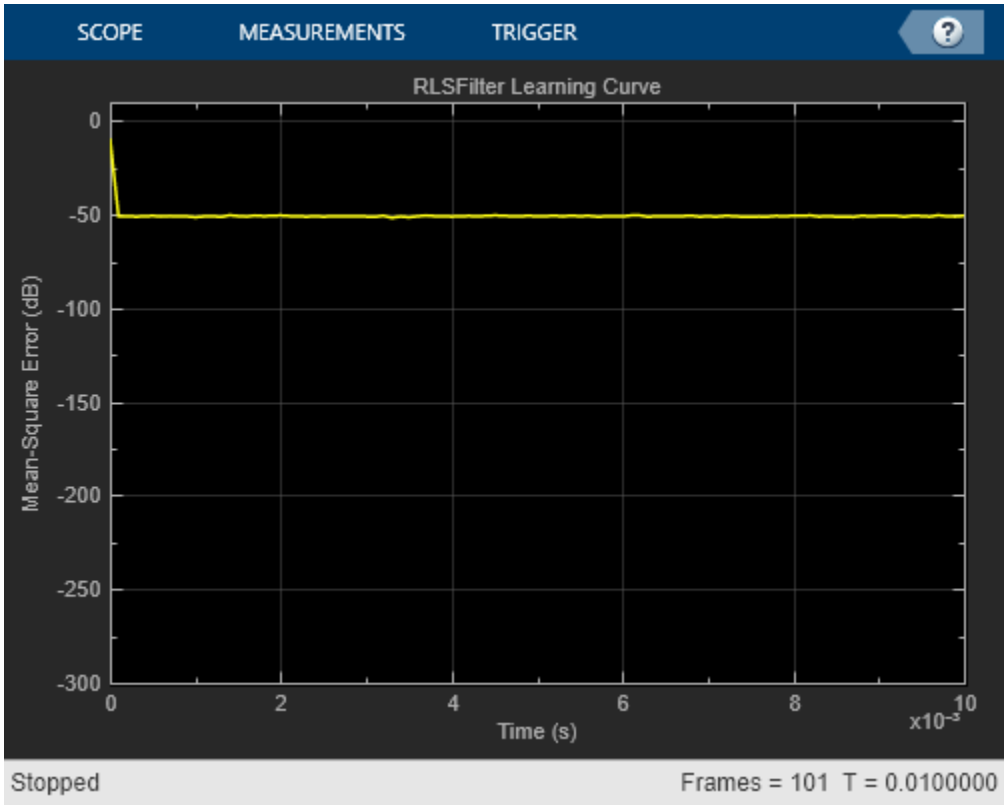
```
%  forgetting factor
Fs = 1e4;
param = struct([]);
param(1).Name = 'Cutoff Frequency (Hz)';
param(1).InitialValue = 0.48 * Fs/2;
param(1).Limits = Fs/2 * [1e-5, .9999];
param(2).Name = 'RLS Forgetting Factor';
param(2).InitialValue = 0.99;
param(2).Limits = [.3, 1];
hUI = HelperCreateParamTuningUI(param, '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);
```


PLOT
MEASUREMENTS
Channel 1
Channel 2
Desired and Estimated Transfer Functions
Amplitude (dB)
20
0
-20
-40
-60
-80
-4000
-2000
0
2000
4000
Frequency (Hz)
Stopped
Frames = 64


SCOPE
MEASUREMENTS
TRIGGER
RLSFilter Learning Curve
Mean-Square Error (dB)
0
-50
-100
-150
-200
-250
-300
0
2
4
6
8
10
x10⁻³
Time (s)
Stopped
Frames = 101 T = 0.0100000

Prepare Algorithm for Acceleration

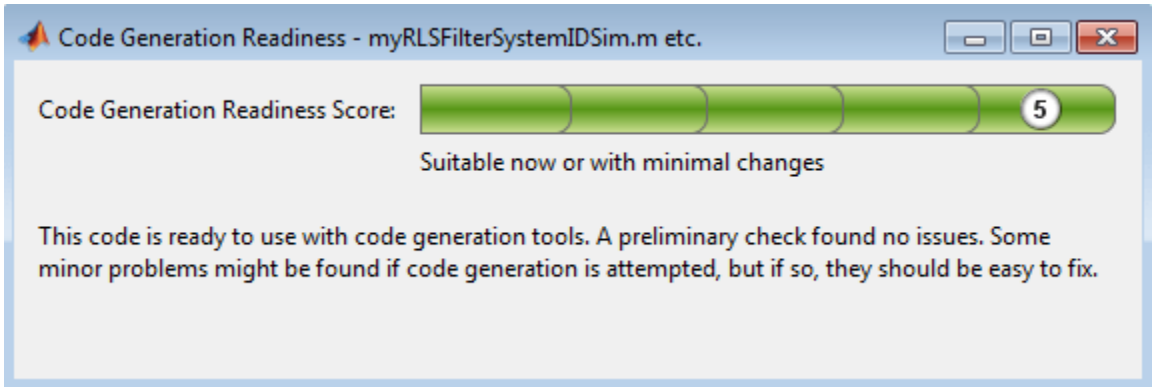
When you use MATLAB Coder to accelerate a MATLAB algorithm, the code must be suitable for code generation.

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};
```

```
Code generation successful.
```

`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: 2.551 seconds
Time for MEX function: 0.38898 seconds
The MEX function is 6.5582 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.

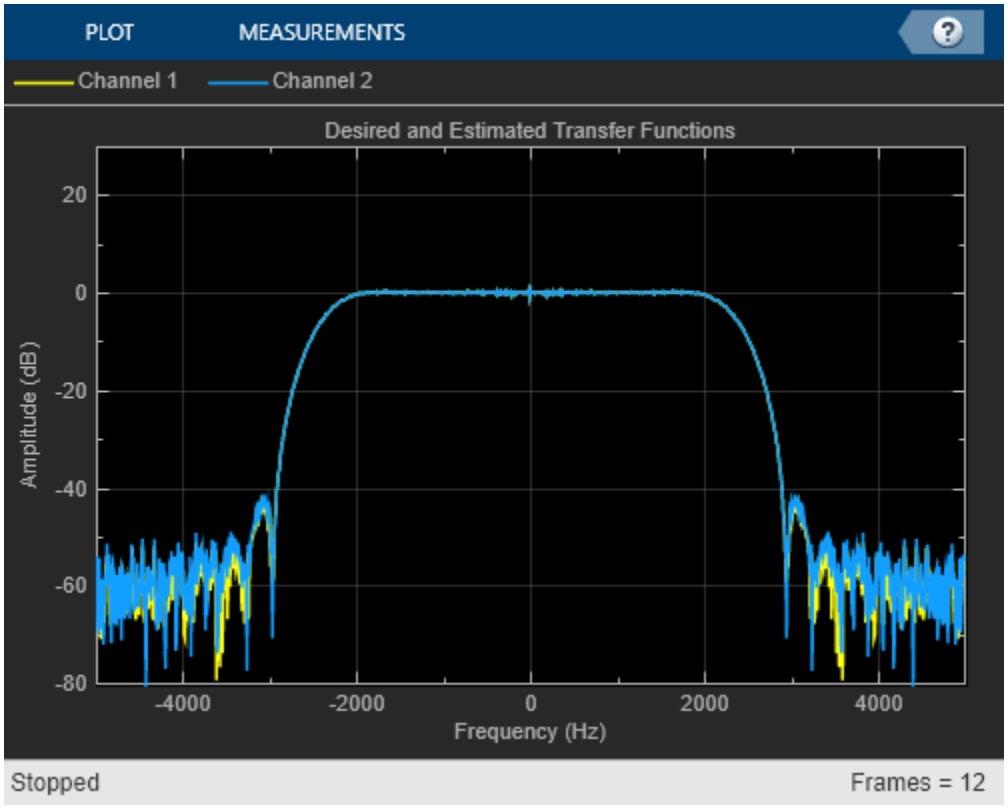
Modify the Application to Call the MEX Function

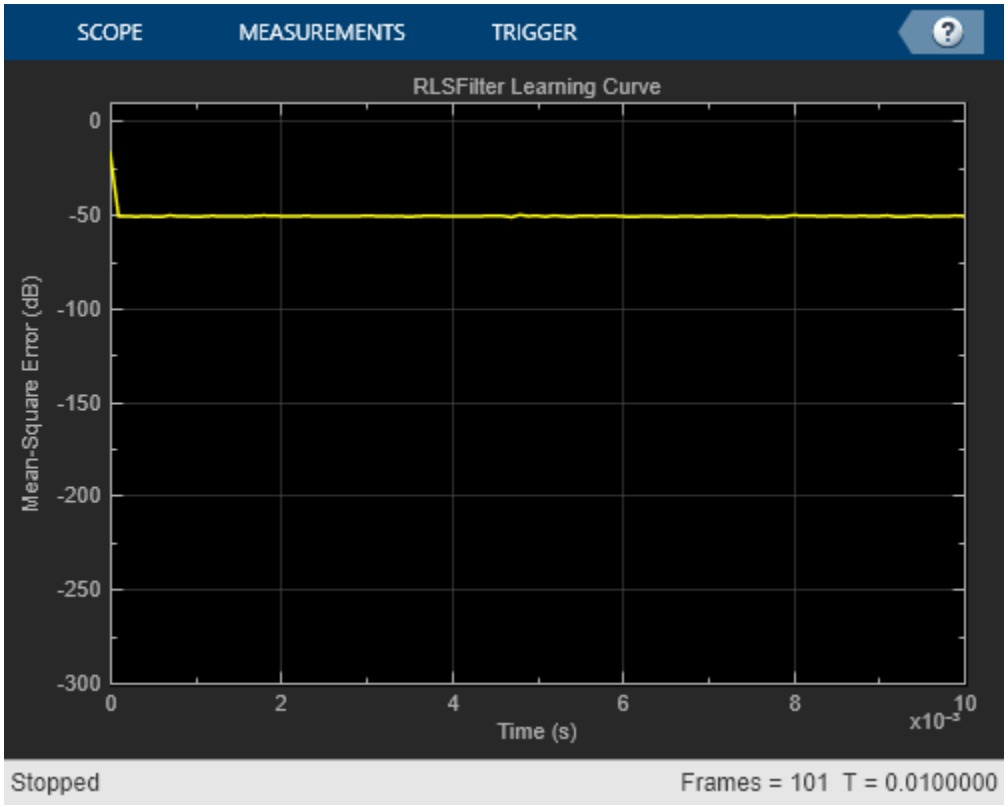
Modify `myRLSFilterSystemIDApp` so that it calls `myRLSFilterSystemIDSim_mex` instead of `myRLSFilterSystemIDSim`.

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);
```


PLOT
MEASUREMENTS
Channel 1
Channel 2
Desired and Estimated Transfer Functions
Amplitude (dB)
20
0
-20
-40
-60
-80
-4000
-2000
0
2000
4000
Frequency (Hz)
Stopped
Frames = 12


SCOPE
MEASUREMENTS
TRIGGER
RLSFilter Learning Curve
Mean-Square Error (dB)
0
-50
-100
-150
-200
-250
-300
0
2
4
6
8
10
x10⁻³
Time (s)
Stopped
Frames = 101 T = 0.0100000

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

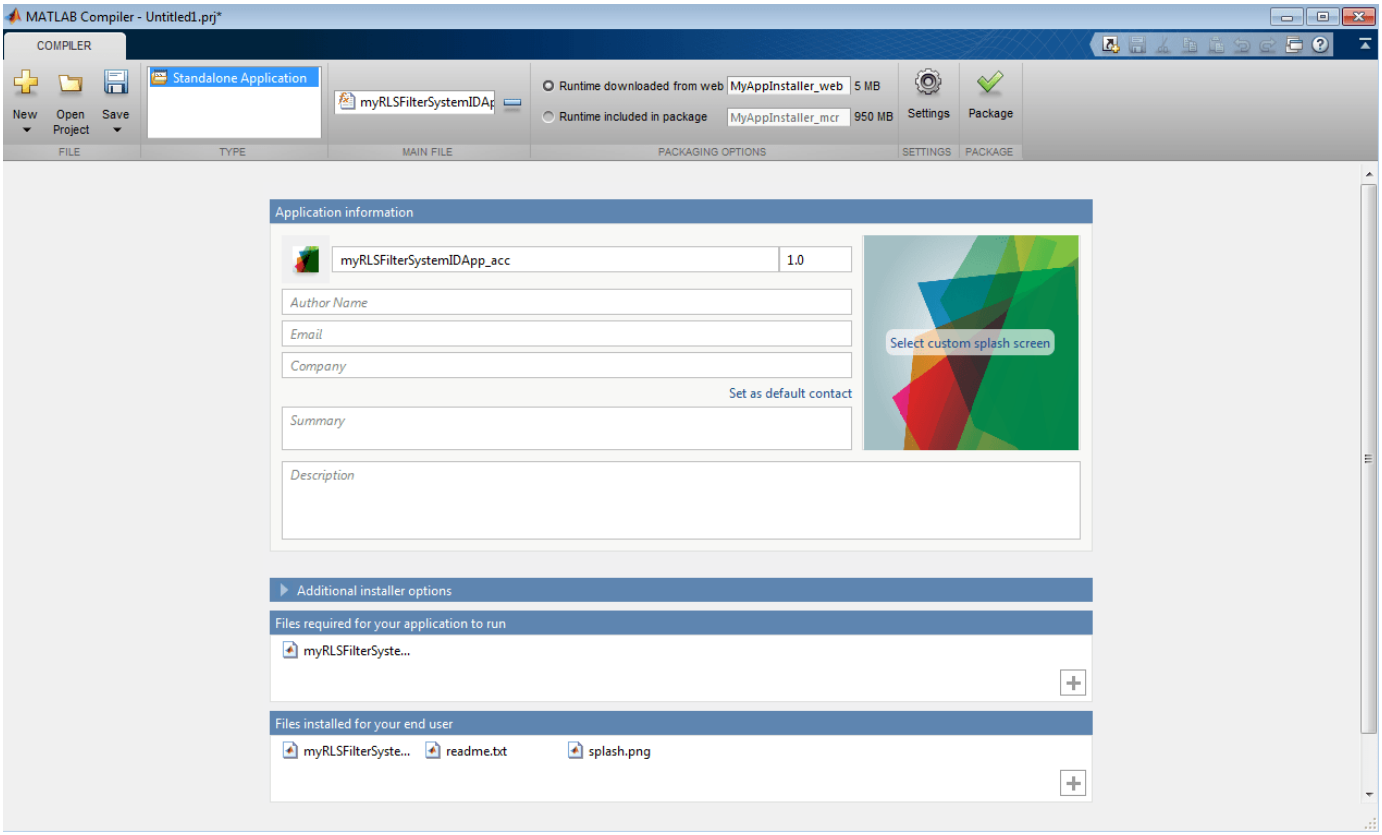
1. To open the Application Compiler App, on the **Apps** tab, under **Application Deployment**, click the app icon.

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.

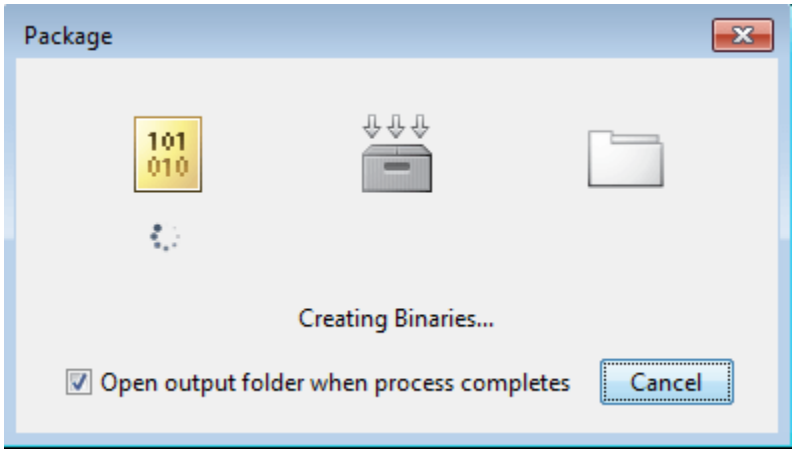
3. In the **Packaging Options** section of the toolstrip, make sure that the **Runtime downloaded from web** check box is selected.

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” on page 4-264
- “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 20-41
“dspunfold Introduces Latency” on page 20-41
“parfor Requires Significant Restructuring in Code” on page 20-41
“parfor Used with dspunfold” on page 20-41

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 20-47
- “Why Does the Analyzer Choose a Zero State Length?” on page 20-49

Why Does the Analyzer Choose the Wrong State Length?

In this section...
“Reason for Verification Failure” on page 20-48 “Recommendation” on page 20-48

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 20-43
- “Why Does the Analyzer Choose a Zero State Length?” on page 20-49

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 20-43
- “Why Does the Analyzer Choose the Wrong State Length?” on page 20-47

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” on page 18-65.
- **androidarrayplot**: Showcases how the power spectrum estimate is displayed on the 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

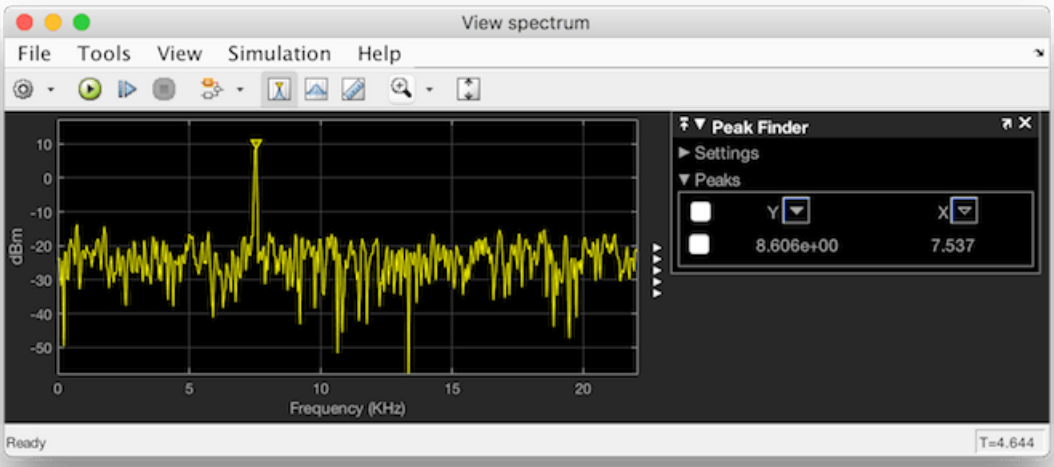
Task 1 - Display Signals on the Development Computer

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.

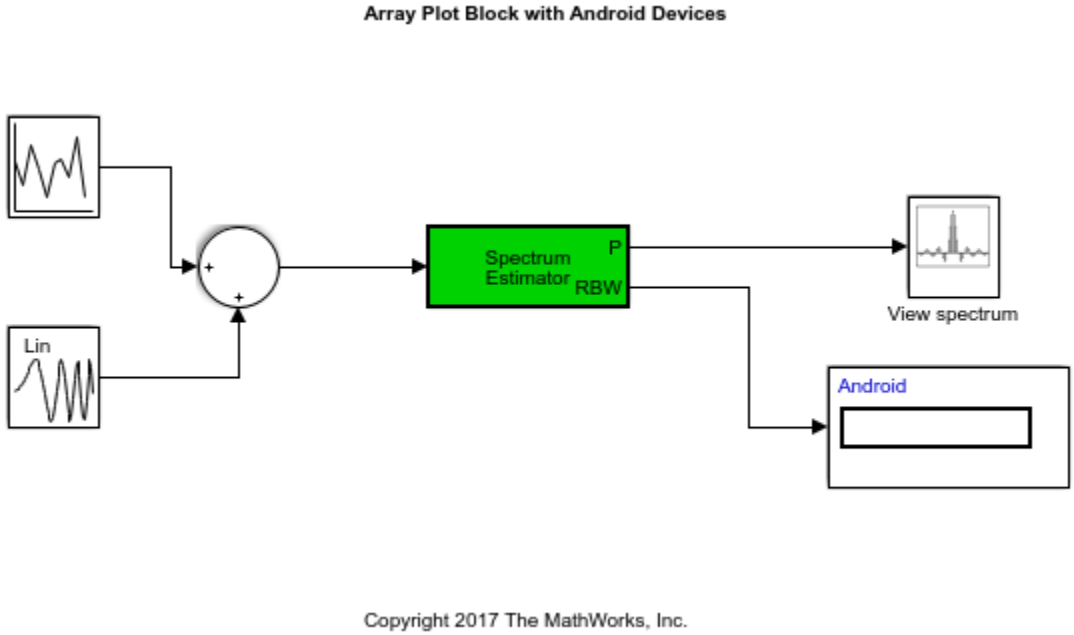
3. In the Simulink model, select the **Simulation** tab and click **Run** to see the output of the streaming power spectrum estimate.



Task 2 - Display Signals on Your Android Device

Display the power spectrum estimate on your Android device

1. Open the androidarrayplot model.



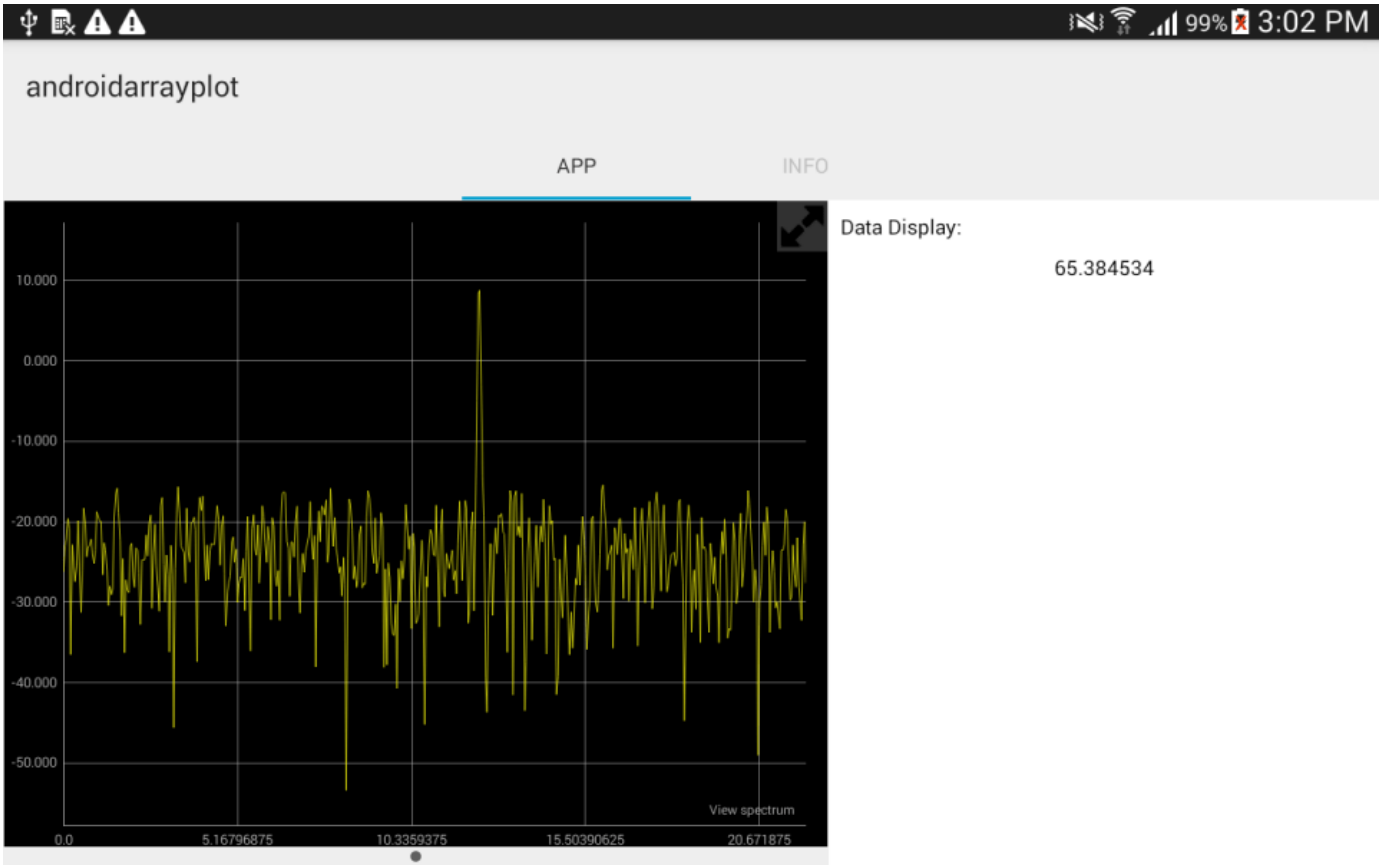
2. Note how the Simulink **Display** block has been replaced with a **Data Display** block from the Android Support Package. This allows you to view the resolution bandwidth (RBW) on your Android Device.

3. On 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. On the **Hardware** tab of the Simulink model, in the **Mode** section, select **Run on board** and then click **Build, Deploy & Start** to run the Simulink 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.

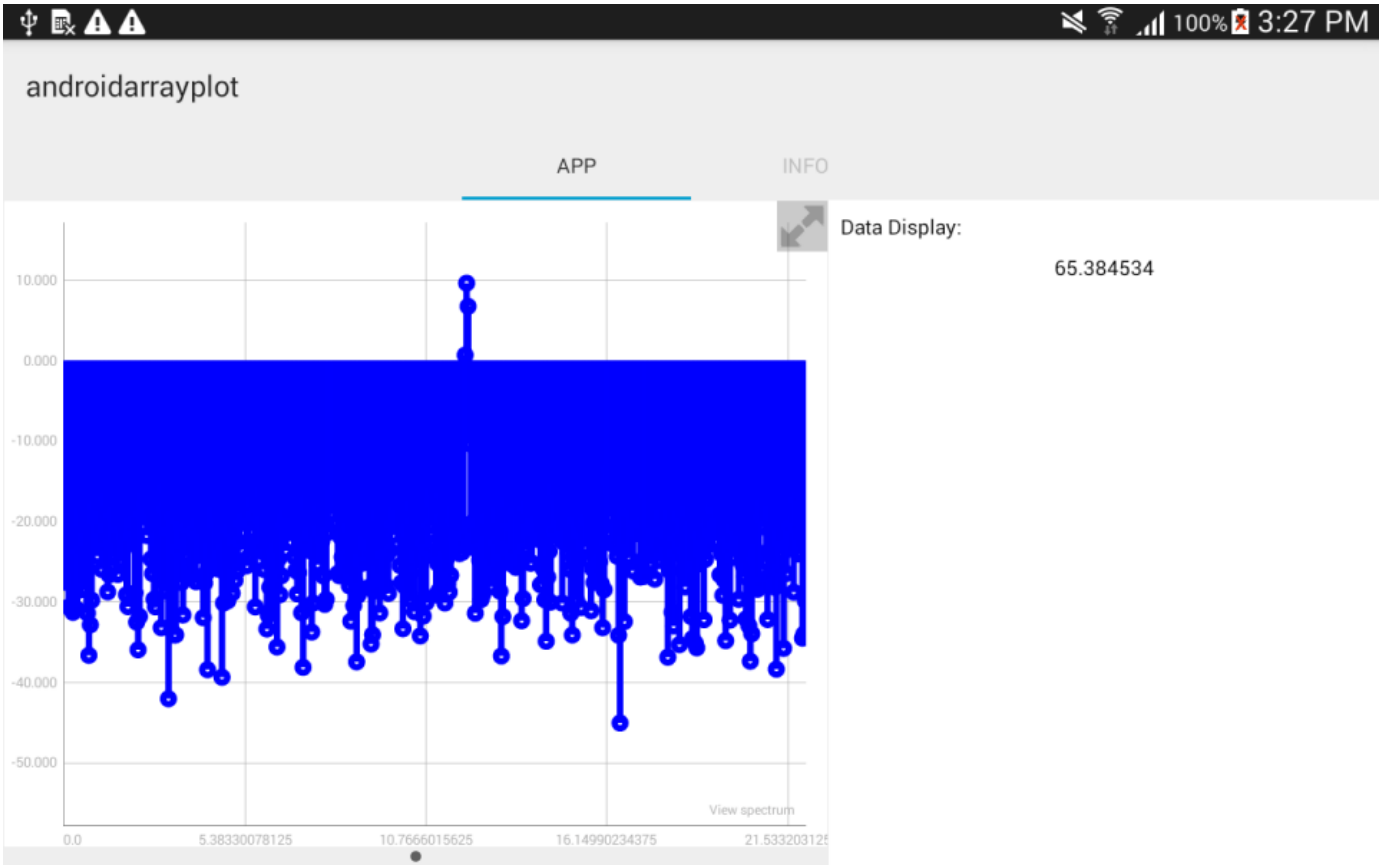
6. Set **Plot type** to **Stem**.

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 **Build, Deploy & Start** button of the Simulink model.



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
Spectrum Estimator | Array Plot | Spectrum Analyzer | Random Source | Chirp

Related Examples

- “Streaming Power Spectrum Estimation Using Welch's Method” on page 18-65

System objects in DSP System Toolbox that Support SIMD Code Generation

When certain conditions are met, you can generate SIMD code using Intel AVX2 technology from certain MATLAB System objects in DSP System Toolbox. For information on how to generate SIMD code from MATLAB algorithms, see “Generate High Performance SIMD Code on Intel from MATLAB Algorithms in DSP System Toolbox” on page 20-57.

The following table contains a list of System objects in DSP System Toolbox that support SIMD code generation. The table also details the conditions under which the support is available.

MATLAB System objects	Conditions
`dsp.AnalyticSignal`	• Input signal is real-valued. • Input signal has a data type of `single` or `double`.
`dsp.ComplexBandpassDecimator`	• Input signal is complex-valued. • Input signal has a data type of `single` or `double`.
`dsp.DCBlocker`	• Input signal has a data type of `single` or `double`.
`dsp.Differentiator`	• Input signal has a data type of `single` or `double`.
`dsp.DigitalDownConverter`	• Input signal has a data type of `single` or `double`.
`dsp.DigitalUpConverter`	• Input signal has a data type of `single` or `double`.
`dsp.FIRFilter`	• Filter structure is set to `'Direct form'` or `'Direct form transposed'`. • Input signal is real-valued with real filter coefficients. • When the filter structure is set to `'Direct form'`, the input signal can also be complex-valued with real or complex filter coefficients. • Input signal has a data type of `single` or `double`.
`dsp.FIRDecimator`	• Filter structure is set to `'Direct form'`. • Input signal is real-valued with real filter coefficients. • Input signal is complex-valued with real or complex filter coefficients. • Input signal has a data type of `single` or `double`.
`dsp.FIRHalfbandInterpolator`	• Input signal has a data type of `single` or `double`.

MATLAB System objects	Conditions
`dsp.FIRInterpolator`	• Input signal is real-valued with real filter coefficients. • Input signal is complex-valued with real or complex filter coefficients. • Input signal has a data type of `single` or `double`.
`dsp.HighpassFilter`	• `FilterType` is set to `'FIR'`. • Input signal has a data type of `single` or `double`.
`dsp.LMSFilter`	• `Method` is set to `'LMS'` or `'Normalized LMS'`. • `WeightsOutput` is set to `'None'` or `'Last'`. • Input signal is real-valued. • Input signal has a data type of `single` or `double`.
`dsp.LowpassFilter`	• `FilterType` is set to `'FIR'`. • Input signal has a data type of `single` or `double`.
`dsp.SampleRateConverter`	• For upsampling, the ratio of output sample rate to input sample rate must be an integer. • For downsampling, the ratio of input sample rate to output sample rate must be an integer. • Input signal has a data type of `single` or `double`.
`dsp.VariableBandwidthFIRFilter`	• Input signal has a data type of `single` or `double`.

See Also

More About

- “Simulink Blocks in DSP System Toolbox that Support SIMD Code Generation” on page 20-59
- “Generate High Performance SIMD Code on Intel from MATLAB Algorithms in DSP System Toolbox” on page 20-57

Generate High Performance SIMD Code on Intel from MATLAB Algorithms in DSP System Toolbox

To generate SIMD code from MATLAB System objects, create a `coder.config` object, set the `CodeReplacementLibrary` property to `'DSP Intel AVX2-FMA (Windows)'`, `'DSP Intel AVX2-FMA (Linux)'`, or `'DSP Intel AVX2-FMA (Mac)'`, and use the object with the `codegen` command.

This workflow requires you to install MATLAB Coder and Embedded Coder on your machine.

Consider this MATLAB function that filters a random multichannel signal using the `dsp.FIRFilter` System object.

```
function y = firsingle()

persistent fir
if isempty(fir)
    b = fir1(250,.4);
    fir = dsp.FIRFilter(b);
end

frameSize = 512;
numChannels = 8;
numFrames = 1e3;

for k = 1:numFrames
    x = randn(frameSize,numChannels,'single');
    y = fir(x);
end
```

Generate plain C code executable of this function using the `codegen` command.

```
cfg = coder.config('exe');
% provides starter main.c
cfg.GenerateExampleMain = 'GenerateCodeAndCompile';
switch computer('arch')
case 'glnxa64'
    codegen firsingle -config cfg -report -o firsingle_std
case 'win64'
    codegen firsingle -config cfg -report -o firsingle_std.exe
case 'maci64'
    codegen firsingle -config cfg -report -o firsingle_std
end
```

Measure the time it takes to run the generated executable.

```
tic;
system('firsingle_std');
tplain = toc

tplain =

   1.1700
```

Generate AVX2 C code executable by setting the `CodeReplacementLibrary` parameter to either `'DSP Intel AVX2-FMA (Windows)'`, `'DSP Intel AVX2-FMA (Linux)'`, or `'DSP Intel AVX2-FMA (Mac)'`, and calling the `codegen` command on the `coder.config` object.

```
cfg = coder.config('exe');
switch computer('arch')
case 'glnxa64'
    cfg.CodeReplacementLibrary = 'DSP Intel AVX2-FMA (Linux)';
case 'win64'
    cfg.CodeReplacementLibrary = 'DSP Intel AVX2-FMA (Windows)';
case 'maci64'
    cfg.CodeReplacementLibrary = 'DSP Intel AVX2-FMA (Mac)';
end
cfg.GenerateExampleMain = 'GenerateCodeAndCompile'; % provides starter main.c
switch computer('arch')
case 'glnxa64'
    codegen firsingle -config cfg -report -o firsingle_avx2
case 'win64'
    codegen firsingle -config cfg -report -o firsingle_avx2.exe
case 'maci64'
    codegen firsingle -config cfg -report -o firsingle_avx2
end
```

Measure the time it takes to run the generated executable.

```
tic;
system('firsingle_avx2');
tavx2 = toc

tavx2 =

   0.2611
```

The generated SIMD code is around 4.5x faster compared to the plain C code on a Windows 10 machine.

```
tplain/tavx2

 ans =
4.4815
```

You can also generate a static library and a dynamic library by specifying the build type as `'lib'` and `'dll'`, respectively.

```
cfg = coder.config('lib');
cfg.CodeReplacementLibrary = 'DSP Intel AVX2-FMA (Windows)';
codegen MATLABfunctionName -config cfg

cfg = coder.config('dll');
cfg.CodeReplacementLibrary = 'DSP Intel AVX2-FMA (Windows)';
codegen MATLABfunctionName -config cfg
```

`MATLABfunctionName` is the MATLAB function that calls the System object you are trying to generate SIMD code from. For a list of System objects that support SIMD code generation, see “System objects in DSP System Toolbox that Support SIMD Code Generation” on page 20-55.

See Also

More About

- “System objects in DSP System Toolbox that Support SIMD Code Generation” on page 20-55
- “Generate High Performance SIMD Code on Intel from Simulink Blocks in DSP System Toolbox” on page 20-64

Simulink Blocks in DSP System Toolbox that Support SIMD Code Generation

When certain conditions are met, you can generate SIMD code using Intel AVX2 technology from certain Simulink blocks in DSP System Toolbox. For information on how to generate SIMD code from Simulink blocks, see “Generate High Performance SIMD Code on Intel from Simulink Blocks in DSP System Toolbox” on page 20-64.

The following table contains a list of Simulink blocks in DSP System Toolbox that support SIMD code generation. The table also details the conditions under which the support is available.

Simulink blocks	Conditions
Arbitrary Response Filter	• **Filter type** is set to `Single-rate`, `Decimator`, or `Interpolator`. • For **Filter type** that is set to `Single-rate`, **Structure** is set to `Direct-form FIR` or `Direct-form FIR transposed`. • For **Filter type** that is set to `Decimator`, **Structure** is set to `Direct-form FIR polyphase decimator` and **Rate options** is set to `Enforce single-rate processing`. • For **Filter type** that is set to `Interpolator`, **Rate options** is set to `Enforce single-rate processing`. • **Input processing** is set to `Columns as channels (frame based)`. • Input signal has a data type of `single` or `double`.
Analytic Signal	• **Input processing** is set to `Columns as channels (frame based)`. • Input signal has to be real-valued. • Input signal has a data type of `single` or `double`.
Bandpass Filter	• **Impulse response** is set to `FIR`. • **Filter type** is set to `Single-rate`. • **Structure** is set to `Direct-form FIR` or `Direct-form FIR transposed`. • **Use basic elements to enable filter customization** parameter is not selected. • **Input processing** is set to `Columns as channels (frame based)`. • Input signal has a data type of `single` or `double`.

Simulink blocks	Conditions
Bandstop Filter	• **Impulse response** is set to `FIR`. • **Filter type** is set to `Single-rate`. • **Structure** is set to `Direct-form FIR` or `Direct-form FIR transposed`. • **Use basic elements to enable filter customization** parameter is not selected. • **Input processing** is set to `Columns as channels (frame based)`. • Input signal has a data type of `single` or `double`.
Complex Bandpass Decimator	• Input signal is complex-valued. • Input signal has a data type of `single` or `double`.
DC Blocker	• Input signal has a data type of `single` or `double`.
Differentiator Filter	• Input signal has a data type of `single` or `double`.
Digital Filter Design	• **Input processing** is set to `Columns as channels (frame based)`. • **Filter Structure** (in **Import Filter from Workspace** pane) is set to `Direct-Form FIR`. You can generate SIMD code even when the filter is a `Direct-Form FIR Transposed` filter. To create a `Direct-Form FIR Transposed` filter, select **Edit** > **Convert Structure**, and click **Direct-Form FIR Transposed**. • Input signal has a data type of `single` or `double`.
Discrete FIR Filter	• **Filter structure** is set to `Direct form` or `Direct form transposed`. • **Input processing** is set to `Columns as channels (frame based)`. • Input signal is real-valued with real filter coefficients. • When **Filter structure** is set to `Direct form`, the input signal can also be complex-valued with real or complex filter coefficients. • Input signal has a data type of `single` or `double`.

Simulink blocks	Conditions
FIR Decimation	• **Filter structure** is set to `Direct form`. • **Input processing** is set to `Columns as channels (frame based)`. • **Rate options** is set to `Enforce single-rate processing`. • Input signal is real-valued with real filter coefficients. • Input signal is complex-valued with real or complex filter coefficients. • Input signal has a data type of `single` or `double`.
FIR Halfband Interpolator	• Input signal has a data type of `single` or `double`.
FIR Interpolation	• **Input processing** is set to `Columns as channels (frame based)`. • **Rate options** is set to `Enforce single-rate processing`. • Input signal is real-valued with real filter coefficients. • Input signal is complex-valued with real or complex filter coefficients. • Input signal has a data type of `single` or `double`.
Highpass Filter	• **Filter type** is set to `FIR`. • Input signal has a data type of `single` or `double`.

Simulink blocks	Conditions

Hilbert Filter	• **Filter type** is set to `Single-rate`, `Decimator`, or `Interpolator`.
	• For **Filter type** that is set to `Single-rate`, **Structure** is set to `Direct-form FIR` or `Direct-form FIR transposed`.
	• For **Filter type** that is set to `Decimator`, **Structure** is set to `Direct-form FIR polyphase decimator` and **Rate options** is set to `Enforce single-rate processing`.
	• For **Filter type** that is set to `Interpolator`:
	  • **Interpolation Factor** cannot be equal to `1`.
	  • **Rate options** is set to `Enforce single-rate processing`.
	• **Input processing** is set to `Columns as channels (frame based)`.
	• Input signal has a data type of `single` or `double`.
	• Input port dimensions cannot be equal to [1 1].

Inverse Sinc Filter	• **Filter type** is set to `Single-rate`, `Decimator`, or `Interpolator`. • For **Filter type** that is set to `Single-rate`, **Structure** is set to `Direct-form FIR` or `Direct-form FIR transposed`. • For **Filter type** that is set to `Decimator`, **Structure** is set to `Direct-form FIR polyphase decimator` and **Rate options** is set to `Enforce single-rate processing`. • For **Filter type** that is set to `Interpolator`, **Rate options** is set to `Enforce single-rate processing`. • **Input processing** is set to `Columns as channels (frame based)`. • Input signal has a data type of `single` or `double`.
LMS Filter	• **Algorithm** parameter is set to `LMS` or `Normalized LMS`. • Input signal is real-valued. • Input signal has a data type of `single` or `double`.

Simulink blocks	Conditions
Lowpass Filter	• **Filter type** is set to `FIR`. • Input signal has a data type of `single` or `double`.

Nyquist Filter	• **Filter type** is set to `Single-rate`, `Decimator`, or `Interpolator`.
	• For **Filter type** that is set to `Single-rate`, **Structure** is set to `Direct-form FIR` or `Direct-form FIR transposed`.
	• For **Filter type** that is set to `Decimator`, **Structure** is set to `Direct-form FIR polyphase decimator` and **Rate options** is set to `Enforce single-rate processing`.
	• For **Filter type** that is set to `Interpolator`:
	  • **Interpolation Factor** cannot be equal to `1`.
	  • **Rate options** is set to `Enforce single-rate processing`.
	• **Input processing** is set to `Columns as channels (frame based)`.
	• Input signal has a data type of `single` or `double`.
	• Input port dimensions cannot be equal to [1 1].

Sample-Rate Converter	• For upsampling, the ratio of output sample rate to input sample rate must be an integer. • For downsampling, the ratio of input sample rate to output sample rate must be an integer. • Input signal has a data type of `single` or `double`.
Variable Bandwidth FIR Filter	• Input signal has a data type of `single` or `double`.

See Also

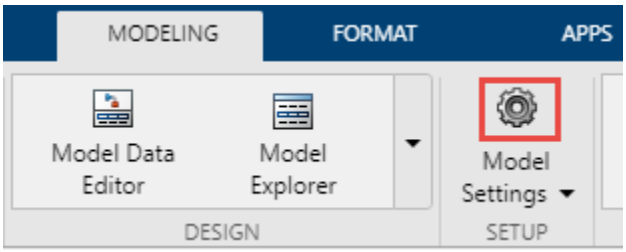
More About

- “System objects in DSP System Toolbox that Support SIMD Code Generation” on page 20-55
- “Generate High Performance SIMD Code on Intel from Simulink Blocks in DSP System Toolbox” on page 20-64

Generate High Performance SIMD Code on Intel from Simulink Blocks in DSP System Toolbox

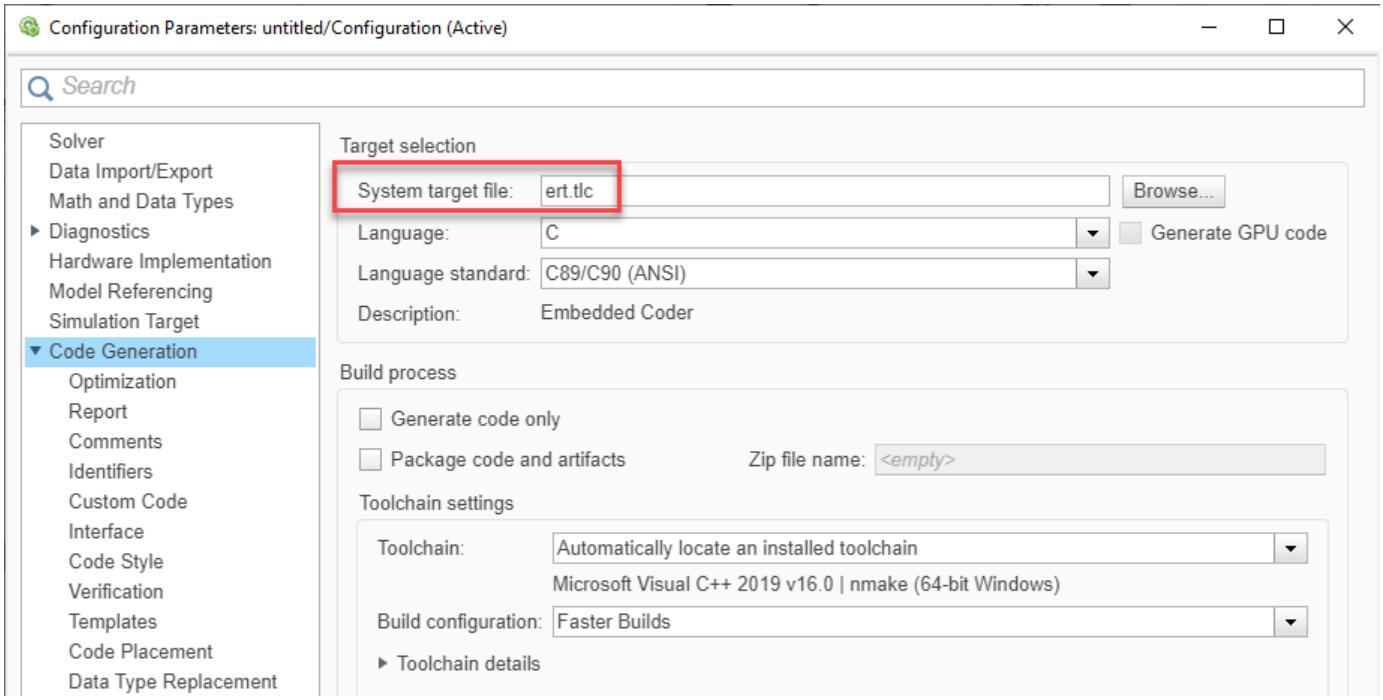
To configure a Simulink model to generate SIMD code on Intel®:

- In the **Modeling** tab of the model toolstrip, click **Model Settings**.

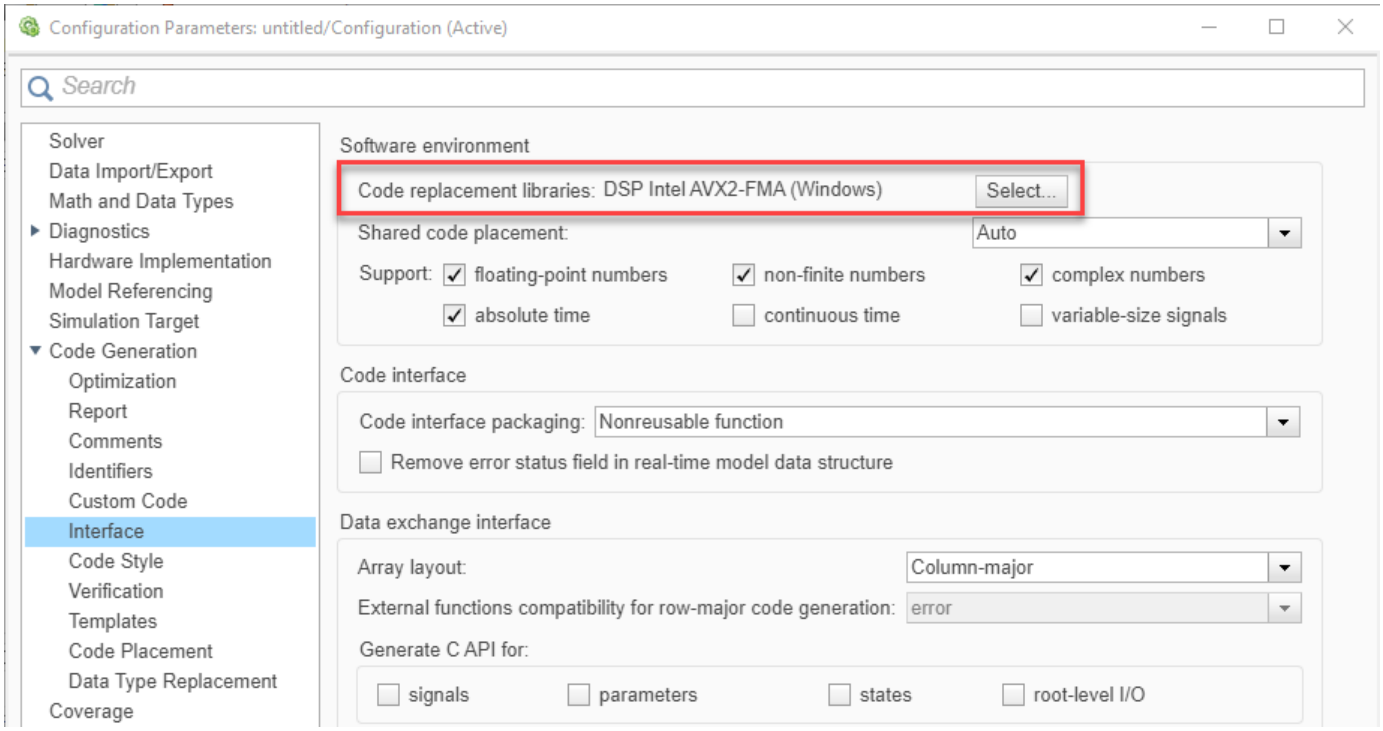


- In the Configuration Parameters dialog box that opens, in the **Code Generation** pane, set the **System target file** to `ert.tlc`.

> **Note** This workflow does not support the MinGW C/C++ compiler. You can choose any other compiler listed in Supported Compilers. To select a different compiler that is installed on your machine, run `mex -setup` in the MATLAB command prompt and follow the instructions.



- Under **Code Generation**, in the **Interface** pane, set the **Code Replacement libraries** to either `DSP Intel AVX2-FMA (Windows)`, `DSP Intel AVX2-FMA (Linux)`, or `DSP Intel AVX2-FMA (Mac)`. Using these libraries, you can generate high performance SIMD code. For more information on Code Replacement Libraries, see “What Is Code Replacement?” (Embedded Coder).



- In the model window, initiate code generation and the build process for the model by using one of these common options:

  - Click the Build Model button.
  - Press **Ctrl+B**.

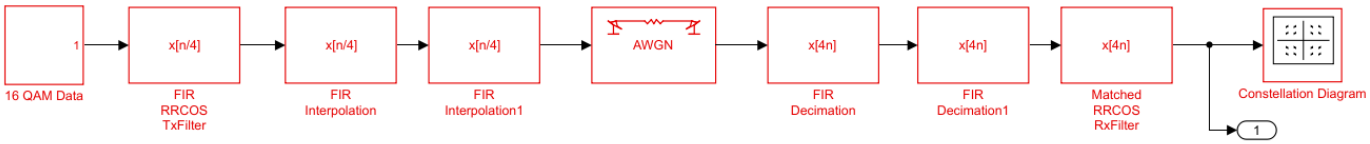
  For an example on how to select a system target file for a Simulink model and how to generate C code for embedded systems, see “Generate Code Using Embedded Coder®” (Embedded Coder).

  The SIMD code is generated using Intel AVX2-FMA technology. The Intel AVX2 SIMD intrinsics significantly improve the performance of the code generated from the supported algorithms on Intel platforms, in most cases meeting or exceeding the performance of the simulation and plain C code.

Compare the Performance of SIMD Code with Generated Plain C Code

Consider this Simulink model that models a digital communication system. The model contains a root-raised cosine filter on the transmitter and the receiver side, a couple of FIR Interpolation and FIR Decimation blocks to increase and decrease the sample rate of the signal, respectively, and an additive white Gaussian noise (AWGN) communication channel to transmit the signal. The root-raised cosine filters on both sides perform matched filtering. The combined response of the two root-raised cosine filters forms a raised-cosine filter, which helps in minimizing the intersymbol interference (ISI). Due to matched filtering, the signal received at the output has a high signal to noise ratio (SNR) and low probability of error. To confirm, view the output in the constellation diagram that follows.

To open the model, type `ex_qam_matchedfilter` in the MATLAB command prompt.



Copyright 2020 The MathWorks, Inc.

In the **Modeling** tab of the model, click **Model Settings**. In the configuration parameters window that opens, under **Code Generation** in the **Interface** pane, set **Code replacement libraries** to `None`. Build the model and this setting generates plain C code executable in the current MATLAB directory. However, if you specify a code generation folder (Simulink) in Simulink preferences, building the model generates plain C code executable in the specified folder. Measure the time it takes to run the executable.

```
tic;
system('ex_qam_matchedfilter');
tplain = toc

tplain =

   37.4883
```

Repeat the process by setting the **Code replacement libraries** to `DSP Intel AVX2-FMA (Windows)`, `DSP Intel AVX2-FMA (Linux)`, or `DSP Intel AVX2-FMA (Mac)`, depending on the platform of the machine you are using. Build the model and measure the time it takes to run the generated AVX2 executable.

```
tic;
system('ex_qam_matchedfilter');
tavx2 = toc

tavx2 =

   8.29
```

The generated SIMD code is around 4.5x compared to the plain C code on a Windows 10 machine.

```
tplain/tavx2

ans =

    4.5221
```

See Also

More About

- “Simulink Blocks in DSP System Toolbox that Support SIMD Code Generation” on page 20-59
- “Generate High Performance SIMD Code on Intel from MATLAB Algorithms in DSP System Toolbox” on page 20-57
- “Generate Code Using Embedded Coder®” (Embedded Coder)

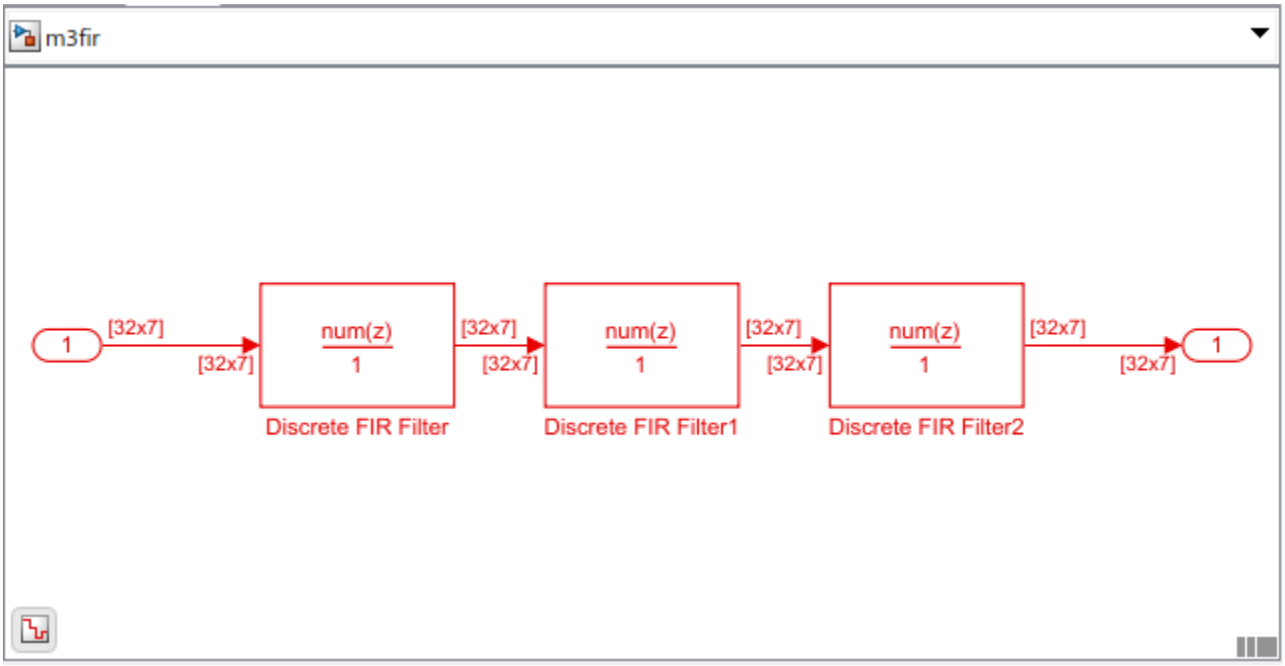
In-Place Memory Optimization

In-place optimization is a memory optimization technique that uses a single buffer, that is, the same memory to store the input and output data. Every time there is an intermediary output value in the algorithm, the same buffer is overwritten to store this value. This technique optimizes the memory usage and generates code that uses very less memory.

These following features in DSP System Toolbox support in-place memory optimization.

- Discrete FIR Filter
- Array-Vector Add
- Array-Vector Subtract
- Array-Vector Multiply
- Array-Vector Divide
- `dsp.FIRFilter`

To illustrate the technique of in-place optimization, consider this model that contains a sequence of three, connected Discrete FIR Filter blocks. Each block filters the input it receives and generates an output that is of the same size as the input.



When you generate code from such a model, you can see the in-place optimization in the generated code.

This section shows the in-place optimized generated code. The three `for` loops with the iteration index *n* correspond to the respective Discrete FIR Filter block in the Simulink model. In each of these three `for` loops, the filter output is computed and stored in the **m3fir_Y.Output[]** buffer. The **m3fir_Y.Output[]** buffer rewrites its value every time the Discrete FIR Filter block has an updated output.

```
/* DiscreteFir: '<Root>/Discrete FIR Filter' incorporates:
 *  DiscreteFir: '<Root>/Discrete FIR Filter2'
 *  Inport: '<Root>/Input'
 */
memcpy(&m3fir_Y.Output[0], &m3fir_U.Input[0], 224U * sizeof(real_T));
for (chan = 0; chan < 7; chan++) {
  for (k = 0; k < 32; k++) {
    acc1 = 0.0;

    /* DiscreteFir: '<Root>/Discrete FIR Filter1' incorporates:
     *  DiscreteFir: '<Root>/Discrete FIR Filter2'
     */
    /* load input sample */
    zNext_tmp_tmp = chan << 5;
    zNext_tmp = zNext_tmp_tmp + k;
    zNext = m3fir_Y.Output[zNext_tmp];
    for (n = 0; n < 16; n++) {
      /* shift state */
      zCurr = zNext;
      zNext_tmp_0 = (chan << 4) + n;
      zNext = m3fir_DW.DiscreteFIRFilter_states[zNext_tmp_0];
      m3fir_DW.DiscreteFIRFilter_states[zNext_tmp_0] = zCurr;

      /* compute one tap */
      acc1 += m3fir_ConstP.DiscreteFIRFilter_Coefficients[n] * zCurr;
    }

    /* compute last tap */
    m3fir_Y.Output[zNext_tmp] = m3fir_ConstP.DiscreteFIRFilter_Coefficients[n]
      * zNext + acc1;
```

```
    /* DiscreteFir: '<Root>/Discrete FIR Filter1' */
    /* store output sample */
    accl = 0.0;

    /* load input sample */
    zNext = m3fir_Y.Output[zNext_tmp];
    for (n = 0; n < 32; n++) {
      /* shift state */
      zCurr = zNext;
      zNext_tmp_0 = zNext_tmp_tmp + n;
      zNext = m3fir_DW.DiscreteFIRFilter1_states[zNext_tmp_0];
      m3fir_DW.DiscreteFIRFilter1_states[zNext_tmp_0] = zCurr;

      /* compute one tap */
      accl += m3fir_ConstP.DiscreteFIRFilter1_Coefficients[n] * zCurr;
    }

    /* compute last tap */
    m3fir_Y.Output[zNext_tmp] = m3fir_ConstP.DiscreteFIRFilter1_Coefficients[n]
      * zNext + accl;

    /* DiscreteFir: '<Root>/Discrete FIR Filter2' */
    /* store output sample */
    accl = 0.0;

    /* load input sample */
    zNext = m3fir_Y.Output[zNext_tmp];
    for (n = 0; n < 48; n++) {
      /* shift state */
      zCurr = zNext;
      zNext_tmp_0 = chan * 48 + n;
      zNext = m3fir_DW.DiscreteFIRFilter2_states[zNext_tmp_0];
      m3fir_DW.DiscreteFIRFilter2_states[zNext_tmp_0] = zCurr;

      /* compute one tap */
      accl += m3fir_ConstP.DiscreteFIRFilter2_Coefficients[n] * zCurr;
    }

    /* compute last tap */
    m3fir_Y.Output[zNext_tmp] = m3fir_ConstP.DiscreteFIRFilter2_Coefficients[n]
      * zNext + accl;

    /* store output sample */
  }
}

/* End of DiscreteFir: '<Root>/Discrete FIR Filter' */
```

The generated code reuses the output buffer and hence is efficient and uses less memory.

When you generate code from a MATLAB algorithm containing a sequence of `dsp.FIRFilter` objects, you see a similar optimization in the generated code.

See Also

Functions
`codegen`

Related Examples

- “Generate C Code from Simulink Model” on page 20-19
- “Generate C Code from MATLAB Code” on page 20-10

HDL Code Generation

- “Find Blocks That Support HDL Code Generation” on page 21-2
- “High Throughput HDL Algorithms” on page 21-4
- “HDL Filter Architectures” on page 21-6
- “Subsystem Optimizations for Filters” on page 21-11
- “Multichannel FIR Filter for FPGA” on page 21-21
- “Programmable FIR Filter for FPGA” on page 21-24
- “Implementing the Filter Chain of a Digital Down-Converter in HDL” on page 21-30
- “HDL Implementation of a Digital Down-Converter for LTE” on page 21-50
- “HDL Implementation of a Digital Up-Converter for LTE” on page 21-69

Find Blocks That Support HDL Code Generation

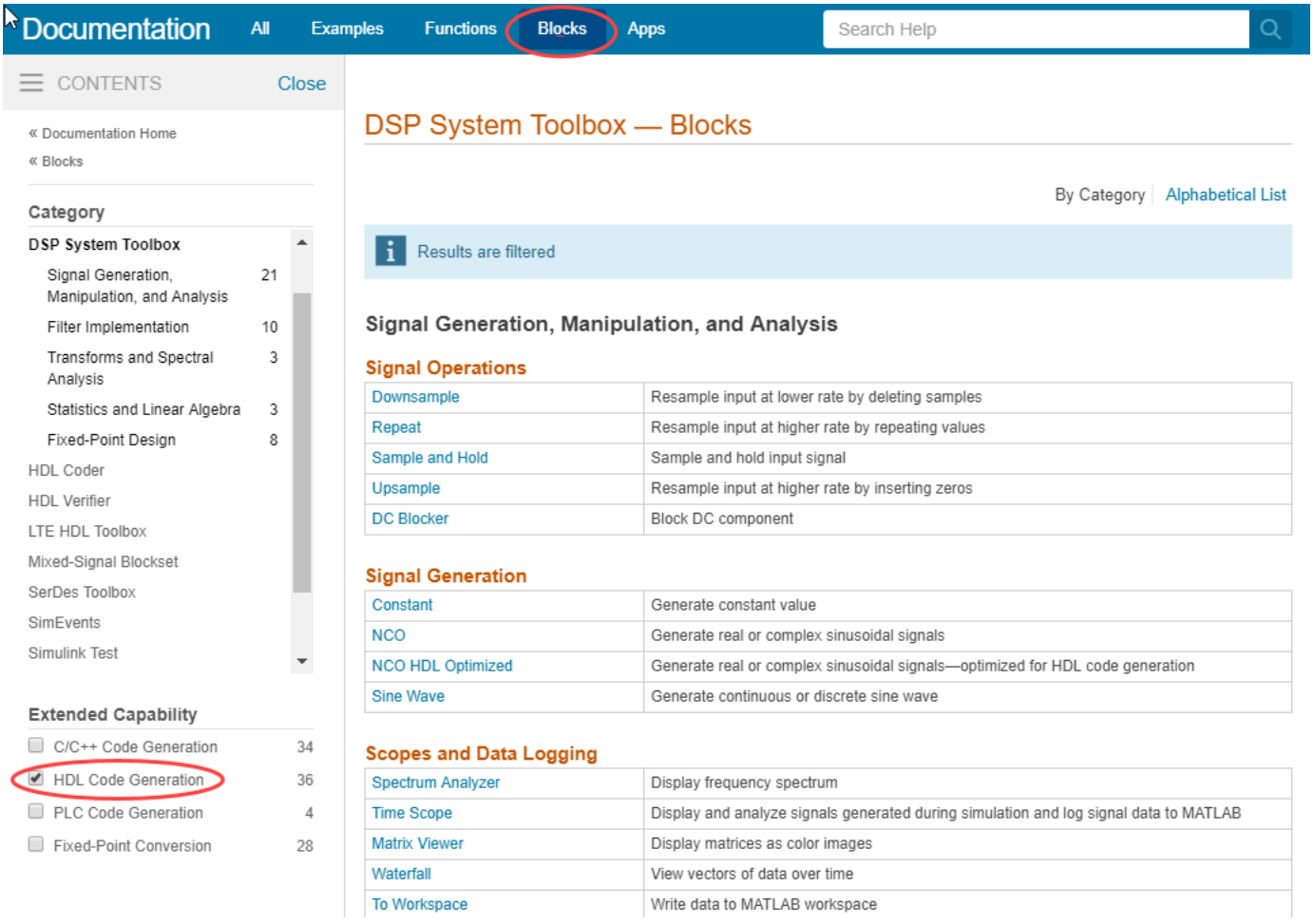
Blocks

In the Simulink library browser, you can find libraries of blocks supported for HDL code generation in the **HDL Coder**, **Communications Toolbox HDL Support**, **DSP System Toolbox HDL Support** block libraries, and others.

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.

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.

Refer to the "Extended Capabilities > HDL Code Generation" section of each block page for block implementations, properties, and restrictions for HDL code generation.



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 can use 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.	Frame-based input is supported only when **Architecture** is set to `Streaming Radix 2^2`.
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.	
FIR Decimation HDL Optimized	Connect a column vector to the input `data` port. The vector size must be less than or equal to 64 samples.	The decimation factor must be an integer multiple of the input vector size.

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.	
CIC Decimation HDL Optimized	Connect a column vector to the input `data` port. The input vector size can be up to 64 samples.	Frame-based input is supported only when **Variable decimation factor** is not selected.
Complex to Magnitude-Angle HDL Optimized	Connect a column vector to the input `data` port. The input vector size can be up to 64 samples.	
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” on page 9-23

HDL Filter Architectures

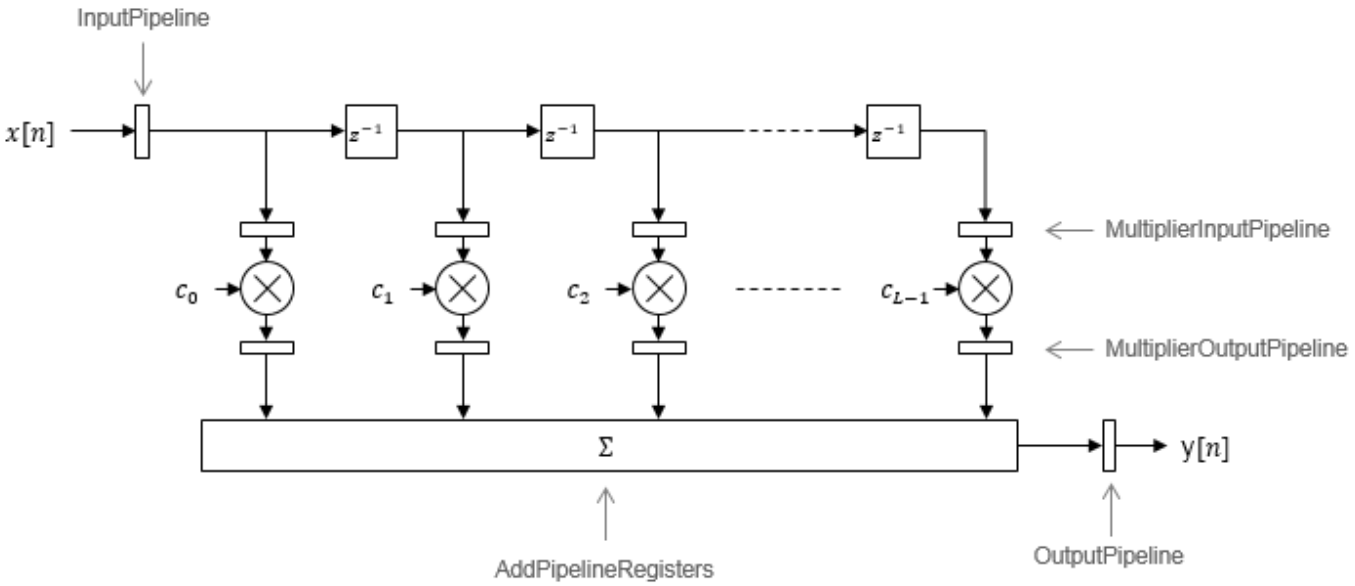
The HDL Coder software provides architecture options that extend your control over speed vs. area tradeoffs in the realization of filter designs. To achieve the desired tradeoff for generated HDL code, you can either specify a fully parallel architecture, or choose one of several serial architectures. 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

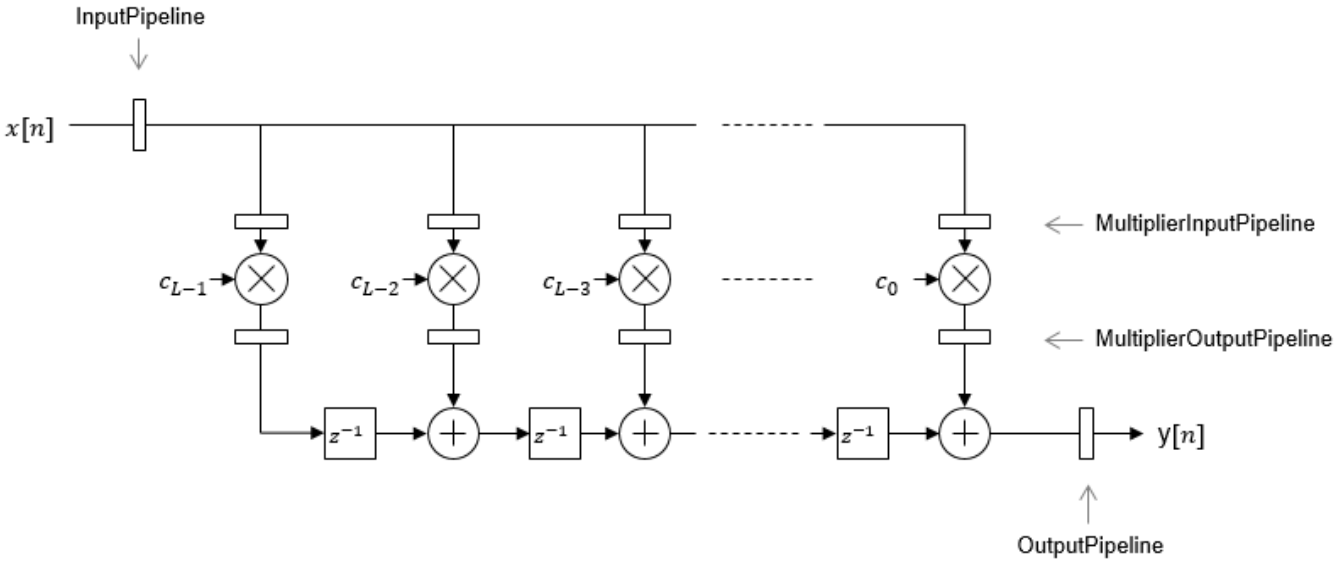
This option is the default architecture. A fully parallel architecture uses a dedicated multiplier and adder for each filter tap. The taps execute in parallel. A fully parallel architecture is optimal for speed. However, it requires more multipliers and adders than a serial architecture, and therefore 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

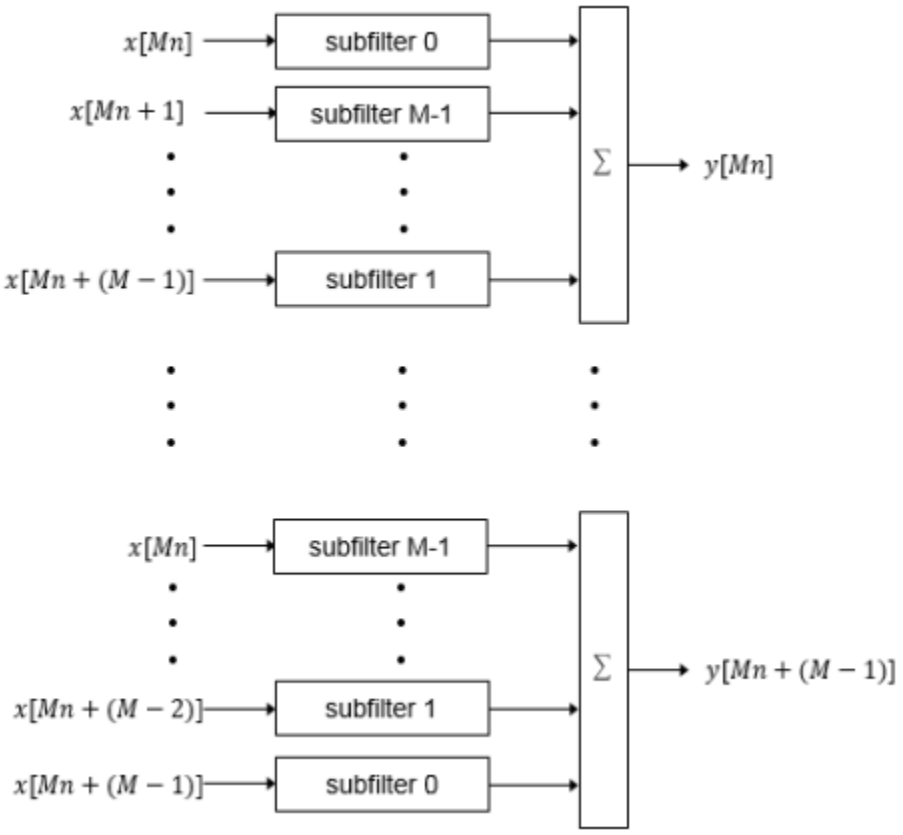
Serialization of a filter increases the total latency of the design by one clock cycle. The serial architectures use an accumulator (an adder with a register) to add the products sequentially. An additional final register is used to store the summed result of all the serial partitions, requiring an extra 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.

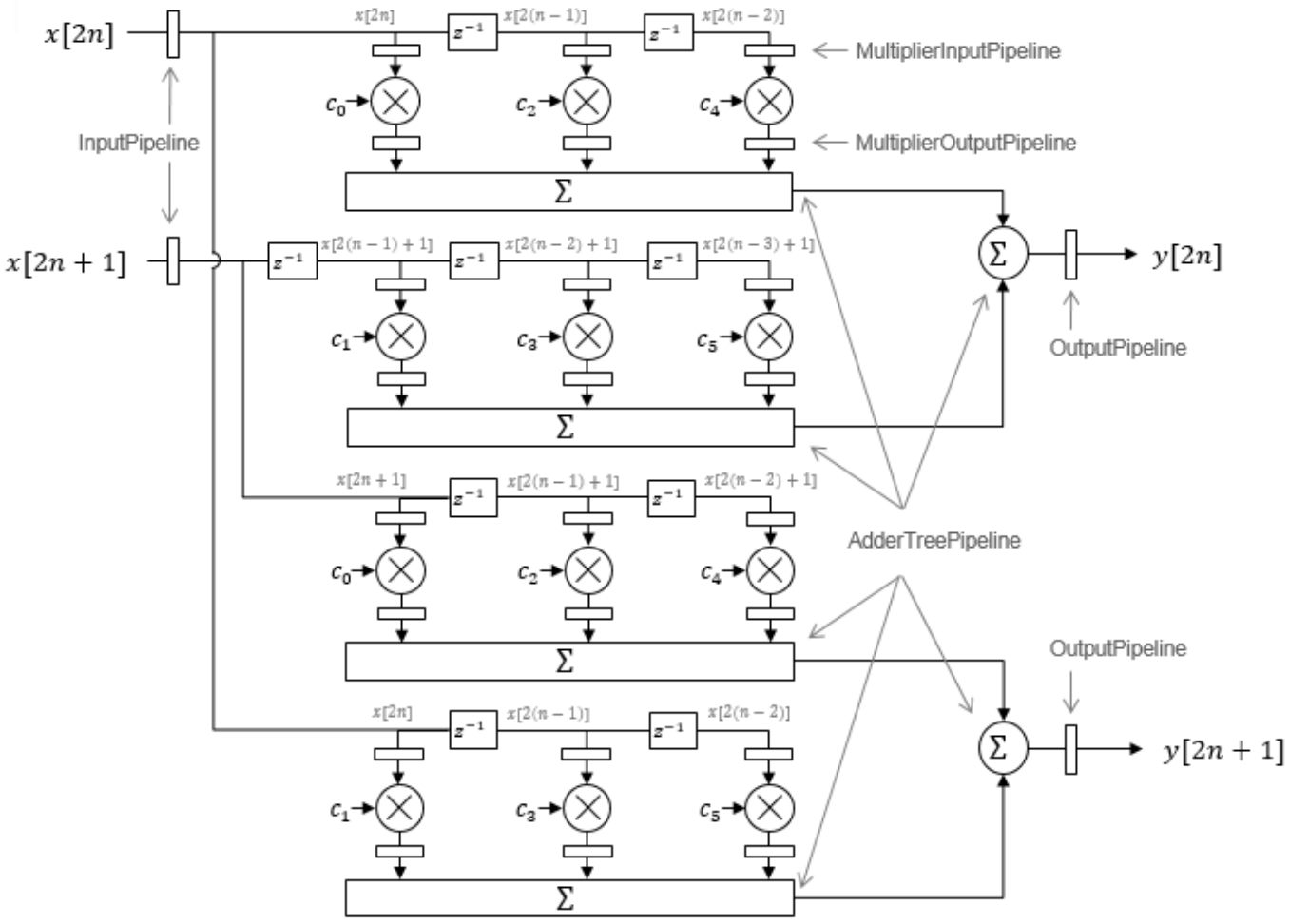


Each of the subfilters includes every *M*th coefficient. The subfilter results are added so that each output sample is the sum of each of the coefficients multiplied with one input sample.

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 Multichannel Filter Subsystem” on page 21-12 example.

You can also use a **SharingFactor** with multichannel filters. See “Area Reduction of Filter Subsystem” on page 21-17.

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 Filter Subsystem” on page 21-17 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:

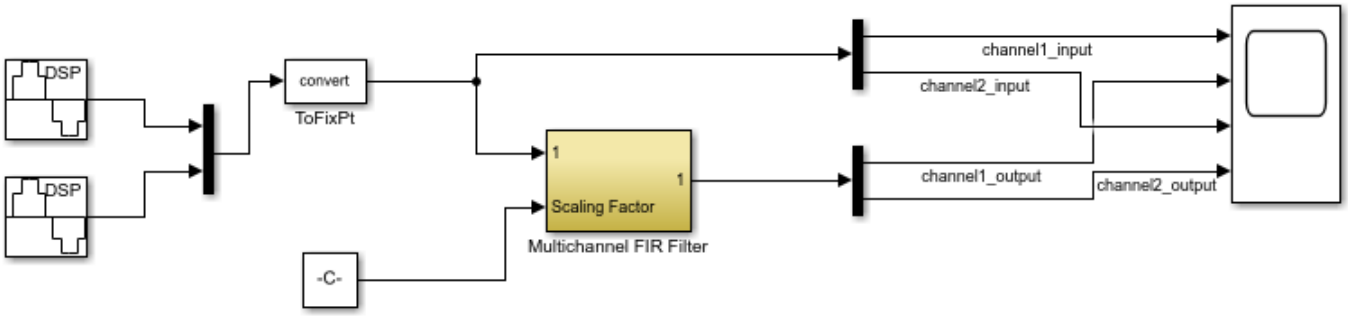
- **InputPipeline** and **OutputPipeline**
- **MultiplierInputPipeline** and **MultiplierOutputPipeline**
- **AddPipelineRegisters**

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).

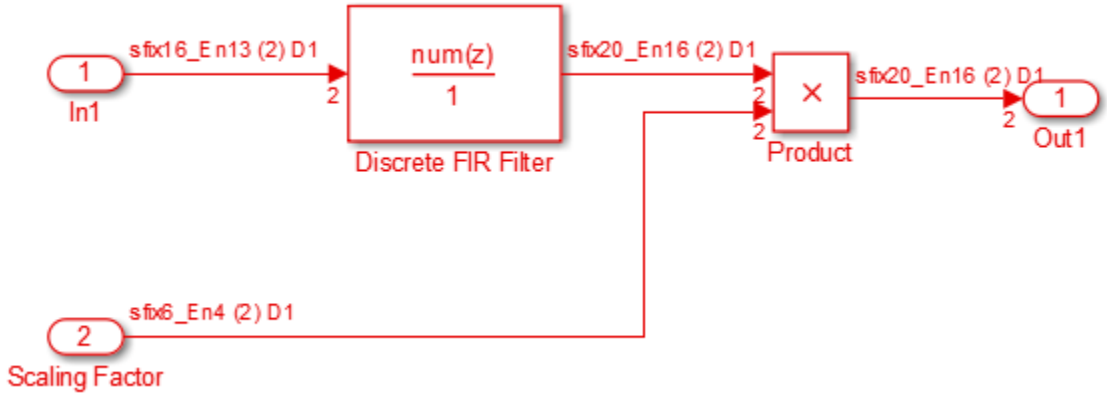
Area Reduction of Multichannel Filter Subsystem

To reduce the number of multipliers in the HDL implementation of a multichannel filter and surrounding logic, use the **StreamingFactor** HDL Coder™ optimization.



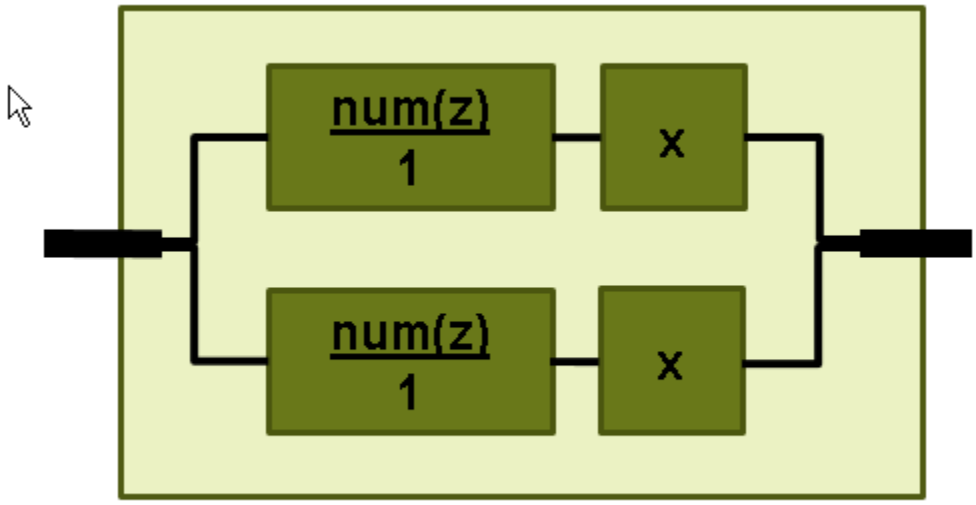
Copyright 2016 The MathWorks, Inc.

The model includes a two-channel sinusoidal signal source feeding a filter subsystem targeted for HDL code generation.

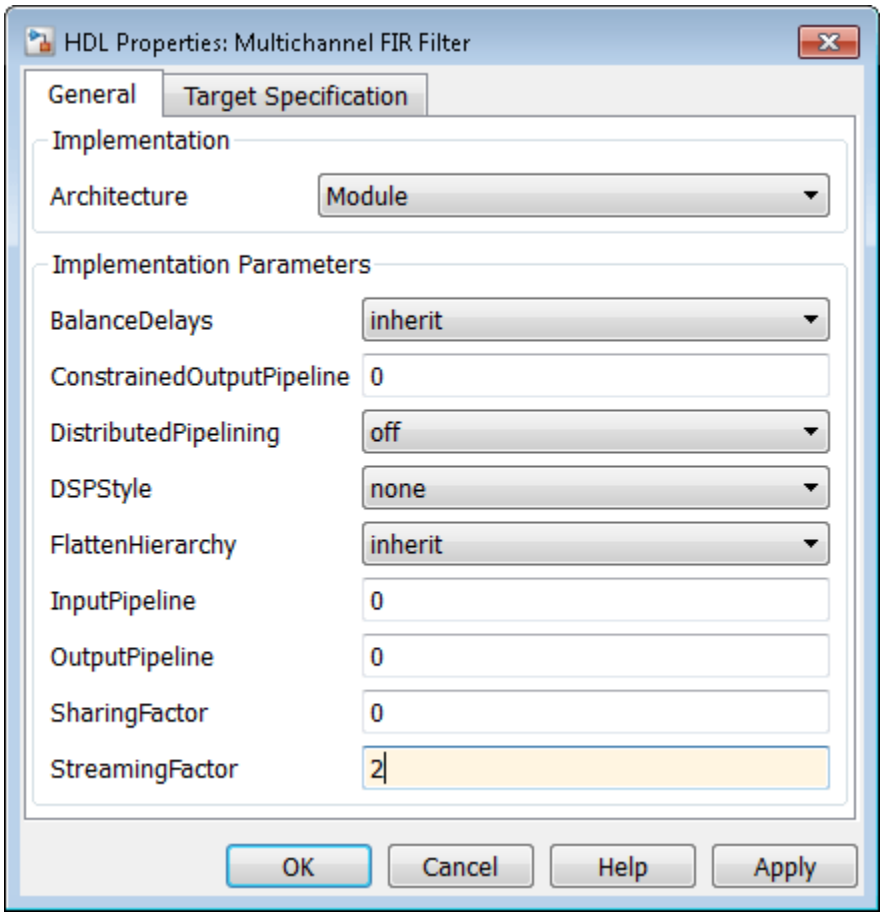


The subsystem contains a Discrete FIR Filter block and a constant multiplier. The multiplier is included to show the optimizations operating over all eligible logic in a subsystem.

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.



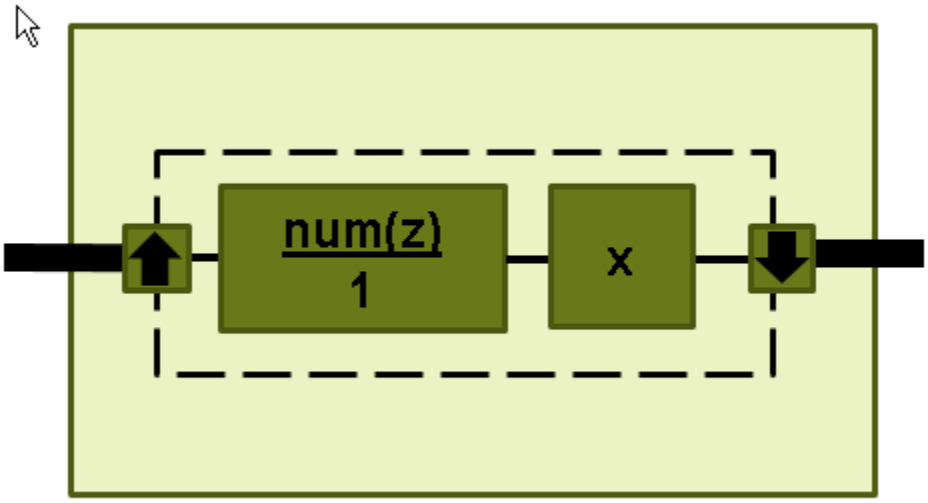
To enable streaming optimization for the Multichannel FIR Filter Subsystem, right-click the subsystem and select **HDL Code > HDL Block Properties**.



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**.

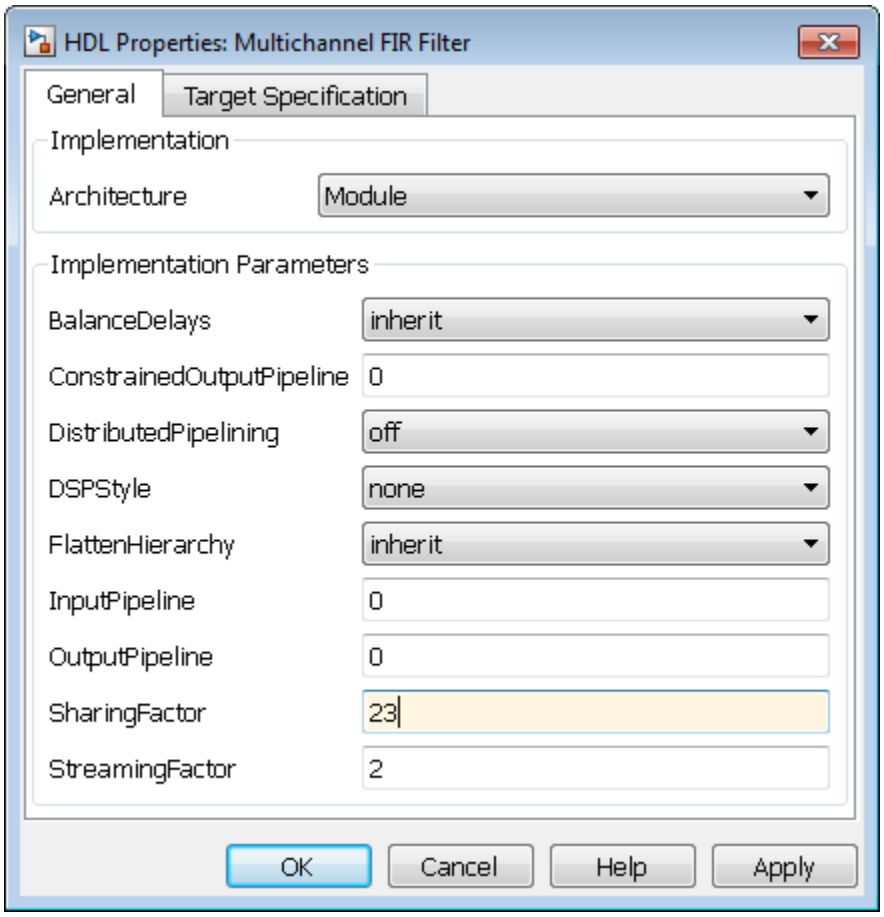
With the streaming factor applied, the logic for one channel is instantiated once and run at twice the rate of the original model.



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

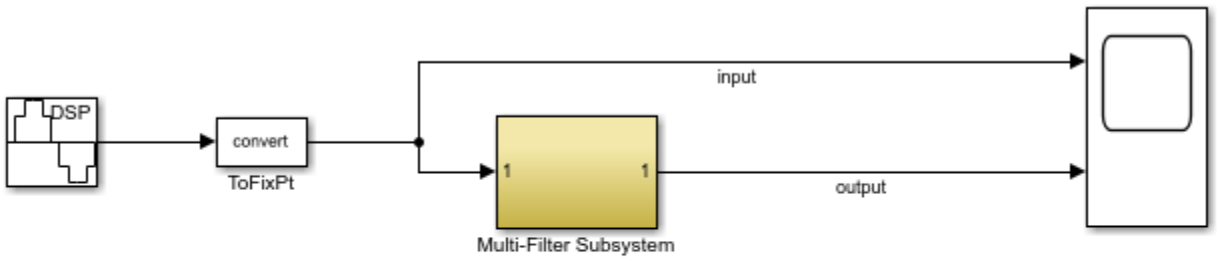
Report for Subsystem: *Multichannel FIR Filter*

Multipliers (2)

```
16x16-bit Multiply : 1
[+] 16x6-bit Multiply : 1
```

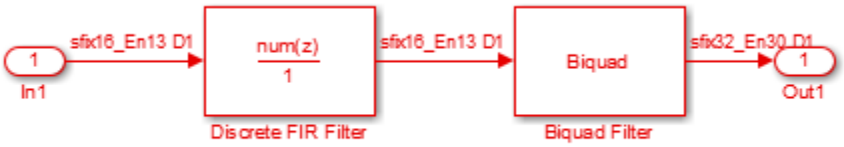
Area Reduction of Filter Subsystem

To reduce the number of multipliers in the HDL implementation of a multifilter design, use the **SharingFactor** HDL Coder™ optimization.



Copyright 2016 The MathWorks, Inc.

The model includes a sinusoidal signal source feeding a filter subsystem targeted for HDL code generation.

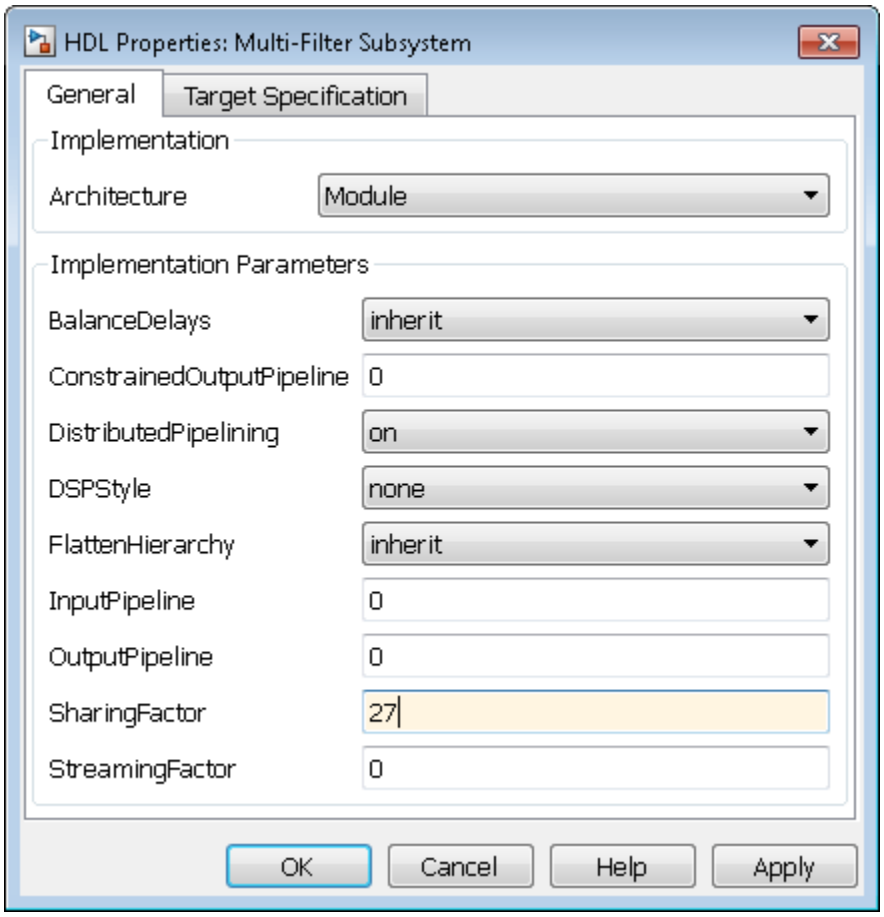


The subsystem contains a Discrete FIR Filter block and a Biquad Filter block. This design demonstrates how the optimization tools share resources between multiple filter blocks.

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

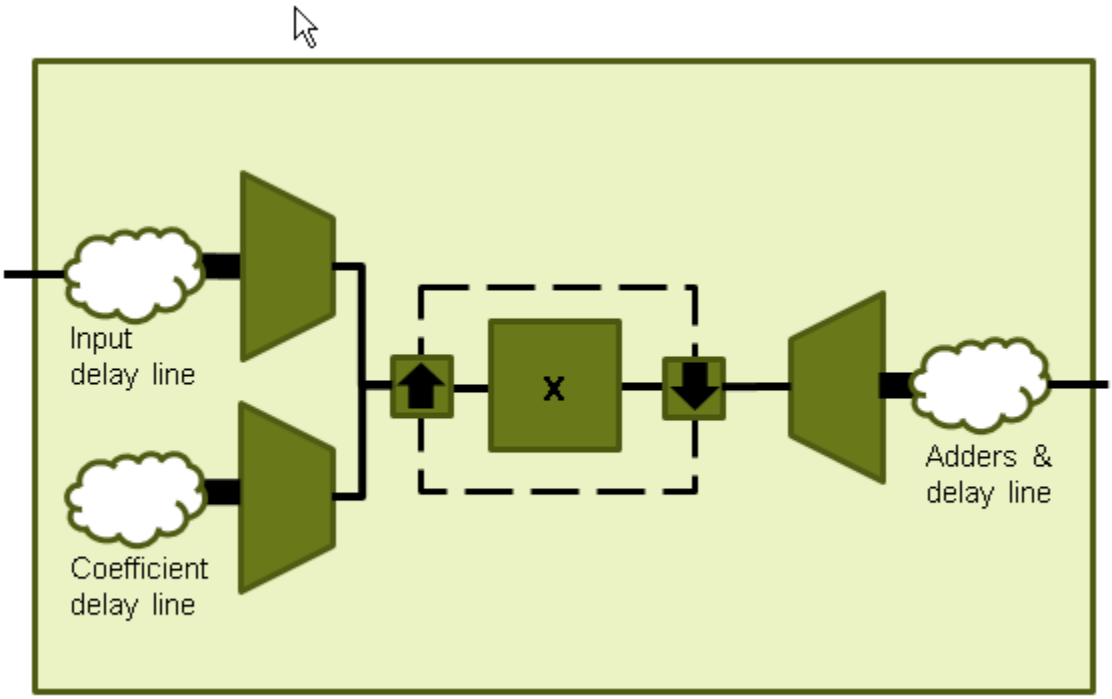
To enable streaming optimization for the **Multi-Filter Subsystem**, right-click the subsystem and select **HDL Code > HDL Block Properties**.



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

See Also

More About

- “Resource Sharing” (HDL Coder)
- “Streaming” (HDL Coder)
- “Clock-Rate Pipelining” (HDL Coder)

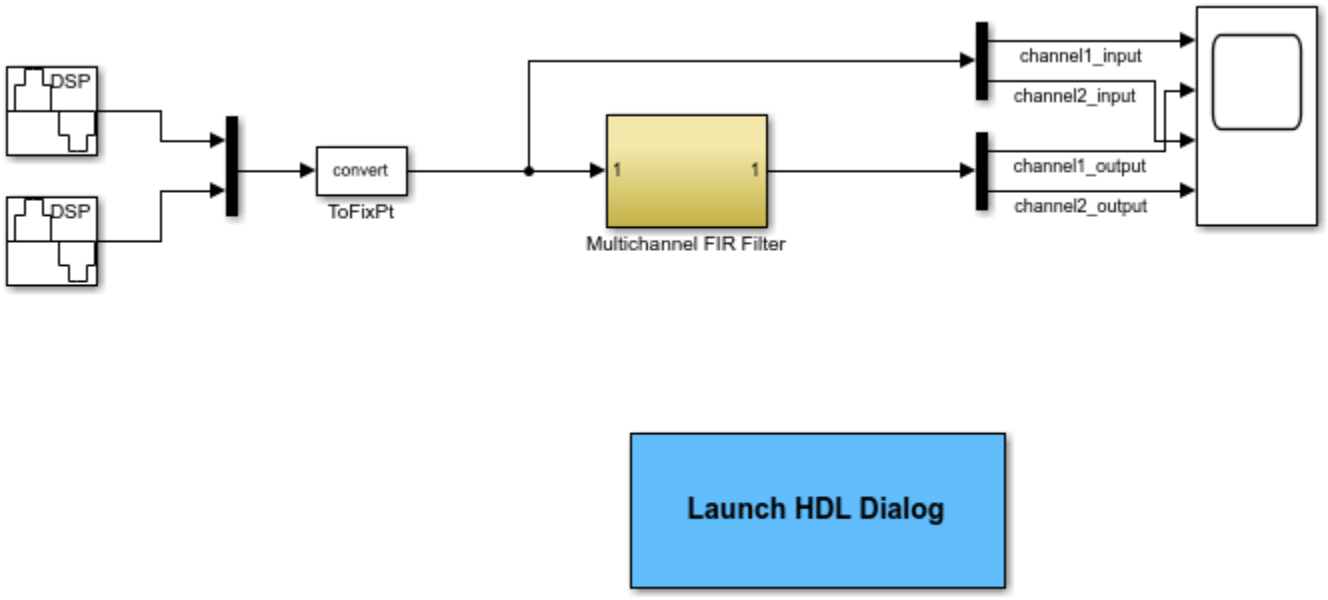
Multichannel FIR Filter for FPGA

This example shows how to implement a discrete FIR filter with multiple input data streams for hardware.

In many DSP applications, multiple data streams are filtered by the same filter. The straightforward solution is to implement a separate filter for each channel. You can create a more area-efficient structure by sharing one filter implementation across multiple channels. The resulting hardware requires a faster clock rate compared to the clock rate used for a single channel filter.

Model Multichannel FIR Filter

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



The model contains a two-channel FIR filter. The input data vector includes two streams of sinusoidal signal with different frequencies. The input data streams are processed by a lowpass filter whose coefficients are specified by the Model Properties `InitFcn` Callback function.

Select a fully parallel architecture for the Discrete FIR Filter block, and enable resource sharing across multiple channels.

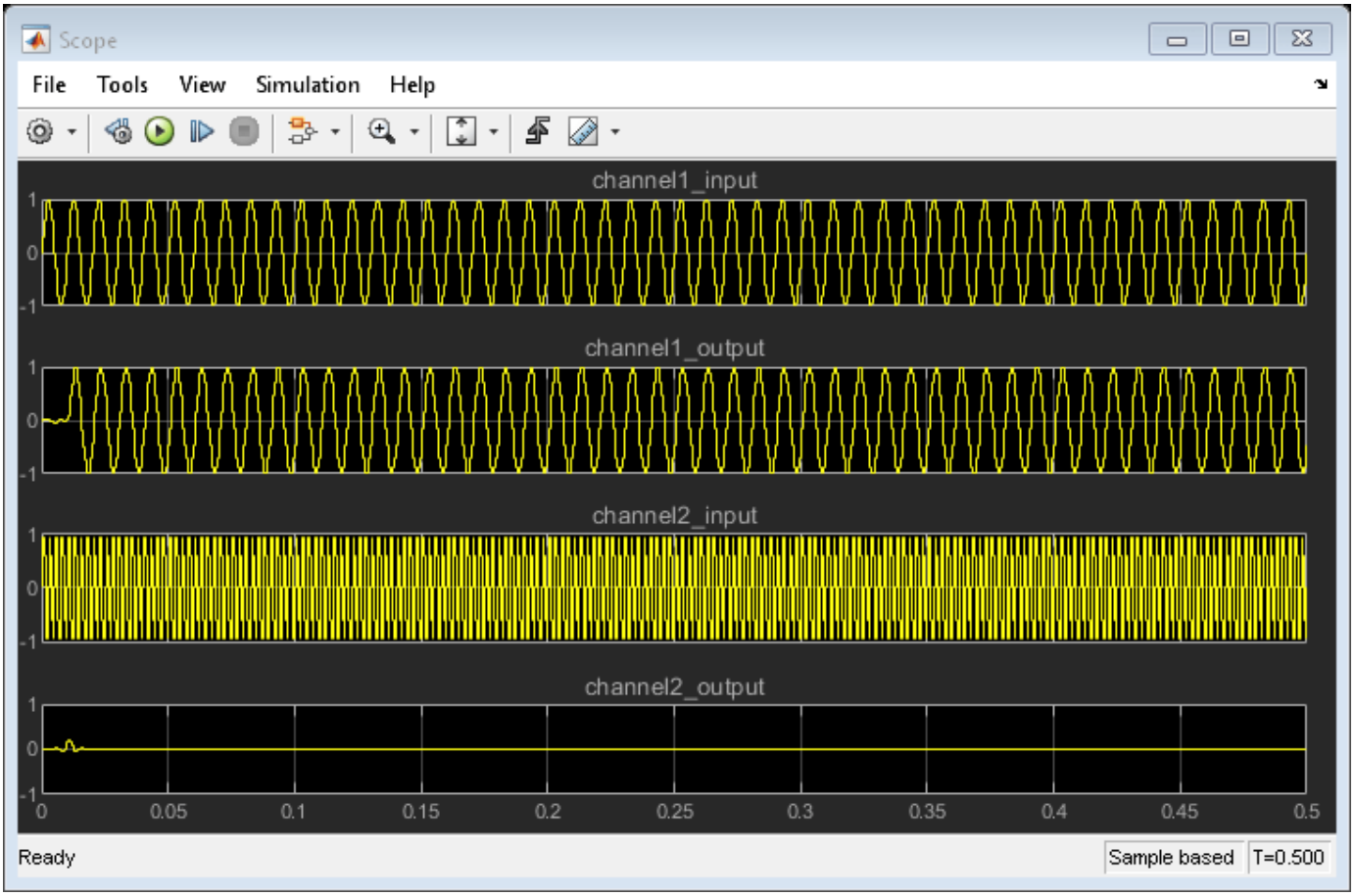
```
systemname = [modelname '/Multichannel FIR Filter'];
blockname = [systemname '/Discrete FIR Filter'];
set_param(blockname,'FilterStructure','Direct form symmetric');
hdlset_param(blockname,'Architecture','Fully Parallel');
hdlset_param(blockname,'ChannelSharing','On');
```

You can alternatively specify these settings on the **HDL Block Properties** menu, which you access by right-clicking a block and selecting **HDL Code > HDL Block Properties**.

Simulation Results

Run the example model and open the scope to compare the two data streams.

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



Generate HDL Code and Test Bench

You must have an HDL Coder™ license to generate HDL code for this example model. Use this command to generate HDL code for the Multichannel FIR Filter subsystem. Enable the resource use report.

```
makehdl(systemname,'resource','on');
```

Use this command to generate a test bench that compares the HDL simulation results with the Simulink model results.

```
makehdltb(systemname);
```

Compare Resource Utilization

To compare resource use with and without sharing, you can disable sharing resources across channels and generate HDL code again, then compare the resource use reports.

```
hdlset_param(blockname,'ChannelSharing','Off');
makehdl(systemname,'resource','on');
```

Summary

Multipliers	44
Adders/Subtractors	86
Registers	86
RAMs	0
Multiplexers	0

Channels are not shared

Summary

Multipliers	22
Adders/Subtractors	43
Registers	91
RAMs	0
Multiplexers	1

Channels are shared

Programmable FIR Filter for FPGA

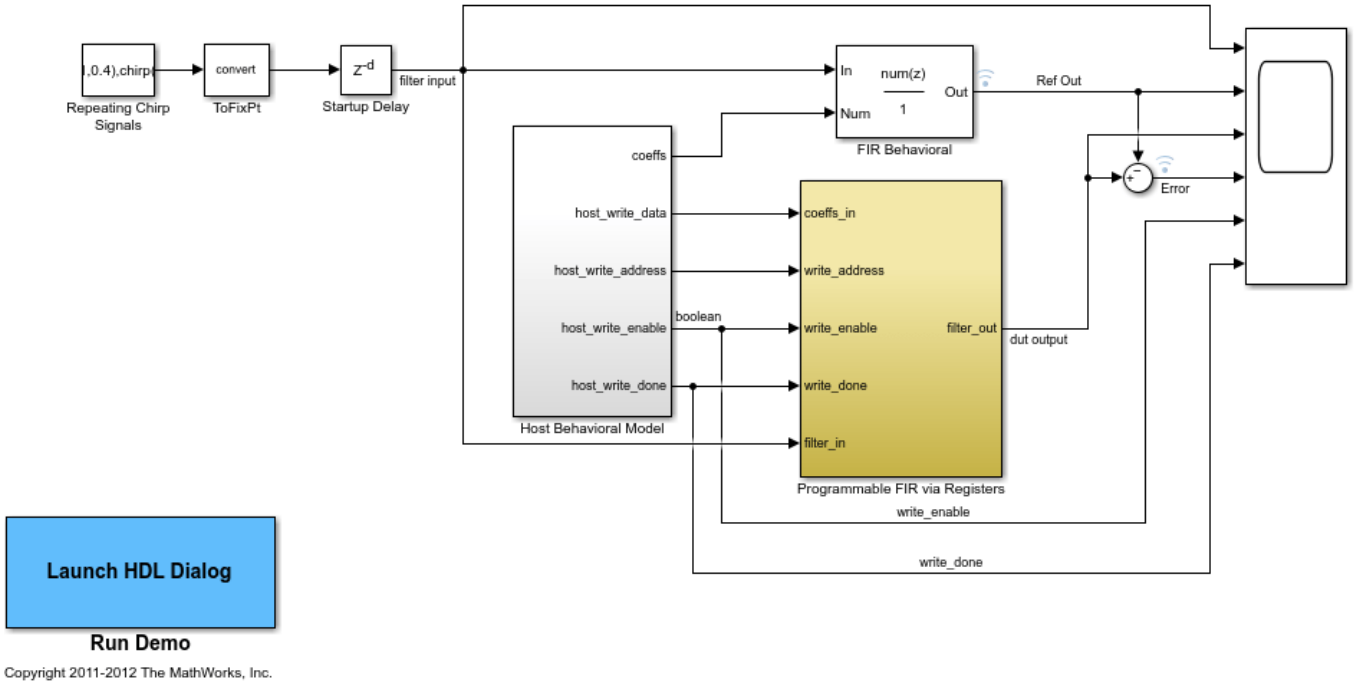
This example shows how to implement a programmable FIR filter for hardware. You can program the filter to a desired response by loading the coefficients into internal registers using the host interface.

In this example, we will implement a bank of filters, each having different responses, on a chip. If all of the filters have a direct-form FIR structure, and the same length, then we can use a host interface to load the coefficients for each response to a register file when needed.

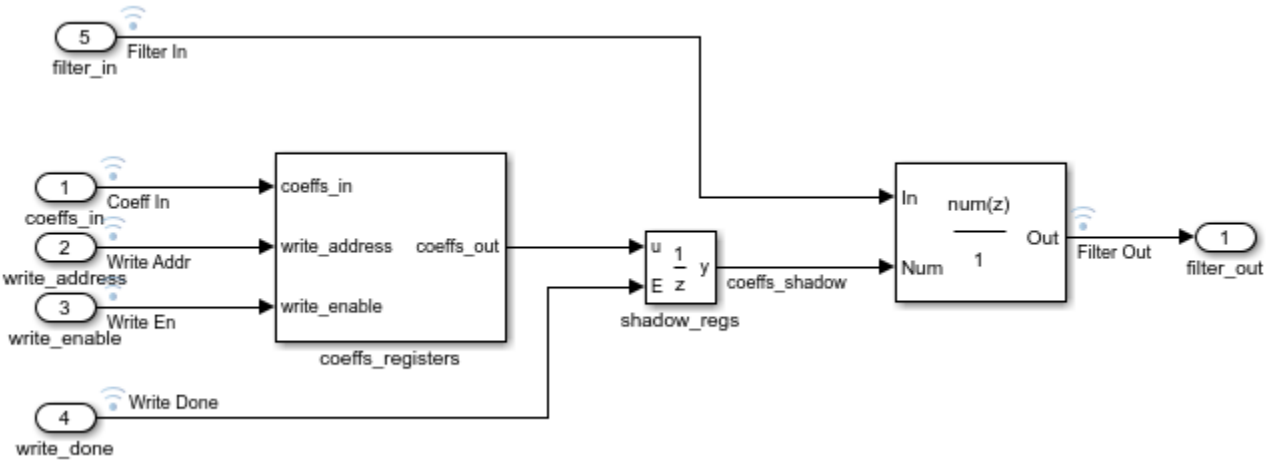
This design adds latency of a few cycles before the input samples can be processed with the loaded coefficients. However, it has the advantage that the same filter hardware can be programmed with new coefficients to obtain a different filter response. This saves chip area, as otherwise each filter would be implemented separately on the chip.

Model Programmable FIR Filter

Consider two FIR filters, one with a lowpass response and the other with a highpass response. The coefficients are specified by using the Model Properties>Callbacks>InitFcn function.



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

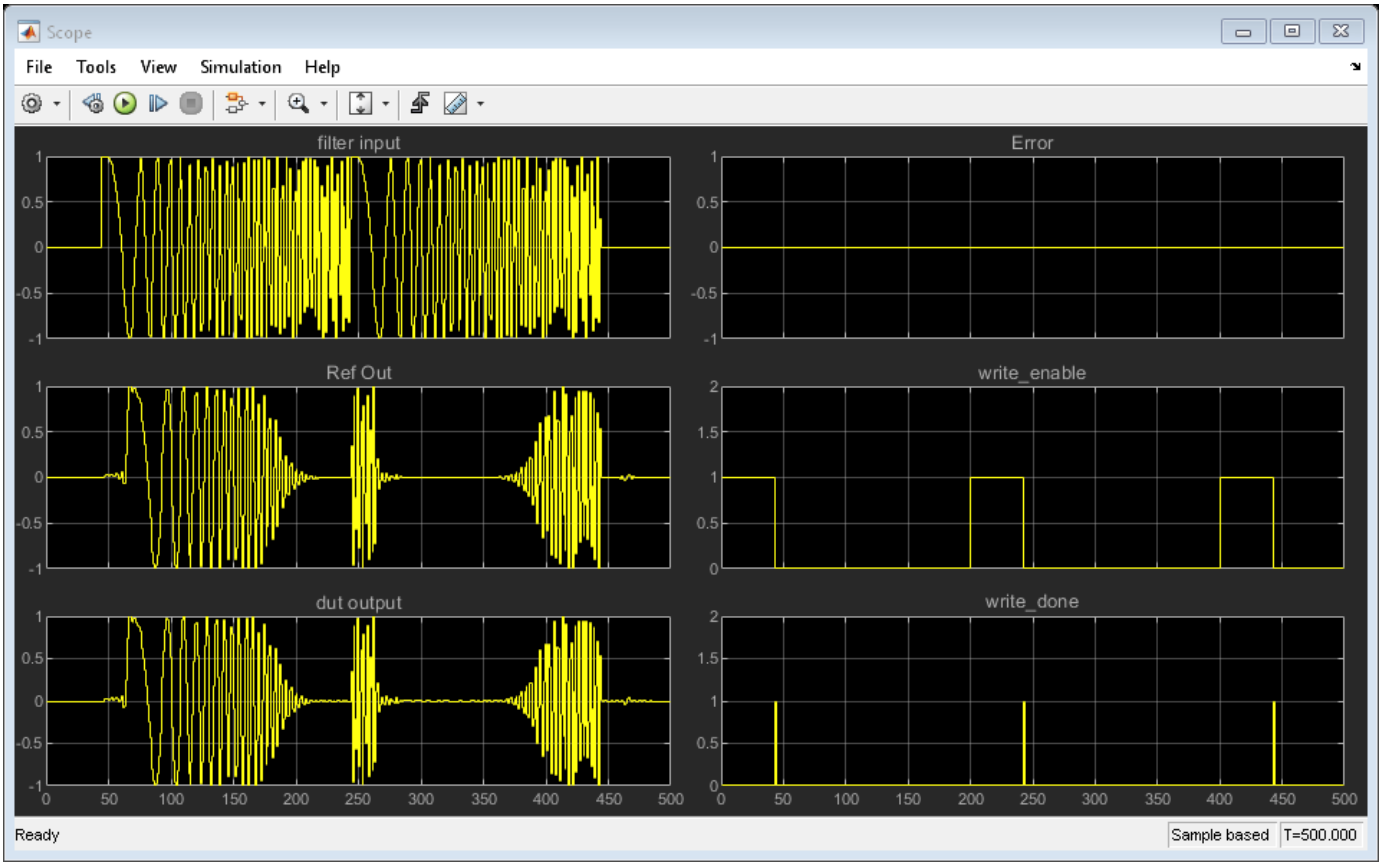


The *coeffs_registers* block loads the coefficients into internal registers when the `write_enable` signal is high. The shadow registers are updated from the coefficients registers when the `write_done` signal is high. The shadow registers enable simultaneous loading and processing of data by the filter entity. The blocks load the second set of coefficients at the same time as processing the last few input samples.

This model is configured to use a fully parallel architecture for the Discrete FIR Filter block. You can also choose serial architectures from the **HDL Block Properties** menu.

Simulink® Simulation Results

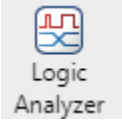
To compare the Design Under Test (DUT) with the reference filter, open the Scope and run the example model.



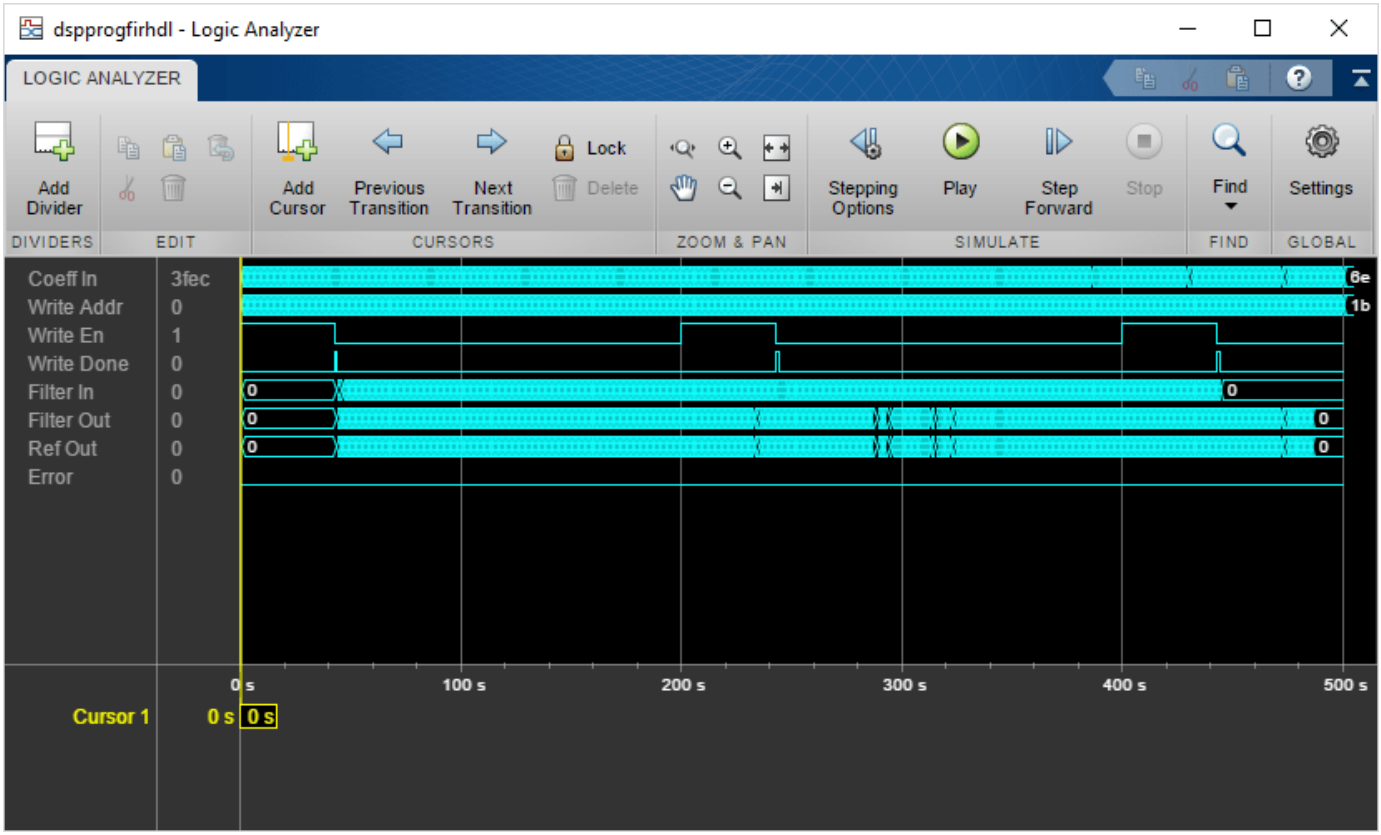
Using the Logic Analyzer

You can also view the signals in the Logic Analyzer. The Logic Analyzer enables you to view multiple signals in one window. It also makes it easy to spot the transitions in the signals.

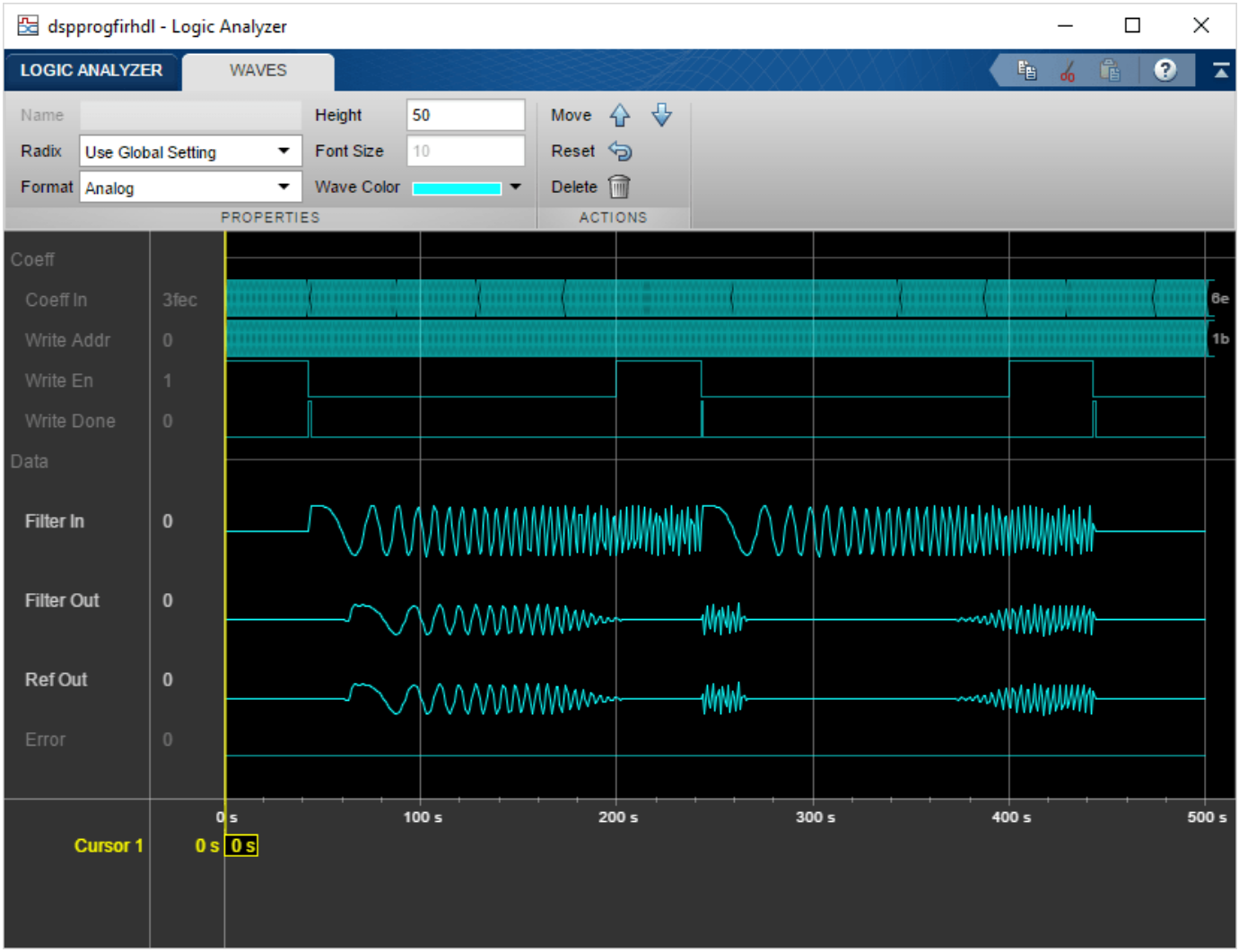
Launch the Logic Analyzer from the model's toolstrip.



The signals of interest -- input coefficients, write address, write enable, write done, filter in, filter out, reference out, and error have been added to the Logic Analyzer for observation.



The Logic Analyzer display can also be controlled on a per-wave or per-divider basis. To modify an individual wave or divider, select a wave or divider and then click on the "Wave" tab. A useful mode of visualization in the Logic Analyzer is the Analog format.



For further information on the Logic Analyzer, refer to the Logic Analyzer documentation.

Generate HDL Code and Test Bench

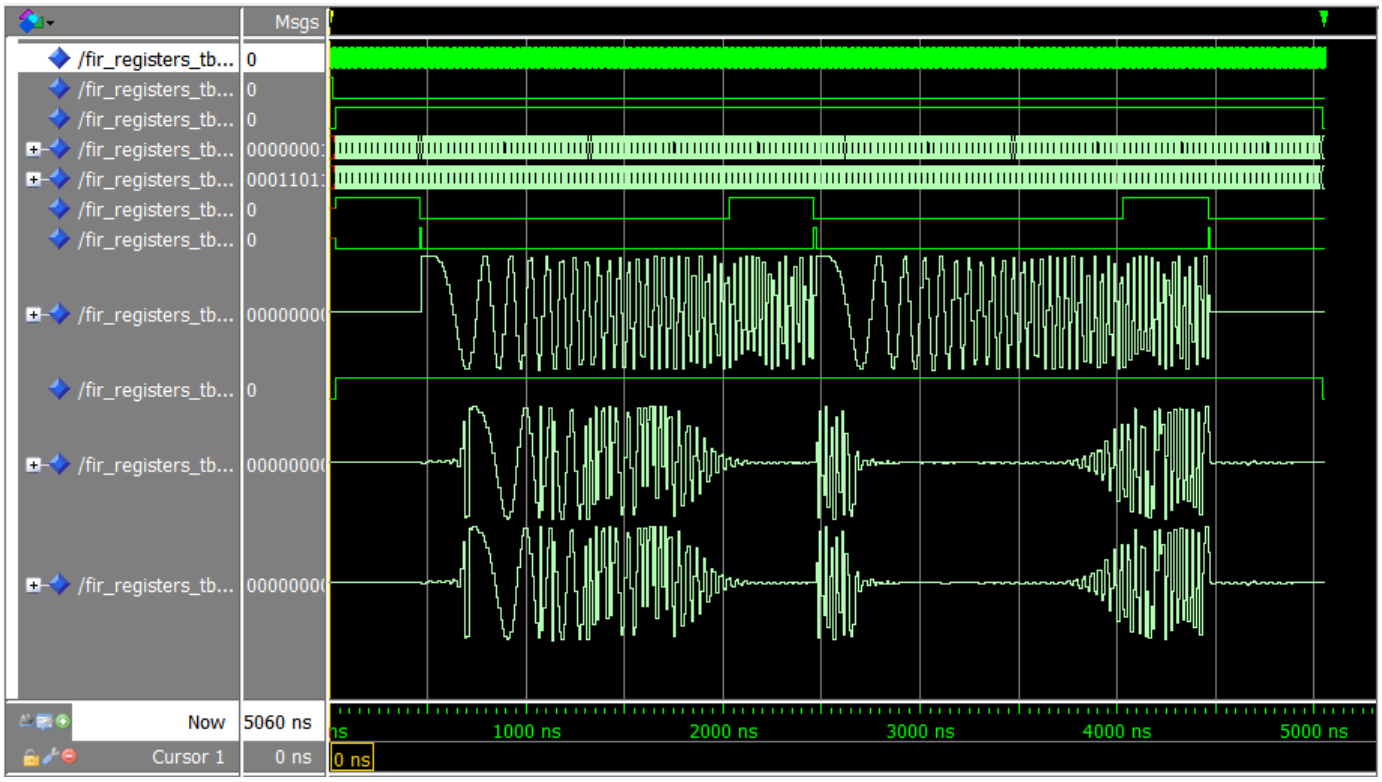
You must have an HDL Coder™ license to generate HDL code for this example model. Use this command to generate HDL code.

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

Use this command to generate a test bench that compares the results of an HDL simulation against the Simulink simulation behavior. makehdltb(systemname);

ModelSim™ Simulation Results

The following figure shows the ModelSim HDL simulator after running the generated .do file scripts for the test bench. Compare the ModelSim result with the Simulink result as plotted before.


Msgs
/fir_registers_tb... 0
/fir_registers_tb... 0
/fir_registers_tb... 0
/fir_registers_tb... 0000000
/fir_registers_tb... 0001101
/fir_registers_tb... 0
/fir_registers_tb... 0
/fir_registers_tb... 0000000
/fir_registers_tb... 0
/fir_registers_tb... 0000000
/fir_registers_tb... 0000000
Now 5060 ns
Cursor 1 0 ns
1000 ns
2000 ns
3000 ns
4000 ns
5000 ns
0 ns

Implementing the Filter Chain of a Digital Down-Converter in HDL

This example shows how to use the DSP System Toolbox™ and Fixed-Point Designer™ to design a three-stage, multirate, fixed-point filter that implements the filter chain of a Digital Down-Converter (DDC) designed to meet the Global System for Mobile (GSM) specification.

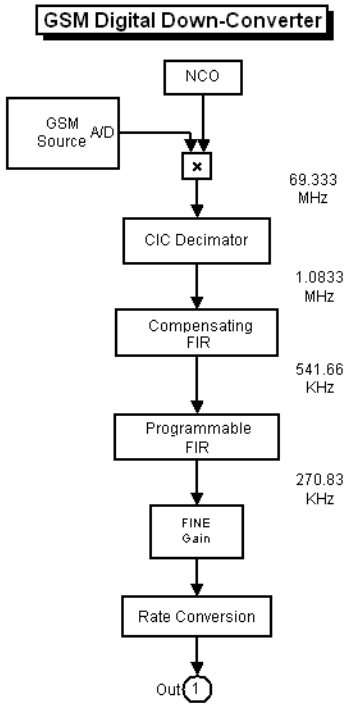
Using the Filter Design HDL Coder™ we will generate synthesizable HDL code for the same three-stage, multirate, fixed-point filter. Finally, using Simulink® and HDL Verifier™ MS, we will co-simulate the fixed-point filters to verify that the generated HDL code produces the same results as the equivalent Simulink behavioral model.

Digital Down-Converter

Digital Down-Converters (DDC) are a key component of digital radios. The DDC performs the frequency translation necessary to convert the high input sample rates found in a digital radio, down to lower sample rates for further and easier processing. In this example, the DDC operates at approximately 70 MHz and must reduce the rate down to 270 KHz.

To further constrain our problem we will model one of the DDCs in Graychip's GC4016 Multi-Standard Quad DDC Chip. The GC4016, among other features, provides the following filters: a five-stage CIC filter with programmable decimation factor (8-4096); a 21-tap FIR filter which decimates by 2 and has programmable 16-bit coefficients; and a 63-tap FIR filter which also decimates by 2 and has programmable 16-bit coefficients.

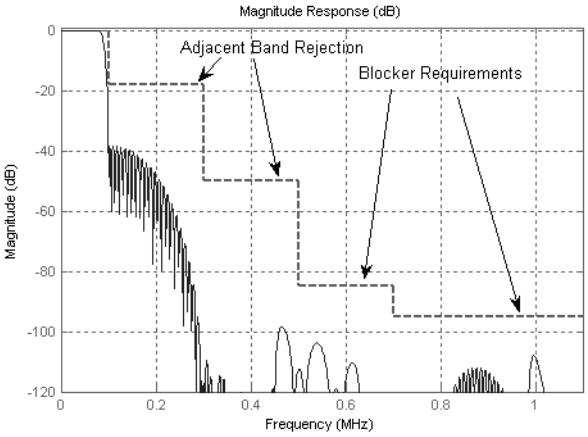
The DDC consists of a Numeric Controlled Oscillator (NCO) and a mixer to quadrature down convert the input signal to baseband. The baseband signal is then low pass filtered by a Cascaded Integrator-Comb (CIC) filter followed by two FIR decimating filters to achieve a low sample-rate of about 270 KHz ready for further processing. The final stage often includes a resampler which interpolates or decimates the signal to achieve the desired sample rate depending on the application. Further filtering can also be achieved with the resampler. A block diagram of a typical DDC is shown below.



This example focuses on the three-stage, multirate, decimation filter, which consists of the CIC and the two decimating FIR filters.

GSM Specifications

The GSM bandwidth of interest is 160 KHz. Therefore, the DDC's three-stage, multirate filter response must be flat over this bandwidth to within the passband ripple, which must be less than 0.1 dB peak to peak. Looking at the GSM out of band rejection mask shown below, we see that the filter must also achieve 18 dB of attenuation at 100 KHz.



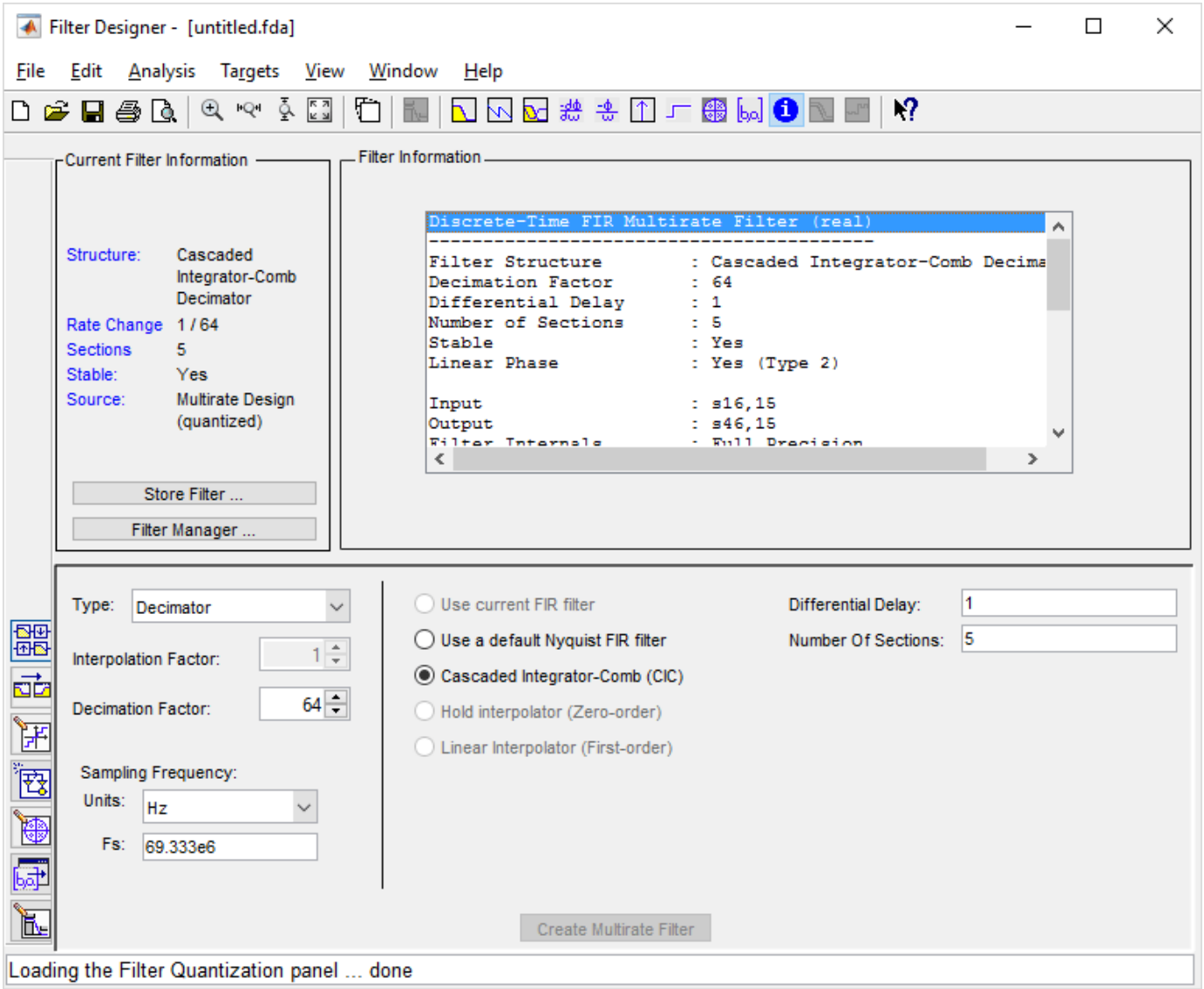
In addition, GSM requires a symbol rate of 270.833 Ksps. Since the Graychip's input sample rate is the same as its clock rate of 69.333 MHz, we must downsample the input down to 270.833 KHz. This requires that the three-stage, multirate filter decimate by 256.

Cascaded Integrator-Comb (CIC) Filter

CIC filters are multirate filters that are very useful because they can achieve high decimation (or interpolation) rates and are implemented without multipliers. CICs are simply boxcar filters implemented recursively cascaded with an upsampler or downsampler. These characteristic make CICs very useful for digital systems operating at high rates, especially when these systems are to be implemented in ASICs or FPGAs.

Although CICs have desirable characteristics they also have some drawbacks, most notably the fact that they incur attenuation in the passband region due to their sinc-like response. For that reason CICs often have to be followed by a compensating filter. The compensating filter must have an inverse-sinc response in the passband region to lift the droop caused by the CIC.

The design and cascade of the three filters can be performed via the graphical user interface Filter Designer,



but we'll use the command line functionality.

We define the CIC as follows:

```
R    = 64; % Decimation factor
D    = 1;  % Differential delay
Nsecs = 5;  % Number of sections

OWL = 20; % Output word length

cic = dsp.CICDecimator('DecimationFactor',R,'NumSections',Nsecs,...
    'FixedPointDataType','Minimum section word lengths',...
    'OutputWordLength',OWL);
```

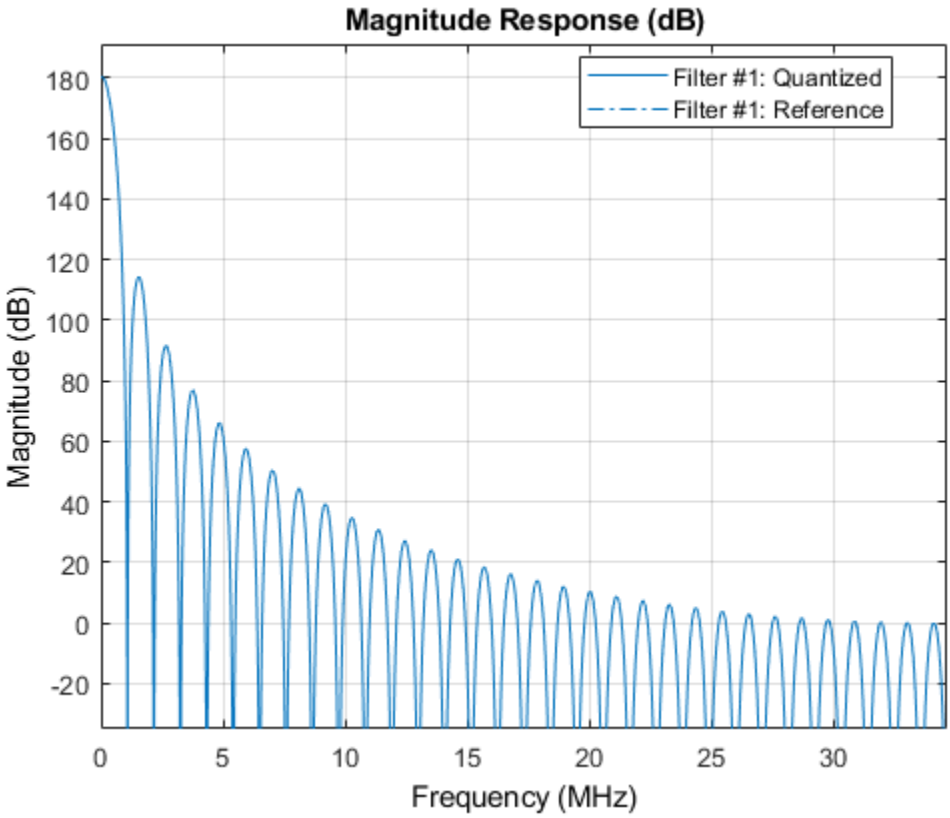
We can view the CIC's details by invoking the info method.

```
info(cic)
```

```
ans =

  9x56 char array

    'Discrete-Time FIR Multirate Filter (real)               '
    '----------------------------------------               '
    'Filter Structure    : Cascaded Integrator-Comb Decimator'
    'Decimation Factor   : 64                                '
    'Differential Delay  : 1                                 '
    'Number of Sections  : 5                                 '
    'Stable              : Yes                               '
    'Linear Phase        : Yes (Type 2)                      '
    '                                                        '
```

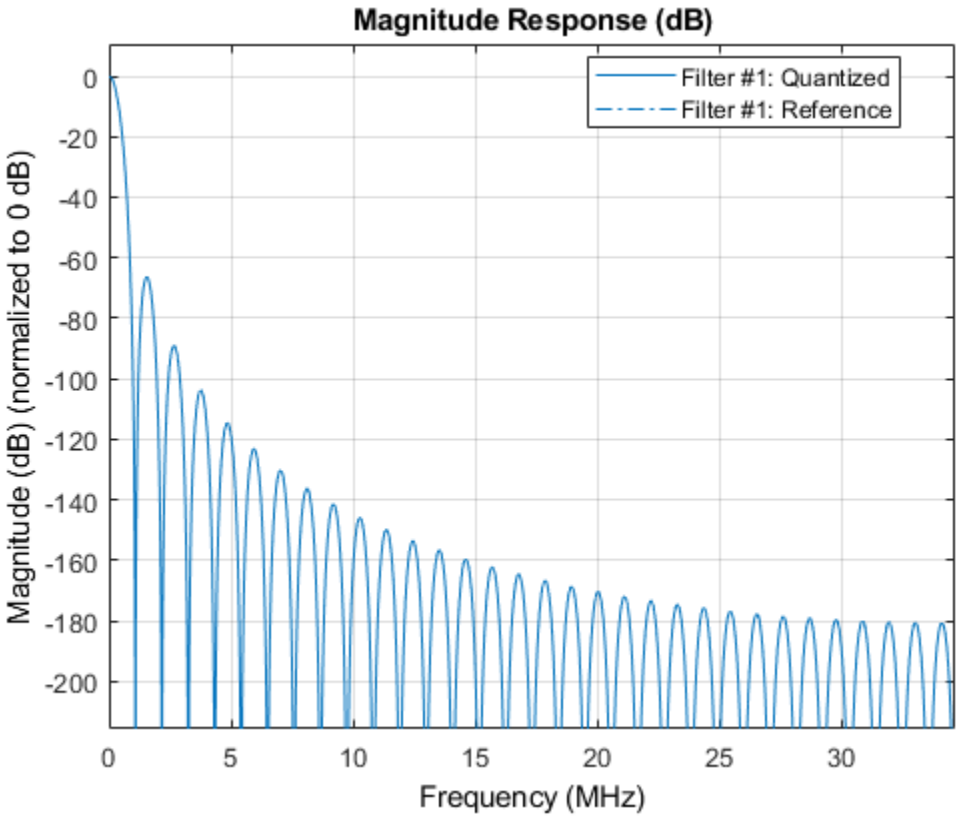
Let's plot and analyze the theoretical magnitude response of the CIC filter which will operate at the input rate of 69.333 MHz.

```
Fs_in = 69.333e6;
fvt = fvtool(cic,'Fs',Fs_in);
fvt.Color = 'White';
```



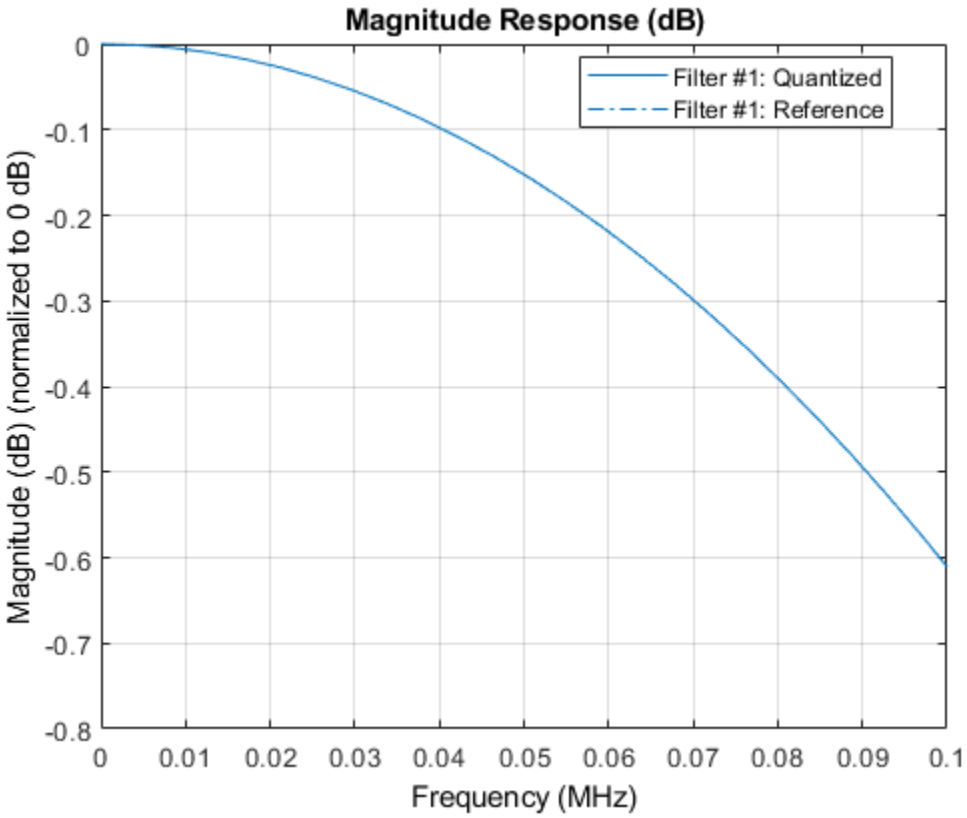
The first thing to note is that the CIC filter has a huge passband gain, which is due to the additions and feedback within the structure. We can normalize the CIC's magnitude response by using the corresponding setting in FVTool. Normalizing the CIC filter response to have 0 dB gain at DC will make it easier to analyze the overlaid filter response of the next stage filter.

```
fvt.NormalizeMagnitudeto1 = 'on';
```



The other thing to note is that zooming in the passband region we see that the CIC has about -0.4 dB of attenuation (droop) at 80 KHz, which is within the bandwidth of interest. A CIC filter is essentially a cascade of boxcar filters and therefore has a sinc-like response which causes the droop. This droop needs to be compensated by the FIR filter in the next stage.

```
axis([0 .1 -0.8 0]);
```



Compensation FIR Decimator

The second stage of our DDC filter chain needs to compensate for the passband droop caused by the CIC and decimate by 2. Since the CIC has a sinc-like response, we can compensate for the droop with a lowpass filter that has an inverse-sinc response in the passband. This filter will operate at 1/64th the input sample rate which is 69.333 MHz, therefore its rate is 1.0833MHz. Instead of designing a lowpass filter with an inverse-sinc passband response from scratch, we'll use a canned function which lets us design a decimator with a CIC Compensation (inverse-sinc) response directly.

```
% Filter specifications
Fs     = 1.0833e6; % Sampling frequency 69.333MHz/64
Apass  = 0.01;     % dB
Astop  = 70;       % dB
Fpass  = 80e3;     % Hz passband-edge frequency
Fstop  = 293e3;    % Hz stopband-edge frequency

% Design decimation filter. D and Nsecs have been defined above as the
% differential delay and number of sections, respectively.
compensator = dsp.CICCompensationDecimator('SampleRate',Fs,...
    'CICRateChangeFactor',R,'CICNumSections',Nsecs,...
    'CICDifferentialDelay',D,'PassbandFrequency',Fpass,...
    'StopbandFrequency',Fstop,'PassbandRipple',Apass,...
    'StopbandAttenuation',Astop);

% Now we have to define the fixed-point attributes of our multirate filter.
% By default, the fixed-point attributes of the accumulator and multipliers
% are set to ensure that full precision arithmetic is used, i.e. no
```

```
% quantization takes place. By default, 16 bits are used to represent the
% filter coefficients. Since that is what we want in this case, no changes
% from default values are required.
```

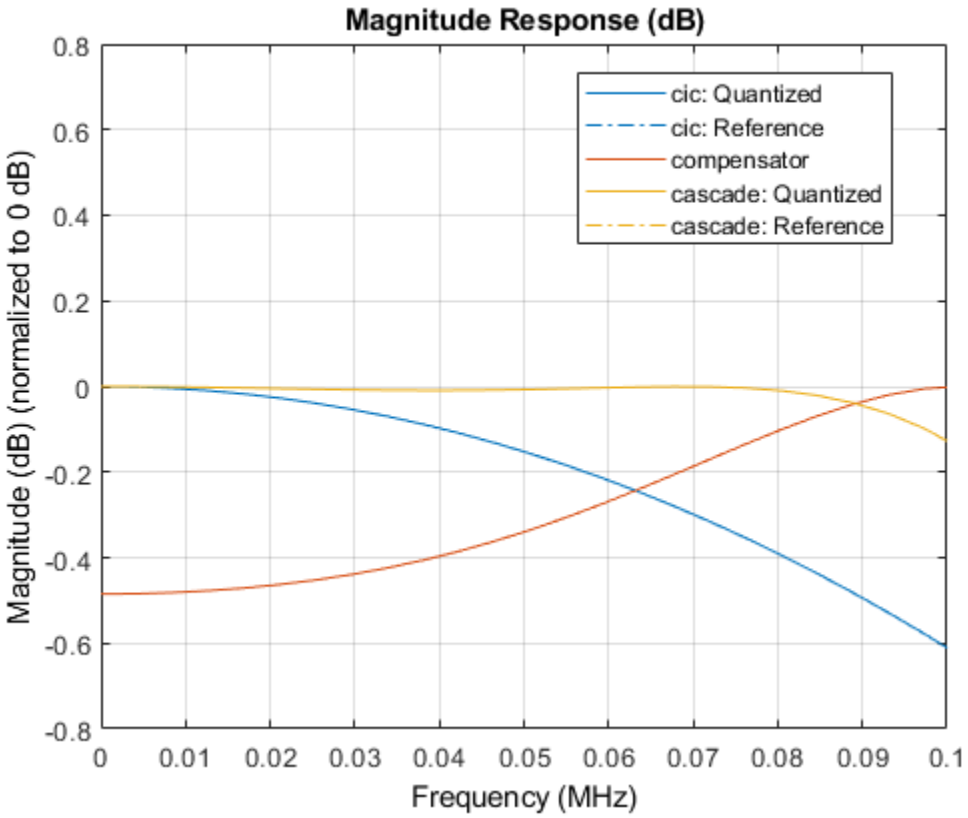
Using the info command we can get a comprehensive report of the FIR compensation filter, including the word lengths of the accumulator and product, which are automatically determined.

```
info(compensator)
```

```
ans =

  10x56 char array

    'Discrete-Time FIR Multirate Filter (real)               '
    '----------------------------------------               '
    'Filter Structure   : Direct-Form FIR Polyphase Decimator'
    'Decimation Factor  : 2                                  '
    'Polyphase Length   : 11                                 '
    'Filter Length      : 21                                 '
    'Stable             : Yes                                '
    'Linear Phase       : Yes (Type 1)                       '
    '                                                        '
    'Arithmetic         : double                             '
```

Cascading the CIC with the inverse sinc filter we can see if we eliminated the passband droop caused by the CIC.

```
cicCompCascade = cascade(cic,compensator);
fvt = fvtool(cic,compensator,cicCompCascade,'Fs',[Fs_in,Fs_in/64,Fs_in]);
fvt.Color = 'White';
fvt.NormalizeMagnitudeto1 = 'on';
axis([0 .1 -0.8 0.8]);
legend(fvt,'cic','compensator','cascade');
```



As we can see in the filter response of the cascade of the two filters, which is between the CIC response and the compensating FIR response, the passband droop has been eliminated.

Third Stage FIR Decimator

As indicated earlier the GSM spectral mask requires an attenuation of 18 dB at 100 KHz. So, for our third and final stage we can try a simple equiripple lowpass filter. Once again we need to quantize the coefficients to 16 bits (default). This filter also needs to decimate by 2.

```
N = 62;       % 63 taps
Fs = 541666;  % 541.666 kHz
Fpass = 80e3;
Fstop = 100e3;

spec = fdesign.decimator(2,'lowpass','N,Fp,Fst',N,Fpass,Fstop,Fs);
% Give more weight to passband
decimator = design(spec,'equiripple','Wpass',2,'SystemObject',true);
```

When defining a multirate filter by default the accumulator word size is determined automatically to maintain full precision. However, because we only have 20 bits for the output let's set the output format to a word length of 20 bits and a fraction length of -12. First, we must change the FullPrecisionOverride property's default value from true to false.

```
decimator.FullPrecisionOverride = false;
decimator.OutputDataType = 'custom';
decimator.RoundingMethod = 'nearest';
decimator.OverflowAction = 'Saturate';
decimator.CustomOutputDataType = numerictype([],20,-12);
```

We can use the info method to view the filter details.

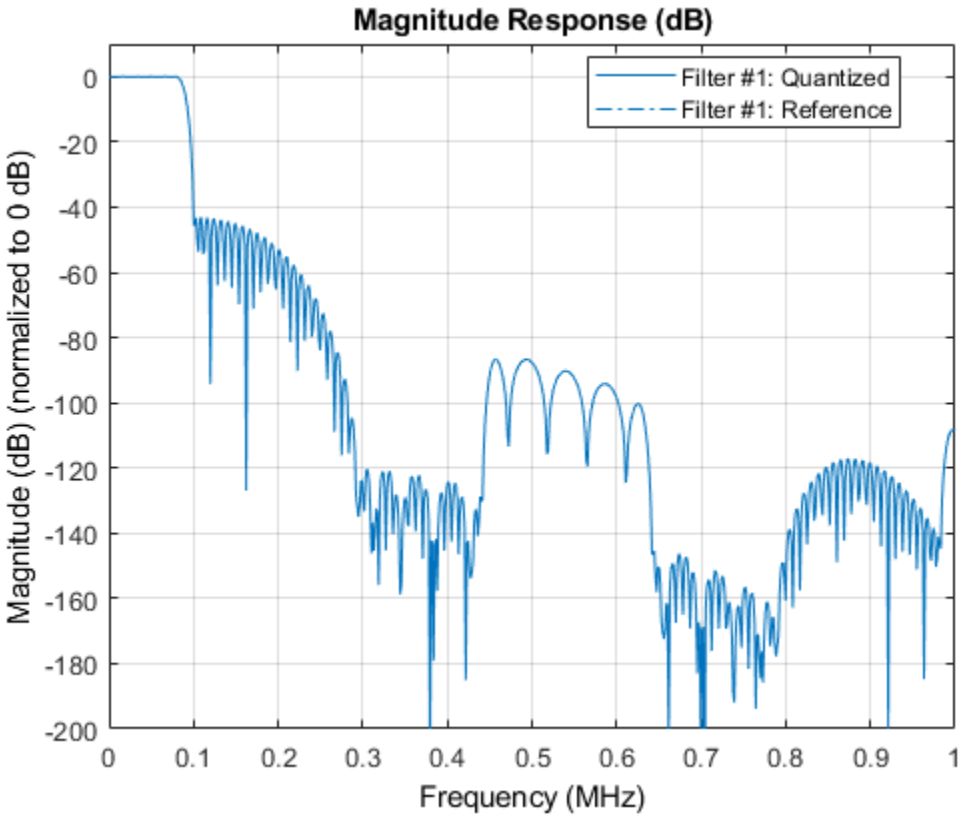
```
info(decimator)
```

```
ans =

  10x56 char array

    'Discrete-Time FIR Multirate Filter (real)               '
    '----------------------------------------               '
    'Filter Structure   : Direct-Form FIR Polyphase Decimator'
    'Decimation Factor  : 2                                  '
    'Polyphase Length   : 32                                 '
    'Filter Length      : 63                                 '
    'Stable             : Yes                                '
    'Linear Phase       : Yes (Type 1)                       '
    '                                                        '
    'Arithmetic         : double                             '
```

Multistage Multirate DDC Filter Chain

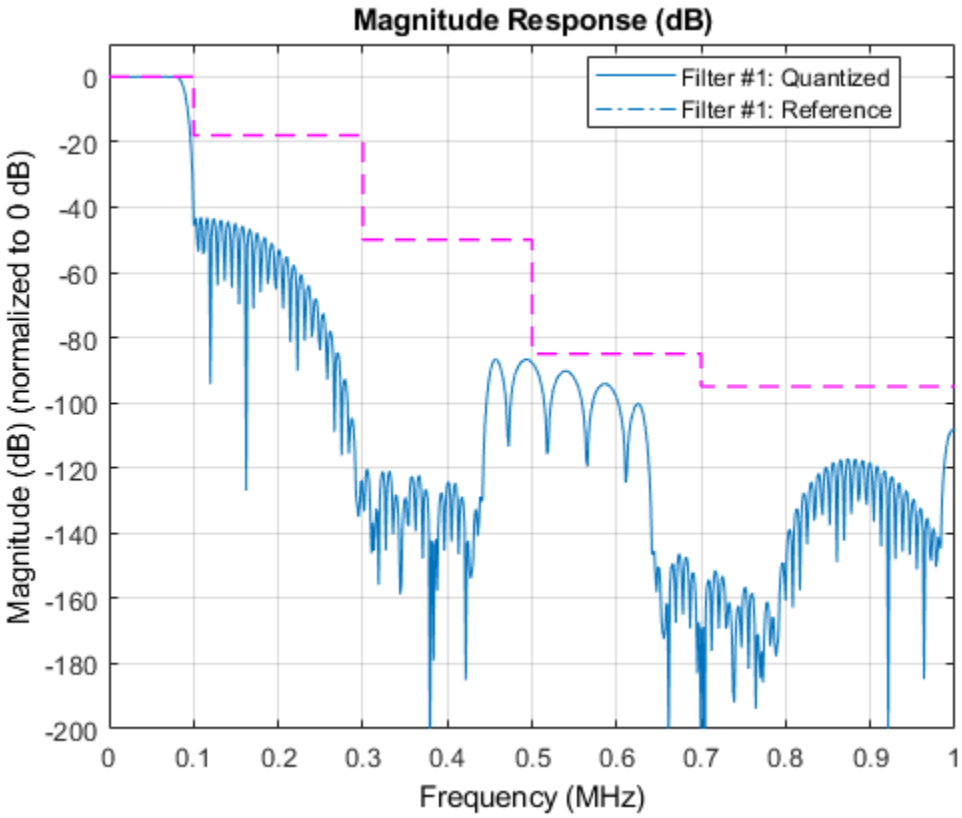
Now that we have designed and quantized the three filters, we can get the overall filter response by cascading the normalized CIC and the two FIR filters. Again, we're using normalized magnitude to ensure that the cascaded filter response is normalized to 0 dB.

```
ddc = cascade(cic,compensator,decimator);
fvt = fvtool(ddc,'Fs',Fs_in);
fvt.Color = 'White';
fvt.NormalizeMagnitudeto1 = 'on';
fvt.NumberofPoints = 8192*3;
axis([0 1 -200 10]);  % Zoom-in
```



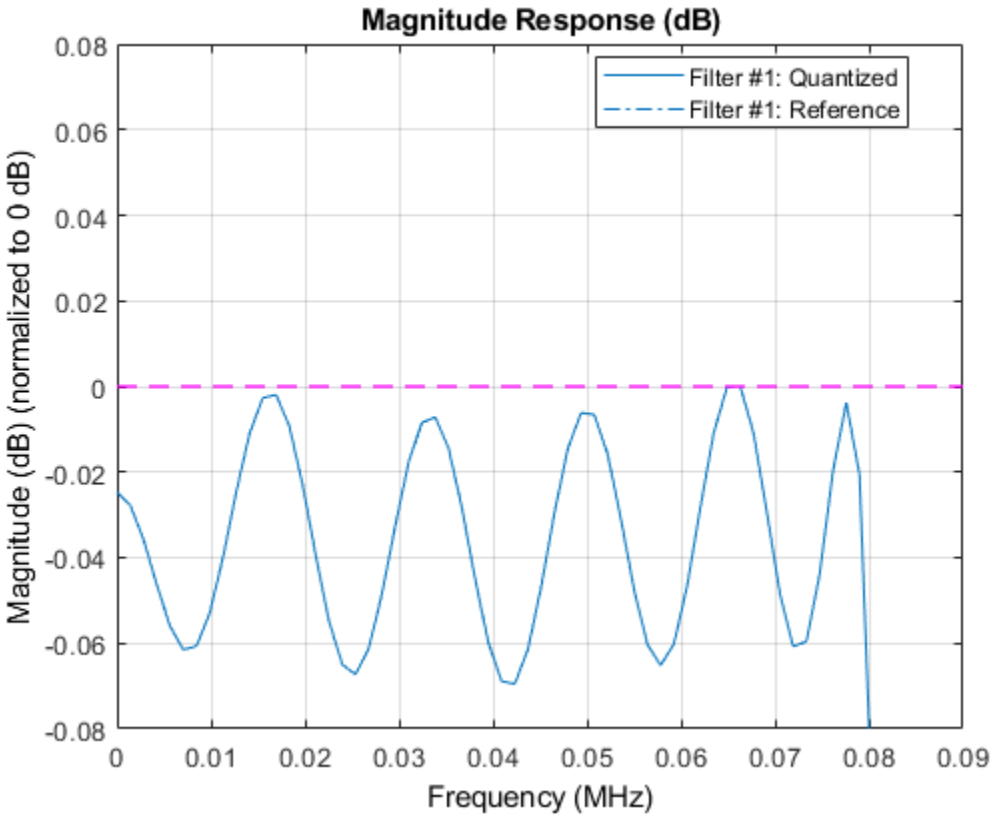
To see if the overall filter response meets the GSM specifications, we can overlay the GSM spectral mask on the filter response.

```
drawgsmmask;
```



We can see that our overall filter response is within the constraints of the GSM spectral mask. We also need to ensure that the passband ripple meets the requirement that it is less than 0.1 dB peak-to-peak. We can verify this by zooming in using the axis command.

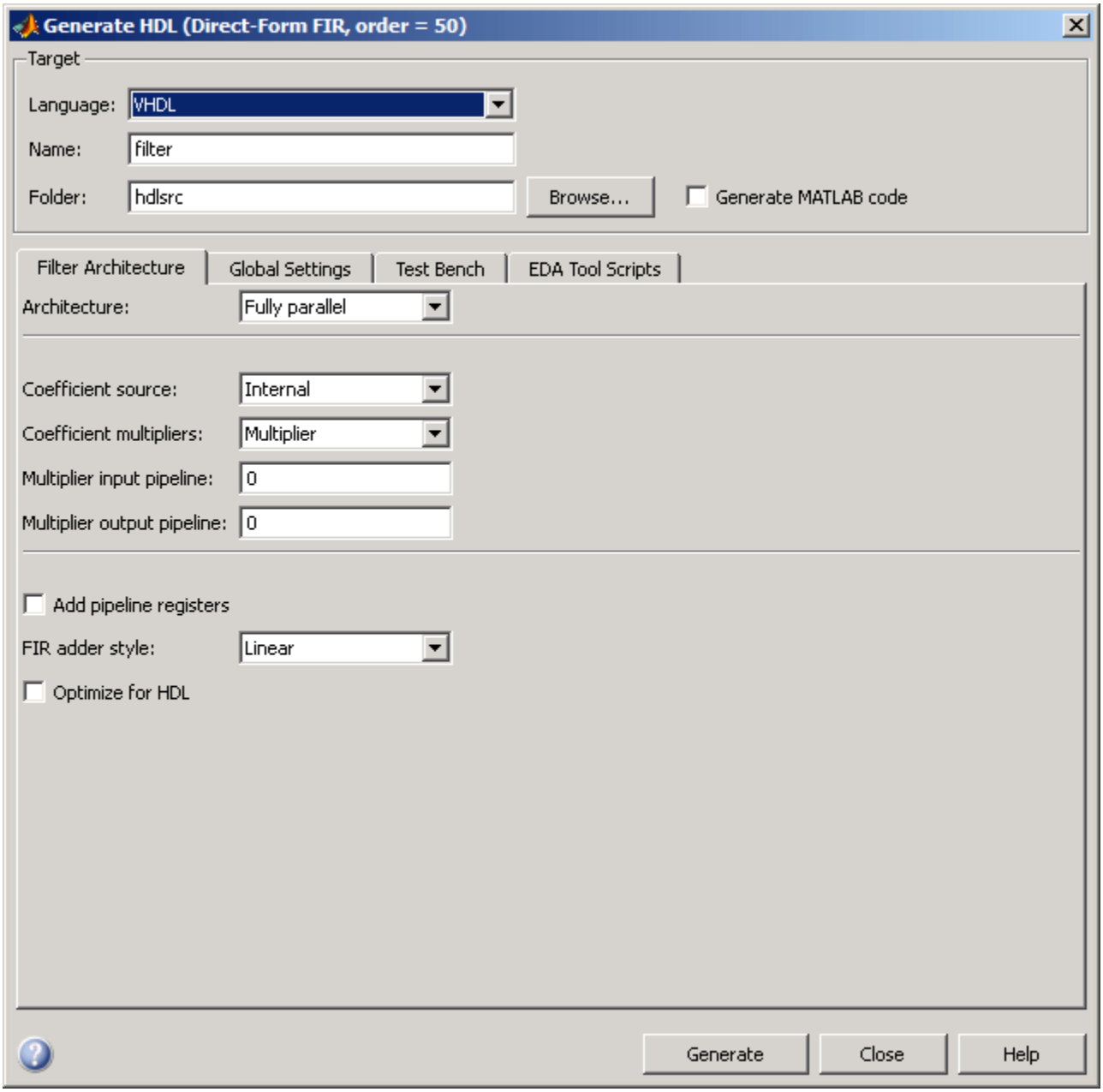
```
axis([0 .09 -0.08 0.08]);
```



Indeed the passband ripple is well below the 0.1 dB peak-to-peak GSM requirement.

Generate VHDL Code

Filter Designer also supports the generation of HDL code from the dialog shown below.



From Filter Designer as well as the command line you can generate VHDL or Verilog code as well as test benches in VHDL or Verilog files. Also, you have the ability to customize your generated HDL code by specifying many options to meet your coding standards and guidelines.

However, here we will use the command line functionality to generate the HDL code.

Now that we have our fixed-point, three-stage, multirate filter meeting the specs we are ready to generate HDL code.

Cascade of CIC and two FIR filters and generate VHDL.

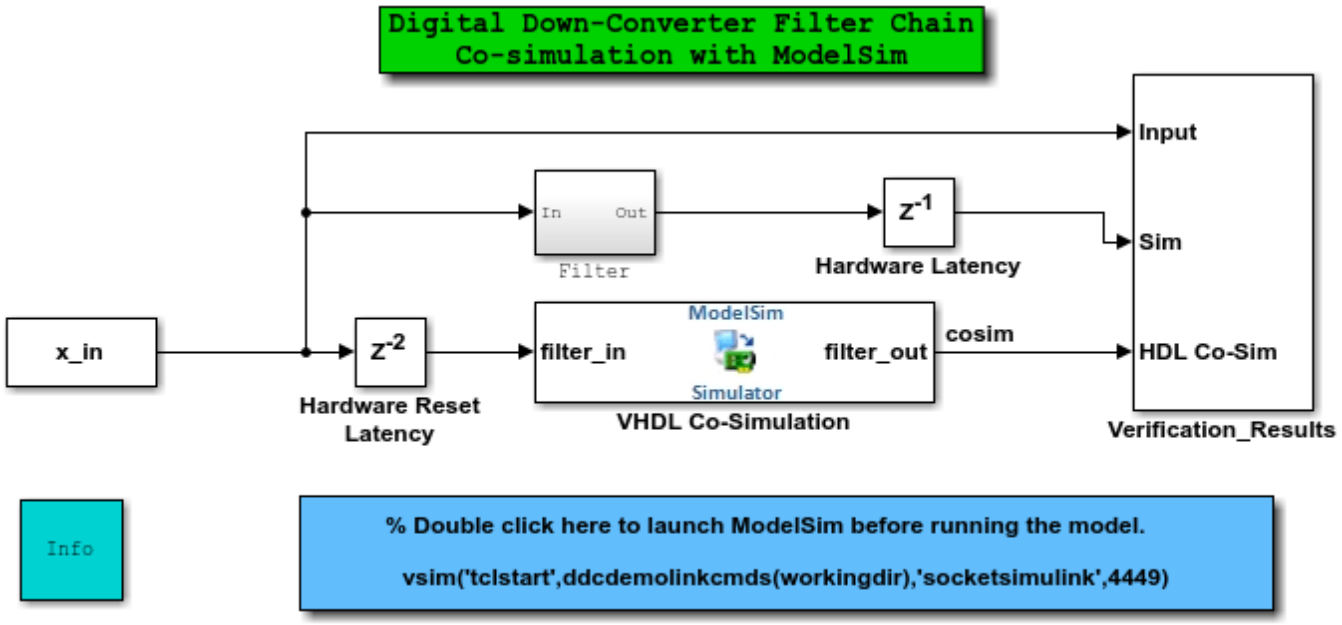
To avoid quantizing the fixed-point data coming from the mixer, which has a word length of 20 bits and a fraction length of 18 bits, (S20,18), we'll set the input word length and fraction length of the CIC to the same values, S20,18.

```
%hcas = cascade(hcic,hcfir,hpfir);
workingdir = tempname;
inT = numerictype(1,20,18);
generatehdl(ddc,'InputDataType', inT,...
    'Name','filter','TargetLanguage','VHDL',...
    'TargetDirectory',fullfile(workingdir,'hdlsrc'));

### Starting VHDL code generation process for filter: filter
### Cascade stage # 1
### Starting VHDL code generation process for filter: filter_stage1
### Generating: <a href="matlab:edit('C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\13\tp9a7149f9_6237_41
### Starting generation of filter_stage1 VHDL entity
### Starting generation of filter_stage1 VHDL architecture
### Section # 1 : Integrator
### Section # 2 : Integrator
### Section # 3 : Integrator
### Section # 4 : Integrator
### Section # 5 : Integrator
### Section # 6 : Comb
### Section # 7 : Comb
### Section # 8 : Comb
### Section # 9 : Comb
### Section # 10 : Comb
### Successful completion of VHDL code generation process for filter: filter_stage1
### Cascade stage # 2
### Starting VHDL code generation process for filter: filter_stage2
### Generating: <a href="matlab:edit('C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\13\tp9a7149f9_6237_41
### Starting generation of filter_stage2 VHDL entity
### Starting generation of filter_stage2 VHDL architecture
### Successful completion of VHDL code generation process for filter: filter_stage2
### Cascade stage # 3
### Starting VHDL code generation process for filter: filter_stage3
### Generating: <a href="matlab:edit('C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\13\tp9a7149f9_6237_41
### Starting generation of filter_stage3 VHDL entity
### Starting generation of filter_stage3 VHDL architecture
### Successful completion of VHDL code generation process for filter: filter_stage3
### Generating: <a href="matlab:edit('C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\13\tp9a7149f9_6237_41
### Starting generation of filter VHDL entity
### Starting generation of filter VHDL architecture
### Successful completion of VHDL code generation process for filter: filter
### HDL latency is 2 samples
```

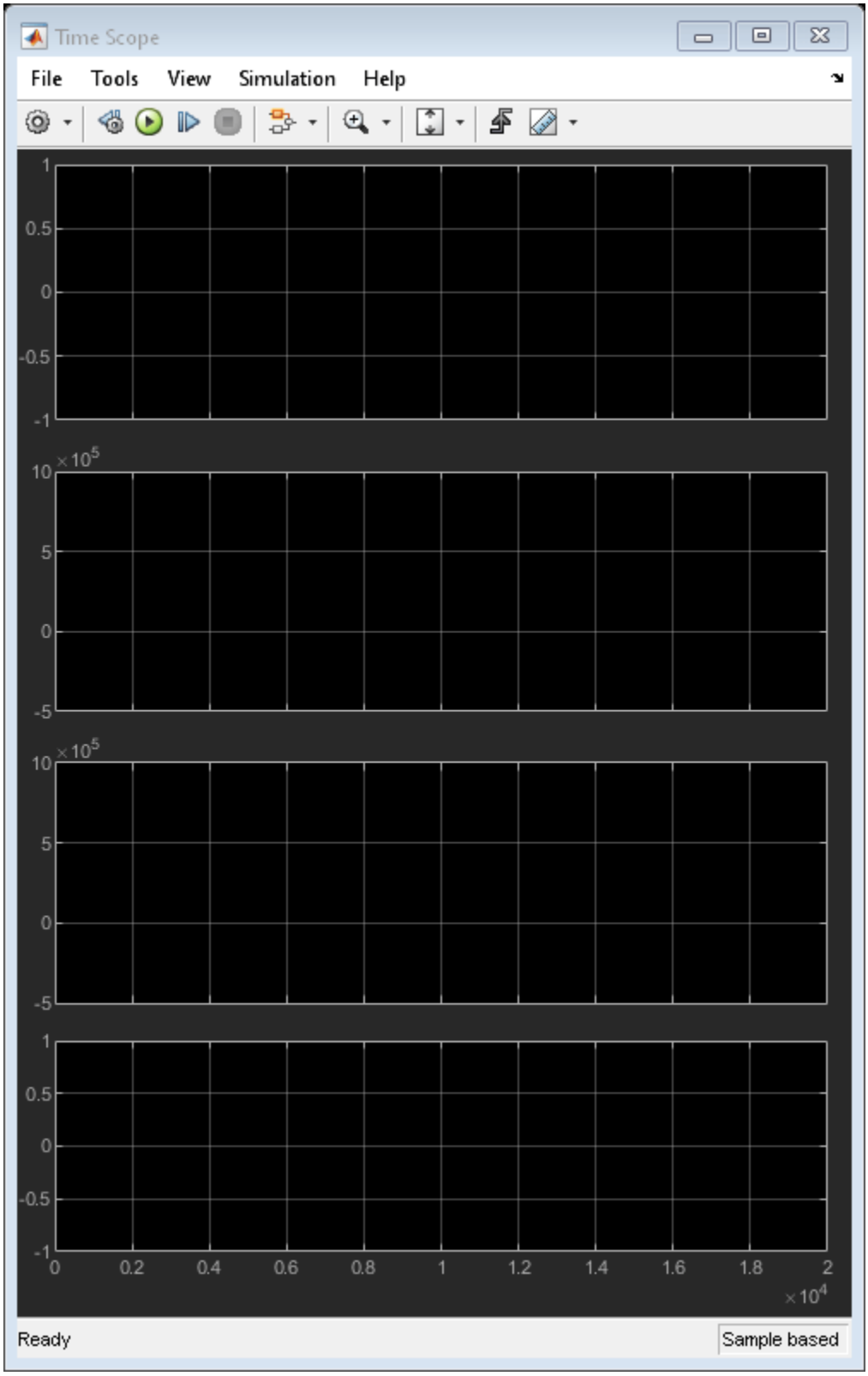
HDL Co-simulation with ModelSim in Simulink

To verify that the generated HDL code is producing the same results as our Simulink model, we'll use HDL Verifier MS to co-simulate our HDL code in Simulink. We have a pre-built Simulink model that includes two signal paths. One signal path produces Simulink's behavioral model results of the three-stage, multirate filter. The other path produces the results of simulating, with ModelSim®, the VHDL code we generated.

```
open_system('ddcfilterchaindemo_cosim');
```

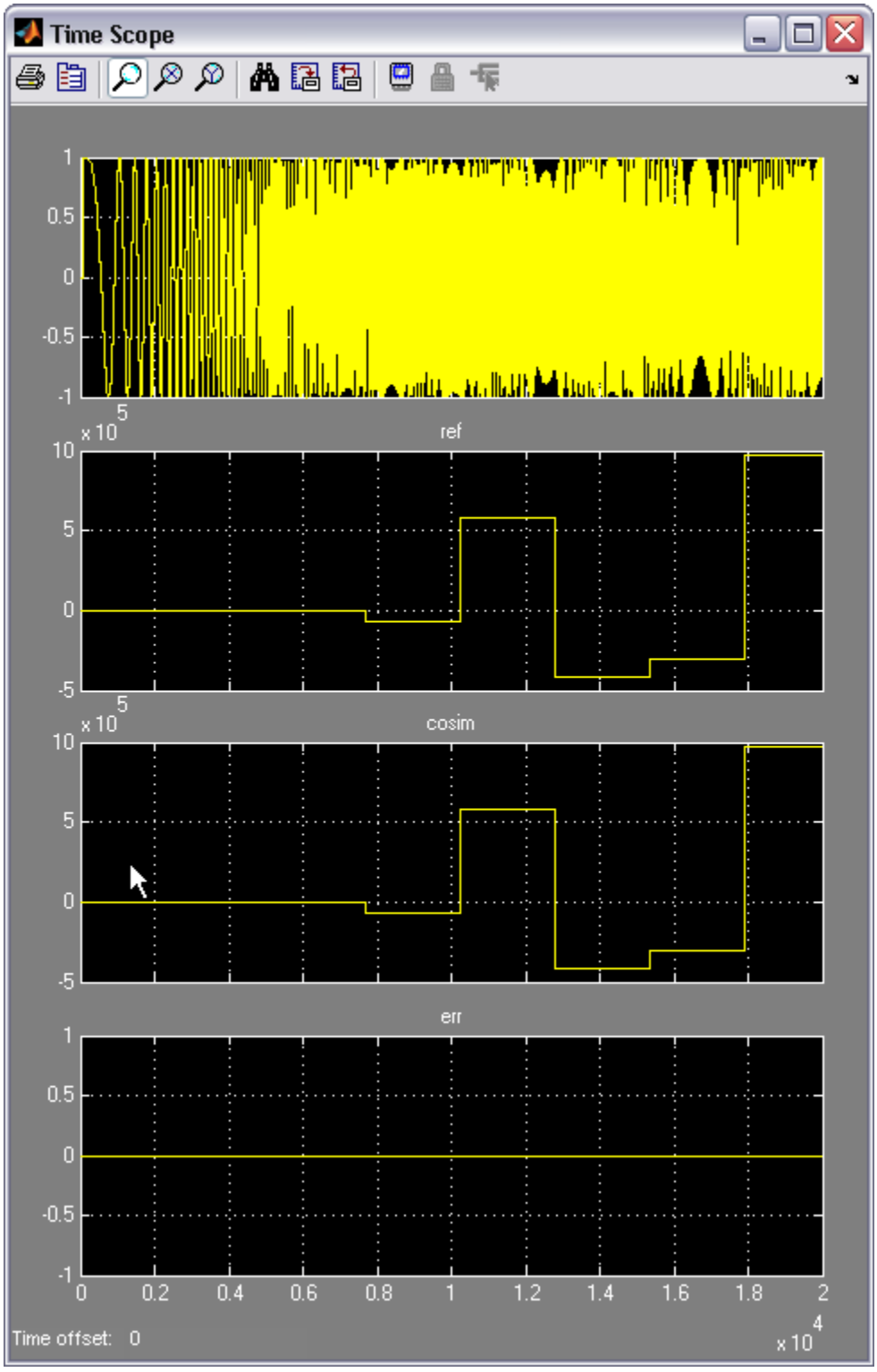


Copyright 1999-2016 The MathWorks, Inc.

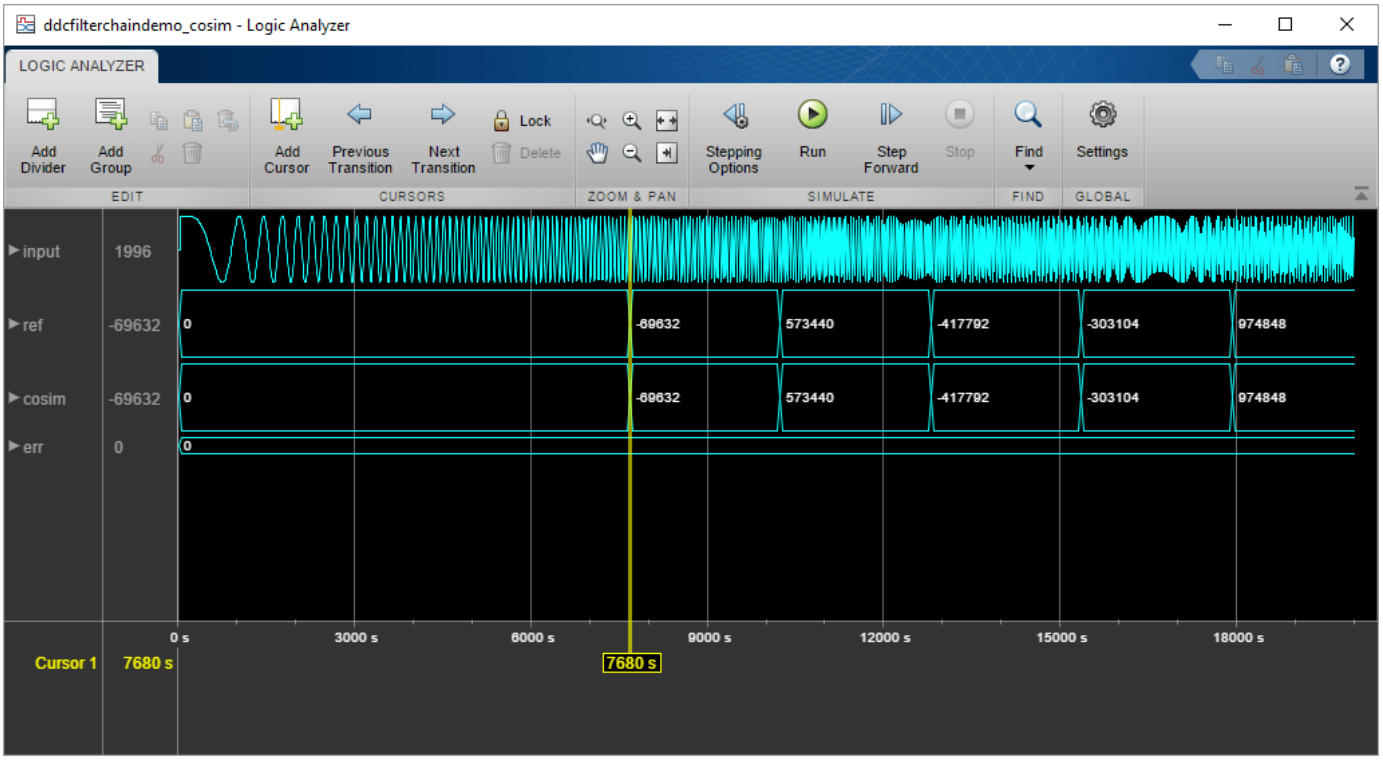

Time Scope
File Tools View Simulation Help
1
0.5
0
-0.5
-1
×10^5
10
5
0
-5
×10^5
10
5
0
-5
1
0.5
0
-0.5
-1
0 0.2 0.4 0.6 0.8 1 1.2 1.4 1.6 1.8 2
×10^4
Ready
Sample based

Start ModelSim by double clicking on the button in the Simulink model. Note that ModelSim must be installed and on the system path. ModelSim will automatically compile the HDL code, initialize the simulation and open the Wave viewer.

When ModelSim is ready run the Simulink model. This will execute co-simulation with ModelSim and automatically open a Time Scope to view the results.



Use the Logic Analyzer to view the results.



Verifying Results

The trace on the top is the excitation chirp signal. The next signal labeled "ref" is the reference signal produced by the Simulink behavioral model of the three-stage multirate filter. The bottom trace labeled "cosim" on the scope is of the ModelSim simulation results of the generated HDL code of the three-stage multirate filter. The last trace shows the error between Simulink's behavioral model results and ModelSim's simulation of the HDL code.

Summary

We used several MathWorks™ products to design and analyze a three-stage, multirate, fixed-point filter chain of a DDC for a GSM application. Then we generated HDL code to implement the filter and verified the generated code by comparing Simulink's behavioral model with HDL code simulated in ModelSim via HDL Verifier MS.

HDL Implementation of a Digital Down-Converter for LTE

This example shows how to design a digital down-converter (DDC) for radio communication applications such as LTE, and generate HDL code.

Introduction

DDCs are widely used in digital communication receivers to convert radio frequency (RF) or intermediate frequency (IF) signals to baseband. The DDC operation shifts the signal to a lower frequency and reduces its sampling rate to facilitate subsequent processing stages. The DDC in this example performs complex frequency translation followed by sample rate conversion using a four-stage filter chain. The example starts by designing the DDC with DSP System Toolbox™ functions in floating point. Then, each stage is converted to fixed point, and used in a Simulink® model that generates synthesizable HDL code. The example uses these two test signals to demonstrate and verify the DDC operation:

- A sinusoid that is modulated onto a 32 MHz IF carrier.
- An LTE downlink signal with a bandwidth of 1.4 MHz that is modulated onto a 32 MHz IF carrier.

The example compares the signal quality at the output of the floating-point DDC with the signal quality at the output of the fixed-point DDC.

Finally, the example presents an implementation of the filter chain for FPGAs, and synthesis results.

This example uses `DDCTestUtils`, a helper class that contains functions for generating stimulus and analyzing the DDC output. For more information, see the `DDCTestUtils.m` file.

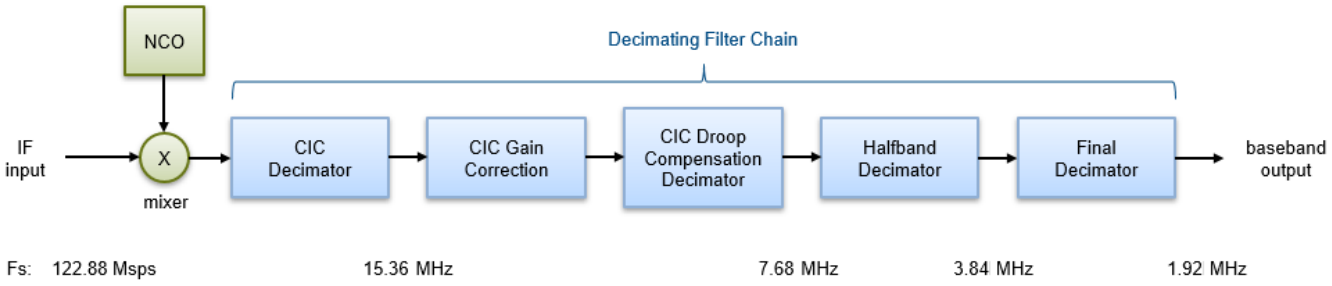
DDC Structure

The DDC consists of a numerically controlled oscillator (NCO), mixer, and decimating filter chain. The filter chain consists of a CIC decimator, CIC gain correction, a CIC compensation decimator (FIR), a halfband FIR decimator, and a final FIR decimator.

The overall response of the filter chain is equivalent to that of a single decimation filter with the same specification. However, splitting the filter into multiple decimation stages results in a more efficient design that uses fewer hardware resources.

The cascade integrator-comb (CIC) decimator provides a large initial decimation factor, which enables subsequent filters to work at lower rates. The CIC compensation decimator improves the spectral response by compensating for the CIC droop while decimating by two. The halfband is an intermediate decimator, and the final decimator implements the precise `Fpass` and `Fstop` characteristics of the DDC. The lower sampling rates near the end of the chain mean the later filters can optimize resource use by sharing multipliers.

This figure shows a block diagram of the DDC.



The sample rate of the input to the DDC is 122.88 Msps, and the output sample rate is 1.92 Msps. These rates give an overall decimation factor of 64. LTE receivers use 1.92 Msps as the typical sampling rate for cell search and master information block (MIB) recovery. The DDC filters are designed to suit this application. The DDC is optimized to run at a clock rate of 122.88 MHz.

DDC Design

This section explains how to design the DDC using floating-point operations and filter-design functions in MATLAB®.

DDC Parameters

This example designs the DDC filter characteristics to meet these desired response values for the given input sampling rate and carrier frequency.

```
FsIn = 122.88e6;    % Sampling rate of DDC input
FsOut = 1.92e6;     % Sampling rate of DDC output
Fc = 32e6;          % Carrier frequency
Fpass = 540e3;      % Passband frequency, equivalent to 36x15kHz LTE subcarriers
Fstop = 700e3;      % Stopband frequency
Ap = 0.1;           % Passband ripple
Ast = 60;           % Stopband attenuation
```

CIC Decimator

The first filter stage is a CIC decimator because of its ability to efficiently implement a large decimation factor. The response of a CIC filter is similar to a cascade of moving average filters, but a CIC filter uses no multiplication or division. As a result,the CIC filter has a large DC gain.

```
cicParams.DecimationFactor = 8;
cicParams.DifferentialDelay = 1;
cicParams.NumSections = 3;
cicParams.FsOut = FsIn/cicParams.DecimationFactor;

cicFilt = dsp.CICDecimator(cicParams.DecimationFactor, ...
    cicParams.DifferentialDelay,cicParams.NumSections) %#ok<*NOPTS>
cicGain = gain(cicFilt)


cicFilt =
```

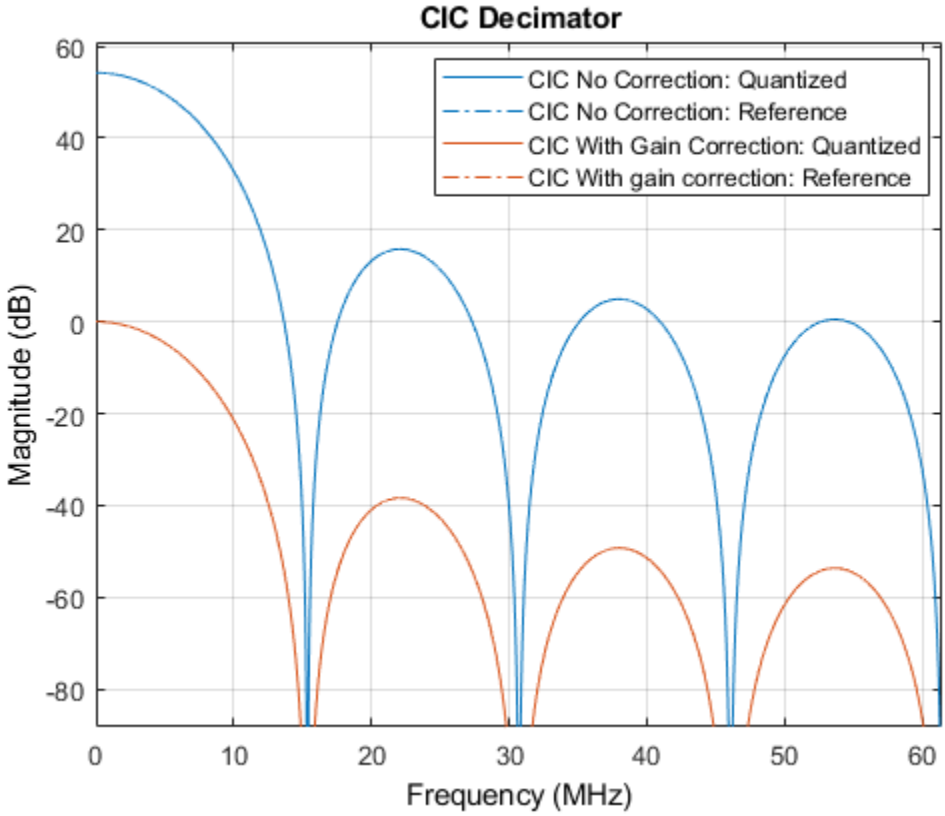
```
  dsp.CICDecimator with properties:

      DecimationFactor: 8
     DifferentialDelay: 1
           NumSections: 3
    FixedPointDataType: 'Full precision'


cicGain =

   512
```

Because the CIC gain is a power of two, a hardware implementation can easily correct for the gain factor by using a shift operation. For analysis purposes, the example represents the gain correction in MATLAB with a one-tap `dsp.FIRFilter` System object™.

```
cicGainCorr = dsp.FIRFilter('Numerator',1/cicGain)


cicGainCorr =

  dsp.FIRFilter with properties:

            Structure: 'Direct form'
      NumeratorSource: 'Property'
            Numerator: 0.0020
    InitialConditions: 0

  Use get to show all properties
```

Display the magnitude response of the CIC filter with and without gain correction by using `fvtool`. For analysis, combine the CIC filter and the gain correction filter into a `dsp.FilterCascade` System object. CIC filters use fixed-point arithmetic internally, so `fvtool` plots both the quantized and unquantized responses.

```
ddcPlots.cicDecim = fvtool(...
    cicFilt, ...
    dsp.FilterCascade(cicFilt,cicGainCorr), ...
    'Fs',[FsIn,FsIn]);
DDCTestUtils.setPlotNameAndTitle('CIC Decimator');
legend(...
    'CIC No Correction: Quantized', ...
    'CIC No Correction: Reference', ...
    'CIC With Gain Correction: Quantized', ...
    'CIC With gain correction: Reference');
```



CIC Droop Compensation Filter

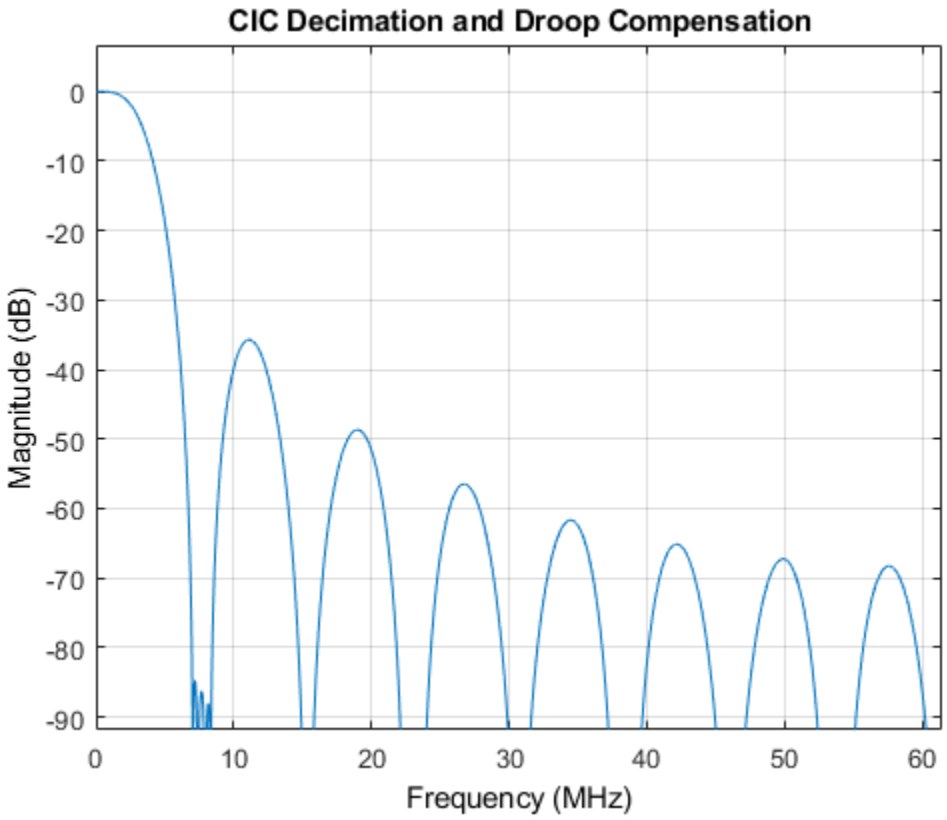
Because the magnitude response of the CIC filter has a significant *droop* within the passband region, the example uses a FIR-based droop compensation filter to flatten the passband response. The droop compensator has the same properties as the CIC decimator. This filter implements decimation by a factor of two, so you must also specify bandlimiting characteristics for the filter. Use the `design` function to return a filter System object with the specified characteristics.

```
compParams.R = 2;                                % CIC compensation decimation factor
compParams.Fpass = Fstop;                        % CIC compensation passband frequency
compParams.FsOut = cicParams.FsOut/compParams.R; % New sampling rate
compParams.Fstop = compParams.FsOut - Fstop;     % CIC compensation stopband frequency
compParams.Ap = Ap;                              % Same passband ripple as overall filter
compParams.Ast = Ast;                            % Same stopband attenuation as overall filter

compSpec = fdesign.decimator(compParams.R,'ciccomp', ...
    cicParams.DifferentialDelay, ...
    cicParams.NumSections, ...
    cicParams.DecimationFactor, ...
    'Fp,Fst,Ap,Ast', ...
    compParams.Fpass,compParams.Fstop,compParams.Ap,compParams.Ast, ...
    cicParams.FsOut);
compFilt = design(compSpec,'SystemObject',true)


compFilt =
```

```
  dsp.FIRDecimator with properties:

    DecimationFactor: 2
     NumeratorSource: 'Property'
           Numerator: [-0.0398 -0.0126 0.2901 0.5258 0.2901 -0.0126 -0.0398]
           Structure: 'Direct form'

  Use get to show all properties
```

Plot the combined response of the CIC filter (with gain correction) and droop compensation.

```
ddcPlots.cicComp = fvtool(...
    dsp.FilterCascade(cicFilt,cicGainCorr,compFilt), ...
    'Fs',FsIn,'Legend','off');
DDCTestUtils.setPlotNameAndTitle('CIC Decimation and Droop Compensation');
```



Halfband Decimator

The halfband filter provides efficient decimation by two. Halfband filters are efficient because approximately half of their coefficients are equal to zero, and those multipliers are excluded from the hardware implementation.

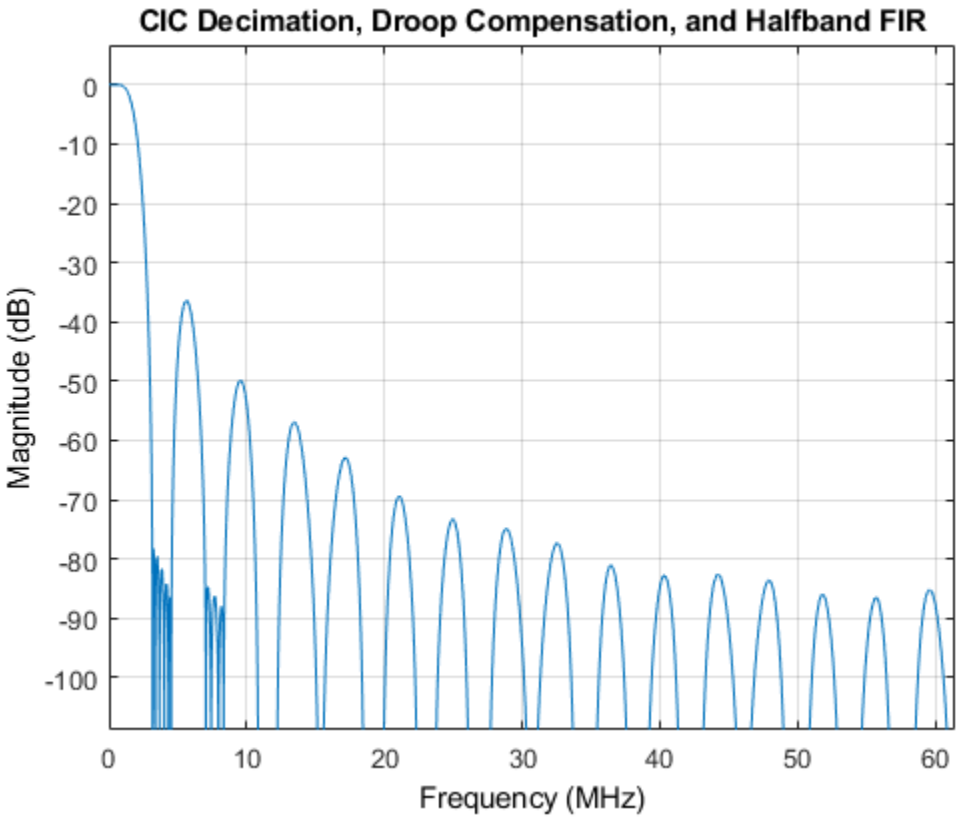
```
hbParams.FsOut = compParams.FsOut/2;
hbParams.TransitionWidth = hbParams.FsOut - 2*Fstop;
hbParams.StopbandAttenuation = Ast;

hbSpec = fdesign.decimator(2,'halfband',...
```

```
    'Tw,Ast', ...
    hbParams.TransitionWidth, ...
    hbParams.StopbandAttenuation, ...
    compParams.FsOut);
hbFilt = design(hbSpec,'SystemObject',true)
```

```
hbFilt =

  dsp.FIRDecimator with properties:

    DecimationFactor: 2
     NumeratorSource: 'Property'
           Numerator: [0.0089 0 -0.0565 0 0.2977 0.5000 0.2977 0 -0.0565 ... ]
           Structure: 'Direct form'

  Use get to show all properties
```

Plot the response of the DDC up to the halfband filter output.

```
ddcPlots.halfbandFIR = fvtool(...
    dsp.FilterCascade(cicFilt,cicGainCorr,compFilt,hbFilt), ...
    'Fs',FsIn,'Legend','off');
DDCTestUtils.setPlotNameAndTitle('CIC Decimation, Droop Compensation, and Halfband FIR');
```



Final FIR Decimator

The final FIR implements the detailed passband and stopband characteristics of the DDC. This filter has more coefficients than the earlier FIR filters, but because it operates at a lower sampling rate it can use resource sharing for an efficient hardware implementation.

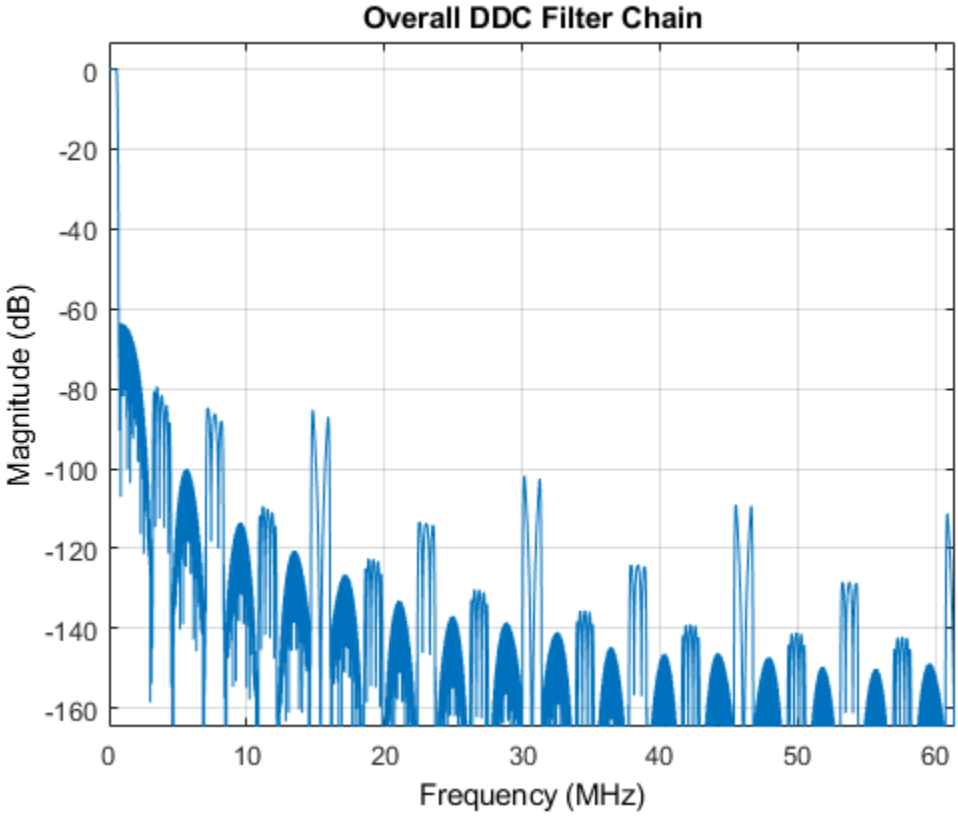
Add 3 dB of headroom to the stopband attenuation so that the DDC still meets the specification after fixed-point quantization. This value was found empirically by using `fvtool`.

```
finalSpec = fdesign.decimator(2,'lowpass', ...
    'Fp,Fst,Ap,Ast',Fpass,Fstop,Ap,Ast+3,hbParams.FsOut);
finalFilt = design(finalSpec,'equiripple','SystemObject',true)
```

```
finalFilt =

  dsp.FIRDecimator with properties:

    DecimationFactor: 2
     NumeratorSource: 'Property'
           Numerator: [9.3365e-04 0.0013 9.3466e-04 -5.3189e-04 -0.0022 ... ]
           Structure: 'Direct form'

  Use get to show all properties
```

Visualize the overall magnitude response of the DDC.

```
ddcFilterChain = dsp.FilterCascade(cicFilt,cicGainCorr, ...
    compFilt,hbFilt,finalFilt);
ddcPlots.overallResponse = fvtool(ddcFilterChain, ...
    'Fs',FsIn,'Legend','off');
DDCTestUtils.setPlotNameAndTitle('Overall DDC Filter Chain');
```



Fixed-Point Conversion

The frequency response of the floating-point DDC filter chain now meets the specification. Next, quantize each filter stage to use fixed-point types and analyze them to confirm that the filter chain still meets the specification.

Filter Quantization

This example uses 16-bit coefficients, which are sufficient to meet the specification. Using fewer than 18 bits for the coefficients minimizes the number of DSP blocks that are required for an FPGA implementation. The input to the DDC filter chain is 16-bit data with 15 fractional bits. The filter outputs are 18-bit values, which provide extra headroom and precision in the intermediate signals.

For the CIC decimator, choosing the `'Minimum section word lengths'` fixed-point data type option automatically optimizes the internal wordlengths based on the output wordlength and other CIC parameters.

```
cicFilt.FixedPointDataType = 'Minimum section word lengths';
cicFilt.OutputWordLength = 18;
```

Configure the fixed-point properties of the gain correction and FIR-based System objects. The object uses the default `RoundingMethod` and `OverflowAction` property values (`'Floor'` and `'Wrap'` respectively).

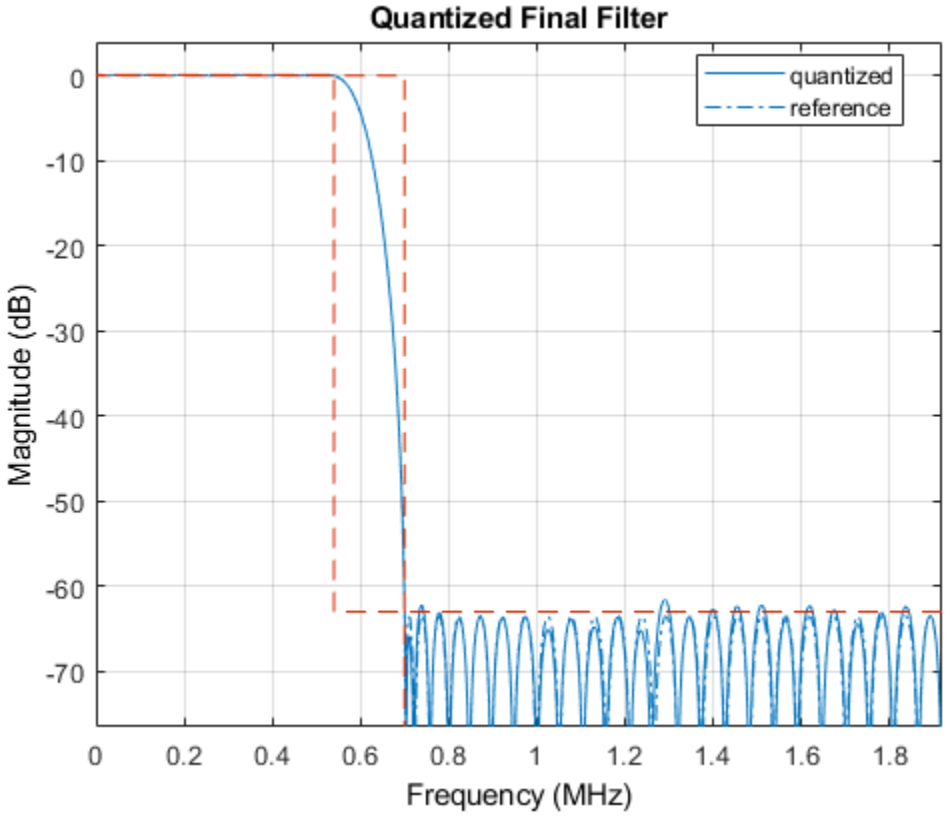
```
% CIC Gain Correction
cicGainCorr.FullPrecisionOverride = false;
cicGainCorr.CoefficientsDataType = 'Custom';
```

```
cicGainCorr.CustomCoefficientsDataType = numerictype(fi(cicGainCorr.Numerator,1,16));
cicGainCorr.OutputDataType = 'Custom';
cicGainCorr.CustomOutputDataType = numerictype(1,18,16);

% CIC Droop Compensation
compFilt.FullPrecisionOverride = false;
compFilt.CoefficientsDataType = 'Custom';
compFilt.CustomCoefficientsDataType = numerictype([],16,15);
compFilt.ProductDataType = 'Full precision';
compFilt.AccumulatorDataType = 'Full precision';
compFilt.OutputDataType = 'Custom';
compFilt.CustomOutputDataType = numerictype([],18,16);

% Halfband
hbFilt.FullPrecisionOverride = false;
hbFilt.CoefficientsDataType = 'Custom';
hbFilt.CustomCoefficientsDataType = numerictype([],16,15);
hbFilt.ProductDataType = 'Full precision';
hbFilt.AccumulatorDataType = 'Full precision';
hbFilt.OutputDataType = 'Custom';
hbFilt.CustomOutputDataType = numerictype([],18,16);

% FIR
finalFilt.FullPrecisionOverride = false;
finalFilt.CoefficientsDataType = 'Custom';
finalFilt.CustomCoefficientsDataType = numerictype([],16,15);
finalFilt.ProductDataType = 'Full precision';
finalFilt.AccumulatorDataType = 'Full precision';
finalFilt.OutputDataType = 'Custom';
finalFilt.CustomOutputDataType = numerictype([],18,16);
```

Fixed-Point Analysis

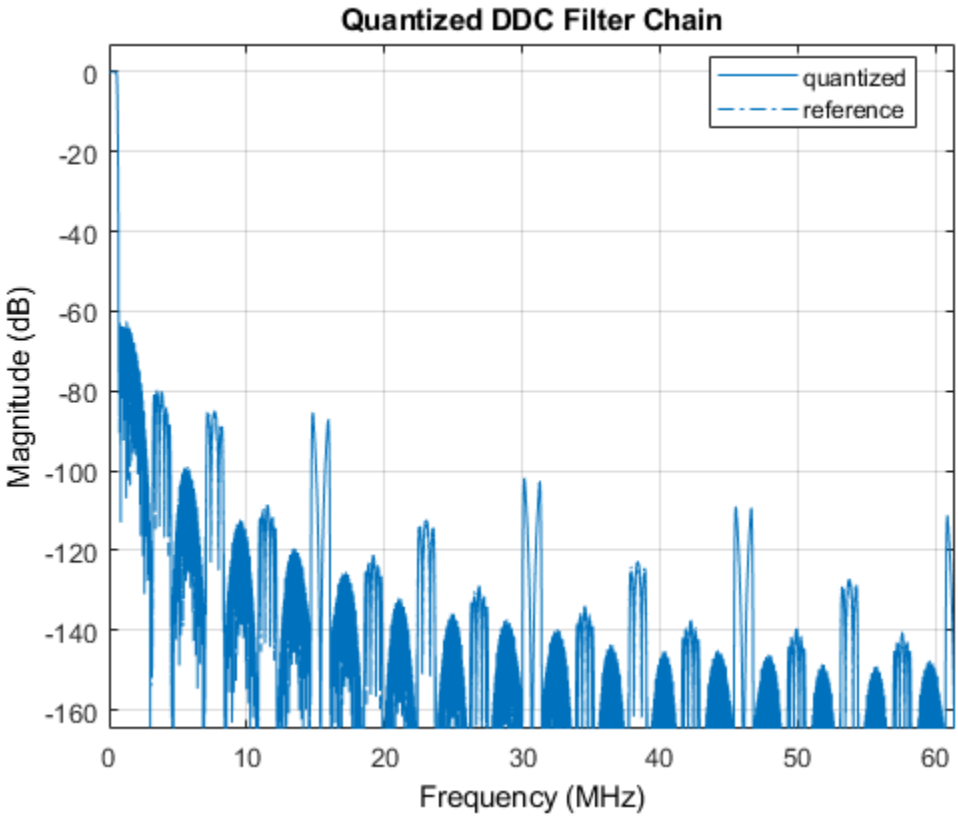
Inspect the quantization effects with `fvtool`. You can analyze the filters individually or in a cascade. `fvtool` shows the quantized and unquantized (reference) responses overlayed. For example, this figure shows the effect of quantizing the final FIR filter stage.

```
ddcPlots.quantizedFIR = fvtool(finalFilt, ...
    'Fs',hbParams.FsOut,'arithmetic','fixed');
DDCTestUtils.setPlotNameAndTitle('Quantized Final Filter');
legend('quantized','reference');
```



Redefine the `ddcFilterChain` cascade object to include the fixed-point properties of the individual filters. Then, use `fvtool` to analyze the entire filter chain and confirm that the quantized DDC still meets the specification.

```
ddcFilterChain = dsp.FilterCascade(cicFilt, ...
    cicGainCorr,compFilt,hbFilt,finalFilt);
ddcPlots.quantizedDDCResponse = fvtool(ddcFilterChain, ...
    'Fs',FsIn,'Arithmetic','fixed');
DDCTestUtils.setPlotNameAndTitle('Quantized DDC Filter Chain');
legend('quantized','reference');
```



HDL-Optimized Simulink Model

The next step in the design flow is to implement the DDC in Simulink using blocks that support HDL code generation.

Model Configuration

The model relies on variables in the MATLAB workspace to configure the blocks and settings. It uses the same filter chain variables defined earlier in the example. Next, define the NCO characteristics and the input signal. The example uses these characteristics to configure the NCO block.

Specify the desired frequency resolution. Calculate the number of accumulator bits that are required to achieve the desired resolution, and then define the number of quantized accumulator bits. The NCO uses the quantized output of the accumulator to address the sine lookup table. Also compute the phase increment the NCO must use to generate the specified carrier frequency. The NCO applies phase dither to those accumulator bits that are removed during quantization.

```
nco.Fd = 1;
nco.AccWL = nextpow2(FsIn/nco.Fd)+1;
nco.QuantAccWL = 12;
nco.PhaseInc = round((-Fc*2^nco.AccWL)/FsIn);
nco.NumDitherBits = nco.AccWL-nco.QuantAccWL;
```

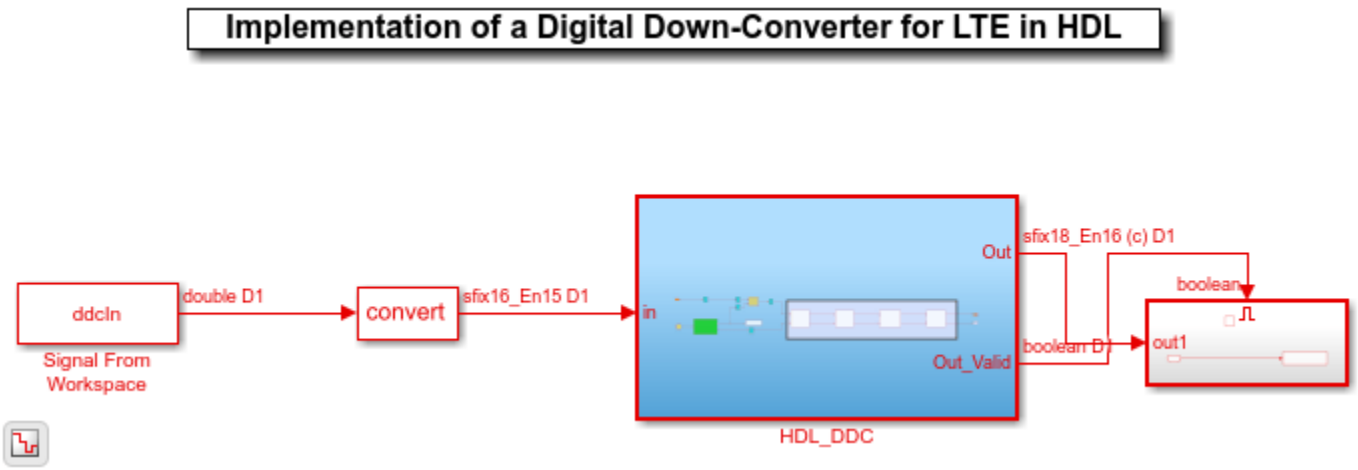
The input to the DDC comes from the `ddcIn` variable. For now, assign a dummy value for `ddcIn` so that the model can compute its data types. During testing, `ddcIn` provides input data to the model.

```
ddcIn = 0; %#ok<NASGU>
```

Model Structure

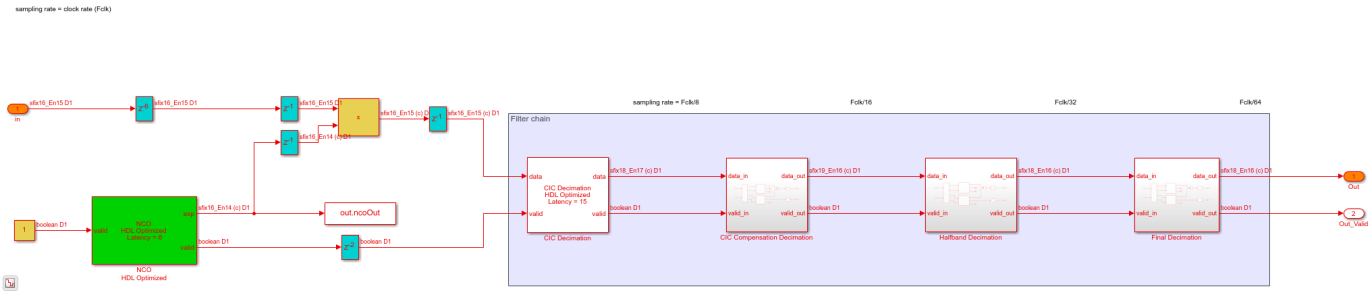
This figure shows the top level of the DDC Simulink model. The model imports the `ddcIn` variable from the MATLAB workspace by using a Signal From Workspace block, converts the input signal to 16-bit values, and applies the signal to the DDC. You can generate HDL code from the `HDL_DDC` subsystem.

```
modelName = 'DDCHDLImplementation';
open_system(modelName);
set_param(modelName,'SimulationCommand','Update');
set_param(modelName,'Open','on');
```



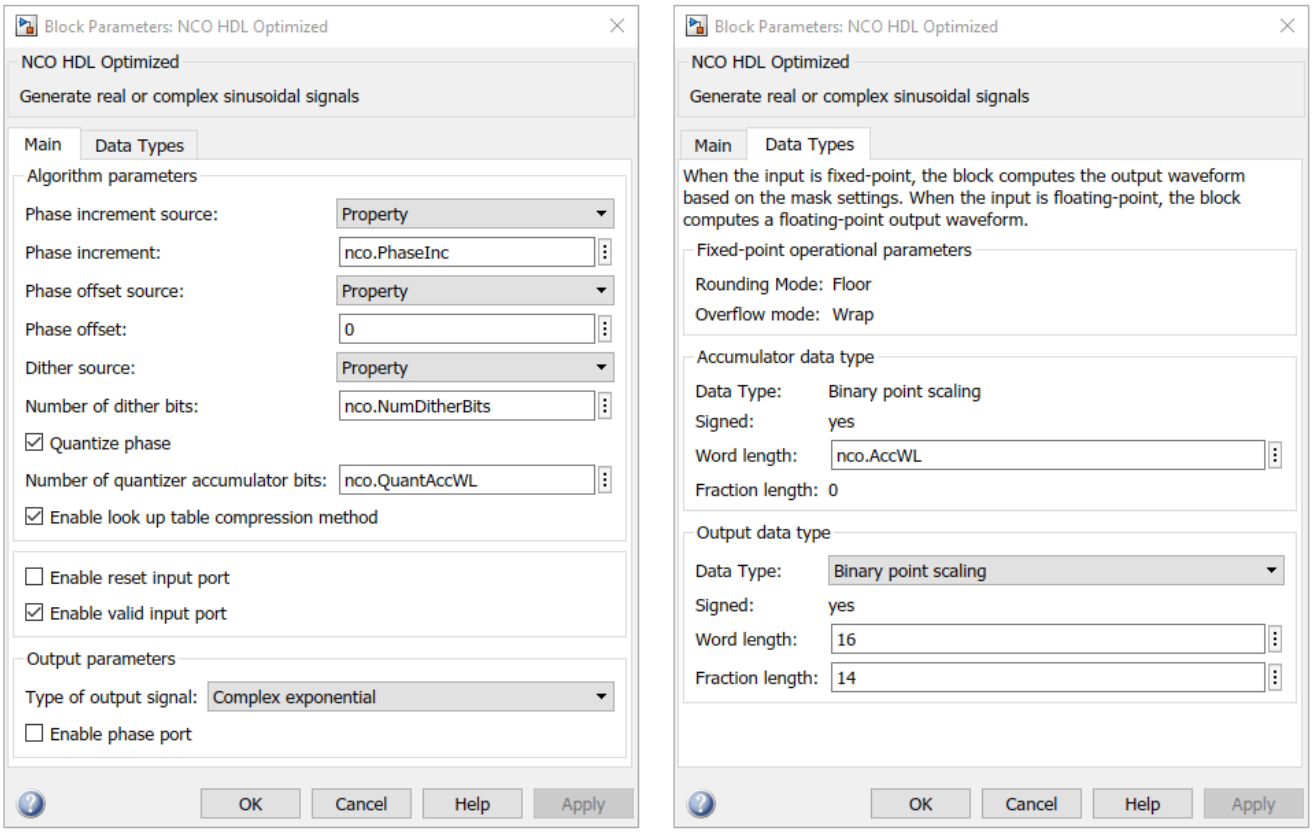
The DDC implementation is inside the `HDL_DDC` subsystem. The NCO HDL Optimized block generates a complex phasor at the carrier frequency. This signal goes to a mixer that multiplies the phasor with the input signal. Then, the output of the mixer is passed to the filter chain and decimated to 1.92 Msps.

```
set_param([modelName '/HDL_DDC'],'Open','on');
```



NCO Block Parameters

The NCO block in the model is configured with the parameters defined in the `nco` structure. This figure shows both tabs of the NCO HDL Optimized block parameters dialog.



CIC Decimation and Gain Correction

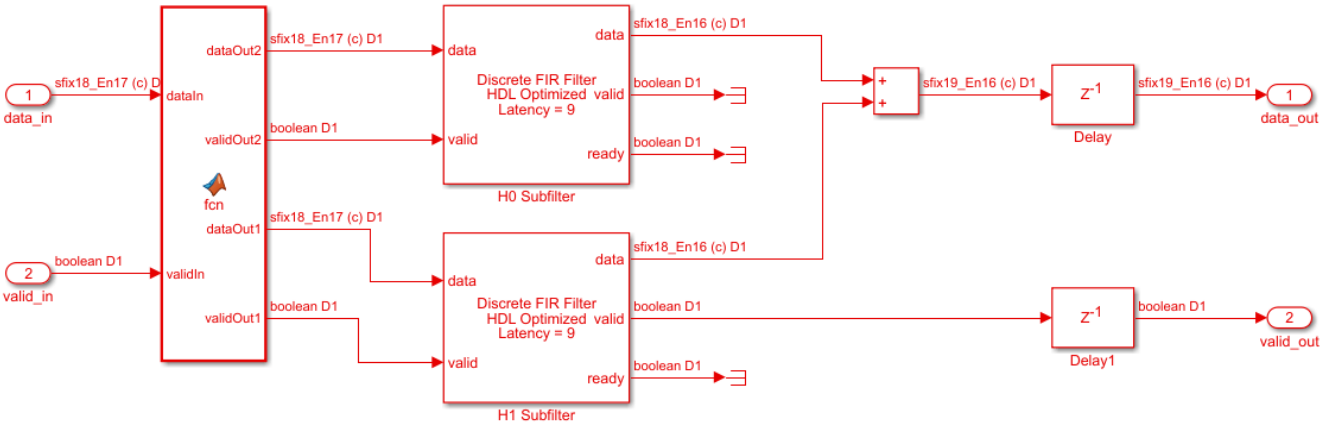
The first filter stage is a CIC Decimator that is implemented with a CIC Decimation HDL Optimized block. The block parameters are set to the `cicParams` structure values. To implement the gain correction, the model selects the **Gain correction** parameter.

The model configures the filters by using the properties of the corresponding System objects. The CIC compensation, halfband decimation, and final cecimation filters operate at effective sample rates that are lower than the clock rate by factors of 8, 16, and 32, respectively. The model implements these sample rates by using the **valid** input signal to indicate which samples are valid at each rate. The signals in the filter chain all have the same Simulink sample time.

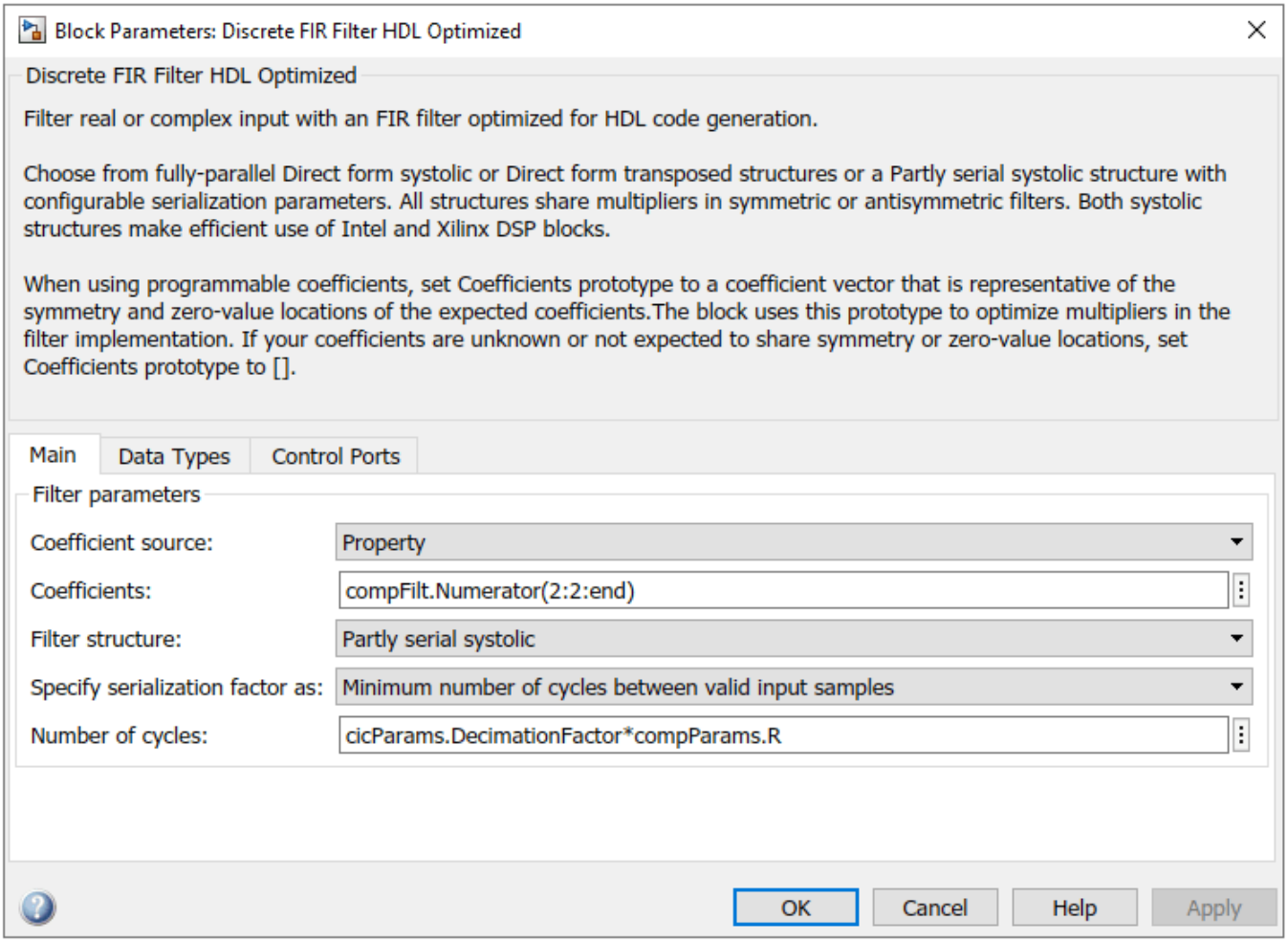
The CIC compensation, halfband decimation, and final decimation filters are each implemented by a MATLAB Function block and two Discrete FIR Filter HDL Optimized blocks in a polyphase decomposition. Polyphase decomposition implements the transform function $H(z) = H_0(z) + z^{-1}H_1(z)$, where $H(z) = a_0 + a_1z^{-1} + a_2z^{-2} + a_3z^{-3} + \ldots$, $H_0(z) = a_0 + a_2z^{-2} + \ldots$, and $H_1(z) = a_1 + a_3z^{-3} + \ldots$. Polyphase decomposition is a resource-efficient way to implement decimation filters. The MATLAB Function block holds two input samples every two cycles and passes them at the same time to the parallel pair of Discrete FIR Filter HDL Optimized blocks, $H_0(z)$ and $H_1(z)$. The lower subfilter, $H_1(z)$, and the upper subfilter, $H_0(z)$, each contain half of the filter coefficients and process half of the input data.

For example, the `CIC Compensation Decimation` subsystem implements a seven-coefficient filter. The upper subfilter, $H_0(z)$, has four coefficients and the lower subfilter, $H_1(z)$, has three coefficients. Each filter receives a sample and generates an output every 16 cycles. Because each filter processes one sample every 16 cycles, the subfilter blocks can share hardware resources in time. To optimize hardware resources in this way, both of the subfilters have the **Filter structure** parameter set to `Partly serial systolic`.

This figure shows the `CIC Compensation Decimation` subsystem. The `Halfband Decimation` and `Final Decimation` subsystems use the same structure.



The model configures all the filter blocks by using the parameters defined in their corresponding structures. For example, this figure shows the block parameters for the CIC Compensation Decimation block. The **Number of cycles** parameter is the minimum number of cycles between input samples. The sample rate of the input to the CIC Compensation Decimation block is `cicParams.DecimationFactor*compParams.R`, which is one sample every 16 cycles for each subfilter.



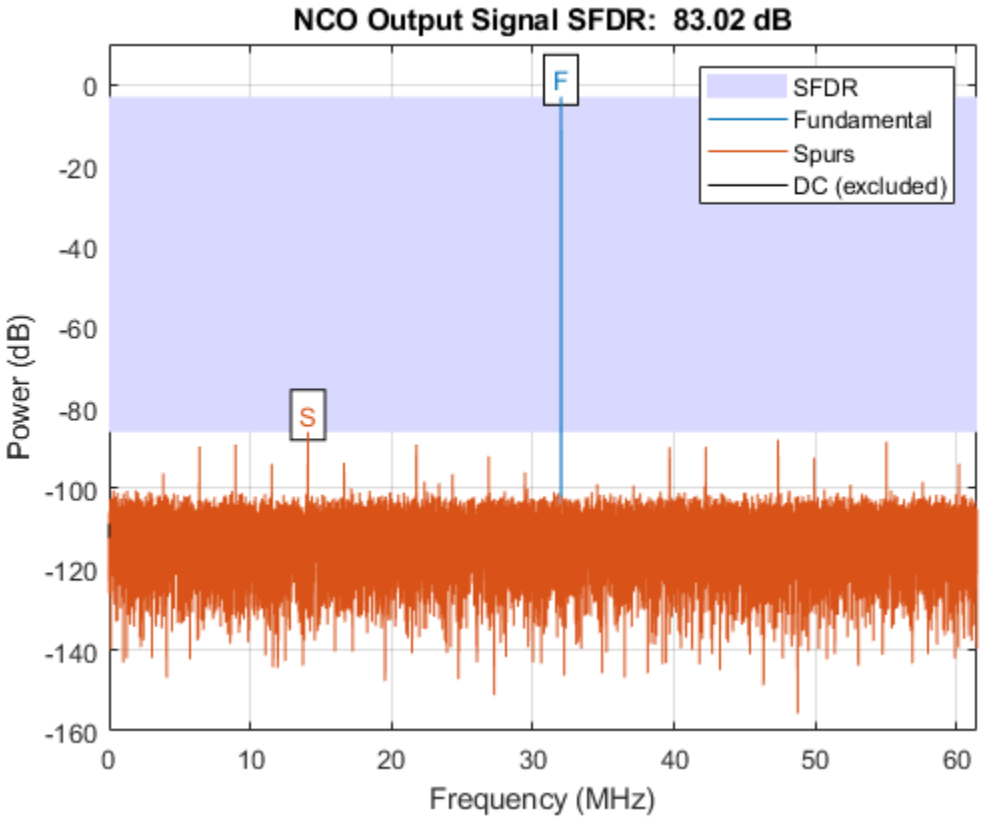
The serial filter implementation reuses the multipliers in time over the number of clock cycles you specify. Without this optimization, the `CIC Compensation Decimation` subsystem with complex input data uses 14 multipliers. After the optimization, $H_0(z)$ and $H_1(z)$ each use use two multipliers for a total of four. Similarly, the `Halfband Decimation` and `Final Decimation` subsystems use four multipliers each.

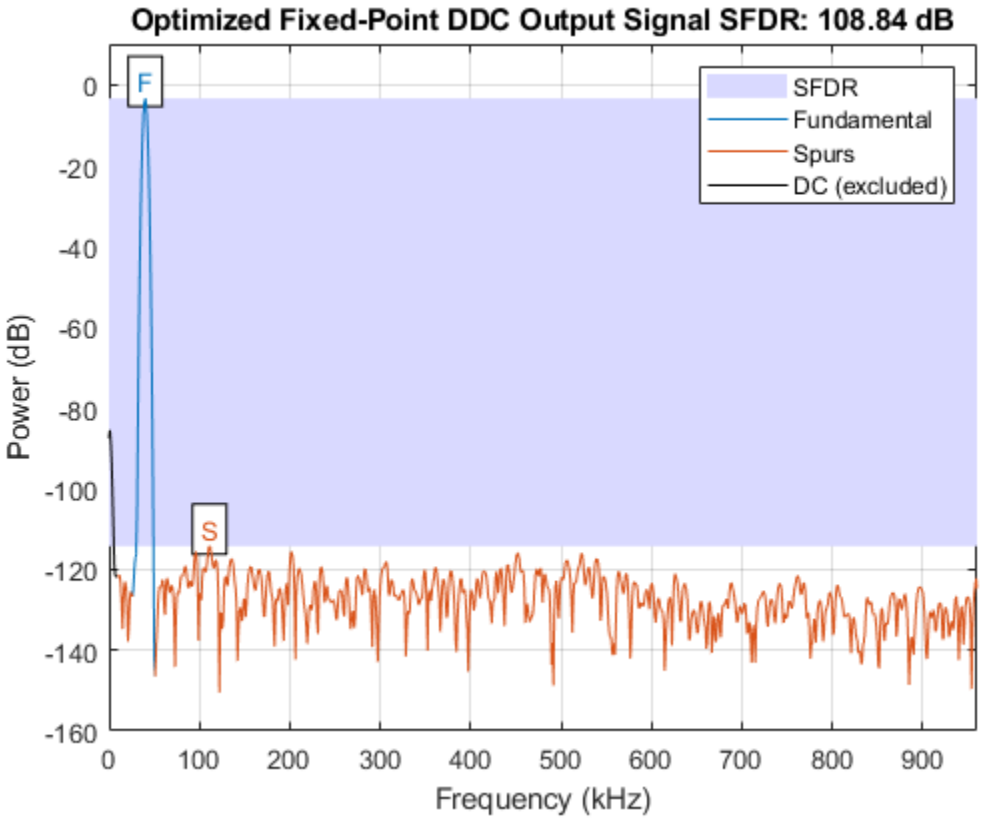
Sinusoid on Carrier Test and Verification

To test the DDC, modulate a 40 kHz sinusoid onto the carrier frequency and pass the modulated sine wave through the DDC. Then, measure the spurious free dynamic range (SFDR) of the resulting tone and the SFDR of the NCO output. Plot the SFDR of the NCO and the fixed-point DDC output.

```
% Initialize random seed before executing any simulations.
rng(0);

% Generate a 40 kHz test tone, modulated onto the carrier.
ddcIn = DDCTestUtils.GenerateTestTone(40e3,Fc);

% Demodulate the test signal with the floating-point DDC.
ddcOut = DDCTestUtils.DownConvert(ddcIn,FsIn,Fc,ddcFilterChain);
release(ddcFilterChain);
```

```
% Demodulate the test signal by running the Simulink model.
out = sim(modelName);

% Measure the SFDR of the NCO, floating-point DDC outputs, and fixed-point
% DDC outputs.
results.sfdrNCO = sfdr(real(out.ncoOut),FsIn);
results.sfdrFloatDDC = sfdr(real(ddcOut),FsOut);
results.sfdrFixedDDC = sfdr(real(out.ddcOut),FsOut);

disp('SFDR Measurements');
disp(['   Floating-point DDC SFDR: ',num2str(results.sfdrFloatDDC) ' dB']);
disp(['   Fixed-point NCO SFDR: ',num2str(results.sfdrNCO) ' dB']);
disp(['   Optimized fixed-point DDC SFDR: ',num2str(results.sfdrFixedDDC) ' dB']);
fprintf(newline);

% Plot the SFDR of the NCO and fixed-point DDC outputs.
ddcPlots.ncoOutSDFR = figure;
sfdr(real(out.ncoOut),FsIn);
DDCTestUtils.setPlotNameAndTitle(['NCO Output Signal ' get(gca,'Title').String]);

ddcPlots.OptddcOutSFDR = figure;
sfdr(real(out.ddcOut),FsOut);
DDCTestUtils.setPlotNameAndTitle(['Optimized Fixed-Point DDC Output Signal ' get(gca,'Title').St

SFDR Measurements
   Floating-point DDC SFDR: 291.3779 dB
   Fixed-point NCO SFDR: 83.0249 dB
   Optimized fixed-point DDC SFDR: 108.8419 dB
```


NCO Output Signal SFDR: 83.02 dB
SFDR
Fundamental
Spurs
DC (excluded)
F
S
Power (dB)
0
-20
-40
-60
-80
-100
-120
-140
-160
0
10
20
30
40
50
60
Frequency (MHz)



LTE Signal Test

You can use an LTE test signal to perform more rigorous testing of the DDC. Generate a standard compliant LTE waveform by using LTE Toolbox™ functions. Then, downconvert the waveform with the DDC model. Use LTE Toolbox functions to measure the error vector magnitude (EVM) of the resulting signals.

```
rng(0);
% Execute this test only if you have the LTE Toolbox product.
if license('test','LTE_Toolbox')

    % Generate a modulated LTE test signal by using the LTE Toolbox functions.
    [ddcIn,sigInfo] = DDCTestUtils.GenerateLTETestSignal(Fc);

    % Downconvert the signal with the floating-point DDC.
    ddcOut = DDCTestUtils.DownConvert(ddcIn,FsIn,Fc,ddcFilterChain);
    release(ddcFilterChain);

    % Downconvert the signal with the Simulink model.
    % Pad the input with zeros to represent propagation latency and return the complete result.
    ddcIn = [ddcIn;zeros(320,1)];
    out = sim(modelName);

    results.evmFloat = DDCTestUtils.MeasureEVM(sigInfo,ddcOut);
    results.evmFixed = DDCTestUtils.MeasureEVM(sigInfo,out.ddcOut);

    disp('LTE Error Vector Magnitude (EVM) Measurements');
```

```
    disp(['   Floating-point DDC RMS EVM: '  num2str(results.evmFloat.RMS*100,3) '%']);
    disp(['   Floating-point DDC Peak EVM: ' num2str(results.evmFloat.Peak*100,3) '%']);
    disp(['   Fixed-point HDL-optimized DDC RMS EVM: '     num2str(results.evmFixed.RMS*100,3) '
    disp(['   Fixed-point HDL-optimized DDC Peak EVM: '    num2str(results.evmFixed.Peak*100,3)
    fprintf(newline);

end
```

```
LTE Error Vector Magnitude (EVM) Measurements
   Floating-point DDC RMS EVM: 0.633%
   Floating-point DDC Peak EVM: 2.44%
   Fixed-point HDL-optimized DDC RMS EVM: 0.731%
   Fixed-point HDL-optimized DDC Peak EVM: 2.69%
```

HDL Code Generation and FPGA Implementation

To generate the HDL code for this example you must have the HDL Coder™ product. Use the `makehdl` and `makehdltb` commands to generate HDL code and an HDL test bench for the `HDL_DDC` subsystem. The DDC was synthesized on a Xilinx® Zynq®-7000 ZC706 evaluation board. The table shows the post place-and-route resource utilization results. The design met timing with a clock frequency of 313 MHz.

```
T = table(...
    categorical({'LUT'; 'LUTRAM'; 'FF'; 'BRAM'; 'DSP'}), ...
    categorical({'2660'; '318'; '5951'; '1.0'; '18'}), ...
    'VariableNames',{'Resource','Usage'})
```

```
T =

  5x2 table

    Resource    Usage
    ________    _____

     LUT        2660
     LUTRAM     318
     FF         5951
     BRAM       1.0
     DSP        18
```

HDL Implementation of a Digital Up-Converter for LTE

This example shows how to design a digital up-converter (DUC) for radio communication applications such as LTE, and generate HDL code.

Introduction

DUCs are widely used in digital communication transmitters to convert baseband signals to radio frequency (RF) or intermediate frequency (IF) signals. The DUC operation increases the sample rate of the signal and shifts it to a higher frequency to facilitate subsequent processing stages. The DUC in this example performs sample rate conversion using a four-stage filter chain followed by complex frequency translation. The example starts by designing the DUC with DSP System Toolbox™ functions in floating point. Then, each stage is converted to fixed point, and used in a Simulink® model that generates synthesizable HDL code. The example uses these two test signals to demonstrate and verify the DUC operation:

- A sinusoid that is modulated onto a 32 MHz IF carrier.
- An LTE downlink signal with a bandwidth of 1.4 MHz, modulated onto a 32 MHz IF carrier.

The example downconverts the outputs of the floating-point and fixed-point DUCs, and compares the signal quality of the two outputs.

Finally, the example presents implementation of the filter chain for FPGAs, and synthesis results.

This example uses `DUCTestUtils`, a helper class that contains functions for generating stimulus and analyzing the DUC output. For more information, see the `DUCTestUtils.m` file.

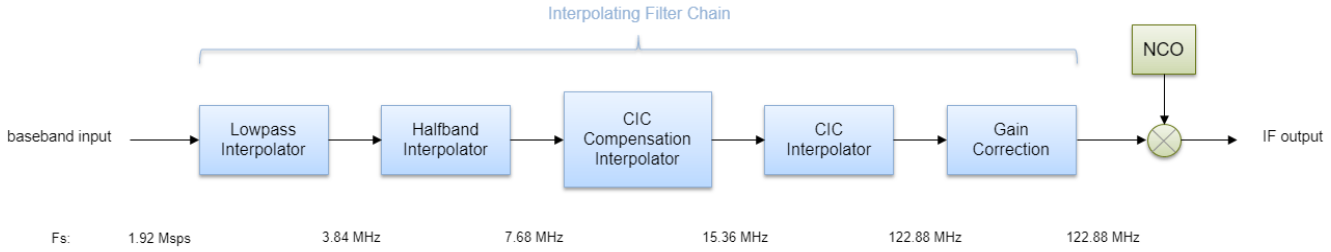
DUC Structure

The DUC consists of an interpolating filter chain, numerically controlled oscillator (NCO), and mixer. The filter chain consists of a lowpass interpolator, halfband interpolator, CIC compensation interpolator (FIR), CIC interpolator, and CIC gain correction.

The overall response of the filter chain is equivalent to that of a single interpolation filter with the same specification. However, splitting the filter into multiple interpolation stages results in a more efficient design that uses fewer hardware resources.

The first lowpass interpolator implements the precise `Fpass` and `Fstop` characteristics of the DUC. The halfband filter is an intermediate interpolator. The lower sampling rates at the beginning of the chain mean the earlier filters can optimize resource use by sharing multipliers. The CIC compensation interpolator improves the spectral response by compensating for the later CIC droop while interpolating by two. The CIC interpolator provides a large interpolation factor, which meets the filter chain upsampling requirements.

This figure shows a block diagram of the DUC.



The sample rate of the input to the DUC is 1.92 Msps, and the output sample rate is 122.88 Msps. These rates give an overall interpolation factor of 64. LTE receivers use 1.92 Msps as the typical sampling rate for cell search and master information block (MIB) recovery. The DUC filters are designed to suit this application. The DUC is optimized to run at a clock rate of 122.88 MHz.

DUC Design

This section explains how to design the DUC using floating-point operations and filter-design functions in MATLAB®. The DUC object enables you to specify several characteristics that define the response of the cascade for the four filters, including passband and stopband frequencies, passband ripple, and stopband attenuation.

DUC Parameters

This example designs the DUC filter characteristics to meet these desired response values for the given input sampling rate and carrier frequency.

```
FsIn = 1.92e6;      % Sampling rate at input to DUC
Fc = 32e6;          % Carrier frequency
Fpass = 540e3;      % Passband frequency, equivalent to 36*15kHz LTE subcarriers
Fstop = 700e3;      % Stopband frequency
Ap = 0.1;           % Passband ripple
Ast = 60;           % Stopband attenuation
```

First Lowpass Interpolator

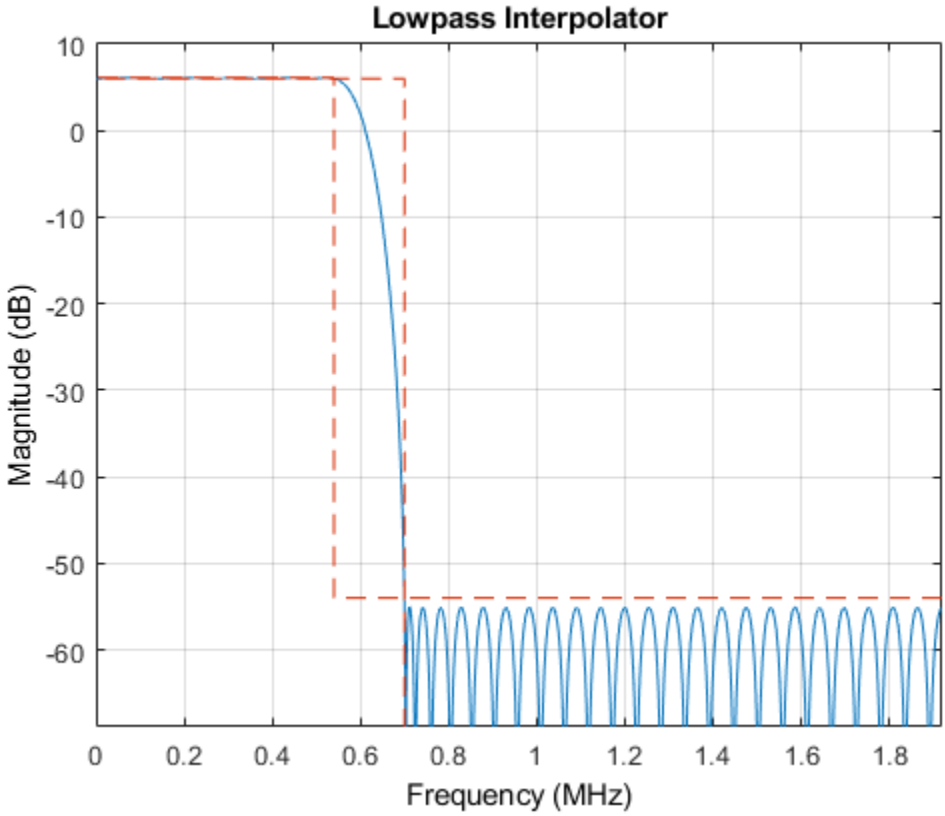
This filter interpolates by two, and operates at the lowest sampling rate of the filter chain. The low sample rate means this filter can use resource sharing for an efficient hardware implementation.

```
lowpassParams.FsIn = FsIn;
lowpassParams.InterpolationFactor = 2;
lowpassParams.FsOut = FsIn*lowpassParams.InterpolationFactor;

lowpassSpec = fdesign.interpolator(lowpassParams.InterpolationFactor, ...
    'lowpass','Fp,Fst,Ap,Ast',Fpass,Fstop,Ap,Ast,lowpassParams.FsOut);
lowpassFilt = design(lowpassSpec,'SystemObject',true)
```

```
lowpassFilt =

  dsp.FIRInterpolator with properties:

    InterpolationFactor: 2
        NumeratorSource: 'Property'
              Numerator: [0.0020 0.0021 4.9115e-04 -0.0027 -0.0050 ... ]

  Use get to show all properties
```

Display the magnitude response of the lowpass filter without gain correction.

```
ducPlots.lowpass = fvtool(lowpassFilt,'Fs',FsIn*2,'Legend','off');
DUCTestUtils.setPlotNameAndTitle('Lowpass Interpolator');
```



Second Halfband Interpolator

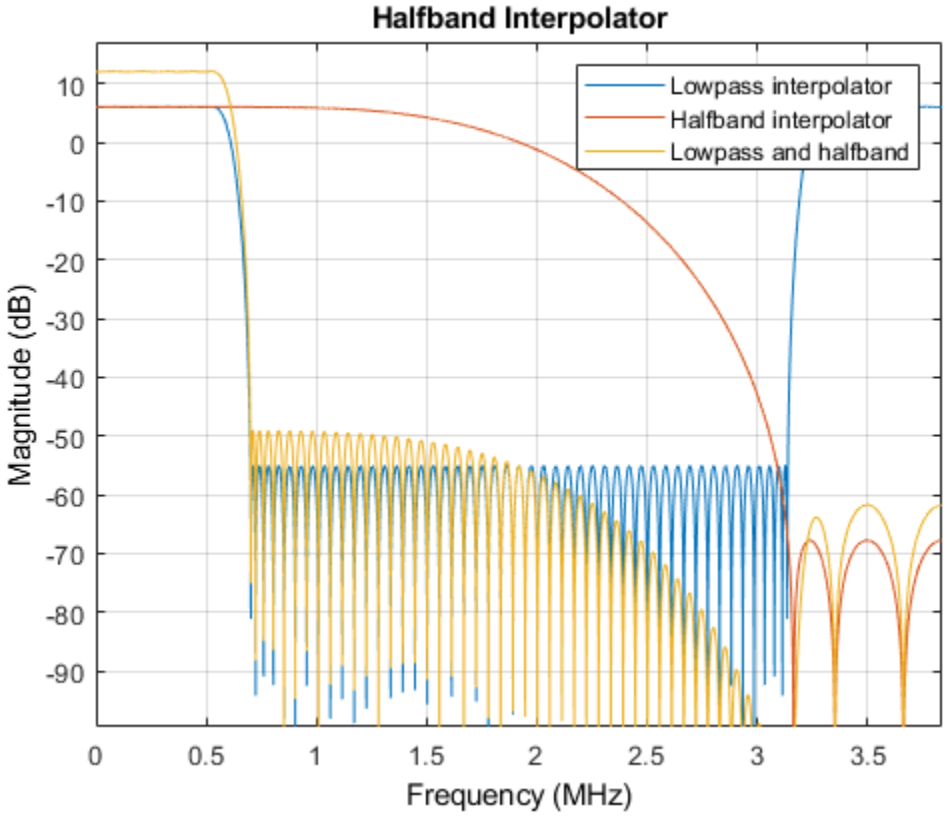
The halfband filter provides efficient interpolation by two. Halfband filters are efficient for hardware because approximately half of their coefficients are equal to zero, and those multipliers are excluded from the hardware implementation.

```
hbParams.FsIn = lowpassParams.FsOut;
hbParams.InterpolationFactor = 2;
hbParams.FsOut = lowpassParams.FsOut*hbParams.InterpolationFactor;
hbParams.TransitionWidth = hbParams.FsIn - 2*Fstop;
hbParams.StopbandAttenuation = Ast;

hbSpec = fdesign.interpolator(hbParams.InterpolationFactor,'halfband', ...
         'TW,Ast', ...
         hbParams.TransitionWidth, ...
         hbParams.StopbandAttenuation, ...
         hbParams.FsOut);

hbFilt = design(hbSpec,'SystemObject',true)


hbFilt =

  dsp.FIRInterpolator with properties:

    InterpolationFactor: 2
        NumeratorSource: 'Property'
```

```
                Numerator: [0.0178 0 -0.1129 0 0.5953 1 0.5953 0 -0.1129 0 ... ]

  Use get to show all properties
```

Visualize the magnitude response of the halfband interpolation.

```
ducFilterChain = dsp.FilterCascade(lowpassFilt,hbFilt);
ducPlots.hbFilt = fvtool(lowpassFilt,hbFilt,ducFilterChain, ...
                            'Fs',[FsIn*2,FsIn*4,FsIn*4]);
legend( ...
       'Lowpass interpolator', ...
       'Halfband interpolator', ...
       'Lowpass and halfband');

DUCTestUtils.setPlotNameAndTitle('Halfband Interpolator');
```



CIC Compensation Interpolator

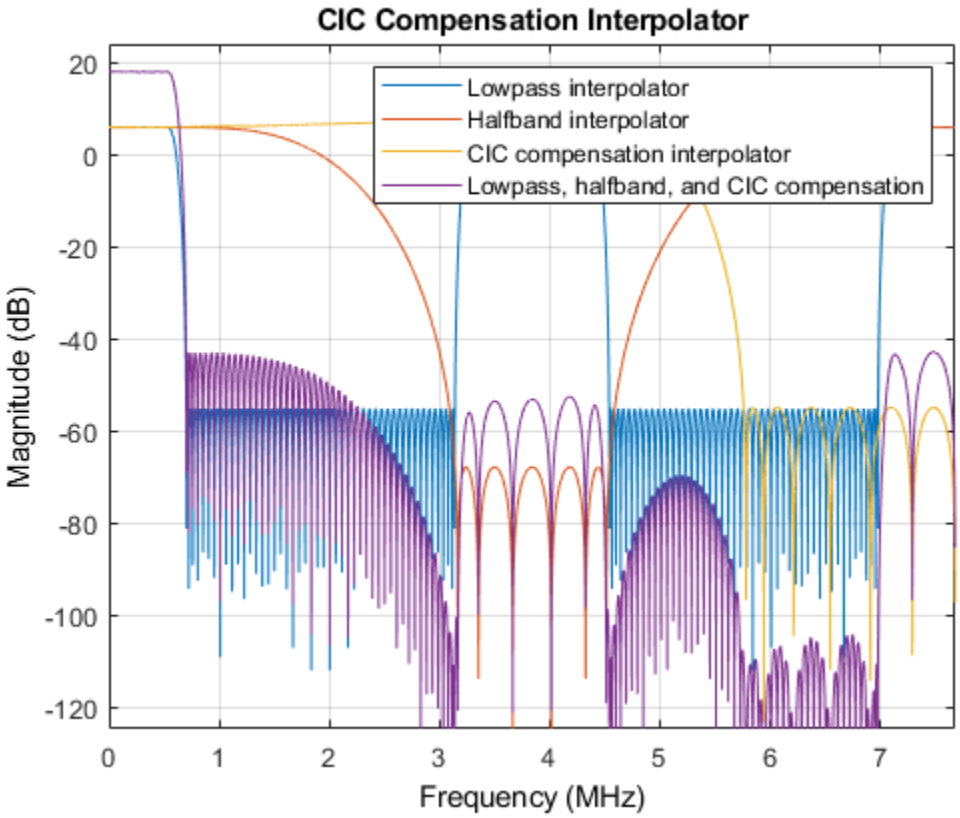
Because the magnitude response of the last CIC filter has a significant *droop* within the passband region, the example uses an FIR-based droop compensation filter to flatten the passband response. The droop compensator has the same properties as the CIC interpolator. This filter implements interpolation by a factor of two, so you must also specify bandlimiting characteristics for the filter. Also, specify the CIC interpolator properties that are used for this compensation filter as well as the later CIC interpolator.

Use the `design` function to return a filter System object with the specified characteristics.

```
compParams.FsIn = hbParams.FsOut;
compParams.InterpolationFactor = 2;                                    % CIC compensation interpolat
compParams.FsOut = compParams.FsIn*compParams.InterpolationFactor;  % New sampling rate
compParams.Fpass = 1/2*compParams.FsIn + Fpass;                        % CIC compensation passband f
compParams.Fstop = 1/2*compParams.FsIn + 1/4*compParams.FsIn;       % CIC compensation stopband f
compParams.Ap = Ap;                                                    % Same passband ripple as ove
compParams.Ast = Ast;                                                  % Same stopband attenuation a

cicParams.InterpolationFactor = 8;  % CIC interpolation factor
cicParams.DifferentialDelay = 1;    % CIC interpolator differential delay
cicParams.NumSections = 3;          % CIC interpolator number of integrator and comb sections

compSpec = fdesign.interpolator(compParams.InterpolationFactor,'ciccomp', ...
           cicParams.DifferentialDelay, ...
           cicParams.NumSections, ...
           cicParams.InterpolationFactor, ...
           'Fp,Fst,Ap,Ast', ...
           compParams.Fpass,compParams.Fstop,compParams.Ap,compParams.Ast, ...
           compParams.FsOut);
compFilt = design(compSpec,'SystemObject',true)


compFilt =

  dsp.FIRInterpolator with properties:

    InterpolationFactor: 2
        NumeratorSource: 'Property'
              Numerator: [-0.0012 0.0078 0.0030 -0.0141 0.0125 0.0118 ... ]

  Use get to show all properties
```

Plot the response of the CIC compensation interpolator.

```
ducFilterChain = dsp.FilterCascade(lowpassFilt,hbFilt,compFilt);
ducPlots.cicComp = fvtool(lowpassFilt,hbFilt,compFilt,ducFilterChain, ...
                          'Fs',[FsIn*2,FsIn*4,FsIn*8,FsIn*8]);
legend( ...
       'Lowpass interpolator', ...
       'Halfband interpolator', ...
       'CIC compensation interpolator', ...
       'Lowpass, halfband, and CIC compensation');
DUCTestUtils.setPlotNameAndTitle('CIC Compensation Interpolator');
```



CIC Interpolator

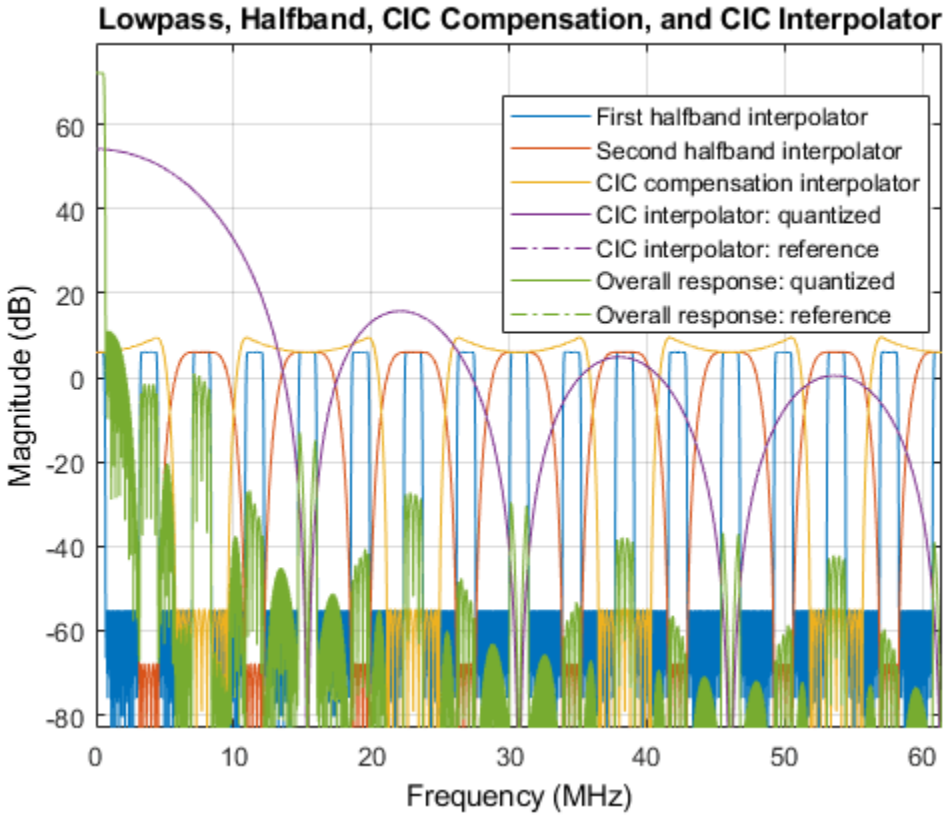
The last filter stage is implemented as a CIC interpolator because of this type of filter's ability to efficiently implement a large decimation factor. The response of a CIC filter is similar to a cascade of moving average filters, but a CIC filter uses no multiplication or division. As a result, the CIC filter has a large DC gain.

```
cicParams.FsIn = compParams.FsOut;
cicParams.FsOut = cicParams.FsIn*cicParams.InterpolationFactor;

cicFilt = dsp.CICInterpolator(cicParams.InterpolationFactor, ...
          cicParams.DifferentialDelay,cicParams.NumSections) %#ok<*NOPTS>
```

```
cicFilt =

  dsp.CICInterpolator with properties:

    InterpolationFactor: 8
      DifferentialDelay: 1
             NumSections: 3
     FixedPointDataType: 'Full precision'
```

Visualize the magnitude response of the CIC interpolation. CIC filters use fixed-point arithmetic internally, so `fvtool` plots both the quantized and unquantized responses.

```
ducFilterChain = dsp.FilterCascade(lowpassFilt,hbFilt,compFilt,cicFilt);
ducPlots.cicInter = fvtool(lowpassFilt,hbFilt,compFilt,cicFilt,ducFilterChain, ...
                     'Fs',[FsIn*2,FsIn*4,FsIn*8,FsIn*64,FsIn*64]);

legend( ...
       'First halfband interpolator', ...
       'Second halfband interpolator', ...
       'CIC compensation interpolator', ...
       'CIC interpolator: quantized', ...
       'CIC interpolator: reference', ...
       'Overall response: quantized', ...
       'Overall response: reference');
DUCTestUtils.setPlotNameAndTitle('Lowpass, Halfband, CIC Compensation, and CIC Interpolator');
```



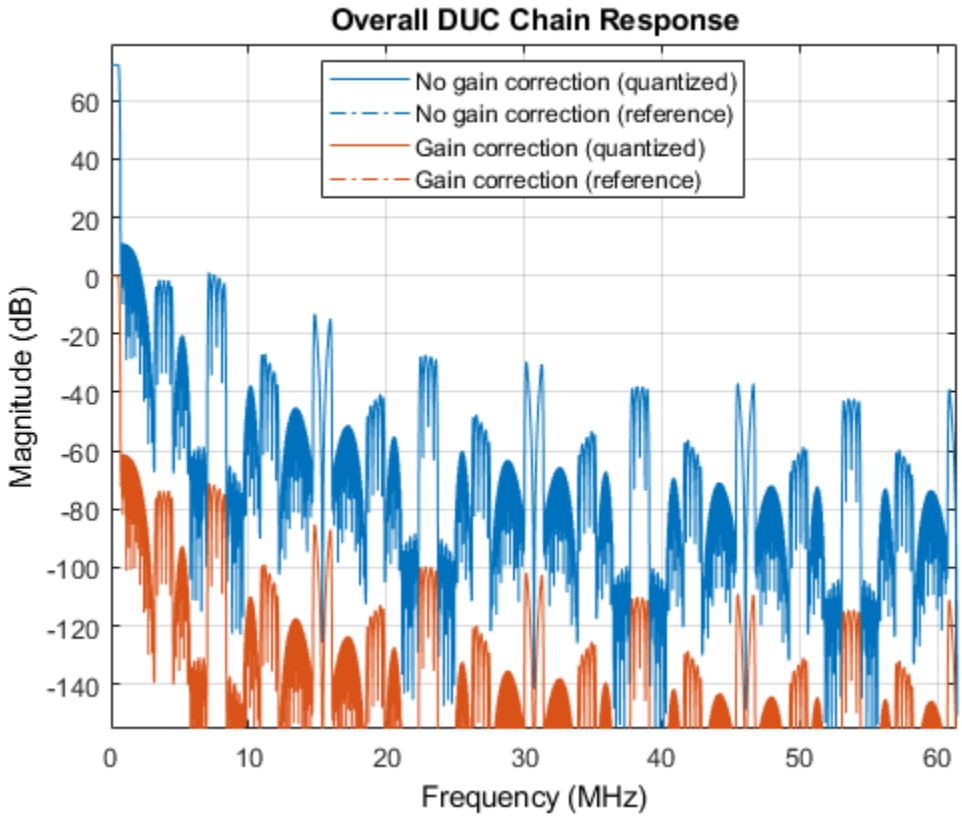
Every interpolator has a DC gain that is determined by its interpolation factor. The CIC interpolator has a larger gain than other filters. Call the `gain` function to get the gain factor of this filter.

```
% Because the CIC gain is a power of two, a hardware implementation can easily
% correct for the gain factor by using a shift operation.
% For analysis purposes, the example represents the gain correction by using
% a one-tap |dsp.FIRFilter| System object. Combine the filter chain and the
% gain correction filter into a |dsp.FilterCascade| System object.

cicGain = gain(cicFilt)
Gain = lowpassParams.InterpolationFactor* ...
    hbParams.InterpolationFactor*compParams.InterpolationFactor* ...
    cicParams.InterpolationFactor*cicGain;
GainCorr = dsp.FIRFilter('Numerator',1/Gain)
```

```
cicGain =

    64


GainCorr =

  dsp.FIRFilter with properties:

           Structure: 'Direct form'
     NumeratorSource: 'Property'
           Numerator: 2.4414e-04
    InitialConditions: 0

  Use get to show all properties
```

Plot the overall chain response with and without gain correction.

```
ducPlots.overallResponse = fvtool(ducFilterChain,dsp.FilterCascade(ducFilterChain,GainCorr), ...
                           'Fs',[FsIn*64,FsIn*64]);
legend( ...
       'No gain correction (quantized)', ...
       'No gain correction (reference)', ...
       'Gain correction (quantized)', ...
       'Gain correction (reference)');
DUCTestUtils.setPlotNameAndTitle('Overall DUC Chain Response');
```



Fixed-Point Conversion

The frequency response of the floating-point DUC filter chain now meets the specification. Next, quantize each filter stage to use fixed-point types and analyze them to confirm that the filter chain still meets the specification.

Filter Quantization

This example uses 16-bit coefficients, which are sufficient to meet the specification. Using fewer than 18 bits for the coefficients minimizes the number of DSP blocks that are required for an FPGA implementation. The input to the DUC filter chain is 16-bit data with 15 fractional bits. The filter outputs are 18-bit values, which provide extra headroom and precision in the intermediate signals.

```
% First Lowpass Interpolator
lowpassFilt.FullPrecisionOverride = false;
lowpassFilt.CoefficientsDataType = 'Custom';
lowpassFilt.CustomCoefficientsDataType = numerictype([],16,15);
lowpassFilt.ProductDataType = 'Full precision';
lowpassFilt.AccumulatorDataType = 'Full precision';
lowpassFilt.OutputDataType = 'Custom';
lowpassFilt.CustomOutputDataType = numerictype([],18,14);

% Halfband
hbFilt.FullPrecisionOverride = false;
hbFilt.CoefficientsDataType = 'Custom';
hbFilt.CustomCoefficientsDataType = numerictype([],16,14);
hbFilt.ProductDataType = 'Full precision';
hbFilt.AccumulatorDataType = 'Full precision';
hbFilt.OutputDataType = 'Custom';
hbFilt.CustomOutputDataType = numerictype([],18,14);

% CIC Compensation Interpolator
compFilt.FullPrecisionOverride = false;
compFilt.CoefficientsDataType = 'Custom';
compFilt.CustomCoefficientsDataType = numerictype([],16,14);
compFilt.ProductDataType = 'Full precision';
compFilt.AccumulatorDataType = 'Full precision';
compFilt.OutputDataType = 'Custom';
compFilt.CustomOutputDataType = numerictype([],18,14);
```

For the CIC interpolator, choosing the `'Minimum section word lengths'` fixed-point data type option automatically optimizes the internal wordlengths based on the output wordlength and other CIC parameters.

```
cicFilt.FixedPointDataType = 'Minimum section word lengths';
cicFilt.OutputWordLength = 18;
```

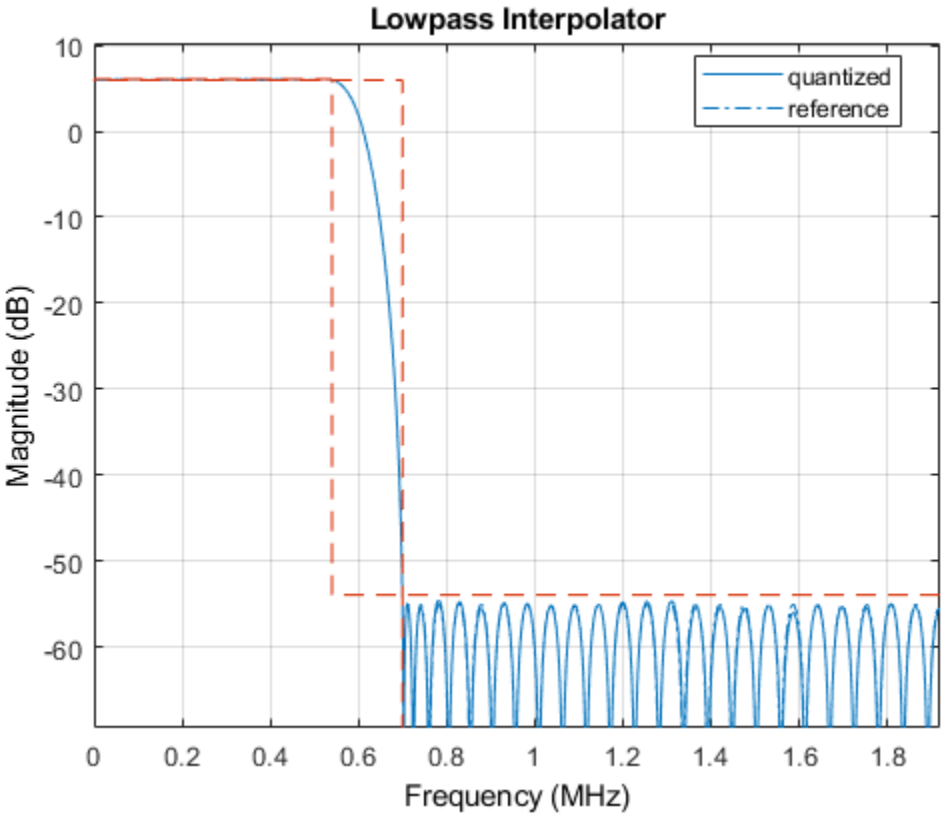
Configure the fixed-point properties of the gain correction and FIR-based System objects. The object uses the default `RoundingMethod` and `OverflowAction` property values (`'Floor'` and `'Wrap'` respectively).

```
% CIC Gain Correction
GainCorr.FullPrecisionOverride = false;
GainCorr.CoefficientsDataType = 'Custom';
GainCorr.CustomCoefficientsDataType = numerictype(fi(GainCorr.Numerator,1,16));
GainCorr.OutputDataType = 'Custom';
GainCorr.CustomOutputDataType = numerictype(1,18,14);
```

Fixed-Point Analysis

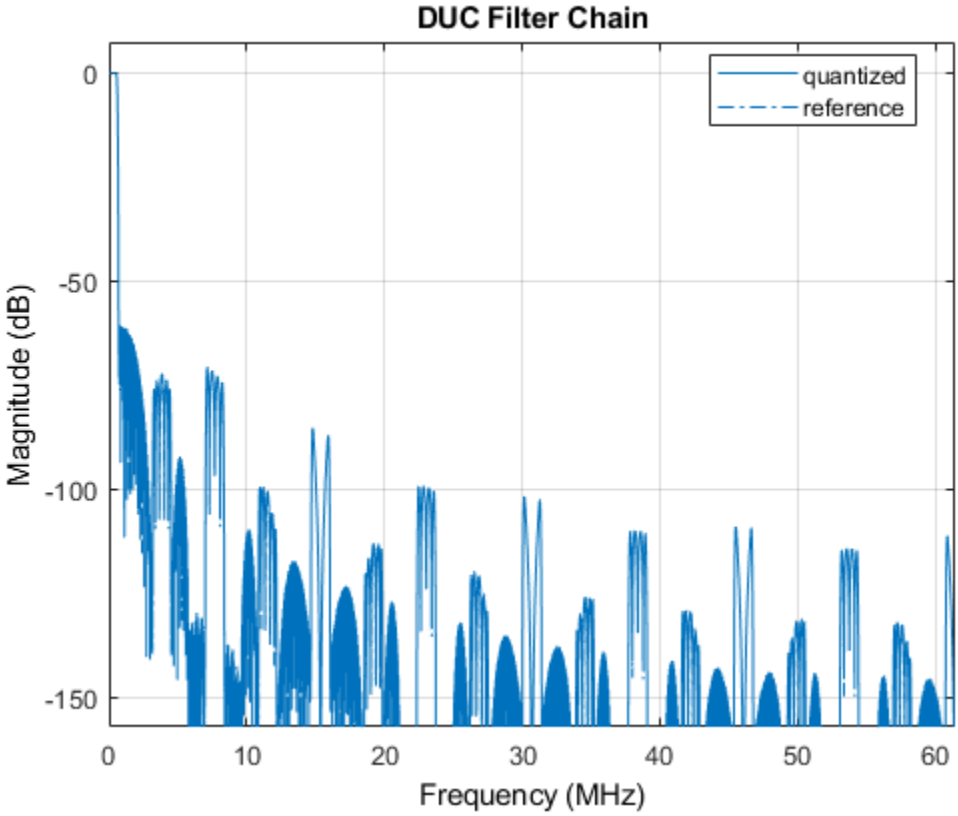
Inspect the quantization effects with `fvtool`. You can analyze the filters individually or in a cascade. `fvtool` shows the quantized and unquantized (reference) responses overlayed. For example, this figure shows the effect of quantizing the first FIR filter stage.

```
ducPlots.quantizedFIR = fvtool(lowpassFilt, ...
    'Fs',lowpassParams.FsIn*2,'arithmetic','fixed');
DUCTestUtils.setPlotNameAndTitle('Lowpass Interpolator');
legend('quantized', 'reference');
```



Redefine the `ducFilterChain` cascade object to include the fixed-point properties of the individual filters. Then use `fvtool` to analyze the entire filter chain and confirm that the quantized DUC still meets the specification.

```
ducFilterChain = dsp.FilterCascade(lowpassFilt,hbFilt,compFilt,cicFilt,GainCorr);
ducPlots.quantizedDUCResponse = fvtool(ducFilterChain, ...
    'Fs',FsIn*64,'Arithmetic','fixed');

DUCTestUtils.setPlotNameAndTitle('DUC Filter Chain');
legend('quantized', 'reference');
```



HDL-Optimized Simulink Model

The next step in the design flow is to implement the DUC in Simulink using blocks that support HDL code generation.

Model Configuration

The model relies on variables in the MATLAB workspace to configure the blocks and settings. It uses the same filter chain variables defined earlier in the example.

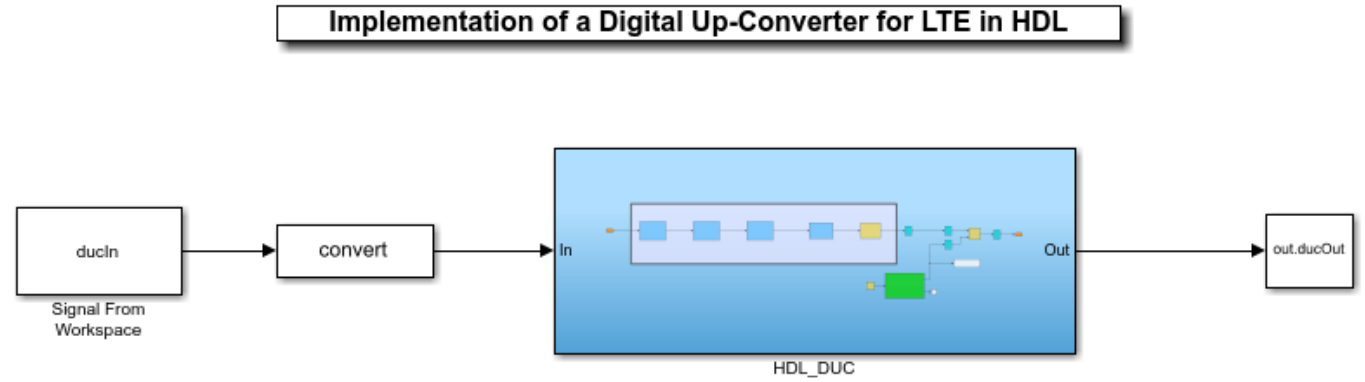
The input to the DUC comes from the `ducIn` variable. For now, assign a dummy value for `ducIn` so that the model can compute its data types. During testing, `ducIn` provides input data to the model.

```
ducIn = 0; %#ok<NASGU>
```

Model Structure

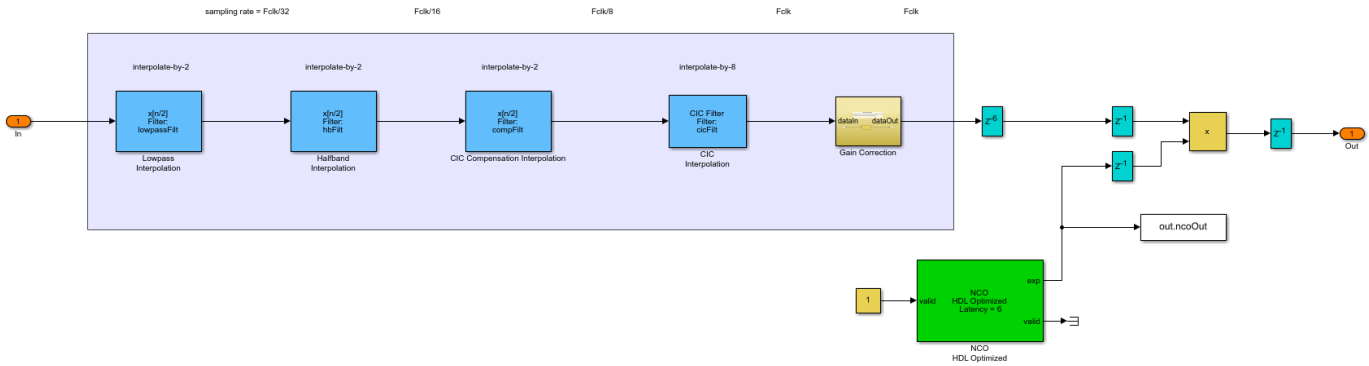
This figure shows the top level of the DUC Simulink model. The model imports the `ducIn` variable from the MATLAB workspace by using a **Signal From Workspace** block, converts the signal to 16-bit values, and applies the signal to the DUC. You can generate HDL code from the `HDL_DUC` subsystem.

```
modelName = 'DUCforLTEHDL';
open_system(modelName);
set_param(modelName,'Open','on');
```



The DUC implementation is inside the `HDL_DUC` subsystem.

```
set_param([modelName '/HDL_DUC'],'Open','on');
```

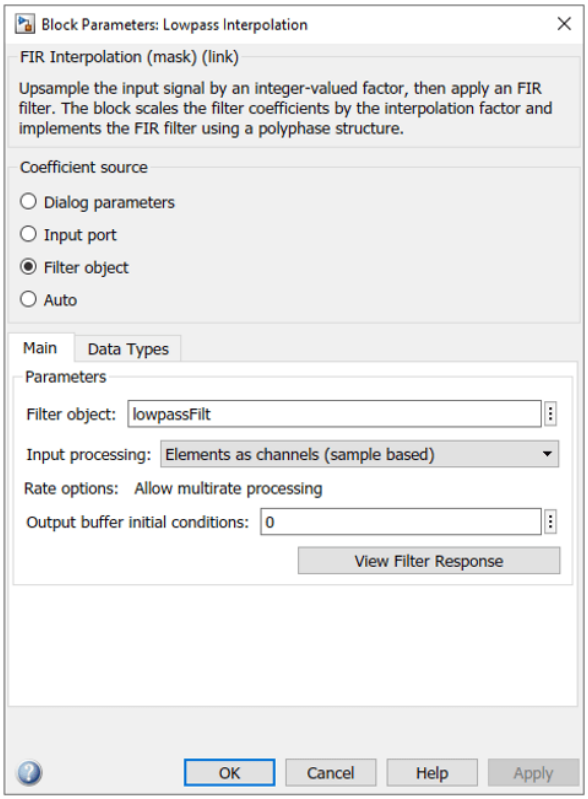


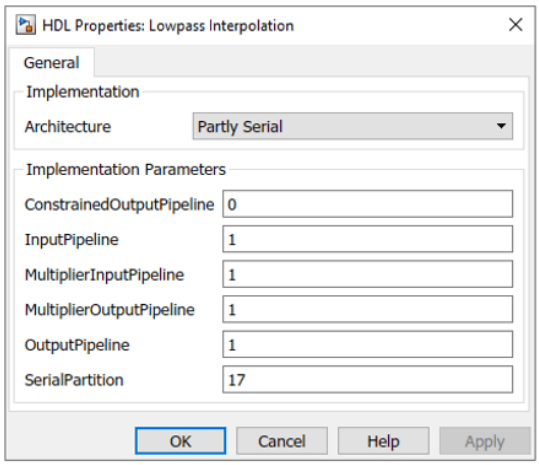
Filter Block Parameters

All of the filters are configured to inherit the properties of the corresponding System objects. Each block also has a set of HDL Properties that are used to optimize the resulting HDL code. The lowpass, halfband, and CIC compensation filters operate at sampling rates that are lower than the clock rate (*Fclk*) by factors of 32, 16, and 8, respectively. These filter blocks use serialization techniques (resource sharing) to minimize hardware resource utilization.

For example, because the sample rate of the input to the Lowpass Interpolation block is *Fclk*/32, 32 clock cycles are available to process each input sample. When you generate HDL code for the DUC, the log file lists all of the `SerialPartition` property values available for the filter blocks, based on their coefficients. These values apply when you set the **Architecture** parameter to `Partly Serial`. The largest value in each `SerialPartition` vector represents the sharing factor of the filter. The sharing factor means that the multipliers are shared in time inside the filter and the filter needs that many cycles to operate on each sample. For the Lowpass Interpolation block in this model, the minimum number of multipliers occurs when the `SerialPartition` property is `17`. This setting means the filter can process one sample every 17 clock cycles. Because one sample arrives at this filter every 32 cycles, the filter has the cycles available to use a sharing factor of 17 to reduce hardware resources. The figures show the block parameters and HDL Properties settings for the Lowpass Interpolation block.

For more information, see “SerialPartition” (HDL Coder) and “HDL Filter Architectures” (HDL Coder).

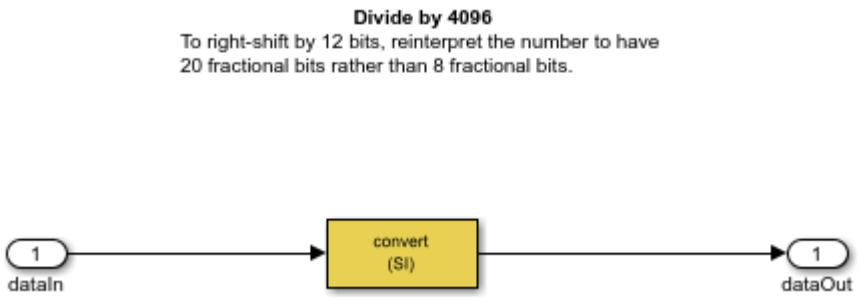




Gain Correction

The gain correction divides the output by 4096, which is equivalent to shifting right by 12 bits. Because the input and output signals of the gain correction each are expressed with 18 bits, the model implements this shift by reinterpreting the data type of the output signal. The Conversion block reinterprets the 12-bit number to have 20 fractional bits rather than 8 fractional bits.

```
set_param([modelName '/HDL_DUC/Gain Correction'],'Open','on');
```



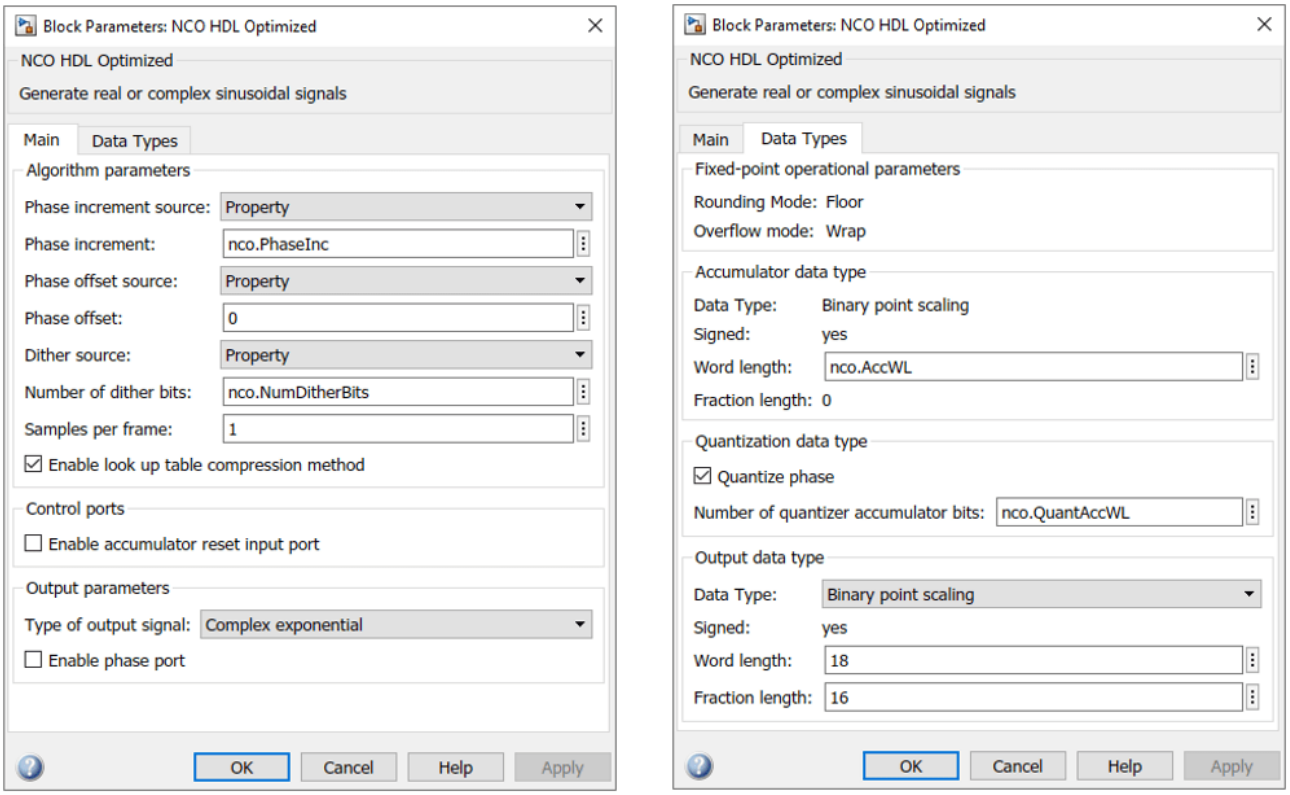
NCO Block Parameters

The NCO HDL Optimized block generates a complex phasor at the carrier frequency. This signal goes to a mixer that multiplies the phasor with the output signal. The output of the mixer is sampled at 122.88 Msps.

Specify the desired frequency resolution. Calculate the number of accumulator bits required to achieve the desired resolution, and define the number of quantized accumulator bits. The NCO uses the quantized output of the accumulator to address the sine lookup table. Also compute the phase increment the NCO must use to generate the specified carrier frequency. The NCO applies phase dither to the accumulator bits that are removed during quantization.

```
nco.Fd = 1;
nco.AccWL = nextpow2(FsIn*64/nco.Fd) + 1;
nco.QuantAccWL = 12;
nco.PhaseInc = round((Fc*2^nco.AccWL)/(FsIn*64));
nco.NumDitherBits = nco.AccWL-nco.QuantAccWL;
```

The NCO block in the model is configured with the parameters defined in the `nco` structure. This figure shows both tabs of the NCO HDL Optimized block parameters dialog.



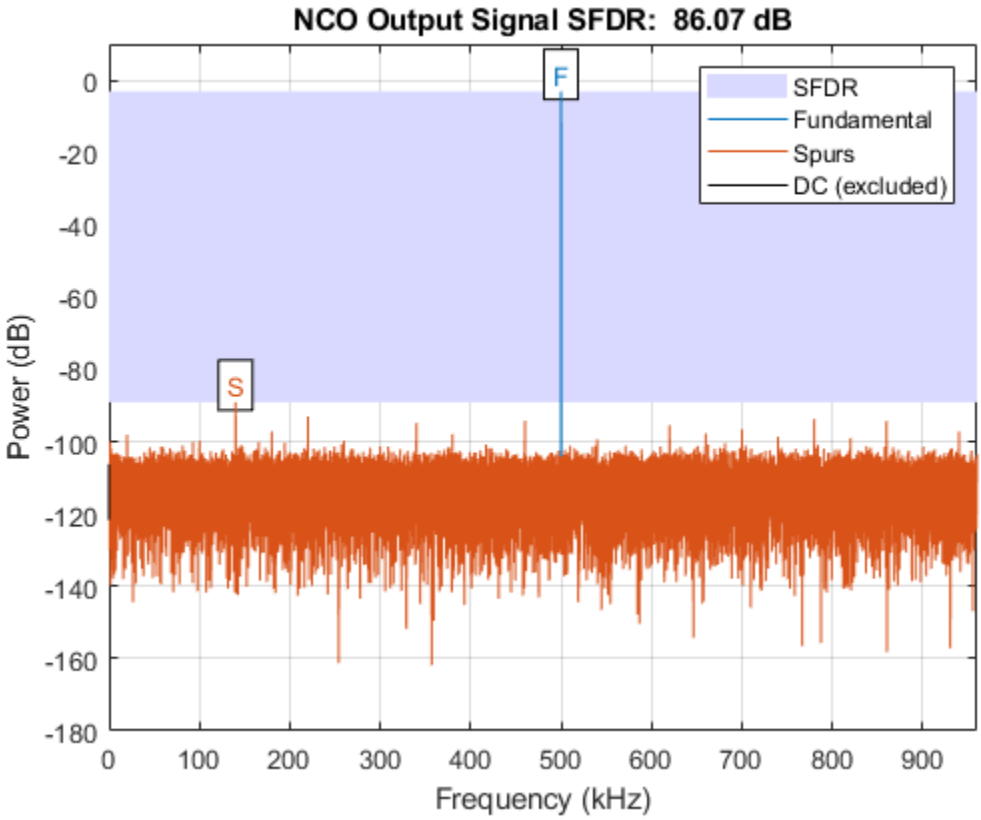
Sinusoid on Carrier Test

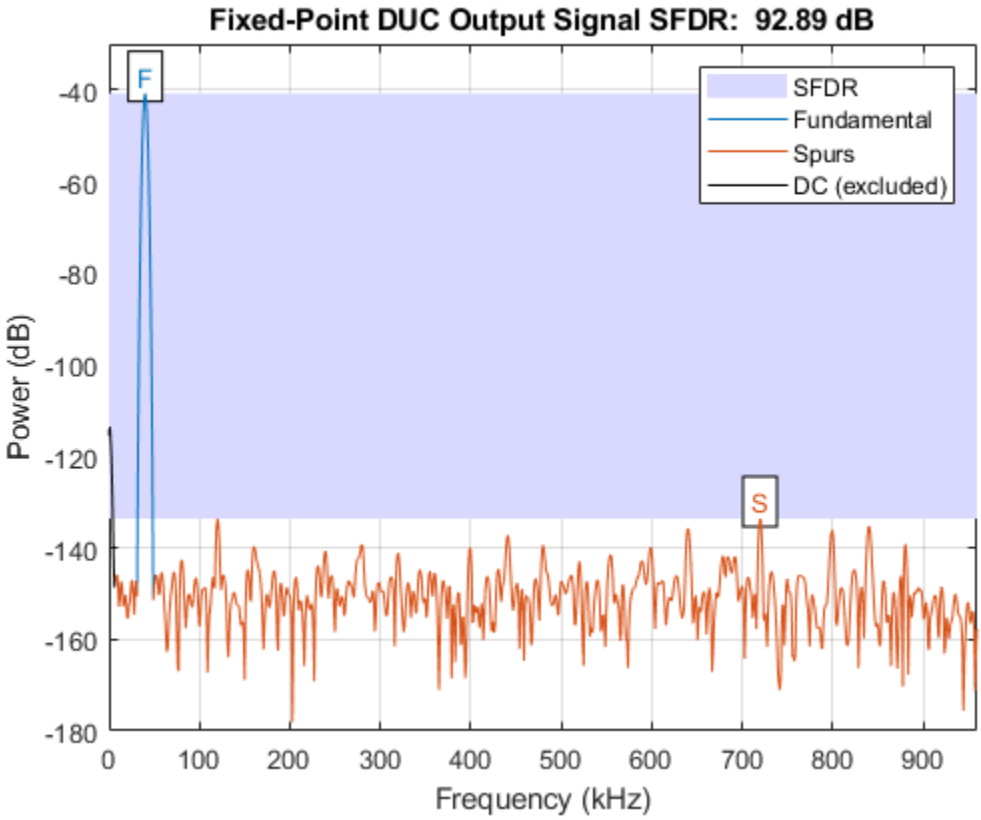
To test the DUC, pass a 40kHz sinusoid through the DUC and modulate the output signal onto the carrier frequency. Demodulate and resample the signal. Then measure the spurious free dynamic range (SFDR) of the resulting tone and the SFDR of the NCO output.

```
% Initialize random seed before executing any simulations.
rng(0);

% Generate a 40kHz test tone.
ducIn = DUCTestUtils.GenerateTestTone(40e3);
```

```
% Upconvert the test signal with the floating-point DUC.
ducTx = DUCTestUtils.UpConvert(ducIn,FsIn*64,Fc,ducFilterChain);
release(ducFilterChain);

% Down convert the output of DUC.
ducRx = DUCTestUtils.DownConvert(ducTx,FsIn*64,Fc);

% Upconvert the test signal by running the fixed-point Simulink model.
simOut = sim(modelName);

% Downconvert the output of DUC.
simTx = simOut.ducOut;
simRx = DUCTestUtils.DownConvert(simTx,FsIn*64,Fc);

% Measure the SFDR of the NCO, floating point DUC, and fixed-point DUC outputs.
results.sfdrNCO = sfdr(real(simOut.ncoOut),FsIn);
results.sfdrFloatDUC = sfdr(real(ducRx),FsIn);
results.sfdrFixedDUC = sfdr(real(simRx),FsIn);

disp('SFDR Measurements');
disp(['   Floating-point DUC SFDR: ',num2str(results.sfdrFloatDUC) ' dB']);
disp(['   Fixed-point NCO SFDR: ',num2str(results.sfdrNCO) ' dB']);
disp(['   Fixed-point DUC SFDR: ',num2str(results.sfdrFixedDUC) ' dB']);
fprintf(newline);

% Plot the SFDR of the NCO and fixed-point DUC outputs.
ducPlots.ncoOutSDFR = figure;
sfdr(real(simOut.ncoOut),FsIn);
DUCTestUtils.setPlotNameAndTitle(['NCO Output Signal ' get(gca,'Title').String]);

ducPlots.ducOutSDFR = figure;
sfdr(real(simRx),FsIn);
DUCTestUtils.setPlotNameAndTitle(['Fixed-Point DUC Output Signal ' get(gca,'Title').String]);
```

```
SFDR Measurements
   Floating-point DUC SFDR: 287.7416 dB
   Fixed-point NCO SFDR: 86.0718 dB
   Fixed-point DUC SFDR: 92.8885 dB
```


NCO Output Signal SFDR: 86.07 dB
F
S
SFDR
Fundamental
Spurs
DC (excluded)
Power (dB)
0
-20
-40
-60
-80
-100
-120
-140
-160
-180
0
100
200
300
400
500
600
700
800
900
Frequency (kHz)



LTE Signal Test

You can use an LTE test signal to perform more rigorous testing of the DUC. Generate a standard compliant LTE waveform by using LTE Toolbox™ functions. Then, upconvert the waveform with the DUC model. Use LTE Toolbox functions to measure the error vector magnitude (EVM) of the resulting signals.

```
rng(0);
% Execute this test only if you have the LTE Toolbox product.
if license('test','LTE_Toolbox')

    % Generate an LTE test signal by using LTE Toolbox functions.
    [ducIn, sigInfo] = DUCTestUtils.GenerateLTETestSignal();

    % Upconvert the signal with the floating-point DUC and modulate onto carrier.
    ducTx = DUCTestUtils.UpConvert(ducIn,FsIn*64,Fc,ducFilterChain);
    release(ducFilterChain);

    % Add noise to the transmit signal.
    ducTxAddNoise = DUCTestUtils.AddNoise(ducTx);

    % Downconvert the received signal.
    ducRx = DUCTestUtils.DownConvert(ducTxAddNoise,FsIn*64,Fc);

    % Upconvert the signal by using the Simulink model.
    simOut = sim(modelName);
```

```
    % Add noise to the transmit signal.
    simTx = simOut.ducOut;
    simTxAddNoise = DUCTestUtils.AddNoise(simTx);

    % Downconvert the received signal.
    simRx = DUCTestUtils.DownConvert(simTxAddNoise,FsIn*64,Fc);

    results.evmFloat = DUCTestUtils.MeasureEVM(sigInfo,ducRx);
    results.evmFixed = DUCTestUtils.MeasureEVM(sigInfo,simRx);

    disp('LTE EVM Measurements');
    disp(['   Floating-point DUC RMS EVM: '  num2str(results.evmFloat.RMS*100,3) '%']);
    disp(['   Floating-point DUC Peak EVM: ' num2str(results.evmFloat.Peak*100,3) '%']);
    disp(['   Fixed-point DUC RMS EVM: '     num2str(results.evmFixed.RMS*100,3) '%']);
    disp(['   Fixed-point DUC Peak EVM: '    num2str(results.evmFixed.Peak*100,3) '%']);
    fprintf(newline);

end
```

```
LTE EVM Measurements
   Floating-point DUC RMS EVM: 0.782%
   Floating-point DUC Peak EVM: 2.42%
   Fixed-point DUC RMS EVM: 0.755%
   Fixed-point DUC Peak EVM: 2.76%
```

HDL Code Generation and FPGA Implementation

To generate the HDL code for this example you must have the HDL Coder™ product. Use the `makehdl` and `makehdltb` commands to generate HDL code and an HDL test bench for the `HDL_DUC` subsystem. The DUC was synthesized on a Xilinx® Zynq®-7000 ZC706 evaluation board. The table shows the post place-and-route resource utilization results. The design met timing with a clock frequency of 188 MHz.

```
T = table(...
    categorical({'LUT'; 'LUTRAM'; 'FF'; 'BRAM'; 'DSP'}),...
    categorical({'3497'; '370'; '4871'; '0.5'; '10'}),...
    'VariableNames',{'Resource','Usage'})
```

```
T =

  5x2 table

    Resource    Usage
    ________    _____

     LUT        3497
     LUTRAM     370
     FF         4871
     BRAM       0.5
     DSP        10
```

Links to Category Pages

- “Signal Management Library” on page 22-2
- “Sinks Library” on page 22-3
- “Math Functions Library” on page 22-4
- “Filtering Library” on page 22-5

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 23-2
- “Controlling Design Specifications in Lowpass FIR Design” on page 23-7
- “Designing Filters with Non-Equiripple Stopband” on page 23-12
- “Minimizing Lowpass FIR Filter Length” on page 23-16

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

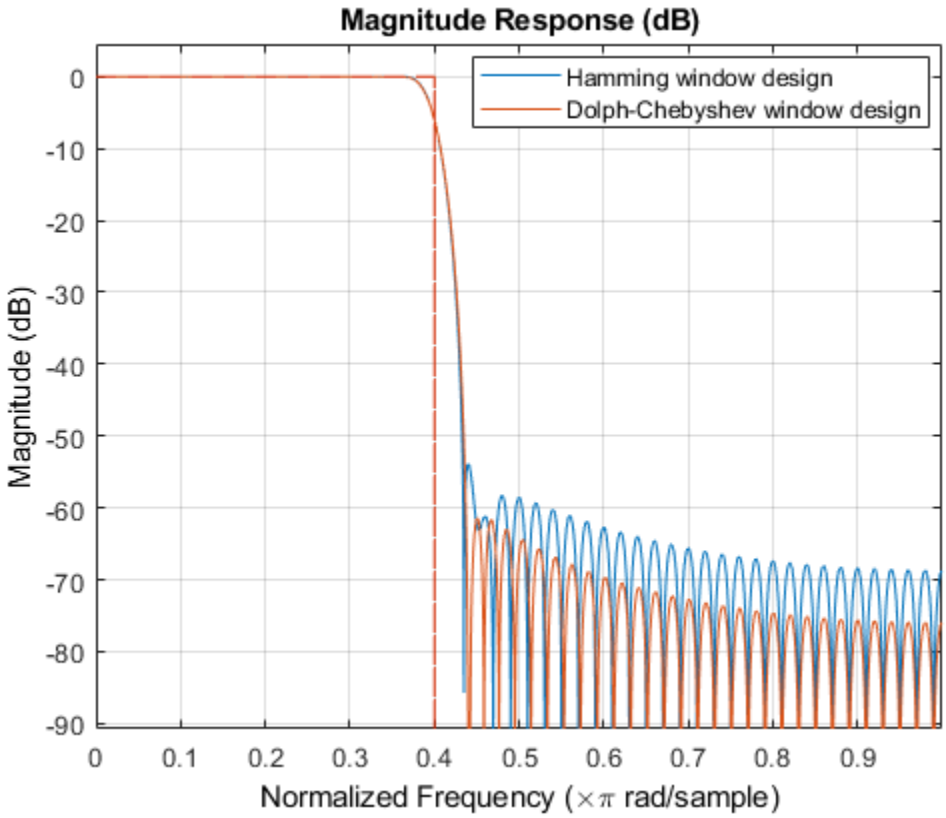
FIR filters are widely used due to the powerful design algorithms that exist for them, their inherent stability when implemented in non-recursive form, the ease with which one can attain linear phase, their simple extensibility to multirate cases, and the ample hardware support that exists for them among 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);
```

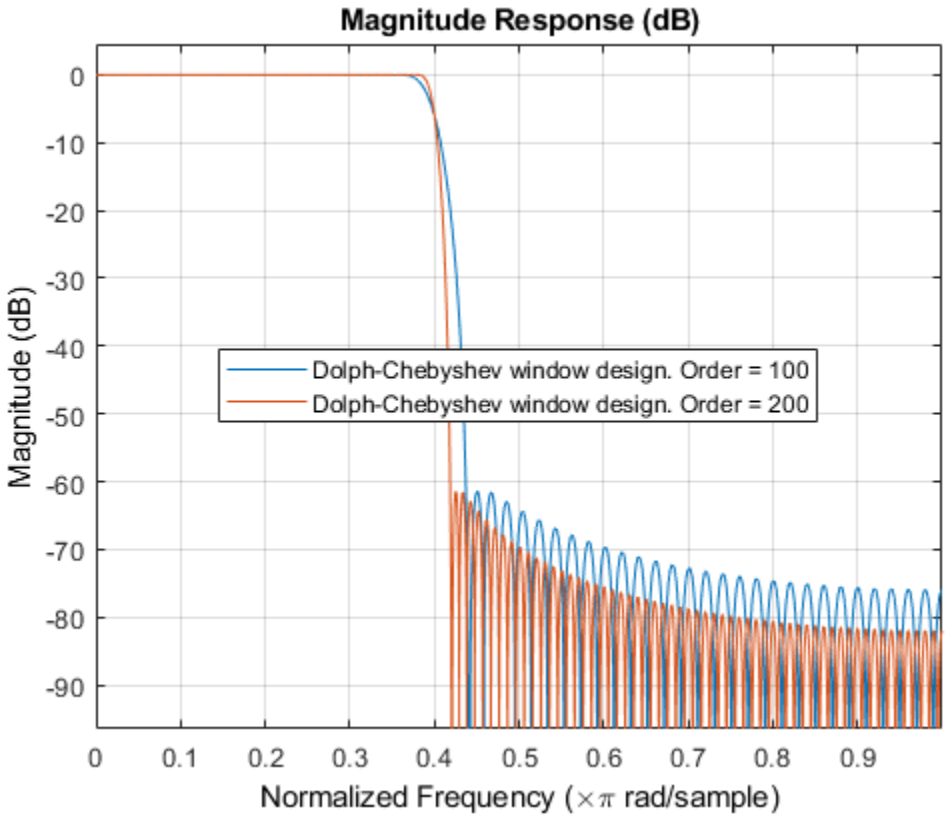
We can design this lowpass filter using the window method. For example, we can use a Hamming window or a Dolph-Chebyshev window:

```
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
```

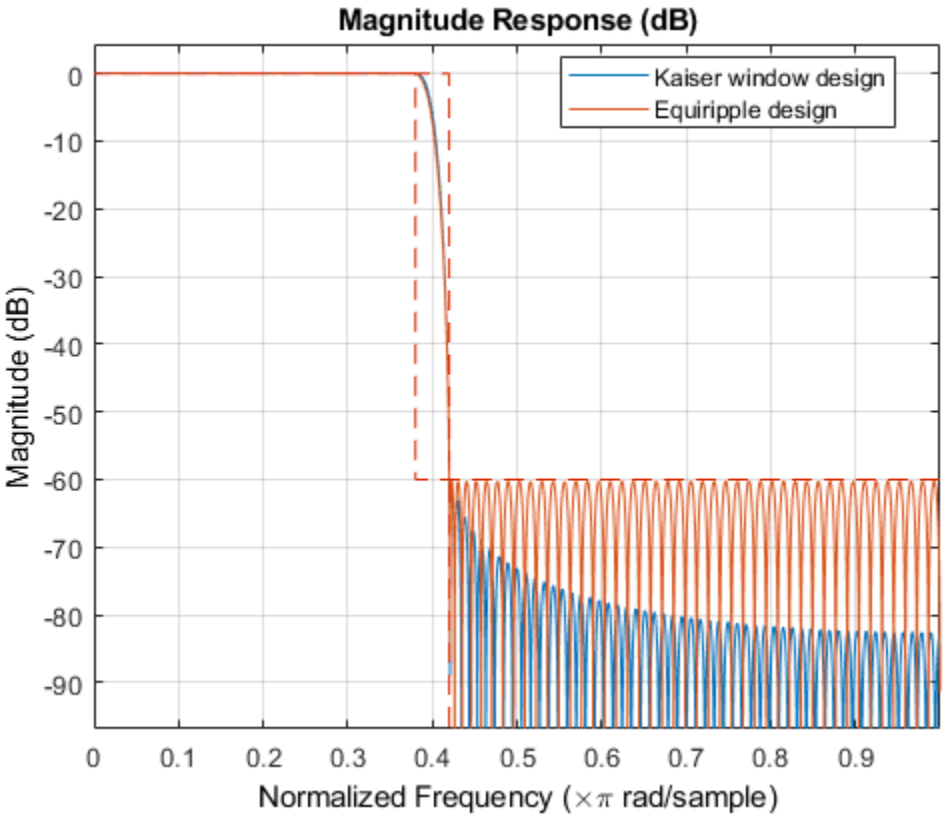
```
Transition Width : 0.04


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

`fdesign` | `design` | `fdesign.lowpass` | `fvtool`

More About

- “Design a Filter in Fdesign — Process Overview” on page 5-2
- “Controlling Design Specifications in Lowpass FIR Design” on page 23-7
- “Designing Filters with Non-Equiripple Stopband” on page 23-12
- “Minimizing Lowpass FIR Filter Length” on page 23-16

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:

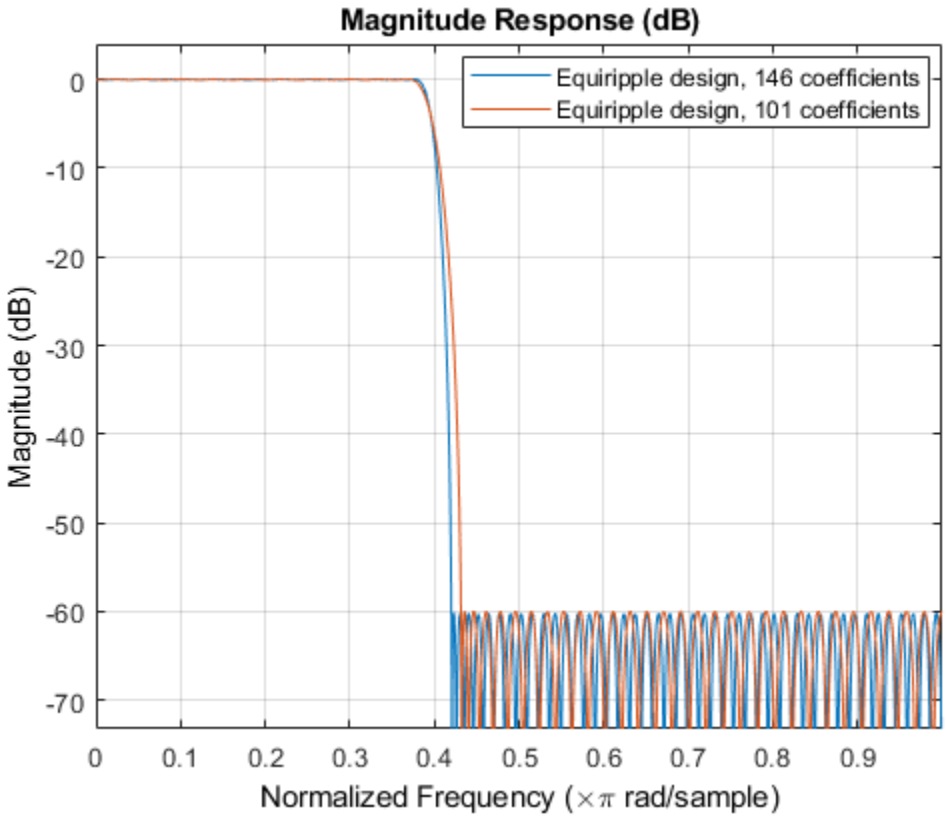
```
Hd2 = design(Hf,'equiripple','systemobject',true);
```

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.

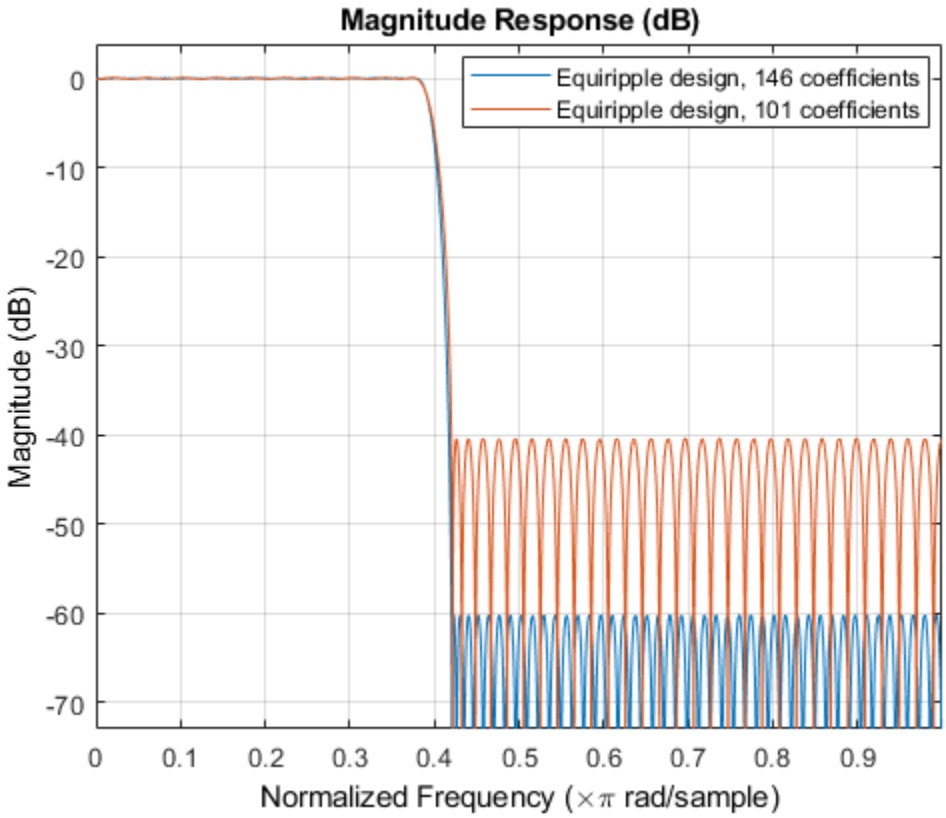
Controlling the Transition Region Width

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')
```



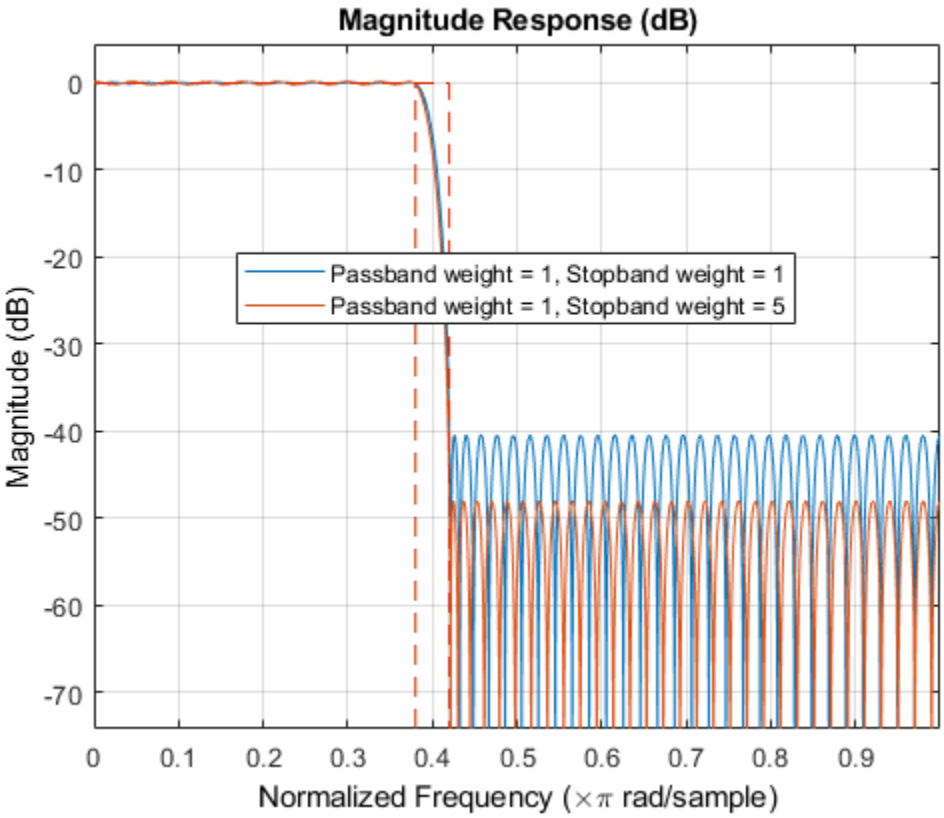
The differences between using 146 coefficients and using 101 coefficients is reflected in a larger passband ripple and a smaller stopband attenuation.

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)
```

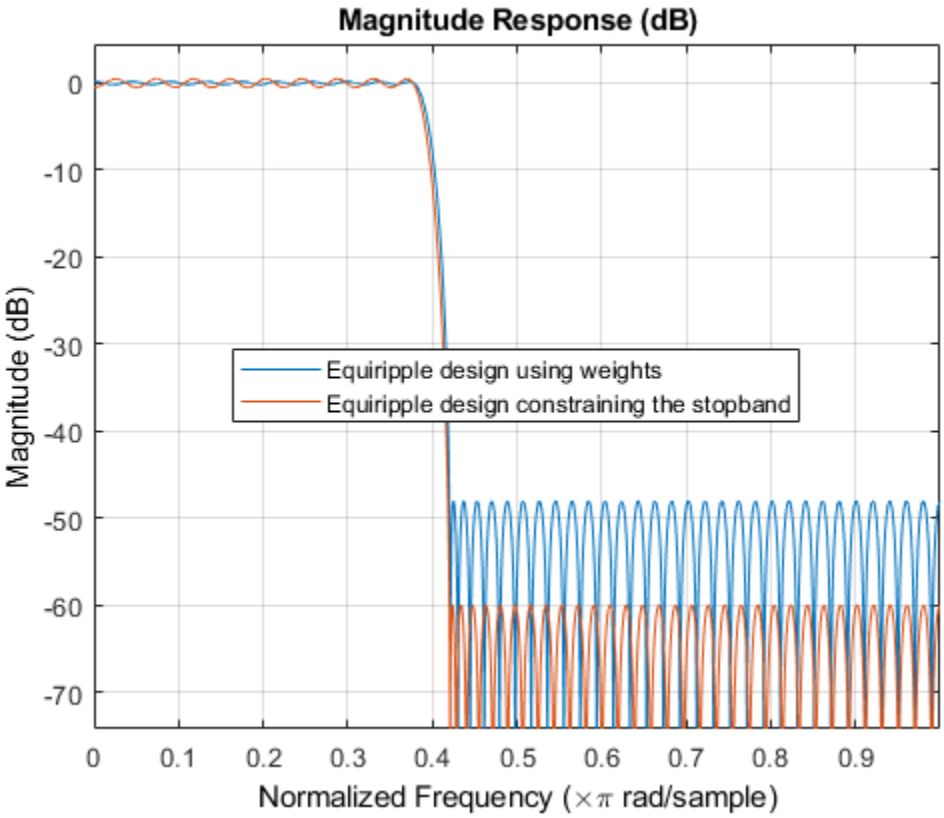
```
ans =
Sample Rate      : N/A (normalized frequency)
Passband Edge    : 0.38
3-dB Point       : 0.39143
6-dB Point       : 0.39722
Stopband Edge    : 0.42
Passband Ripple  : 0.34529 dB
Stopband Atten.  : 48.0068 dB
Transition Width : 0.04
```

```
  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.

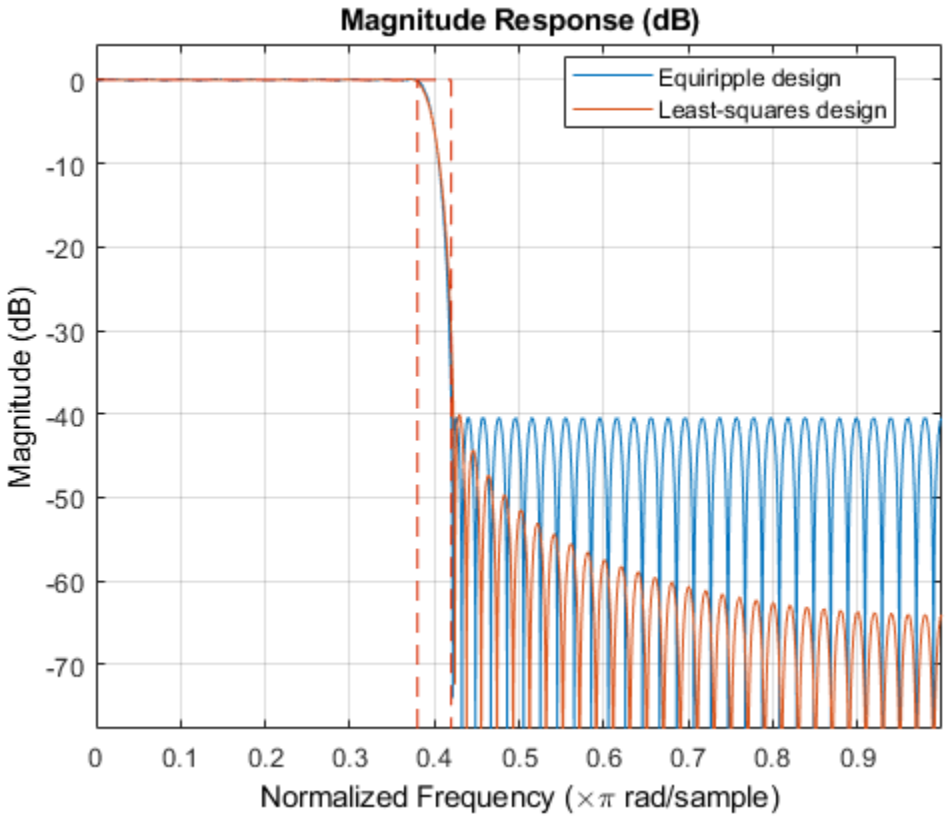
Optimal Non-Equiripple Lowpass Filters

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')
```

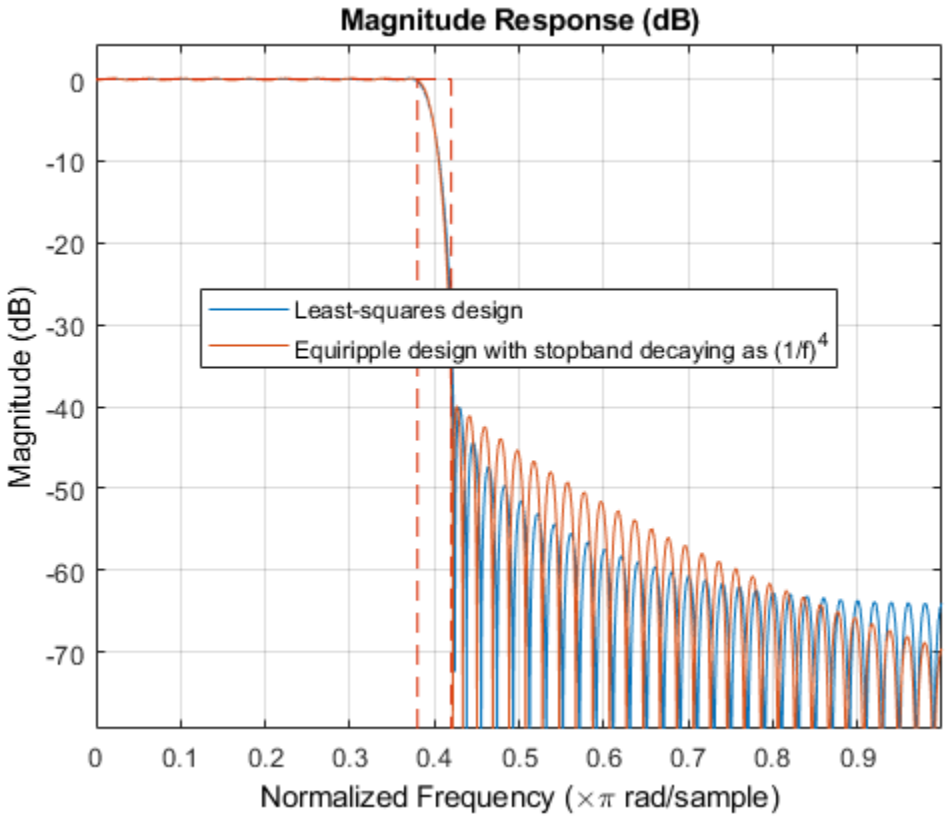


Notice how the attenuation in the stopband increases with frequency for the least-squares designs while it remains constant for the equiripple design. The increased attenuation in the least-squares case minimizes the energy in that band of the signal to be filtered.

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
```

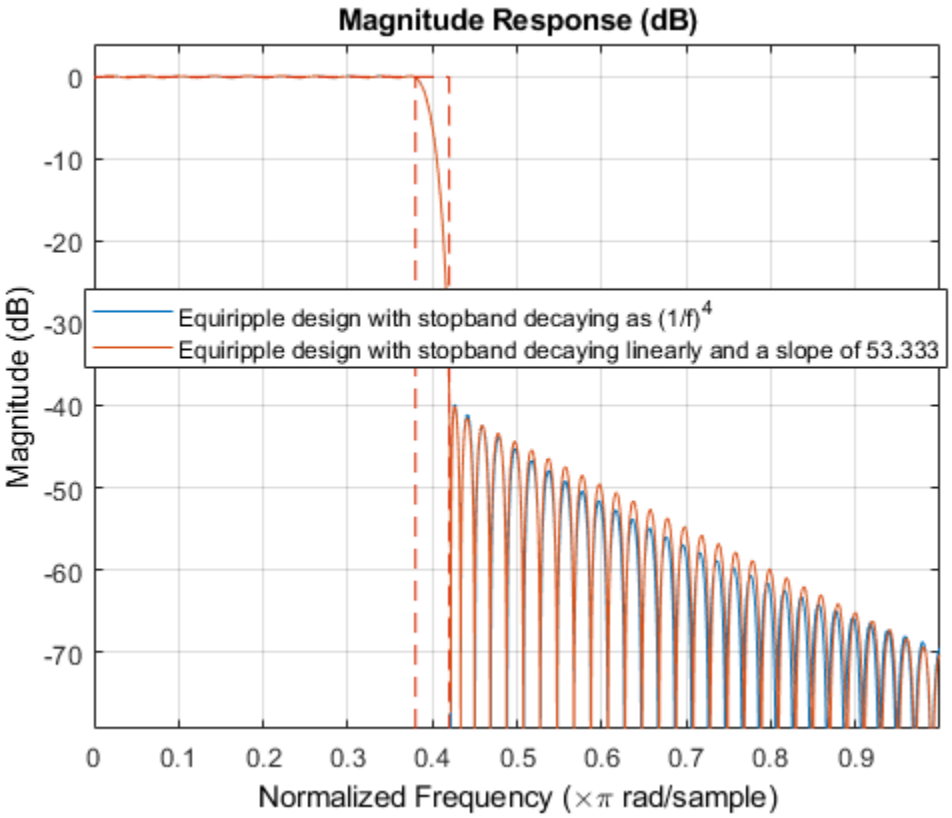
```
ans = 0.3504
```

```
meq.Apass
```

```
ans = 0.1867
```

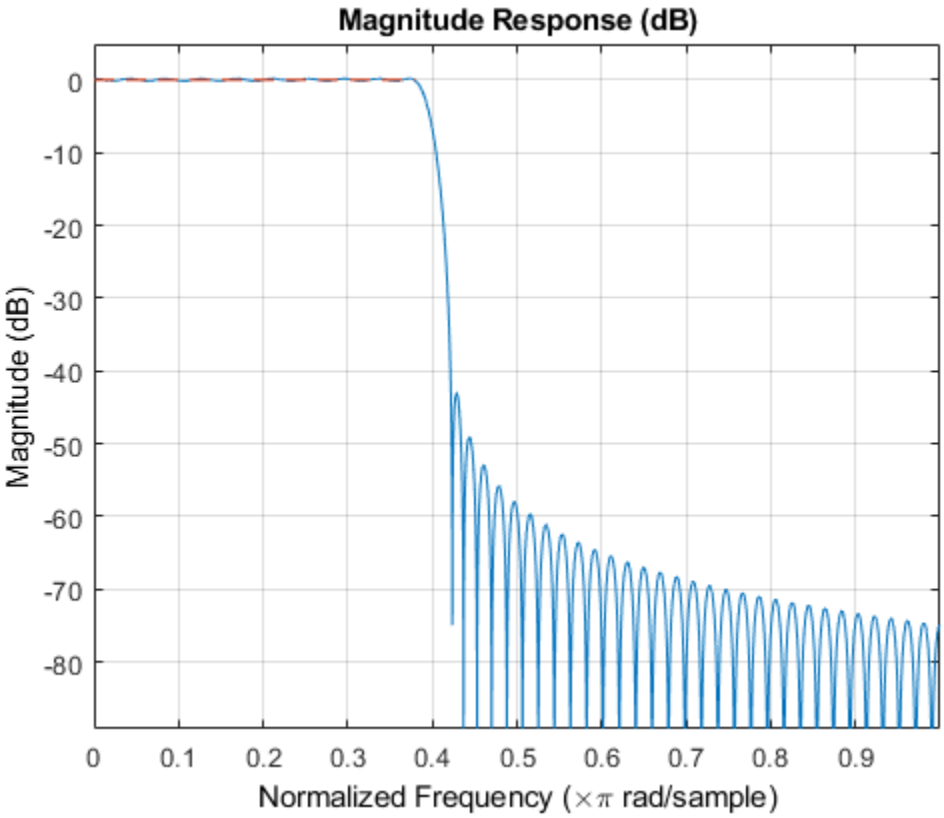
Filters with a stopband that decays as (1/f)^M will decay at 6M dB per octave. Another way of shaping the stopband is using a linear decay. For example given an approximate attenuation of 38 dB at 0.4*pi, if an attenuation of 70 dB is desired at pi, and a linear decay is to be used, the slope of the 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.

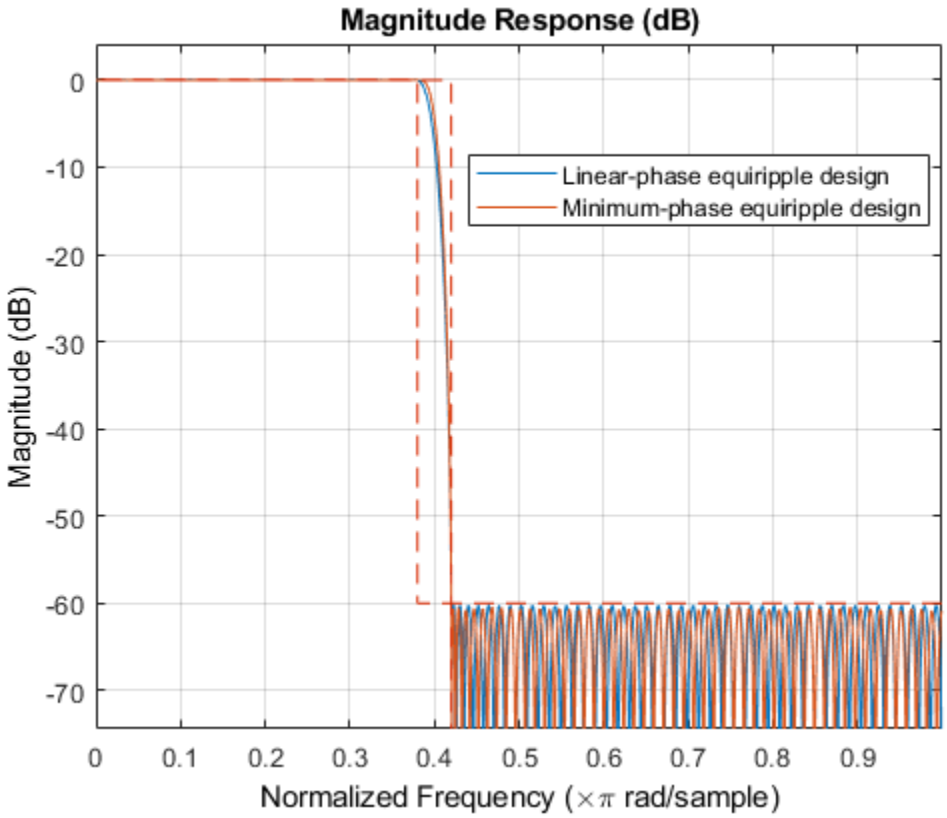
Minimum-Phase Lowpass Filter Design

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);
```

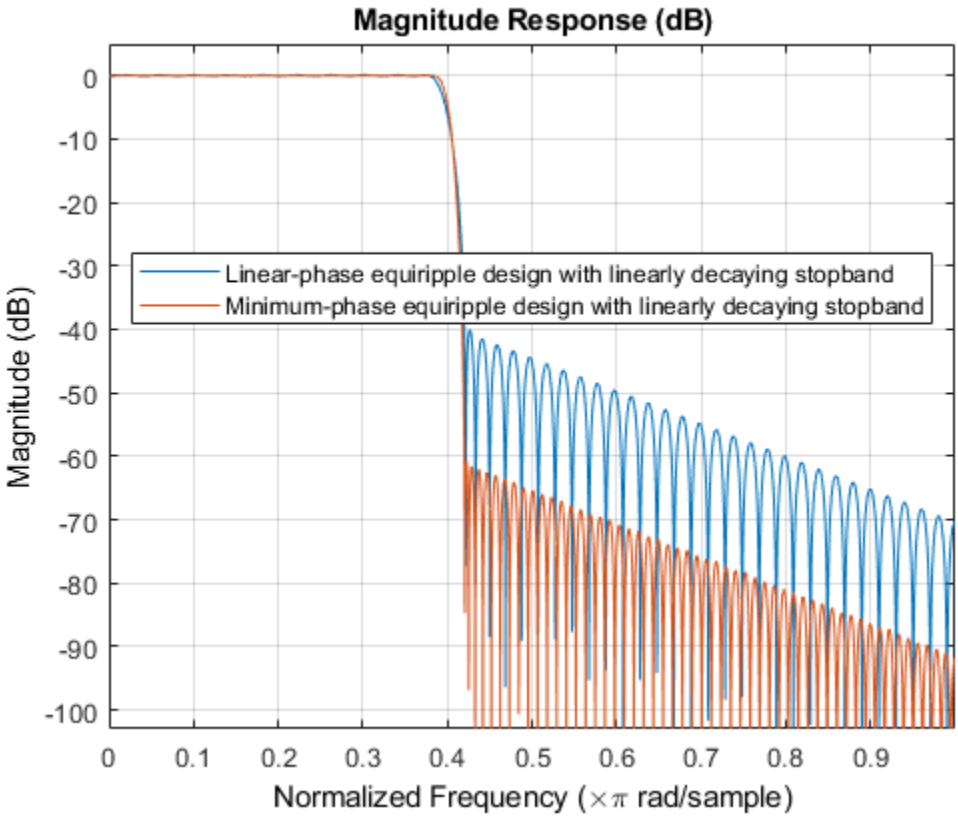
So far, we have only considered linear-phase designs. Linear phase is desirable in many applications. Nevertheless, if linear phase is not a requirement, minimum-phase designs can provide significant improvements over linear phase counterparts. For instance, returning to the minimum order case, a minimum-phase/minimum-order 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.

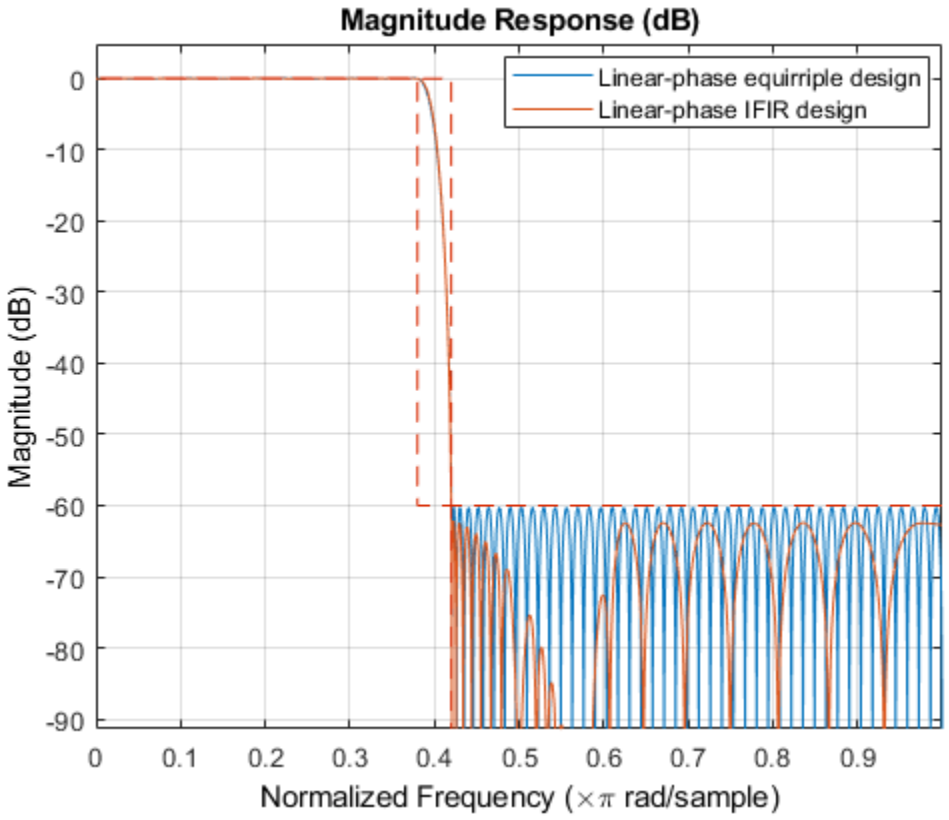
```
setspecs(Hf,'N,Fp,Fst',N,Fp,Fst);
Hd3 = design(Hf,'equiripple','StopbandShape','linear',...
    'StopbandDecay',53.333,'systemobject',true);
setspecs(Hf,'Fp,Fst,Ap,Ast',Fp,Fst,Ap,Ast);
Hd4 = design(Hf,'equiripple','StopbandShape','linear',...
    'StopbandDecay',53.333,'minphase',true,'systemobject',true);
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 24-2
- “Importing a Filter Design” on page 24-24

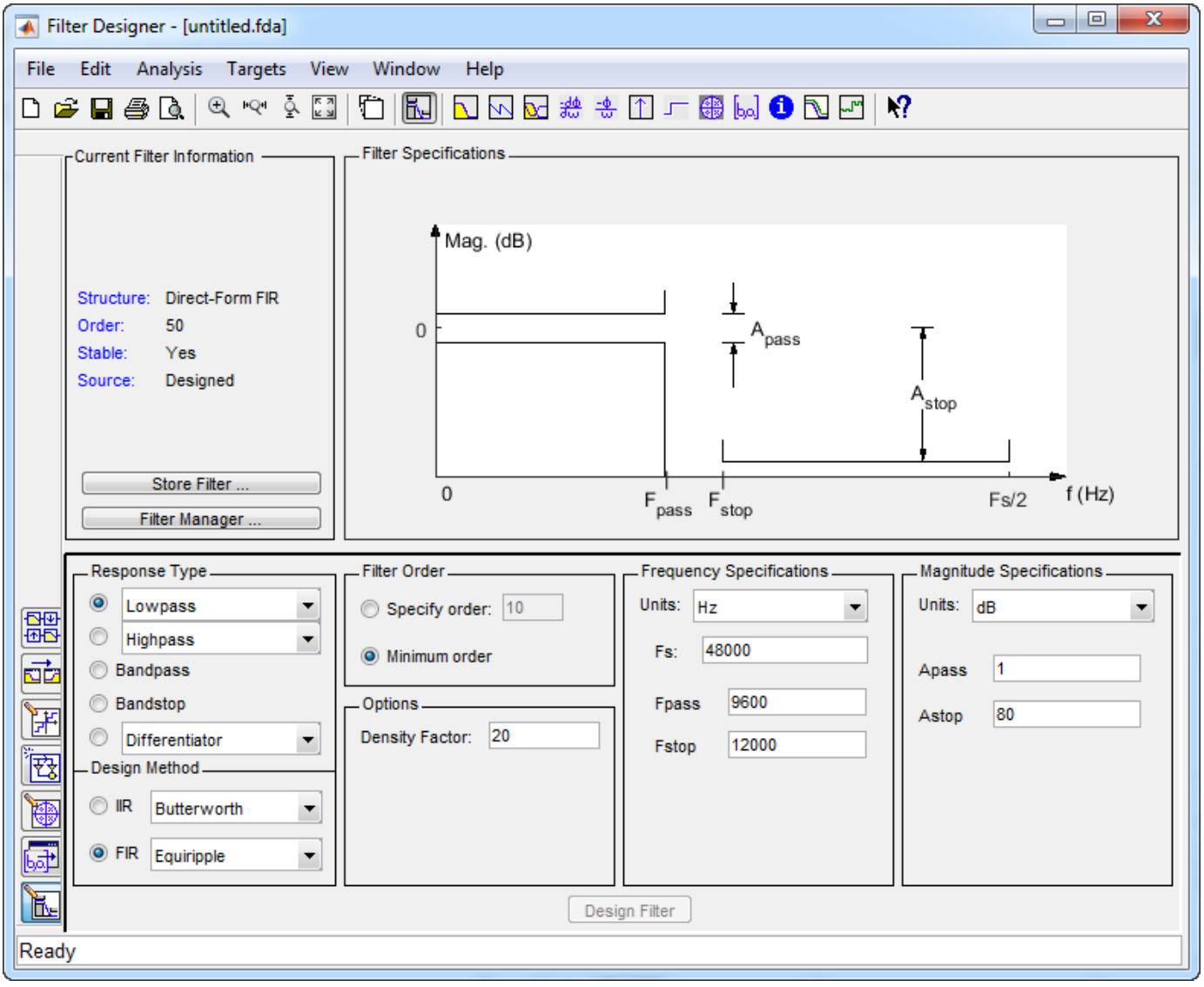
Using Filter Designer

To open filter designer, type

```
filterDesigner
```

at the MATLAB command prompt.

The filter designer opens with the `Design filter` panel displayed.



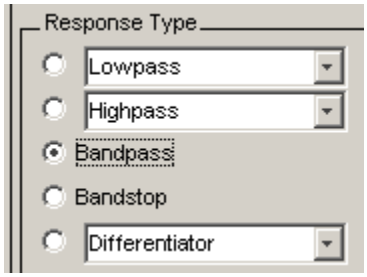
Note that when you open filter designer, **Design Filter** is not enabled. You must make a change to the default filter design in order to enable **Design Filter**. This is true for each time you want to change the filter design. Changes to radio button items or drop down menu items such as those under **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.

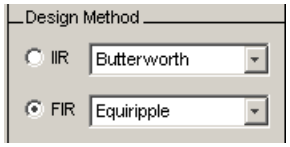


Note Not all filter design methods are available for all response types. Once you choose your response type, this may restrict the filter design methods available to you. Filter design methods that are not available for a selected response type are removed from the Design Method region of the app.

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.



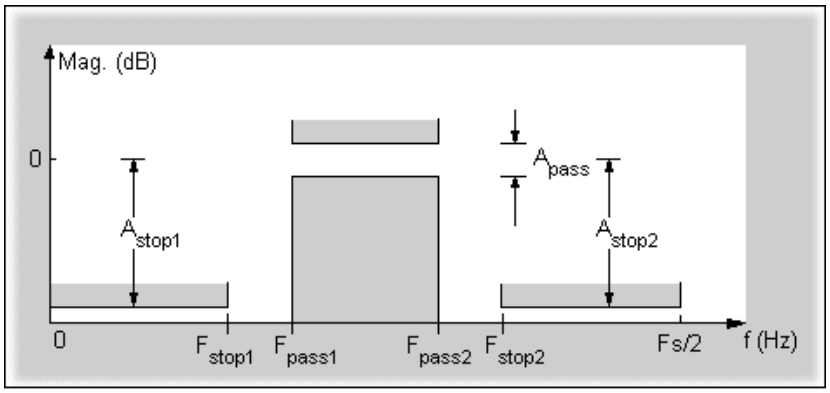
Setting the Filter Design Specifications

- “Viewing Filter Specifications” on page 24-4
- “Filter Order” on page 24-4
- “Options” on page 24-5
- “Bandpass Filter Frequency Specifications” on page 24-5
- “Bandpass Filter Magnitude Specifications” on page 24-6

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**.

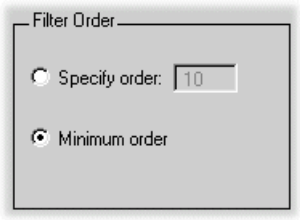


Filter Order

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.

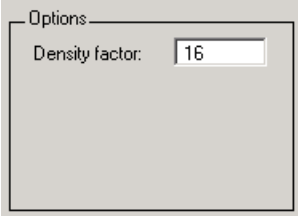
Options

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

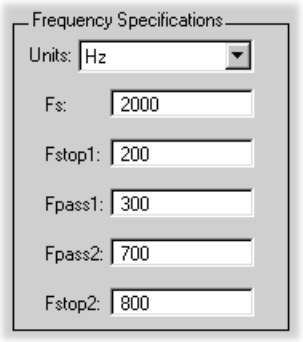
- Units of frequency:
  - Hz
  - kHz
  - MHz
  - GHz
  - Normalized (0 to 1)
- Sampling frequency
- Passband frequencies
- Stopband frequencies

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.



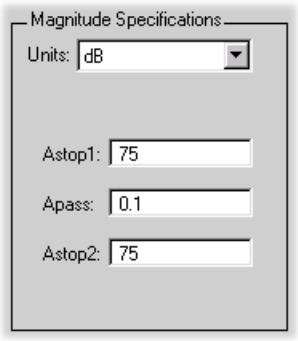
Bandpass Filter Magnitude Specifications

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 24-7
- “Using Data Tips” on page 24-8
- “Drawing Spectral Masks” on page 24-9
- “Changing the Sampling Frequency” on page 24-10
- “Displaying the Response in FVTool” on page 24-10

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.

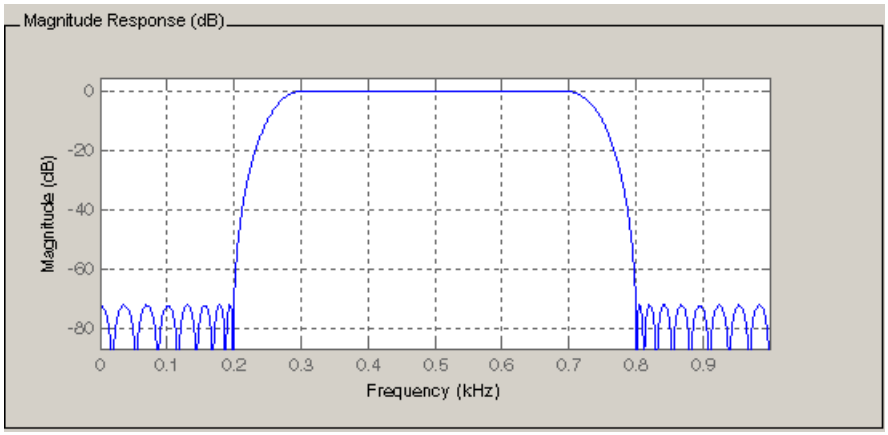
For descriptions of the above responses and their associated toolbar buttons and other filter designer toolbar buttons, see `FVTool`.

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.

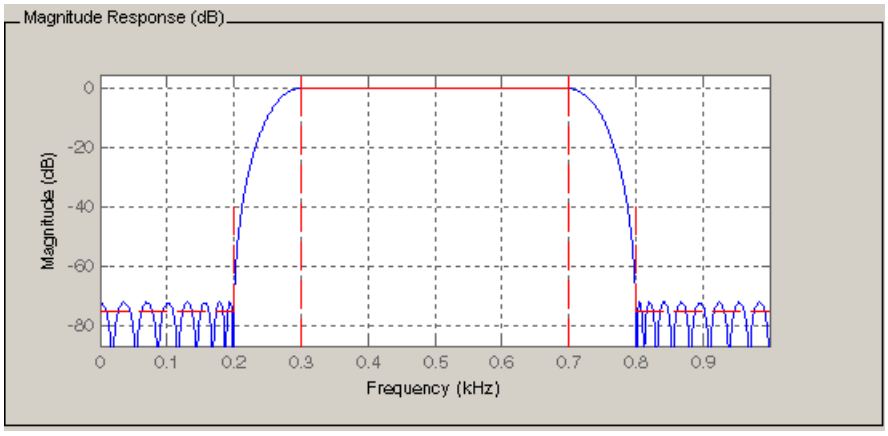
For all the analysis methods, except zero-phase response, you can access them from the **Analysis** menu, the Analysis Parameters dialog box from the context menu, or by using the toolbar buttons. For zero-phase, right-click the *y*-axis of the plot and select **Zero-phase** from the context menu.

For example, to look at the filter's magnitude response, select the **Magnitude Response** button on the toolbar.



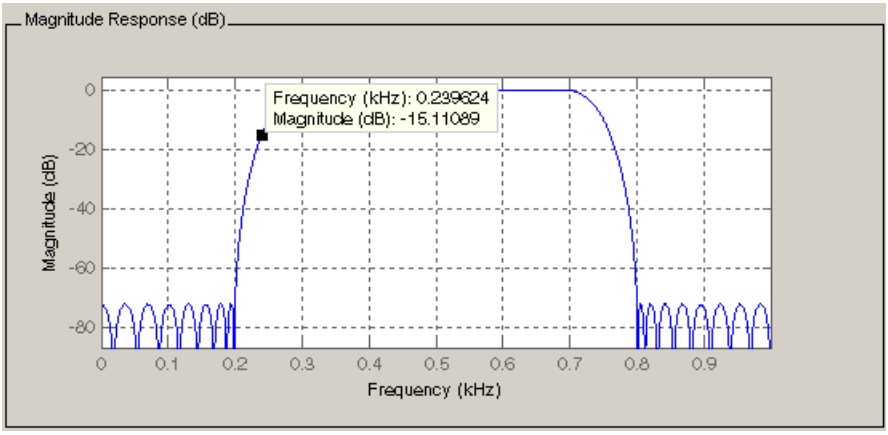
You can also overlay the filter specifications on the Magnitude plot by selecting **View** > **Specification Mask**.

Note You can use specification masks in FVTool only if FVTool was launched from filter designer.



Using Data Tips

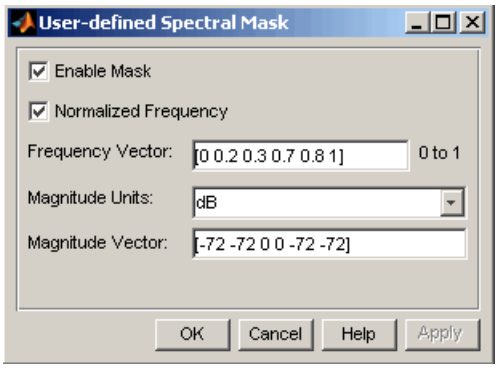
You can click the response to add plot data tips that display information about particular points on the response.



For information on using data tips, see “Interactively Explore Plotted Data”.

Drawing Spectral Masks

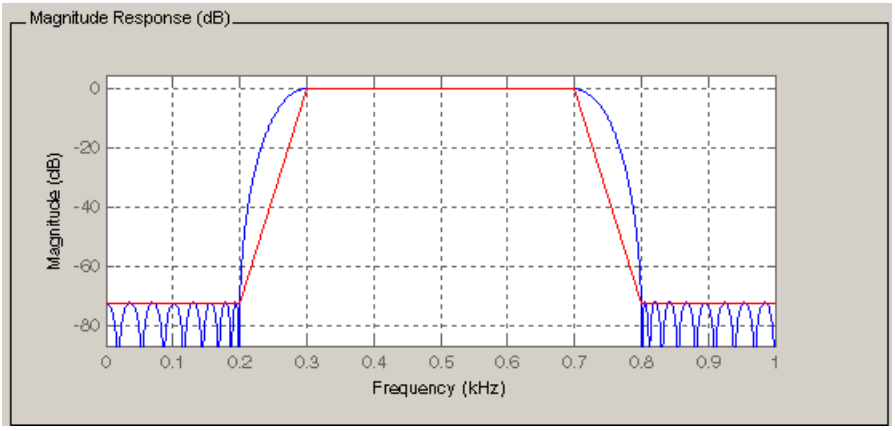
To add spectral masks or rejection area lines to your magnitude plot, click **View > User-defined Spectral Mask**.



The mask is defined by a frequency vector and a magnitude vector. These vectors must be the same length.

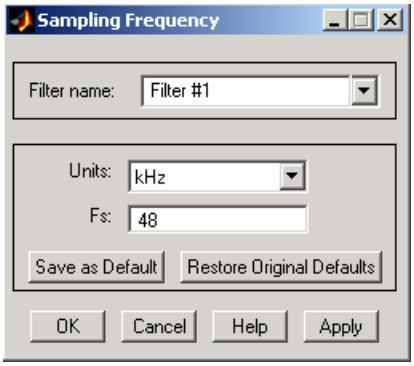
- `Enable Mask` — Select to turn on the mask display.
- `Normalized Frequency` — Select to normalize the frequency between 0 and 1 across the displayed frequency range.
- `Frequency Vector` — Enter a vector of *x*-axis frequency values.
- `Magnitude Units` — Select the desired magnitude units. These 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**.

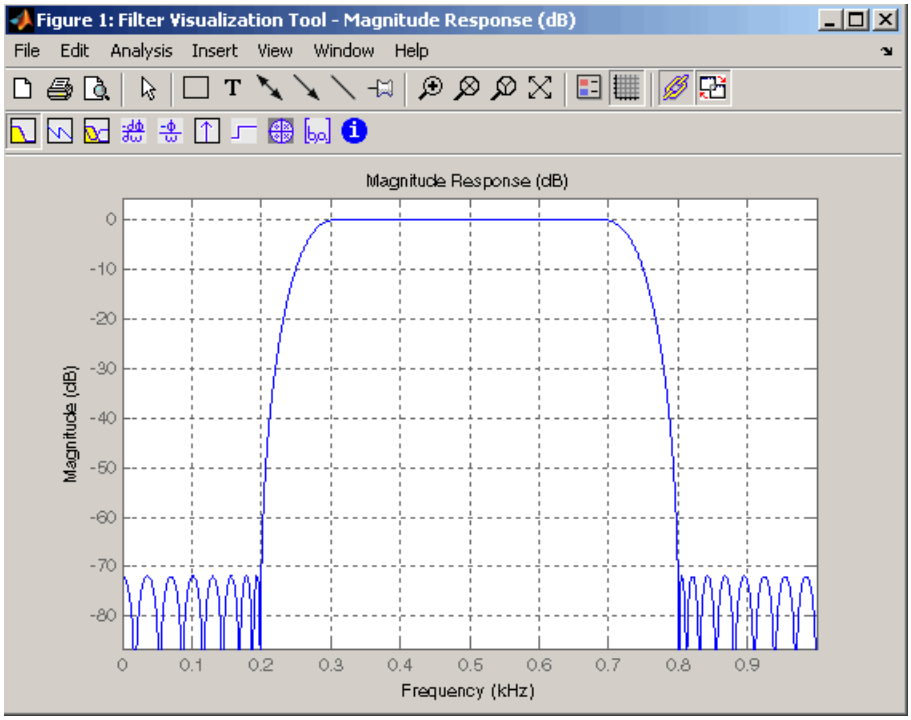
Displaying the Response in FVTool

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". 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 24-11
- "Changing the Pole-Zero Plot" on page 24-12

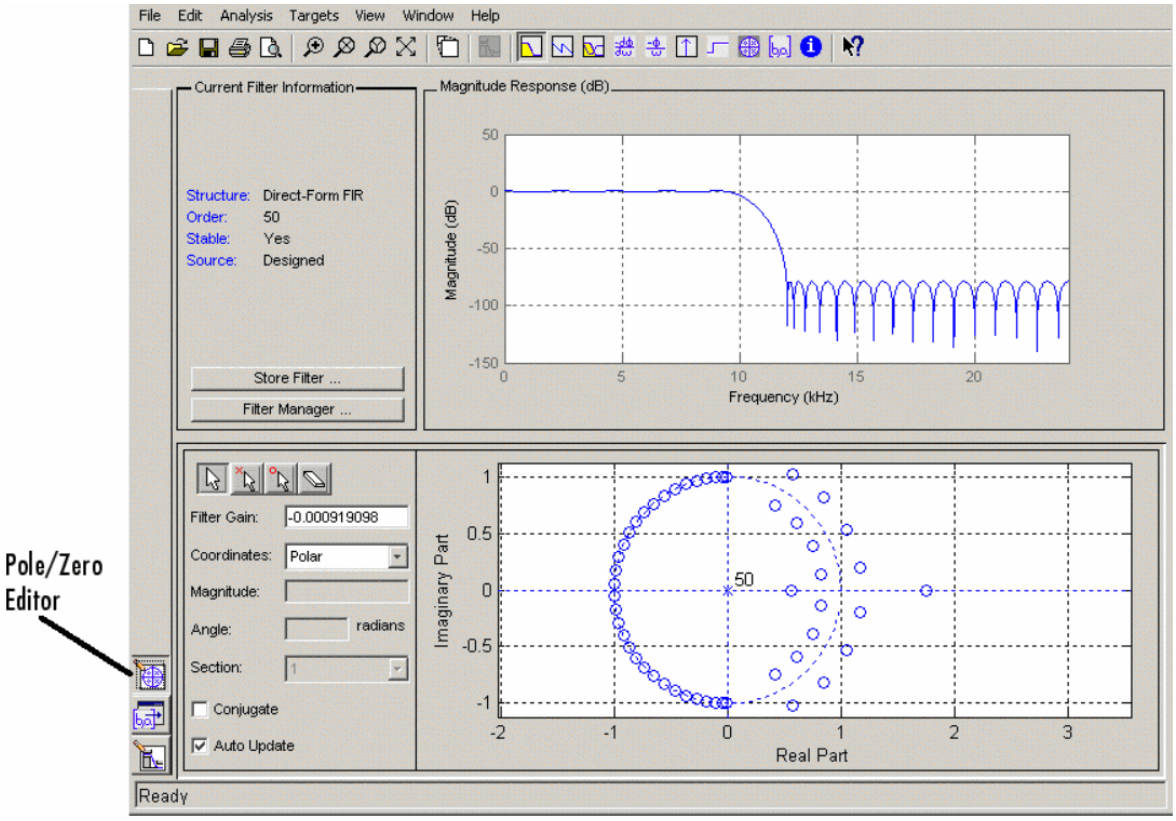
Displaying the Pole-Zero Plot

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.

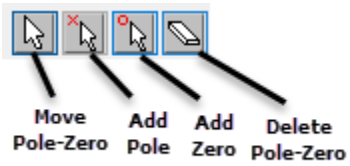
Click the **Pole/Zero Editor** button in the sidebar or select **Edit > Pole/Zero Editor** to display this panel.



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.

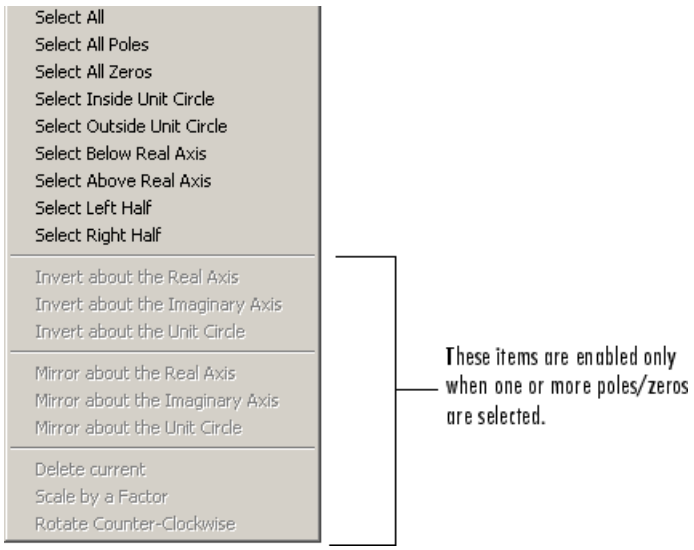


The following plot parameters and controls are located to the left of the pole/zero plot and below the plot mode buttons.

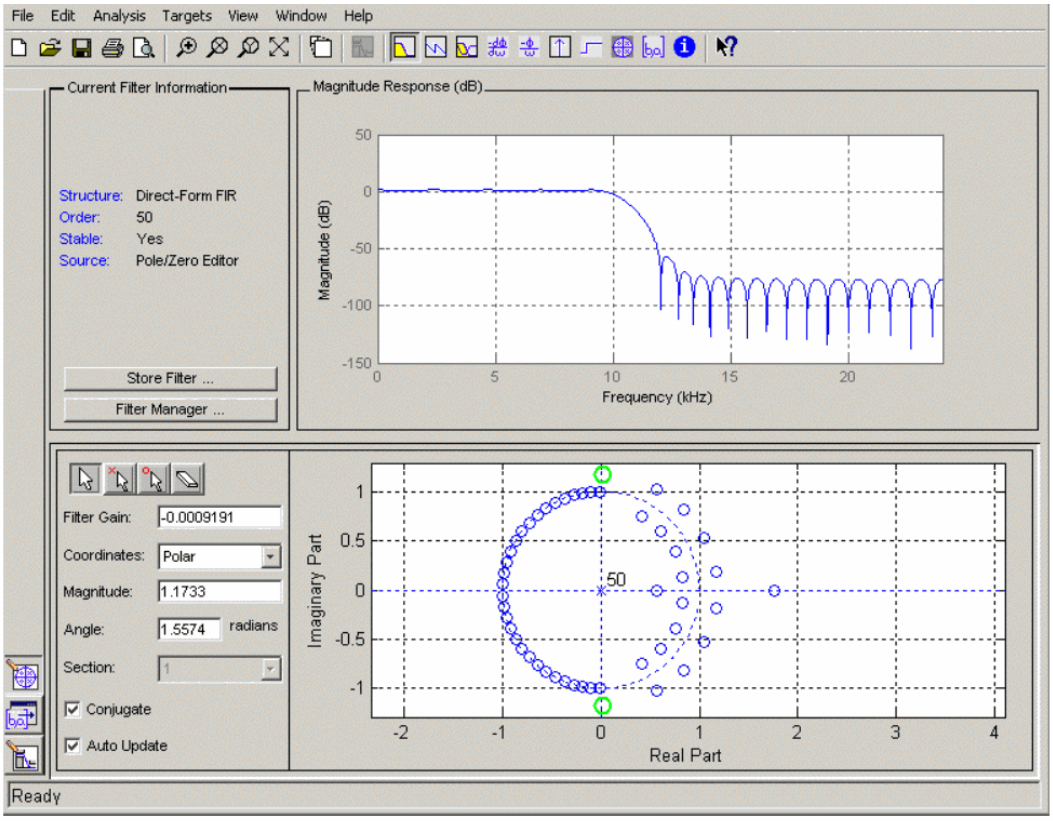
- **Gain** — factor to compensate for the filter's pole(s) and zero(s) gains
- **Coordinates** — units (`Polar` or `Rectangular`) of the selected pole or zero
- **Magnitude** — if polar coordinates is selected, magnitude of the selected pole or zero
- **Angle** — if polar coordinates is selected, angle 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 24-14
- “Converting to Second-Order Sections” on page 24-15

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.

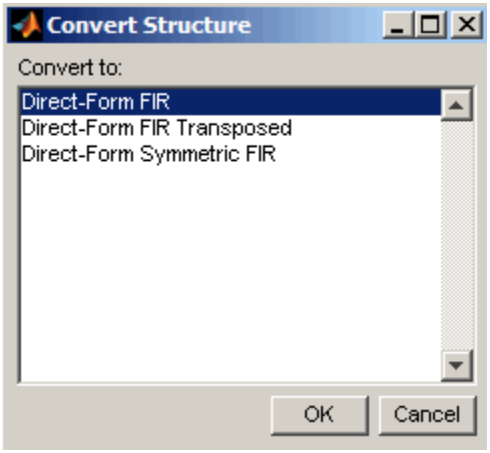
In addition, the following conversions are available for particular classes of filters:

- 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 24-16
- "Exporting Coefficients to an ASCII File" on page 24-16
- "Exporting Coefficients or Objects to a MAT-File" on page 24-16
- "Exporting to a Simulink Model" on page 24-17
- "Other Ways to Export a Filter" on page 24-19

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.

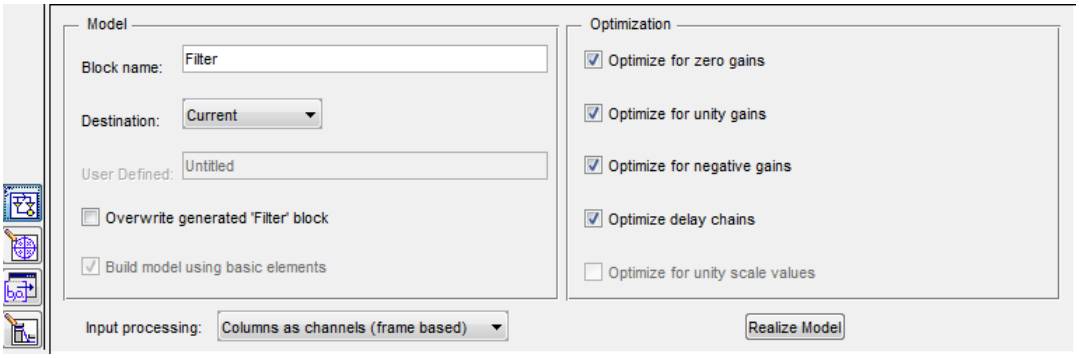
Exporting to a Simulink Model

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.

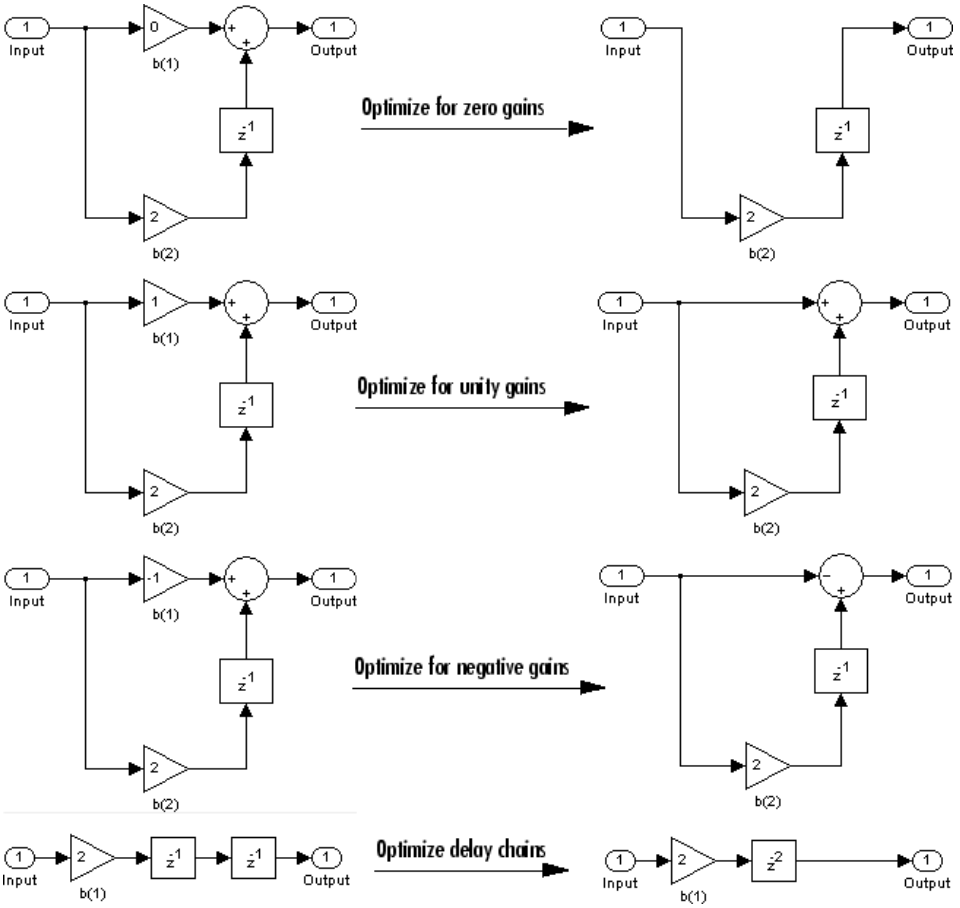
1. After designing your filter, click the **Realize Model** sidebar button or select **File** > **Export to Simulink Model**. The Realize Model panel is displayed.



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

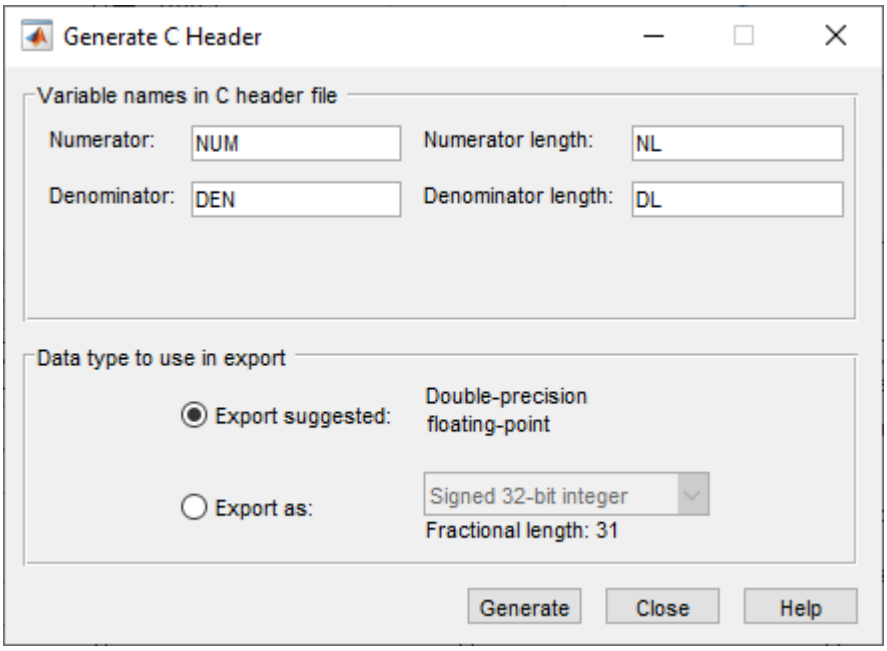
You can also send your filter to a C header file or generate MATLAB code to construct your filter from the command line. For detailed instructions, see the following sections:

- “Generating a C Header File” on page 24-19
- “Generating MATLAB Code” on page 24-20

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)
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 **Generate** to generate the C header file. Click **Close** to close the dialog box.

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.

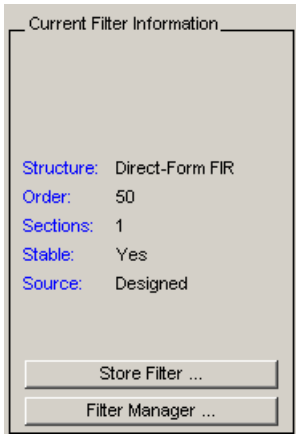
```
function Hd = ExFilter
%EXFILTER Returns a discrete-time filter object.

% 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]
```

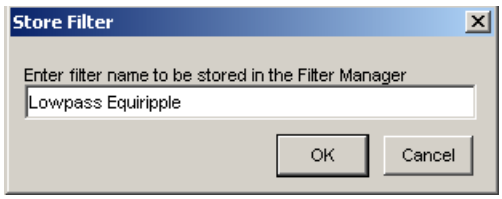
Managing Filters in the Current Session

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.

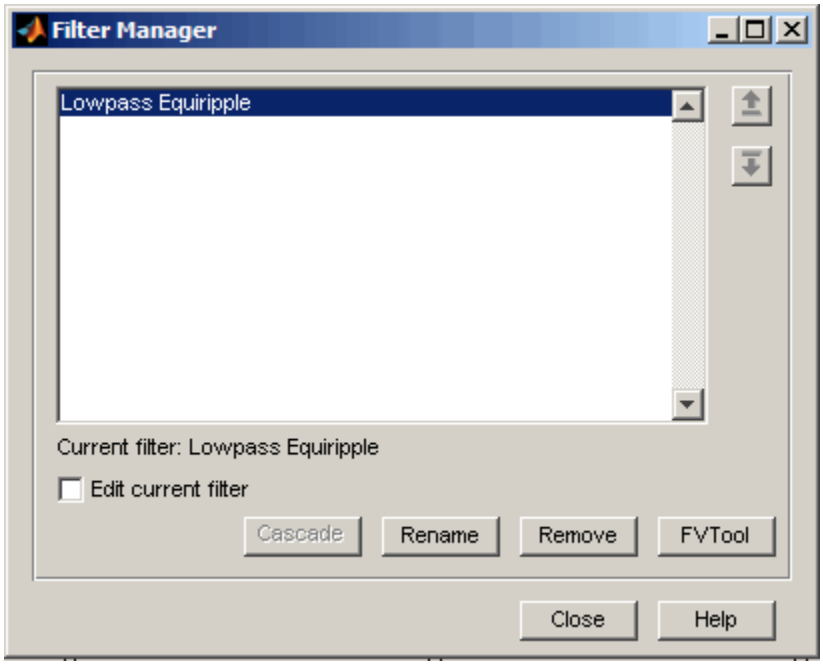
You store and access saved filters with the **Store filter** and **Filter Manager** buttons, respectively, in the Current Filter Information pane.



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.

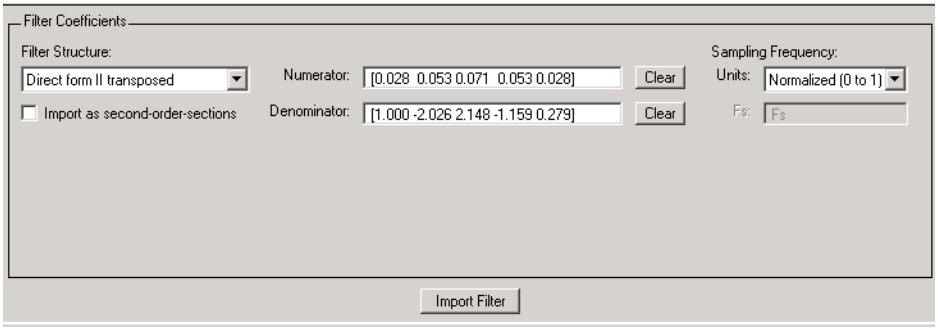
You can load existing sessions into the Filter Design and Analysis Tool by selecting the **Open session** button, **Open session** button, or **File > Open session** . A Load Filter Design Session browser opens that allows you to select from your previously saved filter design sessions.

Importing a Filter Design

In this section...
“Import Filter Panel” on page 24-24 “Filter Structures” on page 24-24

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

$$H(z) = \frac{b(1) + b(2)z^{-1} + b(3)z^{-2} + \ldots b(m+1)z^{-m}}{a(1) + a(2)z^{-1} + a(3)Z^{-3} + \ldots a(n+1)z^{-n}}$$

- 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” 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)” 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” for more information.

Designing a Filter in the Filter Builder GUI

Filter Builder Design Process

In this section...
“Introduction to Filter Builder” on page 25-2
“Design a Filter Using Filter Builder” on page 25-2
“Select a Response” on page 25-2
“Select a Specification” on page 25-4
“Select an Algorithm” on page 25-5
“Customize the Algorithm” on page 25-6
“Analyze the Design” on page 25-7
“Realize or Apply the Filter to Input Data” on page 25-7

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

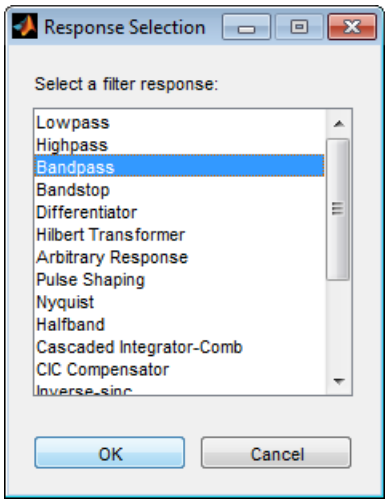
The basic workflow in using `filterBuilder` is to choose the constraints and specifications of the filter, and to use those as a starting point in the design. Postponing the choice of algorithm for the filter allows the best design method to be determined automatically, based upon the desired performance 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.

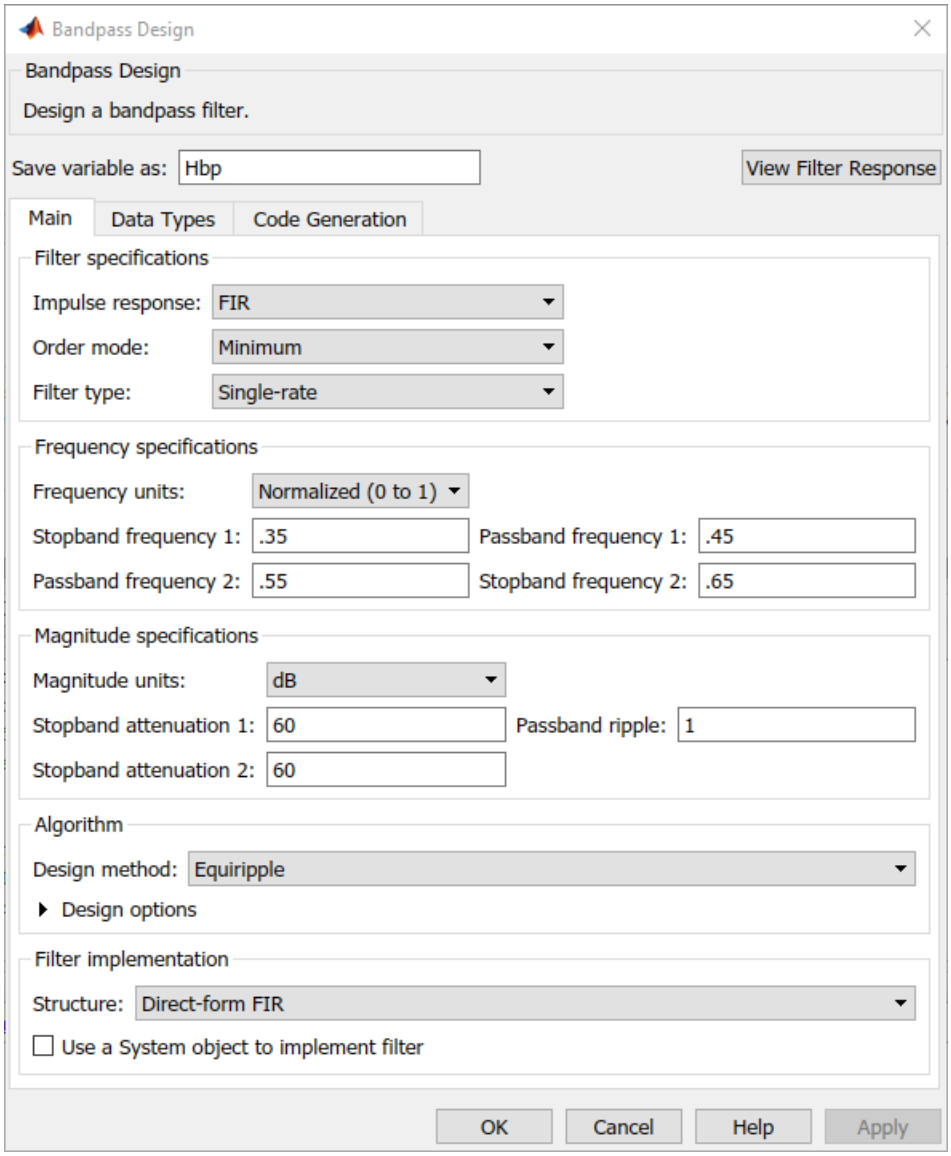


Note This step cannot be skipped because it is not automatically completed for you by the software. You must select a response to initiate the filter design process.

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.

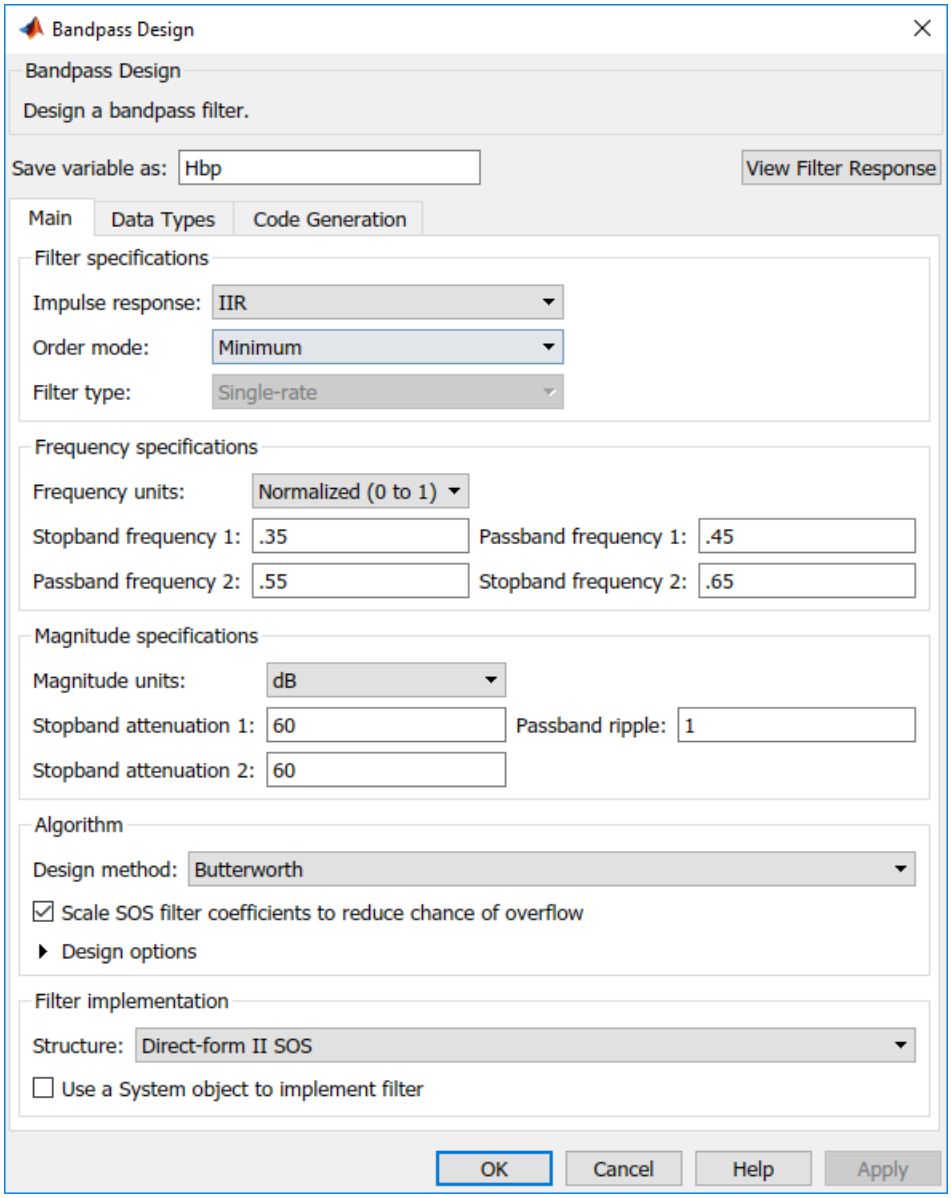
Select a Specification

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.

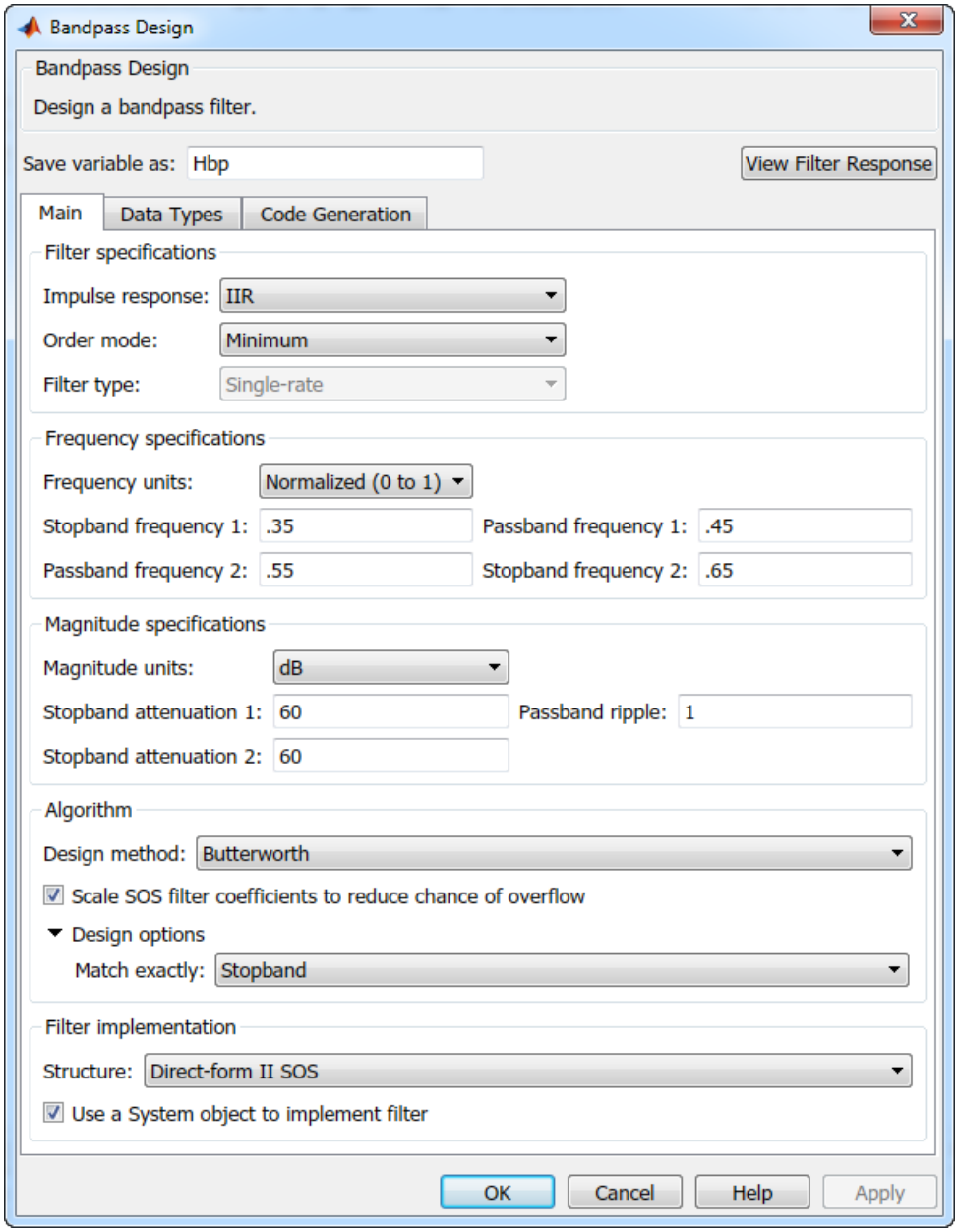
Select an Algorithm

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`.



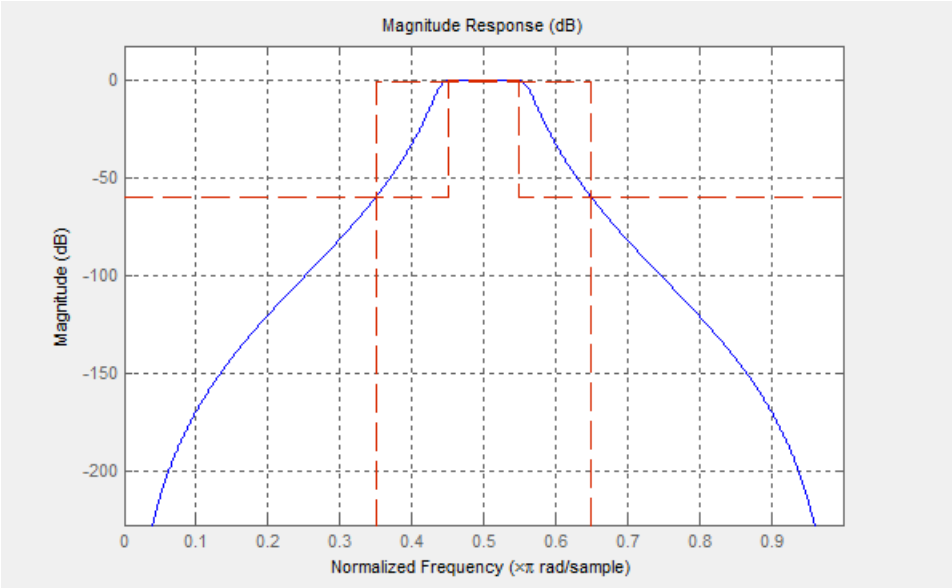
Customize the Algorithm

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.

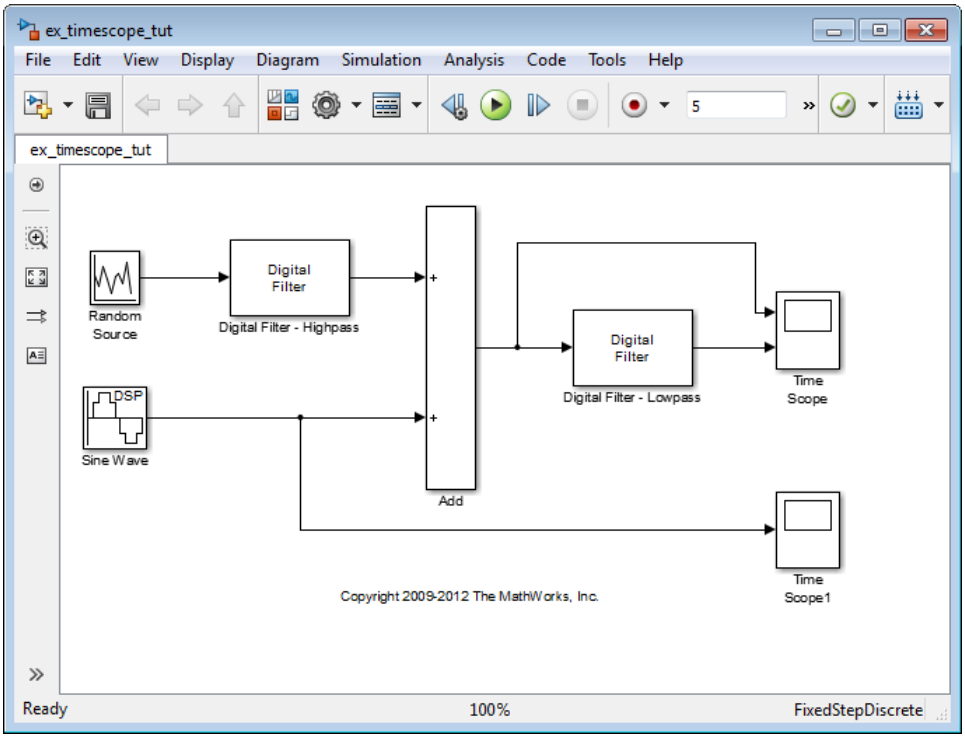
Display Time-Domain Data

In this section...
“Configure the Time Scope Properties” on page 26-3
“Use the Simulation Controls” on page 26-6
“Modify the Time Scope Display” on page 26-7
“Inspect Your Data (Scaling the Axes and Zooming)” on page 26-8
“Manage Multiple Time Scopes” on page 26-10
“Display Complex-Valued Input Signal” on page 26-12
“Display Input Signal of Changing Size” on page 26-12
“Display Simulink Enumeration Input Signal” on page 26-12

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 26-3
2. “Use the Simulation Controls” on page 26-6
3. “Modify the Time Scope Display” on page 26-7
4. “Inspect Your Data (Scaling the Axes and Zooming)” on page 26-8

5 “Manage Multiple Time Scopes” on page 26-10

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.

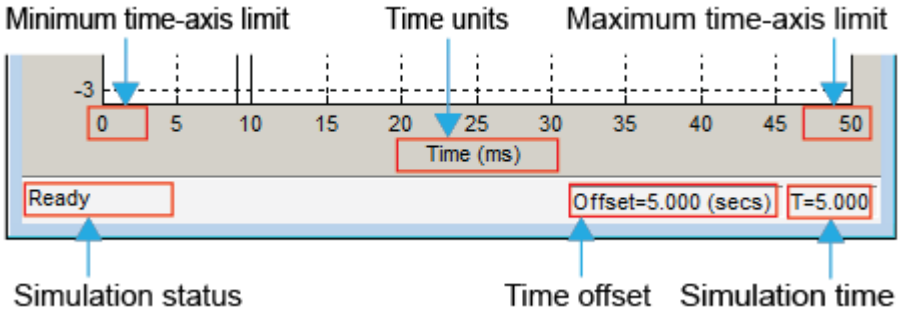
Parameter	Setting
Axes scaling	`Manual`
Scale axes limits at stop	Checked
Data range (%)	`80`
Align	`Center`
Autoscale X-axis limits	Unchecked

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.



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**.

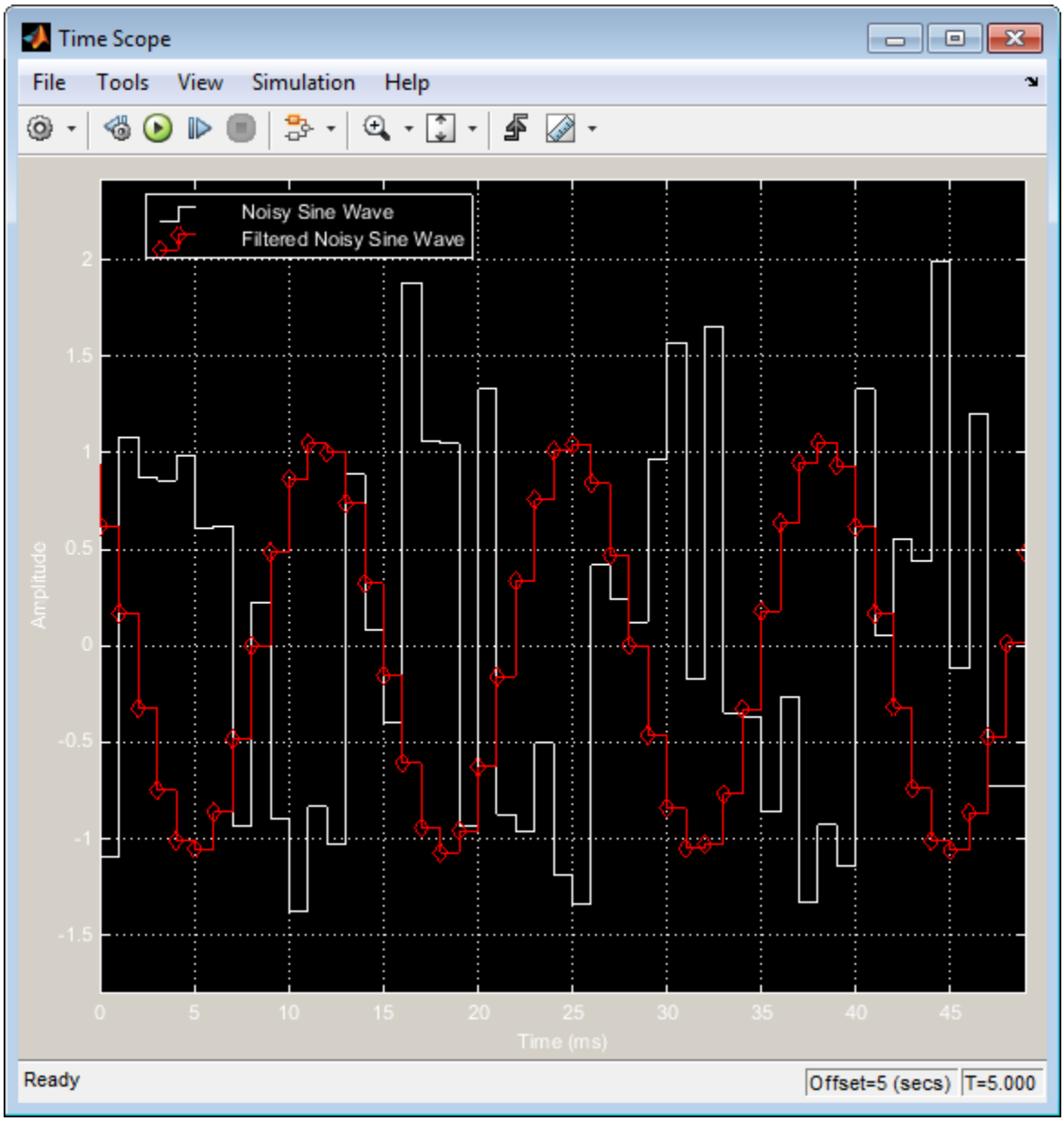
Adjust White Space Around the Signal

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.

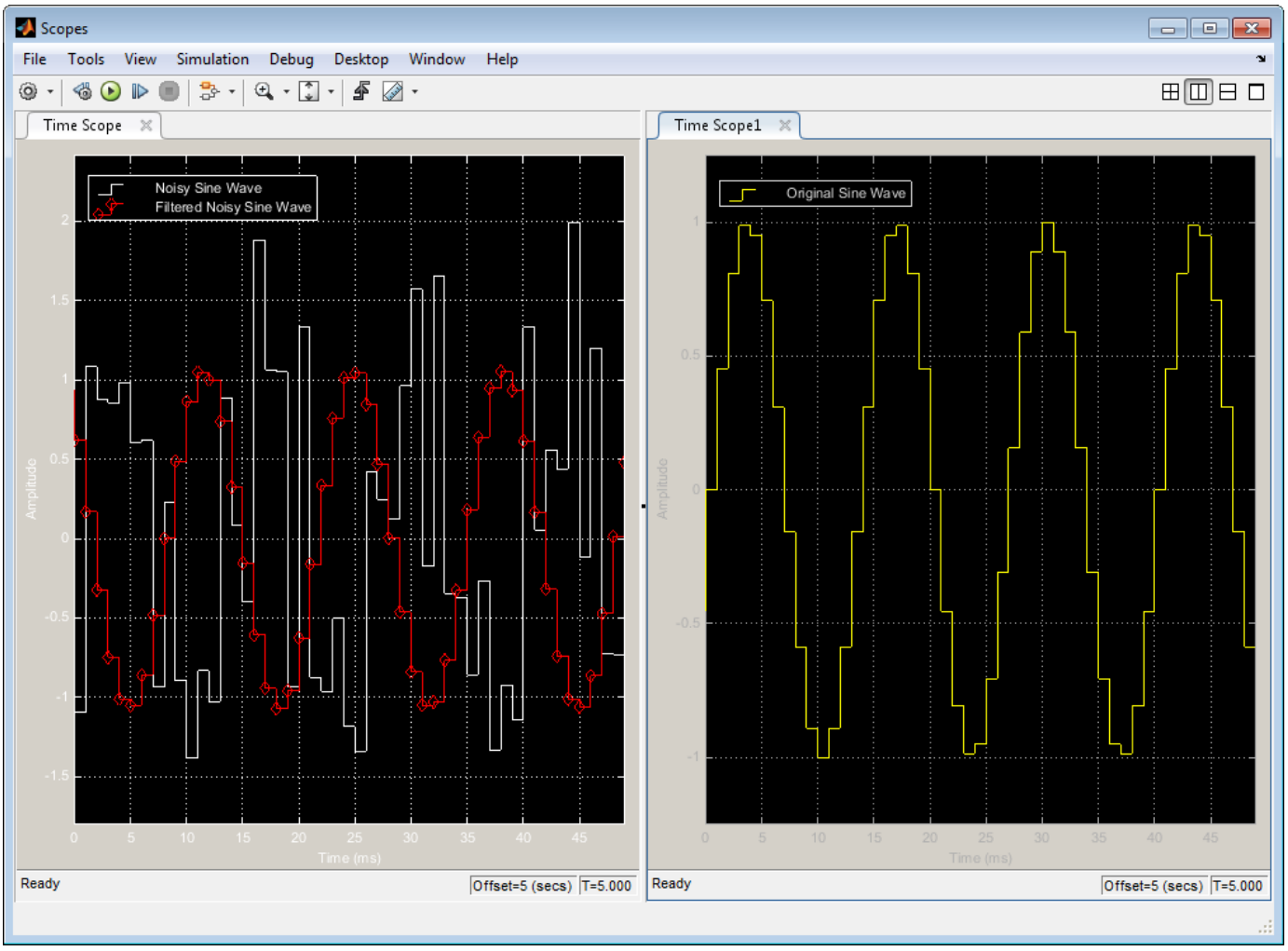
To try this feature, open the Time Scope window, and on the simulation toolbar, click the Highlight Simulink Block button. Doing so opens the `ex_timescope_tut` model. The Time Scope block flashes three times in the model window, allowing you to see where in your model the block of interest is located.

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

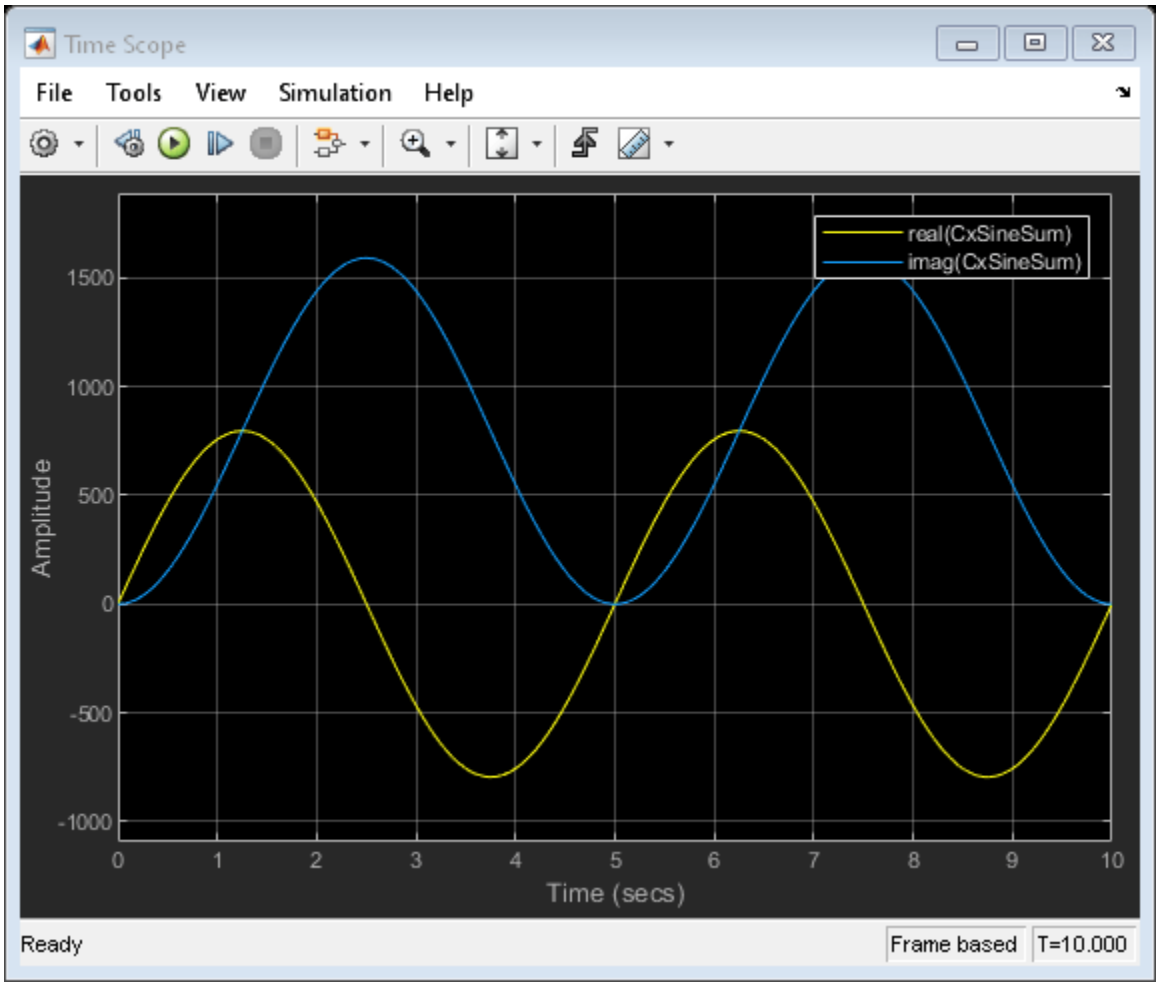
If you save your model with Time Scope windows open, those windows will reopen the next time you open the model. Reopening the Time Scope windows when you open your model can increase the amount of time it takes your model to load. If you are working with a large model, or a model containing multiple 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 Complex-Valued Input Signal

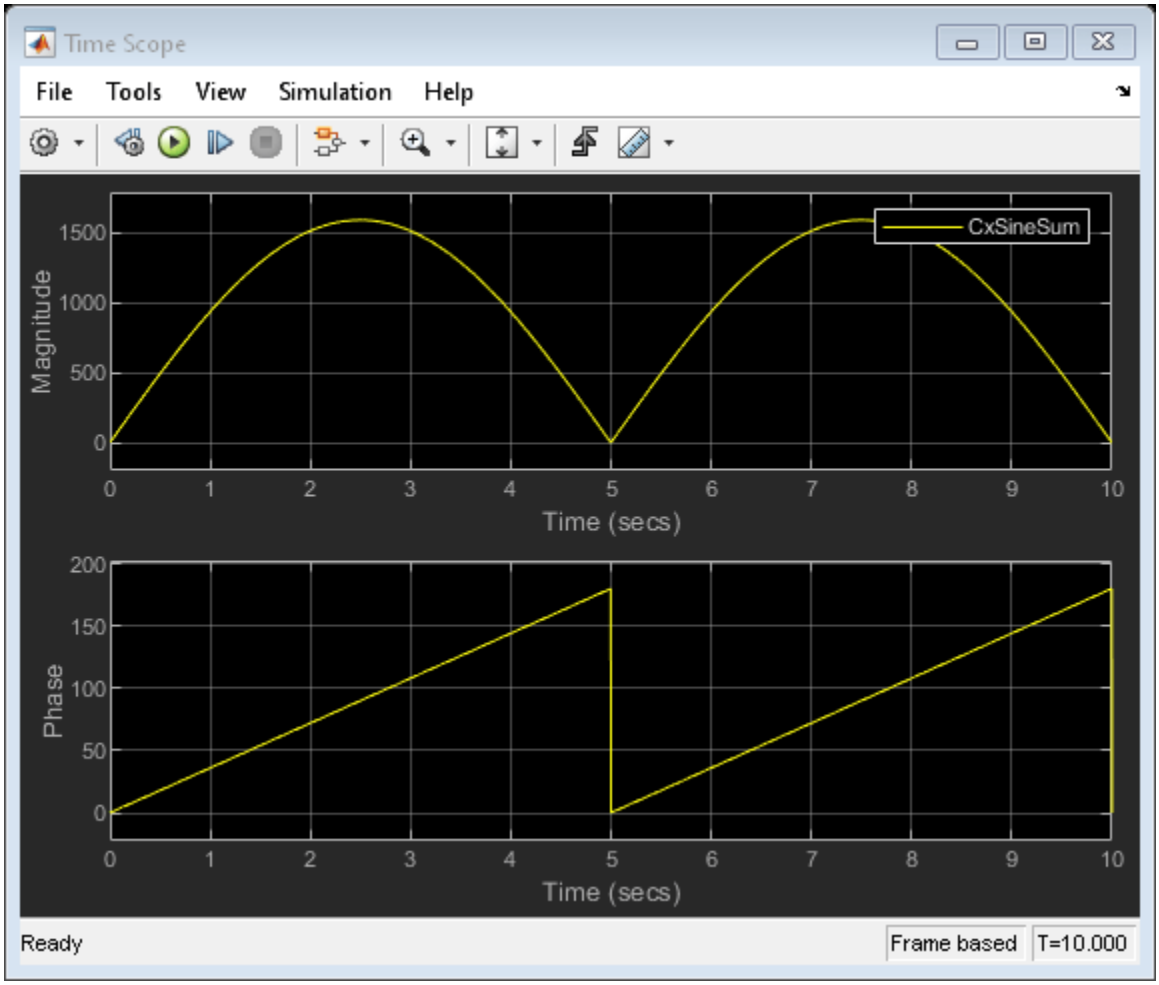
This example shows how to display complex-valued signals in the Time Scope block.

By default, when the input is a complex-valued signal, Time Scope plots the real and imaginary portions on the same axes. These real and imaginary portions appear as different-colored lines on the same axes within the same active display.



To separate the real and imaginary portions of the input, change the scope to plot magintude and phase.

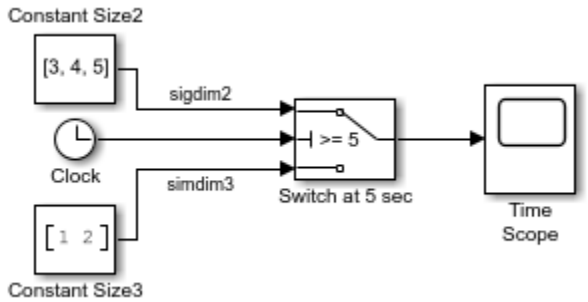
1. From the Scope menu, select **View > Configuration Properties**.
2. In the Configuration Properties window, on the Display tab, select **Plot signal(s) as magnitude and phase**.
3. Click **OK**. The active display shows the magnitude of the input signal on the top axes. The signal phase, in degrees, appears on the bottom axes.



Display Input Signal of Changing Size

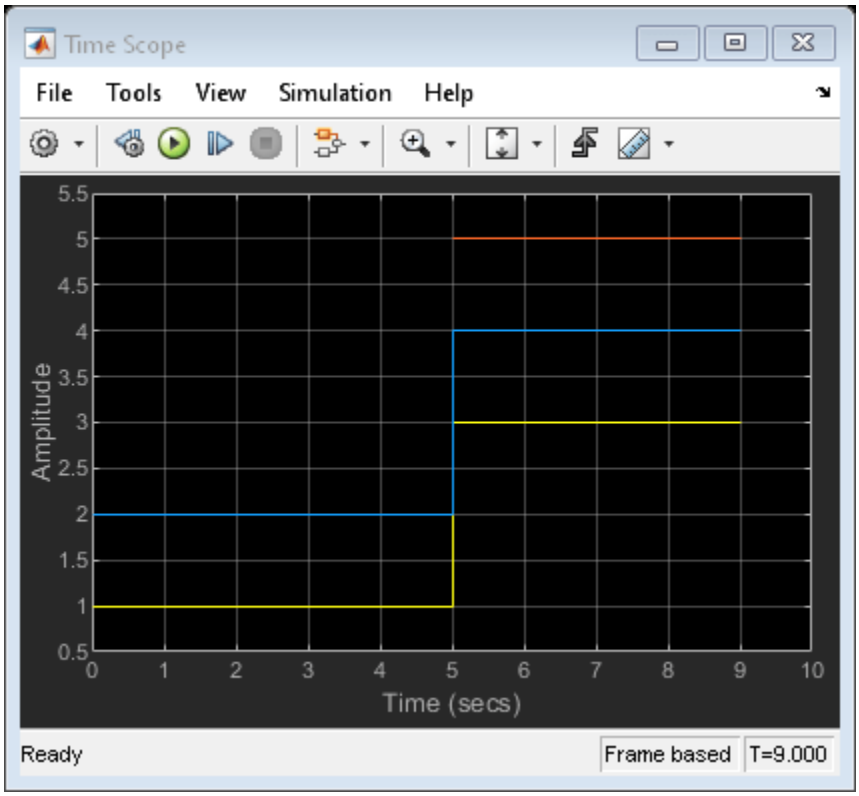
In this example, the size of the input signal to the Time Scope block changes as the simulation progresses.

```
open_system("ex_timescope_varsize")
```



When the simulation time is less than 5 seconds, Time Scope plots the signal connected to the third input port of the Switch block, which has signal dimensions 1 by 2. After 5 seconds, Time Scope plots the signal connected to the first input port of the Switch block, which has signal dimensions 1 by 3.

```
sim("ex_timescope_varsize")
open_system("ex_timescope_varsize/Time Scope")
```

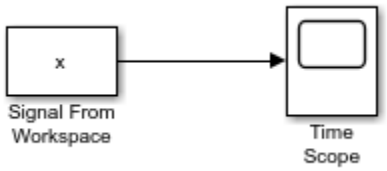


As you can see in the figure, the third line on the display, colored red, appears only after 5 seconds.

Display Simulink Enumeration Input Signal

This example shows how to use the Time Scope to display an enumerated input signal.

```
open_system("ex_timescope_slenum")
```



In this example, Simulink® imports the variable *x* , from the MATLAB® workspace. This variable is created when the model loads because the commands that construct it reside in the model Preload function. To view these commands,

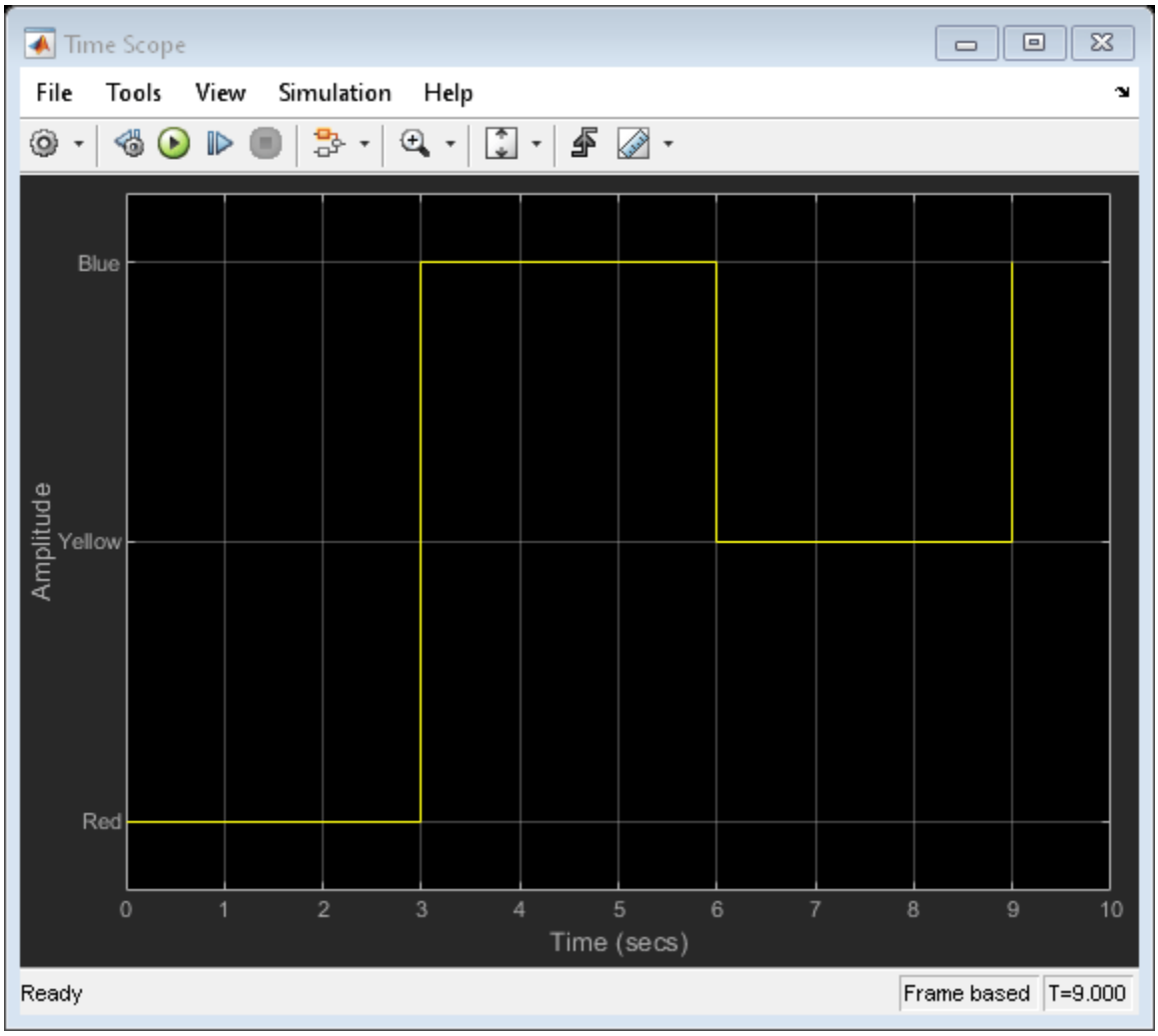
1. On the Simulink toolbar, on the Modeling tab, in the Setup section, in the drop-down, select `Model Properties`.
2. In the Model Properties dialog box, select the `Callbacks` tab. The following lines of MATLAB® code appear.

```
if ~exist('BasicColors','class')
    Simulink.defineIntEnumType('BasicColors', ...
        {'Red', 'Yellow', 'Blue'}, ...
        [0;1;2], ...
        'Description', 'Basic colors', ...
        'DefaultValue', 'Blue', ...
        'AddClassNameToEnumNames', true);
end
x = [BasicColors(0), BasicColors(2), BasicColors(1)]
```

```
x =

  1x3 BasicColors enumeration array

    Red       Blue      Yellow
```

The Signal from Workspace block has a sample time of 3 seconds. Thus, the input signal changes to the next value in vector *x* every 3 seconds.

```
sim("ex_timescope_slenum")
open_system("ex_timescope_slenum/Time Scope")
```

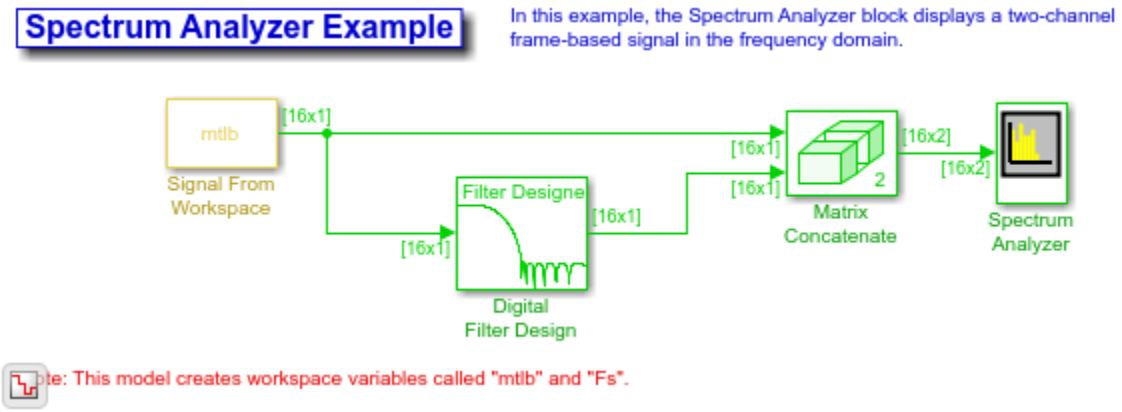

Time Scope
File
Tools
View
Simulation
Help
Blue
Yellow
Red
Amplitude
0
1
2
3
4
5
6
7
8
9
10
Time (secs)
Ready
Frame based
T=9.000

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.

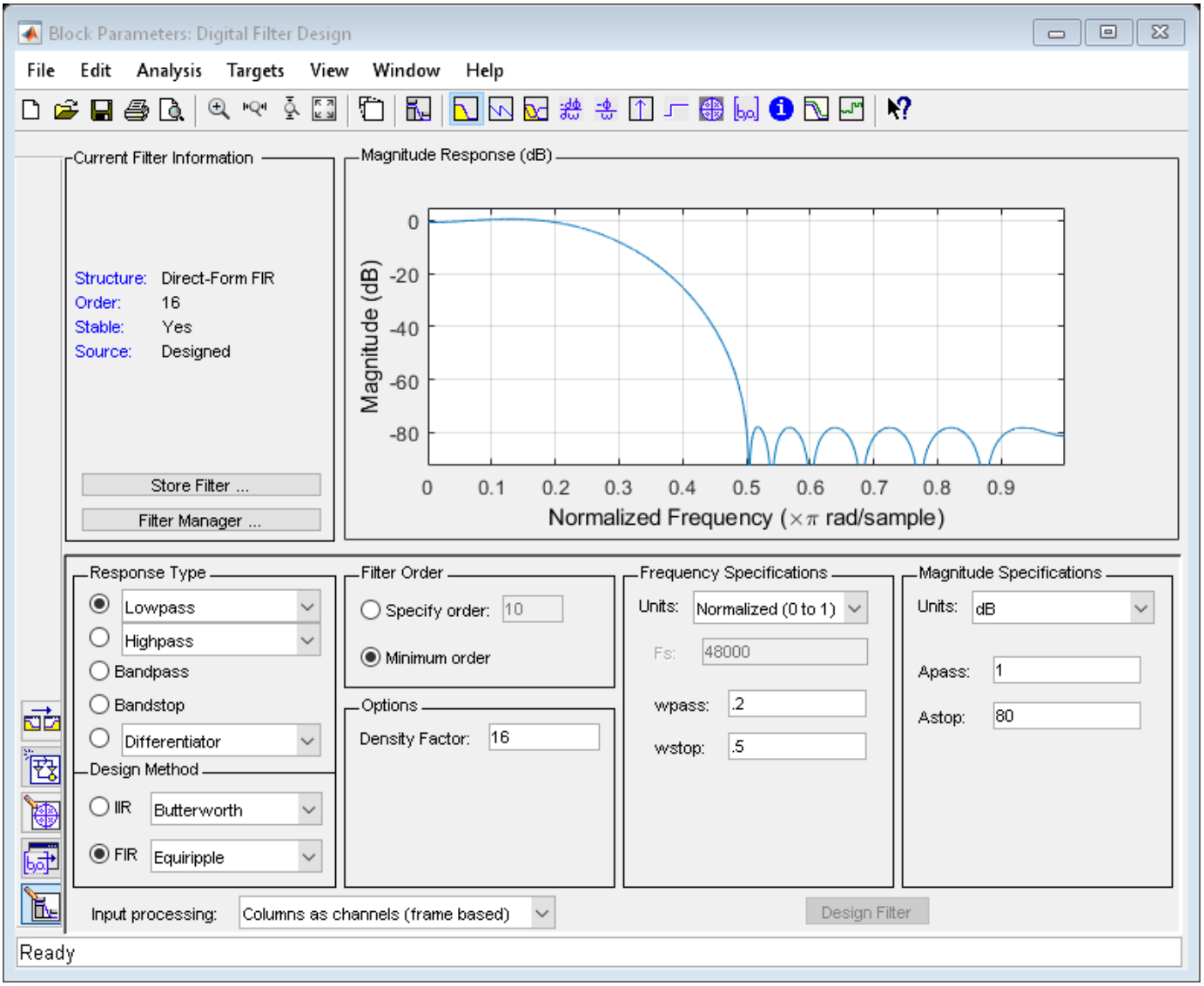
```
model = 'ex_spectrumanalyzer_tut';
open_system(model)
```



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:

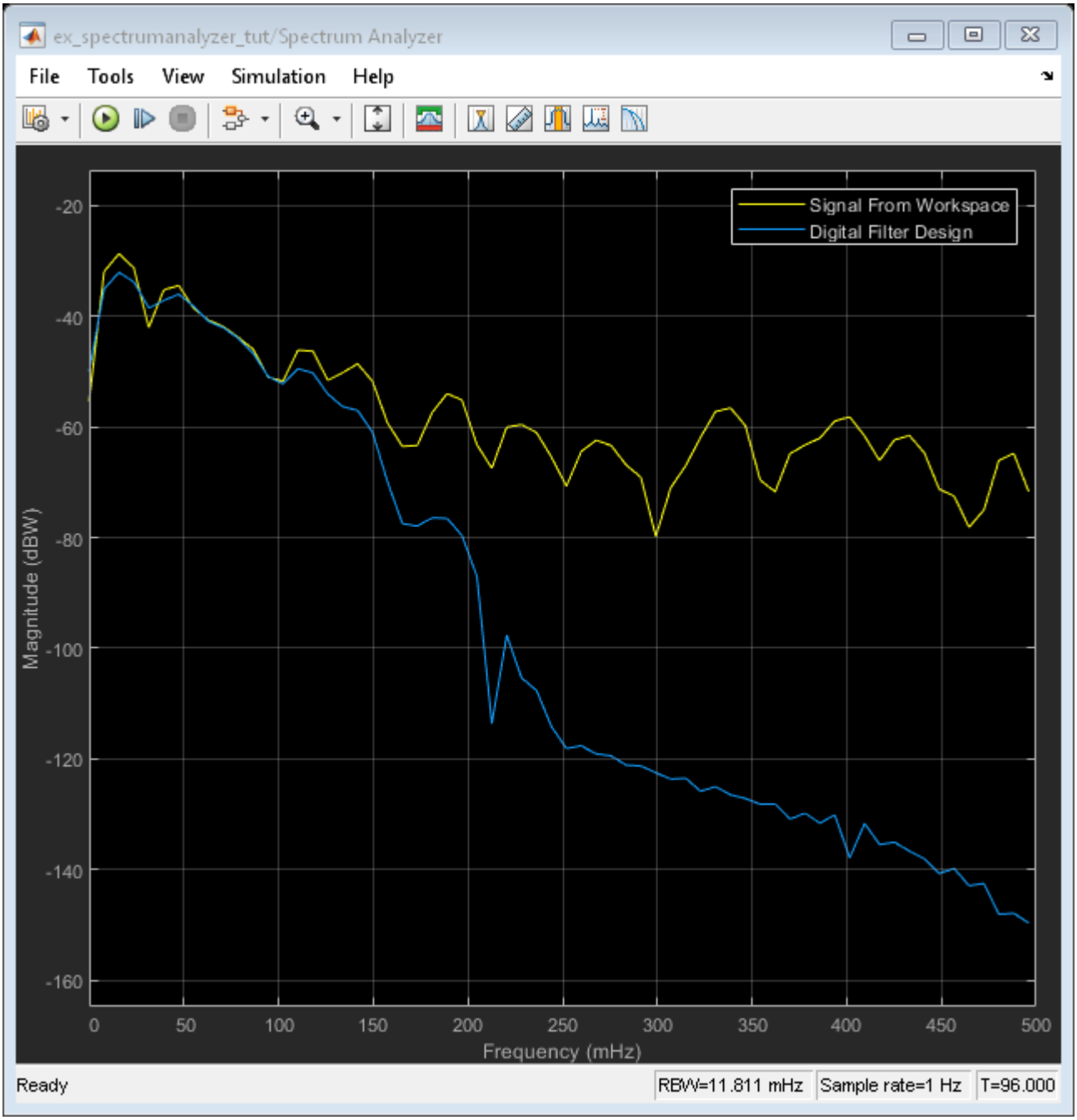
$$N_s = \frac{NENBW \times F_s}{RBW} = \frac{1.512 \times 1}{11.8125 \times 10^{-3}} = 128$$

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.

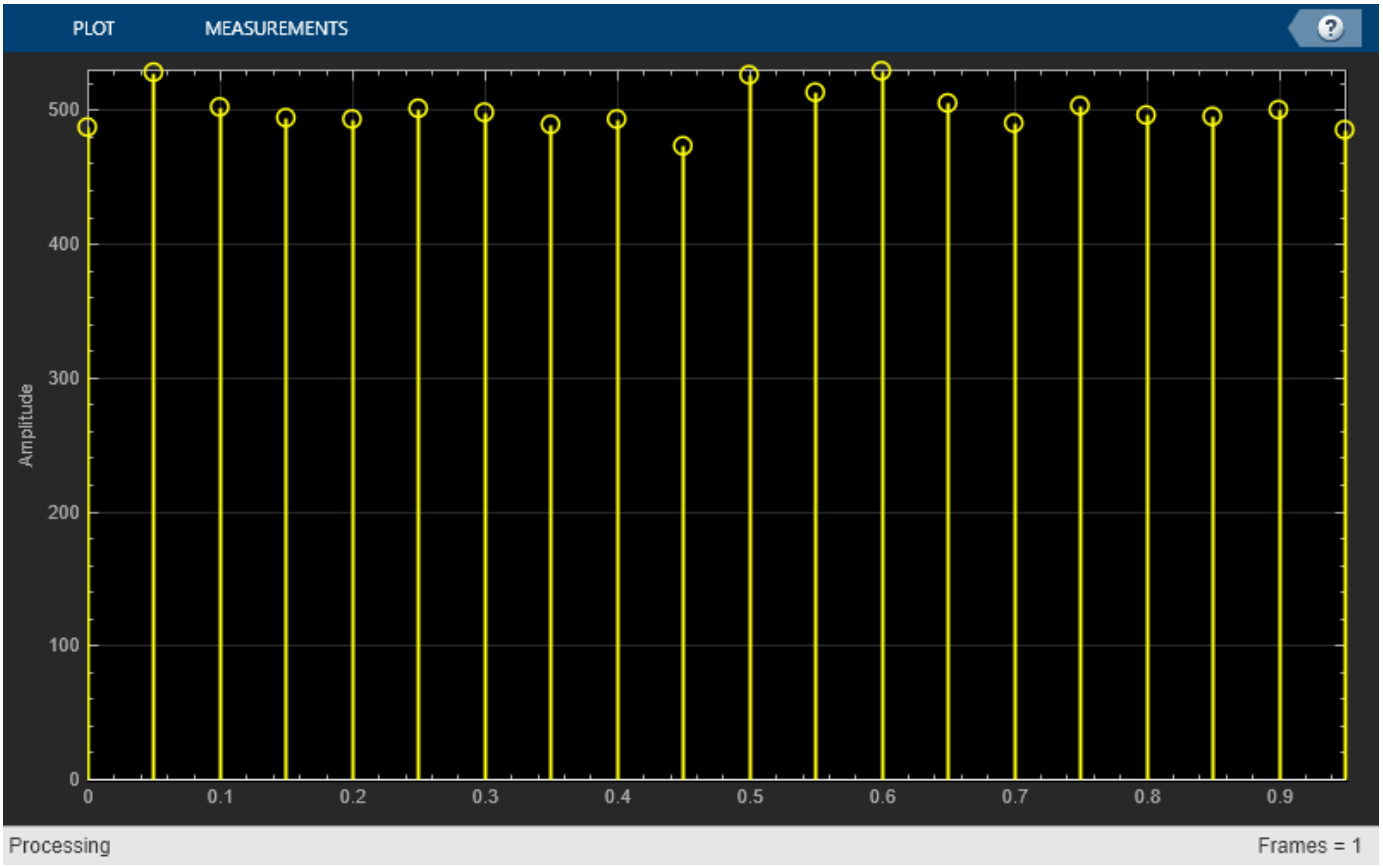
```
numsamples = 1e4;
numbins = 20;
r = rand(numsamples,1);
hst = histcounts(r,numbins);
```

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);
```

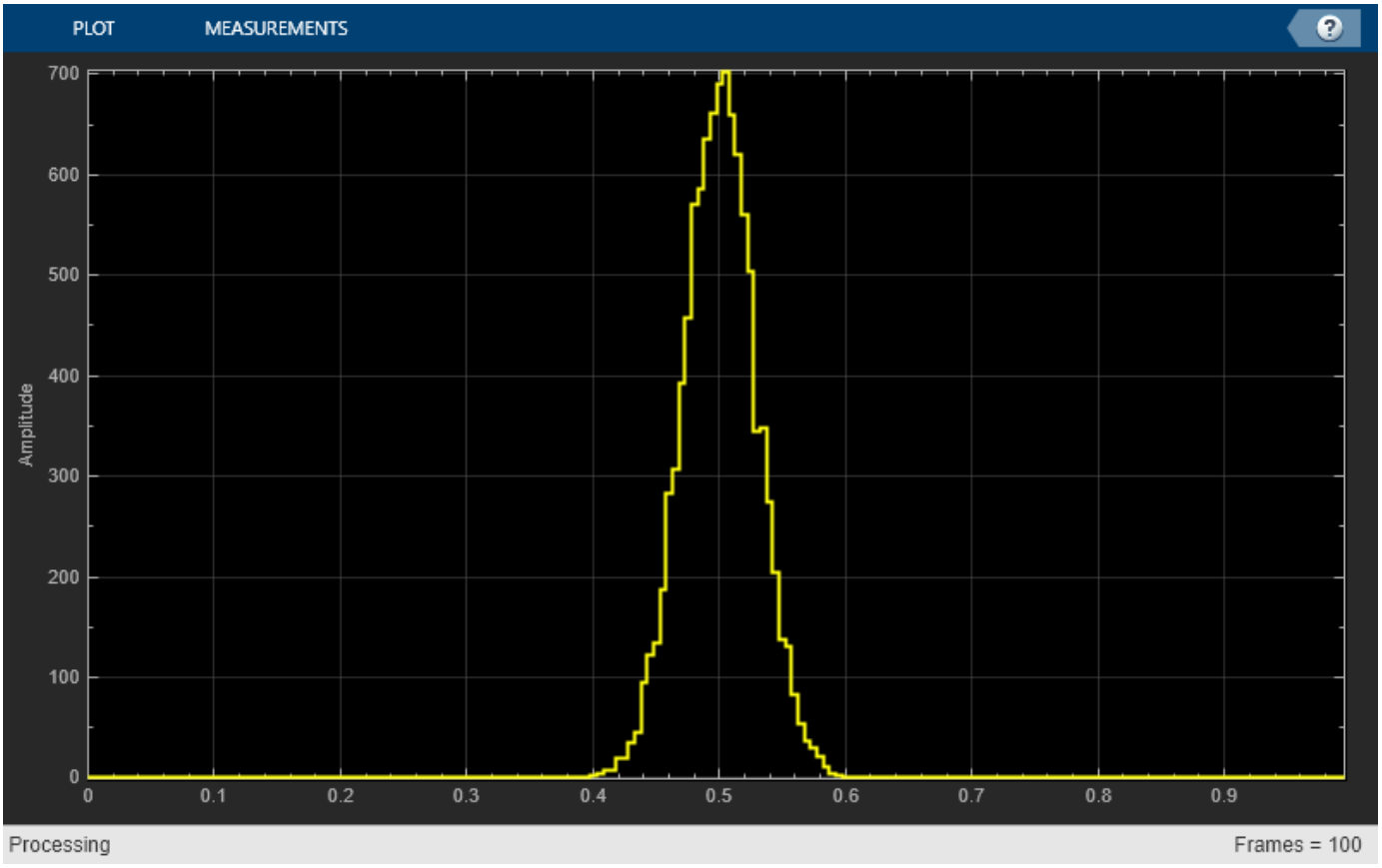
Change the configuration of the Array Plot properties for the display of a distribution function.

```
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

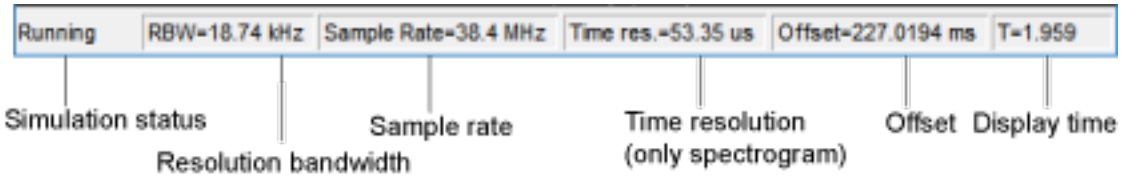
`dsp.ArrayPlot` | `histcounts` | `rand`

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.

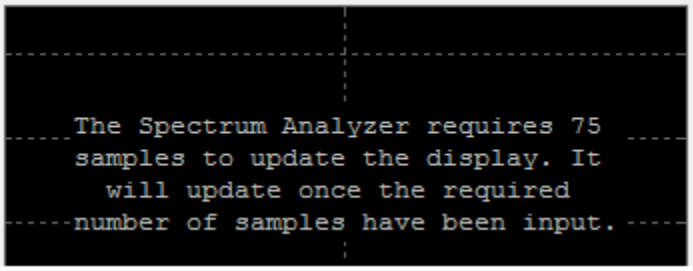
  Details

  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.

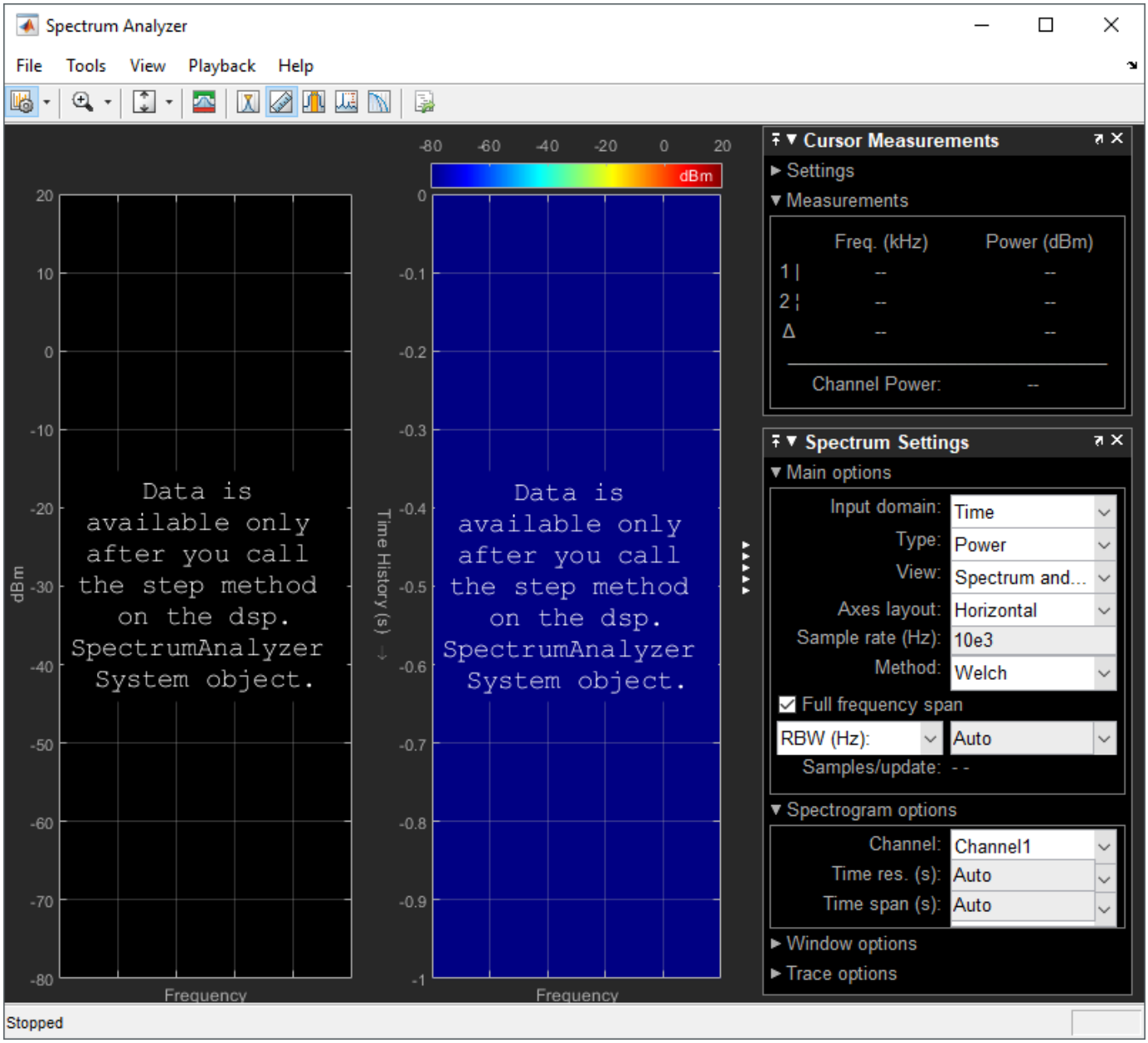
Note The script only generates commands for settings that are available from the command line, applicable to the current visualization, and changed from the default value.

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:

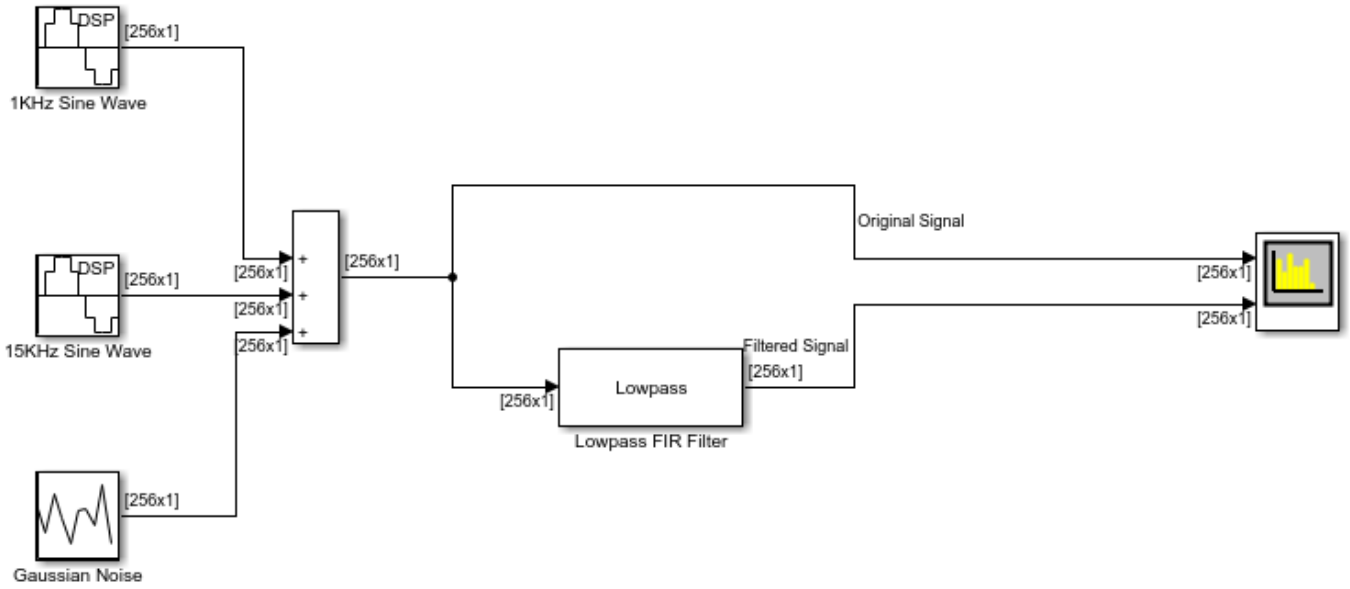
- In the Spectrum Analyzer window, select the spectral mask button . In the **Spectral Mask** pane, the **Statistics** section shows statistics about how often the masks fail, which channels have caused a failure, and which masks are currently failing.
- To get the current status of the spectral masks, 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”.

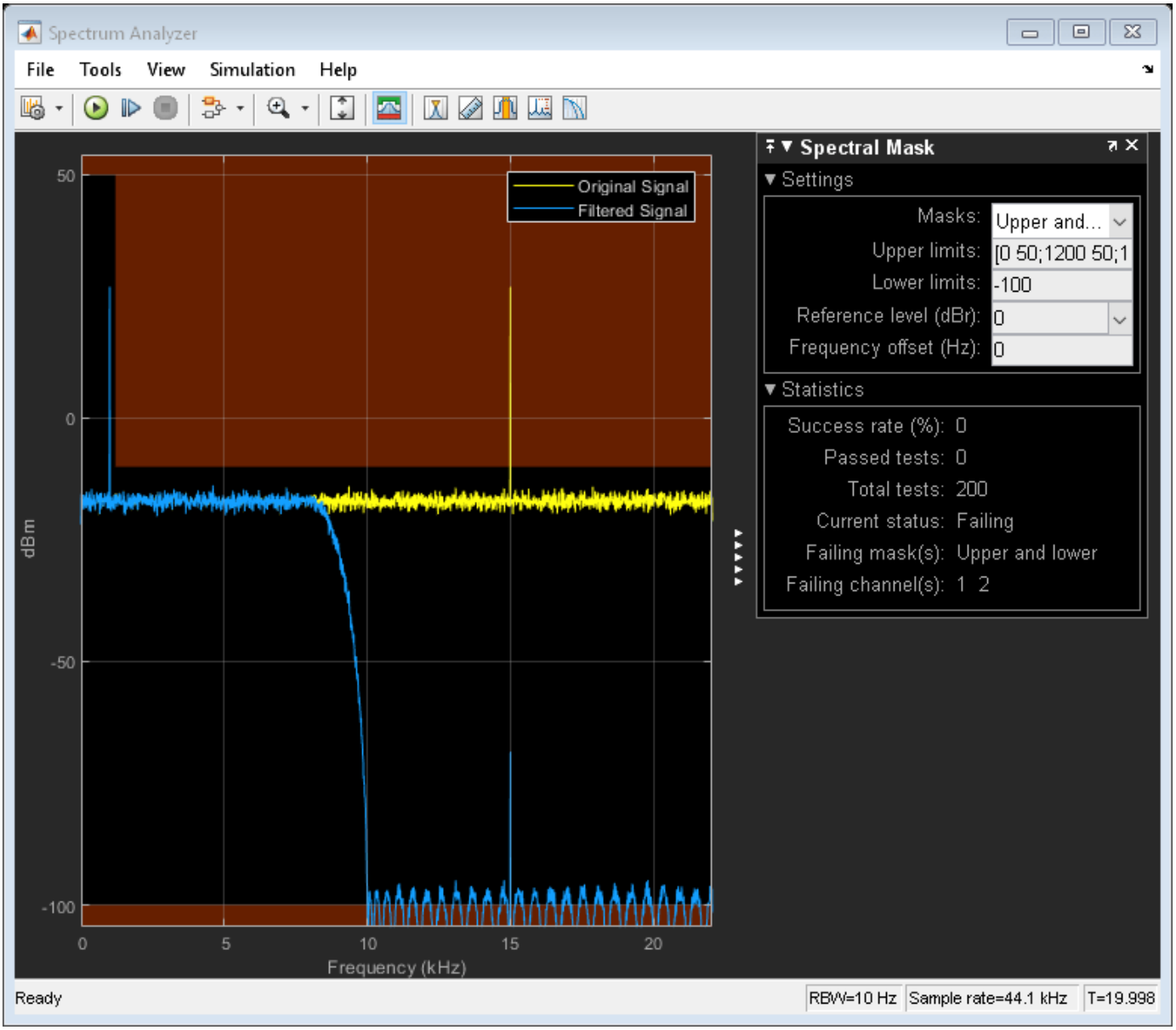
Spectral Mask in Spectrum Analyzer Block

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');
```





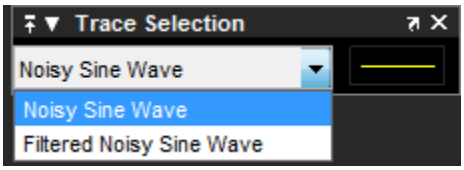
Measurements Panels

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.

- 

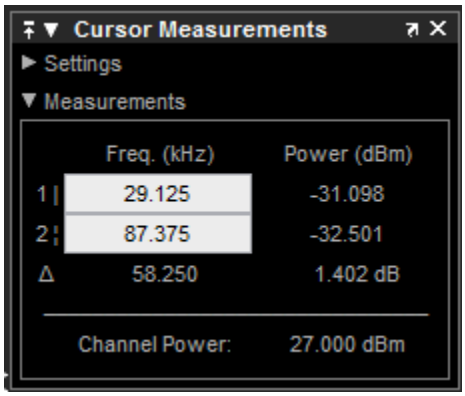
Cursor Measurements Panel

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.

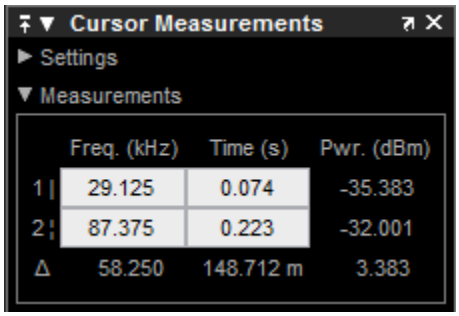
Note If a data point in your signal has more than one value, the cursor measurement at that point is undefined and no cursor value is displayed.

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:

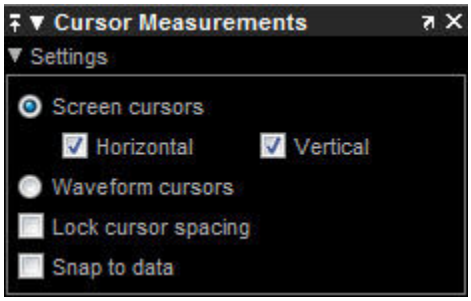


The **Cursor Measurements** panel for the spectrogram view. You must pause the spectrogram display before you can use cursors.



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.

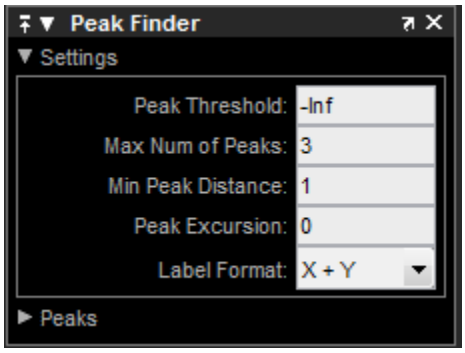
The letter after the value associated with a measurement indicates the abbreviation for the appropriate International System of Units (SI) prefix.

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.

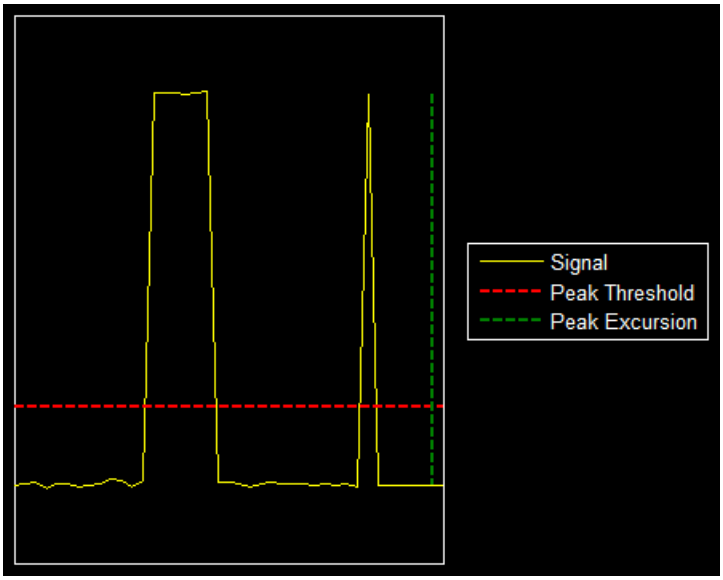
- From the menu, select **Tools** > **Measurements** > **Peak Finder**.
- On the toolbar, click the Peak Finder button.

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

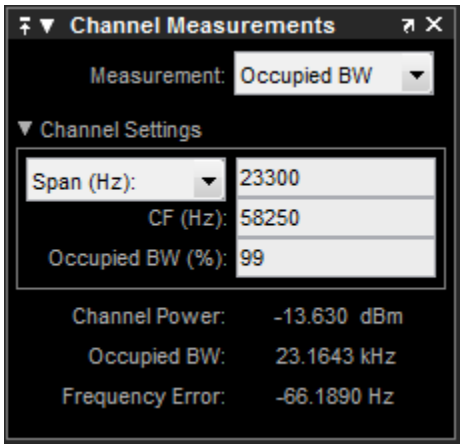
The **Channel Measurements** panel displays occupied bandwidth or adjacent channel power ratio (ACPR) measurements.

- 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.

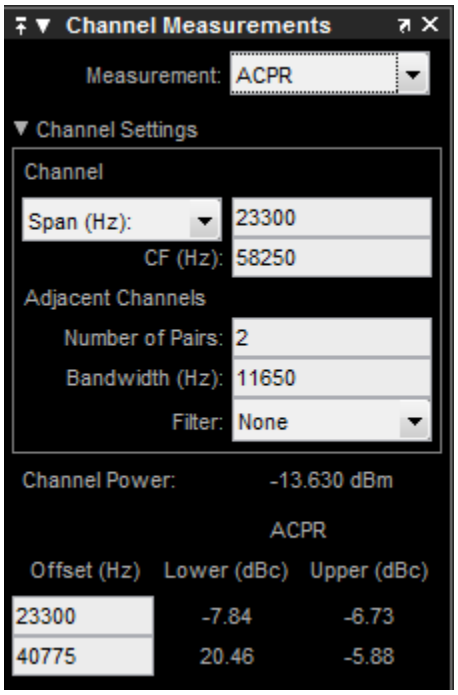


- **Channel Settings** — Modify the parameters for calculating the channel measurements.

  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.

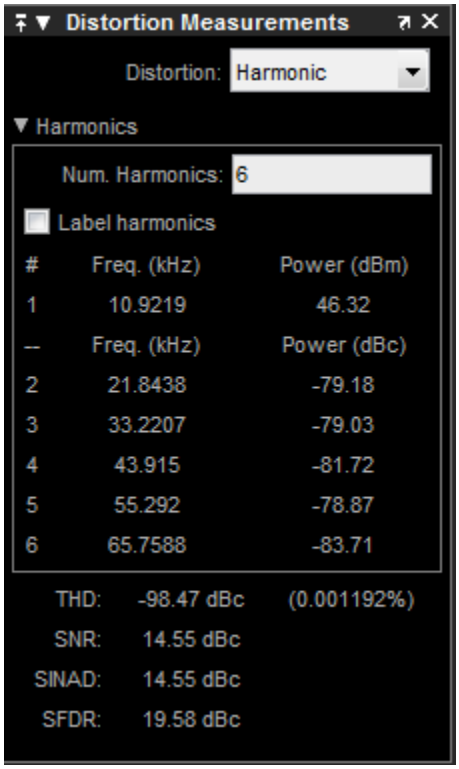
The **Distortion Measurements** panel has an expandable **Harmonics** pane, which shows measurement results for the specified number of harmonics.

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.

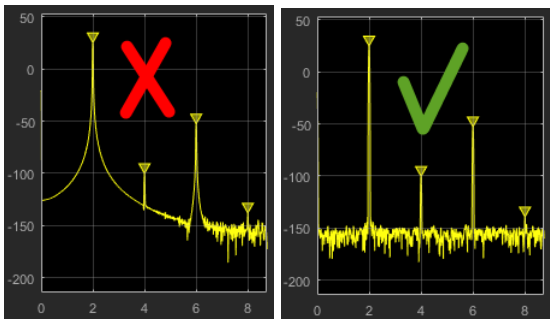


The harmonic distortion measurement automatically locates the largest sinusoidal component (fundamental signal frequency). It then computes the harmonic frequencies and power in each harmonic in your signal. Any DC component is ignored. Any harmonics that are outside the spectrum

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).

  $$SNR = 10 \cdot \log_{10}(S/N)$$

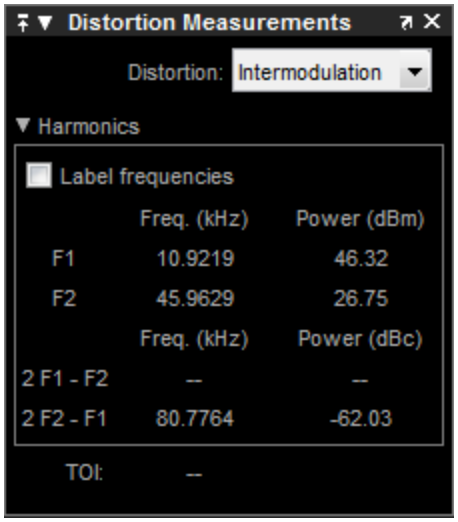
  If you see **—** as the reported SNR, the total non-harmonic content of your signal is less than 30% of the total signal.
- **SINAD** — Signal-to-noise-and-distortion. This value represents the ratio of the power in the fundamental frequency, S to all other content (including noise, N, and harmonic distortion, 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

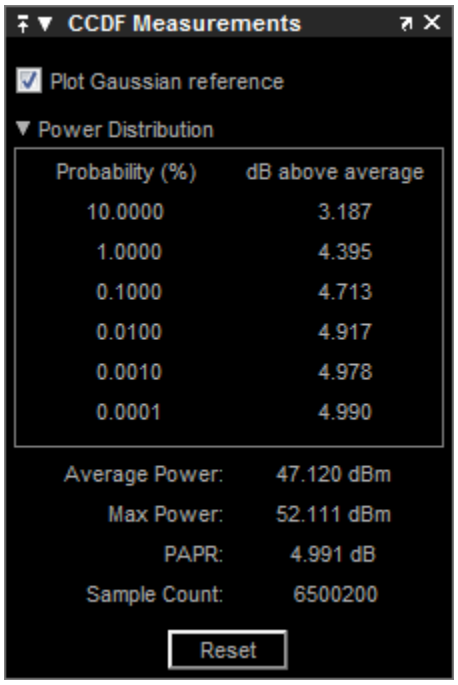
The **CCDF Measurements** panel displays complimentary cumulative distribution function measurements. CCDF measurements in this scope show the probability of the instantaneous power of a signal being a specified level above the average power of the signal. These measurements are useful indicators of the dynamic range of a signal.

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 complex white Gaussian noise reference signal on the plot.
- **Probability (%)** — Percentage of the signal that contains the power level above the value listed in the **dB above average** column
- **dB above average** — Expected minimum power level at the associated **Probability (%)**.
- **Average Power** — Average power level of the signal since the start of simulation or from the last reset.

  Max Power — Maximum power level of the signal since the start of simulation or from the last reset.
- **PAPR** — 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** — 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 grid lines, 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**.

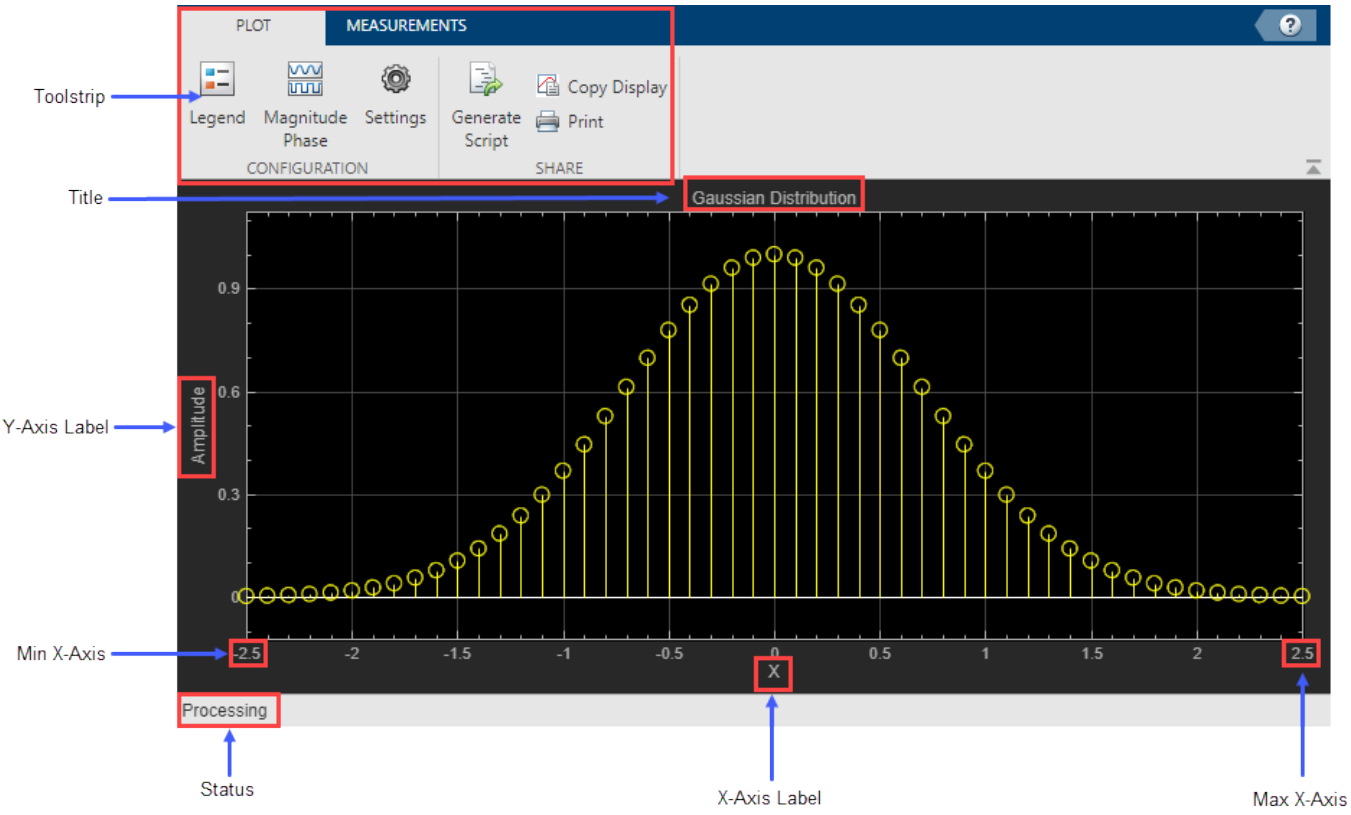
If you are viewing only the spectrum or the spectrogram, you see only the relevant options. If you are using the CCDF measurements, you will also see *x*-axis scaling options. For more details about these options, see “Axes Scaling”.

Configure Array Plot

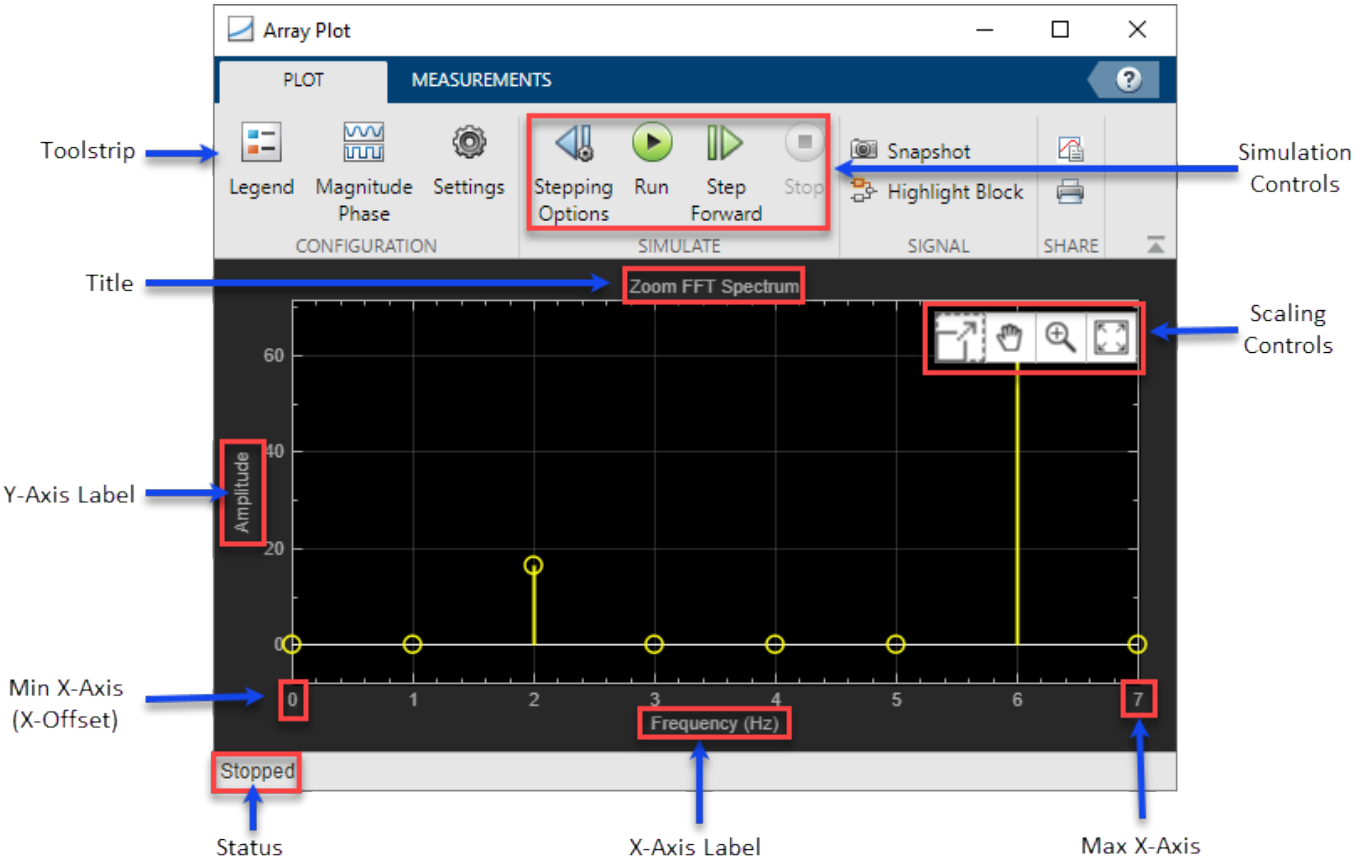
When you use the `dsp.ArrayPlot` object in MATLAB or the Array Plot block in Simulink you can configure many settings and tools from the interface. These sections show you how to use the Array Plot interface and the tools available.

Signal Display

This figure highlights the important aspects of the Array Plot window, in MATLAB:



And in Simulink:



- **Min X-Axis** — Array Plot sets the minimum *x*-axis limit using the value of the **X-Offset** 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 adding the value of **X-Offset** parameter with the span of *x*-axis values multiplied by the **Sample Increment** property as determined by this equation:

  $$x_{\max} = SampleIncrement * (\text{length}(x) - 1) + XOffset$$

  To modify the **Sample Increment** from the Array Plot window, click **Settings** and set the **Sample Increment**. If you set **Sample Increment** 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`
    - Object — Occurs after you run the object and before you run the `release` method.
    - Block — Occurs during the simulation.
  - `Stopped`
    - Object — Occurs after you call `release`.
    - Block — Occurs before and after the simulation.

  - `Ready`
    - Object — Occurs after you construct the scope object and before you first call the object.
    - Block — Occurs after you open the scope and before your first run a simulation.
  - `Paused`
    - Block — Occurs when you pause the simulation.
- **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.
- (Block only) **Simulation Controls** — Control your Simulink simulation from the Array Plot window.

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, the legend for a two-channel signal will display the default names `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 displayed is:

By default, the scope has a black axes background and chooses line colors for each channel in a manner similar to the 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 legend. 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 this 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

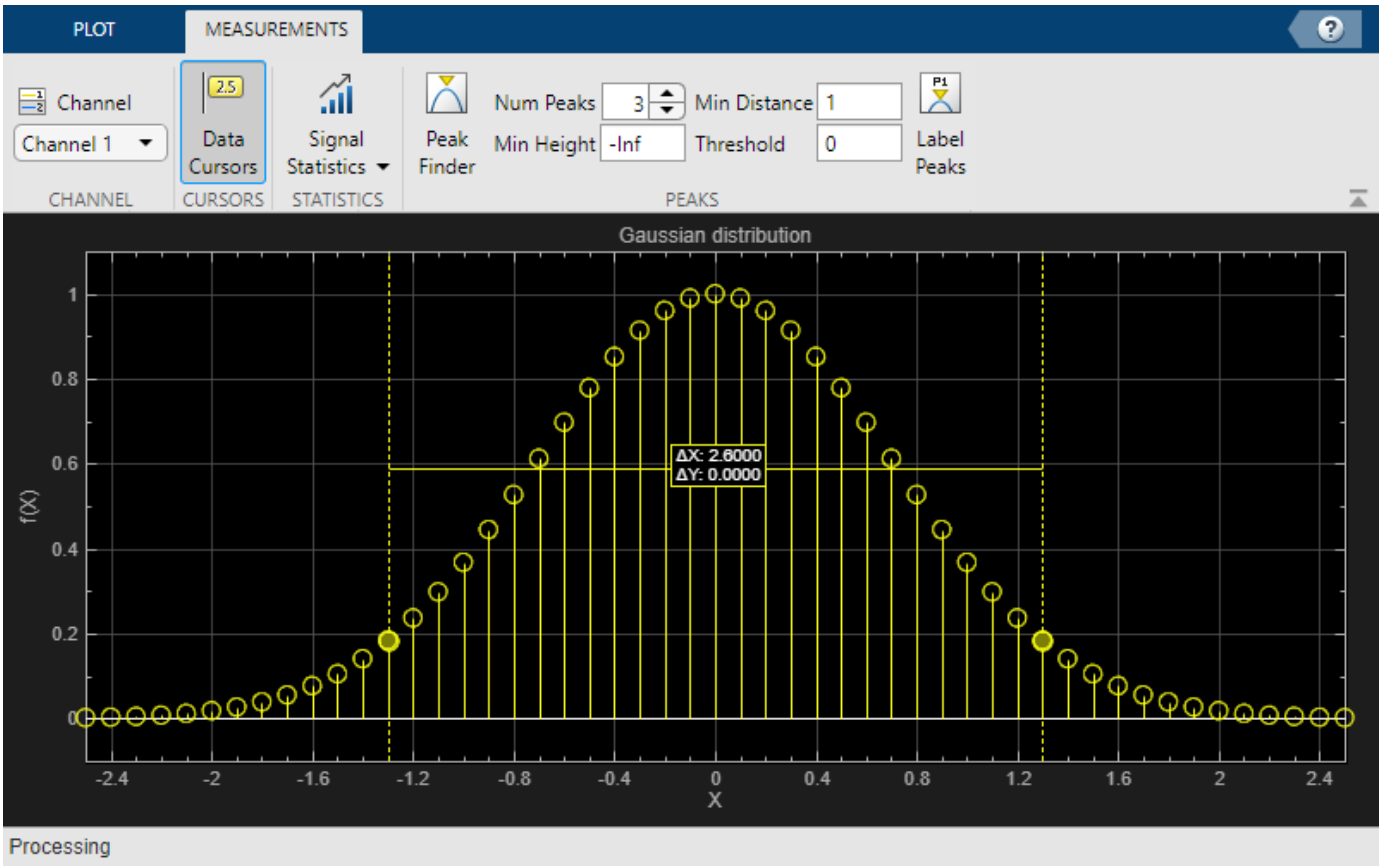
On the **Plot** tab, the Configuration section allows you to modify the plot.

- 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 or the Show legend property on the block.
- 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 or the Plot as Magnitude and Phase property on the block.
- The **Settings** button opens the settings window which allows you to customize the *x*-axis, *y*-limits, plot labels, and signal colors.

Use Array Plot Measurements

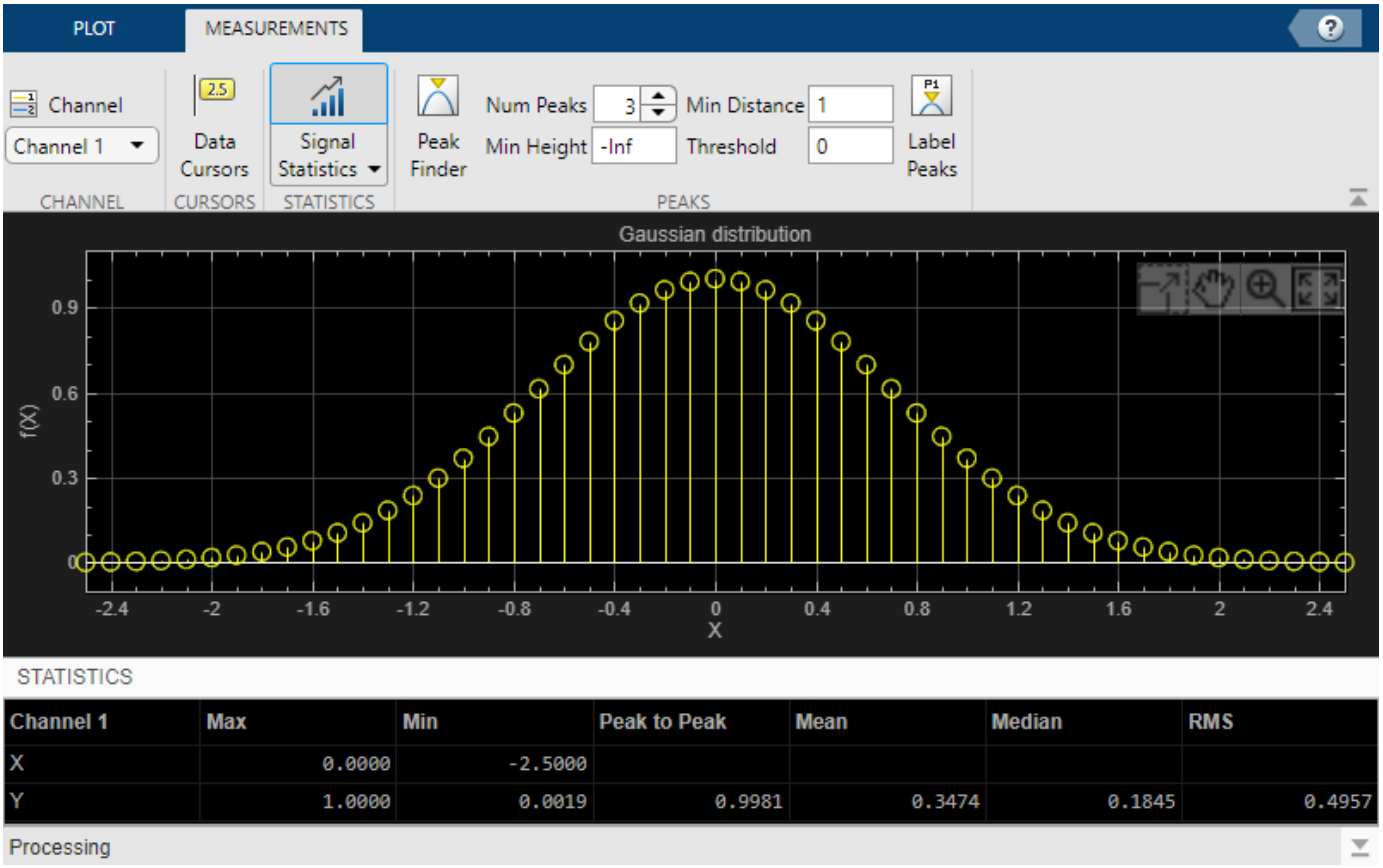
All measurements are made for a specified channel. By default, measurements are applied to the first channel. To change which channel is being measured, use the **Select Channel** drop-down in the **Measurements** tab.

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

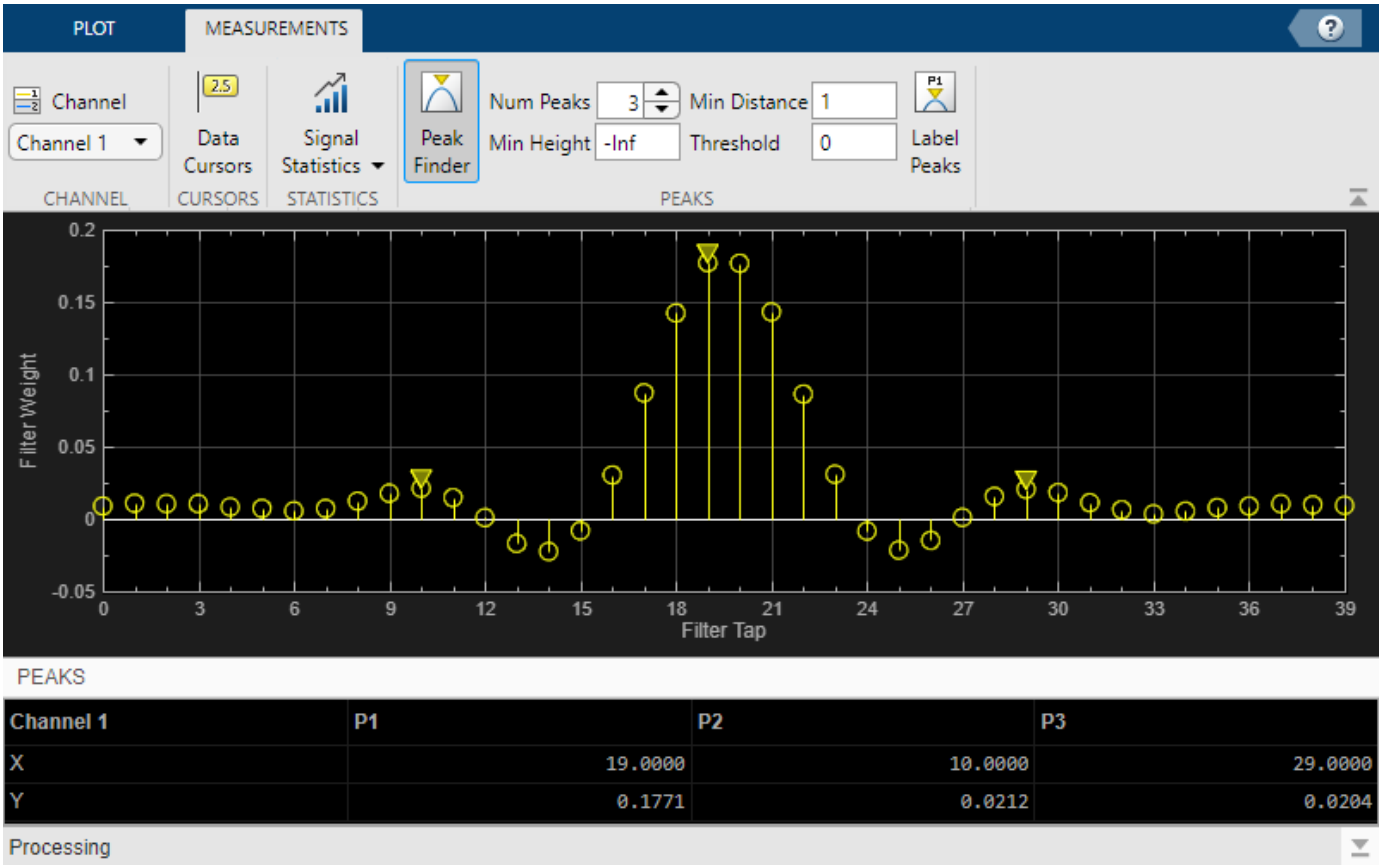


Use the **Signal Statistics** button to display various statistics about the selected signal at the bottom of the array plot window. You can hide or show the **Statistics** panel using the arrow button in the bottom right of the panel.

- **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 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 `findpeaks`.

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.

- (Object only) **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.
- **Snapshot** — During a simulation, use the **Snapshot** button to pause the visualization at an interesting point so you can take a screenshot of the Array Plot window.

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 on the plot.
- — Pan the plot.
- — Autoscale the axes to fit the shown data.

Set Additional Properties

You can only change some Array Plot properties outside the Array Plot window, such as the signal names in the legend or the number of inputs. For the `dsp.ArrayPlot` object, you can set those properties from the command-line. For the Array Plot block, set those properties using the Property Inspector (“Set Block Parameter Values” (Simulink)) or from the command-line using `get_param` and `set_param`.

Find the Array Plot Block in Your Model

To highlight the Array Plot block within your model, on the **Plot** tab, select the **Highlight Block** button. On the Simulink canvas, the Array Plot block is outlined in a highlight color so you can more easily identify it in your model.

See Also

`dsp.ArrayPlot` | Array Plot

More About

- “Visualize Central Limit Theorem in Array Plot” on page 26-21
- “Reconstruction Through Two-Channel Filter Banks” on page 4-241
- “Analyze a Subband of Input Frequencies Using Zoom FFT” on page 17-2
- “Group Delay Estimation in Simulink” on page 17-4

Configure Time Scope Block

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 26-3.

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.

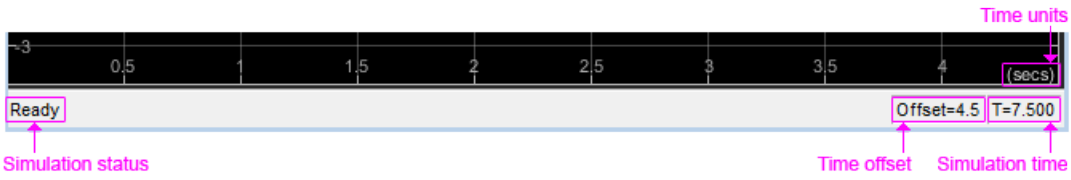
  The *Simulation status* is part of the status bar in the scope window. You can choose to hide or display the entire status bar. From the scope menu, select **View > Status Bar**.
- *Time offset* — The `Offset` value helps you determine the simulation times for which the scope is displaying data. The 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.

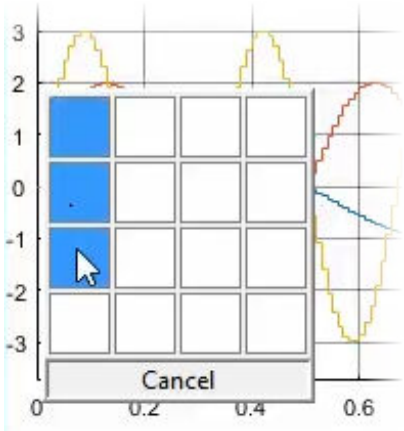
If there are more than seven channels, then the scope repeats this order to assign line colors to the remaining channels. To choose line colors for each channel, change the axes background color to any color except black. To change the axes background color to white, select **View > Style**, click the 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.



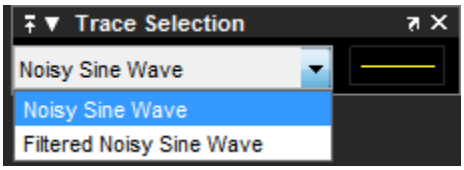
When you use the Layout option to tile the window into multiple displays, the display highlighted in blue is referred to as the active display. The scope dialog boxes reference the active display.

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

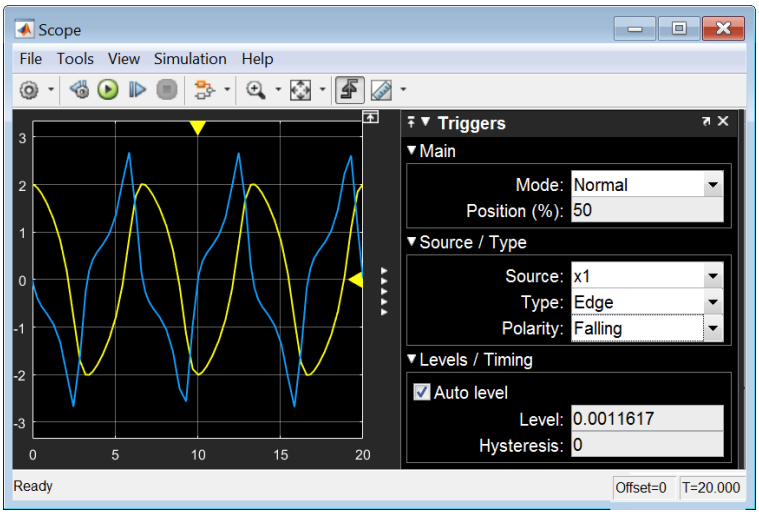
What Is the Trigger Panel

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.

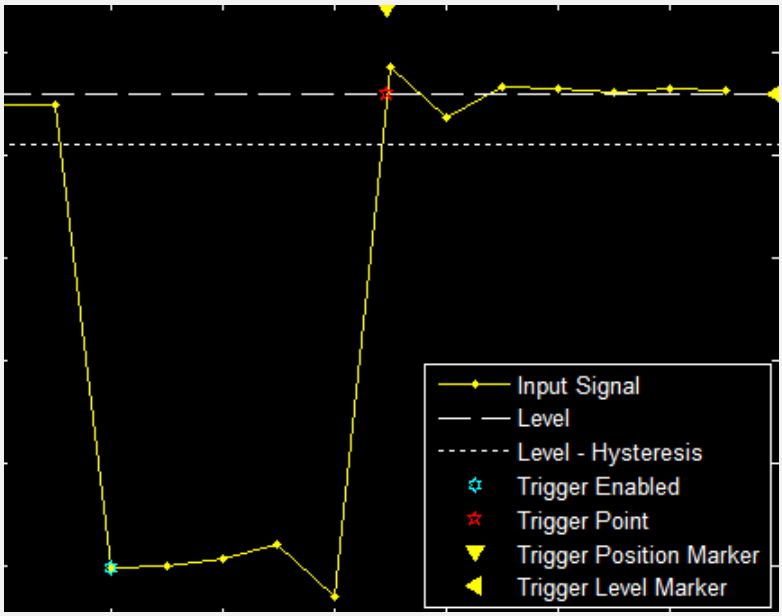
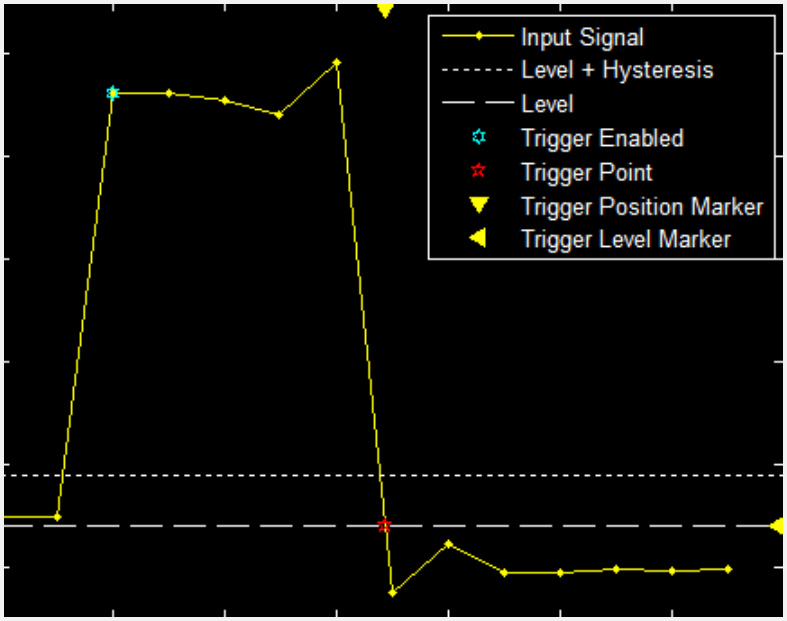
  Normal — Display data from the last trigger event. If no event occurs, the display remains blank.
- **Once** — Display data from the last trigger event and freeze the display. If no event occurs, the display remains blank. Click the **Rearm** button to look for the next trigger event.
- **Off** — Disable triggering.

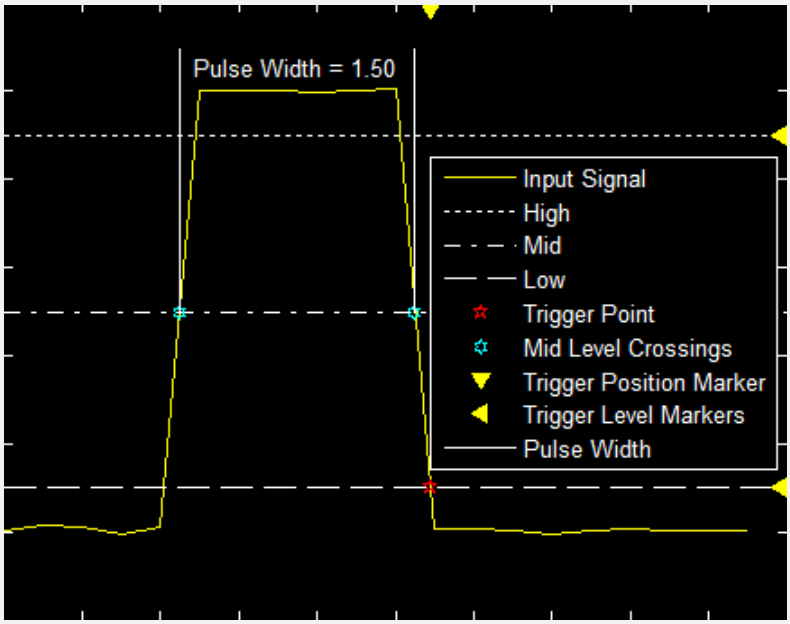
Position (%) — Specify the position of the time pointer along the y-axis. You can also drag the time pointer to the left or right to adjust its position.

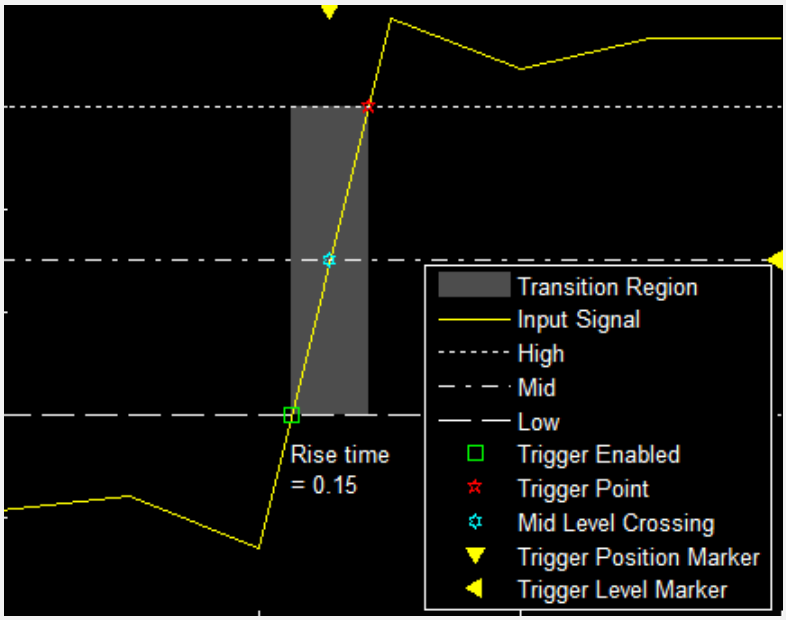
Source/Type and Levels/Timing Panes

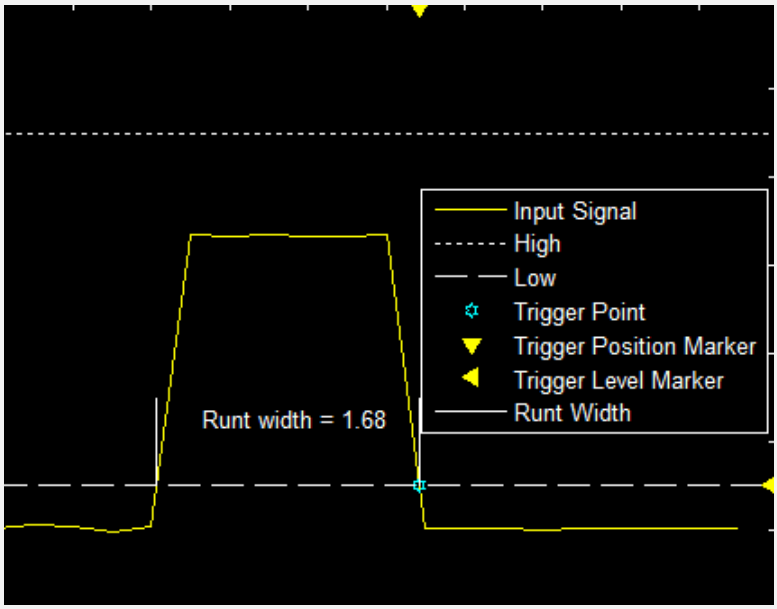
Source — Select a trigger signal. For magnitude and phase plots, select either the magnitude or the phase.

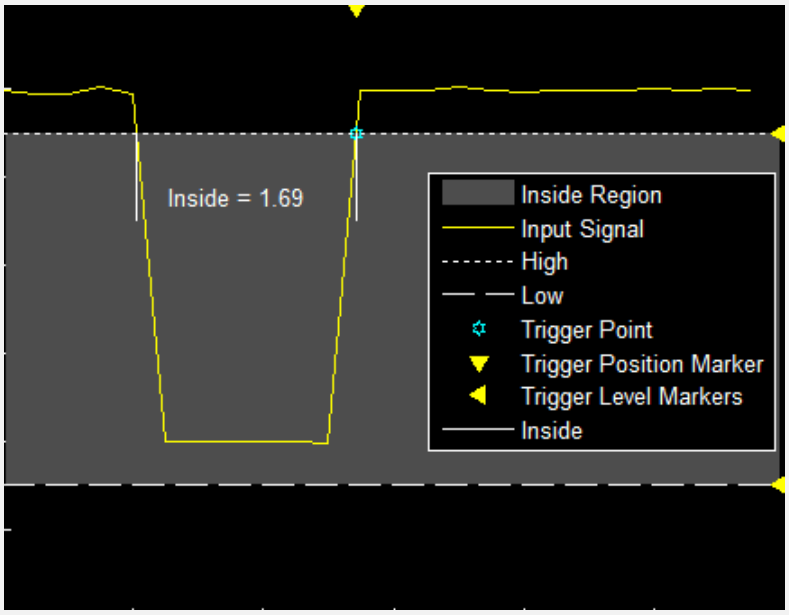
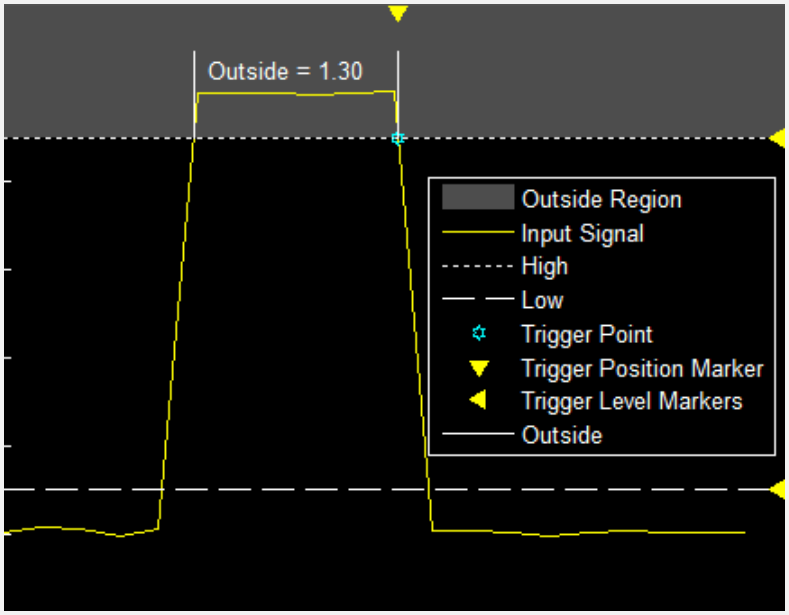
Type — Select the type of trigger.

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.  • `Either` — Trigger when the signal is increasing or decreasing. **Level** — Enter a threshold value for an edge triggered signal. **Auto level** is 50%

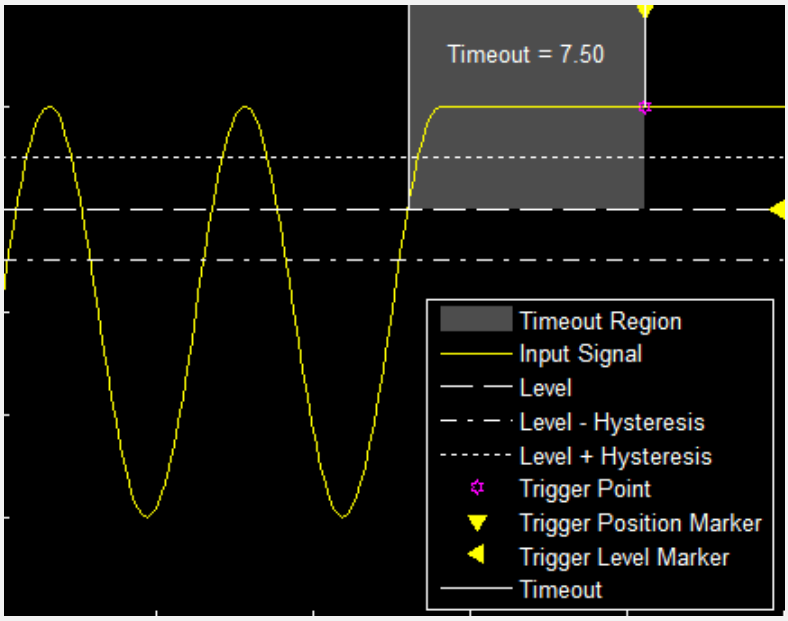
Trigger Type	Trigger Parameters
	Hysteresis — Enter a value for an edge-triggered signal. See “Hysteresis of Trigger Signals” on page 26-63
`Pulse Width` — Trigger when the signal crosses a low threshold and a high threshold twice within a specified time.	**Polarity** — Select the polarity for a pulse 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. **Max Width** — Enter the maximum pulse width for a pulse width triggered signal.

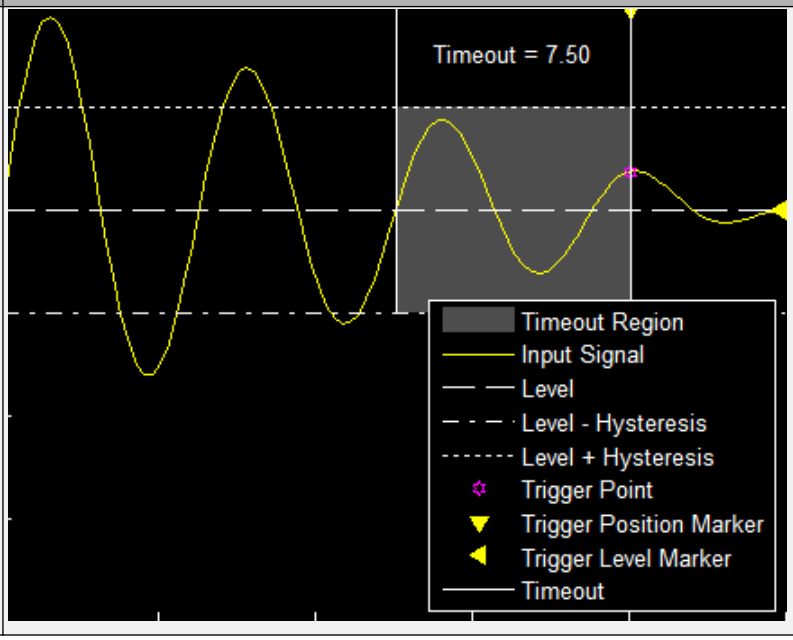
Trigger Type	Trigger Parameters
`Transition` — Trigger on the rising or falling edge of a signal that crosses the high and low levels within a specified time range.	**Polarity** — Select the polarity for a transition-triggered signal. • `Rise Time` — Trigger on an increasing signal when the signal crosses 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.  • `Either` — Trigger when a signal leaves or enters a region between the low and high levels.

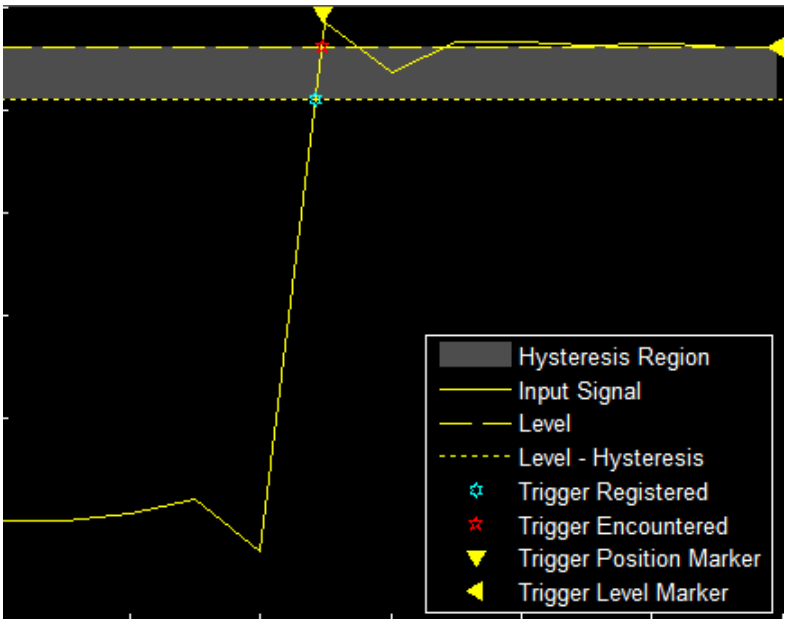
Trigger Type	Trigger Parameters
	High — Enter a high value for a window-triggered signal. **Auto level** is 90%. **Low** — Enter a low value for a window-trigger signal. **Auto level** is 10%. **Min Time** — Enter the minimum time duration for a window-triggered signal. **Max Time** — Enter the maximum 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 26-63. **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.

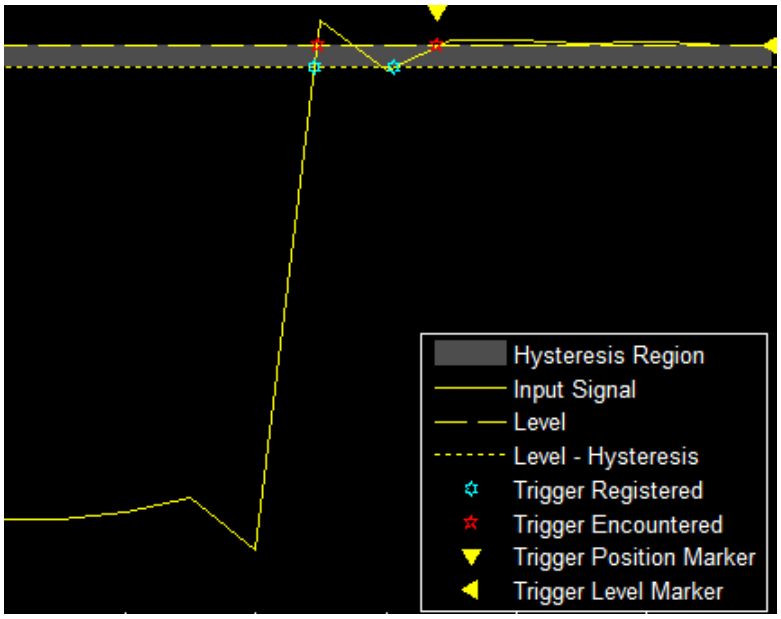
Trigger Type	Trigger Parameters
	

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

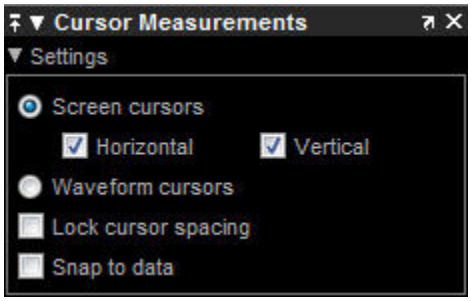
The **Cursor Measurements** panel displays screen cursors. The panel provides two types of cursors for measuring signals. Waveform cursors are vertical cursors that track along the signal. Screen cursors are both horizontal and vertical cursors that you can place anywhere in the display.

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:

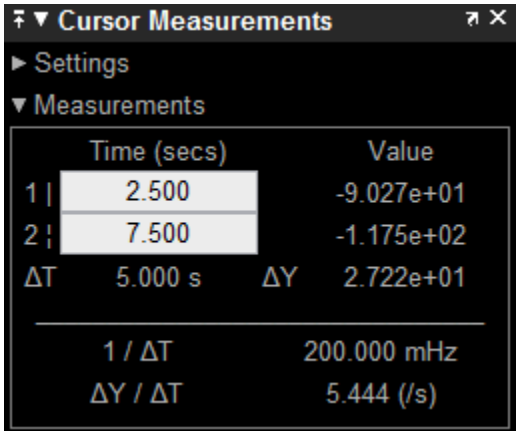
- From the menu, select **Tools > Measurements > Cursor Measurements**.
- On the toolbar, click the Cursor Measurements button.

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.

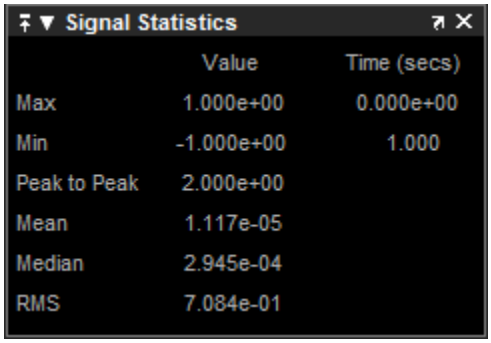


- **1** — View or modify the time or value at cursor number one (solid line cursor).
- **2** — View or modify the time or value at cursor number two (dashed line cursor).
- **ΔT** or **ΔX** — Shows the absolute value of the time (*x*-axis) difference between cursor number one and cursor number two.
- **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

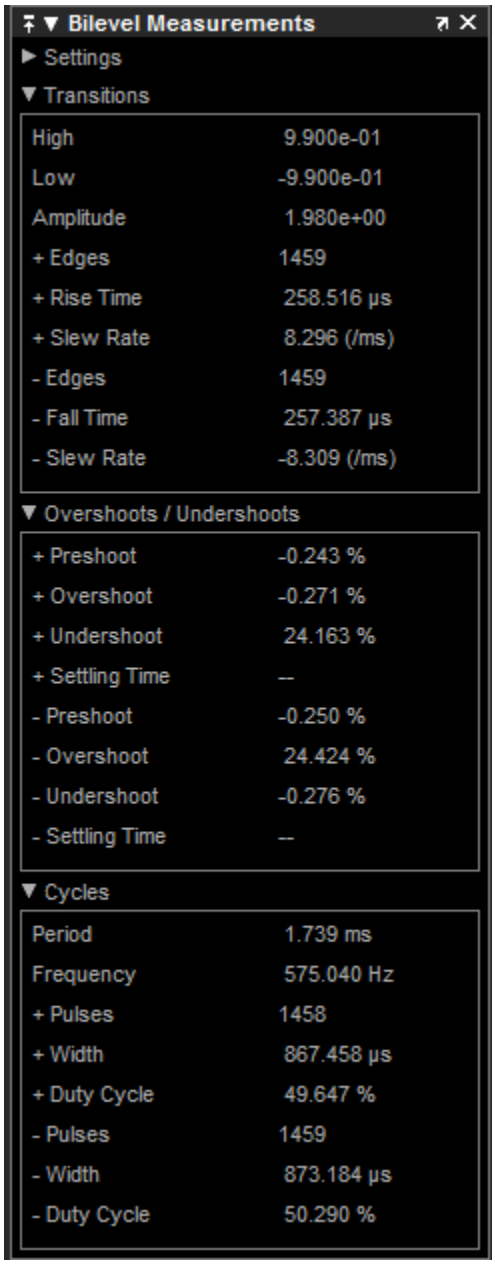
Bilevel Measurements

Display information about signal transitions, overshoots, undershoots, and cycles. To open the Bilevel Measurements panel:

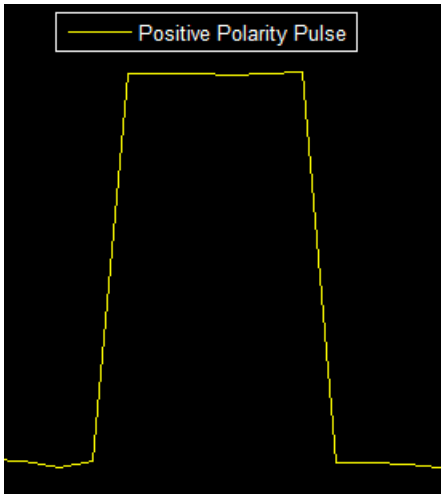
- From the menu, select **Tools > Measurements > Bilevel Measurements**.
- On the toolbar, click the Bilevel Measurements button.

Settings

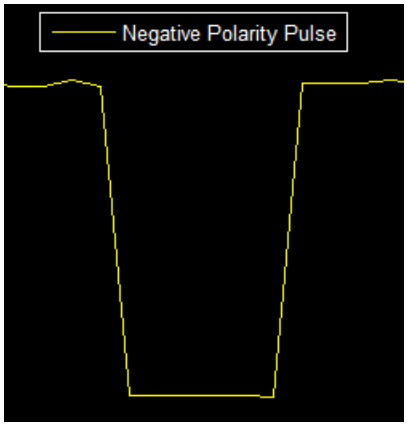
The **Settings** pane enables you to modify the properties used to calculate various measurements involving transitions, overshoots, undershoots, and cycles. You can modify the high-state level, low-state level, state-level tolerance, upper-reference level, mid-reference level, and lower-reference level.



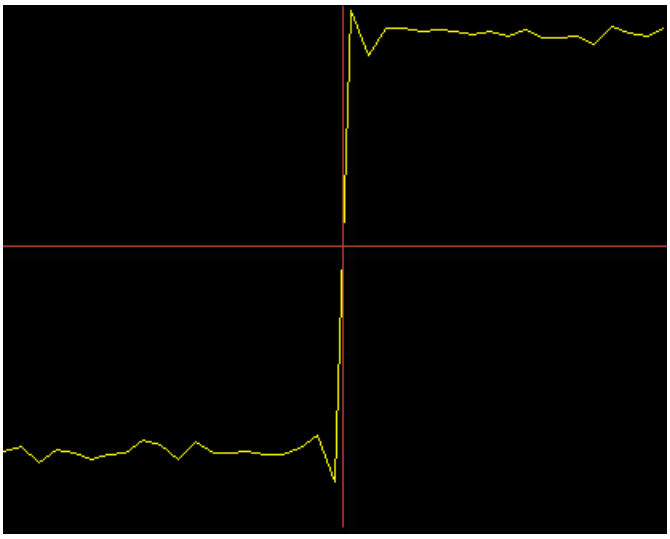
- **Auto State Level** — When this check box is selected, the Bilevel measurements panel detects the high- and low- state levels of a bilevel waveform. When this check box is cleared, you can enter in values for the high- and low- state levels manually.
    - **High** — Used to specify manually the value 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.

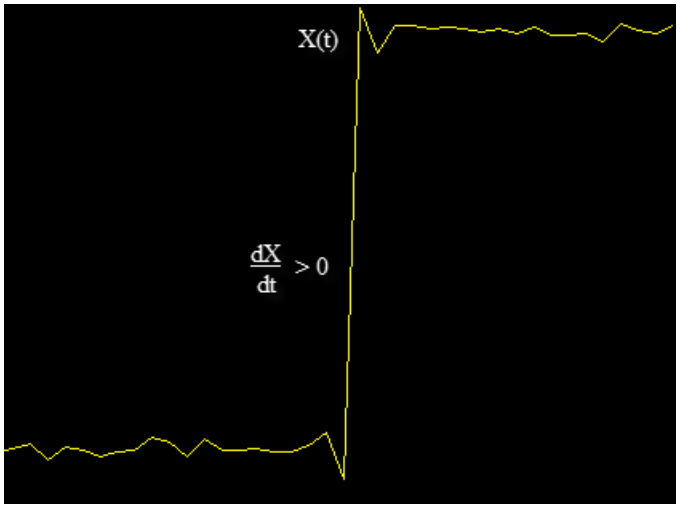


- **Lower Ref Level** — Used to compute the end of the fall-time measurement or the start of the rise-time measurement. This value is expressed as a percentage of the difference between the high- and low-state levels.
- **Settle Seek** — The duration after the mid-reference level instant when each transition 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

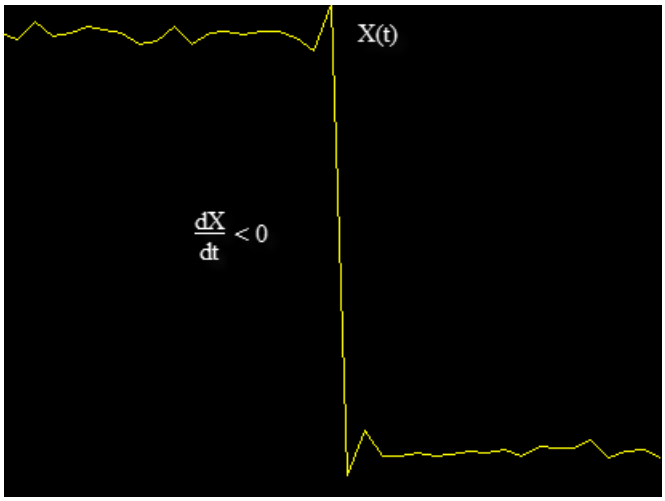
Display calculated measurements associated with the input signal changing between its two possible state level values, high and low.

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.

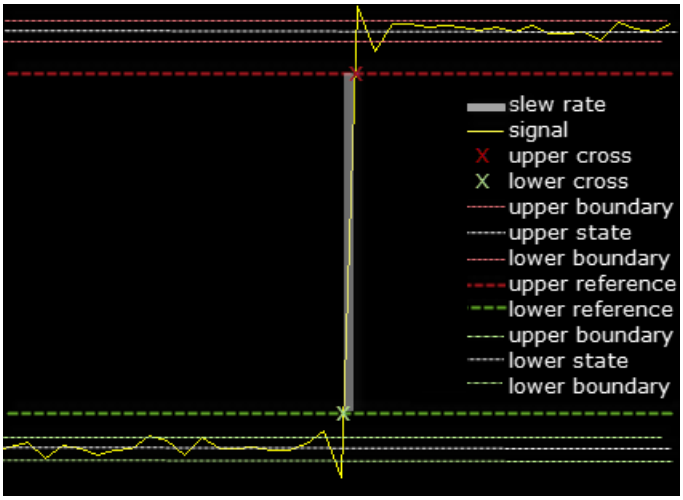
A negative-going transition, or falling edge, in a bilevel waveform is a transition from the high-state level to the low-state level. A negative-going transition has a slope value less than zero. The following figure shows a negative-going transition.



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.

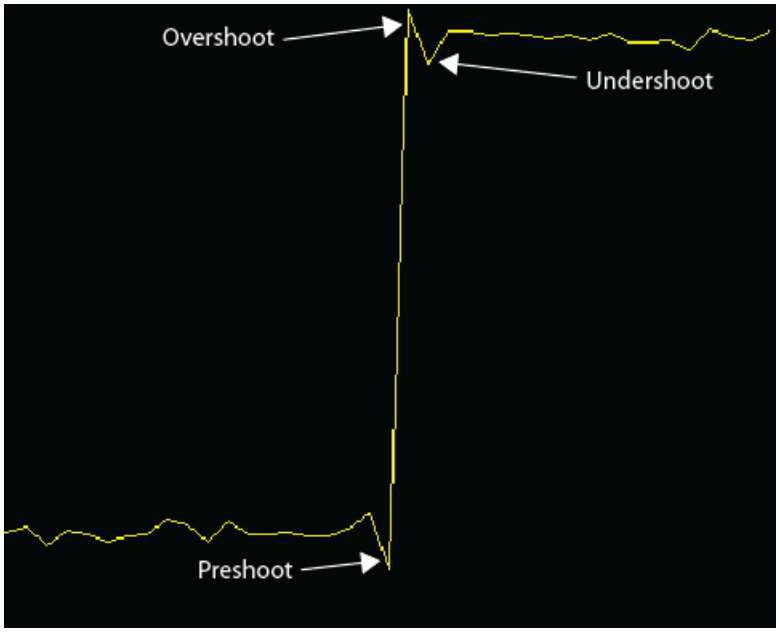


- **– Edges** — Total number of negative-polarity or falling edges counted within the displayed portion of the input signal.
- **– Fall Time** — Average amount of time required for each falling edge to cross from the upper-reference level to the lower-reference level.
- **– Slew Rate** — Average slope of each falling edge transition line within the upper- and lower-percent reference levels in the displayed portion of the input signal.

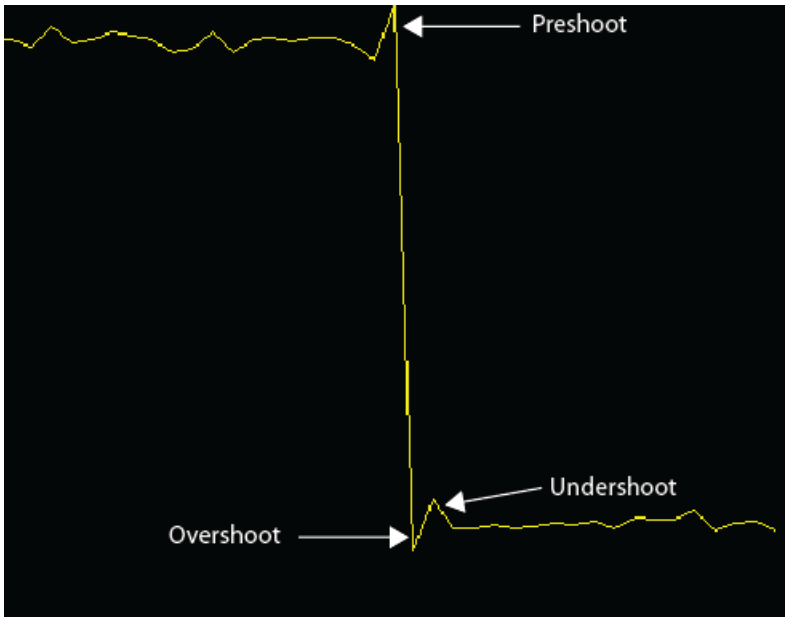
Overshoots / Undershoots Pane

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.

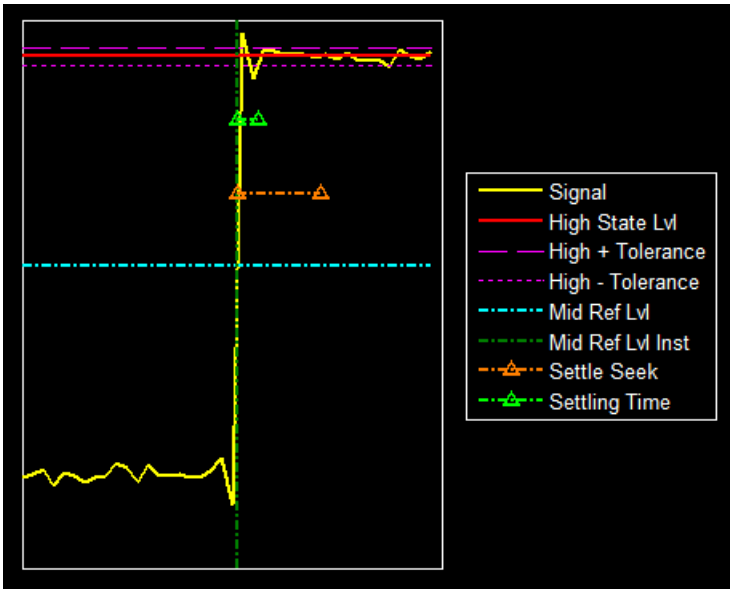
This figure shows preshoot, overshoot, and undershoot for a rising-edge transition.



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.

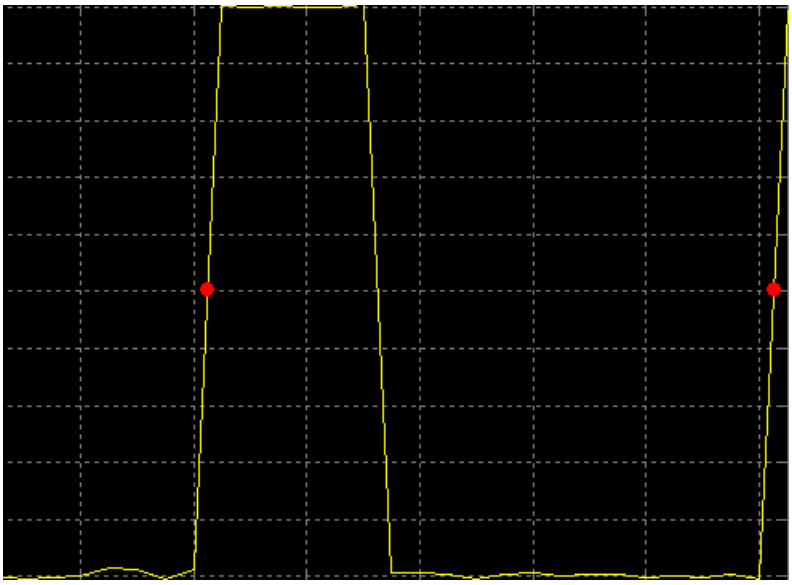
- **– Preshoot** — Average highest aberration in the region immediately preceding each falling transition.
- **– Overshoot** — Average highest aberration in the region immediately following each falling transition.
- **– Undershoot** — Average lowest aberration in the region immediately following each 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.

Use Bilevel Measurements Panel with Clock Input Signal

This example shows how to use the Bilevel Measurements panel in the Time Scope block.

Open the example model `ex-timescope-clockex`:

```
open_system("ex_timescope_clockex")
```

In this example, Simulink® imports the variable *x* , from the MATLAB® workspace. This variable is created when the model loads because the commands that construct it reside in the model Preload function. To view these commands,

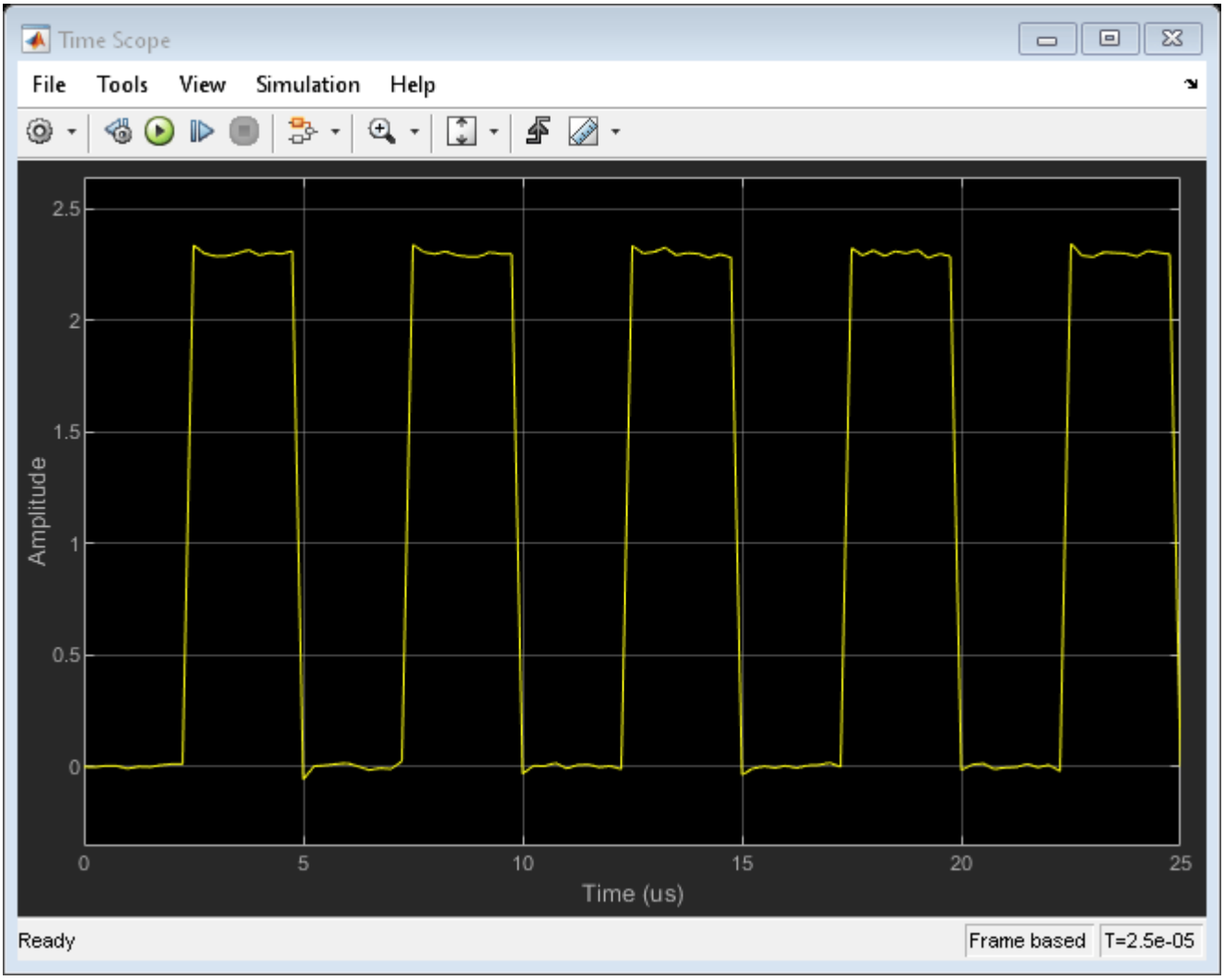
1. On the Simulink toolbar, on the Modeling tab, in the Setup section, in the drop-down, select `Model Properties`.
2. In the Model Properties dialog box, select the `Callbacks` tab. The following lines of MATLAB® code appear.

```
load clockex;
ts = t(2)-t(1);
```

Run your model and open the Time Scope block to see the time domain output.

To show the `Bilevel Measurements` panel: # In the Time Scope menu, select `Tools > Measurements > Bilevel Measurements`. # Collapse the `Transitions` section and Expand the `Settings` and `Overshoots/Undershoots` sections.

```
sim("ex_timescope_clockex")
open_system("ex_timescope_clockex/Time Scope")
```



The value for the rising edge `Settling Time` parameter is not displayed because the default value of `Settle Seek` is longer than the entire simulation.

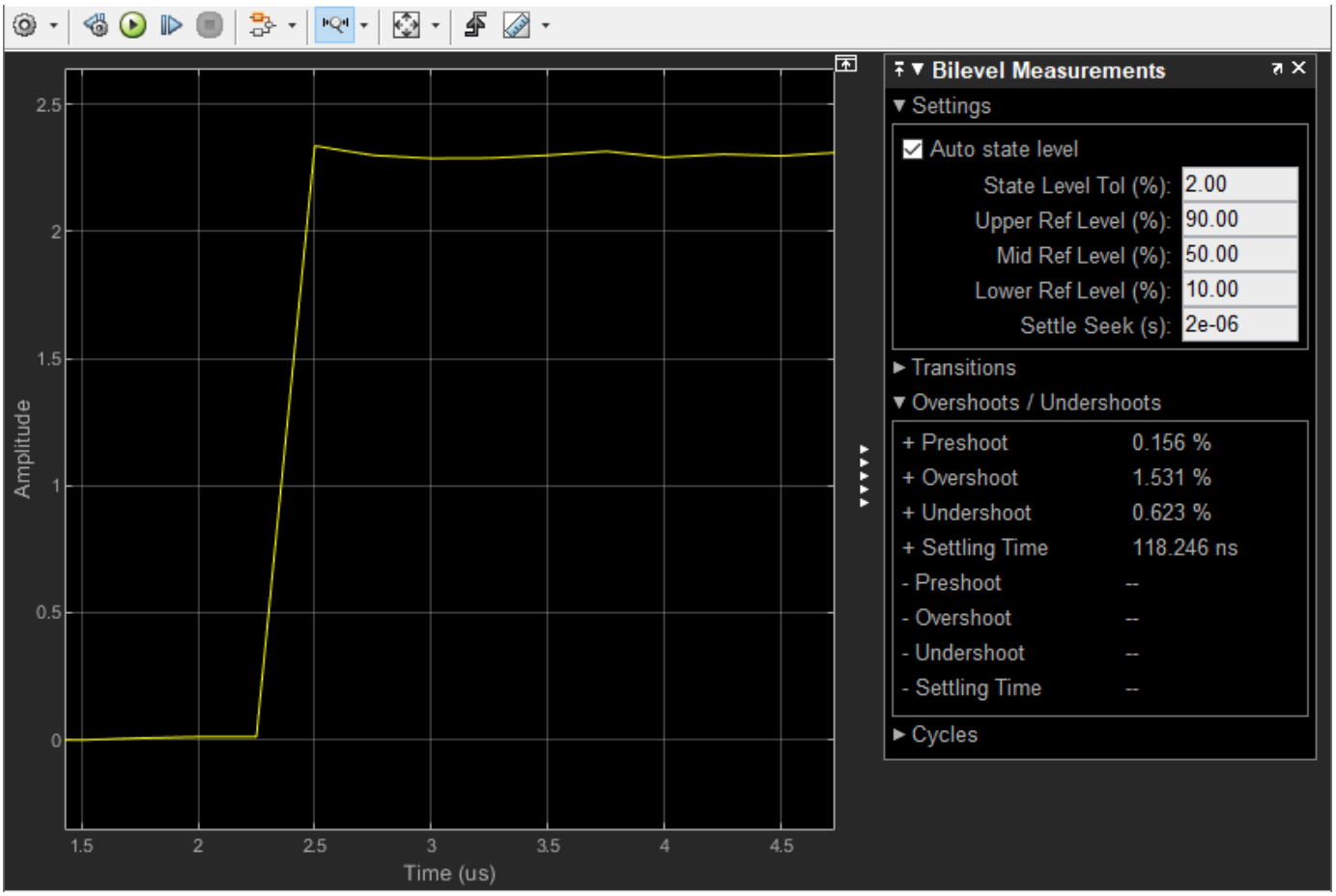
1. Enter a smaller value for `Settle seek` of 2e-6 and press the Enter key. The Time Scope now displays a rising edge settling time value of 118.392 ns.

The settling time value displayed is the statistical average of the settling times for all five rising edges.

To show the settling time for one rising edge, zoom in on that transition.

1. In the Time Scope toolbar, click the Zoom X button.
2. Click the display near a value of 2 microseconds on the time-axis. Drag to the right and release near a value of 4 microseconds on the time-axis.

The Time Scope updates the rising edge settling time value to reflect the new time window.

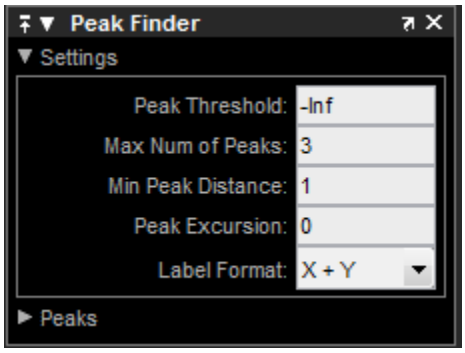


Peak Finder Panel

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.

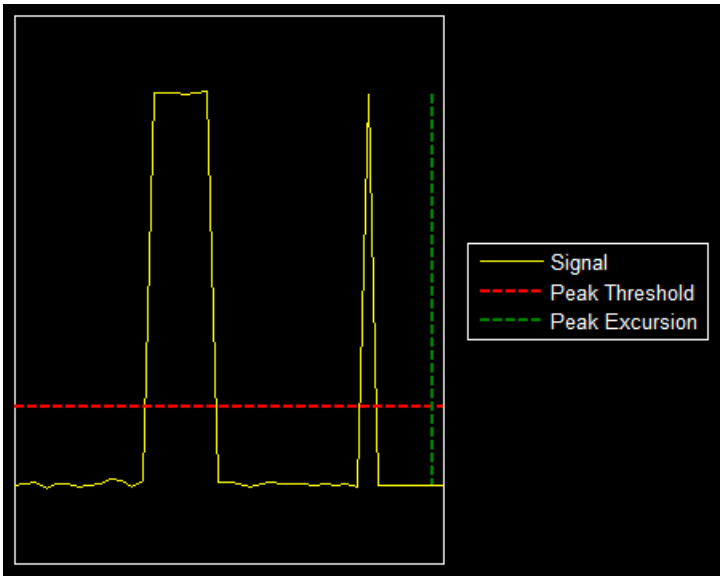
- From the menu, select **Tools** > **Measurements** > **Peak Finder**.
- On the toolbar, click the Peak Finder button.

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

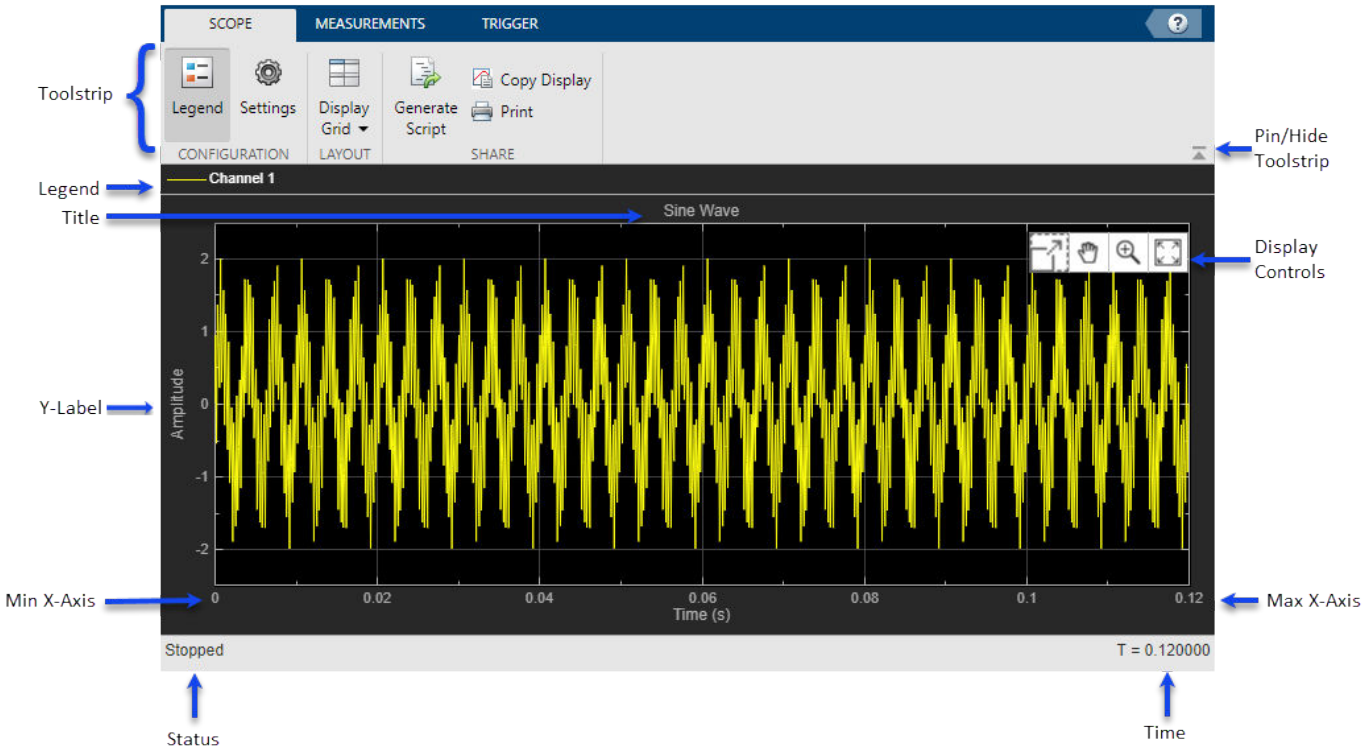
Specify the size of the buffer that the scope holds in its memory cache. Memory is limited by available memory on your system. If your signal has *M* rows of data and *N* data points in each row, *M* x *N* is the number of data points per time step. Multiply this result by the number of time steps for 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.

Configure Time Scope MATLAB Object

When you use the `timescope` object in MATLAB, you can configure many settings and tools from the window. These sections show you how to use the Time Scope interface and the available tools.

Signal Display

This figure highlights the important aspects of the Time Scope window in MATLAB.



- **Min X-Axis** — Time scope sets the minimum *x*-axis limit using the value of the `TimeDisplayOffset` property. To change the **Time Offset** from the Time Scope window, click **Settings** ( ) on the **Scope** tab. Under **Data and Axes**, set the **Time Offset**.
- **Max X-Axis** — Time scope sets the maximum *x*-axis limit by summing the value of the **Time Offset** property with the span of the *x*-axis values. If **Time Span** is set to `Auto`, the span of *x*-axis is `10/SampleRate`.

  The values on the *x*-axis of the scope display remain the same throughout the simulation.
- **Status** — Provides the current status of the plot. The status can be:
  - `Processing` — Occurs after you run the object and before you run the `release` function.
  - `Stopped` — Occurs after you create the scope object and before you first call the object. This status also occurs after you call `release`.
- **Title**, **YLabel** — You can customize the title and the *y*-axis label from **Settings** or by using the `Title` and `YLabel` properties.

- **Toolstrip**

  - **Scope** tab — Customize and share the time scope. For example, showing and hiding the legend
  - **Measurements** tab — Turn on and control different measurement tools.
  - **Trigger** tab — Turn on and modify triggers.

  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, the legend for a two-channel signal will display the default names `Channel 1`, `Channel 2`. To show the legend, on the **Scope** tab, click **Settings** ( ). Under **Display and Labels**, select **Show Legend**. If there are a total of seven input channels, the legend displayed is:

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 legend. 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 this order:

To choose line colors or background colors, on the **Scope** tab click **Settings**.Use the **Axes** color pallet 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 Scope Settings

On the **Scope** tab, the **Configuration** section allows you to modify the scope.

- 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 **Settings** button opens the settings window which allows you to customize the data, axes, display settings, labels, and color settings.

On the **Scope** tab, the **Layout** section allows you to modify the scope layout dimensions.

The **Display Grid** button enables you to select the display layout of the scope.

Use timescope Measurements and Triggers

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 on the **Measurements** tab.

Data Cursors

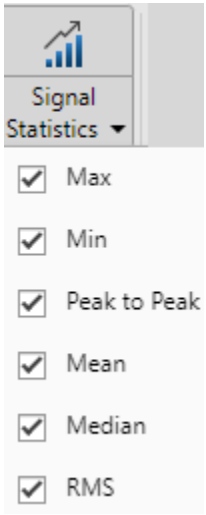
Use the **Data Cursors** button to display screen cursors. Each cursor tracks a vertical line along the signal. The difference between *x*- and *y*-values of the signal at the two cursors is displayed in the box between the cursors.

Signal Statistics

Use the **Signal Statistics** button to display various statistics about the selected signal at the bottom of the time scope 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. End points are not considered peaks. For more information on the algorithms used, see the `findpeaks` function.

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 `timescope` 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.

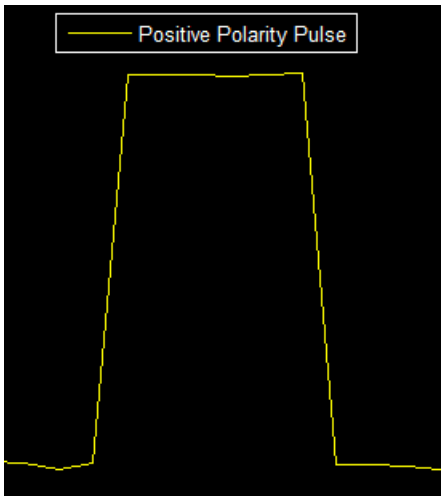
Bilevel Measurements

With bilevel measurements, you can measure transitions, aberrations, and cycles.

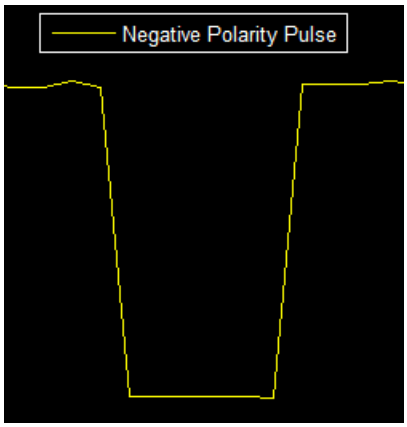
Bilevel Settings

When using bilevel measurements, you can set these properties:

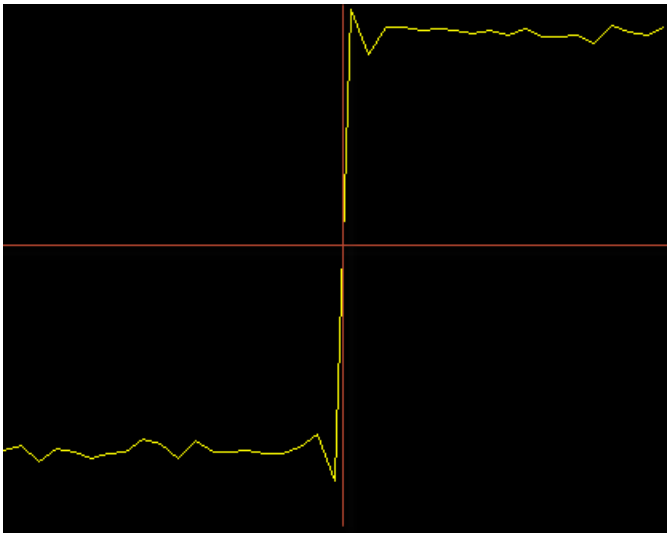
- **Auto State Level** — When this check box is selected, the Bilevel measurements panel detects the high- and low-state levels of a bilevel waveform. When this check box is cleared, you can enter in values for the high- and low-state levels manually.
  - **High** — Manually specify the value that denotes a positive polarity or high-state level.



  - **Low** — Manually specify the value that denotes a negative polarity or low-state level.



- **State Level Tol. %** — Tolerance levels within which the initial and final levels of each transition must lie to 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 a horizontal line, and its corresponding mid-reference level instant is shown as a vertical line.

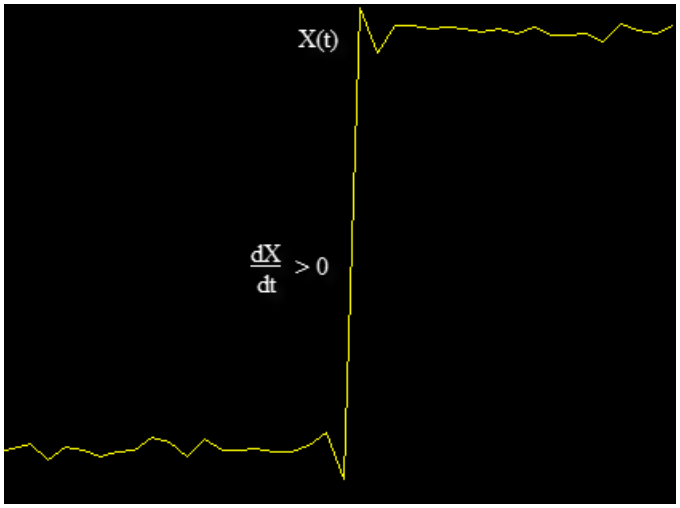


- **Lower Ref Level** — Used to compute the end of the fall-time measurement or the start of the rise-time measurement. This value is expressed as a percentage of the difference between the high- and low-state levels.
- **Settle Seek** — The duration after the mid-reference level instant when each transition occurs is used for computing a valid settling time. Settling time is displayed in the **Aberrations** pane.

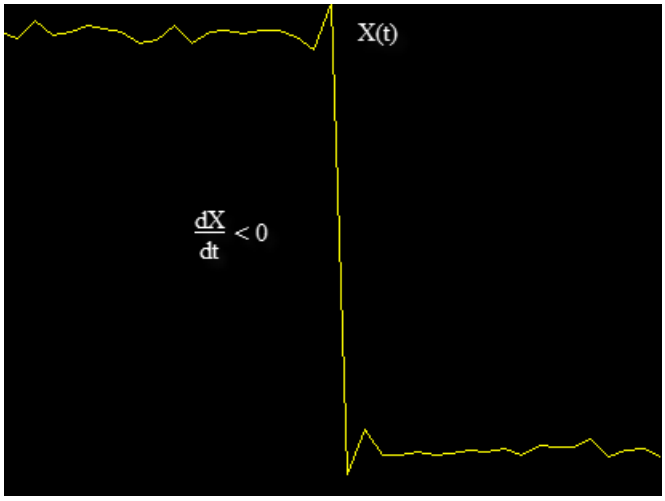
Transitions

Select **Transitions** to display calculated measurements associated with the input signal changing between its two possible state level values, high and low. The measurements are displayed in the **Transitions** pane at the bottom of the scope window.

The **+ Edges** row measures rising edges or a positive-going transition. A rising edge in a bilevel waveform is a transition from the low-state level to the high-state level with a slope value greater than zero.



The **- Edges** row measures falling edges or a negative-going transition. A falling edge in a bilevel waveform is a transition from the high-state level to the low-state level with a slope value less than zero.



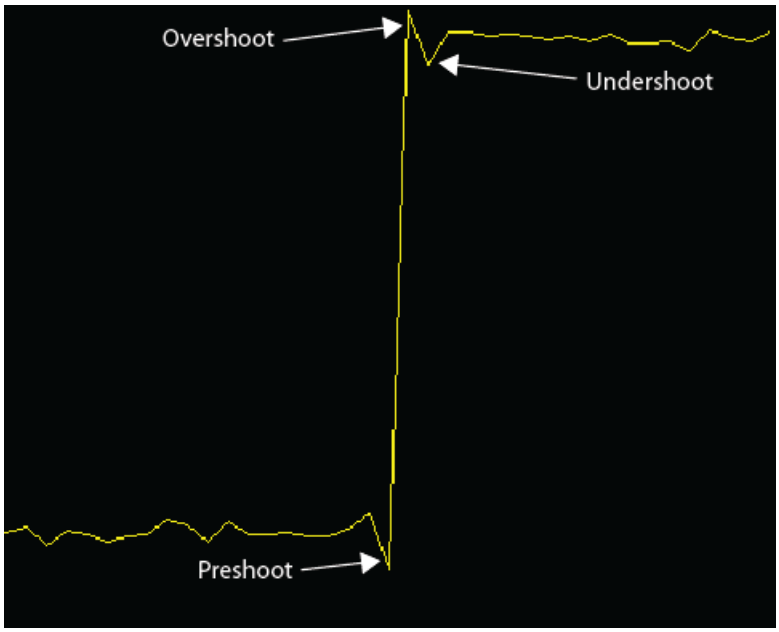
The Transition measurements assume that the amplitude of the input signal is in units of volts. For the transition measurements to be valid, you must convert all input signals to volts.

Aberrations

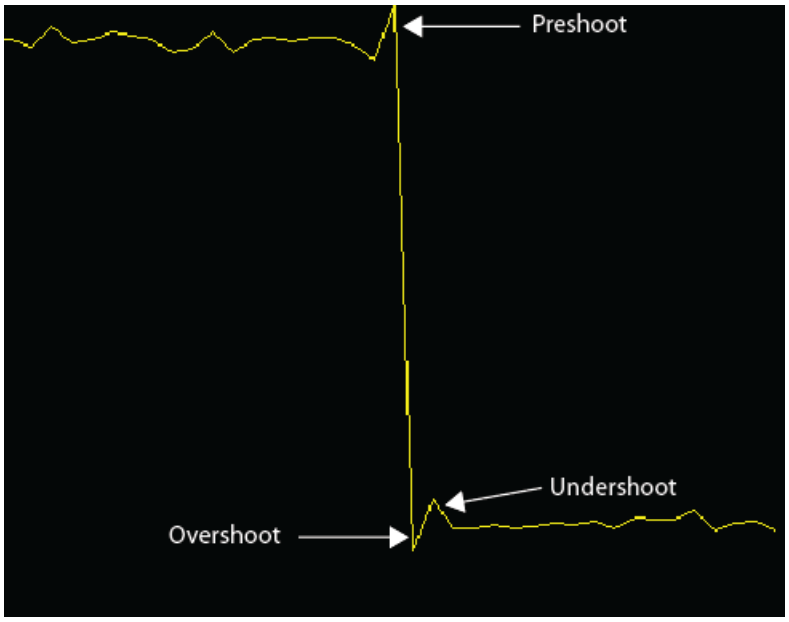
Select **Aberrations** to display calculated measurements involving the distortion and damping of the input signal such as preshoot, overshoot, and undershoot. Overshoot and undershoot, respectively,

refer to the amount that a signal 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. The measurements are displayed in the **Transitions** pane at the bottom of the scope window.

This figure shows preshoot, overshoot, and undershoot for a rising-edge transition.



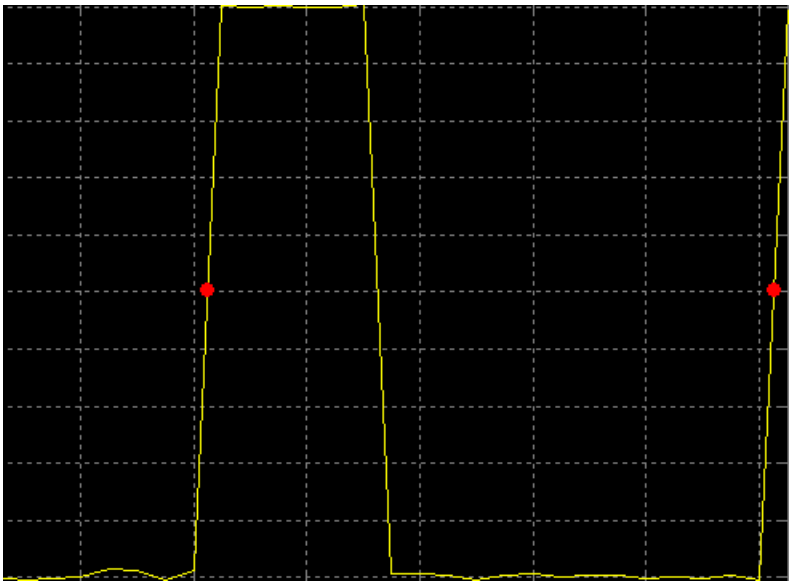
The next figure shows preshoot, overshoot, and undershoot for a falling-edge transition.



Cycles

Select **Cycles** calculates repetitions or trends in the displayed portion of the input signal. The measurements are displayed in the **Cycles** pane at the bottom of the scope window in two rows: **+ Pulses** for the positive-polarity pulses and **- Pulses** for the negative-polarity pulses.

- **Period** — Average duration between adjacent edges of identical polarity within the displayed portion of the input signal. To calculate period, the `timescope` takes the difference between the mid-reference level instants of the initial transition of each pulse and the next identical-polarity transition. These mid-reference level instants for a positive-polarity pulse appear as red dots in the following figure.



- **Frequency** — Reciprocal of the average period, measured in hertz.
- **Count** — Number of positive- or negative-polarity pulses counted.
- **Width** — Average duration between rising and falling edges of each pulse within the displayed portion of the input signal.
- **Duty Cycle** — Average ratio of pulse width to pulse period for each pulse within the displayed portion of the input signal.

Triggers

Define a trigger event to identify the simulation time of specified input signal characteristics. 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 define a trigger:

1. On the Trigger tab of the scope window, select the channel you want to trigger.
2. Specify when the display updates by selecting a triggering **Mode**.

   - **Auto** — Display data from the last trigger event. If no event occurs after one time span, display the last available data.

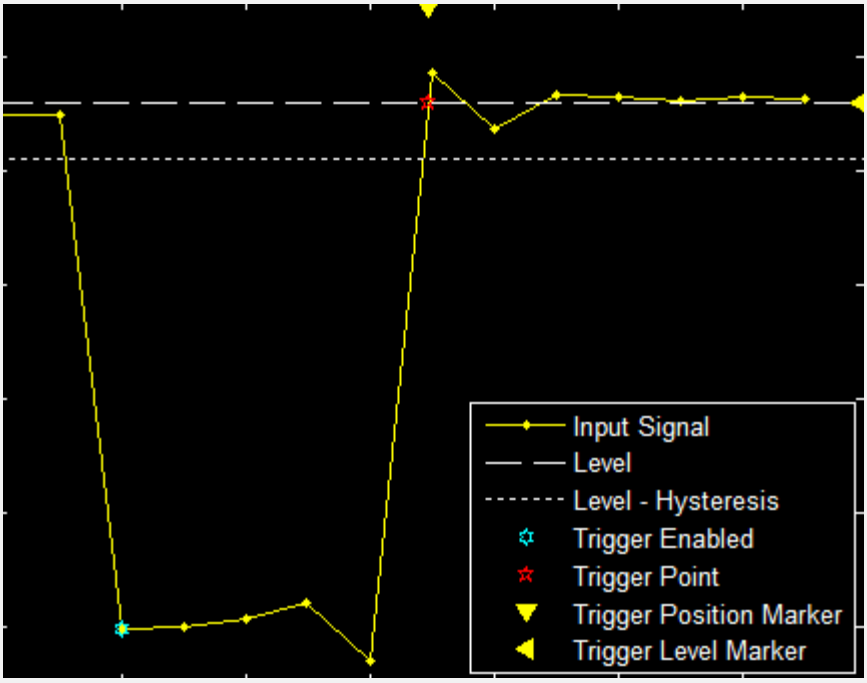
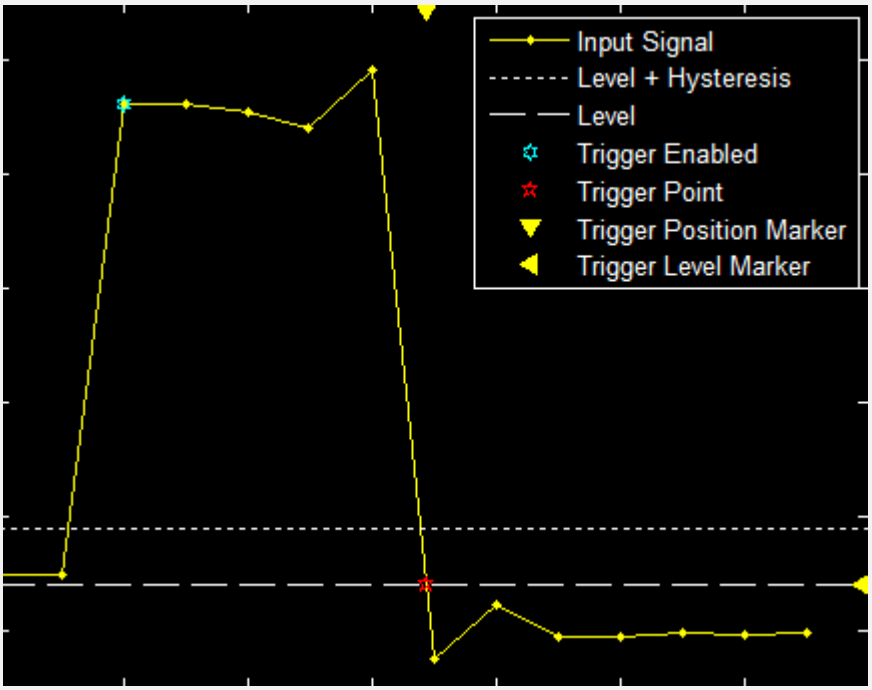
     Normal — Display data from the last trigger event. If no event occurs, the display remains blank.

   - **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.
3. Select a triggering type, polarity, and any other properties. See the Trigger Properties table.

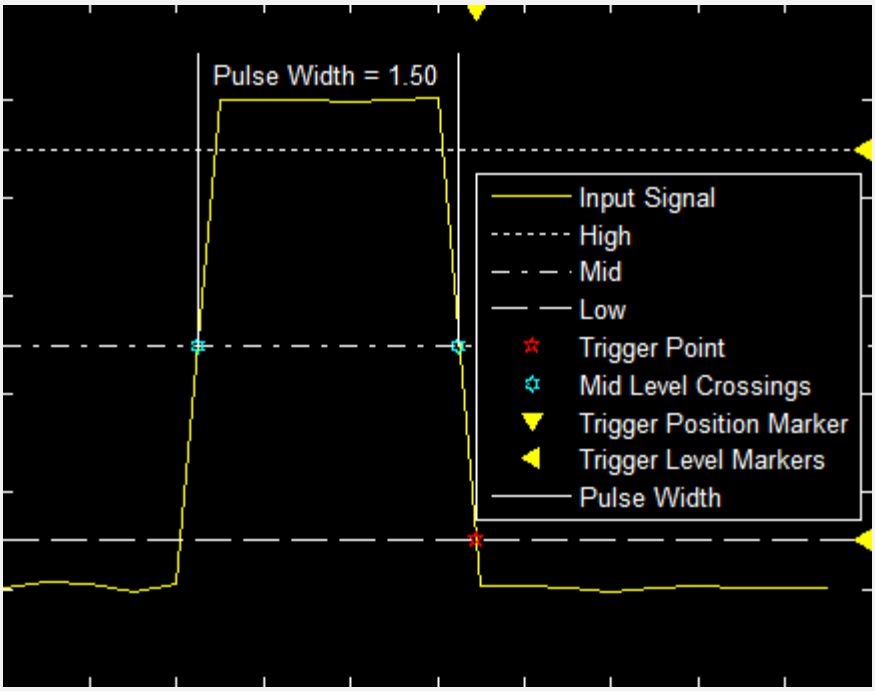
4 Click **Enable Trigger**.

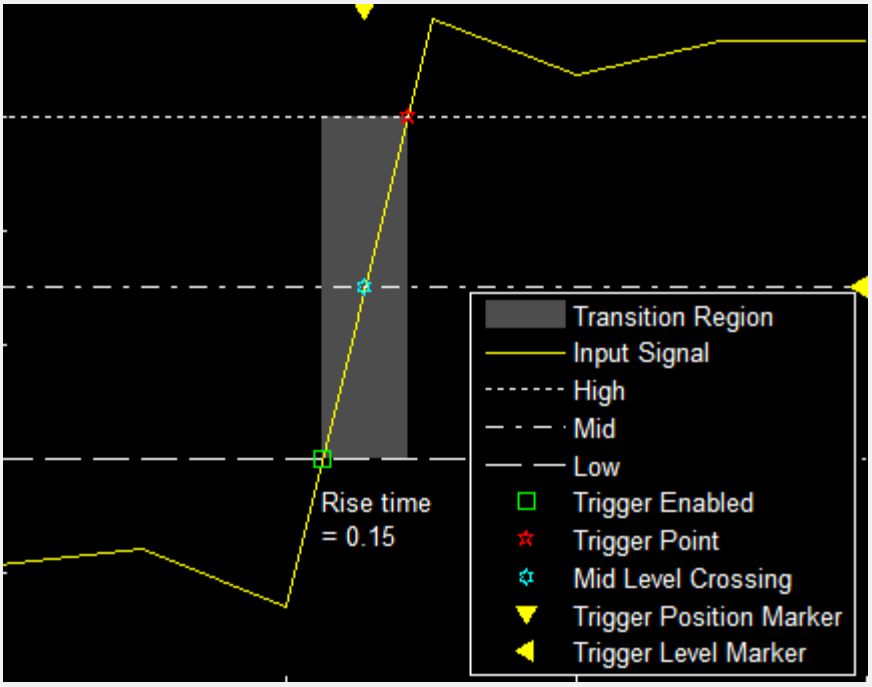
You can set the trigger position to specify the position of the time pointer along the y-axis. You can also drag the time pointer to the left or right to adjust its position.

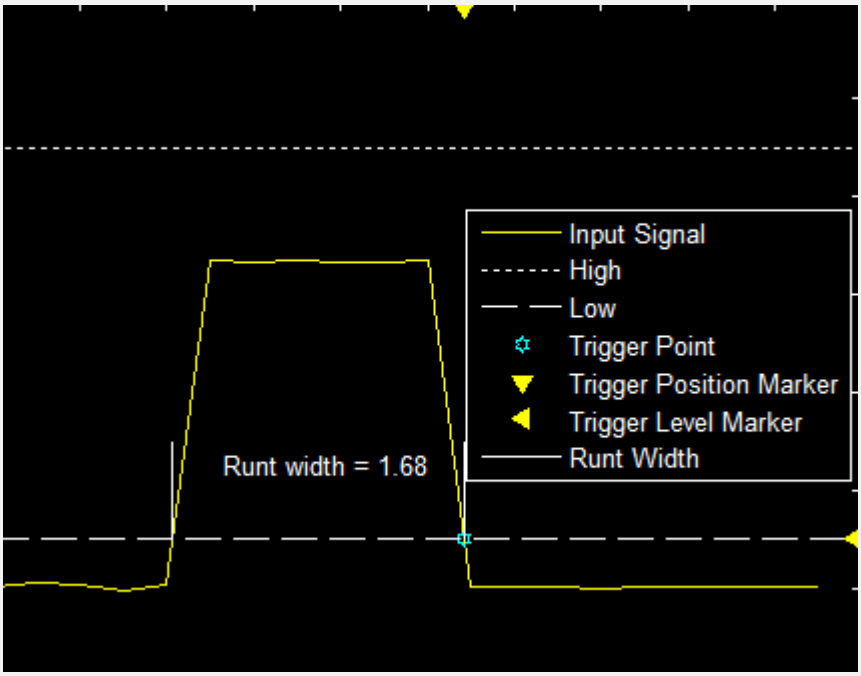
Trigger Properties

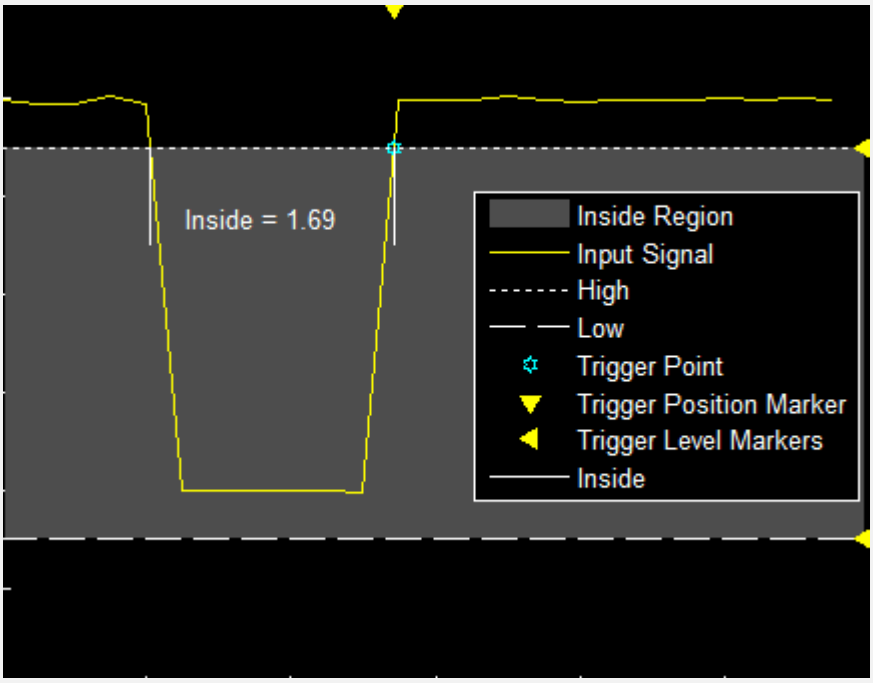
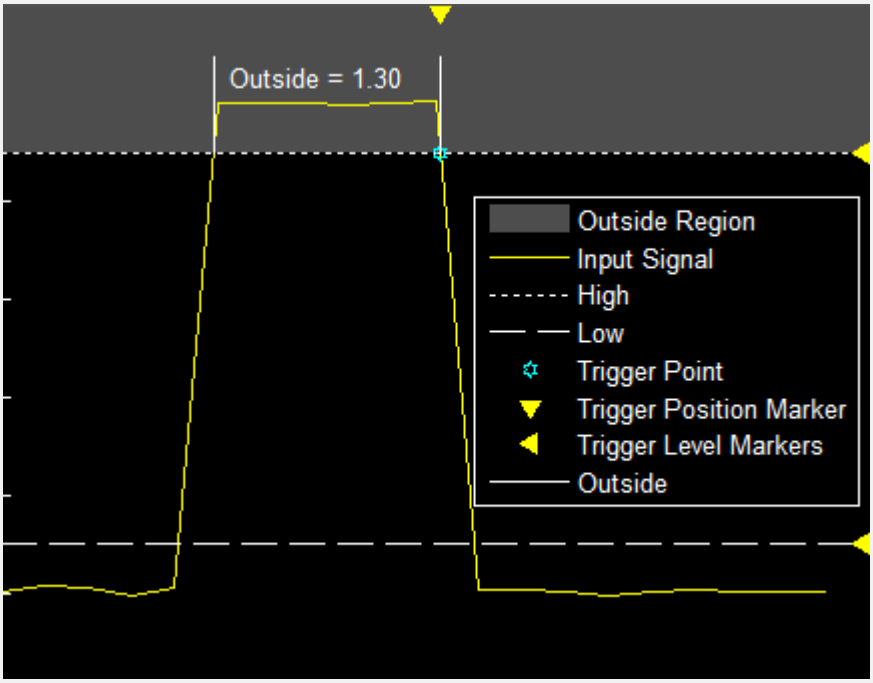
Trigger Type	Trigger Parameters
`Edge` — Trigger when the signal crosses a threshold.	**Polarity** — Select the polarity for an edge-triggered signal. • `Rising` — Trigger when the signal is increasing.  • `Falling` — Trigger when the signal value is decreasing. 

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 26-97 **Delay** — Offset the trigger by a fixed delay in seconds. **Holdoff** — Set the minimum possible time between triggers. **Position** — Set horizontal position of the trigger on the screen.

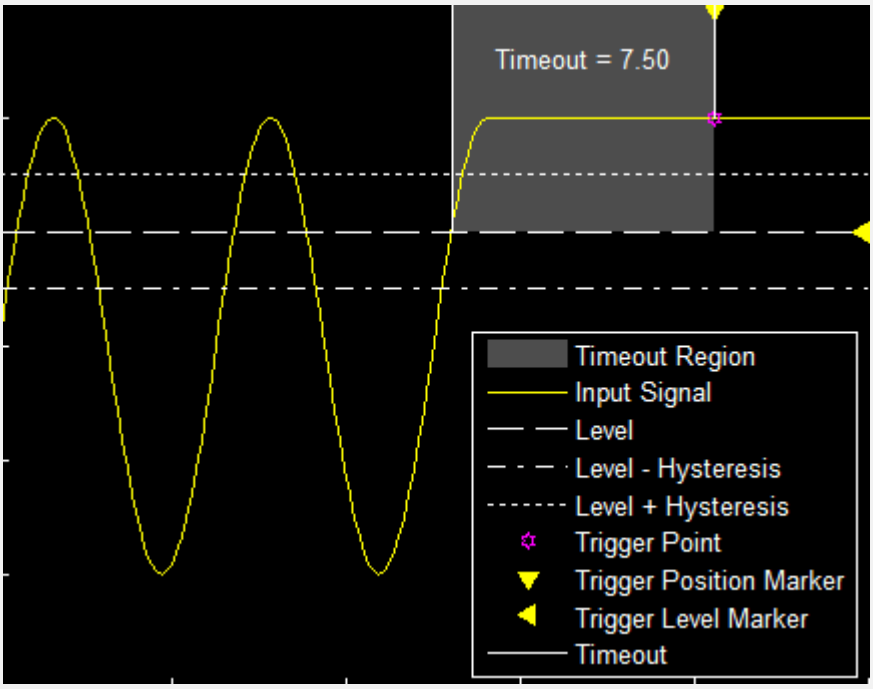
Trigger Type	Trigger Parameters
`Pulse Width` — Trigger when the signal crosses a low threshold and a high threshold twice within a specified time.	**Polarity** — Select the polarity for a pulse width-triggered signal. • `Positive` — Trigger on a positive-polarity pulse when the pulse crosses the low 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. **Max Width** — Enter the maximum pulse width for a pulse-width-triggered signal. **Delay** — Offset the trigger by a fixed delay in seconds.

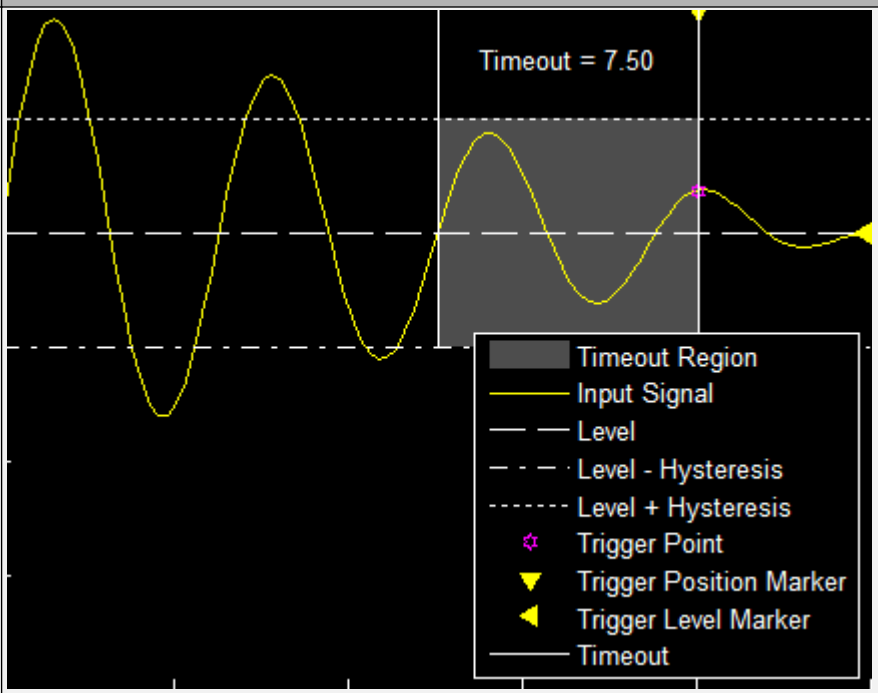
Trigger Type	Trigger Parameters
	Holdoff — Set the minimum possible time between triggers. **Position** — Set horizontal position of the trigger on the screen.
`Transition` — Trigger on the rising or falling edge of a signal that crosses the high and low levels within a specified time range.	**Polarity** — 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. **Delay** — Offset the trigger by a fixed delay in seconds. **Holdoff** — Set the minimum possible time between triggers. **Position** — Set horizontal position of the trigger on the screen.

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. **Delay** — Offset the trigger by a fixed delay in seconds. **Holdoff** — Set the minimum possible time between triggers. **Position** — Set horizontal position of the trigger on the screen.

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. 

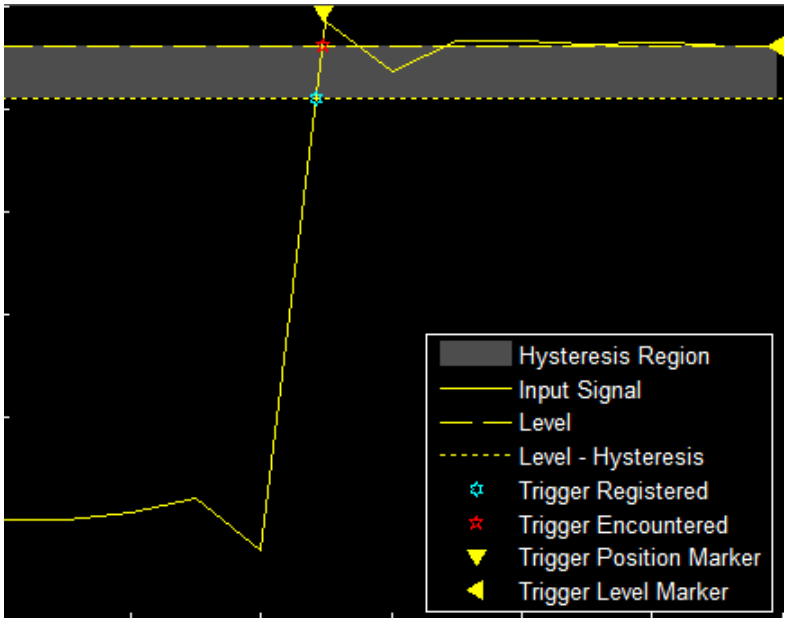
Trigger Type	Trigger Parameters
	• `Either` — Trigger when a signal leaves or enters a region between the low and high levels. **High** — Enter a high value for a window-triggered signal. **Auto level** is 90%. **Low** — Enter a low value for a window-trigger signal. **Auto level** is 10%. **Min Time** — Enter the minimum time duration for a window-triggered signal. **Max Time** — Enter the maximum time duration for a window-triggered signal. **Delay** — Offset the trigger by a fixed delay in seconds. **Holdoff** — Set the minimum possible time between triggers. **Position** — Set horizontal position of the trigger on the screen.

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 26-97. **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.

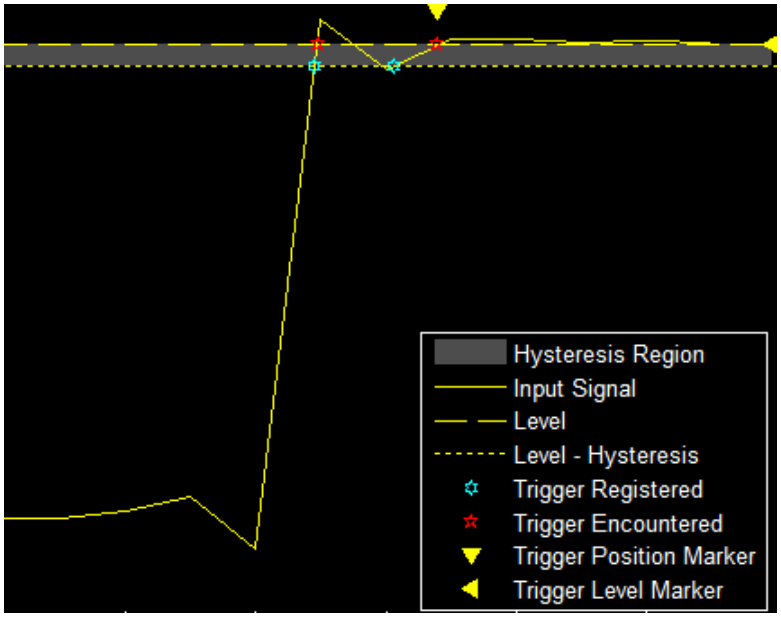
Trigger Type	Trigger Parameters
	 **Delay** — Offset the trigger by a fixed delay in seconds. **Holdoff** — Set the minimum possible time between triggers. **Position** — Set horizontal position of the trigger on the screen.

Hysteresis of Trigger Signals

Hysteresis — 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.



Share or Save the Time Scope

If you want to save the time scope for future use or share it with others, use the buttons in the **Share** section of the **Scope** tab.

- **Generate Script** — Generate a script to re-create your time scope with the same settings. An editor window opens with the code required to re-create your `timescope` object.

- **Copy Display** — Copy the display to your clipboard. You can paste the image in another program to save or share it.
- **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 on the plot.
- — Pan the plot.
- — Autoscale the axes to fit the shown data.

See Also

Objects
`timescope`

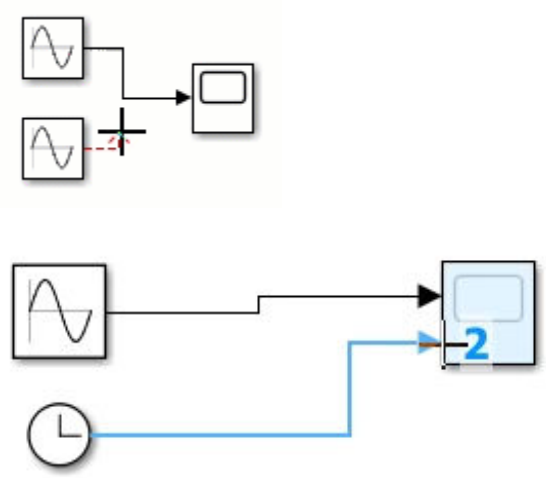
Common Scope Block Tasks

In this section...
“Connect Multiple Signals to a Scope” on page 26-100
“Save Simulation Data Using Scope Block” on page 26-102
“Pause Display While Running” on page 26-104
“Copy Scope Image” on page 26-104
“Plot an Array of Signals” on page 26-106
“Scopes in Referenced Models” on page 26-107
“Scopes Within an Enabled Subsystem” on page 26-111
“Modify x-axis of Scope” on page 26-112
“Show Signal Units on a Scope Display” on page 26-115
“Select Number of Displays and Layout” on page 26-117
“Dock and Undock Scope Window to MATLAB Desktop” on page 26-118

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.



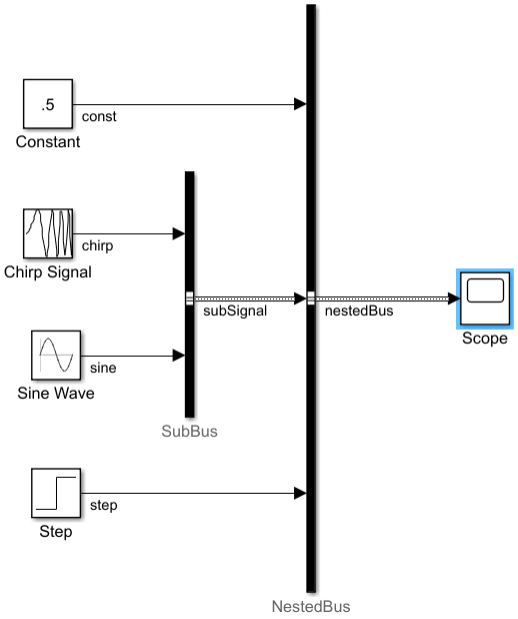
To specify the number of input ports:

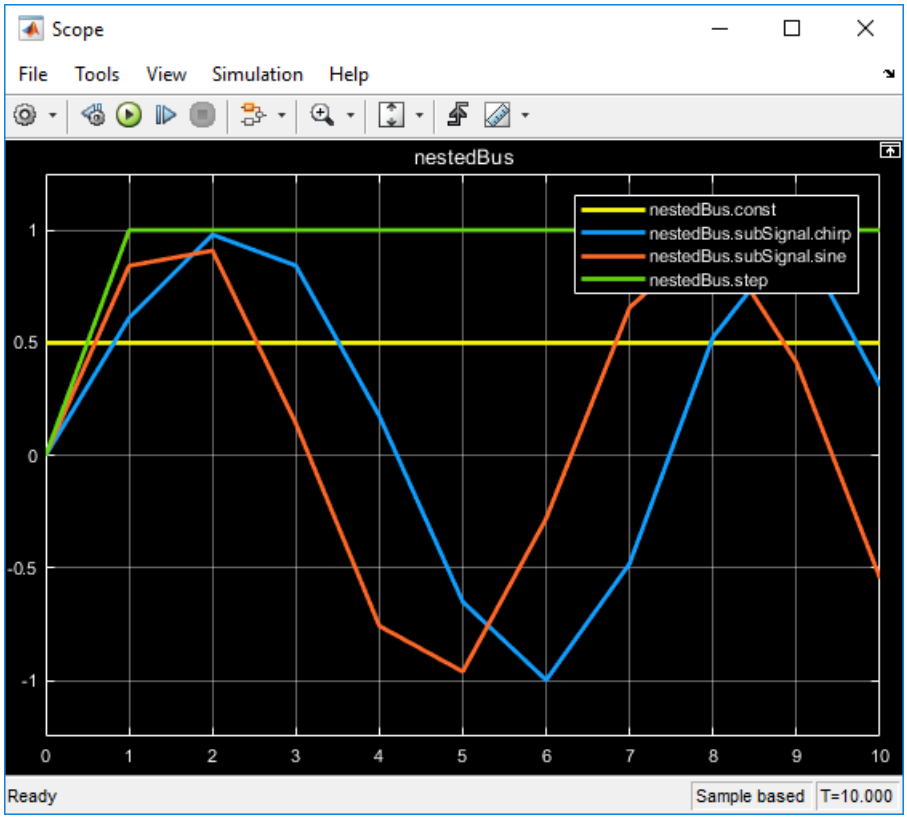
1. Open a scope window.
2. From the toolbar, select **File** > **Number of Input Ports** > **More**.
3. Enter the number of input ports, up to 96.

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.



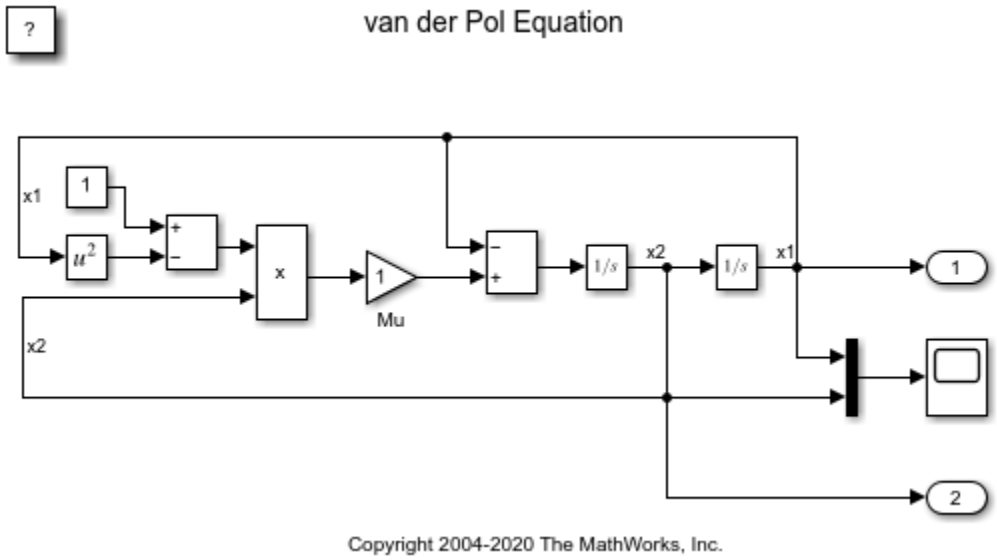


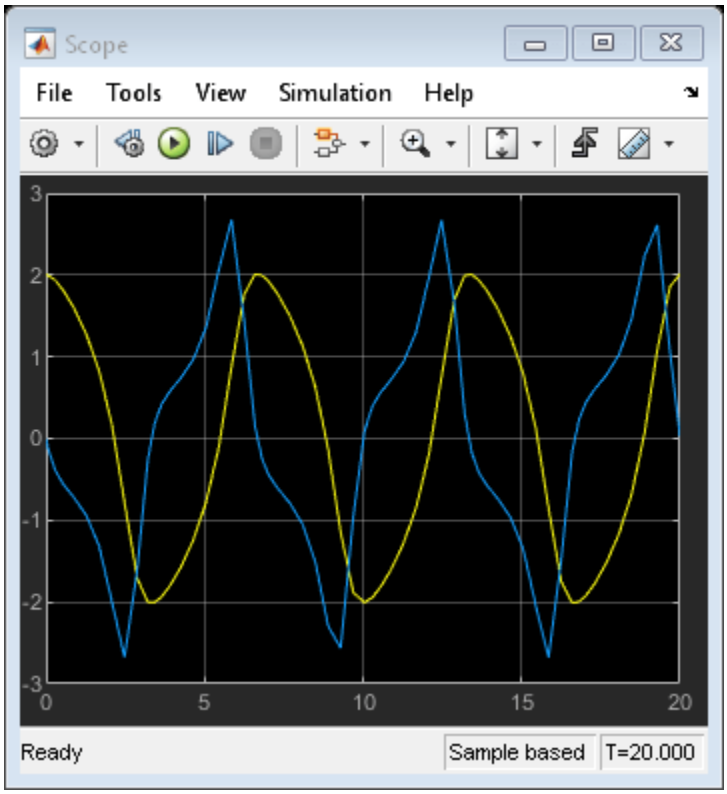
Save Simulation Data Using Scope Block

This example shows how to save signals to the MATLAB Workspace using the Scope block. You can us these steps for the Scope or Time Scope blocks. To save data from the Floating Scope or Scope viewer, see “Save Simulation Data from Floating Scope” (Simulink).

Using the vdp model, turn on data logging to the workspace. You can follow the commands below, or in the Scope window, click the Configuration Properties button and navigate to the Logging tab, turn on **Log data to workspace**.

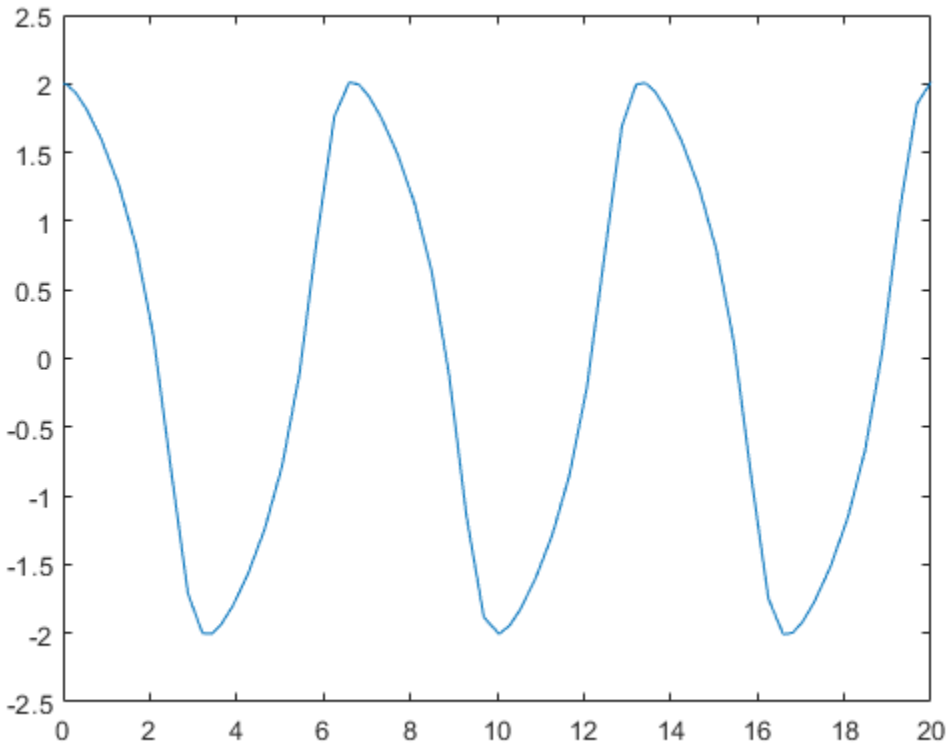
```
vdp
scopeConfig = get_param('vdp/Scope','ScopeConfiguration');
scopeConfig.DataLogging = true;
scopeConfig.DataLoggingSaveFormat = 'Dataset';
out = sim('vdp');
```





In the MATLAB Command window, view the logged data from the `out.ScopeData` structure.

```
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.

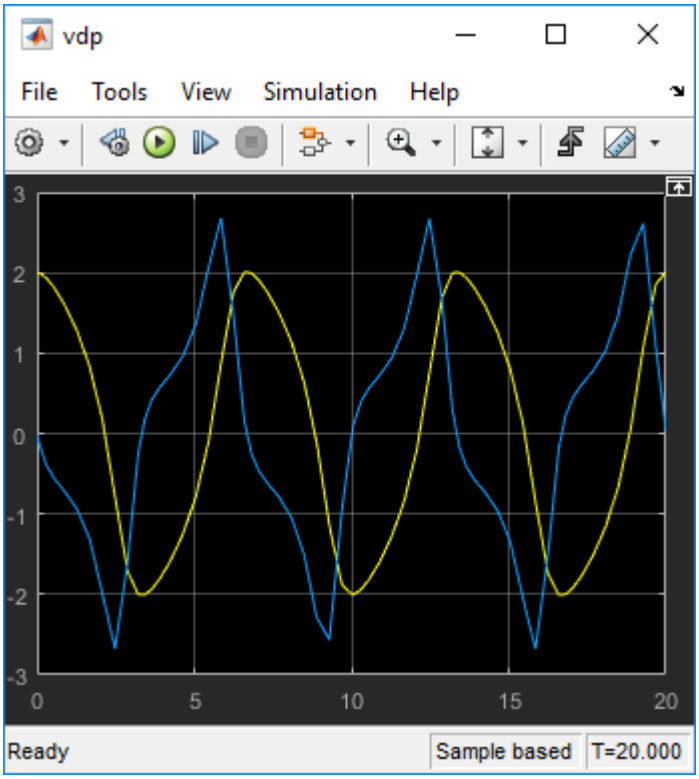
1. Open a scope window and start the simulation.
2. Select **Simulation** > **Simulink Snapshot**.

   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.

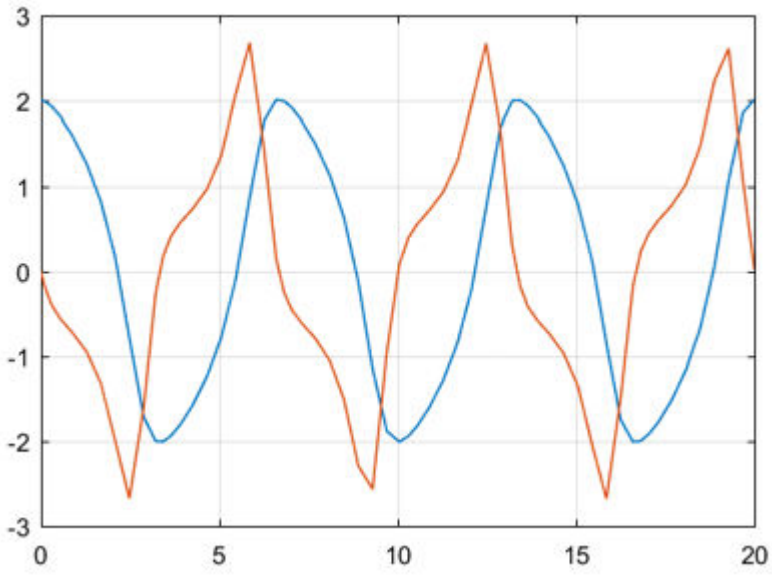
1. Add a scope block to your model.
2. Connect signals to scope ports. See “Connect Multiple Signals to a Scope” on page 26-100. For example, in the `vdp` model, connect the signals `x1` and `x2` to a scope.
3. Open the scope window and run the simulation.



4 Select **File > Copy to Clipboard**.

5 Paste the image into a document.

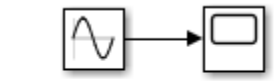
The van der Pol Equation results from my model:



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.

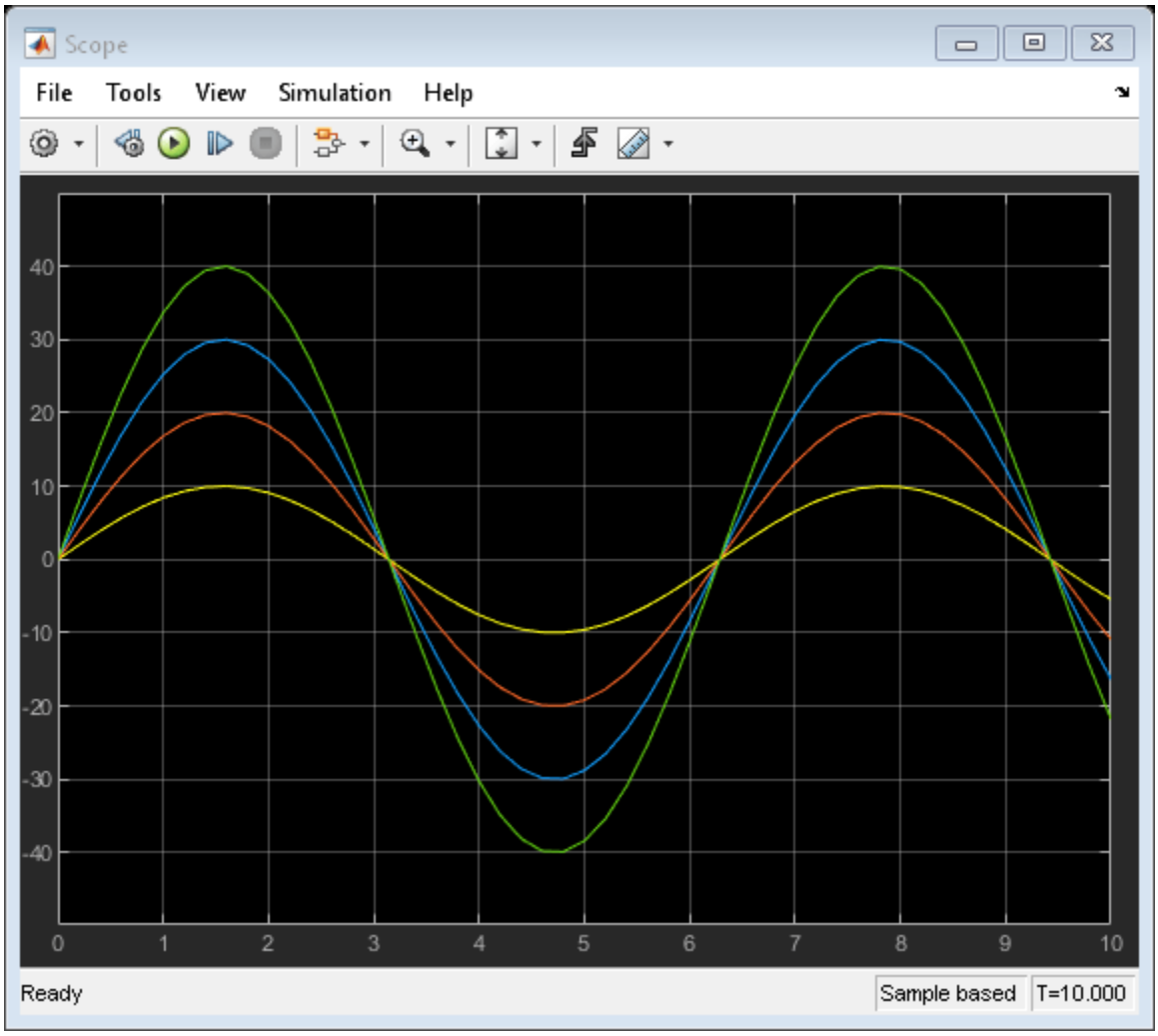
Plot an Array of Signals

This example shows how the scope plots an array of signals.



Copyright 2019 The MathWorks, Inc.

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.

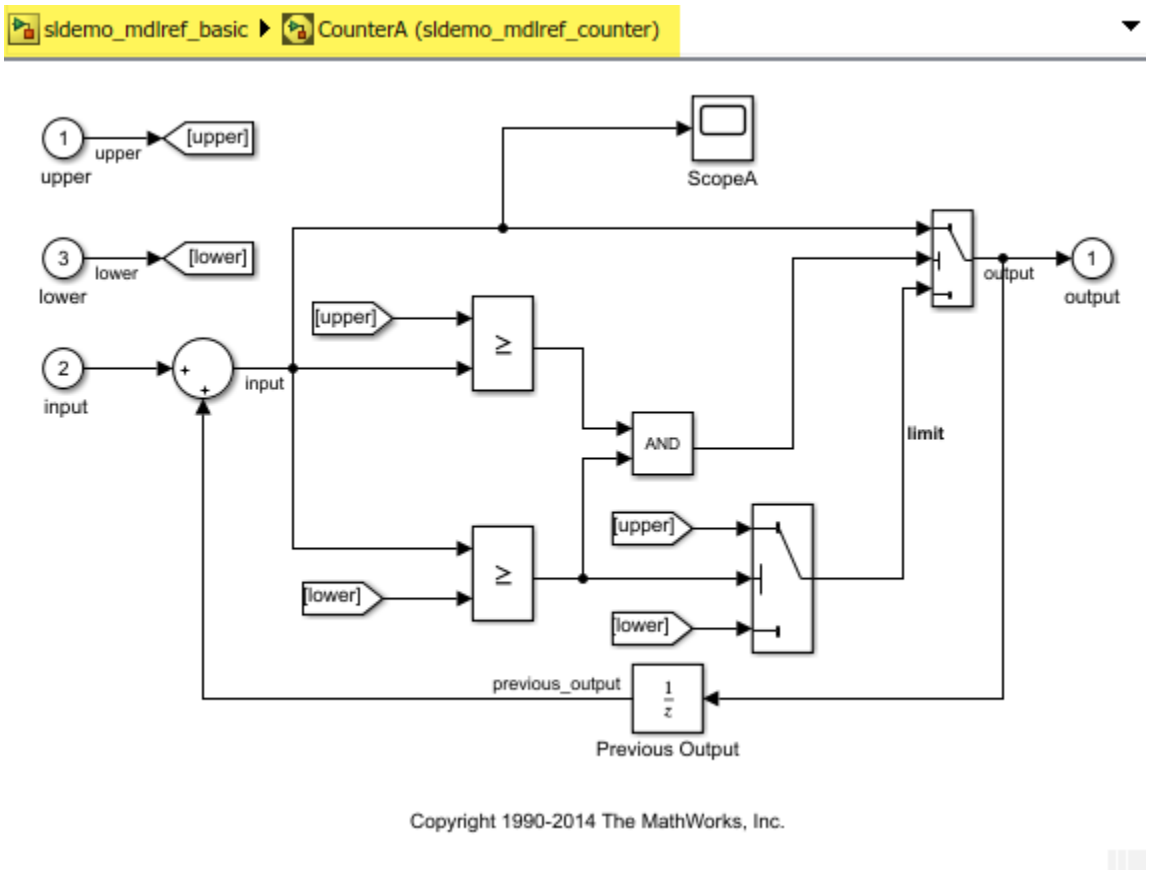
Note Scope windows display simulation results for the most recently opened top model. Playback controls in scope blocks and viewers simulate the model containing that block or viewer.

This example uses the `sldemo_mdlref_counter` model both as a top model and as a referenced model from the `sldemo_mdlref_basic` model.

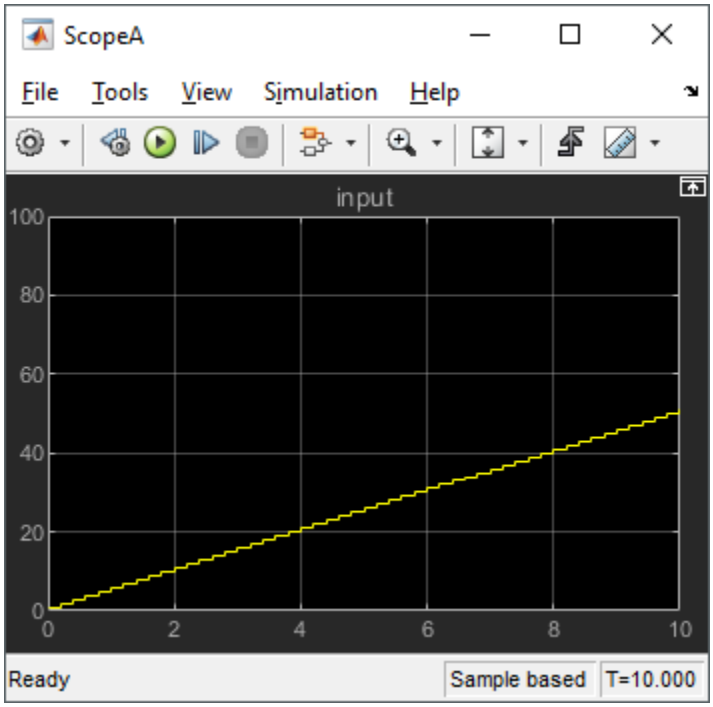
Open the model:

```
openExample('sldemo_mdlref_basic')
```

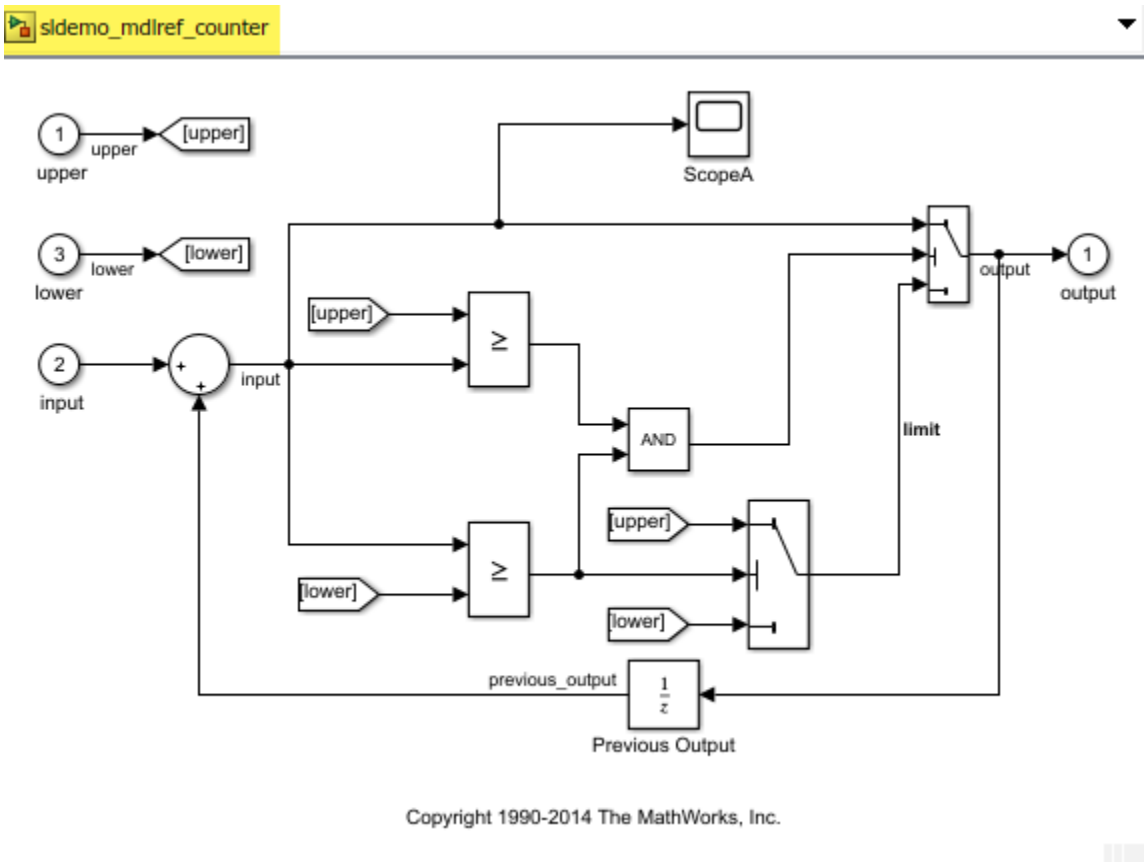
Double-click the CounterA block. The `sldemo_mdlref_counter` model opens as a referenced model, as evidenced by the breadcrumb above the canvas.



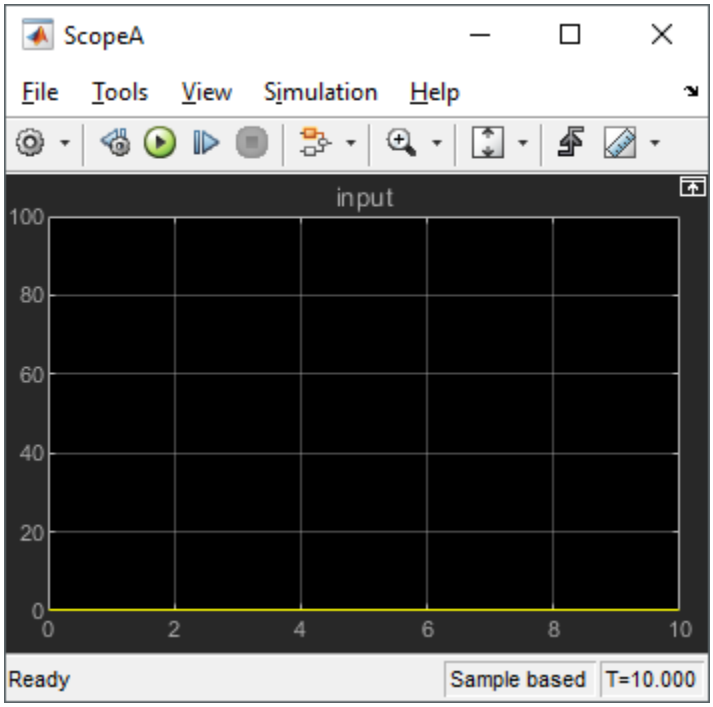
Run the simulation using the main run button, then open up the ScopeA scope. The scope visualizes the data from the entire model.



If you specifically want to visualize a referenced model in isolation, open the model as a top model. In this example, right-click the CounterA block and select `Open as Top Model`. The model opens in another window and the breadcrumb only shows the referenced model name.



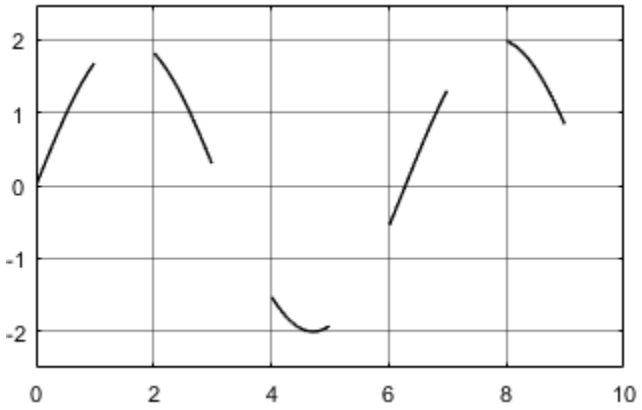
When you run the simulation from either the Simulink window or the scope window, the scope visualizes the model without any reference to another model. In this case, the model input is zero the entire time.



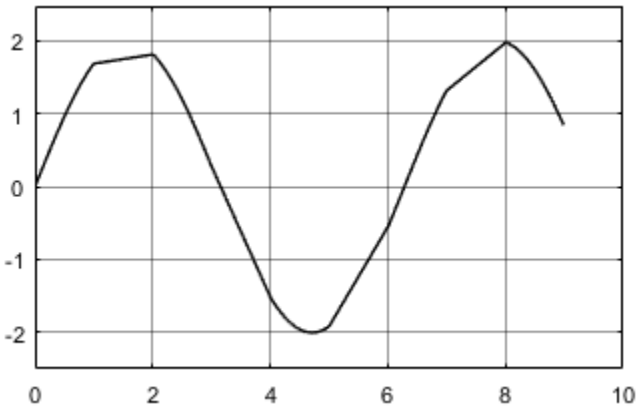
Scopes Within an Enabled Subsystem

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.



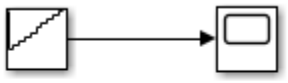
Modify x-axis of Scope

This example shows how to modify the *x*-axis values of the Scope block using the **Time span** and **Time display offset** parameters. The **Time span** parameter modifies how much of the simulation time is shown and offsets the *x*-axis labels. The **Time display offset** parameter modifies the labels used on the *x*-axis.

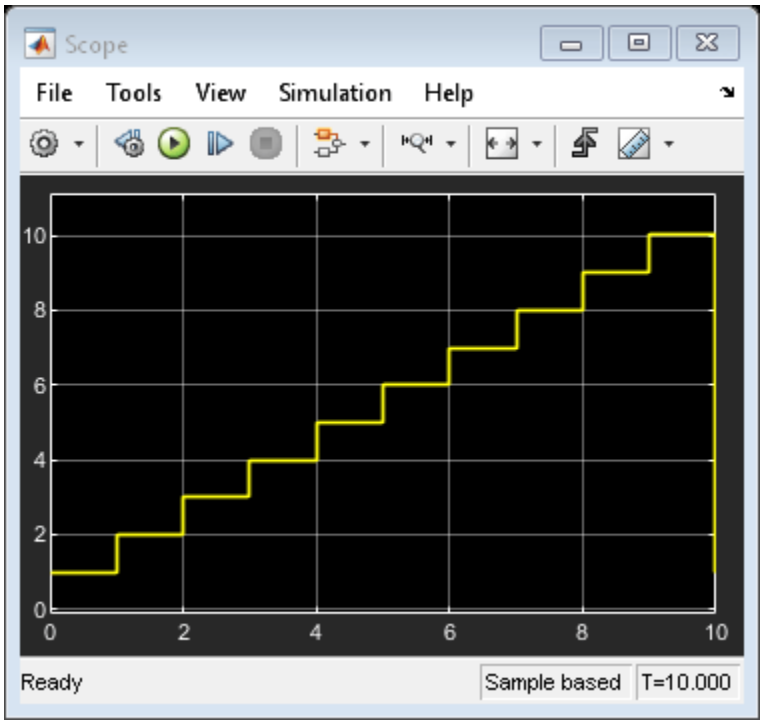
You can also use this procedure for the Time Scope block, Floating Scope block, or Scope viewer.

Open the model and run the simulation to see the original scope output. The simulation runs for 10 time steps stepping up by 1 at each time step.

```
model = 'ModifyScopeXAxis';
open_system(model);
sim(model);
open_system([model,'/Scope']);
```



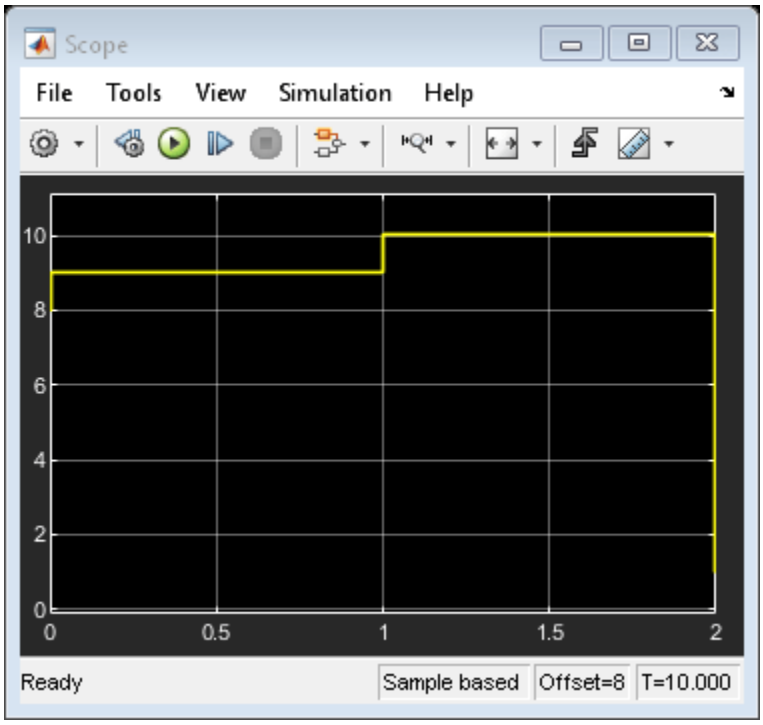
Copyright 2020 The MathWorks, Inc



Modify Time Span Shown

Modify the **Time span** parameter to 2. You can follow the commands below, or in the Scope window, click the Configuration Properties button and navigate to the Time tab.

```
scopeConfig = get_param([model,'/Scope'],'ScopeConfiguration');
scopeConfig.TimeSpan = '2';
sim(model);
open_system([model,'/Scope']);
```



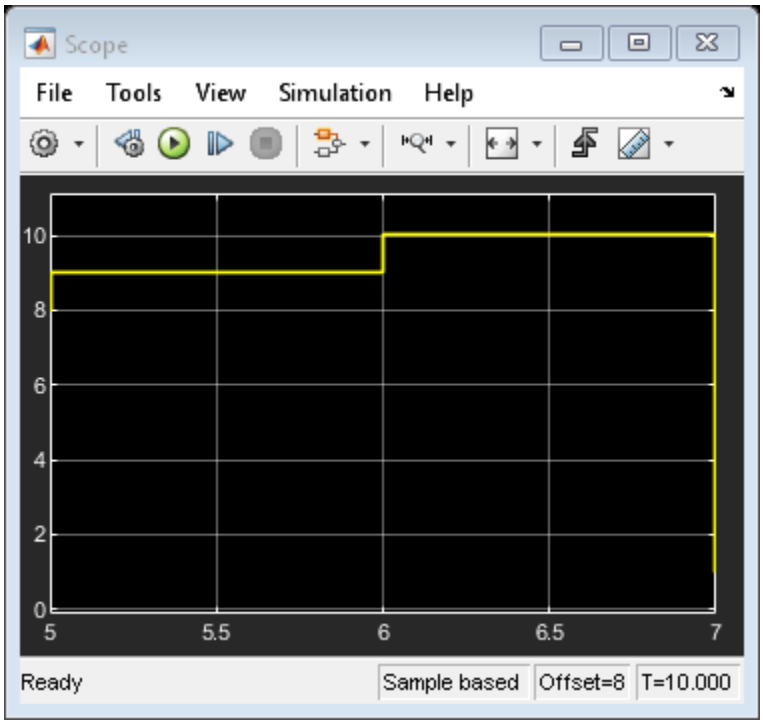
The *x*-axis of the scope now shows only the last 2 time steps and offsets the *x*-axis labels to show 0-2. The bottom toolbar shows that the *x*-axis is offset by 8. This offset is different from the **Time display offset** value.

The **Time span** parameter is useful if you do not want to visualize signal initialization or other start-up tasks at the beginning of a simulation. You can still see the full simulation time span if you click the **Span x-axis** button.

Offset *x*-axis Labels

Modify the **Time display offset** parameter to 5. Again, use the commands below, or in the Scope window, click the Configuration Properties button and navigate to the Time tab.

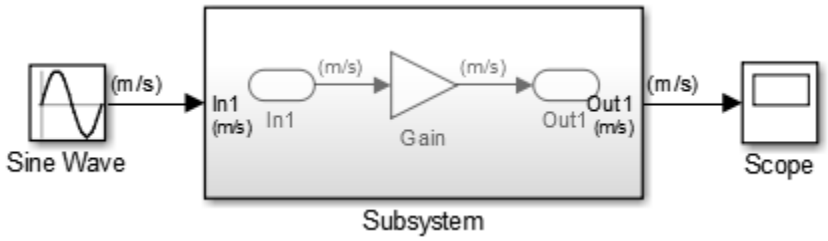
```
scopeConfig.TimeDisplayOffset = '5';
sim(model);
open_system([model,'/Scope']);
```



Now, the same time span of 2 is show in the scope, but the *x*-axis labels are offset by 5, starting at 5 and ending at 7. If you click the **Span x-axis** button, the *x*-axis labels still start at 5.

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`.

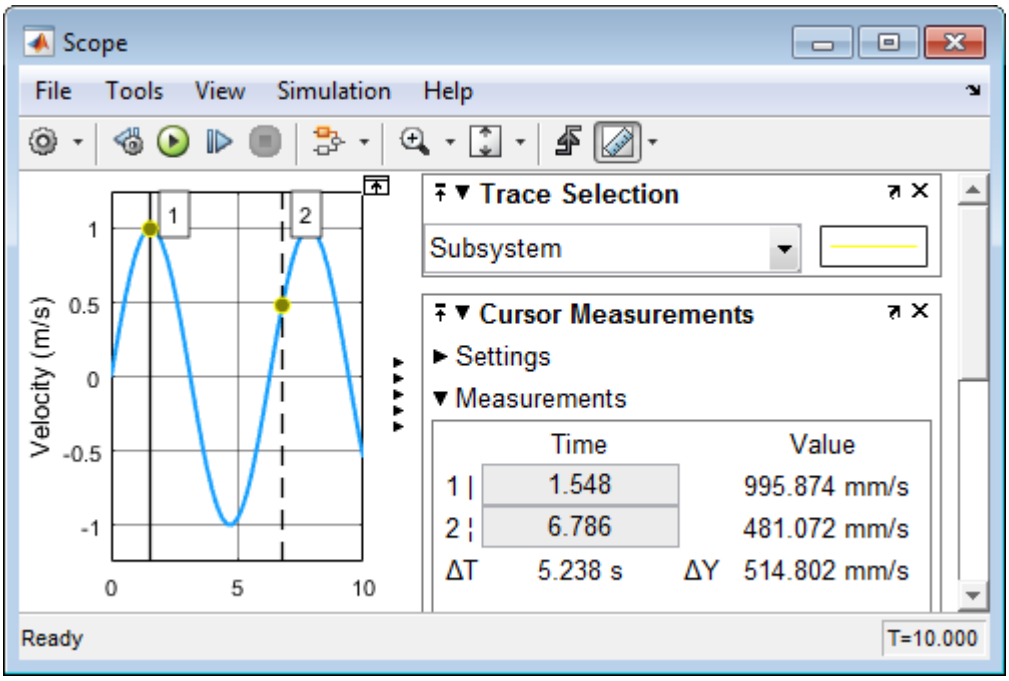


Show Units on a Scope Display

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.

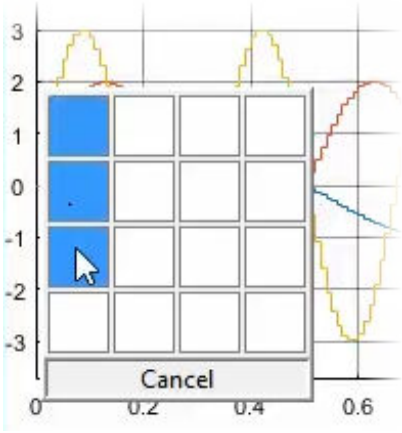
Scopes show units depending on the number of ports and displays:

- **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.

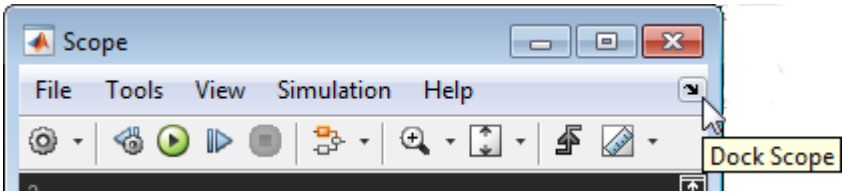
   You can select more than four displays in a row or column. Click within the layout, and then drag your mouse pointer to expand the layout to a maximum of 16 rows by 16 columns.



4 Click to apply the selected layout to the Scope window.

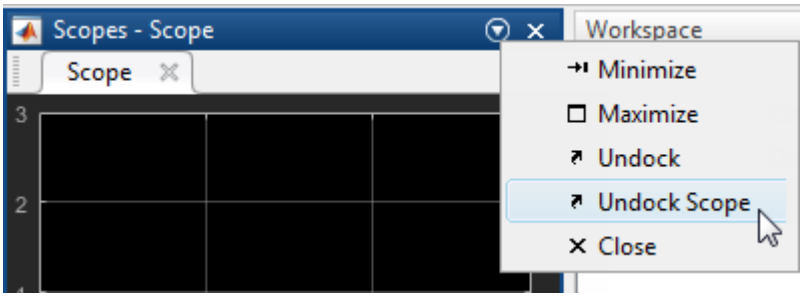
Dock and Undock Scope Window to MATLAB Desktop

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

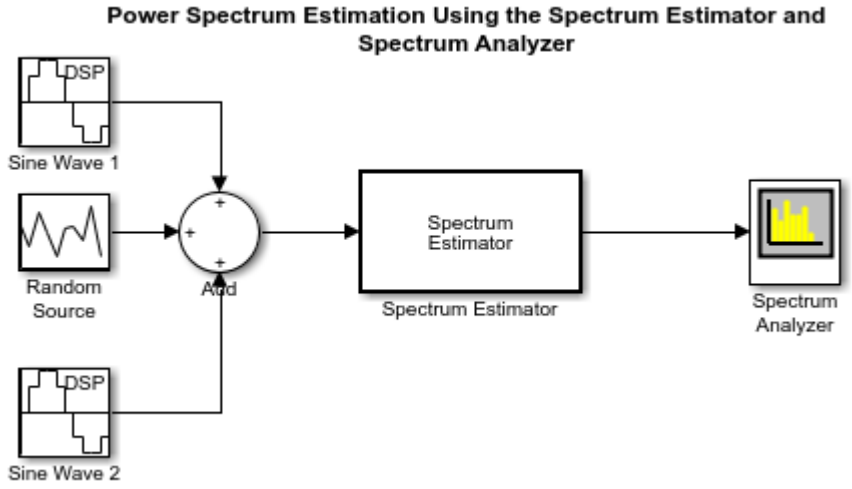
Scope | Floating Scope | Scope Viewer

Related Examples

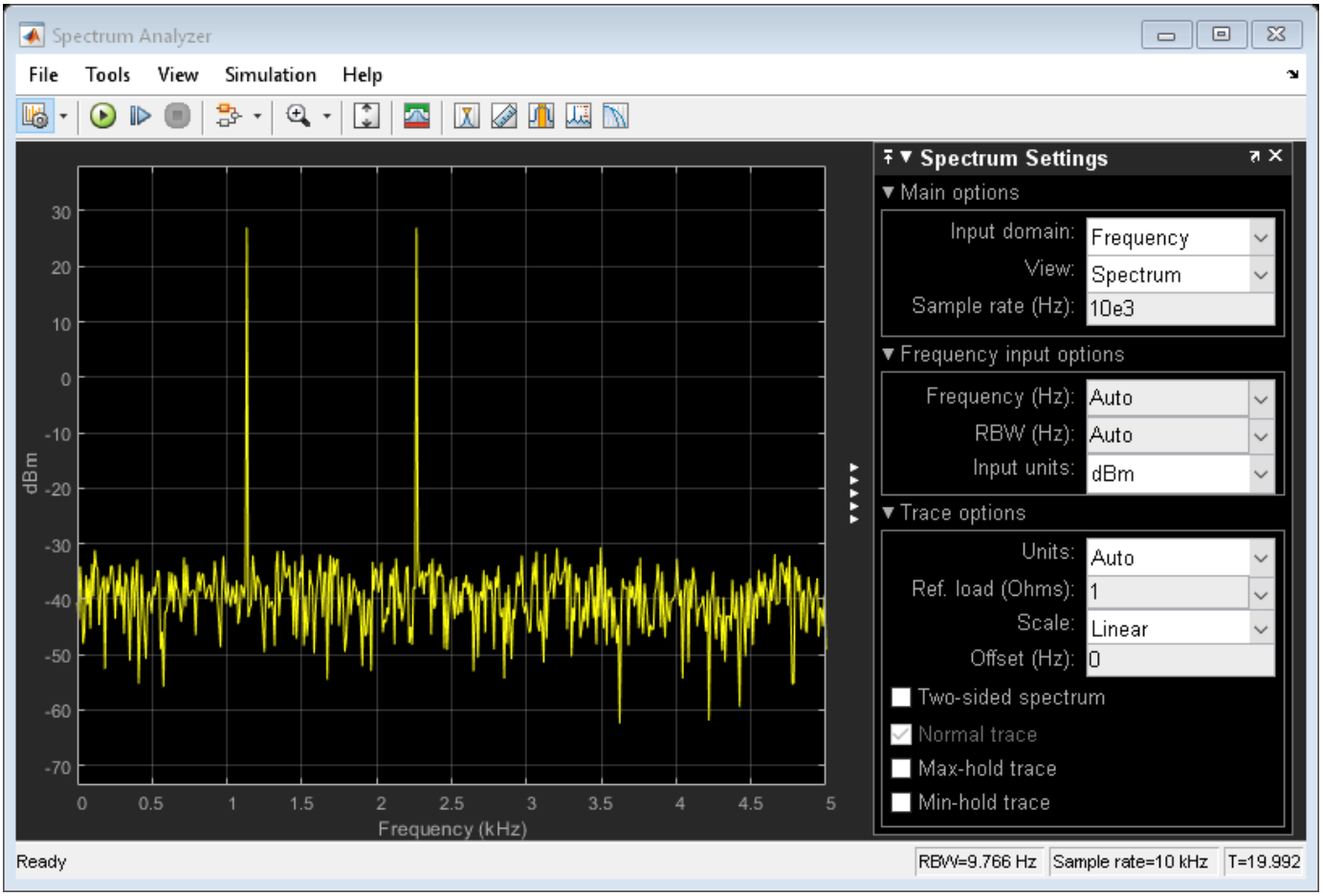
- “Scope Blocks and Scope Viewer Overview” (Simulink)
- “Floating Scope and Scope Viewer Tasks” (Simulink)

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.



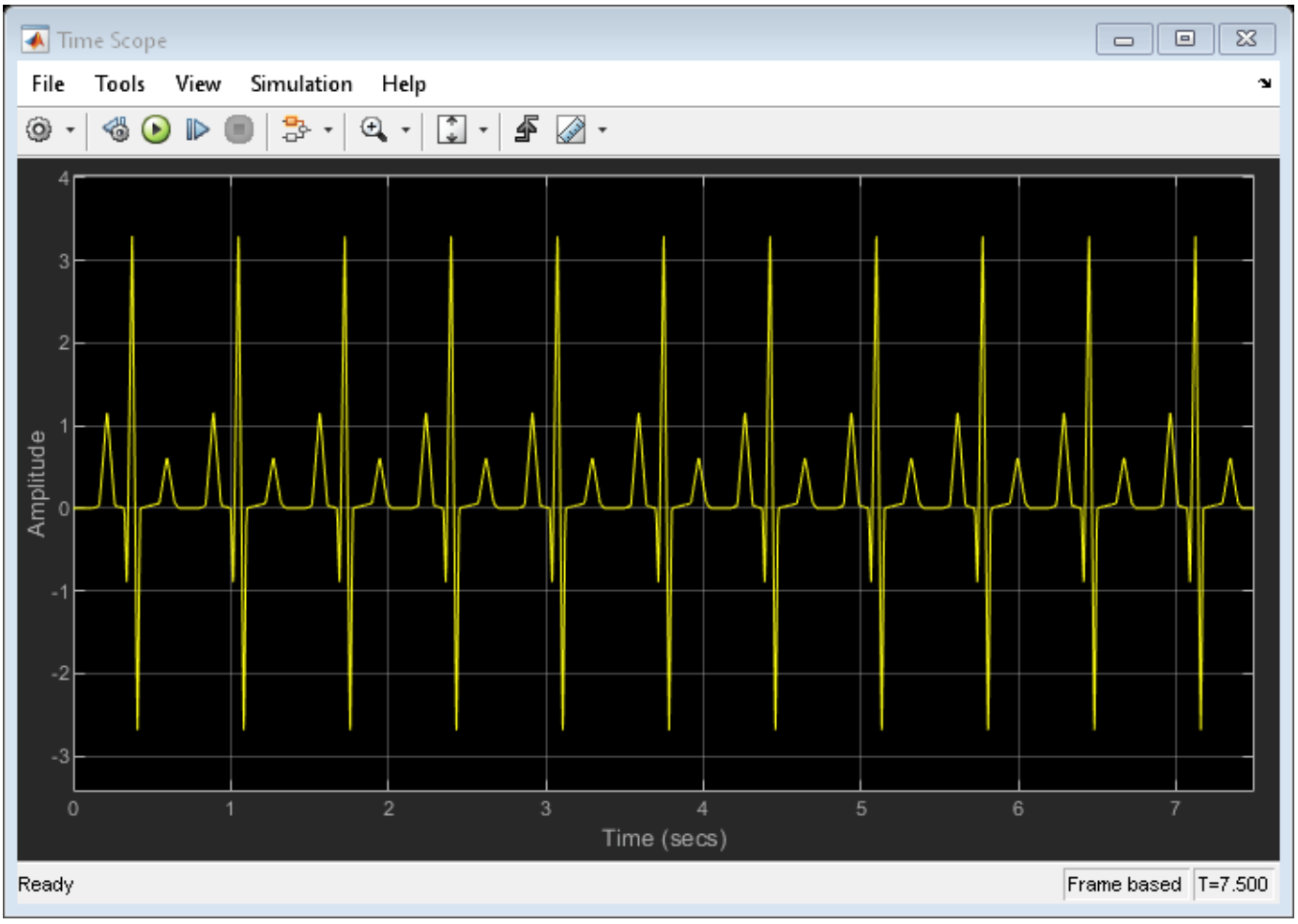
You can see two peaks at 5 kHz and 10 kHz. To measure these peaks, use the Peak Finder measurement tool.

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.

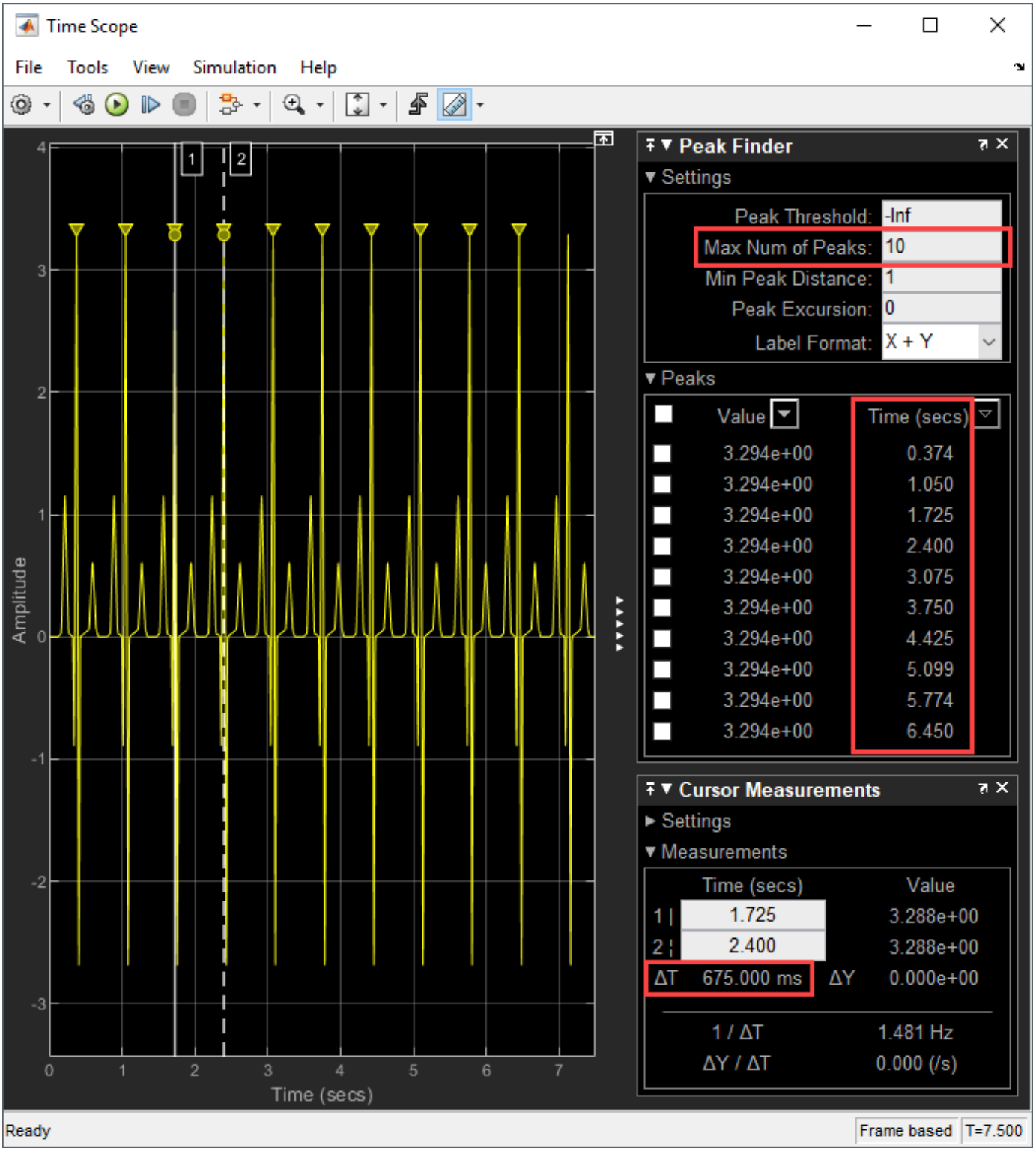
To emulate a heart beat, the model Preload creates the variable `mhb` in the MATLAB® workspace. This variable is then called by the Signal From Workspace block. To see all the model Preload commands, open the **Model Explorer** and look at the Callback functions.

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:

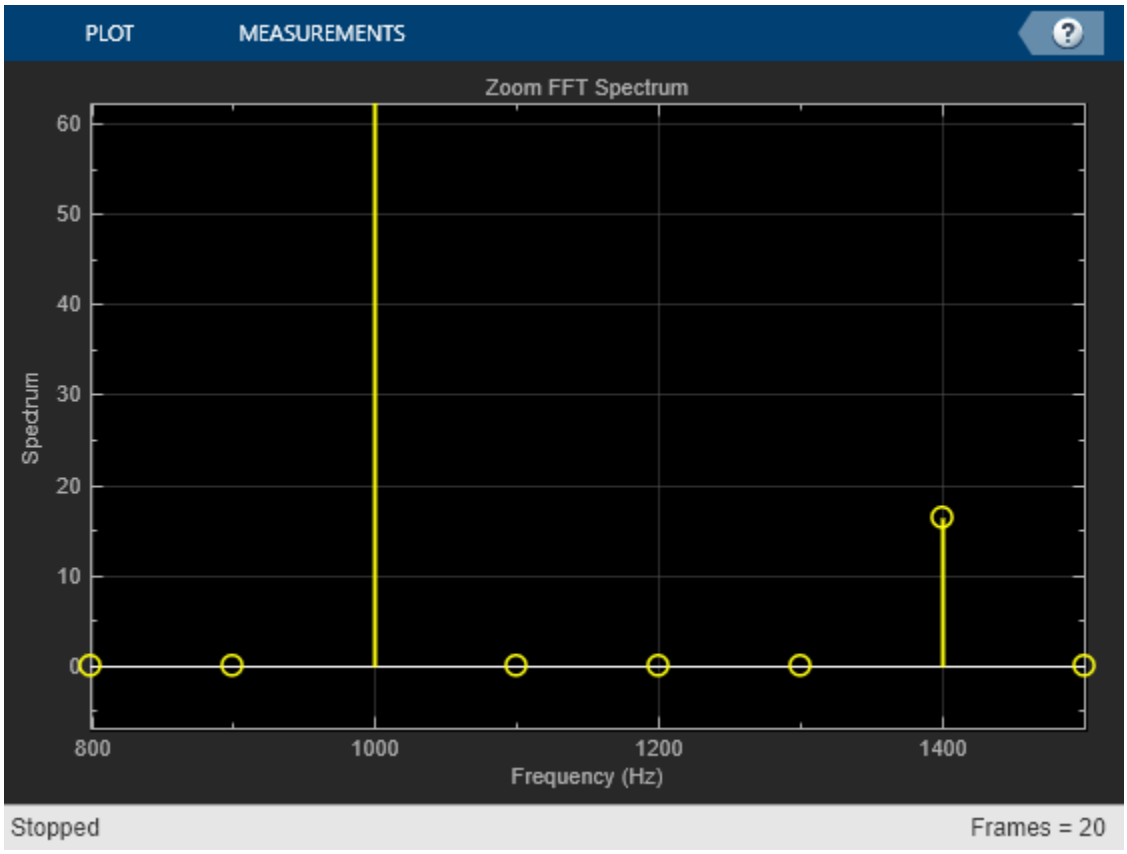
$$\frac{1\ beat}{675\ ms} * \frac{60000\ ms}{1\ min} = 88.88\ bpm$$

Configure Array Plot From the Command-Line

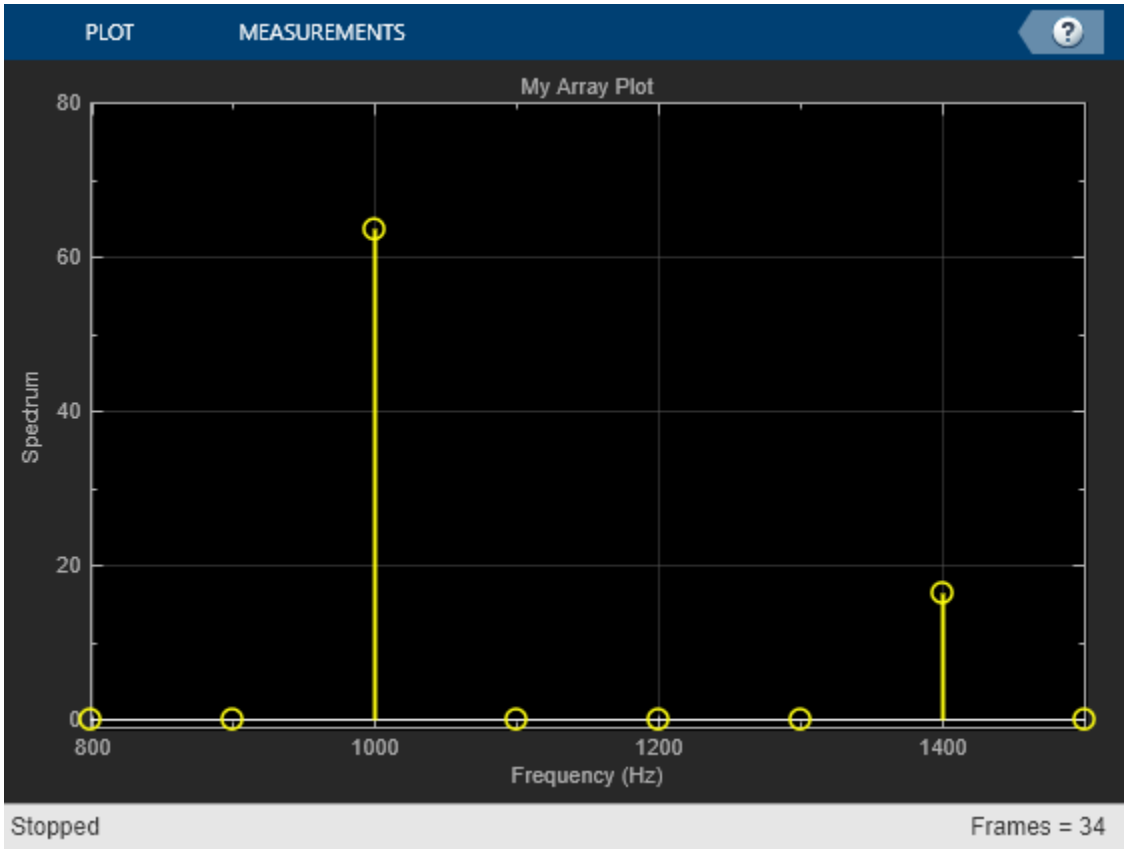
This example shows how to change Array Plot block behavior and appearance from the MATLAB command line.

Load the model. Then, change the title of the Array Plot and the axes scaling.

```
model = 'zoomfftExample';

load_system(model)
sim(model)
open_system([model '/View Spectrum'])
```



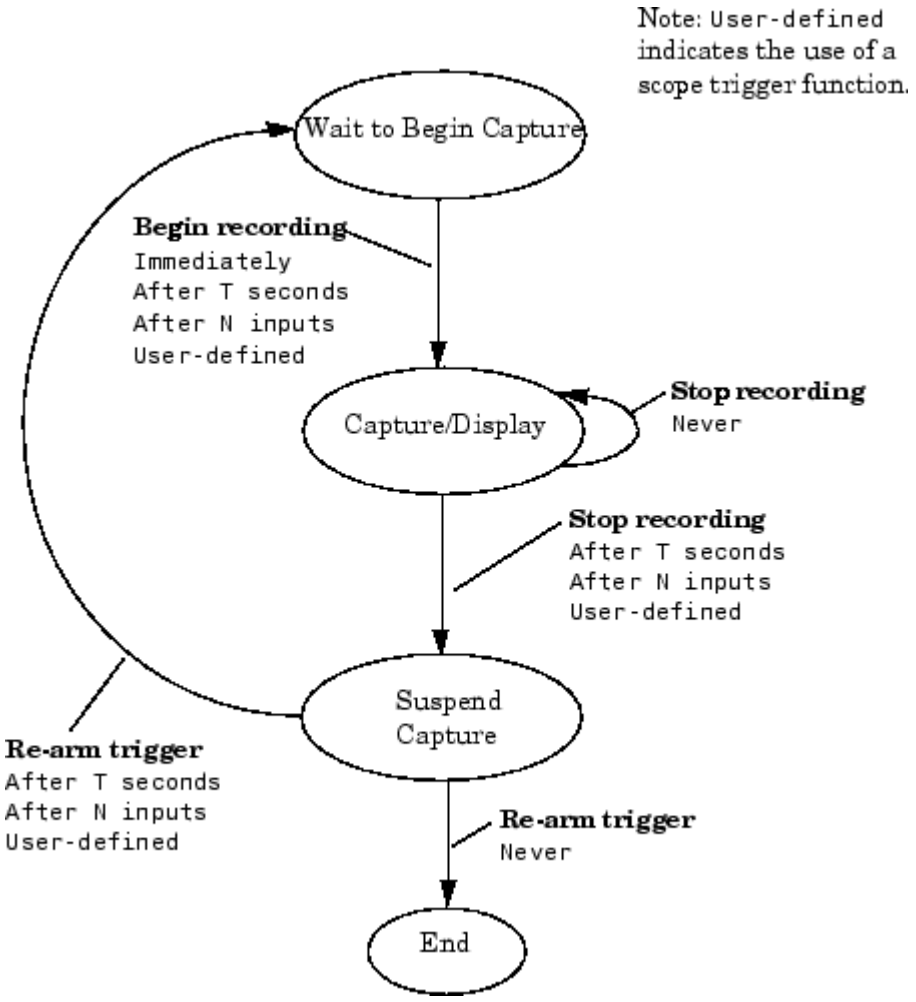

```
sim(model)
set_param([model '/View Spectrum'],'Title','My Array Plot');
set_param([model '/View Spectrum'],'AxesScaling','Manual');
set_param([model '/View Spectrum'],'YLimits','[-1 80]');
open_system([model '/View Spectrum'])
```


PLOT
MEASUREMENTS
My Array Plot
80
60
40
20
0
Spectrum
800
1000
1200
1400
Frequency (Hz)
Stopped
Frames = 34

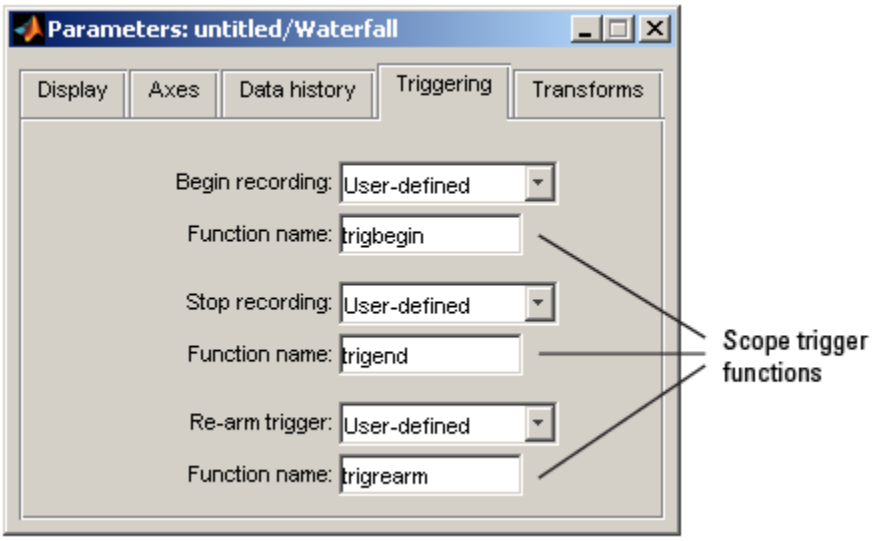
Waterfall Tasks

Scope Trigger Function

The diagram below explains the different states of the Waterfall scope triggering process.



You can create custom scope trigger functions to control when the scope starts, stops, or begins waiting to capture data.



These functions must be valid MATLAB functions and be located either in the current folder or on the MATLAB path.

Each scope trigger function must have the form

```
y = functionname(blk,t,u),
```

where `functionname` refers to the name you give your scope trigger function. The variable `blk` is the Simulink block handle. When the scope trigger function is called by the block, Simulink automatically populates this variable with the handle of the Waterfall block. The variable `t` is the current simulation time, represented by a real, double-precision, scalar value. The variable `u` is the vector input to the block. The output of the scope trigger function, `y`, is interpreted as a logical signal. It is either true or false:

- Begin recording scope trigger function
  - When the output of this scope trigger function is true, the Waterfall block starts capturing data.
  - When the output is false, the block remains in its current state.
- Stop recording scope trigger function
  - When the output of this scope trigger function is true, the block stops capturing data.
  - When the output is false, the block remains in its current state.
- Re-arm trigger scope trigger function
  - When the output of this scope trigger function is true, the block waits for a triggering even to begin recording again.
  - When the output is false, the block remains in its current state.

> **Note** The Waterfall block passes its input data directly to the scope trigger functions. These functions do not use the transformed data defined by the Transform parameters.

The following function is an example of a scope trigger function. This function, called `trigPower`, detects when the energy in the input `u` exceeds a certain threshold.

```
function y = trigPower(blk, t, u)

y = (u'*u > 2300);
```

The following function is another example of a scope trigger function. This function, called `count3`, triggers the scope once three vectors with positive means are input to the block. Then the function resets itself and begins searching for the next three input vectors with positive means. This scope trigger function is valid only when one Waterfall block is present in your model.

```
function y = count3(blk, t, u)

persistent state;
if isempty(state); state = 0; end
if mean(u)>0; state = state+1; end
y = (state>=3);
if y; state = 0; end
```

Scope Transform Function

You can create a scope transform function to control how the Waterfall block transforms your input data. This function must have a valid MATLAB function name and be located either in the current folder or on the MATLAB path.

Your scope transform function must have the form

```
y = functionname(u)
```

where `functionname` refers to the name you give your function. The variable `u` is the real or complex vector input to the block. The output of the scope transform function, `y`, must be a double-precision, real-valued vector. When it is not, the simulation stops and Simulink displays an error. Note that the output vector does not need to be the same size as the input vector.

Exporting Data

You can use the Waterfall block to export data to the MATLAB workspace:

1. Open and run the `dspanc` example.
2. While the simulation is running, click the Export to Workspace button.
3. Type `whos` at the MATLAB command line.

   The variable `ExportData` appears in your MATLAB workspace. `ExportData` is a 40-by-6 matrix. This matrix represents the six data vectors that were present in the Waterfall window at the time you clicked the Export to Workspace button. Each column of this matrix contains 40 filter coefficients. The columns of data were captured at six consecutive instants in time.

   You can control what data is exported using the **Data logging** parameter on the **Data history** tab of the Parameters dialog box. For more information, see “Data History”.

Capturing Data

You can use the Waterfall block to interact with your data while it is being captured:

1. Open and run the `dspanc` example.

2 While the simulation is running, click the **Suspend data capture** button.

The Waterfall block no longer captures or displays the data coming from the Downsample block.

3 To continue capturing data, click the **Resume data capture** button.

4 To freeze the data display while continuing to capture data, click the **Snapshot display** button.

5 To view the Waterfall block that the data is coming from, click the **Go to scope block** button.

In the Simulink model window, the Waterfall block that corresponds to the active Waterfall window flashes. This feature is helpful when you have more than one Waterfall block in a model and you want to clarify which data is being displayed.

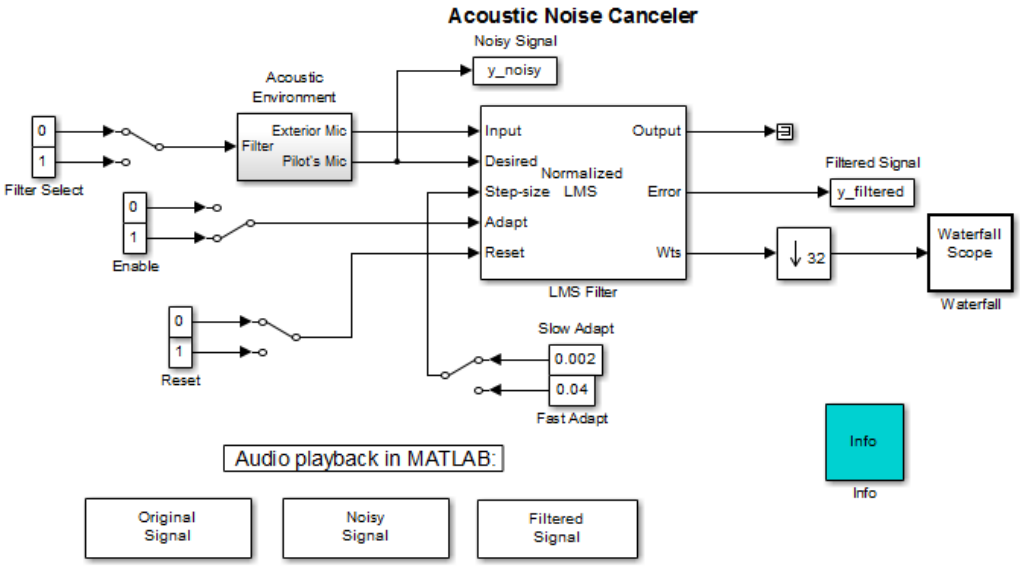
Linking Scopes

You can link several Waterfall blocks together in order to capture the effect of a model event in all of the Waterfall windows in the model:

1 Open the `dspanc` example.

2 Drag a second Waterfall block into the example model.

3 Connect this block to the Output port of the LMS Filter block as shown in this figure:



4 Run the model and view the model behavior in both Waterfall windows.

5 In the dspanc/Waterfall window, click the **Link scopes** button.

6 In the same window, click the **Suspend data capture** button.

The data capture is suspended in both scope windows.

7 Click the **Resume data capture** button.

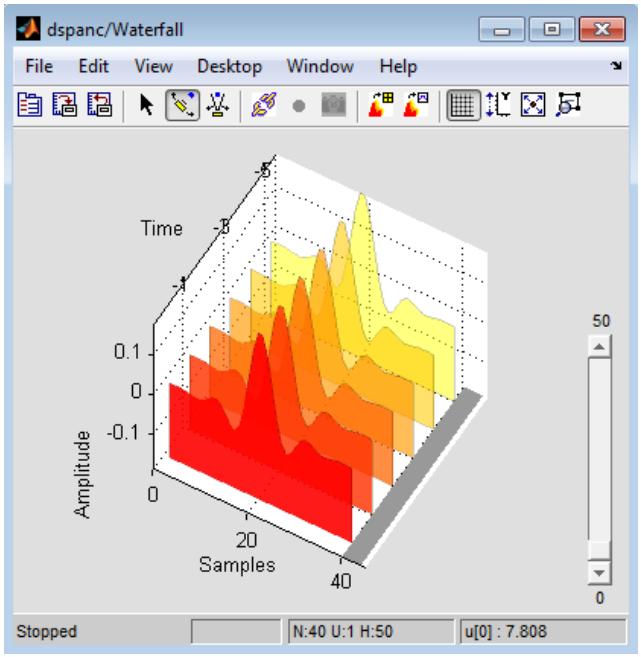
The data capture resumes in both scope windows.

8 In the dspanc/Waterfall window, click the **Snapshot display** button.

In both scope windows, the data display freezes while the block continues to capture data.

9 To continue displaying the captured data, click the **Resume display** button.

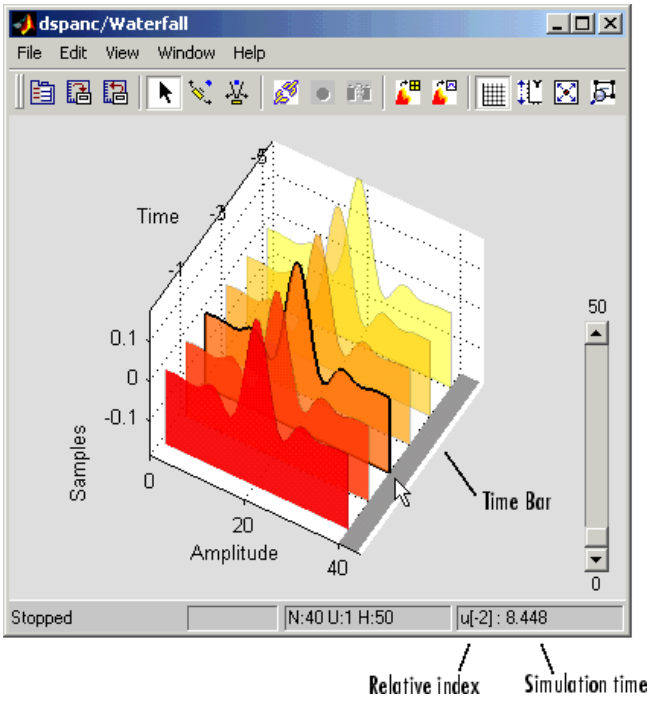
Selecting Data



1 To select a particular set of data, click the **Select** button.

2 Click on the Time Bar at the bottom right of the axes to select a vector of data.

The Waterfall block highlights the selected trace.



While the simulation is running, in the bottom right corner, the Waterfall window displays the relative index of the selected trace. For example, in the previous figure, the selected vector is two sample times away from the most current data vector. When the simulation is stopped, the Waterfall window displays both the relative index and the simulation time associated with the selected trace.

3. To deselect the data vector, click it again.
4. Click-and-drag along the Time Bar.

   Your selection follows the movement of the pointer.

You can use this feature to choose a particular vector to export to the MATLAB workspace. For more information, see “Data History”.

Zooming

You can use the Waterfall window to zoom in on data:

1. Click the **Zoom camera** button.
2. In the Waterfall window, click and hold down the left mouse button.
3. Move the mouse up and down and side-to-side to move closer and farther away from the axes.
4. To resize the axes to fit the Waterfall window, click the **Fit to view** button.

Rotating the Display

You can rotate the data displayed in the Waterfall window:

1. Click on the **Orbit camera** button.
2. In the Waterfall window, click and hold down the left mouse button.
3. Move the mouse in a circular motion to rotate the axes.
4. To return to the position of the original axes, click the **Restore scope position and view** button.

Scaling the Axes

You can use the Waterfall window to rescale the *y*-axis values:

1. Open and run the `dspanc` example.
2. Click the **Rescale amplitude** button.

   The *y*-axis changes so that its minimum value is zero. The maximum value is scaled to fit the data displayed.

Alternatively, you can scale the *y*-axis using the **Y Min** and **Y Max** parameters on the **Axes** tab of the Parameters dialog box. This is helpful when you want to undo the effects of rescaling the amplitude. For more information, see “Axes”.

Saving Scope Settings

The Waterfall block can save the screen position and viewpoint of the Waterfall window:

1. Click the **Save scope position and view** button.
2. Close the Waterfall window.
3. Reopen the Waterfall window.

   It reopens at the same place on your screen. The viewpoint of the axes also remains the same.

See Also

Waterfall

Logic Analyzer

- “Inspect and Measure Transitions Using the Logic Analyzer” on page 27-2
- “Configure Logic Analyzer” on page 27-8

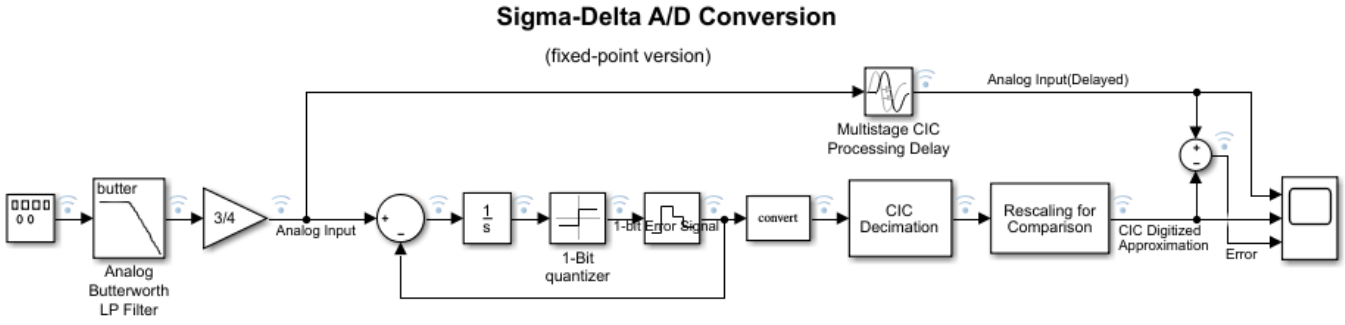
Inspect and Measure Transitions Using the Logic Analyzer

In this section...
“Open a Simulink Model” on page 27-2 “Open the Logic Analyzer” on page 27-2 “Configure Global Settings and Visual Layout” on page 27-3 “Set Stepping Options” on page 27-4 “Run Model” on page 27-5 “Configure Individual Wave Settings” on page 27-5 “Inspect and Measure Transitions” on page 27-5 “Step Through Simulation” on page 27-7 “Save Logic Analyzer Settings” on page 27-7

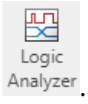
In this tutorial, you explore key functionality of the **Logic Analyzer**, such as choosing and configuring signals to visualize, stepping through a simulation, and measuring transitions.

Open a Simulink Model

To follow along with this tutorial, open the Sigma-Delta A/D Conversion (fixed-point version) model (`dspsdadc_fixpt`).

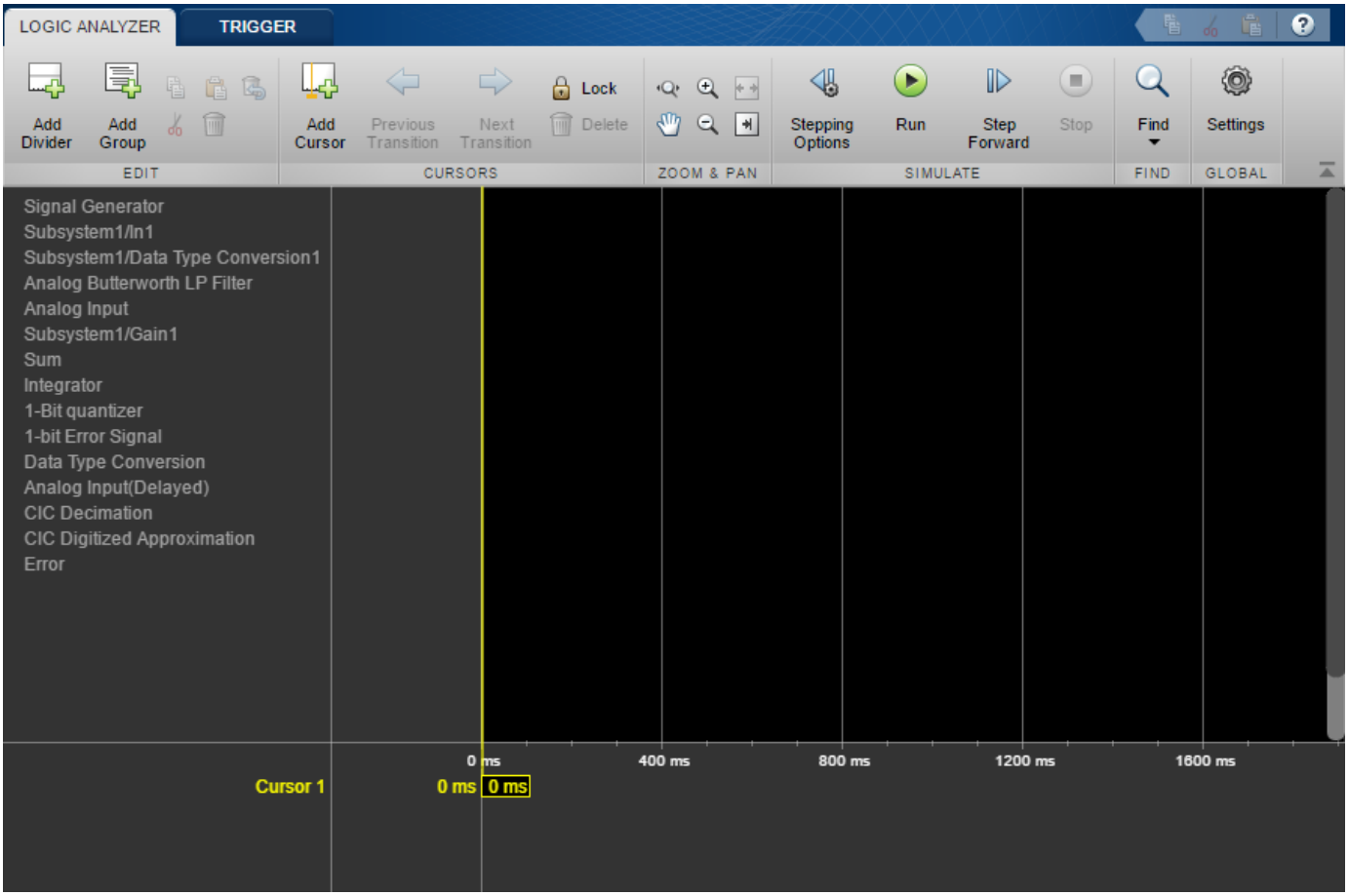


Open the Logic Analyzer

From the Simulink toolstrip, on the **Simulation** tab, in the Review Results gallery, click the Logic Analyzer button 

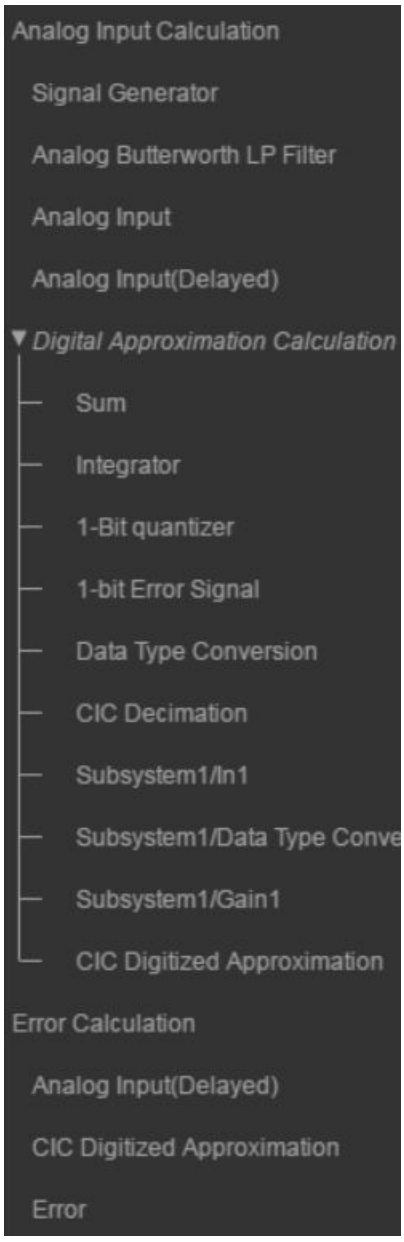
.

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.

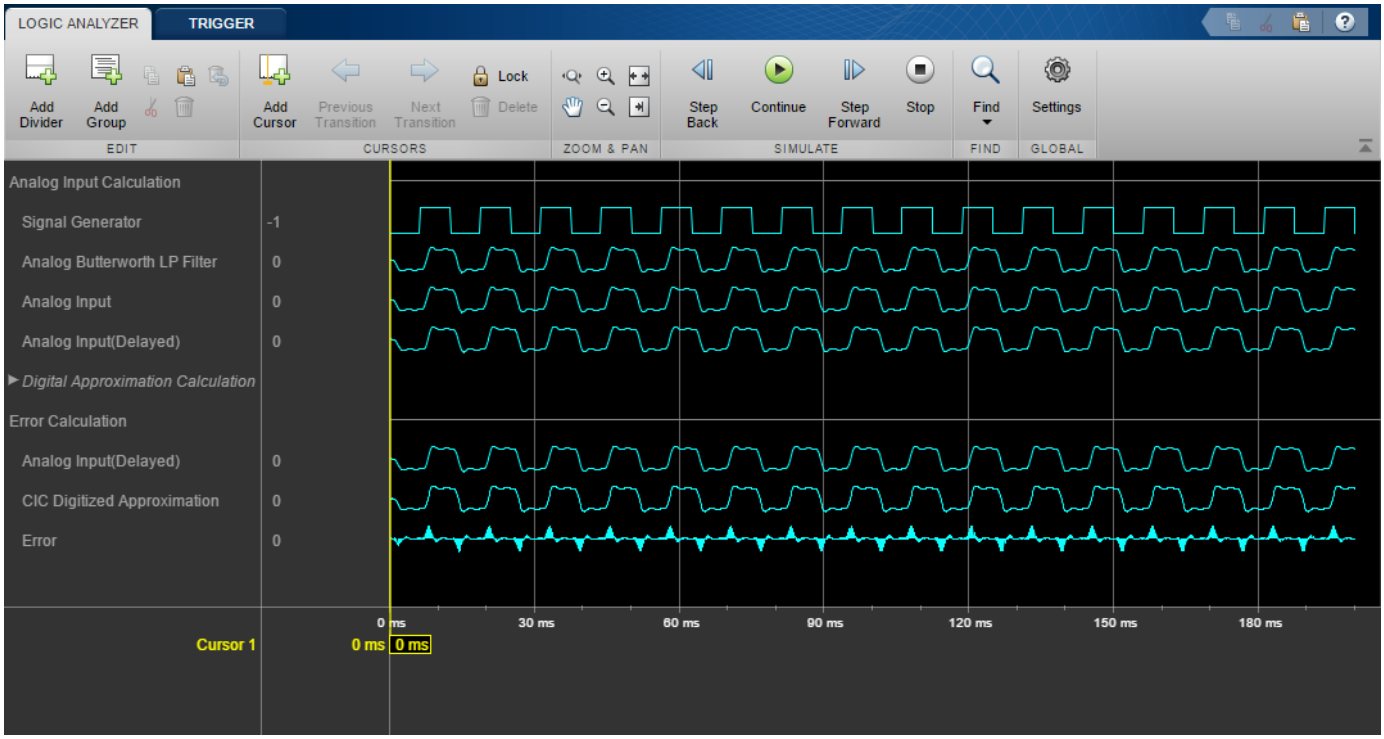


Set Stepping Options

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.

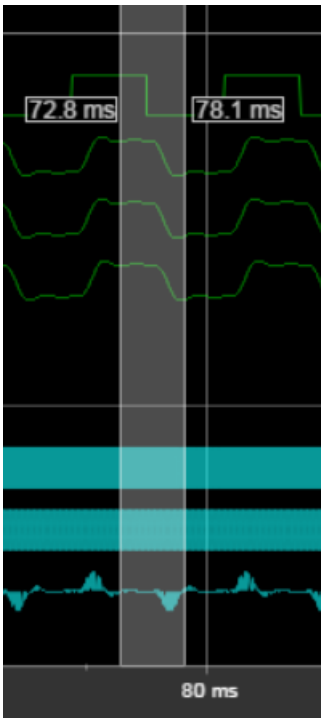


Configure Individual Wave Settings

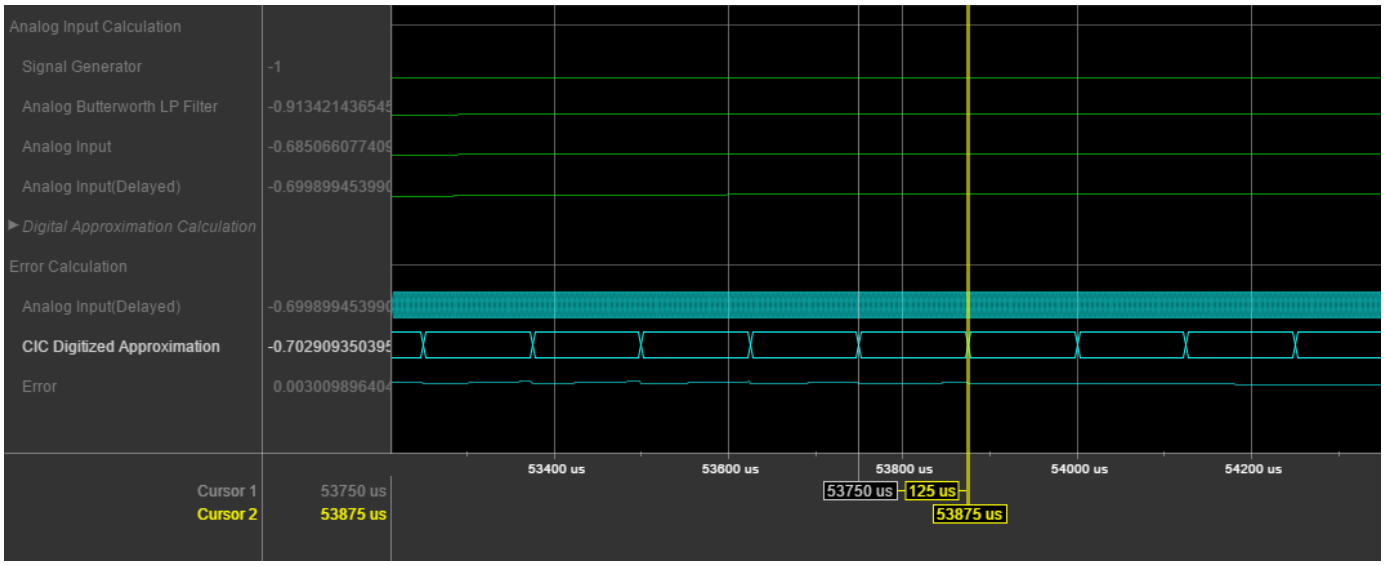
1. Select all waves under your `Analog Input Calculation` divider. Then on the **Waves** tab, select a new **Wave Color** for the selected waves.
2. Under the `Error Calculation` divider, select the `Analog Input(Delayed)` and `CIC Digitized Approximation` waves. On the **Waves** tab, modify the **Format** 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.



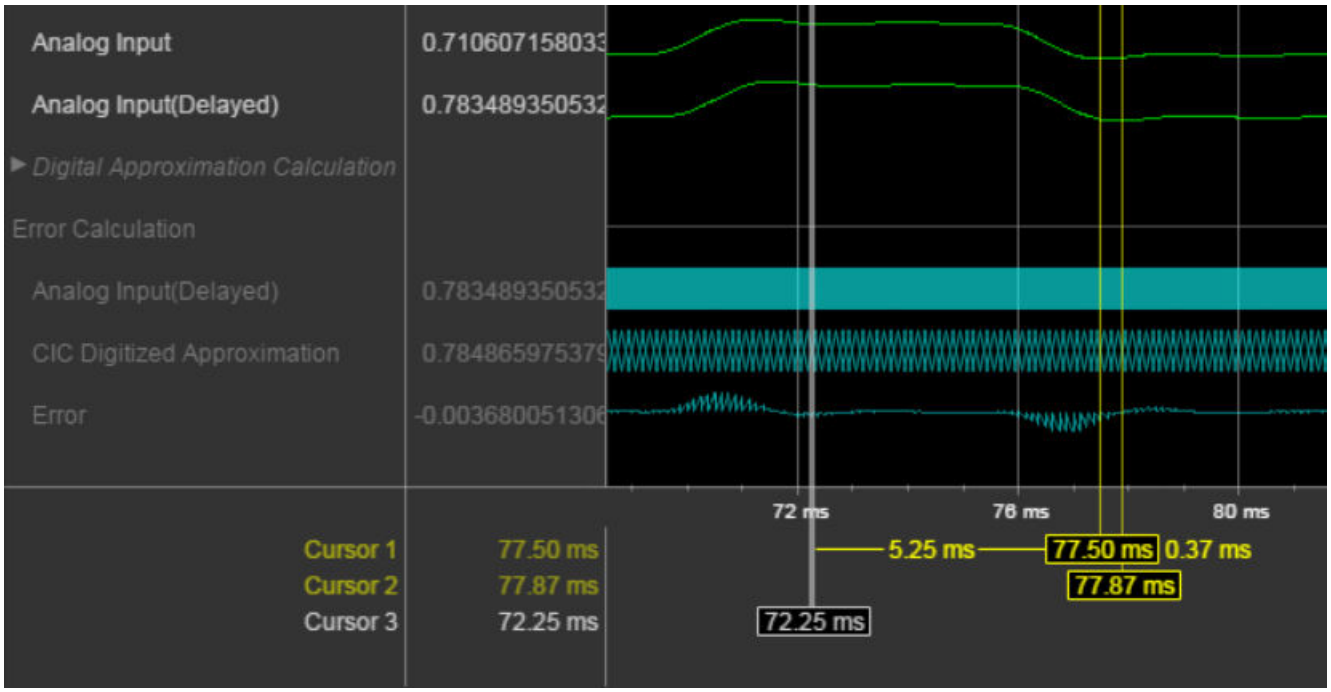
2. For waves displayed as digital, you can use the **Next Transition** and **Previous Transition** buttons. To move the active cursor to the next transition, click **Next Transition**.
3. Click **Lock** to lock the active cursor in place.
4. Click **Add Cursor** to add another cursor to the axes. The 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

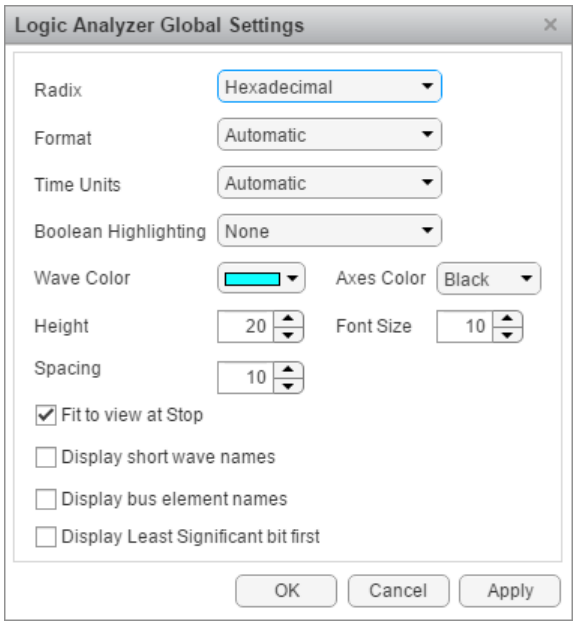
When you save your model, the logic analyzer settings are also saved for that model.

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

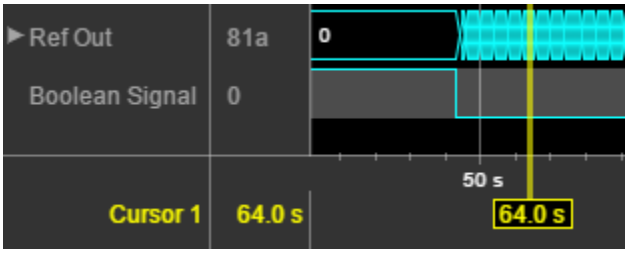
Set the display **Time Units** to one of the following:

- `Automatic` — Uses a time scale appropriate to the time range shown in the current plot
- `seconds`
- `milliseconds`
- `microseconds`
- `nanoseconds`
- `picoseconds`

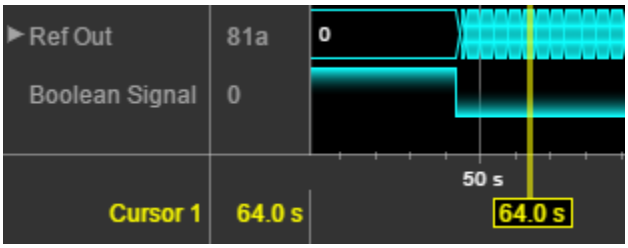
- `femtoseconds`

Set the **Boolean Highlighting** to one of the following:

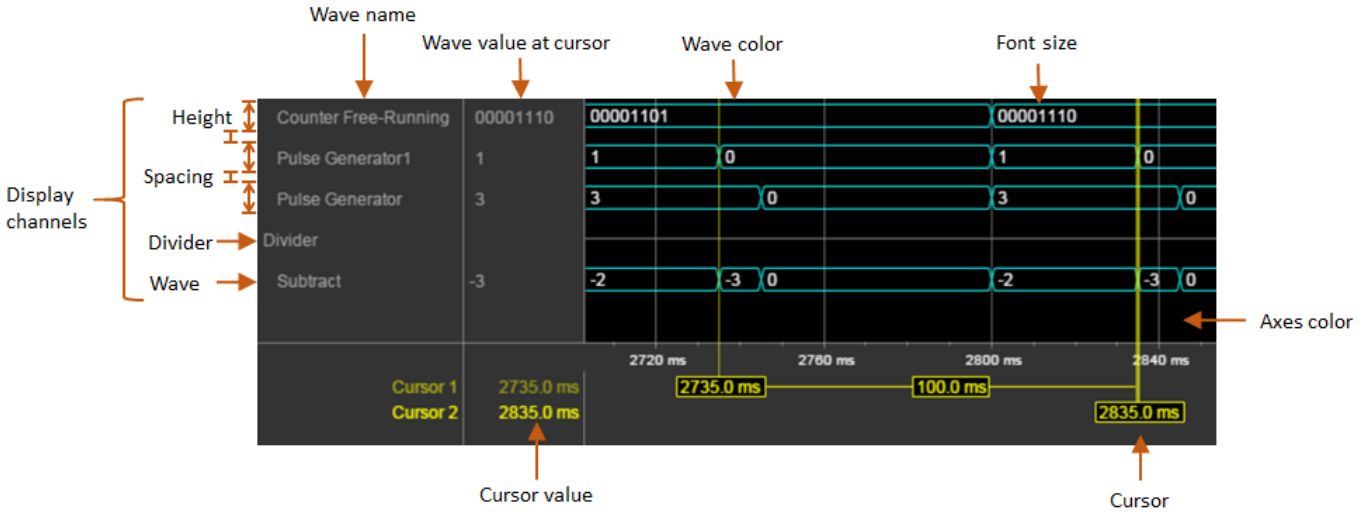
- `None`
- `Rows` — Adds a highlighted background for the entire Boolean signal row.



  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.

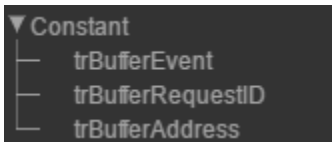
To display the short names of waves without path information, select **Display short wave names**.

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:

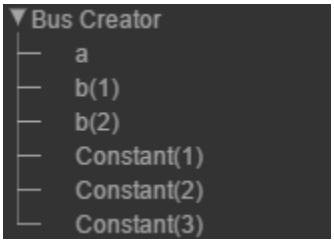
1. Add the bus signal for logging.
2. In the Logic Analyzer settings, select the **Display bus element names** check box.
3. Run the simulation.

When you expand the bus signals, you will see the bus signal names.

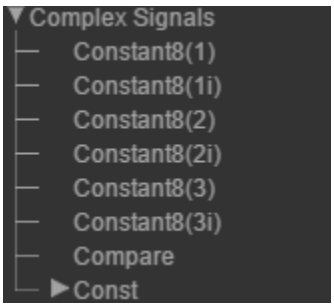


Some special situations:

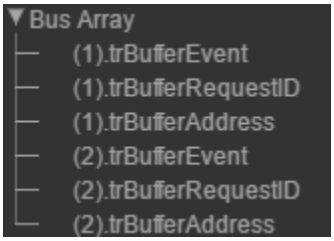
- If the signal has no name, the Logic Analyzer shows the block name instead.
- If the bus is a bus object, the Logic Analyzer shows the bus element names specified in the Bus Object Editor.
- If one of the bus elements contains an array, each element of the array is appended with the element index.



- 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 28-2
- “Sliding Window Method and Exponential Weighting Method” on page 28-5
- “Measure Statistics of Streaming Signals” on page 28-14
- “How Is a Moving Average Filter Different from an FIR Filter?” on page 28-17
- “Energy Detection in the Time Domain” on page 28-21
- “Remove High-Frequency Noise from Gyroscope Data” on page 28-24
- “Linear Algebra and Least Squares” on page 28-26

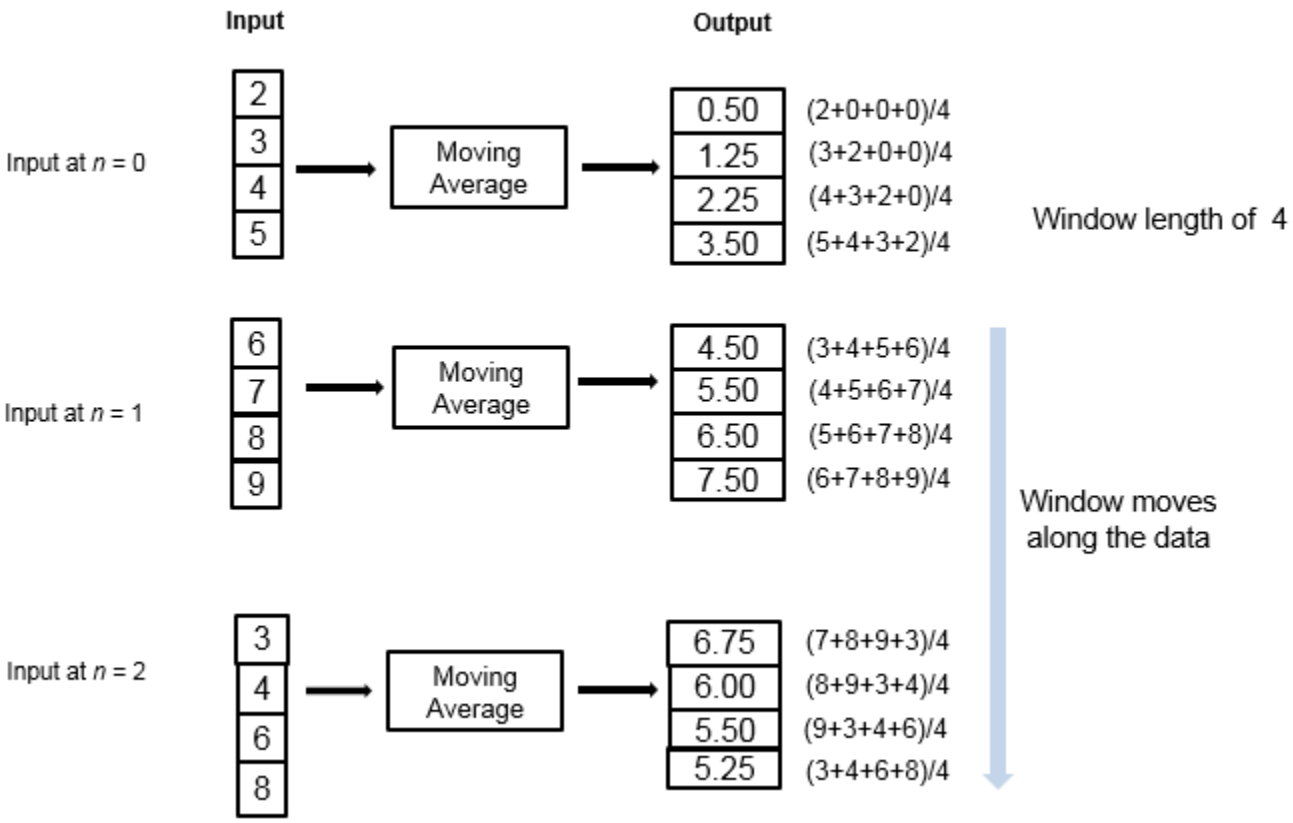
What Are Moving Statistics?

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 28-5.

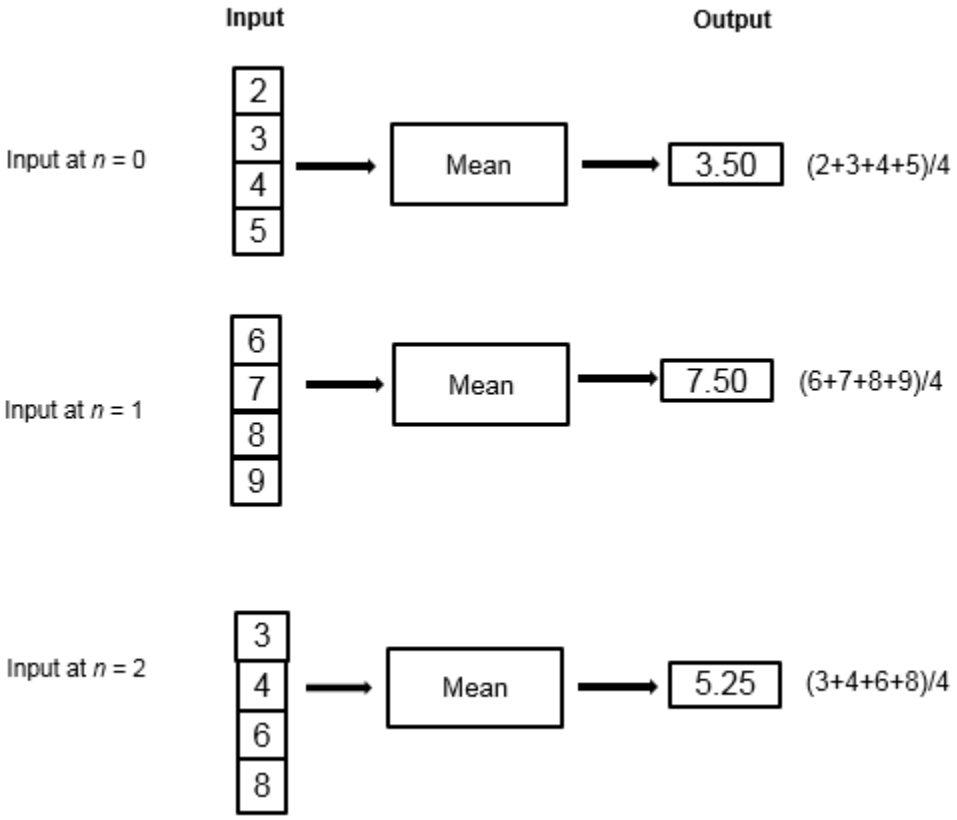
Consider an example of computing the moving average of a streaming input data using the sliding window method. The algorithm uses a window length of 4. At the first time step, the algorithm fills the window with three zeros to represent the first three samples. In the subsequent time steps, to fill the 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 28-14
- “Sliding Window Method and Exponential Weighting Method” on page 28-5
- “How Is a Moving Average Filter Different from an FIR Filter?” on page 28-17
- “Streaming Signal Statistics” on page 4-12
- “Energy Detection in the Time Domain” on page 28-21
- “Remove High-Frequency Noise from Gyroscope Data” on page 28-24

Sliding Window Method and Exponential Weighting Method

In this section...
“Sliding Window Method” on page 28-5 “Exponential Weighting Method” on page 28-7

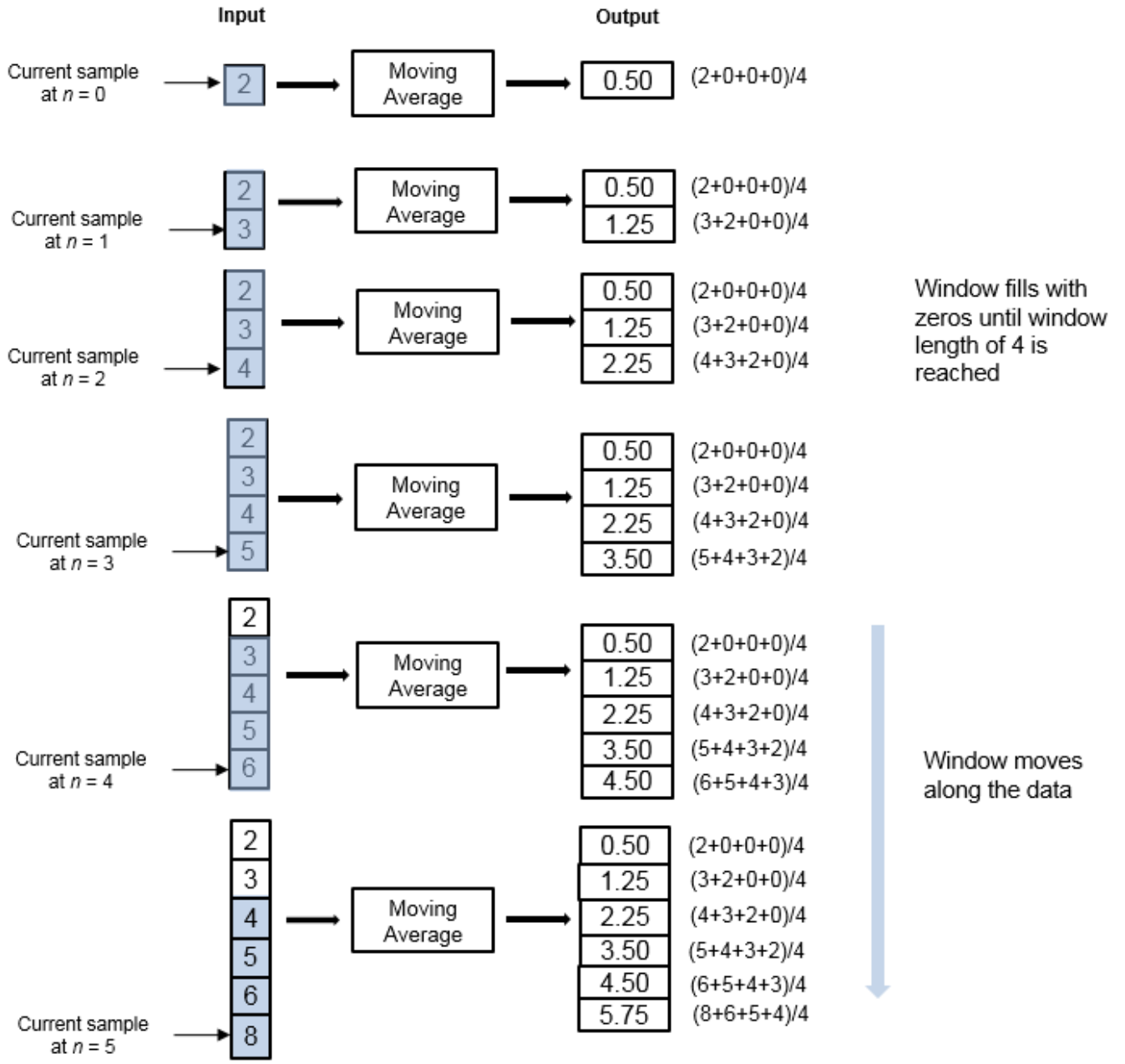
The moving objects and blocks compute the moving statistics of streaming signals using one or both of the sliding window method and exponential weighting method. The sliding window method has a finite impulse response, while the exponential weighting method has an infinite impulse response. To analyze 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

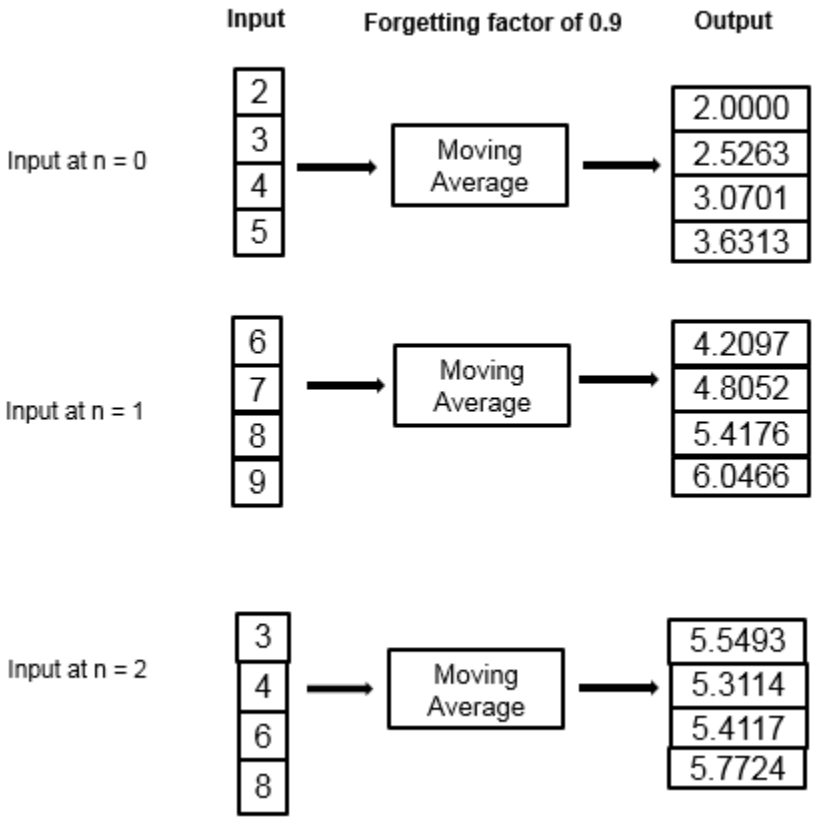
The window length defines the length of the data over which the algorithm computes the statistic. The window moves as the new data comes in. If the window is large, the statistic computed is closer to the stationary statistic of the data. For data that does not change rapidly, use a long window to get 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

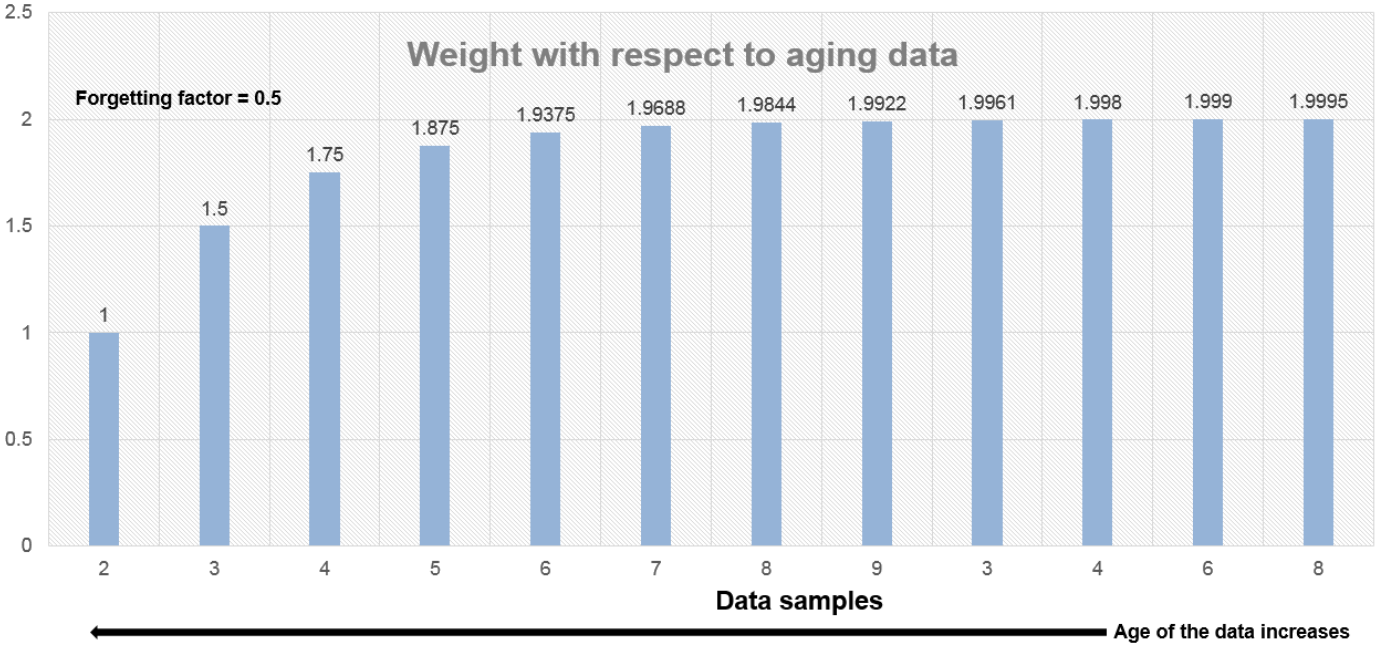
Data	**Weight** $w_{N,\lambda} = \lambda w_{N-1,\lambda} + 1$	**Average** $\bar{x}_{N,\lambda} = \left(1 - \frac{1}{w_{N,\lambda}}\right)\bar{x}_{N-1,\lambda} + \left(\frac{1}{w_{N,\lambda}}\right)x_N$
6	0.9×6.5132+1 = 6.8619	(1–(1/6.8619)) ×5.3114+(1/6.8619)×6 = 5.4117
8	0.9×6.8619+1 = 7.1751	(1–(1/7.1751)) ×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.

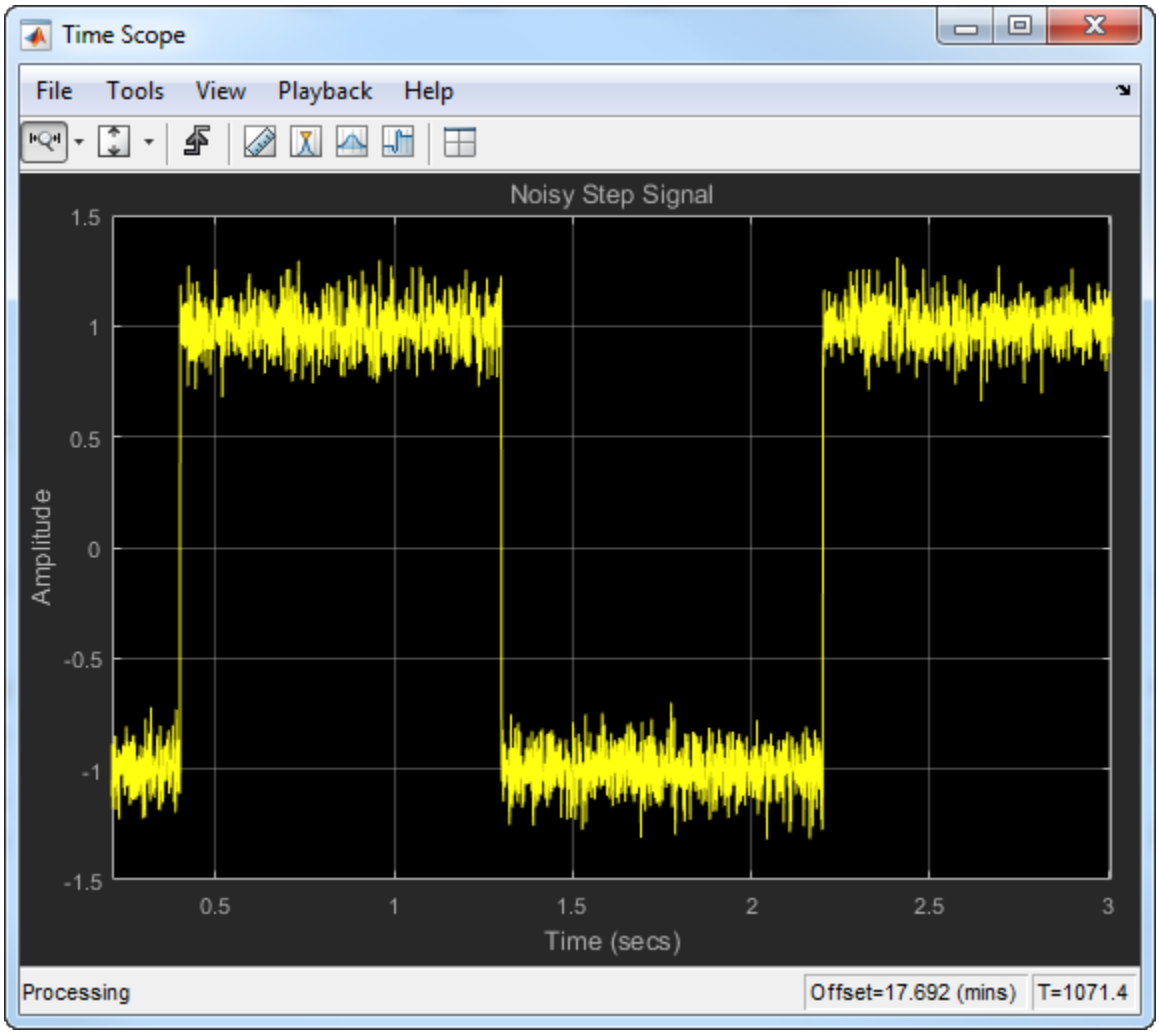
As the age of the data increases, the magnitude of the weighting factor decreases exponentially and never reaches zero. In other words, the recent data has more influence on the current average than the older data.

When the forgetting factor is 0.5, the weights applied to the older data are lower than when the forgetting factor is 0.9.

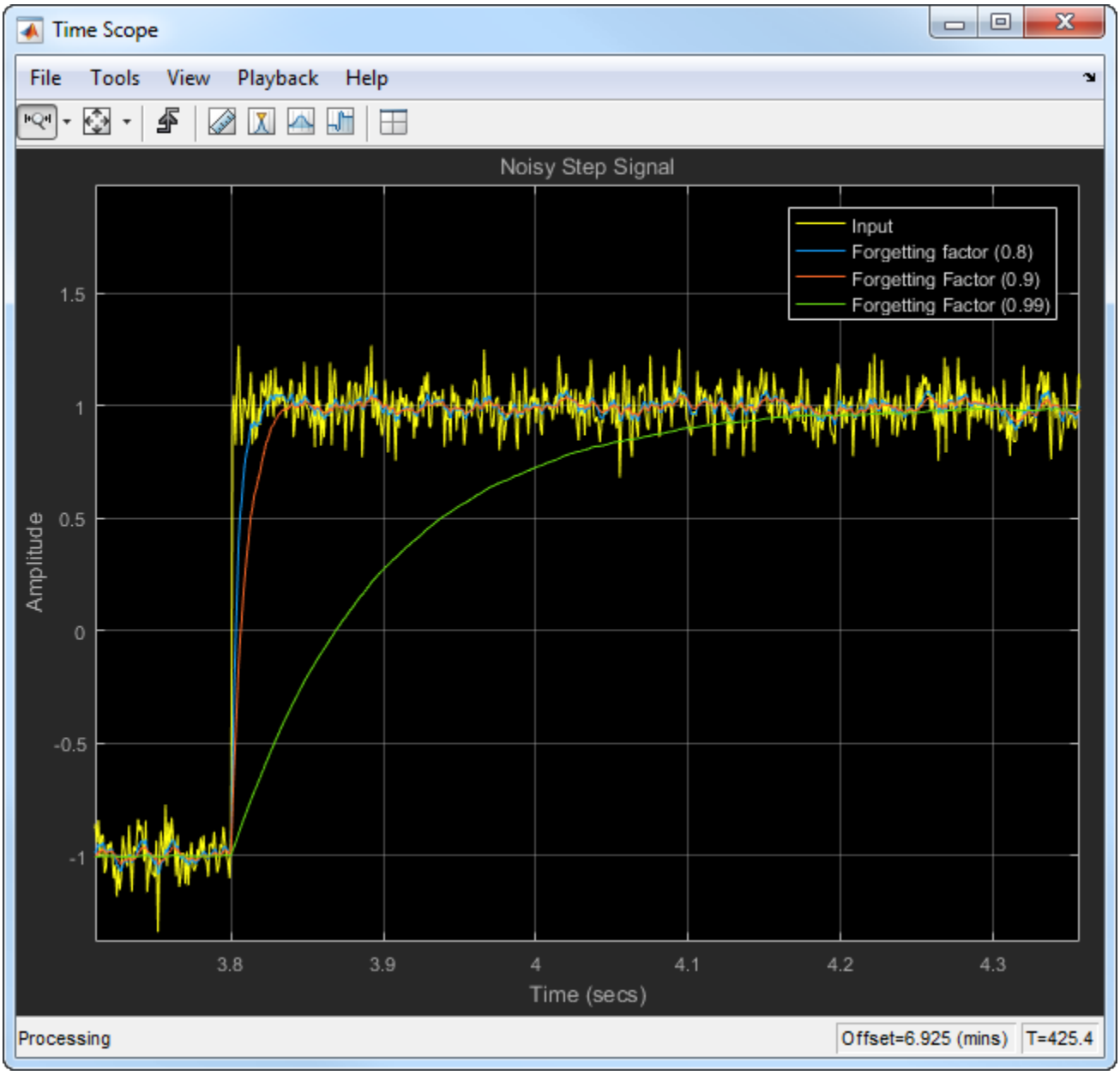


When the forgetting factor is 1, all the data samples are weighed equally. In this case, the exponentially weighted method is the same as the sliding window method with an infinite window length.

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 28-2
- “Measure Statistics of Streaming Signals” on page 28-14
- “How Is a Moving Average Filter Different from an FIR Filter?” on page 28-17
- “Streaming Signal Statistics” on page 4-12
- “Energy Detection in the Time Domain” on page 28-21
- “Remove High-Frequency Noise from Gyroscope Data” on page 28-24

Measure Statistics of Streaming Signals

In this section...
“Compute Moving Average Using Only MATLAB Functions” on page 28-14
“Compute Moving Average Using System Objects” on page 28-15

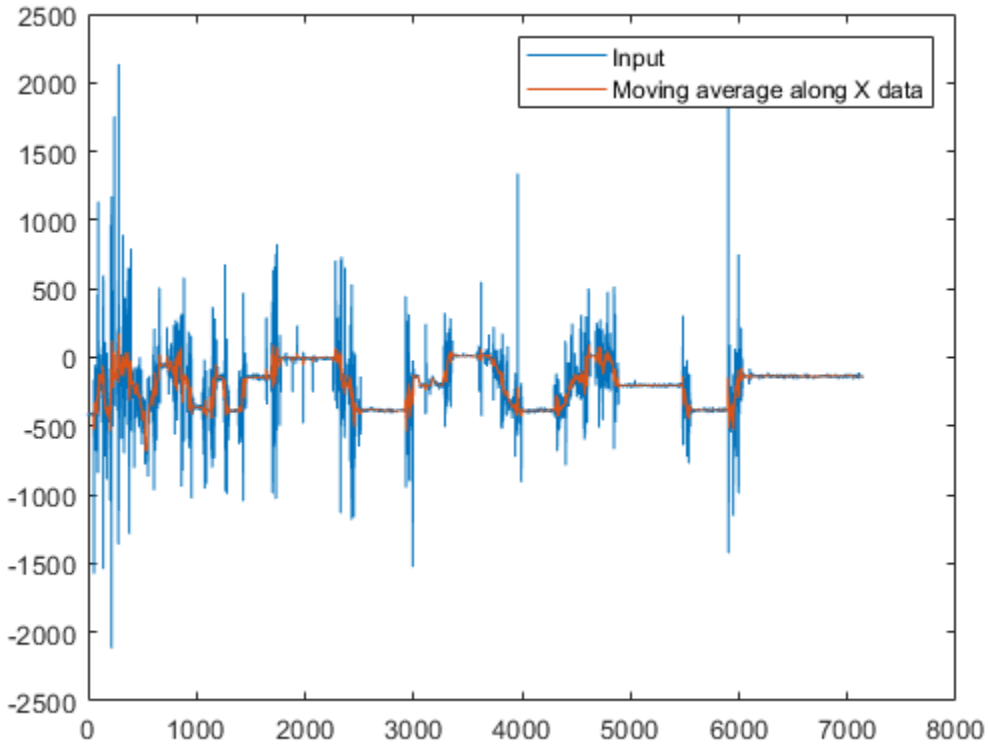
The moving statistics System objects measure statistics of streaming signals in MATLAB. You can also use functions such as `movmean`, `movmedian`, `movstd`, and `movvar` to measure the moving statistics. These functions are more suitable for one-time computations on data that is available in a batch. 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');
```



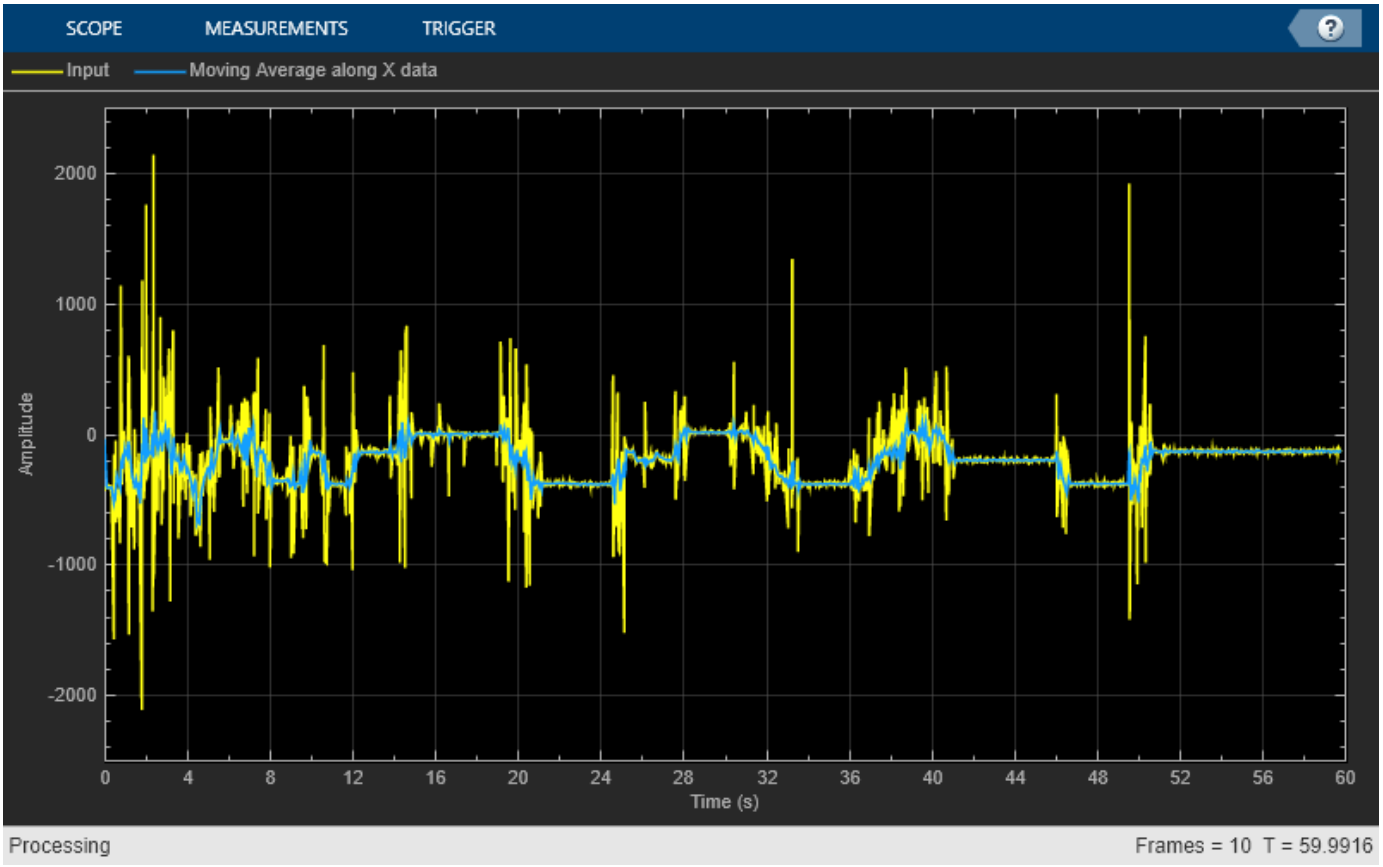
The data is not very large (7140 samples in each column) and is entirely available for processing. The `movmean` function is designed to handle such one-time computations. However, if the data is very large, such as in the order of GB, or if the data is a live stream that needs to be processed in real 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 = timescope('NumInputPorts',2,'SampleRate',119,...
    'YLimits',[-2500 2500],...
    'TimeSpanSource','property','TimeSpan',60,...
    'ChannelNames',{'Input','Moving Average along X data'},...
    '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 28-2
- “Sliding Window Method and Exponential Weighting Method” on page 28-5
- “How Is a Moving Average Filter Different from an FIR Filter?” on page 28-17
- “Streaming Signal Statistics” on page 4-12
- “Energy Detection in the Time Domain” on page 28-21
- “Remove High-Frequency Noise from Gyroscope Data” on page 28-24

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

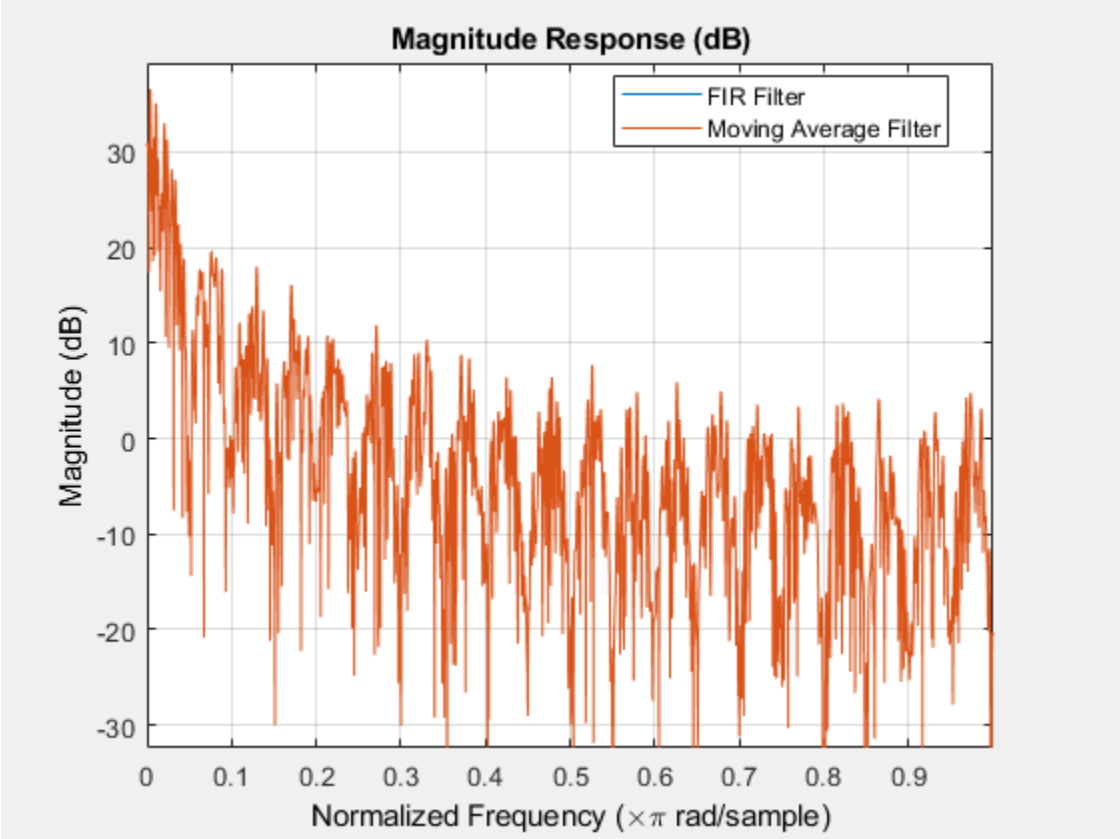
Compare the frequency response of the moving average filter with that of the regular FIR filter. Set the coefficients of the regular FIR filter as a sequence of scaled 1's. The scaling factor is 1/| filterLength|.

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.

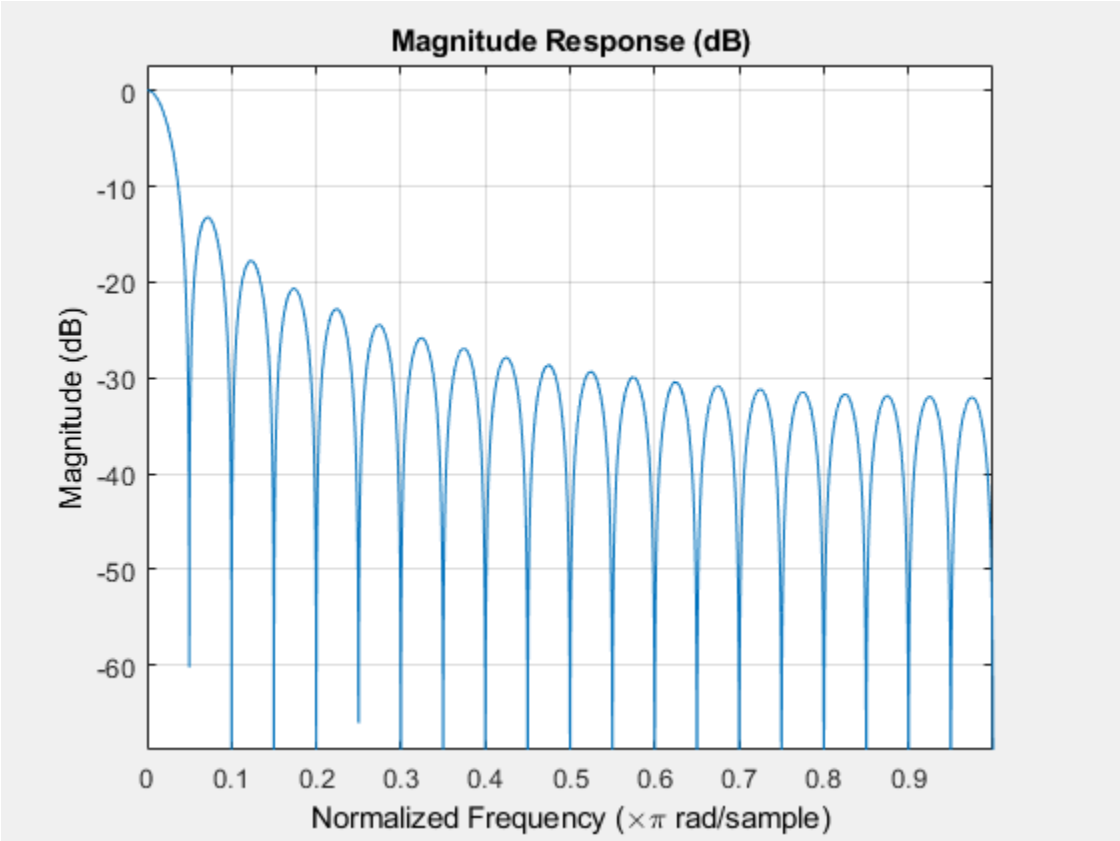
```
hfvt = fvtool(filterOutput,1,mvgAvgOutput,1);
legend(hfvt,'FIR Filter','Moving Average Filter');
```



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);
```

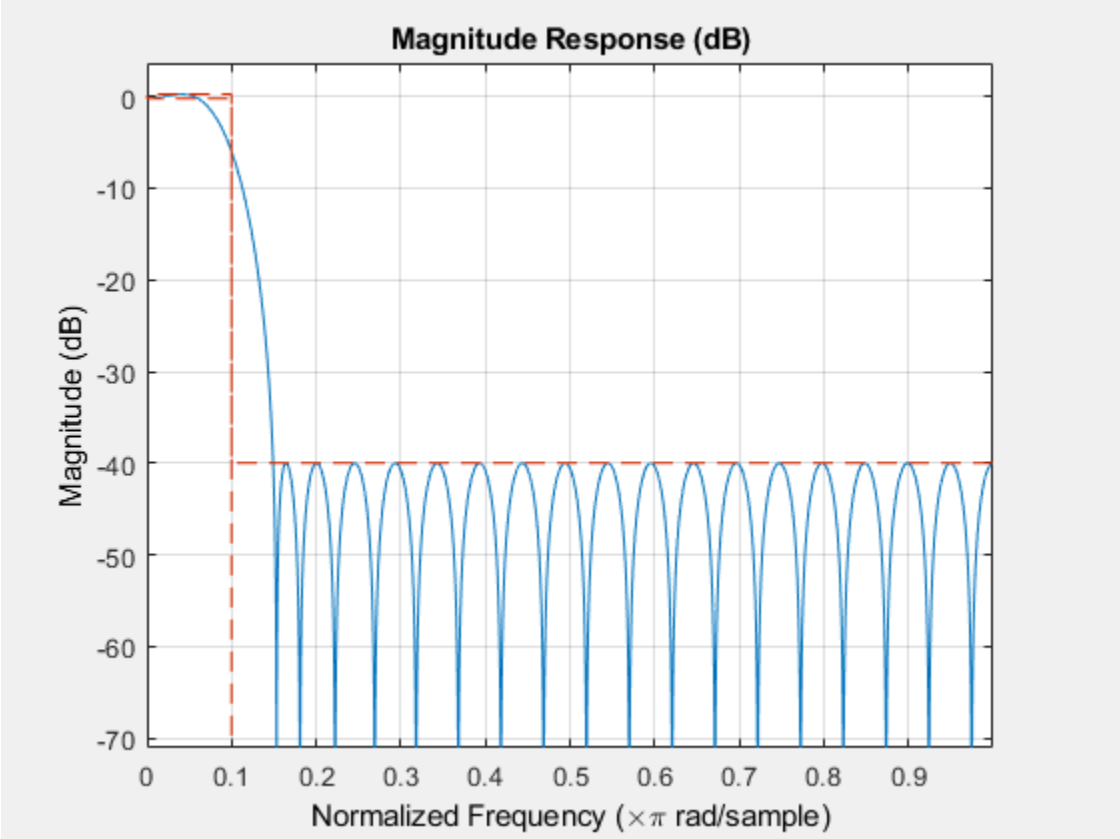


Compare the filter's frequency response to that of the ideal filter. You can see that the main lobe in the passband is not flat and the ripples in the stopband are not constrained. The moving average filter's frequency response does not match the frequency response of the ideal filter.

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 28-2
- “Measure Statistics of Streaming Signals” on page 28-14
- “Sliding Window Method and Exponential Weighting Method” on page 28-5
- “Streaming Signal Statistics” on page 4-12
- “Energy Detection in the Time Domain” on page 28-21
- “Remove High-Frequency Noise from Gyroscope Data” on page 28-24

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.

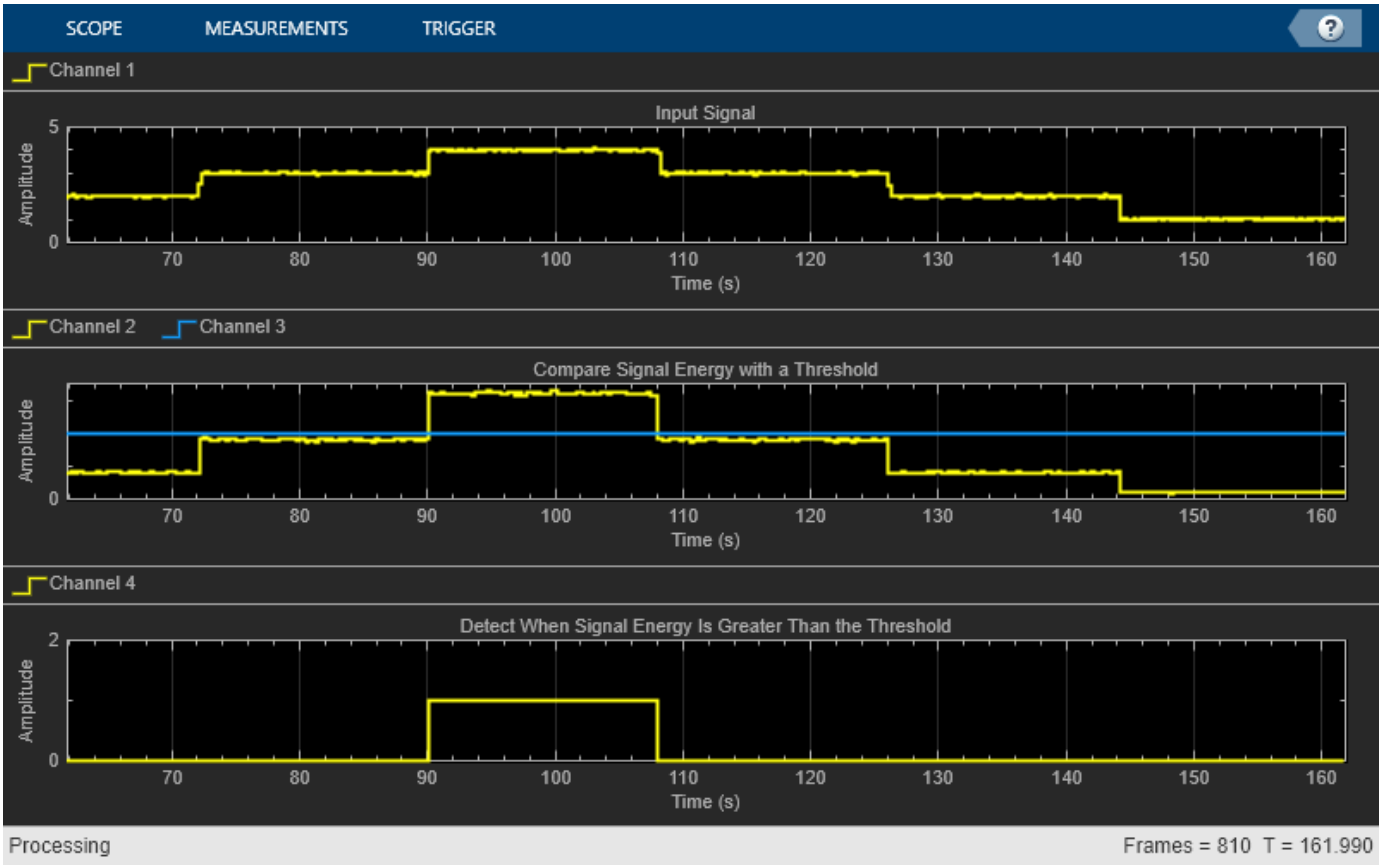
Create a `dsp.MovingRMS` System object™ to compute the moving RMS of the signal. Set this object to use the sliding window method with a window length of 20. Create a `timescope` object to view the output.

```
FrameLength = 20;
Fs = 100;
movrmsWin = dsp.MovingRMS(20);
scope  = timescope('SampleRate',Fs,...
    'TimeSpanOverrunAction','Scroll',...
    'TimeSpanSource','Property','TimeSpan',100,...
    'ShowGrid',true,'LayoutDimensions',[3 1],'NumInputPorts',3);

scope.ActiveDisplay = 1;
scope.YLimits = [0 5];
scope.Title = 'Input Signal';

scope.ActiveDisplay = 2;
scope.YLimits = [0 350];
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 = 1:90
        x = V + 0.1 * randn(FrameLength,1);
        y1 = movrmsWin(x);
        y1ener = (y1(end)^2)*20;
        event = (y1ener>threshold);
        scope(y1,[y1ener.*ones(FrameLength,1),threshold.*ones(FrameLength,1)],event.*ones(FrameLe
    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 28-2

- “Measure Statistics of Streaming Signals” on page 28-14
- “Sliding Window Method and Exponential Weighting Method” on page 28-5
- “Streaming Signal Statistics” on page 4-12
- “How Is a Moving Average Filter Different from an FIR Filter?” on page 28-17
- “Remove High-Frequency Noise from Gyroscope Data” on page 28-24

Remove High-Frequency Noise from Gyroscope Data

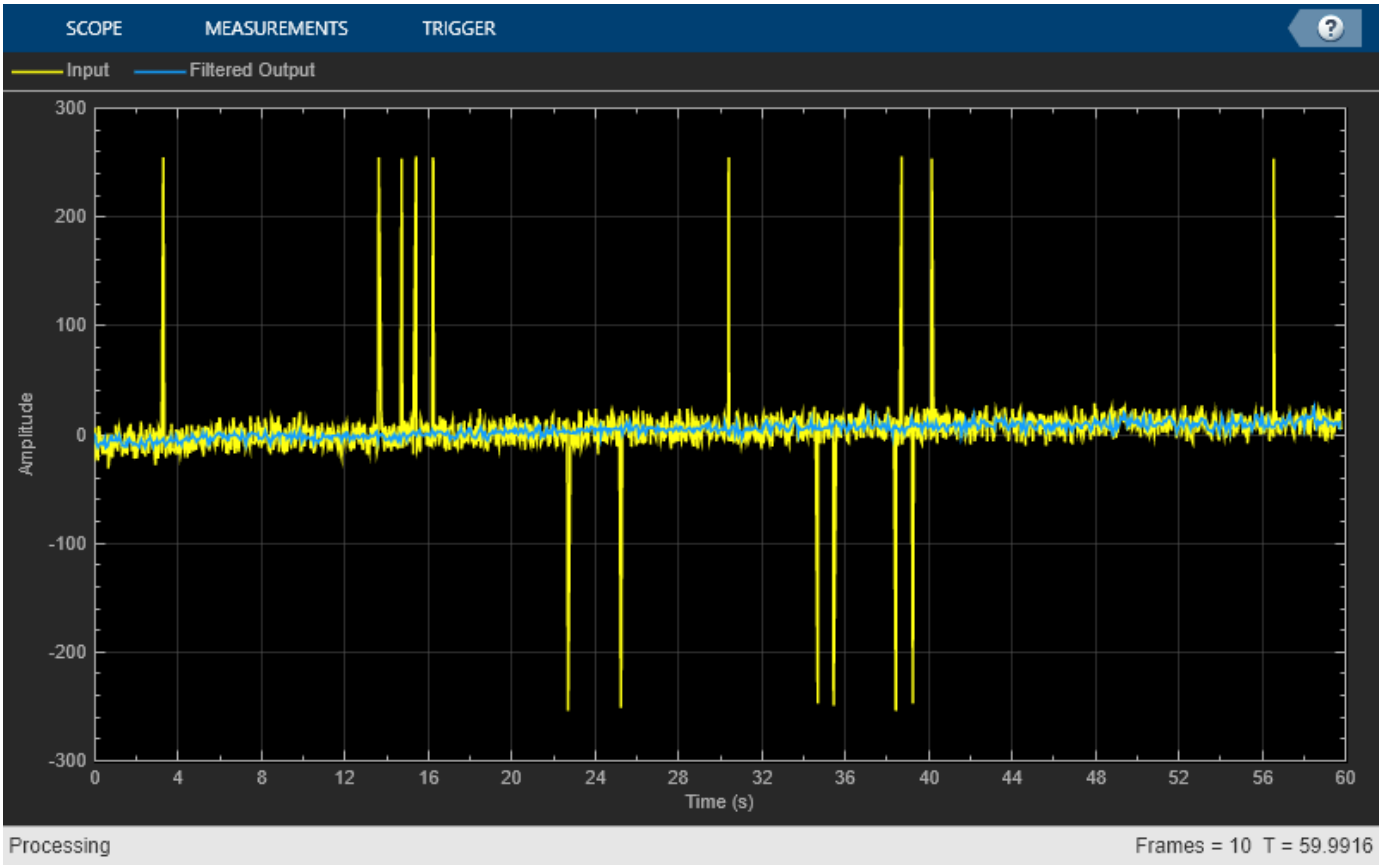
This example shows how to remove the high-frequency outliers from a streaming signal using the `dsp.MedianFilter` System object?.

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 `timescope` object to view the filtered output.

```
reader = dsp.MatFileReader('SamplesPerFrame',714,...
    'Filename','LSM9DS1gyroData73.mat',...
    'VariableName','data');
medFilt = dsp.MedianFilter(10);
scope = timescope('NumInputPorts',1,...
    'SampleRate',119,...
    'YLimits',[-300 300],...
    'ChannelNames',{'Input','Filtered Output'},...
    'TimeSpanSource','Property',...
    '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 28-2
- “Measure Statistics of Streaming Signals” on page 28-14
- “Sliding Window Method and Exponential Weighting Method” on page 28-5
- “Streaming Signal Statistics” on page 4-12
- “How Is a Moving Average Filter Different from an FIR Filter?” on page 28-17
- “Energy Detection in the Time Domain” on page 28-21

Linear Algebra and Least Squares

In this section...
“Linear Algebra Blocks” on page 28-26
“Linear System Solvers” on page 28-26
“Matrix Factorizations” on page 28-27
“Matrix Inverses” on page 28-28

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

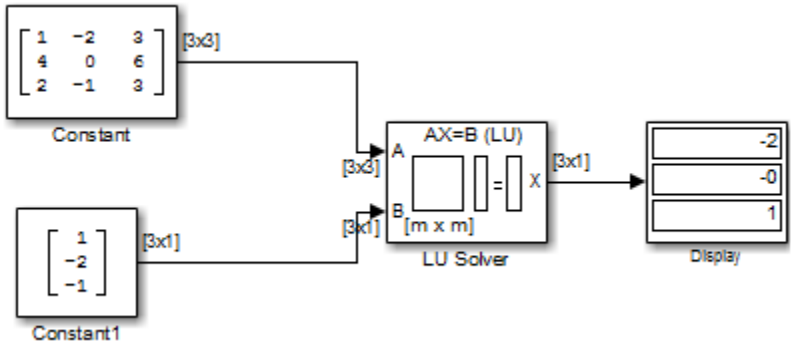
Some of the blocks offer particular strengths for certain classes of problems. For example, the Cholesky Solver block is adapted for a square Hermitian positive definite matrix A, whereas the Backward Substitution block is suited for an upper triangular matrix A.

Solve AX=B Using the LU Solver Block

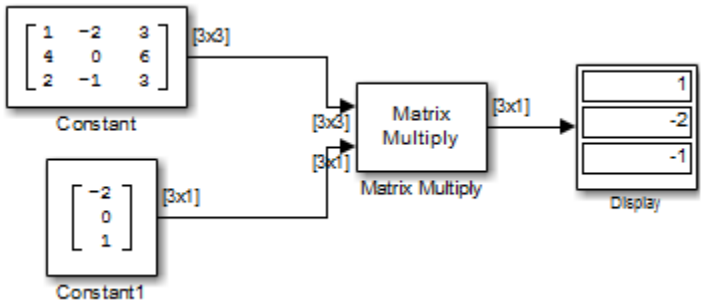
In the following ex_lusolver_tut model, the LU Solver block solves the equation A*x* = b, where

$$\mathbf{A} = \begin{bmatrix} 1 & -2 & 3 \\ 4 & 0 & 6 \\ 2 & -1 & 3 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 1 \\ -2 \\ -1 \end{bmatrix}$$

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:

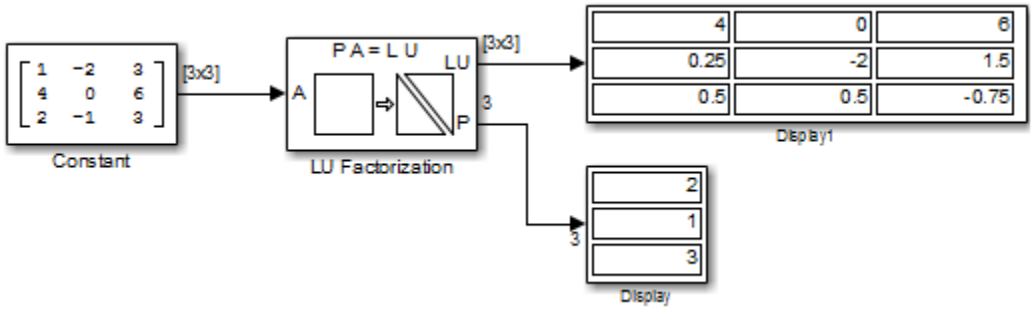
- Cholesky Factorization
- LDL Factorization
- LU Factorization
- QR Factorization
- Singular Value Decomposition

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

$$A = \begin{bmatrix} 1 & -2 & 3 \\ 4 & 0 & 6 \\ 2 & -1 & 3 \end{bmatrix}$$



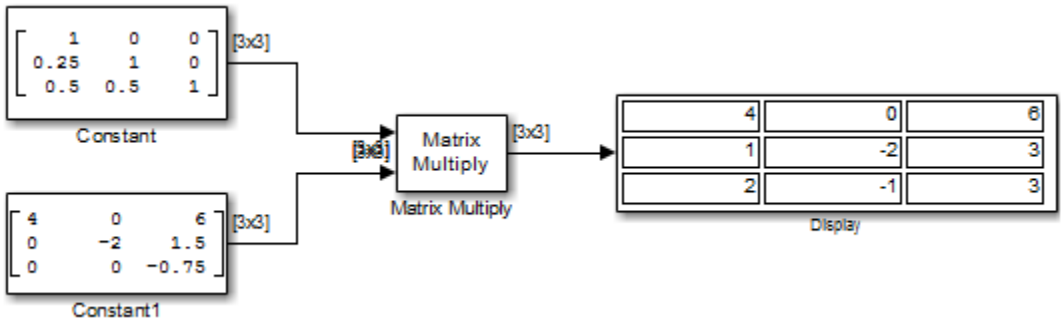
The lower output of the LU Factorization, P, is the permutation index vector, which indicates that the factored matrix A_p is generated from A by interchanging the first and second rows.

$$\mathbf{A}_p = \begin{bmatrix} 4 & 0 & 6 \\ 1 & -2 & 3 \\ 2 & -1 & 3 \end{bmatrix}$$

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:

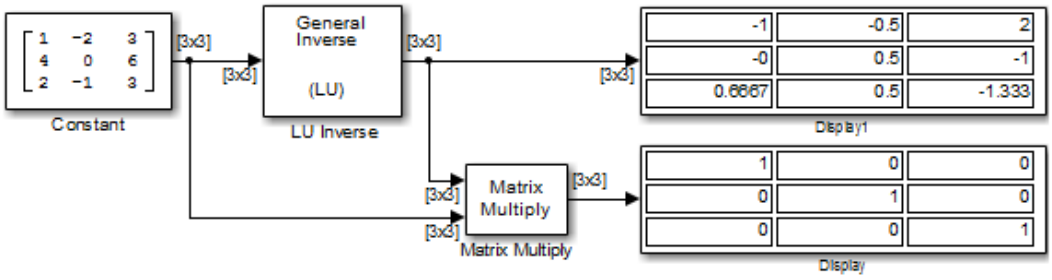
- Cholesky Inverse
- LDL Inverse
- LU Inverse
- Pseudoinverse

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

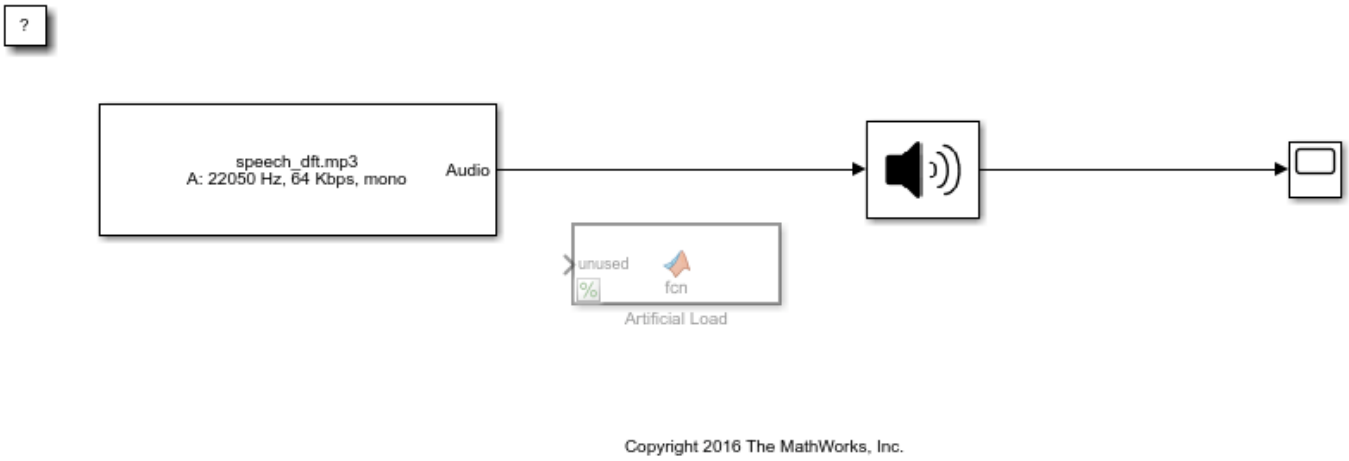
More About

- “Understanding C Code Generation in DSP System Toolbox” on page 20-6
- “MATLAB Programming for Code Generation” (MATLAB Coder)

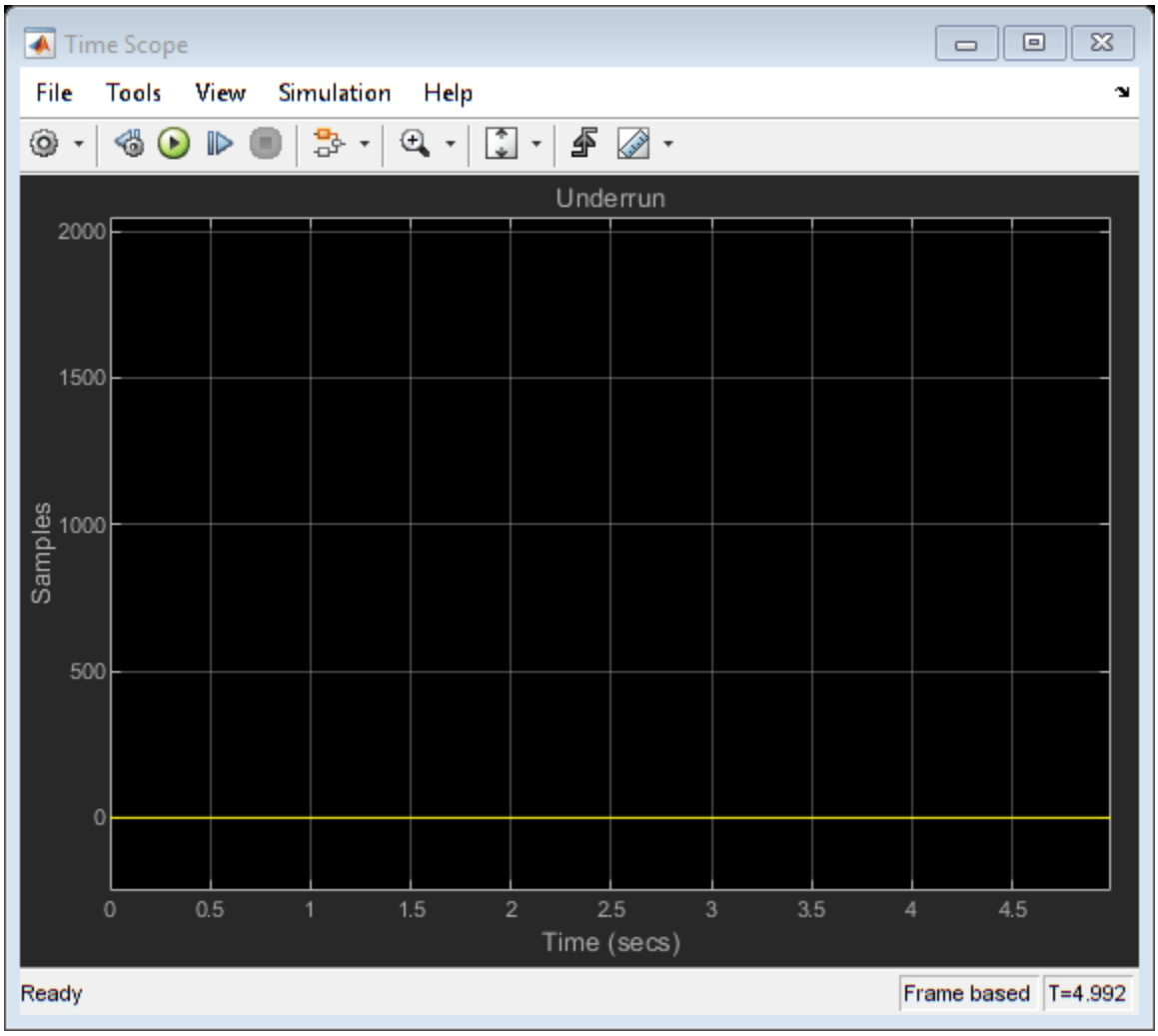
Block Example Repository

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