Filter Design Toolbox

For Use with MATLAB®

Computation

Visualization

Programming

User's Guide

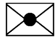
Version 2

How to Contact The MathWorks:

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`support@mathworks.com`	Technical support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 Phone

508-647-7001 Fax



The MathWorks, Inc. Mail
3 Apple Hill Drive
Natick, MA 01760-2098

For contact information about worldwide offices, see the MathWorks Web site.

Filter Design Toolbox User's Guide

© COPYRIGHT 2000 - 2002 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 or for the federal government of the United States. By accepting delivery of the Program, the government hereby agrees that this software qualifies as "commercial" computer software within the meaning of FAR Part 12.212, DFARS Part 227.7202-1, DFARS Part 227.7202-3, DFARS Part 252.227-7013, and DFARS Part 252.227-7014. The terms and conditions of The MathWorks, Inc. Software License Agreement shall pertain to the government's use and disclosure of the Program and Documentation, and shall supersede any conflicting contractual terms or conditions. If this license fails to meet the government's minimum needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to MathWorks.

MATLAB, Simulink, Stateflow, Handle Graphics, and Real-Time Workshop are registered trademarks, and TargetBox is a trademark of The MathWorks, Inc.

Other product or brand names are trademarks or registered trademarks of their respective holders.

Printing History:			
	March 2000	Online only	New for Version 1.0
	September 2000	First Printing	Revised for Vesion 2 (Release 12)
	June 2001	Online only	Revised for Version 2.1 (Release 12.1)
	July 2002	Online only	Revised for Version 2.2 (Release 13)

Contents

Preface

What Is Filter Design Toolbox? . . . . **xiv**

Related Products List . . . . **xv**

Using This Guide . . . . **xvi**
New Users of This Toolbox . . . . **xvi**
Experienced Users of This Toolbox . . . . **xvii**
Organization of This Guide . . . . **xviii**

Configuration Information . . . . **xx**

Technical Conventions . . . . **xxi**

Typographical Conventions . . . . **xxii**

1 Filter Design Toolbox Overview

Filter Design Functions in the Toolbox **1-4**

Quantization Functions in the Toolbox **1-7**
Data Quantizers **1-8**
Quantized Filters **1-9**
Quantized Fast Fourier Transforms **1-9**

Comparison to the Signal Processing Toolbox **1-11**
Filters in Signal Processing Toolbox **1-11**
Filters in Filter Design Toolbox **1-13**

Getting Started with the Toolbox **1-15**
Example - Creating a Quantized IIR Filter **1-15**
Designing the IIR Filter **1-17**
Quantizing the IIR Filter **1-21**

Selected Bibliography **1-28**

2 Designing Advanced Filters

The Optimal Filter Design Problem **2-2**
Optimal Filter Design Theory **2-2**
Optimal Filter Design Solutions **2-5**

Advanced FIR Filter Designs **2-7**
gremez Examples **2-8**
firlpnorm Examples **2-36**

Advanced IIR Filter Designs **2-42**
iirlpnorm Examples **2-45**
iirlpnormc Examples **2-50**
iirgrpdelay Examples **2-56**

Robust Filter Architectures **2-64**
Filter Design Example That Includes Quantization **2-67**

Selected Bibliography **2-73**

3 Designing Adaptive Filters

Overview of Adaptive Filters and Applications **3-4**
Choosing an Adaptive Filter **3-6**
System Identification **3-7**

Inverse System Identification 3-8
Noise Cancellation (or Interference Cancellation) 3-9
Prediction 3-9

Adaptive Filters in the Filter Design Toolbox 3-11

Examples of Adaptive Filters That Use LMS Algorithms . 3-12
adaptlms Example — System Identification 3-13
adaptnlms Example — System Identification 3-18
adaptsd Example — Noise Cancellation 3-21
adaptse Example — Noise Cancellation 3-25
adaptss Example — Noise Cancellation 3-28

Example of Adaptive Filter That Uses RLS Algorithm . . . 3-33
adaptrls Example — Inverse System Identification 3-34

Examples of Adaptive Kalman Filters 3-38
adaptkalman Example — System Identification 3-39

Selected Bibliography 3-41

4 Digital Frequency Transformations

Introduction 4-2

Definition of the Problem 4-3
Selecting Features Subject to Transformation 4-6
Mapping from Prototype Filter to Target Filter 4-8
Summary of Frequency Transformations 4-9

Frequency Transformations for Real Filters 4-11
Real Frequency Shift 4-12
Real Lowpass to Real Lowpass 4-13
Real Lowpass to Real Highpass 4-15
Real Lowpass to Real Bandpass 4-17
Real Lowpass to Real Bandstop 4-19

Real Lowpass to Real Multiband 4-21
Real Lowpass to Real Multipoint 4-23

Frequency Transformations for Complex Filters 4-26
Complex Frequency Shift 4-26
Real Lowpass to Complex Bandpass 4-28
Real Lowpass to Complex Bandstop 4-29
Real Lowpass to Complex Multiband 4-31
Real Lowpass to Complex Multipoint 4-33
Complex Bandpass to Complex Bandpass 4-36

5 Quantization and Quantized Filtering

Binary Data Types 5-3
Digital Filters 5-3
Quantized Filter Types 5-4
Quantized Filter Structures 5-4
Data Format for Quantized Filters 5-5
Quantized FFTs and Quantized Inverse FFTs 5-6

Introductory Quantized Filter Example 5-7
Constructing an Eight-Bit Quantized Filter 5-8
Analyzing Poles and Zeros with zplane 5-10
Analyzing the Impulse Response with impz 5-11
Analyzing the Frequency Response with freqz 5-12
Noise Loading Frequency Response Analysis: nlm 5-13
Analyzing Limit Cycles with limitcycle 5-14

Fixed-Point Arithmetic 5-16
Radix Point Interpretation 5-17
Dynamic Range and Precision 5-17
Overflows and Scaling 5-18

Floating-Point Arithmetic 5-19
Scientific Notation 5-19
The IEEE Format 5-20

The Exponent . . . 5-20
The Fraction . . . 5-20
The Sign Bit . . . 5-20
Single-Precision Format . . . 5-21
Double-Precision Format . . . 5-21
Custom Floating-Point Data Types . . . 5-22
Dynamic Range . . . 5-22
Exceptional Arithmetic . . . 5-24

6 Working with Objects

Objects for Quantized Filtering . . . **6-2**
Constructing Objects . . . 6-3
Copying Objects to Inherit Properties . . . 6-4

Properties and Property Values . . . **6-5**
Setting and Retrieving Property Values . . . 6-5
Setting Property Values Directly at Construction . . . 6-5
Setting Property Values with the set Command . . . 6-6
Retrieving Properties with the get Command . . . 6-8
Direct Property Referencing to Set and Get Values . . . 6-9

Functions Acting on Objects . . . **6-11**

Using Command Line Help . . . **6-12**
Command Line Help For Nonoverloaded Functions . . . 6-12
Command Line Help For Overloaded Functions . . . 6-12

Using Cell Arrays . . . **6-14**
Indexing into a Cell Array of Vectors or Matrices . . . 6-14
Indexing into a Cell Array of Cell Arrays . . . 6-15

7 Working with Quantizers

Quantizers and Unit Quantizers **7-2**

Constructing Quantizers **7-3**
Constructor for Quantizers **7-3**

Quantizer Properties **7-4**
Properties and Property Values **7-4**
Settable Quantizer Properties **7-4**
Setting Quantizer Properties Without Naming Them **7-5**
Read-Only Quantizer Properties **7-5**

Quantizing Data with Quantizers **7-6**
Example — Data-Related Quantizer Information **7-6**

Transformations for Quantized Data **7-8**

Quantizer Data Functions **7-9**

8 Working with Quantized Filters

Constructing Quantized Filters **8-3**
Constructor for Quantized Filters **8-3**
Constructing a Quantized Filter from a Reference **8-4**
Copying Filters to Inherit Properties **8-5**
Changing Filter Property Values After Construction **8-5**

Quantized Filter Properties **8-6**
Properties and Property Values **8-6**
Basic Filter Properties **8-6**
Specifying the Filter Reference Coefficients **8-7**
Specifying the Quantized Filter Structure **8-8**
Specifying the Data Formats **8-9**
Specifying All Data Format Properties at Once **8-10**

Specifying the Format Parameters with setbits **8-11**
Using normalize to Scale Coefficients **8-12**

Filtering Data with Quantized Filters **8-14**

Transformation Functions for Quantized Filter Coefficients **8-15**

9 Working with Quantized FFTs

Constructing Quantized FFTs **9-3**
Constructor for Quantized FFTs **9-3**
Copying Quantized FFTs to Inherit Properties **9-4**

Quantized FFT Properties **9-6**
Properties and Property Values **9-6**
Basic Quantized FFT Properties **9-6**
Specifying the Data Formats **9-7**
Specifying All Data Format Properties at Once **9-8**
Specifying the Format Parameters with setbits **9-8**

Computing a Quantized FFT or Inverse FFT of Data **9-10**

10 Quantized Filtering Analysis Examples

Example — Quantized Filtering of Noisy Speech **10-3**
Loading a Speech Signal **10-3**
Analyzing the Frequency Content of the Speech **10-4**
Adding Noise to the Speech **10-4**
Creating a Filter to Extract the 3000Hz Noise **10-5**
Quantizing the Filter as a Fixed-Point Filter **10-8**
Normalizing the Quantized Filter Coefficients **10-8**

Analyzing the Filter Poles and Zeros Using zplane **10-9**
Creating a Filter with Second-Order Sections **10-12**
Quantized Filter Frequency Response Analysis **10-13**
Filtering with Quantized Filters **10-14**
Analyzing the filter Function Logged Results **10-15**

Example — A Quantized Filter Bank **10-17**
Filtering Data with the Filter Bank **10-18**
Creating a DFT Polyphase FIR Quantized Filter Bank **10-18**

Example — Effects of Quantized Arithmetic **10-23**
Creating a Quantizer for Data **10-23**
Creating a Fixed-Point Filter from a Quantized Reference . . **10-23**
Creating a Double-Precision Quantized Filter **10-24**
Quantizing a Data Set **10-24**
Filtering the Quantized Data with Both Filters **10-24**
Comparing the Results **10-25**

11 Using FDATool with the Filter Design Toolbox

Switching FDATool to Quantization Mode **11-4**

Quantizing Filters in the Filter Design and Analysis Tool **11-7**
To Quantize Reference Filters **11-10**
To Change the Quantization Properties of Quantized Filters **11-11**

Analyzing Filters with the Noise Loading Method **11-12**
Using the Noise Loading Method **11-12**
Comparing the NLM and Theoretical Magnitude Responses **11-16**
Choosing Your Quantized Filter Structure **11-16**
Converting the Structure of a Quantized Filter **11-16**
To Convert Your Filter to Second-Order Sections Form **11-17**

Optimizing the Quantization Process For Your Filter . . **11-19**
Control Coefficient Quantization **11-20**
Limit Coefficient Overflow By Fraction Length Changes . . . **11-20**

Normalizing Transfer Function Coefficients 11-21
Scaling Transfer Function Coefficients 11-24
To Scale Transfer Function Coefficients 11-25
Scaling Inputs and Outputs of Quantized Filters 11-26
To Enter Scale Values for Quantized Filters 11-27

Importing and Exporting Quantized Filters 11-29
To Import Quantized Filters 11-30
To Export Quantized Filters 11-31

Transforming Filters 11-34
Original Filter Type 11-35
Frequency Point To Transform 11-38
Transformed Filter Type 11-39
Specify Desired Frequency Location 11-39
To Transform Filters 11-40

Realizing Filters as Simulink Subsystem Blocks 11-45
About the Realize Model Panel in FDATool 11-45
To Realize a Filter Using FDATool 11-47

Getting Help for FDATool 11-49
Context-Sensitive Help—The What's This? Option 11-49
Additional Help for FDATool 11-49

12 Property Reference

A Quick Guide to Quantizer Properties 12-2

Quantizer Properties Reference 12-3
Format 12-3
Max 12-5
Min 12-5
Mode 12-5
NOperations 12-6
NOverflows 12-6

NUnderflows **12-7**
OverflowMode **12-7**
RoundMode **12-8**

A Quick Guide to Quantized Filter Properties **12-10**

Quantized Filter Properties Reference **12-11**
CoefficientFormat **12-11**
FilterStructure **12-12**
InputFormat **12-38**
NumberOfSections **12-38**
MultiplicandFormat **12-38**
OutputFormat **12-39**
ProductFormat **12-40**
QuantizedCoefficients **12-40**
ReferenceCoefficients **12-40**
ScaleValues **12-48**
StatesPerSection **12-50**
SumFormat **12-50**

A Quick Guide to Quantized FFT Properties **12-51**

Quantized FFT Properties Reference **12-52**
CoefficientFormat **12-52**
InputFormat **12-52**
Length **12-53**
NumberOfSections **12-53**
MultiplicandFormat **12-53**
OutputFormat **12-54**
ProductFormat **12-54**
Radix **12-54**
ScaleValues **12-54**
SumFormat **12-55**

13 Function Reference

Functions—By Category **13-2**
Quantized Filter Construction and Property Functions **13-2**
Quantized Filter Analysis Functions **13-3**
Second-Order Sections Conversion Functions **13-4**
Quantizer Construction and Property Functions **13-4**
Quantizer Analysis Functions **13-5**
Quantized FFT Construction and Property Functions **13-6**
Quantized FFT Analysis Functions **13-6**
Filter Design Functions **13-7**
Filter Conversion Functions **13-8**
Adaptive Filter Design Functions and
Their Initialization Functions **13-9**

Functions Operating on Quantized Filters **13-10**

Functions Operating on Quantizers **13-12**

Functions Operating on Quantized FFTs **13-14**

Functions for Designing Digital Filters **13-16**

Functions—Alphabetical List **13-19**

14 Bibliography

Advanced Filters **14-2**
Adaptive Filters **14-2**
Frequency Transformations **14-3**

Preface

What Is Filter Design Toolbox? (p. xiv)	Briefly introduces the toolbox
Related Products List (p. xv)	Lists products that enhance your toolbox capabilities
Using This Guide (p. xvi)	Describes the structure and contents of this User's Guide
Configuration Information (p. xx)	Explains how you get information about the toolbox
Technical Conventions (p. xxi)	Details some technical conventions used in the text
Typographical Conventions (p. xxii)	Lists the conventions used to identify certain information in the User's Guide, such as variables and functions

What Is Filter Design Toolbox?

Filter Design Toolbox is a collection of tools built on top of the MATLAB® computing environment and the Signal Processing Toolbox. The toolbox includes a number of advanced filter design techniques that support designing, simulating, and analyzing fixed-point and custom floating-point filters for a wide range of precisions.

Note A preliminary version of Filter Design Toolbox, was released as Quantized Filtering Toolbox, Version 1.

Related Products List

The MathWorks provides several products that are especially relevant to the tasks you perform with Filter Design Toolbox.

For more information about any of these products, refer to either

- The online documentation for that product if it is installed or if you are reading the documentation from the CD
- The MathWorks Web site, at `http://www.mathworks.com`; see the “products” section

Note The toolboxes listed below all include functions that extend the capabilities of MATLAB. The blocksets all include blocks that extend the capabilities of Simulink.

Product	Description
DSP Blockset	Design and simulate DSP systems
Fixed-Point Blockset	Design and simulate fixed-point systems
Signal Processing Toolbox	Perform signal processing, analysis, and algorithm development
Simulink	Design and simulate continuous- and discrete-time systems

Using This Guide

All users of the toolbox should read this guide. You should be generally familiar with basic digital signal processing concepts before you use the toolbox and this User's Guide. The quantization portion of this toolbox assumes some familiarity with fixed-point and floating-point arithmetic in the context of digital filtering applications.

New Users of This Toolbox

You can use this toolbox to:

- Design filters using advanced design methods
- Design adaptive filters
- Transform filters from one frequency response type to another, such as from lowpass to bandstop
- Convert filters to and from coupled-allpass forms
- Convert filters to second-order section form
- Quantize filters and filter data
- Quantize data
- Compute quantized FFTs and IFFTs

This toolbox relies on object-oriented programming techniques using objects for quantized filtering and analysis. You do not need to be familiar with these techniques to use this toolbox. However, you may want to review the concepts of MATLAB structures and cell arrays, as these are used in the syntax for several toolbox methods. For more information on MATLAB structures and cell arrays, refer to "Programming and Data Types" in your MATLAB documentation.

As a new user of this toolbox, read the entire guide. Of particular interest are:

- Chapter 2, "Designing Advanced Filters" for its background information on the advanced filter design techniques in this toolbox
- Chapter 5, "Quantization and Quantized Filtering" for its background information on fixed-point and floating-point filters
- Chapter 6, "Working with Objects" for an introduction to the object-oriented techniques you need for this toolbox

- Chapter 7, “Working with Quantizers” for information on constructing and using quantizers
- Chapter 8, “Working with Quantized Filters” for information on constructing and using quantized filters
- Chapter 9, “Working with Quantized FFTs” for information on constructing and using quantized FFTs
- “Example — Quantized Filtering of Noisy Speech” on page 10-3 for a detailed example of designing and analyzing a fixed-point filter
- “Example — A Quantized Filter Bank” on page 10-17 for an example of designing and analyzing a fixed-point polyphase DFT filter bank
- Chapter 11, “Using FDATool with the Filter Design Toolbox” for information about using Filter Design and Analysis Tool to quantize filters and investigate the effects of quantization on filter performance
- “Quantizer Properties Reference” on page 12-3 for a description of the quantizer properties
- “Quantized Filter Properties Reference” on page 12-11 for a description of the quantized filter properties
- “Quantized FFT Properties Reference” on page 12-52 for a description of the quantized FFT properties
- “Functions—By Category” on page 13-2 for a brief description of every function in the toolbox

Experienced Users of This Toolbox

As an experienced user of this toolbox, you may find the following sections to be useful reference guides for the toolbox:

- “Quantizer Properties Reference” on page 12-3
- “Quantized Filter Properties Reference” on page 12-11
- “Quantized FFT Properties Reference” on page 12-52
- “Functions—By Category” on page 13-2

Organization of This Guide

This guide is organized as follows.

Chapter Title	Description
"Filter Design Toolbox Overview"	Offers an overview of the toolbox and an example to get you started using toolbox features and functions
"Designing Advanced Filters"	Provides background information on the advanced filter design methods in this toolbox
"Designing Adaptive Filters"	Introduces adaptive filtering and the functions available in the toolbox
"Digital Frequency Transformations"	Develops the theory of transforming filters and discusses the transformation functions provided
"Quantization and Quantized Filtering"	Introduces: • The concepts of quantization and filtering • An example of using, creating, and analyzing quantized filters • Some tutorial information on fixed- and floating-point arithmetic
"Working with Objects"	Introduces the object-oriented programming techniques relevant to this toolbox
"Working with Quantizers"	Provides information about constructing and using quantizers
"Working with Quantized Filters"	Covers quantized-filter specific characteristics and analysis techniques
"Working with Quantized FFTs"	Introduces constructing and using quantized FFTs
"Quantized Filtering Analysis Examples"	Presents approaches to solving some applied problems with this toolbox
"Using FDATool with the Filter Design Toolbox"	Presents a detailed reference covering the quantization page of the Filter Design and Analysis Tool

Chapter Title	Description (Continued)
"Property Reference"	Provides: • A summary of the quantized filter properties • A detailed quantized filter property reference, including descriptions of the filter structures
"Function Reference" (online only)	Provides: • Tables that include short descriptions of the functions in this toolbox • A detailed alphabetical function reference
"Bibliography"	Lists references for quantized filtering

Configuration Information

To determine whether Filter Design Toolbox is installed on your system, type this command at the MATLAB prompt.

```
ver
```

When you enter this command, MATLAB displays information about the version of MATLAB you are running, including a list of all toolboxes installed on your system and their version numbers.

For information about installing the toolbox, refer to the installation documentation for your platform.

Note For up-to-date information about system requirements, visit the system requirements page, available in the products area at the MathWorks Web site (`www.mathworks.com`).

Technical Conventions

This manual and the functions in Filter Design Toolbox use the following technical notations.

Nyquist frequency	One-half the sampling frequency. Some Signal Processing Toolbox functions normalize this to 1.
`x(1)`	The first element of a data sequence or filter, corresponding to zero lag.
`w` (used in syntax examples)	Digital frequency in radians per sample.
`f` (used in syntax examples)	Digital frequency in hertz.
$[x, y)$	The interval from x to y, including x but not including y.
`...` (used in syntax examples)	Ellipses in the argument list for a given syntax on a function reference page. These indicate that all argument options listed prior to the current syntax are valid for the function.

Typographical Conventions

This manual uses some or all of these conventions.

Item	Convention	Example
Example code	`Monospace` font	To assign the value 5 to `A`, enter `A = 5`
Function names, syntax, filenames, directory/folder names, and user input	`Monospace` font	The `cos` function finds the cosine of each array element. Syntax line example is `MLGetVar ML_var_name`
Buttons and keys	**Boldface** with book title caps	Press the **Enter** key.
Literal strings (in syntax descriptions in reference chapters)	**`Monospace bold`** for literals	`f = freqspace(n,'`**`whole`**`')`
Mathematical expressions	*Italics* for variables Standard text font for functions, operators, and constants	This vector represents the polynomial $p = x^2 + 2x + 3$.

MATLAB output	`Monospace font`	MATLAB responds with
		`A =`
		`5`

Menu and dialog box titles	**Boldface** with book title caps	Choose the **File Options** menu.
New terms and for emphasis	*Italics*	An *array* is an ordered collection of information.
Omitted input arguments	(...) ellipsis denotes all of the input/output arguments from preceding syntaxes.	`[c,ia,ib] = union(...)`
String variables (from a finite list)	*`Monospace italics`*	`sysc = d2c(sysd,'`*`method`*`')`

Filter Design Toolbox Overview

Filter Design Functions in the Toolbox (p. 1-4)	Outlines the filter design functions available in the toolbox
Quantization Functions in the Toolbox (p. 1-7)	Outlines the quantization functions available in the toolbox
Comparison to the Signal Processing Toolbox (p. 1-11)	Explains where the toolbox differs from the Signal Processing Toolbox—the different and more advanced features
Getting Started with the Toolbox (p. 1-15)	Provides an introduction to the toolbox by presenting examples that design various filters
Selected Bibliography (p. 1-28)	Lists some books that offer details about digital filtering and digital signal processing

When you install Filter Design Toolbox in your MATLAB® environment, you can perform digital filter design, fixed- and floating-point filter quantization, and filter performance analysis on your desktop computer. But what are filtering and quantization and what benefits do they provide?

Designers use filtering and its variant, digital filtering, for many tasks:

- To separate signals that have been combined, such as a musical recording and the noise added during the recording process
- To separate signals into their constituent frequencies
- To demodulate signals
- To restore signals that have been degraded by some process, known or unknown

You can use analog filters to accomplish these tasks, but digital filters offer greater flexibility and accuracy than analog filters. In addition, digital signal processing (DSP) depends in large measure on digital filtering to meet the needs of its users.

Analog filters can be cheaper, faster, and have greater dynamic range; digital filters outstrip their analog cousins in flexibility. The ability to create filters that have arbitrary shape frequency response curves, and filters that meet performance constraints, such as bandpass width and transition region width, is well beyond that of analog filters.

Quantization is a natural outgrowth of digital filtering and digital signal processing development. Also, there is a growing need for fixed-point filters that meet power, cost, and size restrictions. When you convert a filter from floating-point to fixed-point, you use quantization to perform the conversion.

As filter designers began to use digital filters in applications where power limitations and size constraints drove the filter design, they moved from double-precision, floating-point filters to fixed-point filters. When you have enough power to run a floating-point digital signal processor, such as on

desktop PC or in your car, fixed-point processing and filtering are unnecessary. But, when your filter needs to run in a cellular phone, or you want to run a hearing aid for hours instead of seconds, fixed-point processing can be essential to ensure long battery life and small size.

Filter Design Toolbox provides the functions you need to develop filters that meet the needs of fixed-point algorithms and electronics systems. In addition

to offering tools for analyzing the effects of quantization on filter performance and signal processing performance, the toolbox offers filter structures for you to use to develop prototype filter designs. With structures ranging from finite impulse response (FIR) filters to infinite impulse response (IIR) filters, you can investigate alternative fixed-point realizations of filters that might meet your goals.

This section contains the following subsections introducing filter design:

- "Filter Design Functions in the Toolbox" on page 1-4
- "Quantization Functions in the Toolbox" on page 1-7
- "Comparison to the Signal Processing Toolbox" on page 1-11
- "Getting Started with the Toolbox" on page 1-15
- "Selected Bibliography" on page 1-28

Filter Design Functions in the Toolbox

In a system that has unlimited power and size, any filter structure that met your performance specifications would do. You would design a floating-point filter whose frequency response achieved your aims and implement that filter in your system.

When you need a fixed-point filter to meet your requirements, the filter structure you choose can depend very much on how quantization affects the performance of the filter. Filter Design Toolbox offers both FIR and IIR filter design tools and structures that let you experiment with multiple filter designs to see how each responds to quantization effects.

Filter Structures

The following tables detail some of the quantized FIR and IIR filter structures available in the toolbox. For lists of all the architectures available in the toolbox, refer to the section "Quantized Filter Properties Reference" on page 12-11 in this guide.

Table 1-1: Finite Impulse Response Filter Structures

FIR Filter Structures	Description
`'antisymmetricfir`	Antisymmetric finite impulse response (FIR)
`'fir'`	Finite impulse response (FIR)
`'firt'`	Transposed finite impulse response (FIR)
`'latticema'`	Moving average (MA) lattice form
`'symmetricfir'`	Symmetric FIR

Table 1-2: Infinite Impulse Response Filter Structures

IIR Filter Structures	Description
`'df1'`	Direct form I
`'df1t'`	Direct form I transposed
`'df2'`	Direct form II

Table 1-2: Infinite Impulse Response Filter Structures

IIR Filter Structures (Continued)	Description
`'df2t'`	Direct form II transposed
`'latticeca'`	Coupled allpass lattice
`'latticecapc'`	Power-complementary output coupled allpass lattice form
`'latticear'`	Autoregressive (AR) lattice form
`'latticearma'`	Autoregressive, moving average (ARMA) lattice form
`'statespace'`	Single-input/single-output state-space

Each of the structures supports floating-point or fixed-point realizations, and you use the same toolbox function, `qfilt`, to create each one. To review schematics of the filter structures available in this toolbox, perform the following steps to run the demo "Quantized Filter Construction" in the `Filter Design` folder in MATLAB demos.

To run the filter construction demo.

1. Enter `demo` at the MATLAB command line prompt.

   The MATLAB Demo window opens on the desktop.

2. Double-click the entry **Toolboxes** in the left pane. The list of available toolboxes appears in the left pane.

3. Click **Filter Design**.

4. Click **Fixed-point Filter Construction**.

5. Click **Quantized Filter Construction** in the list of demos on the lower right.
6. Click **Run this demo** to run the demonstration model.

To access these demos directly from the MATLAB command line, enter `qfiltconstruction` at the prompt.

Quantization Functions in the Toolbox

Designing floating-point filters solves only part of the filter design problem. In most cases, floating-point filter realizations are not appropriate for digital signal processing applications. Many real-world DSP systems require that their filters use minimum power, generate minimum heat, and do not induce computational overload in their processors. Meeting these constraints often means using fixed-point filters. Unfortunately, converting a floating-point filter to fixed-point realization (called quantizing) can result in lost filter performance and accuracy. To simulate and determine the effects of quantization, and allow you to investigate how switching from floating-point to fixed-point arithmetic affects the performance of your filter, the toolbox includes quantization functions. You use the toolbox quantization functions for constructing, applying, and analyzing quantizers, quantized filters, and quantized fast Fourier transforms (FFT).

The following sections introduce the quantization functions in the toolbox. You can find details about the functions in these sections:

- "Quantizer Properties Reference" on page 12-3
- "Quantized Filter Properties Reference" on page 12-11
- "Quantized FFT Properties Reference" on page 12-52

As you read the sections about the properties, you will see that quantizers, quantized filters, and quantized FFTs share common properties and methods. At the lowest level, a quantizer forms the basis of all the quantizers in the toolbox. Each property of a quantized object is an instantiation of a data quantizer. The relationship between quantizers, quantized filters, quantized FFTs, and their underlying quantizer is shown in the following figure.

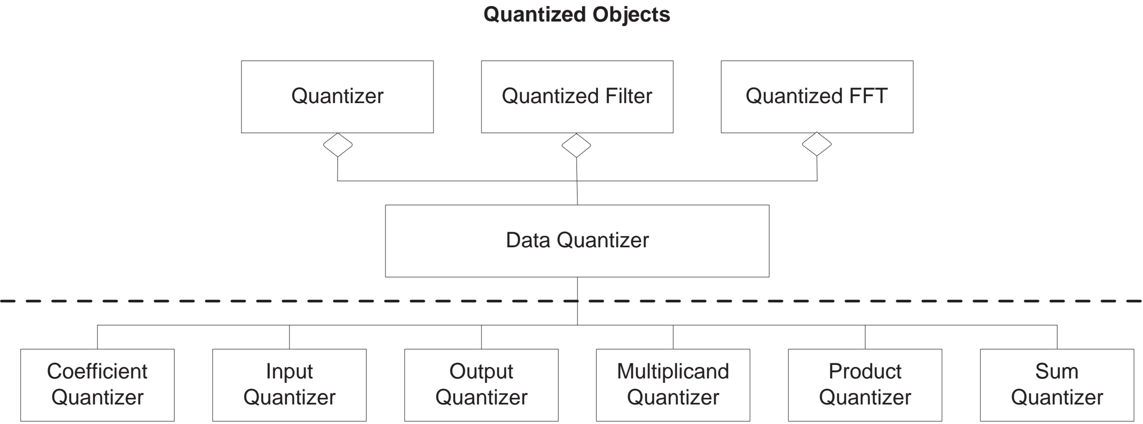


Figure 1-1: Unified Modeling Language Diagram for Filter Design Toolbox Objects

Data Quantizers

To determine how quantization affects a signal, you construct quantizers that you use to quantize a signal or data set in MATLAB. By adjusting the quantization parameters of your quantizer, you can investigate the output from various quantization schemes when you apply them to a data set or signal. In addition to experimenting with data quantization, quantizers determine how quantized filters and quantized FFTs quantize data to which they are applied.

Each quantizer you construct has the following properties that you can set when you construct the quantizer:

- `format` — determines the quantization format properties
- `mode` — determines the arithmetic data type
- `overflowmode` — determines how overflows are handled during arithmetic operations
- `roundmode` — determines the rounding method applied to data values

When you apply a quantizer, five more properties report the results of the operation:

- `max` — reports the maximum value encountered while quantizing input signals
- `min` — reports the minimum value encountered while quantizing input signals
- `noperations` — reports the total number of quantized operations performed while quantizing input signals
- `noverflows` — reports the total number of overflows, both negative and positive, that occurred while quantizing input signals
- `nunderflows` — reports the number of underflows that occurred while quantizing input signals

You cannot set these properties; they are read-only, and reflect the results of all the quantization operations that you perform with a given quantizer. Use `reset` to return quantizers to their initial settings.

Quantized Filters

Quantization, or the effect of word length on filter performance, can lead to erroneous behavior in filter designs. Finite word lengths can change the frequency response of a filter from its desired performance. To help you investigate quantization effects that occur during filtering, the toolbox provides two ways to construct a quantized filter:

- Use the function `qfilt` to create a default quantized filter.
- Use `qfilt` and specify a reference filter to quantize as an input argument.

In both techniques, your quantized filters have the same properties as quantizers.

Quantized Fast Fourier Transforms

In developing digital signal processing (DSP) algorithms, the fast Fourier transform (FFT) is one of the essential building blocks. It may be the most common transform for handling data and signals. To implement an FFT on a fixed-point DSP, you must consider the effects of word length on the output of the transform, in much the same way that you must consider the quantization effects in a digital filter. Filter Design Toolbox includes a quantized FFT (`qfft`)

function that you use to construct and apply quantized FFTs to signals and data in MATLAB. To help you investigate quantization effects that occur during the FFT, the toolbox provides you two ways to construct quantized FFTs:

- Use the function `qfft` to create a default quantized fast Fourier transform.
- Use `qfft` and specify a reference filter to quantize as an input argument.

In both techniques, your quantized FFTs have the same properties as quantizers and quantized filters.

Quantized FFTs have other properties as well; some you can set and some are read-only:

- `length` — determines the length of the FFT. Must be a power of the radix
- `numberofsections` — a read-only property reporting the number of sections in your quantized FFT
- `radix` — indicates the form of the FFT to use
- `scalevalues` — specifies the scaling for the input for each stage of the FFT

Comparison to the Signal Processing Toolbox

You use Signal Processing Toolbox and Filter Design Toolbox to design and analyze filters. Filter Design Toolbox offers advanced filter design methods for FIR and IIR filters that enhance the functionality of Signal Processing Toolbox.

Filters in Signal Processing Toolbox

Signal Processing Toolbox is data-oriented. You create separate variables for each parameter required to characterize a given filter type. For instance, to specify a state-space realization of a filter, you need four variables: one for each of the four parameters that characterize a state-space model.

Filters you design in Signal Processing Toolbox are in double-precision. You cannot design single-precision, custom-precision, or fixed-point filters. The filter design methods in Signal Processing Toolbox are listed in the following tables. Each table includes brief descriptions of the methods and functions, separated into IIR and FIR architectures:

- Table 1-3 — describes IIR filter design methods
- Table 1-4 — describes filter order estimation functions
- Table 1-5 — describes FIR filter design methods

Table 1-3: IIR Filter Design Methods in Signal Processing Toolbox

IIR Filter Design—Classical and Direct	
`besself`	Bessel analog filter design
`butter`	Butterworth analog and digital filter design
`cheby1`	Chebyshev type I filter design (passband ripple)
`cheby2`	Chebyshev type II filter design (stopband ripple)
`ellip`	Elliptic (Cauer) filter design
`maxflat`	Generalized digital Butterworth filter design
`yulewalk`	Recursive digital filter design

Table 1-4: Filter Order Estimation Functions in Signal Processing Toolbox

IIR Filter Order Estimation	
`buttord`	Calculate the order and cutoff frequency for a Butterworth filter
`cheb1ord`	Calculate the order for a Chebyshev type I filter
`cheb2ord`	Calculate the order for a Chebyshev type II filter
`ellipord`	Calculate the minimum order for elliptic filters
`remezord`	Parks-McClellan optimal FIR filter order estimation

Table 1-5: FIR Filter Design Methods in Signal Processing Toolbox

FIR Filter Design	Description
`cremez`	Complex and nonlinear-phase equiripple FIR filter design
`fir1`	Design a window-based finite impulse response filter
`fir2`	Design a frequency sampling-based finite impulse response filter
`fircls`	Constrained least square FIR filter design for multiband filters
`fircls1`	Constrained least square filter design for lowpass and highpass linear phase FIR filters
`firls`	Least square linear-phase FIR filter design
`firrcos`	Raised cosine FIR filter design
`intfilt`	Interpolation FIR filter design
`kaiserord`	Estimate parameters for an FIR filter design with Kaiser window
`remez`	Compute the Parks-McClellan optimal FIR filter design

Table 1-5: FIR Filter Design Methods in Signal Processing Toolbox (Continued)

FIR Filter Design	Description
`remezord`	Parks-McClellan optimal FIR filter order estimation
`sgolay`	Savitzky-Golay filter design

Filters in Filter Design Toolbox

To help you create and analyze quantized filters, Filter Design Toolbox is object-oriented. You encapsulate the parameters needed to specify your quantized filter under one variable name in a quantized filter object. To specify the parameters associated with a quantized filter, you set the property values for its associated named properties. These properties are assigned to the quantized filter object that represents your quantized filter.

You can design a wide range of fixed-point and custom floating-point filters in Filter Design Toolbox. You use the double-precision filters you design in Signal Processing Toolbox and Filter Design Toolbox as reference filters to create quantized filters in this toolbox. To develop a quantized filter, use either toolbox to create a double-precision filter that meets your requirements, then use the quantization functions in this toolbox to convert the double-precision filter to a quantized filter.

Refer to Table 1-6 for a list of the filter design methods in this toolbox.

Table 1-6: Filter Design Methods in the Toolbox—FIR and IIR

Filter Function	Filter Description
`firlpnorm`	Design minimax solution FIR filters using the least-pth algorithm
`gremez`	Use the generalized Remez exchange algorithm to design optimal solution FIR filters with arbitrary response curves
`iirgrpdelay`	Design optimal solution IIR filters where you specify the group delay in the passband frequencies

Table 1-6: Filter Design Methods in the Toolbox—FIR and IIR (Continued)

Filter Function	Filter Description
`iirlpnorm`	Design minimax solution IIR filters using the least-pth algorithm
`iirlpnormc`	Design minimax solution IIR filters using the least-pth algorithm. In addition, restrict the filter poles and zeros to lie within a fixed radius around the origin of the z-plane

You can construct these filters as single-precision, double-precision, custom-precision floating-point, or fixed-point structures.

Getting Started with the Toolbox

This section provides an example to get you started using Filter Design Toolbox. You can run the code in this example from the Help browser (select the code, right-click the selection, and choose **Evaluate Selection** from the context menu) or you can enter the code on the command line. This exercise also introduces Filter Design and Analysis Tool (FDATool). You use it to design and analyze filters, and to quantize filters.

As you follow the example, you are introduced to some of the basic tasks of designing a filter and using FDATool. You will engage some of the quantization capabilities of the toolbox, and a few of the filter design architectures provided as well.

Before you begin this example, start MATLAB and verify that you have installed Signal Processing and Filter Design Toolboxes (type `ver` at the command prompt). You should see Filter Design Toolbox, version 2.0 and Signal Processing Toolbox, version 5.0, among others, in the list of installed products.

Example - Creating a Quantized IIR Filter

Example Background. Wireless communications technologies, such as cellular telephones, need to account for the receiver's motion relative to the transmitter and for path changes between the stations. To model the channel fading and frequency shifting that occurs when the receiver is moving, wireless communications models apply a lowpass filter to the transmitted signal. With a narrow passband of 0 to 40Hz that modifies the transmitted signal, the lowpass filter simulates the Doppler shift caused by the motion between the transmitter and receiver. As the lowpass filter requires a rather peculiar rising shape across the passband and an extremely sharp transition region, designing and quantizing the filter presents an interesting study in filter design. In Figure 1-2, you see the frequency response curve for the RFC filter. Notice the narrow passband with the rising shape and the sharp cutoff transition. Also note that the *y*-axis is a linear scale that dramatizes the shape of the passband.

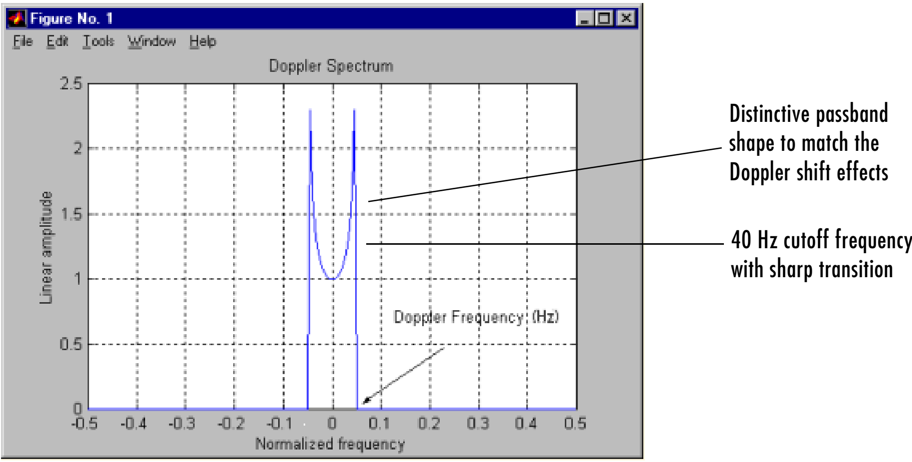


Figure 1-2: Frequency Response of the Filter Used to Simulate the Rayleigh Fading Channel Phenomenon

To create a filter with the passband shape in the figure, we define four vectors that describe the shape.

Vector	Definition
Frequency vector	Specifies frequency points along the frequency response curve. frequency can be in Hz or normalized. In our example, we are using normalized entries.
Edge vector	Specifies the edges, in Hz or in normalized values, of the passband and stopband for the filter. In our example, we are using normalized entries.
Magnitude vector	Specifies the filter response magnitude at each frequency specified in the frequency vector. These values produce the distinctive passband we require.
Weight vector	Specifies the weighting for each frequency in the frequency vector.

Most filter designs do not require you to define four vectors to specify the filter response. Because the passband of the filter we want is not standard, we are going to use the Arbitrary Magnitude filter type in FDATool when we design our filter. This type requires four input vectors to specify the filter. You can also design filters with more normal passband specifications directly in FDATool. You can enter the four vectors in FDATool, but long vectors are easier to enter at the command line. If the vectors exist as files, you can use MATLAB commands to import the vectors into your MATLAB workspace.

Designing the IIR Filter

Start to design the filter by clearing the MATLAB workspace and defining the four required vectors:

1 Clear your MATLAB workspace of all variables and close all your open figure windows. Enter

```
clear; close all;
```

2 At the MATLAB prompt, enter the following commands to create the four vectors that define the desired IIR filter frequency response.

```
PBfreq = 0:.0005:.0175; % Define the passband frequencies
```

Now specify the amplitude at each passband frequency. We use the right array divide operator (`./`) to perform element-wise division.

```
PBamp = .4845 ./ (1-(PBfreq ./ 0.0179).^2).^0.25;
```

Use `PBfreq` and `PBamp` to generate the final frequency `F` and amplitude `A` vectors for our IIR filter. While defining these vectors, define `edges` and `W`, the edge and weight vectors.

```
F = [PBfreq .02 .0215 .025 1];
edges = [0 .0175 .02 .0215 .025 1];
A = [PBamp 0 0 0 0];
W = [ones(1,length(A)-1) 300];
```

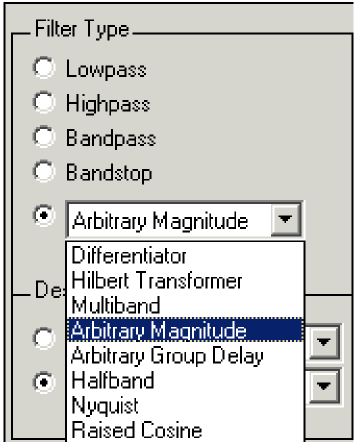
Issuing these commands created four vectors in your MATLAB workspace. FDATool uses these vectors to create an IIR lowpass filter with a specified magnitude response curve. Vectors `F` and `A` each contain 40 elements, and vectors `W` and `edges` contain 40 and 6 elements. If we were not designing

a specific passband shape, you would not have needed to define these vectors.

3 Open FDATool by typing `fdatool` at the command prompt.

   FDATool opens in Design Filter mode.

4 Under **Filter Type**, select `Arbitrary Magnitude` from the list.



   Although we want a lowpass filter, **Lowpass** does not let us specify the shape of the passband. So we use the `Arbitrary Magnitude` option to get precisely the curve we need. You could plot `F` and `A` to see that the curve is similar to the response in Figure 1-2. Use the command `plot(F,A)` to view a simple plot of the specified passband shape.

   When you select `Arbitrary Magnitude` from the list, the options under **Frequency and Magnitude Specifications** change to require three vectors: **Freq. vector**, **Mag. vector**, and **Weight vector**.

5 Continue your IIR filter design by selecting **IIR** under **Design Method**, choosing `Least Pth-norm` from the list.

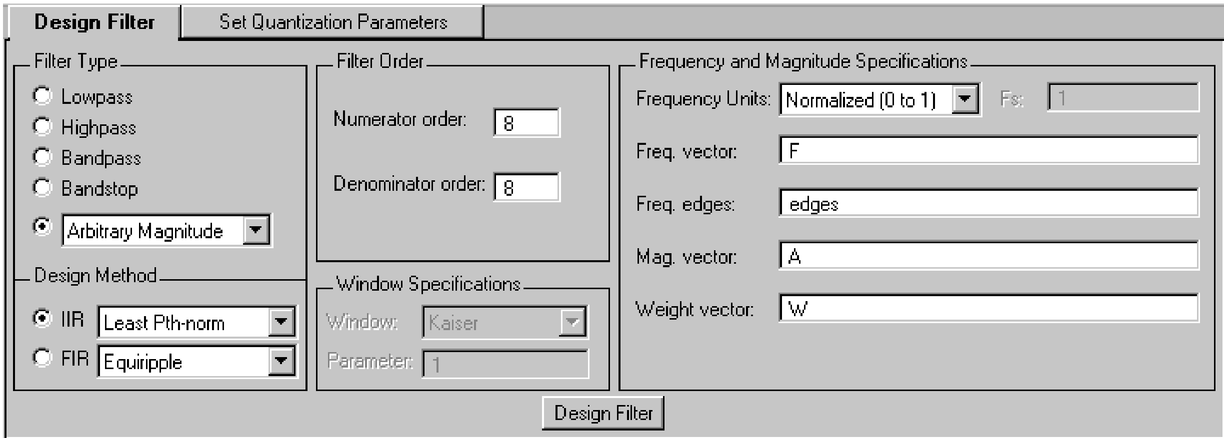
   A new vector appears under **Frequency and Magnitude Specifications — Freq. edges.**

6 Under **Frequency and Magnitude Specifications**, select `Normalized (0 to 1)` from the **Frequency Units** list.

7 Under **Frequency and Magnitude Specifications**, enter the variable names that define the four vectors required to specify the filter response.

Freq. vector, **Freq. edges**, **Mag. vector**, and **Weight vector**: `F`, `edges`, `A`, and `W`.

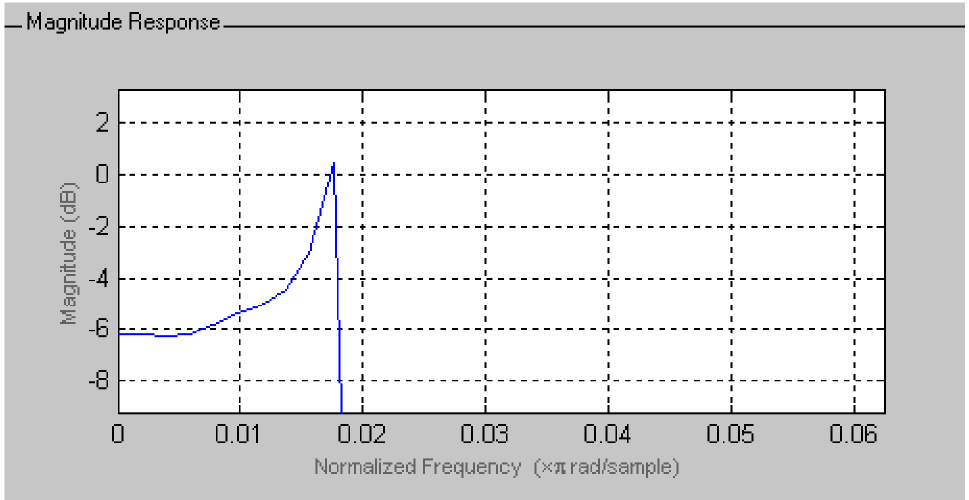
Required Vector	Variable
Freq. vector	`F`
Freq. edges	`edges`
Mag. vector	`A`
Weight vector	`W`



8 Specify the filter order by entering `8` for the numerator and denominator orders under **Filter Order**.

9 Click **Design Filter**.

FDATool designs the filter and computes the filter response. In the analysis area, you see the magnitude response of the filter displayed on a logarithmic scale.



In the upper left corner, the plot shows the region of interest for this filter. Click on the FDATool toolbar and use the zoom feature to inspect the filter passband between 0 and 0.05 (as shown in the figure). You see that the shape of the passband for the IIR filter generally matches the shape in Figure 1-2 (accounting for the shift from a linear to a logarithmic *y*-axis).

For now, we have an eighth-order, stable filter based on the direct form II transposed structure. It consists of one section.

10 To see the poles and zeros for the filter, select **Pole/Zero Plot** from the **Analysis** menu in FDATool.

For this filter, which is stable, the poles lie on or very close to the unit circle, and close to one another. Generally, when roots are close, they can be sensitive to coefficient quantization effects. Changes to the positions of the poles or zeros could cause the filter to become unstable. This is your first hint that quantizing this double-precision filter might be difficult.

Quantizing the IIR Filter

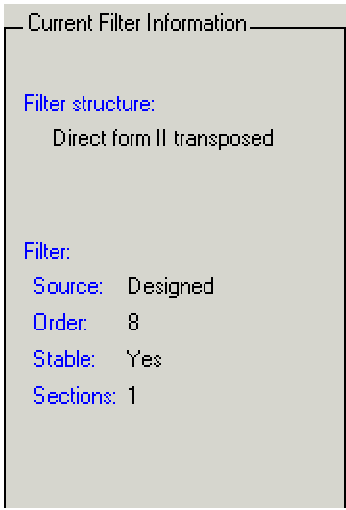
You used the filter design tools in FDATool to design an IIR filter with a passband you defined. To demonstrate the effects of quantization on this filter, we can convert the filter to fixed-point arithmetic and quantize its transfer function coefficients. So, to complete the design process, we need to quantize the IIR filter, keeping its performance intact through the quantization process. You use FDATool in quantization mode to accomplish this operation:

1 In FDATool, click to switch FDATool to quantization mode.

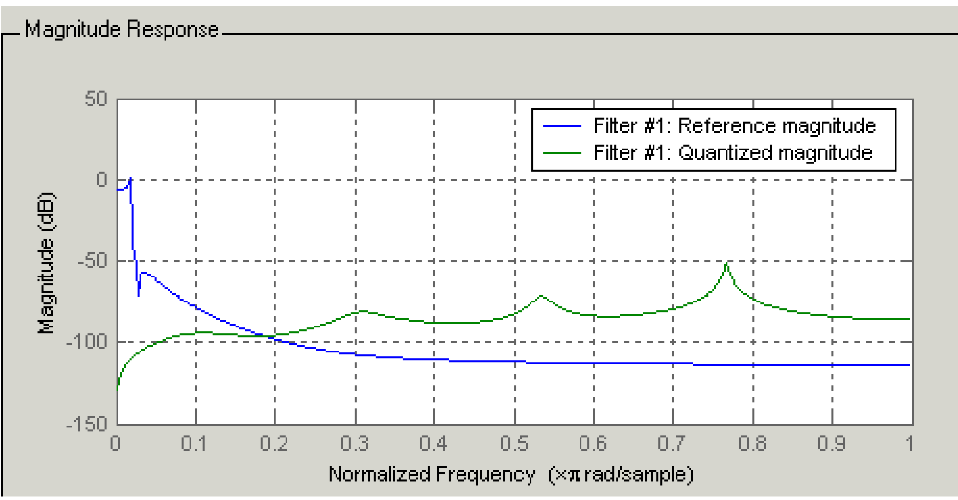
2 To quantize your IIR filter, click

   You have quantized the current filter using the defaults. Under Current Filter Information you see the filter is still stable, eighth-order, and consists of one section. Notice that Source now reads `Designed(Quantized)`. If you import a filter into FDATool, **Source** changes to read `Imported`.

   For now the filter uses the default structure — Direct form II transposed.



3 Look at the Magnitude Response in the FDATool analysis area, which now shows the response curves for both your original IIR filter (Reference) and the quantized version (Quantized).



While the new filter is stable, quantizing the filter coefficients seriously degraded its response. Truncating some of the coefficients, as you did when you quantized the filter, caused the coefficients to exceed the limits [-1,1) of the fixed data type (called *overflow*). Those coefficients were truncated to fall within the range -1 to 1. Maybe we can scale the transfer function coefficients of the reference filter so that quantizing the filter does not do such damage.

If you select **Filter Coefficients** from the **Analysis** menu in FDATool, you can review the coefficients of the reference and quantized filters. When you scroll to the bottom of the display in the analysis area, you see that eight coefficients overflowed during quantization. In the left column of the analysis area, the symbols `+`,`-`, and `0` appear to indicate which coefficients overflowed or underflowed, and in which direction (toward ±infinity or toward zero. The following table summarizes the meaning of the symbols.

Symbol	Meaning
+	Coefficients marked with this symbol overflowed toward positive infinity. FDATool handled the overflow as directed by the `Overflow mode` property value for the `Coefficient` property. In this case the setting is `saturate`.
-	Coefficients marked with this symbol overflowed toward negative infinity. FDATool handled the overflow as directed by the `Overflow mode` property value for the Coefficient property. In this case the setting is `saturate`.
0	Coefficients marked with this symbol underflowed to zero. FDATool handled the underflow as directed by the `round mode` property value for the `Coefficient` property. In this case the setting is round toward nearest.

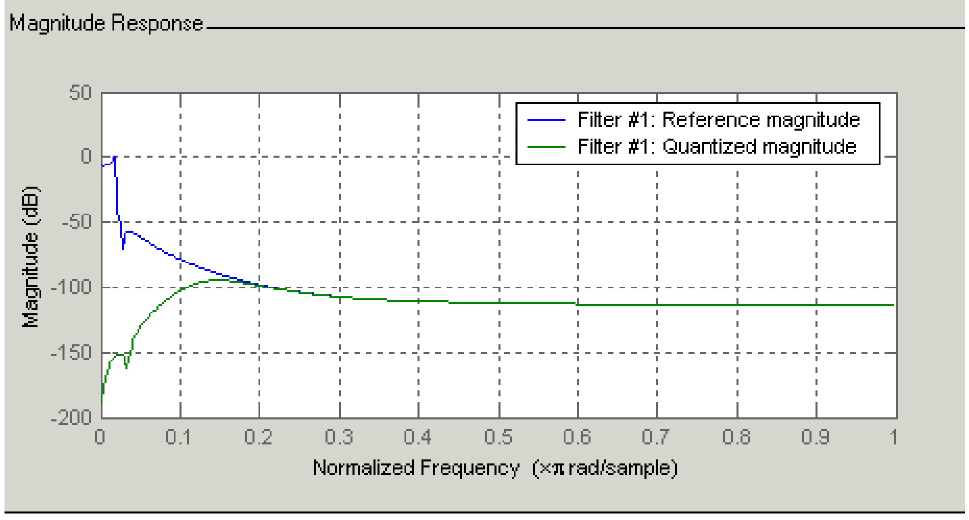
For example, the ninth numerator coefficient underflowed toward zero, and eight of the nine denominator coefficients overflowed toward plus or minus infinity and were saturated to (1-`eps`) or -1.0. The following data table shows the filter coefficients.

```
Numerator
    QuantizedCoefficients{1}   ReferenceCoefficients{1}
0 (1)      0.000000000000000   0.000006805499528066
  (2)     -0.000030517578125  -0.000037137669916463
  (3)      0.000091552734375   0.000087218236138384
  (4)     -0.000122070312500  -0.000115464066452111
  (5)      0.000091552734375   0.000094505093411602
  (6)     -0.000061035156250  -0.000048910514376539
  (7)      0.000030517578125   0.000015426381218102
0 (8)      0.000000000000000  -0.000002610607681069
0 (9)      0.000000000000000   0.000000167701400337
Denominator
    QuantizedCoefficients{2}   ReferenceCoefficients{2}
+ (1)      0.999969482421875    1.000000000000000000
- (2)     -1.000000000000000   -7.532602606298016000
+ (3)      0.999969482421875   24.769238091848504000
```

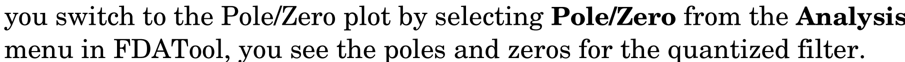
```
- (4)      -1.000000000000000  -46.426612663362768000
+ (5)       0.999969482421875   54.235802381593018000
- (6)      -1.000000000000000  -40.420742464796888000
+ (7)       0.999969482421875   18.759623438876325000
- (8)      -1.000000000000000   -4.954375991471868800
  (9)       0.569671630859375    0.569669813719955510

Warning: 8 overflows in coefficients.
```

4 Click **Scale transfer-fcn coeffs<=1**.

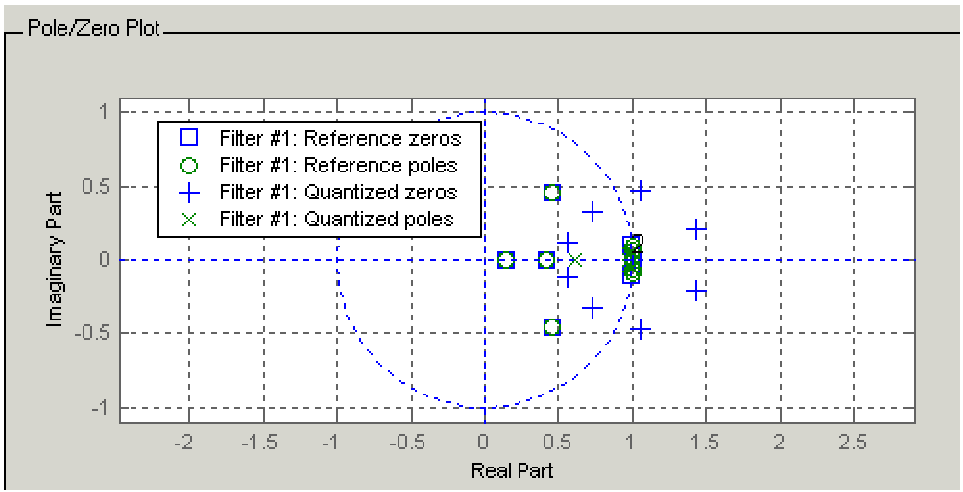
FDATool scales the reference filter coefficients, then quantizes the reference filter again. This time, the coefficients do not overflow or underflow and the filter response in the stop band appears to closely match the reference filter response, as shown in the next figure.



Your quantized filter is now unstable (check FDATool for the **Current Filter Information**). When the reference filter poles and zeros are so close to one another, they can be very sensitive to the effects of quantization. In this case, quantizing the filter moved some of the poles outside the unit circle. If

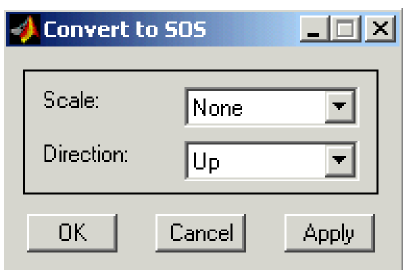


you switch to the Pole/Zero plot by selecting **Pole/Zero** from the **Analysis** menu in FDATool, you see the poles and zeros for the quantized filter.



We resolve this problem by converting our filter structure to one that is more robust to quantization effects. For example, we could change from direct form II transposed to a lattice structure, or we could use second-order sections (SOS) to implement our quantized filter. Second-order section form offers a strong option because when we convert to SOSs, we reduce the order of the polynomials that define the filter, and thus reduce the filter sensitivity to quantization.

5 To convert the filter to second-order section form, select **Edit->Convert to Second-Order Sections**.



In FDATool, you can keep your filter structure the same and convert to SOS form. Or you can change your filter structure and adopt SOS form. We want to keep the transposed direct form II structure, but use second-order sections to implement the filter.

When you convert to second-order sections (SOS), you have the option of treating the error between the reference filter magnitude response and the quantized filter magnitude response in one of three ways. The **Scale** option determines which method FDATool uses:

- `None` — ignore scaling when determining the SOS coefficients
- `L-2` — use Euclidean norm when determining the SOS coefficients
- `L-infininty` — use L_∞ scaling when determining the SOS coefficients

FDATool optimizes the order of the second-order sections according to the scaling option you choose. (The `tf2sos` function that performs the conversion is called with option `'down'` for `L-2` and `'up'` for `L-infinity` scaling.)

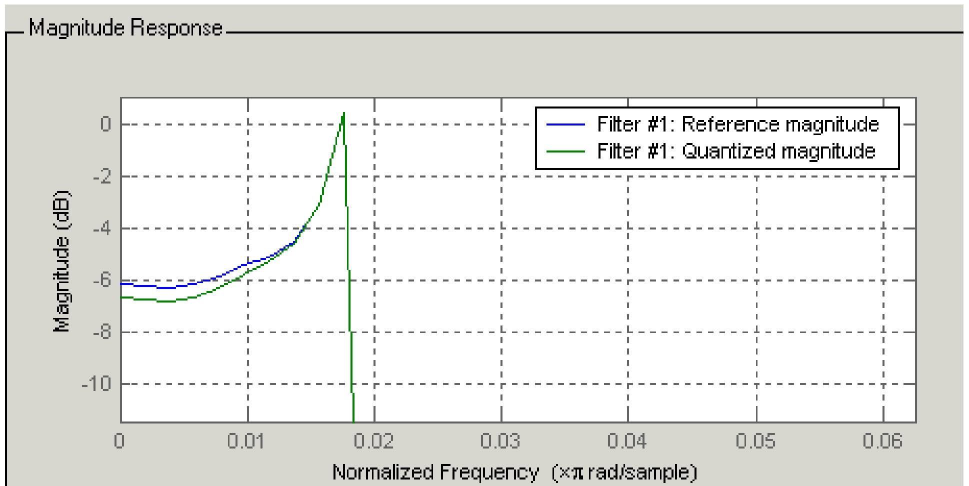
Our IIR filter does not need to be scaled to meet our needs, so select `None` from the **Scale** list and `Up` from the **Direction** list.

6 Click **OK** to close the dialog and convert the filter according to your settings.

7 Select **Magnitude Response** from the FDATool **Analysis** menu.

Our quantized second-order section filter now has the magnitude response we require, and matches the unquantized filter specifications. In the following figure showing the magnitude response curves for both filters, you cann distinguish between the reference and quantized filter curves only

within the beginning of the passband. To emphasize the match between the reference and quantized filters in the passband, use the zoom function to look more closely at the passband as shown.



As you followed this example, you created an arbitrary magnitude IIR filter to match an ideal filter response. Then you quantized the filter and converted it to second-order section form. All of this you accomplished using FDATool, although you could have used the command line to perform the same filter design and quantization operations.

To save the filter you created in FDATool, either select **File->Save Session** to save the session and your FDATool interface settings, or choose **File->Export** to export the filter to your MATLAB workspace in transfer function form.

Selected Bibliography

For further information about the algorithms and computer models used to design filters and apply quantization in the toolbox, refer to one or more of the following references.

Digital Filters

[1] Antoniou, Andreas, *Digital Filters*, Second Edition, McGraw-Hill, Inc., 1993

[2] Mitra, Sanjit K., *Digital Signal Processing: A Computer-Based Approach*, McGraw-Hill, Inc, 1998

[3] Oppenheim, Alan. V., R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, Inc, 1989

Quantization and Signal Processing

[4] Lapsley, Phil, J, Bier, A. Shoham, E.A. Lee, *DSP Processor Fundamentals*, IEEE Press, 1997

[5] McClellan, James H., C.S. Burrus, A.V. Oppenheim, T.W. Parks, R.W. Schafer, H.W. Schuessler, *Computer-Based Exercises for Signal Processing Using MATLAB 5*, Prentice-Hall, Inc., 1998

[6] Roberts, Richard A., C.T. Mullis, *Digital Signal Processing*, Addison-Wesley Publishing Company, 1987

[7] Van Loan, Charles, *Computational Frameworks for the Fast Fourier Transform,* SIAM,1992

2

Designing Advanced Filters

The Optimal Filter Design Problem (p. 2-2)	Reviews the theory of optimal filter design
Advanced FIR Filter Designs (p. 2-7)	Discusses and presents examples of advanced FIR filter designs
Advanced IIR Filter Designs (p. 2-42)	Discusses and proesents examples of advance IIR filter designs
Robust Filter Architectures (p. 2-64)	Talks about robust filters and provides some examples of robust architectures
Selected Bibliography (p. 2-73)	Offers a limited list of books that cover filter design in detail

The Optimal Filter Design Problem

Filter Design Toolbox provides you with the tools to design optimal filters in the finite impulse response (FIR) and infinite impulse response (IIR) domains.

Often, filter design techniques and algorithms result in filters that are easy to apply and put relatively light demands on computational systems. While these filters are acceptable in many instances, they are not optimal solutions to the filtering needs of some digital signal processing implementations. Suboptimal filter designs can meet the performance specifications for the filter, but generally at the expense of increased filter order. This can result in increased arithmetic computational load for each input sample and lower operating speed than may be possible and necessary.

You use the functions `firlpnorm`, `gremez`, `iirlpnorm`, and `iirlpnormc` to design optimal filters. The following sections review the optimal filter design problem and introduce the filter design functions included in the toolbox:

- "Optimal Filter Design Theory" on page 2-2
- "Optimal Filter Design Solutions" on page 2-5
- "Advanced FIR Filter Designs" on page 2-7
- "Examples—Using gremez to Design FIR Filters" on page 2-9
- "Advanced IIR Filter Designs" on page 2-42
- "Examples — Using Filter Design Toolbox Functions to Design IIR Filters" on page 2-43

Optimal Filter Design Theory

How do you design a filter that meets your performance needs, such as having the required passbands, stopbands, or transition regions, and is also the optimal solution? (The optimal solution filter minimizes a measure of the error between your desired frequency response and the actual filter response.)

Consider two filter frequency response curves:

- D(ω) — the response of your ideal filter, as defined by your signal processing needs and specifications
- H(ω) — the frequency response of the filter implementation you select

In the following figure you see the response curves for D(ω) and H(ω), both lowpass filters.

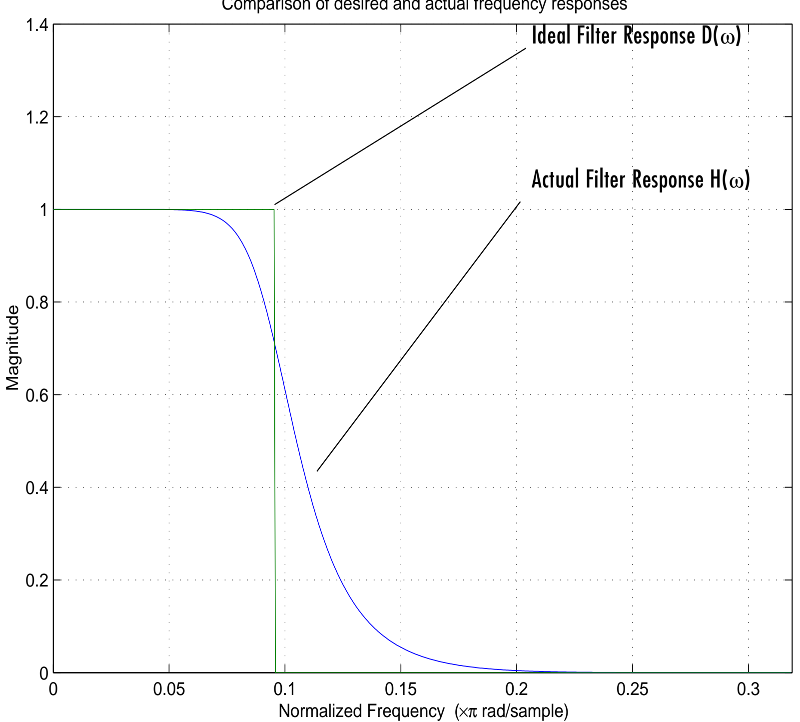


Figure 2-1: Response Curves for Ideal and Actual Lowpass Filters

Optimal filter design theory seeks to make H(ω) match D(ω) as closely as possible by a given measure of closeness.

More precisely, if we define a weighted error

$$E(\omega) = W(\omega)[H(\omega) - D(\omega)]$$

where $E(\omega)$ is the error between the ideal and actual filter response values and $W(\omega)$ is the weighting factor, the optimal filter design problem is to determine an $H(\omega)$ that minimizes some measure or norm of $E(\omega)$ given a particular weighting function $W(\omega)$ and a desired response $D(\omega)$.

$W(\omega)$, the weighting function, lets you determine which portions of the actual filter response curve are most important to your filter performance, whether passband response or attenuation in the stopband.

Usually, developers use the L_p norm to measure the error. This norm is given by

$$\int_{\Omega} [E[\omega]]^p$$

and this is the quantity we minimize.

In practice, the two most commonly used norms are L_2 and L_∞, meaning that p equals 2 and p equals infinity.

Filter designs that minimize the L_∞ are attractive because they lead to equiripple solutions. Their equiripple characteristics tend to produce the lowest order filter that satisfies some prescribed specification.

When p goes to infinity, L_∞ norm simplifies to

$$\max_{\omega \varepsilon \Omega} |E(\omega)|$$

meaning that when p equals ∞, the optimal design minimizes the maximum magnitude of the weighted error. Hence, it yields a *minimax* solution.

Notice that the L_p norm is computed over a region Ω that uses a subset of the positive Nyquist interval $[0,\pi]$. Ω covers the positive Nyquist interval except for certain frequency bands deemed to be “don’t care” regions or transition bands that are not included in the optimization.

Optimal Filter Design Solutions

We have stated that the optimal filter design problem is to find the filter whose magnitude response, $|H(\omega)|$, minimizes

$$\int_{\omega} [W(w)(|H(w)| - D(w))]^{p} dw$$

for a given Ω, p, $W(\omega)$ and $D(\omega)$. You can use both FIR and IIR filters to meet this requirement.

For the FIR case, with p equal to ∞, and the additional constraint that the filter must have linear phase, you can use a very efficient design method, based on the Remez exchange algorithm to determine the optimal solution.

Function `gremez` in the toolbox implements this method. Additionally, `gremez` provides optional calling syntaxes that enable variations and enhancements to the filter design problem.

To design optimal FIR solutions in the general case where p is not necessarily equal to infinity, the toolbox includes the function `firlpnorm`. You may find this useful in cases where minimax solutions lead to abrupt time-domain responses. `firlpnorm` does not use the Remez exchange algorithm and generally takes longer to design a filter than `gremez` and other filter design functions. Moreover, `firlpnorm` is not constrained to linear phase filters.

Note that Signal Processing Toolbox provides the function `firls`, an efficient FIR linear phase solution to the optimal filter design problem in the least-squares sense, that is, when p equals 2.

IIR solutions to the optimal filter design problem are more involved than their FIR counterparts. Filter Design Toolbox offers two functions that design IIR filters that are optimal in the least-p norm sense: `iirlpnorm` and `iirlpnormc`.

`iirplnorm` uses a somewhat faster, unconstrained algorithm, while `iirplnormc` uses a constrained algorithm that designs an optimal filter that meets the specifications while restricting the maximum radius of its poles to a specified value less than one.

Elliptic filters, such as those you use the function `ellip` (in Signal Processing Toolbox) to design, are optimal IIR filters for the case p equals infinity, when the desired magnitude response is piecewise constant, and the filter numerator and denominator orders are the same.

The Parks-McClellan method, which implements the Remez exchange algorithm, produces a filter design that just meets your design requirements, but does not exceed them. In many instances, when you use the window method to design a filter, the result is a filter that performs too well in the stopband. This wastes performance and taxes computational power by using more filter coefficients than necessary. When you use a rectangular window in the window design method, the resulting filter can be shown to be the optimal, unweighted least squares solution to the filter design problem. In summary, the optimal solution is not always a good solution to the filter design problem.

Filters designed using the Parks-McClellan method have equal ripple in their passbands and stopbands. For this reason, they are often called *equiripple* filters. They represent the most efficient filter designs for a given specification, meeting your frequency response specification with the lowest order filter.

Advanced FIR Filter Designs

Filter Design Toolbox includes a function, `gremez`, for designing FIR filters that represent the optimal solutions to filter design requirements. `gremez` provides a minimax filter design algorithm that you use to design the following real FIR filters:

- Types 1 through 4 linear phase
- Minimum phase
- Maximum phase
- Minimum order, even or odd
- Extra-ripple
- Maximal-ripple
- Constrained-ripple
- Single-point band
- Forced gain
- Arbitrarily shaped frequency response

For examples of filters that use `gremez` design features, refer to "gremez Examples" on page 2-8.

`gremez` implements the Shpak-Antoniou algorithm described in "A generalized Remez method for the design of FIR digital filters," D.J. Shpak and A. Antoniou, published in *IEEE Trans. Circuits and Systems*, pp. 161-174, Feb. 1990.

FIR filters, when implemented nonrecursively, do not use feedback in their architectures. This limits the filter design so that you include current inputs to the filter, as opposed to including past outputs (feedback) to calculate the current output of the filter. In this toolbox, you use the function `gremez` to design FIR filters. Among other features, `gremez` lets you:

- Define filters that have arbitrary shape frequency response curves
- Set a range of performance limits for a filter
- Set the weighting for the error between the desired response and the actual response in each band of interest in a filter

`remez` and `gremez` respond the same way to the same input and output arguments, where the input arguments are valid for both functions. `gremez`

extends the `remez` algorithm to support the new filter designs by adding new input argument options.

Note To provide improved FIR filter design optimization, `gremez` uses a generalized Remez algorithm that is not identical to the Remez algorithm used by `remez`. Specifically, `gremez` uses a higher density frequency grid in filter transition regions, such as at the cutoff points. Thus the frequency grid is not constant, but changes density across the frequency spectrum, letting the algorithm more closely optimize filter performance in those areas.

For more straightforward filter designs, `remez` and `gremez` generate the same filter coefficients and the same design. As the filter gets more complex, such as higher order or more bands or steeper transition regions, the filter designs may diverge. Generally, `gremez` provides better optimized filter designs in these cases.

Using `gremez` to design filters places the following restrictions on your designs:

- Design must be FIR.
- You can select the number of filter coefficients.
- The frequency response curve must be divided into a series of passbands and stopbands separated by transition or “don’t care” bands.
- Within each passband and stopband, you specify your desired amplitude response as a piecewise constant function.
- You cannot constrain the amplitude response in transition bands.

With these considerations in place, `gremez` designs *equiripple*, or *minimax*, filters to meet your specifications.

gremez Examples

Each of these examples uses one or more features provided in the function `gremez`. Review each example to get an overview of the capabilities of the function.

Examples—Using gremez to Design FIR Filters

`gremez` provides a wide range of new capabilities for FIR filter design. Because of the comprehensive nature of the generalized Remez algorithm, the best way to learn what you can do with the new function is by example. This section presents a series of examples that investigate the filters you can design through `gremez`. You can view these examples as a demonstration program in MATLAB by opening the MATLAB demos and selecting `Filter Design` from `Toolboxes`. Listed there you see a number of demonstration programs. Select `Minimax FIR Filter Design` to see function `gremez` used to create many filters, from a lowpass filter to a constrained stopband design to a minimum phase, lowpass filter with a constrained stopband.

To open the FIR filter design demo.

Follow these steps to open the FIR filter design demo in MATLAB.

1. Start MATLAB.
2. At the MATLAB prompt, enter `demos`.

   The **MATLAB Demo Window** dialog opens.
3. On the left-hand list, double-click `Toolboxes` to expand the directory tree.

   You see a list of the toolbox demonstration programs available in MATLAB.
4. Select `Filter Design`.
5. From the right-hand list, select `Minimax FIR Filter Design`.

A few examples include comparisons to other filter designs and some include analysis notes. For details about using function `gremez`, refer to Chapter 13, “Function Reference.” While this set of examples covers some of the options for `gremez`, many options exist that do not appear in these examples. Examples cover common or interesting `gremez` options to demonstrate some of the capabilities.

In each of the examples in this section, we use the output argument res to return the structure res that contains information about the filter.

Structure res Element	Contents
res.order	Filter order.
res.fgrid	Vector containing the frequency grid used in the filter design optimization.
res.H	Actual frequency response on the grid in fgrid.
res.error	Error at each point on the frequency grid (desired response- actual response).
res.des	Desired response at each point on fgrid.
res.wt	Weights at each point on fgrid.
res.iextr	Vector of indices into fgrid of extremal frequencies.
res.fextr	Vector of extremal frequencies.
res.iterations	Number of Remez iterations for the optimization.
res.evals	Number of function evaluations for the optimization.
res.edgeCheck	Results of the transition-region anomaly check. Computed when the '**check**' option is specified. One element returned per band edge. Returned values can be: • 1 = OK • 0 = Probable transition-region anomaly • -1 = Edge not checked. In the normalized frequency domain, the edges at f=0 and f=1 cannot have anomalies and are not checked.

Example—Designing a Minimax Filter

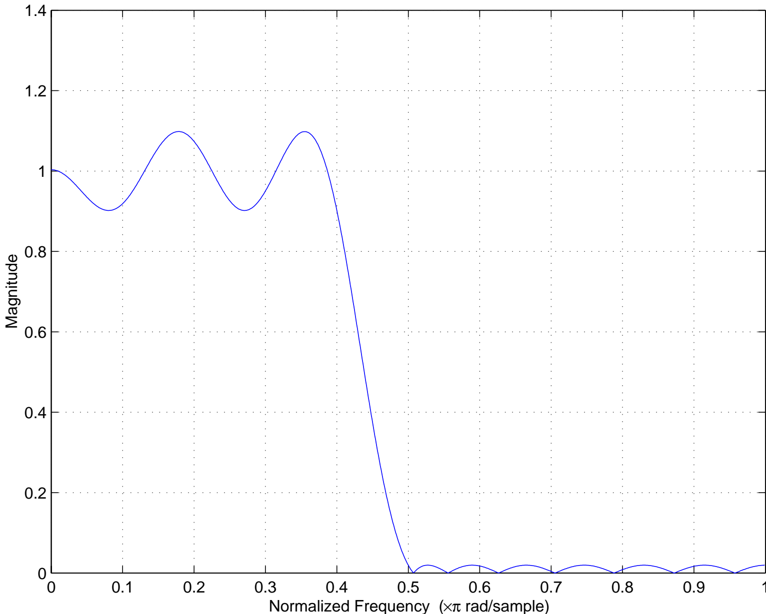
To use `gremez` to design an equiripple or minimax filter, we use the following statement.

```
[b,err,res] = gremez(22,[0 0.4 0.5 1],[1 1 0 0],[1,5]);
```

If you use the same statement, replacing `gremez` with `remez`, you get the same filter. You can reproduce any filter that `remez` generates by replacing `remez` with `gremez` in the statement. `gremez` retains full compatibility with `remez`.

Here's a plot of the magnitude response of the minimax filter as created by `gremez`. The following code creates this figure.

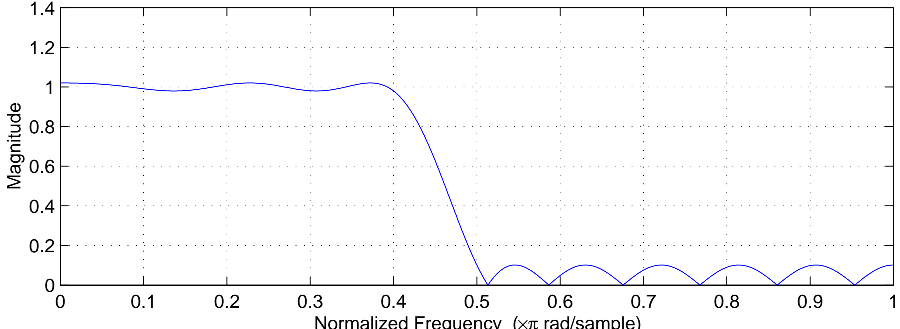
```
[h,w]=freqz(b); plot(w,abs(h))
```



Our filter ends up as a 22nd-order filter with magnitude response that has ripples about 1 in the passband and ripples about 0 in the stopband. Using the weight vector, we chose to emphasize meeting the stopband performance five times as much as meeting the passband performance. Hence the reduced ripple

in the stopband relative to the passband. In the next figure, we switch the weighting to emphasize the passband, and see that the passband ripple is much smaller than the stopband ripple.

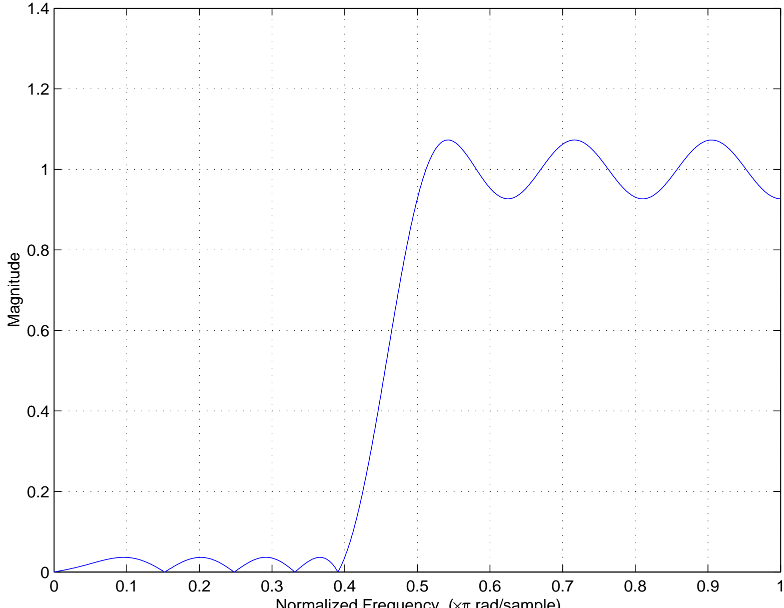
```
[b,err,res] = gremez(22,...,[5,1]);
plot(res.fgrid,abs(res.H))
```



Example—Designing a Minimax Filter, Odd-Order, Antisymmetric

In this example, `gremez` designs a filter that `remez` cannot. When you evaluate the following code in MATLAB, the result is a minimax FIR filter, this time having odd-order and antisymmetric structure, known as type 4. You can see from the figure that the magnitude response now represents a high pass filter. In this example, we specify the filter as type 4 (`'4'` in the statement) to get the odd-order, antisymmetric design we want.

```
[b,err,res]=gremez(21,[0 0.4 0.5 1], [0 0 1 1],[2 1],'4');
[H,W,S]=freqz(b,1,1024);
S.plot = 'mag'; S.yunits = 'linear';
freqzplot(H,W,S);
```



We have weighted the stopband more heavily than the passband ([2 1]) in the function syntax. The 2 and 1 tell `gremez` that we care about meeting the stopband specification twice as much as the passband specification. Notice that the weighting is relative, not absolute. Our weights say that the stopband is twice as important as the passband. They do not specify the weighting in absolute terms.

Example—Designing a "Least Squares-Like" Filter

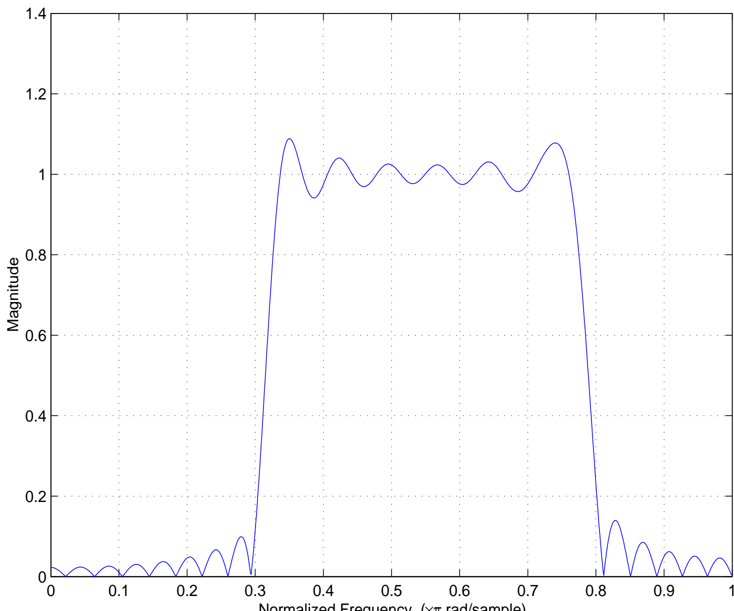
`gremez` lets you design filters that resemble least squares design. In this example, we design a 53rd-order filter and use the user-supplied file `taperedresp.m` to specify a frequency response weighting function to perform the error weighting for the design. So you can reproduce this example, the file `taperedresp.m` is in the *`matlabroot`*`\toolbox\filterdesign\filtdesdemos` folder. `taperedresp.m` contains the following code to specify the weighting.

```
% Example for a user-supplied frequency-response function
% taperedresp.m

function [des,wt] = taperedresp(order, ff, grid, wtx, aa)
nbands = length(ff)/2;
% Create output vectors of the appropriate size
des=grid;
wt=grid;

for i=1:nbands
   k = find(grid >= ff(2*i-1) & grid <= ff(2*i));
   npoints = length(k); t = 0:npoints-1;
   des(k) = linspace(aa(2*i-1), aa(2*i), npoints);
   if i == 1
      wt(k) = wtx(i) * (1.5 + cos((t)*pi/(npoints-1)));
   elseif i == nbands
      wt(k) = wtx(i) * (1.5 + cos(pi+(t)*pi/(npoints-1)));
   else
      wt(k) = wtx(i) * (1.5 - cos((t)*2*pi/(npoints-1)));
   end
end
```

To generate the least-squares-like filter, use the following code.

```
[b,err,res]=gremez(53, [0 0.3 0.33 0.77 0.8 1],...
{'taperedresp',[0 0 1 1 0 0]}, [2 2 1]);
[H,W,S]=freqz(b,1,1024);
S.plot = 'mag'; S.yunits = 'linear';
freqzplot(H,W,S);
```

When you issue these statements at the MATLAB prompt, you get the following plot for the filter magnitude response.

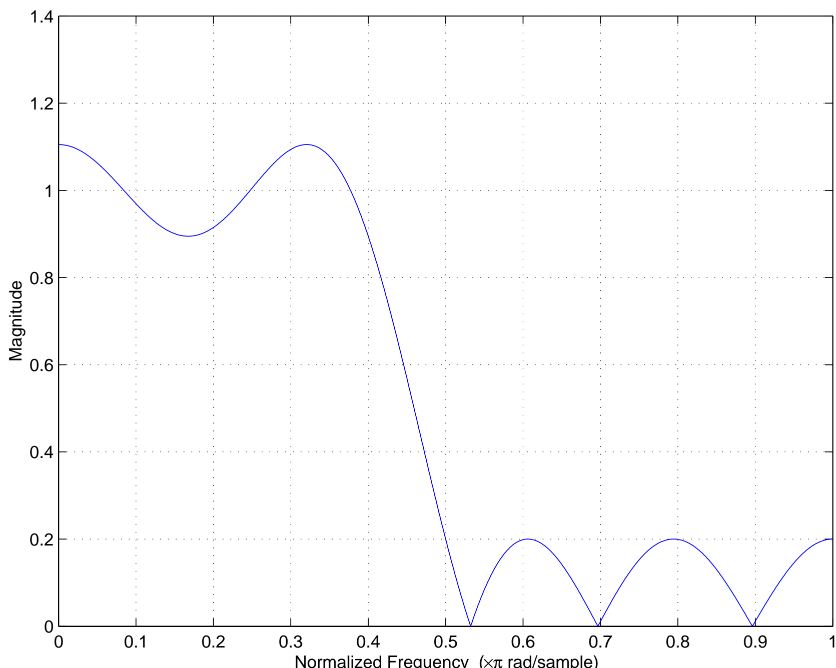


Example—Designing a Constrained Lowpass Filter

With gremez, you can both apply weighting to the passband and apply a limit or *constraint* to the error in the stopband, called constraining. Limiting the stopband error can be useful in circumstances where your filter must meet a specified stopband requirement. To create a lowpass filter with a constrained stopband and weighted passband response, we use gremez with the 'w' optional input argument to weight the passband. The optional input argument 'c' constrains the filter stopband error not to exceed 0.2. Note that to use the constraining and weighting options, your filter must have at least one unconstrained band. That is, cell array c must contain at least one 'w' entry. In our example, c is {'w' 'c'}.

```
[b,err,res]=gremez(12,[0 0.4 0.5 1], [1 1 0 0], [1 0.2],...
{'w' 'c'});
[H,W,S]=freqz(b,1,1024);
S.plot = 'mag'; S.yunits = 'linear';
freqzplot(H,W,S);
```

The next figure shows the lowpass filter with the constraints applied.



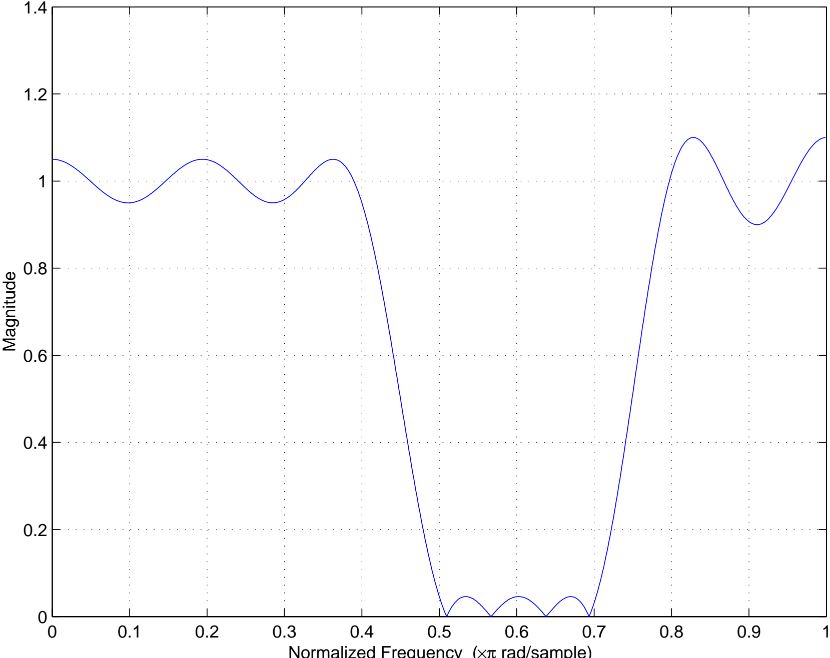
When you use constraining values in your `gremez` filter design, check to see that your filter actually touches the constraining value in the stopband. If it does not, increase the error weighting (`'w'`) for your unconstrained bands. This change causes the constrained errors to approach the constraint value more quickly. Notice that the plot shows our filter just touches the desired constraint of 0.2.

Example—Designing a Constrained Bandstop Filter

Continuing with the concept of using weighting in `gremez`, we design a bandstop filter whose passband ripple we constrain not to exceed 0.05 and 0.1. In this instance, cell array `c` is `{'c' 'w' 'c'}` to constrain the passbands and we use the optional input vector `W=[0.05 1 0.1]` to constrain the passband ripple not to exceed 0.05 and 0.1.

```
[b,err,res]=gremez(22,[0 0.4 0.5 0.7 0.8 1], [1 1 0 0 1 1],...
[0.05 1 0.1], {'c' 'w' 'c'});
[H,W,S]=freqz(b,1,1024);
S.plot = 'mag'; S.yunits = 'linear';
freqzplot(H,W,S);
```

As expected the magnitude response shows different peak ripple values in the passbands — 0.05 for the low frequency band and 0.1 for the high frequency band.



Example—Designing a Single-Point Band Filter

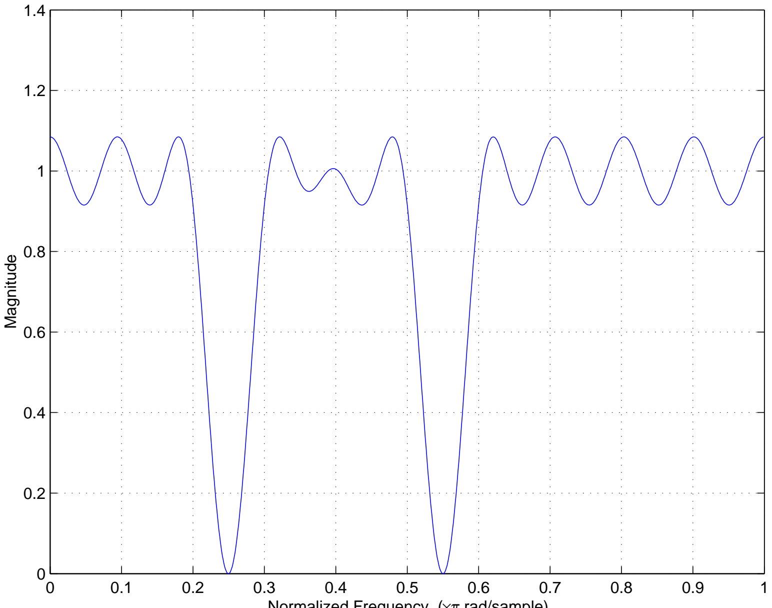
The following statements

```
[b,err,res]=gremez(42,[0 0.2 0.25 0.3 0.5 0.55 0.6 1],...
[1 1 0 1 1 0 1 1], {'n' 'n' 's' 'n' 'n' 's' 'n' 'n'});
[H,W,S]=freqz(b,1,1024);
S.plot = 'mag'; S.yunits = 'linear';
freqzplot(H,W,S);
```

generate an interesting filter that you cannot design when you use functions in Signal Processing Toolbox: a multiple stopband filter where the stop bands are defined by single points. In the `gremez` command in this example, the syntax is `b=gremez(N,F,A,S)`. The input vectors `F`, `A`, and `S`, each containing eight values, define the response curve for the filter.

Input Vector	Use
F=[0 0.2 0.25 0.3 0.5 0.55 0.6 1]	Defines the points of interest in the frequency response. In this case, you are working with frequencies normalized between 0 and 1.
A=[1 1 0 1 1 0 1 1]	Set the gain at each frequency point.
S={'n' 'n' 's' 'n' 'n' 's' 'n' 'n'}	Specifies whether the frequency points represent normal or single-point bands. By comparing the frequency and type vector entries, we see that `F=0.25` and `F=0.55` are single point bands (marked by `s`), and the gain at those points is `0`. The other bands are normal bands (marked with `n`) with gain =1.

From the next figure, you see that the filter has just the response we defined, with zeros at F = 0.25 and F = 0.55.

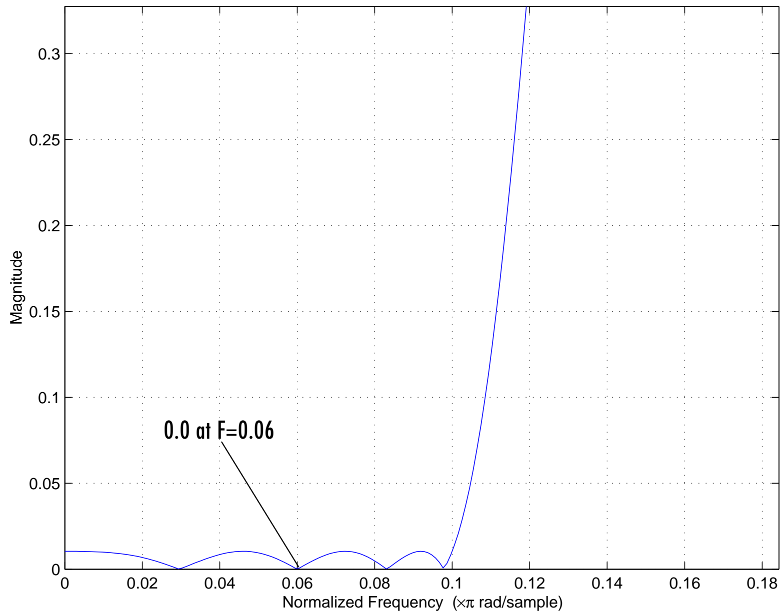


Example—Designing a Filter with a Specified In-band Value

In some filter design tasks, you want a filter whose inband value you determine exactly. For example, you might want a 60 Hz noise rejection filter to have zero gain at F = 0.06 (F = 60 Hz in real frequency). For this example, the sampling frequency is 2 KHz, so 60 Hz is F = 0.06 when we normalize the frequency. We use the following code example to design such a filter.

```
[b,err,res]=gremez(82,[0 0.055 0.06 0.1 0.15 1], [0 0 0 0 1 1],...
{'n' 'i' 'f' 'n' 'n' 'n'});
[H,W,S]=freqz(b,1,1024);
S.plot = 'mag'; S.yunits = 'linear';
freqzplot(H,W,S);
```

At F = 0.06, we require the gain of the filter response to be exactly 0.0. So we force the gain at F = 0.06 to zero by adding the `'f'` input option to the `S` vector. As shown in the plot, the filter response is zero at F = 0.06, and the resulting filter rejects 60 Hz noise quite effectively.



You might have noticed in the `gremez` statement that the `S` vector includes an `'i'` option. Entries in the `S` vector have any of the following values.

Vector Symbol	Meaning
`n`	Represents a normal frequency point
`s`	Represents a single-point band frequency
`f`	Forces the gain at this frequency to a fixed value, as specified in the weighting vector `W`
`i`	Represents an indeterminate frequency point. Usually used when the band should abut the next band

For our noise rejecting filter, the sampling frequency is 2 KHz, so 60 Hz is f=0.06 in normalized frequency.

Example—Designing Extra-Ripple and Maximal-Ripple Filters

Extra-ripple and maximal-ripple filters have some interesting properties:

- They have locally minimum transition region widths
- They tend to converge very quickly

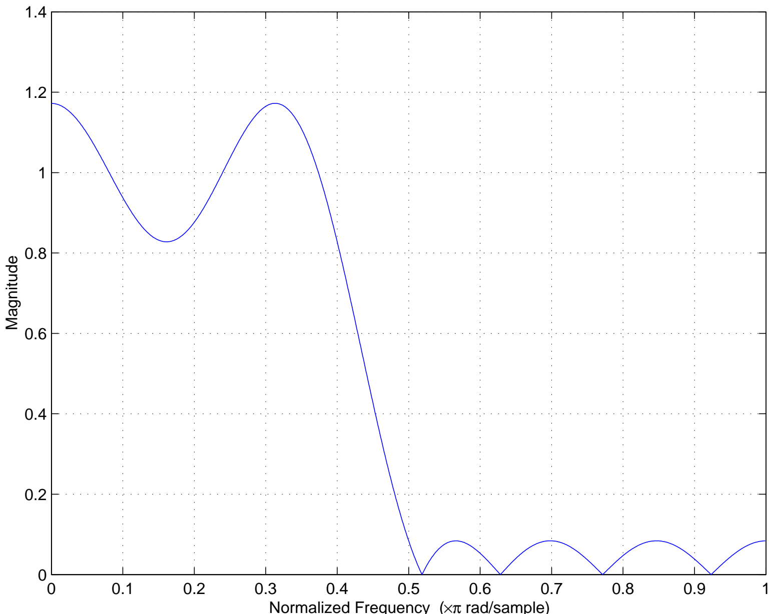
`gremez` lets you use multiple independent approximation errors to directly design extra- and maximal ripple filters. In this example, we use independent errors to design two filters, then we revisit our 60 Hz noise rejection filter to compare these two different approaches to designing the same filter.

Example of an Extra-Ripple Lowpass Filter

The code to design our extra-ripple filter is

```
[b,err,res]=gremez(12,[0 0.4 0.5 1], [1 1 0 0], [1 1],...
{'e1' 'e2'});
[H,W,S]=freqz(b,1,1024);
S.plot = 'mag'; S.yunits = 'linear';
freqzplot(H,W,S);
```

The last entries in the command, `[1 1]` and `{'e1' 'e2'}`, are the vectors `W` and `E` that determine the weights and independent approximation errors for filters with special properties. `'e1'` is applied to the passband and `'e2'` applied to the stopband. Where the `gremez` algorithm usually results in equiripple filters, using the approximations lets `gremez` adjust the ripple in each band separately, as we have done in this design.

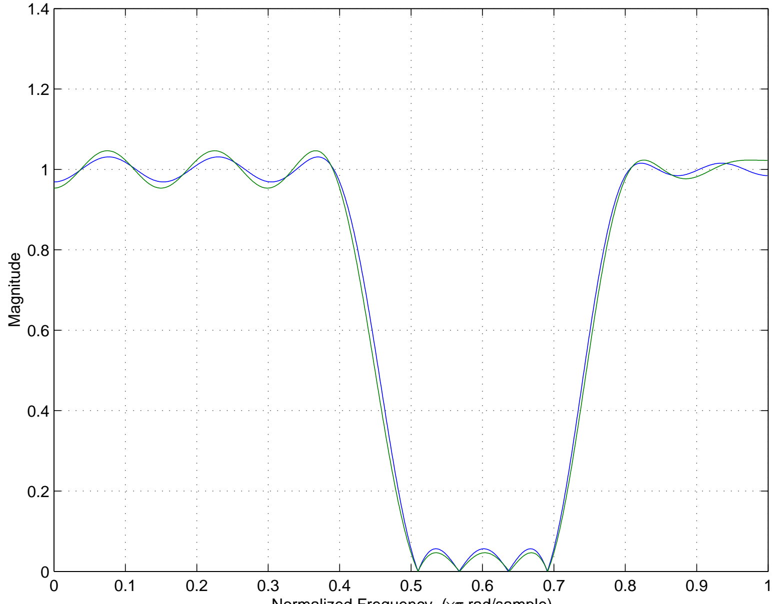


Example of an Extra-Ripple Bandstop Filter With Two Independent Approximation Errors

Now we extend the extra-ripple concept by using two independent error approximations. The two passbands share the first approximation error 'e1'. The stopband uses 'e2'. So you can see the effectiveness of this design approach, also create and plot a single approximation error filter for comparison.

```
[b,err,res]=gremez(28,[0 0.4 0.5 0.7 0.8 1], [1 1 0 0 1 1],...
[1 1 2], {'e1' 'e2' 'e1'}); % Extra-rippple filter design
[b2,err2,res2]=gremez(28,[0 0.4 0.5 0.7 0.8 1],...
[1 1 0 0 1 1],[1 1 2]); % Weighted-Chebyshev design
[H,W]=freqz(b,1,1024);[H2,W]=freqz(b2,1,1024);
S.plot = 'mag'; S.yunits = 'linear';
freqzplot([H H2],W,S);
```

In the figure, the responses are similar for the two designs, but the extra-ripple design shows less ripple in the passbands and slightly more in the stopband. If you evaluate the example code in MATLAB to create the plot, you can select **Zoom in** from the **Tools** menu in the figure window to examine the curves more closely.



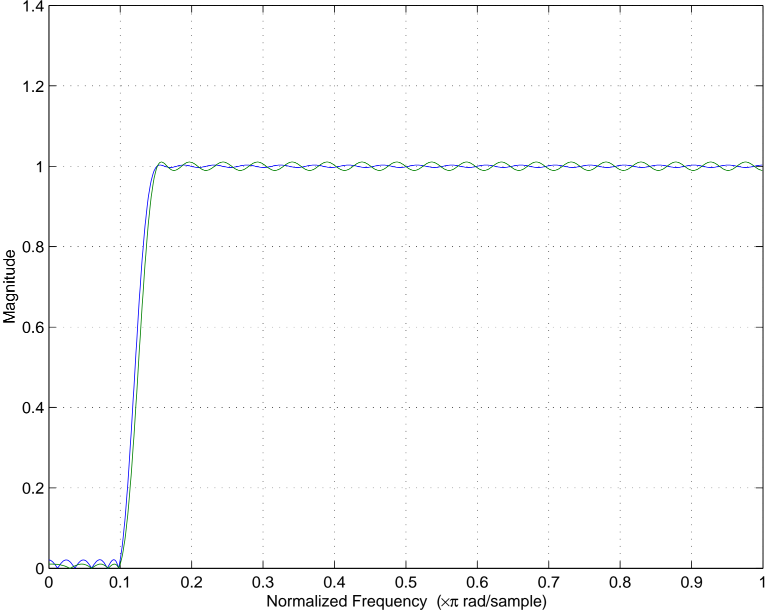
For this design, we let `gremez` use the same error approximation for the passbands and a different one in the stopband. The result is a filter that has minimum total error in the passbands, and minimum error in the stopband.

Example—Comparing Two 60 Hz Noise Rejection Filters

With the extra-ripple filter design technique available in `gremez`, we can use two different design techniques to redo our 60 Hz noise rejection filter. We use three independent error approximations in this design, one for each band, as shown in the following code.

```
[b,err,res]=gremez(82,[0 0.055 0.06 0.1 0.15 1],[0 0 0 0 1 1],...
{'n' 'i' 'f' 'n' 'n' 'n'},[10 1 1],{'e1' 'e2' 'e3'}); % New filter
[b2,err,res]=gremez(82,[0 0.055 0.06 0.1 0.15 1],...
[0 0 0 0 1 1], {'n' 'i' 'f' 'n' 'n' 'n'}); % Original filter
[H,W]=freqz(b,1,1024);
[H2,W]=freqz(b2,1,1024);
S.plot = 'mag'; S.yunits = 'linear';
freqzplot([H H2],W,S);
```

We have included the second gremez statement in this example to reproduce the earlier noise rejection filter for comparison. We plot them on the same figure for easy reference. In the stopband, the original design has lower ripple; the new, independent error design has less ripple in the passband. Also, the new filter has slightly steeper transition region performance.



Using independent approximation errors, as we did in this filter when we specified 'e1', 'e2', and 'e3', can result in better filter performance. The strings 'e1', 'e2', and so on direct gremez to consider the associated band alone, or with

other bands that use the same error approximation. By assigning independent errors to each band, we let the generalized Remez algorithm used by gremez minimize the error in each band without considering the error in the other bands. If we do not use independent errors, the algorithm minimizes the total error in all bands at once.

At times, you need to use independent approximation errors to get designs that use forced inband values to converge. Error approximations are needed where the polynomial used to approximate the filter becomes undetermined when you try to force the inband values to converge.

Example—Checking for Transition-Region Anomalies

To allow you to check your filter designs for anomalies, gremez provides an input option called 'check'. With the check option included in the command, gremez reports anomalies in the response curve for the filter. An anomaly in gremez is defined as out-of-the-ordinary response behavior in a transition, or “don’t care,” region of the filter response.

To demonstrate anomaly checking, we use gremez to design a filter with an anomaly, and include the 'check' optional input argument.

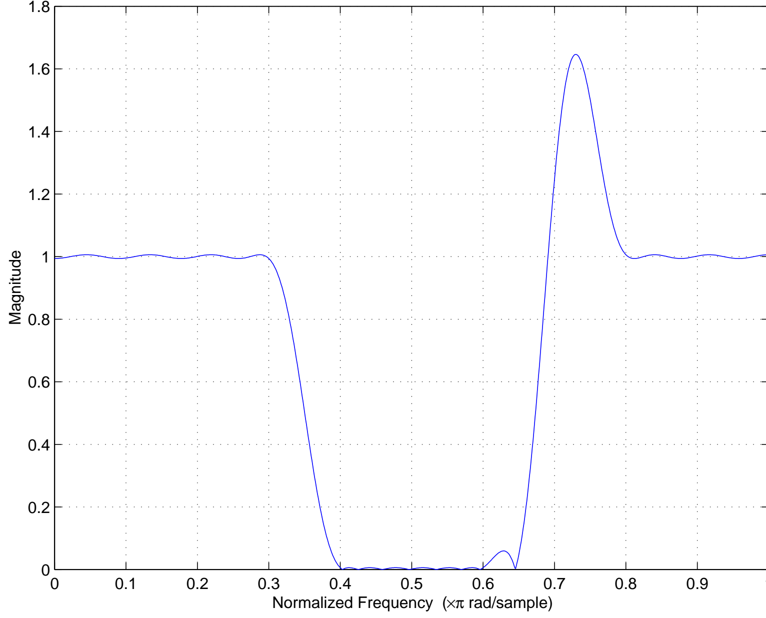
```
[b,err,res]=gremez(44,[0 0.3 0.4 0.6 0.8 1],...
[1 1 0 0 1 1],'check');
[H,W]=freqz(b,1,1024);
S.plot = 'mag'; S.yunits = 'linear';
freqzplot(H,W,S);
```

With the 'check' option, gremez returns the results vector res.edgeCheck in the structure res. Each zero-valued entry in this vector represents the location of a probable anomaly in the filter response. Entries that are not checked, such as the edges at f=1 and f=0, have -1 entries in res.edgeCheck.

To check for anomalies, the following command returns the vector containing the edge check results.

```
res.edgeCheck

ans =

    -1
     1
     1
     0
     0
    -1
```

There are anomalies between the f=0.6 and f=0.8 edges, as shown clearly in the figure. This represents a transition region for our filter. Notice that the edges at f=0 and f=1 were not checked.



In our example, the anomalous behavior happened because of the width of the transition region. When we define a narrower transition band, the anomaly disappears. Generally, reducing the transition region width eliminates anomalies in the filter response.

Example—Using Automatic Minimum Filter Order Determination

Rather than entering the filter order `N` in the `gremez` command, you can let the generalized Remez algorithm determine the minimum order for your filter. You set the specifications for the filter and the generalized Remez algorithm repeatedly designs the filter until the design just meets your specifications.

You have three options for setting the minimum order option for the filter:

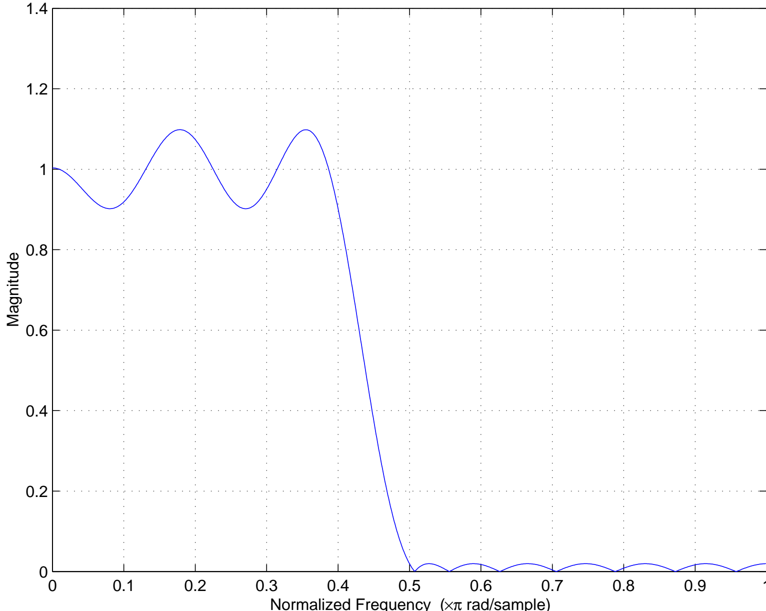
- `'minorder'` directs the Remez algorithm to iterate over the filter design until it finds a design that just fulfills your design specifications and is the lowest possible order. Using this option directs `gremez` to use `remezord` to get an initial estimate of the filter order.
- `'mineven'` directs the Remez algorithm to iterate over the filter design until it finds a design that just fulfills your design specifications and is the lowest possible even order.
- `'minodd'` directs the Remez algorithm to iterate over the filter design until it finds a design that just fulfills your design specifications and is the lowest possible odd order.

Note When you use the minimum order option `'minorder'`, `gremez` treats the weights in the `W` vector as maximum error values for the associated frequencies in the frequency vector `F`. Also, constraints become absolute limits; `gremez` designs a filter that does not exceed the constraints.

For this example, we let the Remez algorithm find a minimum order filter that implements a lowpass filter with a transition band between f=0.4 and f=0.5.

```
[b,err,res]=gremez('minorder',[0 0.4 0.5 1], [1 1 0 0],...
[0.1 0.02]);
[H,W,S]=freqz(b,1,1024);
S.plot = 'mag'; S.yunits = 'linear';
freqzplot(H,W,S);
```

Our filter, shown in the figure, demonstrates the desired ripple in the passbands and stopbands, 0.1 and 0.02; the transition region meets our specifications; and the filter order (found from `res.order`) is 22.



When you use the minimum order feature, you can specify the initial order (your best guess) in the `gremez` statement. When you estimate the order, `gremez` does not use `remezord` to make an estimate of the filter order. This is important when `remezord` does not support your desired filter type, such as differentiators and Hilbert transformers, as well as for filters that use frequency response functions that you supply. For the following filter example, we provide an initial estimate of 18 for the filter order, and we specify that we want our filter to have the minimum even order possible by adding the '`mineven`' option.

```
[b,err,res]=gremez({'mineven',18},[0 0.4 0.5 1], [1 1 0 0],...
[0.1 0.02]);
[H,W,S]=freqz(b,1,1024);
S.plot = 'mag'; S.yunits = 'linear';
```

```
freqzplot(H,W,S);
```

Though we provided an initial estimate of 18 for the order, the final order for our filter is again 22. If we had specified 'minodd', the result would be a 23rd-order filter.

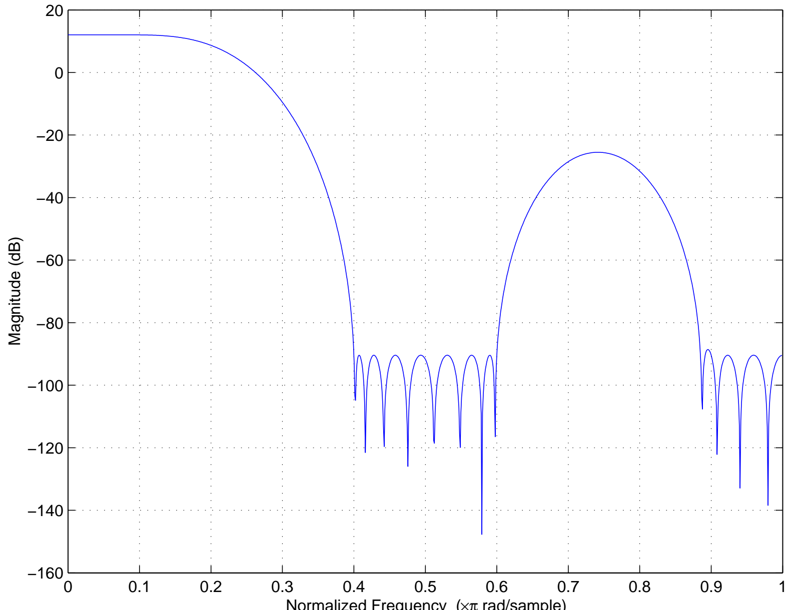
Example — Designing an Interpolation Filter

Now let us design an interpolation filter. These are usually used to upsample a band-limited signal by an integer factor, for example after the signal has been decimated by downsampling. Upsampling is often used while designing multirate filters to reduce the computational load required to use a filter. In Signal Processing Toolbox, you can use the function `intfilt` to design an interpolation filter. While `intfilt` provides a way to design the filter, it does not provide the control that `gremez` offers. Input options for `gremez` let you define the filter response and errors in each passband and stopband, and the weighting of the band responses in the filter design.

```
[b,err,res]=gremez(30,[0 0.1 0.4 0.6 0.9 1], [4 4 0 0 0 0],...
[1 100 100]);
[H,W]=freqz(b,1,1024);
S.plot = 'mag'; S.yunits = 'db';
freqzplot(H,W,S);
```

We specify a 30th-order filter with edges at 0.1, 0.4, 0.6, and 0.9, and weight them as [1 100 100]. The resulting design has stopbands between f=0.4 and f=0.6, and f=0.9 and f=1.0.

The next figure shows a filter designed by `gremez`.



Example—Comparing Filters Designed by gremez and intfilt

Now, to see that `gremez` lets you develop a better interpolation filter than `intfilt`, we compare filters designed by both functions. We need three sets of code to display the filters for our comparison — the first set generates the detail plot of the first stopband, the second set displays the second stopband in detail, and the third plot focuses on the stopband ripple. To keep the frequency response displays consistent, we use the MATLAB `plot` function to ensure that the axes and labels are the same for both filters. `freqzplot` does not provide enough control of the plotting functions.

Code to display the first stopband.

```
[b,err]=gremez(30,[0 0.1 0.4 0.6 0.9 1], [4 4 0 0 0 0],...
[1 100 100]);
b2=intfilt(4, 4, 0.4);
w=linspace(0.4, 0.6)*pi; h=freqz(b,1,w); h2=freqz(b2,1,w);
plot(w/pi,20*log10(abs([h' h2']))); ylabel('Stopband #1 (dB)');
v=axis; v=[0.4 0.6 -100 v(4)]; axis(v);
```

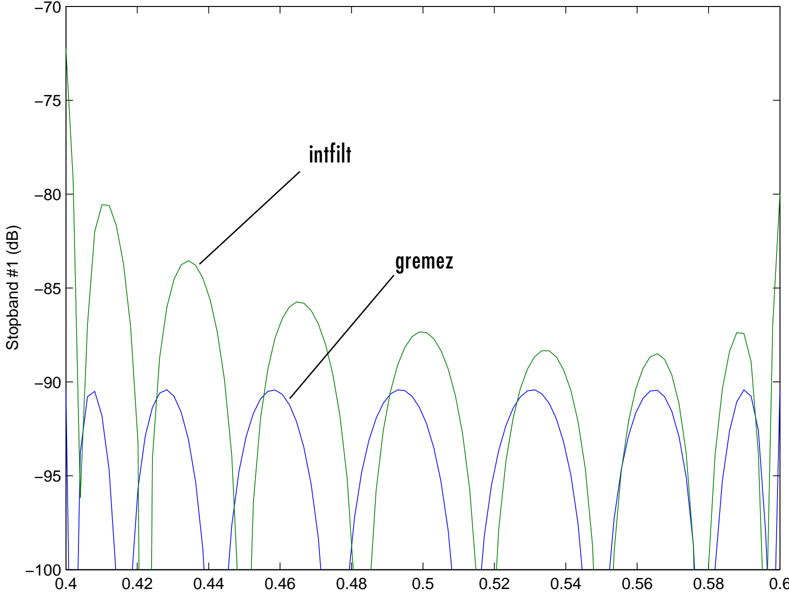
Code set to display the second stopband.

```
[b,err]=gremez(30,[0 0.1 0.4 0.6 0.9 1], [4 4 0 0 0 0],...
[1 100 100]);
b2=intfilt(4, 4, 0.4);
w=linspace(0.9, 1)*pi; h=freqz(b,1,w); h2=freqz(b2,1,w);
plot(w/pi,20*log10(abs([h' h2'])));  ylabel('Stopband #2 (dB)');
v=axis; v=[0.9 1 -100 v(4)]; axis(v);
```

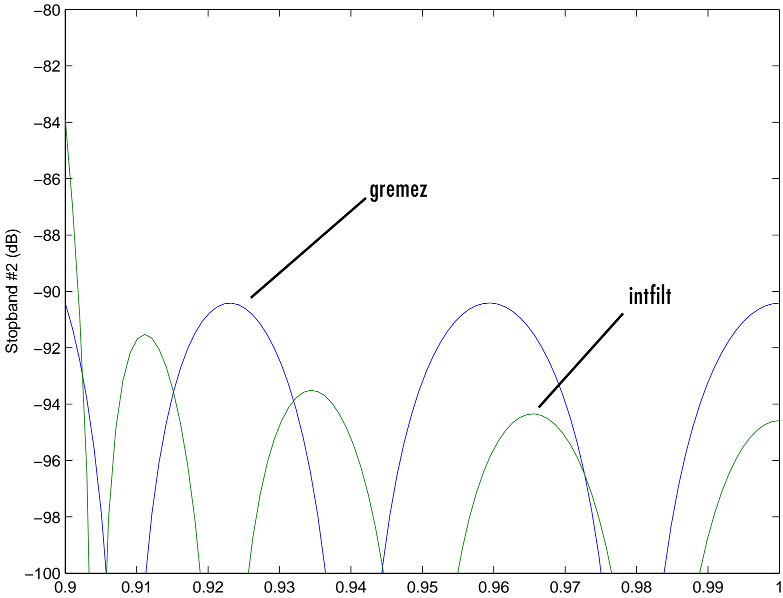
Code set to display the passband ripple.

```
[b,err]=gremez(30,[0 0.1 0.4 0.6 0.9 1], [4 4 0 0 0 0],...
[1 100 100]);
b2=intfilt(4, 4, 0.4);
w=linspace(0, .1)*pi; h=freqz(b,1,w); h2=freqz(b2,1,w);
plot(w/pi,20*log10(abs([h' h2'])));  ylabel('Passband (dB)');
```

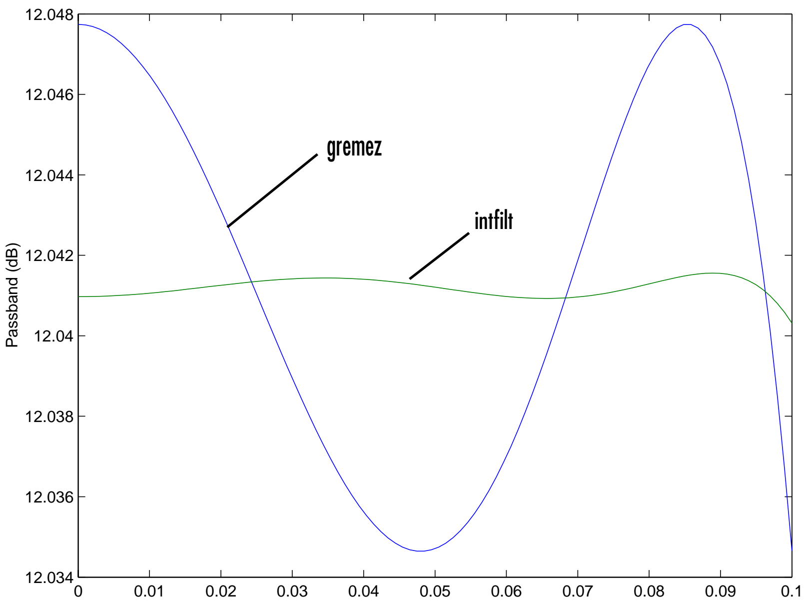
In the next figure, showing the first stopband in detail, you see that using the weighting function in `gremez` improved the minimum stopband attenuation by almost 20 dB over the `intfilt` design.



If we switch to a plot of the second stopband, shown in the next figure, you see that the equiripple attenuation throughout the band is about 6 dB larger for the `gremez`-generated filter than the minimum stopband attenuation of the filter designed by `intfilt`.



Finally, let's look at the passbands of the two filters, shown in the next figure. Here, the ripple in the `gremez`-designed filter is slightly larger than the passband ripple for the `intfilt` design. Still, both are very small, less than 0.014 dB peak-to-peak.



Example—Designing a Minimum Phase Lowpass Filter with a Constrained Stopband

With `gremez` you can determine whether the FIR filter you design is minimum phase, maximum phase, or linear phase. Through this example we show a minimum phase filter and look at the roots of the filter transfer function to see that no roots lie outside the unit circle in the z-plane. First, we create the minimum phase filter by using `gremez` with the `'minphase'` optional input argument.

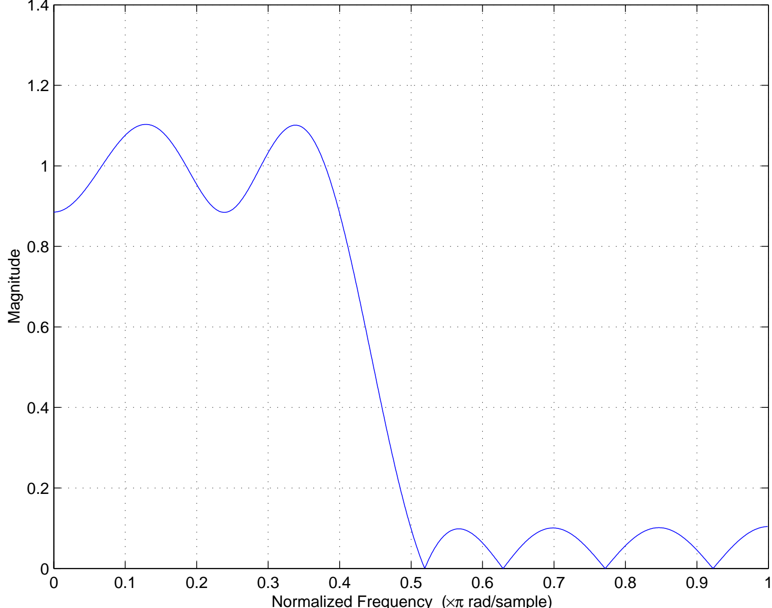
```
[b,err,res]=gremez(12,[0 0.4 0.5 1], [1 1 0 0],[1 0.1],...
{'w' 'c'},{64},'minphase');
```

`gremez` generates a lowpass filter with constrained stopband magnitude equal to 0.1, and the filter is minimum phase as well. We could have specified a maximum phase design by replacing the `'minphase'` option with `'maxphase'`. In the `gremez` statement, you might have noticed the cell array `{64}` entry. The cell array entries define the grid density for points across the frequency spectrum.

Now, plot the filter to view the frequency response.

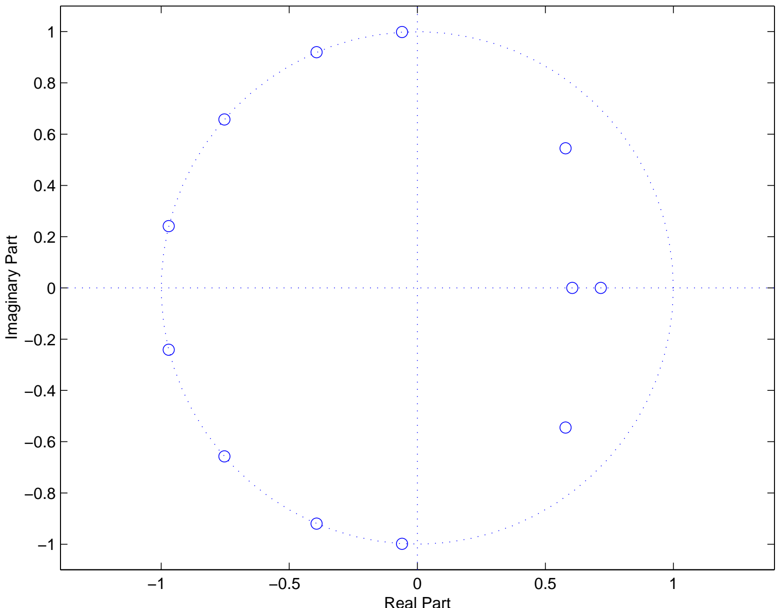
```
[H,W]=freqz(b,1,1024);
S.plot = 'mag'; S.yunits = 'linear';
freqzplot(H,W,S);
```

We have a lowpass filter with stopband ripple not exceeding 0.1, as desired.



In the next figure, viewing our filter roots on the z-plane plot shows us that the roots lie in or on the unit circle. The zeros of a minimum phase delay FIR filter lie on or inside the unit circle. Maximum phase delay filters have zeros that lie on or outside the unit circle.

```
[b,err,res]=gremez(12,[0 0.4 0.5 1], [1 1 0 0],[1 0.1],...
{'w' 'c'},{64},'minphase');
[H,W]=freqz(b,1,1024);
zplane(roots(b));
```



Notice that the filter, with eight zeros on the unit circle, could be very sensitive to quantization. You could use FDATool to investigate the effects of quantizing this filter, and to convert the filter to second order sections or make other changes the reduce the sensitivity to quantization.

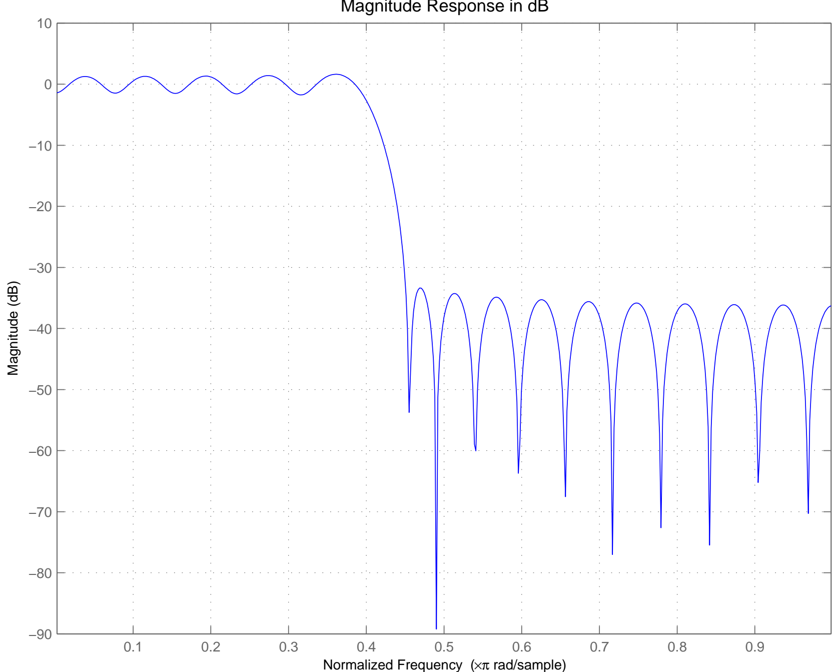
firlpnorm Examples

Review the following examples for an overview of the capabilities of the function—each example uses one or more features provided by `firlpnorm` and the least-Pth unconstrained optimization algorithm. Among the filter designs you can create are filters with arbitrarily defined magnitude response or minimum phase.

Example—Design a Lowpass Filter With pmin = 4 and pmax = 12

With the filter specifications in this example, the result is a quasi-equiripple response lowpass filter. You can see from the plot that follows the code the shape of the magnitude response.

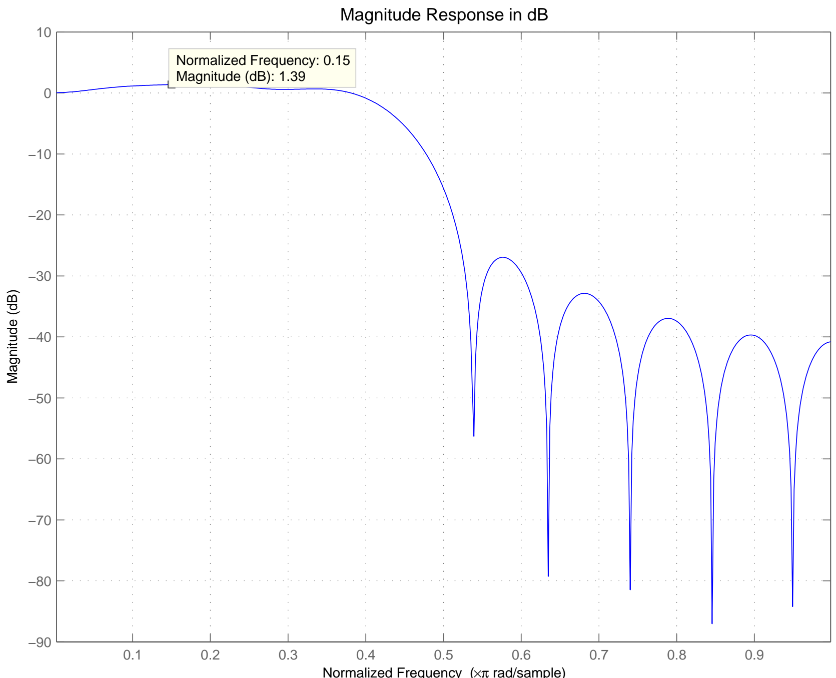
```
b=firlpnorm(30,[0 0.4 0.45 1],[0 0.4 0.45 1],[1 1 0 0],...
[1 1 10 10],[4 12]);
[H,W,S]=freqz(b,1,1024);
S.plot = 'mag';
fvtool(b);
```



Example—Design a Lowpass Least-Squares Filter With a "Peak" In The Passband

Using the appropriate set of input arguments, you can add a slight peak in the passband of the filter. The following code creates a lowpass filter that demonstrates just such tweaking of its passband to add gain. Notice the set of inputs for a (the specification of the passband response) `[1 1.2 1 0 0]` in the calling syntax. The `1.2` raises the passband response at the `0.15` normalized frequency point defined in `f`.

```
b=firlpnorm(15, [0 0.15 0.4 0.5 1], [0 0.4 0.5 1],...
[1 1.2 1 0 0],[2 2 2 1 1], [2 2]);
fvtool(b)
```

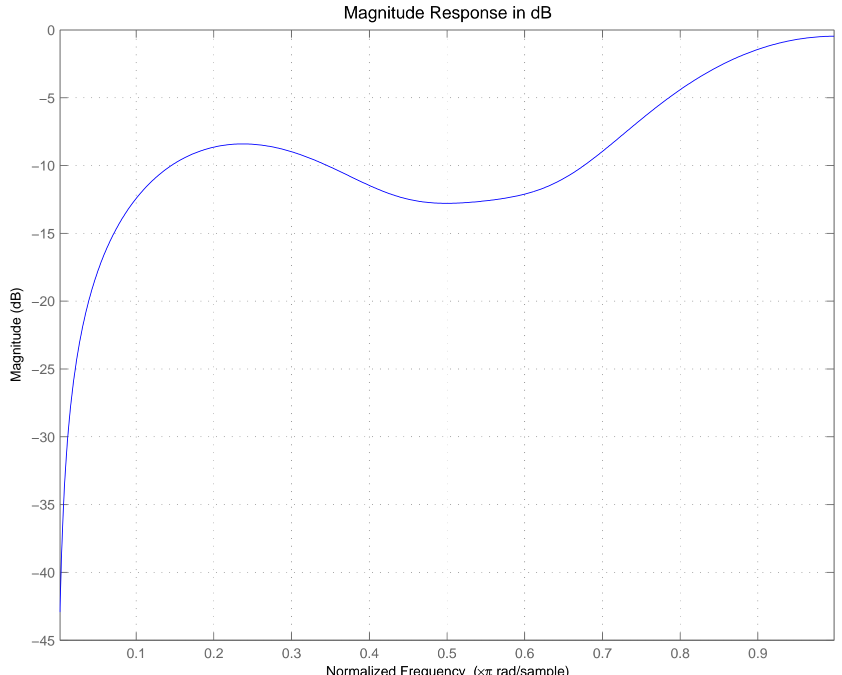


Example—Create a Low-order Filter With One Band

By using the weighting input arguments and the pmin and pmax options, this example creates a low order, n = 5, FIR filter with one band. When you define pmin and pmax as 2 and 16, the optimization algorithm starts at pmin = 2 and continues to optimize in the filter in the pmax sense. By default, pmin and pmax are 2 and 128, achieving the L-infinity or Chebyshev norms.

```
b=firlpnorm(5, [0 .2 .6  1], [0 1], [0 .4 .2  1], [1 1 1 1],...
[2 16]);
fvtool(b)
```

Reviewing the figure from FVTool shows the single band nature of the filter response.

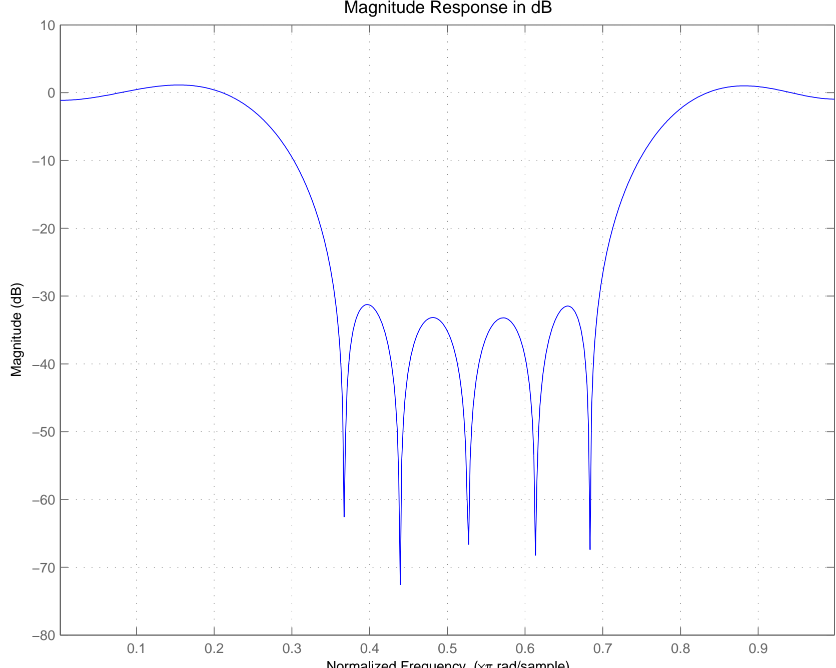


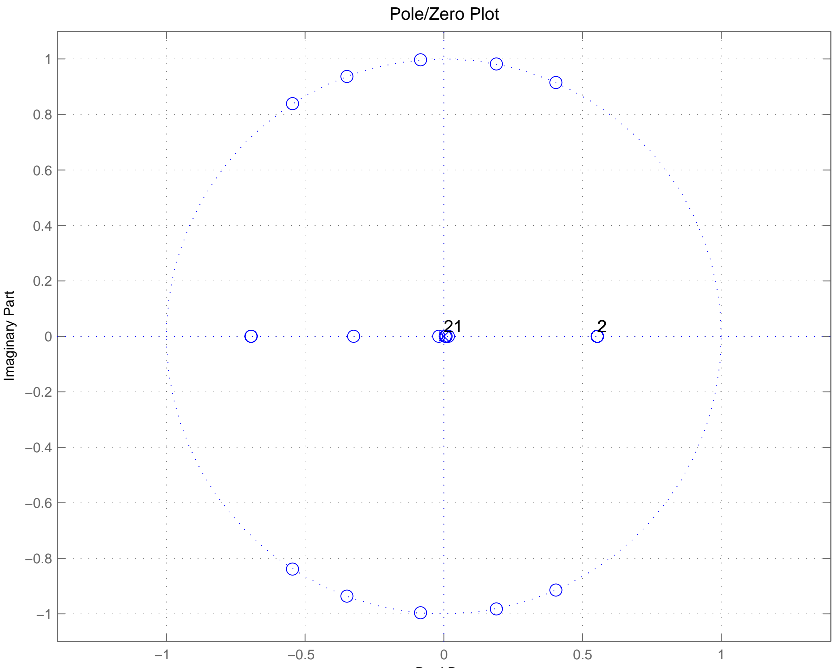
Example—Return a Minimum Phase Bandstop Filter

To generate a minimum phase filter, `firlpnorm` uses the '`minphase`' optional input argument. For this example of creating a bandstop filter, `p = [2 4]` and the filter order is set to `21`. Notice that weight vector `w` emphasizes the error in the stopband region by using `[1 1 5 5 1 1]`. Combined with the a vector of `[1 1 0 0 1 1]`, the result is a bandstop filter, as shown in the figure that follows the code for designing the filter.

```
b=firlpnorm(21, [0 .25 .35 .7 .8 1], [0 .25 .35 .7 .8 1],...
[1 1 0 0 1 1], [1 1 5 5 1 1], [2 4], 'minphase');
fvtool(b)
```

Plotting the zeros on the unit circle shows the minimum phase nature of the filter.




Pole/Zero Plot
Imaginary Part
Real Part
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
21
2

Advanced IIR Filter Designs

Many digital filters use both input values and previous output values from the filter to calculate the current output value. FIR filters can be implemented with feedback, although this is unusual. Cascaded integrated comb filters are one example.

For IIR filters, the transfer function is a ratio of polynomials:

- The numerator of the transfer function. When this expression falls to zero, the value of the transfer function is zero as well. Called a zero of the function.
- The denominator of the transfer function. When this expression goes to zero (division by zero), the value of the transfer function tends to infinity; called a pole of the function or filter.

Filter Design Toolbox introduces three functions: `iirlpnorm`, `iirlpnormc`, and `iirgrpdelay` for designing IIR filters that design optimal solutions to your filter requirements. With these new filter functions, you can design filters to meet your specifications that you could not design using the IIR filter design functions in Signal Processing Toolbox.

Function `iirlpnorm` uses a least-pth norm unconstrained optimization algorithm to design IIR filters that have arbitrary shape magnitude response curves. `iirlpnormc` uses a least-pth norm optimization algorithm as well, only this version is constrained to let you restrict the radius of the poles of the IIR filter.

To let you design allpass IIR filters that meet a prescribed group delay specification, `iirgrpdelay` uses a least-pth constrained optimization algorithm. For basic information about the least-pth algorithms used in the IIR filter design functions, refer to *Digital Filters* by Antoniou [1].

This section uses examples to introduce the IIR filter design functions in the toolbox. As you review these examples, you may notice that the IIR design functions use the same syntax, input, and output arguments. Because the design functions use very similar algorithms, common input and output arguments apply. Arguments are used in the same way, and carry the same defaults and restrictions. That said, if an example of one IIR function uses a syntax that does not appear under another IIR design function, chances are you can use the first syntax in the other design function as well.

Examples — Using Filter Design Toolbox Functions to Design IIR Filters

Filter Design Toolbox provides new capabilities for IIR filter design. Because of the comprehensive nature of the new IIR design functions, learning by example is the best way to discover what you can do with them. This section presents a series of examples that investigate the filters you can implement through IIR filter design in Filter Design Toolbox. You can view these examples as a demonstration program in MATLAB by opening the MATLAB demos and selecting `Filter Design` from `Toolboxes`. Listed there you see a number of demonstration programs. Select one of the following demos to see the IIR filter design functions being used to design a variety of filters:

- Least P-norm Optimal IIR Filter Design demonstrates IIR filter design function `iirlpnorm`. Examples include:
  - "Example — Using iirlpnorm to Design a Lowpass Filter" on page 2-45
  - "Example — Using iirlpnorm to Design a Low Order Filter" on page 2-46
  - "Example — Using iirlpnorm to Design a Bandstop Filter" on page 2-47
  - "Example — Using iirlpnorm to Design a Noise-Shaping Filter" on page 2-49
- Constrained Least P-norm IIR Filter Design demonstrates IIR filter design function `iirlpnormc`. This set of examples includes:
  - "Example — Using iirlpnormc to Design a Lowpass Filter" on page 2-50
  - "Example — Using iirlpnormc to Design a Bandstop Filter with a Constrained Pole Radius" on page 2-52
  - "Example — Using iirlpnormc to Design a High-Order Notch Filter" on page 2-53
  - "Example — Using iirlpnormc to Change an Elliptic Filter to a Constrained Lowpass Filter" on page 2-54
- `IIR Filter Design Given a Prescribed Group Delay` demonstrates IIR filter design function `iirgrpdelay`. These examples include:
  - "Example — Using iirgrpdelay to Design a Filter with a User-Specified Group Delay Contour" on page 2-57
  - "Example — Using iirgrpdelay to Design a Lowpass Elliptic Filter with Equalized Group Delay" on page 2-59

To Open the IIR Filter Design Demos

Follow these steps to open the IIR filter design demos:

1. Start MATLAB.
2. At the MATLAB prompt, enter `demos`.

   The **MATLAB Demo Window** dialog opens.
3. On the list on the left, double-click `Toolboxes` to expand the directory tree.

   You see a list of the toolbox demonstration programs available in MATLAB.
4. Select `Filter Design`.
5. From the list on the right, select one of the following demonstration programs:
   - `Least P-norm Optimal IIR Filter Design`
   - `Constrained Least P-norm IIR Filter Design`
   - `IIR Filter Design Given a Prescribed Group Delay`

A few examples include comparisons to other filter design functions and analysis notes. For details about using the IIR design functions `iirlpnorm`, `iirlpnormc`, and `iirgrpdelay`, refer to Chapter 13, "Function Reference." While this set of examples covers many of the options for the functions, more options exist that do not appear in these examples. Examples cover common or interesting IIR design options to highlight some of the capabilities of the design functions.

In these examples, you can see that `iirlpnorm`, `iirlpnormc`, and `iirgrpdelay` use many of the input arguments used by `gremez`, plus others such as the denominator order. At the most basic level, each IIR filter design function uses the input arguments `N`, `D`, `F`, `Edges`, and `A` — the filter order for the numerator and denominator (so you can specify different order numerators and denominators), the vector containing the filter cutoff frequencies, the band edge frequencies, and the filter response at each frequency point. `F` and `A` must have matching numbers of elements; they can exceed the number of elements in `Edges`. You use this feature to specify a gain contour within a band defined by the entries in `Edges`. Every frequency that appears in `Edges` must also be an element of `F`. Also, the first band edge must equal the first frequency and the last band edge must equal the last frequency in `F`.

iirlpnorm Examples

Each of these examples uses one or more feature provided in the function iirlpnorm. The examples build on one another, although they can be run separately. Review each example to get an overview of the capabilities of the function.

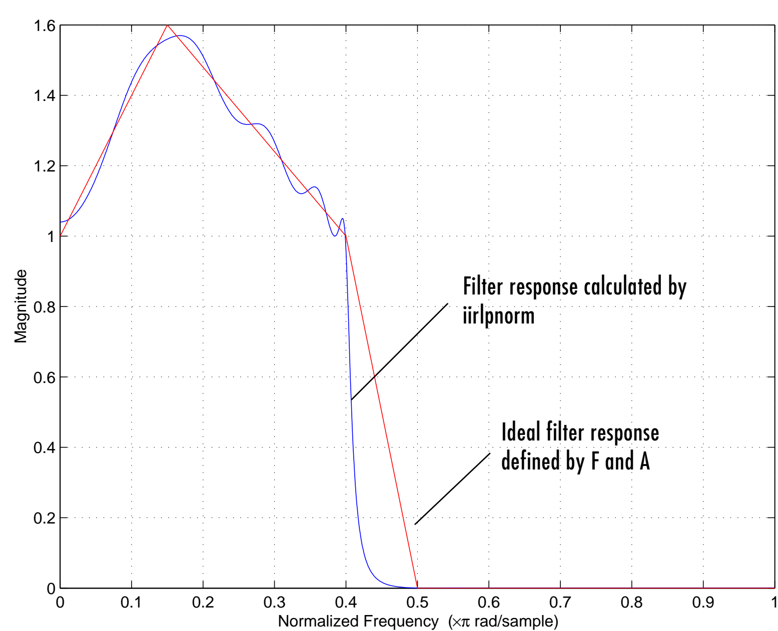
Example — Using iirlpnorm to Design a Lowpass Filter

To design a lowpass filter with maximum gain of 1.6 in the passband, we use the syntax iirlpnorm(n,d,f,edges,a,w). To duplicate the filter in the figure, use this code.

```
[b,a]=iirlpnorm(3, 11, [0 0.15 0.4 0.5 1], [0 0.4 0.5 1],...
[1 1.6 1 0 0], [1 1 1 100 100]);
[h,w,s]=freqz(b,a,1024);
s.plot = 'mag'; s.yunits = 'linear';
freqzplot(h,w,s);
hold on; plot([0 0.15 0.4 0.5 1], [1 1.6 1 0 0], 'r'); hold off;
```

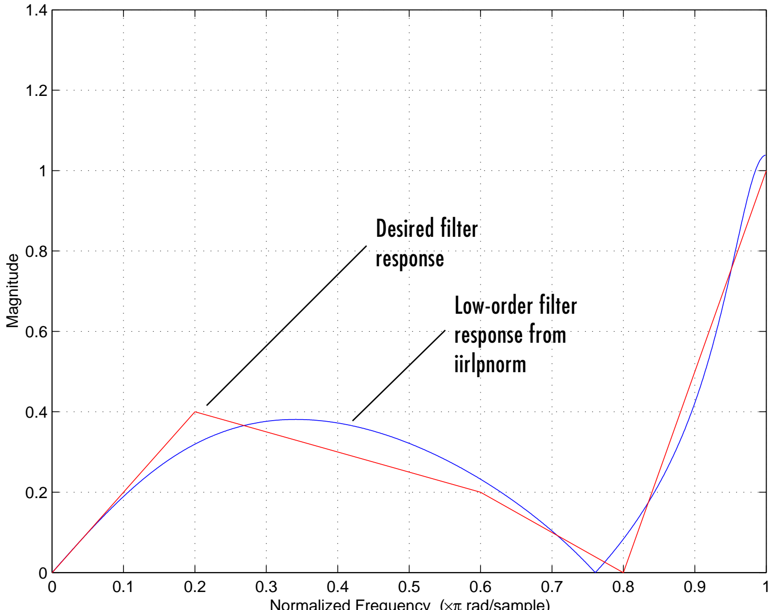
When you look at the magnitude response curve, notice the response reaches 1.6 in the passband.

s



Example — Using iirlpnorm to Design a Low Order Filter

The curves in the next figure show the results of using `iirlpnorm` to design a low-order filter with a single band. For this design, we introduce a new two-element vector `P=[Pmin,Pmax]` that defines the minimum and maximum values of `P` in the least-pth norm algorithm. If you do not specify `P`, the default values are `[2 128]`, resulting in the L_∞ or Chebyshev norm. Specify `Pmin` and `Pmax` to be even numbers. To view the placement of the poles and zeros for your filter before the optimization takes place, replace `[Pmin Pmax]` with the string `'inspect'`. With the option `'inspect'` in use, the algorithm does not optimize the filter design.



We specified a lowpass filter with third-order numerator and denominator, and used the P vector to limit the optimization range, by using the function syntax `iirlpnorm(n,d,f,edges,a,w,p)`.

```
[b,a]=iirlpnorm(3, 3, [0 .2 .6 .8 1], [0 1], [0 .4 .2 0 1],...
[1 1 1 1 1], [2 64]);
[h,w,s]=freqz(b,a,1024);
s.plot = 'mag'; s.yunits = 'linear';
freqzplot(h,w,s);
hold on; plot([0 .2 .6 .8 1], [0 .4 .2 0 1], 'r'); hold off;
```

Setting `W=[1 1 1 1 1]` is the same as not setting weight values.

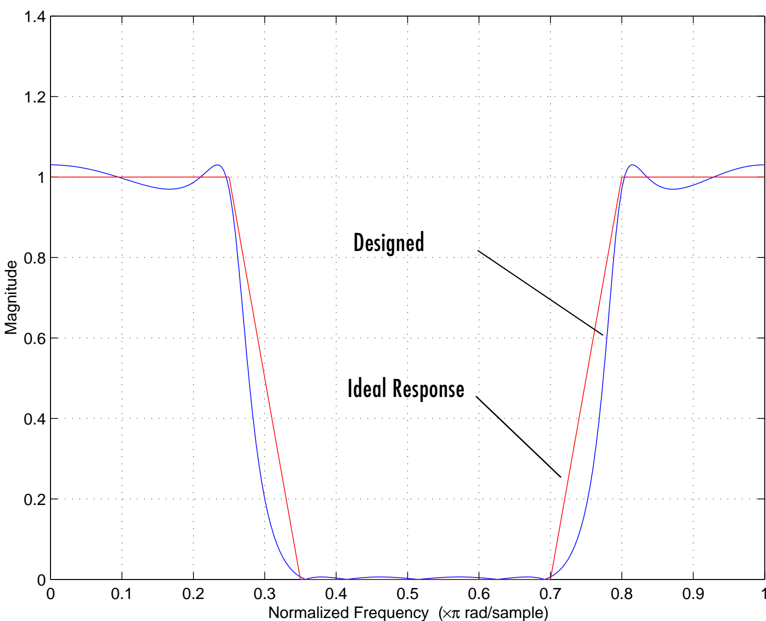
Example — Using iirlpnorm to Design a Bandstop Filter

Designing IIR bandstop filters is straightforward. Enter the frequency, magnitude, edges, and weight vectors using the syntax `iirlpnorm(n,d,f,edges,a,w)` as shown here. To ensure that the stopband

rejects undesired frequencies aggressively, we weight the magnitude response in the stopband more heavily by entering the weight vector [1 1 5 5 1 1], telling the optimization algorithm that meeting the inband response specification is five times as important as meeting the out-of-band response.

```
[b,a]=iirlpnorm(10, 7, [0 .25 .35 .7 .8 1],...
[0 .25 .35 .7 .8 1], [1 1 0 0 1 1], [1 1 5 5 1 1]);
[h,w,s]=freqz(b,a,1024);
s.plot = 'mag'; s.yunits = 'linear';
freqzplot(h,w,s);
hold on; plot([0 .25 .35 .7 .8 1], [1 1 0 0 1 1], 'r'); hold off;
```

As you can see from the following figure, the filter meets our design needs quite closely.

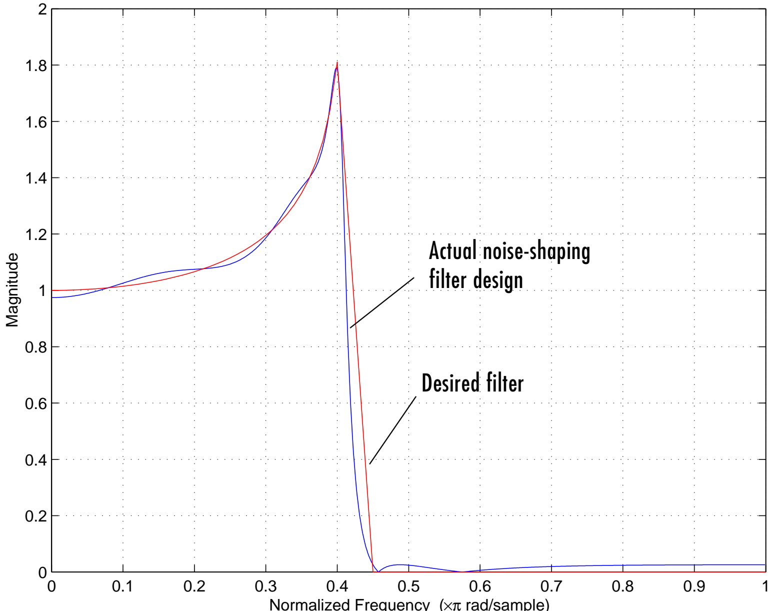


Example — Using iirlpnorm to Design a Noise-Shaping Filter

In this example, we create a lowpass filter with a rising magnitude in the passband. Communications designers use the filter when they simulate the effects of motion between a transmitter and receiver, such as you find in cellular telephone networks. Here, we use `iirlpnorm` to design the same filter. Because of the complex shape of the passband, we define the vectors `f`, `a`, `w`, and `edges` in the workspace, then use the vector names in the `iirlpnorm` statement.

```
f = 0:0.01:0.4;
a = 1.0 ./ (1 - (f./0.42).^2).^0.25;
f = [f 0.45 1];
a = [a 0 0];
edges = [0 0.4 0.45 1];
w = ones(1, length(a));
[b,a]=iirlpnorm(4, 6, f, edges, a, w);
[h,w,s]=freqz(b,a,1024);
s.plot = 'mag'; s.yunits = 'linear';
freqzplot(h,w,s);
hold on; plot(F,A, 'r'); hold off;
```

When you compare the figure below to the filter design in “Getting Started with the Toolbox” on page 1-15, you see they match very well.



iirlpnormc Examples

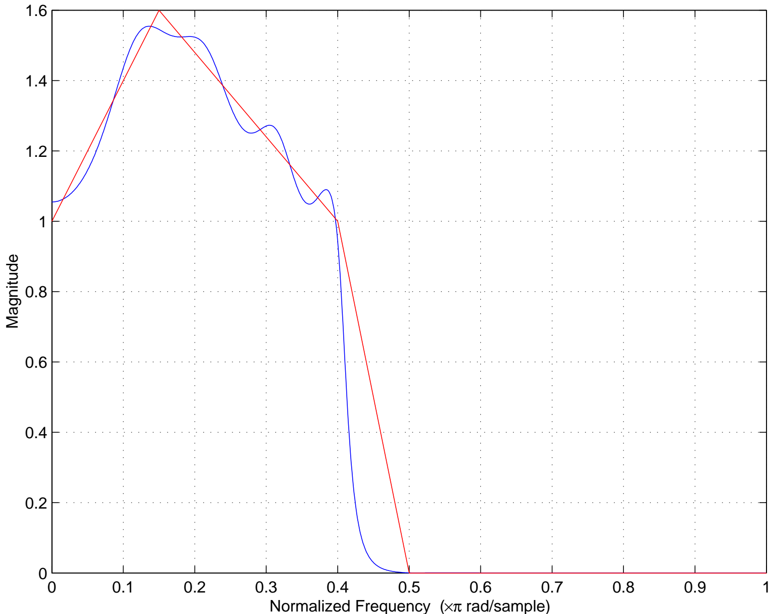
Each of these examples uses one or more feature provided in the function `iirlpnormc`. Review each example to get an overview of the capabilities of the function.

Example — Using iirlpnormc to Design a Lowpass Filter

Just as you use `iirlpnorm` to design lowpass filters, you can use `iirlpnormc` to design them as well. `iirlpnormc` lets you limit the radius of the filter poles when you specify the filter in the function. By restricting the poles to be less than a certain distance from the origin of the unit circle in the z-plane, the filter remains stable, while possibly improving the robustness of the filter to quantization effects. In this lowpass filter example, we restrict the pole radius not to exceed 0.95, using the function syntax `iirlpnormc(n,d,f,edges,a,w,radius)`.

```
[b,a]=iirlpnormc(3, 11, [0 0.15 0.4 0.5 1], [0 0.4 0.5 1],...
[1 1.6 1 0 0], [1 1 1 100 100], 0.95);
[h,w,s]=freqz(b,a,1024);
s.plot = 'mag'; s.yunits = 'linear';
freqzplot(h,w,s);
hold on; plot([0 0.15 0.4 0.5 1], [1 1.6 1 0 0], 'r'); hold off;
```

`radius` takes values between 0 and 1.



Compared to the unconstrained `iirlpnorm` lowpass filter example (refer to "iirlpnorm Examples" on page 2-45), you see that the filter performance is about the same, although the ripple in the passband is slightly greater, and the transition somewhat sharper. The difference between these two designs is the constraint applied to the poles when you use `iirlpnormc` with a radius value. Both filters demonstrate peaks in their passband.

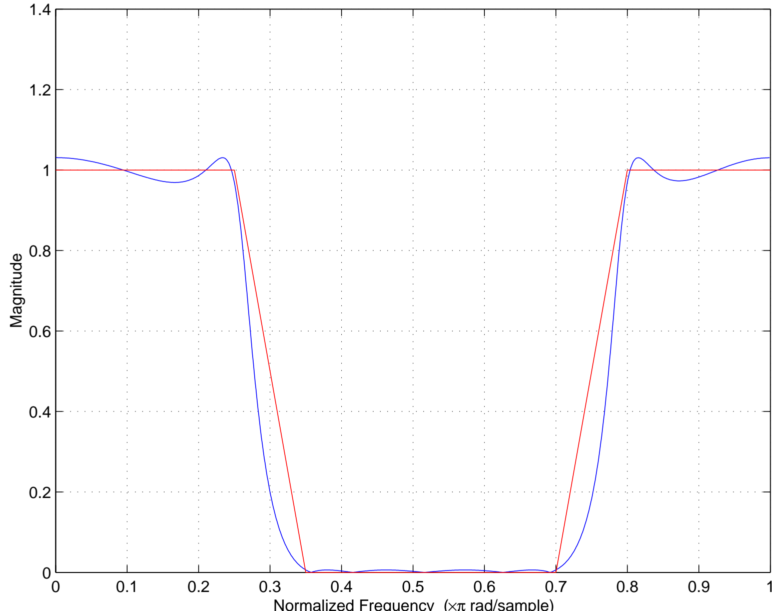
Example — Using iirlpnormc to Design a Bandstop Filter with a Constrained Pole Radius

Here we use `iirlpnormc` to design a bandstop filter. Notice that we specify different orders for the numerator (`n=10`) and denominator (`d=7`) and the frequency and edges vectors are the same. With `radius=.91`, none of the 11 filter poles lies farther than 0.91 away from the origin, as you can see in the zero-pole plot.

```
f = [0 .25 .35 .7 .8 1];
[b,a]=iirlpnormc(10, 7, f, f, [1 1 0 0 1 1], [1 1 5 5 1 1], .91);
[h,w,s]=freqz(b,a,1024);
s.plot = 'mag'; s.yunits = 'linear';
freqzplot(h,w,s);
hold on; plot([0 .25 .35 .7 .8 1], [1 1 0 0 1 1], 'r'); hold off;
```

To generate the zero-pole plot, use `zplane(b,a)` at the MATLAB prompt.

When we plot the magnitude response curve, the emphasis we placed on reducing the error in the stopband is clear — note the close match between the desired and calculated responses. (We weighted the magnitude response `w=[1 1 5 5 1 1]` to minimize the error in the vicinity of the stopband frequency points.)



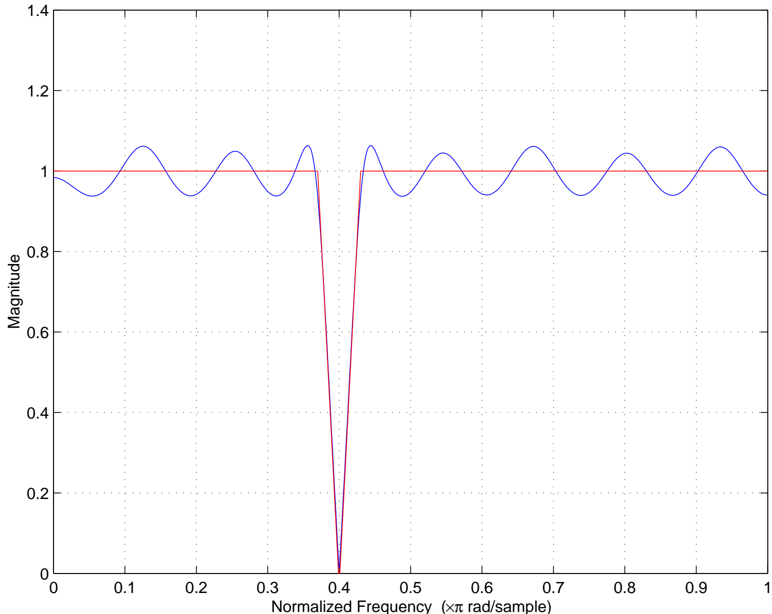
Example — Using iirlpnormc to Design a High-Order Notch Filter

To create an optimized design for an IIR high-order notch filter, use `iirlpnormc` to design the filter. The following code results in the optimal solution to creating a filter with different numerator and denominator orders, and with a maximum pole radius of `0.92`.

```
f = [0 0.37 0.399 0.401 0.43 1];
[b,a]=iirlpnormc(2, 17, f, f, [1 1 0 0 1 1], [1 1 2 2 1 1], 0.92);
[h,w,s]=freqz(b,a,1024);
s.plot = 'mag'; s.yunits = 'linear';
freqzplot(h,w,s);
hold on;
plot([0 0.37 0.399 0.401 0.43 1], [1 1 0 0 1 1], 'r'); hold off;
```

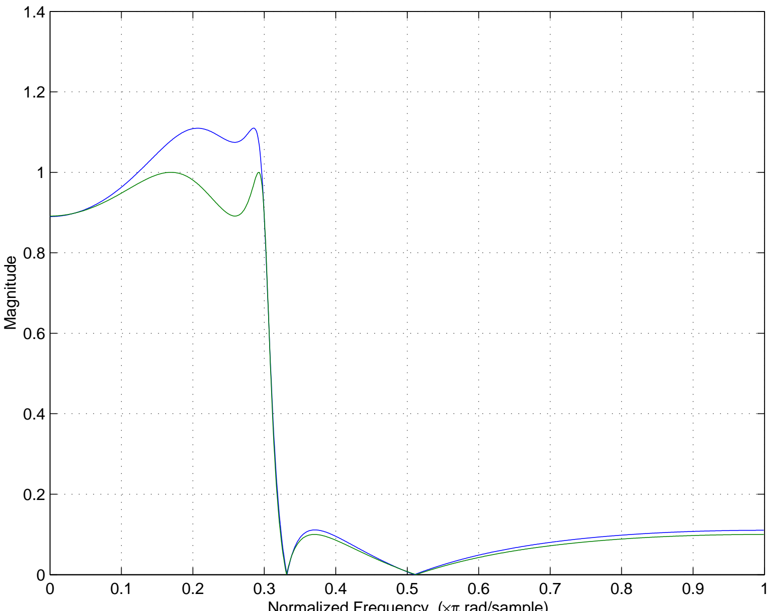
Note the frequency vector entries `0.37`, `0.399`, `0.401`, and `0.43`. These represent the cutoff points for the filter stopband, a fairly narrow filter. Looking at the filter response plot, you see it is similar to the single-point filter

example we designed with the gremez function (refer to “Example—Designing a Single-Point Band Filter” on page 2-18). This filter has two pairs of constrained poles.



Example — Using iirlpnormc to Change an Elliptic Filter to a Constrained Lowpass Filter

Using an elliptic filter design as the initial conditions, with a maximum pole radius of 0.96, we reduce the pole radius to 0.95 when we use iirlpnormc to create an optimal filter solution. The result is a filter with the same band edge frequencies, and a gain in the passband greater than one. The following code uses the function ellip from Signal Processing Toolbox to create an elliptical filter. Then we use the function iirlpnormc with the syntax iirlpnormc(n,d,f,edges,a,w,radius,p, dens,initnum,initden). initnum and initden are the initial estimates of the filter numerator and denominator coefficients. We use be and ae from our elliptic filter as the vectors Initnum and initden.




```
[be,ae]=ellip(4,1,20,0.3);
f = [0 0.3 0.323 1];
[b,a]=iirlpnormc(4, 4, f, f, [1 1 0 0], [1 1 1 1], .95,...
[128 128], 20, be, ae);
[h,w,s]=freqz(b,a,1024);
he=freqz(be,ae,1024);
s.plot = 'mag'; s.yunits = 'linear';
freqzplot([h he],w,s);
```

A few points to think about when you use `iirlpnormc`. These hints can help you converge on a good filter design:

- `iirlpnormc` implements a weighted, least-pth optimization algorithm.
- Check the location and radii of the designed filter poles and zeros.
- If the zeros are on the unit circle and the poles are well inside the circle, try increasing the numerator order `N`, or reducing the error weighting (`W`) in the stopband.

- If several poles have large radii, and the zeros are well inside the unit circle, try increasing D, the denominator order, or reducing the error weighting in the passband.
- As you reduce the pole radius, you may need to increase the denominator order.

iirgrpdelay Examples

Filter Design Toolbox provides a new filter design function iirgrpdelay for designing allpass IIR filters that have group delay characteristics that meet your needs. When you cascade these allpass filters with other IIR filters, they act as compensating elements. They produce equalized or specified group delay across the combined filter frequency response while maintaining the IIR filter pass and stop bands. For more information about group delay in filters, refer to “Signal Processing Basics” in *Signal Processing Toolbox User’s Guide*.

Note iirgrpdelay creates allpass filters you use to compensate for the phase changes caused by other filters. You cannot use iirgrpdelay to create filters that both filter input signals and compensate for phase changes in output signals.

In this section, we introduce the function iirgrpdelay through a series of examples. Each of these examples uses one or more feature provided in the function. The examples build on one another, although they can be run separately. By reviewing each example you get an overview of the capabilities of the design function.

In much the same way that you use other IIR filter design functions to create filters with arbitrary magnitude response curves, you use iirgrpdelay to create filters that have arbitrary group delay curves in the filter passband and stopband. (In most cases, specifying the group delay in the stopband is not useful; the filter rejects those frequencies by design. Nonetheless, you can specify the group delay for frequencies that fall within filter stopbands.)

To specify a filter that approximates a given relative group delay, use iirgrpdelay with the following input argument syntax

```
iirgrpdelay(N,F,Edges,Gd)
```

where N is the filter order, F is a vector containing frequencies between 0 and 1, Gd is a vector whose elements are the desired group delay at the frequencies specified in F, and Edges specifies the band edges. Filter order N must be an even number, and the vectors F and Gd must have the same number of elements. To let you specify the shape of the group delay within a band or bands, vectors F and Gd can contain more elements than Edges.

Considering the following ideas can help you design your group delay compensator:

- After you use iirgrpdelay to design a filter, use freqz, grpdelay, and zplane to check your design for undesirable features.
- Remember that allpass filters have positive group delay. You cannot develop allpass filters that have negative group delay characteristics.
- For some difficult filter optimization problems, use the iirgrpdelay syntax

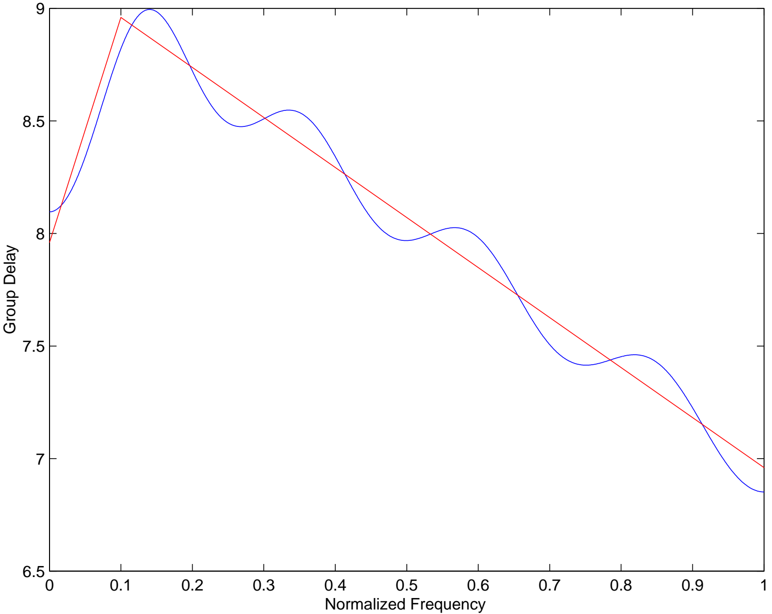
  ```
  iirgrpdelay(n,d,edgees,a,w,radius,p,dens,initden)
  ```

  where initden is a vector containing your estimates of the transfer function coefficients for the denominator. You can use the Pole-Zero editor in Signal Processing Toolbox to generate values for initden.
- If the poles and zeros of your filter design cluster together, you may need to increase the filter order or relax the pole radius restriction (if you used one).

Example — Using iirgrpdelay to Design a Filter with a User-Specified Group Delay Contour

To show the ability to create an arbitrary shape group delay contour in the passband of an IIR filter, we use iirgrpdelay and specify the group delay we desire. Notice that we also specify the maximum pole radius of 0.99. We plot the ideal group delay contour on the figure as well to compare the desired result to the designed filter.

```
[b,a,tau] = iirgrpdelay(8, [0 0.1 1], [0 1], [2 3 1],...
[1 1 1], 0.99);
[G,F] = grpdelay(b,a, 0:0.001:1, 2);
plot(F, G); hold on; plot([0 0.1 1], [2 3 1]+tau, 'r'); hold off;
```



The straight lines represent the desired group delay contour, the wavy line the designed contour. The desired group delay, `[2 3 1]`, is relative. Note that the actual group delay approximates `[8 9 7]`. If we increase the filter order, to 10 for example, the approximation improves, but the absolute group delay increases.

One of the output arguments for `iirgrpdelay` is `tau`, the resulting group delay offset. In all cases, filters created by `iirgrpdelay` have a group delay that approximates (`gd` + `tau`) where `gd` is the specified relative group delay of the filter.

When you look at the zero-pole plot for our filter (use the function `zplane`), you can see that the poles stay well within the radius constraint. Optimizing the filter may not result in poles that are near the constraint. Pole constraints come into play only when needed to limit the optimization. In this example, our design did not require the constraint to stay within the bounds of the unit circle.

You can verify that this is an allpass filter by plotting the magnitude response curve for the design. Use `freqz(b,a)` to plot the curve.

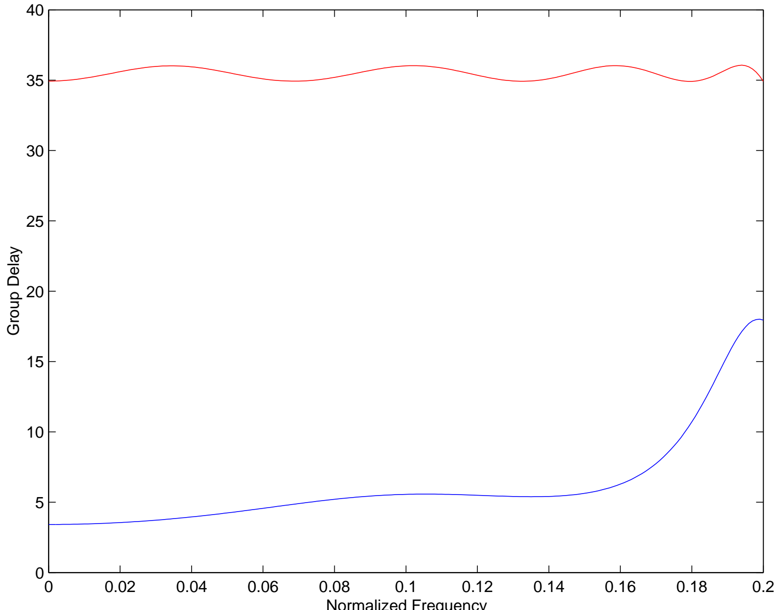
In general, you determine the contour to use for the group delay equalization of an IIR filter by subtracting the filter group delay from the filter maximum group delay. In the next example, we use this process to create our lowpass filter.

Example — Using iirgrpdelay to Design a Lowpass Elliptic Filter with Equalized Group Delay

The following code designs a pair of filters that together create a lowpass filter with equalized group delay.

```
[be,ae] = ellip(4,1,40,0.2); % Lowpass filter
f = 0:0.001:0.2;
g = grpdelay(be,ae,f,2);
g1 = max(g)-g;
[b,a,tau] = iirgrpdelay(8, f, [0 0.2], g1); % Phase compensator
gd = grpdelay(b,a,f,2);
plot(f, g); hold on; plot(f, g+gd, 'r'); hold off;
```

Cascading the filters is the same as adding the group delay for each filter frequency-point by frequency-point (`g+gd` in the `plot` function input arguments). In the figure, the lower curve is the group delay for the elliptic filter. The compensated, or equalized, group delay is the upper curve — an essentially flat group delay across the passband from 0 to 0.2. Since this example used the lowpass elliptic filter from our earlier `iirlpnorm` examples, you can see that combining these filters results in a lowpass filter with equalized group delay. Note that the group delay of the combination is twice the maximum group delay of the reference filter. When you use an allpass filter to equalize the group delay of a reference filter, the final group delay is the sum of the group delays of the reference and allpass filters.



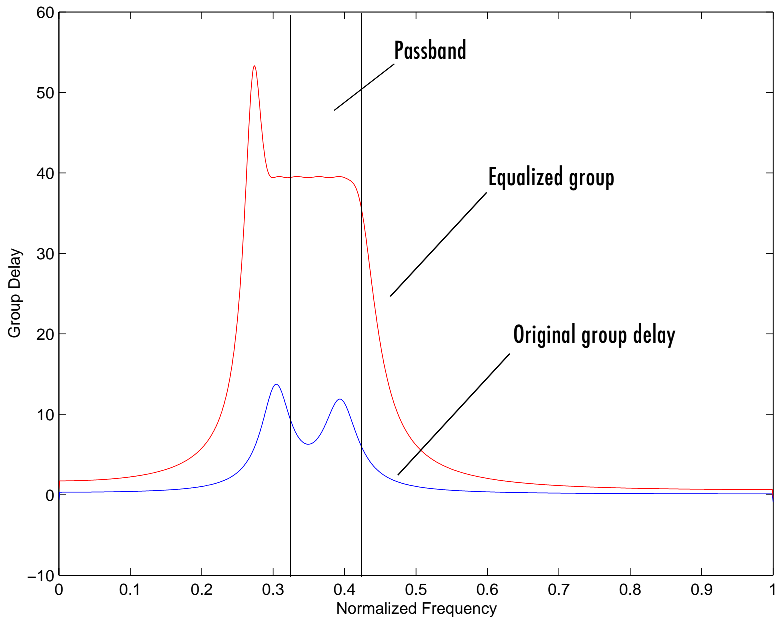
To determine the group delay contour necessary to compensate for the phase effects of our elliptic filter, we use the elliptic filter group delay as a reference.

In the example, we used `grpdelay` to return vector `g` containing the group delay value at many frequencies across the elliptic filter passband. After determining the maximum group delay in the elliptic filter passband (returned by `max(g)` in the example code), we subtract each individual group delay from the maximum group delay `(g1=max(g)-g)`. The result is vector `g1` containing values that define a curve that is the mirror image of the group delay contour of our elliptic filter. Then we use `g1` as the input group delay values to `iirgrpdelay`, and the resulting allpass filter has a group delay contour that equalizes the group delay of our lowpass elliptic filter, as shown in the figure.

Example — Demonstrating Passband Equalization for a Bandpass Chebyshev Filter

You can use `iirgrpdelay` to create filters that compensate for the group delay of many kinds of filters. In this example, we create an allpass filter that

equalizes the group delay of a bandpass filter. In the figure, the lower curve is the group delay of the bandpass filter and the upper curve is the equalized group delay for the combination of the bandpass filter and the allpass filter. Group delay variation across the passband is less than 0.2.



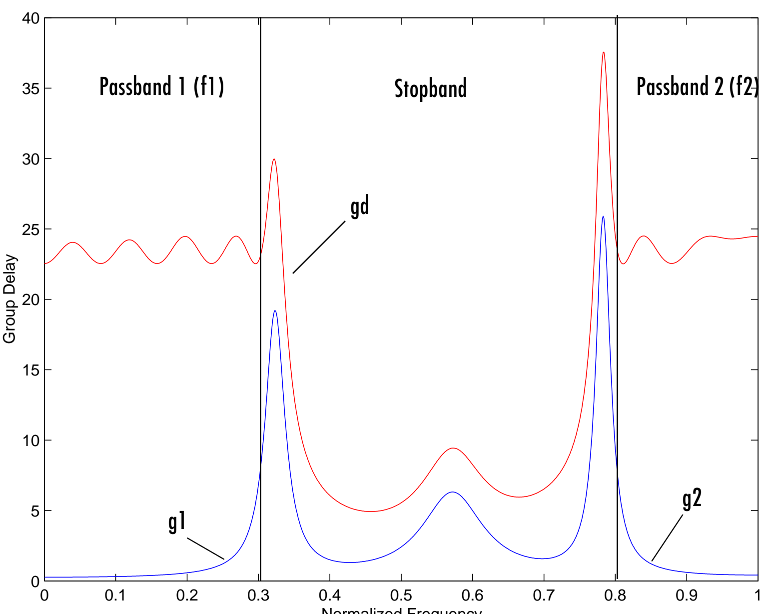

```
[bc,ac] = cheby1(2,1,[0.3 0.4]); % Bandpass filter design
f = 0.3:0.001:0.4;
g = grpdelay(bc,ac,f,2);
g1 = max(g)-g;
wt = ones(1, length(f));
[b,a,tau] = iirgrpdelay(8, f, [0.3 0.4], g1, wt, 0.95);
f = 0:0.001:1;
g = grpdelay(bc,ac,f,2);
gd = grpdelay(b,a,f,2);
plot(f, g); hold on; plot(f, g+gd, 'r'); hold off;
```

Example — Demonstrating Passband Equalization for a Bandstop Chebyshev Filter

Our final example shows how to equalize the group delay in the passband of a bandstop filter. Since this filter has two passbands, we equalize the group delay in each band according to the needs of each band. Vectors `g1` and `g2` in the example code contain the group delays within each passband of the bandpass filter. We ignore the stopband group delay for this case. To determine the group delay contour across both passbands, we concatenate `g1` and `g2` (using the command `g = [g1; g2]`), then use the vector `g` as the basis for the group delay input argument `gx` to `iirgrpdelay`.

```
[bc,ac] = cheby2(3,1,[0.3 0.8], 'stop'); % Bandstop filter
f1 = 0.0:0.001:0.3;
g1 = grpdelay(bc,ac,f1,2);
f2 = 0.8:0.001:1;
g2 = grpdelay(bc,ac,f2,2);
f = [f1 f2]; g = [g1; g2]; % Concatenate the passband group delays
gx = max(g)-g;
wt = ones(1, length(f));
[b,a,tau] = iirgrpdelay(14, f, [0 0.3 0.8 1], gx, wt, 0.95);
f = 0:0.001:1;
g = grpdelay(bc,ac,f,2);
gd = grpdelay(b,a,f,2);
plot(f, g); hold on; plot(f, g+gd, 'r'); hold off;
```

The figure shows that our approach works. You see that the group delay in the passbands is well-equalized (illustrated by the upper curve; the lower curve presents the nonequalized group delay). The stop band is unaffected, and the overall equalized group delay variation in the passbands is close to a constant.


Passband 1 (f1)
Stopband
Passband 2 (f2)
gd
g1
g2
Group Delay
Normalized Frequency
0
5
10
15
20
25
30
35
40
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

Robust Filter Architectures

We have been considering FIR and IIR filters whose transfer function is represented by constant coefficients and where the input signals and coefficients can be any double-precision value from -∞ to +∞. These systems are in the discrete time domain, with infinite precision values for the dependent variable, often magnitude.

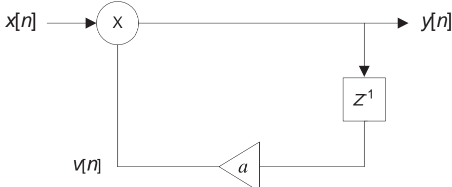
When you represent filters in software, or in general purpose or special purpose computing hardware, the inputs to the filters and the filter coefficients can be represented only by discrete values. The process of converting the infinite precision variables to discrete values is called *quantization* and represents a source of error when you implement digital filters.

Converting to the discrete domain produces three sources of errors:

- Error caused by the discrete representation of infinitely precise information, such as filter transfer function coefficients or signal amplitude values. Real systems create error when they quantize amplitude values.
- Analog-to-digital conversion error in the input signal.
- Arithmetic round off errors caused by the limited word length available to represent the data in the arithmetic process.

Transfer Function Coefficient Quantization Error

To illustrate the effects of converting from continuous to discrete representations, and to show error sources resulting from quantization, consider the following first-order IIR filter.



The constant coefficient difference equation that defines this filter is

$$y[n] = \alpha y[n-1] + x[n]$$

where $y[n]$ and $x[n]$ are the output and input signal variables. In transfer function form, the following equation describes our IIR filter.

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

When you implement this filter form in hardware, the filter coefficient α assumes discrete values that approximate the design value. Therefore, the actual transfer function that you implement is

$$\hat{H}(z) = \frac{z}{z - \hat{\alpha}}$$

where $\hat{H}$ and $\hat{\alpha}$ are the close approximations to the original H and α in the filter design. Notice that this transfer function differs from the theoretical function $H(z)$. As a result, the actual filter response can differ substantially from the ideal response.

The main effect of transfer function coefficient quantization is to move the poles and zeros to different locations in the z-plane, away from their desired, or designed locations (the locations for the ideal, nonquantized coefficient filter). Moving the poles or zeros can have two effects:

- Changing the frequency response of the quantized filter so it is not the same as the ideal or designed filter.
- Moving poles from inside to outside the unit circle, causing the quantized IIR filter to be unstable. Applies only to IIR filters.

Input Sampling Error (A/D Error)

Given the difference equation for our IIR filter, from earlier

$$y[n] = \alpha y[n-1] + x[n]$$

where $x[n]$ is the sampled output from an analog to digital converter. Sampling the continuous signal $x_a(t)$ results in $x[n]$. Then the sampled input to the filter from the A/D convertor, $\hat{x}[n]$, is

$$\hat{x}[n] = x[n] + e[n]$$

and $e[n]$ is the error in the A/D conversion process. Our discrete input to the filter no longer matches the continuous signal $x_a(t)$. Discrete-time input $x_a(t)$ does not match $x[n]$ because analog-to-digital conversion made the input

discrete in time. Similarly, quantized input $\hat{x}[n]$ does not match $x[n]$ because it has been convert to discrete data in amplitude.

Arithmetic Quantization Error

Quantization in arithmetic operations causes another error. For our first-order filter example, the output from our multiplier $v[n]$ is generated by multiplying the signal, $y[n\text{-}1]$ with the transfer function coefficients, α

$$v[n] = \alpha y[n-1]$$

and storing the result. When we quantize the result to fit it into a storage register, we generate a quantized value $\hat{v}[n]$ that we write as

$$v[n] = v[n] + e_{\alpha}[n]$$

where $e_{\alpha}[n]$ is the error sequence resulting from the product quantization process.

Limitcycles and Arithmetic Quantization

There is another source of errors in digital filter implementation, caused by the nonlinearity of quantized arithmetic operations. These errors are apparent in an effect called limit cycling that occurs at the filter output. Limit cycles usually appear when there is no input to the filter, or the input to the filter is constant or sinusoidal. For more information on limit cycles and the function `limitcycle`, refer to the `limitcycle` reference page in the online documentation. To learn more about quantization, refer to Chapter 5, "Quantization and Quantized Filtering."

Low Sensitivity Filter Architectures

Quantizing filter coefficients can have serious effects on the performance of digital filters. As a result of coefficient quantization, the frequency response of the filter with quantized coefficients can be significantly different from the desired filter without quantized coefficients. In some cases, the performance of the quantized filter can make it unsuitable for your application.

Low sensitivity filter architectures, or robust architectures as they are sometimes called, are interesting because they can reduce the effects of coefficient quantization. By being inherently less sensitive to coefficient quantization, these filter architectures withstand the quantization process and result in filters that retain the performance of the original filter.

Approaches to Designing Low Sensitivity Filters

Consider either of two approaches to designing low sensitivity filters:

- Convert low sensitivity analog filters composed of inductors, capacitors, and resistors to digital architectures by replacing the analog components and connections with their digital equivalents so the digital filter approximates the analog version.
- Develop digital filter implementations that respond directly to the conditions that create low coefficient sensitivity in a digital filter designs.

Filter Design Toolbox uses the latter approach to provide low sensitivity filter architectures.

Generally, filter architecture sensitivity ranges from high for direct forms to very low for coupled allpass forms. For reference, the following list ranks the filter forms in the toolbox by their sensitivity to coefficient quantization, from high sensitivity to low:

1. Direct forms—often very sensitive to quantization
2. Lattice forms—moderately sensitive to quantization
3. Allpass forms—quite robust under quantization

Quantization sensitivity is also a function of the locations of the poles and zeros for a filter, so use this list for guidance only.

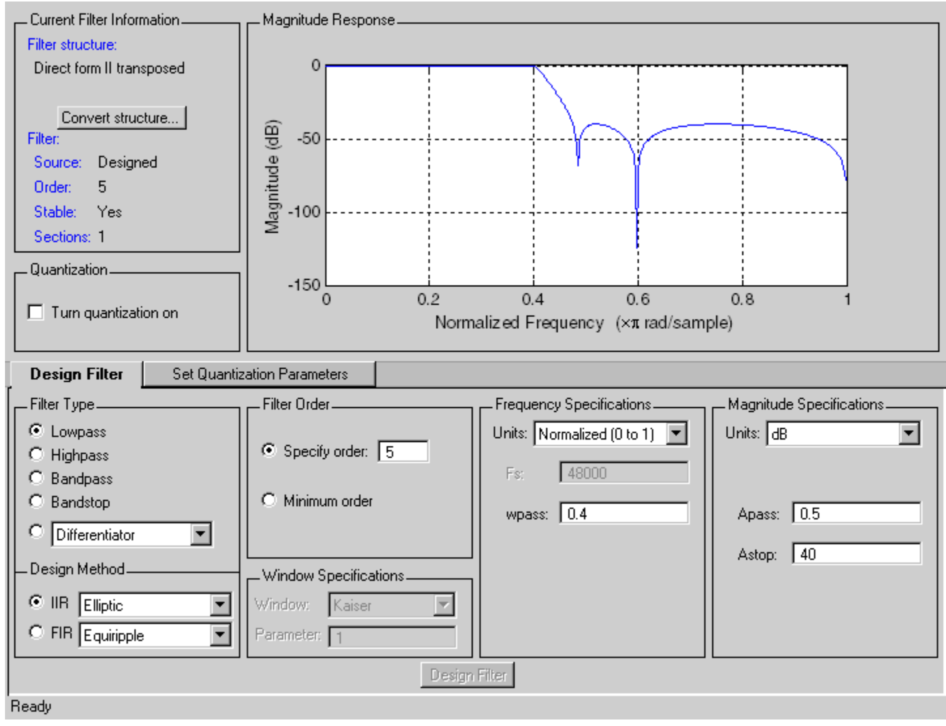
Within the forms

- FIR filters tend to be less sensitive than IIR filters
- For the direct forms, second-order section implementations are often less sensitive to coefficient quantization

Filter Design Example That Includes Quantization

To demonstrate the effects of coefficient quantization on the performance of a filter, this example creates a 5th-order, lowpass elliptic IIR filter. We choose a cutoff frequency of 0.4π radians (normalized frequency from 0 to 1), passband ripple less than 0.5 dB, and stopband attenuation of at least 40 dB. In the figure you see the filter response. We used the Filter Design and Analysis tool (FDATool) to design the filter. Notice that we used the default filter structure

`df2t`, or Direct form 2 transposed. When we want to compare the quantized version of the filter to the floating-point filter, FDATool lets us quantize the filter and display the filter response curves together.



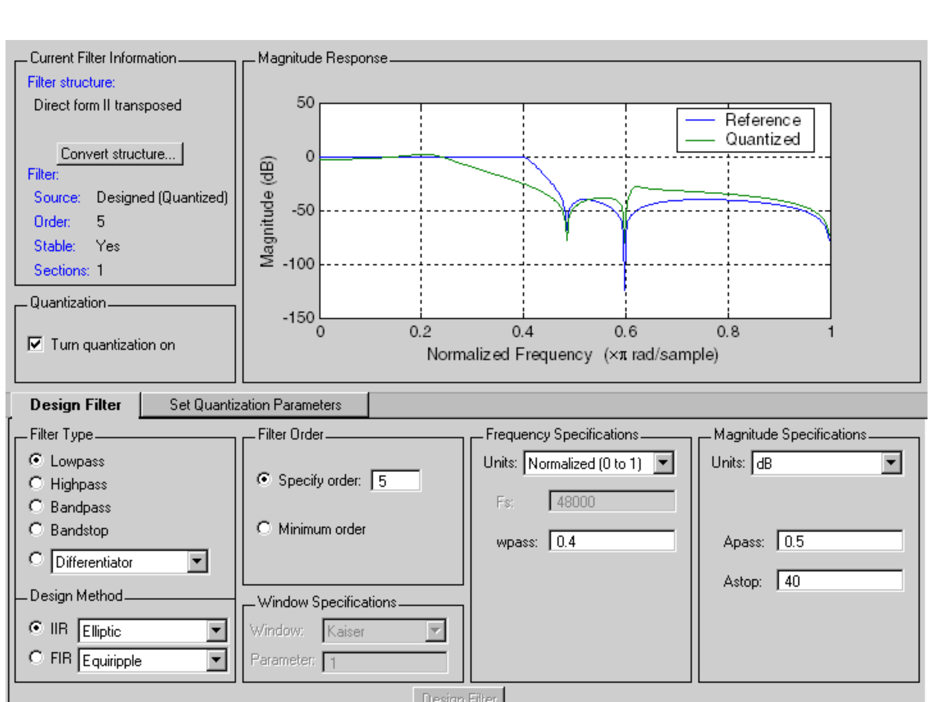
We could have used the function `ellip` from Signal Processing Toolbox to create the filter.

```
[b,a] = ellip(5,0.5,40,0.4);
```

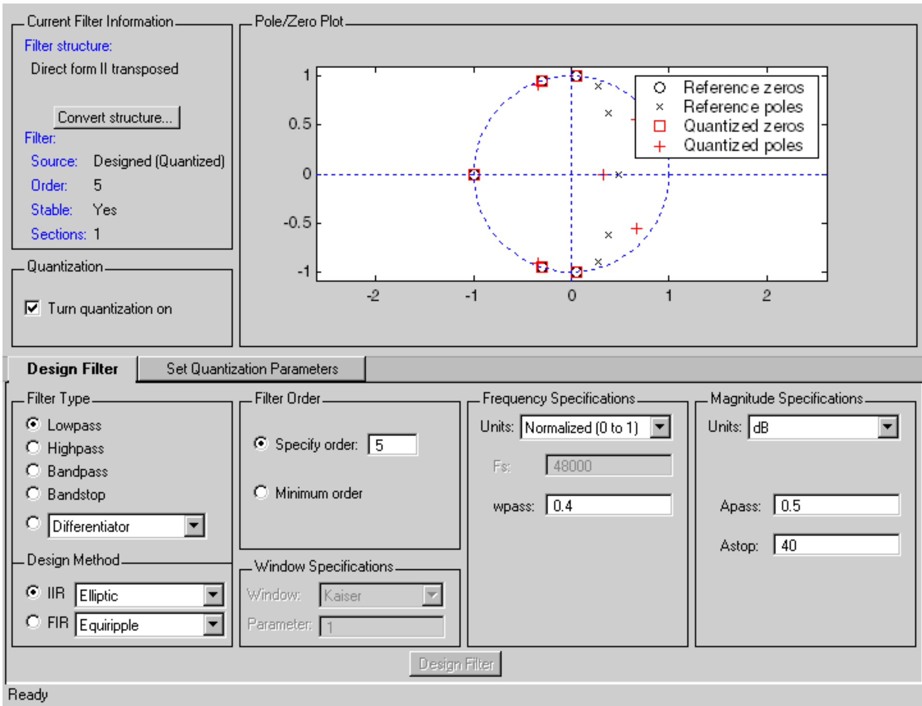
The results are identical because FDATool uses the same function to design the lowpass filter.

We quantize the filter by selecting **Turn quantization on**. FDATool quantizes our elliptic filter and displays the magnitude response for both the original (or reference) filter and the quantized filter. For this quantization process we use the default coefficient format settings in FDATool. Later in this example we

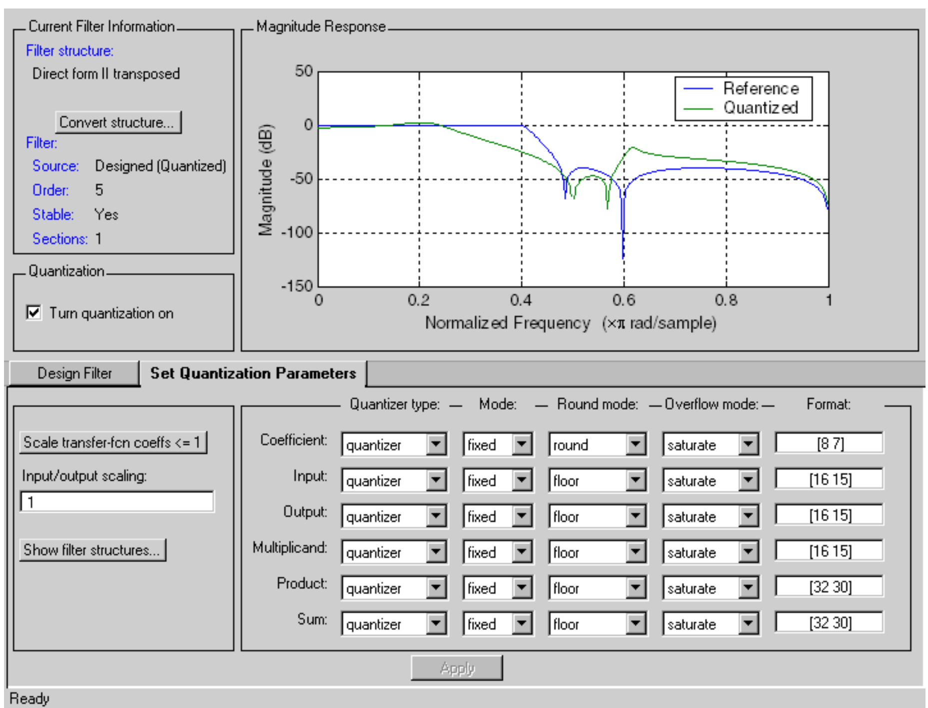
change the coefficient format to illustrate the effects of changing the word length used to represent the filter coefficients.



Quantizing the coefficients has damaged our filter magnitude response. Our quantized filter transition band starts much earlier and is much shallower, and the stopband attenuation has been reduced. When we look at the zero-pole plot for the unquantized and quantized versions of our filter, we see that quantization has moved the poles from their designed locations. Coefficient overflow, rather than sensitivity to quantization, caused the terrible quantized response in this filter. Coefficient quantization changes filter coefficients by at most one quantization level. Overflow can change the coefficients by an arbitrarily large amount. In this case, quantization changed the largest magnitude coefficient from 2.49 to saturation at 1.0. You can see this from the coefficient view by selecting **Analysis -> View Filter Coefficients**. Thus we see how sensitive this direct form IIR filter is to coefficient quantization.



To continue this example, we look at the effects of changing the coefficient format from fixed-point, 16-bits to fixed-point, 8-bits. After we make the desired change, we see the response curves shown in this figure.

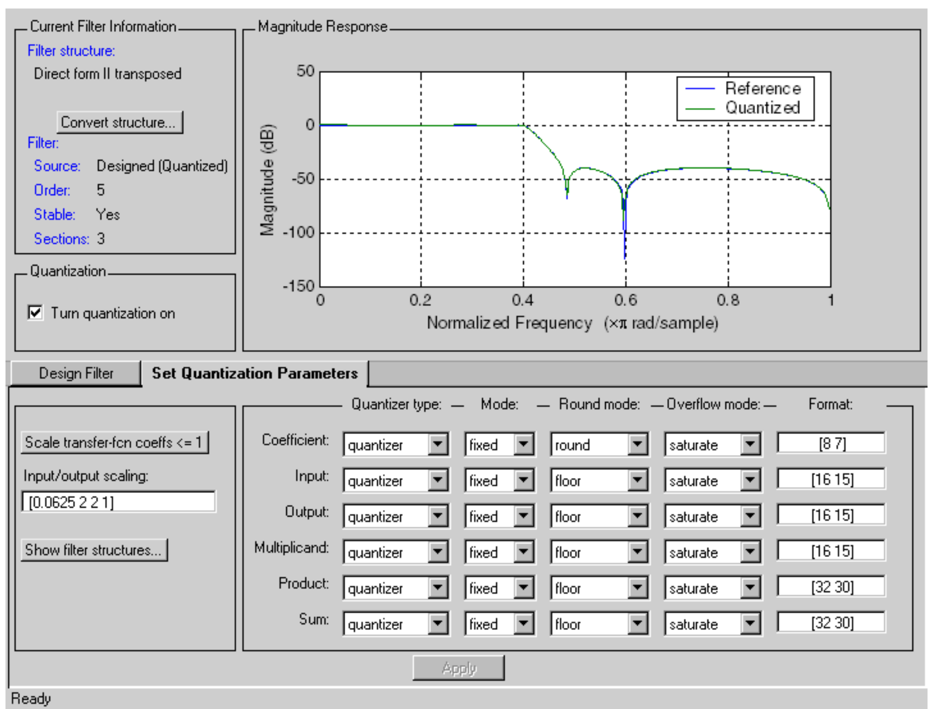


When you inspect the entries in the **Set Quantization Parameters** dialog, you see that we changed the coefficient format to [8 7], meaning we are using eight-bit wordlength and seven-bit fraction length to represent each filter coefficient. Changing the coefficient format to 8-bit, fixed point representation causes the effects shown in the figure — the passband rolls off early, the transition is less sharp, and the cutoff frequency lies beyond our 0.4 specification.

In FDATool, select **Analysis->Pole/Zero Plot** to view the poles and zeros for the 8-bit filter plotted on the unit circle. Or you might select **Analysis->View Filter Coefficients** to see the coefficient numerical values for the filter.

One more experiment in this example. We try changing the Direct form II transposed (`df2t`) filter structure to use second-order sections, which tend to be resistant to quantization effects. As we see in the figure, the elliptic filter

that uses second-order sections, even with the 8-bit coefficient format, performs identically to our reference filter.



In the Quantization Parameters options, you may note that the **Input/output scaling** changed when we converted our filter to second-order sections. Although we did not explicitly change the scaling by using the **Scale transfer-fcn <=1** option, converting the filter structure required that the gain for the new sections be changed to maintain the same overall gain for the filter. Thus our converted filter, which now has three sections, has unique scale factors for each section. The vector entries `[0.0625 1 2 1]` represent the scale factors applied to each section. `0.0625` is the scale factor applied to the input, `1` and `2` are the factors applied to the inputs of the second and third sections, and `1` is applied to the output from the third section. The resulting filter has the same gain as the original filter.

Selected Bibliography

[1] Antoniou, A., *Digital Filters: Analysis, Design, and Applications*, Second Edition, McGraw-Hill, Inc., 1993, 330–360.

[2] Mitra, S. K., *Digital Signal Processing: A Computer-Based Approach*, McGraw-Hill, Inc., 1998, 573–584.

3

Designing Adaptive Filters

Overview of Adaptive Filters and Applications (p. 3-4)	Read a short section about adaptive filters and their uses
Adaptive Filters in the Filter Design Toolbox (p. 3-11)	Learn about the adaptive filters provided in the toolbox
Examples of Adaptive Filters That Use LMS Algorithms (p. 3-12)	Presents examples of adaptive filters that use LMS algorithms to determine filter coefficients
Example of Adaptive Filter That Uses RLS Algorithm (p. 3-33)	Presents examples of adaptive filters that use RLS algorithms to determine filter coefficients
Examples of Adaptive Kalman Filters (p. 3-38)	Offers necessarily brief set of examples of adaptive Kalman filters
Selected Bibliography (p. 3-41)	Lists a few books that cover adaptive filters in both detail and with broad scope

Over the past three decades, digital signal processors have made great advances in increasing speed and complexity, and reducing power consumption. As a direct result, real-time adaptive filtering is quickly becoming an enabling technology for the future of communications, both wired and wireless. In the following sections, this guide presents an overview of adaptive filtering; discussions of some of the common applications for adaptive filters; and details about the adaptive filters available in the toolbox.

Listed below are the sections that cover adaptive filters in this guide. Within each section, examples and a short discussion of the theory of the filters introduces the adaptive filter concepts.

- "Overview of Adaptive Filters and Applications" on page 3-4 — presents a general discussion of adaptive filters and their applications.
  - "System Identification" on page 3-7 — talks using adaptive filters to identify the response of an unknown system such as a communications channel or a telephone line.
  - "Inverse System Identification" on page 3-8 — talks about using adaptive filters to develop a filter which has a response that is the inverse of an unknown system. You can overcome echoes in modem connections and local telephone lines by inserting an inverse adaptive filter and using it to compensate for the induced noise on the lines.
  - "Noise Cancellation (or Interference Cancellation)" on page 3-9 — useful for performing active noise cancellation where the filter adapts in real-time to keep the error small. Compare this to system identification where the filter adapts once and stays fixed thereafter.
  - "Prediction" on page 3-9 — describes using adaptive filters to predict a signals future values.
- "Adaptive Filters in the Filter Design Toolbox" on page 3-11 lists the adaptive filters included in the toolbox.
- "Examples of Adaptive Filters That Use LMS Algorithms" on page 3-12 presents a discussion of using LMS techniques to perform the filter adaptation process.
- "Example of Adaptive Filter That Uses RLS Algorithm" on page 3-33 discusses adaptive filters based on the RMS techniques for minimizing the total error between the known and unknown systems.

- “Examples of Adaptive Kalman Filters” on page 3-38 presents an example of as adaptive filter that uses the Kalman algorithm to determine filter coefficients.

For more detailed information about adaptive filters and adaptive filter theory, refer to the books listed in “Selected Bibliography” on page 3-41.

Overview of Adaptive Filters and Applications

Adaptive filters self learn. As the signal into the filter continues, the adaptive filter coefficients adjust themselves to achieve the desired result, such as identifying an unknown filter or cancelling noise in the input signal. In Figure 3-1, the shaded box represents the adaptive filter, comprising the adaptive filter and the adaptive RLS algorithm. For the general adaptive algorithm block diagram, look at Figure 3-2.

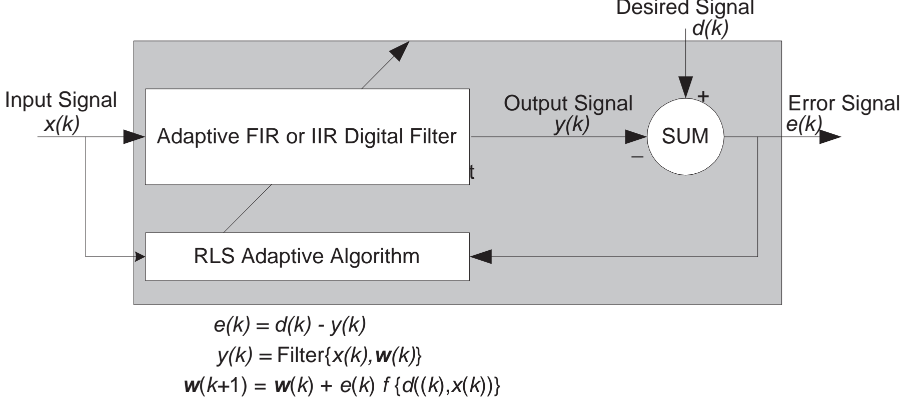


Figure 3-1: Block Diagram That Defines the Inputs and Output of a Generic RLS Adaptive Filter

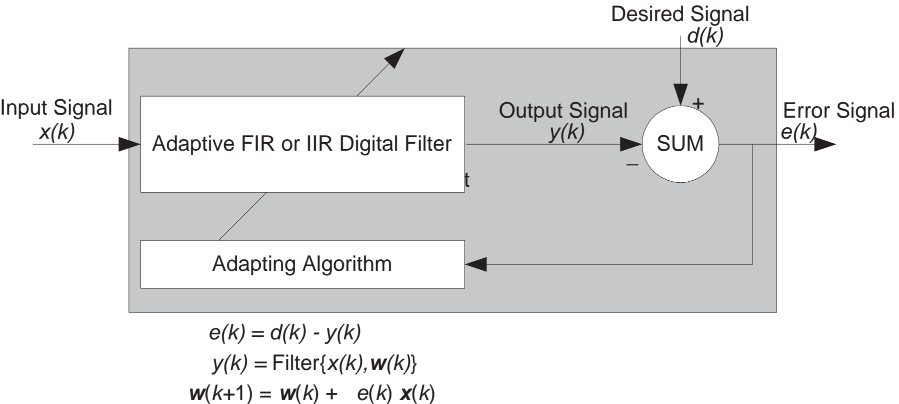


Figure 3-2: Block Diagram Defining General Adaptive Filter Algorithm Inputs and Outputs

An adaptive FIR or IIR filter designs itself based on the characteristics of the input signal to the filter and a signal which represent the desired behavior of the filter on its input. Designing the filter does not require any other frequency response information or specification. To define the self learning process the filter uses, you select the adaptive algorithm used to reduce the error between the output signal *y(k)* and the desired signal *d(k)*. When the LMS performance criteria for *e(k)* has achieved its minimum value through the iterations of the adapting algorithm, the adaptive filter is finished and its coefficients have converged to a solution. Now the output from the adaptive filter matches closely the desired signal *d(k)*. When you change the input data characteristics, sometimes called the filter environment, the filter adapts to the new environment by generating a new set of coefficients for the new data. Notice that when *e(k)* goes to zero and remains there you achieve perfect adaptation; the ideal result but not likely in the real world.

The adaptive filter functions in this toolbox implement the shaded portion of Figure 3-1, replacing the adaptive algorithm with an appropriate technique. Therefore, to use one of the functions you provide the input signal or signals and the initial values for the filter. A later section in this *User's Guide*, "Adaptive Filters in the Filter Design Toolbox" offers details about the algorithms available and the inputs required to use them in MATLAB.

Choosing an Adaptive Filter

With many adaptive filters to choose from, selecting the one that best meets your needs requires careful consideration. An exhaustive discussion of the criteria for selecting your approach is beyond the scope of this *User's Guide*. However, a few guidelines can help you make your choice.

Two main considerations frame the decision — the filter job to do and the filter algorithm to use.

When you begin to develop an adaptive filter for your needs, most likely the primary concern is whether using an adaptive filter is a cost-competitive approach to solving your filtering needs. Generally many areas determine the suitability of adaptive filters (these areas are common to most filtering and signal processing applications). Four such areas are:

- Filter consistency — does your filter performance degrade when the filter coefficients change slightly as a result of quantization, or you switch to fixed-point arithmetic? Will excessive noise in the signal hurt the performance of your filter?
- Filter performance — does your adaptive filter provide sufficient identification accuracy or fidelity, or does the filter provide sufficient signal discrimination or noise cancellation to meet your requirements?
- Tools — do tools exist that make your filter development process easier? Better tools can make it practical to use more complex adaptive algorithms.
- DSP requirements — can your filter perform its job within the constraints of your application. Does your processor have sufficient memory, throughput, and time to use your proposed adaptive filtering approach? Can you trade memory for throughput: use more memory to reduce the throughput requirements or use a faster signal processor?

Of the preceding considerations, characterizing filter consistency or robustness may be the most difficult.

Using the simulations in the Filter Design Toolbox offers a good first step in developing and studying these issues. Often, beginning your study using one of the least mean squares (LMS) algorithm filters provides both a relatively straightforward filter to implement and a sufficiently powerful tool for evaluating whether adaptive filtering can be useful for your problem.

Additionally, starting with an LMS approach can form a solid baseline against which you can study and compare the more complex adaptive filters available in the toolbox. Finally, your development process should, at some time, test your algorithm and adaptive filter with real data. For truly testing the value of your work there is no substitute for actual data.

With these considerations in mind, here are some applications that commonly use adaptive filters.

System Identification

One common 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.

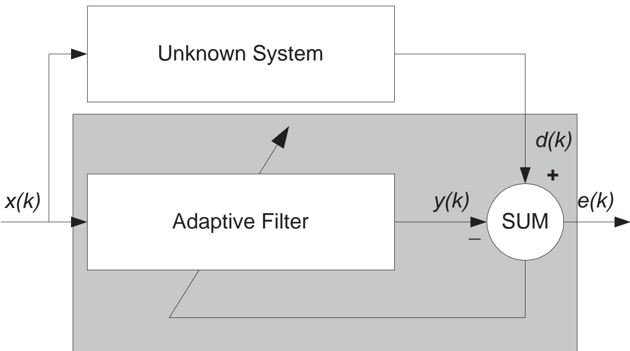


Figure 3-3: Using an Adaptive Filter to Identify an Unknown System

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. When the unknown system is a modem, the input often represents white noise, and is the sound you hear from your modem when you log in to your Internet service provider.

Inverse System Identification

By placing the unknown system in series with your adaptive filter, your filter becomes the inverse of the unknown system when *e(k)* gets 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.

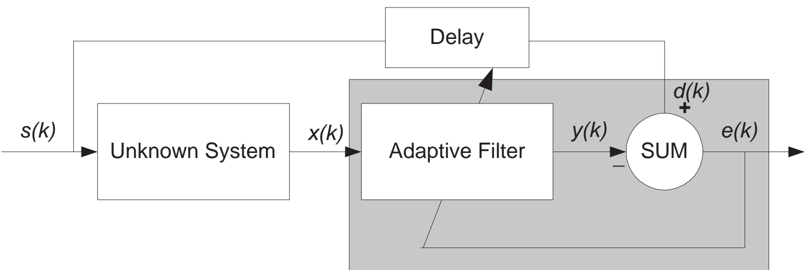


Figure 3-4: Determining an Inverse Response to an Unknown System

Without the delay element, the adaptive filter algorithm tries to match the output from the adaptive filter (*y(k)*) to input data (*x(k)*) that has not yet reached the adaptive elements because it is passing through the unknown system. In essence, the filter ends up trying to look ahead in time. As hard as it tries, the filter can never adapt: *e(k)* never reaches a very small value and your adaptive filter never compensates for the unknown system response. And it never provides a true inverse response to the unknown system. Including a delay equal to 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 possibly having other anomalies as well. Adding an adaptive filter which has a response that is the inverse of the wire response, adapting in real time, removes the rolloff and the anomalies, increasing the available frequency range and data rate for the telephone system.

Noise Cancellation (or Interference Cancellation)

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 represents noise that is correlated to the noise to remove from our desired signal.

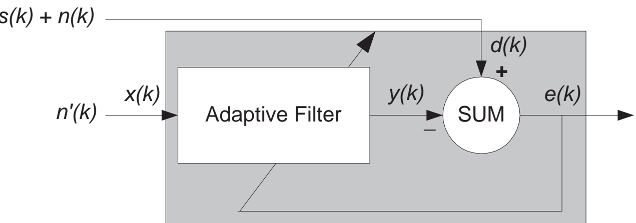


Figure 3-5: Using an Adaptive Filter to Remove Noise from an Unknown System

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 signals may seem to be an impossible task, without some limiting assumptions. Assume that the signal is either steady or slowly varying over time, and periodic over time as well.

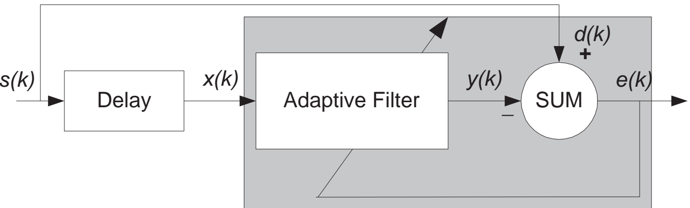


Figure 3-6: Predicting Future Values of a Periodic Signal

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.

Adaptive Filters in the Filter Design Toolbox

Filter Design Toolbox contains more than a half-dozen functions for applying adaptive filters to data. As you see in Table 3-1, the functions use various algorithms to determine the weights for the filter coefficients of the adapting filter. While the algorithms differ in their detail implementations, the LMS and RLS share a common operational approach — minimizing the error between the filter output and the desired signal; the Kalman algorithm function is somewhat different in how it determines the filter coefficients.

Table 3-1: Adaptive Filter Functions in the Toolbox

Function	Description
`adaptkalman`	Use a Kalman algorithm to determine the coefficients for a filter to model an unknown system.
`adaptlms`	Use a least mean squares (LMS) algorithm to determine the coefficients for a filter to model an unknown system.
`adaptnlms`	Use a normalized least mean squares algorithm to determine the coefficients for a filter to model an unknown system.
`adaptrls`	Use a recursive least squares algorithm to determine the coefficients for a filter to model an unknown system.
`adaptsd`	Use a sign-data LMS algorithm to determine the coefficients for a filter to model an unknown system.
`adaptse`	Use a sign-error LMS algorithm to determine the coefficients for a filter to model an unknown system.
`adaptss`	Use a sign-sign LMS algorithm to determine the coefficients for a filter to model an unknown system.

Presenting a detailed derivation of the Wiener-Hopf equation and determining solutions to it is beyond the scope of this *User's Guide*. Full descriptions of the theory appear in the adaptive filter references provided in the "Selected Bibliography" on page 3-41.

Examples of Adaptive Filters That Use LMS Algorithms

This section provides introductory examples using each of the least mean squares (LMS) adaptive filter functions in the toolbox.

The Filter Design Toolbox provides five adaptive filter design functions that use the LMS algorithms to search for the optimal solution to the adaptive filter:

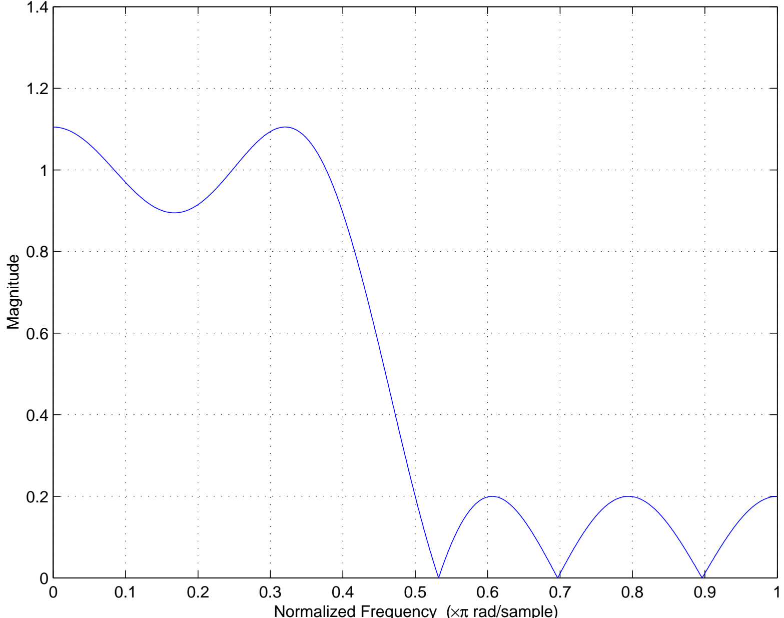
- `adaptlms` — implement the LMS algorithm to solve the Weiner-Hopf equation and find the filter coefficients for an adaptive filter.
- `adaptnlms` — implement the normalized variation of the LMS algorithm to solve the Weiner-Hopf equation and determine the filter coefficients of an adaptive filter.
- `adaptsd` — implement the sign-data variation of the LMS algorithm to solve the Weiner-Hopf equation and determine the filter coefficients of an adaptive filter. The correction to the filter weights at each iteration depends on the sign of the input $x(k)$.
- `adaptse` — implement the sign-error variation of the LMS algorithm to solve the Weiner-Hopf equation and determine the filter coefficients of an adaptive filter. The correction applied to the current filter weights for each successive iteration depends on the sign of the error, $e(k)$.
- `adaptss` — implement the sign-sign variation of the LMS algorithm to solve the Weiner-Hopf equation and determine the filter coefficients of an adaptive filter. The correction applied to the current filter weights for each successive iteration depends on both the sign of $x(k)$ and the sign of $e(k)$.

To demonstrate the differences and similarities between the various LMS algorithms supplied in the toolbox, the LMS and NLMS adaptive filter examples use the same filter for the unknown system. In this case, the unknown filter is one of the filters used in the examples from “gremez Examples” on page 2-8 — the constrained lowpass filter.

```
[b,err,res]=gremez(12,[0 0.4 0.5 1], [1 1 0 0], [1 0.2],...
{'w' 'c'});
```

From the figure you see that the filter is indeed lowpass and constrained to 0.2 ripple in the stopband. With this as the baseline, the adaptive LMS filter examples use the adaptive LMS algorithms and their initialization functions, to identify this filter in a system identification role. To review the general

model for system ID mode, look at “System Identification” on page 3-7 for the layout.



For the sign variations of the LMS algorithm, the examples use noise cancellation as the demonstration application, as opposed to the system identification application used in the LMS examples.

adaptlms Example — System Identification

To use the adaptive filter functions in the toolbox you need to provide three things:

- An unknown system or process to adapt to. In this example, the filter designed by `gremez` is the unknown system.
- Appropriate input data to exercise the adaptation process. In terms of the generic LMS model, these are the desired signal *d(k)* and the input signal *x(k)*.

- Both the adaptive LMS function to use and the matching initialization function to set up the adapting algorithm. Here we use `adaptlms` and `initlms`.

Start by defining an input signal `x`.

```
x = 0.1*randn(1,500);
```

The input is broadband noise. For the unknown system filter, use `gremez` to create a twelfth-order lowpass filter:

```
[b,err,res] = gremez(12,[0 0.4 0.5 1], [1 1 0 0], [1 0.2],...
{'w' 'c'});
```

Although you do not need them here, include the `err` and `res` output arguments.

Now filter the signal through the unknown system to get the desired signal.

```
d = filter(b,1,x);
```

With the unknown filter designed and the desired signal in place you can apply the adaptive LMS filter to identify the unknown.

Preparing the adaptive filter algorithm requires that you provide starting values for estimates of the filter coefficients and the LMS step size in a single structure `s`. We use `initlms` to populate the structure. You could start with estimated coefficients of some set of nonzero values; this example uses zeros for the 12 initial filter weights. For the step size, 0.8 is a reasonable value — a good compromise between being large enough to converge well within the 500 iterations (500 input sample points) and small enough to create an accurate copy of the unknown filter.

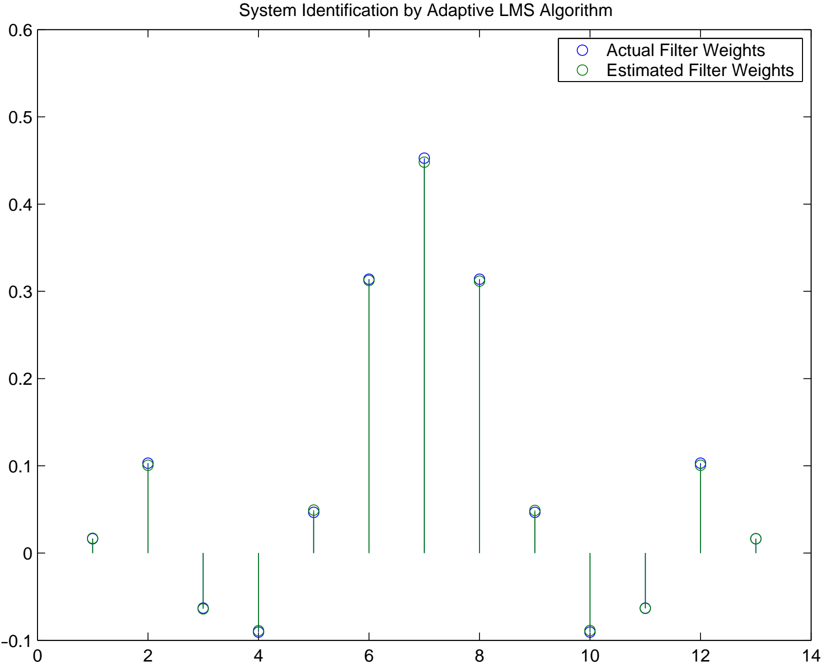
```
w0 = zeros(1,13);
mu = 0.8;
s = initlms(w0,mu);
```

Structure `s` now comprises the following fields.

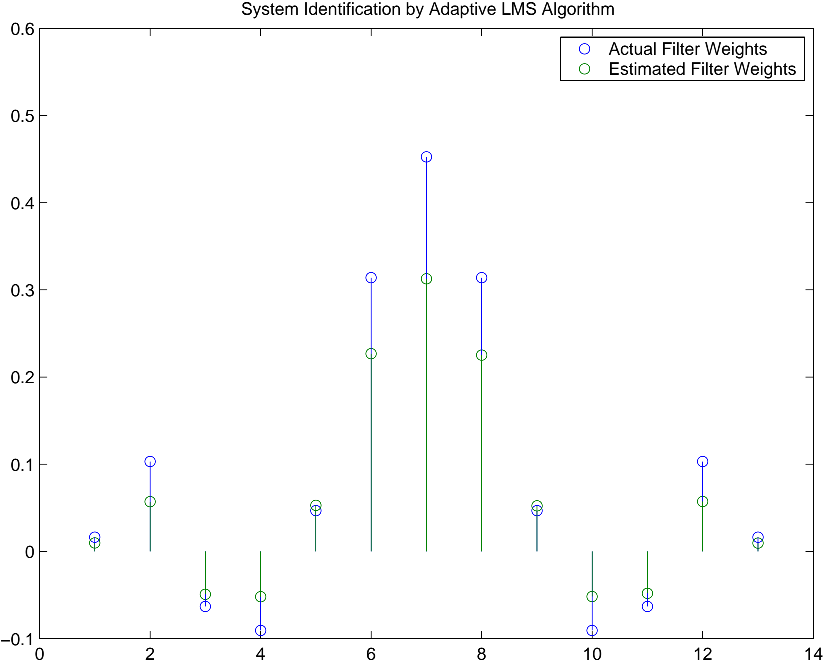
Structure Element	Element Contents	initlms Element
`s.coeffs`	LMS FIR filter coefficients. Should be initialized with the initial coefficients for the FIR filter prior to adapting. You need (adapting filter order + 1) entries in `s.coeffs`. Updated filter coefficients are returned in `s.coeffs` when you use `s` as an output argument.	`wo`
`s.step`	Sets the LMS algorithm step size. Determines both how quickly and how closely the adative filter adapts to the filter solution.	`mu`
`s.states`	Returns the states of the FIR filter after adaptation. This is an optional element. If omitted, it defaults to a zero vector of length equal to the filter order. When you use `adaptlms` in a loop structure, use this element to specify the initial filter states for the adapting FIR filter.	`zi`
`s.leakage`	Specifies the LMS leakage parameter. Allows you to implement a leaky LMS algorithm. Including a leakage factor can improve the results of the algorithm by forcing the LMS algorithm to continue to adapt even after it reaches a minimum value. Ranges between 0 and 1. This is an optional field. Defaults to one if omitted (specifying no leakage) or set to empty, [ ].	`lf`
`s.iter`	Total number of iterations in the adaptive filter run. Although you can set this in s, you should not. Consider it a read-only value.	

Finally, using the desired signal, d, the input to the filter, x, and the structure that contains the algorithm initialization settings, s, we run the adaptive filter to determine the unknown system and plot the results, comparing the actual coefficients from gremez to the coefficients found by adaptlms.

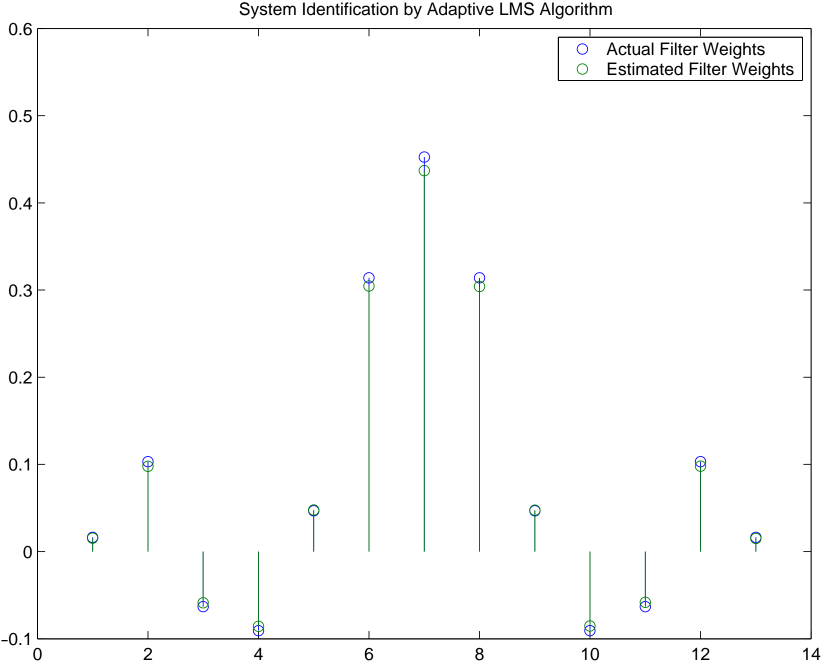
```
[y,e,s] = adaptlms(x,d,s);
stem([b.' s.coeffs.'])
```



In the stem plot the actual and estimated filter weights are the same. As an experiment, try changing the step size to 0.2. Repeating the example with mu = 0.2 results in the following stem plot. The estimated weights fail to approximate the actual weights closely.



Since this may be because we did not iterate over the LMS algorithm enough times, try using 1000 samples. With 1000 samples, the stem plot, shown in the next figure, looks much better, albeit at the expense of much more computation. Clearly you should take care to select the step size with both the computation required and the fidelity of the estimated filter in mind.



adaptnlms Example — System Identification

To improve the convergence performance of the LMS algorithm, the normalized variant 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. Thus the step size changes with time. As a result, the normalized algorithm converges more quickly with fewer samples in many cases. For input signals that change slowly over time, the normalized LMS can represent a more efficient LMS approach.

In the `adaptlms` example, we used `gremez` to create the filter that we would identify. So you can compare the results, we use the same filter, and replace `adaptlms` with `adaptnlms`, to use the normalized LMS algorithm variation. You should see better convergence with similar fidelity.

First, generate the input signal and the unknown filter.

```
x = 0.1*randn(1,500);
[b,err,res] = gremez(12,[0 0.4 0.5 1], [1 1 0 0], [1 0.2],...
{'w' 'c'});
d = filter(b,1,x);
```

Again `d` represents the desired signal *d(x)* as we defined it in Figure 3-1 and `b` contains the filter coefficients for our unknown filter.

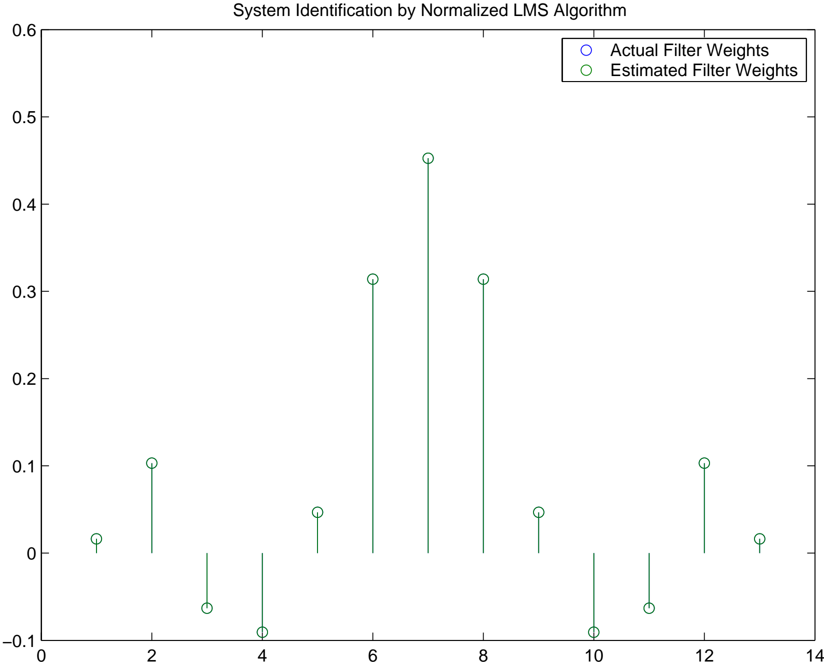
```
w0 = zeros(1,13);
mu = 0.8;
s = initnlms(w0,mu);
```

We use the preceding code to initialize the normalized LMS algorithm, just as we used `initlms` to prepare the LMS algorithm in the `adaptlms` example. You can see the input arguments are identical in this case. While there are optional input arguments that you use to refine the normalized algorithm, such as `offset` and `leakage` factor, to maintain the comparison to our LMS example we use the same set of input arguments used earlier. For more information about the optional input arguments, refer to `initnlms` in the reference section of this *User's Guide*.

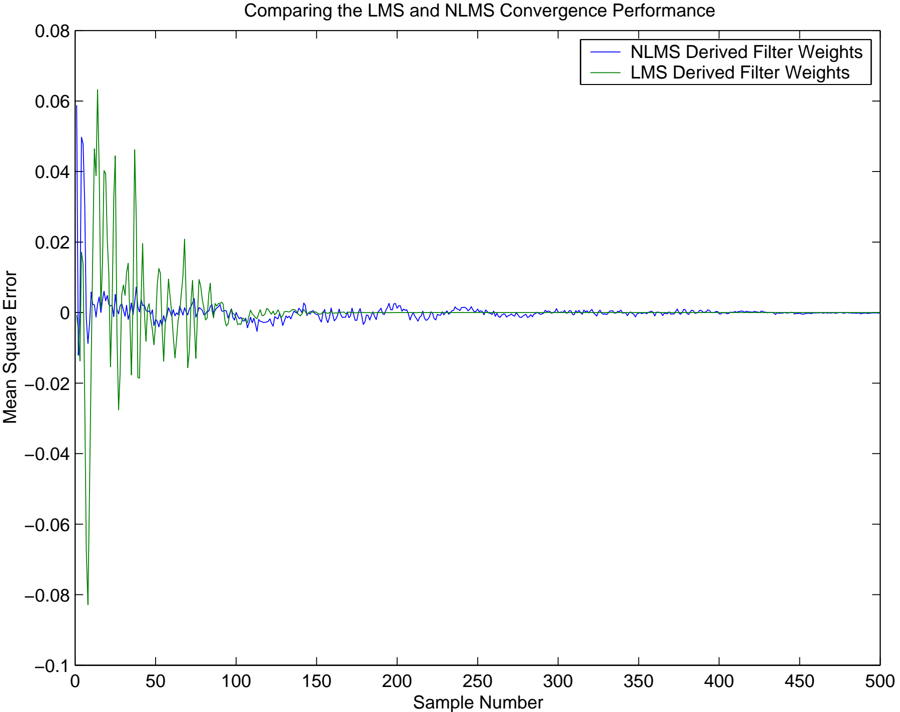
Running the system identification process is a matter of using `adaptnlms` with the desired signal, the input signal, and the initial filter coefficients and conditions specified in `s` as input arguments. Then plot the results to compare the adapted filter to the actual filter.

```
[y,e,s] = adaptnlms(x,d,s);
stem([b.' s.coeffs.'])
```

As shown in the following stem plot (a convenient way to compare the estimated and actual filter coefficients), the two are close to identical.



If we compare the convergence performance of the regular LMS algorithm to the normalized LMS variant, you see the normalized version adapts in far fewer iterations to a result almost as good as the nonnormalized version.



adaptsd Example — Noise Cancellation

When the amount of computation required to derive an adaptive filter drives your development process, the sign-data variant of the LMS (SDLS) algorithm may be a very good choice. Fortunately, the current state of digital signal processor (DSP) design has relaxed the need to minimize the operations count by making DSPs whose multiply and shift operations are as fast as add operations. Thus some of the impetus for the sign-data algorithm (and the sign-error and sign-sign variations) has been lost to DSP technology improvements.

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. Using 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

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \mu e(k) sgn[\mathbf{x}(k)] \ , \ sgn[\mathbf{x}(k)] = \begin{cases} 1, & \mathbf{x}(k) > 0 \\ 0, & \mathbf{x}(k) = 0 \\ -1, & \mathbf{x}(k) < 0 \end{cases}$$

with vector **w** containing the weights applied to the filter coefficients and vector **x** containing the input data. *e(k)* (equal to desired signal - filtered signal) is the error at time k and is the quantity the SDLMS algorithm seeks to minimize. μ (`mu`) is the step size. As you specify `mu` smaller, the correction to the filter weights gets smaller for each sample and the SDLMS error falls more slowly. Larger `mu` 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 good convergence rate and stability, select `mu` within the following practical bounds

$$0 < \mu < \frac{1}{N\{InputSignalPower\}}$$

where N is the number of samples in the signal. Also, define `mu` 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. 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 may result in the error growing beyond all bounds. You restrain the tendency of the sign-data algorithm to get out of control by choosing a small step size (μ<< 1) and setting the initial conditions for the algorithm to nonzero positive and negative values.

In this noise cancellation example, `adaptsd` requires two input data sets:

- Data containing a signal corrupted by noise. In Figure 3-5, this is *d(k)*, the desired signal. The noise cancellation process removes the noise, leaving the signal.
- Data containing random noise (*x(k)* in Figure 3-5) that 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, then add the filtered noise to the signal.

```
noise=randn(1,1000);
nfilt=fir1(11,0.4); % Eleventh order lowpass filter
fnoise=filter(nfilt,1,noise); % Correlated noise data
d=signal.'+fnoise;
```

`fnoise` is the correlated noise and `d` is now the desired input to the sign-data algorithm.

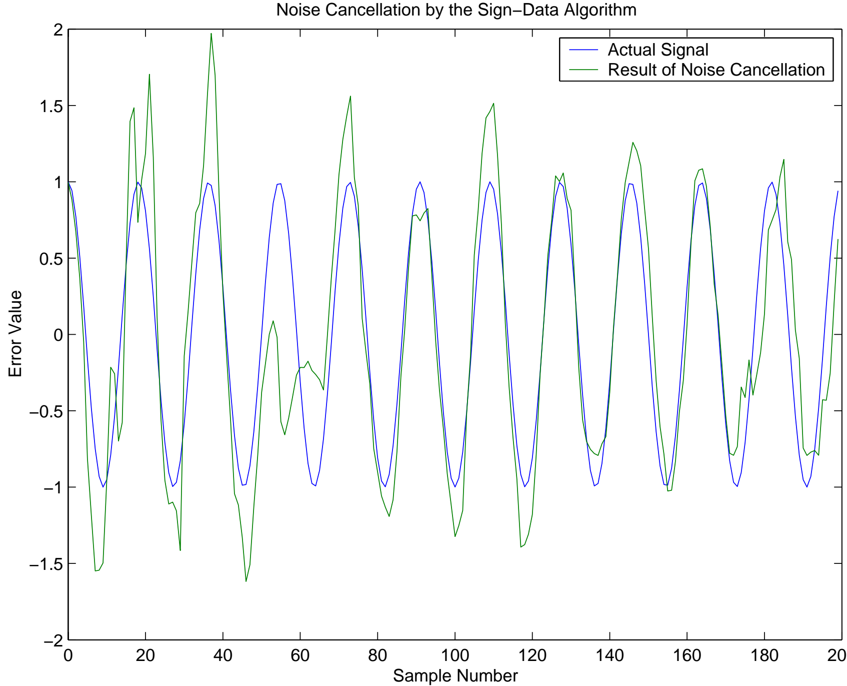
To prepare the algorithm for processing, set the input conditions `w0` and `mu` in structure `s`. As noted earlier in this section, the values you set for `w0` and `mu` determine whether the adaptive filter can remove the noise from the signal path. In "adaptlms Example — System Identification" on page 3-13, you set `w0`, the filter coefficients, to zeros. Except in rare 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, we start with the coefficients in the filter we used to filter the noise (`nfilt`), and modify them slightly so the algorithm has to adapt.

```
w0 = nfilt.' -0.01; % Set the filter initial conditions.
mu = 0.05;          % Set the set size for algorithm updating.
s=initsd(w0,mu);   % Initialize the input structure for adaptsd.
```

With the required input arguments for adaptsd prepared, run the adaptation and view the results.

```
[y,e,s] = adaptsd(noise,d,s);
plot(0:199,signal(1:200),0:199,e(1:200));
```

When adaptsd runs, it uses far fewer multiply operations than either of the LMS algorithms. Also, performing the sign-data adaptation requires only bit shifting multiplys when the step size is a power of two. Although the performance of the sign-data algorithm as shown in the next figure is quite good, the sign-data algorithm is much less stable than the standard LMS variations. In this noise cancellation example, the signal after processing is a very good match to the input signal, but the algorithm could very easily grow without bound rather than achieve good performance. Changing w0, mu, or even the lowpass filter you used to create the correlated noise can cause noise cancellation to fail and the algorithm to become useless.



adaptse Example — Noise Cancellation

In some cases, the sign-error variant of the LMS algorithm may be a very good choice for an adaptive filter application. In the standard and normalized variations of the LMS adaptive filter, the coefficients for the adapting filter arise from calculating the mean square error between the desired signal and the output signal from the unknown system, and applying the result to the current filter coefficients. Using the sign-error algorithm replaces the mean square error calculation by using the sign of the error to modify 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-error LMS algorithm is

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \mu \operatorname{sgn}[e(k)][\mathbf{x}(k)] \ , \ sgn[e(k)] = \begin{cases} 1, & e(k) > 0 \\ 0, & e(k) = 0 \\ -1, & e(k) < 0 \end{cases}$$

with vector **w** containing the weights applied to the filter coefficients and vector **x** containing the input data. *e(k)* (equal to desired signal - filtered signal) is the error at time *k* and is the quantity the SELMS algorithm seeks to minimize. μ (`mu`) is the step size. As you specify `mu` smaller, the correction to the filter weights gets smaller for each sample and the SELMS error falls more slowly. Larger `mu` 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 good convergence rate and stability, select `mu` within the following practical bounds

$$0 < \mu < \frac{1}{N\{InputSignalPower\}}$$

where *N* is the number of samples in the signal. Also, define `mu` as a power of two for efficient computation.

Note How you set the initial conditions of the sign-data algorithm profoundly influences the effectiveness of the adaptation. Because the algorithm essentially quantizes the error signal, the algorithm can become unstable easily. A series of large error values, coupled with the quantization process may result in the error growing beyond all bounds. You restrain the tendency of the sign-error algorithm to get out of control by choosing a small step size (μ<< 1) and setting the initial conditions for the algorithm to nonzero positive and negative values.

In this noise cancellation example, `adaptse` requires two input data sets:

- Data containing a signal corrupted by noise. In Figure 3-5, this is *d(k)*, the desired signal. The noise cancellation process removes the noise, leaving the signal.
- Data containing random noise (*x(k)* in Figure 3-5) that 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, then add the filtered noise to the signal.

```
noise=randn(1,1000);
nfilt=fir1(11,0.4); % Eleventh order lowpass filter.
fnoise=filter(nfilt,1,noise); % Correlated noise data.
d=signal.'+fnoise;
```

`fnoise` is the correlated noise and `d` is now the desired input to the sign-data algorithm.

To prepare the SSLMS algorithm for processing, set the input conditions in structure `s`. As noted earlier in this section, the values you set for `w0` and `mu` determine whether the adaptive filter can remove the noise from the signal path. In the LMS tutorial, you set `w0`, the filter coefficients, to zeros. Except in rare cases, that approach does not work for the sign-data algorithm. The closer

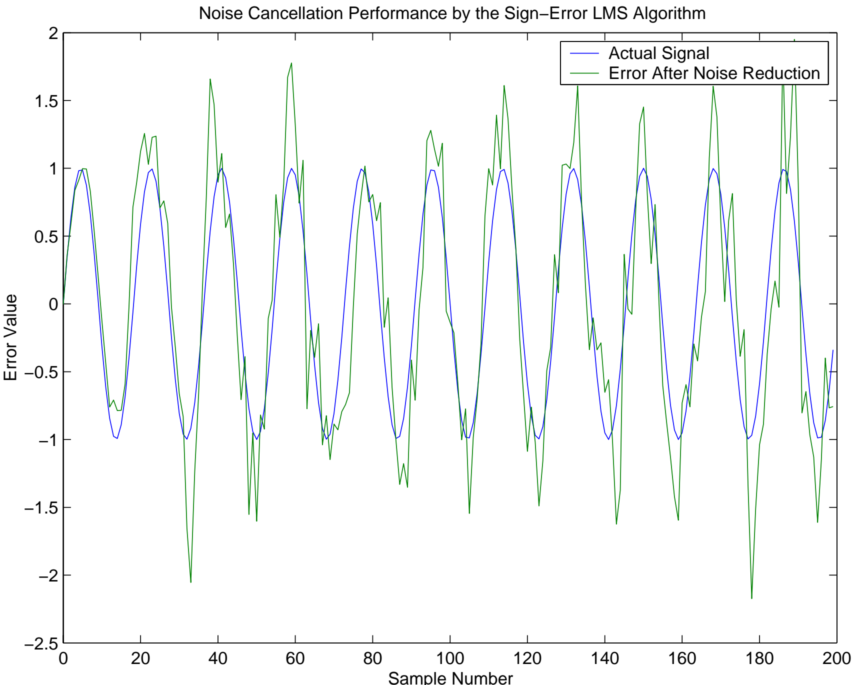
you set your initial filter coefficients to the expected values, the more likely it is that the algorithm remains well-behaved and converges to a filter solution that removes the noise effectively. For this example, we start with the coefficients of the filter we used to filter the noise (`nfilt`), and modify them slightly so the algorithm has to adapt.

```
w0 = nfilt.' -0.01; % Set the filter initial conditions.
mu = 0.05;          % Set the set size for algorithm updating.
s=initse(w0,mu);   % Initialize the input structure for adaptse.
```

With the required input arguments for `adaptse` prepared, run the adaptation and view the results.

```
[y,e,s] = adaptse(noise,d,s);
plot(0:199,signal(1:200),0:199,e(1:200));
```

When `adaptse` runs, it uses far fewer multiply operations than either of the LMS algorithms. Also, performing the sign-error adaptation requires only bit shifting multiplys when the step size is a power of two. Although the performance of the sign-data algorithm as shown in the next figure is quite good, the sign-data algorithm is much less stable than the standard LMS variations. In this noise cancellation example, the signal after processing is a very good match to the input signal, but the algorithm could very easily become unstable rather than achieve good performance. Changing `w0`, `mu`, or even the lowpass filter you used to create the correlated noise can cause noise cancellation to fail and the algorithm to become useless.



adaptss Example — Noise Cancellation

The final variation of the LMS algorithm in the toolbox is the sign-sign variant (SSLMS). The rationale for this version matches those for the sign-data and sign-error algorithms presented in preceding sections. For more details, refer to “adaptsd Example — Noise Cancellation” on page 3-21.

The sign-sign algorithm (SSLMS) replaces the mean square error calculation to 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 essence, the algorithm quantizes both the error and the input by applying the sign operator to them.

In vector form, the sign-sign LMS algorithm is

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \mu \operatorname{sgn}[e(k)] \operatorname{sgn}[\mathbf{x}(k)] \;, \operatorname{sgn}[z(k)] = \begin{cases} 1, & z(k) > 0 \\ 0, & z(k) = 0 \\ -1, & z(k) < 0 \end{cases}$$

where

$$z(k) = [e(k)] \operatorname{sgn}[\mathbf{x}(k)]$$

Vector **w** contains the weights applied to the filter coefficients and vector **x** contains the input data. *e(k)* (= desired signal - filtered signal) is the error at time *k* and is the quantity the SSLMS algorithm seeks to minimize. μ (mu) is the step size. As you specify mu smaller, the correction to the filter weights gets smaller for each sample and the SSLMS error falls more slowly. Larger mu 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 good convergence rate and stability, select mu within the following practical bounds

$$0 < \mu < \frac{1}{N\{InputSignalPower\}}$$

where *N* is the number of samples in the signal. Also, define mu as a power of two for efficient computation.

Note How you set the initial conditions of the sign-sign algorithm profoundly influences the effectiveness of the adaptation. Because the algorithm essentially quantizes the input signal and the error signal, the algorithm can become unstable easily. A series of large error values, coupled with the quantization process may result in the error growing beyond all bounds. You restrain the tendency of the sign-sign algorithm to get out of control by choosing a small step size (μ<< 1) and setting the initial conditions for the algorithm to nonzero positive and negative values.

In this noise cancellation example, adaptss requires two input data sets:

- Data containing a signal corrupted by noise. In Figure 3-5, this is *d(k)*, the desired signal. The noise cancellation process removes the noise, leaving the cleaned signal as the content of the error signal.
- Data containing random noise (*x(k)* in Figure 3-5) that is correlated with the noise that corrupts the signal data, called. 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, then add the filtered noise to the signal.

```
noise=randn(1,1000);
nfilt=fir1(11,0.4); % Eleventh order lowpass filter
fnoise=filter(nfilt,1,noise); % Correlated noise data
d=signal.'+fnoise;
```

`fnoise` is the correlated noise and `d` is now the desired input to the sign-data algorithm.

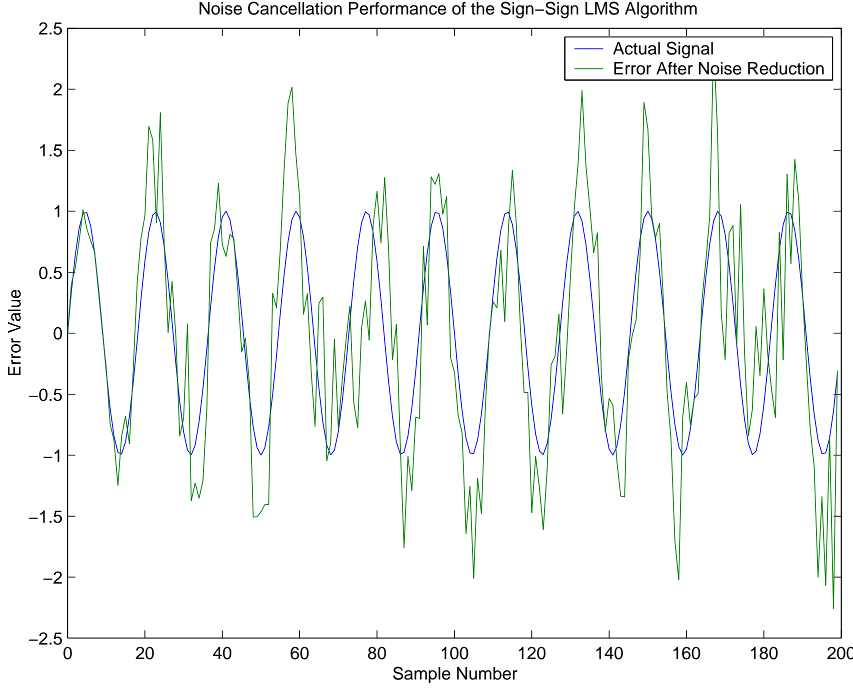
To prepare the algorithm for processing, set `w0` and `mu` — the input conditions in structure `s`. As noted earlier in this section, the values you set for `w0` and `mu` determine whether the adaptive filter can remove the noise from the signal path. In the LMS tutorial, you set `w0`, the filter coefficients, to zeros. Except in rare cases, that approach does not work for the sign-sign 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, we start with the coefficients in the filter we used to filter the noise (`nfilt`), and modify them slightly so the algorithm has to adapt.

```
w0 = nfilt.' -0.01; % Set the filter initial conditions.
mu = 0.05;          % Set the set size for algorithm updating.
s=initss(w0,mu);   % Initialize the input structure for adaptss.
```

With the required input arguments for `adaptsd` prepared, run the adaptation and view the results.

```
[y,e,s] = adaptss(noise,d,s);
plot(0:199,signal(1:200),0:199,e(1:200));
```

When adaptss runs, it uses far fewer multiply operations than either of the LMS algorithms. Also, performing the sign-sign adaptation requires only bit shifting multiplys when the step size is a power of two. Although the performance of the sign-sign algorithm as shown in the next figure is quite good, the sign-sign algorithm is much less stable than the standard LMS variations. In this noise cancellation example, the signal after processing is a very good match to the input signal. The algorithm could very easily fail to converge rather than achieve good performance if the quantization process produces very small or large changes. Changing w0, mu, or even the lowpass filter you used to create the correlated noise can cause noise cancellation to fail and the algorithm to become useless.



As an aside, the sign-sign LMS algorithm is part of the international CCITT standard for 32 Kb/s ADPCM telephony.

Example of Adaptive Filter That Uses RLS Algorithm

This section provides an introductory example that uses the RLS adaptive filter function `adaptrls`.

If LMS algorithms represent the simplest and most easily applied adaptive algorithms, the recursive least squares (RLS) algorithms represents increased complexity, computational cost, and fidelity. In performance, RLS approaches the Kalman filter (`adaptkalman`) in adaptive filtering applications, at 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.

In contrast to the least mean squares algorithm, from which it can be derived, the RLS adaptive algorithm minimizes the total squared error between the desired signal and the output from the unknown system. Referring to Figure 3-2, you see the signal flow graph (or model) for the RLS adaptive filter system. Note that the signal paths and identifications are the same whether the filter uses RLS or LMS. The difference lies in the adapting portion. 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. Examples of the sign variants of the LMS algorithms demonstrated this feature to demonstrate the differences between the sign-data, sign-error, and sign-sign variations of the LMS algorithm.

One interesting input option that applies to RLS algorithms is not present in the LMS processes — a forgetting factor, λ, that determines how the algorithm treats past data input to the algorithm. When the LMS algorithm looks at the error to minimize, it considers only the current error value. In the RLS method, the error considered is the total error from the beginning to the current data point. Said another way, the RLS algorithm has infinite memory — all error data is given the same consideration in the 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 value of older error data by multiplying the old data by the forgetting factor. Since $0 \leq \lambda < 1$, applying the factor is equivalent to weighting the older error. When $\lambda = 1$, all previous error is 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 $\lambda = 0.9$, the RLS algorithm multiplies an error value from 50 samples in the past by an

attenuation factor of $0.9^{50} = 5.15 \times 10^{-3}$, considerably deemphasizing the influence of the past error on the current total error.

adaptrls Example — Inverse System Identification

Rather than use a system identification application, or a noise cancellation model, this example use the inverse system identification model shown in Figure 3-4. Cascading the adaptive filter with the 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 is H(z) and the adaptive filter transfer function is G(z), the error measured between the desired signal and the signal from the cascaded system reaches its minimum when the product of H(z) and G(z) is 1, G(z)*H(z) = 1. For this relation to be true, G(z) must equal -H(z), the inverse of the transfer function of the unknown system.

To demonstrate that this is true, create a signal to input to the cascaded filter pair.

```
x = randn(1,3000);
```

In the cascaded filters case, like this one, 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 32 samples, 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 about the number of samples that is equal to half the order of the adaptive filter. Delaying the input requires prepending 12 zero-values samples to `x`.

```
delay = zeros(1,12);
d = [delay x(1:2988)]; %Concatenate the delay and the signal.
```

You have to keep the desired signal vector `d` the same length as `x`, hence adjust the signal element count to allow for the delay samples. Although it is not generally true, for this example you know the order of the unknown filter, so you add a delay equal to the order of the unknown filter.

For the unknown system, use a lowpass, 12th-order FIR filter.

```
ufilt = fir1(12,0.55,'low');
```

Filtering `x` provides the input data signal for the adaptive algorithm function.

```
xdata = filter(ufilt,1,x);
```

Use `initrls` to set the initial conditions for the algorithm. `initrls` produces the structure `s`, one of the input arguments to `adaptrls`. You can set the initial conditions without using `initrls` by including each input argument to adaptrls on its own — `w0`, `p0`, `lambda`, `zi`, and `alg`. For more about `initrls` and the input conditions to prepare the RLS algorithm, refer to `initrls` and `adaptrls` in the reference section of this *User's Guide*.

```
w0 = zeros(13,1);
p0 = 2*eye(13);
lambda = 0.99;
s = initrls(w0,p0,lambda);
```

Most of the process to this point is the same as the preceding examples. However, since this example is looking 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 (`xdata`) carries the unknown system information. With Gaussian distribution and variance of 1, the unfiltered noise `d` is the desired signal. The code to run this adaptive filter example is

```
[y,e,s] = adaptrls(xdata,d,s);
```

where `y` returns the coefficients of the adapted filter and `e` contains the error signal as the filter adapts to find the inverse of the unknown system. You can review the returned elements of the adapted filter in `s`.

Figure 3-7 presents the results of the adaptation. In the figure, we present the magnitude response curves for the unknown and adapted filters. As a reminder, the unknown filter was a lowpass filter with cutoff at 0.55, on the normalized frequency scale from 0 to 1.

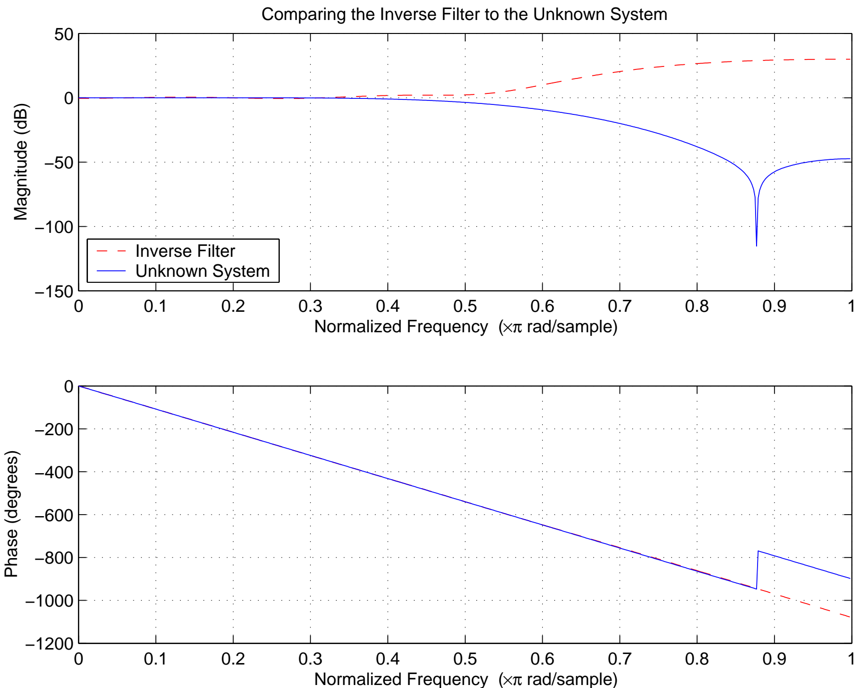


Figure 3-7: Comparing the Results of the RLS Inverse System Identification

Viewed alone in Figure 3-8, the inverse system looks like a fair compensator for the unknown lowpass filter — a high pass filter with linear phase.

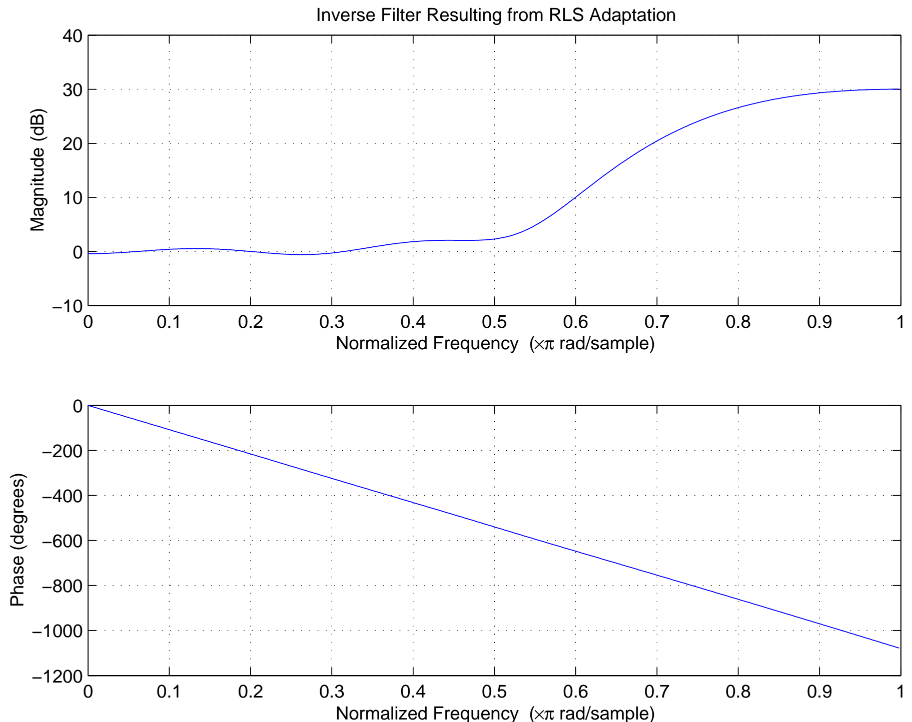


Figure 3-8: After Adapting, the RLS Algorithm Produces a Highpass Filter

Examples of Adaptive Kalman Filters

Without going into details because the specifics are beyond the scope of this *User's Guide*, the adaptive filter functions in the toolbox represent variations of Kalman filtering. Thus, Kalman filters are the basis of all the other functions, and perhaps the most effective and efficient since each succeeding filter update in the Kalman algorithm depends only on the most recent input data.

`adaptkalman` shares many input arguments with the LMS and RLS adaptive functions. To completely specify the Kalman algorithm requires a few additional inputs — `k0`, `qm`, and `qp` as listed in the following table.

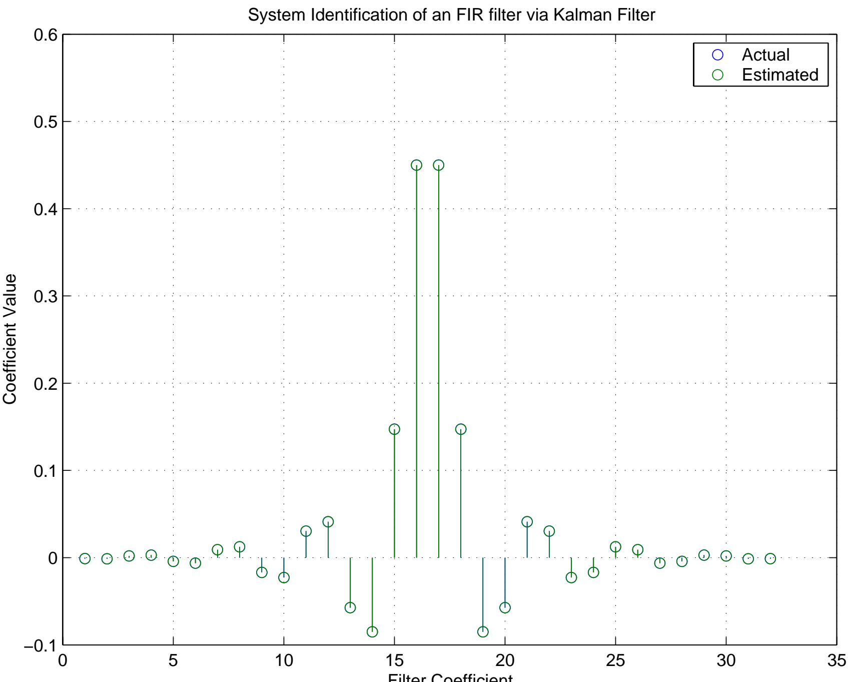
Structure Element	Element Description	initkalman Argument
`s.coeffs`	Kalman adaptive filter coefficients. Should be initialized with the initial values for the FIR filter coefficients. Updated coefficients are returned when you use `s` as an output argument.	`w0`
`s.errcov`	The state error covariance matrix. Initialize this element with the initial error state covariance matrix. An updated matrix is returned when you use `s` as an output argument.	`k0`
`s.measvar`	Contains the measurement noise variance matrix.	`qm`
`s.procov`	Contains the process noise covariance matrix.	`qp`
`s.states`	Returns the states of the FIR filter. This is an optional element. If omitted, it defaults to a zero vector of length equal to the filter order.	`zi`

Structure Element	Element Description	initkalman Argument
`s.gain`	Kalman gain vector. Computed and returned after every iteration. This is a read-only value.	Not applicable
`s.iter`	Total number of iterations in the adaptive filter run. Although you can set this number in `s`, you should not.	Not applicable

Befitting the nature of the Kalman approach to adaptive filtering, arguments `k0`, `qm`, and `qp` are matrices that define the known parameters for the algorithm — the initial conditions. Often you do not know the initial state of the update process equation that defines each filter update. To overcome this fact, we use mean and correlation matrices of the initial state to define the equation.

adaptkalman Example — System Identification

```
x  = 0.1*randn(1,500);
b  = fir1(31,0.5);
d  = filter(b,1,x);
w0 = zeros(1,32);
k0 = 0.5*eye(32);
qm = 2;
qp = 0.1*eye(32);
s = initkalman(w0,k0,qm,qp);
[y,e,s] = adaptkalman(x,d,s);
stem([b.',s.coeffs.']);
legend('Actual','Estimated');
title('System Identification of an FIR Filter via Kalman Filter');
grid on;
```


System Identification of an FIR filter via Kalman Filter
Actual
Estimated
Coefficient Value
Filter Coefficient
0.6
0.5
0.4
0.3
0.2
0.1
0
−0.1
0
5
10
15
20
25
30
35

Selected Bibliography

[1] Hayes, Monson H., *Statistical Digital Signal Processing and Modeling*, John Wiley & Sons, 1996, 493–552.

[2] Haykin, Simon, *Adaptive Filter Theory*, Prentice-Hall, Inc., 1996

4

Digital Frequency Transformations

Introduction (p. 4-2)	Provides background about digital frequency transformations for filters
Definition of the Problem (p. 4-3)	Presents and defines the problem of using digital frequency transformation
Frequency Transformations for Real Filters (p. 4-11)	Discusses the functions in the toolbox used for transforming real filters to other real filters
Frequency Transformations for Complex Filters (p. 4-26)	Talks about the functions in the toolbox used for transforming complex filters to other complex filters, or real filters to complex filters

Introduction

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.

This tutorial gives an overview and interpretation of the frequency transformations, and describes the range of transformations available to the toolbox user. To aid this purpose the tutorial has been arranged into three sections:

- "Definition of the Problem" on page 4-3 introduces the frequency transformation concept and provides its mathematical and intuitive interpretations.
- "Frequency Transformations for Real Filters" on page 4-11 describes the real frequency transformations available in the toolbox. Such transformations start from a real prototype filter and return a real target filter.
- "Frequency Transformations for Complex Filters" on page 4-26 describes complex frequency transformations available in the toolbox. Such transformations start from the any real or complex prototype filter and return a complex target filter.

Definition of the Problem

The basic form of mapping in common use 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.

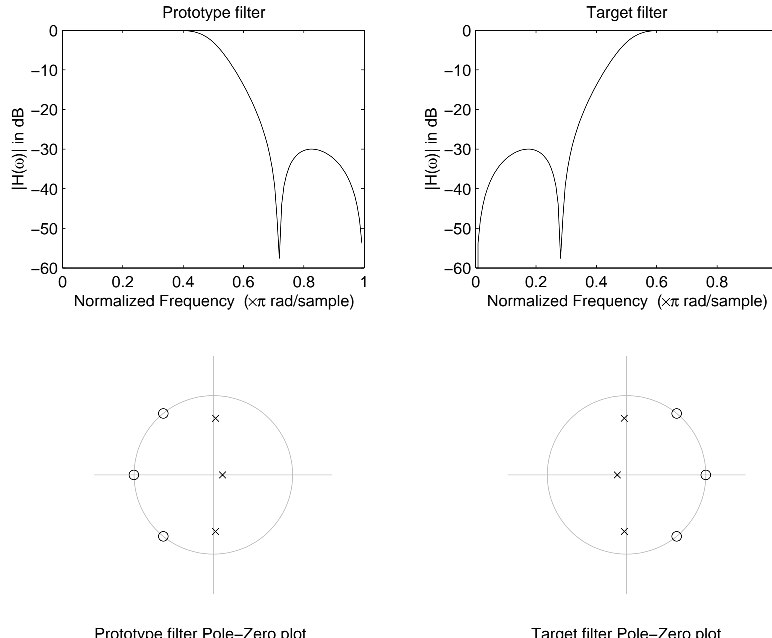


Figure 4-1: Example of a Simple Mirror Transformation

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.

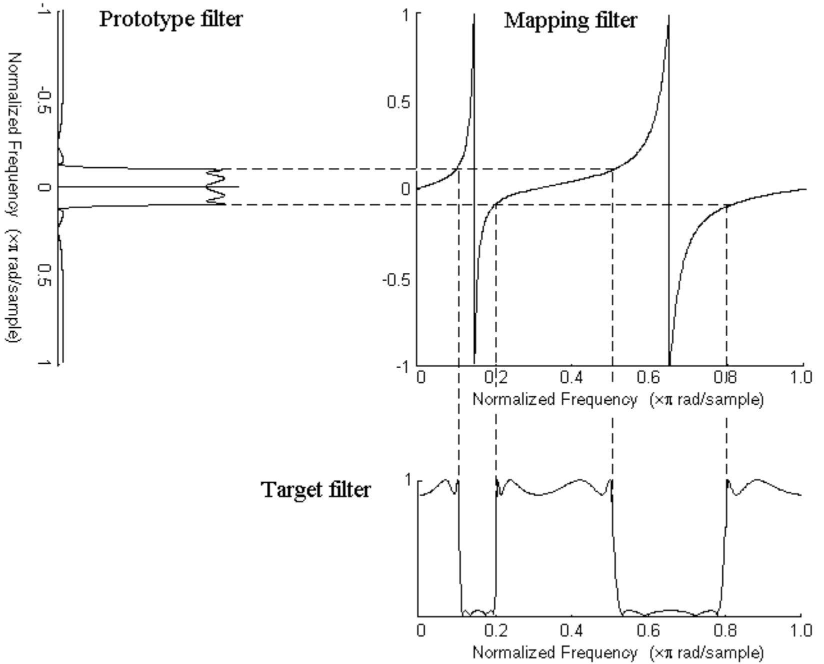


Figure 4-2: Graphical Interpretation of the Mapping Process

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 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).

All the transformations forming the package are explained in the next sections of the tutorial. They are separated into those operating on real filters and those generating or working with complex filters. The choice of transformation ranges from standard Constantinides first and second-order ones [13][14] up to the real muliband filter by Mullis and Franchitti [15], and the complex multiband filter and real/complex multipoint ones by Krukowski, Cain and Kale [16].

Selecting 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.

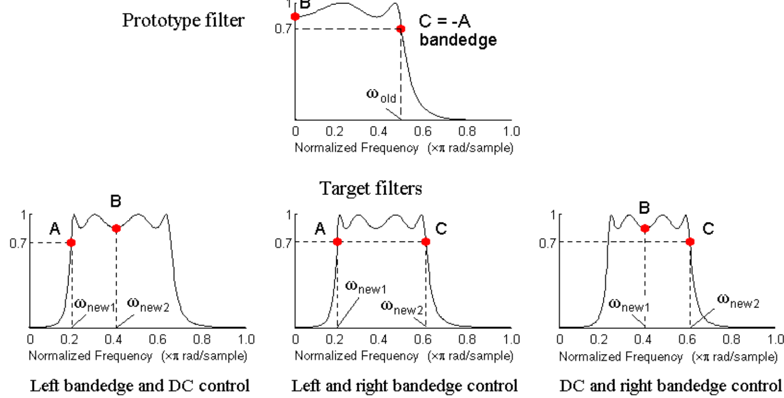


Figure 4-3: Feature Selection for Real Lowpass to Bandpass Transformation

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.

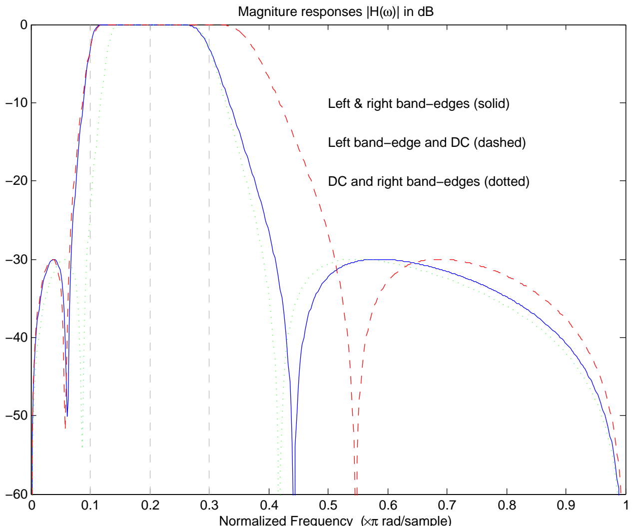


Figure 4-4: Result of choosing different features

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 Equation . 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:

$$H_o(z) = \frac{\sum_{i=1} \left(S \sum_{k=0} \alpha_k z^k - z_i \cdot \sum_{k=0} \alpha_k z^{N-k} \right)}{\sum_{i=1}^{N} \left(S \sum_{k=0}^{N-1} \alpha_k z^k - p_i \cdot \sum_{k=0}^{N-1} \alpha_k z^{N-k} \right)}$$

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 polymonials 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

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" on page 4-12
- "Real Lowpass to Real Lowpass" on page 4-13
- "Real Lowpass to Real Highpass" on page 4-15
- "Real Lowpass to Real Bandpass" on page 4-17
- "Real Lowpass to Real Bandstop" on page 4-19
- "Real Lowpass to Real Multiband" on page 4-21
- "Real Lowpass to Real Multipoint" on page 4-23

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

$$\alpha = \begin{cases} \dfrac{\cos\frac{\pi}{2}(\omega_{old} - 2\omega_{new})}{\cos\frac{\pi}{2}\omega_{old}} & for \quad \left|\cos\frac{\pi}{2}(\omega_{old} - 2\omega_{new})\right| < 1 \\[2ex] \dfrac{\sin\frac{\pi}{2}(\omega_{old} - 2\omega_{new})}{\sin\frac{\pi}{2}\omega_{old}} & otherwise \end{cases}$$

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 example below 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);
```

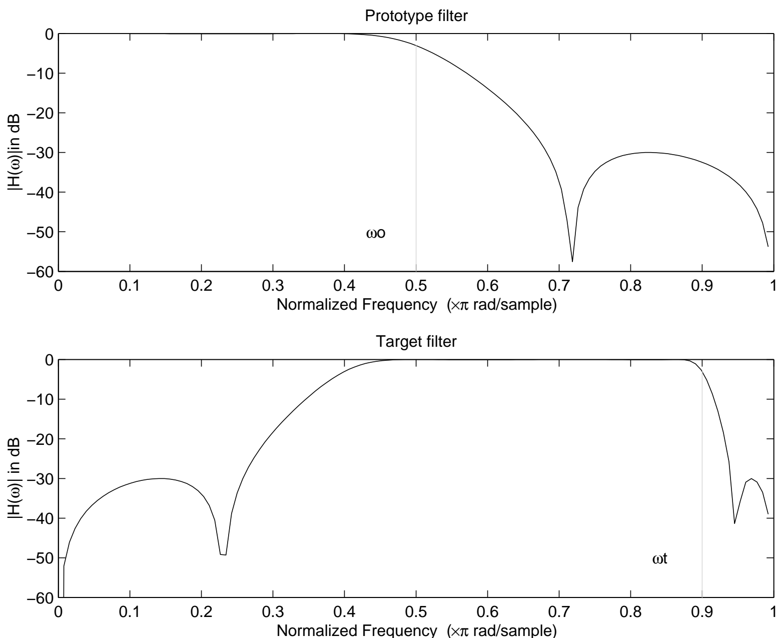


Figure 4-5: Example of Real Frequency Shift Mapping

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

$$\alpha = \frac{\sin\frac{\pi}{2}(w_{old} - w_{new})}{\sin\frac{\pi}{2}(w_{old} + w_{new})}$$

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 figure below.

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);
```

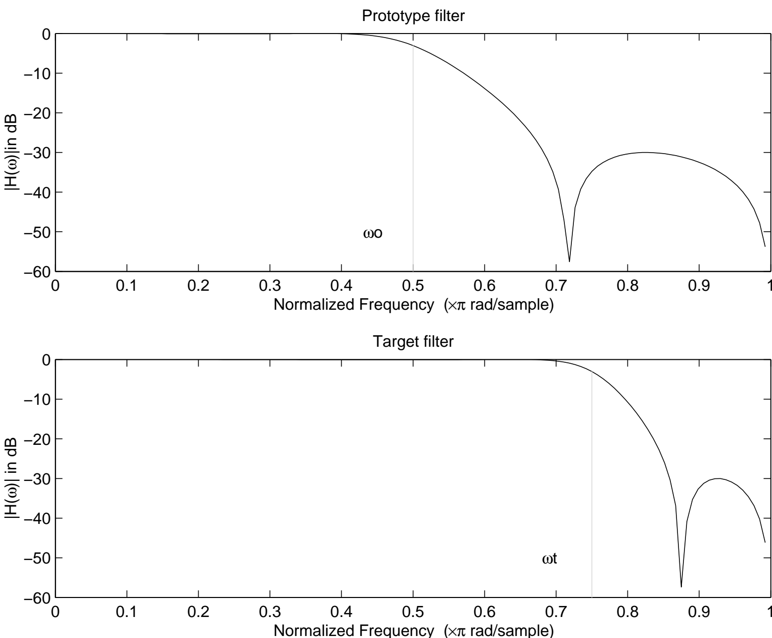


Figure 4-6: Example of Real Lowpass to Real Lowpass Mapping

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

$$\alpha = -\left(\frac{\cos\frac{\pi}{2}(w_{old} + w_{new})}{\cos\frac{\pi}{2}(w_{old} - w_{new})}\right)$$

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 from0.5 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] = iirlp2lp(b, a, 0.5, 0.75);
```

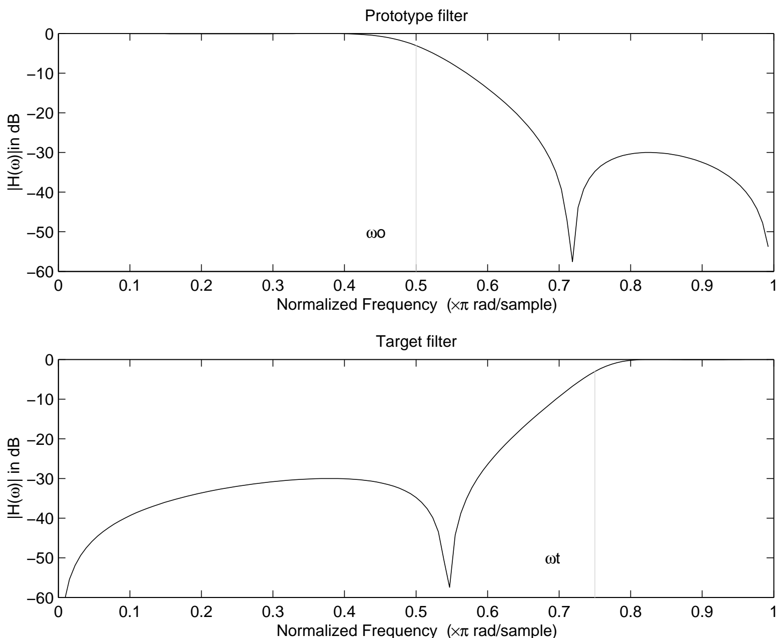


Figure 4-7: Example of Real Lowpass to Real Highpass Mapping

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

$$\alpha = \frac{\sin\frac{\pi}{4}(2w_{old} - w_{new,2} + w_{new,1})}{\sin\frac{\pi}{4}(2w_{old} + w_{new,2} - w_{new,1})}$$

$$\beta = \cos\frac{\pi}{2}(w_{new,1} + w_{new,2})$$

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]);
```

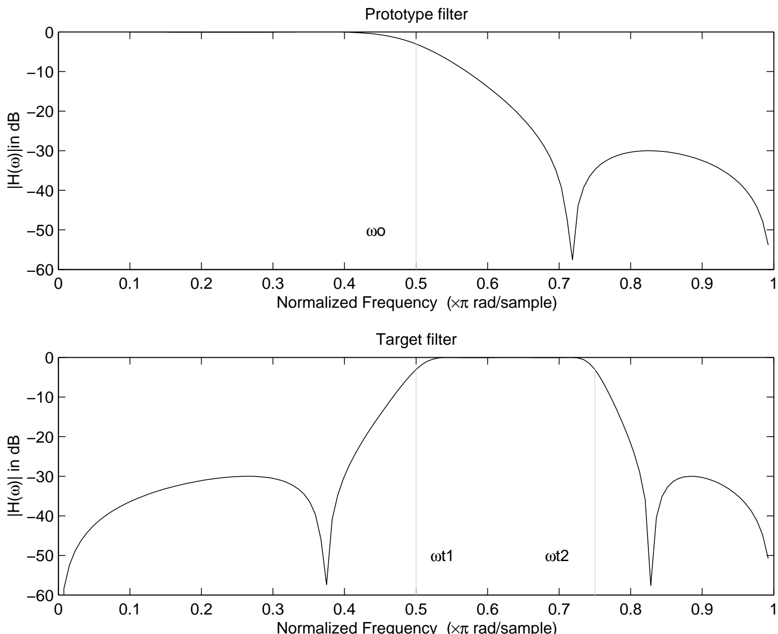


Figure 4-8: Example of Real Lowpass to Real Bandpass Mapping

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

$$\alpha = \frac{\cos\frac{\pi}{4}(2w_{old} + w_{new,2} - w_{new,1})}{\cos\frac{\pi}{4}(2w_{old} - w_{new,2} + w_{new,1})}$$

$$\beta = \cos\frac{\pi}{2}(w_{new,1} + w_{new,2})$$

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 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]);
```

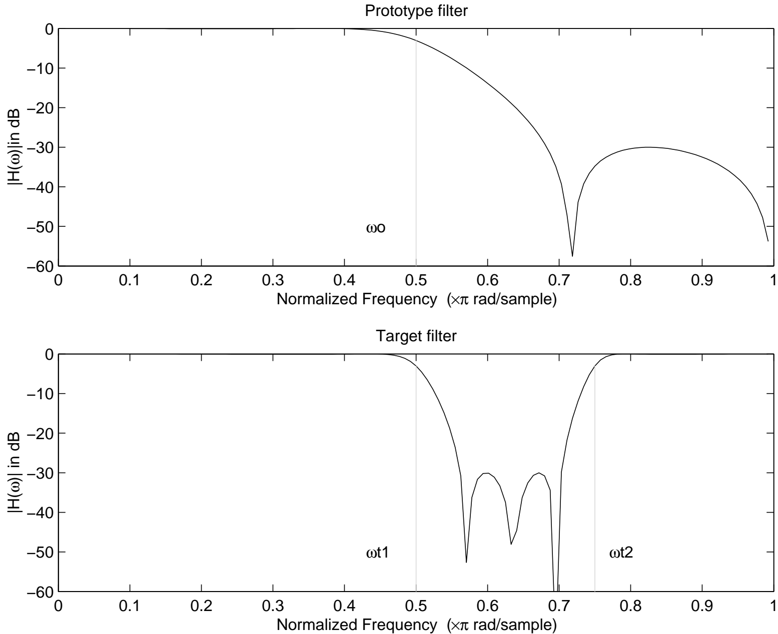


Figure 4-9: Example of Real Lowpass to Real Bandstop Mapping

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 passbands, 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 );
```

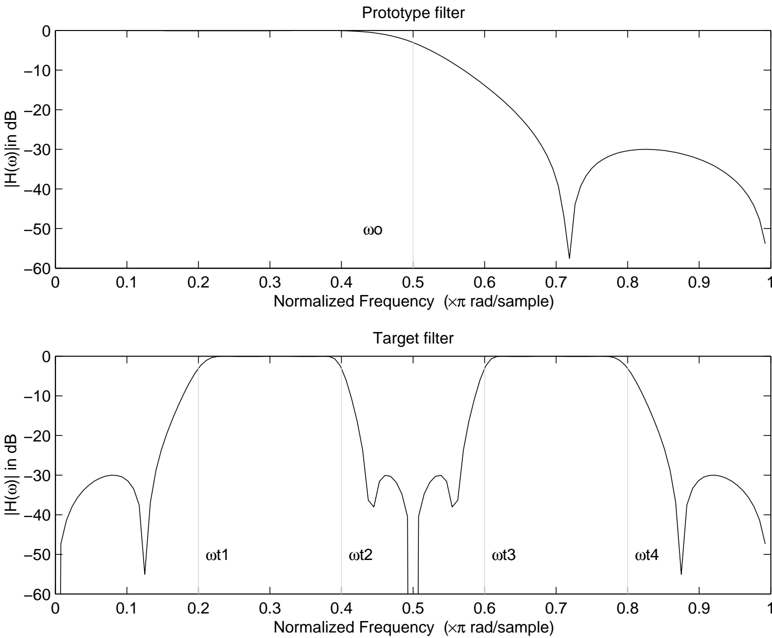


Figure 4-10: Example of Real Lowpass to Real Multiband Mapping

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 *N*th-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 );
```

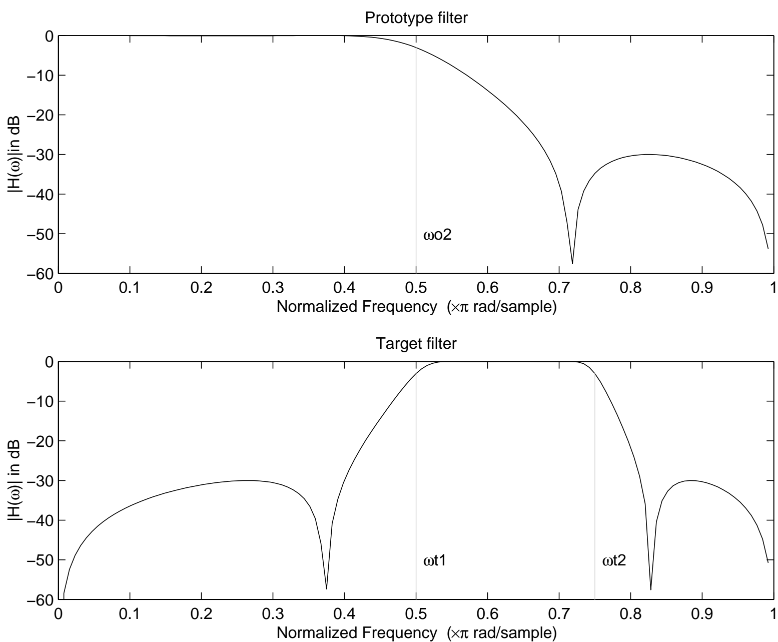


Figure 4-11: Example of Real Lowpass to Real Multipoint Mapping

Frequency Transformations for Complex Filters

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" on page 4-26
- "Real Lowpass to Complex Bandpass" on page 4-28
- "Real Lowpass to Complex Bandstop" on page 4-29
- "Real Lowpass to Complex Multiband" on page 4-31
- "Real Lowpass to Complex Multipoint" on page 4-33
- "Complex Bandpass to Complex Bandpass" on page 4-36

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 0.9:

```
[num,den] = iirshift(b, a, 0.5, 0.9);
```

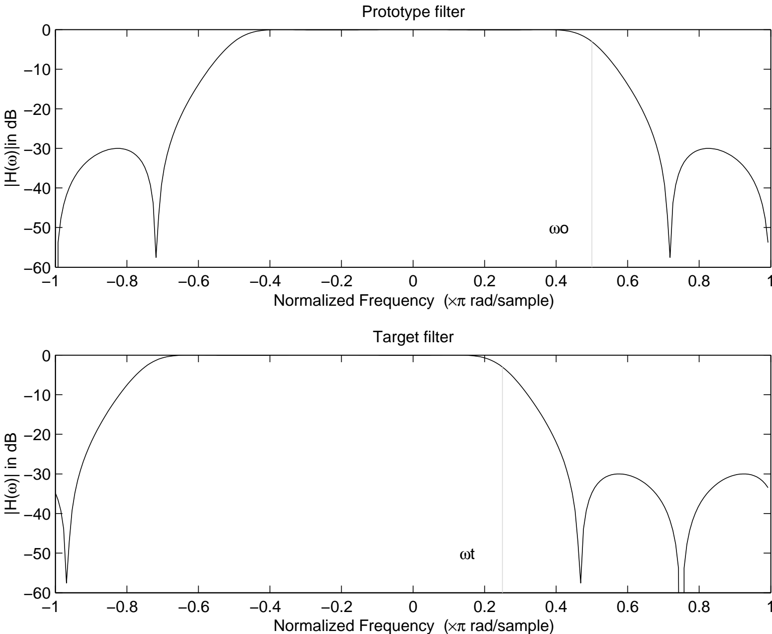


Figure 4-12: Example of Complex Frequency Shift Mapping

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

$$\alpha = \frac{\sin\frac{\pi}{4}(2w_{old} - w_{new,2} + w_{new,1})}{\sin\pi(2w_{old} + w_{new,2} - w_{new,1})}$$

$$\beta = e^{-j\pi(w_{new} - w_{old})}$$

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 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]);
```

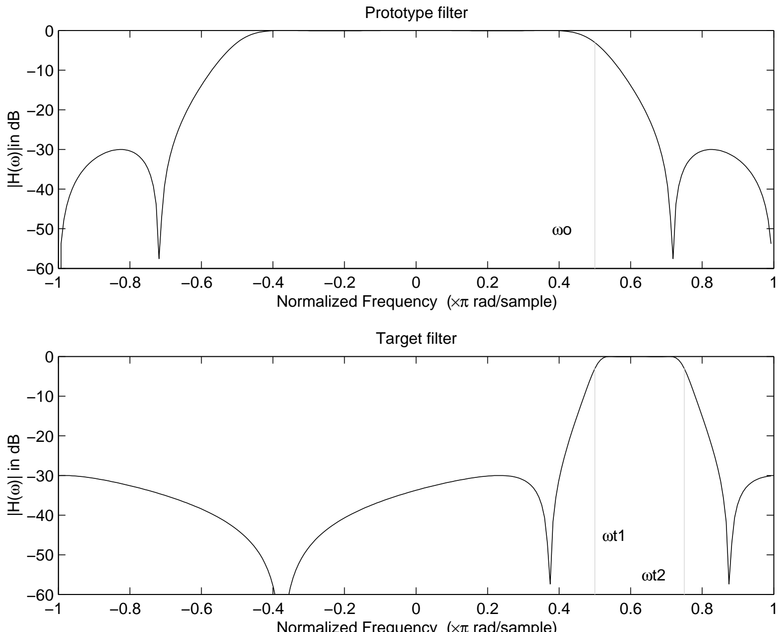


Figure 4-13: Example of Real Lowpass to Complex Bandpass Mapping

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

$$\alpha = \frac{\cos\pi(2w_{old} + \nu_{new,2} - \nu_{new,1})}{\cos\pi(2w_{old} - \nu_{new,2} + \nu_{new,1})}$$

$$\beta = e^{-j\pi(w_{new} - w_{old})}$$

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]);
```

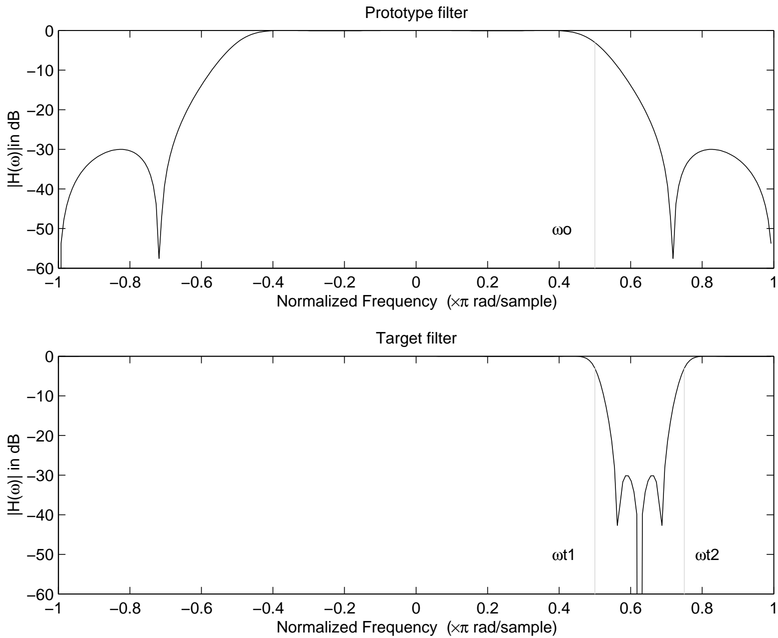


Figure 4-14: Example of Real Lowpass to Complex Bandstop Mapping

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 general transfer function form as shown below.

$$H_A(z) = S\frac{\sum_{i=0}^{N} \alpha_i z^{-i}}{\sum_{i=0}^{N} \alpha_i^* z^{-N+i}}$$

$$\alpha_0 = 1$$

The coefficients α are calculated from the system of linear equations:

$$\begin{cases} \sum_{i=1}^{N} \Re(\alpha_i)\cdot[\cos\beta_{1,k} - \cos\beta_{2,k}] + \Im(\alpha_i)\cdot[\sin\beta_{1,k} + \sin\beta_{2,k}] = \cos\beta_{3,k} \\ \sum_{i=1}^{N} \Re(\alpha_i)\cdot[\sin\beta_{1,k} - \sin\beta_{2,k}] - \Im(\alpha_i)\cdot[\cos\beta_{1,k} + \cos\beta_{2,k}] = \sin\beta_{3,k} \\ \beta_{1,k} = -\pi[\nu_{old}\cdot(-1)^k + \nu_{new,k}(N-k)] \\ \beta_{2,k} = -\pi[\Delta C + \nu_{new,k}k] \\ \beta_{3,k} = -\pi[\nu_{old}\cdot(-1)^k + \nu_{new,k}N] \\ k = 1\ldots N \end{cases}$$

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 Δ`C`, 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.

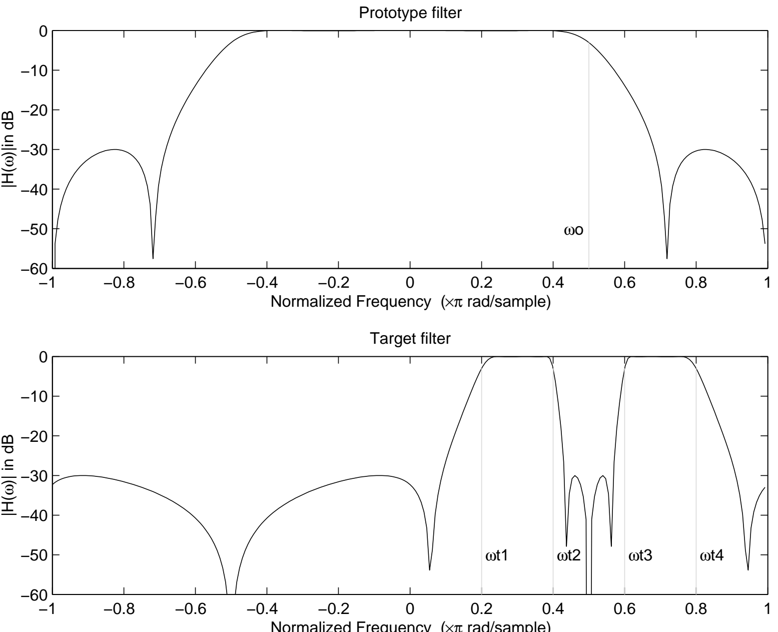


Figure 4-15: Example of Real Lowpass to Complex Multiband Mapping

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 Nth-order complex allpass mapping filter is given by the general transfer function form as shown below.

$$H_A(z) = S\frac{\sum_{i=0}^{N} \alpha_i z^{-i}}{\sum_{i=0}^{N} \alpha_i^* z^{-N+i}}$$

$$\alpha_0 = 1$$

The coefficients α can be calculated from the system of linear equations:

$$\begin{cases} \sum_{i=1}^{N} \Re(\alpha_i) \cdot [\cos\beta_{1,k} - \cos\beta_{2,k}] + \Im(\alpha_i) \cdot [\sin\beta_{1,k} + \sin\beta_{2,k}] = \cos\beta_{3,k} \\ \sum_{i=1}^{N} \Re(\alpha_i) \cdot [\sin\beta_{1,k} - \sin\beta_{2,k}] - \Im(\alpha_i) \cdot [\cos\beta_{1,k} + \cos\beta_{2,k}] = \sin\beta_{3,k} \\ \beta_{1,k} = -\frac{\pi}{2}[w_{old,k} + w_{new,k}(N-k)] \\ \beta_{2,k} = -\frac{\pi}{2}[2\Delta C + w_{new,k}k] \\ \beta_{3,k} = -\frac{\pi}{2}[w_{old,k} + w_{new,k}N] \\ k = 1 \ldots N \end{cases}$$

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 example below 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]);
```

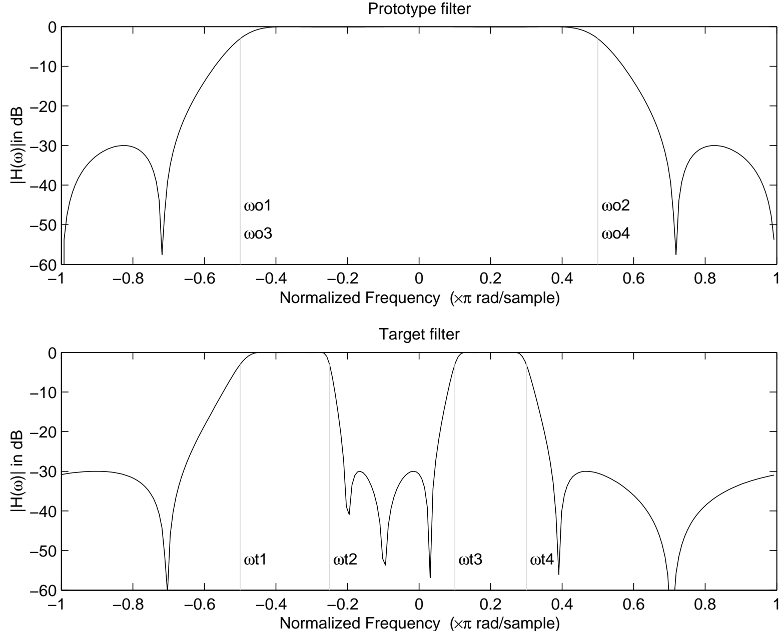


Figure 4-16: Example of Real Lowpass to Complex Multipoint Mapping

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

$$\alpha = \frac{\sin\frac{\pi}{4}((w_{old,2} - w_{old,1}) - (w_{new,2} - w_{new,1}))}{\sin\frac{\pi}{4}((w_{old,2} - w_{old,1}) + (w_{new,2} - w_{new,1}))}$$

$$\alpha = e^{-j\pi(w_{old,2} - w_{old,1})}$$

$$\gamma = e^{-j\pi(w_{new,2} - w_{new,1})}$$

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 example below 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]);
```

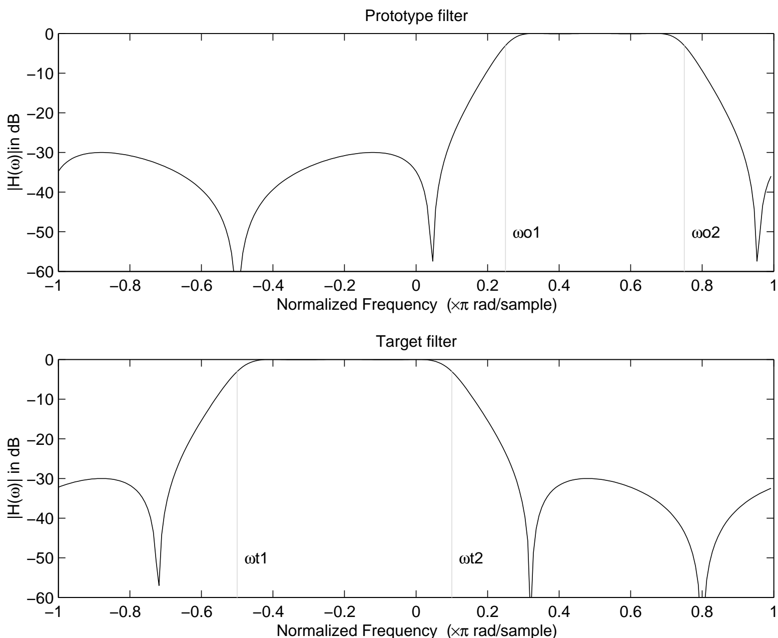


Figure 4-17: Example of Complex Bandpass to Complex Bandpass Mapping

5

Quantization and Quantized Filtering

Binary Data Types (p. 5-3)	Read this for an introduction to binary data types in the toolbox
Introductory Quantized Filter Example (p. 5-7)	To help you become familiar with quantized filters, this provides an example of using and analyzing a quantized filter
Fixed-Point Arithmetic (p. 5-16)	Reviews the fundamentals of fixed-point arithmetic and how the toolbox uses it
Floating-Point Arithmetic (p. 5-19)	Reviews the fundamentals of floating-point arithmetic and the data types in the toolbox

In the Filter Design Toolbox you can implement and analyze single-input single-output filters either as fixed-point filters, or as custom floating-point filters. Either type of filter is referred to as a *quantized filter*.

You can create a quantized filter from a reference filter, that is, a filter whose coefficients and arithmetic operations you want to quantize in some fashion.

You can also implement quantized FFTs and quantized inverse FFTs in this toolbox.

You can specify quantization parameters for quantized filters and FFTs with quantizers. Quantizers specify how data is quantized. You can also quantize any data set with a quantizer.

When you apply a quantized filter to data, not only are the filter coefficients quantized to your specification, but so are:

- The data that you filter
- The results of any arithmetic operations that occur during filtering

See “Bibliography” for a list of relevant references on quantized filtering.

Binary Data Types

Binary data is coded and stored as ones and zeros.

Binary Data for Fixed-Point Arithmetic

Binary data that is coded for fixed-point arithmetic is characterized by word length (in bits) and the placement of the radix (binary) point. The radix point placement determines the fraction length of a binary word, and also determines how the binary words are scaled. You can specify fixed-point words in this toolbox with word lengths up to the limits of memory on your PC. The fraction length can range from 0 bits (for integers) to one bit less than the word length.

Binary Data for Floating-Point Arithmetic

Binary data that is coded for floating-point arithmetic is characterized by the lengths of the fraction (mantissa) and the exponent (or equivalently, by the word length and the exponent length). In addition to having the ability to specify the standard IEEE single-precision and double-precision formats, you can specify filters in a custom floating-point format, with word lengths ranging from 2 to up to the limits of memory on your PC, and exponent lengths ranging from 0 to 11 bits.

Different data coding methods and precisions affect the following:

- The numerical range of the result
- The quantization error

You can use the Filter Design Toolbox to analyze quantized filters, quantized FFTs, or quantizers, and see how all of these factors affect your filter performance on data sets.

Digital Filters

Digital computers generate coded binary data. Binary data is usually coded in a fixed-point or floating-point format. You use digital filters to process binary data. Digital filters are modeled as discrete-time linear systems.

You can use digital filters to:

- Filter out noise in measurements

- Enhance signals
- Represent signals

Quantized Filter Types

You can specify any type of filter in this toolbox as a quantized filter:

- Fixed-point filters

  Fixed-point filters are useful for modeling fixed-point Digital Signal Processing (DSP) processors that operate on data using fixed-point arithmetic.
- Double-precision floating-point filters
- Single-precision floating-point filters
- Custom floating-point filters

  You can use custom floating-point filters in this toolbox to model floating-point DSP processors that operate on data using specific floating-point formats.

Quantized Filter Structures

The quantized filters you can implement in this toolbox can have any of the following structures:

- Direct form I
- Direct form I transposed
- Direct form II
- Direct form II transposed
- Direct form finite impulse response (FIR)
- Direct form FIR transposed filters
- Direct form antisymmetric FIR filters
- Direct form symmetric FIR filters
- Lattice allpass
- Lattice coupled-allpass filters
- Lattice moving average (MA) minimum phase filters
- Lattice MA maximum phase filters

- Lattice autoregressive (AR) filters
- Lattice ARMA filters
- Lattice coupled-allpass power complementary filters
- Single-input single-output state-space filters

Data Format for Quantized Filters

You can specify the precision and dynamic range for fixed-point filters with two *fixed-point data format parameters*:

- Word length
- Fraction length

The *word length* is the length in bits of any binary word. The *fraction length* is the length in bits of the binary word up to the radix point.

You can specify the precision, dynamic range, and other quantization parameters for floating-point filters with two *floating-point data format parameters*:

- Word length
- Exponent length

You can specify the precision, dynamic range, and other quantization parameters when you specify the data format properties. You can specify these properties using quantizers.

Except for when you specify a double- or single-precision quantized filter, you can specify the precision and dynamic range for each of the following quantization results individually:

- Inputs to a filtering operation
- Outputs of a filtering operation
- Quantized filter coefficients
- Sums that result from filtering
- Products that result from filtering
- Terms that are multiplied by filter coefficients (multiplicands)

These filter characteristics allow you to specify different quantization parameters for data and arithmetic instructions.

Quantized FFTs and Quantized Inverse FFTs

You can specify any type of radix two or radix four quantized FFT in this toolbox with the following data formats:

- Fixed-point FFTs
- Double-precision floating-point FFTs
- Single-precision floating-point FFTs
- Custom floating-point FFTs

The data formats for quantized FFTs are identical to those of quantized filters.

Introductory Quantized Filter Example

Follow the example in this section to:

- Construct a quantized filter.
- Plot the filter's poles and zeros.
- Plot the filter's impulse response.
- Plot the filter's frequency response from the quantized filter coefficients. The method used does not account for other quantization effects on the frequency response computation.
- Plot the filter's frequency response using the noise loading method. This method takes all quantization effects into account.
- Test the filter for limit cycles.

You can construct quantized filters by either:

- Using the quantized filter constructor function `qfilt`
- Copying a quantized filter from an existing one

Quantized filters have many properties, including the filter structure and the quantization formats.

When you use the function `qfilt` to create a quantized filter `Hq`, you can either:

- Type

  ```
  Hq = qfilt
  ```

  at the command line to accept the default filter properties, and change the property values later.
- Use a modified syntax for `qfilt` to set property values when you create `Hq`.

You can construct quantized filters with any of several filter structures.

Once you construct a filter, use the `filter` command to apply it to data. In addition, the following analysis functions apply to quantized filters:

- `zplane` for pole/zero plots
- `impz` for quantized impulse response plots
- `freqz` for computing and plotting the linear frequency response from the quantized filter coefficients

- `nlm` for estimating and plotting the frequency response using the noise loading method
- `limitcycle` for limit cycle detection and analysis

The first three of these Filter Design Toolbox functions are overloaded for quantized filters. They behave similarly to the functions with the same name in the Signal Processing Toolbox.

The introductory example presented in this section is included to illustrate some of the features of this toolbox. In this example, you can use the code presented to construct an eight-bit fixed-point quantized FIR filter, and analyze it with the response functions listed above.

To learn more about quantized filters, see Chapter 8, "Working with Quantized Filters" and Chapter 10, "Quantized Filtering Analysis Examples."

Constructing an Eight-Bit Quantized Filter

1. Use `gremez` to design an FIR low-pass filter in the frequency domain with a normalized cutoff frequency of approximately 0.4 radians/sample. Specify:
   - 27 filter coefficients
   - Four frequency points `[0 .4 .6 1]`
   - Four corresponding gains `[1 1 0 0]`

   ```
   b = gremez(27,[0 .4 .6 1],[1 1 0 0]);
   ```

   The entries in the vector `b` are the coefficients of the (numerator) polynomial of the FIR filter. This is your reference filter.

2. Construct a fixed-point quantized FIR filter from your reference filter with the following characteristics:
   - 8-bit word length for all data formats
   - 7-bit fraction length for all data formats
   - Direct form finite impulse response (FIR) filter structure
   - The `'convergent'` method used to round quantized numbers to the nearest allowable quantized value.

You can create quantized filters using the `qfilt` command. When you create a quantized filter, you must enter the vector of reference filter coefficients `b` in a cell array by enclosing the coefficients in curly braces, `{b}`.

```
Hq = qfilt('fir',{b},'Format',{[8,7],'convergent'})

Hq =
Quantized Direct form FIR filter.
Numerator
 QuantizedCoefficients{1}  ReferenceCoefficients{1}
0       ( 1)  0.0000000    0.001722275146612721
0       ( 2)  0.0000000    0.003409515867936453
        ( 3) -0.0078125   -0.004898115162792102
        ( 4) -0.0078125   -0.006325311495727597
        ( 5)  0.0078125    0.009418759615173328
        ( 6)  0.0156250    0.012524352890295399
        ( 7) -0.0156250   -0.017394015777896423
        ( 8) -0.0234375   -0.022634462311564768
        ( 9)  0.0312500    0.030838037214625479
        (10)  0.0390625    0.040914907859937441
        (11) -0.0546875   -0.057578619191314129
        (12) -0.0859375   -0.084552886463529736
        (13)  0.1484375    0.147259140040687880
        (14)  0.4453125    0.448759881582659110
        (15)  0.4453125    0.448759881582659110
        (16)  0.1484375    0.147259140040687880
        (17) -0.0859375   -0.084552886463529736
        (18) -0.0546875   -0.057578619191314129
        (19)  0.0390625    0.040914907859937441
        (20)  0.0312500    0.030838037214625479
        (21) -0.0234375   -0.022634462311564768
        (22) -0.0156250   -0.017394015777896423
        (23)  0.0156250    0.012524352890295399
        (24)  0.0078125    0.009418759615173328
        (25) -0.0078125   -0.006325311495727597
        (26) -0.0078125   -0.004898115162792102
0       (27)  0.0000000    0.003409515867936453
0       (28)  0.0000000    0.001722275146612721

   FilterStructure = fir
       ScaleValues = [1]
  NumberOfSections = 1
  StatesPerSection = [27]
```

```
 CoefficientFormat = unitquantizer('fixed', 'convergent',
'saturate', [8  7])
      InputFormat = quantizer('fixed', 'convergent', 'saturate',
[8  7])
     OutputFormat = quantizer('fixed', 'convergent', 'saturate',
[8  7])
MultiplicandFormat = quantizer('fixed', 'convergent', 'saturate',
[8  7])
    ProductFormat = quantizer('fixed', 'convergent', 'saturate',
[8  7])
        SumFormat = quantizer('fixed', 'convergent', 'saturate', [8  7])
```

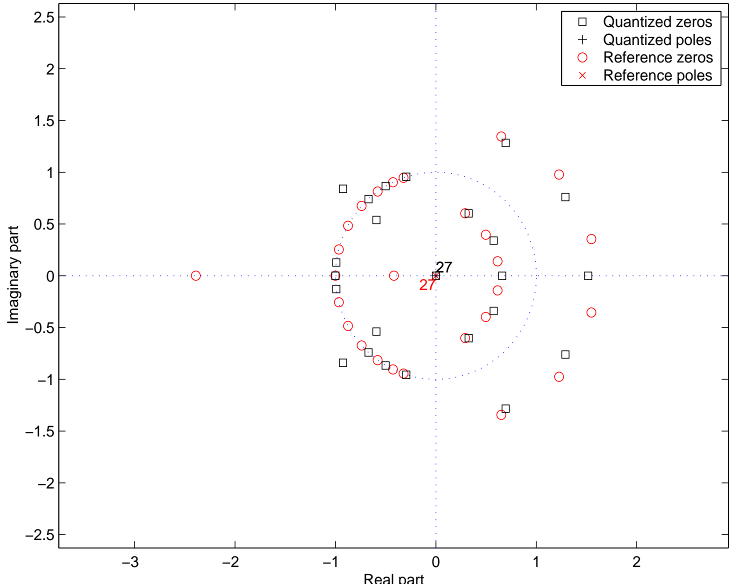
Notice that the display provides information about the filter and its property values. For this example, we created a filter whose product and sum quantizer formats are the same size as the coefficient format to illustrate the quantization effects.

Analyzing Poles and Zeros with zplane

To compare poles and zeros of the reference filter to those of the quantized filter `Hq` you just constructed, type

```
zplane(Hq)
```

Notice that the quantized zeros are not very close to the reference poles and zeros on the plot.

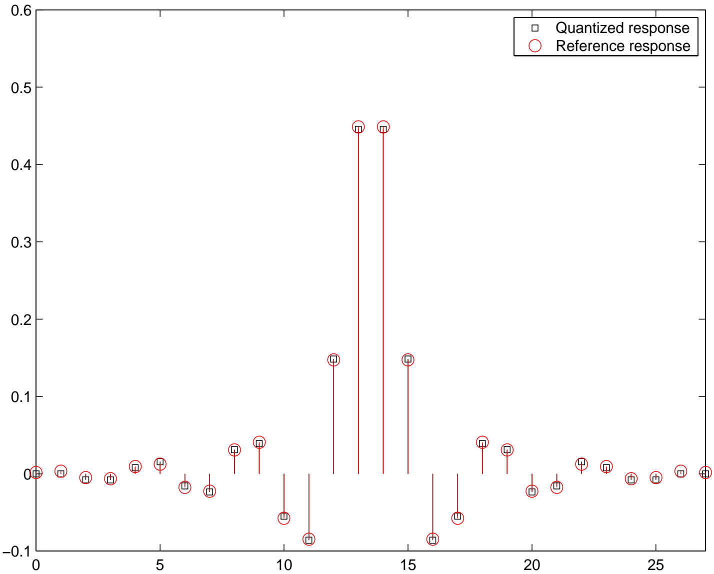


Analyzing the Impulse Response with impz

To compare the impulse response plot of the quantized filter `Hq` you just constructed to that of its floating-point reference (`b`), use the `impz` command.

```
impz(Hq)
```

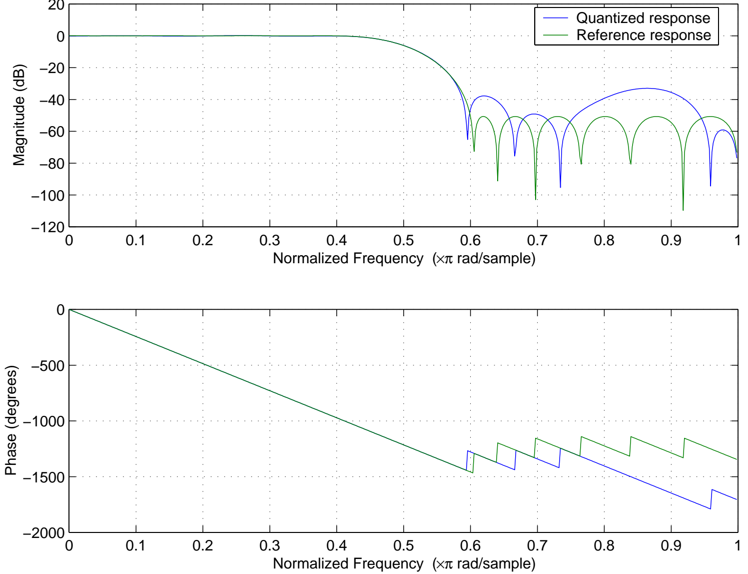
The impulse response computed by `impz` is the response of the fixed-point quantized filter `Hq` to a quantized impulse.



Analyzing the Frequency Response with freqz

To compare the frequency response plot of the quantized filter `Hq` you just constructed to that of its floating-point reference (`b`), use the `freqz` command.

```
freqz(Hq)
```

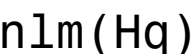


The `freqz` command computes the linear frequency response of the two filters whose coefficients are, respectively:

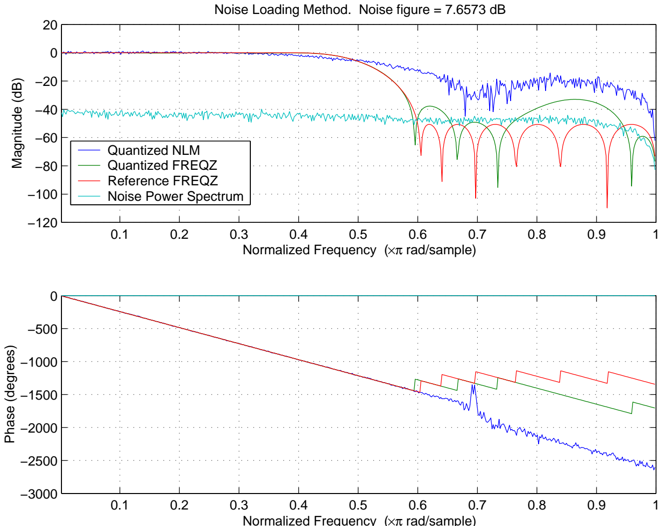
- The quantized filter coefficients
- The reference filter coefficients

Noise Loading Frequency Response Analysis: nlm

You can estimate the frequency response of the filter `Hq` you just created using the noise loading method computed with `nlm`. The noise loading method takes quantization effects into account. This method estimates the quantization noise figure when it runs a set of Monte Carlo frequency response calculations by filtering a set of sinusoids with randomly varying phase.



```
nlm(Hq)
```



Difference Between nlm and freqz for Frequency Response Analysis

The frequency response computed by `freqz` is determined using the true linear frequency response of the transfer function associated with the quantized filter coefficients. It does not take any other quantization effects into account, and is not computed from the filter structure you specify.

The frequency response computed by `nlm` is an estimate of the frequency response that accounts for nonlinear quantization effects due to your choice of:

- Filter structure
- Other quantization parameters

Analyzing Limit Cycles with limitcycle

You can analyze limit cycles of the filter `Hq` with `limitcycle`. This function computes a Monte Carlo simulation to detect the presence of limit cycles.

```
limitcycle(Hq)
```

```
No limit cycles detected after 20 Monte Carlo trials.
```

As is guaranteed for FIR filters, no limit cycles are detected for this model.

Fixed-Point Arithmetic

You can specify how numbers are quantized using fixed-point arithmetic in this toolbox with two quantities:

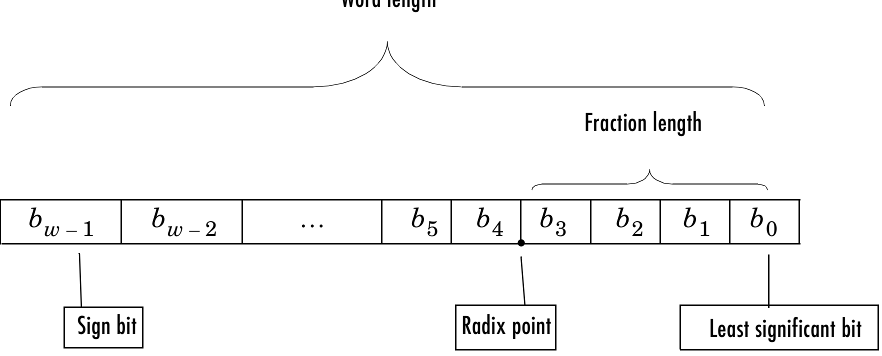
- Word length in bits
- Fraction length in bits

Note This toolbox does bit-true fixed-point arithmetic for word lengths of 53 bits and fewer. It simulates fixed-point arithmetic for word lengths greater than 53 bits, such as 64 bits.

Although the 64-bit fixed-point arithmetic is not be bit-true to the last bit, it properly handles overflows and the results are almost indistinguishable from bit-true results when the numbers are scaled properly. For example, (small numbers + small numbers) work correctly and (large numbers + large numbers) are right as well, but (large numbers + small numbers) will be dominated by the large number and some precision loss will occur.

Fraction length can be up to one bit less than the word length.

A general representation for a two's complement binary fixed-point number is



where:

- b_i are the binary digits (bits, zeros or ones).

- The word length in bits is given by w.
- The most significant bit (MSB) is the leftmost bit. It is represented by the location of b_{w-1}. In Filter Design Toolbox, this number represents the sign bit; a 1 indicates the number is negative, and a 0 indicates it is nonnegative.
- The least significant bit (LSB) is the rightmost bit, represented by the location of b_0.
- The radix (binary) point is shown four places to the left of the LSB for this example.
- The fraction length f is the distance from the LSB to the radix point.

Radix Point Interpretation

Where you place the radix point determines how fixed-point numbers are interpreted in two's complement arithmetic. For example, the five bit binary number:

- 10110 represents the integer $-2^4+2^2+2 = -10$.
- 10.110 represents $-2+2^{-1}+2^{-2} = -1.25$.
- 1.0110 represents $-2^{-0}+2^{-2}+2^{-3} = -0.625$.

Dynamic Range and Precision

A fixed-point quantization scheme determines the dynamic range of the numbers that can be applied to it. Numbers outside of this range are always mapped to fixed-point numbers within the range when you quantize them. The precision is the distance between successive numbers occurring within the dynamic range in a fixed-point representation. The dynamic range and precision depend on the word length and the fraction length.

For a signed fixed-point number with word length w and fraction length f, the range is from -2^{w-f-1} to $2^{w-f-1}-2^{-f}$.

For an unsigned fixed-point number with word length w and fraction length f, the range is from 0 to $2^{w-f}-2^{-f}$.

In either case the precision is 2^{-f}.

Overflows and Scaling

When you quantize a number that is outside of the dynamic range for your specified precision, *overflows* occur. Overflows occur more frequently with fixed-point quantization than with floating-point, because the dynamic range of fixed-point numbers is much less than that of floating-point numbers with equivalent word lengths.

Overflows can occur when you create a fixed-point quantized filter from an arbitrary floating-point design. You can normalize your fixed-point filter coefficients and introduce a corresponding scaling factor for filtering to avoid overflows in the coefficients.

In this toolbox you can specify how you want overflows to be handled:

- Saturate on the overflow
- Wrap on the overflow

Floating-Point Arithmetic

Fixed-point numbers are limited in that they cannot simultaneously represent very large or very small numbers using a reasonable word length. This limitation is overcome by using scientific notation. With scientific notation, you can dynamically place the radix point at a convenient location and use powers of the radix to keep track of that location. Thus, a range of very large and very small numbers can be represented with only a few digits.

Any binary floating-point number can be represented in floating-point using scientific notation form as $\pm F \times 2^{\pm E}$ where F is the fraction or mantissa (of length f), 2 is the radix or base (binary in this case), and E is the exponent of the radix (of length e). The floating-point word length w is f+e+1. The extra bit is for the sign bit.

You can specify single-precision and double-precision floating-point quantized filters with the Filter Design Toolbox. In addition, you can specify custom floating-point quantized filters with word lengths of up to 64 bits, and exponent lengths of up to 11 bits.

See `http://www.mathworks.com/company/newsletter/pdf/Fall96Cleve.pdf` for more information on floating-point computation.

Scientific Notation

A direct analogy exists between scientific notation and radix point notation. For example, scientific notation using five decimal digits for the mantissa would take the form

$$\pm d.dddd \times 10^{p} = \pm ddddd.0 \times 10^{p-4} = \pm 0.ddddd \times 10^{p+1}$$

where p is an integer of unrestricted range. Radix point notation using 5 bits for the mantissa is the same except for the number base

$$\pm b.bbbb \times 2^{q} = \pm bbbbb.0 \times 2^{q-4} = \pm 0.bbbbb \times 2^{q+1}$$

where q is an integer of unrestricted range. The previous equation is valid for both fixed- and floating-point numbers. For both these data types, the mantissa can be changed at any time by the processor. However, for fixed-point numbers, the exponent never changes, while for floating-point numbers, the exponent can be changed any time by the processor.

The IEEE Format

The IEEE 754 Standard for binary floating-point arithmetic has been widely adopted for use on DSP processors.

This standard specifies four floating-point number formats including single- and double-precision. Each format contains three components:

- Exponent
- Fraction
- Sign bit

The Exponent

In the IEEE format, exponent representations are biased. This means a fixed value (the bias) is subtracted from the field to get the true exponent value. For example, if the exponent field is 8 bits, then the numbers 0 through 255 are represented, and there is a bias of 127. Some values of the exponent are reserved for flagging `inf`, `NaN`, and denormalized numbers, so the true exponent values range from –126 to 127. If the exponent length is e, the bias is given by $2^{e-1}-1$.

The Fraction

In general, floating-point numbers can be represented in many different ways by shifting the number to the left or right of the radix point and decreasing or increasing the exponent of the radix by a corresponding amount. To simplify operations on these numbers, they are *normalized* in the IEEE format. A normalized binary number has a fraction with the form $1.F$ where F has a fixed size for a given data type. Since the leftmost fraction bit is always a 1, it is unnecessary to store this bit and is therefore implicit (or hidden). Thus, an n-bit fraction stores an n+1-bit number. If the exponent length is e and the word length is w, then the fraction length $f = w-e-1$. IEEE also supports denormalized numbers.

The Sign Bit

IEEE floating-point numbers use a sign/magnitude representation where the sign bit is explicitly included in the word. Using this representation, a sign bit of 0 represents a positive number and a sign bit of 1 represents a negative number. Both the fraction and the exponent can be positive or negative, but

only the fraction has a sign bit. The sign of the exponent is determined by the bias.

Single-Precision Format

The IEEE 754 single precision floating-point format is a 32-bit word divided into a 1-bit sign indicator s, an 8-bit biased exponent E, and a 24-bit fraction F. A representation of this format is given below.

b_{31}	b_{30} ...	b_{22} ... b_0
s	E	F

The relationship between this format and the representation of real numbers is given below.

Number Characterization	**Value**
Normalized, $0<E<255$	$(-1)^s(2^{E-127})(1.F)$
Denormalized, $E=0$; $F\neq0$	$(-1)^s(2^{-126})(0.F)$
Zero, $E=0$; $F=0$	$(-1)^s(0)$
Otherwise	exceptional value

Denormalized values are discussed in “Exceptional Arithmetic” on page 5-24.

Double-Precision Format

The IEEE 754 double precision (64-bit) floating-point format consists of a 1-bit sign indicator s, an 11-bit biased exponent E, and a 52-bit fraction F.
A representation of this format is given below.

b_{63}	b_{62} ...	b_{51} ... b_0
s	E	F

The relationship between this format and the representation of real numbers is given below.

Number Characterization	Value
Normalized, $0<E<2047$	$(-1)^s(2^{E-1023})(1.F)$
Denormalized, $E=0$; $F \neq 0$	$(-1)^s(2^{-1022})(0.F)$
Zero, $E=0$; $F=0$	$(-1)^s(0)$
Otherwise	exceptional value

Denormalized values are discussed in "Exceptional Arithmetic" on page 5-24.

Custom Floating-Point Data Types

This toolbox supports custom (nonstandard) IEEE-style floating-point data types. These data types adhere to the definitions and formulas previously given for IEEE single- and double-precision numbers.

The fraction length and the bias for the exponent are calculated from the word length and exponent length you supply. You can specify:

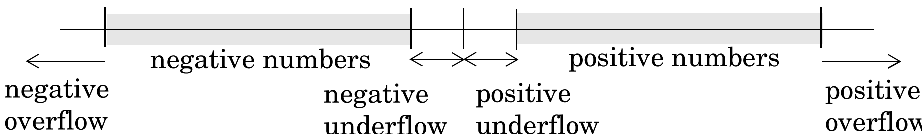
- Any exponent length up to 11 bits
- Any word length greater than the exponent length up to 64 bits

When specifying a custom format, keep in mind that the exponent length largely determines the dynamic range, while the fraction length largely determines the precision of the result.

Dynamic Range

A floating-point quantization scheme determines the dynamic range of the numbers that can be applied to it. Numbers outside of this range are always mapped to `±inf`.

The range of representable numbers for an IEEE floating-point number with word length w, exponent length e, fraction length $f = w–e–1$, and the exponent bias given by $bias = 2^{e-1} - 1$ is described in the following diagram



where:

- Normalized positive numbers are defined within the range 2^{1-bias} to $(2 - 2^{-f}) \cdot 2^{bias}$.
- Normalized negative numbers are defined within the range -2^{1-bias} to $-(2 - 2^{-f}) \cdot 2^{bias}$.
- Positive numbers greater than $(2 - 2^{-f}) \cdot 2^{bias}$, and negative numbers greater than $-(2 - 2^{-f}) \cdot 2^{bias}$ are called *overflows*.
- Positive numbers less than 2^{1-bias}, and negative numbers less than -2^{1-bias} are either *underflows* or denormalized numbers.
- Zero is specified by a E=0; F=0.

Overflows and underflows result from exceptional arithmetic conditions. Exceptional arithmetic is discussed “Exceptional Arithmetic” on page 5-24.

Note You can use the MATLAB functions `realmin` and `realmax` to determine the dynamic range of double-precision floating-point values for your computer.

Precision

The precision is the distance between 1.0 and the next largest floating-point number. The dynamic range and precision depend on the word length and the exponent length.

The precision for floating-point numbers is 2^{-f}.

Note In MATLAB, floating-point relative accuracy is given by the command `eps` which returns the distance from 1.0 to the next largest floating-point number. For computers that support the IEEE standard for floating-point numbers, `eps` = 2^{-52} or 2.2204×10^{-16}.

Floating-Point Data Type Parameters

The maximum and minimum absolute values, exponent bias, and precision for the floating-point formats supported by this toolbox are given below.

Table 5-1: Floating-Point Data Type Parameters

Floating-Point Data Type	Normalized Minimum	Maximum	Exponent Bias	Precision
Single	$2^{-126} \approx 10^{-38}$	$(2-2^{-23})2^{127} \approx 3(10^{38})$	127	$2^{-23} \approx 10^{-7}$
Double	$2^{-1022} \approx 2(10^{-308})$	$(2-2^{-52})2^{1023} \approx 1.7(10^{308})$	1023	$2^{-52} \approx 10^{-16}$
Custom	2^{1-bias}	$(2-2^{-f})2^{bias}$	$2^{e-1}-1$	2^{-f}

Due to the sign/magnitude representation of floating-point numbers, there are two representations of zero, one positive and one negative. For both representations $E = 0$ and $F = 0$.

Exceptional Arithmetic

In addition to specifying a floating-point format, the IEEE 754 Standard for binary floating-point arithmetic specifies practices and procedures so that predictable results are produced independent of the hardware platform. Specifically, denormalized numbers, are defined to deal with exceptional arithmetic (underflow and overflow).

Denormalized Numbers

Denormalized numbers are used to handle cases of exponent underflow. When the exponent of the result is too small (such as a negative exponent whose magnitude is too large), the result is denormalized by right-shifting the fraction and leaving the exponent at its minimum value. Using denormalized numbers is also referred to as gradual underflow. Without denormalized

numbers, the gap between the smallest representable nonzero number and zero is much wider than the gap between the smallest representable nonzero number and the next larger number. Gradual underflow fills that gap and reduces the impact of exponent underflow to a level comparable with roundoff among the normalized numbers. Thus, denormalized numbers provide extended range for small numbers at the expense of precision.

For more information about denormalized single- and double-precision numbers, refer to "Single-Precision Format" on page 5-21 and "Double-Precision Format" on page 5-21.

6

Working with Objects

Objects for Quantized Filtering (p. 6-2)	Describes the objects the toolbox uses for quantized filtering
Properties and Property Values (p. 6-5)	Outlines the properties of the quantized filter objects
Functions Acting on Objects (p. 6-11)	Lists the functions (methods) that apply to quantized filter objects
Using Command Line Help (p. 6-12)	Describes how to get help on quantized objects, properties, and methods
Using Cell Arrays (p. 6-14)	Provides information on using cell arrays, which are common when you use the objects in the toolbox

Objects for Quantized Filtering

The Filter Design Toolbox uses objects to create:

- Quantizers
- Quantized filters
- Quantized FFTs

Concepts you need to know about the objects for quantized filtering in this toolbox are covered in these sections:

- “Constructing Objects”
- “Copying Objects to Inherit Properties”
- “Properties and Property Values”
- “Setting and Retrieving Property Values”
  - “Setting Property Values Directly at Construction”
  - “Setting Property Values with the set Command”
  - “Retrieving Properties with the get Command”
  - “Direct Property Referencing to Set and Get Values”
- “Functions Acting on Objects”
- “Using Command Line Help”
- “Using Cell Arrays”
  - “Indexing into a Cell Array of Vectors or Matrices”
  - “Indexing into a Cell Array of Cell Arrays”

Note Although the examples in this section use quantized filters, the techniques discussed here apply to quantizers and quantized FFTs. See “MATLAB Classes and Objects” in your MATLAB documentation for more details on object-oriented programming in MATLAB.

Constructing Objects

You use one of the two methods Filter Design Toolbox offers to construct objects:

- Use the object constructor function
- Copy an existing object

For example, when you create a quantized filter using the `qfilt` command, you are creating a *Qfilt object*. The Qfilt object implementation relies on MATLAB object-oriented programming capabilities.

Like other MATLAB structures, objects in this toolbox have predefined fields called *object properties*.

You specify object property values by either:

- Specifying the property values when you create the object
- Creating an object with default property values, and changing some or all of these property values later

For examples of setting quantized filter properties, see "Quantized Filter Properties" on page 8-6.

Example — Constructor for Quantized Filters

The easiest way to create a quantized filter (qfilt object) is to create one with the default properties. You can create a quantized filter `Hq` by typing

```
Hq = qfilt
```

MATLAB lists the properties of the filter `Hq` you created along with the associated default property values.

```
Quantized Direct form II transposed filter
Numerator
     QuantizedCoefficients{1}    ReferenceCoefficients{1}
+ (1)       0.999969482421875  1.000000000000000000
Denominator
     QuantizedCoefficients{2}    ReferenceCoefficients{2}
+ (1)       0.999969482421875  1.000000000000000000

   FilterStructure = df2t
       ScaleValues = [1]
  NumberOfSections = 1
```

```
  StatesPerSection = [0]
 CoefficientFormat = quantizer('fixed', 'round', 'saturate', [16  15])
       InputFormat = quantizer('fixed', 'floor', 'saturate', [16  15])
      OutputFormat = quantizer('fixed', 'floor', 'saturate', [16  15])
MultiplicandFormat = quantizer('fixed', 'floor', 'saturate', [16  15])
     ProductFormat = quantizer('fixed', 'floor', 'saturate', [32  30])
         SumFormat = quantizer('fixed', 'floor', 'saturate', [32  30])
Warning: 2 overflows in coefficients.
```

The properties of this filter are described in Table 12-3, Quick Guide to Quantized Filter Properties, on page 12-10, and in more detail in “Quantized Filter Properties Reference” on page 12-11. All of these properties are set to default values when you construct them.

For information on quantizer properties, see “A Quick Guide to Quantizer Properties” on page 12-2 or “Quantizer Properties Reference” on page 12-3 for more details.

For information on quantized FFT properties, see “A Quick Guide to Quantized FFT Properties” on page 12-51, or “Quantized FFT Properties Reference” on page 12-52 for more details.

Copying Objects to Inherit Properties

If you already have an object with the property values set the way you want them, you can create a new one with the same property values by copying the first object.

This feature is convenient to use when you want to change a small number of properties on a set of objects.

Example — Copying Quantized Filters to Inherit Properties

To create a new quantized filter `Hq2` with the same property values as an existing quantized filter `Hq`, type

```
Hq2 = copyobj(Hq);
```

Properties and Property Values

All objects in this toolbox have properties associated with them. Each property associated with an object is assigned a value. You can set the values of most properties. However, some properties have read-only values.

To learn about properties that are specific to quantized filters, see "Quantized Filter Properties" on page 8-6.

To learn about properties that are specific to quantizers, see "Quantizer Properties Reference" on page 12-3.

To learn about properties that are specific to quantized FFTs, see "Quantized FFT Properties Reference" on page 12-52.

Setting and Retrieving Property Values

You can set Filter Design Toolbox object property values:

- Directly when you create the object
- By using the `set` command with an existing object

You can retrieve quantized filter property values using the `get` command.

In addition, direct property referencing lets you either set or retrieve property values.

Setting Property Values Directly at Construction

To set property values directly when you construct an object, simply include the following in the argument list for the object construction command:

- A string for the property name you want to set followed by a comma
- The associated property value. Sometimes this value is also a string

Include as many property names in the argument list for the object construction command as there are properties you want to set directly.

Example — Setting Quantized Filter Property Values at Construction

Suppose you want to set the following filter characteristics when you create a fixed-point quantized filter:

- The filter structure has a direct form II transposed structure

- The reference filter transfer function has numerator [1 .5] and denominator [1 .7 .89]

Do this by typing

```
Hq = qfilt('FilterStructure','df2t','ReferenceCoefficients',...
     {[1 .5] [1 .7 .89]});
```

These properties are described in "Quantized Filter Properties Reference" on page 12-11.

Note When you set any object property values, the strings for property names and their values are case-insensitive. In addition, you only need to type the shortest uniquely identifying string in each case. For example, you could have typed the above code as

```
Hq = qfilt('filt','df2t','ref',{[1 .5] [1 .7 .89]});
```

Setting Property Values with the set Command

Once you construct an object, you can modify its property values using the `set` command.

You can use the `set` command to both:

- Set specific property values
- Display a listing of all property values you can set

Example — Setting Fixed-Point Quantized Filter Property Values Using set

For example, set the following specifications for the fixed-point filter `Hq` you just created:

- Set the input quantization format to `[24 23]`
- Set the filter structure to a direct form I structure

To do this, type

```
set(Hq,'inputformat',[24 23],'filterstructure','df1')
Hq.input.format
```

```
ans =
    24    23

Hq.filt

ans =
df1
```

Notice how the display reflects the changes in the property values.

To display a listing of all of the properties associated with a quantized filter `Hq` that you can set, type

```
set(Hq)

QuantizedCoefficients: Quantized from reference coefficients.
ReferenceCoefficients: Cell array of coefficients.  One cell per
section.
                     {num,den} | {{num1,den1},{num2,den2},...} |
                       {num} | {{num1},{num2},...} |
                       {k} | {{k1},{k2},...} |
                       {k,v} | {{k1,v1},{k2,v2},...} |
                       {k1,k2,beta} |
{{k11,k21,beta1},{k12,k22,beta2},...} |
                   {A,B,C,D} | {{A1,B1,C1,D1},{A2,B2,C2,D2},...}
      FilterStructure: [df1 | df1t | df2 | <df2t> | fir | firt |
                        symmetricfir | antisymmetricfir |
                        latticear | latcallpass |
                        latticema | latcmax | latticearma |
                        latticeca | latticecapc | statespace]
          ScaleValues: Vector of scale values between sections.
    CoefficientFormat: quantizer
          InputFormat: quantizer
         OutputFormat: quantizer
   MultiplicandFormat: quantizer
        ProductFormat: quantizer
            SumFormat: quantizer
```

Retrieving Properties with the get Command

You can use the `get` command to:

- Retrieve property values for an object
- Display a listing of all the properties associated with an object and the associated property values

Example — Retrieving Quantized Filter Property Values

For example, to retrieve the value of the quantization data format for the input, type

```
v = get(Hq,'FilterStructure')

v =
df1
```

Note When you retrieve properties, the strings for property names and their values are case-insensitive. In addition, you only need to type the shortest uniquely identifying string in each case. For example, you could have typed the above code as

```
v = get(Hq,'filt');
```

Note To display a listing of the properties of a quantized filter `Hq`, and their values, type

---

```
get(Hq)

Quantized Direct Form I (df1) filter.
Numerator
 QuantizedCoefficients{1}    ReferenceCoefficients{1}
    (1) 1.000000000000000    1.000000000000000000
    (2) 0.500000000000000    0.500000000000000000
```

```
Denominator
 QuantizedCoefficients{2}    ReferenceCoefficients{2}
    (1) 1.000000000000000    1.000000000000000000
    (2) 0.699981689453125    0.699999999999999960
    (3) 0.889984130859375    0.890000000000000010

   FilterStructure = df1
       ScaleValues = [1]
  NumberOfSections = 1
  StatesPerSection = [3]
 CoefficientFormat = unitquantizer('fixed', 'floor', 'saturate',
[16  15])
      InputFormat = quantizer('fixed', 'floor', 'saturate', [24
23])
     OutputFormat = quantizer('fixed', 'floor', 'saturate', [16
15])
MultiplicandFormat = quantizer('fixed', 'floor', 'saturate', [16
15])
    ProductFormat = quantizer('fixed', 'floor', 'saturate', [32
30])
        SumFormat = quantizer('fixed', 'floor', 'saturate', [32
30])
```

Direct Property Referencing to Set and Get Values

You can reference directly into a property for setting or retrieving property values using MATLAB structure-like referencing. You do this by using a period to index into a property by name.

Example — Direct Property Referencing in Quantized Filters

For example:

1 Create a filter with default values.

2 Change its reference filter coefficients.

```
Hq = qfilt;
Hq.ref = {[1 .5],[1 .7 .89]};
```

Notice that you do not have to type the full name of the `ReferenceCoefficients` property, and you can use lower case to refer to the property.

To retrieve any property values, you can also use direct property referencing.

```
v = Hq.ref

v =
    [1x2 double]    [1x3 double]
```

Notice that `v` is a cell array, and you need to index into it to retrieve its values. See “Using Cell Arrays” on page 6-14 for help about indexing into cell arrays.

Functions Acting on Objects

Several functions in this toolbox have the same name as functions in the Signal Processing Toolbox or in MATLAB. These Filter Design Toolbox functions behave similarly to their original counterparts, but you apply these functions directly to an object. This concept of having functions with the same name operate on different types of objects (or on data) is called *overloading* of functions.

For example, the `filter` command is overloaded for quantized filters (qfilt objects). Once you specify your quantized filter by assigning values to its properties, you can apply many of the functions in this toolbox (such as `freqz` for frequency response analysis) directly to the variable name you assign to your quantized filter, without having to specify filter parameters again.

For a complete list of the functions that act on quantizers, see “Functions Operating on Quantizers” on page 13-12.

For a complete list of the functions that act on quantized filters, see “Functions Operating on Quantized FFTs” on page 13-14.

For a complete list of the functions that act on quantized FFTs, see “Functions Operating on Quantized FFTs” on page 13-14.

Using Command Line Help

How you get command line help on a function depends on whether the function is overloaded.

Command Line Help For Nonoverloaded Functions

You can use the usual syntax for getting command line help on functions that are not overloaded.

Type

```
help FuncName
```

to get command line help on functions in this toolbox that are not overloaded.

Command Line Help For Overloaded Functions

Because many of the toolbox functions are overloaded, you need to refer to the object name when you are trying to get command line help for overloaded functions.

Command Line Help for Overloaded Functions on Quantized Filters

To get command line help for an overloaded function `MethodName` that operates on quantized filters (Qfilt objects), type

```
help qfilt/MethodName
```

Similarly, for command line help on overloaded methods for quantizers or quantized FFTs, type

```
help quantizer/MethodName
help qfft/MethodName
```

For example, to get help on the `zplane` function in this toolbox, type

```
help qfilt/zplane
```

You can find a list of the overloaded functions for quantized filters in “Functions Operating on Quantized FFTs” on page 13-14.

You can find a list of the overloaded functions for quantizers, in “Functions Operating on Quantizers” on page 13-12.

You can find a list of the overloaded functions for quantized FFTs, in "Functions Operating on Quantized FFTs" on page 13-14.

Note Many of the toolbox functions are overloaded. MATLAB does not necessarily display the appropriate help text for a given object command `MethodName` when you type

```
help MethodName
```

To get the appropriate help for an overloaded function, you may need to specify the type of object to which you are applying the function. For example,

```
help qfilt/MethodName
help qfft/MethodName
```

Using Cell Arrays

The syntax for constructing quantized filters requires you to enter the reference filter coefficients as cell arrays.

Cell arrays can store any type of data: strings, vectors, matrices, cell arrays, and so forth. You specify a cell array using curly braces (`{}`). You need to use these braces to index into a cell array to retrieve its contents.

When you index into a cell array you use one set of braces to index into each layer of a cell array.

For details on constructing and using quantized filters in this toolbox, see Chapter 8, "Working with Quantized Filters." For detailed information on cell arrays, see *Using MATLAB*.

The next sections provide guidance and examples of how to index into a cell array:

- "Indexing into a Cell Array of Vectors or Matrices" on page 6-14
- "Indexing into a Cell Array of Cell Arrays" on page 6-15

Indexing into a Cell Array of Vectors or Matrices

To index into a cell array of matrices (as opposed to a cell array of cell arrays), you only need one set of braces to index into the cell array.

Here's an example of accessing cell array information from a quantized filter with a single section. In this case, the filter coefficient information is stored as a cell array of vectors.

Example — Accessing Coefficient Information from Filters with One Section

You can specify a sample quantized filter by typing

```
Hq = qfilt('ref',{[1 .5],[1 .7 .89]});
Hq.ReferenceCoefficients

ans =
     [1x2 double]    [1x3 double]
```

Notice that the filter reference coefficients are stored in a two-by-one cell array of vectors, the way you specified them.

Suppose that you want to retrieve the values stored in this property.

Use curly braces to index into and access the first entry of the cell array `Hq.ReferenceCoefficients`. You can use the shorthand for property names when you index into the properties of `Hq`.

```
Hq.ref{1}

ans =
    1.0000    0.5000
```

Similarly,

```
Hq.ref{2}

ans =
    1.0000    0.7000    0.8900
```

To access the third entry in `Hq.ref{2}`, index into `Hq.ref{2}` in the standard way.

```
Hq.ref{2}(3)

ans =
    0.8900
```

Indexing into a Cell Array of Cell Arrays

To index into a cell array of cell arrays, you have to use as many sets of braces as you have layers of cells.

Here's an example of indexing into the cell arrays of multisection quantized filters.

Example — Accessing Coefficient Information from Multisection Filters

When you create quantized filters with multiple sections, specify the reference filter coefficients as a cell array of cell arrays, using one cell array to enter the numerator and denominator of each section. In this case, use sequences of curly braces to index into these cell arrays.

For example, suppose you want to quantize and design a sixth-order Butterworth filter you create using the Signal Processing Toolbox.

```
[b,a] = butter(6,.5);
```

Filters whose transfer functions are factored into second-order sections are much more robust against quantization error, so use `sos` to put your direct form II filter into a second-order sections form.

```
Hq = sos(qfilt('df2',{b,a}));
Hq.ReferenceCoefficients

ans =
    {1x2 cell}    {1x2 cell}    {1x2 cell}
```

The reference coefficients are contained in a three-by-one cell array of cells `Hq.ReferenceCoefficients`. This cell array is created from the values you set for the `ReferenceCoefficients` property. You can index into one of the three cell arrays of cells by:

1 Creating a cell array c from the cell array `Hq.ReferenceCoefficients`

2 Indexing into it

```
c = Hq.ref;
c{2}{1:2}

ans =
    0.2500    0.5012    0.2511
ans =
    1.0000    0.0000    0.1716
```

Notice that you can use the colon operator to obtain the contents of both entries in the cell array contained in the cell array `c{2}`.

Note You do not have to create another cell array to index into the reference coefficients data for one section of the filter. You do have to create another cell array if you want to index into multiple entries of the cell array, as in this example.

7

Working with Quantizers

Quantizers and Unit Quantizers (p. 7-2)	Describes the two kinds of quantizers in the toolbox
Constructing Quantizers (p. 7-3)	Explains how to create quantizer objects
Quantizer Properties (p. 7-4)	Outlines the properties of the quantizer objects
Quantizing Data with Quantizers (p. 7-6)	Talks about using quantizers to quantize data—how and what quantizing data does
Transformations for Quantized Data (p. 7-8)	Offers a brief explanation of transforming quantized data between representations, such as hex
Quantizer Data Functions (p. 7-9)	Introduces the functions in the toolbox that work on quantized data

Quantizers and Unit Quantizers

There are two types of quantizers you can construct in this toolbox:

- Quantizers
- Unit quantizers

These two types of quantizers are the same, except that *unit quantizers* quantize any number within the quantization level (`eps(q)`) of 1 to 1, where `q` is a quantizer.

You can construct quantizers to specify quantization parameters you want to use when you quantize data sets. You can also use quantizers for:

- Specifying data formats for quantized filters or FFTs
- Obtaining information about the data sets you quantize

This chapter covers quantizer-specific information:

- Constructing quantizers
- Quantizer properties
- Quantizing data with quantizers
  - Accessing data-related quantization information using a quantizer
- Displaying quantized data in binary or hexadecimal format
- Accessing quantizer data

The quantizers you create in this toolbox are objects with properties. Most of the basic information you need to know about setting and retrieving property values is found in Chapter 6, “Working with Objects.” See “Quantizer Properties Reference” on page 12-3 for information on quantizer properties.

Constructing Quantizers

You can construct quantizers by either:

- Using either quantizer constructor function:
  - `quantizer`
  - `unitquantizer`
- Copying a quantizer from an existing one using the `copyobj` function

---

Note You can also use the constructor `unitquantizer` to transform an existing quantizer into a unit quantizer.

All quantizer parameters are stored as properties that you can set or retrieve. Some of these quantizer parameters include:

- Quantization format
- Data type (signed or unsigned fixed-point, or double-, single-, or custom-precision floating-point)
- Rounding method used in quantization
- Overflow method used in quantization

Constructor for Quantizers

The easiest way to create a quantizer is to create one with the default properties. You can create a quantizer `q` by typing

```
q = quantizer
```

A listing of all of the properties of the quantizer `q` you just created is displayed along with the associated property values. All property values are set to defaults when you construct a quantizer this way. See "Example — Constructor for Quantized Filters" on page 6-3 for more details.

To construct a unit quantizer `q` with all of the default quantizer properties, type

```
q = unitquantizer
```

Quantizer Properties

Since a quantizer is an object, it has properties associated with it. You can set the values of some quantizer properties. However, some properties have read-only values. This sections covers both settable and read-only properties:

- "Settable Quantizer Properties" on page 7-4
- "Read-Only Quantizer Properties" on page 7-5

Properties and Property Values

Each property associated with a quantizer is assigned a value. When you construct a quantizer, you can assign some of the property values.

Most of the basic information you need to know about setting and retrieving property values is found in Chapter 6, "Working with Objects."

A complete list of properties of quantized filters is provided in Table 12-3, Quick Guide to Quantized Filter Properties, on page 12-10. Properties are described in more detail in "Quantized Filter Properties Reference" on page 12-11.

Settable Quantizer Properties

You can set the following four quantizer properties:

- `Mode` property — specifying the data type:
  - Fixed-point (signed or unsigned)
  - Custom floating-point
  - Double-precision floating-point
  - Single-precision floating-point
- `Format` property — specifying quantization format parameters
- `OverflowMode` property — specifying how overflows are handled in arithmetic operations
- `RoundMode` property — specifying the rounding method used in quantization

See "Quantizer Properties Reference" on page 12-3 for full details on all properties.

For example, create a fixed-point quantizer with:

- The `Format` property value set to `[16,14]`
- The `OverflowMode` property value set to `'saturate'`
- The `RoundMode` property value set to `'ceil'`

You can do this with the following command.

```
q = quantizer('mode','fixed','format',[16,14],'overflowmode',...
    'saturate','roundmode','ceil')
```

Setting Quantizer Properties Without Naming Them

You don't have to include quantizer property names when you set quantizer property values.

For example, you can create quantizer `q` from the previous example by typing

```
q = quantizer('fixed',[16,14],'saturate','ceil')
```

Note You do not have to include default property values when you construct a quantizer. In this example, you could leave out `'fixed'` and `'saturate'`.

Read-Only Quantizer Properties

Quantizers have five read-only properties:

- `Max`
- `Min`
- `NOperations`
- `NOverflows`
- `NUnderflows`

These properties log quantization information each time you use `quantize` to quantize data with a quantizer. The associated property values change each time you use `quantize` with a given quantizer. You can reset these values to the default value using `reset`.

For an example, see "Example — Data-Related Quantizer Information" on page 7-6.

Quantizing Data with Quantizers

You construct a quantizer to specify the quantization parameters to use when you quantize data sets. You can use the `quantize` function to quantize data according to a quantizer's specifications.

Once you quantize data with a quantizer, its data-related, read-only property values may change.

The following example shows:

- How you use `quantize` to quantize data
- How quantization affects the read-only properties
- How you reset the read-only properties to their default values using `reset`

Example — Data-Related Quantizer Information

1 Construct an example data set and a quantizer.

```
randn('state',0);
x = randn(100,4);
q = quantizer([16,14]);
```

2 Retrieve the values of the `Max` and `Noverflows` properties.

```
q.max

ans =
reset

q.noverflows

ans =
     0
```

3 Quantize the data set according to the quantizer's specifications.

```
y = quantize(q,x);
```

4 Check the quantizer property values.

```
q.max
```

```
ans =
2.3726

q.noverflows

ans =
     15
```

5 Reset the read-only properties and check them.

```
reset(q)
q.max

ans =
reset

q.noverflows

ans =
     0
```

Transformations for Quantized Data

You can convert data values from numeric to hexadecimal or binary according to a quantizer's specifications.

Use:

- `num2bin` to convert data to binary
- `num2hex` to convert data to hexadecimal
- `hex2num` to convert hexadecimal data to numeric
- `bin2num` to convert binary data to numeric

For example,

```
q = quantizer([3 2]);
      x = [0.75   -0.25
           0.50   -0.50
           0.25   -0.75
           0      -1   ];
      b = num2bin(q,x)

b =
011
010
001
000
111
110
101
100
```

produces all two's complement fractional representations of three-bit fixed-point numbers.

Quantizer Data Functions

Filter Design Toolbox provides a number of data functions to retrieve information about a quantizer. These functions include:

- `denormalmax` — the largest denormalized quantized number
- `denormalmin` — the smallest denormalized quantized number
- `eps` — the quantization level
- `exponentbias` — the exponent bias of a quantizer
- `exponentlength` — the exponent length of a floating-point quantizer
- `exponentmax` — the maximum exponent allowable for a floating-point quantizer
- `fractionlength` — the fraction length of a fixed-point quantizer
- `range` — the numerical range of a quantizer
- `realmax` — the largest positive number a quantizer can produce
- `realmin` — the smallest positive normal number a quantizer can produce
- `wordlength` — the word length of a quantizer

For example, to find the largest positive quantized number the default quantizer can create, type

```
format long
q = quantizer;
r = realmax(q)

r =
   0.99996948242188
```

8

Working with Quantized Filters

Constructing Quantized Filters (p. 8-3)	Describes how you construct quantized filters in the toolbox
Quantized Filter Properties (p. 8-6)	Lists and explains the properties associated with quantized filter objects, called quantized filters
Filtering Data with Quantized Filters (p. 8-14)	Uses examples to show you how to use quantized filters to filter data
Transformation Functions for Quantized Filter Coefficients (p. 8-15)	Introduces the hex and binary functions for changing the way you display quantized filters

This chapter covers what you need to know to construct and use quantized filters:

- Constructing quantized filters
- Quantized filter properties
- Filtering data with quantized filters
- Transformation Functions for Quantized Filter Coefficients

The quantized filters you create in this toolbox are objects with properties. Most of the basic information you need to know about setting and retrieving property values is found in Chapter 6, “Working with Objects.”

Constructing Quantized Filters

You can construct quantized filters in the Filter Design Toolbox by either:

- Using the quantized filter constructor function `qfilt`
- Copying an existing one

All filter characteristics are stored as properties that you can set or retrieve. Some of these quantized filter characteristics include:

- Filter structure.
- Reference filter coefficients.
- Filter topology (single section or cascaded nth-order sections). The syntax you use to enter the reference filter coefficients determines the topology.
- Quantized filter data format parameters:
  - Quantization parameters (precisions).
  - Data type (signed or unsigned fixed-point, or, double-, single-, or custom-precision floating-point).
  - Rounding method used in quantization.
  - Overflow method used in quantization.
- Scaling factors for each section of a cascade of nth-order sections.

You can specify quantized filter properties by either:

- Specifying all of the filter properties when you create it
- Creating a quantized filter with default property values, and changing some or all of these property values later

Constructor for Quantized Filters

The most direct way to create a quantized filter (Qfilt object) is to create one with the default properties. You create a default quantized filter `Hq` by typing

```
Hq = qfilt
```

A listing of all of the properties of the filter `Hq` you just created is displayed along with the associated property values. All property values are set to defaults when you construct a quantized filter this way.

To construct a quantized filter with properties other than the default values, follow the procedure outlined in “Setting Property Values Directly at Construction” on page 6-5.

For some examples of using the quantized filter constructor to construct a filter while specifying some properties at construction, see:

- “Constructing an Eight-Bit Quantized Filter” on page 5-8
- “Example — Accessing Coefficient Information from Filters with One Section” on page 6-14
- “Example — Accessing Coefficient Information from Multisection Filters” on page 6-15

Constructing a Quantized Filter from a Reference

In general you construct quantized filters from reference filters. You begin with a set of unquantized (or quantized) reference filter coefficients to implement in a quantized filter.

Suppose you design a quantized filter from a fourth-order elliptic filter. You can use the Signal Processing Toolbox filter design functions to help you. First, design a filter with parameters in transfer function form.

```
[b,a] = ellip(4,3,20,.6);
```

Filters designed with a second-order section topology are more robust against quantization errors than those composed of higher order transfer functions.

Converting a Filter to Second-Order Sections Form

You can construct a quantized filter in second-order sections form as follows:

1. Create a quantized filter using the elliptic filter’s transfer function parameters as reference coefficients.

   ```
   Hq = qfilt('df2t',{b,a});
   ```

   This filter is not in second-order sections form and has coefficient overflow.

2. Use sos to convert the filter to second-order sections form.

   ```
   Hq = sos(Hq);
   ```

Copying Filters to Inherit Properties

If you already have a quantized filter Hq with the property values set the way you want them, you can create a new quantized filter Hq2 with the same property values as Hq by typing

```
Hq2 = copyobj(Hq)
```

This function is convenient to use when you are changing a small number of properties on a set of filters.

For example, create a 16-bit precision filter Hq from an FIR reference filter with

```
b = fir1(80,0.5,kaiser(81,8)); % Reference filter
Hq = qfilt('fir',{b})
```

Except for the filter coefficients provided by {b}, Hq inherits the default property values for a quantized filter.

Changing Filter Property Values After Construction

Now suppose you want to analyze the response of this same reference filter b when you:

- Change all of the data format property values using setbits
- Change the ScaleValues property value to [0.5 0.5]

You can do this by first copying Hq, and then changing only those properties you want to change.

```
Hq2 = copyobj(Hq);
setbits(Hq2,[16,14])
Hq2.ScaleValues = [0.5 0.5];
Hq2.scale

ans =
    0.5000    0.5000
```

For more information on setting filter properties, see “Setting Property Values with the set Command” on page 6-6 and “Direct Property Referencing to Set and Get Values” on page 6-9.

Quantized Filter Properties

Since a quantized filter is a Qfilt object, it has properties associated with it. These properties prescribe the most basic filter qualities, such as the data format for each data path or the rounding methods used for quantization and filtering. You can set the values of most properties. However, some properties have read-only values.

Properties and Property Values

Each property associated with a quantized filter is assigned a value. When you construct a quantized filter, you assign some of the quantized filter property values to design a quantized filter to your own specifications. You can set or retrieve quantized filter properties according to the information in "Setting and Retrieving Property Values" on page 6-5.

A complete list of properties of quantized filters is provided in Table 12-3, Quick Guide to Quantized Filter Properties, on page 12-10. Properties are described in more detail in "Quantized Filter Properties Reference" on page 12-11.

Basic Filter Properties

Basic filter properties include:

- The `ReferenceCoefficients` property — specifying the filter reference coefficients
- The `FilterStructure` property — specifying the quantized filter structure
- The data format properties for setting quantization parameters for data and arithmetic operations:
  - `CoefficientFormat` — specifying how the reference filter coefficients are quantized
  - `InputFormat` — specifying how the inputs are quantized
  - `MultiplicandFormat` — specifying how data is quantized before it is multiplied by a coefficient
  - `OutputFormat` — specifying how the outputs are quantized
  - `ProductFormat` — specifying how the results of multiplication are quantized

- `SumFormat` — specifying how the results of addition are quantized

See “Quantized Filter Properties Reference” on page 12-11 for full details on all properties.

Specifying the Filter Reference Coefficients

The `ReferenceCoefficients` property value contains the filter parameters for the reference filter that specifies your quantized filter. “Constructing a Quantized Filter from a Reference” on page 8-4 uses the `ReferenceCoefficients` property in an example of quantized filter construction.

The syntax you use for assigning reference filter coefficients depends on the filter structure and topology to assign. See “Assigning Reference Filter Coefficients” on page 12-40 for more information on the syntax for each filter structure and topology.

For example, to assign a direct form II transposed filter structure with one second-order section for the transfer function

$$H(z) = \frac{1 + 0.5z^{-1}}{1 + 0.7z^{-1} + 0.89z^{-2}}$$

type

```
Hq = qfilt('FilterStructure','df2t','ReferenceCoefficients',...
    {[1 .5] [1 .7 .89]});
```

In this example, you use the constructor `qfilt` to specify the quantized filter. You set the `FilterStructure` and the `ReferenceCoefficients` property values at the same time that you specify the filter. All other filter properties retain their default values.

Notice that you enter the numerator and denominator polynomial coefficients in one cell array for this filter with one second-order section. In general you use a separate cell array to specify the reference filter coefficients for each cascaded section in a quantized filter.

Specifying the Quantized Filter Structure

In Filter Design Toolbox, you can create quantized filters with 16 different filter structures:

- Direct form I
- Direct form I transposed
- Direct form II
- Direct form II transposed
- Direct form Finite Impulse Response (FIR)
- Direct form FIR transposed
- Direct form antisymmetric FIR (odd and even orders)
- Direct form symmetric FIR filters (odd and even orders)
- Lattice allpass
- Lattice coupled-allpass filters
- Lattice coupled allpass power-complementary filters
- Lattice Moving Average (MA) minimum phase filters
- Lattice MA maximum phase filters
- Lattice Autoregressive (AR) filters
- Lattice ARMA filters
- Single-input single-output state-space filters

Filter structures are described in detail in the description for the property `FilterStructure` on page 12-12.

You can create filters with two possible filter topologies:

- A single section
- Cascaded nth-order sections

Topology. You set the topology when you specify the reference filter coefficients for your quantized filter. See "Assigning Reference Filter Coefficients" on page 12-40 for more information. After you create your quantized filter with the topology you choose, use Filter Design and Analysis Tool (FDATool) in quantization mode to change the filter topology. For more information about FDATool, refer to Chapter 11, "Using FDATool with the Filter Design Toolbox."

For example, you can construct a quantized filter with a lattice AR structure by:

1 Specifying a vector of AR reflection coefficients

```
k = [.66 .7 .44];
```

2 Constructing a quantized filter with a lattice AR filter structure

```
Hq = qfilt('latticear',{k});
```

Notice that:

- You don't have to type the `'FilterStructure'` property name at construction
- You specify the reflection reference filter coefficients in a cell array

Specifying the Data Formats

Quantized filters have six data format properties you can set:

- `CoefficientFormat`
- `InputFormat`
- `MultiplicandFormat`
- `OutputFormat`
- `ProductFormat`
- `SumFormat`

Specify the data format property values for quantized filters using quantizers. For each data format, you can specify:

- Data type
- Quantization format parameters
- Method for handling quantization overflows
- Method for rounding

For example, the quantization format of the `CoefficientFormat` property for `Hq` has the default value of `[16,15]` (as do all data format properties for this filter). To change the quantization format for the `CoefficientFormat` property value to `[16,14]`, type

```
Hq.CoefficientFormat.Format = [16,14];
Hq.CoefficientFormat.Format
ans =
    16    14
```

Here you are changing the `Format` property of the quantizer for the `CoefficientFormat` property. This syntax leaves all other property values for the quantizer for the `CoefficientFormat` property unchanged.

Specifying All Data Format Properties at Once

To implement the quantized lattice filter `Hq` you just specified using floating-point calculations, you need to set the `Mode` property value for each data format property quantizer for `Hq` to `'float'`. You can do this using the *`quantizer`* syntax for accessing the data format properties. See `qfilt` for more information on this syntax.

```
Hq.quantizer = {'float', [24,8]}

Hq =

Quantized Autoregressive Lattice (latticear) filter.
Lattice
 QuantizedCoefficients{1}    ReferenceCoefficients{1}
    (1) 0.659988403320313    0.660000000000000030
    (2) 0.699996948242188    0.699999999999999960
    (3) 0.439994812011719    0.440000000000000000

   FilterStructure = latticear
       ScaleValues = [1]
  NumberOfSections = 1
  StatesPerSection = [3]
 CoefficientFormat = quantizer('float', 'round', [24   8])
       InputFormat = quantizer('float', 'floor', [24   8])
      OutputFormat = quantizer('float', 'floor', [24   8])
MultiplicandFormat = quantizer('float', 'floor', [24   8])
     ProductFormat = quantizer('float', 'floor', [24   8])
         SumFormat = quantizer('float', 'floor', [24   8])
```

Note The `quantizer` syntax lets you use one line of code to change the `Mode` and `Format` property values for all data format quantizers. You can also do this with the following six commands.

```
Hq.CoefficientFormat = quantizer('float',[24,8])
Hq.InputFormat = quantizer('float',[24,8])
Hq.MultiplicandFormat = quantizer('float',[24,8])
Hq.OutputFormat = quantizer('float',[24,8])
Hq.ProductFormat = quantizer('float',[24,8])
Hq.SumFormat = quantizer('float',[24,8])
```

Specifying the Format Parameters with setbits

Suppose you want to change all of the arithmetic and quantization data format parameters for the custom floating-point filter `Hq` in the previous example to `[24 8]`. You can do this in three ways:

- Using the `setbits` command
- Using the `quantizer` syntax
- Setting each data format property separately

To do this using the `setbits` command, type

```
setbits(Hq,[24,8])
```

To do this using the `quantizer` syntax, type

```
Hq.quantizer = [24,8];
```

These two commands are equivalent for floating-point filters.

Note The `setbits` command behaves slightly differently for fixed-point filters. It doubles the quantization data formats for products and sums.

Using normalize to Scale Coefficients

Even though you can specify how overflows are treated, they are not corrected for automatically. You can use `normalize` to account for coefficient quantization overflows for all of the direct form and FIR fixed-point filter structures. This function normalizes the coefficients and modifies the filter scaling.

For example, if you create an elliptic filter with Signal Processing Toolbox and directly quantize it with fixed-point arithmetic, there may be some coefficient overflows.

```
[b,a] = ellip(5,2,40,0.4);
Hq = qfilt('df2t',{b,a})

Warning: 5 overflows in coefficients.
```

A warning is displayed indicating that there are coefficient overflows in this fixed-point filter. This type of warning is displayed whenever you create a filter with coefficient overflow and you have MATLAB `warning` set `on`.

You can normalize the coefficients and modify the scaling using `normalize`.

```
Hq = normalize(Hq)

Hq.ScaleValues
ans =
    0.0313    1.0000
```

Notice that:

- The `ScaleValues` property value has been modified from its original (default) value of 1.
- There is no longer any coefficient overflow in `Hq`.

You can apply `normalize` to direct form IIR and FIR filters. The `FilterStructure` property value must be one of the following:

- `'antisymmetricfir'`
- `'df1'`
- `'df1t'`
- `'df2'`

- `'df2t'`
- `'fir'`
- `'firt'`
- `'symmetricfir'`

Filtering Data with Quantized Filters

You can filter data with quantized filters using the `filter` function.

Example: Filtering Data with a Quantized Filter

```
warning on
randn('state',0);
x = randn(100,2);
[b,a] = butter(3,.9,'high');
Hq = sos(qfilt('ReferenceCoefficients',{0.5* b,0.5*a},...
'CoefficientFormat',unitquantizer([26 24])));
y = filter(Hq,x);

Warning: 64 overflows in QFILT/FILTER.

                    Max          Min     NOverflows   NUnderflows   NOperations
 Coefficient       1.187           -1            0             0            12
                   1.648           -2            0             0            18
       Input       2.183       -2.202           64             0           200
      Output      0.4345      -0.4477            0             0           200
Multiplicand           1           -1            0             2           800
                  0.4345      -0.4477            0             0          1000
     Product    0.009246    -0.008869            0             0           800
                  0.7158      -0.7377            0             0          1000
         Sum     0.01274      -0.0122            0             0           600
                  0.4345      -0.4477            0             0          1000
```

Notice that a record of the overflows that occurred in filtering is displayed if you have set `warning on`.

Use `qreport` to get this listing when needed as well.

Transformation Functions for Quantized Filter Coefficients

You can change the display for quantized filter coefficients to:

- Binary, using `num2bin`
- Hexidecimal, using `num2hex`

For example, to display the coefficients of the filter `Hq` you just created as hexidemimal numbers, type

```
num2hex(Hq)
Hq.QuantizedCoefficients{1} =

05D8
0655
0B99
0B99
0655
05D8

Hq.QuantizedCoefficients{2} =

7FFF
8000
7FFF
8000
7FFF
CAE8
```

9

Working with Quantized FFTs

Constructing Quantized FFTs (p. 9-3)	Talks about how you construct quantized FFTs
Quantized FFT Properties (p. 9-6)	Explains the properties of quantized FFT objects
Computing a Quantized FFT or Inverse FFT of Data (p. 9-10)	Shows you how to compute both the FFT and inverse FFT of a data set in MATLAB

Use quantized fast Fourier transforms (FFTs) to specify quantization parameters for computing a quantized FFT or inverse FFT.

This chapter covers what you need to know to construct and use quantized FFTs:

- Constructing quantized FFTs
- Quantized FFT properties
- Computing quantized FFTs and quantized inverse FFTs

The quantized FFTs you create in this toolbox are called *QFFT objects*. These objects have properties. Most of the basic information you need to know about setting and retrieving property values is found in Chapter 6, “Working with Objects.”

Constructing Quantized FFTs

You can construct quantized FFTs in the Filter Design Toolbox by either:

- Using the quantized FFT constructor function `qfft`
- Copying a quantized FFT from an existing one

All quantized FFT characteristics are stored as properties that you can set or retrieve. Some of these quantized FFT characteristics include:

- The FFT length.
- The radix number. Either 2 or 4.
- The number of sections in the FFT. Computed from the length and radix of the FFT.
- Quantized FFT data format parameters:
  - Quantization parameters (precisions).
  - Data type (signed or unsigned fixed-point; or double-, single-, or custom-precision floating-point).
  - Rounding method used in quantization.
  - Overflow method used in quantization.
- Scaling factors for each stage of the FFT.

You can specify quantized FFT properties by either:

- Specifying them when you create a quantized FFT
- Creating a quantized FFT with default property values, and changing some or all of these property values later

Constructor for Quantized FFTs

The easiest way to create a quantized FFT (QFFT object) is to create one with the default properties. You create a default quantized FFT `F` by typing

```
F = qfft
```

A listing of the properties of the FFT `F` you just created is displayed along with the associated property values. All property values are set to defaults when you create a quantized FFT this way.

To construct a quantized FFT with properties other than the default values, follow the procedure outlined in "Setting Property Values Directly at Construction" on page 6-5.

Copying Quantized FFTs to Inherit Properties

If you have a quantized FFT `F` with the property values set the way you want them, you can create a new quantized FFT `F2` with the same property values as `F` by typing

```
F2 = copyobj(F)
```

For example, create a length 32, radix 2, FFT `F` by typing

```
F = qfft('length',32, 'radix', 2)

F =
            Radix = 2
            Length = 32
 CoefficientFormat = quantizer('fixed', 'round', 'saturate', [16
15])
      InputFormat = quantizer('fixed', 'floor', 'saturate', [16
15])
     OutputFormat = quantizer('fixed', 'floor', 'saturate', [16
15])
MultiplicandFormat = quantizer('fixed', 'floor', 'saturate', [16
15])
    ProductFormat = quantizer('fixed', 'floor', 'saturate', [32
30])
        SumFormat = quantizer('fixed', 'floor', 'saturate', [32
30])
  NumberOfSections = 5
       ScaleValues = [1]
```

Except for the length and the number of sections, `F` inherits all of the default property values for a quantized filter.

Changing Some FFT Property Values After Construction

You can create another quantized FFT `F2`, which has the same properties as `F`, but scales each stage of the FFT differently. To do this:

1 Copy F.

2 Change the ScaleValues property value.

For example, you can do this by typing

```
F2 = copyobj(F);
F2.ScaleValues = [1 0.5 0.25 0.5 1];
```

For more information on setting FFT properties, see “Setting Property Values with the set Command” on page 6-6 and “Direct Property Referencing to Set and Get Values” on page 6-9.

Quantized FFT Properties

Since a quantized FFT is a QFFT object, it has properties associated with it. These properties prescribe the FFT characteristics, such as the FFT length and the radix number. You can set the values of most properties. However, some properties have read-only values.

Properties and Property Values

Each property associated with a quantized FFT is assigned a value. When you construct a quantized FFT, you can assign some of the quantized FFT property values. You can set or retrieve quantized FFT properties according to the information in “Setting and Retrieving Property Values” on page 6-5.

A complete list of properties of quantized FFTs is provided in Table 12-6, Quick Guide to Quantized FFT Properties, on page 12-51. Properties are described in more detail in “Quantized FFT Properties Reference” on page 12-52.

Basic Quantized FFT Properties

Basic quantized FFT properties include:

- The `Radix` property — specifying the FFT’s radix number (2 or 4)
- The `Length` property — specifying the quantized FFT length (a power of the radix number)
- The data format properties for setting quantization parameters for data and arithmetic operations:
  - `CoefficientFormat` — specifying how the FFT coefficients (twiddle factors) are quantized
  - `InputFormat` — specifying how the inputs are quantized
  - `MultiplicandFormat` — specifying how data is quantized before it is multiplied by a coefficient
  - `OutputFormat` — specifying how the outputs are quantized
  - `ProductFormat` — specifying how the results of multiplication are quantized
  - `SumFormat` — specifying how the results of addition are quantized

See “Quantized FFT Properties Reference” on page 12-52 for full details on all properties.

Specifying the Data Formats

Quantized FFTs have six data format properties you can set:

- `CoefficientFormat`
- `InputFormat`
- `MultiplicandFormat`
- `OutputFormat`
- `ProductFormat`
- `SumFormat`

Specify the data format property values for quantized FFTs using quantizers. For each data format, you can specify:

- Data type
- Quantization format parameters
- Method for handling quantization overflows
- Method for rounding

For example:

1. Create a default quantized FFT `F`.

2. Change the quantization format parameters for the `CoefficientFormat` property value to `[16,14]`.

```
% Create a default quantized FFT.
F = qfft;
% Display the format of the coefficient quantization.
F.CoefficientFormat.Format

ans =
    16    15
% Change the coefficient quantization to [16,14].
F.CoefficientFormat.Format = [16,14];
F.CoefficientFormat.Format

ans =
    16    14
```

Here you are changing the `Format` property of the quantizer for the quantized FFT's `CoefficientFormat` property. This syntax leaves all other property values for the quantizer for the `CoefficientFormat` property unchanged.

Specifying All Data Format Properties at Once

To implement the quantized FFT `F` you just specified using floating-point calculations, set the `Mode` property value for each data format property quantizer for `F` to `'float'`. You do this using the `quantizer` syntax for accessing the data format properties. See `qfft` for more information on this syntax.

```
F.quantizer = {'float', [24,8]}

F =
             Radix = 2
            Length = 16
 CoefficientFormat = quantizer('float', 'floor', [24 8])
       InputFormat = quantizer('float', 'floor', [24 8])
      OutputFormat = quantizer('float', 'floor', [24 8])
MultiplicandFormat = quantizer('float', 'floor', [24 8])
     ProductFormat = quantizer('float', 'floor', [24 8])
         SumFormat = quantizer('float', 'floor', [24 8])
    NumberOfSections = 4
       ScaleValues = [1]
```

Specifying the Format Parameters with setbits

Suppose you want to change all of the arithmetic and quantization data format parameters for the custom floating-point FFT `F` in the previous example to `[24 4]`. You can do this in three ways:

- Using the `setbits` command
- Using the `quantizer` syntax
- Setting each data format property separately

To do this using the `setbits` command, type

```
setbits(F,[24,4])
```

To do this using the `quantizer` syntax, type

```
F.quantizer = [24,4];
```

These two commands are equivalent for floating-point FFTs.

Note The `setbits` command behaves slightly differently for fixed-point FFTs in that it doubles the quantization data formats for products and sums.

Computing a Quantized FFT or Inverse FFT of Data

To compute a quantized FFT or inverse FFT of a data set:

1 Create a quantized FFT `F`.

2 Obtain or create the data set.

3 Apply `fft` to `F` for a quantized FFT or `ifft` to `F` for a quantized inverse FFT.

For example, type

```
warning on
randn('state',0)
F = qfft;         % Create a quantized FFT.
x = randn(100,3); % Create a sample data set x.
y = fft(F,x);     % Compute a quantized FFT of x.
```

```
Warning: 542 overflows in quantized fft.
```

	Max	Min	NOverflows	NUnderflows	NOperations
Coefficient	1	-1	5	4	62
Input	2.309	-2.365	97	0	300
Output	2	-2	71	0	192
Multiplicand	2	-2	350	0	3840
Product	1	-1	0	0	960
Sum	2.414	-2.414	24	0	2400

Notice that a record of the overflows that occurred in filtering is displayed if you have warnings turned on.

You can also use `qreport` to get this report.

10

Quantized Filtering Analysis Examples

Example — Quantized Filtering of Noisy Speech (p. 10-3)	To help explain quantized filtering, this example demonstrates one way to remove noise from a signal
Example — A Quantized Filter Bank (p. 10-17)	In this example, you see how to create a filter bank to filter data
Example — Effects of Quantized Arithmetic (p. 10-23)	Using quantized filters on data may change the data; this example describes some of those changes and how to account for them

This chapter includes the following examples of how you use the quantized filtering features of this toolbox:

- “Example — Quantized Filtering of Noisy Speech”
- “Example — A Quantized Filter Bank”
- “Example — Effects of Quantized Arithmetic”

Example — Quantized Filtering of Noisy Speech

This example covers the following procedure that demonstrates filtering of a noisy signal:

1 "Loading a Speech Signal" on page 10-3
2 "Analyzing the Frequency Content of the Speech" on page 10-4
3 "Adding Noise to the Speech" on page 10-4
4 "Creating a Filter to Extract the 3000Hz Noise" on page 10-5
5 "Quantizing the Filter as a Fixed-Point Filter" on page 10-8
6 "Normalizing the Quantized Filter Coefficients" on page 10-8
7 "Analyzing the Filter Poles and Zeros Using zplane" on page 10-9
8 "Creating a Filter with Second-Order Sections" on page 10-12
9 "Quantized Filter Frequency Response Analysis" on page 10-13
10 "Filtering with Quantized Filters" on page 10-14
11 "Analyzing the filter Function Logged Results" on page 10-15

Loading a Speech Signal

To load a speech signal contained in a matrix `mtlb`, along with its associated sampling frequency `Fs`, type

```
load mtlb
```

If you have speakers and a sound card, you can type

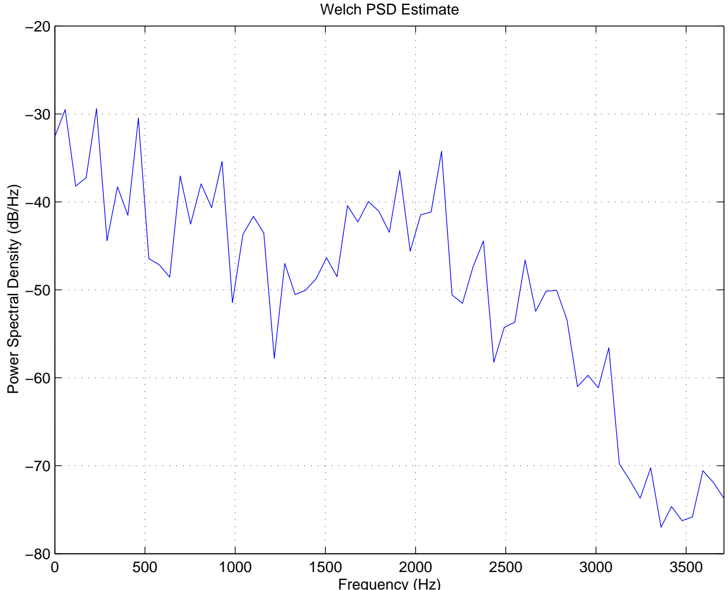
```
sound(mtlb)
```

and hear this speech signal.

Analyzing the Frequency Content of the Speech

Next look at the power spectral density of this signal using the `pwelch` command.

```
n = length(mtlb);
nfft = 128;
pwelch(mtlb,[],[],nfft,Fs)
```



Adding Noise to the Speech

Now add noise to the speech signal at 3000 hertz (Hz) and 3100 Hz and look at its power spectral density.

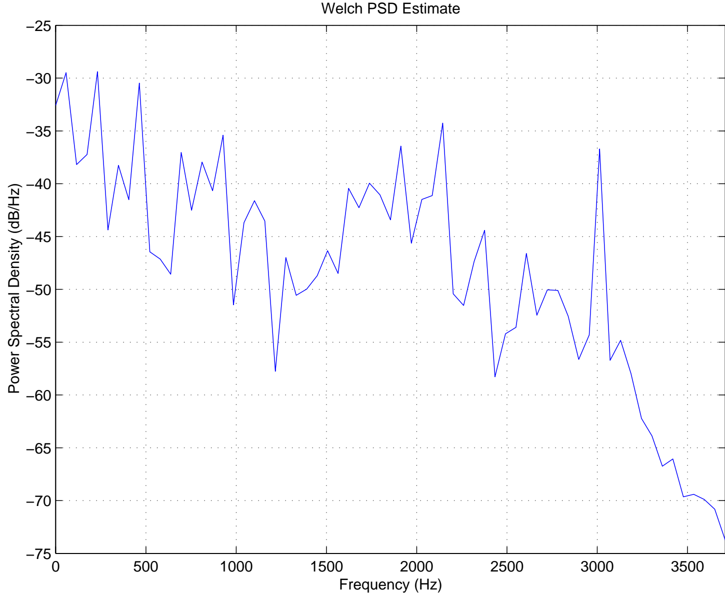
```
f1 = 3000;            %  Noise frequency in Hz.
f2 = 3100;            %  Noise frequency in Hz.
t = (0:n-1)'/Fs;     %  Time duration of the noise signal.

noise = sin(2*pi*f1*t) + 0.8*sin(2*pi*f2*t);
u = mtlb + noise;     %  Add noise to the mtlb signal.
```

If you have speakers and a sound card, type

```
sound(u)
```

Otherwise, use `pwelch` to look at the power spectral density for `u` and compare it to that of `mtlb`.

```
pwelch(u,[],[],nfft,Fs);
```



Notice the difference between the two power spectral densities in the 3000 to 3100 Hz range.

Creating a Filter to Extract the 3000Hz Noise

Consider this simple notched filter design to remove the 3000Hz noise.

A Notched Filter Design

To design a notched filter in MATLAB to remove noise at a given frequency, for each frequency you want to remove:

1 Calculate the (normalized) frequency you want to remove in rad/sample.

2. Place a complex zero on the unit circle at this normalized frequency.

3. Place a stable complex pole close to this zero, but inside the unit circle.

4. Determine the filter numerator and denominator by:

   a. Specifying factors of the numerator and denominator polynomials using the pole and zero

   b. Using `conv` to multiply the factors by their conjugates

For this example, you want to remove noise at both 3000 Hz and 3100 Hz, so you can follow these steps for both `f1=3000` and `f2=3100`, and put the two notched filters together.

Here are the steps for `f1=3000`. Repeat these for `f2=3100` for the final design.

The frequency you want to remove is calculated in rad/sample as

```
wo = 2*pi*f1/Fs;
```

A notched filter has a zero on the unit circle at a frequency corresponding to an angle of `wo` radians. This removes any noise at this frequency. You can find the real and imaginary parts (`x` and `y`) of the corresponding zero using

```
rez = cos(wo);
imz = sin(wo);
```

The next step in the notched filter design is to add a pole close to the zero, but inside the unit circle. This essentially eliminates the effect of the notched filter at frequencies other than 3000 Hz, while keeping the filter stable. The closer the pole is to the zero, the narrower the notch will be.

```
rez1 = .99*cos(wo);
imz1 = .99*sin(wo);
```

You can define this portion of the filter's numerator and denominator polynomials `b` and `a` by introducing the complex conjugate factors and using `conv`.

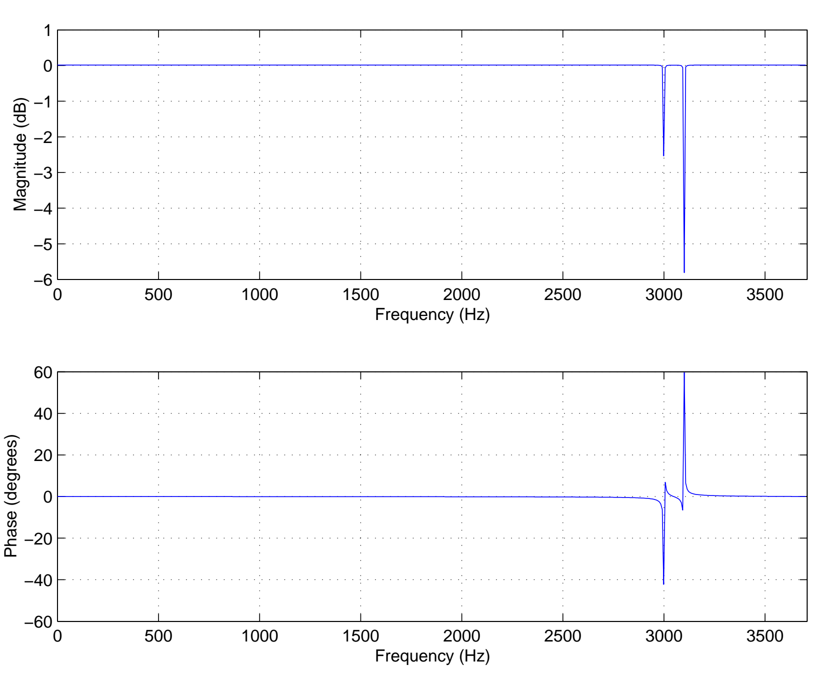
```
b1 = conv([1 -rez-i*imz],[1 -rez+i*imz]);
a1 = conv([1 -rez1-i*imz1],[1 -rez1+i*imz1]);
```

Similarly, you can follow these steps to remove 3100 Hz noise.

```
b2 = conv([1 -cos(2*pi*f2/Fs)-i*sin(2*pi*f2/Fs)],...
        [1 -cos(2*pi*f2/Fs)+i*sin(2*pi*f2/Fs)]);
a2 = conv([1 -0.99*cos(2*pi*f2/Fs)-i*0.99*sin(2*pi*f2/Fs)],...
        [1 -0.99*cos(2*pi*f2/Fs)+i*0.99*sin(2*pi*f2/Fs)]);
```

Finally, put these two filters together and look at the frequency response.

```
b = conv(b1,b2);
a = conv(a1,a2);
freqz(b,a,512,Fs);
```



Quantizing the Filter as a Fixed-Point Filter

You can create a direct form II transposed fixed-point quantized filter using the elliptic filter you just created as a reference. Name the filter `Hq1`.

```
Hq1 = qfilt('df2t',{b,a});

Warning: 9 Overflows in coefficients.
```

Normalizing the Quantized Filter Coefficients

MATLAB displays a warning because the filter you just created has some coefficient overflow associated with it. You can use the `normalize` command to scale the coefficients and account for this overflow.

```
Hq1 = normalize(Hq1);
```

In addition to scaling the filter coefficients, the `normalize` also modifies the `ScaleValues` property value to account for the coefficient scaling when you filter.

```
Hq1

Hq1 =
Quantized Direct form II transposed filter
Numerator
    QuantizedCoefficients{1}   ReferenceCoefficients{1}
  (1)      0.125000000000000  0.125000000000000000
  (2)      0.423736572265625  0.423728525076514370
  (3)      0.608825683593750  0.608840299517598550
  (4)      0.423736572265625  0.423728525076514370
  (5)      0.125000000000000  0.125000000000000000
Denominator
    QuantizedCoefficients{2}   ReferenceCoefficients{2}
  (1)      0.125000000000000  0.125000000000000000
  (2)      0.419494628906250  0.419491239825749210
  (3)      0.596710205078125  0.596724377557198320
  (4)      0.411132812500000  0.411143364153216890
  (5)      0.120086669921875  0.120074501250000020

   FilterStructure = df2t
       ScaleValues = [1  1]
```

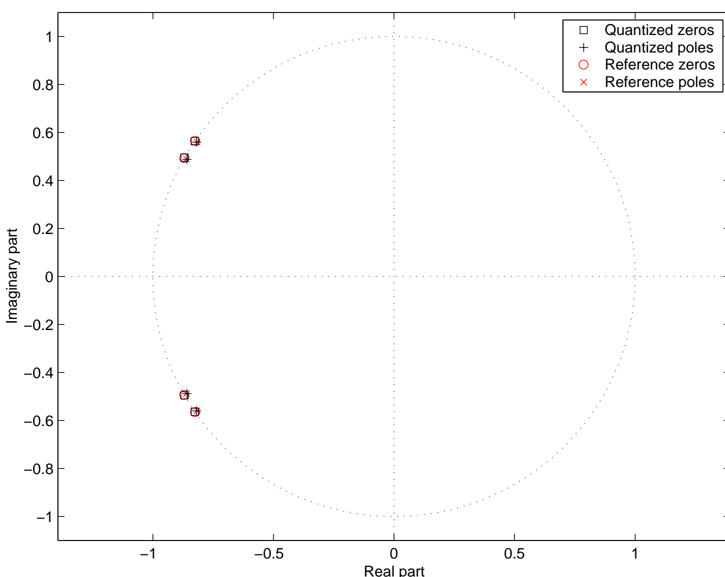
```
  NumberOfSections = 1
  StatesPerSection = [4]
 CoefficientFormat = quantizer('fixed', 'round', 'saturate', [16
15])
      InputFormat = quantizer('fixed', 'floor', 'saturate', [16
15])
     OutputFormat = quantizer('fixed', 'floor', 'saturate', [16
15])
MultiplicandFormat = quantizer('fixed', 'floor', 'saturate', [16
15])
    ProductFormat = quantizer('fixed', 'floor', 'saturate', [32
30])
        SumFormat = quantizer('fixed', 'floor', 'saturate', [32  30])
```

Note In this example, the `ScaleValues` property value is `[1 1]`. There is effectively no scaling associated with the sections of this particular filter, even after it has been normalized. This is because the required scaling for the numerator and denominator of each filter section is the same.

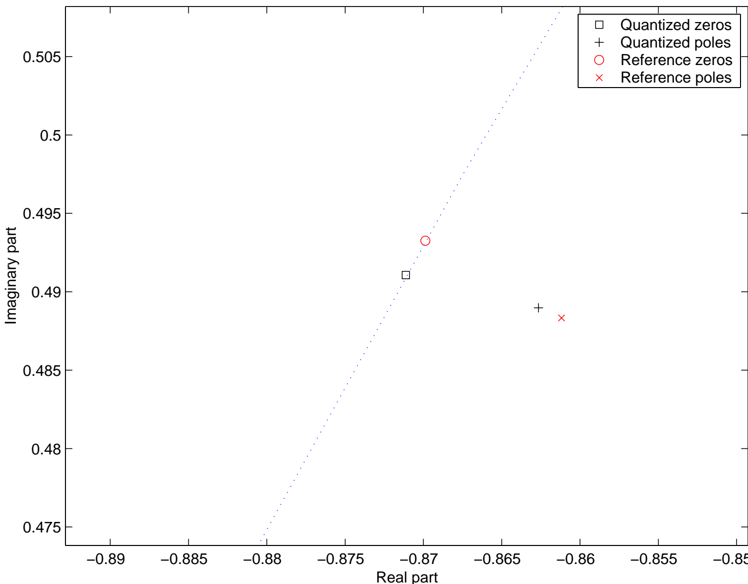
Analyzing the Filter Poles and Zeros Using zplane

You can apply `zplane` to a quantized filter to analyze its poles and zeros.

```
zplane(Hq1)
```



At first glance, this looks like you've done a good job at the fixed point notched filter design. If you zoom in, you can see that the quantized poles are not really at the correct angles for the notched filter. This is caused by quantization error.



Having poles located at incorrect angles is not the only problem in the filter. There are overflow limit cycles that you detect by

```
rand('state',0)
limitcycle(Hq1)
```

resulting in the warning

```
Overflow limit cycle detected.
```

To see the destructive behavior of the limit cycles, look at the plot from the noise loading method `nlm`.

```
nlm(Hq1)
```

The quantized noise loading method is random noise around the filter notches. Also, `zplane(Hq1)` shows oscillating behavior for the filter.

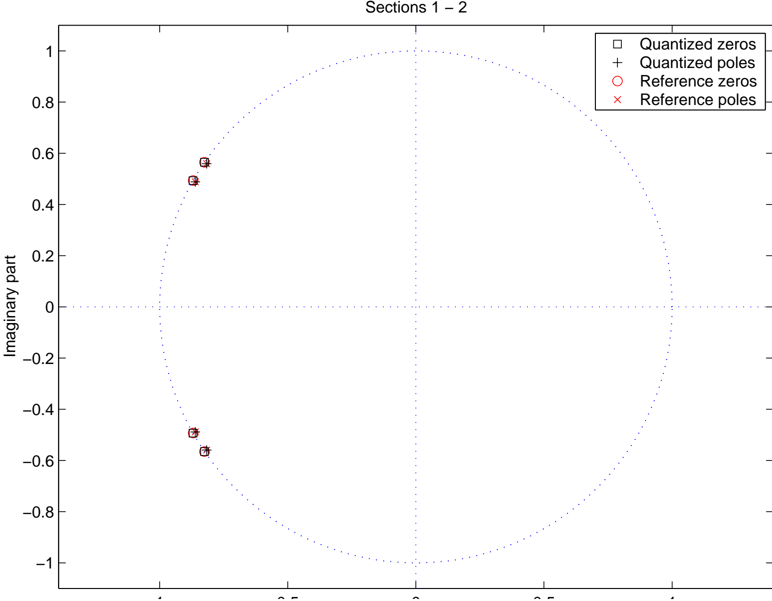
Creating a Filter with Second-Order Sections

Filters whose transfer functions have been factored into second-order sections are less susceptible to coefficient quantization errors. If you are using a quantized filter with a transfer function filter structure, you can use `sos` to convert the normalized quantized filter to second-order sections form.

```
Hq2 = sos(Hq1);
```

Now look at the poles and zeros using `zplane`.

```
zplane(Hq2)
```



Zoom in, as shown in the next figure, to see that the quantized notched filter design poles and zeros are lined up the way you designed them.

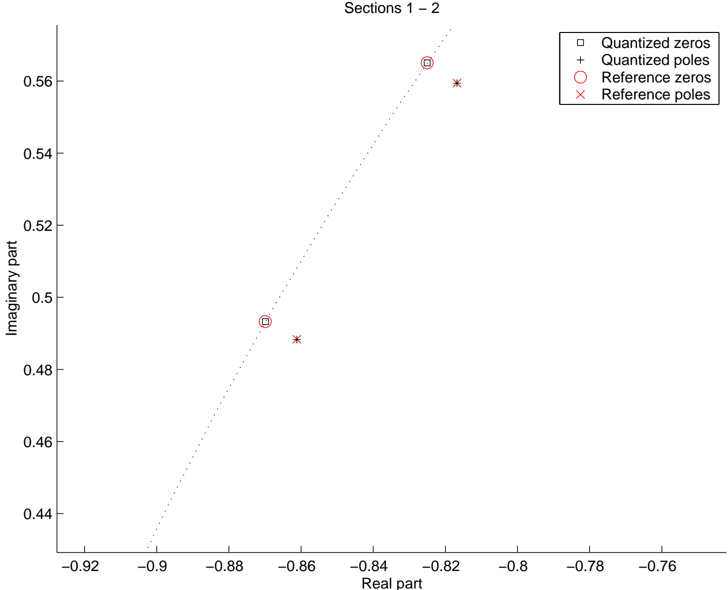
Also, the overflow limit cycle problem has cleared up. You can verify this with

```
limitcycle(Hq2)
```

and

```
nlm(Hq2)
```

By zooming in on the tail of the impulse response, plotted by `impz(Hq2)`, you see the granular limit cycle, but this is not as big an issue as overflow limit cycles.



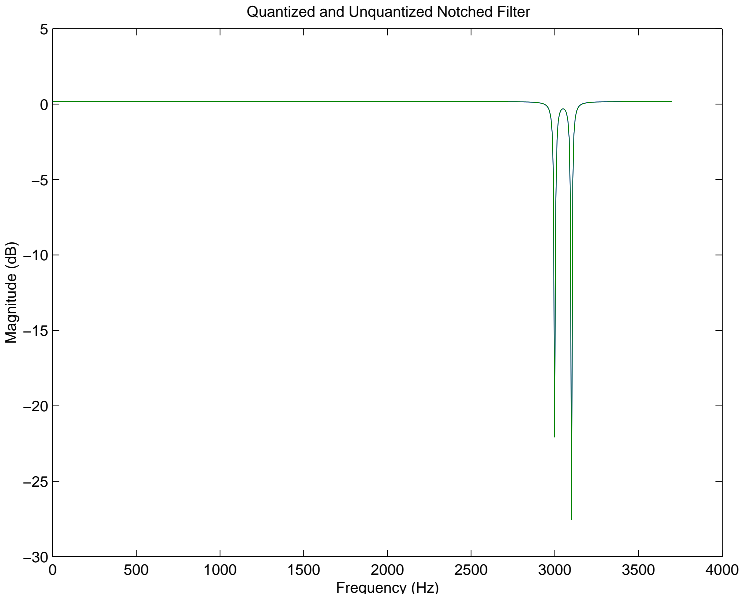
Quantized Filter Frequency Response Analysis

You can use `freqz` to analyze the frequency response of a quantized filter.

```
[H,F,units,Hr] = freqz(Hq2,512,Fs);
```

This syntax allows you to compare the frequency response `H` of the quantized filter, to that (`Hr`) of the reference filter.

```
plot(F,20*log10(abs([H Hr])));
ylabel('Magnitude (dB)')
xlabel('Frequency (Hz)')
legend('Quantized','Reference',3)
```



The two responses are almost identical.

Filtering with Quantized Filters

Now that you've designed a quantized filter you are happy with, use the `filter` command to apply it to the noisy speech signal and see how well it does.

```
y = filter(Hq2,u/5);
```

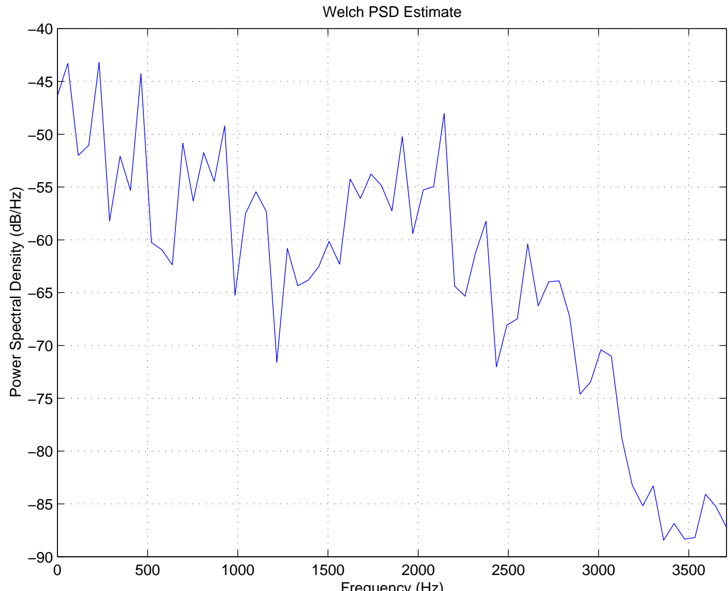
This scaling of the input is to avoid overflows.

You can listen to the filtered speech signal by typing

```
sound(y)
```

Sounds pretty good. The power spectral density also looks like the original.

```
pwelch(y,[],[],nfft,Fs)
```



Analyzing the filter Function Logged Results

You can use an alternate syntax for the `filter` command to monitor the maximum and minimum values as well as the overflows and underflows that occur during filtering. Suppose you didn't realize there would be input overflows and hadn't scaled the input.

```
warning on
y = filter(Hq2,u);
Warning: 1557 overflows in QFILT/FILTER.
                           Max              Min      NOverflows
NUnderflows     NOperations
 Coefficient         0.8612           0.49             0
0               6
                    0.8699        0.4901               0
0               6
```

```
        Input          4.127         -3.665          1557
0            4001
       Output              1             -1             0
0            4001
Multiplicand            1.81         -1.759          2637
0           32008
                           2             -2          1964
0           28007
      Product           1.81         -1.759             0
0           32008
                           2             -2             0
0           28007
          Sum          1.095         -1.248             0
0           20005
                       1.688         -1.604             0
0           20005
```

A report of all underflows and overflows is displayed when you filter the data. `qreport(Hq2)` provides the logged function output as well.

Example — A Quantized Filter Bank

You can use filter banks to create a set of filters that partition input signals into separate frequency bands or channels. Discrete Fourier Transform (DFT) polyphase FIR filter banks [3] provide a computationally efficient way to implement a filter bank that supports a large number of channels. Some cell phone base stations use DFT polyphase FIR fixed-point filter banks.

The polyphase DFT FIR filter bank is equivalent to a bank of long FIR filters operating at a relatively high sample rate.

A model for a polyphase DFT FIR filter bank is shown below. The impulse response coefficients of the original FIR filter are sampled and partitioned among the 16 FIR filters $H_i(z)$, i=1, ... , 16. The incoming signal is successively delayed and downsampled, before it enters any of the FIR filters. The outputs of the FIR filters are then scaled and sent through an FFT. The 16 outputs of the FFT represent the 16 channel signals.

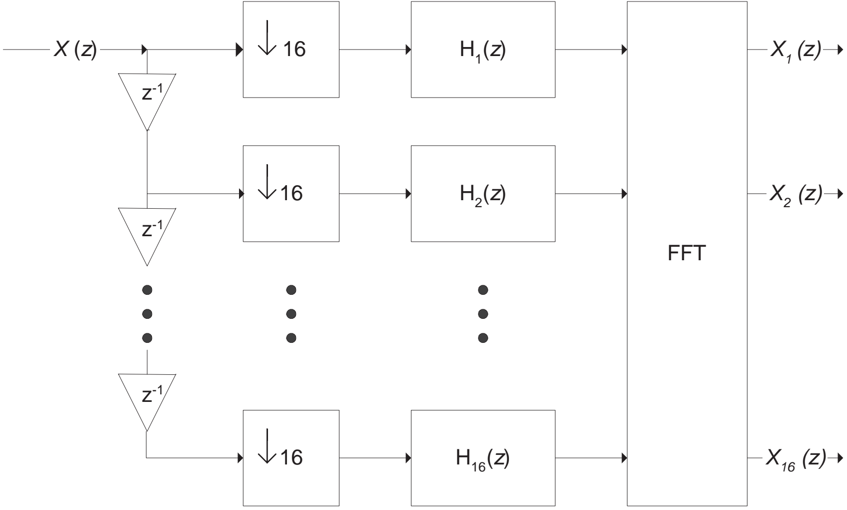


Figure 10-1: Model for a Polyphase DFT FIR Filter Bank

You can follow the example in this section to create a bank of DFT polyphase FIR fixed-point filters using quantized filters and quantized FFTs.

Filtering Data with the Filter Bank

To implement the filter bank shown in Figure 10-1, Model for a Polyphase DFT FIR Filter Bank, on page 10-17:

1. Create a quantized filter bank of 16 FIR filters followed by a quantized FFT. For linear analysis, adjust the `ScaleValues` property of the quantized FFT so that no overflows occur.
2. Successively delay and downsample an incoming data stream so that every ith signal sample enters the ith FIR filter.
3. Filter the data through the bank of FIR filters using `filter` on each quantized filter in the bank.
4. Put the output of the bank of filters through a 16-point FFT using `fft` on the quantized FFT.
5. Rescale the output of the FFT to account for the scaling introduced by its `ScaleValues` property.

Creating a DFT Polyphase FIR Quantized Filter Bank

This example follows the five steps listed in “Filtering Data with the Filter Bank” using a set of unit sinusoids at different frequencies for the incoming data.

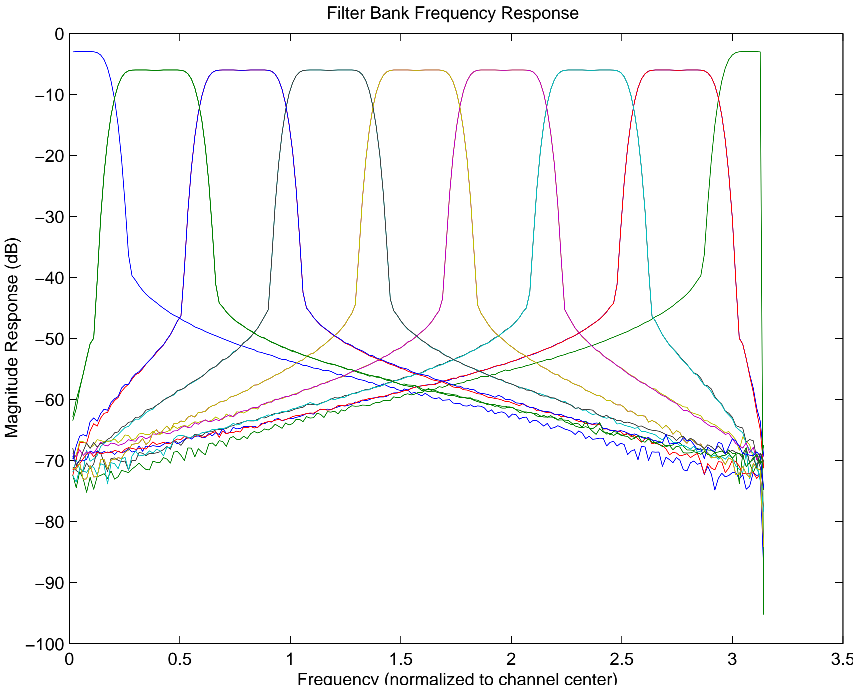
This demo takes some time to run and produces the two frequency response plots shown after the example code. You only see eight channels of filters in the magnitude response of the filter bank because FFTs produce conjugate signals for real-valued inputs. The second figure shows all 16 channels, presenting the channel amplitude for each channel.

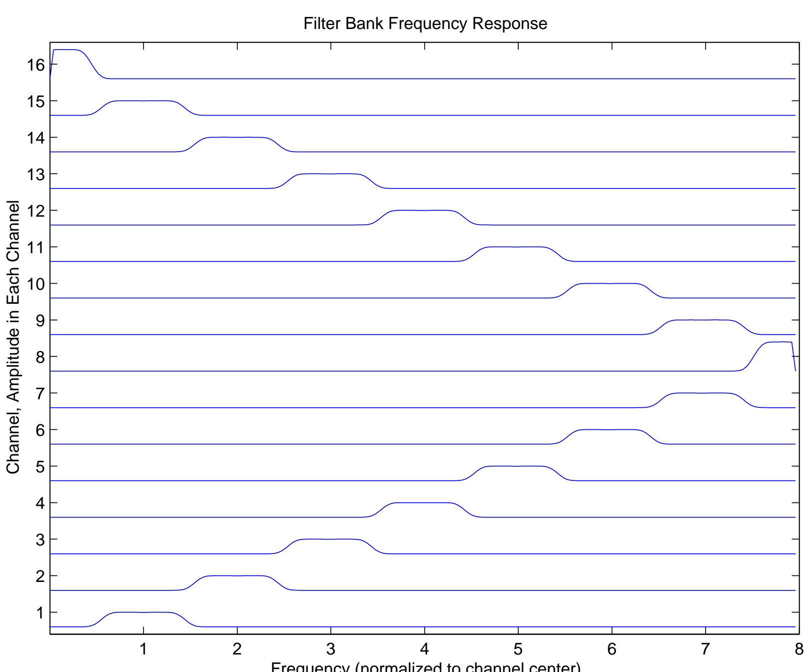
```
% Create a DFT Polyphase FIR Quantized Filter Bank.
% Initialize two variables to define the filters and the filter
% bank.
M = 16;  % Number of channels in the filter bank.
N = 8;   % Number of taps in each FIR filter.

% Calculate the coefficients b for the prototype lowpass filter,
% and zero-pad so that it has length M*N.
b = fir1(M*N-2,1/M);
```

```
b = [b,zeros(1,M*N-length(b))];

% Reshape the filter coefficients into a matrix whos rows
% represent the individual polyphase filters to be distributed
% among the filter bank.
B = flipud(reshape(b,M,N));

Hq = cell(M,1);
for k=1:M
  Hq{k} = qfilt('fir',{B(k,:)});
end

% Create a quantized FFT F of length M.
% Set the ScaleValues property value according to the
% NumberOfSections property value. Scale each section by 1/2.

F = qfft('length',M,'scale',0.5*ones(1,log2(M)));

% Retain the FFT scaling to weight the FFT correctly.
g = 1/prod(F.ScaleValues);

% Construct a bank of M quantized filters and an M-point quantized
% FFT. Filter a sinusoid that is stepped in frequency from 0 to
% pi radians, store the power of the filtered signal, and plot the
% results for each channel in the filter bank.

Nfreq = 200; % Number of frequencies to sweep.
w = linspace(0,pi,Nfreq);  % Frequency vector from 0 to pi.
P = 100;     % Number of output points from each channel.
t = 1:M*N*P; % Time vector.
HH = zeros(M,length(w));  % Stores output power for each channel.
for j=1:length(w)
  disp([num2str(j),' out of ',num2str(length(w))])
  x = sin(w(j)*t);           % Signal to filter

  % EXECUTE THE FILTER BANK:
  % Reshape the input so that it represents parallel channels of
  % data going into the filter bank.
  X = [x(:);zeros(M*ceil(length(x)/M)-length(x), 1)];
  X = reshape(X,M,length(X)/M);
```

```
    % Make the output the same size as the input.
    Y = zeros(size(X));

    % FIR filter bank.
    for k=1:M
      Y(k,:) = filter(Hq{k},X(k,:));
    end

    % FFT
    Y = fft(F,Y);

    HH(:,j) = var(Y.')';       % Store the output power.
  end

  % Compensate for FFT scaling.
  s = 1/prod(scalevalues(F));
  HH = HH*s^2;

  % Plot the results.
  figure(1)
  plot(w,10*log10(HH))
  title('Filter Bank Frequency Response')
  xlabel('Frequency (normalized to channel center)')
  ylabel('Magnitude Response (dB)')
  set(gca,'xtick',(1:M/2)*w(end)/M*2)
  set(gca,'xticklabel',(1:M/2))
  figure(2)
  strips(HH')
  set(gca,'yticklabel',1:M)
  set(gca,'xtick',(1:M/2)*Nfreq/M*2)
  set(gca,'xticklabels',(1:M/2))
  grid off
  title('Filter Bank Frequency Response')
  xlabel('Frequency (normalized to channel center)')
  ylabel('Channel, Amplitude in Each Channel')
```

Look at the next two figures to see the results of the example code.


Filter Bank Frequency Response
Magnitude Response (dB)
Frequency (normalized to channel center)
0
−10
−20
−30
−40
−50
−60
−70
−80
−90
−100
0
0.5
1
1.5
2
2.5
3
3.5


Filter Bank Frequency Response
Channel, Amplitude in Each Channel
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
1
2
3
4
5
6
7
8
Frequency (normalized to channel center)

Example — Effects of Quantized Arithmetic

When you filter data with a fixed-point quantized filter, your results may vary from those obtained by filtering with a double-precision reference filter. This is due to a number of factors, including:

- Quantization of the input to the filter
- Quantization of the output from the filter
- Quantization of the filter coefficients
- Quantization occurring during the various arithmetic operations performed by the filter

You can isolate the effects of fixed-point quantization that result solely from arithmetic operations by:

1. Creating quantizer `q` for data.
2. Creating a fixed-point filter `Hq` from a reference using quantizer `q` data formats.
3. Creating a double-precision quantized filter `Hd` from `Hq`, with the same (quantized) coefficients.
4. Quantizing a data set `x` according to the quantizer specifications.
5. Filtering the quantized data set `x` with both filters.
6. Comparing the results.

Creating a Quantizer for Data

Create a 16-bit default quantizer.

```
q = quantizer;
```

Creating a Fixed-Point Filter from a Quantized Reference

1. Create an example double-precision reference filter and quantize and scale the filter coefficients.

```
[b,a] = ellip(7,.1,40,.4);
c = quantize(q,{b/8, a/8}); % Coefficients are in a cell array.
```

2 Create a fixed-point quantized filter from the coefficients `c`, with data formats specified by the quantizer `q`.

```
Hq = qfilt('df2t',c,'quantizer',q);
```

Creating a Double-Precision Quantized Filter

You can create a quantized double-precision filter `Hd` from `Hq` by changing the value of the `Mode` property for each of the quantizers that specify the data formats of `Hq`.

```
Hd = Hq;
Hd.quantizer = 'double';
```

Quantizing a Data Set

Create a random data set and quantize it.

```
rand('state',0);
n = 1000;
x = quantize(q,0.5*(2*rand(n,1) - 1));
```

This data set is scaled to prevent overflows. If you do not prevent overflows, you cannot isolate the quantization effects of arithmetic.

Filtering the Quantized Data with Both Filters

Filter the quantized data with the double-precision filter and the fixed-point filter.

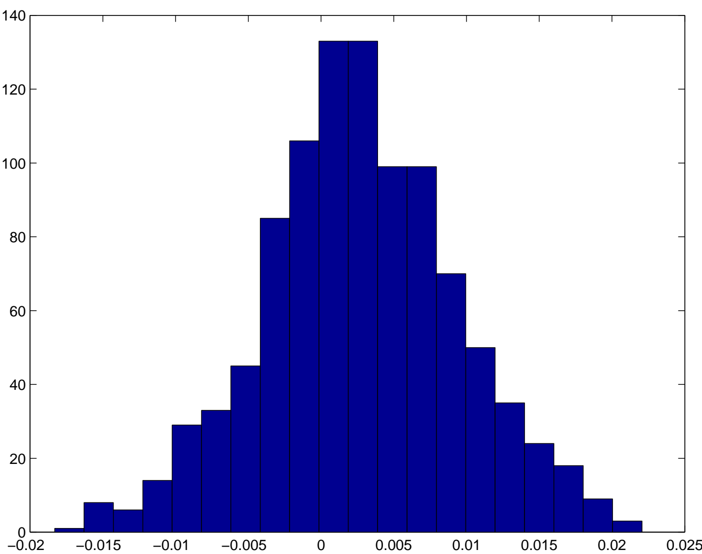
```
yq = filter(Hq,x);
yd = filter(Hd,x);
```

Comparing the Results

Analyze the error signal and its histogram.

```
e = yd - yq;
hist(e,20)
```

The error is approximately normally distributed. The nonzero mean is caused by choosing 'floor' for the rounding method.



11

Using FDATool with the Filter Design Toolbox

Switching FDATool to Quantization Mode (p. 11-4)	After you open FDATool, this section explain how to access the quantization features in the tool.
Quantizing Filters in the Filter Design and Analysis Tool (p. 11-7)	Explains how you quantize a filter in FDATool.
Analyzing Filters with the Noise Loading Method (p. 11-12)	FDATool provides a variety of analysis methods for quantized filters; this section explains how to use them.
Optimizing the Quantization Process For Your Filter (p. 11-19)	You can adjust the way FDATool quantizes filters. To learn how, read this section.
Importing and Exporting Quantized Filters (p. 11-29)	Shows you how to import and export filters to and from your MATLAB workspace, as well as to other destinations.
Transforming Filters (p. 11-34)	Describes how you use the filter transformation capability in FDATool to change the magnitude response of your FIR or IIR filters in the tool.
Realizing Filters as Simulink Subsystem Blocks (p. 11-45)	Using the Realize Model feature to create a Simulink model of your quantized filter as a subsystem block.
Getting Help for FDATool (p. 11-49)	Shows you how to get help about the features in FDATool, such as using Help or using the What's This option.

The Filter Design Toolbox adds a new dialog and operating mode, and a new menu selection, to the Filter Design and Analysis Tool (FDATool) provided by the Signal Processing Toolbox. From the new dialog, titled **Set Quantization Parameters**, you can:

- View Simulink models of the filter structures available in the toolbox.
- Quantize double-precision filters you design in this GUI using the design mode.
- Quantize double-precision filters you import into this GUI using the import mode.
- Perform analysis of quantized filters.
- Scale the transfer function coefficients for a filter to be less than or equal to 1.
- Select the quantization settings for the properties of the quantized filter displayed by the tool:
  - Coefficient
  - Input
  - Output
  - Multiplicand
  - Product
  - Sum
- Change the input and output scale values for a filter.

After you import a filter in to FDATool, the options on the quantization dialog let you quantize the filter and investigate the effects of various quantization settings.

From the new selection on the FDATool 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.

This section presents the following information and procedures for using FDATool:

- "Switching FDATool to Quantization Mode" on page 11-4
- "Quantizing Filters in the Filter Design and Analysis Tool" on page 11-7
- "Choosing Your Quantized Filter Structure" on page 11-16
- "Scaling Transfer Function Coefficients" on page 11-24
- "Scaling Inputs and Outputs of Quantized Filters" on page 11-26

Switching FDATool to Quantization Mode

You use the quantization mode in FDATool to quantize filters. Quantization represents the fourth operating mode for FDATool, along with the filter design, filter transformation, and import modes. To switch to quantization mode, open FDATool from the MATLAB command prompt by entering

```
fdatool
```

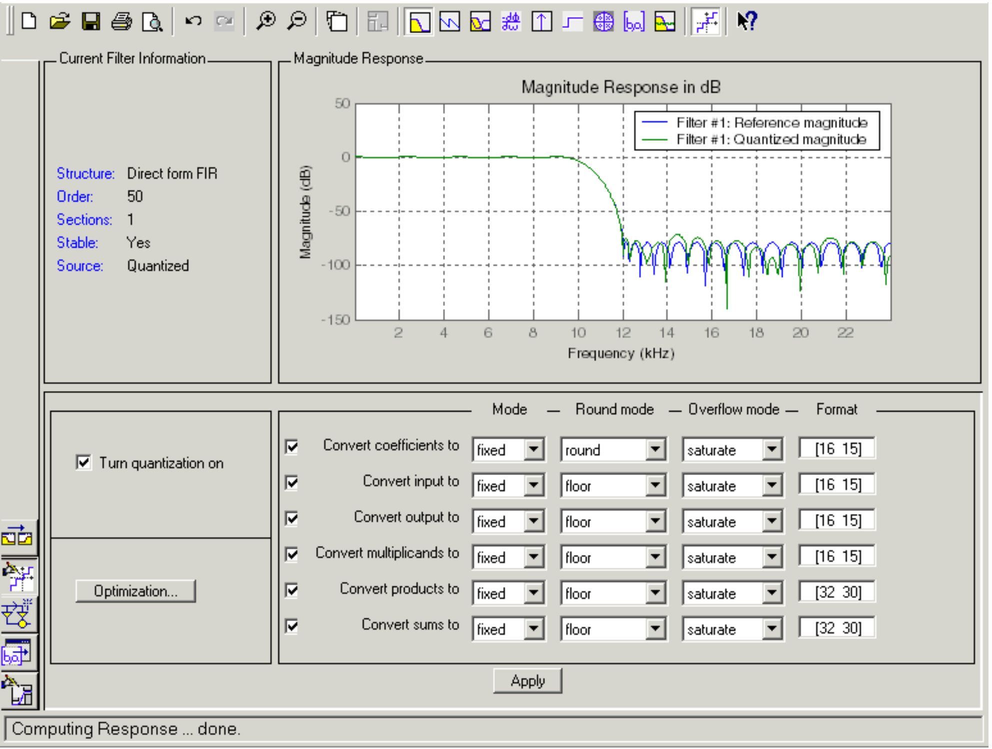
When FDATool opens, click **Set Quantization Parameters**. FDATool switches to quantization mode and you see the following panel at the bottom of FDATool, with the default values shown. Controls within the dialog let you quantize filters and investigate the effects of changing quantization settings. To enable the quantization options, perform these steps:

1. Click on the FDATool menu bar.

   If you have designed or imported a filter into FDATool, you now see two filter magnitude plots in the analysis area. One is your original filter, the other is your filter after quantization.

2. Click in the side bar.

   The quantization options appear in the lower panel of FDATool. You see the settings for each quantizer in the filter.



You use the following controls in the dialog to perform tasks related to quantizing filters in FDATool:

- **Turn quantization on** button —quantizes the filter displayed in **Current Filter Information**.
- **Set quantization parameters** —changes Filter Design and Analysis Tool to quantization mode to configure and quantize filters that you design or import.
- **Optimization**—lets you set a variety of options for quantizing your filter, such as scaling the filter transfer function coefficients to be less than or equal to one.
- **Apply**—applies changes you make to the quantization parameters for your filter.

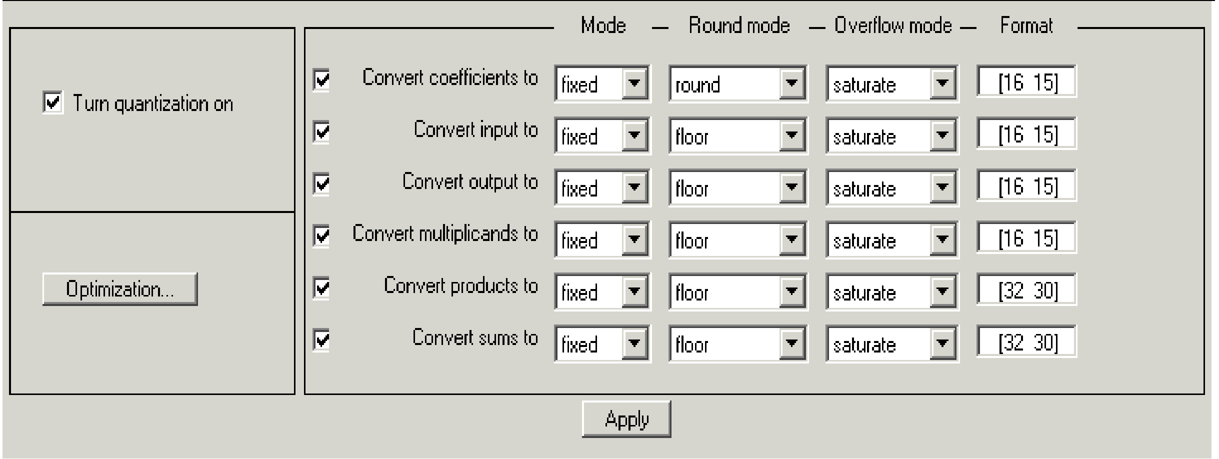
- Quantizer property lists, such as **Convert coefficient to** and **Convert multiplicand to**— these lists let you set values for the properties of the quantizers that constitute your quantized filter. Under **Format**, the entries contain [`wordlength fractionlength`] for each quantizer property.

Quantizing Filters in the Filter Design and Analysis Tool

Quantized filters have properties that define how they quantize data you filter. Use the **Set Quantization Parameters** dialog in FDATool to set the properties. Using options in the **Set Quantization Parameters** dialog, FDATool lets you perform a number of tasks:

- Create a quantized filter from a reference filter after either importing the reference filter from your workspace, or using FDATool to design the reference filter.
- Create a quantized filter that has the default structure (Direct form II transposed) 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**, the quantized filter panel opens in FDATool, with all options set to default values.



To let you set the properties for the six quantizers that make up a quantized filter, FDATool lists each quantizer. Table 11-1 lists each component

quantizer, its full property name, and includes a short description of what the quantizer does in the filter.

Table 11-1: These Quantizers Define the Behavior of a Quantized Filter

Quantizer	Filter Property Name	Description
Convert coefficient to	`CoefficientFormat`	Determines how the coefficient quantizer handles filter coefficients. When you quantize a filter, the properties of this quantizer govern the quantization.
Convert input to	`InputFormat`	Specifies how data input to the filter is quantized.
Convert output to	`OutputFormat`	Specifies how date output by the filter is quantized.
Convert multiplicand to	`MultiplicandFormat`	Specifies how filter multiplicands are quantized. Multiplicands are the inputs to multiply operations.
Convert product to	`ProductFormat`	Determines how to quantize the results of multiply operations.
Convert sum to	`SumFormat`	Determines how to quantize the results of arithmetic sums in the filter.

Every quantizer has four properties. For each quantizer, such as **Convert Coefficient** and **Convert Output**, you select values for its properties to determine how the filter performs quantization. The properties that make up each quantizer in a quantized filter are listed in Table 9-2.

Table 11-2: Four Properties Specify Each Quantizer

Quantizer Property	Description
Mode	Selects one of four arithmetic modes for the quantizer: • `fixed`—to specify fixed-point arithmetic. `fixed` is the default setting. • `float`—to specify floating-point arithmetic • `double`—to specify double-precision arithmetic • `single`—to specify single-precision arithmetic
Round mode	Sets the way in which the quantizer handles values after it quantizes them. You have five options to choose from: • `ceil`—round values to the nearest integer towards plus infinity. • `convergent`—round values to the nearest integer, except in a tie, then round down if the next-to-last bit is even, up if odd. • `fix`—round values to the nearest integer towards zero. • `floor`—round values to the nearest integer towards minus infinity. The default setting for all quantizers except the **Coefficient** quantizer. • `round`—round values to the nearest integer. Negative numbers that lie halfway between two values are rounded towards negative infinity. Positive numbers that lie halfway between two values are rounded towards positive infinity. Ties round toward positive infinity. The default setting for the **Coefficient** quantizer.

Table 11-2: Four Properties Specify Each Quantizer (Continued)

Quantizer Property	Description
Overflow mode	When the result of a quantization operation exceeds the range that the format can represent, this value tells the quantizer how to handle the overflow. Choices are • `saturate`—set values that fall outside the representable range to the minimum or maximum values in the range. Values greater than the maximum value are set to the maximum range value. Values less than the minimum value are set to the minimum range value. This is the default setting. • `wrap`—map values that fall outside the representable range of the format back into the range using modular arithmetic.
Format	Specifies the word length and fraction length for the **Mode** value you specified. `[16 15]` is the default setting for word length and fraction length. Notice that the Product and Sum quantizers default to [2*word length 2*fraction length], or `[32 30]`.

To Quantize Reference Filters

When you are quantizing a reference filter, follow these steps to set the **Coefficient** property values that control the quantization process. Before you begin, verify that **Turn quantization on** is not selected:

1. Click **Set Quantization Parameters** to open the **Set Quantization Parameters** dialog.
2. Select **Turn quantization on**.

   When you turn quantization on, FDATool quantizes the current filter according to the **Coefficient** properties, and changes the information displayed in the analysis area to show quantized filter data.

3. Review the settings for the **Convert coefficient to** properties: **Mode**, **Round mode**, **Overflow mode**, and **Format**.

4. Change the **Convert coefficient to** properties as required to quantize your filter correctly.

5. Click **Apply**.

   FDATool quantizes your filter using the new settings.

6. Use the analysis features in FDATool to determine whether the new quantized filter meets your requirements.

To Change the Quantization Properties of Quantized Filters

When you are changing the property values for a quantized filter, or after you import a quantized filter from your MATLAB workspace, follow these steps to set the property values for the quantized 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 quantizers: **Convert coefficient to**, **Convert input to**, **Convert output to**, **Convert multiplicand to**, **Convert product to**, and **Convert sum to**. Settings for these properties 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 2.

5. Use the analysis features in FDATool to determine whether your new quantized filter meets your requirements.

Analyzing Filters with the Noise Loading Method

One technique for estimating the frequency response for quantized filters is the noise loading method (NLM) provided by function `nlm` in this toolbox. FDATool offers the noise loading method as a filter analysis tool accessible from the toolbar.

Using the Noise Loading Method

After you design and quantize your filter, the noise loading method on the **Analysis** menu lets you apply the noise loading method to your filter. When you select **Analysis -> Noise Loading Method** from the menubar, FDATool 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 FDATool.

With the NLM, you estimate the complex frequency response for your filter as determined by applying a noise- like signal to the filter input. NLM 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, NLM uses default settings for the various conditions that define the process, such as the number of test points and the number of trials.

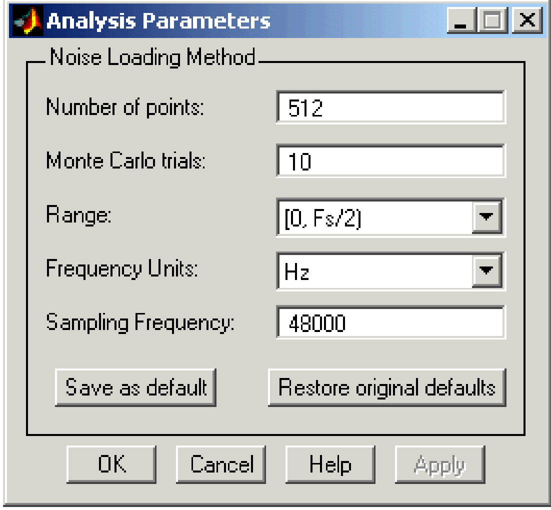
Analysis Parameter	Default Setting	Description
Number of points	512	Number of equally spaced points around the upper half of the unit circle.
Number of Monte Carlo trials	10	Number of times to repeat the Monte Carlo test to get an average frequency response.
Range	0 to Fs/2	Frequency range of the plot x-axis.
Frequency units	Hz	Units for specifying the frequency range.
Sampling frequency	48000	Inverse of the sampling period.

After your first analysis run ends, open the **Analysis Parameters** dialog and adjust your settings appropriately, such as changing the numer of trials or number of points.

To open the Analysis Parameters dialog, use either of the next procedures when you have a quantized filter in FDATool:

- Select **Analysis -> Analysis Parameters...** from the menu bar
- Right-click in the analysis area and select **Analysis Parameters...** from the context menu

Whichever option you choose opens the dialog as shown in the figure. Notice that the settings for the options reflect the defaults.



Example—Noise Loading Method Applied to a Filter

To demonstrate the NLM in use, start by creating a quantized filter. For this example, use FDATool to design a sixth-order Butterworth IIR filter.

To Use NLM Analysis in FDATool

1. Type `fdatool` at the MATLAB prompt to launch FDATool.
2. In **Design Method**, select **IIR** and **Butterworth** from the list.

3 Under **Filter Type**, select **Highpass**.

4 To set the filter order to 6, select **Specify order** under **Filter Order**. Type 6 in the text box.

5 Click **Design Filter**.

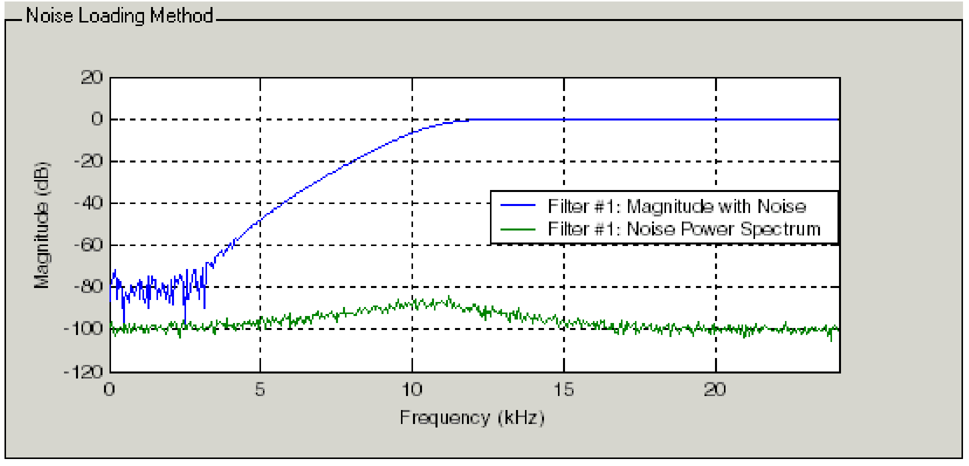
In FDATool, the analysis area changes to display the magnitude response for your filter.

6 To generate the quantized version of your filter, using default quantizer settings, click [icon] on the toolbar.

Now the analysis areas shows the magnitude response for both filters—your original filter and the quantized version.

7 Finally, to use NLM on your quantized filter, select **Analysis -> Noise Loading Method** from the menubar.

FDATool runs the NLM Monte Carlo trials, calculates the average magnitude response for the filter, and displays the result in the analysis area as shown in this figure.



In the figure you see both the magnitude response and the noise power spectrum used to determine the response.

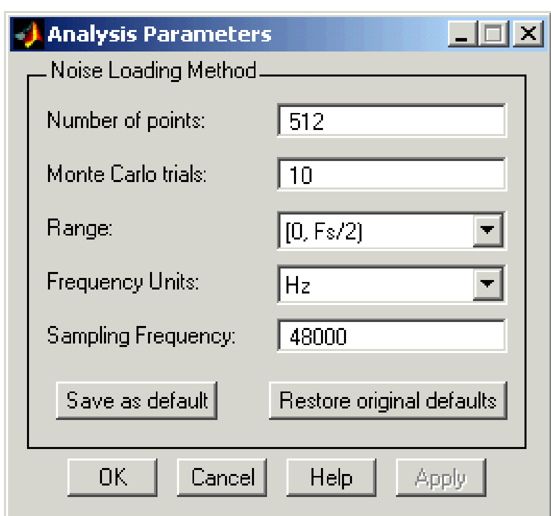
To Change Your NLM Analysis Parameters

In “Example—Noise Loading Method Applied to a Filter”, you used NLM to estimate the magnitude response for a quantized highpass Butterworth filter. Since you ran NLM only once FDATool, your noise loading analysis used the default NLM settings shown in “Using the Noise Loading Method”.

To change the settings, follow these steps after the first time you use NLM on your quantized filter.

1 With the results from running the noise loading method displayed in the FDATool analysis area, right-click in the area and select **Analysis Parameters...**.

To give you access to the analysis parameters, the Analysis Parameters dialog opens as shown here (with default settings).



2 To use more trials to estimate the magnitude response, change **Monte Carlo trials** to 20 and click **OK** to run the analysis.

   FDATool closes the **Analysis Parameters...** dialog and reruns NLM, returning the results in the analysis area.

   To rerun NLM without closing the dialog, press **Enter** after you type your new value into a setting, then click **Apply**. Now FDATool runs NLM without closing the dailog. When you want to try many different settings for the noise loading analysis, this is a useful shortcut.

Comparing the NLM and Theoretical Magnitude Responses

An important measure of the effectiveness of the noise loading method for estimating the magnitude response of a quantized filter is to compare the NLM response to the theoretical response. To see a comparison of the two diverse methods, refer to the online reference page for `nlm`.

Choosing Your Quantized Filter Structure

FDATool 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 quantized filter in FDATool, refer to "Converting to a New Structure" in your Signal Processing Toolbox documentation.

Converting 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" on page 12-24
- "Direct Form I Transposed Filter Structure" on page 12-20
- "Direct Form II Filter Structure" on page 12-22
- "Direct Form I Filter Structure" on page 12-18
- "Direct Form Finite Impulse Response (FIR) Filter Structure" on page 12-26

- "Direct Form FIR Transposed Filter Structure" on page 12-27
- "Lattice Autoregressive Moving Average (ARMA) Filter Structure" on page 12-34
- "Lattice Coupled-Allpass Filter Structure" on page 12-30
- "Lattice Coupled-Allpass Power Complementary Filter Structure" on page 12-31
- "State-Space Filter Structure" on page 12-35

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
- State space
- Lattice ARMA

Additionally, FDATool lets you do the following conversions:

- Minimum phase FIR filter to Lattice MA minimum phase
- Maximum phase FIR filter to Lattice MA maximum phase
- Allpass filters to Lattice allpass

Refer to "FilterStructure" on page 12-12 for details about each of these structures.

To Convert Your Filter to Second-Order Sections Form

To learn about using FDATool to convert your quantized filter to use second-order sections, refer to "Converting to Second-Order Sections" in your Signal Processing Toolbox documentation.

To View Schematics of Filter Structures in the Toolbox

Often it helps to see the structure of a filter. From the **Set Quantization Parameters** dialog in FDATool, the **Show filter structures** option opens

a demonstration program that provides Simulink models of each filter structure included in the toolbox.

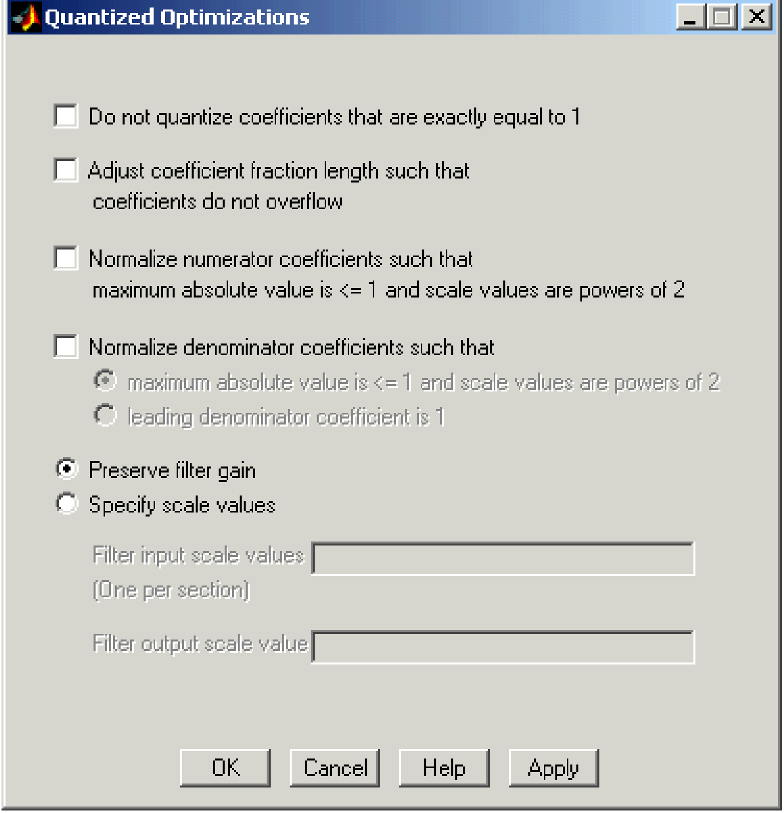
To View Filter Structures in FDATool

To open the demonstration, click **Help -> Show filter structures**. After the Help browser opens, select the filter structure to view from the table of filter structures.

Optimizing the Quantization Process For Your Filter

By clicking **Optimization...** on the Quantized Filter panel, you launch the **Quantized Optimizations** dialog. Using the controls provided on the dialog, you direct FDATool about specific features of the quantization process.

As shown here, the **Quantized Optimizations** dialog lets you determine how quantization affects your filter coefficients and what happens when you scale your filter.



Control Coefficient Quantization

With the **Quantized Optimizations** dialog open, you can use the **Do not quantize coefficients that are exactly equal to 1** option to prevent FDATool from quantizing coefficients, both numerator and denominator (also called b and a) whose value is 1.0

Directing FDATool not to quantize 1.0 coefficients has two advantages for your filter:

1. Eliminates one multiply operation for each such coefficient during filter quantization. When your coefficient is equal to 1.0, the quantization process skips the multiply associated with the coefficient, making the process more efficient.

2. Reduces the possible error that can result when a coefficient whose value is 1.0 gets quantized to a value that is not exactly 1.0. Changing the value might alter your filter performance.

To stop FDATool from quantizing coefficients equal to 1.0, follow these steps:

1. Open the **Quantized Optimizations** dialog.

2. Select **Do not quantize coefficients that are exaactly equal to 1**.

3. Click **Apply** to quantize your filter with the new setting, or click **OK** to quantize your filter and close the dialog.

Limit Coefficient Overflow By Fraction Length Changes

One way to reduce the possibility that the coefficients of you filter overflow during quantization is to let the fraction length of the coefficient format change during quantization. FDATool provides an option that automatically adjusts the fraction length for a quantizer to prevent the coefficients from exceeding the range of the format.

The **Set Quantization Parameters** panel in FDATool shows the format in place for the **Convert coefficient to** quantizer in the **Format** column. Given as a vector, such as [16 15] (the default value), you can check the word length and fraction length for representing your filter coefficients here.

When you select **Adjust coefficient fraction length such that coefficients do not overflow** on the **Set Quantization Parameters** panel, FDATool varies the fraction length from the format you set on the **Quantized Filter** panel in FDATool.

To elect to let FDATool adjust the fraction length for your filter coefficients to prevent overflows, perform the following steps:

1. Open the **Quantized Optimizations** dialog.
2. Select **Adjust coefficient fraction length such that coefficients do not overflow**.
3. Click **Apply** to quantize your filter with the new setting, or click **OK** to quantize your filter and close the dialog.

Normalizing Transfer Function Coefficients

One way to prevent your filter coefficients from overflowing and to maintain well-behaved filters after quantization is to normalize the coefficients so the absolute value of every coefficient is 0.5< = coefficient value < 1.

To provide you with the flexibility to decide how and which coefficients to normalize, the **Quantized Optimizations** dialog provides several options for specifying the treatment of filter function coefficients. In this table, you see a summary of the options and what they do. Following the table are detailed

descriptions of the options and how you use them. Note that some of the options depend on one another.

Control Name	Description
“Normalize numerator coefficients such that maximum absolute value is < = 1 and scale values are powers of 2”	Performs a two step process of normalizing and scaling the transfer function coefficients (b) to produce well-behaved filters after quantization.
“Normalize denominator coefficients such that”	Specify whether to normalize the denominator coefficients (a) of the filter transfer function. Selecting this option enables two other options that determine how to quantize the denominator coefficients.
“maximum absolute value is < = 1 and scale values are powers of 2”	In combination with the **Normalize denominator...** option preceding, specifies that the coefficients after normalization are between -1 and 1 and that the scale values used for normalizing are powers of 2. Compare to the **Normalize numerator...** option.
“leading denominator coefficient is 1”	Combined with the **Normalize denominator...** option, directs FDATool to normalize the leading denominator coefficient a(1) to be exactly 1.0
“Preserve filter gain when normalizing coefficients”	Directs FDATool to adjust the gain of the filter so the response after normalizing the coefficients is the same as before. This is the default setting.

Normalize numerator coefficients such that maximum absolute value is < = 1 and scale values are powers of 2

Since in some cases your filter coefficient vector can contain values that are greater than 1, you can choose to normalize and scale the coefficients before quantization. Normalizing the coefficients reduces the sensitivity of your filter to the effects of quantization. Adjusting the normalization so the resulting scale factors are powers of two makes the quantization process be more efficient. When your scale factors are powers of 2, the multiply operations required to apply the scale factors can be replaced by a simple shift—much more efficient.

Normalize denominator coefficients such that

To reduce the effects of quantization on your filter, normalizing the denominator coefficients adjust the values in the coefficient vector so that all values in the vector are > = 0.5 and < 1. When you select this option, you enable two more options that let you tailor the quantization to your needs:

- **maximum absolute value is < = 1 and scale values are powers of 2**
- **leading denominator coefficient is 1**

These controls produce the same results for denominator coefficients that the normalization options produce for numerator coefficients.

maximum absolute value is < = 1 and scale values are powers of 2

Since in some cases your filter coefficient vector can contain values that are greater than 1, you can choose to normalize and scale the coefficients before quantization. Normalizing the coefficients reduces the sensitivity of your filter to the effects of quantization. Adjusting the normalization so the resulting scale factors are powers of two makes the quantization process be more efficient. When your scale factors are powers of 2, the multiply operations required to apply the scale factors can be replaced by a simple shift—much more efficient.

leading denominator coefficient is 1

With this option selected, FDATool divides the members of the denominator coefficient vector by the value of the leading coefficient, forcing the leading coefficient to be 1. Note that this occurs after normalization and scaling

Preserve filter gain when normalizing coefficients

Filters in FDATool are in transfer function form. To mitigate the effects of quantization on the performance of your filter, you can normalizethe transfer function coefficients. After you import or design a filter in FDATool (to create your reference filter), you can opt to normalize the filter transfer function coefficients not to exceed ±1. Normalizing the coefficients prevents overflow and underflow conditions from occurring during quantization.

A few things to note about using normalizing:

- When you choose to normalize your transfer function coefficients, in either the numerator or denominator, FDATool does two things:
  - It normalizes the coefficients as directed by your choice of options on the **Quantized Optimizations** dialog.
  - It changes the filter gain to keep the filter magnitude response the same after normalizing the coefficients. If FDATool did not change the gain, the response of the filter to a given input would change when you chose to normalize the coefficients.
- FDATool cannot restore the transfer function coefficients back to their values before normalization. Clearing and applying the options on the dialog does not restore your filter to the state before normalization. So the resulting filter may demonstrate changed magnitude response after you remove the normalization. To get back to your original filter, either redesign or reimport the filter.

Scaling Transfer Function Coefficients

All filters in FDATool are in transfer function form. To mitigate the effects of quantization on the performance of your filter, you can scale the transfer function coefficients. After you import or design a filter in FDATool (to create your reference filter), you can scale the filter transfer function coefficients not to exceed ±1. Scaling the coefficients prevents overflow and underflow conditions from occurring during quantization.

A few things to note about using scaling:

- When you choose to scale your transfer function coefficients, FDATool does two things:
  - It scales the coefficients as directed.

- It changes the filter gain to keep the filter magnitude response the same after scaling. If FDATool did not change the gain, the response of the filter to a given input would change when you scaled the coefficients.

- If you remove the scaling factors, FDATool restores the transfer function coefficients to their values before scaling. FDATool does not remove the filter gain it added when you scaled the coefficients. So the resulting filter may demonstrate changed magnitude response after you remove the scale factors.

To Scale Transfer Function Coefficients

To scale the transfer function coefficients of a filter in FDATool, follow these steps:

1. Design a filter, or import a filter into FDATool. This is your reference filter.

   Under **Current Filter Information,** the characteristics of your filter are structure, source, order, and whether the filter is stable.

2. Click **Set Quantization Parameters**.

   The bottom half of the FDATool window (the quantization region) shows the options for quantizing a filter, including options for scaling filter transfer function coefficients and setting the property values for the quantization properties of the filter.

3. Select **Turn quantization on** to quantize the filter in Current Filter Information.

   You can review the transfer function coefficients for your filter. Select **View Filter Coefficients** from the **Analysis** menu. The analysis area changes to list the coefficients for the reference and quantized filters. Scroll through the list to review the coefficients and to check for coefficient overflow or underflow that can occur during quantization.

   Notice that the left column in the analysis area contains symbols. They indicate whether the quantized coefficient over- or underflowed during quantization. A minus sign signals that the coefficient on that line overflowed toward positive infinity. A plus sign indicates an overflow toward negative infinity. Coefficients marked with zero had reference values that underflowed to zero.

4. Click **Scale transfer-fnc coeffs <=1**.

5. Review the scaled coefficients to see that no overflow warning appears at the end of the list of coefficients

   ```
   Warning: 1 overflow in coefficients.
   ```

   and no plus, zero, or minus symbols appear in the left column.

Once you have scaled a filter, you cannot remove the scale factors. You must recreate the filter from the beginning by redesigning or reimporting the filter.

Scaling Inputs and Outputs of Quantized Filters

For any filter structure, each filter section has two scale values associated with it—an input and an output. When you select **Help -> Show filter structures...** to look at the filter structures provided by FDATool, you do not see that each structure includes at least two scale values, `s(1)` and `s(2)`. If the filter has multiple sections, the number of scale values is (number of sections +1). For example, a filter with three sections has four scale values because the output scale value for each section is the input value to the next section:

- `s(1)`—input scale value for the first section
- `s(2)`—output scale value from the first section and the input scale value to the second section
- `s(3)`—output scale value from the second section and input scale to the third section
- `s(4)`—output scale value from the third section

So the number of scale values you need for your filter depends on the filter structure.

Enter input and output scale values in four ways in **Filter input values** and **Filter output value**:

1. Select Specify scale values.

2. Do one of the following to enter your input scale values:

- Enter a scalar. FDATool uses the scalar in **Filter input values** for every scale value in your structure.

- Enter a vector of scale values in **Filter input values**. The vector can be up to length (number of sections +1), where each scale value entry is a real number. FDATool assigns the scale values in the order `s(1)`, `s(2)`, `s(3)`,...`s` (number of sections + 1). When your vector contains fewer values than the number of scale values required for your filter structure, FDATool assigns the values in order until it uses all the values in the vector. Remaining scale values are set to one and are omitted during scaling or filtering.
- Enter a variable name that represents a vector in your MATLAB workspace. The length of the vector can be up to (number of sections +1).

3 (Optional) Enter a scale value for the output scaling by doing one of the following steps:

- Enter a scalar. FDATool uses the scalar in **Filter output value** for the output scale value in your structure.
- Enter a variable name that represents a vector in your MATLAB workspace.

4 Click **Apply**.

Scale values that are exactly equal to one are omitted during filtering and scaling, avoiding the associated multiplication operation.

To Enter Scale Values for Quantized Filters

Scale values apply to quantized filters. To specify the scale values for the current quantized filter in FDATool, follow these steps:

1 Click on the side bar.

2 Check or determine the number of sections in your filter.

The number of scale values you need for your filter depends on the number of sections used in the filter design. For example, a filter with four sections requires you to enter either one scale value or up to 5 (the number of sections +1).

3 Enter one of the following into **Filter input scale values**:

   a A scalar. FDATool uses the scalar for the input scale value in the filter.

   b A vector of scale values. The vector can be up to (number of sections +1) elements, where each entry is a real number.

   c A variable name that represents a vector in your MATLAB workspace. The length of the vector in the workspace can be up to (number of sections +1) elements.

4 (Optional) Enter one of the following into **Filter output scale value**:

   a A scalar. FDATool uses the scalar for the output scale value in the filter.

   b A variable name that represents a scalar in your MATLAB workspace. FDATool uses the scalar for the output scale value in the filter.

5 Click **Apply**.

Importing and Exporting Quantized Filters

When you import a quantized filter into FDATool, or export a quantized filter from FDATool 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.

This section includes:

- "To Import Quantized Filters"
- "To Export Quantized Filters"

For general information about importing and exporting filters in FDATool, refer to "Filter Design and Analysis Tool" section in your *Signal Processing Toolbox User's Guide*.

FDATool imports quantized filters having the following structures:

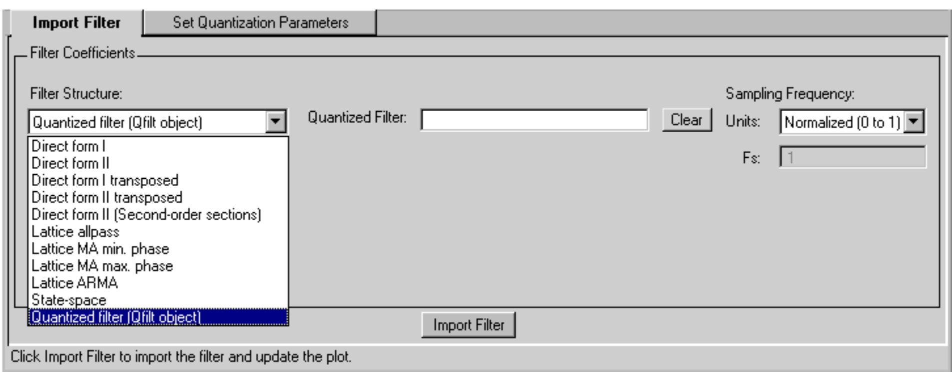
- Direct form I
- Direct form II
- Direct form I transposed
- Direct form II transposed
- Direct form symmetric FIR
- Direct form antisymmetric FIR
- Lattice allpass
- Lattice AR
- Lattice MA minimum phase
- Lattice MA maximum phase
- Lattice ARMA
- Lattice coupled-allpass
- Lattice coupled-allpass power complementary
- State-space

To Import Quantized Filters

After you design or open a quantized filter in your MATLAB workspace, FDATool lets you import the filter for analysis. Follow these steps to import your filter in to FDATool:

1. Open FDATool.
2. Select **Filter->Import Filter** from the menu bar.

   In the lower region of FDATool, the **Design Filter** tab becomes **Import Filter**, and options appear for importing quantized filters, as shown.



3. From the **Filter Structure** list, select `Quantized filter (Qfilt object)`.

   The options for importing filters change to include:

   - **Quantized filter**—Enter the variable name for the quantized filter in your workspace. You can also enter `qfilt` to direct FDATool to construct a quantized filter. When you enter `qfilt`, FDATool creates a quantized filter according to the `qfilt` syntax you use.
   - **Frequency units**—Select the frequency units from the **Units** list, and specify the sampling frequency value in **Fs**. Your sampling frequency must correspond to the units you select. For example, when you select `Normalized (0 to 1)`, **Fs** should be one.

4 Click **Import** to import or construct the filter.

FDATool 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 you entered the quantized filter constructor in **Quantized filter**, FDATool creates the filter and displays the filter magnitude response.

To Export Quantized Filters

To save your filter design, FDATool lets you export the quantized filter to your MATLAB workspace (or you can save the current session in FDATool). When you choose to save the quantized filter by exporting it, you select one of these options:

- Export to your MATLAB workspace
- Export to a text file
- Export to a MAT-file

Exporting Coefficients or Objects to the Workspace

You can save the filter as filter coefficients variables or as a `dfilt` or `qfilt` filter object variable. To save the filter to the MATLAB workspace:

1 Select **Export** from the **File** menu. The **Export** dialog appears.

2 Select `Workspace` from the **Export To** list.

3 Select `Coefficients` from the **Export As** list 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** and **Denominator** options under **Variable Names**. For objects, assign the variable name in the **Discrete** or **Quantized 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 the **OK** button

If you try to export the filter to a variable name that exists in your workspace, and you did not select **Overwrite existing variables**, FDATool stops the export operation and 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 dialog, select the **Overwrite existing variables** check box and click **OK** or **Apply**.

Getting Filter Coefficients after Exporting

To extract the filter coefficients from your quantized filter after you export the filter to MATLAB, use the `celldisp` function in MATLAB. For example, create a quantized filter in FDATool and export the filter as `Hq`. To extract the filter coefficients for `Hq`, use

```
celldisp(Hq.referencecoefficients)
```

which returns the cell array containing the filter reference coefficients, or

```
celldisp(Hq.quantizedcoefficients)
```

to return the quantized coefficients.

Exporting 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. Click **Apply** to export the filter without closing the **Export** dialog. Clicking **Apply** lets you export your quantized filter to more than one name without leaving the **Export** dialog.

   The **Export Filter Coefficients to Text-file** dialog appears. This is the standard Microsoft Windows save file dialog.
4. Choose or enter a directory and filename for the text file and click **OK**.

   FDATool exports your quantized filter as a text file with the name you provided, and the MATLAB editor opens, displaying the file for editing.

Exporting 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. Click **Apply** to export the filter without closing the **Export** dialog. Clicking **Apply** lets you export your quantized filter to more than one name without leaving the **Export** dialog.

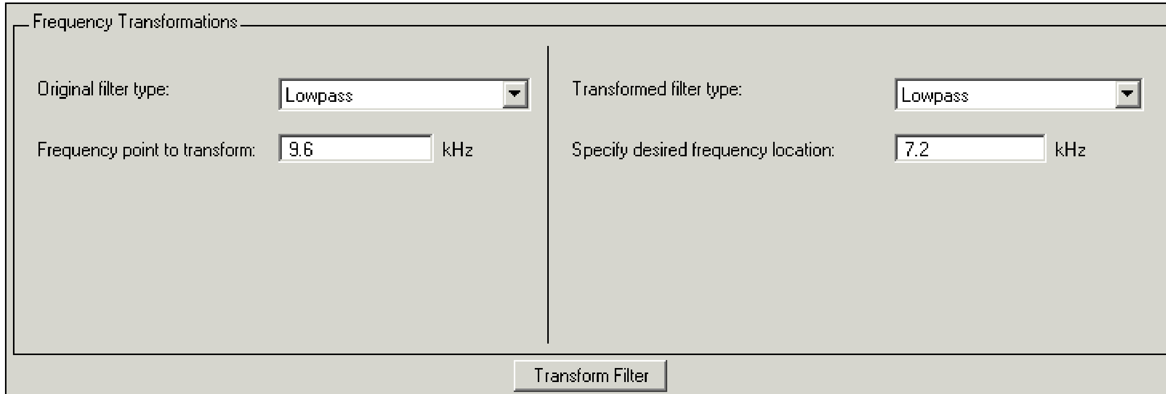
   The **Export Filter Coefficients to MAT-file** dialog appears. This is the standard Microsoft Windows save file dialog.
5. Choose or enter a directory and filename for the text file and click **OK**.

   FDATool exports your quantized filter as a MAT-file with the specified name.

Transforming Filters

The toolbox provides functions for transforming filters between various forms. When you use FDATool with the Toolbox installed, a new side bar button enables you to use the **Transform Filter** panel to transform filters as well as using the command line functions.

When you click the **Transform Filter** button on the side bar, the **Transform Filter** panel opens in FDATool, 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)**

Original Filter	Available Transformed Filter Types
Highpass FIR	• **Lowpass** • **Lowpass (FIR) narrowband** • **Lowpass (FIR) wideband** • **Highpass (FIR)** • **Highpass** • **Bandpass** • **Bandstop** • **Multiband** • **Bandpass (complex)** • **Bandstop (complex)** • **Multiband (complex)**
Highpass IIR	• **Lowpass** • **Highpass** • **Bandpass** • **Bandstop** • **Multiband** • **Bandpass (complex)** • **Bandstop (complex)** • **Multiband (complex)**
Bandpass FIR	• **Bandpass** • **Bandpass (FIR)**
Bandpass IIR	**Bandpass**

Original Filter	Available Transformed Filter Types
Bandstop FIR	• **Bandstop** • **Bandstop (FIR)**
Bandstop IIR	**Bandstop**

Note also that the options change depending on whether you original filter is FIR or IIR. Starting from an IIR filter, you can transform to IIR or FIR forms. With an FIR original filter, you are limited to FIR target filters.

After selecting your response type, use **Frequency point to transform** to specify the magnitude response point in your original filter to transfer to your target filter. Your target filter inherits the performance features of your original filter, such as passband ripple, while changing to the new response form.

For more information about transforming filters, refer to "Frequency Transformations for Real Filters" on page 4-11 and "Frequency Transformations for Complex Filters" on page 4-26.

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 indentified by the frequency point you enter in this field.

While you can enter any location, generally you should specify a filter passband or stopband edge, or a value in the passband or stopband.

The **Frequency point to transform** sets the magnitude response at the values you enter in **Specify desired frequency location**. Specify a value that lies at either the edge of the stopband or the edge of the passband. If, for example, you are creating a bandpass filter from a highpass filter, the transformation algorithm sets the magnitude response of the transformed filter at the **Specify desired frequency location** to be the same as the response at the **Frequency point to transform** value. Thus you get a bandpass filter whose response at the low and high frequency locations is the same. Notice that the passband

between them is undefined. In the next two figures you see the original highpass filter and the transformed bandpass filter.

For more information about transforming filters, refer to “Digital Frequency Transformations” on page 4-1.

Transformed Filter Type

Select the magnitude response for the target filter from the list. The complete list of transformed filter types is:

- **Lowpass**
- **Lowpass (FIR)**
- **Highpass**
- **Highpass (FIR) narrowband**
- **Highpass (FIR) wideband**
- **Bandpass**
- **Bandstop**
- **Multiband**
- **Bandpass (complex)**
- **Bandstop (complex)**
- **Multiband (complex)**

Not all types of transformed filters are available for all filter types on the **Original filter types** list. You can transform bandpass filters only to bandpass filters. Or bandstop filters to bandstop filters. Or IIR filters to IIR filters.

For more information about transforming filters, refer to “Frequency Transformations for Real Filters” on page 4-11 and “Frequency Transformations for Complex Filters” on page 4-26.

Specify Desired Frequency Location

The frequency point you enter in **Frequency point to transform** matched a magnitude response value. At each frequency you enter here, the transformation tries to make the magnitude response the same as the response identified by your **Frequency point to transform** value.

While you can enter any location, generally you should specify a filter passband or stopband edge, or a value in the passband or stopband.

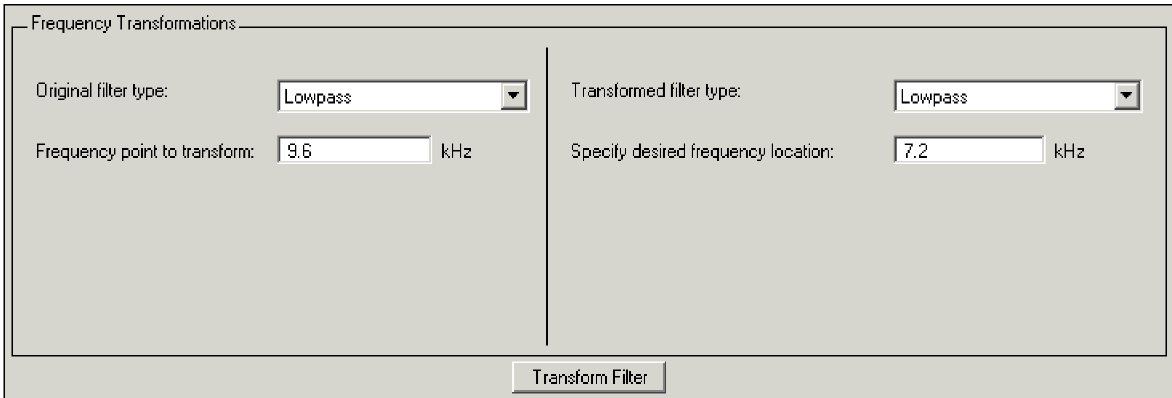
For more information about transforming filters, refer to “Digital Frequency Transformations” on page 4-1.

To 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 FDATool.
2 Click **Transform Filter**, , on the side bar.

   FDATool opens the **Transform Filter** panel in FDATool.



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**, FDATool recognizes whether your filter form is FIR or IIR. Using both your filter type selection and the filter form, FDATool 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.

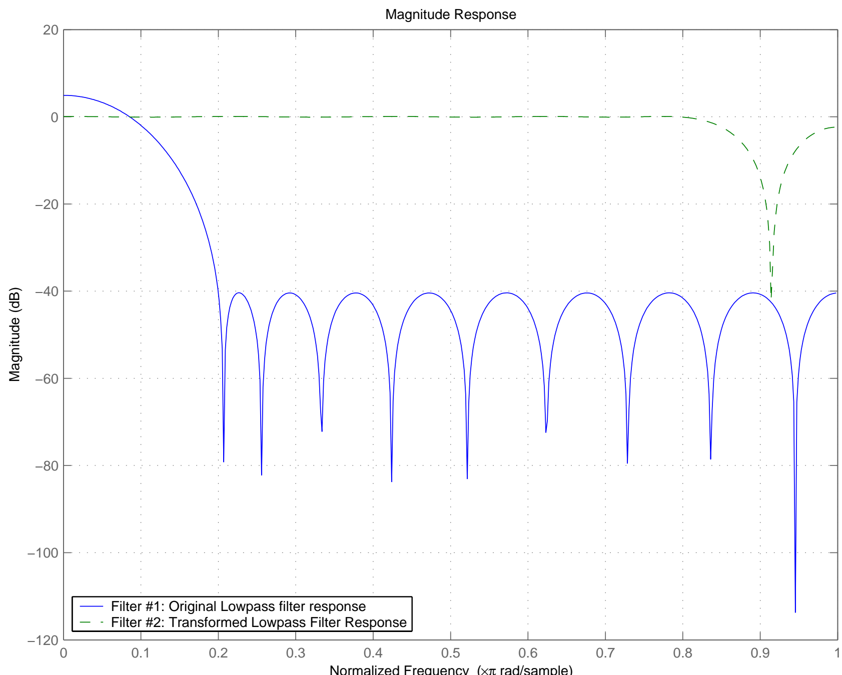
5 From the **Transformed filter type** list, select the type of filter you want to transform to.

   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 or desired frequency locations— one element for each desired location. Your values define the edges of the passbands and stopbands.

     After you click **Transform Filter**, FDATool 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**.

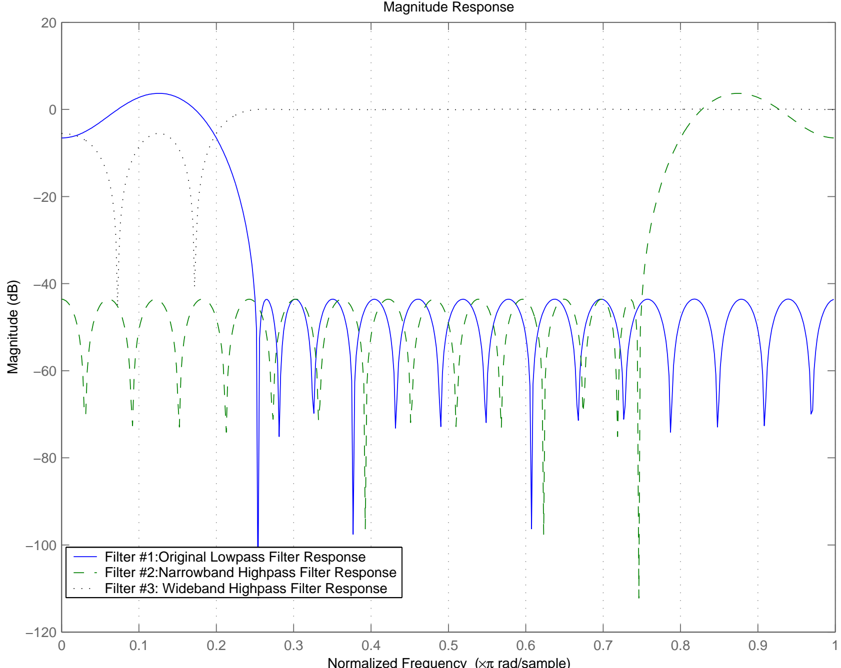
     For example, the figure shown here includes the magnitude response curves for two filter. The original filter is a lowpass filter with rolloff between 0.2 and 0.25. The transformed filter is a lowpass filter with rolloff region between 0.8 and 0.85.



- To transform your lowpass filter to a highpass filter, select **Lowpass to Highpass**.

When you select **Lowpass to Highpass**, FDATool returns the dialog shown here. More information about the **Select Transform...** dialog follows the figure.

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.

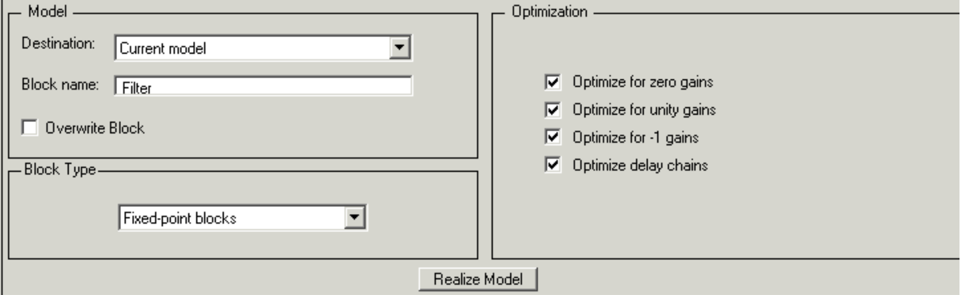
Realizing Filters as Simulink Subsystem Blocks

After you design or import a filter in FDATool, the realize model feature lets you create a Simulink subsystem block that implements your filter. The generated filter subsystem block uses the delay, gain, and sum fixed-point blocks from the Fixed-Point Blockset. If you do not own the Fixed-Point Blockset, FDATool still realizes your model using fixed-point blocks from Simulink, but you cannot run any model that includes your filter subsystem block in Simulink.

About the Realize Model Panel in FDATool

Switching FDATool to realize model mode, by clicking on the sidebar, gives you access to the Realize Model panel and the options for realizing your quantized filter as a Simulink subsystem block.

On the panel, as shown here, are the options provided for configuring how FDATool realizes your model.



Model Options

Under **Model**, you set options that direct FDATool where to put your new subsystem block and what to name the block.

Destination. Tells FDATool whether to put the new block in your current Simulink model or open a new Simulink model and add the block to that window. Select `Current model` to add the block to your current model, or select `New model` to create a new model for the block.

Block name. Provides FDATool with a name to assign to your block. When you realize your filter as a subsystem, the resulting block shows the name you enter here as the block name, positioned just below the block.

Overwrite block. Directs FDATool whether to overwrite an existing block with this block in the destination model. The result is that the new filter realization subsystem block replaces the existing filter subsystem block. Selecting this option replaces your existing filter realization subsystem block with the one you create when you click **Realize Model**. Clearing **Overwrite block** causes FDATool to create a new block in the destination model, rather than replacing the existing block.

Block Type Option

To realize your quantized filter as a subsystem block, the most appropriate choice is to select `Fixed-point blocks` from the list. When you are licensed to use the fixed-point blocks in DSP Blockset, you have the option of realizing your model as either fixed- or floating-point blocks. Since your filter is designed to use quantized coefficients, the fixed-point blocks option usually matches your needs most closely.

You can elect to realize your filter using floating-point blocks, with the understanding that while the coefficients and gains of your filter retain their fixed-point values (the filter uses the fixed-point values for both gain and coefficients, in floating-point format), the math performed during filtering uses floating-point arithmetic and does not truly match the output of your filter running in fixed-point mode. Although realizing your quantized filter with floating-point blocks is not recommended, selecting `Floating-point blocks` from the list creates your filter from blocks in Simulink and the DSP Blockset.

If you do not own a license for the fixed-point blockset, realizing your quantized filter as a subsystem generates a subsystem block that uses fixed-point blocks, but you cannot run or edit the block. If you use the filter subsystem in a Simulink model, you cannot run the model.

Optimization Options

Four options enable you to tailor the way the realized model optimizes various filter features such as delays and gains. When you open the Realize Model panel, these options are selected by default.

Optimize for zero gains. Specify whether to remove zero-gain blocks from the realized filter.

Optimize for unity gains. Specify whether to replace unity-gain blocks with direct connections in the filter subsystem.

Optimize for -1 gains. Specify whether to replace negative unity-gain blocks with a sign change at the nearest sum block in the filter.

Optimize delay chains. Specify whether to replace cascaded chains of delay blocks with a single integer delay block to provide an equivalent delay.

Each of these options can optimize the way your filter performs in simulation and in code you might generate from your model.

To Realize a Filter Using FDATool

After your quantized filter in FDATool 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 FDATool 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 **Overwrite block** to direct FDATool which way to go—overwrite or not.
5. Select `Fixed-point blocks` from the list in **Block Type**.
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 connetions to adjacent blocks
  - **Optimize for -1 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

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 FDATool, 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.

Getting Help for FDATool

To find out more about the buttons or options in the FDATool dialogs, use the **What's This?** button to access context-sensitive help.

Context-Sensitive Help—The What's This? Option

To find information on a particular option or region of the dialog:

1 Click the **What's This?** button .

Your cursor changes to .

2 Click on the region or option of interest.

For example, click **Turn quantization on** to find out what this option does.

You can also select **What's this?** from the **Help** menu to launch context-sensitive help.

Additional Help for FDATool

For help about importing filters into FDATool, or for details about using FDATool to create and analyze double-precision filters, refer to the "Filter Design and Analysis Tool Overview" in your Signal Processing Toolbox documentation.

12

Property Reference

A Quick Guide to Quantizer Properties (p. 12-2)	Provides an overview of the properties of quantizers
Quantizer Properties Reference (p. 12-3)	Gives the details about the quantizer properties
A Quick Guide to Quantized Filter Properties (p. 12-10)	Provides an overview of the properties of quantized filters
Quantized Filter Properties Reference (p. 12-11)	Explains the details about the quantized filter properties
A Quick Guide to Quantized FFT Properties (p. 12-51)	Provides an overview of the properties of quantized FFTs
Quantized FFT Properties Reference (p. 12-52)	Offers details about the quantized FFT properties

A Quick Guide to Quantizer Properties

The following table summarizes the quantizer properties and provides a brief description of each. A table providing a full description of each property follows in the next section.

Table 12-1: Quick Guide to Quantizer Properties

Property	Brief Description of What the Property Specifies
Format	Quantization format
Max	Maximum value encountered when the quantizer quantizes data
Min	Minimum value encountered when the quantizer quantizes data
Mode	Type of quantized arithmetic
NOperations	Number of quantization operations performed by a quantizer
NOverflows	Number of overflows encountered when the quantizer quantizes data
NUnderflows	Number of underflows encountered when the quantizer quantizes data
OverflowMode	Handling of arithmetic overflows
RoundMode	Rounding method used in quantization

Quantizer Properties Reference

To quantize data using `quantize`, you need to specify quantization parameters in a quantizer. When you create a quantizer, you are creating a MATLAB object. You specify the quantization parameters as values assigned to the quantizer properties. With these property values, you specify the quantizer:

- Data format
- Arithmetic method
- Rounding method
- Overflow method

For a quick reference to properties, see Table 12-1, Quick Guide to Quantizer Properties, on page 12-2. Details of all of the properties associated with quantizers are described in the following sections in alphabetical order.

Format

You can set the data format of a quantizer according to its `Format` property value. The interpretation of this property value depends on the value of the `Mode` property.

For example, whether you specify the `Mode` property with fixed- or floating-point arithmetic affects the interpretation of the data format property. For some `Mode` property values, the data format property is read-only.

The following table shows you how to interpret the values for the `Format` property value when you specify it, or how it is specified in read-only cases.

Table 12-2: Interpreting Format Property for Different Arithmetic Types (Mode Property Values)

Filter Arithmetic	Mode Property Value	Interpreting the Format Property Values
Fixed-point	`'fixed'` or `'ufixed'`	You specify the `Format` property value as a vector. The number of bits for the quantizer word length is the first entry of this vector, and the number of bits for the quantizer fraction length is the second entry. The word length can range from 2 to the limits of memory on your PC. The fraction length can range from 0 to one less than the word length.
Floating-point	`'float'`	You specify the `Format` property value as a vector. The number of bits you want for the quantizer word length is the first entry of this vector, and the number of bits you want for the quantizer exponent length is the second entry. The word length can range from 2 to the limits of memory on your PC. The exponent length can range from 0 to 11.
Floating-point	`'double'`	The `Format` property value is specified automatically (is read-only) when you set the Mode property to `'double'`. The value is [64 11], specifying the word length and exponent length, respectively.
Floating-point	`'single'`	The `Format` property value is specified automatically (is read-only) when you set the Mode property to `'single'`. The value is [32 8], specifying the word length and exponent length, respectively.

Default value: `'fixed'`

The `Format` property for quantizers affects the following quantized filter and quantized FFT data format properties:

- The `CoefficientFormat` property
- The `InputFormat` property

- The `MultiplicandFormat` property
- The `OutputFormat` property
- The `ProductFormat` property
- The `SumFormat` property

Set each of these data format properties using a quantizer.

Max

The `Max` property is read-only. The value of the `Max` property is the maximum value data has before a quantizer is applied to it, that is, before quantization using `quantize`. This value accumulates if you use the same quantizer to quantize several data sets. You can reset the value using `reset`.

Default value: `reset`

Min

The `Min` property is read-only. The value of the `Min` property is the minimum value data has before a quantizer is applied to it, that is, before quantization using `quantize`. This value accumulates if you use the same quantizer to quantize several data sets. You can reset the value using `reset`.

Default value: `reset`

Mode

You specify `Mode` property values as one of the following strings to indicate the type of arithmetic used in filtering and quantization.

Mode Property Setting	Description
`'fixed'`	Signed fixed-point calculations
`'float'`	User-specified floating-point calculations
`'double'`	Floating-point calculations using double-precision

Mode Property Setting	Description
'single'	Floating-point calculations using single-precision
'ufixed'	Unsigned fixed-point calculations

Default value: 'fixed'

Remarks: When you set the Mode property value to 'double' or 'single' the Format property value becomes read-only.

The Mode property for quantizers affects the following quantized filter and quantized FFT data format properties:

- The CoefficientFormat property
- The InputFormat property
- The MultiplicandFormat property
- The OutputFormat property
- The ProductFormat property
- The SumFormat property

Set each of these data format properties using a quantizer.

NOperations

The NOperations property is read-only. The value of the NOperations property is the number of quantization operations that occurred during quantization when you use a quantizer, quantized filter, or quantized FFT. This value accumulates when you use the same quantizer, quantized filter, or quantized FFT to process several data sets. You reset the value using reset.

Default value: 0

NOverflows

The NOverflows property is read-only. The value of the NOverflows property is the number of overflows that occur during quantization using quantize. This value accumulates if you use the same quantizer to quantize several data sets. You can reset the value using reset.

Default value: `0`

NUnderflows

The `NUnderflows` property is read-only. The value of the `NUnderflows` property is the number of underflows that occur during quantization using `quantize`. This value accumulates when you use the same quantizer to quantize several data sets. You can reset the value using `reset`.

Default value: `0`

OverflowMode

The `OverflowMode` property values are specified as one of the following two strings indicating how overflows in fixed-point arithmetic are handled:

- `'saturate'` — saturate overflows.

  When the values of data to be quantized lie outside of the range of the largest and smallest representable numbers (as specified by the data format properties), these values are quantized to the value of either the largest or smallest representable value, depending on which is closest.
- `'wrap'` — wrap all overflows to the range of representable values.

  When the values of data to be quantized lie outside of the range of the largest and smallest representable numbers (as specified by the data format properties), these values are wrapped back into that range using modular arithmetic relative to the smallest representable number.

Default value: `'saturate'`

Note Numbers in floating-point filters that extend beyond the dynamic range overflow to `±inf`.

The `OverflowMode` property value is set to `'saturate'` and becomes a read-only property when you set the value of the `Mode` property to either `'float'`, `'double'`, or `'single'`.

The `OverflowMode` property for quantizers affects the following quantized filter and quantized FFT data format properties:

- The `CoefficientFormat` property
- The `InputFormat` property
- The `MultiplicandFormat` property
- The `OutputFormat` property
- The `ProductFormat` property
- The `SumFormat` property

Set each of these data format properties using a quantizer.

RoundMode

The `RoundMode` property values specify the rounding method used for quantizing numerical values. Specify the `RoundMode` property values as one of the following five strings.

RoundMode String	Description of Rounding Algorithm
`'ceil'`	Round up to the next allowable quantized value.
`'convergent'`	Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to 1.
`'fix'`	Round negative numbers up and positive numbers down to the next allowable quantized value.
`'floor'`	Round down to the next allowable quantized value.
`'round'`	Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up.

Default value: `'floor'`

Remarks: The `RoundMode` property for quantizers affects the following quantized filter and quantized FFT data format properties:

- The `CoefficientFormat` property
- The `InputFormat` property
- The `MultiplicandFormat` property
- The `OutputFormat` property
- The `ProductFormat` property
- The `SumFormat` property

Use a quantizer to set each of these data format properties.

A Quick Guide to Quantized Filter Properties

The following table summarizes the quantized filter properties and provides a brief description of each. A table providing a full description of each property follows in the next section.

Table 12-3: Quick Guide to Quantized Filter Properties

Property	Brief Description of What the Property Specifies
CoefficientFormat	Quantization format for filter coefficients
FilterStructure	Filter structure
InputFormat	Quantization format applied to inputs during filtering
NumberOfSections	Number of cascaded sections in the filter
MultiplicandFormat	Quantization format for inputs that are multiplied by coefficients in filtering operations
OutputFormat	Quantization format applied to outputs during filtering
ProductFormat	Quantization format for results of multiplication in filtering
QuantizedCoefficients	Filter coefficients after quantization
`ReferenceCoefficients`	Filter coefficients before quantization
ScaleValues	Scaling for the quantized filter
StatesPerSection	Number of states (delays) in each section of the filter
SumFormat	Quantization format for results of addition in filtering

Quantized Filter Properties Reference

When you create a quantized filter, you are creating a MATLAB object. The quantized filter object you create has many properties to which you assign values. You use these property values to assign the characteristics of the quantized filters you create, including:

- The filter structure
- The double-precision coefficients that specify the original reference filter (before quantization)
- The data formats used in quantization and filtering operations

You specify the `ReferenceCoefficients` property value as a cell array. For more information, see "Using Cell Arrays" on page 4-13.

For a quick reference to properties, see Table 12-1, Quick Guide to Quantizer Properties. Details of all of the properties associated with quantized filters are described in the following sections in alphabetical order.

CoefficientFormat

The `CoefficientFormat` property values specify how filter coefficients are quantized. You specify these values with a quantizer. You set them according to the quantizer property values:

- `Format` (except when the `Mode` property value is set to `'double'` or `'single'`)
- `Mode`
- `OverflowMode`
- `RoundMode`

The value you set for this property is used to calculate the `QuantizedCoefficients` property values.

Default value: `quantizer('fixed','round','saturate',[16,15])`

Note Coefficient overflows that occur due to quantization are not corrected automatically. You can use `normalize` with several filter structures to account for coefficient overflows.

FilterStructure

The `FilterStructure` property values are specified as one of the following strings indicating the quantized filter architecture:

'**Default value:** `'df2t'`

FilterStructure Property Name	Filter Description
`'antisymmetricfir'`	Antisymmetric finite impulse response (FIR). Even and odd forms.
`'df1'`	Direct form I.
`'df1t'`	Direct form I transposed.
`'df2'`	Direct form II.
`'df2t'`	Direct form II transposed. Default filter structure.
`'fir'`	Direct form FIR.
`'firt'`	Direct form FIR transposed.
`'latcallpass'`	Lattice allpass.
`'latticeca'`	Lattice coupled-allpass.
`'latticecapc'`	Lattice coupled-allpass power-complementary.
`'latticear'`	Lattice autoregressive (AR).
`'latticema'`	Lattice moving average (MA) minimum phase.
`'latcmax'`	Lattice moving average (MA) maximum phase.
`'latticearma'`	Lattice ARMA.
`'statespace'`	Single-input/single-output state-space.
`'symmetricfir'`	Symmetric FIR. Even and odd forms.

Remarks: The syntax for entering values for the `ReferenceCoefficients` property is constrained by the `FilterStructure` property value. See Table

12-4: Syntax for Assigning Reference Filter Coefficients (Single Section) on page 12-41, for information on how to enter these coefficients for each filter architecture.

Filter Structure with Quantizers in Place

To help you understand how the quantizers work in filter structures like those provided in the Toolbox, Figure 12-1 presents the structure for a Direct Form 2 filter, including the quantizers that compose the quantized filter. You see that one or more quantizers accompany each filter element, such as a delay, coefficient, or a summation element. The input to or output from each element reflects the result of the associated quantizer. Wherever a particular filter element appears in a structure, recall the quantizers that accompany it as they appear in this figure. For example, a multiplicand quantizer precedes every coefficient element and a product quantizer follows every coefficient element. Or a sum quantizer follows each sum element.

Notice that in this diagram, the first denominator coefficient in your filter, 1/a(1), appears because a(1) is not equal to 1.

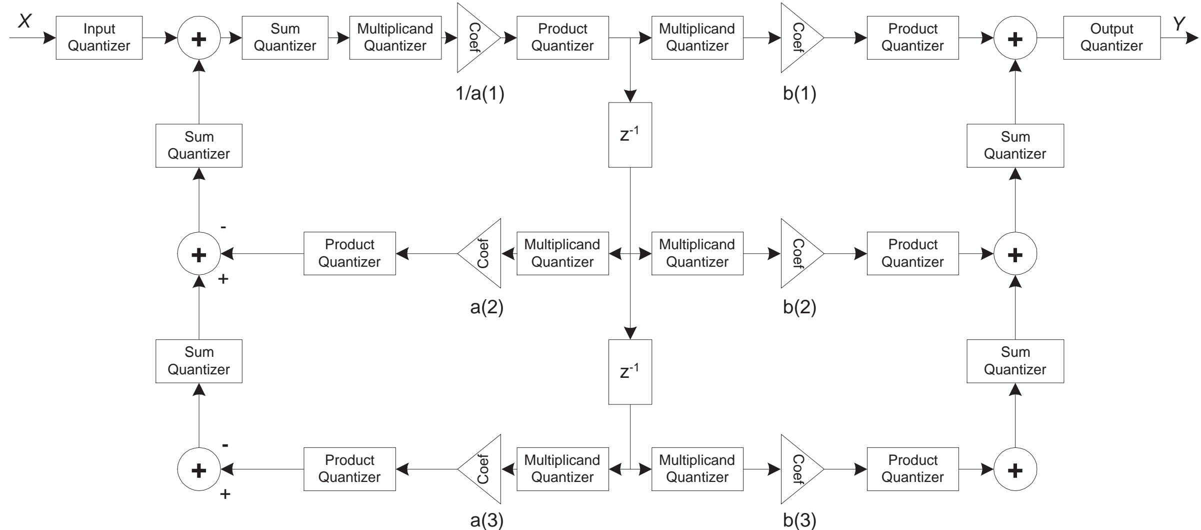


Figure 12-1: df2 Filter Structure Including the Quantizers, with a(1) ≠ 1

When your filter sets a(1) = 1, the df2 structure changes as shown in the next diagram, where the multiplicand and product quantizers for a(1) are not included and are not used when you quantize your filter. Skipping these quantizers removes potential errors that arise when a(1) ends up not quite equal to 1 after quantization, although it should be exactly 1.

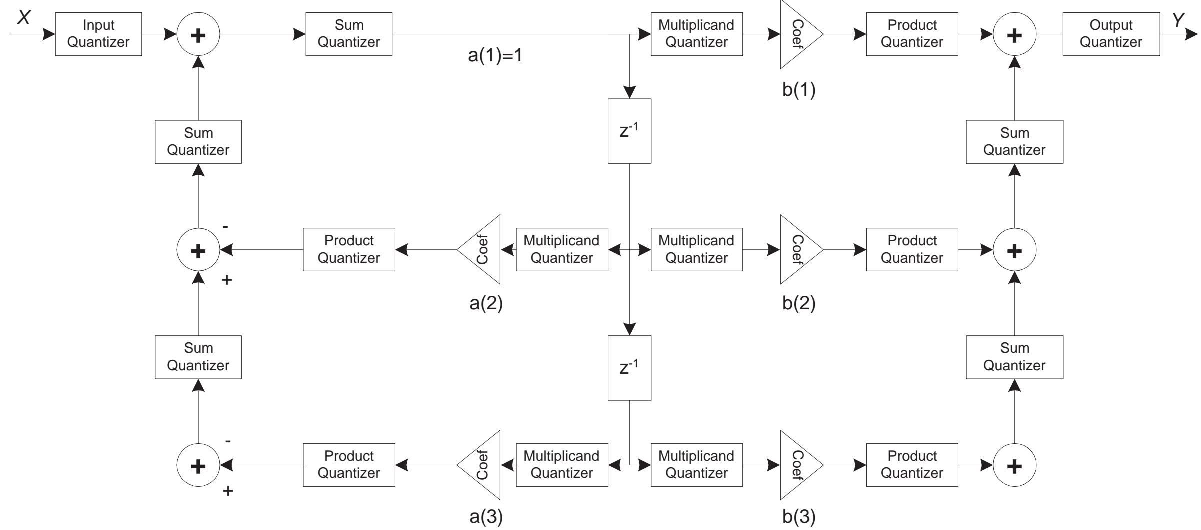


Figure 12-2: df2 Filter Structure Without Input Quantizers, where a(1) = 1

When the leading denominator coefficient a(1) is not 1, choose it to be a power of two so that a shift replaces the multiply that would otherwise be used.

Note The quantized filter structures in the toolbox include the first denominator coefficient a(1) in the feedback loop of direct-form IIR filters (df1, df1t, df2, df2t), although customarily a(1) = 0.

However, when a(1) ≠ 1, the coefficient is needed to ensure accurate quantization analysis. For examples of instances where the leading denominator coefficient is not 1, check references [7] and [10] in the Bibliography.

Quantized Filter Structures

You can choose among several different filter structures when you create a quantized filter. You can also specify filters with single or multiple cascaded sections of the same type. Because quantization is a nonlinear process, different filter structures produce different results.

You specify the filter structure by assigning a specific string to the `FilterStructure` property. Refer to the function reference listings for `qfilt` and `set` for information on setting property values.

The `FilterStructure` property value constrains the syntax you can use for specifying the filter reference coefficients. For details on the syntax to use for specifying a filter with either a single section, or multiple (L) cascaded sections, see Table 12-4, Syntax for Assigning Reference Filter Coefficients (Single Section), and Table 12-5, Syntax for Assigning Reference Filter Coefficients (L Sections).

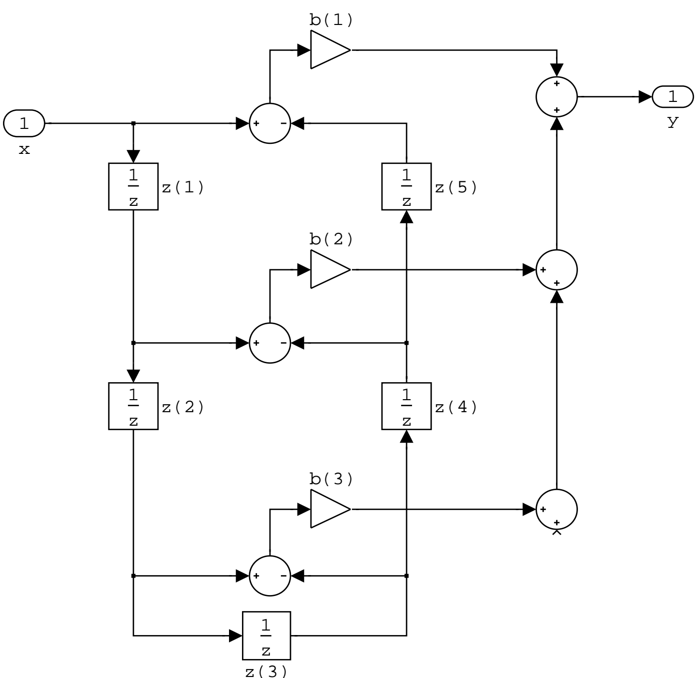
The figures in the following subsections of this section serve as visual aids to help you determine how to enter the reference filter coefficients for each filter structure. Each subsection contains a simple example for constructing a filter of a given structure.

Scale factors for the inputs and output for the filters do not appear in the block diagrams. The default filter structures do not include, nor assume, the scale factors.

Direct Form Antisymmetric FIR Filter Structure (Odd Order)

The following figure depicts a *direct form antisymmetric FIR* filter structure that directly realizes a fifth-order antisymmetric FIR filter. The filter coefficients are labeled $b(i)$, i = 1, ..., 6, and the initial and final state values in filtering are labeled $z(i)$.

antisymmetricfir
(Antisymmetric FIR)
Even number of coefficients, length(b) = 6.
b(i) == – b(end – i + 1)



Use the string `'antisymmetricfir'` for the value of the `FilterStructure` property to design a quantized filter with this structure.

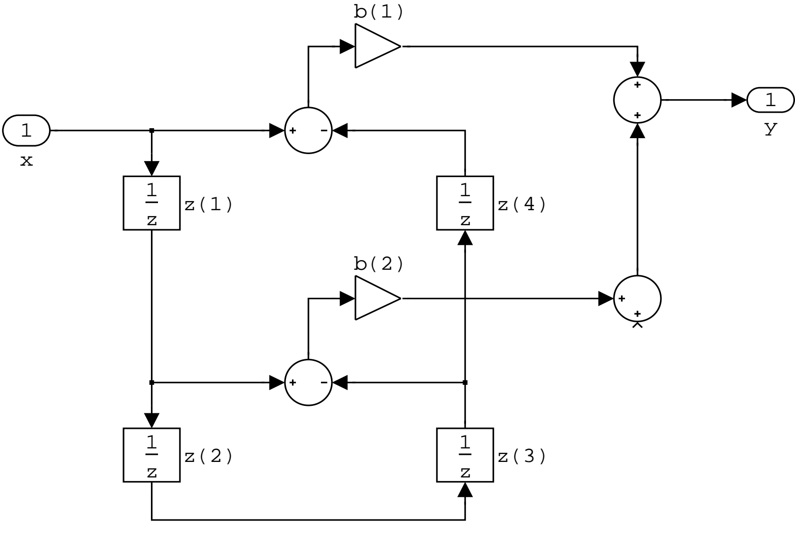
Example — Specifying an Odd-Order Direct Form Antisymmetric FIR Filter Structure. Specify a fifth-order direct form antisymmetric FIR filter structure for a quantized filter `Hq` with the following code.

```
b = [-0.008 0.06 -0.44 0.44 -0.06 0.008];
Hq = qfilt('antisymmetricfir',{b});
```

Antisymmetric FIR Filter Structure (Even Order)

The following figure depicts a *direct form antisymmetric FIR* filter structure that directly realizes a fourth-order antisymmetric FIR filter. The filter coefficients are labeled $b(i)$, $i = 1, ..., 5$, and the states (used for initial and final state values in filtering) are labeled $z(i)$.

antisymmetricfir
(Antisymmetric FIR)
Odd number of coefficients, length(b) = 5.
Note that antisymmetry is defined as
b(i) == –b(end – i + 1)
so that the middle coefficient is zero for odd length
b((end+1)/2) = 0



Use the string `'antisymmetricfir'` to specify the value of the `FilterStructure` property for a quantized filter with this structure.

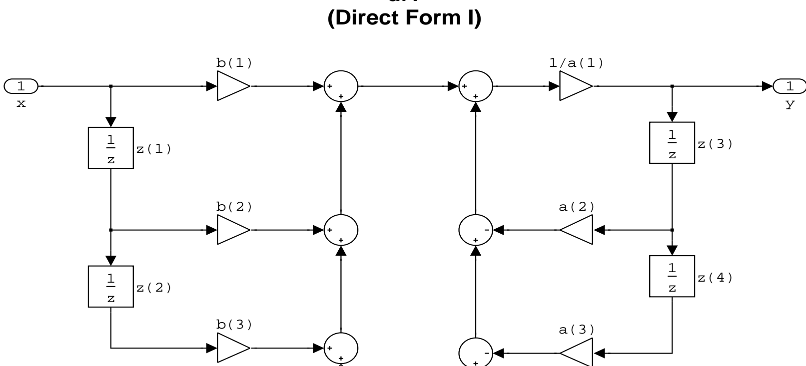
Example — Specifying an Even-Order Direct Form Antisymmetric FIR Filter Structure. You can specify a fourth-order direct form antisymmetric FIR filter structure for a quantized filter `Hq` with the following code.

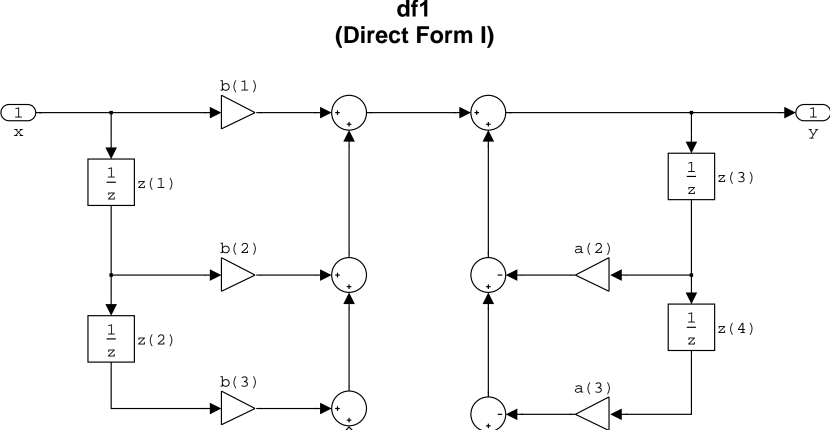
```
b = [-0.01 0.1 0.0 -0.1 0.01];
Hq = qfilt('antisymmetricfir',{b});
```

Direct Form I Filter Structure

The following figures depict *direct form I* filter structures that directly realize a transfer function with a second-order numerator and denominator. The numerator coefficients are labeled $b(i)$, the denominator coefficients are labeled $a(i)$, $i = 1, 2, 3$, and the states (used for initial and final state values in filtering) are labeled $z(i)$. In the first figure, *a(1)* is not equal to one and appears in the structure. When *a(1)* is equal to one, the realized structure does not include the coefficient, as you see in the second figure.

df1
(Direct Form I)





Use the string 'df1' to specify the value of the FilterStructure property for a quantized filter with this structure.

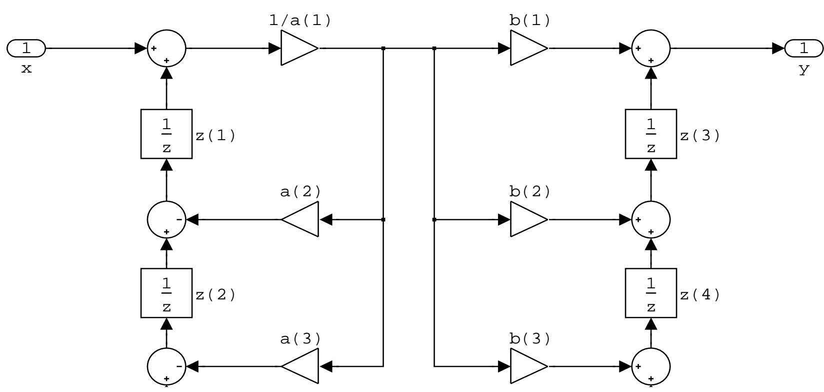
Example — Specifying a Direct Form I Filter Structure. You can specify a second-order direct form I structure for a quantized filter Hq with the following code.

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
Hq = qfilt('df1',{b,a});
```

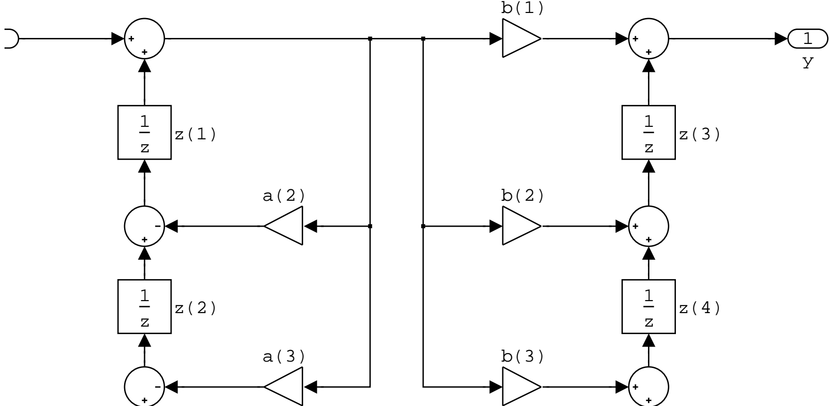
Direct Form I Transposed Filter Structure

The following figures depict *direct form I transposed* filter structures that directly realize a transfer function with a second-order numerator and denominator. The numerator coefficients are labeled $b(i)$, the denominator coefficients are labeled $a(i)$, $i = 1, 2, 3$, and the states (used for initial and final state values in filtering) are labeled $z(i)$. In the first figure, *a(1)* is not equal to one and appears in the structure. When *a(1)* is equal to one, the realized structure does not include the coefficient, as you see in the second figure.

df1t
(Transposed Direct Form I)



df1t
(Transposed Direct Form I)



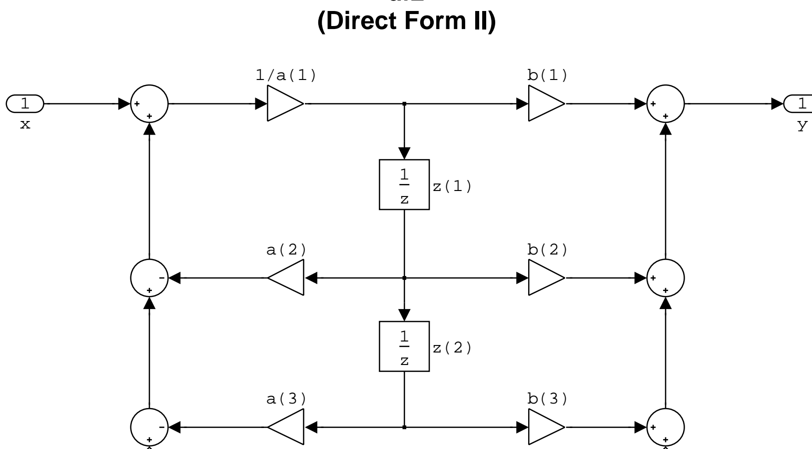
Use the string `'df1t'` to specify the value of the `FilterStructure` property for a quantized filter with this structure.

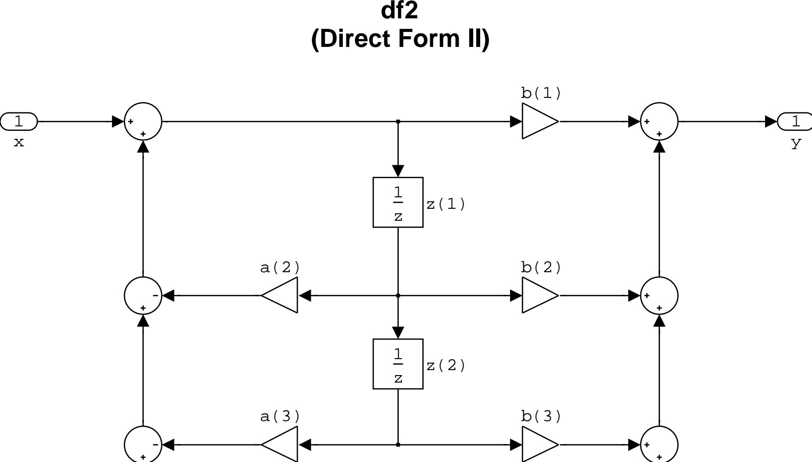
Example — Specifying a Direct Form I Transposed Filter Structure. You can specify a second-order direct form I transposed filter structure for a quantized filter `Hq` with the following code.

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
Hq = qfilt('df1t',{b,a});
```

Direct Form II Filter Structure

The following figures depict *direct form II* filter structures that directly realize a transfer function with a second-order numerator and denominator. The numerator coefficients are labeled $b(i)$, the denominator coefficients are labeled $a(i)$, $i = 1, 2, 3$, and the states (used for initial and final state values in filtering) are labeled $z(i)$. In the first figure, *a(1)* is not equal to one and appears in the structure. When *a(1)* is equal to one, the realized structure does not include the coefficient, as you see in the second figure.





Use the string `'df2'` to specify the value of the `FilterStructure` property for a quantized filter with this structure.

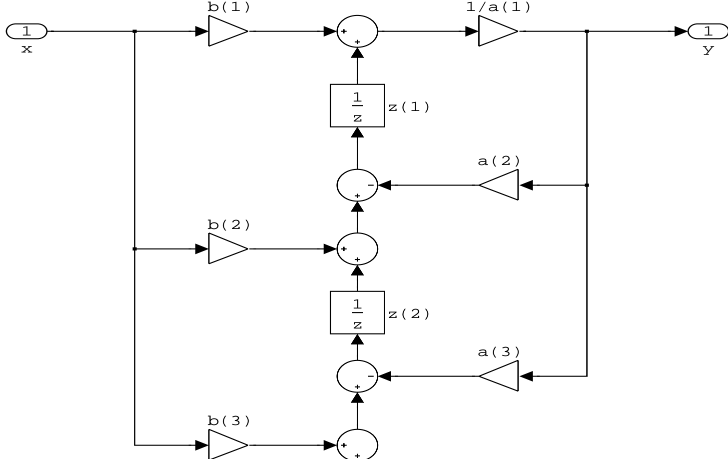
Example — Specifying a Direct Form II Filter Structure. You can specify a second-order direct form II filter structure for a quantized filter `Hq` with the following code.

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
Hq = qfilt('df2',{b,a});
```

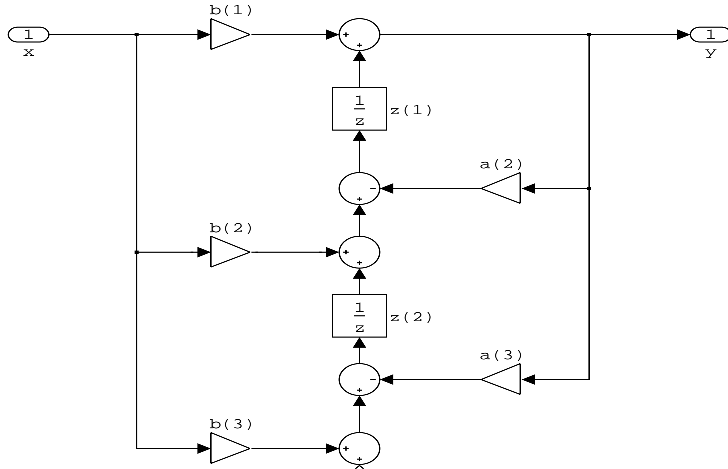
Direct Form II Transposed Filter Structure

The following figures depict *direct form II transposed* filter structures that directly realize a transfer function with a second-order numerator and denominator. The numerator coefficients are labeled $b(i)$, the denominator coefficients are labeled $a(i)$, i = 1, 2, 3, and the states (used for initial and final state values in filtering) are labeled $z(i)$. In the first figure, *a(1)* is not equal to one and appears in the structure. When *a(1)* is equal to one, the realized structure does not include the coefficient, as you see in the second figure.

df2t
(Transposed Direct Form II)



df2t
(Transposed Direct Form II)



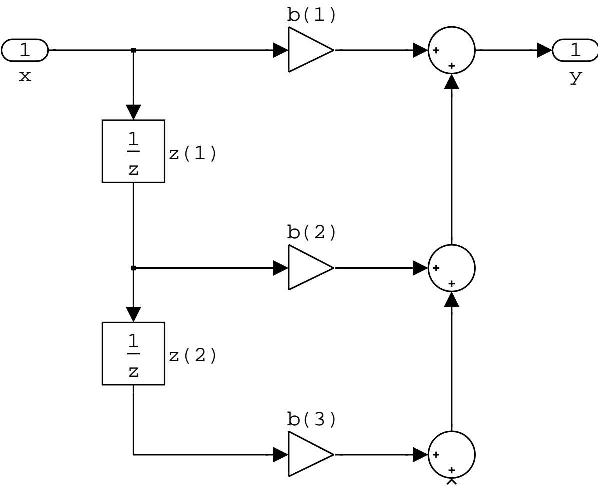
Use the string `'df2t'` to specify the value of the `FilterStructure` property for a quantized filter with this structure.

Example — Specifying a Direct Form II Transposed Filter Structure. You can specify a second-order direct form II transposed filter structure for a quantized filter `Hq` with the following code.

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
Hq = qfilt('df2t',{b,a});
```

Direct Form Finite Impulse Response (FIR) Filter Structure

The following figure depicts a *direct form finite impulse response (FIR)* filter structure that directly realizes a second-order FIR filter. The filter coefficients are labeled $b(i)$, $i = 1, 2, 3$, and the states (used for initial and final state values in filtering) are labeled $z(i)$.



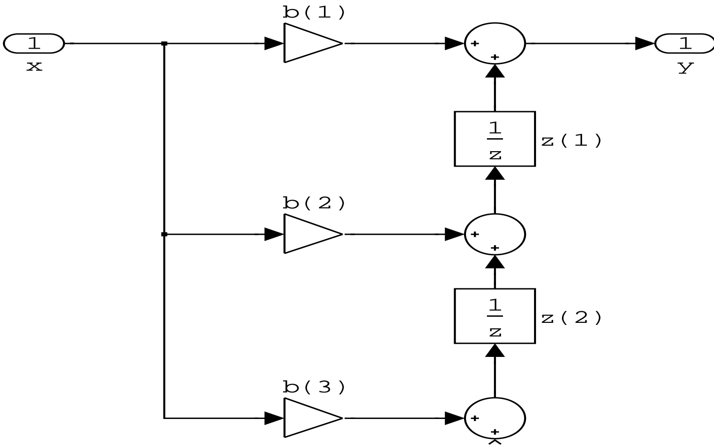
Use the string `'fir'` to specify the value of the `FilterStructure` property for a quantized filter with this structure.

Example — Specifying a Direct Form FIR Filter Structure. You can specify a second-order direct form FIR filter structure for a quantized filter `Hq` with the following code.

```
b = [0.05 0.9 0.05];
Hq = qfilt('fir',{b});
```

Direct Form FIR Transposed Filter Structure

The following figure depicts a *direct form finite impulse response (FIR) transposed* filter structure that directly realizes a second-order FIR filter. The filter coefficients are labeled $b(i)$, $i = 1, 2, 3$, and the states (used for initial and final state values in filtering) are labeled $z(i)$.



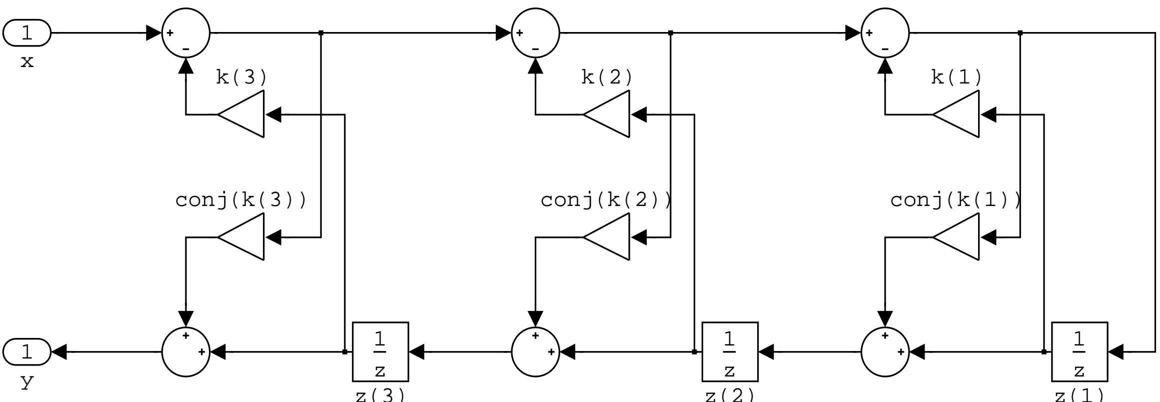
Use the string `'firt'` to specify the value of the `FilterStructure` property for a quantized filter with this structure.

Example — Specifying a Direct Forn FIR Transposed Filter Structure. You can specify a second-order direct form FIR transposed filter structure for a quantized filter `Hq` with the following code.

```
b = [0.05 0.9 0.05];
Hq = qfilt('firt',{b});
```

Lattice Allpass Filter Structure

The following figure depicts a *lattice allpass* filter structure. The pictured structure directly realizes third-order lattice allpass filters. The filter reflection coefficients are labeled $k1(i)$, i = 1, 2, 3. The states (used for initial and final state values in filtering) are labeled $z(i)$.



Use the string `'latcallpass'` to specify the value of the `FilterStructure` property for a quantized filter with this structure.

Example — Specifying a Lattice Allpass Filter Structure. You can specify a third-order lattice allpass filter structure for a quantized filter `Hq` with the following code.

```
k = [.66 .7 .44];
Hq = qfilt('latcallpass',{k});
```

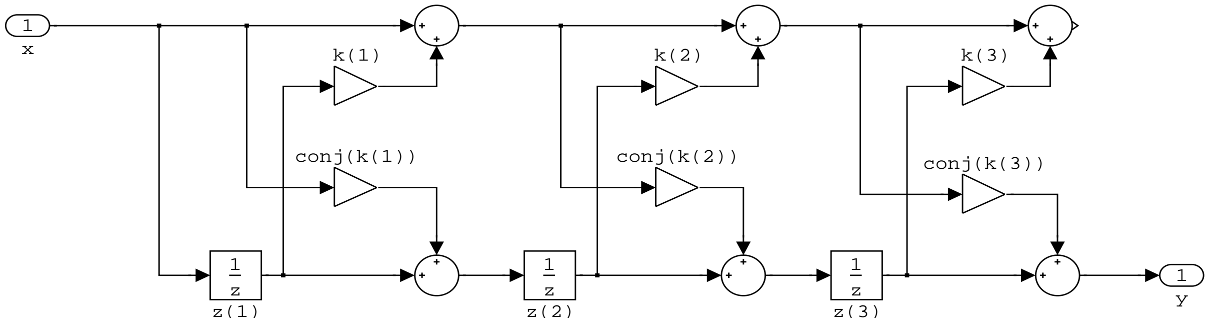
Lattice Moving Average Maximum Phase Filter Structure

The following figure depicts a *lattice moving average maximum phase* filter structure that directly realizes a third-order lattice moving average (MA) filter with the following phase form depending on the initial transfer function:

- When you start with a minimum phase transfer function, the upper branch of the resulting lattice structure returns a minimum phase filter. The lower branch returns a maximum phase filter.
- When your transfer function is neither minimum phase nor maximum phase, the lattice moving average maximum phase structure will not be maximum phase.
- When you start with a maximum phase filter, the resulting lattice filter is maximum phase also.

The filter reflection coefficients are labeled $k(i)$, $i = 1, 2, 3$. The states (used for initial and final state values in filtering) are labeled $z(i)$.

latcmax
(Lattice MA Max phase)



Use the string `'latcmax'` to specify the value of the `FilterStructure` property for a quantized filter with this structure.

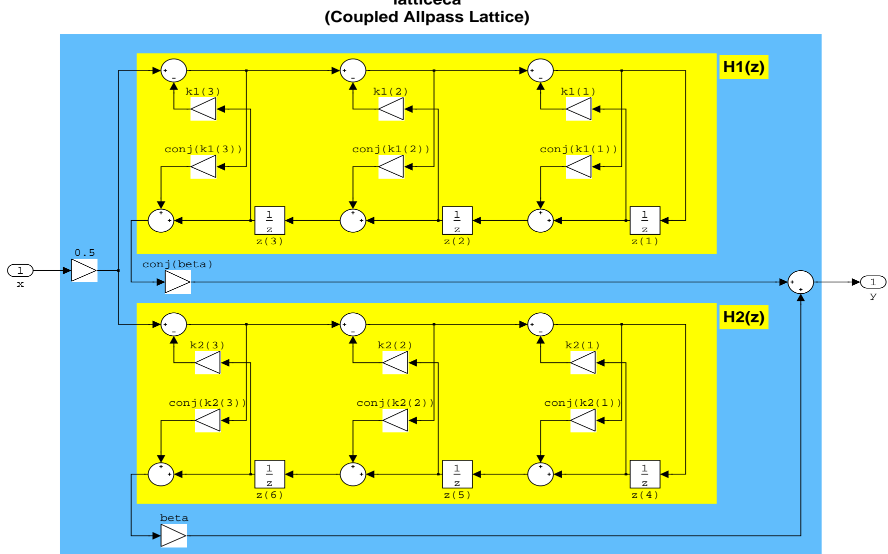
Example—Specifying a Lattice Moving Average Maximum Phase Filter Structure. You can specify a fourth-order lattice MA maximum phase filter structure for a quantized filter `Hq` with the following code.

```
k = [.66 .7 .44 .33];
```

```
Hq = qfilt('latcmax',{k});
```

Lattice Coupled-Allpass Filter Structure

The following figure depicts a *lattice coupled-allpass* filter structure. The filter is composed of two third-order allpass lattice filters. The filter reflection coefficients for the first filter are labeled *k1*(i), i = 1, 2, 3. The filter reflection coefficients for the second filter are labeled *k2*(i), i = 1, 2, 3. The unity gain complex coupling coefficient is *beta*. The states (used for initial and final state values in filtering) are labeled $z(i)$.



Use the string `'latticeca'` to specify the value of the `FilterStructure` property for a quantized filter with this structure.

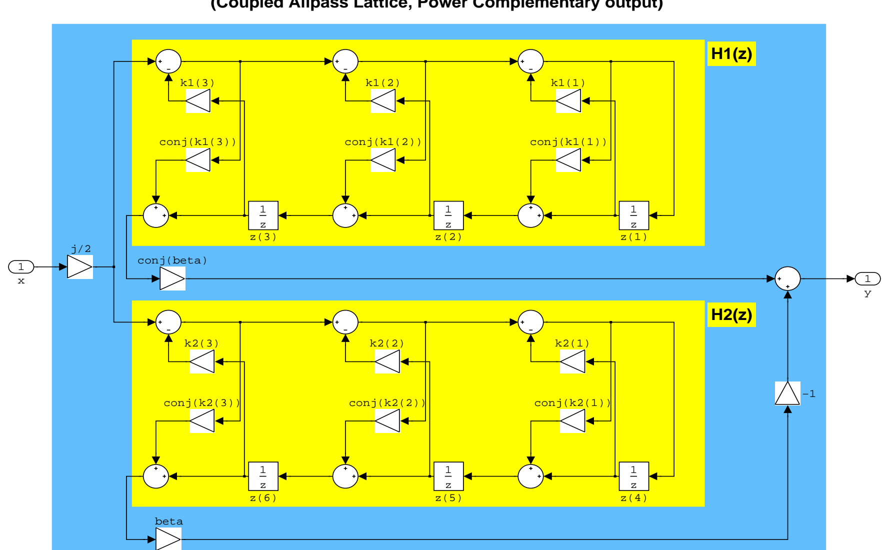
Example — Specifying a Lattice Coupled-Allpass Filter Structure. You can specify a third-order lattice coupled allpass filter structure for a quantized filter `Hq` with the following code.

```
k1 = [0.9511 + 0.3088i; 0.7511 + 0.1158i]
k2 = 0.7502 - 0.1218i
beta = 0.1385 + 0.9904i
Hq = qfilt('latticeca',{k1,k2,beta});
```

Lattice Coupled-Allpass Power Complementary Filter Structure

The following figure depicts a *lattice coupled-allpass power complementary* filter structure. The filter is composed of two third-order allpass lattice filters. The filter reflection coefficients for the first filter are labeled $k1(i)$, $i = 1, 2, 3$. The filter reflection coefficients for the second filter are labeled $k2(i)$, $i = 1, 2, 3$. The unity gain complex coupling coefficient is *beta*. The states used for initial and final state values in filtering are labeled $z(i)$. The resulting filter transfer function is the power-complementary transfer function of the coupled allpass lattice filter (formed from the same coefficients).

latticecapc
(Coupled Allpass Lattice, Power Complementary output)



Use the string `'latticecapc'` to specify the value of the `FilterStructure` property for a quantized filter with this structure.

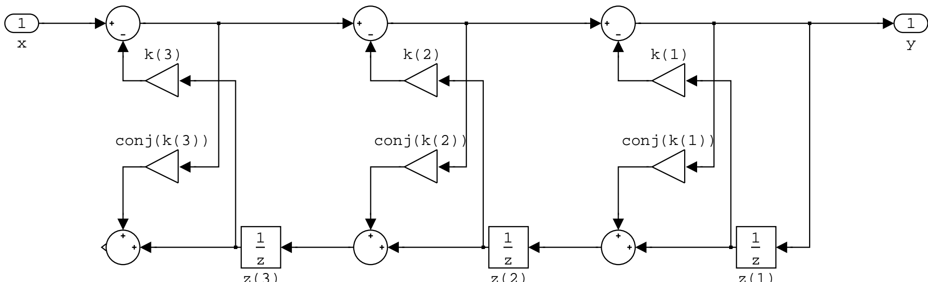
Example — Specifying a Lattice Coupled-Allpass Power Complementary Filter Structure.

Specify a third-order lattice coupled-allpass power complementary filter structure for a quantized filter `Hq` with the following code.

```
k1 = [0.9511 + 0.3088i; 0.7511 + 0.1158i]
k2 = 0.7502 - 0.1218i
beta = 0.1385 + 0.9904i
Hq = qfilt('latticecapc',{k1,k2,beta});
```

Lattice Autoregressive (AR) Filter Structure

The following figure depicts a *lattice autoregressive (AR)* filter structure that directly realizes a third-order lattice AR filter. The filter reflection coefficients are labeled $k(i)$, $i = 1, 2, 3$, and the states (used for initial and final state values in filtering) are labeled $z(i)$.



Use the string `'latticear'` to specify the value of the `FilterStructure` property for a quantized filter with this structure.

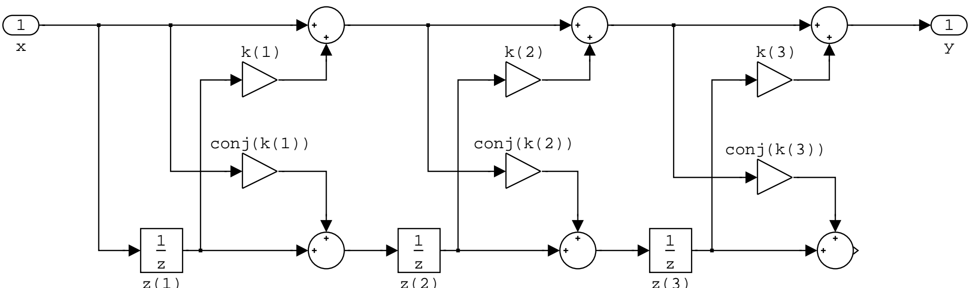
Example — Specifying an Lattice AR Filter Structure. You can specify a third-order lattice AR filter structure for a quantized filter `Hq` with the following code.

```
k = [.66 .7 .44];
Hq = qfilt('latticear',{k});
```

Lattice Moving Average (MA) Filter Structure

The following figure depicts a *lattice moving average (MA)* filter structure that directly realizes a third-order lattice MA filter. The filter reflection coefficients are labeled $k(i)$, $i = 1, 2, 3$, and the states (used for initial and final state values in filtering) are labeled $z(i)$.

latticema
(Moving Average Lattice)



Use the string `'latticema'` to specify the value of the `FilterStructure` property for a quantized filter with this structure.

Example — Specifying an Lattice MA Filter Structure. You can specify a third-order lattice MA filter structure for a quantized filter `Hq` with the following code.

```
k = [.66 .7 .44];
Hq = qfilt('latticema',{k});
```

Lattice Autoregressive Moving Average (ARMA) Filter Structure

The following figure depicts a *lattice autoregressive moving average (ARMA)* filter structure that directly realizes a fourth-order lattice ARMA filter. The filter reflection coefficients are labeled $k(i)$, $i = 1, ..., 4$, the ladder coefficients are labeled $v(i)$, $i = 1, 2, 3$, and the states (used for initial and final state values in filtering) are labeled $z(i)$.

Use the string `'latticearma'` to specify the value of the `FilterStructure` property for a quantized filter with this structure.

Example — Specifying an Lattice ARMA Filter Structure. You can specify a fourth-order lattice ARMA filter structure for a quantized filter `Hq` with the following code.

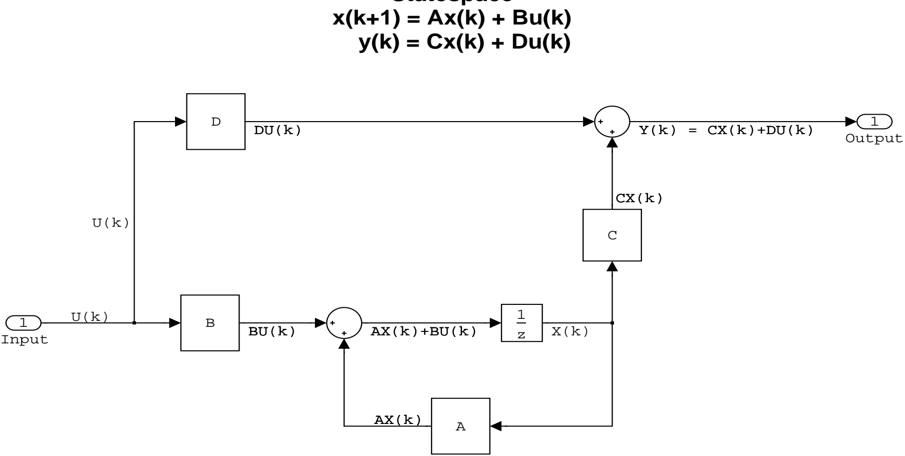
```
k = [.66 .7 .44 .66];
v = [1 0 0];
Hq = qfilt('latticearma',{k,v});
```

State-Space Filter Structure

State-space models with input sequence x_k and output sequence y_k have the following form.

$$z_{k+1} = Az_k + Bx_k$$
$$y_k = Cz_k + Dx_k$$

If the states z_k are vectors of length n, then the matrices A, B, C, and D are n-by-n, n-by-1, 1-by-n, and 1-by-1 respectively.



Use the string `'statespace'` to specify the value of the `FilterStructure` property for a quantized filter with this structure.

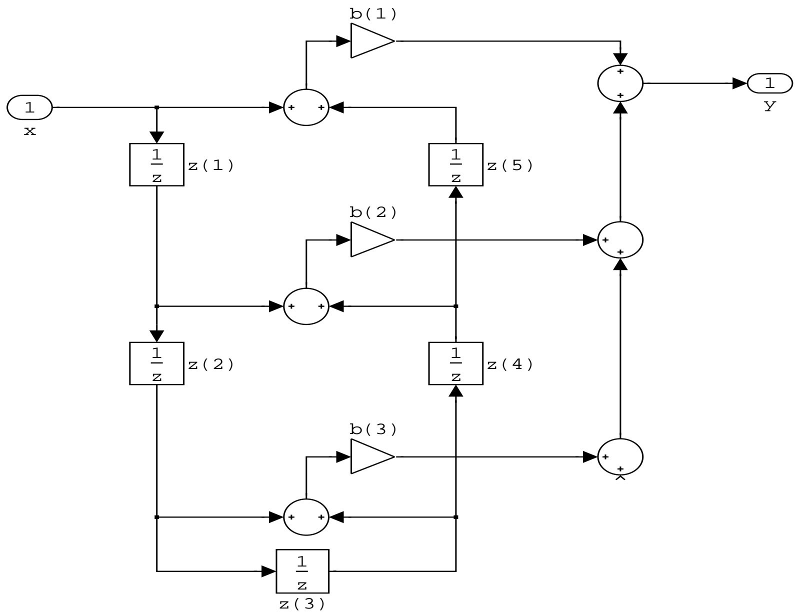
Example — Specifying a State-Space Filter Structure. You can specify a second-order state-space filter structure for a quantized filter `Hq` with the following code.

```
[A,B,C,D] = butter(2,0.5);
Hq = qfilt('statespace',{A,B,C,D});
```

Direct Form Symmetric FIR Filter Structure (Odd Order)

The following figure depicts a *direct form symmetric FIR* filter structure that directly realizes a fifth-order direct form symmetric FIR filter. The filter coefficients are labeled $b(i)$, $i = 1, ..., 6$, and the states (used for initial and final state values in filtering) are labeled $z(i)$.

symmetricfir
(Symmetric FIR)
Even number of coefficients, length(b) = 6.
b(i) == b(end – i + 1)



Use the string `'symmetricfir'` to specify the value of the `FilterStructure` property for a quantized filter with this structure.

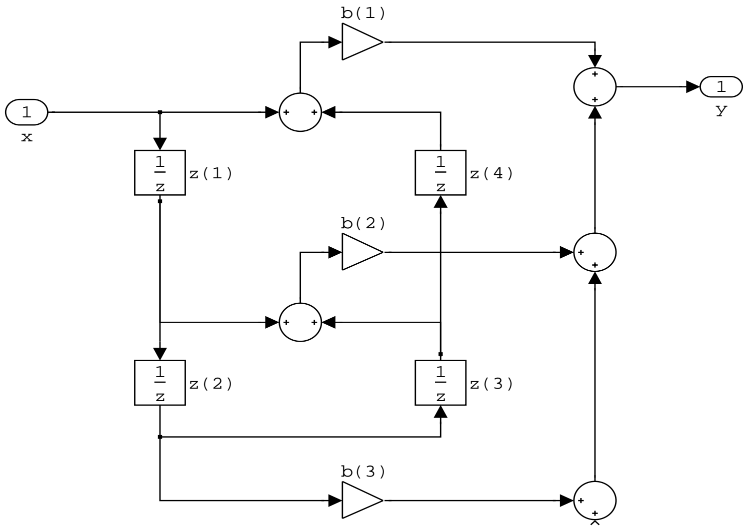
Example — Specifying an Odd-Order Direct Form Symmetric FIR Filter Structure. You can specify a fifth-order direct form symmetric FIR filter structure for a quantized filter `Hq` with the following code.

```
b = [-0.008 0.06 0.44 0.44 0.06 -0.008];
Hq = qfilt('symmetricfir',{b});
```

Direct Form Symmetric FIR Filter Structure (Even Order)

The following figure depicts a *direct form symmetric FIR* filter structure that directly realizes a fourth-order direct form symmetric FIR filter. The filter coefficients are labeled $b(i)$, $i = 1, ..., 5$, and the states (used for initial and final state values in filtering) are labeled $z(i)$.

symmetricfir
(Symmetric FIR)
Odd number of coefficients, length(b) = 5.
b(i) == b(end – i + 1)



Use the string `'symmetricfir'` to specify the value of the `FilterStructure` property for a quantized filter with this structure.

Example — Specifying an Even-Order Direct Form Symmetric FIR Filter Structure. You can specify a fourth-order direct form symmetric FIR filter structure for a quantized filter `Hq` with the following code.

```
b = [-0.01 0.1 0.8 0.1 -0.01];
Hq = qfilt('symmetricfir',{b});
```

InputFormat

The `InputFormat` property values specify how inputs are quantized during the filtering operation. You specify these values with a quantizer. You set them according to the property values of a quantizer's:

- `Format` (except when the `Mode` property value is set to `'double'` or `'single'`)
- `Mode`
- `OverflowMode`
- `RoundMode`

Default value: `quantizer('fixed','floor','saturate',[16,15])`

NumberOfSections

The value of this read-only property is a scalar that specifies the number of cascaded sections in your quantized filter. You specify the number of sections for your filter by the way you specify the `ReferenceCoefficients` property value.

Default value: `1`

MultiplicandFormat

Products in quantized filters always involve two types of multiplicands:

- Inputs (data)
- Coefficients

`MultiplicandFormat` property values specify how inputs that are multiplied by coefficients are quantized during the filtering operation.

`multiplicandFormat` property values specify how to quantize the filter multiplicands. Multiplicands are the inputs to multiply operations.

Not all inputs to multiplications are directly from the input to the filter. Sometimes, as in the direct form I filter, the input to one multiplication may be the output of another multiplication or addition. The output of a multiplication (a product) and the output of an addition (a sum) is usually double the wordlength of their inputs. For example, multiplying a 16-bit multiplicand by a 16-bit multiplier (here a coefficient in a filter) yields a 32-bit product. Also, sums are usually kept in double-wordlength accumulators. If any one of these

double-wordlength numbers is fed back into another multiplication as a multiplicand, they need to be quantized back into a single-wordlength (16-bit) number before using them in another calculation. Not quantizing the result back to single word length causes (16-bit * 16-bit) = 32-bit result, then (32-bit result * 16-bit) = 48-bit result, and so on. Hence the multiplicand quantizer prevents the result from growing beyond 32 bits.

When multiplicand quantizers are not necessary, as in direct-form FIR filters, `multiplicandFormat` should be set to be the same as the `inputFormat` because the inputs to the multiplications are exactly the input to the filter.

Although multiplicand quantizers are not always necessary for a given filter structure, filter structures in FD Toolbox have them available to provide full generality in the specification of the arithmetic of any filter.

You specify these values with a quantizer. You set them according to the property values of a quantizer's:

- `Format` (except when the `Mode` property value is set to `'double'` or `'single'`)
- `Mode`
- `OverflowMode`
- `RoundMode`

Default value: `quantizer('fixed','floor','saturate',[16,15])`

OutputFormat

The `OutputFormat` property values specify how outputs are quantized during the filtering operation. You specify these values with a quantizer. You set them according to the property values of a quantizer's:

- `Format` (except when the `Mode` property value is set to `'double'` or `'single'`)
- `Mode`
- `OverflowMode`
- `RoundMode`

Default value: `quantizer('fixed','floor','saturate',[16,15])`

ProductFormat

The `ProductFormat` property values specify how the results of multiplication are quantized during the filtering operation. You specify these values with a quantizer. You set them according to the property values of a quantizer's:

- `Format` (except when the `Mode` property value is set to `'double'` or `'single'`)
- `Mode`
- `OverflowMode`
- `RoundMode`

Default value: `quantizer('fixed','floor','saturate',[32,30])`

QuantizedCoefficients

The values for this read-only property are stored in a cell array containing the quantized filter coefficients calculated from the value of the `ReferenceCoefficients` property. The quantization is specified by the value of the `CoefficientFormat` property.

Default value: `{1 1}`

Remarks: If any filter coefficient overflows occur as a result of quantization, a warning is displayed.

The cell array for the `QuantizedCoefficients` property value has the same form as that of the corresponding `ReferenceCoefficients` property value, described in "Assigning Reference Filter Coefficients" on page 12-40.

ReferenceCoefficients

The `ReferenceCoefficients` property values are specified as a cell array that specifies the original (unquantized) reference filter coefficients. You specify these in double-precision, using a syntax specific to the value of the `FilterStructure` property.

Default value: `{1 1}`

Assigning Reference Filter Coefficients

To assign the coefficients that specify the filter that serves as the reference for your quantized filter, specify the value of the `ReferenceCoefficients` property. The syntax you use to assign reference filter coefficients for your

quantized filter depends on the value you assign to the `FilterStructure` property. These syntaxes are described in the following two tables. The first table explains the syntax for the `ReferenceCoefficients` property value when you want to specify one section for your filter. The next table explains how to specify the coefficients for filters with *L* cascaded sections.

Table 12-4: Syntax for Assigning Reference Filter Coefficients (Single Section)

FilterStructure Property Value	Syntax for ReferenceCoefficients Property Value
'antisymmetricfir'	`{b}`: • This is a cell array containing one vector. • `b(i) = -b(n-i+1); i = 1, ..., n` • `n-1` is the order of the polynomial represented by `b`. • When `n` is odd, the center coefficient, `b((n+1)/2)` should be 0. If you don't supply an antisymmetric vector `b`, it is converted to be antisymmetric automatically.
`'df1'`	`{b,a}`: • This is a cell array of vectors. • `b` is the vector representing the coefficients of the transfer function numerator polynomial. • `a` is the vector representing the coefficients of the transfer function denominator polynomial.
`'df1t'`	`{b,a}`: This is a cell array of vectors.
`'df2'`	`{b,a}`: This is a cell array of vectors.
`'df2t'`	`{b,a}`: This is a cell array of vectors.
`'fir'`	`{b}`: This is a cell array containing one vector.

Table 12-4: Syntax for Assigning Reference Filter Coefficients (Single Section) (Continued)

FilterStructure Property Value	Syntax for ReferenceCoefficients Property Value
`'firt'`	`{b}`: This is a cell array containing one vector.
`'latticeca'`	`{k1,k2,beta}`: • This is a cell array of vectors. • `k1` and `k2` are the vectors of reflection coefficients for the two lattice allpass filters in the coupled allpass structure. • `beta` is the unity gain complex scalar coupling coefficient.
`'latticecapc'`	`{k1,k2,beta}`: • This is a cell array of vectors. • `k1` and `k2` are the vectors of reflection coefficients for the two lattice allpass filters in the coupled allpass structure. • `beta` is the unity gain complex scalar coupling coefficient.
`'latticear'`	`{k}`: • This is a cell array containing one vector. • `k` is the vector of reflection coefficients for an all-pole (AR) lattice filter.
`'latticema'`	`{k}`: • This is a cell array containing one vector. • `k` is the vector of reflection coefficients for an FIR (MA) lattice filter.

Table 12-4: Syntax for Assigning Reference Filter Coefficients (Single Section) (Continued)

FilterStructure Property Value	Syntax for ReferenceCoefficients Property Value
`'latticearma'`	`{k,v}`: • This is a cell array of vectors. • `k` is the vector of reflection coefficients for an IIR (ARMA) lattice filter. • `v` is the vector of ladder coefficients for an IIR lattice filter.
`'statespace'`	`{A,B,C,D}`: • This is a cell array of matrices. • `A` is the *n*-by-*n* state transition matrix (*n* states). • `B` is the *n*-by-1 input to state transmission vector. • `C` is the 1-by-*n* state to output transmission vector. • `D` is the input to output transmission scalar.
`'symmetricfir'`	`{b}`: • This is a cell array containing one vector. • `b(i) = b(n-i+1); i = 1, ..., n` • `n-1` is the order of the polynomial represented by `b`. • If you don't supply a symmetric vector b, it is converted to be symmetric automatically.

You can specify quantized filters with multiple sections for all of the filter structures.

The following table describes the syntax for entering reference coefficients to specify a quantized filter with L second-order or arbitrary-order sections.

Table 12-5: Syntax for Assigning Reference Filter Coefficients (L Sections)

Section Structure	Syntax for ReferenceCoefficients Property Value
L second-order sections	`{ {b1 a1} {b2 a2} ... {bL aL} }` • This is a 1-by-L cell array of 1-by-2 cell arrays. • `bi` is a 1-by-3 row vector for the numerator of the ith section, i=1, ... , L. • `ai` is a 1-by-3 row vector for the denominator of the ith section, i=1, ... , L. You can use `tf2sos` and `sos2cell` to covert a transfer function directly into this format (a cell array of cells). You can also use `sos` to convert quantized filters with other topologies directly to a second-order sections form.
L sections, each of arbitrary order (except FIR filters)	`{ {b1 a1} {b2 a2} ... {bL aL} }` • This is a 1-by-L cell array of 1-by-2 cell arrays. • `bi` is a 1-by-nb_i row vector for the numerator of the ith section, i=1, ... , L. • `ai` is a 1-by-na_i row vector for the denominator of the ith section, i=1, ... , L. • nb_i is the order of the numerator of the ith section. • na_i is the order of the denominator of the ith section.
L sections, each of arbitrary order (only FIR)	`{ {b1} {b2} ... {bL} }` • This is a 1-by-L cell array of one-dimensional cell arrays. • `bi` is a 1-by-nb_i row vector for the numerator of the ith FIR section, i=1, ... , L. • nb_i is the order of the ith FIR section.

Table 12-5: Syntax for Assigning Reference Filter Coefficients (L Sections) (Continued)

Section Structure	Syntax for ReferenceCoefficients Property Value
L sections of coupled allpass lattice filters	`{{k11,k21,beta1},...,{k1L,k2L,betaL)}`: • This is a 1-by-L cell array of 1-by-3 cell arrays. • `k1i` and `k2i` are the vectors of reflection coefficients for the two lattice allpass filters in the ith coupled allpass structure, i=1, ... , L. • `betai` is the unity gain complex scalar coupling coefficient in the ith coupled allpass structure, i=1, ... , L.
L sections of lattice ARMA filters	`{{k1,v1},...,{kL,vL}}`: • This is a 1-by-L cell array of 1-by-2 cell arrays. • `ki` is the vector of reflection coefficients for the ith IIR lattice (ARMA) filter in the cascade, i=1, ... , L. • `vi` is the vector of ladder coefficients for ith IIR lattice (ARMA) filter in the cascade, i=1, ... , L.

Table 12-5: Syntax for Assigning Reference Filter Coefficients (L Sections) (Continued)

Section Structure	Syntax for ReferenceCoefficients Property Value
L sections of lattice AR or MA filters	`{{k1},...,{kL}}`: • This is a 1-by-*L* cell array of one-dimensional cell arrays. • `ki` is the vector of reflection coefficients for the *i*th lattice AR or MA filter in the cascade, *i*=1, ... , *L*.
L sections of state-space filters	`{{A1,B1,C1,D1},...,{AL,BL,CL,DL}}`: • This is a cell array of 1-by4 cell arrays. • `Ai` is the *ni*-by-*ni* state transition matrix (*ni* states) of the *i*th state-space filter in the cascade, *i*=1, ... , *L*. • `Bi` is the *ni*-by-1 input to state transmission vector of the *i*th state-space filter in the cascade, *i*=1, ... , *L*. • `Ci` is the 1-by-*ni* state to output transmission vector of the *i*th state-space filter in the cascade, *i*=1, ... , *L*. • `Di` is the input to output transmission scalar of the *i*th state-space filter in the cascade, *i*=1, ... , *L*.

Conversion functions in this toolbox and in Signal Processing Toolbox let you convert transfer functions to other filter forms and from filter forms to transfer functions. Relevant conversion functions include the following functions.

Conversion Function	Description
`ca2tf`	Converts from a coupled allpass filter to a transfer function.
`cl2tf`	Converts from a lattice coupled allpass filter to a transfer function.

Conversion Function (Continued)	Description
`sos`	Converts quantized filters to create second-order sections. This is the recommended method for converting quantized filters to second-order sections.
`tf2ca`	Converts from a transfer function to a coupled allpass filter.
`tf2cl`	Converts from a transfer function to a lattice coupled allpass filter.
`tf2latc`	Converts from a transfer function to a lattice filter.
`tf2sos`	Converts from a transfer function to a second-order section form.
`tf2ss`	Converts from a transfer function to state-space form.
`tf2zp`	Converts from a rational transfer function to its factored (single section) form (zero-pole-gain form).
`zp2sos`	Converts a zero-pole-gain form to a second-order section form.
`zp2ss`	Conversion of zero-pole-gain form to a state-space form.
`zp2tf`	Conversion of zero-pole-gain form to transfer functions of multiple order sections.

You can specify a filter with L sections of arbitrary order by:

1. Factoring your entire transfer function with `tf2zp`.
2. Using `zp2tf` to compose the transfer function for each section from the selected first-order factors obtained in step 1.

Note You are not required to normalize the leading coefficients of each section's denominator polynomial when specifying second-order sections, though `tf2sos` does.

ScaleValues

The `ScaleValues` property values are specified as a scalar (or vector) that introduces scaling for inputs (and the outputs from cascaded sections in the vector case) during filtering:

- When you only have a single section in your filter:
  - Specify the `ScaleValues` property value as a scalar if you only want to scale the input to your filter.
  - Specify the `ScaleValues` property as a vector of length 2 if you want to specify scaling to the input (scaled with the first entry in the vector) and the output (scaled with the last entry in the vector).
- When you have L cascaded sections in your filter:
  - Specify the `ScaleValues` property value as a scalar if you only want to scale the input to your filter.
  - Specify the value for the `ScaleValues` property as a vector of length L+1 if you want to scale the inputs to every stage in your filter, along with the output:
    - The first entry of your vector specifies the input scaling
  - Each successive entry specifies the scaling at the output of the next section
  - The final entry specifies the scaling for the filter output.

The interpretation of this property is described below with diagrams in "Interpreting the ScaleValues Property".

Default value: `1`

Remarks: The value of the `ScaleValues` property is not quantized. Data affected by the presence of a scaling factor in the filter is quantized according to the appropriate data format.

When you apply `normalize` to a quantized filter, the value for the `ScaleValues` property is changed accordingly.

It is good practice to choose values for this property that are either positive or negative powers of two.

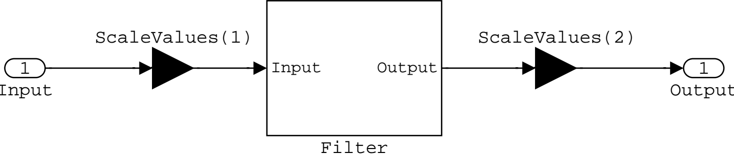
Interpreting the ScaleValues Property

When you specify the values of the `ScaleValues` property of a quantized filter, the values are entered as a vector, the length of which is determined by the number of cascaded sections in your filter:

- When you have only one section, the value of the `Scalevalues` property can be a a scalar or a two-element vector.
- When you have L cascaded sections in your filter, the value of the `Scalevalues` property can be a scalar or an L+1-element vector.

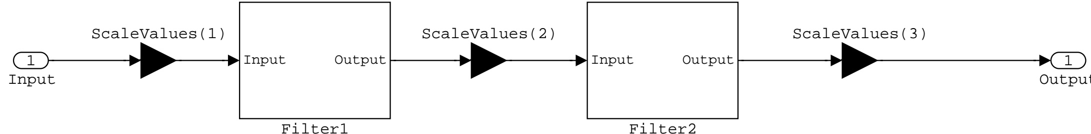
The following diagram shows how the `ScaleValues` property values are applied to a quantized filter with only one section.

Application of ScaleValues to a Single Section



The following diagram shows how the ScaleValues property values are applied to a quantized filter with two sections.

Application of ScaleValues to Multiple Sections



StatesPerSection

This read-only property value is an 1-by-L vector that specifies the number of states (delays) in each section of a quantized filter with L cascaded sections.

Default value: 0

SumFormat

The SumFormat property values specify how the results of addition are quantized during the filtering operation. You specify these values with a quantizer. You set them according to the property values of a quantizer's:

- Format (except when the Mode property value is set to 'double' or 'single')
- Mode
- OverflowMode
- RoundMode

Default value: quantizer('fixed','floor','saturate',[32,30])

A Quick Guide to Quantized FFT Properties

The following table summarizes the quantized FFT properties and provides a brief description of each. These properties are implemented when you use a quantized FFT in conjunction with an FFT or inverse FFT (IFFT)) algorithm (`fft` or `ifft`). A table providing a full description of each property follows in the next section.

Table 12-6: Quick Guide to Quantized FFT Properties

Property	Brief Description of What the Property Specifies
CoefficientFormat	Quantization format for FFT or IFFT coefficients (twiddle factors)
InputFormat	Quantization format applied to inputs to the FFT or IFFT algorithm
Length	Length of the quantized FFT or IFFT
NumberOfSections	Number of sections used in the quantized FFT algorithm
MultiplicandFormat	Quantization format for inputs that are multiplied by coefficients in the FFT or IFFT algorithm
OutputFormat	Quantization format applied to outputs of the FFT or IFFT algorithm
ProductFormat	Quantization format for results of multiplication within the FFT or IFFT algorithm
Radix	Radix value for the FFT algorithm
ScaleValues	Scaling for the inputs and stages of the FFT or IFFT algorithm
SumFormat	Quantization format for results of addition within the FFT or IFFT algorithm

Quantized FFT Properties Reference

To implement an FFT or inverse FFT (IFFT) algorithm, you specify a quantized FFT, along with its property values. When you create a quantized FFT, you are creating a MATLAB object. You specify the FFT quantization parameters as values assigned to the quantized FFT properties. With these property values, you specify the quantized FFT:

- Data formats
- Length
- Radix number (2 or 4)
- Scaling values for each stage

For a quick reference to properties, see "A Quick Guide to Quantized FFT Properties" on page 12-51. Details of all of the properties associated with quantized FFTs are described in the following sections in alphabetical order.

CoefficientFormat

The `CoefficientFormat` property values specify how FFT coefficients (*twiddle factors*) are quantized in the FFT algorithm. You specify these values with a quantizer. You set them according to the quantizer property values:

- `Format` (except when the `Mode` property value is set to `'double'` or `'single'`)
- `Mode`
- `OverflowMode`
- `RoundMode`

Default value: `quantizer('fixed','round','saturate',[16,15])`

InputFormat

The `InputFormat` property values specify how inputs are quantized in the FFT algorithm. You specify these values with a quantizer. You set them according to the property values of a quantizer's:

- `Format` (except when the `Mode` property value is set to `'double'` or `'single'`)
- `Mode`
- `OverflowMode`

- RoundMode

Default value: quantizer('fixed','floor','saturate',[16,15])

Length

The Length property value is a scalar integer indicating the length of the FFT. Specify the length as a power of the radix number.

Default value: 16

NumberOfSections

The value of this read-only property is a scalar that specifies the number of sections (stages) in your FFT algorithm. This number is computed from the Length and the Radix property values as

$\log_2(\texttt{Length})/\log_2(\texttt{Radix})$

Default value: 4

MultiplicandFormat

Products in quantized FFTs always involve two types of multiplicands:

- Inputs (data)
- Coefficients

The MultiplicandFormat property values specify how inputs that are multiplied by coefficients are quantized in the FFT algorithm. You specify these values with a quantizer. You set them according to the property values of a quantizer's:

- Format (except when the Mode property value is set to 'double' or 'single')
- Mode
- OverflowMode
- RoundMode

Default value: quantizer('fixed','floor','saturate',[16,15])

OutputFormat

The `OutputFormat` property values specify how outputs are quantized in the FFT algorithm. You specify these values with a quantizer. You set them according to the property values of a quantizer's:

- `Format` (except when the `Mode` property value is set to `'double'` or `'single'`)
- `Mode`
- `OverflowMode`
- `RoundMode`

Default value: `quantizer('fixed','floor','saturate',[16,15])`

ProductFormat

The `ProductFormat` property values specify how the results of multiplication are quantized in the FFT algorithm. You specify these values with a quantizer. You set them according to the property values of a quantizer's:

- `Format` (except when the `Mode` property value is set to `'double'` or `'single'`)
- `Mode`
- `OverflowMode`
- `RoundMode`

Default value: `quantizer('fixed','floor','saturate',[32,30])`

Radix

The `Radix` property indicates the form of the FFT algorithm you want to apply. The `Radix` property value can be either:

- `2` (default)
- `4`

ScaleValues

The `ScaleValues` property values are specified as a scalar (or vector) that introduces scaling for inputs (and the outputs from each FFT section in the vector case) to the FFT algorithm:

- Specify the `ScaleValues` property value as a scalar if you only want to scale the input to the FFT algorithm.
- Specify the `ScaleValues` property as a vector of length L if you have L sections in your FFT, and you want to scale:
  - The input to the first section (with the first entry in the vector you supply)
  - The input to each subsequent section (with each corresponding entry in the vector you supply)

Default value: `1`

Remarks: The value of the `ScaleValues` property is not quantized. Data affected by the presence of a scaling factor within the FFT algorithm is quantized according to the appropriate data format.

It is good practice to choose values for this property that are positive or negative powers of two.

SumFormat

The `SumFormat` property values specify how the results of addition are quantized in the FFT algorithm. You specify these values with a quantizer. You set them according to the property values of a quantizer's:

- `Format` (except when the `Mode` property value is set to `'double'` or `'single'`)
- `Mode`
- `OverflowMode`
- `RoundMode`

Default value: `quantizer('fixed','floor','saturate',[32,30])`

13

Function Reference

Functions—By Category (p. 13-2)	Lists the functions in the toolbox, by category, such as object constructors or analysis functions
Functions Operating on Quantized Filters (p. 13-10)	Lists the functions used on quantized filters
Functions Operating on Quantizers (p. 13-12)	Lists the functions used on quantizers
Functions Operating on Quantized FFTs (p. 13-14)	Lists the functions used on quantized FFTs
Functions for Designing Digital Filters (p. 13-16)	List the filter design functions
Functions—Alphabetical List (p. 13-19)	Introduces the alphabetical listing of reference pages for every function in the toolbox

Functions—By Category

With the Filter Design (FD) Toolbox, you can create, apply, and analyze quantized filters, quantizers, and quantized fast Fourier transforms (FFTs). This chapter contains brief descriptions of all FD Toolbox functions grouped by subject area, and continues with the detailed reference entries listed alphabetically. The following tables list the functions in the FD Toolbox, separated by quantization application—quantized filter, quantizer, or quantized FFT. In many instances, you can apply a function to more than one application; those functions are called *overloaded* functions and they appear in more than one table.

Quantized Filter Construction and Property Functions

Function	Description
`get`	Get properties of a quantized filter
`isreal`	Test if filter coefficients are real
`num2bin`	Convert a number to two's-complement binary string
`num2hex`	Convert a number to hexadecimal string
`qfilt`	Construct a quantized filter (Qfilt object)
`qreport`	Returns the listing of a quantize filter and its properties
`reset`	Reset the properties of a quantized filter to their initial values
`set`	Set properties of a quantized filter
`setbits`	Set the data format property values for a quantized filter

Quantized Filter Analysis Functions

Function	Description
freqz	Compute the frequency response for a quantized filter
impz	Compute the impulse response for a quantized filter
isallpass	Test quantized filters to determine if they are allpass structures
isfir	Test quantized filters to see if they are FIR filters
islinphase	Test quantized filters to see if they are linear phase
ismaxphase	Test quantized filters to see if they are maximum phase filters
isminphase	Test quantized filters to see if they are minimum phase filters
isreal	Test qauntized filters for purely real coefficients
issos	Test whether quantized filters are composed of second-order sections
isstable	Test for stability of quantized filters
limitcycle	Detect limit cycles in a quantized filter
nlm	Use the Noise Loading Method to estimate the frequency response of a quantized filter
zplane	Compute a pole-zero plot for a quantized filter

Table 13-1: Quantized Filtering Functions

Function	Description
filter	Filter data with a quantized filter
normalize	Normalize quantized filter coefficients

Second-Order Sections Conversion Functions

Function	Description
`cell2sos`	Convert a cell array to a second-order sections matrix
`sos`	Convert a quantized filter to second-order sections form, order, and scale
`sos2cell`	Convert a second-order sections matrix to a cell array

Quantizer Construction and Property Functions

Function	Description
`bin2num`	Convert binary string to number
`get`	Return the property values for a quantizer
`num2bin`	Convert a number to two's-complement binary string
`num2hex`	Convert a number to hexadecimal string
`qreport`	Returns the listing of a quantizer and its properties
`quantize`	Apply a quantizer to a data set
`quantizer`	Construct a quantizer object
`reset`	Reset the properties of a quantizer to their initial values
`set`	Set and display the property values of a quantizer
`unitquantize`	Set numbers between `eps(q)` and 1 equal to 1
`wordlength`	Return the wordlength of a quantizer

Quantizer Analysis Functions

Function	Description
`denormalmax`	Return the largest denormalized quantized number
`denormalmin`	Return the smallest denormalized quantized number
`eps`	Return the quantized relative accuracy of a quantizer
`errmean`	Return the mean of the quantization error resulting from quantizing a signal
`errpdf`	Calculate the probability density function (pdf) of the quantization error
`errvar`	Return the variance of the quantization error resulting from quantizing a signal
`exponentbias`	Return the exponent bias for a quantizer
`exponentlength`	Return the exponent length for a quantizer
`exponentmax`	Return the maximum exponent for a quantizer
`exponentmin`	Return the minimum exponent for a quantizer
`fractionlength`	Return the fraction length for a quantizer
`isfixed`	Test whether quantizers are fixed point
`isfloat`	Test whether quantizers are floating point
`isnone`	Test whether a quantizer has quantization mode equal to none
`noverflows`	Return the number of overflows encountered while using a quantizer on one or more data sets
`range`	Return the numerical range of a quantizer
`realmax`	Return the largest positive quantized number

Quantizer Analysis Functions

Function	Description
`realmin`	Return the smallest positive quantized number
`nunderflows`	Return the number of underflows encountered while using a quantizer on one or more data sets

Quantized FFT Construction and Property Functions

Function	Description
`fft`	Apply a quantized FFT to a data set
`get`	Return the property values for a quantized FFT
`ifft`	Apply an inverse quantized FFT to a data set
`qfft`	Construct a quantized FFT
`qreport`	Returns the listing of a quantized FFT and its properties
`quantizer`	Return all the quantizers associated with a quantized FFT
reset	Reset the properties of a quantized FFT to their initial values
`set`	Set and display the property values of a quantized FFT
`setbits`	Set and one or more property values of a quantized FFT

Quantized FFT Analysis Functions

Function	Description
`noverflows`	Return the number of overflows resulting from the most recent application of a quantized FFT

Quantized FFT Analysis Functions

Function	Description
range	Return the numerical range of a quantized FFT
twiddles	Return the twiddle factors for a quantized FFT

Filter Design Functions

Function	Description
cicdecimate	Use a cascaded integrator-comb (CIC) decimation filter to decrease the sampling rate for a signal
cicinterpolate	Use a cascaded integrator-comb (CIC) interpolation filter to increase the sampling rate for a signal
firceqrip	Design constrained, equiripple, finite impulse response (FIR) filters
firlpnorm	Design least-pth norm optimal FIR filters
firhalfband	Design half-band FIR filters
firminphase	Compute the minimum phase FIR spectral factor of linear phase FIR filters
firnyquist	Design lowpass Nyquist (L-th band) FIR filters
gremez	Design optimal equiripple FIR (finite impulse response) digital filters based on the Parks-McClellan algorithm
ifir	Design interpolated FIR filters
iircomb	Design comb IIR filters with periodic frequency response
iirgrpdelay	Design least-pth norm IIR filters with given group delay
iirlpnorm	Design least-pth norm IIR filters

Filter Design Functions

Function	Description
iirlpnormc	Design constrained least-pth norm IIR filters
iirnotch	Design notch IIR filters to attenuate a fixed frequency
iirpeak	Design peaking IIR filters for boosting or cutting specific frequencies

Filter Conversion Functions

Function	Description
ca2tf	Convert coupled allpass filters to transfer function form
cl2tf	Convert lattice coupled allpass filters to transfer function form
firlp2lp	Transform lowpass FIR filters to lowpass filters with different passband specifications
firlp2hp	Transform lowpass FIR filters to highpass FIR filters
iirlp2bp	Transform lowpass IIR filters to bandpass filters
iirlp2bs	Transform lowpass IIR filters to bandstop filters
iirlp2hp	Transform lowpass IIR filters to highpass filters
iirlp2lp	Transform lowpass IIR filters to lowpass filters
iirpowcomp	Compute the power complementary IIR filter
tf2ca	Convert transfer function form to coupled allpass form
tf2cl	Convert transfer function form to lattice coupled allpass form

Adaptive Filter Design Functions and Their Initialization Functions

Function	Initializing Function	Description
adaptkalman	initkalman	Use a Kalman filter in an adaptive filtering application
adaptlms	initlms	Use a least mean squares (LMS) algorithm filter in an adaptive filtering application
adaptnlms	initnlms	Use a normalized LMS algorithm filter in an adaptive filtering application
adaptrls	initrls	Use a recursive least squares algorithm filter in an adaptive filtering application
adaptsd	initsd	Use a sign-data variant of the LMS algorithm filter in an adaptive filtering application
adaptse	initse	Use a sign-error variant of the LMS algorithm filter in an adaptive filtering application
adaptss	initss	Use a sign-sign variant of the LMS algorithm filter in an adaptive filtering application

Functions Operating on Quantized Filters

The following table lists functions that operate directly on quantized filters. Some are overloaded and operate on other quantized objects, such as quantized FFTs as well. Overloaded functions are marked in the table.

Functions	Functions That Operate Directly on Quantized Filters	Overloaded Functions
convert	÷	
copyobj	÷	÷
disp	÷	÷
eps	÷	÷
filter	÷	÷
freqz	÷	÷
get	÷	÷
impz	÷	÷
isallpass	÷	
isfir	÷	
islinphase	÷	
ismaxphase	÷	
isminphase	÷	
isreal	÷	÷
issos	÷	
isstable	÷	
limitcycle	÷	
nlm	÷	

Functions	Functions That Operate Directly on Quantized Filters	Overloaded Functions
noperations	÷	÷
normalize	÷	
noverflows	÷	÷
num2bin	÷	
num2hex	÷	
optimizeunitygains	÷	
order	÷	
qfilt	÷	
qfilt2tf	÷	
range	÷	÷
reset	÷	÷
set	÷	÷
setbits	÷	÷
sos	÷	
zplane	÷	÷

To get command line help on an overloaded function FunctionName for quantized filters, type

```
help qfilt/FunctionName
```

Functions Operating on Quantizers

The following table lists functions that operate directly on quantizers. Some are overloaded and operate on other quantized objects, such as quantized FFTs as well. Overloaded functions are marked in the table

Functions	Functions That Operate Directly on Quantizers	Overloaded Functions
bin2num	÷	÷
copyobj	÷	÷
denormalmax	÷	
denormalmin	÷	
disp	÷	÷
eps	÷	÷
exponentbias	÷	
exponentlength	÷	
exponentmax	÷	
exponentmin	÷	
fractionlength	÷	
get	÷	÷
hex2num	÷	
max	÷	
min	÷	
noperations	÷	÷
noverflows	÷	÷
num2bin	÷	÷

Functions	Functions That Operate Directly on Quantizers	Overloaded Functions
num2hex	÷	÷
nunderflows	÷	
qreport	÷	÷
quantize	÷	
quantizer	÷	
randquant	÷	
range	÷	÷
realmax	÷	
realmin	÷	
reset	÷	÷
set	÷	÷
tostring	÷	÷
unitquantize	÷	
unitquantizer	÷	
wordlength	÷	

To get command line help for an overloaded function `FunctionName` for quantizers, type

```
help quantizer/FunctionName
```

Functions Operating on Quantized FFTs

The following table lists functions that operate directly on quantized FFTs. Some are overloaded and operate on other quantized objects, such as quantized filters as well. Overloaded functions are marked in the table

Functions	Functions That Operate Directly on Quantized FFTs	Overloaded Functions
copyobj	÷	÷
disp	÷	÷
eps	÷	÷
fft	÷	
get	÷	÷
ifft	÷	
noperations	÷	÷
noverflows	÷	÷
optimizeunitygains	÷	÷
qfft	÷	
qreport	÷	÷
quantizer	÷	÷
range	÷	÷
reset	÷	÷
set	÷	÷
setbits	÷	÷
tostring	÷	÷
twiddles	÷÷	

To get command line help on an overloaded function `FunctionName` for quantized FFTs, type

```
help qfft/FunctionName
```

Functions for Designing Digital Filters

The following functions design digital FIR filters:

- `firceqrip`
- `firlpnorm`
- `firhalfband`
- `firminphase`
- `firnyquist`
- `gremez`
- `ifir`

The following functions design digital IIR filters:

- `cicdecimate`
- `cicinterpolate`
- `iircomb`
- `iirgrpdelay`
- `iirlpnorm`
- `iirlpnormc`
- `iirnotch`
- `iirpeak`

The following functions design adaptive filters:

- `adaptkalman`
- `adaptlms`
- `adaptrls`
- `adaptsd`
- `adaptse`
- `adaptss`

The following functions transform the frequency response of digital filters from one type to another, such as lowpass to highpass:

IIR transforms

- `firlp2lp`
- `firlp2hp`
- `iirlp2bp`
- `iirlp2bs`
- `iirlp2hp`
- `iirlp2lp`
- `iirlp2mb`
- `iirlp2xn`
- `iirlp2bpc`
- `iirlp2bsc`
- `iirshiftc`
- `iirlp2mbc`
- `iirlp2xc`
- `iirbpc2bpc`
- `iirrateup`
- `iirftransf`

ZPK transforms

- `zpklp2lp`
- `zpklp2hp`
- `zpklp2bp`
- `zpklp2bs`
- `zpkshift`
- `zpklp2mb`
- `zpklp2xn`
- `zpklp2bpc`
- `zpklp2bsc`
- `zpkshiftc`
- `zpklp2mbc`
- `zpklp2xc`

- `zpkbpc2bpc`
- `zpkrateup`
- `zpkftransf`

The following functions convert the structures of digital filters:

- `ca2tf`
- `cl2tf`
- `iirpowcomp`
- `qfilt2tf`
- `tf2ca`
- `tf2cl`

To get command line help on a design or conversion function such as `gremez` or `quantizer`, type either

- `help gremez`
- `help objecttype/quantizer` where `objecttype` is one of the following strings that specify the version of help to see:
  - `qfilt`
  - `qfft`
  - `quant`

Functions—Alphabetical List

The following reference pages list the functions included in the Filter Design Toolbox. Each function listing provides a purpose, syntax, description, algorithm (optional), and examples for the function.

adaptkalman

Purpose

Use a discrete-time Kalman filter in an adaptive filtering application

Syntax

```
y = adaptkalman(x,d,s)
[y,e] = adaptkalman(x,d,s)
[y,e,s] = adaptkalman(x,d,s)
```

Description

`y = adaptkalman(x,d,s)` applies a Kalman adaptive filter to the data vector `x` and the desired signal `d`. The filtered data is returned in `y`. To return the filter states after adaptation, specify the output argument `s`.

`s` is a structure containing the initialization settings that define the Kalman filter you are using and some output results, as shown in the table that follows. In the third column of the table, you see a list showing how the input arguments to `initkalman` correspond to elements in `s`.

Structure Element	Element Description	initkalman argument
`s.coeffs`	Kalman adaptive filter coefficients. Should be initialized with the initial values for the FIR filter coefficients. Updated coefficients are returned when you use `s` as an output argument. Contains filter order plus one elements in a vector.	`w0`
`s.errcov`	The state error covariance matrix. Initialize this element with the initial error state covariance matrix. An updated matrix is returned when you use `s` as an output argument. This is a square matrix of dimension filter order plus one. For example, for a 32nd-order filter, `s.errcov` is a 33-by-33 matrix.	`k0`

Structure Element	Element Description	initkalman argument
s.measvar	Contains the measurement noise variance matrix. Use the same value for all the elements in the matrix and adaptkalman returns a matrix of noise variance values — a square matrix of dimension filter order plus one. For example, for a 32nd-order filter, s.measvar is a 33-by-33 matrix.	qm
s.procov	Contains the process noise covariance matrix. This is a square matrix of dimension filter order plus one. For example, for a 32nd-order filter, s.procov is a 33-by-33 matrix.	qp
s.states	Returns the states of the FIR filter when use s as an output argument. This is an optional input element. If omitted on input, it defaults to a zero vector of length equal to the filter order.	zi
s.gain	Kalman gain vector. Computed and returned after every iteration. This is a read-only value.	
s.iter	Total number of iterations in adaptive filter run. This is a read-only value.	

Use initkalman to configure the elements of input argument structure s.

[y,e] = adaptkalman(...) also returns the prediction error e.

[y,e,s] = adaptkalman(...) returns the updated structure s.

In applications where you need to know the intermediate filter states as the filter adapts to the unknown system, call adaptkalman inside a conditional program statement such as the following for-loop example.

```
for n = 1:length(x)
[y(n),e(n),s] = adaptkalman(x(n),d(n),s);
% States (The fields of s) here may be modified here.
```

```
end
```

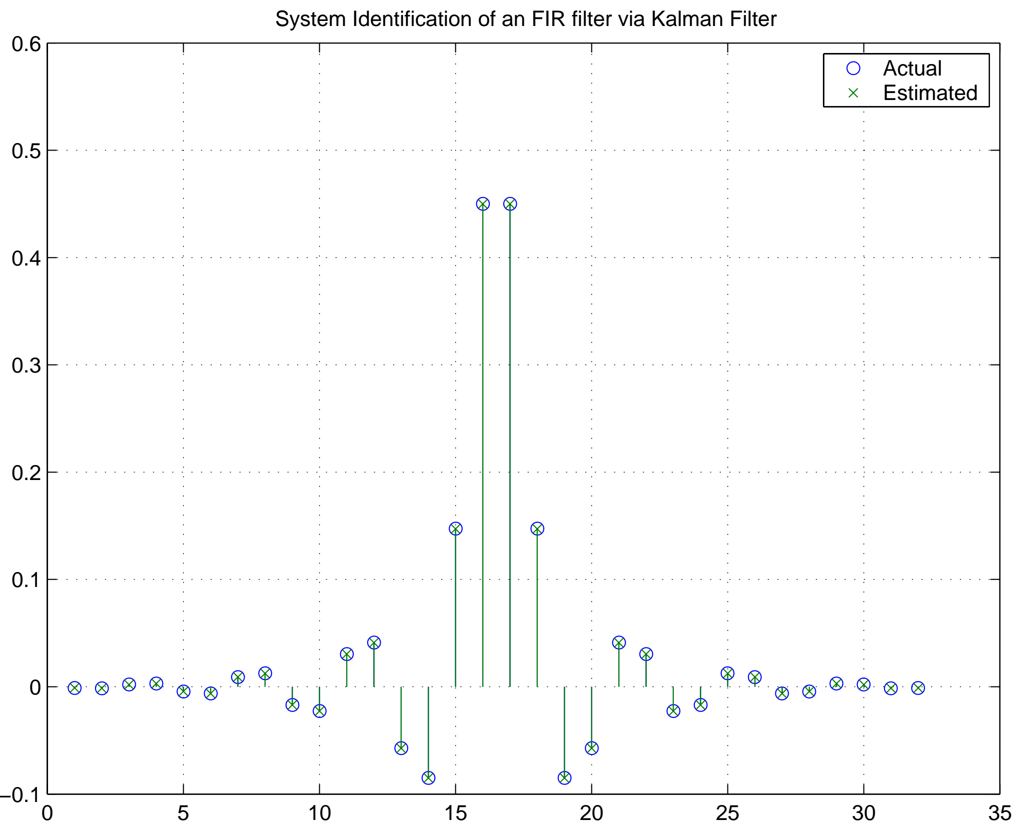
In lieu of assigning the structure fields for s manually, use initkalman to populate structure s.

Examples

Use an adaptive Kalman filter to identify an unknown 32nd-order FIR filter (500 iterations). From Signal Processing Toolbox we use fir1 to create our unknown windowed lowpass FIR filter.

```
x  = 0.1*randn(1,500); % Input to the filter
b  = fir1(31,0.5);     % FIR system to be identified
d  = filter(b,1,x);    % Desired signal
w0 = zeros(1,32);      % Intial filter coefficients
k0 = 0.5*eye(32);      % Initial state error correlation matrix
qm = 2;                % Measurement noise covariance
qp = 0.1*eye(32);      % Process noise covariance
s = initkalman(w0,k0,qm,qp);
[y,e,s] = adaptkalman(x,d,s);
stem([b.',s.coeffs.']);
legend('Actual','Estimated');
title('System Identification of an FIR filter via Kalman Filter');
grid on;
```

In the stem plot, you see that the original filter and the Kalman approximation/identification filter have identical response characteristics.



See Also

`initkalman`, `adaptlms`, `adaptnlms`, `adaptrls`, `adaptsd`, `adaptse`, `adaptss`

References

Haykin, S., *Adaptive Filter Theory*, Third Edition, Prentice-Hall, Inc., 1996.

Purpose

Use a least mean squared (LMS) FIR adaptive filter in an adaptive filtering application

Syntax

```
y = adaptlms(x,d,s)
[y,e] = adaptlms(x,d,s)
[y,e,s] = adaptlms(x,d,s)
```

Description

`Y = adaptlms(x,d,s)` applies an FIR LMS adaptive filter to the data vector `x` and the desired signal `d`. The filtered data is returned in `y`. `s` is a structure that contains initialization settings that define the LMS adaptive algorithm you plan to use, as well as some output from the filter adaptation process. The following table details the contents of `s`, both input and output.The column headed initlms Element shows you which element in `s` corresponds to each input argument to `initlms`.

Structure Element	Element Contents	initlms Element
`s.coeffs`	LMS FIR filter coefficients. Should be initialized with the initial coefficients for the FIR filter prior to adapting. You need (adapting filter order + 1) entries in `s.coeffs`. Updated filter coefficients are returned in `s.coeffs` when you use `s` as an output argument.	`wo`
`s.step`	Sets the LMS algorithm step size. Determines both how quickly and how closely the adative filter adapts to the filter solution.	`mu`
`s.states`	Returns the states of the FIR filter after adaptation. This is an optional element. If omitted, it defaults to a zero vector of length equal to the filter order. When you use `adaptlms` in a loop structure, use this element to specify the initial filter states for the adapting FIR filter.	`zi`

Structure Element	Element Contents	initlms Element
`s.leakage`	Specifies the LMS leakage parameter. Allows you to implement a leaky LMS algorithm. Including a leakage factor can improve the results of the algorithm by forcing the LMS algorithm to continue to adapt even after it reaches a minimum value. Ranges between 0 and 1. This is an optional field. Defaults to one if omitted (specifying no leakage) or set to empty, [ ].	`lf`
`s.iter`	Total number of iterations in the adaptive filter run. Although you can set this in s, you should not. Consider it a read-only value.	

`[y,e] = adaptlms(...)` also returns the prediction error `e`. Ultimately this shows you how well the filter adapted to the desired signal and input data. Or how well `y` approximates `d`.

`[y,e,s] = adaptlms(...)` returns the updated structure `s`.

`adaptlms` can be called for a block of data, when `x` and `d` are vectors, or in "sample by sample mode" using programming constructs such as the following for-loop code.

```
for n = 1:length(x)
[y(n),e(n),s] = adaptlms(x(n),d(n),s);
 % The fields of s may be modified here.
end
```

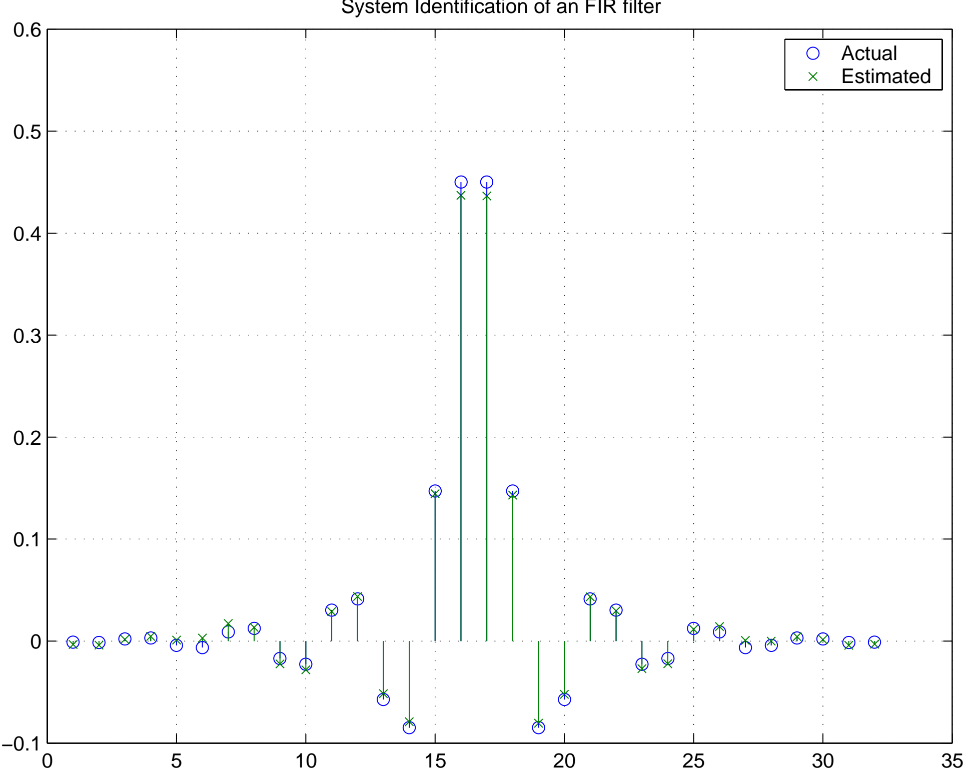
In lieu of assigning the structure fields manually, call `initlms` to populate the structure `s` more easily.

Examples

System Identification of a 31st-order FIR filter (500 iterations). Identifying the characteristics of an unknown filter is a classic problem for adaptive filtering. This example uses an FIR filter as the unknown, and uses the LMS algorithm to calculate weights for the adapting filter. The stem plot that follows the

example code demonstrates that the adapted filter matches the unknown quite closely.

```
x  = 0.1*randn(1,500); % Input to the filter
b  = fir1(31,0.5);     % FIR system to be identified
d  = filter(b,1,x);    % Desired signal
w0 = zeros(1,32);      % Intial filter coefficients
mu = 0.8;              % LMS step size.
s = initlms(w0,mu);
[y,e,s] = adaptlms(x,d,s);
stem([b.',s.coeffs.']);
legend('Actual','Estimated');
title('System Identification of an FIR filter');grid on;
```



Algorithm

In vector form, the LMS algorithm is

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \mu e(k)\mathbf{x}(k)$$

with vector **w** containing the weights applied to the filter coefficients (s.coeffs) and vector **x** containing the input data. *e(k)* (equal to desired signal - filtered signal) is the error at time k and is the quantity the LMS algorithm seeks to minimize. μ (`mu`, and `s.step`)) is the step size. As you specify `mu` smaller, the correction to the filter weights gets smaller for each sample and the LMS error falls more slowly. Larger `mu` 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 good convergence rate and stability, select mu within the following practical bounds:

$$0 < \frac{\mu}{2} < \frac{1}{N\{InputSignalPower\}}$$

where N is the number of samples in the signal.

When you add a leakage factor lf, the algorithm changes to

$$\mathbf{w}(k+1) = c\mathbf{w}(k) + \mu e(k)\mathbf{x}(k)$$

with c representing lf.

See Also

initlms, adaptkalman, adaptnlms, adaptrls, adaptsd, adaptse, adaptss

References

Haykin, S., *Adaptive Filter Theory*, Third Edition, Prentice-Hall, Inc., 1996.

Purpose

Use a normalized least mean squared (NLMS) FIR adaptive filter in an adaptive filtering application

Syntax

```
y = adaptnlms(x,d,s)
[y,e] = adaptnlms(x,d,s)
[y,e,s] = adaptnlms(x,d,s)
```

Description

`y = adaptlms(x,d,s)` applies an FIR normalized LMS adaptive filter to the data vector `x` and the desired signal `d`. The filtered data is returned in `y`. Structure `s` contains initialization settings that define the NLMS adaptive algorithm you plan to use, as well as some output from the filter adaptation process. The following table details the contents of `s`, both input and output. The column headed **initnlms Element** shows you which element in `s` corresponds to each input argument to `initlms`.

Structure Element	Element Contents	initnlms Element
`s.coeffs`	NLMS FIR filter coefficients. Should be initialized with the initial coefficients for the FIR filter prior to adapting. You need (adapting filter order + 1) entries in `s.coeffs`. Updated filter coefficients are returned in `s.coeffs` when you use `s` as an output argument.	`wo`
`s.offset`	Specifies an optional offset for the normalization term. Use this to avoid divide by zero (or by very small numbers) when the square of input data norm becomes very small. When omitted, it defaults to zero.	`offset`
`s.step`	Sets the NLMS algorithm step size. Determines both how quickly and how closely the adative filter adapts to the filter solution.	`mu`

Structure Element	Element Contents	initnlms Element
s.states	Returns the states of the FIR filter after adaptation. This is an optional element. If omitted, it defaults to a zero vector of length equal to the filter order. When you use adaptlms in a loop structure, use this element to specify the initial filter states for the adapting FIR filter.	zi
s.leakage	Specifies the NLMS leakage parameter. Allows you to implement a leaky NLMS algorithm. Including a leakage factor can improve the results of the algorithm by forcing the NLMS algorithm to continue to adapt even after it reaches a minimum value. This is an optional field. Defaults to one if omitted (specifying no leakage) or set to empty, [ ].	lf
s.iter	Returns the total number of iterations in the adaptive filter run. Although you can set this in s, you should not. Consider it a read-only value.	

[y,e] = adaptnlms(...) also returns the prediction error e. Ultimately this shows you how well the filter adapted to the desired signal and input data — how well y approximates d.

[y,e,s] = adaptnlms(...) returns the updated structure s.

adaptlms can be called for a block of data, when x and d are vectors, or in“sample by sample mode” using conditional program statements such as the following For-loop code.

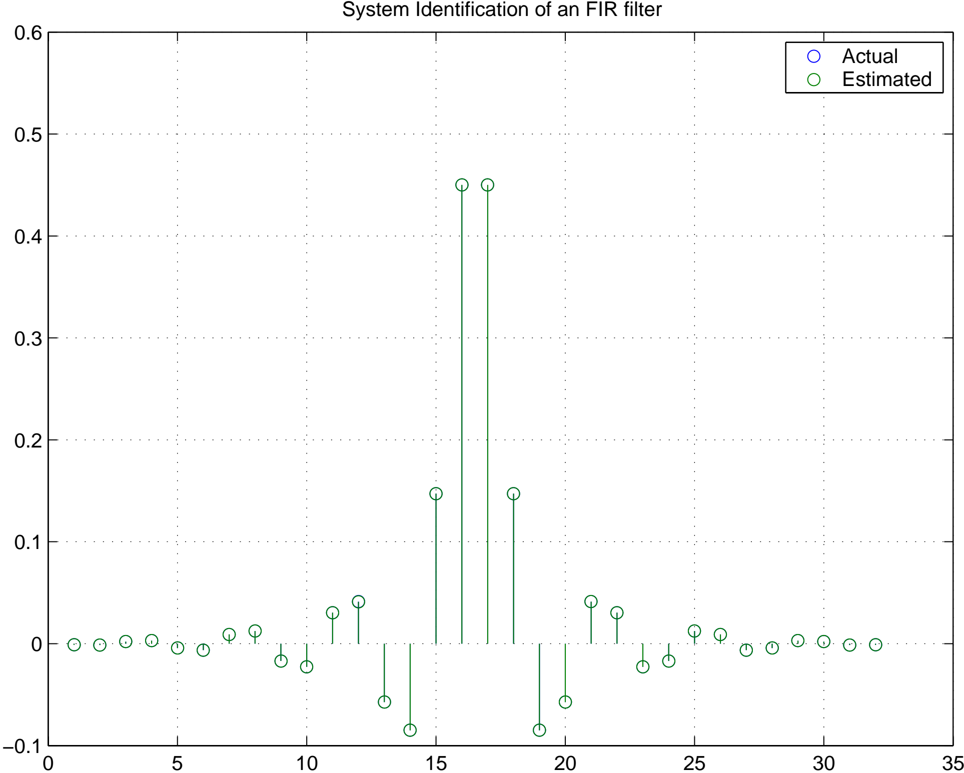
```
for n = 1:length(x)
[y(n),e(n),s] = adaptnlms(x(n),d(n),s);
 % The fields of s may be modified here.
end
```

In lieu of assigning the structure fields manually, call initlms to populate the structure s more easily.

Examples

System Identification of a 31st-order FIR filter (500 iterations). Identifying the characteristics of an unknown filter is a classic problem for adaptive filtering. This example uses an FIR filter as the unknown, and uses the normalized LMS algorithm to calculate weights for the adapting filter. The stem plot that follows the example code demonstrates that the adapted filter matches the unknown quite closely.

```
x  = 0.1*randn(1,500); % Input to the filter
b  = fir1(31,0.5);     % FIR system to be identified
d  = filter(b,1,x);    % Desired signal
w0 = zeros(1,32);      % Intial filter coefficients
mu = 0.8;              % NLMS step size.
s = initnlms(w0,mu);
[y,e,s] = adaptnlms(x,d,s);
stem([b.',s.coeffs.']);
legend('Actual','Estimated');
title('System Identification of an FIR filter');grid on;
```



Algorithm

In vector form, the NLMS algorithm is

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \mu_n e(k)\mathbf{x}(k)$$

where

$$\mu_n = \frac{1}{\varepsilon + \|x(k)\|^2}$$

and ε is the offset that prevents divide by zero situations and $\|x(k)\|$ is the 2-norm of vector $x(k)$.

Vector **w** contains the weights applied to the filter coefficients (`s.coeffs`) and vector **x** contains the input data. $e(k)$ (equal to desired signal - filtered signal) is the error at time k and is the quantity the NLMS algorithm seeks to minimize. μ (`mu`) is the normalized step size (`s.step`). As you specify `mu` smaller, the correction to the filter weights gets smaller for each sample and the NLMS error falls more slowly. Larger `mu` 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 good convergence rate and stability, select `mu` within the following practical bounds:

$$0 < \mu < \frac{1}{N\{InputSignalPower\}}$$

where N is the number of samples in the signal.

When you add a leakage factor, the NLMS algorithm changes to

$$\mathbf{w}(k+1) = c\mathbf{w}(k) + \mu_n e(k)\mathbf{x}(k)$$

with `c` representing the leakage factor.

See Also

`initnlms`, `adaptkalman`, `adaptlms`, `adaptrls`, `adaptsd`, `adaptse`, `adaptss`

References

Haykin, S., *Adaptive Filter Theory*, Third Edition, Prentice-Hall, Inc., 1996.

adaptrls

Purpose

Use a recursive least-squares (RLS) FIR adaptive filter in an adaptive filtering application

Syntax

```
y = adaptrls(x,d,s)
[y,e] = adaptrls(x,d,s)
[y,e,s] = adaptrls(x,d,s)
```

Description

`y = adaptrls(x,d,s)` applies an FIR RLS adaptive filter to the data vector `x` and the desired signal `d`. The filtered data is returned in `y`. Structure `s` contains the RLS adaptive filter information that defines the algorithm being used. In addition, the final states of the adapted filter appear in `s.states` when you use `s` as an output argument.

Structure Element	initrls Element	Element Contents
`s.coeffs`	`w0`	RLS adaptive filter coefficients. Initialize `s.coeffs` with the initial values for the FIR filter coefficients. Updated filter coefficients after adapting are returned when `s` is an output argument.
`s.invcov`	`p0`	The inverse of the input covariance matrix. Initialize with the initial input covariance matrix inverse. The updated covariance matrix is returned when `s` is an output argument and you specify the '`direct`' RLS algorithm.
`s.lambda`	`lambda`	The forgetting factor. Determines how the RLS algorithm handles past input data — whether all data weighs equally in the algorithm or earlier data loses weight as it falls farther into the past.

Structure Element	initrls Element	Element Contents
s.states	zi	Returns the states of the FIR filter after adaptation. This is an optional element. If omitted, it defaults to a zero vector of length equal to the filter order. When you use adaptrls in a loop structure, use this element to specify the initial filter states for the adapting FIR filter.
s.gain		RLS algorithm gain value. Computed and returned after every iteration. This is a read-only value.
s.iter		Returns the total number of iterations in the adaptive filter run. Although you can set this in s, you should not. Consider it a read-only value.
s.alg	alg	Algorithm to use. Optional field. Can be one of '**direct**' for the conventional RLS algorithm or '**sqrt**' for the more stable square root (QR) method.

[y,e] = adaptrls(x,d,s) also returns the prediction error e. Ultimately this shows you how well the filter adapted to the desired signal and input data — how well y approximates d.

[y,e,s] = adaptrls(x,d,s) returns the updated structure s.

In an application where the intermediate states are important, call this function in a “sample by sample mode” using a For-loop.

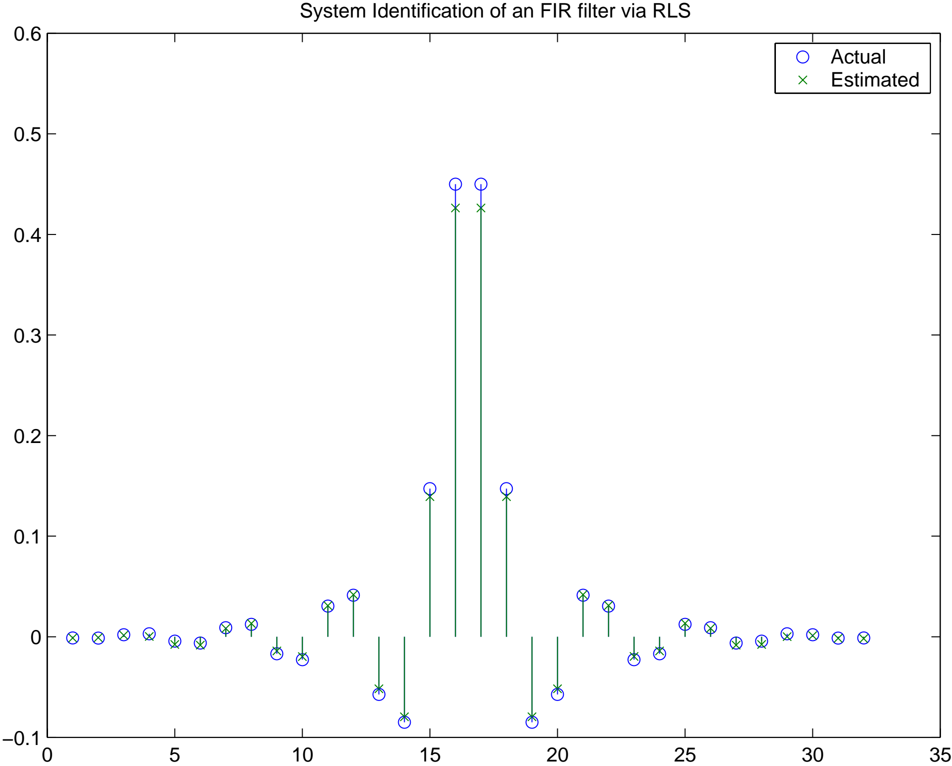
```
for n = 1:length(x)
[y(n),e(n),s] = adaptrls(x(n),d(n),s);
% States (The fields of S) here may be modified here.
end
```

In lieu of assigning the strucure fields manually, the initrls function can be called to populate the structure S.

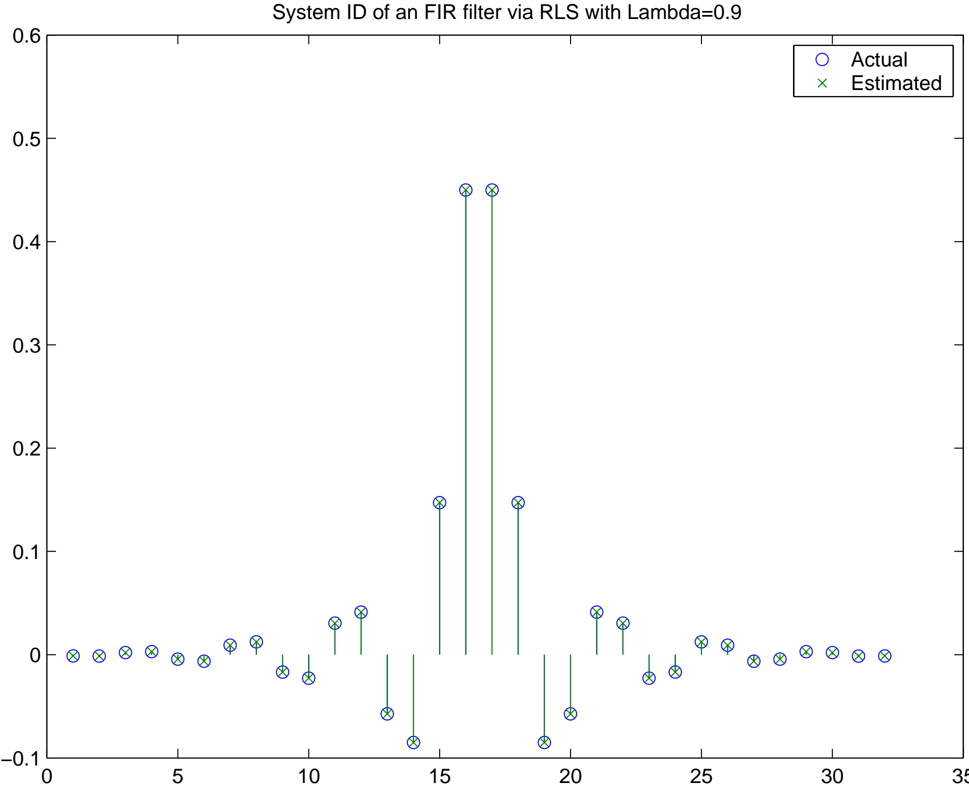
Examples

System Identification of a 32nd-order FIR filter (500 iterations). Identifying the characteristics of an unknown filter is a classic problem for adaptive filtering. This example uses an FIR filter as the unknown, and uses the RLS algorithm to calculate weights for the adapting filter. The stem plot that follows the example code demonstrates that the adapted filter matches the unknown quite closely.

```
x  = 0.1*randn(1,500); % Desired signal.
b  = fir1(32,0.55);     % FIR system to be identified.
d  = filter(b,1,x);    % Input to the adapting filter.
w0 = zeros(1,33);      % Intial filter coefficients.
p0 = 5*eye(33);       % Initial input correlation matrix inverse.
lambda = 1.0;         % Exponential memory weighting factor.
s = initrls(w0,p0,lambda);
[y,e,s] = adaptrls(x,d,s);
stem([b.',s.coeffs.']);
legend('Actual','Estimated');
title('System Identification via RLS'); grid on;
```



Notice that the estimated filter misses on the actual coefficients between 15 and 20. By changing `lambda` from `1.0` to `0.9`, we can make the actual and estimated match more closely, as shown in the next figure.



Algorithm

In vector form, the RLS algorithm, using exponential weighting, is]

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \mathbf{m}_k e(k)$$

where $\mathbf{m}_k$ and $\mathbf{P}_k$ are defined as

$$\mathbf{m}_k = \frac{\mathbf{P}_{k-1} - \mathbf{x}_k}{\left[\lambda + \mathbf{x}_k^T \mathbf{P}_{k-1} \mathbf{x}_k\right]}$$

$$\mathbf{P}_k = \frac{\mathbf{P}_{k-1} - \mathbf{m}_k \mathbf{x}_k^T \mathbf{P}_{k-1}}{\lambda}$$

Compared to the LMS algorithm used by `adaptlms`, `adaptnlms`, and others, the RLS algorithm can provide smaller error and faster convergence.

See Also

`initrls`, `adaptkalman`, `adaptlms`, `adaptnlms`, `adaptsd`, `adaptse`, `adaptss`

References

Haykin, S., *Adaptive Filter Theory*, Third Edition, Prentice-Hall, Inc., 1996.

A.H. Sayed and Kailath, T., “A State-space Approach to RLS Adaptive Filtering,” IEEE Signal Processing Magazine, July 1994, pp. 18-60.

adaptsd

Purpose

Use a sign-data FIR adaptive filter in an adaptive filter application

Syntax

```
y = adaptsd(x,d,s)
[y,e] = adaptsd(x,d,s)
[y,e,s] = adaptsd(x,d,s)
```

Description

`y = adaptsd(x,d,s)` applies an FIR LMS adaptive filter to the data vector `x` and the desired signal `d`. In this variation of the LMS algorithm, called sign-data LMS, the filtered data is returned in `y`. Structure `s` contains initialization settings that define the SDLMS adaptive algorithm you plan to use, as well as some output from the filter adaptation process. The following table details the contents of `s`, both input and output. The column **initlms Element** shows you which element in `s` corresponds to each input argument to `initsd`.

Structure Element	Element Contents	initlms Element
`s.coeffs`	SDLMS FIR filter coefficients. Should be initialized with the initial coefficients for the FIR filter prior to adapting. You need (adapting filter order + 1) entries in `s.coeffs`. Updated filter coefficients are returned in `s.coeffs` when you use `s` as an output argument.	`wo`
`s.step`	Sets the SDLMS algorithm step size. Determines both how quickly and how closely the adaptive filter adapts to the filter solution.	`mu`
`s.states`	Returns the states of the FIR filter after adaptation. This is an optional element. If omitted, it defaults to a zero vector of length equal to the filter order. When you use `adaptsd` in a loop structure, use this element to specify the initial filter states for the adapting FIR filter.	`zi`

Structure Element	Element Contents	initlms Element
`s.leakage`	Specifies the SDLMS leakage parameter. Allows you to implement a leaky LMS algorithm. Including a leakage factor can improve the results of the algorithm by forcing the LMS algorithm to continue to adapt even after it reaches a minimum value. This is an optional field. Defaults to one if omitted (specifying no leakage) or set to empty, [ ].	`lf`
`s.iter`	Total number of iterations in the adaptive filter run. Although you can set this in `s`, you should not. Consider it a read-only value.	

`[y,e] = adaptsd(...)` also returns the prediction error `e`. Ultimately this shows you how well the filter adapted to the desired signal and input data. Or how well `y` approximates `d`.

`[y,e,s] = adaptsd(...)` returns the updated structure `s`.

`adaptsd` can be called for a block of data, when `x` and `d` are vectors, or in a "sample by sample mode" using conditional program statements such as the following for-loop code.

```
for n = 1:length(x)
[y(n),e(n),s] = adaptsd(x(n),d(n),s);
 % The fields of s may be modified here.
end
```

In lieu of assigning the structure fields manually, call `initsd` to populate the structure `s` more easily.

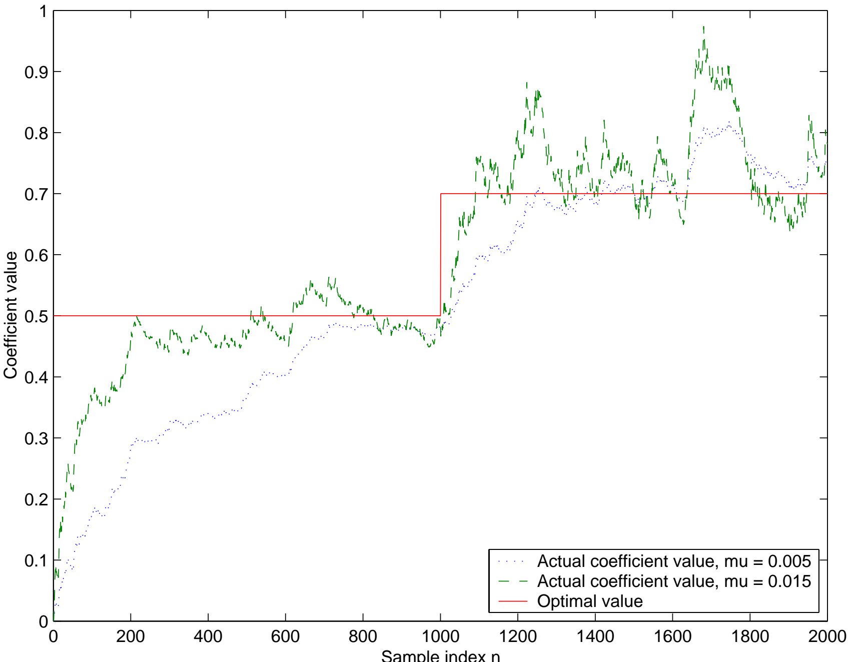
Examples

To demonstrate the effects of using different step sizes, we use adaptive linear prediction with two different step sizes to identify an FIR filter whose coefficients change with time. This example generates two sets of filter coefficients to compare to the ideal coefficients.

```
u = randn(1,2000); % Input
y1 = filter(1,[1,-.5],u(1:1000));
```

```
y2 = filter(1,[1,-.7],u(1001:2000));
y = [y1,y2]; % Construct a filter with non-stationary
coefficients
mu1 = 0.005; mu2 = 0.015; w0 = zeros(1,2);
s1 = initsd(w0,mu1); s2 = initsd(w0,mu2);
for n = 1:length(y),
   [z1(n),e1(n),s1] = adaptsd(u(n),y(n),s1);
   [z2(n),e2(n),s2] = adaptsd(u(n),y(n),s2);
   coeffs1(n,:) = s1.coeffs; coeffs2(n,:) = s2.coeffs;
end
plot([coeffs1(:,2),coeffs2(:,2),[.5*ones(1000,1);...
0.7*ones(1000,1)]])
legend('Actual coefficient value, mu = 0.005',...
'Actual coefficient value, mu = 0.015','Optimal value',4);
xlabel('Sample index n'),ylabel('Coefficient value');
```

In the figure, the coefficients generated using `mu=0.005` converge more closely to the ideal; the `mu=0.015` case coefficients converge more quickly but less closely. In the end, the resulting coefficients for both cases are quite similar if not the same. When the FIR filter coefficients change at sample index = 1000, the adapting SDLMS algorithm changes to match the new filter.



Algorithm

In vector form, the SDLMS algorithm is

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \mu e(k) sgn[\mathbf{x}(k)] \;,\; sgn[\mathbf{x}(k)] = \begin{cases} 1, & \mathbf{x}(k) > 0 \\ 0, & \mathbf{x}(k) = 0 \\ -1, & \mathbf{x}(k) < 0 \end{cases}$$

with vector **w** containing the weights applied to the filter coefficients and vector **x** containing the input data. *e(k)* (equal to desired signal - filtered signal) is the error at time *k* and is the quantity the SDLMS algorithm seeks to minimize. μ (`mu`) is the step size. As you specify `mu` smaller, the correction to the filter weights gets smaller for each sample and the SDLMS error falls more slowly. Larger `mu` 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 good convergence rate and stability, select `mu` within the following practical bounds:

$$0 < \mu < \frac{1}{N\{InputSignalPower\}}$$

where N is the number of samples in the signal. Also, define `mu` as a power of two for efficient computing.

See Also

`initsd`, `adaptse`, `adaptss`, `adaptkalman`, `adaptlms`, `adaptnlms`, `adaptrls`

References

Hayes, M.H., *Statistical Digital Signal Processing and Modeling*, John Wiley and Sons, 1996

Diniz, P, "Adaptive Filtering. Algorithms and Practical Implementation," Kluwer Academic Publishers, Bostom, 1997.

Purpose

Apply a sign-error FIR adaptive filter in an adaptive filter application

Syntax

```
y = adaptse(x,d,s)
[y,e] = adaptse(x,d,s)
[y,e,s] = adaptse(x,d,s)
```

Description

`y = adaptse(x,d,s)` applies a sign-error LMS (SELMS) FIR adaptive filter to the data vector `x` and the desired signal `d`. Note that for this algorithm both `x` and `d` must be real. The filtered data is returned in `y`. Structure `s` contains the adaptive filter algorithm information.

Structure Element	Element Contents Description	initse Element
`s.coeffs`	SELMS FIR filter coefficients. Should be initialized with the initial coefficients for the FIR filter prior to adapting. You need (adapting filter order + 1) entries in `s.coeffs`. Updated filter coefficients are returned in `s.coeffs` when you use `s` as an output argument.	`w0`
`s.step`	Sets the SELMS algorithm step size. Determines both how quickly and how closely the adaptive filter adapts to the filter solution.	`mu`
`s.states`	Returns the states of the FIR filter after adaptation. This is an optional element. If omitted, it defaults to a zero vector of length equal to the filter order. When you use `adaptse` in a loop structure, use this element to specify the initial filter states for the adapting FIR filter.	`zi`

Structure Element	Element Contents Description	initse Element
s.leakage	Specifies the SELMS leakage parameter. Allows you to implement a leaky SELMS algorithm. Including a leakage factor can improve the results of the algorithm by forcing the SELMS algorithm to continue to adapt even after it reaches a minimum value. This is an optional field. Defaults to one specifying no leakage if omitted or set to empty, [ ].	lf
s.iter	Total number of iterations in the adaptive filter run. Although you can set this in s, you should not. Consider it a read-only value.	

`[y,e] = adaptse(x,d,s)` also returns the prediction error `e`. Ultimately this shows you how well the filter adapted to the desired signal and input data. Or how well `y` approximates `d`.

`[y,e,s] = adaptse(x,d,s)` returns the updated structure S.

`adaptse` can be called for a block of data, when `x` and `d` are vectors, or in a 'sample by sample mode' using a for loop:

```
for n = 1:length(x)
[y(n),e(n),s] = adaptse(x(n),d(n),s);
% The fields of S may be modified here.
end
```

In lieu of assigning the structure fields manually, function `initse` can be called to populate the structure `s`.

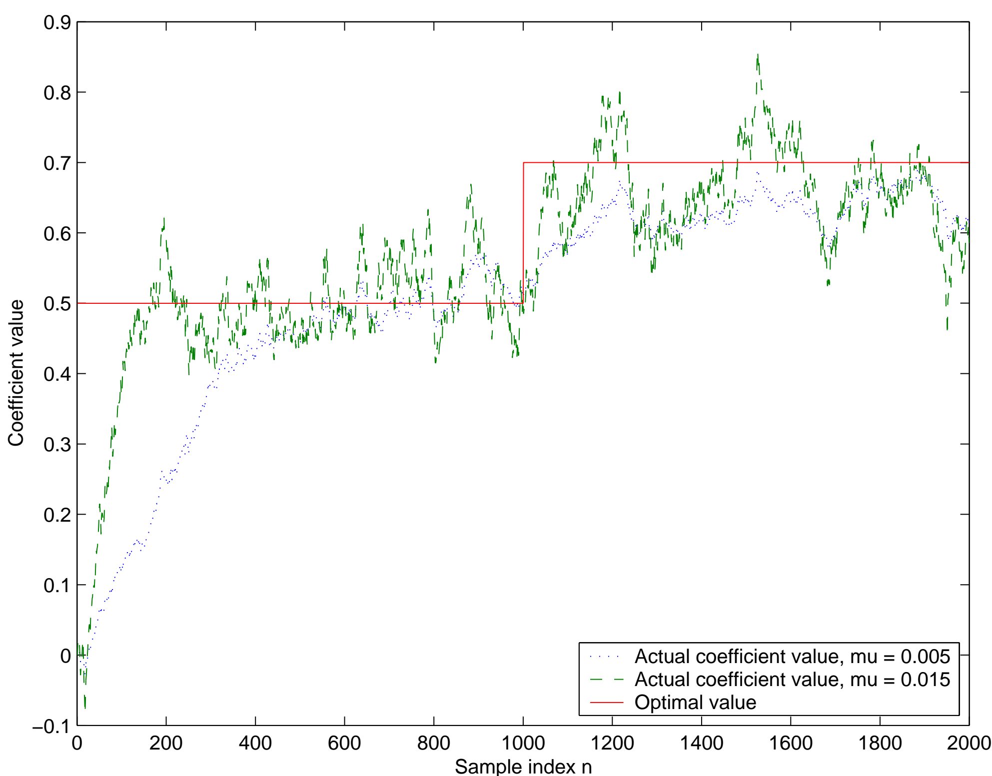
Examples

To demonstrate the effects of using different step sizes, we use adaptive linear prediction with two different step sizes to identify an FIR filter whose coefficients change with time. This example generates two sets of filter coefficients to compare to the ideal coefficients.

```
u = randn(1,2000); % Input
y1 = filter(1,[1,-.5],u(1:1000));
y2 = filter(1,[1,-.7],u(1001:2000));
```

```
y = [y1,y2]; % Construct a filter with non-stationary
coefficients
mu1 = 0.005; mu2 = 0.015; w0 = zeros(1,2);
s1 = initse(w0,mu1); s2 = initse(w0,mu2);
for n = 1:length(y),
   [z1(n),e1(n),s1] = adaptse(u(n),y(n),s1);
   [z2(n),e2(n),s2] = adaptse(u(n),y(n),s2);
   coeffs1(n,:) = s1.coeffs; coeffs2(n,:) = s2.coeffs;
end
plot([coeffs1(:,2),coeffs2(:,2),[.5*ones(1000,1);...
0.7*ones(1000,1)]])
legend('Actual coefficient value, mu = 0.005',...
'Actual coefficient value, mu = 0.015','Optimal value',4);
xlabel('Sample index n'),ylabel('Coefficient value');
```

In the figure, the coefficients generated using `mu=0.005` converge more closely to the ideal; the `mu=0.015` case coefficients converge more quickly but less closely. In the end, the resulting coefficients for both cases are quite similar if not the same. When the FIR filter coefficients change at sample index = 1000, the adapting SELMS algorithm changes to match the new filter.



Algorithm

In vector form, the SELMS algorithm is

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \mu\,\mathrm{sgn}[e(k)][\mathbf{x}(k)]\ ,\quad sgn[e(k)] = \begin{cases} 1, & e(k) > 0 \\ 0, & e(k) = 0 \\ -1, & e(k) < 0 \end{cases}$$

with vector **w** containing the weights applied to the filter coefficients and vector **x** containing the input data. *e(k)* (equal to desired signal - filtered signal) is the error at time k and is the quantity the SELMS algorithm seeks to

minimize. μ (`mu`) is the step size. As you specify `mu` smaller, the correction to the filter weights gets smaller for each sample and the SELMS error falls more slowly. Larger `mu` 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 good convergence rate and stability, select `mu` within the following practical bounds:

$$0 < \mu < \frac{1}{N\{InputSignalPower\}}$$

where N is the number of samples in the signal. Also, define `mu` as a power of two for efficient computation.

See Also

`initse`, `adaptsd`, `adaptss`, `adaptkalman`, `adaptlms`, `adaptnlms`, `adaptrls`

References

Hayes, M.H., *Statistical Digital Signal Processing and Modeling,* John Wiley and Sons, 1996

Diniz, P, "Adaptive Filtering. Algorithms and Practical Implementation," Kluwer Academic Publishers, Bostom, 1997.

Purpose Sign-sign FIR adaptive filter.

Syntax

```
y = adaptss(x,d,s)
[y,e] = adaptss(x,d,s)
[y,e,s] = adaptss(x,d,s)
```

Description `y = adaptss(x,d,s)` applies a sign-sign FIR adaptive filter to the data vector `x` and the desired signal `d`. Note that for this algorithm both `x` and `d` must be real. The filtered data is returned in `y`. `s` is a structure that contains the adaptive filter information.

Structure Element	Element Contents Description	initss Element
`s.coeffs`	SSLMS FIR filter coefficients. Initialize with the initial coefficients for the FIR filter prior to adapting. You need [adapting filter order + 1] entries in `s.coeffs`. Updated filter coefficients are returned in `s.coeffs` when you use `s` as an output argument.	`w0`
`s.step`	Sets the SSLMS algorithm step size. Determines both how quickly and how closely the adaptive filter adapts to the filter solution.	`mu`
`s.states`	Returns the states of the FIR filter after adaptation. This is an optional element. If omitted, it defaults to a zero vector of length equal to the filter order. When you use `adaptss` in a loop structure, use this element to specify the initial filter states for the adapting FIR filter.	`zi`

Structure Element	Element Contents Description	initss Element
`s.leakage`	Specifies the SSLMS leakage parameter. Allows you to implement a leaky SSLMS algorithm. Including a leakage factor can improve the results of the algorithm by forcing the SSLMS algorithm to continue to adapt even after it reaches a minimum value. This is an optional field. Defaults to one if omitted (specifying no leakage) or set to empty, [ ].	`lf`
`s.iter`	Total number of iterations in the adaptive filter run. Although you can set this in `s`, you should not. Consider it a read-only value.	

`[y,e] = adaptss(x,d,s)` also returns the prediction error `e`. Ultimately this shows you how well the filter adapted to the desired signal and input data. Or how well `y` approximates `d`.

`[y,e,s] = adaptss(x,d,s)` returns the updated structure `s`.

`adaptss` can be called for a block of data, when `x` and `d` are vectors, or in "sample by sample mode" using a For-loop:

```
for n = 1:length(x)
[y(n),e(n),s] = adaptss(x(n),d(n),s);
% The fields of S may be modified here.
end
```

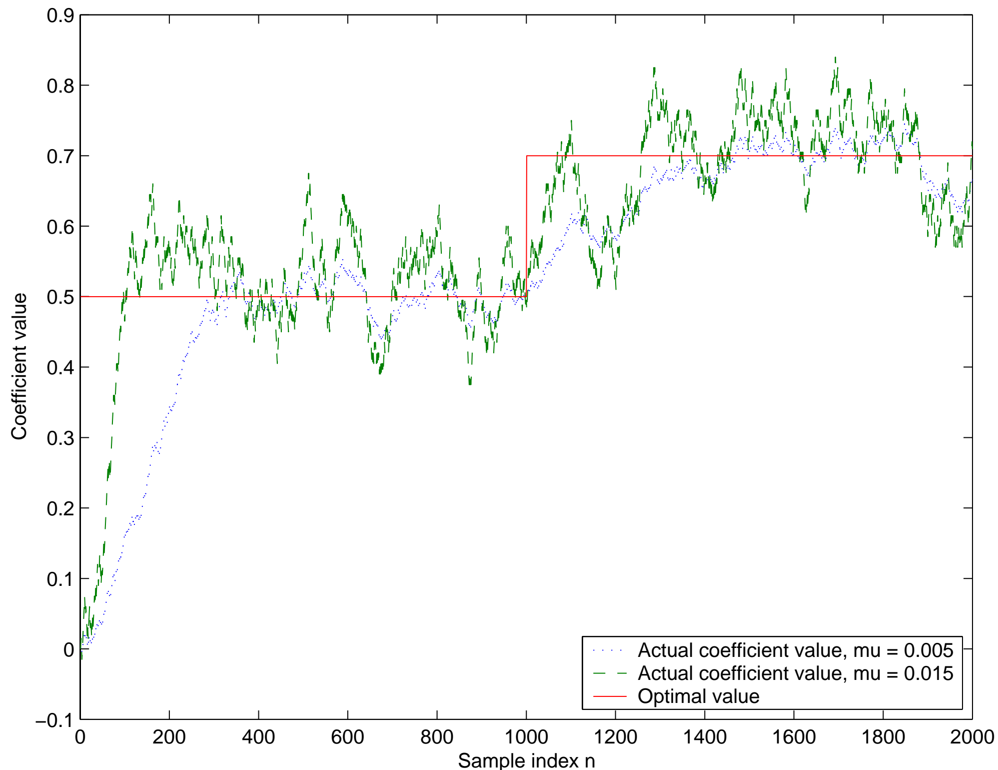
In lieu of assigning the structure fields manually, function `initss` can be called to populate the structure `s`.

Examples

To demonstrate the effects of using different step sizes, we use adaptive linear prediction with two different step sizes to identify an FIR filter whose coefficients change with time. This example generates two sets of filter coefficients to compare to the ideal coefficients.

```
u = randn(1,2000); % Input
y1 = filter(1,[1,-.5],u(1:1000));
y2 = filter(1,[1,-.7],u(1001:2000));
```

```
y = [y1,y2]; % Construct a filter with non-stationary
coefficients
mu1 = 0.005; mu2 = 0.015; w0 = zeros(1,2);
s1 = initss(w0,mu1); s2 = initss(w0,mu2);
for n = 1:length(y),
   [z1(n),e1(n),s1] = adaptss(u(n),y(n),s1);
   [z2(n),e2(n),s2] = adaptss(u(n),y(n),s2);
   coeffs1(n,:) = s1.coeffs; coeffs2(n,:) = s2.coeffs;
end
plot([coeffs1(:,2),coeffs2(:,2),[.5*ones(1000,1);...
0.7*ones(1000,1)]])
legend('Actual coefficient value, mu = 0.005',...
'Actual coefficient value, mu = 0.015','Optimal value',4);
xlabel('Sample index n'),ylabel('Coefficient value');
```



In the figure, the coefficients generated using `mu=0.005` converge more closely to the ideal; the `mu=0.015` case coefficients converge more quickly but less closely. In the end, the resulting coefficients for both cases are quite similar if not the same. When the FIR filter coefficients change at sample index = 1000, the adapting SSLMS algorithm changes to match the new filter.

Algorithm

In vector form, the SSLMS algorithm is

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \mu \operatorname{sgn}[e(k)] \operatorname{sgn}[\mathbf{x}(k)] \quad ,\operatorname{sgn}[z(k)] = \begin{cases} 1, & z(k) > 0 \\ 0, & z(k) = 0 \\ -1, & z(k) < 0 \end{cases}$$

where, for convenience,

$$z(k) = [e(k)] \operatorname{sgn}[\mathbf{x}(k)]$$

Vector **w** contains the weights applied to the filter coefficients and vector **x** contains the input data. *e(k)* (= desired signal - filtered signal) is the error at time *k* and is the quantity the SSLMS algorithm seeks to minimize. μ (`mu`) is the step size. As you specify `mu` smaller, the correction to the filter weights gets smaller for each sample and the SSLMS error falls more slowly. Larger `mu` 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 good convergence rate and stability, select `mu` within the following practical bounds:

$$0 < \mu < \frac{1}{N\{InputSignalPower\}}$$

where N is the number of samples in the signal. Also, define `mu` as a power of two for efficient computation.

See Also

`initss`, `adaptsd`, `adaptse`, `adaptkalman`, `adaptlms`, `adaptnlms`, `adaptrls`

References

Hayes, M.H., *Statistical Digital Signal Processing and Modeling,* John Wiley and Sons, 1996

Diniz, P, “Adaptive Filtering. Algorithms and Practical Implementation,” Kluwer Academic Publishers, Bostom, 1997.

Purpose

Return an allpass filter for complex bandpass transformation

Syntax

```
[AllpassNum,AllpassDen] = allpassbpc2bpc(Wo,Wt)
```

Description

`[AllpassNum,AllpassDen] = allpassbpc2bpc(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the first-order allpass mapping filter for performing a complex bandpass to complex bandpass frequency transformation. This transformation effectively places two features of an original filter, located at frequencies W_{o1} and W_{o2}, at the required target frequency locations W_{t1} and W_{t2}. It is assumed that W_{t2} is greater than W_{t1}. In most of the cases the features selected for the transformation are the band edges of the filter passbands. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

This transformation can also be used for transforming other types of filters; e.g., complex notch filters or resonators can be repositioned at two distinct desired frequencies at any place around the unit circle. This is very attractive for adaptive systems.

Examples

Design the allpass filter changing the complex bandpass filter with the band edges originally at W_{o1}=0.2 and W_{o2}=0.4 to the new band edges of W_{t1}=0.3 and W_{t2}=0.6 precisely defined:

```
Wo = [0.2, 0.4];
Wt = [0.3, 0.6];
[AllpassNum, AllpassDen] = allpassbpc2bpc(Wo, Wt);
```

Calculate the frequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$:

```
plot(f/pi, angle(h)/pi, Wt, Wo, 'ro');
```

```
title('Mapping function Wo(Wt)');
xlabel('New frequency, Wt');
ylabel('Old frequency, Wo');
```

Arguments

`Wo`
Frequency values to be transformed from the prototype filter

`Wt`
Desired frequency locations in the transformed target filter

`AllpassNum`
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

See Also

`iirbpc2bpc`, `zpkbpc2bpc`

Purpose

Return an allpass filter for lowpass to bandpass transformation

Syntax

```
[AllpassNum,AllpassDen] = allpasslp2bp(Wo,Wt)
```

Description

`[AllpassNum,AllpassDen] = allpasslp2bp(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the second-order allpass mapping filter for performing a real lowpass to real bandpass frequency transformation. This transformation effectively places one feature of an original filter, located at frequency $-W_o$, at the required target frequency location, W_{t1}, and the second feature, originally at $+W_o$, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}. This transformation implements the "DC mobility," which means that the Nyquist feature stays at Nyquist, but the DC feature moves to a location dependent on the selection of W_t.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandpass transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Lowpass to bandpass transformation can also be used for transforming other types of filters; e.g., real notch filters or resonators can be doubled and repositioned at two distinct desired frequencies.

Examples

Design the allpass filter changing the lowpass filter with cutoff frequency at W_o=0.5 to the real bandpass filter with cutoff frequencies at W_{t1}=0.25 and W_{t2}=0.375:

```
Wo = 0.5;
Wt = [0.25, 0.375];
[AllpassNum, AllpassDen] = allpasslp2bp(Wo, Wt);
```

Calculate the frequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$. Please note that the transformation works in the same way for both positive and negative frequencies:

```
plot(f/pi, abs(angle(h))/pi, Wt, Wo, 'ro');
title('Mapping function Wo(Wt)');
xlabel('New frequency, Wt');
ylabel('Old frequency, Wo');
```

Arguments

`Wo`
Frequency value to be transformed from the prototype filter

`Wt`
Desired frequency locations in the transformed target filter

`AllpassNum`
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`iirlp2bp`, `zpklp2bp`

References

[1] Constantinides, A.G., “Spectral transformations for digital filters,” *IEE 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] Nowrouzian, B. and L.T. Bruton, “Closed-form solutions for discrete-time elliptic transfer functions,” *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

[4] Constantinides, A.G., “Design of bandpass digital filters,” *IEEE Proceedings*, vol. 1, pp. 1129-1231, June 1969.

Purpose Return an allpass filter for lowpass to complex bandpass transformation

Syntax

```
[AllpassNum,AllpassDen] = allpasslp2bpc(Wo,Wt)
```

Description `[AllpassNum,AllpassDen] = allpasslp2bpc(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the first-order allpass mapping filter for performing a real lowpass to complex bandpass frequency transformation. This transformation effectively places one feature of an original filter, located at frequency $-W_o$, at the required target frequency location, W_{t1}, and the second feature, originally at $+W_o$, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandpass transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Lowpass to bandpass transformation can also be used for transforming other types of filters; e.g., real notch filters or resonators can be doubled and positioned at two distinct desired frequencies at any place around the unit circle forming a pair of complex notches/resonators. This transformation can be used for designing bandpass filters for radio receivers from the high-quality prototype lowpass filter.

Examples Design the allpass filter changing the real lowpass filter with the cutoff frequency of W_o=0.5 into a complex bandpass filter with band edges of W_{t1}=0.2 and W_{t2}=0.4 precisely defined:

```
Wo = 0.5;
Wt = [0.2,0.4];
[AllpassNum, AllpassDen] = allpasslp2bpc(Wo, Wt);
```

Calculate the frequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$:

```
plot(f/pi, angle(h)/pi, Wt, Wo.*[-1,1], 'ro');
title('Mapping function Wo(Wt)');
xlabel('New frequency, Wt');
ylabel('Old frequency, Wo');
```

Arguments

`Wo`
Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

`Wt`
Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

`AllpassNum`
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

See Also

`iirlp2bpc`, `zpklp2bpc`

Purpose

Return an allpass filter for lowpass to bandstop transformation

Syntax

```
[AllpassNum,AllpassDen] = allpasslp2bs(Wo,Wt)
```

Description

`[AllpassNum,AllpassDen] = allpasslp2bs(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the second-order allpass mapping filter for performing a real lowpass to real bandstop frequency transformation. This transformation effectively places one feature of an original filter, located at frequency -W_o, at the required target frequency location, W_{t1}, and the second feature, originally at +W_o, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}. This transformation implements the "Nyquist Mobility," which means that the DC feature stays at DC, but the Nyquist feature moves to a location dependent on the selection of W_o and W_t.

Relative positions of other features of an original filter change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. After the transformation feature F_2 will precede F_1 in the target filter. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandstop transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Examples

Design the allpass filter changing the lowpass filter with cutoff frequency at W_o=0.5 to the real bandstop filter with cutoff frequencies at W_{t1}=0.25 and W_{t2}=0.375:

```
Wo = 0.5;
Wt = [0.25, 0.375];
[AllpassNum, AllpassDen] = allpasslp2bs(Wo, Wt);
```

Calculate the frequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$. Please note that the transformation works in the same way for both positive and negative frequencies:

```
plot(f/pi, abs(angle(h))/pi, Wt, Wo, 'ro');
title('Mapping function Wo(Wt)');
xlabel('New frequency, Wt');
ylabel('Old frequency, Wo');
```

Arguments

`Wo`
Frequency value to be transformed from the prototype filter

`Wt`
Desired frequency locations in the transformed target filter

`AllpassNum`
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`iirlp2bs`, `zpklp2bs`

References

[1] Constantinides, A.G., “Spectral transformations for digital filters,” *IEE 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] Nowrouzian, B. and L.T. Bruton, “Closed-form solutions for discrete-time elliptic transfer functions,” *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

[4] Constantinides, A.G., “Design of bandpass digital filters,” *IEEE Proceedings*, vol. 1, pp. 1129-1231, June 1969.

Purpose

Return an allpass filter for lowpass to complex bandstop transformation

Syntax

```
[AllpassNum,AllpassDen] = allpasslp2bsc(Wo,Wt)
```

Description

`[AllpassNum,AllpassDen] = allpasslp2bsc(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the first-order allpass mapping filter for performing a real lowpass to complex bandstop frequency transformation. This transformation effectively places one feature of an original filter, located at frequency $-W_o$, at the required target frequency location, W_{t1}, and the second feature, originally at $+W_o$, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}. Additionally the transformation swaps passbands with stopbands in the target filter.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandstop transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Lowpass to bandpass transformation can also be used for transforming other types of filters; e.g., real notch filters or resonators can be doubled and positioned at two distinct desired frequencies at any place around the unit circle forming a pair of complex notches/resonators. This transformation can be used for designing bandstop filters for band attenuation or frequency equalizers, from the high-quality prototype lowpass filter.

Examples

Design the allpass filter changing the real lowpass filter with the cutoff frequency of W_o=0.5 into a complex bandstop filter with band edges of W_{t1}=0.2 and W_{t2}=0.4 precisely defined:

```
Wo = 0.5;
Wt = [0.2,0.4];
[AllpassNum, AllpassDen] = allpasslp2bsc(Wo, Wt);
```

Calculate the frequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$:

```
plot(f/pi, angle(h)/pi, Wt, Wo.*[1,-1], 'ro');
title('Mapping function Wo(Wt)');
xlabel('New frequency, Wt');
ylabel('Old frequency, Wo');
```

Arguments

`Wo`
Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

`Wt`
Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

`AllpassNum`
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

See Also

`iirlp2bsc`, `zpklp2bsc`

Purpose

Return an allpass filter for lowpass to highpass transformation

Syntax

```
[AllpassNum,AllpassDen] = allpasslp2hp(Wo,Wt)
```

Description

`[AllpassNum,AllpassDen] = allpasslp2hp(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the first-order allpass mapping filter for performing a real lowpass to real highpass frequency transformation. This transformation effectively places one feature of an original filter, located originally at frequency, W_o, at the required target frequency location, W_t, at the same time rotating the whole frequency response by half of the sampling frequency. Result is that the DC and Nyquist features swap places.

Relative positions of other features of an original filter change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. After the transformation feature F_2 will precede F_1 in the target filter. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to highpass transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband.

Lowpass to highpass transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can change their position in a simple way by using the lowpass to highpass transformation.

Examples

Design the allpass filter changing the lowpass filter to the highpass filter with its cutoff frequency moved from W_o=0.5 to W_t=0.25:

```
Wo = 0.5;
Wt = 0.25;
[AllpassNum, AllpassDen] = allpasslp2hp(Wo, Wt);
```

Calculate the frequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$. Please note that the transformation works in the same way for both positive and negative frequencies:

```
plot(f/pi, abs(angle(h))/pi, Wt, Wo, 'ro');
title('Mapping function Wo(Wt)');
xlabel('New frequency, Wt');
ylabel('Old frequency, Wo');
```

Arguments

`Wo`
Frequency value to be transformed from the prototype filter

`Wt`
Desired frequency location in the transformed target filter

`AllpassNum`
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`iirlp2hp`, `zpklp2hp`

References

[1] Constantinides, A.G., "Spectral transformations for digital filters," *IEE 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] Nowrouzian, B. and L.T. Bruton, "Closed-form solutions for discrete-time elliptic transfer functions," *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

[4] Constantinides, A.G., "Frequency transformations for digital filters," *Electronics Letters*, vol. 3, no. 11, pp. 487-489, November 1967.

Purpose

Return an allpass filter for lowpass to lowpass transformation

Syntax

```
[AllpassNum,AllpassDen] = allpasslp2lp(Wo,Wt)
```

Description

`[AllpassNum,AllpassDen] = allpasslp2lp(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the first-order allpass mapping filter for performing a real lowpass to real lowpass frequency transformation. This transformation effectively places one feature of an original filter, located originally at frequency W_o, at the required target frequency location, W_t.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to lowpass transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband and so on.

Lowpass to lowpass transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can change their position in a simple way by applying the lowpass to lowpass transformation.

Examples

Design the allpass filter changing the lowpass filter cutoff frequency originally at W_o=0.5 to W_t=0.25:

```
Wo = 0.5;
Wt = 0.25;
[AllpassNum, AllpassDen] = allpasslp2lp(Wo, Wt);
```

Calculate the frequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$. Please note that the transformation works in the same way for both positive and negative frequencies:

```
plot(f/pi, abs(angle(h))/pi, Wt, Wo, 'ro');
```

```
title('Mapping function Wo(Wt)');
xlabel('New frequency, Wt');
ylabel('Old frequency, Wo');
```

Arguments

`Wo`
Frequency value to be transformed from the prototype filter

`Wt`
Desired frequency location in the transformed target filter

`AllpassNum`
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`iirlp2lp`, `zpklp2lp`

References

[1] Constantinides, A.G., “Spectral transformations for digital filters,” *IEE 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] Nowrouzian, B. and L.T. Bruton, “Closed-form solutions for discrete-time elliptic transfer functions,” *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

[4] Constantinides, A.G., “Frequency transformations for digital filters,” *Electronics Letters*, vol. 3, no. 11, pp. 487-489, November 1967.

Purpose

Return an allpass filter for lowpass to M-band transformation

Syntax

```
[AllpassNum,AllpassDen] = allpasslp2mb(Wo,Wt)
[AllpassNum,AllpassDen] = allpasslp2mb(Wo,Wt,Pass)
```

Description

`[AllpassNum,AllpassDen] = allpasslp2mb(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the `M`th-order allpass mapping filter for performing a real lowpass to real multipassband frequency transformation. Parameter `M` is the number of times an original feature is replicated in the target filter. This transformation effectively places one feature of an original filter, located at frequency W_o, at the required target frequency locations, W_{t1},...,W_{tM}. By default the DC feature is kept at its original location.

`[AllpassNum,AllpassDen]=allpasslp2mb(Wo,Wt,Pass)` allows you to specify an additional parameter, `Pass`, which chooses between using the “DC Mobility” and the “Nyquist Mobility”. In the first case the Nyquist feature stays at its original location and the DC feature is free to move. In the second case the DC feature is kept at an original frequency and the Nyquist feature is movable.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations without redesigning them. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

Examples

Design the allpass filter changing the real lowpass filter with the cutoff frequency of W_o=`0.5` into a real multiband filter with band edges of W_t=`[1:2:9]/10` precisely defined:

```
Wo = 0.5;
Wt = [1:2:9]/10;
[AllpassNum, AllpassDen] = allpasslp2mb(Wo, Wt);
```

Calculate the frequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$. Please note that the transformation works in the same way for both positive and negative frequencies:

```
plot(f/pi, abs(angle(h))/pi, Wt, Wo, 'ro');
title('Mapping function Wo(Wt)');
xlabel('New frequency, Wt');
ylabel('Old frequency, Wo');
```

Arguments

`Wo`
Frequency value to be transformed from the prototype filter

`Wt`
Desired frequency locations in the transformed target filter

`Pass`
Choice (`'pass'`/`'stop'`) of passband/stopband at DC, `pass` being the default

`AllpassNum`
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`iirlp2mb`, `zpklp2mb`

References

[1] Franchitti, J.C., “All-pass filter interpolation and frequency transformation problems,” MSc Thesis, Dept. of Electrical and Computer Engineering, University of Colorado, 1985.

[2] Feyh, G., J.C. Franchitti and C.T. Mullis, “All-pass filter interpolation and frequency transformation problem,” *Proceedings 20th Asilomar Conference on*

Signals, Systems and Computers, Pacific Grove, California, pp. 164-168, November 1986.

[3] Mullis, C.T. and R.A. Roberts, *Digital Signal Processing*, section 6.7, Reading, Massachussetts, Addison-Wesley, 1987.

[4] Feyh, G., W.B. Jones and C.T. Mullis, *An extension of the Schur Algorithm for frequency transformations, Linear Circuits, Systems and Signal Processing: Theory and Application*, C. J. Byrnes et al Eds, Amsterdam: Elsevier, 1988.

allpasslp2mbc

Purpose

Return an allpass filter for lowpass to complex M-band transformation

Syntax

```
[AllpassNum,AllpassDen] = allpasslp2mbc(Wo,Wt)
```

Description

`[AllpassNum,AllpassDen] = allpasslp2mbc(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the `M`th-order allpass mapping filter for performing a real lowpass to complex multipassband frequency transformation. Parameter `M` is the number of times an original feature is replicated in the target filter. This transformation effectively places one feature of an original filter, located at frequency W_o, at the required target frequency locations, W_{t1},...,W_{tM}.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations without the need to design them again. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

Examples

Design the allpass filter changing the real lowpass filter with the cutoff frequency of W_o=`0.5` into a complex multiband filter with band edges of W_t=`[-3+1:2:9]/10` precisely defined:

```
Wo = 0.5;
Wt = [-3+1:2:9]/10;
[AllpassNum, AllpassDen] = allpasslp2mb(Wo, Wt);
```

Calculate the frequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$. Please note that the transformation works in the same way for both positive and negative frequencies:

```
plot(f/pi, angle(h)/pi, Wt, Wo, 'ro');
title('Mapping function Wo(Wt)');
xlabel('New frequency, Wt');
ylabel('Old frequency, Wo');
```

Arguments

`Wo`
Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

`Wt`
Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

`AllpassNum`
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

See Also

`iirlp2mbc`, `zpklp2mbc`

allpasslp2xc

Purpose Return an allpass filter for lowpass to complex N-point transformation

Syntax

```
[AllpassNum,AllpassDen] = allpasslp2xc(Wo,Wt)
```

Description `[AllpassNum,AllpassDen] = allpasslp2xc(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the `N`th-order allpass mapping filter, where `N` is the allpass filter order, for performing a real lowpass to complex multipoint frequency transformation. Parameter `N` also specifies the number of replicas of the prototype filter created around the unit circle after the transformation. This transformation effectively places `N` features of the, original filter located at frequencies $W_{o1},\ldots,W_{oN}$, at the required target frequency locations, $W_{t1},\ldots,W_{tM}$.

Relative positions of other features of an original filter are the same in the target filter for the Nyquist mobility and are reversed for the DC mobility. For the Nyquist mobility this means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation. For DC mobility feature F_2 will precede F_1 after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones. The only condition is that the features must be selected in such a way that when creating `N` bands around the unit circle, there will be no band overlap.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

Examples Design the allpass filter moving four features of an original complex filter given in W_o to the new independent frequency locations W_t. Please note that the transformation creates `N` replicas of an original filter around the unit circle, where `N` is the order of the allpass mapping filter:

```
Wo = [-0.2, 0.3, -0.7, 0.4];
Wt = [ 0.3, 0.5,  0.7, 0.9];
```

```
[AllpassNum, AllpassDen] = allpasslp2xc(Wo, Wt);
```

Calculate the frequency responsefrequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$:

```
plot(f/pi, angle(h)/pi, Wt, Wo, 'ro');
title('Mapping function Wo(Wt)');
xlabel('New frequency, Wt');
ylabel('Old frequency, Wo');
```

Arguments

`Wo`
Frequency values to be transformed from the prototype filter

`Wt`
Desired frequency locations in the transformed target filter

`AllpassNum`
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

See Also

`iirlp2xc`, `zpklp2xc`

allpasslp2xn

Purpose

Return an allpass filter for lowpass to N-point transformation

Syntax

```
[AllpassNum,AllpassDen] = allpasslp2xn(Wo,Wt)
[AllpassNum,AllpassDen] = allpasslp2xn(Wo,Wt,Pass)
```

Description

`[AllpassNum,AllpassDen] = allpasslp2xn(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the `N`th-order allpass mapping filter, where `N` is the allpass filter order, for performing a real lowpass to real multipoint frequency transformation. Parameter `N` also specifies the number of replicas of the prototype filter created around the unit circle after the transformation. This transformation effectively places `N` features of an original filter, located at frequencies W_{o1},...,W_{oN}, at the required target frequency locations, W_{t1},...,W_{tM}. By default the DC feature is kept at its original location.

`[AllpassNum,AllpassDen]=allpasslp2xn(Wo,Wt,Pass)` allows you to specify an additional parameter, `Pass`, which chooses between using the "DC Mobility" and the "Nyquist Mobility". In the first case the Nyquist feature stays at its original location and the DC feature is free to move. In the second case the DC feature is kept at an original frequency and the Nyquist feature is movable.

Relative positions of other features of an original filter are the same in the target filter for the Nyquist mobility and are reversed for the DC mobility. For the Nyquist mobility this means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation. For DC mobility feature F_2 will precede F_1 after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones. The only condition is that the features must be selected in such a way that when creating `N` bands around the unit circle, there will be no band overlap.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations without the need of designing them again. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

Examples

Design the allpass filter moving three features of an original filter given in W_o to the new independent frequency locations W_t. Please note that the transformation creates N replicas of an original filter around the unit circle, where N is the order of the allpass mapping filter:

```
Wo = [-0.2, 0.3, -0.7];
Wt = [ 0.3, 0.5,  0.8];
[AllpassNum, AllpassDen] = allpasslp2xn(Wo, Wt);
```

Calculate the frequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$:

```
plot(f/pi, angle(h)/pi, Wt, Wo, 'ro');
title('Mapping function Wo(Wt)');
xlabel('New frequency, Wt');
ylabel('Old frequency, Wo');
```

Arguments

`Wo`
Frequency values to be transformed from the prototype filter

`Wt`
Desired frequency locations in the transformed target filter

`Pass`
Choice (`'pass'`/`'stop'`) of passband/stopband at DC, `pass` being the default

`AllpassNum`
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`iirlp2xn`, `zpklp2xn`

References

[1] Cain, G.D. , A. Krukowski and I. Kale, "High Order Transformations for Flexible IIR Filter Design," *VII European Signal Processing Conference*

(EUSIPCO'94), vol. 3, pp. 1582-1585, Edinburgh, United Kingdom, September 1994.

[2] 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.

Purpose

Return an allpass filter for integer upsample transformation

Syntax

```
[AllpassNum,AllpassDen] = allpassrateup(N)
```

Description

`[AllpassNum,AllpassDen] = allpassrateup(N)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the `N`th-order allpass mapping filter for performing the rateup frequency transformation, which creates `N` equal replicas of the prototype filter frequency response.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Examples

Design the allpass filter creating the effect of upsampling the digital filter four times:

```
N = 4;
```

Choose any feature from an original filter, say at W_o=0.2:

```
Wo = 0.2;
Wt = Wo/N + 2*[0:N-1]/N;
[AllpassNum, AllpassDen] = allpassrateup(N);
```

Calculate the frequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$:

```
plot(f/pi, angle(h)/pi, Wt, Wo, 'ro');
title('Mapping function Wo(Wt)');
xlabel('New frequency, Wt');
ylabel('Old frequency, Wo');
```

Arguments

`N`
Frequency replication ratio (upsampling ratio)

`AllpassNum`
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

See Also

`iirrateup`, `zpkrateup`

Purpose

Return an allpass filter for real shift transformation

Syntax

```
[AllpassNum,AllpassDen] = allpassshift(Wo,Wt)
```

Description

`[AllpassNum,AllpassDen] = allpassshift(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the second-order allpass mapping filter for performing a real frequency shift transformation. This transformation places one selected feature of an original filter, located at frequency W_o, at the required target frequency location, W_t. This transformation implements the "DC mobility," which means that the Nyquist feature stays at Nyquist, but the DC feature moves to a location dependent on the selection of W_o and W_t.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the real shift transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be moved to a different frequency by applying a shift transformation. In such a way you can avoid designing the filter from the beginning.

Examples

Design the allpass filter precisely shifting one feature of the lowpass filter originally at W_o=0.5 to the new frequencies of W_t=0.25:

```
Wo = 0.5;
Wt = 0.25;
[AllpassNum, AllpassDen] = allpassshift(Wo, Wt);
```

Calculate the frequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$. Please note that the transformation works in the same way for both positive and negative frequencies:

```
plot(f/pi, abs(angle(h))/pi, Wt, Wo, 'ro');
title('Mapping function Wo(Wt)');
xlabel('New frequency, Wt');
ylabel('Old frequency, Wo');
```

Arguments

`Wo`
Frequency value to be transformed from the prototype filter

`Wt`
Desired frequency location in the transformed target filter

`AllpassNum`
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`iirshift`, `zpkshift`

Purpose

Return an allpass filter for complex shift transformation

Syntax

```
[AllpassNum,AllpassDen] = allpassshiftc(Wo,Wt)
```

Description

`[AllpassNum,AllpassDen] = allpassshiftc(Wo,Wt)` returns the numerator, `AllpassNum`, and denominator, `AllpassDen`, vectors of the allpass mapping filter for performing a complex frequency shift of the frequency response of the digital filter by an arbitrary amount.

`[AllpassNum,AllpassDen]=allpassshiftc(0,0.5)` calculates the allpass filter for doing the Hilbert transformation, i.e. a 90 degree counterclockwise rotation of an original filter in the frequency domain.

`[AllpassNum,AllpassDen]=allpassshiftc(0,-0.5)` calculates the allpass filter for doing an inverse Hilbert transformation, i.e. a 90 degree clockwise rotation of an original filter in the frequency domain.

Examples

Design the allpass filter precisely rotating the whole filter by the amount defined by the location of the selected feature from an original filter, W_o=0.5, and its required position in the target filter, W_t=0.25:

```
Wo = 0.5;
Wt = 0.25;
[AllpassNum, AllpassDen] = allpassshiftc(Wo, Wt);
```

Calculate the frequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$:

```
plot(f/pi, angle(h)/pi, Wt, Wo, 'ro');
title('Mapping function Wo(Wt)');
xlabel('New frequency, Wt');
ylabel('Old frequency, Wo');
```

Arguments

`Wo`

Frequency value to be transformed from the prototype filter

`Wt`
Desired frequency location in the transformed target filter

`AllpassNum`
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

See Also

`iirshiftc`, `zpkshiftc`

References

[1] Oppenheim, A.V., R.W. Schafer and J.R. Buck, *Discrete-Time Signal Processing*, Prentice-Hall International Inc., 1989.

[2] Dutta-Roy, S.C. and B. Kumar, "On digital differentiators, Hilbert transformers, and half-band low-pass filters," *IEEE Transactions on Education*, vol. 32, pp. 314-318, August 1989.

Purpose Convert a two's complement binary string to a number

Syntax

```
y = bin2num(q,b)
```

Description

`y = bin2num(q,b)` uses the properties of quantizer `q` to convert binary string `b` to numeric array `y`. When `b` is a cell array containing binary strings, `y` will be a cell array of the same dimension containing numeric arrays. The fixed-point binary representation is two's complement. The floating-point binary representation is in IEEE Standard 754 style.

`bin2num` and `num2bin` are inverses of one another. Note that `num2bin` always returns columnwise.

Examples

Create a quantizer and an array of numeric strings. Convert the numeric strings to binary strings, then use `bin2num` to convert them back to numeric strings.

```
q=quantizer([4 3]);
[a,b]=range(q);
x=(b:-eps(q):a)';
b = num2bin(q,x)

b =

0111
0110
0101
0100
0011
0010
0001
0000
1111
1110
1101
1100
1011
1010
1001
1000
```

bin2num performs the inverse operation of num2bin.

```
y=bin2num(q,b)

y =

    0.8750
    0.7500
    0.6250
    0.5000
    0.3750
    0.2500
    0.1250
         0
   -0.1250
   -0.2500
   -0.3750
   -0.5000
   -0.6250
   -0.7500
   -0.8750
   -1.0000
```

See Also

num2bin

Purpose

Convert coupled allpass filter form to transfer function forms

Syntax

```
[b,a] = ca2tf(d1,d2)
[b,a] = ca2tf(d1,d2,beta)
[b,a,bp] = ca2tf(d1,d2)
[b,a,bp] = ca2tf(d1,d2,beta)
```

Description

`[b,a]=ca2tf(d1,d2)` returns the vector of coefficients `b` and the vector of coefficients `a` corresponding to the numerator and the denominator of the transfer function

$$H(z) = B(z)/A(z) = \frac{1}{2}[H1(z) + H2(z)]$$

`d1` and `d2` are real vectors corresponding to the denominators of the allpass filters `H1(z)` and `H2(z)`.

`[b,a]=ca2tf(d1,d2,beta)` where `d1`, `d2` and `beta` are complex, returns the vector of coefficients `b` and the vector of coefficients `a` corresponding to the numerator and the denominator of the transfer function

$$H(z) = B(z)/A(z) = \frac{1}{2}[-(\bar{\beta}) \bullet H1(z) + \beta \bullet H2(z)]$$

`[b,a,bp]=ca2tf(d1,d2)`, where `d1` and `d2` are real, returns the vector `bp` of real coefficients corresponding to the numerator of the power complementary filter G(z)

$$G(z) = Bp(z)/A(z) = \frac{1}{2}[H1(z) - H2(z)]$$

`[b,a,bp]=ca2tf(d1,d2,beta)`, where `d1`, `d2` and `beta` are complex, returns the vector of coefficients `bp` of real or complex coefficients that correspond to the numerator of the power complementary filter G(z)

$$G(z) = Bp(z)/A(z) = \frac{1}{2j}[-(\bar{\beta}) \bullet H1(z) + \beta \bullet H2(z)]$$

Examples

Create a filter, convert the filter to coupled allpass form, and convert the result back to the original structure (create the power complementary filter as well).

```
[b,a]=cheby1(10,.5,.4);
[d1,d2,beta]=tf2ca(b,a);           % tf2ca returns the
                                   % denominators of the
                                   % allpasses.
[num,den,numpc]=ca2tf(d1,d2,beta); % Reconstruct the original
                                   % filter plus the power
                                   % complementary one.
[h,w,s]=freqz(num,den);
hpc = freqz(numpc,den);
s.plot = 'mag';
s.yunits = 'sq';
freqzplot([h hpc],w,s);            % Plot the mag response of the
                                   % original filter and the
                                   % power complementary one.
```

See Also

cl2tf, iirpowcomp, tf2ca, tf2cl

Purpose

Convert cell array to second-order-section matrix

Syntax

```
s = cell2sos(c)
[s,g] = cell2sos(c)
```

Description

`s = cell2sos(c)` converts cell array `c` of the form

```
c = { {b1,a1}, {b2,a2}, ... {bi,ai} }
```

where each numerator vector `bi` and denominator vector `ai` contains the coefficients of a linear or quadratic polynomial, to an L-by-6 second-order section matrix `s` of the form

```
s = [b1 a1
       b2 a2
        ...
       bi ai]
```

When `cell2sos` encounters linear sections, it zero-pads the sections on the right.

`[s,g] = cell2sos(c)` when the first element of `c` is a pair of scalars, forms the scalar gain `g` with the first pair and uses the remaining elements of `c` to build the `s` matrix.

Examples

```
c = {{[0.0181 0.0181],[1.0000 -0.5095]},{[1 2 1],[1 -1.2505
0.5457]}}

c =
{1x2 cell}    {1x2 cell}

s = cell2sos(c)
s =

    0.0181    0.0181         0    1.0000   -0.5095         0
    1.0000    2.0000    1.0000    1.0000   -1.2505    0.5457
```

See Also

`sos2ss`, `sos2tf`, `sos2zp`, `ss2sos`, `tf2sos`, `zp2sos`

Purpose Use a cascaded integrator-comb (CIC) decimation filter to decrease the sampling rate for a signal

Syntax

```
y = cicdecimate(m,n,r,x,q)
```

Description `y = cicdecimate(m,n,r,x,q)` filters the data in input vector `x`, applying a decimation factor (or sample rate reduction) `r` to the signal. `r` must be a positive integer. For example, when `r = 5`, the decimation filter reduces the signal length to one-fifth of the original length.

Input arguments `m` and `n` define the number of integrator and comb stages (`n`) and the number of differential delays (`m`) in the CIC decimation filter. Although `m` can be any positive integer, most often it is 1 or 2. Each integrator stage in the CIC filter comprises a single-pole infinite impulse response (IIR) filter with a unity feedback coefficient.

`q` represents a quantizer operating in signed fixed-point mode, as specified by the `fixed` keyword argument to the function `quantizer`.

`cicdecimate` uses the `int32` data type for all arithmetic operations it performs while decimating the input signal. Limiting the data type to int32 means when the most significant bit (MSB) at the filter output is greater than 32, the overall sum can overflow causing the result to be wrong. When the MSB exceeds 32, `cicdecimate` generates a warning message that the MSB is too large.

With reference to the high sampling rate, the transfer function for the composite CIC filter is

$$H(z) = \frac{(1-z^{-rm})^n}{(1-z^{-1})^n} = \left(\frac{1-z^{-rm}}{1-z^{-1}}\right)^n = (1+z^{-1}+z^{-2}+\ldots+z^{-rm+1})^n$$

Design Considerations

When you design your CIC decimation filter, remember the following general points:

- The filter output spectrum has nulls at $\omega = k * 2\pi/\mathrm{rm}$ radians, $k = 1,2,3\ldots$.
- Aliasing and imaging occur in the vicinity of the nulls.
- n, the number of stages in the filter, determines the passband attentuation. Increasing `n` improves the filter ability to reject aliasing and imaging, but it

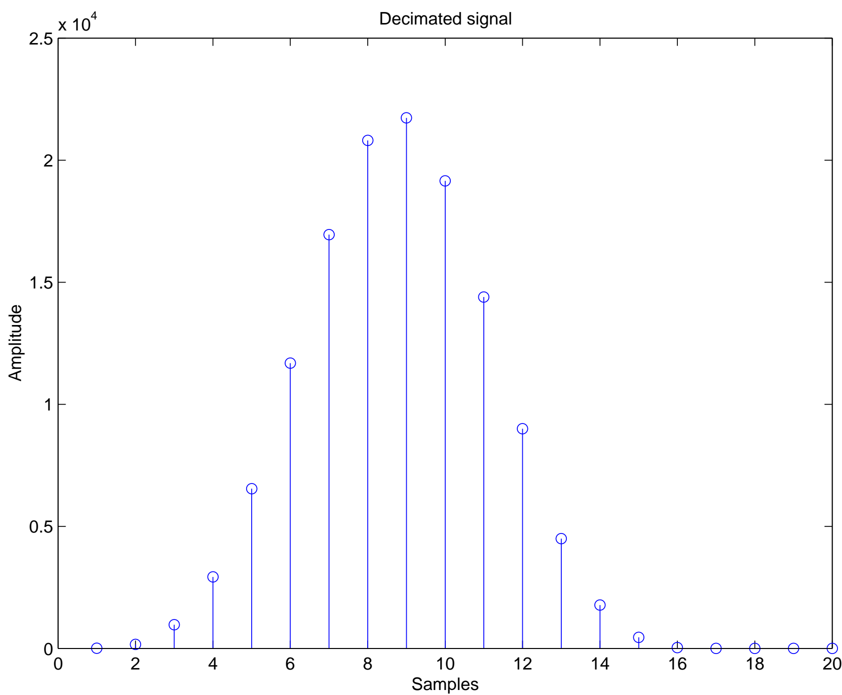
also increases the droop (or rolloff) in the filter passband. Using an appropriate FIR filter in series after the CIC decimation filter can help you compensate for the induced droop.

- The DC gain for the filter is a function of the decimation factor. Raising the decimation factor increases the DC gain.

Examples

This example applies a decimation factor r equal to 8 to a 160-point impulse signal. The signal output from the filter has 160/r, or 20, points or samples. Choosing 10 bits for the quantizer wordlength represents a fairly common setting for analog to digital converters. The plot shown after the code presents the stem plot of the decimated signal, with 20 samples remaining after decimation:

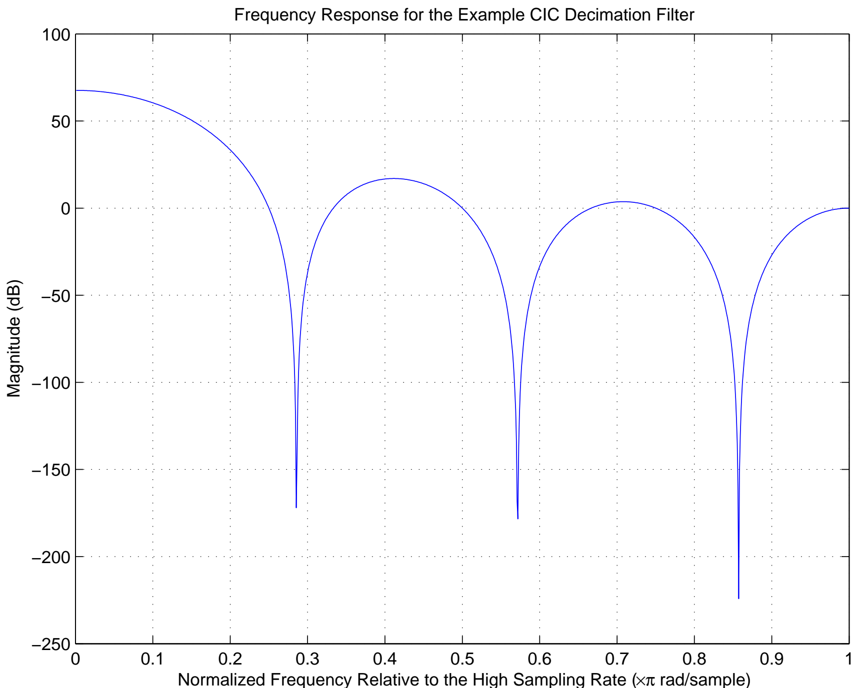
```
m = 4;  % Differential delays in the filter
n = 4;  % filter stages
r = 8   % decimation factor
x = zeros(160,1); x(1) = 1;    % Create a 160-point impulse signal
q = quantizer([10 0],'fixed'); % Define the quantizer
y = cicdecimate(m,n,r,x,q)
stem(y)                        % Plot the output as a stem plot
xlabel('Samples'); ylabel('Amplitude');
title('Decimated Signal');
```



This example demonstrates one way to compute the frequency response, using a 4-stage decimation filter with the decimation factor set to 7:

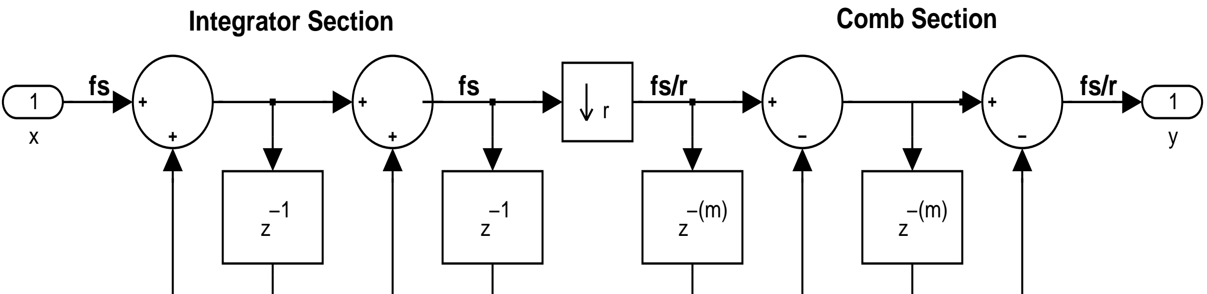
```
m = 1;n = 4; r = 7; % Define the filter parameters
w = linspace(0,pi,1024); % Set the frequency in radians
% Calculate the frequency response of the filter
h = exp(i*n*w/2*(1-r*m)).*(sin(r*m*w/2)./sin(w/2)).^n;
plot(w/pi,20*log10(abs(h))); grid on;
xlabel('Normalized Frequency Relative to the High Sampling...
Rate (\times\pi rad/sample)');
ylabel('Magnitude (dB)');
title('Frequency Response for the Example CIC...
Decimation Filter');
```

Here's the frequency response plot for the filter. For details about the transfer function used to produce the frequency response, refer to [1] in the References section.



Algorithm

To show how the CIC decimation filter is constructed, the following figure presents a block diagram of the filter structure for a two-stage CIC decimation filter (n = 2). *fs* is the high sampling rate, the input to the decimation process.



See Also `cicinterpolate`, `filterdesign`

References

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

[2] Meyer-Baese, Uwe, “Hogenauer CIC Filters,” in *Digital Signal Processing with Field Programmable Gate Arrays*, Springer, 2001, pp. 155-172

Purpose

Use a cascaded integrator-comb (CIC) interpolation filter to increase the sampling rate for a signal

Syntax

```
y = cicinterpolate(m,n,r,x,q)
```

Description

`y = cicinterpolate(m,n,r,x,q)` filters the data in input vector `x`, applying an interpolation factor (or sample rate increase) `r` to the signal. For example, when `r` = `5`, the CIC interpolation filter increases the signal length to five times the original length.

Input arguments `m` and `n` define the number of integrator and comb stages (`n`) and the number of differential delays (`m`) in the CIC interpolation filter. Although `m` can be any positive integer, most often it is 1 or 2. Each integrator stage comprises a single-pole infinite impulse response (IIR) filter with a unity feedback coefficient.

`q` represents a quantizer operating in signed fixed-point mode, as specified by the `fixed` keyword argument to the function `quantizer`.

`cicinterpolate` uses the `int32` data type for all arithmetic operations it performs while interpolating the input signal. Limiting the data type to `int32` means when the most significant bit (MSB) at the filter output is greater than 32, the overall sum can overflow causing the result to be wrong. When the MSB exceeds 32, `cicinterpolate` generates a warning message that the MSB is too large.

With reference to the high, sampling rate, the transfer function for the composite CIC filter is

$$\frac{(1-z^{-rm})^n}{(1-z^{-1})^n} = \left(\frac{1-z^{-rm}}{1-z^{-1}}\right)^n = (1+z^{-1}+z^{-2}+\ldots+z^{-r}$$

Design Considerations

When you design your CIC interpolation filter, remember the following general points:

- The filter output spectrum has nulls at $\omega = k * 2\pi/\mathrm{rm}$ radians, k = 1,2,3….
- Aliasing and imaging occur in the vicinity of the nulls.

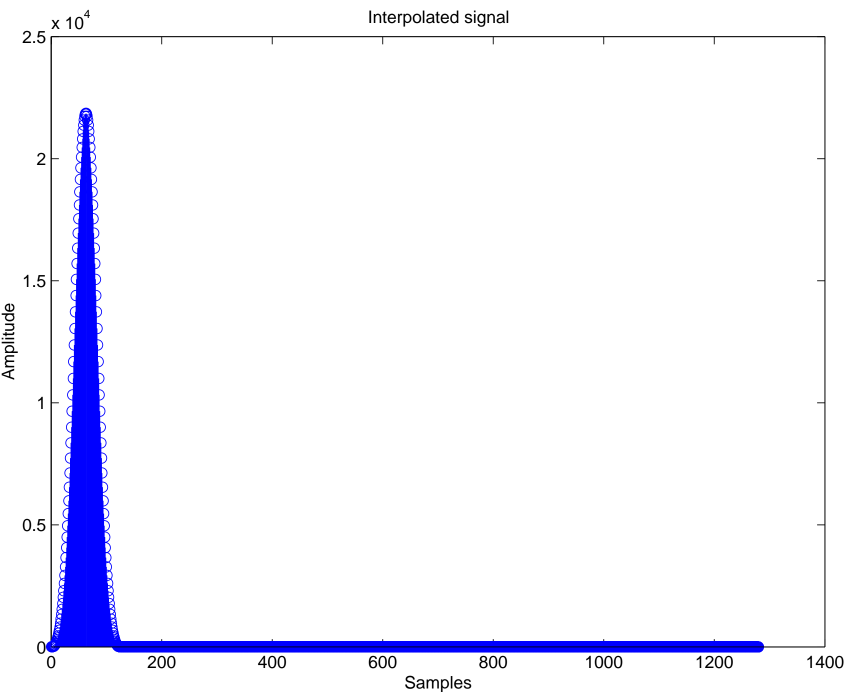
- n, the number of stages in the filter, determines the passband attentuation. Increasing n improves the filter ability to reject aliasing and imaging, but it also increases the droop or rolloff in the filter passband. Using an appropriate FIR filter in series after the CIC interpolation filter can help you compensate for the induced droop.
- The DC gain for the filter is a function of the interpolation factor. Raising the interpolation factor increases the DC gain.

Examples

This example applies an interpolation factor r equal to 8 to a 160 point impulse signal. The signal output from the filter has 160*r, or 1280, points or samples. Choosing 10 bits for the quantizer wordlength represents a fairly common setting for analog to digital converters:

```
m = 4;  % Differential delays in the filter
n = 4;  % Filter stages
r = 8   % Interpolation factor
x = zeros(160,1); x(1) = 1;   % Create a 160-point impulse signal
q = quantizer([10 0],'fixed'); % Define the quantizer
y = cicinterpole(m,n,r,x,q)
stem(y)                        % Plot the output as a stem plot
xlabel('Samples'); ylabel('Amplitude');
title('Interpolated Signal');
```

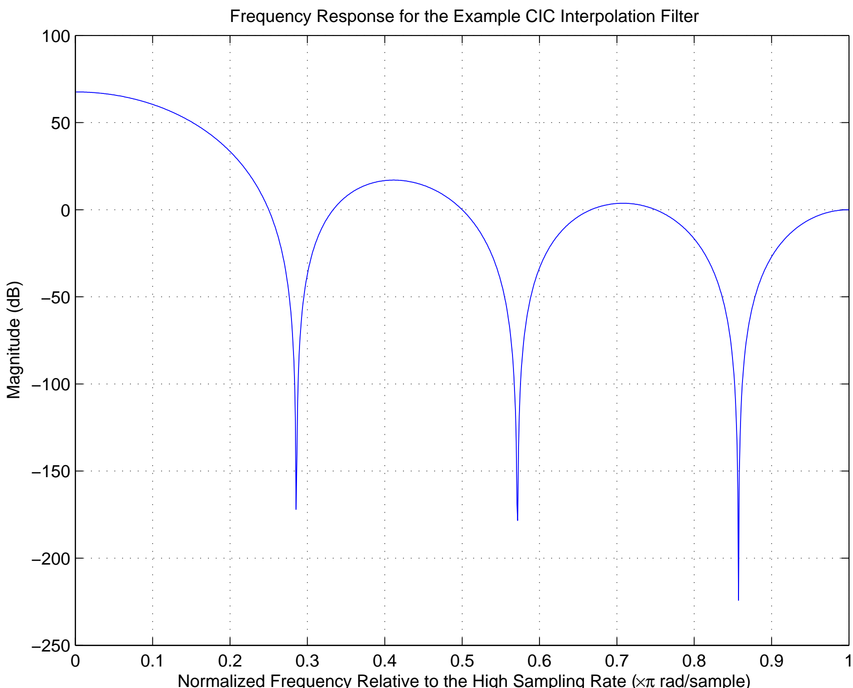
After interpolating the signal, y contains 1280 samples, as you see in the figure shown.



Computing and plotting the frequency response of an interpolating filter can be valuable. This example demonstrates one way to compute and display the frequency response, using a 4-stage CIC interpolation filter with the interpolation factor set to 7:

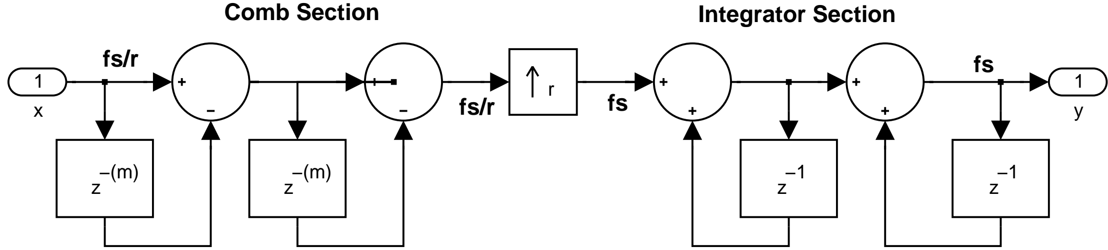
```
m = 1;n = 4; r = 7; % Define the filter parameters
w = linspace(0,pi,1024); % Set the frequency in radians
% Calculate the frequency response of the filter
h = exp(i*n*w/2*(1-r*m)).*(sin(r*m*w/2)./sin(w/2)).^n;
plot(w/pi,20*log10(abs(h))); grid on;
xlabel('Normalized Frequency Relative to the High Sampling'...
'Rate (\times\pi rad/sample)');
ylabel('Magnitude (dB)');
title('Frequency Response for the Example CIC'...
'Interpolation Filter');
```

As with the CIC decimation filter, the figure shows the clear comb nature of the interpolation filter.For details about the transfer function used to produce this frequency response plot, refer to [2] the References section.



Algorithm

To show how the CIC interpolation filter is constructed, the following figure provides a block diagram of the filter structure for a two-stage CIC interpolation filter (n = 2). *fs* is the high sampling rate, the output from the interpolation process.



See Also

`cicdecimate`, `filterdesign`

References

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

[2] Meyer-Baese, Uwe, "Hogenauer CIC Filters," in *Digital Signal Processing with Field Programmable Gate Arrays*, Springer, 2001, pp. 155-172

Purpose

Convert coupled allpass lattice to transfer function form

Syntax

```
[b,a] = cl2tf(k1,k2)
[b,a] = cl2tf(k1,k2,beta)
[b,a,bp] = cl2tf(k1,k2)
[b,a,bp] = cl2tf(k1,k2,beta)
```

Description

`[b,a] = cl2tf(k1,k2)` returns the numerator and denominator vectors of coefficients `b` and `a` corresponding to the transfer function

$$H(z) = B(z)/A(z) = \frac{1}{2}[H1(z) + H2(z)]$$

where *H1(z)* and *H2(z)* are the transfer functions of the allpass filters determined by `k1` and `k2`, and `k1` and `k2` are real vectors of reflection coefficients corresponding to allpass lattice structures.

`[b,a] = cl2tf(k1,k2,beta)` where `k1`, `k2` and `beta` are complex, returns the numerator and denominator vectors of coefficients `b` and `a` corresponding to the transfer function

$$H(z) = B(z)/A(z) = \frac{1}{2}[-(\bar{\beta}) \bullet H1(z) + \beta \bullet H2(z)]$$

`[b,a,bp] = cl2tf(k1,k2)` where `k1` and `k2` are real, returns the vector `bp` of real coefficients corresponding to the numerator of the power complementary filter *G(z)*

$$G(z) = Bp(z)/A(z) = \frac{1}{2}[H1(z) - H2(z)]$$

`[b,a,bp] = cl2tf(k1,k2,beta)` where `k1`, `k2` and `beta` are complex, returns the vector of coefficients `bp` of possibly complex coefficients corresponding to the numerator of the power complementary filter *G(z)*

$$G(z) = Bp(z)/A(z) = \frac{1}{2j}[-(\bar{\beta}) \bullet H1(z) + \beta \bullet H2(z)]$$

Examples

```
[b,a]=cheby1(10,.5,.4);
[k1,k2,beta]=tf2cl(b,a); %TF2CL returns the reflection coeffs
```

```
% Reconstruct the original filter
% plus the power complementary one.
[num,den,numpc]=cl2tf(k1,k2,beta);
[h,w,s1]=freqz(num,den);
hpc = freqz(numpc,den);
s.plot = 'mag';
s.yunits = 'sq';
% Plot the mag response of the original filter and the power
% complementary one.
freqzplot([h hpc],w,s1);
```

See Also

`tf2cl`, `tf2ca`, `ca2tf`, `tf2latc`, `latc2tf`, `iirpowcomp`

coeread

Purpose Read a XILINX CORE Generator™ coefficient (.COE) file

Syntax

```
hq = coeread('filename')
```

Description `hq = coeread(filename)` extracts the Distributed Arithmetic FIR filter coefficients defined in the XILINX CORE Generator .COE file specified by `filename`. It returns the quantized filter object `hq`. If you do not provide the file type extension `.coe` with the `filename`, the function assumes the `.coe` extension.

See Also `coewrite`

Purpose

Write a Xilinx CORE Generator™ coefficient (.COE) file

Syntax

```
coewrite(h)
coewrite(hq,radix)
coewrite(...,filename)
```

Description

`coewrite(hq)` writes a XILINX Distributed Arithmetic FIR filter coefficient .COE file which can be loaded into the XILINX CORE Generator. The coefficients are extracted from the quantized filter object `hq`. Your quantized filter must be a direct-form FIR structure with one section. You cannot export nonquantized filters as `.coe` files, nor multiple-section filters. To enable you to provide a name for the file, `coewrite` displays a dialog where you fill in the file name. If you do not specify the name of the output file, the default file name is `untitled.coe`.

`coewrite(hq,radix)` indicates the radix (number base) used to specify the FIR filter coefficients. Valid `radix` values are `2` for binary, `10` for decimal, and `16` for hexadecimal (default).

`coewrite(...,filename)` writes a XILINX.COE file to `filename`. If you omit the file extension, `coewrite` adds the `.coe` extension to the name of the file.

Examples

`coewrite` generates an ASCII text file that contains the filter coeffcents in a format the XILINX CORE Generator can read and load. In this example, you create a 30th-order quantized filter and generate the `.coe` file that include the filter coefficients as well as associated information about the filter.

```
b = firceqrip(30,0.4,[0.05 0.03]);
hq = qfilt('fir',{b});
coewrite(hq,10,'mycoefile');
```

when you look at `mycoefile.coe`, you see the following:

```
;
; XILINX CORE Generator(tm) Distributed Arithmetic FIR filter
coefficient (.COE) File
; Generated by MATLAB(tm) and the Filter Design Toolbox.
;
; Generated on: 15-Mar-2002 13:47:15
;
```

```
Radix = 10;
Coefficient_Width = 16;
CoefData =   -41,
 -851,
 -366,
  308,
  651,
   22,
 -873,
 -658,
  749,
 1504,
   21,
-2367,
-2012,
 3014,
 9900,
....
```

`coewrite` puts the filter coefficients in column-major order and reports the radix, the coefficient width, and the coefficients. These represent the minimum set of data needed in a `.coe` file.

See Also

`coeread`

Purpose Apply convergent rounding

Syntax `convergent(x)`

Description `convergent(x)` rounds the elements of `x` to the nearest integer, except in a tie, then round to the nearest even integer.

Examples `round` and `convergent` differ in the way they treat values whose fractional part is 0.5. In `round`, every tie is rounded up in absolute value. `convergent` rounds ties to the nearest even integer.

```
x=[-3.5:3.5]';
[x convergent(x) round(x)]
ans =

   -3.5000   -4.0000   -4.0000
   -2.5000   -2.0000   -3.0000
   -1.5000   -2.0000   -2.0000
   -0.5000         0   -1.0000
    0.5000         0    1.0000
    1.5000    2.0000    2.0000
    2.5000    2.0000    3.0000
    3.5000    4.0000    4.0000
```

See Also `quantizer/quantize`

convert

Purpose Convert filter structures of quantized filters

Syntax

```
hq = convert(hq,newstruct)
```

Description

`hq = convert(hq,newstruct)` returns a quantized filter whose structure has been transformed to the filter structure specified by string `newstruct`. You can enter any one of the following quantized filter structures:

- `'antisymmetricfir'`: Antisymmetric finite impulse response (FIR).
- `'df1'`: Direct form I.
- `'df1t'`: Direct form I transposed.
- `'df2'`: Direct form II.
- `'df2t'`: Direct form II transposed. Default filter structure.
- `'fir'`: FIR.
- `'firt'`: Direct form FIR transposed.
- `'latcallpass'`: Lattice allpass.
- `'latticeca'`: Lattice coupled-allpass.
- `'latticecapc'`: Lattice coupled-allpass power-complementary.
- `'latticear'`: Lattice autoregressive (AR).
- `'latticema'`: Lattice moving average (MA) minimum phase.
- `'latcmax'`: Lattice moving average (MA) maximum phase.
- `'latticearma'`: Lattice ARMA.
- `'statespace'`: Single-input/single-output state-space.
- `'symmetricfir'`: Symmetric FIR. Even and odd forms.

All filters can be converted to the following structures:

- `df1`
- `df1t`
- `df2`
- `df2t`
- `statespace`
- `latticearma`

For the following filter classes, you can specify other conversions as well:

- Minimum phase FIR filters can be converted to `latticema`
- Maximum phase FIR filters can be converted to `latcmax`
- Allpass filters can be converted to `latcallpass`

`convert` generates an error if you specify a conversion that is not possible.

Examples

```
[b,a]=ellip(5,3,40,.7);
Hq = qfilt('df2t',{b,a});
Hq2 = convert(Hq,'statespace')
Hq2 =
Quantized State-space filter
⋮
FilterStructure = statespace
      ScaleValues = [1]
  NumberOfSections = 1
  StatesPerSection = [5]
 CoefficientFormat = quantizer('fixed', 'round', 'saturate', [16  15])
      InputFormat = quantizer('fixed', 'floor', 'saturate', [16  15])
     OutputFormat = quantizer('fixed', 'floor', 'saturate', [16  15])
MultiplicandFormat = quantizer('fixed', 'floor', 'saturate', [16  15])
    ProductFormat = quantizer('fixed', 'floor', 'saturate', [32  30])
        SumFormat = quantizer('fixed', 'floor', 'saturate', [32  30])
Warning: 9 overflows in coefficients.
```

See Also

`qfilt`

copyobj

Purpose

Make an independent copy of a quantizer, quantized filter, or quantized FFT

Syntax

```
obj1 = copyobj(obj)
[obj1,obj2,...] = copyobj(obja,objb,...)
```

Description

`obj1 = copyobj(obj)` makes a copy of obj and returns it in `obj1`. `obj` can be a quantizer, quantized filter, or quantized FFT.

`[obj1,obj2,...] = copyobj(obja,objb,...)` copies `obja` into `obj1`, `objb` into `obj2`, and so on. All objects can be quantizers, quantized filters, or quantized FFTs.

Using `copyobj` to copy a quantizer, quantized filter, or quantized FFT is not the same as using the command syntax `object1 = object` to copy a quantized object. Quantized filters, quantized FFTs, and quantizers have memory (their read-only properties). When you use `copyobj`, the resulting copy is independent of the original item—it does not share the original object's memory, such as the values of the properties `min`, `max`, `noverflows`, or `noperations`. Using `object1 = object` creates a new object that is an alias for the original and shares the original object's memory, and thus its property values.

Examples

```
q = quantizer('CoefficientFormat',[8 7]);
q1 = copyobj(q);
```

You can combine quantizers and quantized filters in the same `copyobj` command. You cannot include quantized FFTs with other quantized objects in one `copyobj` command.

```
hq = qfilt;
q=quantizer;
[hq1,q1] = copyobj(hq,q)
```

See Also

`qfilt`, `qfft`, `quantizer`, `get`, `set`

denormalmax

Purpose Return the largest denormalized quantized number for a quantizer

Syntax

```
x = denormalmax(q)
```

Description `x = denormalmax(q)` is the largest positive denormalized quantized number where `q` is a quantizer. Anything larger than `x` is a normalized number. Denormalized numbers apply only to floating-point format. When `q` represents fixed-point numbers, this function returns `eps(q)`.

Examples

```
q = quantizer('float',[6 3]);
x = denormalmax(q)
```

returns the value `x = 0.1875 = 3/16`.

Algorithm When `q` is a floating-point quantizer,
`denormalmax(q) = realmin(q) - denormalmin(q)`.

When `q` is a fixed-point quantizer, `denormalmax(q) = eps(q)`.

See Also `denormalmin`, `eps`, `quantizer`

denormalmin

Purpose Return the smallest denormalized quantized number for a quantizer

Syntax

```
x = denormalmin(q)
```

Description `x = denormalmin(q)` is the smallest positive denormalized quantized number where `q` is a quantizer. Anything smaller than `x` underflows to zero with respect to the quantizer `q`. Denormalized numbers apply only to floating-point format. When `q` represents a fixed-point number, `denormalmin` returns `eps(q)`.

Examples

```
q = quantizer('float',[6 3]);
denormalmin(q)
```

returns the value 0.0625 = 1/16.

Algorithm When `q` is a floating-point quantizer, $x = 2^{Emin-f}$, where $Emin$ is equal to `exponent(q)`.

When `q` is a fixed-point quantizer, $x = \mathrm{eps(q)} = 2^{-f}$, where f is equal to `fractionlength(q)`.

See Also `denormalmax`, `eps`, `quantizer`

Purpose Construct a discrete-time, coupled-allpass, lattice filter object

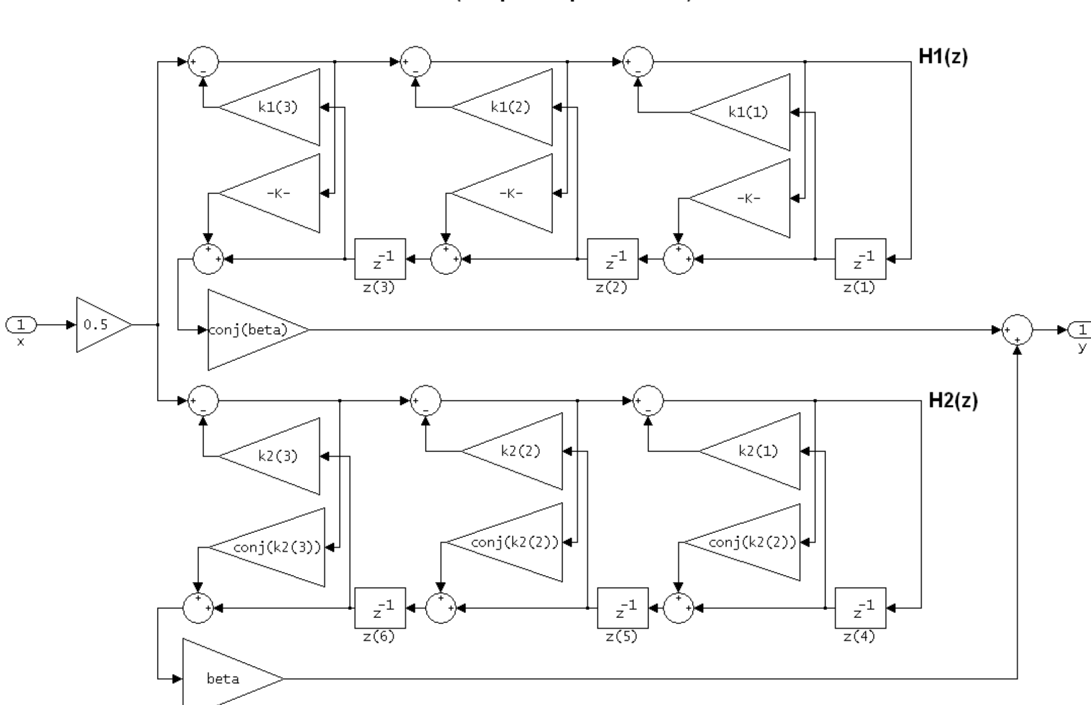
Syntax

```
Hd = dfilt.calattice(k1,k2,beta)
Hd = dfilt.calattice
```

Description

`Hd = dfilt.calattice(k1,k2,beta)` returns a discrete-time, coupled-allpass, lattice filter object, `Hd`, which is two allpass, lattice filter structures coupled together. The lattice coefficients for each structure are vectors, `k1` and `k2`. Input argument `beta` is shown in the diagram below

`Hd = dfilt.calattice` returns a default, discrete-time coupled-allpass, lattice filter object, `Hd`. The default values are `k1 = k2 =` [ ], which is the default value for `dfilt.latticeallpass`, and `beta = 1`. This filter passes the input through to the output unchanged.



Example

Specify a third-order lattice coupled-allpass filter structure for a dfilt filter, Hd with the following code.

```
k1 = [0.9511 + 0.3088i; 0.7511 + 0.1158i]
k2 = 0.7502 - 0.1218i
beta = 0.1385 + 0.9904i
Hd = dfilt.calattice(k1,k2,beta)

k1 =
   0.9511 + 0.3088i
   0.7511 + 0.1158i
k2 =
```

```
   0.7502 - 0.1218i
beta =
   0.1385 + 0.9904i
Hd =
     FilterStructure: 'Lattice Coupled Allpass'
           Allpass1: [1x1 dfilt.latticeallpass]
           Allpass2: [1x1 dfilt.latticeallpass]
               Beta: 0.1385+ 0.9904i
```

See Also

`dfilt.calatticepc`

`dfilt.latticeallpass`, `dfilt.latticear`, `dfilt.latticearma`, `dfilt.latticemamax`, `dfilt.latticemamin` in your Signal Processing Toolbox documentation

dfilt.calatticepc

Purpose

Construct a discrete-time, coupled-allpass, power-complementary lattice filter object

Syntax

```
Hd = dfilt.calatticepc(k1,k2,beta)
Hd = dfilt.calatticepc
```

Description

`Hd = dfilt.calatticepc(k1,k2)` returns a discrete-time, coupled-allpass, lattice filter object, `Hd`, with power-complementary output. This object is two allpass lattice filter structures coupled together to produce complementary output. The lattice coefficients for each structure are vectors, `k1` and `k2`, respectively. `beta` is shown in the diagram below

`Hd = dfilt.calatticepc` returns a default, discrete-time, coupled-allpass, lattice filter object, `Hd`, with power-complementary output. The default values are `k1=k2=[]`, which is the default value for the `dfilt.latticeallpass`. The default for `beta=1`. This filter passes the input through to the output unchanged.

calatticepc
(Coupled Allpass Lattice, Power Complementary output)

Example

Specify a third-order lattice coupled-allpass power complementary filter structure for a filter Hd with the following code.

```
k1 = [0.9511 + 0.3088i; 0.7511 + 0.1158i]
k2 = 0.7502 - 0.1218i
beta = 0.1385 + 0.9904i
Hd = dfilt.calattice(k1,k2,beta)

k1 =
   0.9511 + 0.3088i
   0.7511 + 0.1158i
k2 =
```

```
   0.7502 - 0.1218i
beta =
   0.1385 + 0.9904i
Hd =
    FilterStructure: 'Coupled-Allpass Lattice, Power
                        Complementary Output'
           Allpass1: [1x1 dfilt.latticeallpass]
           Allpass2: [1x1 dfilt.latticeallpass]
               Beta: 0.1385+ 0.9904i
```

See Also

dfilt.calattice

dfilt.latticeallpass, dfilt.latticear, dfilt.latticearma, dfilt.latticemamax, dfilt.latticemamin in your Signal Processing Toolbox documentation

Purpose

Display a quantizer, quantized FFT, or quantized filter

Description

Similar to omitting the closing semicolon from an expression on the command line, except that `disp` does not display the variable name. `disp` lists the property names and property values for a quantizer, quantized filter, or quantized FFT.

The following examples illustrate the default display for a quantized FFT `F` and a quantizer `q`.

```
F = qfft;
disp(F)
             Radix = 2
            Length = 16
 CoefficientFormat = quantizer('fixed', 'round', 'saturate', [16  15])
       InputFormat = quantizer('fixed', 'floor', 'saturate', [16  15])
      OutputFormat = quantizer('fixed', 'floor', 'saturate', [16  15])
     OperandFormat = quantizer('fixed', 'floor', 'saturate', [16  15])
     ProductFormat = quantizer('fixed', 'floor', 'saturate', [32  30])
         SumFormat = quantizer('fixed', 'floor', 'saturate', [32  30])
  NumberOfSections = 4
       ScaleValues = [1]

q = quantizer
q =
            Mode = fixed
       RoundMode = floor
    OverflowMode = saturate
          Format = [16  15]
             Max = reset
             Min = reset
      NOverflows = 0
     NUnderflows = 0
```

See Also

`set`

Purpose

Return the quantized relative accuracy for quantized filters, quantizers, and quantized FFTs

Syntax

```
eps(q)
eps(hq)
eps(f)
```

Description

`eps(q)` returns the quantization level of quantizer object `q`, or the distance from 1.0 to the next largest floating-point number when `q` is a floating-point quantizer object.

`eps(hq)` returns the quantization level of quantized filter `hq`, or the distance from 1.0 to the next largest floating-point number when `hq` is a floating-point quantized filter.

`eps(f)` returns the quantization level of quantized FFT `f`, or the distance from 1.0 to the next largest floating-point number when `f` is a floating-point quantized FFT.

Examples

```
q = quantizer('float',[6 3]);
eps(q)
```

returns the value 0.25. The following code

```
hq = qfilt;
f = qfft;
eps(hq)
eps(f)
```

returns the values

```
coefficient   3.051757813e-005
       input   3.051757813e-005
      output   3.051757813e-005
multiplicand   3.051757813e-005
     product   9.313225746e-010
         sum   9.313225746e-010
```

for the quantizers in the quantized filter, and

```
 coefficient   3.051757813e-005
       input   3.051757813e-005
      output   3.051757813e-005
multiplicand   3.051757813e-005
     product   9.313225746e-010
         sum   9.313225746e-010
```

for the quantizers in the quantized FFT.

Algorithm

For fixed-point or floating-point numbers, $e = 2^{-f}$ where e is the relative accuracy of the quantizer and f is the fraction length of the quantizer.

See Also

eps, exponentbias, exponentlength, exponentmax, exponentmin

Purpose

Return the mean of the quantization error resulting from quantizing a signal

Syntax

```
qerr = errmean(q)
```

Description

`qerr = errmean(q)` returns the mean value of the uniformly distributed random quantization error that results when you use quantizer `q` to quantize a signal.

The value of `qerr` does not depend on the signal quantized unless the precision (the value of the least significant bit) of your signal and your quantizer are very nearly the same. Use `eps` to determine the precision for quantizers or varied wordlengths.

When the precision of your signal is close to the precision of your quantizer, `qerr` may not match the theoretical value. When your signal has infinite extent and infinite precision, the value calculated for `qerr` matches the theoretical value of the mean of the uniformly distributed quantization error.

For most purposes, when the difference in precision between a signal and the quantizers is greater then 16 bits, the result calculated by `errmean` is exact. When you reduce the wordlength by three or four bits through quantization, `errmean` generates an excellent approximation. For wordlength changes that exceed four bits, `errmean` provides a less good match to the theoretical mean. For fixed-point quantizers, the wordlength property defines the precision.

As you change the rounding mode for your quantizer, the mean error value changes as well, as shown in this table.

Round Mode	Probability Density Function ($f(x)$ = pdf)	Mean (μ)	Variance (σ^2)	dB = $10\log_{10}\sigma^2$
ceil	$1/\varepsilon$; $0 \le x < \varepsilon$; 0 otherwise	$\varepsilon/2$	$\varepsilon^2/12$	$-6.02f - 10.79$
convergent	$1/\varepsilon$; $-\varepsilon/2 \le x \le \varepsilon/2$; 0 otherwise	0	$\varepsilon^2/12$	$-6.02f - 10.79$
fix	$1/(2\varepsilon)$; $-\varepsilon < x < \varepsilon$; 0 otherwise	0	$\varepsilon^2/3$	$-6.02f - 4.77$
floor	$1/\varepsilon$; $-\varepsilon < x \le 0$; 0 otherwise	$-\varepsilon/2$	$\varepsilon^2/12$	$-6.02f - 10.79$
round	$1/\varepsilon$; $-\varepsilon/2 < x \le \varepsilon/2$; 0 otherwise	0	$\varepsilon^2/12$	$-6.02f - 10.79$

In the table, ε represents the quantization level (`eps(q)`) for your quantizer, x is the uniformly distributed random quantization error, and `f` is the wordlength of the quantizer.

For more information about the `errmean` algorithm, and for a discussion about correction factors for quantizing from one fixed-point format and precision to another, refer to [1] in the References section.

Examples

Compare the mean value determined by Monte Carlo methods to the mean value computed by `errmean`. In this example, the fraction length for `q` equals 15 bits (`eps` = 3.0518e-005) and the fraction length for the signal `u` is 31 bits (`eps` = 4.6566e-010) .

```
q = quantizer('fixed','floor',[16 15]);
m = errmean(q)
m =

 -1.5259e-005 % =-eps(q/2) from the table
% Compare m to the sample mean from a Monte Carlo experiment
r = realmax(q);
u = 2*r*rand(1000,1)-r;  % Original signal
y = quantize(q,u);       % Quantized signal
e = y - u;               % Error
m_est = mean(e)          % Estimate of the error mean
m_est =

 -1.5471e-005
abs(m-m_est)

ans =

  2.1174e-007           % Difference between the error estimates
```

Algorithm

You use similar equations to calculate the mean value for the five rounding modes. In the following equations, $x = y\text{-}u$, where u is the original signal and y is the signal value after quantization. ε is the minimum quantization step for the quantizer. For all of the following, $f(x)$ denotes the probability density function of the error.

Ceil mode

$$f(x) = \begin{cases} 1/\varepsilon, & 0 \le x < \varepsilon \\ 0, & \text{otherwise} \end{cases}$$

$$\mu = E(x) = \int_{-\infty}^{\infty} xf(x)dx = \varepsilon/2$$

Convergent mode

$$f(x) = \begin{cases} 1/\varepsilon, & -\varepsilon/2 \le x \le \varepsilon/2 \\ 0, & \text{otherwise} \end{cases}$$

$$\mu = E(x) = \int_{-\infty}^{\infty} xf(x)dx = 0$$

Fix mode

$$f(x) = \begin{cases} 1/(2\varepsilon), & -\varepsilon < x < \varepsilon \\ 0, & \text{otherwise} \end{cases}$$

$$\mu = E(x) = \int_{-\infty}^{\infty} xf(x)dx = 0$$

Floor mode

$$f(x) = \begin{cases} 1/\varepsilon, & -\varepsilon < x \le 0 \\ 0, & \text{otherwise} \end{cases}$$

$$\mu = E(x) = \int_{-\infty}^{\infty} xf(x)dx = -\varepsilon/2$$

Round mode

$$f(x) = \begin{cases} 1/\varepsilon, & -\varepsilon/2 < x \le \varepsilon/2 \\ 0, & \text{otherwise} \end{cases}$$

$$\mu = E(x) = \int_{-\infty}^{\infty} x f(x) dx = 0$$

See Also

`quantizer/errpdf`, `quantizer/errvar`

References

[1] Schlichthärle, Dietrich, *Digital Filter*, Springer, 2000, Section 8.3 "Quantization," pp. 233-240

errpdf

Purpose

Calculate the probability density function (pdf) of the quantization error

Syntax

```
(qf,x) = errpdf(q)
(qf) = errpdf(q,x)
```

Description

`(qf,x) = errpdf(q)` returns `qf`, the pdf of the quantization error, evaluated at the values returned in `x`. When you do not provide `x` as an input vector to define the values at which to calculate `qf`, `errpdf` uses 128 equally spaced points between (-2*`eps`) and (2*`eps`) as the values at which it calculates `qf`.

`(qf) = errpdf(q,x)` returns `qf`, the pdf of the quantization error, evaluated at the values specified in vector `x`. Values in `qf` result from using `q` to quantize a signal. The error generated by the quantization process is random and uniformly distributed around zero.

When the precision of your signal is close to the precision of your quantizer, `qf` may not match the theoretical precision. When your signal has infinite extent and infinite precision, the value calculated for `qf` matches the theoretical value of the pdf of the uniformly distributed quantization error.

For most purposes, when the difference in precision between a signal and the quantizers is greater then 16 bits, the result calculated by `errpdf` is exact. When you reduce the wordlength by 3 or 4 bits through quantization, `errpdf` generates an excellent approximation. For wordlength changes that exceed four bits, `errpdf` provides a less good match to the theoretical mean. For fixed-point quantizers, the wordlength property defines the precision.

As you change the rounding mode for your quantizer, the pdf changes as well, as shown in this table.

Round Mode	Probability Density Function ($f(x)$ = pdf)	Mean (μ)	Variance (σ^2)	dB = $10\log_{10}\sigma^2$
ceil	$1/\varepsilon$; $0 \le x < \varepsilon$; 0 otherwise	$\varepsilon/2$	$\varepsilon^2/12$	$-6.02f - 10.79$
convergent	$1/\varepsilon$; $-\varepsilon/2 \le x \le \varepsilon/2$; 0 otherwise	0	$\varepsilon^2/12$	$-6.02f - 10.79$
fix	$1/(2\varepsilon)$; $-\varepsilon < x < \varepsilon$; 0 otherwise	0	$\varepsilon^2/3$	$-6.02f - 4.77$

Round Mode	Probability Density Function ($f(x)$ = pdf)	Mean (μ)	Variance (σ^2)	dB = $10\log_{10}\sigma^2$
floor	$1/\varepsilon$; $-\varepsilon < x \le 0$; 0 otherwise	$-\varepsilon/2$	$\varepsilon^2/12$	$-6.02f - 10.79$
round	$1/\varepsilon$; $-\varepsilon/2 < x \le \varepsilon/2$; 0 otherwise	0	$\varepsilon^2/12$	$-6.02f - 10.79$

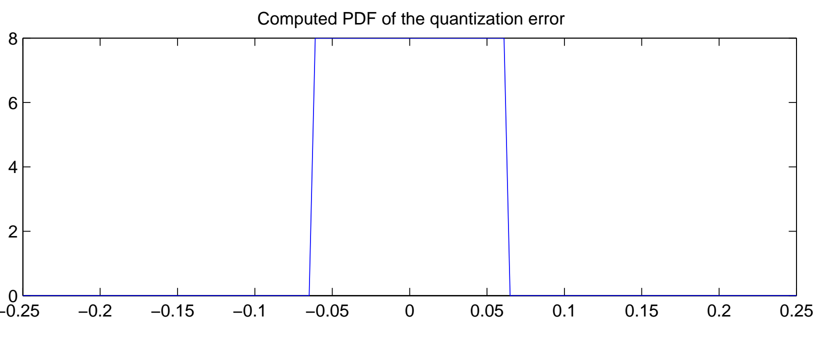
In the table, ε represents the quantization level (eps(q)) for your quantizer, x is the uniformly distributed random quantization error, and f is the wordlength of the quantizer.

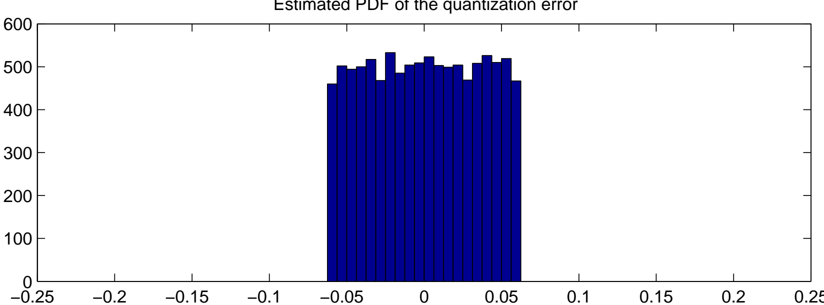
Examples

Using a quantizer on a signal, compare the pdf calculated by errpdf to the error generated by a Monte Carlo experiment. Notice that the quantizer uses 4 bits with 3 bits for the fraction length. Signal u in the Monte Carlo experiment is a double array.

```
q = quantizer('round',[4 3]);
[f,x] = errpdf(q);
subplot(211)
plot(x,f)
title('Computed PDF of the quantization error.')

% Compare f to the sample pdf from a Monte Carlo experiment
r = realmax(q);
u = 2*r*rand(10000,1)-r;  % Original signal
y = quantize(q,u);        % Quantized signal
e = y - u;                % Error
subplot(212)
hist(e,20);set(gca,'xlim',[min(x) max(x)])
title('Estimate of the PDF of the quantization error.')
```

Looking at the plot shown here you see that the computed, or theoretical, and estimated pdfs agree closely.





Algorithm

Here are the methods for calculating the pdf for the five rounding modes. In the equations, $x = y\text{-}u$, where u is the original signal and y is the signal value after quantization. ε is the minimum quantization step for the quantizer. For all of the following, *f(x)* denotes the probability density function of the error.

Ceil mode

$$f(x) = \begin{cases} 1/\varepsilon, & 0 \le x < \varepsilon \\ 0, & \text{otherwise} \end{cases}$$

Convergent mode

$$f(x) = \begin{cases} 1/\varepsilon, & -\varepsilon/2 \le x \le \varepsilon/2 \\ 0, & \text{otherwise} \end{cases}$$

Fix mode

$$f(x) = \begin{cases} 1/(2\varepsilon), & -\varepsilon < x < \varepsilon \\ 0, & \text{otherwise} \end{cases}$$

Floor mode

$$f(x) = \begin{cases} 1/\varepsilon, & -\varepsilon < x \le 0 \\ 0, & \text{otherwise} \end{cases}$$

Round mode

$$f(x) = \begin{cases} 1/\varepsilon, & -\varepsilon/2 < x \le \varepsilon/2 \\ 0, & \text{otherwise} \end{cases}$$

See Also

`quantizer/errmean`, `quantizer/errvar`

References

[1] Schlichthärle, Dietrich, *Digital Filter*, Springer, 2000, Section 8.3 “Quantization,” pp. 233-240

Purpose

Return the variance of the quantization error resulting from quantizing a signal

Syntax

```
qvar = errvar(q)
```

Description

`qvar = errvar(q)` returns the variance of the uniformly distributed random quantization error that results when you use quantizer `q` to quantize a signal.

The value of `errvar` does not depend on the signal quantized unless the precision (the value of the least significant bit) of your signal and your quantizer are very nearly the same. Use `eps` to determine the precision for quantizers or various wordlengths.

When the precision of your signal is close to the precision of your quantizer, `qvar` may not match the theoretical value. When your signal has infinite extent and infinite precision, the value calculated for `qvar` matches the theoretical value of the variance of the uniformly distributed quantization error.

For most purposes, when the difference in precision between a signal and the quantizers is greater then 16 bits, the result calculated by `errvar` is exact. When you reduce the wordlength by 3 or 4 bits through quantization, `errvar` generates an excellent approximation. For wordlength changes that exceed 4 bits, `errvar` provides a less good match to the theoretical mean. For fixed-point quantizers, the wordlength property defines the precision.

As you change the rounding mode for your quantizer, the variance changes as well, as shown in this table.

Round Mode	Probability Density Function ($f(x)$ = pdf)	Mean (μ)	Variance (σ^2)	dB = $10\log_{10}\sigma^2$
ceil	$1/\varepsilon$; $0 \le x < \varepsilon$; 0 otherwise	$-\varepsilon/2$	$\varepsilon^2/12$	$-6.02f - 10.79$
convergent	$1/\varepsilon$; $-\varepsilon/2 \le x \le \varepsilon/2$; 0 otherwise	0	$\varepsilon^2/12$	$-6.02f - 10.79$
fix	$1/(2\varepsilon)$; $-\varepsilon < x < \varepsilon$; 0 otherwise	0	$\varepsilon^2/3$	$-6.02f - 4.77$
floor	$1/\varepsilon$; $-\varepsilon < x \le 0$; 0 otherwise	$-\varepsilon/2$	$\varepsilon^2/12$	$-6.02f - 10.79$
round	$1/\varepsilon$; $-\varepsilon/2 < x \le \varepsilon/2$; 0 otherwise	0	$\varepsilon^2/12$	$-6.02f - 10.79$

In the table, ε represents the quantization level (eps(q)) for your quantizer, x is the uniformly distributed random quantization error, and f is the wordlength of the quantizer.

Examples

To demonstrate the accuracy of errvar, compare the theoretical variance for the quantization error as determined by Monte Carlo analysis using a signal to the result from errvar:

```
q = quantizer;
v = errvar(q)

% Compare to the sample variance from a Monte Carlo experiment
r = realmax(q);
u = 2*r*rand(1000,1)-r;  % Original signal
y = quantize(q,u);       % Quantized signal
e = y - u;               % Error
v_est = var(e)           % Estimate of the error variance
v =

  7.7610e-011 % =eps(q)^2/12 from the table

v_est =

  7.5534e-011

v_est-v

ans =

 -2.0758e-012
```

Algorithm

The variance depends on the rounding mode of the quantizer. Ceil, convergent, floor, and round share the same variance through different calculations. Fix differs by a factor of four. For the definition and derivation of μ for each mode, refer to errvar. $E(x)$ is the expected value of the random variable; the variance is σ^2. In the equations, $x = y\text{-}u$, where u is the original signal and y is the signal value after quantization. ε is the minimum quantization step for the quantizer.

Ceil and floor modes

$$E(x)^2 = 1/\varepsilon \int_{-\varepsilon}^{0} x^2 dx$$

$$= \varepsilon^2/3$$

$$\sigma^2 = E(x^2) - \mu^2 = \varepsilon^2/3 - \varepsilon^2/4 = \varepsilon^2/12$$

Convergent and round modes

$$E(x)^2 = 1/\varepsilon \int_{-\varepsilon/2}^{\varepsilon/2} x^2 dx$$

$$= \varepsilon^2/12$$

$$\sigma^2 = E(x^2) - \mu^2 = \varepsilon^2/12 - 0 = \varepsilon^2/12$$

Fix mode

$$E(x)^2 = 1/\varepsilon \int_{-\varepsilon}^{0} x^2 dx$$

$$= \varepsilon^2/3$$

$$\sigma^2 = E(x^2) - \mu^2 = \varepsilon^2/3 - 0 = \varepsilon^2/3$$

See Also

`quantizer/errmean`, `quantizer/errpdf`

References

[1] Schlichthärle, Dietrich, *Digital Filter*, Springer, 2000, Section 8.3 “Quantization,” pp. 233-240

Purpose Return the exponent bias for a quantizer

Syntax

```
b = exponentbias(q)
```

Description `b = exponentbias(q)` returns the exponent bias of the quantizer `q`. For fixed-point quantizers, `exponentbias(q)` returns 0.

Examples

```
q = quantizer('double');
b = exponentbias(q)
```

returns the value `b = 1023`.

Algorithm For floating-point quantizers, $b = 2^{e-1} - 1$, where `e = eps(q)`, and `exponentbias` is the same as the exponent maximum.

For fixed-point quantizers, `b = 0` by definition.

See Also `eps`, `exponentlength`, `exponentmax`, `exponentmin`

exponentlength

Purpose Return the exponent length of a quantizer

Syntax

```
e = exponentlength(q)
```

Description `e = exponentlength(q)` returns the exponent length of quantizer `q`. When `q` is a fixed-point quantizer, `exponentlength(q)` returns 0. This is useful because exponent length is valid whether the quantizer mode is floating-point or fixed-point.

Examples

```
q = quantizer('double');
e = exponentlength(q);
```

returns the value `e = 11`.

Algorithm The exponent length is part of the format of a floating-point quantizer `[w, e]`. For fixed-point quantizers, $e = 0$ by definition.

See Also `eps`, `exponentbias`, `exponentmax`, `exponentmin`

exponentmax

Purpose Return the maximum exponent for a quantizer

Syntax

```
exponentmax(q)
```

Description

exponentmax(q) returns the maximum exponent for quantizer q. When q is a fixed-point quantizer, it returns 0.

```
q = quantizer('double');
exponentmax(q)
```

returns the value ans = 1023.

Algorithm

For floating-point quantizers, $E_{max} = 2^{e-1} - 1$.

For fixed-point quantizers, $E_{max} = 0$ by definition.

See Also

eps, exponentbias, exponentlength, exponentmin

exponentmin

Purpose Return the minimum exponent for a quantizer

Syntax

```
emin = exponentmin(q)
```

Description `emin = exponentmin(q)` returns the minimum exponent for quantizer `q`. If `q` is a fixed-point quantizer, `exponentmin` returns 0.

Examples

```
q = quantizer('double');
emin = exponentmin(q)
```

returns the value `emin =  1022`.

Algorithm For floating-point quantizers, $E_{min} = -2^{e-1} + 2$.

For fixed-point quantizers, $E_{min} = 0$.

See Also `eps`, `exponentbias`, `exponentlength`, `exponentmax`

Purpose Open the Filter Design and Analysis Tool.

Syntax

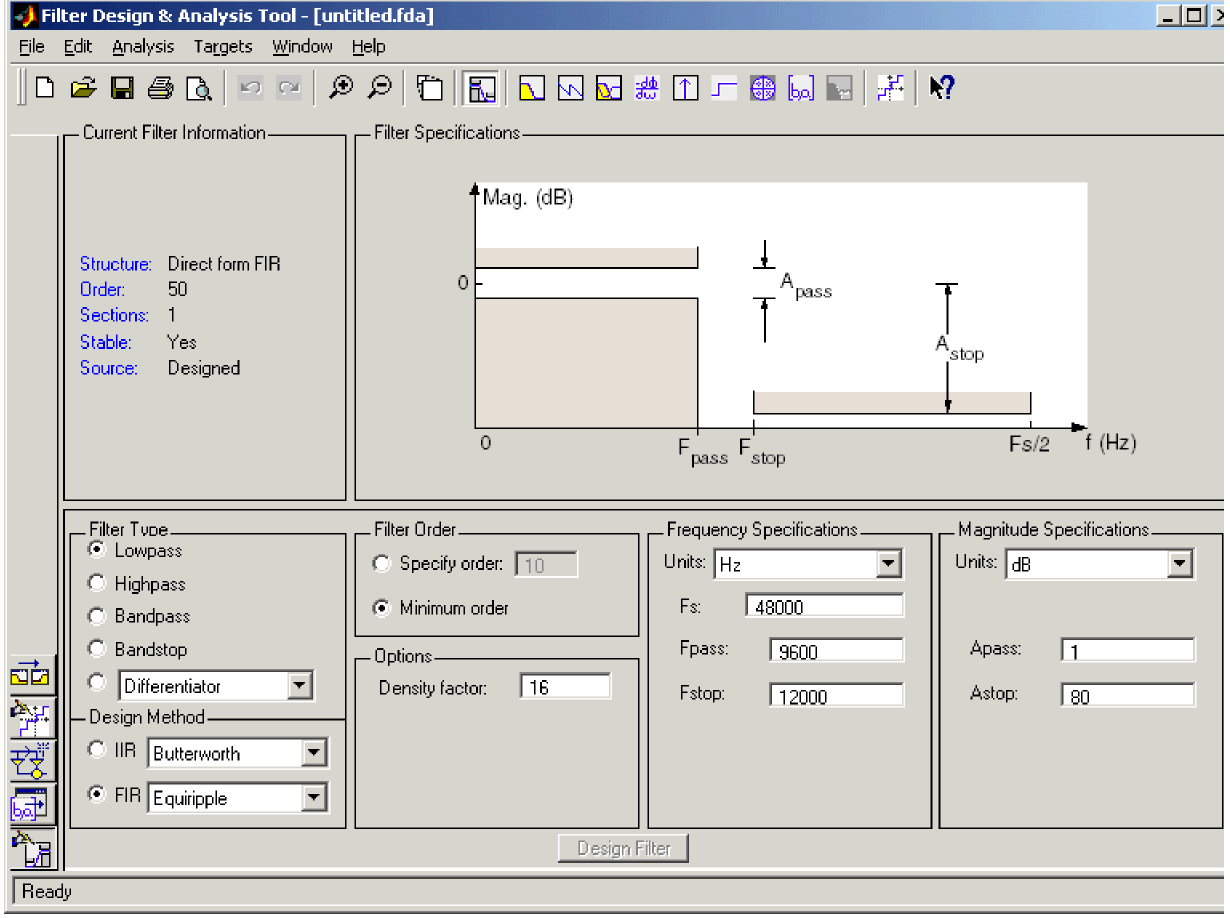
```
fdatool
```

Description `fdatool` opens the Filter Design and Analysis Tool (FDATool). Use this tool to:

- Design filters
- Quantize filters
- Analyze filters
- Modify existing filter designs
- Realize Simulink models of quantized, direct form, FIR filters
- Perform digital frequency transformations of filters

Refer to “Using FDATool with the Filter Design Toolbox” for more information about using the quantization features of FDATool. For general information about using FDATool, refer to “Filter Design and Analysis Tool” in your Signal Processing Toolbox documentation.

When you open FDATool and you have Filter Design Toolbox installed, FDATool incorporates additional features that are provided by Filter Design Toolbox. With Filter Design Toolbox installed, FDATool lets you design and analyze quantized filters, as well as convert quantized filters to various filter structures, transform filters, and realize models of filters.

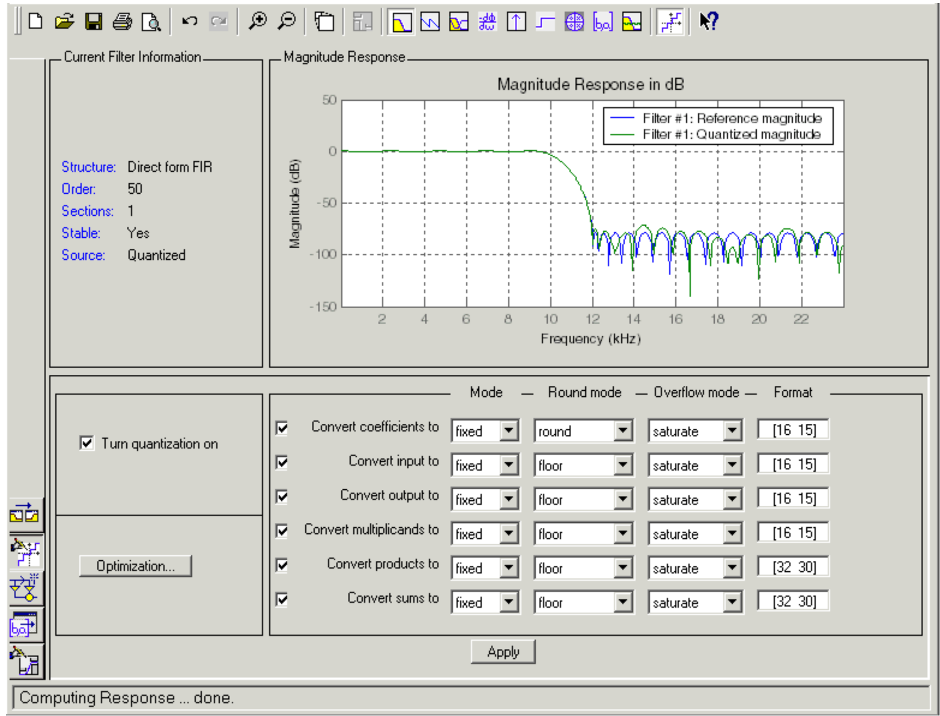


Use the **Set Quantization Parameters** option to configure the quantization settings for a quantized filter, or to access the tools to scale the filter coefficients.

Set Quantization Parameters — provides access to the properties of the quantizers that compose a quantized filter. When you click **Set Quantization Parameters**, you see FDATool displaying the quantization parameters at the bottom of the dialog, as shown in the figure.

Transform Filter—clicking this button opens the **Frequency Transformations** pane so you can use digital frequency transformations to change the magnitude response of your filter.

Realize Model—starting from your quantized, direct form, FIR filter, clicking this button creates a Simulink model of your filter structure in new model window.



Turn quantization on—enables the **Quantized Filter** panel and quantizes the current filter. Select this option when you want to quantize a filter or set the quantization properties for a filter.

Optimize...—opens the **Quantized Optimizations** dialog to let you specify how to quantize and scale your filter.

Remarks

By incorporating many advanced filter design methods from Filter Design Toolbox,FDATool provides more design methods than the SPTool Filter Designer.

See Also

`fdatool`, `fvtool`, `sptool` in your Signal Processing Toolbox documentation

Purpose Apply a quantized fast Fourier transform to data

Syntax

```
y = fft(F,x)
y = fft(F,x,dim)
```

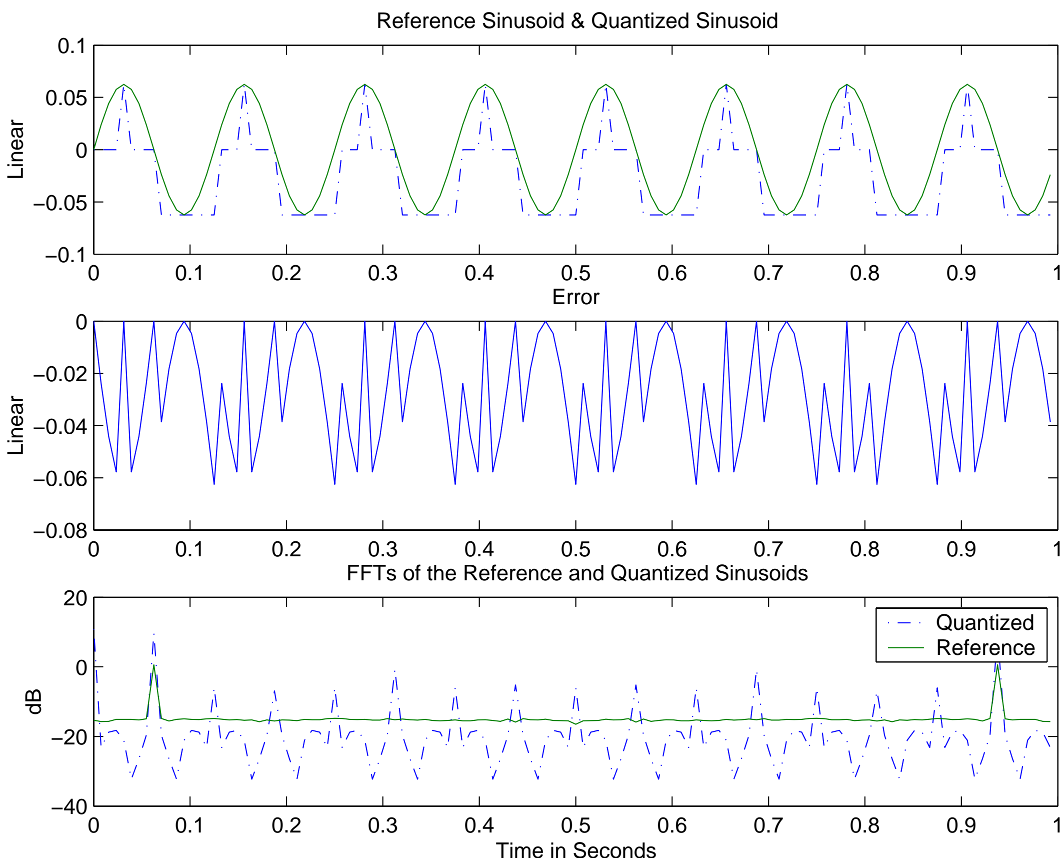
Description `y = fft(F,x)` uses quantized FFT (fast Fourier transform) `F` to compute the FFT of vector `x`. The parameters of the quantized FFT are specified in quantized FFT `F`. The radix is specified by `F.radix`. The decimation is specified by `F.decimation`. The length of the FFT is specified by `F.length`. When the length of `x` is less than `F.length`, `x` is padded with zeros. When `x` is longer than `F.length`, `x` is truncated. For matrices, the FFT operation is applied to each column. For N-D arrays, the FFT operation operates on the first nonsingleton dimension.

`y = fft(F,x,dim)` applies the quantized FFT operation across the dimension `dim`.

Examples When you quantize a sinusoid, you generate errors as a result of the quantization process. This example demonstrates this effect. We create a sinusoid, quantize it, and look at the error between the quantized and unquantized sinusoids. Then we plot the FFTs for both signals.

```
n = 128;
t = (0:n-1/n);
x = sin(2*pi*16*t)/16;% Reference sinusoid
q = quantizer([5 4]);
f = qfft('length',n,'inputformat',q);
plot(t,[quantize(q,x);x]);% Plot both signals
plot(t,[quantize(q,x)-x]);% Plot the error
plot(t,[20*log10(abs(fft(f,x)));...
(20*log10(abs(fft(x)))/20)])% Plot the FFTs for both signals
```

The following figure presents the results.



Looking at the subplot of the error between the reference and quantized sinusoids, you see that the error is periodic. Because the error is periodic, the FFT of the quantized sinusoid includes periodic frequency content not in the reference signal, as seen in the FFTs subplot.

See Also

`get`, `ifft`, `qreport`, `qfft`, `set`

Purpose Apply a quantized filter to data and access states and filtering information

Syntax

```
y = filter(Hq,x)
[y,zf] = filter(Hq,x)
[...] = filter(Hq,x,zi)
[...] = filter(Hq,x,zi,dim)
[y,zf,s,z,v] = filter(Hq,x...)
```

Description `y = filter(Hq,x)` filters a vector of real or complex input data `x` through a quantized filter `Hq`, producing filtered output data `y`. The vectors `x` and `y` have the same length.

If `x` is a matrix, `y = filter(Hq,x)` filters each column of `x` to produce a matrix `y`. If `x` is a multidimensional array, `y = filter(Hq,x)` filters `x` along the first nonsingleton dimension of `x`.

`[y,zf] = filter(Hq,x)` produces an additional output argument `zf`. `zf` contains the final values for the state vector calculated from zero initial conditions for the state. The form `zf` takes depends on the data to be filtered and the number of stages in the filter, as detailed in Table 13-3, Final State Form Depends on Filtered Data and Filter Structure.

`[...] = filter(Hq,x,zi)` specifies the initial conditions for the state vector in `zi`. The form for specifying `zi` is described in Table 13-2, Initial State Format Depends on the Filter Structure. To specify the same initial condition for all state components, enter `zi` as a scalar. You can set `zi` to zero, `[]`, or `{}` to specify zero (the default) initial conditions.

The form of the initial and final states associated with a quantized filter `Hq` depends on the filter structure and the data to be filtered. The following tables

give the form for either entering the initial states or retrieving the final states of the quantized filter.

Table 13-2: Initial State Format Depends on the Filter Structure

Number of Filter Sections	Format of the Initial State
1	A column vector of length s_1
n	A 1-by-n cell array of vectors of length s_i, i=1, 2,...,n

Table 13-3: Final State Form Depends on Filtered Data and Filter Structure

Filtered Data	Number of Filter Sections	Form of the Final State
Vector	1	A column vector of length s_1
Vector	n	A 1-by-n cell array of vectors of length s_i, i=1, 2,...,n
Multidimensional array	1	An s_1-by-c matrix
Multidimensional array	n	1-by-n cell array of s_i-by-c matrices, i=1, 2,...,n

The variables in these tables are described as follows:

- s_i is the number of states in the ith section of the filter.
- c is `prod(size(x))/size(x,dim)`, where `dim` is the first nonsingleton dimension into which you are filtering.

To figure out the dimensions of the initial or final conditions, run the filter once with empty initial conditions. Then the final conditions are the right size for the initial conditions:

```
[y,zf] = filter(Hq,x);
```

Look at the size and data type of `zf`. The initial conditions, `zi`, will be the same size as `zf`.

Use the `StatesPerSection` property of the quantized filter `Hq` to access the number of states in each section. See "Quantized Filter Properties Reference" on page 12-11 for more information on filter properties.

`[...] = filter(Hq,x,zi,dim)` applies the quantized filter `Hq` to the input data located along the specific dimension of `x` specified by `dim`.

`[y,zf,s,z,v] = filter(Hq,x...)` returns `s`, a MATLAB structure containing quantization information (refer to `qreport` for details); `z`, the filter's state sequence; and `v`, the number of overflows at each time step of the filter. When you include four or five output arguments, the input argument `x` must be a vector. `z` is a cell array containing the sequence of states at each time step, having 1 element per filter and 1 column per time step. The initial conditions of the kth filter section are in the first column of `z{k}:zi{k}=z{k}(:,1)`. The final conditions of the kth filter section are in the last column of `z{k}:zf{k} = z{k}(:,end)`. Overflows for the kth section are in `v{k}`.

Examples

Find the response of a quantized digital filter.

```
randn('state',0);
x = randn(100,1);
warning on;
[b,a] = butter(3,.9,'high');
Hq = sos(qfilt('referencecoefficients',{b,a}))
Warning: 3 overflows in coefficients.

y = filter(Hq,x);

Warning: 27 overflows in QFILT/FILTER.

                    Max          Min    NOverflows    NUnderflows    NOperations
 Coefficient          1      -0.7419             0              0              4
                 0.8238           -1             0              0              6
       Input      2.183       -2.171            27              0            100
      Output     0.4361        -0.45             0              0            100
Multiplicand          1           -1             0              2            600
                 0.4361        -0.45             0              0            700
     Product    0.01276     -0.01227             0              0            600
                 0.4361        -0.45             0              0            700
         Sum    0.01278     -0.01221             0              0            300
                 0.2181       -0.225             0              0            500
```

```
Hq.filterstructure

ans =

df2t
```

Notice the warnings returned during filter quantization and application. The first warning indicates that one of the filter coefficients overflowed during quantization before converting the filter to second-order section form. Applying the function `sos` to the filter removed the coefficient overflows. The second warning displays the overflow report, listing details about the filtering operation.

Note Use `qreport` to display the information logged during a filtering operation.

Algorithm

The `filter` command implements fixed- or floating-point arithmetic on the quantized filter structure you specify. The *state vector* z associated with the filter is a vector whose components are derived from the values of each of the input signals to each delay in the filter. The length of z is the same as the number of delays in the filter.

The implementation of `filter` depends on the filter structure. For example, the operation of `filter` at sample m for a direct form II transposed filter is given by the quantized time domain difference equations for y and the states z_i shown below. Square brackets denote the quantization that takes place for the input data x, the output data y, the coefficients, the products, and the sums.

$$\begin{aligned}
y(m) &= \left[\frac{[[[b(1)][x(m)]] + z_1(m-1)]}{[a(1)]}\right] \\
z_1(m) &= [[[b(2)][x(m)]] + z_2(m-1) - [[a(2)]y(m)]] \\
\vdots &= \vdots \\
z_{n-2}(m) &= [[[[b(n-1)][x(m)]] + z_{n-1}(m-1)] - [[a(n-1)]y(m)]] \\
z_{n-1}(m) &= [[[b(n)][x(m)]] - [[a(n)]y(m)]]
\end{aligned}$$

Notice that for this `df2t` filter structure, you divide by $a(1)$. For efficient computation, choose $a(1)$ to be a power of 2.

Note `qfilt/filter` does not normalize the filter coefficients automatically. Function `filter` supplied by MATLAB does normalize the coefficients.

See Also

`impz`, `qfilt`, `qreport`

References

[1] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989.

firceqrip

Purpose Design constrained, equiripple, finite impulse response (FIR) filters

Syntax

```
h = firceqrip(n,wo,del)
h = firceqrip(..., slope ,r)
h = firceqrip(...,'passedge')
h = firceqrip(...,'stopedge')
h = firceqrip(...,'high')
h = firceqrip(...,'min')
h = firceqrip(...,'invsinc',c)
```

Description

`h = firceqrip(n,wo,del)` design an order `n` filter (filter length equal `n+1`) lowpass FIR filter with linear phase.

`firceqrip` produces the same equiripple lowpass filters that `remez` produces using the Parks-McClellan algorithm. The difference is how you specify the filter characteristics for the function.

Input argument `wo` specifies the cutoff frequency. The two-element vector `del` specifies the peak or maximum error allowed in the passband and stopbands. Enter `[d1 d2]` for `del` where `d1` sets the passband error and `d2` sets the stopband error. Since `firceqrip` works in the normalized frequency domain, you must set `wo` to be between 0 and 1 (0 < `wo` < 1).

`h = firceqrip(..., slope ,r)` uses the input keyword '**`slope`**' and input argument `r` to design a filter with a stopband that does not demonstrate equiripple characteristics. `r` determines the slope of the stopband in dB when `r` > 0. Try setting `r` to `10` to see the effect on the filter frequency response. In the Examples section, Example 3 designs a filter with `r` equal to 20.

`h = firceqrip(...,'passedge')` designs a filter where `wo` specifies the frequency at which the passband starts to roll off.

`h = firceqrip(...,'stopedge')` designs a filter where `wo` specifies the frequency at which the stopband begins.

`h = firceqrip(...,'high')` designs a high pass FIR filter instead of a lowpass filter.

`h = firceqrip(...,'min')` designs an FIR filter with minimum phase.

`h = firceqrip(...,'invsinc',c))` designs a lowpass filter whose passband has the shape of the inverse sinc function. For this syntax, keyword **`invsinc`** applies the inverse sinc function as defined by whether `c` is a scalar or a two-element vector:

- When you use `c` as a scalar with the **`invsinc`** keyword, `firceqrip` applies the function 1/sinc(`c*w`), where `w` is the normalized frequency, to the passband.
- When you use `c` as a two-element vector entered as `[c p]`,with the **`invsinc`** keyword, `firceqrip` applies the function 1/sinc$(c*w)^p$ to the passband, where `w` is the normalized frequency.

In both cases, `c` must meet the condition $c < 1/w_o$.

When you use a cascaded-integrated comb (CIC) filter in series with this FIR filter, argument `p` lets you compensate for the droop in the passband of the CIC filter. Setting `p` equal to the number of stages in your CIC generally produces an FIR filter whose passband neatly compensates for the CIC passband shape.

To let you specify precisely the FIR filter to design, use any or all of the optional input arguments together. Any ordering of the optional arguments works—order is not important in the function call. Refer to Examples 3 and 4 to see multiple optional input arguments being used.

Note If the `wo` you specify is too small or too large, or if either `c` or `p` is too large, your filter specifications may be unfeasible, causing the design algorithm to fail to generate your filter.

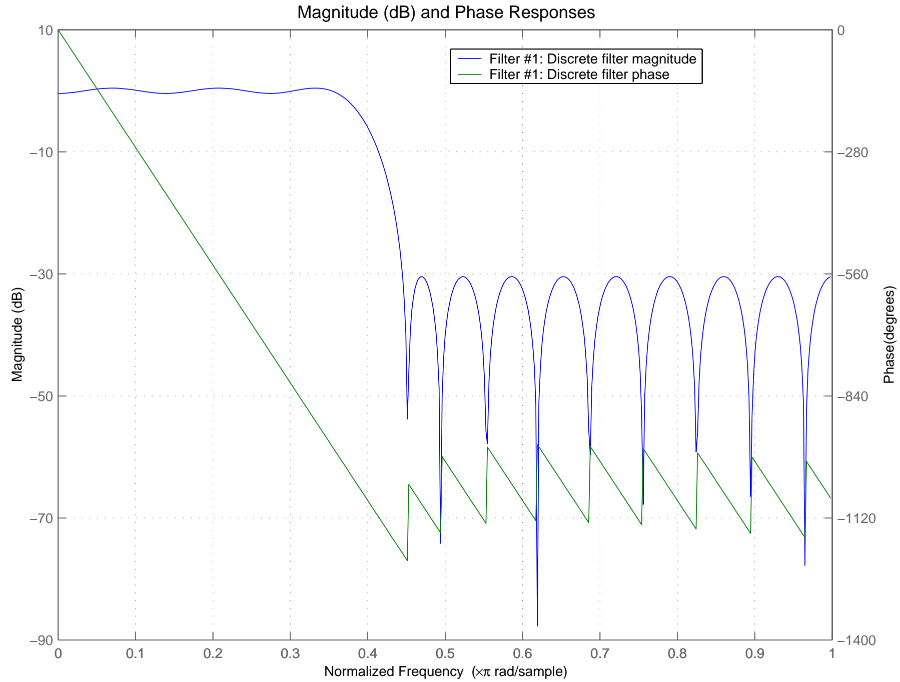
Examples

To introduce a few of the variations on FIR filters that you design with `firceqrip`, these five examples cover both the default syntax `h = firceqrip(n,wo,del)` and some of the optional input arguments. For each example, the input arguments `n`, `wo`, and `del` remain the same.

Example 1—Design an order = 30 FIR filter without using optional input arguments or keywords.

```
h = firceqrip(n,wo,del); fvtool(h)
```

Both the phase and magnitude response for the resulting lowpass filter appear in the plot shown here.



Example 2—Design an order = 30 FIR filter with the **stopedge** keyword to define the response at the edge of the filter stopband.

```
h = firceqrip(n,wo,del,'stopedge'); fvtool(h)
```

Example 3—Design an order = 30 FIR filter with the **slope** keyword and r = 20.

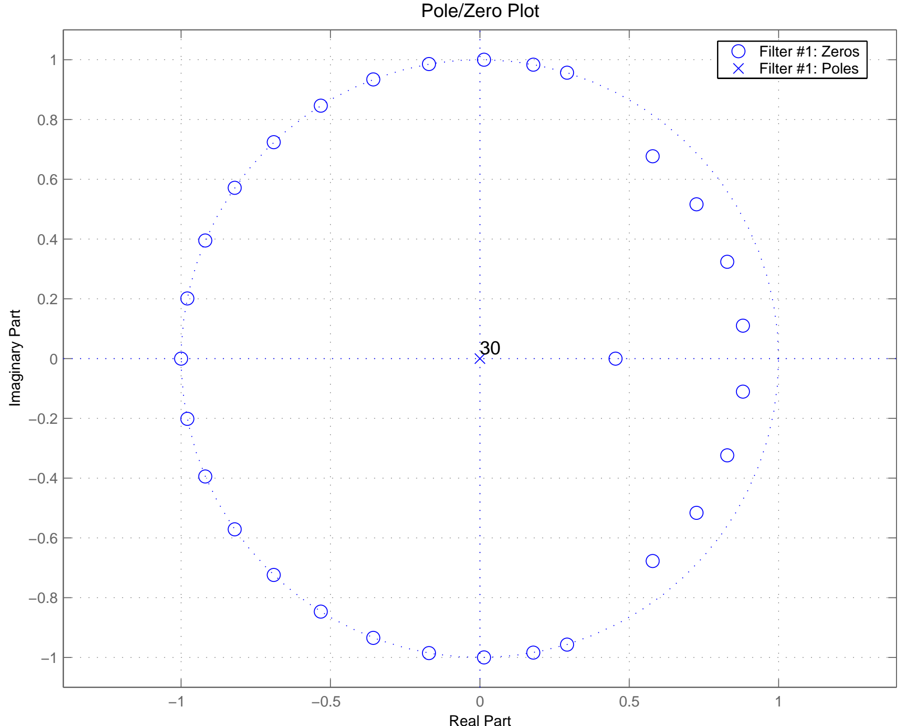
```
h = firceqrip(n,wo,del,'slope',20,'stopedge'); fvtool(h)
```

Example 4—Design an order = 30 FIR filter defining the stopband and specifying that the resulting filter is minimum phase with the **min** keyword.

```
h = firceqrip(n,wo,del,'stopedge','min'); fvtool(h)
```

Comparing this filter to the filter in Example 1, notice that the cutoff frequency `wo = 0.4` now applies to the edge of the stopband rather than the point at which the frequency response magnitude is 0.5.

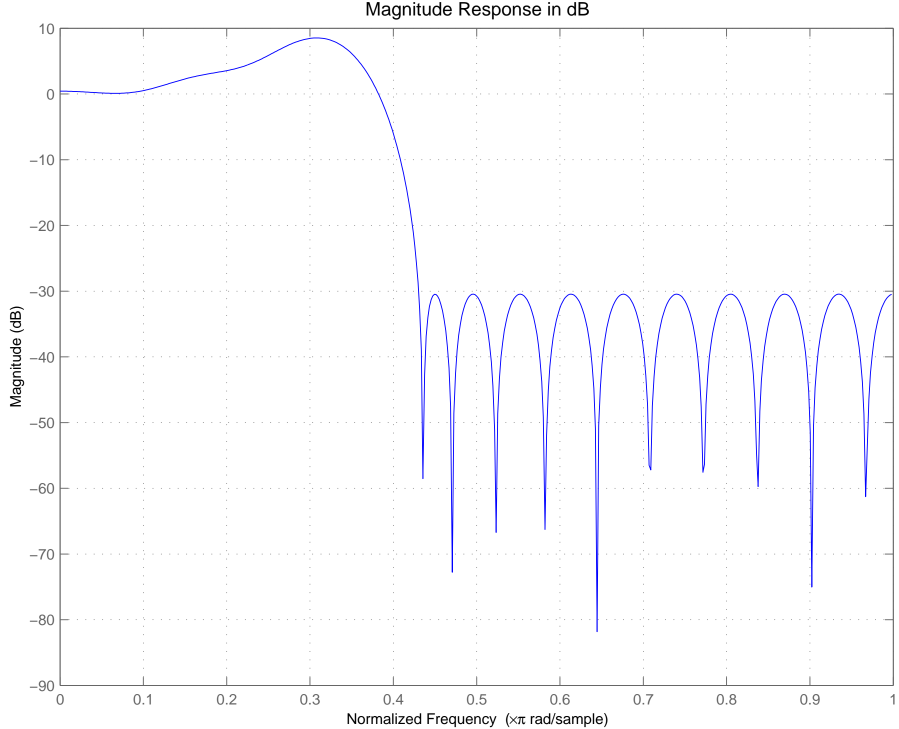
Viewing the zero-pole plot shown here reveals this is a minimum phase FIR filter—the zeros lie on or inside the unit circle, z = 1.



Example 5—Design an order = 30 FIR filter with the **invsinc** keyword to shape the filter passband with an inverse sinc function.

```
h = firceqrip(n,wo,del,'invsinc',[2 1.5]); fvtool(h)
```

With the inverse sinc function being applied defined as 1/sinc(2*w)$^{1.5}$, the figure shows the reshaping of the passband that results from using the **`invsinc`** keyword option, and entering `c` as the two-element vector `[2 1.5]`.



See Also

`firhalfband`, `firnyquist`, `gremez`, `ifir`, `iirgrpdelay`, `iirlpnorm`, `iirlpnormc`

`fircls`, `firls`, `remez` in your Signal Processing Toolbox documentation

Purpose

Design a halfband FIR filter

Syntax

```
b = firhalfband(n,fp)
b = firhalfband(n,win)
b = firhalfband('minorder',fp,dev)
b = firhalfband('minorder',fp,dev,'kaiser')
b = firhalfband(...,'high')
```

Description

`b = firhalfband(n,fp)` designs a lowpass halfband FIR filter of order N with an equiripple characteristic. N must be selected such that N/2 is an odd integer. `fp` determines the passband edge frequency, and it must satisfy 0 < `fp` < 1/2, where 1/2 corresponds to $\pi/2$ rad/sample.

`b = firhalfband(n,win)` designs a lowpass Nth-order filter using the truncated, windowed-impulse response method instead of the equiripple method. `win` is an `n+1` length vector. The ideal impulse response is truncated to length `n + 1`, and then multiplied point-by-point with the window specified in `win`.

`b = firhalfband('minorder',fp,dev)` designs a lowpass minimum-order filter, with passband edge `fp`. The peak ripple is constrained by the scalar `dev`. This design uses the equiripple method.

`b = firhalfband('minorder',fp,dev,'kaiser')` designs a lowpass minimum-order filter, with passband edge `fp`. The peak ripple is constrained by the scalar `dev`. This design uses the Kaiser window method.

`b = firhalfband(...,'high')` returns a highpass halfband FIR filter.

Examples

This example designs a minimum order halfband filter with specified maximum ripple:

```
b=firhalfband('minorder',.45,0.0001);
[h,w,s]=freqz(b); s.plot='mag'; s.yunits = 'li';
fvtool(h,w,s); % Plot magnitude only in linear units
figure;
impz(b) % Impulse response is zero for every other sample
```

See Also

`firnyquist`, `gremez`

`fir1`, `firls`, `remez` in your Signal Processing Toolbox documentation

References

Saramaki, T, “Finite Impulse Response Filter Design,” *Handbook for Digital Signal Processing*. S.K. Mitra and J.F. Kaiser Eds. Wiley-Interscience, N.Y., 1993, Chapter 4.

Purpose

Convert FIR Type I lowpass to to FIR Type 1 lowpass with inverse band width.

Syntax

```
g = firlp2lp(b)
```

Description

`g = firlp2lp(b)` transforms the Type I lowpass FIR filter b with zero-phase response Hr(w) to a Type I lowpass FIR filter g with zero-phase response [1 - Hr(π-w)].

If b is a narrowband filter, g will be a wideband filter and vice versa. The passband and stopband ripples of g will be equal to the stopband and passband ripples of b.

Examples

Overlay the original narrowband lowpass and the resulting wideband lowpass

```
b = gremez(36,[0 .2 .25 1],[1 1 0 0],[1 5]);
zerophase(b);
hold on
h = firlp2lp(b);
zerophase(h); hold off
```

See Also

`firlp2hp`

`zerophase` in your Signal Processing Toolbox documentation

References

[1] Saramaki, T,, Finite Impulse Response Filter Design, *Handbook for Digital Signal Processing*. S.K. Mitra and J.F. Kaiser Eds. Wiley-Interscience, N.Y., 1993, Chapter 4.

firlp2hp

Purpose

Convert FIR Type I lowpass filter to Type I FIR highpass filter

Syntax

```
g = firlp2hp(b)
g = firlp2hp(b,'wide')
```

Description

`g = firlp2hp(b)` transforms the type I lowpass FIR filter b with zero-phase response Hr(w) into a type I highpass FIR filter g with zero-phase response Hr(π-w).

The passband and stopband ripples of g will be equal to the passband and stopband ripples of b.

`g = firlp2hp(b,'wide')` transforms the Type I lowpass FIR filter b with zero-phase response Hr(w) into a Type I highpass FIR filter g with zero-phase response 1 - Hr(w).

For this case, the passband and stopband ripples of g will be equal to the stopband and passband ripples of b.

Examples

Overlay the original narrowband lowpass and the resulting narrowband highpass and wideband highpass

```
b = gremez(36,[0 .2 .25 1],[1 1 0 0],[1 3]);
zerophase(b); hold on;
h = firlp2hp(b);
zerophase(h);
g = firlp2hp(b,'wide');
zerophase(g); hold off
```

See Also

`firlp2lp`

`zerophase` in your Signal Processing Toolbox documentation

References

[1] Saramaki, T,, Finite Impulse Response Filter Design, *Handbook for Digital Signal Processing*. S.K. Mitra and J.F. Kaiser Eds. Wiley-Interscience, N.Y., 1993, Chapter 4.

Purpose

Least P-norm optimal FIR filter design

Syntax

```
b = firlpnorm(n,f,edges,a)
b = firlpnorm(n,f,edges,a,w)
b = firlpnorm(n,f,edges,a,w,p)
b = firlpnorm(n,f,edges,a,w,p,dens)
b = firlpnorm(n,f,edges,a,w,p,dens,initnum)
b = firlpnorm(...,'minphase')
[b,err] = firlpnorm(...)
```

Description

`b = firlpnorm(n,f,edges,a)` returns a filter of numerator order `n` which represents the best approximation to the frequency response described by `f` and `a` in the least-Pth norm sense. P is set to 128 by default, which essentially equivalent to the infinity norm. Vector `edges` specifies the band-edge frequencies for multiband designs. `firlpnorm` uses an unconstrained quasi-Newton algorithm to design the specified filter.

`f` and `a` must have the same number of elements, which can exceed the number of elements in `edges`. This lets you specify filters with any gain contour within each band. However, the frequencies in `edges` must also be in vector `f`. Always use `freqz` to check the resulting filter.

`b = firlpnorm(n,f,edges,a,w)` uses the weights in `w` to weight the error. `w` has one entry per frequency point (the same length as `f` and `a`) which tells `firlpnorm` how much emphasis to put on minimizing the error in the vicinity of each frequency point relative to the other points. For example,

```
b = firlpnorm(20,[0 .15 .4 .5 1],[0 .4 .5 1],...
[1 1.6 1 0 0],[1 1 1 10 10])
```

designs a lowpass filter with a peak of 1.6 within the passband, and with emphasis placed on minimizing the error in the stopband.

`b = firlpnorm(n,f,edges,a,w,p)` where `p` is a two-element vector [`pmin` `pmax`] lets you specify the minimum and maximum values of `p` used in the least-pth algorithm. Default is `[2 128]` which essentially yields the L-infinity, or Chebyshev, norm. `pmin` and `pmax` should be even numbers. The design algorithm starts optimizing the filter with `pmin` and moves toward an optimal filter in the `pmax` sense.When `p` is the string `'`**`inspect`**`'`, `firlpnorm` does not

optimize the resulting filter. You might use this feature to inspect the initial zero placement.

`b = firlpnorm(n,f,edges,a,w,p,dens)` specifies the grid density `dens` used in the optimization. The number of grid points is `[dens*(n+1)]`. The default is 20. You can specify `dens` as a single-element cell array. The grid is equally spaced.

`b = firlpnorm(n,f,edges,a,w,p,dens,initnum)` lets you determine the initial estimate of the filter numerator coefficients in vector `initnum`. This can prove helpful for difficult optimization problems. The pole-zero editor in the Signal Processing Toolbox can be used for generating `initnum`.

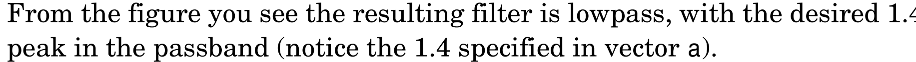
`b = firlpnorm(...,'minphase')` where string '`minphase`' is the last argument in the argument list generates a minimum-phase FIR filter. By default, `firlpnorm` design mixed-phase filters. Specifying input option '`minphase`' causes `firlpnorm` to use a different optimization method to design the minimum-phase filter. As a result of the different optimization used, the minimum-phase filter can yield slightly different results.

`[b,err] = firlpnorm(...)` returns the least-pth approximation error `err`.

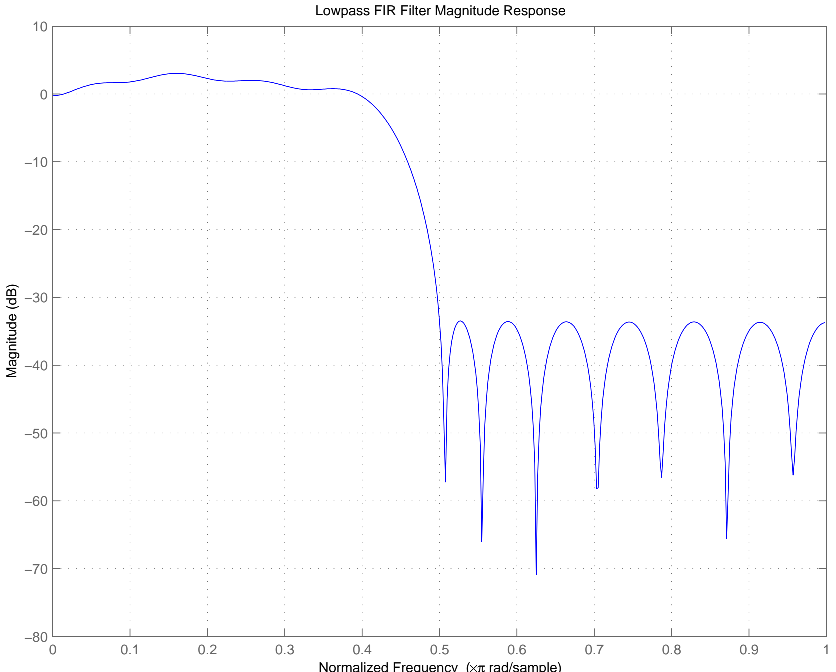
Examples

To demonstrate `firlpnorm`, here are two examples — the first designs a lowpass filter and the second a highpass, minimum-phase filter.

```
% Lowpass filter with a peak of 1.4 in the passband.
b = firlpnorm(22,[0 .15 .4 .5 1],[0 .4 .5 1],[1 1.4 1 0 0],...
[1 1 1 2 2]);
fvtool(b)
```



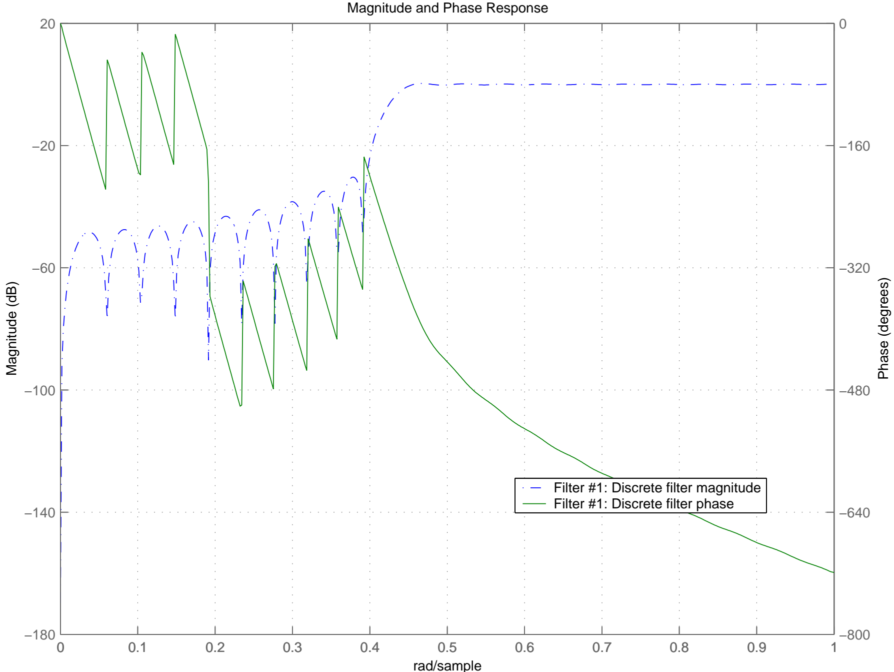
From the figure you see the resulting filter is lowpass, with the desired 1.4 peak in the passband (notice the 1.4 specified in vector a).



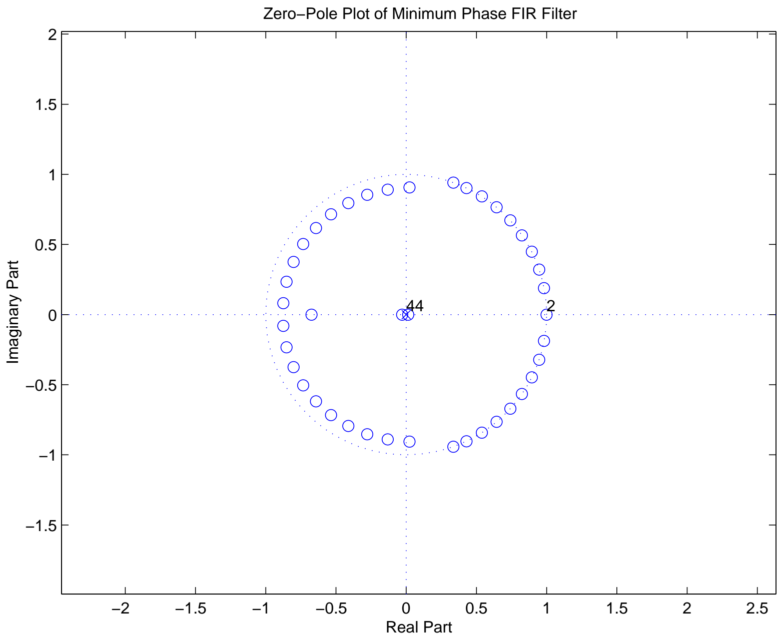
Now for the minimum-phase filter.

```
% Highpass minimum-phase filter optimized for the 4-norm.
b = firlpnorm(44,[0 .4 .45 1],[0 .4 .45 1],[0 0 1 1],[5 1 1 1],...
[2 4],'minphase');
fvtool(b)
```

As shown in the next figure, this is a minimum-phase, highpass filter.



The next zero-pole plot shows the minimum phase nature more clearly.



See Also

`gremez`, `iirgrpdelay`, `iirlpnorm`, `iirlpnormc`
`filter`, `fvtool`, `freqz`, `zplane` in your Signal Processing Toolbox documentation

References

[1] Saramaki, T,, Finite Impulse Response Filter Design, *Handbook for Digital Signal Processing*. S.K. Mitra and J.F. Kaiser Eds. Wiley-Interscience, N.Y., 1993, Chapter 4.

firminphase

Purpose

Compute the minimum-phase FIR spectral factor

Syntax

```
h = firminphase(b)
h = firminphase(b,nz)
```

Description

`h = firminphase(b)` computes the minimum-phase FIR spectral factor `h` of a linear-phase FIR filter `b`. Filter `b` must be real, have even order, and have nonnegative zero-phase response.

`h = firminphase(b,nz)` specifies the number of zeros, `nz`, of `b` that lie on the unit circle. You must specify `nz` as an even number to compute the minimum-phase spectral factor because every root on the unit circle must have even multiplicity. Including `nz` can help `firminphase` calculate the required FIR spectral factor. Zeros with multiplicity greter than two on the unit circle cause problems in the spectral factor determination.

Note You can find the maximum-phase spectral factor, `g`, by reversing `h`, such that $g = fliplr(h)$, and $b = conv(h, g)$.

Example

This example designs a constrained least squares filter with a nonnegative zero-phase response, and then uses `firminphase` to compute the minimum-phase spectral factor.

```
f   = [0 0.4 0.8 1];
a   = [0 1 0];
up  = [0.02 1.02  0.01];
lo  = [0 0.98 0]; % The zeros insure nonnegative zero-phase resp.
n   = 32;
b   = fircls(n,f,a,up,lo);
h   = firminphase(b);
```

See Also

`gremez`
`fircls`, `zerophase` in your Signal Processing Toolbox documentation

References

[1] Saramaki, T,, Finite Impulse Response Filter Design, *Handbook for Digital Signal Processing*. S.K. Mitra and J.F. Kaiser Eds. Wiley-Interscience, N.Y., 1993, Chapter 4.

Purpose

Design a Lowpass Nyquist (L-th band) FIR filter

Syntax

`firnyquist(n,l,r,varargin)`

Description

`b = firnyquist(n,l,r)` designs an N-th order, L-th band, Nyquist FIR filter with a roll-off factor `r` and an equiripple characteristic.

The rolloff factor `r` is related to the normalized transition width `tw` by $tw = 2\pi(r/l)$ (rad/sample). The order, `n`, must be even. `l` must be an integer greater than one. If `l` is not specified, it defaults to `4`. `r` must satisfy `0< r < 1`. If `r` is not specified, it defaults to `0.5`.

`b = firnyquist('`**`minorder`**`',l,r,dev)` designs a minimum-order, L-th band Nyquist FIR filter with a rolloff factor `r` using the Kaiser window. The peak ripple is constrained by the scalar `dev`.

`b = firnyquist(n,l,r,decay)` designs an N-th order, L-th band, Nyquist FIR filter where the scalar `decay`, specifies the rate of decay in the stopband. `decay` must be nonnegative. If omitted or left empty, `decay` defaults to `0` which yields an equiripple stopband. `A` nonequiripple stopband may be desirable for decimation purposes.

`b = firnyquist(n,l,r,'`**`nonnegative`**`')` returns an FIR filter with nonnegative zero-phase response. This filter can be spectrally factored into minimum-phase and maximum-phase “square-root” filters. This allows using the spectral factors in applications such as matched-filtering.

`b = firnyquist(n,l,r,'`**`minphase`**`')` returns the minimum-phase spectral factor `bmin` of order `n`. `bmin` meets the condition `b=conv(bmin,bmax)` so that `b` is an L-th band FIR Nyquist filter of order 2n with rolloff factor `r`. Obtain `bmax`, the maximum phase spectral factor by reversing the coefficients of `bmin`. For example, `bmax = bmin(end:-1:1)`.

Example

Example 1: This example designs a minimum phase factor of a Nyquist filter.

```
bmin = firnyquist(47,10,.45,'minphase');
b = firnyquist(2*47,10,.45,'nonnegative');
[h,w,s] = freqz(b); hmin = freqz(bmin);
fvtool(b,1,bmin,1);
```

Example 2: This example compares filters with different decays.

```
b1 = firnyquist(72,8,.3,0); % Equiripple
b2 = firnyquist(72,8,.3,.5);
b3 = firnyquist(72,8,.3,1);
fvtool(b1,1,b2,1,b3,1);
```

See Also

`firhalfband`, `gremez`, `firminphase`
`firrcos`, `firls` in your Signal Processing Toolbox documentation

References

[1] T. Saramaki, Finite Impulse Response Filter Design, *Handbook for Digital Signal Processing*. S.K. Mitra and J.F. Kaiser Eds. Wiley-Interscience, N.Y., 1993, Chapter 4.

Purpose Return the fraction length for a quantizer

Syntax

```
f = fractionlength(q)
```

Description `f = fractionlength(q)` returns the fraction length of quantizer `q`. This is useful because fraction length is valid whether the quantizer mode is floating-point or fixed-point.

Examples For a floating-point quantizer

```
q = quantizer('float',[32 8]);
f = fractionlength(q);
```

returns $f = 23 = 32 - 8 - 1$.

For a fixed-point quantizer

```
q = quantizer('fixed',[6 4])
f = fractionlength(q);
```

returns $f = 4$.

Algorithm For floating-point quantizers, $f = w - e - 1$, where w is the word length and e is the exponent length.

For fixed-point quantizers, f is part of the format $[w\ f]$.

See Also `quantizer`

freqz

Purpose Compute the frequency response of quantized filters

Syntax

```
[h,w] = freqz(Hq,n)
h = freqz(Hq,w)
[h,w] = freqz(Hq,n,'whole')
[h,w,units,href] = freqz(Hq,...)
[h,f] = freqz(Hq,n,fs)
h = freqz(Hq,f,fs)
[h,f] = freqz(Hq,n,'whole',fs)
[h,f,s] = freqz(Hq,...)
[h,f,units,href] = freqz(Hq,...,fs)
freqz(Hq,...)
```

Description

`[h,w] = freqz(Hq,n)` returns the frequency response vector `h` and the corresponding frequency vector `w` for the quantized filter `Hq`. `freqz` uses the transfer function associated with the quantized filter to calculate the frequency response of the filter. The vectors `h` and `w` are both of length `n`. The frequency vector `w` has values ranging from 0 to π radians per sample. If you do not specify the integer `n`, or you specify it as the empty vector `[]`, the frequency response is calculated using the default value of 512 samples.

`h = freqz(Hq,w)` returns the frequency response vector `h` calculated at the frequencies (in radians per sample) supplied by the vector `w`. The vector `w` can have any length.

`[h,w] = freqz(Hq,n,'whole')` uses `n` sample points around the entire unit circle to calculate the frequency response. Frequency vector `w` has length `n` and values ranging from 0 to 2π radians per sample.

`[h,w,units,href] = freqz(Hq,...)` returns the optional string argument `units`, specifying the units for the frequency vector `w`. The string returned in `units` is `'rad/sample'`, denoting radians per sample. The optional output argument `href` is the frequency response of the transfer function associated with the reference filter used to specify the quantized filter `Hq`.

`[h,f] = freqz(Hq,n,fs)` returns the frequency response vector `h` and the corresponding frequency vector `f` for the quantized filter `Hq`. The vectors `h` and `f` are both of length `n`. The frequency response calculation uses the sampling

frequency specified by the scalar `fs` (in Hz). The frequency vector `f` has values ranging from 0 to `(fs/2)` Hz.

`h = freqz(Hq,f,fs)` returns the frequency response vector `h` calculated at the frequencies (in Hz) supplied in the vector `f`. Vector `f` can be any length.

`[h,f] = freqz(Hq,n,'whole',fs)` uses `n` points around the entire unit circle to calculate the frequency response. Frequency vector `f` has length `n` and has values ranging from 0 to `fs` Hz.

`[h,f,s] = freqz(Hq,...)` returns the structure `s` with the following fields:

- `s.xunits`—a string specifying the frequency axis units. The contents of `s.xunits` can be one of the following:
  - `'rad/sample'` (default)
  - `'Hz'`
  - `'kHz'`
  - `'MHz'`
  - `'GHz'`
  - A user-specified string
- `s.yunits`—a string specifying the vertical axis units. The contents of `s.yunits` can be one of the following:
  - `'dB'` (default)
  - `'linear'`
  - `'squared'`
- `s.plot`—a string specifying the type of plot to produce. The contents of `s.plot` can be one of the following:
  - `'both'` (default)
  - `'mag'`
  - `'phase'`

`[h,f,units,href] = freqz(Hq,...,fs)` returns the optional MATLAB structure `units`, that `freqzplot` uses for plotting. The string returned in `units` is `'Hz'` for hertz. The optional output argument `href` is the frequency response of the transfer function associated with the reference filter used to specify the quantized filter `Hq`.

`freqz(Hq,...)` plots the magnitude and unwrapped phase of the frequency response of the quantized filter `Hq` in the current figure window.

Remarks

There are several ways of analyzing the frequency response of quantized filters. `freqz` accounts for quantization effects in the filter coefficients, but does not account for quantization effects in filtering arithmetic. To account for the quantization effects in filtering arithmetic, refer to function `nlm`.

Algorithm

`freqz` calculates the frequency response for a quantized filter from the filter transfer function $H_q(z)$. The complex-valued frequency response is calculated by evaluating $H_q(e^{j\omega})$ at discrete values of w specified by the syntax you use. The integer input argument `n` determines the number of equally-spaced points around the upper half of the unit circle at which `freqz` evaluates the frequency response. The frequency ranges from 0 to π radians per sample when you do not supply a sampling frequency as an input argument. When you supply the scalar sampling frequency `fs` as an input argument to `freqz`, the frequency ranges from 0 to `fs`/2 Hz.

To calculate the transfer function associated with a quantized filter, `freqz` uses the values of the `QuantizedCoefficients` and `FilterStructure` properties.

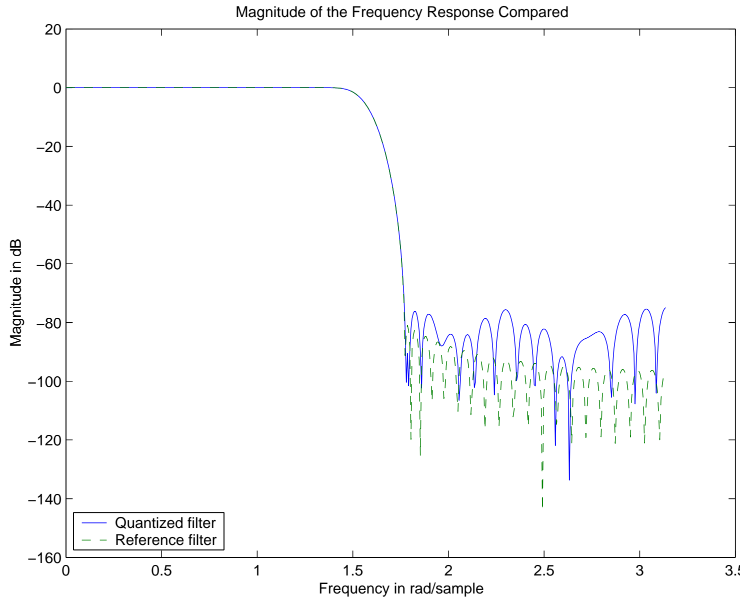
When you include the optional output argument `href` in the command, `freqz` uses the value of the `ReferenceCoefficients` property to calculate the frequency response of the reference filter transfer function.

Examples

Plot the estimated frequency response of a quantized filter.

```
b = fir1(80,0.5,kaiser(81,8));
Hq = qfilt('fir',{b});
[h,w,units,href] = freqz(Hq);
plot(w,20 * log10(abs(h)),'-',w,20 * log10(abs(href)),'--')
legend('Quantized filter','Reference filter',3)
xlabel('Frequency in rad/sample')
ylabel('Magnitude in dB')
```

```
title('Magnitude of the Frequency Response Compared')
```



See Also

`qfilt`
`fvtool` in your Signal Processing Toolbox documentation

get

Purpose

Return the property values for quantized filters, quantizers, and quantized FFTs

Syntax

```
get(obj,pn,pv)
get(hq)
struct = get(hq)
v = get(hq,'propertyname')
value = get(f, 'propertyname')
structure = get(f)
value = get(q, 'propertyname')
structure = get(q)
```

Description

`get(obj,pn,pv)` displays the property names and property values associated with *`obj`*, where *`obj`* is one of the following:

- A quantizer
- A quantized filter
- A quantized FFT

`pn` is the name of a property of the object `obj`, and `pv` is the value associated with `pn`.

`get(hq)` displays a list of the property names and property values associated with quantized filter hq.

`struct = get(hq)` returns the MATLAB structure `struct`, a list of the properties associated with the quantized filter `hq`, along with the properties' associated values. Each field associated with `struct` is named according to the corresponding property name.

`v = get(hq,'propertyname')` returns the property value `v` associated with the property named in the string `'propertyname'` for the quantized filter `hq`. If you replace the string `'propertyname'` by a cell array of a vector of strings containing property names, `get` returns a cell array of a vector of corresponding values.

`value = get(f, 'propertyname')` returns the property value `value` associated with the property named in the string `'propertyname'` for the quantized FFT `f`. If you replace the string `'propertyname'` by a cell array of a

vector of strings containing property names, `get` returns a cell array of a vector of corresponding values.

`structure = get(f)` returns a structure containing the properties and states of quantized FFT `f`.

`value = get(q, 'propertyname')` returns the property value `value` associated with the property named in the string `'propertyname'` for the quantizer `q`. If you replace the string `'propertyname'` by a cell array of a vector of strings containing property names, `get` returns a cell array of a vector of corresponding values.

`structure = get(q)` returns a structure containing the properties and states of quantizer `q`.

Remarks

For more information on the properties associated with quantized filters, see "A Quick Guide to Quantized Filter Properties" on page 12-10. For more information on the properties associated with quantized FFTs, see "A Quick Guide to Quantized FFT Properties" on page 12-51. For more information on the properties associated with quantizers, refer to "A Quick Guide to Quantizer Properties" on page 12-2.

Examples

Use `get` to list the properties of quantized filter `hq`, along with the property values. Then retrieve the value associated with the `OutputFormat` property for this filter in a structure `v`.

```
hq = qfilt;
get(hq)

Quantized Direct form II transposed filter
Numerator
     QuantizedCoefficients{1}    ReferenceCoefficients{1}
+ (1)       0.999969482421875  1.000000000000000000
Denominator
     QuantizedCoefficients{2}    ReferenceCoefficients{2}
+ (1)       0.999969482421875  1.000000000000000000

   FilterStructure = df2t
       ScaleValues = [1]
  NumberOfSections = 1
  StatesPerSection = [0]
 CoefficientFormat = quantizer('fixed', 'round', 'saturate', [16  15])
       InputFormat = quantizer('fixed', 'floor', 'saturate', [16  15])
```

```
      OutputFormat = quantizer('fixed', 'floor', 'saturate', [16  15])
MultiplicandFormat = quantizer('fixed', 'floor', 'saturate', [16  15])
     ProductFormat = quantizer('fixed', 'floor', 'saturate', [32  30])
         SumFormat = quantizer('fixed', 'floor', 'saturate', [32  30])
Warning: 2 overflows in coefficients.


v = get(hq,'OutputFormat')

v =

            Mode = fixed
       RoundMode = floor
    OverflowMode = saturate
          Format = [16  15]

             Max = reset
             Min = reset
      NOverflows = 0
     NUnderflows = 0
     NOperations = 0
```

```
q = quantizer('fixed', 'floor', 'saturate', [16 15])
struct      = get(q)
mode        = get(q, 'mode')
format      = get(q, 'format')
noverflows  = get(q, 'noverflows')
```

get also supports the dot notation for setting and accessing properties.

```
q = quantizer('fixed', 'floor', 'saturate', [16 15])
struct      = get(q)
mode        = q.mode
format      = q.format
noverflows  = q.noverflows
```

See Also

qfft, qfilt, quantizer, set

Purpose Use the Parks-McClellan technique to design digital FIR filters

Syntax

```
b = gremez(n,f,a,w)
b = gremez(n,f,a,'hilbert')
b = gremez(n,f,a,'differentiator')
b = gremez(m,f,a,r)
b = gremez({m,ni},f,a,r)
b = gremez(n,f,a,w,c)
b = gremez(n,f,a,w,e)
b = gremez(n,f,a,s)
b = gremez(n,f,a,s,w,e)
```

Description `gremez` is a minimax filter design algorithm you use to design the following types of real FIR filters:

- Types 1-4 linear phase:
  - Type 1 is even order, symmetric
  - Type 2 is odd order, symmetric
  - Type 3 is even order, antisymmetric
  - Type 4 is odd order, antisymmetric
- Minimum phase
- Maximum phase
- Minimum order (even or odd)
- Extra ripple
- Maximal ripple
- Constrained ripple
- Single-point band (notching and peaking)
- Forced gain
- Arbitrary shape frequency response curve filters

`b = gremez(n,f,a,w)` returns a length `n+1` linear phase FIR filter which has the best approximation to the desired frequency response described by `f` and `a` in the minimax sense. `w` is a vector of weights, one per band. When you omit `w`, all bands are weighted equally. For more information on the input arguments, refer to `remez` in *Signal Processing Toolbox User's Guide*.

`b = gremez(n,f,a,'hilbert')` and `b = gremez(n,f,a,'differentiator')` design FIR Hilbert transformers and differentiators. For more information on designing these filters, refer to `remez` in *Signal Processing Toolbox User's Guide*.

`b = gremez(m,f,a,r)`, where `m` is one of '`minorder`', '`mineven`' or '`minodd`', designs filters repeatedly until the minimum order filter, as specified in `m`, that meets the specifications is found. `r` is a vector containing the peak ripple per frequency band. You must specify `r`. When you specify '`mineven`' or '`minodd`', the minimum even or odd order filter is found.

`b = gremez({m,ni},f,a,r)` where `m` is one of '`minorder`', '`mineven`' or '`minodd`', uses `ni` as the initial estimate of the filter order. `ni` is optional for common filter designs, but it must be specified for designs in which `remezord` cannot be used, such as while designing differentiators or Hilbert transformers.

`b = gremez(n,f,a,w,c)` designs filters having constrained error magnitudes (ripples). `c` is a cell array of strings of length `w`. The entries of `c` must be either '`c`' to indicate that the corresponding element in `w` is a constraint (the ripple for that band cannot exceed `w`) or '`w`' indicating that the corresponding entry in `w` is a weight. There must be at least one unconstrained band—`c` must contain at least one '`w`' entry. For example,

`b = gremez(12,[0 0.4 0.5 1], [1 1 0 0], [1 0.2], {'w' 'c'})` uses a weight of one in the passband, and constrains the stopband ripple to 0.2 or less.

A hint about using constrained values: if the resulting filter does not touch the constraints, increase the error weighting you apply to the unconstrained bands.

`b = gremez(n,f,a,w,e)` specifies independent approximation errors for different bands. Use this syntax to design extra ripple or maximal ripple filters. These filters have interesting properties such as having the minimum transition width. `e` is a cell array of strings specifying the approximation errors to use. Its length must equal the number of bands. Entries of `e` must be in the form '`e#`' where `#` indicates which approximation error to use for the corresponding band. For example, when `e = {'e1','e2','e1'}`, the first and third bands use the same approximation error `'e1'` and the second band uses a different one `'e2'`. Note that when all bands use the same approximation

error, such as `{'e1','e1','e1',...}`, it is equivalent to omitting `e`, as in `b = gremez(n,f,a,w)`.

`b = gremez(n,f,a,s)` is used to design filters with special properties at certain frequency points. `s` is a cell array of strings and must be the same length as `f` and `a`. Entries of `s` must be one of:

- `'n'` - normal frequency point.
- `'s'` - single-point band. The frequency "band" is given by a single point. The corresponding gain at this frequency point must be specified in `a`.
- `'f'` - forced frequency point. Forces the gain at the specified frequency band to be the value specified.
- `'i'` - indeterminate frequency point. Use this argument when adjacent bands abut one another (no transition region).

For example, the following command designs a bandstop filter with zero-valued single-point stop bands (notches) at 0.25 and 0.55.

```
b = gremez(42,[0 0.2 0.25 0.3 0.5 0.55 0.6 1],[1 1 0 1 1 0 1 1],...
{'n' 'n' 's' 'n' 'n' 's' 'n' 'n'})
```

```
b = gremez(82,[0 0.055 0.06 0.1 0.15 1],[0 0 0 0 1 1],...
{'n' 'i' 'f' 'n' 'n' 'n'})
```

designs a highpass filter with the gain at 0.06 forced to be zero. The band edge at 0.055 is indeterminate since the first two bands actually touch. The other band edges are normal.

`b = gremez(n,f,a,s,w,e)` specifies weights and independent approximation errors for filters with special properties. The weights and properties are included in vectors `w` and `e`. Sometimes, you may need to use independent approximation errors to get designs with forced values to converge. For example,

```
b = gremez(82,[0 0.055 0.06 0.1 0.15 1], [0 0 0 0 1 1],...
{'n' 'i' 'f' 'n' 'n' 'n'}, [10 1 1] ,{'e1' 'e2' 'e3'});
```

`b = gremez(...,'1')` designs a type 1 filter (even-order symmetric). You can specify type 2 (odd-order symmetric), type 3 (even-order antisymmetric), and type 4 (odd-order antisymmetric) filters as well. Note that restrictions apply to `a` at f=0 or f=1 for FIR filter types 2, 3, and 4.

`b = gremez(...,'minphase')` designs a minimum-phase FIR filter. You can use the argument `'maxphase'` to design a maximum phase FIR filter.

`b = gremez(..., 'check')` returns a warning when there are potential transition-region anomalies.

`b = remez(...,{lgrid})`, where `{lgrid}` is a scalar cell array. The value of the scalar controls the density of the frequency grid by setting the number of samples used along the frequency axis.

`[b,err] = gremez(...)` returns the unweighted approximation error magnitudes. `err` contains one element for each independent approximation error returned by the function.

`[b,err,res] = gremez(...)` returns the structure `res` comprising optional results computed by `gremez`. `res` contains the following fields.

Structure Field	Contents
`res.fgrid`	Vector containing the frequency grid used in the filter design optimization
`res.des`	Desired response on `fgrid`
`res.wt`	Weights on `fgrid`
`res.h`	Actual frequency response on the frequency grid
`res.error`	Error at each point (desired response - actual response) on the frequency grid
`res.iextr`	Vector of indices into `fgrid` of extremal frequencies
`res.fextr`	Vector of extremal frequencies
`res.order`	Filter order

Structure Field	Contents
`res.edgecheck`	Transition-region anomaly check. One element per band edge. Element values have the following meanings: 1 = OK 0 = probable transition-region anomaly -1 = edge not checked Computed when you specify the `'check'` input option in the function syntax.
`res.iterations`	Number of Remez iterations for the optimization
`res.evals`	Number of function evaluations for the optimization

`gremez` is also a “function function”, allowing you to write a function that defines the desired frequency response.

`b = gremez(n,f,fresp,w)` returns a length N+1 FIR filter which has the best approximation to the desired frequency response as returned by the user-defined function `fresp`. `gremez` uses the following syntax to call `fresp`

```
[dh,dw] = fresp(n,f,gf,w)
```

where:

- `fresp` is the string variable that identifies the function that you use to define your desired filter frequency response.
- `n` is the filter order.
- `f` is the vector of frequency band edges which must appear monotonically between 0 and 1, where 1 is one-half of the sampling frequency. The frequency bands span `f(k)` to `f(k+1)` for `k` odd. The intervals `f(k+1)` to `f(k+2)` for `k` odd are “transition bands” or “don't care” regions during optimization.

- `gf` is a vector of grid points that have been chosen over each specified frequency band by `gremez`, and determines the frequencies at which `gremez` evaluates the response function.
- `w` is a vector of real, positive weights, one per band, for use during optimization. `w` is optional in the call to `gremez`. If you do not specify `w`, it is set to unity weighting before being passed to `fresp`.
- `dh` and `dw` are the desired frequency response and optimization weight vectors, evaluated at each frequency in grid `gf`.

`gremez` includes a predefined frequency response function named `'remezfrf2'`. You can write your own based on the simpler `'remezfrf'`. See the help for `private/remezfrf` for more information.

`b = gremez(n,f,{fresp,p1,p2,...},w)` specifies optional arguments `p1`, `p2`,..., `pn` to be passed to the response function `fresp`.

`b = gremez(n,f,a,w)` is a synonym for `b = gremez(n,f,{'remezfrf2',a},w)`, where `a` is a vector containing your specified response amplitudes at each band edge in `f`. By default, `gremez` designs symmetric (even) FIR filters. `'remezfrf2'` is the predefined frequency response function. If you do not specify your own frequency response function (the `fresp` string variable), `gremez` uses `'remezfrf2'`.

`b = gremez(...,'h')` and `b = gremez(...,'d')` design antisymmetric (odd) filters. When you omit the `'h'` or `'d'` arguments from the `gremez` command syntax, each frequency response function `fresp` can tell `gremez` to design either an even or odd filter. Use the command syntax `sym = fresp('defaults',{n,f,[],w,p1,p2,...})`. `gremez` expects `fresp` to return `sym = 'even'` or `sym = 'odd'`. If `fresp` does not support this call, `gremez` assumes even symmetry.

For more information about the input arguments to `gremez`, refer to `remez`.

See Also

`remez`, `cremez`, `butter`, `cheby1`, `cheby2`, `ellip`, `freqz`, `filter`, `firls`, and `fircls` in your Signal Processing Toolbox documentation

Reference

Shpak, D.J. and A. Antoniou, "A generalized Remez method for the design of FIR digital filters," *IEEE Trans. Circuits and Systems*, pp. 161-174,Feb. 1990.

hex2num

Purpose

Convert hexadecimal string to a number

Syntax

```
x = hex2num(q,h)
[x1,x2,...] = hex2num(q,h1,h2,...)
```

Description

`x = hex2num(q,h)` converts hexadecimal string `h` to numeric matrix `x`. The attributes of the numbers in `x` are specified by quantizer `q`. When `h` is a cell array containing hexadecimal strings, `hex2num` returns `x` as a cell array of the same dimension containing numbers. For fixed-point hexadecimal strings, `hex2num` uses two's complement representation. For floating-point strings, the representation is IEEE Standard 754 style.

When there are fewer hexadecimal digits than needed to represent the number, the fixed-point conversion zero fills on the left. Floating-point conversion zero fills on the right.

`[x1,x2,...] = hex2num(q,h1,h2,...)` converts hexadecimal strings `h1`, `h2`,... to numeric matrices `x1`, `x2`,....

`hex2num` and `num2hex` are inverses of one another, with the distinction that `num2hex` returns the hexadecimal strings in a column.

Examples

To create all of the 4-bit fixed-point two's complement numbers fractional form, use the following code.

```
q = quantizer([4 3]);
h = '7 3 F B';'6 2 E A';'5 1 D 9';'4 0 C 8'];
x = hex2num(q,h)
x =

    0.8750    0.3750   -0.1250   -0.6250
    0.7500    0.2500   -0.2500   -0.7500
    0.6250    0.1250   -0.3750   -0.8750
    0.5000         0   -0.5000   -1.0000
```

See Also

`num2hex`, `bin2num`, `num2bin`

Purpose

Apply the inverse quantized FFT to data

Syntax

```
y = ifft(f,x)
y = ifft(f,x,dim)
```

Description

`y = ifft(f,x)` is the quantized inverse FFT of *x*. The parameters of the quantized FFT are specified in quantized FFT *f*.

`y = ifft(f,x,dim)` is the quantized inverse FFT of *x* across the dimension `dim`.

See Also

`fft`, `get`, `qfft`, `qreport`, `set`

Purpose

Design interpolated FIR filters

Syntax

```
h = ifir(l,type,f,dev)
h = ifir(l,type,f,dev,str)
```

Description

`h = ifir(l,type,f,dev)` finds a periodic filter $f(z^l)$ and an image-suppressor filter *G(z)* such that

$$h = f(z^l)G(z)$$

l is the interpolation factor.

`h` represents the optimal minimax FIR approximation to the desired response specified by the string `type`. Specify the filter band edge frequencies in vector `f`. With `ifir`, you designs a filter that meets the response defined by `type` which does not exceed the peak ripple specified in vector `dev`.

`type` must be a string with either '**`low`**' to generate lowpass filters or '**`high`**' for highpass filters. `f` must be a two-element vector containing two values — the first defining the passband edge frequency and the second that defines the stopband edge frequency. Vector `dev` must contain two values that specify the peak ripple or deviation allowed in the passband and stopband.

`h = ifir(l,type,f,dev,str)` uses the string specified in `str` to select the degree of optimization the interpolation algorithm uses. `str` can be one of three allowed strings:

str String Value	Description
'**`simple`**'	
'**`intermediate`**'	
'**`advanced`**'	

`str` lets you direct the filter design algorithm to trade between the time it takes to design the filter and optimizing the filter order. The '**`advanced`**' option can substantially reduce the filter order, especially for *g(z)*.

Examples

The first example creates a lowpass filter using `ifir` with an interpolation factor of 6. In example 2, the code designs a wideband highpass filter with the same interpolation factor. You can see the plots of the examples after the code sections.

Create a narrowband lowpass design using an interpolation factor of 6.

```
[h,g]=ifir(6,'low',[.12 .14],[.01 .001]);
[Hh,w]=freqz(h,1,1024); Hg=freqz(g,1,1024);
H = Hh.*Hg; % Compounded response
subplot(2,1,1), freqzplot([Hh,Hg],w,'mag');
legend('Periodic Filter','Image Suppressor Filter');
subplot(2,1,2), freqzplot(H,w,'mag');
legend('Overall Filter');
```

Use the '**high**' option to create a wideband highpass design using an interpolation factor of 6.

```
[h,g,d]=ifir(6,'high',[.12 .14],[.001 .01]);
[Hh,w]=freqz(h,1,1024); Hg=freqz(g,1,1024);
H = Hh.*Hg; % Branch 1 compounded response
Hd = freqz(d,1,1024); % Branch 2 response
Hoverall = H+Hd;
freqzplot(Hoverall,w,'mag');
title('Overall Filter');
```

iirbpc2bpc

Purpose Transform an IIR complex bandpass filter to an IIR complex bandpass filter with different frequency response characteristics

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirbpc2bpc(B,A,Wo,Wt)
```

Description `[Num,Den,AllpassNum,AllpassDen] = iirbpc2bpc(B,A,Wo,Wt)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the complex bandpass prototype by applying a first-order complex bandpass to complex bandpass frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with the numerator specified by `B` and the denominator specified by `A`.

This transformation effectively places two features of an original filter, located at frequencies W_{o1} and W_{o2}, at the required target frequency locations, W_{t1}, and W_{t2} respectively. It is assumed that W_{t2} is greater than W_{t1}. In most of the cases the features selected for the transformation are the band edges of the filter passbands. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

This transformation can also be used for transforming other types of filters; e.g., complex notch filters or resonators can be repositioned at two distinct desired frequencies at any place around the unit circle; e.g., in the adaptive system.

Examples Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

Create a complex passband from 0.25 to 0.75:

```
[b, a] = iirlp2bpc (b, a, 0.5, [0.25,0.75]);
[num, den] = iirbpc2bpc(b, a, [0.25, 0.75], [-0.5, 0.5]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

Arguments

`B`
Numerator of the prototype lowpass filter

`A`
Denominator of the prototype lowpass filter

`Wo`
Frequency values to be transformed from the prototype filter

`Wt`
Desired frequency locations in the transformed target filter

`Num`
Numerator of the target filter

`Den`
Denominator of the target filter

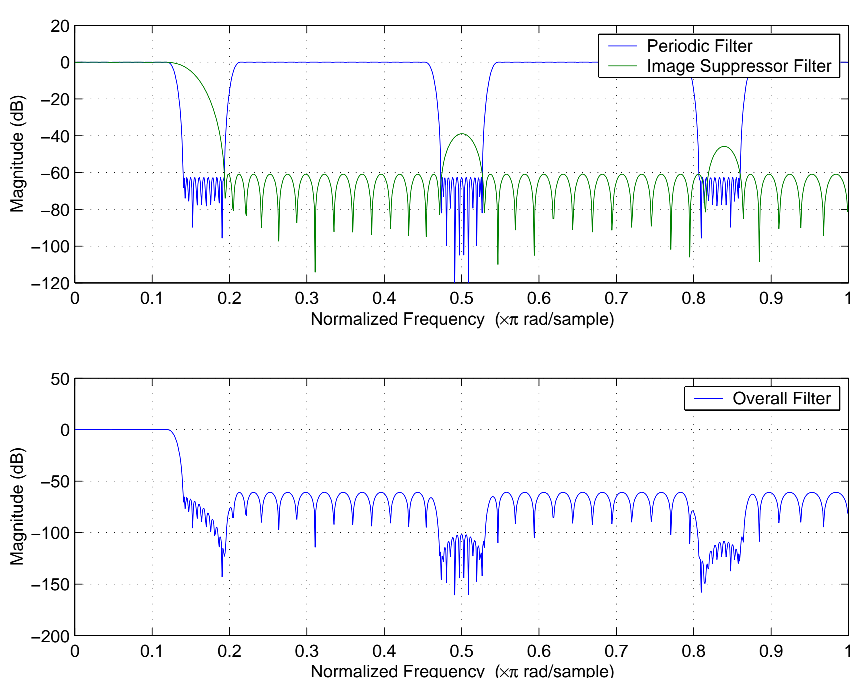
`AllpassNum`
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

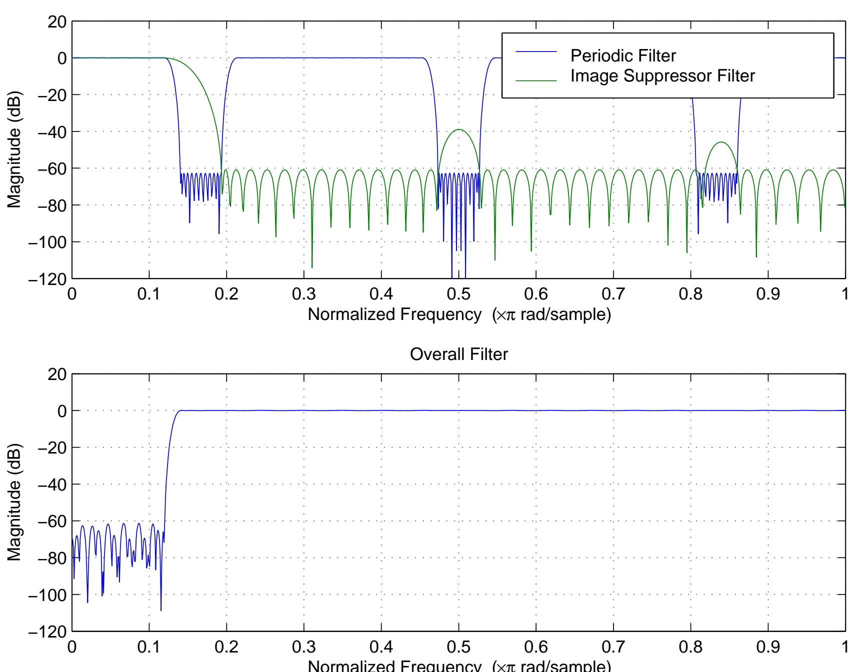
Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

See Also

`iirftransf`, `allpassbpc2bpc`, `zpkbpc2bpc`



20
0
−20
−40
−60
−80
−100
−120
Magnitude (dB)
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
Normalized Frequency (×π rad/sample)
Periodic Filter
Image Suppressor Filter
50
0
−50
−100
−150
−200
Magnitude (dB)
Overall Filter



See Also

`gremez`

`fir1`, `firls`, `remez` in your Signal Processing Toolbox documentation

References

[1] Saramaki, T,, Finite Impulse Response Filter Design, *Handbook for Digital Signal Processing*. S.K. Mitra and J.F. Kaiser Eds. Wiley-Interscience, N.Y., 1993, Chapter 4.

iircomb

Purpose Design an IIR comb notching or peaking digital filter

Syntax

```
[num,den] = iircomb(n,bw)
[num,den] = iircomb(n,bw,ab)
[num,den] = iircomb( , 'type')
```

Description `[num,den] = iircomb(n,bw)` returns a digital notching filter with order `n` and with the width of the filter notch at -3dB set to `bw`, the filter bandwidth. The filter order must be a positive integer. `n` also defines the number of notches in the filter across the frequency range from 0 to 2π—the number of notches equals `n+1`.

For the notching filter, the transfer function takes the form

$$H(z) = b \times \frac{1 - z^{-n}}{1 - az^{-n}}$$

where a and b are the filter coefficients and n is the filter order or the number of notches in the filter minus 1.

The quality factor (Q factor) q for the filter is related to the filter bandwidth by q = ω_0/bw where ω_0 is the frequency to remove from the signal.

`[num,den] = iircomb(n,bw,ab)` returns a digital notching filter whose bandwidth, `bw`, is specified at a level of `-ab` decibels. Including the optional input argument `ab` lets you specify the magnitude response bandwidth at a level that is not the default -3dB point, such as -6 dB or 0 dB.

`[num,den] = iircomb( ,'type')` returns a digital filter of the specified type. The input argument `type` can be either

- `'notch'` to design an IIR notch filter. Notch filters attenuate the response at the specified frequencies. This is the default type. When you omit the `type` input argument, `iircomb` returns a notch filter.
- `'peak'` to design an IIR peaking filter. Peaking filters boost the signal at the specified frequencies.

The transfer function for peaking filters is

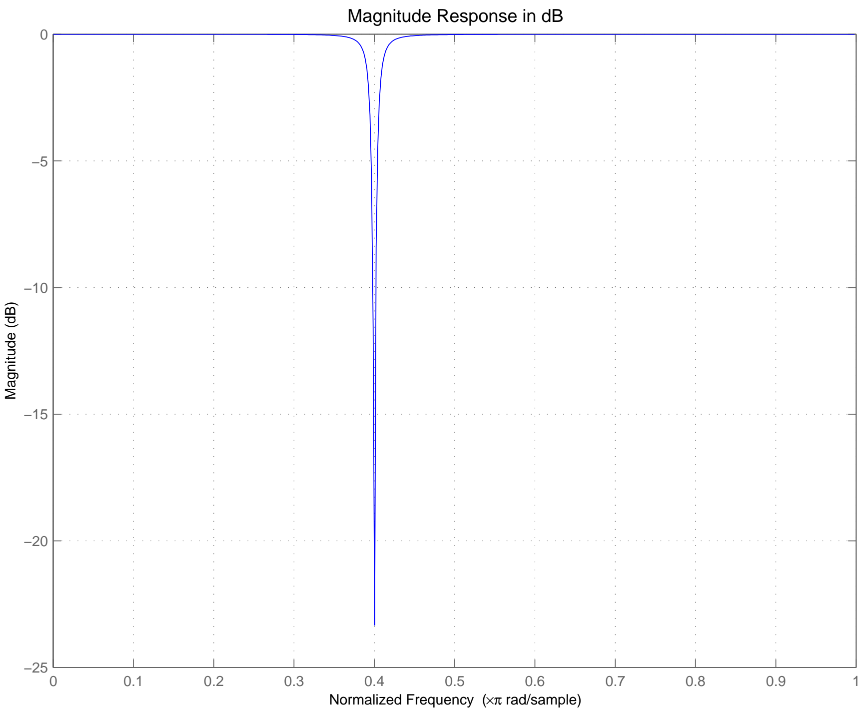
$$H(z) = b \times \frac{1 + z^{-n}}{1 - az^{-n}}$$

Examples

Design and plot an IIR notch filter with 11 notches (equal to filter order plus 1) that removes a 60 Hz tone (`f0`) from a signal at 600 Hz (`fs`). For this example, set the Q factor for the filter to 35 and use it to specify the filter bandwidth.

```
fs = 600; fo = 60;  q = 35; bw = (fo/(fs/2))/q;
[b,a] = iircomb(fs/fo,bw,'notch'); % Note the type flag 'notch'
fvtool(b,a);
```

Using the Filter Visualization Tool (FVTool) generates the following plot showing the filter notches. Note the notches are evenly spaced and one falls at exactly 60 Hz.



See Also

`gremez`, `iirnotch`, `iirpeak`

Purpose

IIR frequency transformation of the digital filter

Syntax

```
[OutNum,OutDen] = iirftransf(OrigNum,OrigDen,FTFNum,FTFDen)
```

Description

`[OutNum,OutDen] = iirftransf(OrigNum,OrigDen,FTFNum,FTFDen)` returns the numerator and denominator vectors, `OutNum` and `OutDen`, of the target filter, which is the result of transforming the prototype filter specified by the numerator, `OrigNum`, and denominator, `OrigDen`, with the mapping filter given by the numerator, `FTFNum`, and the denominator, `FTFDen`. If the allpass mapping filter is not specified, then the function returns an original filter.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
[AlpNum, AlpDen] = allpasslp2lp(0.5, 0.25);
[num, den] = iirftransf(b, a, AlpNum, AlpDen);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

Arguments

`OrigNum`
Numerator of the prototype lowpass filter

`OrigDen`
Denominator of the prototype lowpass filter

`FTFNum`
Numerator of the mapping filter

`FTFDen`
Denominator of the mapping filter

`OutNum`
Numerator of the target filter

`OutDen`
Denominator of the target filter

See Also

`zpkftransf`

iirgrpdelay

Purpose

Optimal IIR filter design with prescribed group-delay

Syntax

```
[num,den] = iirgrpdelay(n,f,edges,a)
[num,den] = iirgrpdelay(n,f,edges,a,w)
[num,den] = iirgrpdelay(n,f,edges,a,w,radius)
[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p)
[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p,dens)
[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p,dens,initden)
[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p,dens,initden,tau)
[num,den,tau] = iirgrpdelay(n,f,edges,a,w)
```

Description

`[num,den] = iirgrpdelay(n,f,edges,a)` returns an allpass IIR filter of order `n` (`n` must be even) which is the best approximation to the relative group-delay response described by `f` and `a` in the least-pth sense. `f` is a vector of frequencies between 0 and 1 and `a` is specified in samples. The vector `edges` specifies the band-edge frequencies for multi-band designs. `iirgrpdelay` uses a constrained Newton-type algorithm. Always check your resulting filter using `grpdelay` or `freqz`.

`[num,den] = iirgrpdelay(n,f,edges,a,w)` uses the weights in `w` to weight the error. `w` has one entry per frequency point and must be the same length length as `f` and `a`). Entries in `w` tell `iirgrpdelay` how much emphasis to put on minimizing the error in the vicinity of each specified frequency point relative to the other points.

`f` and `a` must have the same number of elements. `f` and `a` can contains more elements than the vector `edges` contains. This lets you use `f` and `a` to specify a filter that has any group-delay contour within each band.

`[num,den] = iirgrpdelay(n,f,edges,a,w,radius)` returns a filter having a maximum pole radius equal to `radius`, where 0<`radius`<1. `radius` defaults to 0.999999. Filters whose pole radius you constrain to be less than 1.0 can better retain transfer function accuracy after quantization.

`[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p)`, where `p` is a two-element vector `[pmin pmax]`, lets you determine the minimum and maximum values of `p` used in the least-pth algorithm. `p` defaults to [2 128] which yields filters very similar to the L-infinity, or Chebyshev, norm. `pmin` and

pmax should be even. If p is the string 'inspect', no optimization occurs. You might use this feature to inspect the initial pole/zero placement.

[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p,dens) specifies the grid density dens used in the optimization process. The number of grid points is (dens*(n+1)). The default is 20. dens can be specified as a single-element cell array. The grid is not equally spaced.

[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p,dens,initden) allows you to specify the initial estimate of the denominator coefficients in vector initden. This can be useful for difficult optimization problems. The pole-zero editor in the Signal Processing Toolbox can be used for generating initden.

[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p,dens,initden,tau) allows the initial estimate of the group delay offset to be specified by the value of tau, in samples.

[num,den,tau] = iirgrpdelay(n,f,edges,a,w) returns the resulting group delay offset. In all cases, the resulting filter has a group delay that approximates [a + tau]. Allpass filters can have only positive group delay and a non-zero value of tau accounts for any additional group delay that is needed to meet the shape of the contour specified by (f,a). The default for tau is max(a).

Hint: If the zeros or poles cluster together, your filter order may be too low or the pole radius may be too small (overly constrained). Try increasing n or radius.

For group-delay equalization of an IIR filter, compute a by subtracting the filter's group delay from its maximum group delay. For example,

```
[be,ae] = ellip(4,1,40,0.2);
f = 0:0.001:0.2;
g = grpdelay(be,ae,f,2);   % Equalize only the passband.
a = max(g)-g;
[num,den]=iirgrpdelay(8, f, [0 0.2], a);
```

See Also

freqz, filter, grpdelay, iirlpnorm, iirlpnormc, zplane

References

Antoniou, A., *Digital Filters: Analysis, Design, and Applications*, Second Edition, McGraw-Hill, Inc. 1993.

iirlp2bp

Purpose

Transform an IIR real lowpass filter to an IIR real bandpass filter frequency response

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirlp2bp(B,A,Wo,Wt)
```

Description

`[Num,Den,AllpassNum,AllpassDen] = iirlp2bp(B,A,Wo,Wt)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the real lowpass prototype by applying a second-order real lowpass to real bandpass frequency mapping.

It also returns the numerator, `AllpassNum`, and the denominator `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with a numerator specified by `B` and a denominator specified by `A`.

This transformation effectively places one feature of an original filter, located at frequency $-W_o$, at the required target frequency location, W_{t1}, and the second feature, originally at $+W_o$, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}. This transformation implements the “DC Mobility,” which means that the Nyquist feature stays at Nyquist, but the DC feature moves to a location dependent on the selection of W_ts.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandpass transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Real lowpass to bandpass transformation can also be used for transforming other types of filters; e.g., real notch filters or resonators can be doubled and positioned at two distinct desired frequencies.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

Create the real bandpass filter by placing the cutoff frequencies of the prototype filter at the band edge frequencies W_{t1}=0.25 and W_{t2}=0.75:

```
[num, den] = iirlp2bp(b, a, 0.5, [0.25, 0.75]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

Arguments

`B`
Numerator of the prototype lowpass filter

`A`
Denominator of the prototype lowpass filter

`Wo`
Frequency value to be transformed from the prototype filter

`Wt`
Desired frequency locations in the transformed target filter

`Num`
Numerator of the target filter

`Den`
Denominator of the target filter

`AllpassNum`
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`iirftransf`, `allpasslp2bp`, `zpklp2bp`

References

[1] Constantinides, A.G., “Spectral transformations for digital filters,” *IEE 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] Nowrouzian, B. and L.T. Bruton, “Closed-form solutions for discrete-time elliptic transfer functions,” *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., “Design of bandpass digital filters,” *IEEE Proceedings*, vol. 1, pp. 1129-1231, June 1969.

Purpose IIR lowpass to complex bandpass frequency transformation frequency response

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirlp2bpc(B,A,Wo,Wt)
```

Description

[Num,Den,AllpassNum,AllpassDen] = iirlp2bpc(B,A,Wo,Wt) returns the numerator and denominator vectors, Num and Den respectively, of the target filter transformed from the real lowpass prototype by applying a first-order real lowpass to complex bandpass frequency transformation.

It also returns the numerator, AllpassNum, and the denominator, AllpassDen, of the allpass mapping filter. The prototype lowpass filter is given with a numerator specified by B and a denominator specified by A.

This transformation effectively places one feature of an original filter, located at frequency $-W_o$, at the required target frequency location, W_{t1}, and the second feature, originally at $+W_o$, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandpass transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Lowpass to bandpass transformation can also be used for transforming other types of filters; e.g., real notch filters or resonators can be doubled and positioned at two distinct desired frequencies at any place around the unit circle forming a pair of complex notches/resonators. This transformation can be used for designing bandpass filters for radio receivers from the high-quality prototype lowpass filter.

Examples Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

Move the cutoffs of the prototype filter to the new locations W_{t1}=0.25 and W_{t2}=0.75 creating a complex bandpass filter:

```
[num, den] = iirlp2bpc(b, a, 0.5, [0.25, 0.75]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

Arguments

`B`
Numerator of the prototype lowpass filter

`A`
Denominator of the prototype lowpass filter

`Wo`
Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

`Wt`
Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

`Num`
Numerator of the target filter

`Den`
Denominator of the target filter

`AllpassNum`
Numerator of the mapping filter

`AllpassDen`

Denominator of the mapping filter

See Also

`iirftransf`, `allpasslp2bpc`, `zpklp2bpc`

Purpose

Transform an IIR real lowpass filter to an IIR real bandstop filter frequency response

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirlp2bs(B,A,Wo,Wt)
```

Description

`[Num,Den,AllpassNum,AllpassDen] = iirlp2bs(B,A,Wo,Wt)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the real lowpass prototype by applying a second-order real lowpass to real bandstop frequency mapping.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with a numerator specified by `B` and a denominator specified by `A`.

This transformation effectively places one feature of an original filter, located at frequency $-W_o$, at the required target frequency location, W_{t1}, and the second feature, originally at $+W_o$, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}. This transformation implements the "Nyquist Mobility," which means that the DC feature stays at DC, but the Nyquist feature moves to a location dependent on the selection of W_o and W_ts.

Relative positions of other features of an original filter change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. After the transformation feature F_2 will precede F_1 in the target filter. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandstop transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

Create the real bandstop filter by placing the cutoff frequencies of the prototype filter at the band edge frequencies W_{t1}=0.25 and W_{t2}=0.75:

```
[num, den] = iirlp2bs(b, a, 0.5, [0.25, 0.75]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

Arguments

`B`
Numerator of the prototype lowpass filter

`A`
Denominator of the prototype lowpass filter

`Wo`
Frequency value to be transformed from the prototype filter

`Wt`
Desired frequency locations in the transformed target filter

`Num`
Numerator of the target filter

`Den`
Denominator of the target filter

`AllpassNum`
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`iirftransf`, `allpasslp2bs`, `zpklp2bs`

References

[1] Constantinides, A.G., “Spectral transformations for digital filters,” *IEE 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] Nowrouzian, B. and L.T. Bruton, “Closed-form solutions for discrete-time elliptic transfer functions,” *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., “Design of bandpass digital filters,” *IEEE Proceedings*, vol. 1, pp. 1129-1231, June 1969.

Purpose

Transform an IIR real lowpass filter to an IIR complex bandstop filter frequency response

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirlp2bsc(B,A,Wo,Wt)
```

Description

`[Num,Den,AllpassNum,AllpassDen] = iirlp2bsc(B,A,Wo,Wt)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the real lowpass prototype by applying a first-order real lowpass to complex bandstop frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with a numerator specified by `B` and the denominator specified by `A`.

This transformation effectively places one feature of an original filter, located at frequency $-W_o$, at the required target frequency location, W_{t1}, and the second feature, originally at $+W_o$, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}. Additionally the transformation swaps passbands with stopbands in the target filter.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandstop transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Lowpass to bandpass transformation can also be used for transforming other types of filters; e.g., real notch filters or resonators can be doubled and positioned at two distinct desired frequencies at any place around the unit circle forming a pair of complex notches/resonators. This transformation can be used for designing bandstop filters for band attenuation or frequency equalizers, from the high-quality prototype lowpass filter.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

Move the cutoffs of the prototype filter to the new locations W_{t1}=0.25 and W_{t2}=0.75 creating a complex bandstop filter:

```
[num, den] = iirlp2bsc(b, a, 0.5, [0.25, 0.75]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

Arguments

`B`
Numerator of the prototype lowpass filter

`A`
Denominator of the prototype lowpass filter

`Wo`
Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

`Wt`
Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

`Num`
Numerator of the target filter

`Den`
Denominator of the target filter

`AllpassNum`
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

See Also

`iirftransf`, `allpasslp2bsc`, `zpklp2bsc`.

Purpose Transform a discrete time lowpass IIR filter to a highpass filter

Syntax

```
[num,den] = iirlp2hp(b,a,wc,wd)
```

Description

`[num,den] = iirlp2hp(b,a,wc,wd)` with input arguments `b` and `a`, the numerator and denominator coefficients (zeros and poles) for a lowpass IIR filter, `iirlp2bp` transforms the magnitude response from lowpass to highpass. `num` and `den` return the coefficients for the transformed highpass filter. For `wc`, enter a selected frequency from your lowpass filter. You use the chosen frequency to define the magnitude response value you want in the highpass filter. Enter one frequency for the highpass filter — the value that defines the location of the transformed point — in `wd`. Note that all frequencies are normalized between zero and one. Notice also that the filter order does not change when you transform to a highpass filter.

When you select `wc` and designate `wd`, the transformation algorithm sets the magnitude response at the `wd` values of your bandstop filter to be the same as the magnitude response of your lowpass filter at `wc`. Filter performance between the values in `wd` is not specified, except that the stopband retains the ripple nature of your original lowpass filter and the magnitude response in the stopband is equal to the peak response of your lowpass filter. To accurately specify the filter magnitude response across the stopband of your bandpass filter, use a frequency value from within the stopband of your lowpass filter as `wc`. Then your bandstop filter response is the same magnitude and ripple as your lowpass filter stopband magnitude and ripple.

The fact that the transformation retains the shape of the original filter is what makes this function useful. If you have a lowpass filter whose characteristics, such as rolloff or passband ripple, particularly meet your needs, the transformation function lets you create a new filter with the same characteristic performance features, but in a highpass version. Without designing the highpass filter from the beginning.

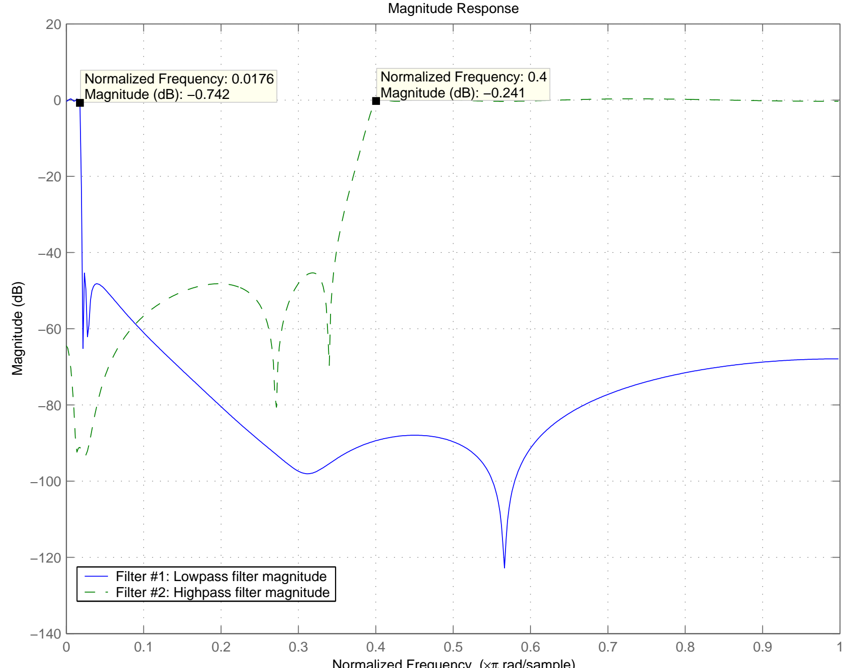
In some cases tranforming your filter may cause numerical problems, resulting in incorrect conversion to the highpass filter. Use `fvtool` to verify the response of your converted filter.

Examples

This example transforms an IIR filter from lowpass to high pass by moving the magnitude response at one frequency in the source filter to a new location in the transformed filter. To generate a highpass filter whose passband flattens

out at 0.4, we select the frequency in the lowpass filter where the passband starts to rolloff (wc = 0.0175) and move it to the new location at wd = 0.4.

```
[b,a] = iirlpnorm(10,6,[0 0.0175 0.02 0.0215 0.025 1],...
[0 0.0175 0.02 0.0215 0.025 1],[1 1 0 0 0 0],[1 1 1 1 10 10]);
wc = 0.0175;
wd = 0.4;
[num,den] = iirlp2hp(b,a,wc,wd);
fvtool(b,a,num,den);
```



In the figure showing the magnitude responses for the two filters, the transition band for the highpass filter is essentially the mirror image of the transition for the lowpass filter from 0.0175 to 0.025, stretched out over a wider frequency range. In the passbands, the filter share common ripple characteristics and magnitude.

See Also `iirlp2bp`, `iirlp2bs`, `iirlp2lp`, `firlp2lp`, `firlp2hp`

References

Sanjit K. Mitra, *Digital Signal Processing. A Computer-Based Approach*, Second Edition, McGraw-Hill, 2001.

Purpose

Transform a discrete time lowpass IIR filter to a different lowpass filter

Syntax

```
[num,den] = iirlp2lp(b,a,wc,wd)
```

Description

`[num,den] = iirlp2hp(b,a,wc,wd)` with input arguments `b` and `a`, the numerator and denominator coefficients (zeros and poles) for a lowpass IIR filter, `iirlp2bp` transforms the magnitude response from lowpass to highpass. `num` and `den` return the coefficients for the transformed highpass filter. For `wc`, enter a selected frequency from your lowpass filter. You use the chosen frequency to define the magnitude response value you want in the highpass filter. Enter one frequency for the highpass filter — the value that defines the location of the transformed point — in `wd`. Note that all frequencies are normalized between zero and one. Notice also that the filter order does not change when you transform to a highpass filter.

When you select `wc` and designate `wd`, the transformation algorithm sets the magnitude response at the `wd` values of your bandstop filter to be the same as the magnitude response of your lowpass filter at `wc`. Filter performance between the values in `wd` is not specified, except that the stopband retains the ripple nature of your original lowpass filter and the magnitude response in the stopband is equal to the peak response of your lowpass filter. To accurately specify the filter magnitude response across the stopband of your bandpass filter, use a frequency value from within the stopband of your lowpass filter as `wc`. Then your bandstop filter response is the same magnitude and ripple as your lowpass filter stopband magnitude and ripple.

The fact that the transformation retains the shape of the original filter is what makes this function useful. If you have a lowpass filter whose characteristics, such as rolloff or passband ripple, particularly meet your needs, the transformation function lets you create a new filter with the same characteristic performance features, but in a highpass version. Without designing the highpass filter from the beginning.

In some cases tranforming your filter may cause numerical problems, resulting in incorrect conversion to the highpass filter. Use `fvtool` to verify the response of your converted filter.

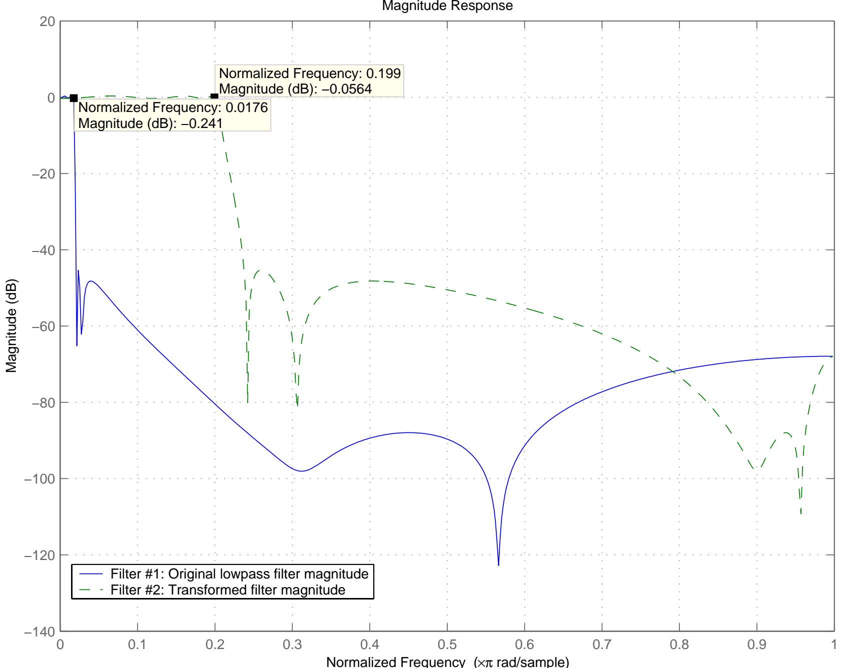
Examples

This example transforms an IIR filter from lowpass to high pass by moving the magnitude response at one frequency in the source filter to a new location in the transformed filter. To generate a lowpass filter whose passband extends

out to 0.2, we select the frequency in the lowpass filter where the passband starts to rolloff (wc = 0.0175) and move it to the new location at wd = 0.2.

```
[b,a] = iirlpnorm(10,6,[0 0.0175 0.02 0.0215 0.025 1],...
[0 0.0175 0.02 0.0215 0.025 1],[1 1 0 0 0 0],[1 1 1 1 10 10]);
wc = 0.0175;
wd = 0.2;
[num,den] = iirlp2lp(b,a,wc,wd);
fvtool(b,a,num,den);
```

Moving the edge of the passband from 0.0175 to 0.2 results in a new lowpass filter whose peak response inband is the same as the original filter: same ripple, same absolute magnitude.



Notice that the rolloff is slightly less steep and the stopband profiles are the same for both filters; the new filter stopband is a “stretched” version of the original, as is the passband of the new filter.

See Also

`iirlp2bp`, `iirlp2bs`, `iirlp2hp`, `firlp2lp`, `firlp2hp`

References

Sanjit K. Mitra, *Digital Signal Processing. A Computer-Based Approach*, Second Edition, McGraw-Hill, 2001.

Purpose

Transform an IIR real lowpass filter to an IIR real M-band filter frequency response

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirlp2mb(B,A,Wo,Wt)
[Num,Den,AllpassNum,AllpassDen] = iirlp2mb(B,A,Wo,Wt,Pass)
```

Description

`[Num,Den,AllpassNum,AllpassDen] = iirlp2mb(B,A,Wo,Wt)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the real lowpass prototype by applying an `M`th-order real lowpass to real multibandpass frequency mapping. By default the DC feature is kept at its original location.

`[Num,Den,AllpassNum,AllpassDen]=iirlp2mb(B,A,Wo,Wt,Pass)` allows you to specify an additional parameter, `Pass`, which chooses between using the "DC Mobility" and the "Nyquist Mobility". In the first case the Nyquist feature stays at its original location and the DC feature is free to move. In the second case the DC feature is kept at an original frequency and the Nyquist feature is movable.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with a numerator specified by `B` and a denominator specified by `A`.

This transformation effectively places one feature of an original filter, located at frequency W_o, at the required target frequency locations, $W_{t1},\ldots,W_{tM}$.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

Example 1: Create the real multiband filter with two passbands:

```
[num1, den1] = iirlp2mb(b, a, 0.5, [2 4 6 8]/10);
[num2, den2] = iirlp2mb(b, a, 0.5, [2 4 6 8]/10, 'pass');
```

Example 2: Create the real multiband filter with two stopbands:

```
[num3, den3] = iirlp2mb(b, a, 0.5, [2 4 6 8]/10, 'stop');
```

Verify the result by comparing the prototype filter with target filters:

```
fvtool(b, a, num1, den1, num2, den2, num3, den3);
```

Arguments

`B`
Numerator of the prototype lowpass filter

`A`
Denominator of the prototype lowpass filter

`Wo`
Frequency value to be transformed from the prototype filter

`Wt`
Desired frequency locations in the transformed target filter

`Pass`
Choice (`'pass'`/`'stop'`) of passband/stopband at DC, `pass` being the default

`Num`
Numerator of the target filter

`Den`
Denominator of the target filter

`AllpassNum`
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`iirftransf`, `allpasslp2mb`, `zpklp2mb`

References

[1] Franchitti, J.C., “All-pass filter interpolation and frequency transformation problems,” MSc Thesis, Dept. of Electrical and Computer Engineering, University of Colorado, 1985.

[2] Feyh, G., J.C. Franchitti and C.T. Mullis, “All-pass filter interpolation and frequency transformation problem,” *Proceedings 20th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, California, pp. 164-168, November 1986.

[3] Mullis, C.T. and R. A. Roberts, *Digital Signal Processing*, section 6.7, Reading, Mass., Addison-Wesley, 1987.

[4] Feyh, G., W.B. Jones and C.T. Mullis, *An extension of the Schur Algorithm for frequency transformations, Linear Circuits, Systems and Signal Processing: Theory and Application*, C. J. Byrnes et al Eds, Amsterdam: Elsevier, 1988.

iirlp2mbc

Purpose

Transform an IIR real lowpass filter to an IIR complex M-band filter frequency response

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirlp2mbc(B,A,Wo,Wc)
```

Description

`[Num,Den,AllpassNum,AllpassDen] = iirlp2mbc(B,A,Wo,Wc)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the real lowpass prototype by applying an `M`th-order real lowpass to complex multibandpass frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with a numerator specified by `B` and a denominator specified by `A`.

This transformation effectively places one feature of an original filter, located at frequency W_o, at the required target frequency locations, W_{t1},...,W_{tM}.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

Example 1: Create the complex multiband filter with two passbands:

```
[num1, den1] = iirlp2mbc(b, a, 0.5, [2 4 6 8]/10);
```

Example 2: Create the complex multiband filter with two passbands:

```
[num2, den2] = iirlp2mbc(b, a, 0.5, [2 4 6 8]/10, 'pass');
```

Example 3: Create the complex multiband filter with two stopbands:

```
[num3, den3] = iirlp2mbc(b, a, 0.5, [2 4 6 8]/10, 'stop');
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num1, den1, num2, den2, num3, den3);
```

Arguments

`B`
Numerator of the prototype lowpass filter

`A`
Denominator of the prototype lowpass filter

`Wo`
Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

`Wc`
Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

`Num`
Numerator of the target filter

`Den`
Denominator of the target filter

`AllpassNum`
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

See Also

`iirftransf`, `allpasslp2mbc`, `zpklp2mbc`

iirlp2xc

Purpose

Transform an IIR real lowpass filter to an IIR complex N-point filter frequency response

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirlp2xc(B,A,Wo,Wt)
```

Description

`[Num,Den,AllpassNum,AllpassDen] = iirlp2xc(B,A,Wo,Wt)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the real lowpass prototype by applying an `N`th-order real lowpass to complex multipoint frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with a numerator specified by `B` and a denominator specified by `A`.

Parameter `N` also specifies the number of replicas of the prototype filter created around the unit circle after the transformation. This transformation effectively places `N` features of an original filter, located at frequencies W_{o1},...,W_{oN}, at the required target frequency locations, W_{t1},...,W_{tM}.

Relative positions of other features of an original filter are the same in the target filter for the Nyquist mobility and are reversed for the DC mobility. For the Nyquist mobility this means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation. For DC mobility feature F_2 will precede F_1 after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., a stopband edge, DC, the deep minimum in the stopband, or other ones. The only condition is that the features must be selected in such a way that when creating `N` bands around the unit circle, there will be no band overlap.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

Create the complex bandpass filter from the real lowpass filter:

```
[num, den] = iirlp2xc(b, a, [-0.5 0.5], [-0.25 0.25]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

Arguments

`B`
Numerator of the prototype lowpass filter

`A`
Denominator of the prototype lowpass filter

`Wo`
Frequency values to be transformed from the prototype filter. They should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

`Wt`
Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

`Num`
Numerator of the target filter

`Den`
Denominator of the target filter

`AllpassNum`
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

See Also

`iirftransf`, `allpasslp2xc`, `zpklp2xc`

iirlp2xn

Purpose

Transform an IIR real lowpass filter to an IIR real N-point filter frequency response

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirlp2xn(B,A,Wo,Wt)
[Num,Den,AllpassNum,AllpassDen] = iirlp2xn(B,A,Wo,Wt,Pass)
```

Description

`[Num,Den,AllpassNum,AllpassDen] = iirlp2xn(B,A,Wo,Wt)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the real lowpass prototype by applying an `N`th-order real lowpass to real multipoint frequency transformation, where `N` is the number of features being mapped. By default the DC feature is kept at its original location.

`[Num,Den,AllpassNum,AllpassDen]=iirlp2xn(B,A,Wo,Wt,Pass)` allows you to specify an additional parameter, `Pass`, which chooses between using the "DC Mobility" and the "Nyquist Mobility". In the first case the Nyquist feature stays at its original location and the DC feature is free to move. In the second case the DC feature is kept at an original frequency and the Nyquist feature is allowed to move.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with the numerator specified by `B` and the denominator specified by `A`.

Parameter `N` also specifies the number of replicas of the prototype filter created around the unit circle after the transformation. This transformation effectively places `N` features of an original filter, located at frequencies W_{o1},...,W_{oN}, at the required target frequency locations, W_{t1},...,W_{tM}.

Relative positions of other features of an original filter are the same in the target filter for the Nyquist mobility and are reversed for the DC mobility. For the Nyquist mobility this means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation. For DC mobility feature F_2 will precede F_1 after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the

stopband, or other ones. The only condition is that the features must be selected in such a way that when creating N bands around the unit circle, there will be no band overlap.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

Move the cutoffs of the prototype filter to the new locations W_{t1}=0.25 and W_{t2}=0.75 creating a real bandpass filter:

```
[num, den] = iirlp2xn(b, a, [-0.5 0.5], [0.25 0.75], 'pass');
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

Arguments

B
Numerator of the prototype lowpass filter

A
Denominator of the prototype lowpass filter

Wo
Frequency values to be transformed from the prototype filter

Wt
Desired frequency locations in the transformed target filter

Pass
Choice ('pass'/'stop') of passband/stopband at DC, pass being the default

Num
Numerator of the target filter

Den
Denominator of the target filter

AllpassNum
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`iirftransf`, `allpasslp2xn`, `zpklp2xn`

References

[1] Cain, G.D., A. Krukowski and I. Kale, "High Order Transformations for Flexible IIR Filter Design," *VII European Signal Processing Conference (EUSIPCO'94)*, vol. 3, pp. 1582-1585, Edinburgh, United Kingdom, September 1994.

[2] 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.

iirlpnorm

Purpose

Least P-norm optimal IIR filter design

Syntax

```
[num,den] = iirlpnorm(n,d,f,edges,a)
[num,den] = iirlpnorm(n,d,f,edges,a,w)
[num,den] = iirlpnorm(n,d,f,edges,a,w,p)
[num,den] = iirlpnorm(n,d,f,edges,a,w,p,dens)
[num,den] = iirlpnorm(n,d,f,edges,a,w,p,dens,initnum,initden)
```

Description

`[num,den] = iirlpnorm(n,d,f,edges,a)` returns a filter having a numerator order `n` and denominator order `d` which is the best approximation to the desired frequency response described by `f` and `a` in the least-pth sense. The vector `edges` specifies the band-edge frequencies for multi-band designs. An unconstrained quasi-Newton algorithm is employed and any poles or zeros that lie outside of the unit circle are reflected back inside. `n` and `d` should be chosen so that the zeros and poles are used effectively. See the “Hints” section. Always use `freqz` to check the resulting filter.

`[num,den] = iirlpnorm(n,d,f,edges,a,w)` uses the weights in `w` to weight the error. `w` has one entry per frequency point (the same length as `f` and `a`) which tells `iirlpnorm` how much emphasis to put on minimizing the error in the vicinity of each frequency point relative to the other points. `f` and `a` must have the same number of elements, which may exceed the number of elements in `edges`. This allows for the specification of filters having any gain contour within each band. The frequencies specified in `edges` must also appear in the vector `f`. For example,

```
[num,den] = iirlpnorm(5,12,[0 .15 .4 .5 1],[0 .4 .5 1],...
[1 1.6 1 0 0],[1 1 1 10 10])
```

is a lowpass filter with a peak of 1.6 within the passband.

`[num,den] = iirlpnorm(n,d,f,edges,a,w,p)` where `p` is a two-element vector `[pmin pmax]` allows for the specification of the minimum and maximum values of `p` used in the least-pth algorithm. Default is [2 128] which essentially yields the L-infinity, or Chebyshev, norm. `Pmin` and `pmax` should be even. If `p` is the string `'inspect'`, no optimization will occur. This can be used to inspect the initial pole/zero placement.

`[num,den] = iirlpnorm(n,d,f,edges,a,w,p,dens)` specifies the grid density `dens` used in the optimization. The number of grid points is `(dens*(n+d+1))`. The default is 20. `dens` can be specified as a single-element cell array. The grid is not equally spaced.

`[num,den] = iirlpnorm(n,d,f,edges,a,w,p,dens,initnum,initden)` allows for the specification of the initial estimate of the filter numerator and denominator coefficients in vectors `initnum` and `initden`. This may be useful for difficult optimization problems. The pole-zero editor in the Signal Processing Toolbox can be used for generating `initnum` and `initden`.

Hints

- This is a weighted least-pth optimization.
- Check the radii and locations of the poles and zeros for your filter. If the zeros are on the unit circle and the poles are well inside the unit circle, try increasing the order of the numerator or reducing the error weighting in the stopband.
- Similarly, if several poles have a large radii and the zeros are well inside of the unit circle, try increasing the order of the denominator or reducing the error weighting in the passband.

See Also

`iirlpnormc`, `filter`, `freqz`, `iirgrpdelay`, `zplane`

References

Antoniou, A., *Digital Filters: Analysis, Design, and Applications*, Second Edition, McGraw-Hill, Inc. 1993.

Purpose Design a constrained least P-norm optimal IIR filter

Syntax

```
[num,den] = iirlpnormc(n,d,f,edges,a)
[num,den] = iirlpnormc(n,d,f,edges,a,w)
[num,den] = iirlpnormc(n,d,f,edges,a,w,radius)
[num,den] = iirlpnormc(n,d,f,edges,a,w,radius,p)
[num,den] = iirlpnormc(n,d,f,edges,a,w,radius,p,dens)
[num,den] = iirlpnormc(n,d,f,edges,a,w,radius,p,...
  dens,initnum,initden)
```

Description

`[num,den] = iirlpnormc(n,d,f,edges,a)` returns a filter having a numerator order `n` and denominator order `d` which is the best approximation to the desired frequency response described by `f` and `a` in the least-pth sense. The vector `edges` specifies the band-edge frequencies for multi-band designs. A constrained Newton-type algorithm is employed. `n` and `d` should be chosen so that the zeros and poles are used effectively. See the "Hints" section. Always check the resulting filter using `freqz`.

`[num,den] = iirlpnormc(n,d,f,edges,a,w)` uses the weights in `w` to weight the error. `w` has one entry per frequency point (the same length as `f` and `a`) which tells `iirlpnormc` how much emphasis to put on minimizing the error in the vicinity of each frequency point relative to the other points. `f` and `a` must have the same number of elements, which can exceed the number of elements in `edges`. This allows for the specification of filters having any gain contour within each band. The frequencies specified in `edges` must also appear in the vector `f`. For example,

```
[num,den] = iirlpnormc(5,12,[0 .15 .4 .5 1],[0 .4 .5 1],...
[1 1.6 1 0 0],[1 1 1 10 10])
```

designs a lowpass filter with a peak of 1.6 within the passband.

`[num,den] = iirlpnormc(n,d,f,edges,a,w,radius)` returns a filter having a maximum pole radius of `radius` where 0<`radius`<1. `radius` defaults to 0.999999. Filters having a reduced pole radius may retain better transfer function accuracy after you quantize them.

`[num,den] = iirlpnormc(n,d,f,edges,a,w,radius,p)` where `p` is a two-element vector `[pmin pmax]` allows for the specification of the minimum

and maximum values of `p` used in the least-pth algorithm. Default is [2 128] which essentially yields the L-infinity, or Chebyshev, norm. `pmin` and `pmax` should be even. If `p` is the string `'inspect'`, no optimization will occur. This can be used to inspect the initial pole/zero placement.

`[num,den] = iirlpnormc(n,d,f,edges,a,w,radius,p,dens)` specifies the grid density `dens` used in the optimization. The number of grid points is `(dens*(n+d+1))`. The default is 20. `dens` can be specified as a single-element cell array. The grid is not equally spaced.

`[num,den] = iirlpnormc(n,d,f,edges,a,w,radius,p,dens,...`
`initnum,initden)` allows for the specification of the initial estimate of the filter numerator and denominator coefficients in vectors `initnum` and `initden`. This may be useful for difficult optimization problems. The pole-zero editor in the Signal Processing Toolbox can be used for generating `initnum` and `initden`.

Hints

- This is a weighted least-pth optimization.
- Check the radii and location of the resulting poles and zeros.
- If the zeros are all on the unit circle and the poles are well inside of the unit circle, try increasing the order of the numerator or reducing the error weighting in the stopband.
- Similarly, if several poles have a large radius and the zeros are well inside of the unit circle, try increasing the order of the denominator or reducing the error weight in the passband.
- If you reduce the pole radius, you might need to increase the order of the denominator.

See Also

`freqz`, `filter`, `iirgrpdelay`, `iirlpnorm`, `zplane`

References

Antoniou, A., *Digital Filters: Analysis, Design, and Applications*, Second Edition, McGraw-Hill, Inc. 1993.

iirnotch

Purpose

Design a second-order IIR notch digital filter

Syntax

```
[num,den] = iirnotch(w0,bw)
[num,den] = iirnotch(w0,bw,ab)
```

Description

`[num,den] = iirnotch(w0,bw)` eturns a digital notching filter with the notch located at `w0`, and with the bandwidth at the -3 dB point set to `bw`. To design the filter, `w0` must meet the condition 0.0 < `w0` < 1.0, where 1.0 corresponds to π radians per sample in the frequency range.

The quality factor (Q factor) q for the filter is related to the filter bandwidth by $q = \omega_0/bw$ where ω_0 is `w0`, the frequency to remove from the signal.

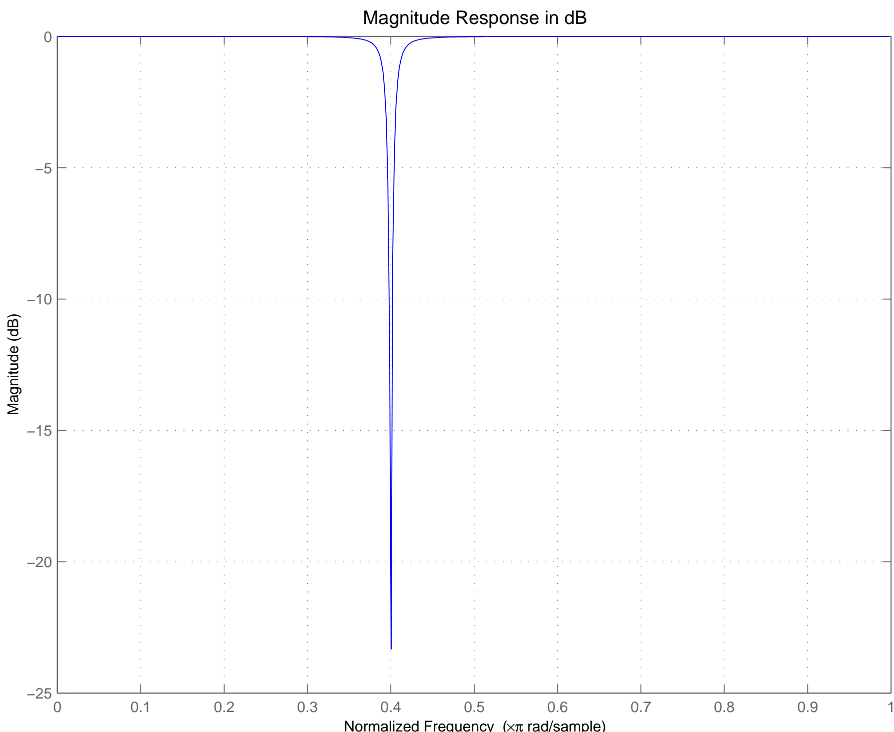
`[num,den] = iirnotch(w0,bw,ab)` returns a digital notching filter whose bandwidth, `bw`, is specified at a level of -`ab` decibels. Including the optional input argument `ab` lets you specify the magnitude response bandwidth at a level that is not the default -3dB point, such as -6 dB or 0 dB.

Examples

Design and plot an IIR notch filter that removes a 60 Hz tone (`f0`) from a signal at 300 Hz (`fs`). For this example, set the Q factor for the filter to 35 and use it to specify the filter bandwidth:

```
wo = 60/(300/2);  bw = wo/35;
[b,a] = iirnotch(wo,bw);
fvtool(b,a);
```

Shown in the next plot, the notch filter has the desired bandwidth with the notch located at 60 Hz, or 0.4π radians per sample. Compare this plot to the comb filter plot shown on the reference page for `iircomb`.



See Also

`gremez`, `iircomb`, `iirpeak`

Purpose

Design a second-order IIR peak or resonator digital filter

Syntax

```
[num,den] = iirpeak(w0,bw)
[num,den] = iirpeak(w0,bw,ab)
```

Description

`[num,den] = iirpeak(w0,bw)` eturns a second-order digital peaking filter with the peak located at `w0`, and with the bandwidth at the +3dB point set to `bw`. To design the filter, `w0` must meet the condition 0.0 < `w0` < 1.0, where 1.0 corresponds to π radians per sample in the frequency range.

The quality factor (Q factor) q for the filter is related to the filter bandwidth by q = ω_0/bw where ω_0 is `w0` the signal frequency to boost.

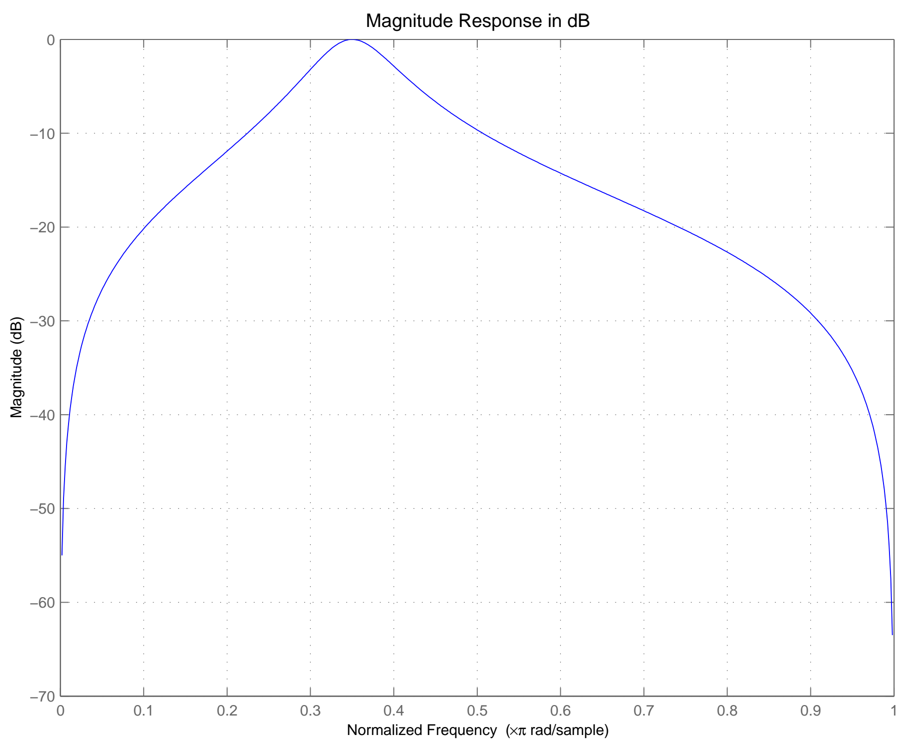
`[num,den] = iirpeak(w0,bw,ab)` returns a digital peaking filter whose bandwidth, `bw`, is specified at a level of +`ab` decibels. Including the optional input argument `ab` lets you specify the magnitude response bandwidth at a level that is not the default +3dB point, such as +6 dB or 0 dB.

Examples

Design and plot an IIR peaking filter that boosts the frequency at 1.75 Khz in a signal and has bandwidth of 500 Hz at the -3 dB point:

```
fs = 10000; wo = 1750/(fs/2);  bw = 500/(fs/2);
[b,a] = iirpeak(wo,bw);
fvtool(b,a);
```

Shown in the next plot, the peak filter has the desired gain and bandwidth at 1.75 KHz.



See Also

`gremez`, `iircomb`, `iirnotch`

Purpose

Compute power complementary filter.

Syntax

```
[bp,ap] = iirpowcomp(b,a)
[bp,ap,c] = iirpowcomp(b,a)
```

Description

`[bp,ap] = iirpowcomp(b,a)` returns the coefficients of the power complementary IIR filter g(z) = bp(z)/ap(z) in vectors `bp` and `ap`, given the coefficients of the IIR filter h(z) = b(z)/a(z) in vectors `b` and `a`. `b` must be symmetric (Hermitian) or antisymmetric (antihermitian) and of the same length as `a`. The two power complementary filters satisfy the relation

$$|H(w)|^2 + |G(w)|^2 = 1.$$

`[bp,ap,c] = iirpowcomp(b,a)` where `c` is a complex scalar of magnitude =1, forces `bp` to satisfy the generalized hermitian property

`conj(bp(end:-1:1)) = c*bp`.

When `c` is omitted, it is chosen as follows:

- When `b` is real, chooses C as 1 or -1, whichever yields `bp` real
- When `b` is complex, C defaults to 1

`ap` is always equal to `a`.

Examples

```
[b,a]=cheby1(10,.5,.4);
[bp,ap]=iirpowcomp(b,a);
[h,w,s]=freqz(b,a); [h1,w,s]=freqz(bp,ap);
s.plot='mag'; s.yunits='sq';freqzplot([h h1],w,s)
```

See Also

`tf2ca`, `tf2cl`, `ca2tf`, `cl2tf`

iirrateup

Purpose

Upsample an IIR filter by an integer factor

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirrateup(B,A,N)
```

Description

`[Num,Den,AllpassNum,AllpassDen] = iirrateup(B,A,N)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter being transformed from any prototype by applying an `N`th-order rateup frequency transformation, where `N` is the upsample ratio. Transformation creates `N` equal replicas of the prototype filter frequency response.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with a numerator specified by `B` and a denominator specified by `A`.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
[num, den] = iirrateup(b, a, 4);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

Arguments

`B`
Numerator of the prototype lowpass filter

`A`
Denominator of the prototype lowpass filter

`N`
Frequency multiplication ratio

`Num`
Numerator of the target filter

`Den`
Denominator of the target filter

`AllpassNum`
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

See Also `iirftransf`, `allpassrateup`, `zpkrateup`

Purpose

Shift the frequency response of an IIR real filter

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirshift(B,A,Wo,Wt)
```

Description

`[Num,Den,AllpassNum,AllpassDen] = iirshift(B,A,Wo,Wt)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the real lowpass prototype by applying a second-order real shift frequency mapping.

It also returns the numerator, `AllpassNum`, and the denominator of the allpass mapping filter, `AllpassDen`. The prototype lowpass filter is given with the numerator specified by `B` and the denominator specified by `A`.

This transformation places one selected feature of an original filter located at frequency W_o to the required target frequency location, W_t. This transformation implements the "DC Mobility," which means that the Nyquist feature stays at Nyquist, but the DC feature moves to a location dependent on the selection of W_o and W_t.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the real shift transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can change their position in a simple way without designing them from the beginning.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

Perform the real frequency shift by defining where the selected feature of the prototype filter, originally at W_o=0.5, should be placed in the target filter, W_t=0.75:

```
Wo = 0.5; Wt = 0.75;

[num, den] = iirshift(b, a, Wo, Wt);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

Arguments

`B`
Numerator of the prototype lowpass filter

`A`
Denominator of the prototype lowpass filter

`Wo`
Frequency value to be transformed from the prototype filter

`Wt`
Desired frequency location in the transformed target filter

`Num`
Numerator of the target filter

`Den`
Denominator of the target filter

`AllpassNum`
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`iirftransf`, `allpassshift`, `zpkshift`.

iirshiftc

Purpose

Shift the frequency response of an IIR complex filter

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirshiftc(B,A,Wo,Wt)
```

Description

`[Num,Den,AllpassNum,AllpassDen] = iirshiftc(B,A,Wo,Wc)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the real lowpass prototype by applying a first-order complex frequency shift transformation. This transformation rotates all the features of an original filter by the same amount specified by the location of the selected feature of the prototype filter, originally at W_o, placed at W_t in the target filter.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with the numerator specified by `B` and the denominator specified by `A`.

`[Num,Den,AllpassNum,AllpassDen] = iirshiftc(B,A,0,0.5)` calculates the allpass filter for doing the Hilbert transformation, i.e. a 90 degree counterclockwise rotation of an original filter in the frequency domain.

`[Num,Den,AllpassNum,AllpassDen] = iirshiftc(B,A,0,-0.5)` calculates the allpass filter for doing an inverse Hilbert transformation, i.e. a 90 degree clockwise rotation of an original filter in the frequency domain.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

Rotate all features of the prototype filter in the frequency domain by the same amount by specifying where the selected feature of an original filter, W_o=0.5, should appear in the target filter, W_t=0.25:

```
[num, den] = iirshiftc(b, a, 0.5, 0.25);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

Arguments

`B`
Numerator of the prototype lowpass filter

`A`
Denominator of the prototype lowpass filter

`Wo`
Frequency value to be transformed from the prototype filter

`Wt`
Desired frequency location in the transformed target filter

`Num`
Numerator of the target filter

`Den`
Denominator of the target filter

`AllpassNum`
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

See Also

`iirftransf`, `allpassshiftc`, `zpkshiftc`

References

[1] Oppenheim, A.V., R.W. Schafer and J.R. Buck, *Discrete-Time Signal Processing*, Prentice-Hall International Inc., 1989.

[2] Dutta-Roy, S.C. and B. Kumar, "On digital differentiators, Hilbert transformers, and half-band low-pass filters," *IEEE Transactions on Education*, vol. 32, pp. 314-318, August 1989.

impz

Purpose

Compute the impulse response of quantized filters

Syntax

```
[h,t] = impz(hq)
[h,t] = impz(hq,n)
[h,t] = impz(hq,n,Fs)
[h,t,ref] = impz(hq,...)
impz(hq,...)
```

Description

`[h,t] = impz(hq)` computes the response of the quantized filter `hq` to an impulse. `impz` returns the computed impulse response in the column vector `h` and the corresponding sample times in the column vector `t` (where `t = [0:n-1]'` and `n = length(t)` is computed automatically).

`[h,t] = impz(hq,n)` computes `n` samples of the quantized impulse response for any positive integer `n`. In this case, `t = [0:n-1]'`. When `n` is a vector of integers, `impz` computes the impulse response at those integer locations, starting the response computation from 0 (and `t=n` or `t=[0 n]`). If, instead of `n`, you include the empty vector `[]` as the second argument, impz computes the number of samples automatically.

`[h,t] = impz(hq,n,Fs)` computes `n` samples and produces a vector `t` of length `n` so that the samples are spaced `1/Fs` units apart.

`[h,t,ref] = impz(hq,...)` returns the impulse response of the quantized filter `hq` in the column vector `h`, and returns the impulse response of the reference filter in the vector `ref`.

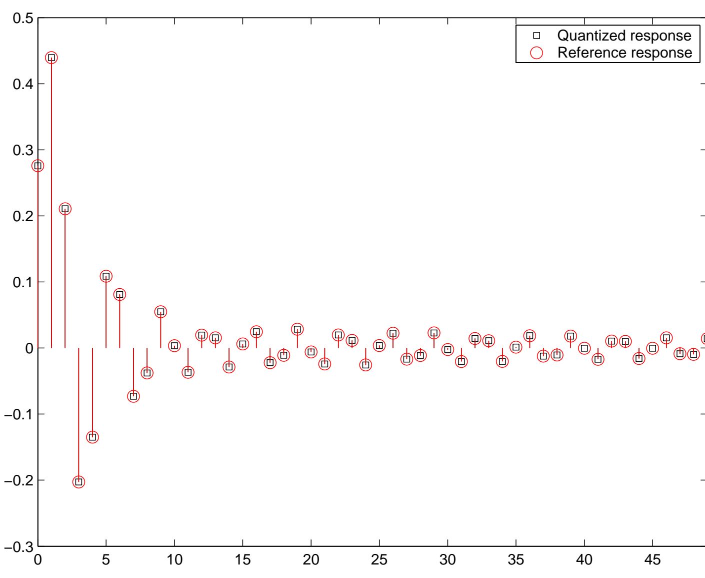
`impz(hq,...)` with no output arguments plots the impulse response of the reference filter associated with `hq`, and the quantized impulse response of quantized filter `hq` in a new figure window. `impz` uses `stem` for plotting the impulse responses.

Note `impz` works for both real and complex quantized filters. When you omit the output arguments, only the real part of the impulse response is plotted.

Examples

Create a quantized filter for a fourth-order, low-pass elliptic filter with a cutoff frequency of 0.4 times the Nyquist frequency. Use a second-order sections structure to resist quantization errors. Plot the first 50 samples of the quantized impulse response, along with the reference impulse response.

```
% Specify transfer function parameters for the reference filter.

[b,a] = ellip(4,3,20,.6);

% Create a quantized filter from the reference filter. Convert the
quantized filter to second-order section form, order, and scale.

hq = sos(qfilt('ref',{b,a}));

Warning: 1 overflow in coefficients.

impz(hq,50)
```



Algorithm

`impz` applied to the quantized filter `hq` applies the `filter` command twice to a length `n` impulse sequence:

- Once for a quantized filter whose coefficients are determined by the `ReferenceCoefficients` property value for `hq`
- Once for a quantized filter whose coefficients are determined by the `QuantizedCoefficients` property value for `hq`

The resulting plots use `stem`.

Warnings that occur with `impz` are a result of the `filter` command. In particular, you get an input overflow warning with `impz` when the `InputFormat` property value for the quantized filter `hq` is fixed-point and has only one bit to the left of the radix point.

For example, when your `InputFormat` property is set to `{'fixed',[16,15]}`, you get an input overflow warning when you implement `impz`.

See Also

`filter`

Purpose

Configure the initialization structure used as an input argument to `adaptkalman`

Syntax

```
s = initkalman(w0,k0,qm,qp)
s = initkalman(w0,k0,qm,qp,zi)
```

Description

`s = initkalman(w0,k0,qm,qp)` returns the fully populated structure `s` that you use when you call `adaptkalman`. Vector `w0` contains the initial values of the filter coefficients. Its length equals the order of the adapting FIR filter plus one.

`k0` contains the initial state error covariance matrix. It should be an Hermitian symmetric square matrix with dimensions equal to `length(w0)`.

`qm` is the measurement noise variance and `qp` is the process noise covariance matrix.

`s = initkalman(w0,k0,qm,qp,zi)` adds input argument `zi` to specify the filter initial conditions. When you omit `zi` or specify it as empty, `[]`, `initkalman` defaults to `zi` equal to a zero vector of length `[length(w0)-1]`. When you use `adaptkalman` in program structures like for-loops, the initial conditions provide the filter weights for the first iteration of the loop. Recall that each iteration of the Kalman filter algorithm uses the weights from the previous iteration. Without initial conditions the first interation has no input to use. For each loop iteration the same problem occurs and the filter never adapts to the unknown.

When you check the contents of `s` after you use `initkalman` MATLAB displays the structure elements, rather than the input argument names. To help you

remember which element in s corresponds to each input argument to initkalman, the following table provides the mapping.

initkalman argument	Structure Field	Argument Description
w0	s.coeffs	Kalman adaptive filter coefficients. Should be initialized with the initial values for the FIR filter coefficients. Updated coefficients are returned when you use s as an output argument.
k0	s.errcov	The state error covariance matrix. Initialize this element with the initial error state covariance matrix. An updated matrix is returned when you use s as an output argument.
qm	s.measvar	Conntains the measurement noise variance matrix.
qp	s.procov	Contains the process noise covariance matrix.
	s.states	Returns the states of the FIR filter. This is an optional element. If omitted, it defaults to a zero vector of length equal to the filter order.
	s.gain	Kalman gain vector. Not required, but computed and returned after every iteration.
	s.iter	Total number of iterations in adaptive filter run. This is read-only.

For example, after you use initkalman to create s, MATLAB returns the structure shown when you enter s at the prompt. In this example, we use a 31st-order filter.

```
s

s =
```

```
      coeffs: [1x32 double]
      states: [31x1 double]
      errcov: [32x32 double]
     measvar: 2
      procov: [32x32 double]
        gain: []
        iter: 0
```

Examples

Prepare the initialization structure needed to identify an unknown FIR filter with 32 coefficients. To see this structure used in an example, refer to `adaptkalman`.

```
w0 = zeros(1,32);       % Intial filter coefficients
k0 = 0.5*eye(32);       % Initial state error correlation matrix
qm = 2;                 % Measurement noise covariance
qp = 0.1*eye(32);       % Process noise covariance
s = initkalman(w0,k0,qm,qp);
```

See Also

`adaptkalman`, `initlms`, `initnlms`, `initrls`, `initse`

Reference

S. Haykin, *Adaptive Filter Theory*, 3rd Edition, Prentice Hall, N.J., 1996.

Purpose

Configure the initialization structure used as an input argument to `adaptlms`

Syntax

```
s = initlms(w0,mu)
s = initlms(w0,mu,zi)
s = initlms(w0,mu,zi,lf)
```

Description

`s = initlms(w0,mu)` returns the fully populated structure `s` that you use when you call `adaptklms`. Vector `w0` contains the initial values of the filter coefficients. Its length should be equal to [order of the adapting FIR filter + 1]. `mu` is the Least Mean Square (LMS) algorithm step size. The step size you specify determines both the time it takes for the LMS algorithm to converge to a solution and the accuracy of that solution (how closely the result approaches the minimum least mean square error). Generally, small step sizes adapt more slowly but more closely and large step sizes adapt more quickly with larger error compared to the true minimum mean square error.

In matrix form, the LMS algorithm is

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \mu e(k)\mathbf{x}(k) \quad \textbf{(13-1)}$$

with vector **w** containing the weights applied to the filter coefficients and vector **x** containing the input data. *e(k)* (equal to desired signal - filtered signal) is the error at time k and is the quantity the LMS algorithm seeks to minimize. μ (`mu`) is the step size. As you specify `mu` smaller, the correction to the filter weights gets smaller for each sample and the LMS error falls more slowly. Larger `mu` 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 good convergence rate and stability, select `mu` within the following bounds:

$$0 < \mu < \frac{1}{N\{InputSignalPower\}}$$

where N is the number of samples in the signal.

`s = initlms(w0,mu,zi)` adds input argument `zi` to specify the filter initial conditions. When you omit `zi` or specify it as empty, `[]`, `initkalman` defaults to `zi` equal to a zero vector of length equal to `[length(w0)-1]`. For conditional processing such as using adaptlms in a for-loop, specifying the initial conditions is very important. Each iteration of the LMS algorithm uses the

weights from the prior interation. You supply the initial conditions so the first iteration has a set of prior filter weights to start from.

`s = initlms(w0,mu,zi,lf)` specifies the leakage factor `lf` as an input argument. Including the leakage factor can improve the behavior of the algorithm. Leaking the weight $\mathbf{w}(k)$ (the leakage factor applies to the weight in equation 11-1) forces the algorithm to continue to adapt even after it reaches its minimum value. While this can mean that the leaky LMS does not achieve quite so accurate a measure of the minimum mean square error, the sensitivity to errors, or to small values in the input is reduced when you use the leakage factor. Typically, set `lf` between 0.9 (considered very leaky) and 1.0, meaning no leakage. If you specify `lf` as empty, it defaults to one.

When you check the contents of `s` after you use `initlms` MATLAB displays the structure elements, rather than the input argument names. To help you remember which element in `s` corresponds to each input argument to `initlms`, the following table provides the mapping.

initlms Argument	Structure Field	Argument Contents
`wo`	`s.coeffs`	LMS FIR filter coefficients. Should be initialized with the initial coefficients for the FIR filter prior to adapting. You need (adapting filter order + 1) entries in `s.coeffs`. Updated filter coefficients are returned in `s.coeffs` when you use `s` as an output argument.
`mu`	`s.step`	Sets the LMS algorithm step size. Determines both how quickly and how closely the adative filter adapts to the filter solution.

initlms Argument	Structure Field	Argument Contents
zi	s.states	Returns the states of the FIR filter after adaptation. This is an optional element. If omitted, it defaults to a zero vector of length equal to the filter order. When you use adaptlms in a loop structure, use this element to specify the initial filter states for the adapting FIR filter.
lf	s.leakage	Specifies the LMS leakage parameter. Allows you to implement a leaky LMS algorithm. Including a leakage factor can improve the results of the algorithm by forcing the LMS algorithm to continue to adapt even after it reaches a minimum value. This is an optional field. Defaults to one if omitted (specifying no leakage) or set to empty, [ ].
	s.iter	Total number of iterations in the adaptive filter run. Although you can set this in s, you should not. Consider it a read-only value.

For example, after you use initlms to create s, MATLAB returns the structure shown when you enter s at the prompt. In this example, we created s for a 31st-order filter.

```
s

s =

      coeffs: [1x32 double]
      states: [31x1 double]
        step: 0.8000
     leakage: 1
        iter: 0
```

Examples

To use adaptlms, you must provide at least two input arguments that define the LMS algorithm to use — w0 and mu. Structure s comprises these data sets

and forms the initialization for `adaptlms`. In this example, use `initlms` to configure `s` to identify an unknown 31st-order FIR filter. To see this structure in use, refer to `adaptlms`.

```
w0 = zeros(1,32);       % Intial filter coefficients
mu = 0.8;               % LMS step size.
s = initlms(w0,mu);
```

See Also

`adaptlms`, `initnlms`, `adaptnlms`, `initrls`

Reference

Hayes, Monson. H., *Statistical Digital Signal Processing and Modeling*, John Wiley and Sons, Inc, 1996.

initnlms

Purpose

Configure the initialization structure used as an input argument to `adaptnlms`

Syntax

```
s = initnlms(w0,mu)
s = initnlms(w0,mu,zi)
s = initnlms(w0,mu,zi,lf)
s = initnlms(w0,mu,zi,lf,offset)
```

Description

`s = initnlms(w0,mu)` returns the fully populated structure `s` that you use when you call `adaptnlms`. Vector `w0` contains the initial values of the filter coefficients. Its length should equal the order of the adapting FIR filter plus one. `mu` is the Normalized Least Mean Square (NLMS) algorithm step size. The step size you specify determines both the time it takes for the NLMS algorithm to converge to a solution and the accuracy of that solution (how closely the result approaches the minimum least mean square error). Generally, small step sizes adapt more slowly but more closely and large step sizes adapt more quickly with larger error compared to the true minimum mean square error.

In vector form, the NLMS algorithm is

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \mu_n e(k)\mathbf{x}(k)$$

where

$$\mu_n = \frac{1}{\varepsilon + \|x(k)\|^2}$$

with vector **w** containing the weights applied to the filter coefficients and vector **x** containing the input data. *e(k)* (equal to desired signal - filtered signal) is the error at time *k* and is the quantity the NLMS algorithm seeks to minimize. μ (`mu`) is the step size. As you specify `mu` smaller, the correction to the filter weights gets smaller for each sample and the NLMS error falls more slowly. Larger `mu` 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 good convergence rate and stability, select `mu` within the following bounds:

$$0 < \mu < \frac{1}{N\{InputSignalPower\}}$$

where *N* is the number of samples in the signal.

`s = initnlms(w0,mu,zi)` adds input argument `zi` to specify the filter initial conditions. When you omit `zi` or specify it as empty, `[]`, `initnlms` defaults to `zi` equal to a zero vector of length `[length(w0)-1]`. For conditional processing such as using `adaptnlms` in a for-loop, specifying the initial conditions is very important. Each iteration of the NLMS algorithm uses the weights from the prior interation. You supply the initial conditions so the first iteration has a set of prior filter weights to start from.

`s = initnlms(w0,mu,zi,lf)` specifies the leakage factor `lf`. Including the leakage factor can improve the behavior of the algorithm. Leaking the weight $\mathbf{w}(k)$ (the leakage factor applies to the weight in equation 11-2) forces the algorithm to continue to adapt even after it reaches its minimum value. This can mean that the leaky NLMS does not achieve quite so accurate a measure of the minimum mean square error. However, the sensitivity to errors, or to small values in the input is reduced when you use the leakage factor. Typically, set `lf` between 0.9 (considered very leaky) and 1.0, meaning no leakage. If you specify `lf` as empty, it defaults to one.

`s = initnlms(w0,mu,zi,lf,offset)` specifies an optional offset for the normalization term. This is useful to avoid divide by zero (or very small numbers) conditions when the square of the input data norm becomes very small. If `offset` is specified as empty, it defaults to zero.

When you check the contents of `s` after you use `initnlms` MATLAB displays the structure elements, rather than the input argument names. To help you

remember which element in `s` corresponds to each input argument to `initlms`, the following table provides the mapping.

initnlms Argument	Structure Field	Argument Contents
`wo`	`s.coeffs`	NLMS FIR filter coefficients. Should be initialized with the initial coefficients for the FIR filter prior to adapting. You need (adapting filter order + 1) entries in `s.coeffs`. Updated filter coefficients are returned in `s.coeffs` when you use `s` as an output argument.
`mu`	`s.step`	Sets the NLMS algorithm step size. Determines both how quickly and how closely the adative filter adapts to the filter solution.
`zi`	`s.states`	Returns the states of the FIR filter after adaptation. This is an optional element. If omitted, it defaults to a zero vector of length equal to the filter order. When you use `adaptlms` in a loop structure, use this element to specify the initial filter states for the adapting FIR filter.
`lf`	`s.leakage`	Specifies the NLMS leakage parameter. Allows you to implement a leaky NLMS algorithm. Including a leakage factor can improve the results of the algorithm by forcing the NLMS algorithm to continue to adapt even after it reaches a minimum value. This is an optional field. Defaults to one if omitted (specifying no leakage) or set to empty, [ ].

initnlms Argument	Structure Field	Argument Contents
offset	s.offset	Specifies an optional offset for the normalization term. Use this to avoid divide by zero (or by very small numbers) when the square of input data norm becomes very small. When omitted, it defaults to zero.
	s.iter	Total number of iterations in the adaptive filter run. Although you can set this in s, you should not. Consider it a read-only value.

For example, after you use `initnlms` to create `s`, MATLAB returns the structure shown when you enter `s` at the prompt. In this example, we created `s` for a 31st-order filter.

```
s

s =

     coeffs: [1x32 double]
     states: [31x1 double]
       step: 0.8000
    leakage: 1
       iter: 0
```

See Also

`adaptnlms`, `adaptlms`, `adaptrls`, `initlms`, `initkalman`

Reference

Hayes, M.H., *Statistical Digital Signal Processing and Modeling*, John Wiley and Sons, 1996

initrls

Purpose

Configure the initialization structure used as an input argument to `adaptrls`

Syntax

```
s = initrls(w0,p0,lambda)
s = initrls(w0,p0,lambda,zi)
s = initrls(w0,p0,lambda,zi,alg)
```

Description

`s = initrls(w0,p0,lambda)` returns the fully populated structure `s` that you use when you call `adaptrls`. Vector `w0` contains the initial values of the filter coefficients. Its length should equal the order of the adapting FIR filter plus one.

`p0` is the inverse of the initial error covariance matrix. It must be an Hermitian symmetric square matrix with dimensions equal to `length(w0)`.

`lambda` is the forgetting factor, also called the exponential weighting factor, in the recursive least squares (RLS) algorithm. RLS algorithms calculate the least squares error vector using all previous data; data from long ago is given the same weight as newly received data. It is possible for bad data from the past to affect the current solution. In RLS terms this is called inifinite memory. lambda lets you determine how the RLS algorithm treats old data. When you specify `lambda`, the RLS algorithm applies a weighting factor to sample data using `lambda` in the following calculation:

weighting factor for a sample = $\texttt{lambda}^{(\text{sample age})}$

where `sample age` represents the age of the sample being weighted. For a recent sample, sample age might be 1 or 2 or 10, meaning that the sample is 1, 2, or 10 iterations old. A sample from 100 iterations earlier would have sample age = 100 and a weighting factor of $0.9^{100} = 2.6 \times 10^{-5}$ when `lambda` = 0.9. Thus earlier samples have less affect on the least squares error vector than recent samples. `lambda` should satisfy 0 < `lambda` <= 1, where `lambda = 1` denotes infinite memory — all previous data is equally weighted in the RLS algorithm.

`s = initrls(w0,p0,lambda,zi)` adds input argument `zi` to specify the filter initial conditions. When you omit `zi` or specify it as empty, `[]`, `initrls` defaults to `zi` equal to a zero vector of length `[length(w0)-1]`.

`s = initrls(w0,p0,lambda,zi,alg)` adds the input argument '`alg`' that specifies which version of the RLS algorithm gets used in RLS computations.

String `alg` can be either '**`direct`**' (default) to use the RLS algorithm or '**`sqrt`**' to use the more stable square root (QR decomposition) RLS algorithm.

String alg	Description
`direct`	Specifies the standard RLS algorithm to determine the least squares weight vector for the adaptive filter weights. This is the default setting.
`sqrt`	Specifies the QR decomposition RLS algorithm to determine the least squares weight vector for the adaptive filter coefficients. The QR algorithm applies the QR decomposition to the incoming data matrix, rather than working with the correlation matrix of the input data as the RLS algorithm does. In the RLS algorithm, the input data correlation matrix is averaged over time. Working directly with the input data matrix makes the QR version more stable numerically.

When you check the contents of `s` after you use `initrls` MATLAB displays the structure elements, rather than the input argument names. To help you

remember which element in s corresponds to each input argument to initrls, the following table provides the mapping

initrls Argument	Structure Field	Argument Contents
w0	s.coeffs	NLMS FIR filter coefficients. Should be initialized with the initial coefficients for the FIR filter prior to adapting. You need (adapting filter order + 1) entries in s.coeffs. Updated filter coefficients are returned in s.coeffs when you use s as an output argument.
p0	s.invcov	The inverse of the input covariance matrix. Should be initialized with the initial input covariance matrix inverse. p0 has dimensions equal to the filter order, or length(w0)-1. When you use s as an output argument to adaptrls, with the '**direct**' algorithm specified, adaptrls returns the updated matrix in s.invcov.
lambda	s.lambda	The forgetting factor that defines how the RLS algorithm weighs more recent and less recent samples. While lambda can be between 0 and 1, usually you set 0.9 < lambda ≤ 1.0
zi	s.states	Returns the states of the FIR filter after adaptation. This is an optional element. If omitted, it defaults to a zero vector of length equal to the filter order. When you use adaptrls in a loop structure, use this element to specify the initial filter states for the adapting FIR filter.

initrls Argument	Structure Field	Argument Contents
	s.gain	RLS gain is a read-only value. For each iteration of the algorithm, the gain from the previous iteration feeds into the algorithm. For the first iteration, the default gain is [ ]. After the algorithm finishes adapting, s.gain contains the final gain value.
	s.iter	Total number of iterations in the adaptive filter run. Although you can set this in s, you should not. Consider it a read-only value.
alg	s.alg	Specifies the RLS algorithm to use for the adapting process. This is an optional element. Enter either '**direct**' for the conventional RLS algorithm or '**sqrt**' for the more stable square root (QR) method. **direct** is the default algorithm; used when you omit alg.

Examples

Create the structure s that you use with adaptrls. In this example we plan to identify an unknown 32nd-order FIR filter. Set w0 to contain 33 initial filter coefficients. Since p0 is the inverse correlation matrix, our correlation matrix must have been the identity matrix with 0.2 on the diagonal and zeros everywhere else. By default we use the direct RLS algorithm and we let all earlier samples be weighted equally, lambda = 1.

```
w0 = zeros(1,33);       % Intial filter coefficients
p0 = 5*eye(33);        % Initial input correlation matrix inverse
lambda = 1;             % Exponential memory weighting factor
s = initrls(w0,p0,lambda);
```

To see the results of using this s, refer to adaptrls.

See Also

adaptrls, initkalman, initlms, initnlms

References

Hayes, Monson. H., *Statistical Digital Signal Processing and Modeling*, John Wiley and Sons, Inc, 1996.

Purpose Configure the initialization structure used as an input argument to `adaptsd`

Syntax

```
s = initsd(w0,mu)
s = initsd(w0,mu,zi)
s = initsd(w0,mu,zi,lf)
```

Description `s = initsd(w0,mu)` returns the fully populated structure `s` that you use when you call `adaptsd`. Vector `w0` contains the initial values of the filter coefficients. Its length should equal the order of the adapting FIR filter plus one. `mu` is the sign data least mean square (SDLMS) algorithm step size. The step size you specify determines both the time it takes for the SDLMS algorithm to converge to a solution and the accuracy of that solution (how closely the result approaches the minimum least mean square error). Generally, small step sizes adapt more slowly but more closely and large step sizes adapt more quickly with larger error compared to the true minimum mean square error.

In matrix form, the SDLMS algorithm is

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \mu e(k) sgn[\mathbf{x}(k)] \text{ , } sgn[\mathbf{x}(k)] = \begin{cases} 1, & \mathbf{x}(k) > 0 \\ 0, & \mathbf{x}(k) = 0 \\ -1, & \mathbf{x}(k) < 0 \end{cases} \quad \textbf{(13-2)}$$

with vector **w** containing the weights applied to the filter coefficients and vector **x** containing the input data. *e(k)* (equal to desired signal - filtered signal) is the error at time *k* and is the quantity the SDLMS algorithm seeks to minimize. μ (`mu`) is the step size. As you specify `mu` smaller, the correction to the filter weights gets smaller for each sample and the SDLMS error falls more slowly. Larger `mu` 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 good convergence rate and stability, select `mu` within the following bounds:

$$0 < \mu < \frac{1}{N\{InputSignalPower\}}$$

where *N* is the number of samples in the signal. Also, define `mu` as a power of two.

`s = initsd(w0,mu,zi)` adds input argument `zi` to specify the filter initial conditions. When you omit `zi` or specify it as empty, `[]`, `initsd` defaults to `zi` equal to a zero vector of length `[length(w0)-1]`. For conditional processing such as using `adaptse` in a for-loop, specifying the initial conditions is very important. Each iteration of the SDLMS algorithm uses the weights from the prior interation. You supply the initial conditions so the first iteration has a set of prior filter weights to start from.

`s = initsd(w0,mu,zi,lf)` specifies the leakage factor `lf`. Including the leakage factor can improve the behavior of the algorithm. Leaking the weight $\mathbf{w}(k)$ (the leakage factor applies to the weight in Equation 13-2) forces the algorithm to continue to adapt even after it reaches its minimum value. This can mean that the leaky SDLMS does not achieve quite so accurate a measure of the minimum mean square error. However, the sensitivity to errors, or to small values in the input is reduced when you use the leakage factor. Typically, set `lf` between 0.9 (considered very leaky) and 1.0, meaning no leakage. If you specify `lf` as empty, it defaults to one.

See Also

`adaptsd`, `initse`, `initss`, `adaptlms`, `adaptrls`, `initlms`, `initnlms`

References

Hayes, Monson. H., *Statistical Digital Signal Processing and Modeling*, John Wiley and Sons, Inc, 1996.

initse

Purpose

Configure the initialization structure used as an input argument to `adaptse`

Syntax

```
s = initse(w0,mu)
s = initse(w0,mu,zi)
s = initse(w0,mu,zi,lf)
```

Description

`s = initse(w0,mu)` returns the fully populated structure `s` that you use when you call `adaptse`. Vector `w0` contains the initial values of the filter coefficients. Its length should equal the order of the adapting FIR filter plus one. `mu` is the sign error least mean square (SELMS) algorithm step size. The step size you specify determines both the time it takes for the SELMS algorithm to converge to a solution and the accuracy of that solution (how closely the result approaches the minimum least mean square error). Generally, small step sizes adapt more slowly but more closely and large step sizes adapt more quickly with larger error compared to the true minimum mean square error.

In matrix form, the SELMS algorithm is

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \mu \operatorname{sgn}[e(k)][\mathbf{x}(k)] \;,\; sgn[e(k)] = \begin{cases} 1, & e(k) > 0 \\ 0, & e(k) = 0 \\ -1, & e(k) < 0 \end{cases} \quad \textbf{(13-3)}$$

with vector **w** containing the weights applied to the filter coefficients and vector **x** containing the input data. *e(k)* (equal to desired signal - filtered signal) is the error at time *k* and is the quantity the SELMS algorithm seeks to minimize. μ (`mu`) is the step size. As you specify `mu` smaller, the correction to the filter weights gets smaller for each sample and the SELMS error falls more slowly. Larger `mu` 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 good convergence rate and stability, select `mu` within the following bounds:

$$0 < \mu < \frac{1}{N\{InputSignalPower\}}$$

where N is the number of samples in the signal. Also, define `mu` as a power of two.

`s = initse(w0,mu,zi)` adds input argument `zi` to specify the filter initial conditions. When you omit `zi` or specify it as empty, `[ ]`, `initss` defaults to `zi` equal to a zero vector of length `[length(w0)-1]`. For conditional processing such as using `adaptss` in a for-loop, specifying the initial conditions is very important. Each iteration of the SELMS algorithm uses the weights from the prior interation. You supply the initial conditions so the first iteration has a set of prior filter weights to start from.

`s = initnlms(w0,mu,zi,lf)` specifies the leakage factor `lf`. Including the leakage factor can improve the behavior of the algorithm. Leaking the weight $\mathbf{w}(k)$ (the leakage factor applies to the weight in Equation 13-3) forces the algorithm to continue to adapt even after it reaches its minimum value. This can mean that the leaky SELMS does not achieve quite so accurate a measure of the minimum mean square error. However, the sensitivity to errors, or to small values in the input is reduced when you use the leakage factor. Typically, set `lf` between 0.9 (considered very leaky) and 1.0, meaning no leakage. If you specify `lf` as empty, it defaults to one.

See Also

`adaptsd`, `initse`, `initss`, `adaptlms`, `adaptrls`, `initlms`, `initnlms`

References

Hayes, Monson. H., *Statistical Digital Signal Processing and Modeling*, John Wiley and Sons, Inc, 1996.

Purpose

Configure the initialization structure used as an input argument to `adaptss`

Syntax

```
s = initss(w0,mu)
s = initss(w0,mu,zi)
s = initss(w0,mu,zi,lf)
```

Description

`s = initss(w0,mu)` returns the fully populated structure `s` that you use when you call `adaptss`. Vector `w0` contains the initial values of the filter coefficients. Its length should equal the order of the adapting FIR filter plus one. `mu` is the sign sign least mean square (SSLMS) algorithm step size. The step size you specify determines both the time it takes for the SSLMS algorithm to converge to a solution and the accuracy of that solution (how closely the result approaches the minimum least mean square error). Generally, small step sizes adapt more slowly but more closely and large step sizes adapt more quickly with larger error compared to the true minimum mean square error.

In matrix form, the SSLMS algorithm is

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \mu\,\mathrm{sgn}[e(k)]\,\mathrm{sgn}[\mathbf{x}(k)] \text{ ,}$$

$$\mathrm{sgn}[z(k)] = \begin{cases} 1, & z(k) > 0 \\ 0, & z(k) = 0 \\ -1, & z(k) < 0 \end{cases} \qquad \textbf{(13-4)}$$

where *z(k)* is

$$z(k) = [e(k)]\,\mathrm{sgn}[\mathbf{x}(k)]$$

Vector **w** contains the weights applied to the filter coefficients and vector **x** contains the input data. *e(k)* (equal to desired signal - filtered signal) is the error at time *k* and is the quantity the SSLMS algorithm seeks to minimize. μ (`mu`) is the step size. As you specify `mu` smaller, the correction to the filter weights gets smaller for each sample and the SSLMS error falls more slowly. Larger `mu` 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 good convergence rate and stability, select `mu` within the following bounds:

$$0 < \mu < \frac{1}{N\{InputSignalPower\}}$$

where N is the number of samples in the signal. Also, define `mu` as a power of two.

`s = initss(w0,mu,zi)` adds input argument `zi` to specify the filter initial conditions. When you omit `zi` or specify it as empty, `[]`, `initss` defaults to `zi` equal to a zero vector of length `[length(w0)-1]`. For conditional processing such as using `adaptss` in a for-loop, specifying the initial conditions is very important. Each iteration of the SSLMS algorithm uses the weights from the prior interation. You supply the initial conditions so the first iteration has a set of prior filter weights to start from.

`s = initss(w0,mu,zi,lf)` specifies the leakage factor `lf`. Including the leakage factor can improve the behavior of the algorithm. Leaking the weight $\mathbf{w}(k)$ (the leakage factor applies to the weight in Equation 13-4) forces the algorithm to continue to adapt even after it reaches its minimum value. This can mean that the leaky SSLMS does not achieve quite so accurate a measure of the minimum mean square error. However, the sensitivity to errors, or to small values in the input is reduced when you use the leakage factor. Typically, set `lf` between 0.9 (considered very leaky) and 1.0, meaning no leakage. If you specify `lf` as empty, it defaults to one.

See Also

`adaptsd`, `initse`, `initss`, `adaptlms`, `adaptrls`, `initlms`, `initnlms`

References

Hayes, Monson. H., *Statistical Digital Signal Processing and Modeling*, John Wiley and Sons, Inc, 1996.

isallpass

Purpose Test quantized filters to determine whether they are allpass structures

Syntax

```
flag = isallpass(f)
flag = isallpass(f,k)
```

Description `flag = isallpass(f)` determines whether the filter object `f` is an allpass filter, returning `1` if true and `0` if false.

`flag = isallpass(f,k)` determines whether the `k`-th section of the filter object `f` is an allpass section and returns `1` if true and `0` if false.

Since lattice coupled allpass filters always have allpass sections, this function always returns `1` for filters whose structure is `latticeca`.

See Also `isfir`, `islinphase`, `ismaxphase`, `isminphase`, `isreal`, `issos`, `isstable`

Purpose Test quantized filters to see whether they are FIR filters

Syntax

```
flag = isfir(hq)
flag = isfir(hq,k)
```

Description

`flag = isfir(hq)` determines whether quantized filter `hq` is an FIR filter, returning `flag` equal to `1` when the quantized filter is an FIR filter, and `0` when it is IIR.

`flag = isfir(hq,k)` determines whether the `k`th-section of quantized filter `hq` is an FIR filter, returning `flag` equal to `1` when the `k`th-section is an FIR filter and `0` when it is IIR.

`isfir(hq)` looks at filter `hq` and determines whether the filter, in transfer function form, has a scalar for its denominator. If it does, it is an FIR filter.

Examples

```
hq = qfilt;
isfir(hq)
ans =

     1
```

returns `1` for the status of filter `hq`; the filter is an FIR structure with denominator reference coefficient equal to one.

See Also `isallpass`, `islinphase`, `ismaxphase`, `isminphase`, `isreal`, `issos`, `isstable`

isfixed

Purpose

Test and return whether a quantizer is fixed point

Syntax

```
res = isfixed(q)
```

Description

`res = isfixed(q)` returns `res = 1` (logical true) when `q` is a fixed-point quantizer. When `q` is not fixed point, `isfixed` returns `res = 0`. When you develop programs that use one or more quantizers, you may find this function useful to determine the mode of a quantizer within your program, and respond to the returned value.

Examples

Demonstrate the results of testing both fixed-point and nonfixed-point quantizers:

```
q = quantizer('double'); % Create a floating-point quantizer
res = isfixed(q)
res =
0
q = quantizer('ufixed'); % Create a fixed-point quantizer
res = isfixed(q)
res =
1
```

See Also

`isfloat`, `quantizer`

Purpose Test and return whether a quantizer is floating point

Syntax

```
res = isfloat(q)
```

Description

`res = isfloat(q)` returns `res = 1` (logical true) when `q` is a floating-point quantizer. When `q` is not floating point, `isfloat` returns `res = 0`. When you develop programs that use one or more quantizers, you may find this function useful to determine the mode of a quantizer within your program, and respond to the returned value.

Examples

Demonstrate the results of testing both fixed-point and nonfixed-point quantizers:

```
q = quantizer('double'); % Create a floating-point quantizer
res = isfloat(q)
res =
1
q = quantizer('ufixed'); % Create a fixed-point quantizer
res = isfloat(q)
res =
0
```

See Also

`isfixed`, `quantizer`

isnone

Purpose

Determine and return whether a quantizer has quantization mode equal to none

Syntax

```
res = isnone(q)
```

Description

`res = isnone(q)` returns `res = 1` (logical true) when qauntizer `q` has mode equal to `none`. Recall that quantizers may have one of five modes—none, double, fixed, float, or single. When you develop programs that use one or more quantizers, you may find this function useful for determining the mode of a quantizer within your program, and responding to the returned value.

Examples

Create and test two quantizers—one operating in double mode and the other with mode set to `none`:

```
q = quantizer('double'); % Create a double data type quantizer
res = isnone(q)
res =
0
q = quantizer('none'); % Create a quantizer with mode = none
res = isnone(q)
res =
1
```

See Also

`isfixed`, `isfloat`

Purpose

Test quantized filters to see whether they are linear phase

Syntax

```
flag = islinphase(hq)
flag = islinphase(hq,k)
```

Description

`flag = islinphase(hq)` determines if the quantized filter `hq` is linear phase, and returns `1` if true and `0` if false.

`flag = islinphase(hq,k)` determines if the `k`th-section of the filter `hq` is a linear phase section and returns `1` if true and `0` if false.

The determination is based on the reference coefficients. A filter has linear phase if it is FIR and its transfer function coefficients are are symmetric or antisymmetric. If it is IIR and it has poles on or outside the unit circle and both numerator and denominator are symmetric or antisymmetric, it is linear phase also.

Examples

This IIR filter has linear phase.

```
num=[1 0 0 0 0 -1];
den=[1 -1];
hq = qfilt('df2t',{num,den});
islinphase(hq)
ans =

     1
```

See Also

`isallpass`, `isfir`, `ismaxphase`, `isminphase`, `isreal`, `issos`, `isstable`

ismaxphase

Purpose

Test quantized filters to see whether they are maximum phase filters

Syntax

```
flag = ismaxphase(hq)
flag = ismaxphase(hq,k)
```

Description

`flag = ismaxphase(hq)` determines whether filter `hq` is maximum phase, returning `1` if true and `0` if false.

`flag = ismaxphase(hq,k)` determines if the `k`th-section of filter `hq` is a maximum phase section and returns `1` if true and `0` if false.

The determination is based on the reference coefficients. A filter is maximum phase when the zeros of its transfer function are on or outside the unit circle, or when the numerator is a scalar.

Examples

```
hq = qfilt;
ismaxphase(hq)
```

returns `1` so this is a maximum phase quantized filter. Notice that the filter coefficients (zeros) are 1.0 before quantization. Compare to `isminphase`.

See Also

`isallpass`, `isfir`, `islinphase`, `isminphase`, `isreal`, `issos`, `isstable`

Purpose

Test quantized filters to see if they are minimum phase

Syntax

```
flag = isminphase(hq)
flag = isminphase(hq,k)
```

Description

`flag = isminphase(hq)` determines if the filter `hq` is minimum phase and returns `1` if true and `0` if false.

`flag = isminphase(hq,k)` determines if the `k`-th section of the filter `hq` is a minimum phase section and returns `1` if true and `0` if false.

The determination is based on the reference coefficients. A filter is minimum phase when the zeros of its transfer function are on or inside the unit circle, or the numerator is a scalar.

Examples

This example creates a minimum phase quantized filter.

```
hq = qfilt;
isminphase(hq)
```

If you look at the example in `ismaxphase`, you may notice that this filter is also maximum phase. Since both the poles and zeros of the filter lie on the unit circle, it passes the tests for minimum and maximum phase designation.

See Also

`isallpass`, `isfir`, `islinphase`, `ismaxphase`, `isreal`, `issos`, `isstable`,

isreal

Purpose

Test quantized filters for purely real coefficients

Syntax

```
r = isreal(hq)
```

Description

`r = isreal(hq)` returns `r = 1` (or true) if all reference filter coefficients for the quantized filter `hq` are real, and returns `r = 0` (or false) otherwise.

`isreal(hq)` returns `1` if all filter coefficients in quantized filter `hq` have zero imaginary part. Otherwise, `isreal(hq)` returns a `0` indicating that the filter is complex. Complex quantized filters have one or more coefficients with nonzero imaginary parts.

Note Quantizing a filter cannot make a real filter into a complex filter.

Examples

```
% Create a reference filter.
[b,a] = ellip(2,0.5,20,0.4);

% Create a quantized filter from the reference filter.

hq = qfilt('df2t',{b,a});

% Test if all filter coefficients are real.

r = isreal(hq)

r =
     1
```

See Also

`isfir`, `islinphase`, `ismaxphase`, `isminphase`, `issos`, `isstable`, `isallpass`

issos

Purpose Test whether quantized filters are composed of second-order sections

Syntax

```
flag = issos(hq)
```

Description

`flag = issos(hq)` determines whether quantized filter `hq` consists of second-order sections. Returns `1` if all sections of quantized filter `hq` have order less than or equal to two, and `0` otherwise.

Examples

```
warning off
[b,a] = butter(5,.5);
hq = sos(qfilt('ref',{b,a}));
v = issos(hq)
v =

     1
hq.statespersection

ans =

     1     2     2
```

Quantized filter `hq` is in second-order section form.

See Also

`isallpass`, `isfir`, `islinphase`, `ismaxphase`, `isminphase`, `isreal`, `isstable`

Purpose

Test whether a quantized filter is stable

Syntax

```
r = isstable(hq)
r = isstable(hq,k)
```

Description

`r = isstable(hq)` tests quantized filter `hq` to determine whether its poles are inside the unit circle. If the poles lie on or outside the circle, `isstable` returns `r = 0`. If the poles are inside the circle, `isstable` returns `r = 1`.

`r = isstable(hq,k)` returns the stability of the `k`th-section of a multiple section quantized filter. Based on the locations of the poles of the specified section, `isstable` returns `r = 1` if the filter section is stable, and `0` otherwise.

To determine the filter stability, `isstable` checks the quantized filter coefficients. When the poles lie on or inside the unit circle, the quantized filter is stable. FIR filters are stable by design since the defining transfer functions do not have denominator polynomials.

Examples

Since filter stability is very important in your design process, use isstable to deterime whether you quantized IIR filter is indeed stable:

```
hq = qfilt;
isstable(hq)
ans =
    1
```

See Also

`isallpass`, `isfir`, `islinphase`, `ismaxphase`, `isminphase`, `isreal`, `issos`, `zplane`

length

Purpose Return the length of a quantized FFT

Syntax

```
length(f)
```

Description `length(f)` returns the value of the length property of quantized FFT `f`. The value of the length property must be a positive integer that is also a power of the radix of the quantized FFT (`f.radix`). The length of the FFT is the length of the data vector that the FFT operates on.

Examples

```
f = qfft;
length(f)
```

returns the default 16 for the length of the FFT.

See Also `qfft`, `get`, `set`

Purpose

Detect limit cycles in a quantized filter

Syntax

```
limitcycle(hq)
limitcycle(hq, ntrials, inputlength, stopcriterion, displaytype)
[limitcycletype, zi, stateperiod, statesequence,...
   overflowsperstep, trial, section] = limitcycle(hq, ...)
```

Description

`limitcycle(hq)` runs 20 Monte Carlo trials with quantized filter `hq`. Each trial uses a new set of initial states (determined randomly) and zero input vector of length 100. Monte Carlo processing stops if a zero-input limit cycle is detected in quantized filter `hq`. At completion, `limitcycle` returns one of the following strings:

- `'granular'` indicating that a granular overflow occurred
- `'overflow'` indicating that an overflow limitcycle occurred
- `'none'` indicating that no limit cycles were detected during the Monte Carlo trials

`limitcycle(hq, ntrials, inputlength, stopcriterion, displaytype)` lets you set the following arguments:

- `ntrials` — the number of monte carlo trials (default is 20).
- `inputLength` — the length of the zero vector used as input to the filter (default is 100).
- `stopcriterion` — the criterion for stopping the Monte Carlo trials processing. `stopcriterion` can be set to `'`**`either`**`'` (the default), `'`**`granular`**`'`, `'`**`overflow`**`'`, or `'`**`none`**`'`. If `stopcriterion` is:

stopcriterion	Description
`'`**`either`**`'`	Monte Carlo trials will stop when either a granular or overflow limit cycle is detected
`'`**`granular`**`'`	Monte Carlo trials stop when a granular limit cycle was detected

stopcriterion	Description
'overflow'	Monte Carlo trials stop when an overflow limit cycle was detected
'none'	Monte Carlo trials do not stop until all trials have been run

- `displaytype` — the display type. When `displaytype` is nonzero, `limitcycle` displays messages about the progress of the Monte Carlo trials.

`[LimitcycleType, Zi, StatePeriod, StateSequence, overflowsperstep, trial, section] = limitcycle(hq,...)` also returns

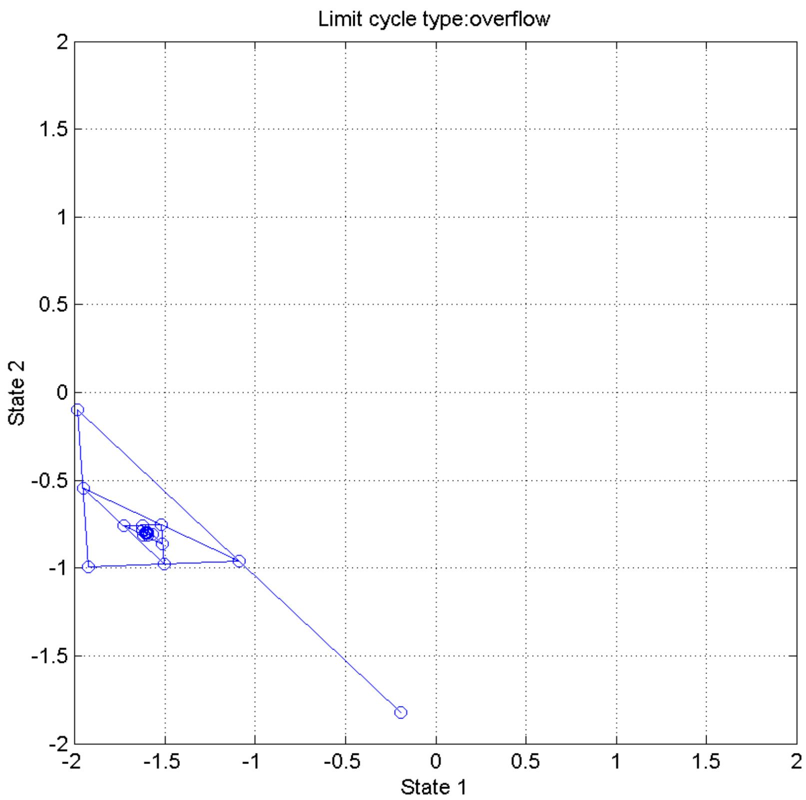
- `limitcycletype` — one of '**granular**' to indicate that a granular overflow occurred; '**overflow**' to indicate that an overflow limitcycle occurred; or '**none**' to indicate that no limit cycles were detected during the Monte Carlo trials.
- `zi` — the initial condition that caused the limit cycle.
- `stateperiod` — an integer indicating the repeat period of the limit cycle (-1 if the filter converged and the last state is zero, 0 if the last state is not zero and no limit cycle was detected).
- `statesequence` — a matrix containing the sequence of states at every time step (one matrix column per time step). The final conditions are in the last column of `statesequence zf = statesequence(:,end)`. The initial conditions of the section are in the first column of `statesequence zi = statesequence(:,1)`.
- `overflowsperstep` — a cell array that contains one vector of integers for each section of the filter that indicates the total number of overflows that occurred during each time step. The overflows from the kth-section are found in `overflowsperstep{k}`.
- `trial` — the number of the trial on which Monte Carlo processing stopped.
- `section` — the number of the section in which the limitcycle was detected.

Only the parameters of the last limit cycle are returned. If Monte Carlo processing does not detect any limit cycles, the parameters of the last Monte Carlo trial are returned.

Examples

In this example, there is a region of initial conditions in which no limit cycles occur, and a region where they do. If no limit cycles are detected before the Monte Carlo trials are over, the state sequence spirals to zero. When a limit cycle is found, the states do not end at zero. Each time you run this example, it uses a different sequence of random initial conditions, so the plot you get may differ from the one displayed in the following figure.

```
a = [-1 -1; 0.5 0];
b = [0; 1];
c = [1 0];
d = 0;
hq = qfilt('statespace',{a,b,c,d},'overflowmode','wrap');
[limitcycletype, zi, stateperiod, statesequence] = limitcycle(hq);
plot(statesequence(1,:), statesequence(2,:),'-o')
xlabel('State 1');
ylabel('State 2');
axis([-2 2 -2 2]); axis square; grid
title(['Limit cycle type:',limitcycletype])
```



See Also

`freqz`, `nlm`

Purpose Return the maximum value of a quantizer object before quantization

Syntax

```
max(q)
```

Description `max(q)` is the maximum value before quantization during a call to `quantize(q,...)` for quantizer `q`. This value is the maximum value encountered over successive calls to `quantize` and is reset with `reset(q)`. `max(q)` is equivalent to `get(q,'max')` and `q.max`.

Examples

```
q = quantizer;
warning on
y = quantize(q,-20:10);
max(q)
```

returns the value 10 and a warning for 29 overflows.

See Also `min`

min

Purpose Return the minimum value of a quantizer object before quantization

Syntax

```
max(q)
```

Description max(q) is the minimum value before quantization during a call to quantize(q,...) for quantizer q. This value is the minimum value encountered over successive calls to quantize and is reset with reset(q). min(q) is equivalent to get(q,'min') and q.min.

Examples

```
q = quantizer;
warning on
y = quantize(q,-20:10);
min(q)
```

returns the value -20 and a warning for 29 overflows.

See Also max

Purpose Use the noise loading method to estimate the frequency response of a quantized filter

Syntax

```
[h,w,pnn,nf] = nlm(hq,n,l)
[h,w,pnn,nf] = nlm(hq,n,l,'whole')
[h,f,...] = nlm(hq,n,l,fs)
[h,f,...] = nlm(hq,n,l,'whole',fs)
nlm(hq,...)
```

Description `[h,w,pnn,nf] = nlm(hq,n,l)` uses the noise loading method to estimate the complex frequency response of quantized filter `hq`. Using `nlm` returns the complex frequency response `h`, frequency vector `w`, in radians/sample, power spectral density `pnn`, and noise figure `nf`, for the quantized filter `hq`, at `n` equally-spaced points around the upper half of the unit circle. Noise figure `nf` and power spectral density `pnn` are given in dB. `nlm` averages over `l` Monte Carlo trials. The Monte Carlo trials result in a noise-like signal that contains complete frequency content across the spectrum. When you omit `n` or `l` from the command, or leave them empty, `n` defaults to 512 and `l` defaults to 10.

`[h,w,pnn,nf] = nlm(hq,n,l,'whole')` uses `n` points around the entire unit circle, rather than the upper half.

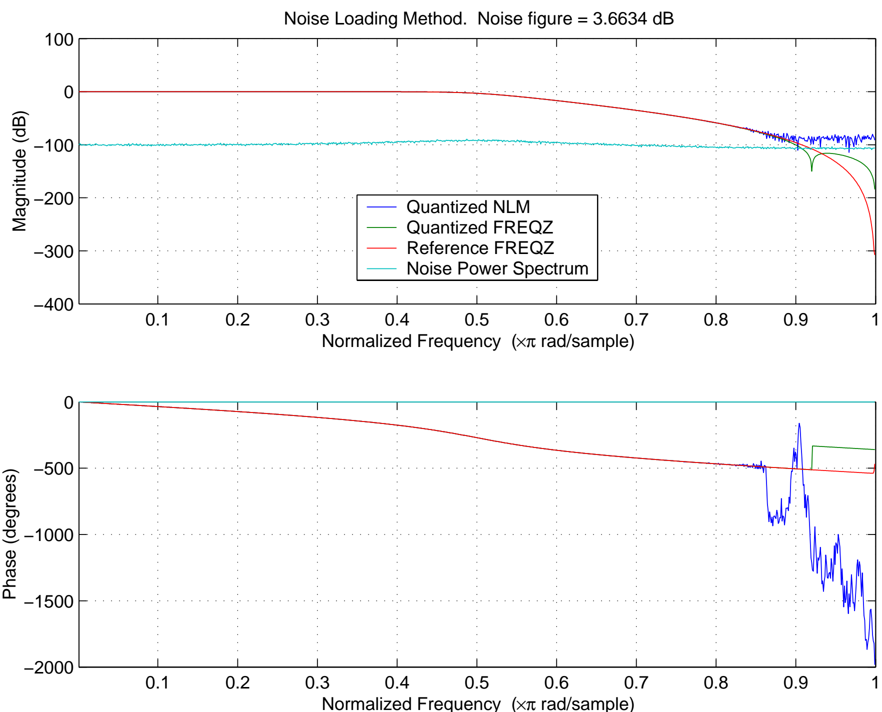
`[h,f,...] = nlm(hq,n,l,fs)` and `[h,f,...] = nlm(hq,n,l,'whole',fs)` returns frequency vector `f`, in Hz, where `fs` is the sampling frequency in Hz.

`nlm(hq,...)` without output arguments plots the magnitude and unwrapped phase of `hq`, comparing the estimated response to the theoretical frequency response calculated by `[h,w] = freqz(hq,n)` in the current figure window.

Examples Use the noise loading method to determine the frequency response of a quantized IIR filter `Hq`.

```
[b,a] = butter(6, 0.5);
hq = qfilt('df2t',{b,a});
nlm(hq,1024,20)
```

For comparison, the plot shows the theoretical response and the response estimated by `nlm`. Additionally, you see the estimated phase response for comparison.



See Also

`freqz`, `qfilt`

References

McClellan, et al., *Computer-Based Exercises for Signal Processing Using MATLAB 5*, Prentice-Hall, 1998, 243.

Purpose

Number of quantization operations performed by a quantizer, quantized filter, or quantized FFT

Syntax

```
noperations(q)
noperations(hq)
noperations(f)
```

Description

`noperations(q)` is the number of quantization operations during a call to `quantize(q,...)` for quantizer `q`. This value accumulates over successive calls to `quantize`. You reset the value of `noperations` to zero by issuing the command `reset(q)`.

`noperations(hq)` is the number of sum and product quantization operations performed during a call to `filter(hq,...)` for quantized filter `hq`.

`noperations(f)` is the number of sum and product quantization operations performed during a call to `fft(f,...)` or `ifft(f,...)` for quantized FFT `f`.

Here's how `noperations` counts quantization operations—each time any data element gets quantized, `noperations` gets incremented by one. Both the real and complex parts count, separately. For example, (complex * complex) counts four quantization operations for products and two for sum—(a+bi)*(c+di) = (a*c - b*d) + (a*d + b*c). In contrast, (real*real) counts one quantization operation.

In addition, the real and complex parts of the inputs get quantized individually. As a result, for a complex input of length 204 elements, `noperations` counts 408 quantizations—204 for the real part of the input and 204 for the complex part.

If any inputs, states, or coefficients are complex-valued, they are all expanded from real values to complex values, with a corresponding increase in the number of quantization operations recorded by `noperations`. In concrete terms, (real*real) requires fewer quantizations than (real*complex) and (complex*complex). Changing all the values to complex because one is complex, such as the coefficient, makes the (real*real) into (real*complex), raising `noperations` count.

Examples

`noperations` returns the number of quantizations it counts. You call it as a quantizer, or as part of designing a quantized filter or quantized FFT.

`noperations` reports the total number of sum and product quantizations for quantized filters and quantized FFTs. For quantizers, `noperations` reports all the quantization operations.

The following code does not perform any adds or multiplies; it quantizes the specified data according to the properties of quantizer `q`:

```
warning on
q=quantizer;
y = quantize(q,-20:10);
noperations(q)
```

and returns 31 and a warning for 29 overflows. Notice that the next example returns an operations count (`NOperations`) that includes only quantizations performed during multiply and add operations.

```
[b,a] = ellip(4,3,20,.6);
hq = qfilt('df2',{b,a},'roundmode','fix');
y=filter(hq,randn(100,1));
```

```
Warning: 32 overflows in QFILT/FILTER.
                    Max           Min      NOverflows     NUnderflows     NOperations
 Coefficient       1.398        0.2259              2               0              10
       Input       2.183        -2.171             27               0             100
      Output      0.9377       -0.8144              0               0             100
Multiplicand       1.972            -2            310               0            1200
     Product           1            -1              0               0            1200
         Sum       1.972        -2.426              3               0            1000
```

```
noperations(hq)
ans =

          2200
```

Returning a total of 2200 operations shows that `noperations` represents the total sum and product quantizations performed, as you see in the result listing from the `filter` call.

See Also

`get`, `qfft`, `qfilt`, `quantizer`

normalize

Purpose

Normalize quantized filter coefficients

Syntax

```
h = normalize(Hq)
```

Description

`h = normalize(Hq)` accounts for quantized filter coefficient overflow by normalizing the quantized filter coefficients in the quantized filter `Hq`. The new quantized filter `h` contains the normalized coefficients. All quantized filter coefficients for `h` stored in the `QuantizedCoefficients` property value are modified to have magnitude less than or equal to one. The result also modifies the `ReferenceCoefficients` property value for `h` accordingly. `normalize` also modifies the `ScaleValues` property value for `h` from that of `Hq`, so that input data to each section of `h` are scaled to compensate for the normalized filter coefficients. The scaling factors used in `normalize` are powers of two. There may be a different scaling factor for each section of the quantized filter. You can apply `normalize` to direct form IIR and FIR filters only. To apply `normalize` to a quantized filter, its property `Hq.FilterStructure` must be one of the following strings:

- `'df1'`
- `'df1t'`
- `'df2'`
- `'df2t'`
- `'fir'`
- `'firt'`
- `antisymmetricfir`
- `'symmetricfir'`

Examples

Create a direct form II transposed quantized filter and use `normalize` to account for overflow.

```
% Create a low pass reference filter in the Signal Proc. Toolbox.

[b,a] = ellip(5,2,40,0.4);

% Create the quantized filter from the reference.

hq = qfilt('df2t',{b,a});

Warning: 5 overflows in coefficients.
```

You are warned that some of the coefficients have overflowed. To account for this overflow, use `normalize` to modify the `ReferenceCoefficients`, `QuantizedCoefficients`, and `ScaleValues` property values for `Hq`.

```
hq = normalize(hq)

hq =
Quantized Direct form II transposed filter
Numerator
     QuantizedCoefficients{1}    ReferenceCoefficients{1}
  (1)       0.365295410156250  0.365289835338219130
  (2)       0.395721435546875  0.395708380608267300
  (3)       0.724884033203125  0.724891008581378560
  (4)       0.724884033203125  0.724891008581378120
  (5)       0.395721435546875  0.395708380608267240
  (6)       0.365295410156250  0.365289835338218350
Denominator
     QuantizedCoefficients{2}    ReferenceCoefficients{2}
  (1)       0.250000000000000   0.250000000000000000
  (2)      -0.541015625000000  -0.541012429707579350
  (3)       0.790557861328125   0.790542752251058410
  (4)      -0.668945312500000  -0.668930473694134720
  (5)       0.365966796875000   0.365965902328318770
  (6)      -0.103698730468750  -0.103697674644671510

   FilterStructure = df2t
       ScaleValues = [0.03125  1]
  NumberOfSections = 1
  StatesPerSection = [5]
 CoefficientFormat = quantizer('fixed', 'round', 'saturate', [16  15])
       InputFormat = quantizer('fixed', 'floor', 'saturate', [16  15])
      OutputFormat = quantizer('fixed', 'floor', 'saturate', [16  15])
MultiplicandFormat = quantizer('fixed', 'floor', 'saturate', [16  15])
     ProductFormat = quantizer('fixed', 'floor', 'saturate', [32  30])
         SumFormat = quantizer('fixed', 'floor', 'saturate', [32  30])
```

Notice that none of the coefficients overflow, and that the `ScaleValues` property value has changed.

See Also

`get`, `set`

Purpose

Return the number of overflows from the most recent FFT or IFFT operation

Syntax

```
noverflows(F)
noverflows(Hq)
noverflows(Hq,'sum')
noverflows (Q)
```

Description

`noverflows(F)` returns the number of overflows resulting from the most recent `fft` or `ifft` operation that used quantized fft (`F`).

`noverflows(Hq)` returns the number of overflows resulting from the most recent filter operation that used quantized filter (`Hq`).

`noverflows(Hq,'sum')` returns the number of overflows that resulted from the most recent `qfilt` operation. When the quantized filter has one section, this returns a scalar. When the filter uses two or more sections, `noverflows` returns a vector containing one element for each filter section.

`noverflows(Q)` returns the number of overflows resulting from the most recent `quantize` operation that used quantizer (`Q`).

Examples

Create a quantized fft `f` and apply it to a data set. Check the number of overflows that result when you use `f`. Then apply f to a second data set and check the overflows again.

```
warning on
n=128;
t = (1:n)/n;
x = sin(2*pi*10*t)/10;
f = qfft('length',n);
plot(t,abs([fft(f,x);fft(x)]))
noverflows(f)
```

returns 24 for the number of overflows and a warning of 24 overflows.

Now, apply `f` to another data set.

```
x = sin(2*pi*10*t)/5;
plot(t,abs([fft(f,x);fft(x)]))
noverflows(f)
```

Now you see 58 overflows.

See Also `get`, `max`, `range`, `reset`

Purpose

Convert a number to a binary string

Syntax

```
num2bin(Hq)
c = num2bin(Hq)
y = num2bin(q,x)
```

Description

`num2bin(Hq)` with no left-hand-side argument displays the quantized coefficients in quantized filter `Hq` as binary strings.

`c = num2bin(Hq)` with a left-hand-side argument `c` returns a cell array of quantized coefficients as binary strings. Cell array `c` inherits the configuration of cell array `Hq.QuantizedCoefficients`.

When the mode of `Hq` is float, double, or single, the coefficients are converted to IEEE Standard 754 style binary strings.

If the mode of `Hq` is fixed, the coefficients are converted to two's complement binary strings.

`y = num2bin(q,x)` converts numeric array `x` into binary strings returned in `y`. When `x` is a cell array, each numeric element of `x` is converted to binary. If `x` is a structure, each numeric field of `x` is converted to binary.

`num2bin` and `bin2num` are inverses of one another, differing in that `num2bin` returns the binary strings in a column.

Examples

```
x=magic(3)/9

x =
    0.8889    0.1111    0.6667
    0.3333    0.5556    0.7778
    0.4444    1.0000    0.2222

q=quantizer([4 3]);
y = num2bin(q,x)

y =
0111
0010
0011
0000
```

```
0100
0111
0101
0110
0001
```

Algorithm Numeric values in the input data are quantized first by quantizer `q`, then converted to their binary equivalents. When `Hq` has coefficients exactly equal to 1, or when the input data set `x` includes values equal to 1 and 1 is outside the quantizer's range, 1 is quantized according to the property values set for `q` because no binary representation for 1 exists. Beware of this behavior when `q.overflowmode = 'wrap'`, because the value 1 in the input data or quantized filter coefficients gets converted and wrapped to –1 (1000 binary).

For example,

```
q = quantizer([4 3],'wrap');
range (q)

ans =
   -1.0000    0.8750

num2bin(q,1)
```

returns the binary $1000_2 = -1_{10}$ because 1 lies outside the maximum value (0.8750) for `q`.

Errors When one or more quantized coefficients have real or imaginary parts that equal 1, and the number format does not include 1 in its range, those coefficients are saturated to $1 - \varepsilon$ (where ε is the epsilon of the coefficient quantizer) and the operation returns a warning message.

See Also `bin2num`, `hex2num`, `num2hex`

Purpose Convert a number to its hexadecimal equivalent

Syntax

```
num2hex(Hq)
c = num2hex(Hq)
y = num2hex(q,x)
```

Description `num2hex(Hq)` with no left-hand-side argument displays the quantized coefficients in quantized filter `Hq` as hexadecimal strings.

`c = num2hex(Hq)` with a left-hand-side argument `c` returns a cell array of quantized coefficients as hexadecimal strings. Cell array `c` inherits the configuration of cell array `Hq.QuantizedCoefficients`.

When the mode of `Hq` is `'float'`, `'double'`, or `'single'`, the coefficients are converted to IEEE Standard 754 style hexadecimal strings.

If the mode of `Hq` is fixed, the coefficients are converted to two's complement hexadecimal strings.

`y = num2hex(q,x)` converts numeric array `x` into hexadecimal strings returned in `y`. When `x` is a cell array, each numeric element of `x` is converted to hexadecimal. If `x` is a structure, each numeric field of `x` is converted to hexadecimal.

For fixed-point quantizers, the representation is two's complement. For floating-point quantizers, the representation is IEEE Standard 754 style.

For example, for `q = quantizer('double')`

```
num2hex(q,nan)

ans =

fff8000000000000
```

The leading fraction bit is 1, all other fraction bits are 0. Sign bit is 1, exponent bits are all 1.

```
num2hex(q,inf)

ans =

7ff0000000000000
```

Sign bit is 0, exponent bits are all 1, all fraction bits are 0.

```
num2hex(q,-inf)

ans =

fff0000000000000
```

Sign bit is 1, exponent bits are all 1, all fraction bits are 0.

`num2hex` and `hex2num` are inverses of each other, except that `num2hex` returns the hexadecimal strings in a column.

Examples

This is a floating-point example using a quantizer `q` that has 6-bit word length and 3-bit exponent length.

```
x=magic(3)
x =
     8     1     6
     3     5     7
     4     9     2

q=quantizer('float',[6 3]);
y = num2hex(q,x)

y =

0
8
0
8
8
8
0
8
0
```

Algorithm

Call the `num2hex` method of the coefficient's quantizer. The numeric values are quantized first by `q`; if you have coefficients that are exactly equal to 1, and 1 is not representable in the arithmetic format, no binary representation for 1

will exist, and 1 is quantized according to `q`. Beware of this when `q.overflowmode = 'wrap'`, because 1 will be quantized to –1.

For example,

```
q = quantizer([4 3],'wrap');
num2hex(q, 1)
```

returns the hexadecimal $8_{16} = -1_{10}$.

Errors

If one or more quantized coefficients has a real or imaginary part that is exactly equal to 1, and 1 is outside the range for the quantizer, those coefficients are saturated to $1 - \varepsilon$ (where ε is the epsilon of the coefficient quantizer) and the operation returns a warning message.

See Also

`bin2num`, `hex2num`, `num2bin`

num2int

Purpose

Convert number to signed integer

Syntax

```
y = num2int(q,x)
y = num2int(hq)
y = num2int(q,c)
[y1,y2, ] = num2int(q,x1,x2, )
```

Description

`y = num2int(q,x)` uses `q.format` to convert numeric `x` to an integer.

`y = num2int(hq)` uses `q.coefficientformat` to convert the coefficients of quantized filter `hq` to integers. This function is equivalent to

```
y = num2int(hq.coefficientquantizer,hq.quantizedcoefficients)
```

`y = num2int(q,{c})` uses `q.format` to convert the entries in cell array `c` to integers, returned in cell array `y`.

`[y1,y2, ] = num2int(q,x1,x2, )` uses `q.format` to convert numeric values `x1`, `x2`, … to integers `y1`,`y2`,….

Examples

All of the four-bit, two's complement, fixed-point numbers in fractional form are given by

```
x = [0.875 0.375 -0.125 -0.625
     0.750 0.250 -0.250 -0.750
     0.625 0.125 -0.375 -0.875
     0.500 0 -0.500 -1.000];

q=quantizer([4 3]);

y = num2int(q,x)
```

```
y =

     7     3    -1    -5
     6     2    -2    -6
     5     1    -3    -7
     4     0    -4    -8
```

For a quantized filter hq

```
[b,a] = butter(3,.9,'high')

b =

    0.0029   -0.0087    0.0087   -0.0029
a =

    1.0000    2.3741    1.9294    0.5321

hq = sos(qfilt('referencecoefficients',{b,a}))
hq.format = [4 3]
Warning: 1 overflow in coefficients.
hq =
Quantized Direct form II transposed filter
------- Section 1 -------
Numerator
    QuantizedCoefficients{1}{1}   ReferenceCoefficients{1}{1}
  (1)                      0.750   0.741915184087109990
  (2)                     -0.750  -0.741922736650797670
Denominator
    QuantizedCoefficients{1}{2}   ReferenceCoefficients{1}{2}
+ (1)                      0.875  0.999969482421875000
  (2)                      0.750  0.726520355687005010
------- Section 2 -------
Numerator
    QuantizedCoefficients{2}{1}   ReferenceCoefficients{2}{1}
  (1)                      0.500   0.500000000000000000
  (2)                     -1.000  -0.999994910089555320
  (3)                      0.500   0.499994910141368990
Denominator
    QuantizedCoefficients{2}{2}   ReferenceCoefficients{2}{2}
  (1)                      0.500  0.500000000000000000
```

```
(2)                        0.875  0.823776107851993070
(3)                        0.375  0.366169458636399440

 FilterStructure = df2t
     ScaleValues = [0.00390625  1  1]
NumberOfSections = 2
StatesPerSection = [1  2]
 CoefficientFormat = quantizer('fixed', 'round', 'saturate', [4  3])
       InputFormat = quantizer('fixed', 'floor', 'saturate', [4  3])
      OutputFormat = quantizer('fixed', 'floor', 'saturate', [4  3])
MultiplicandFormat = quantizer('fixed', 'floor', 'saturate', [4  3])
     ProductFormat = quantizer('fixed', 'floor', 'saturate', [4  3])
         SumFormat = quantizer('fixed', 'floor', 'saturate', [4  3])
Warning: 1 overflow in coefficients.

num2int(hq)

hq.QuantizedCoefficients{1}{1} =

     6    -6

hq.QuantizedCoefficients{1}{2} =

     7     6

hq.QuantizedCoefficients{2}{1} =

     4    -8     4

hq.QuantizedCoefficients{2}{2} =

     4     7     3
```

Algorithm

When `q` is a fixed-point quantizer and f is equal to `fractionlength`(`q`), and x is numeric

$y=x*2^f$.

When `q` is a floating-point quantizer, $y = x$. `num2int` is meaningful only for fixed-point quantizers.

See Also

`bin2num`, `hex2num`, `num2bin`, `num2hex`

numberofsections

Purpose Return the number of sections in a quantized filter

Syntax `numberofsections(hq)`

Description `numberofsections(hq)` returns the number of sections in a quantized filter. The filter reference coefficients determine the number of sections.

Examples Create a double-precision filter using the Butterworth method. Convert the filter to a quantized filter in second-order section form, then use the function `numberofsections` to determine the number of sections that make up the filter.

```
[b,a] = butter(7,.5);
Hq = sos(qfilt('df2t',{b,a}));
numberofsections(Hq)
```

See Also `get`, `qfilt`, `set`, `sos`

Purpose Return the number of underflows from the most recent quantizer operation

Syntax

```
nunderflows(q)
```

Description `nunderflows(q)` is the number of underflows during a call to `quantize(q,...)` for quantizer object `q`. An underflow is defined as a number that is nonzero before it is quantized, and zero after it is quantized. The number of underflows accumulates over successive calls to `quantize`. Use the function `reset(q)` to return `nunderflows` to zero.

Examples

```
q = quantizer('fixed','floor',[4 3]);
      x = (0:eps(q)/4:2*eps(q))';
      y = quantize(q,x);
      nunderflows(q)

ans =

     3
```

By looking at `x` and `y`, you can see which ones went to zero.

```
[x,y]

ans =

         0         0
    0.0313         0
    0.0625         0
    0.0938         0
    0.1250    0.1250
    0.1563    0.1250
    0.1875    0.1250
    0.2188    0.1250
    0.2500    0.2500
```

See Also denormalmin, eps, quantize, quantizer, reset

Purpose

Optimize unity gains for a quantized filter

Syntax

`optimizeunitygains(hq)`

Description

`optimizeunitygains(hq)` returns the value of the `optimizeunitygains` property of quantized filter object `hq`. The value of the property can be one of these two strings:

- **`on`** — optimize for coefficients whose real or imaginary part is exactly equal to 1. Even if 1 cannot be represented by the number format specified by the `CoefficientFormat` property, skip multiplications by a real or imaginary part of a coefficient that is equal to 1.
- **`off`** — do not optimize for coefficients whose real or imaginary part is exactly equal to 1. If 1 cannot be represented by the number format specified by the `CoefficientFormat` property, then quantize real or imaginary parts of coefficients that are equal to 1 to the next lower quantization level.

When `optimizeunitygains` is **`on`**, `quantizer(hq,'coefficient')` returns a unitquantizer. If `optimizeunitygains` is **`off`**, `quantizer(hq,'coefficient')` returns a quantizer.

Examplse

```
Hq = qfilt;
optimizeunitygains(Hq)
```

returns the default '`off`'.

See Also

`qfilt`, `qfilt/get`, `quantizer`, `unitquantizer`

order

Purpose

Return the filter order of a quantized filter

Syntax

```
n=order(hq)
n=order(hq,k)
```

Description

`n = order(hq)` returns the order `n` of the quantized filter `hq`. When `hq` is a single-section filter, `n` is the number of delays required for a minimum realization of the filter.

When `hq` has more than one section, `n` is the number of delays required for a minimum realization of the overall filter.

`n=order(hq,k)` returns the order `n` of the `k`-th section of quantized filter `hq`.

Examples

Create a reference filter. Quantize the filter and convert to second-order section form. Then use `order` to check the filter order of the second section and the overall filter.

```
[b,a] = ellip(4,3,20,.6); % Create the reference filter.

% Quantize the filter and convert to second-order sections.
Hq = sos(qfilt('df2',{b,a},'roundmode','fix'))

n=order(Hq) % Check the order of the overall filter.
n = 4

n=order(Hq,2) % Check the order of the second section, k=2.
n = 2
```

qfft

Purpose

Construct a quantized FFT

Syntax

```
f = qfft
f = qfft('propertyname1',propertyvalue1, ...)
f = qfft(a)
f = qfft(pn,pv)
f = qfft('quantize ,[14 13])
```

Description

`f = qfft` creates a quantized FFT with default property values.

`f = qfft('propertyname1',propertyvalue1,...)` uses property name/property value pairs to set the properties of the quantized FFT.

`f = qfft(a)`, where `a` is a structure whose field names are quantized FFT property names, sets the properties named in each field name to the values contained in the structure.

`f = qfft(pn,pv)` sets the quantized FFT properties specified in the cell array of strings `pn` to the corresponding property values in cell array `pv`.

`f = qfft('quantize ,[14 13])` sets all data format properties for the quantized FFT to the same word length and fraction length.

Refer to "A Quick Guide to Quantized FFT Properties" on page 12-51 for a list of quantized FFT properties.

Examples

Create a quantized FFT `f` and apply it to a data set. Plot the result.

```
warning on
n=128;
t = (1:n)/n;
x = sin(2*pi*10*t)/10;
f = qfft('length',n);
plot(t,abs([fft(f,x);fft(x)]))
```

See Also fft, get, ifft, qreport, set

Purpose Construct a quantized filter

Syntax

```
Hq = qfilt
Hq = qfilt('Structure',{Coef})
Hq = qfilt('prop1',value1,'prop2',value2,...)
Hq = qfilt('Structure',{Coef},'prop1',value1,'prop2',value2,...)
Hq = qfilt( quantizer ,[13, 14])
```

Description

`Hq = qfilt` creates a quantized filter `Hq` with default property settings. The default settings for `Hq` imply `Hq` is a fixed-point quantized filter with a transposed direct form II filter structure. All of the filter properties, along with their default values are listed in "Quantized Filter Properties Reference" on page 12-11.

`Hq = qfilt('`*`Structure`*`',{Coef})` creates a quantized filter `Hq` with all properties set to default values, except that the filter structure is specified by the string `'`*`Structure`*`'`, and the reference filter parameters (the `ReferenceCoefficients` property values) are specified in the cell array `{Coef}`. The syntax for entering reference coefficients is specified in "Specifying the Filter Reference Coefficients" on page 8-7. `'`*`Structure`*`'` can be one of the strings for the `FilterStructure` property values listed in the following table.

Table 13-4: Filter Structure Properties

Property Value String	Description
`'df1'`	Direct form I
`'df1t'`	Direct form I transposed
`'df2'`	Direct form II
`'df2t'`	Direct form II transposed
`'fir'`	Finite impulse response (FIR)
`'firt`	Finite impulse response transposed
`'antisymmetricfir`	Direct form antisymmetric FIR, available odd or even

Table 13-4: Filter Structure Properties (Continued)

Property Value String	Description
'symmetricfir'	Direct form symmetric FIR, available odd or even
'latticear'	Lattice autoregressive (AR)
'latticema'	Lattice moving average (MA)
'latticearma'	Lattice ARMA
'latticeca'	Lattice coupled-allpass
'latticecapc'	Lattice coupled-allpass power complementary
'statespace'	Single-input, single-output state-space

`Hq = qfilt('prop1',value1,'prop2',value2,...)` creates a quantized filter `Hq` with all properties set to the default values, except for those you specify with the input string arguments `'prop1'`, `'prop2'`,`...`, along with the corresponding values in `value1`, `value2`,`...`. Filter properties you can set, with their default values, are listed in "Quantized Filter Properties Reference" on page 12-11. Any properties that you do not explicitly set when you create the quantized filter are assigned default values.

You can also use the shortcut

```
Hq = qfilt('Structure',{Coef},'prop1',value1,'prop2',value2,...)
```

by first specifying the `FilterStructure` property value as `'Structure'` and the reference filter parameters (the `ReferenceCoefficients` property values) in the cell array `{Coef}`.

`Hq = qfilt('quantizer',[13 14]` sets all the data format properties for quantized filter `Hq` to the same word length and fraction length.

Examples

Example 1: Quantized Filter with Two Second-Order Sections

From a reference filter, create a fixed-point quantized filter `Hq` that has two second-order sections, setting the rounding mode to `'fix'` and displaying the results.

```
% Create the reference filter transfer function.
```

```
[b,a] = ellip(4,3,20,.6);

% Create a quantized filter with 2 second-order sections
% and display the results.
hq = sos(qfilt('df2',{b,a},'roundmode','fix'))

hq =
Quantized Direct form II transposed filter
------- Section 1 -------
Numerator
     QuantizedCoefficients{1}{1}    ReferenceCoefficients{1}{1}
  (1)          0.551605224609375  0.551616219027048720

  (2)          0.776458740234375  0.776489000631472080
  (3)          0.551605224609375  0.551616219027047940
Denominator
     QuantizedCoefficients{1}{2}    ReferenceCoefficients{1}{2}
  (1)          0.999969482421875   0.999969482421875000
  (2)         -0.054809570312500  -0.054810658312267876
  (3)          0.473083496093750   0.473108096805785360
------- Section 2 -------
Numerator
     QuantizedCoefficients{2}{1}    ReferenceCoefficients{2}{1}
  (1)          0.499969482421875  0.499984741210937500
  (2)          0.359802246093750  0.359832079066733920
  (3)          0.499969482421875  0.499984741210938170
Denominator
     QuantizedCoefficients{2}{2}    ReferenceCoefficients{2}{2}
  (1)          0.999969482421875  0.999969482421875000
  (2)          0.588378906250000  0.588389482549356520
  (3)          0.957336425781250  0.957363508666007170


   FilterStructure = df2t
       ScaleValues = [0.5  2  1]
  NumberOfSections = 2
  StatesPerSection = [2  2]
 CoefficientFormat = quantizer('fixed', 'fix', 'saturate', [16  15])
       InputFormat = quantizer('fixed', 'fix', 'saturate', [16  15])
      OutputFormat = quantizer('fixed', 'fix', 'saturate', [16  15])
MultiplicandFormat = quantizer('fixed', 'fix', 'saturate', [16  15])
     ProductFormat = quantizer('fixed', 'fix', 'saturate', [32  30])
         SumFormat = quantizer('fixed', 'fix', 'saturate', [32  30])
```

Example 2: Quantized Filter from Table of Filter Coefficients

In this example, you create a sixth-order quantized filter from filter coefficients in a reference table.

Enter the filter coefficients from a table of coefficients. The following coefficients represent a 6-pole Chebyshev high pass filter, with 0.5% ripple in the passband and cutoff at 0.25 in normalized frequency.

Numerator:

```
b=[.0143445 -0.08606701 .2151675 -.28689 -.2151675 0.08606701 0.0143445]
```

Denominator:

```
a=[1.0 1.076051 1.662847 1.191062 0.7403085 0.2752156 0.0572225]
```

Create a quantized filter using the reference coefficients `b` and `a`.

```
hq = qfilt('ref',{b,a})

hq =

Quantized Direct form II transposed filter
Numerator
     QuantizedCoefficients{1}    ReferenceCoefficients{1}
  (1)       0.014343261718750   0.014344500000000000
  (2)      -0.086059570312500  -0.086067009999999999
  (3)       0.215179443359375   0.215167500000000010
  (4)      -0.286895751953125  -0.286889999999999980
  (5)       0.215179443359375   0.215167500000000010
  (6)      -0.086059570312500  -0.086067009999999999
  (7)       0.014343261718750   0.014344500000000000
Denominator
     QuantizedCoefficients{2}    ReferenceCoefficients{2}
+ (1)       0.999969482421875  1.000000000000000000
+ (2)       0.999969482421875  1.076051000000000100
+ (3)       0.999969482421875  1.662847000000000000
+ (4)       0.999969482421875  1.191062000000000100
  (5)       0.740295410156250  0.740308500000000040
  (6)       0.275207519531250  0.275215600000000000
  (7)       0.057220458984375  0.057222500000000003

   FilterStructure = df2t
       ScaleValues = [1]
  NumberOfSections = 1
  StatesPerSection = [6]
 CoefficientFormat = quantizer('fixed', 'round', 'saturate', [16  15])
       InputFormat = quantizer('fixed', 'floor', 'saturate', [16  15])
      OutputFormat = quantizer('fixed', 'floor', 'saturate', [16  15])
```

```
MultiplicandFormat = quantizer('fixed', 'floor', 'saturate', [16  15])
     ProductFormat = quantizer('fixed', 'floor', 'saturate', [32  30])
         SumFormat = quantizer('fixed', 'floor', 'saturate', [32  30])
Warning: 4 overflows in coefficients.
```

Eliminate the overflows by normalizing the coefficients.

```
hq2 = normalize(hq)
```

You have a sixth-order, high pass filter with no overflowing coefficients.

Some things to think about when you use coefficients from a table.

- Take care to assign the numerator and denominator values correctly. In your table, know which coefficients are for the numerator, which for the denominator.
- Verify that the sign of the denominator coefficients is correct for MATLAB.
- Note whether all coefficients are provided. Some tables omit the first coefficient for the denominator. If omitted, set the first denominator coefficient equal to 1.0.

Example 3: Comparing Fixed-Point and Floating-Point Filters

To demonstrate the effect of filtering a signal with a quantized filter that has a leading zero in the denominator coefficients, this example creates a default quantized filter, then changes the reference coefficients to be numerator=1 and denominator=0.

```
q=qfilt

q =
Quantized Direct form II transposed filter
Numerator
     QuantizedCoefficients{1}    ReferenceCoefficients{1}
+ (1)       0.999969482421875  1.000000000000000000
Denominator
     QuantizedCoefficients{2}    ReferenceCoefficients{2}
+ (1)       0.999969482421875  1.000000000000000000

   FilterStructure = df2t
       ScaleValues = [1]
  NumberOfSections = 1
  StatesPerSection = [0]
 CoefficientFormat = quantizer('fixed', 'round', 'saturate', [16  15])
       InputFormat = quantizer('fixed', 'floor', 'saturate', [16  15])
      OutputFormat = quantizer('fixed', 'floor', 'saturate', [16  15])
```

```
MultiplicandFormat = quantizer('fixed', 'floor', 'saturate', [16  15])
     ProductFormat = quantizer('fixed', 'floor', 'saturate', [32  30])
         SumFormat = quantizer('fixed', 'floor', 'saturate', [32  30])
Warning: 2 overflows in coefficients.

q.ref={1 0};
Warning: 1 overflow in coefficients.
```

q is a fixed-point quantized filter with references coefficients of b=1 and a=0. Now filter a signal with q and look at the results.

```
filter(q,rand(1,2))
Warning: 3 overflows in QFILT/FILTER.

                    Max          Min    NOverflows   NUnderflows   NOperations
 Coefficient          1            0             1             0             2
       Input     0.9501       0.2311             0             0             2
      Output          1            1             0             0             2
Multiplicand          2       0.2311             2             0             8
     Product        Inf       0.2311             2             0             8
         Sum     0.9501       0.2311             0             0             2

ans =

    1.0000    1.0000
```

In two's complement fixed-point format, NaNs and Infs cannot be represented. When the division by zero occurs during the filtering process, which happens when the leading coefficient in the denominator is zero, the result saturates to $(1\text{-}2^{15})$. The direct form filter structures, such as df1 and df2t, demonstrate this behavior when they have leading zeros in the denominator.

When you change the filter mode to 'float' from 'fixed', the results return as Inf, as you should expect.

```
q.mode='float';
filter(q,rand(1,2))

Warning: 6 overflows in QFILT/FILTER.

                    Max          Min    NOverflows   NUnderflows   NOperations

 Coefficient          1            0             0             0             2

       Input     0.6068        0.486             0             0             2

      Output        Inf    1.798e+308            2             0             2

Multiplicand        Inf         0.25             2             0             8

     Product        Inf         0.25             4             0             8

         Sum        0.5         0.25             0             0             2
```

```
ans =

    Inf    Inf
```

Changing the mode results in `Inf` because IEEE floating-point arithmetic returns `Inf` as the result of a division by zero operation.

See Also

`get`, `set`, `setbits`

qfilt2tf

Purpose Convert quantized filters to transfer function form

Syntax

```
[Bq,Aq,Br,Ar] = qfilt2tf(Hq)
[Cq,Cr] = qfilt2tf(Hq,'sections')
```

Description

`[Bq,Aq,Br,Ar] = qfilt2tf(Hq)` converts the quantized filter coefficients from quantized filter `Hq` into transfer function form with numerator `Bq` and denominator `Aq`, and the reference coefficients into transfer-function form with numerator `Br` and denominator `Ar`. When quantized filter `Hq` has more than one section, all the numerator polynomials are are convolved into the numerator polynomial of a single transfer function. Similarly, the denominator polynomials are convolved into a denominator polynomial of a single transfer function.

`[Cq,Cr] = qfilt2tf(Hq,'sections')` returns one cell array per section, where `Cq` is the transfer function form of the quantized coefficients and `Cr` is the transfer function form of the reference coefficients.

```
Cq = {{Bq1,Aq1},{Bq2,Aq2},...}
Cr = {{Br1,Ar1},{Br2,Ar2},...}
```

Examples

To demonstrate the conversion, use `butter` to a create a reference filter in statespace form. Make a statespace quantized filter from the reference filter and convert the quantized filter to transfer function form.

```
[A,B,C,D]=butter(3,.2);
Hq=qfilt('statespace',{A,B,C,D},'mode','double');
[bq,aq]=qfilt2tf(Hq)
bq =

    0.0181    0.0543    0.0543    0.0181

aq =

    1.0000   -1.7600    1.1829   -0.2781
```

See Also `qfilt`

Purpose

Display the results of applying a quantizer, quantized FFT or quantized filter to data

Syntax

```
qreport(obj)
s = qreport (obj)
```

where `obj` is one of the following objects:

- Quantizer
- Quantized filter
- Quantized FFT

Description

`qreport(obj)` displays the minimum (`Min`), maximum (`Max`), number of overflows (`NOver`), and underflows (`NUnder`) of the most recent application of `obj` to a data set, where `obj` is a quantized filter or a quantized FFT. Each section of quantized filter `Hq` or stage of quantized FFT `F` is represented by one line of information in the report.

Setting `warning` to `ON` displays this report when a quantized filter or quantized FFT overflows.

`s = qreport(obj)` returns a MATLAB structure containing the information.

Also, `qreport(s)` displays the report for the structure `s`.

Examples

Display the results of filtering a data set with a quantized filter `Hq`.

```
[b,a] = butter(6,.5);
Hq = sos(qfilt('ReferenceCoefficients',{b,a}));
Y = filter(Hq,rand(50,1));
qreport(Hq)
```

```
                    Max          Min        NOverflows NUnderflows NOperations
 Coefficient           1    -5.169e-016          0           1           6
                       1     -1.11e-016          0           1           6
                       1    -8.326e-017          0           1           6
       Input      0.9501       0.009861          0           0          50
      Output      0.9555        0.02808          0           0          50
Multiplicand      0.9501      0.0006161          0           0         400
                   0.394        0.02808          0           0         350
                  0.9556        0.02808          0           0         350
     Product       0.394      -0.001708          0           0         400
                  0.6424       -0.05511          0           0         350
                  0.9556        -0.5626          0           0         350
```

```
        Sum       0.09852    -0.0007188          0          0          250
                   0.3212     -0.003827          0          0          250
                   0.9555       -0.2523          0          0          250
```

Display the results of running `qfft F` on a set of random data.

```
F = qfft('length',64,'scale',1/64);
Y = fft(F,rand(64,1));
qreport(F)
```

```
                   Max           Min       NOverflows    NUnderflows NOperations
 Coefficient           1            -1            6            5          126
       Input      0.9883       0.01176            0            0           64
      Output      0.5364      -0.06312            0            0          128
Multiplicand      0.9883      -0.03622            0            0         1536
     Product      0.2902      -0.02877            0            0          768
         Sum      0.5364      -0.06312            0            0         1920
```

See Also

`disp`, `get`

Purpose

Apply a quantizer to data

Syntax

```
y = quantize(q, x)
[y1,y2,...] = quantize(q,x1,x2,...)
```

Description

`y = quantize(q, x)` uses the quantizer `q` to quantize `x`. When `x` is a numeric array, each element of `x` is quantized. When `x` is a cell array, each numeric element of the cell array is quantized. When `x` is a structure, each numeric field of `x` is quantized. Nonnumeric elements or fields of `x` are left unchanged and `quantize` does not issue warnings for nonnumeric values.

`[y1,y2,...] = quantize(q,x1,x2,...)` is equivalent to `y1 = quantize(q,x1), y2 = quantize(q,x2),...`

The quantizer states

`'max'`	-	Maximum value before quantizing
`'min'`	-	Minimum value before quantizing
`'noverflows'`	-	Number of overflows
`'nunderflows'`	-	Number of underflows
`'noperations'`	-	Number of quantization operations

are updated during the call to `quantize`, and running totals are kept until a call to `reset` is made.

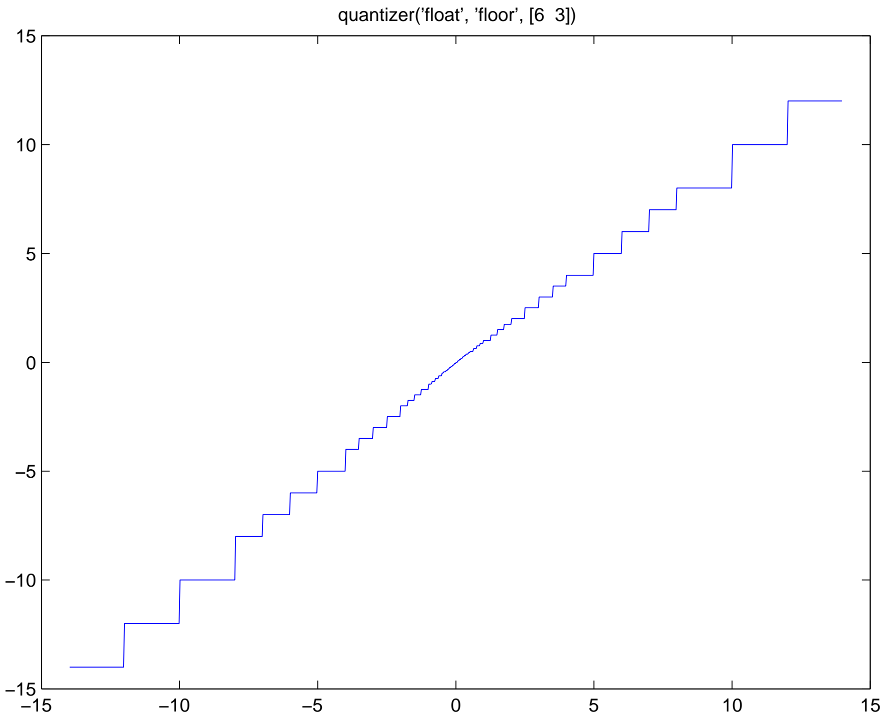
Examples

The following examples demonstrate using `quantize` to quantize data.

Example 1 - Custom Precision Floating-Point

The code listed here produces the plot shown in the following figure.

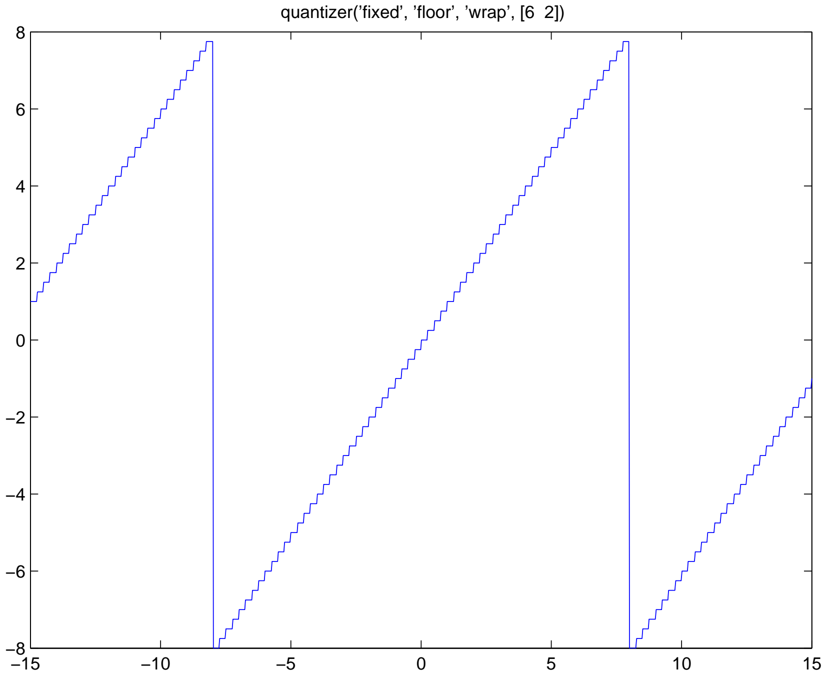
```
u=linspace(-15,15,1000);
q=quantizer([6 3],'float');
range(q)
 ans =
   -14    14
y=quantize(q,u);
plot(u,y);title(tostring(q))
```



Example 2 - Fixed-Point

The code listed here produces the plot shown in the following figure.

```
u=linspace(-15,15,1000);
q=quantizer([6 2],'wrap');
range(q)
 ans =
   -8.0000    7.7500
y=quantize(q,u);
plot(u,y);title(tostring(q))
```



See Also

`quantizer`, `set`

quantizer

Purpose

Construct a quantizer

Syntax

```
q = quantizer
q = quantizer('PropertyName1',PropertyValue1, ... )
q = quantizer(PropertyValue1, PropertyValue2, ... )
q = quantizer(a)
q = quantizer(pn,pv)
[qcoefficient,qinput,qoutput,qmultiplicand,qproduct,...
   qsum] = quantizer(F)
[q1, q2, ...] = quantizer(F, format1, format2, ...)
```

Description

`q = quantizer` creates a quantizer with properties set to their default values.

`q = quantizer('PropertyName1',PropertyValue1,...)` uses property name/ property value pairs.

`q = quantizer(PropertyValue1,PropertyValue2,...)` creates a quantizer with the listed property values. When two values conflict, `quantizer` sets the last property value in the list. Property values are unique; you can set the property names by specifying just the property values in the command.

`q = quantizer(a)` where `a` is a structure whose field names are property names, sets the properties named in each field name with the values contained in the structure.

`q = quantizer(pn,pv)` sets the named properties specified in the cell array of strings `pn` to the corresponding values in the cell array `pv`.

These are the quantizer property values, sorted by associated property name:

Property Name	Property Value	Description
`Mode`	`'double'`	Double-precision mode. Override all other parameters.
	`'float'`	Custom-precision floating-point mode.
	`'fixed'`	Signed fixed-point mode.

Property Name (Continued)	Property Value	Description
	'single'	Single-precision mode. Override all other parameters.
	'ufixed'	Unsigned fixed-point mode.
Roundmode	'ceil'	Round towards negative infinity.
	'convergent'	Convergent rounding.
	'fix'	Round towards zero.
	'floor'	Round towards positive infinity.
	'round'	Round towards nearest.
Overflowmode (fixed-point only)	'saturate'	Saturate at max value on overflow.
	'wrap'	Wrap on overflow.
Format	[wordlength exponentlength]	The format for fixed or ufixed mode.
	[wordlength exponentlength]	The format for float mode.

The default property values for a quantizer are

```
mode = 'fixed';
roundmode = 'floor';
overflowmode = 'saturate';
format = [16 15];
```

Along with the preceding properties, quantizers have read-only properties: 'max', 'min', 'noverflows', 'nunderflows', and 'noperations'. They can be accessed through quantizer/get or q.max, q.min, q.noverflows, q.nunderflows, and q.noperations, but they cannot be set. They are updated during the quantizer/quantize method, and are reset by the quantizer/reset method.

The following table lists the read-only quantizer properties:

Property Name	Description
'max'	Maximum value before quantizing
'min'	Minimum value before quantizing
'noverflows'	Number of overflows
'nunderflows'	Number of underflows.
'noperations'	Number of data points quantized

[qcoefficient,qinput,qoutput,qmultilplicand,qproduct,qsum] = quantizer(F) returns property values associated with the quantized FFT F for the twiddle factors, input, output, product, and sum quantizers.

[q1, q2,...] = quantizer(F, formatName1, formatName2,...) returns quantizers q1, q2,..., associated with formatName1,formatName2,..., where format k is a string that can be one of 'twiddle', 'input', 'output', 'multiplicand', 'product', or 'sum'.

Examples

The following example operations are equivalent.

Setting quantizer properties by listing property values only in the command.

```
q = quantizer('fixed', 'ceil', 'saturate', [5 4])
```

Using a structure a to set quantizer properties.

```
a.mode = 'fixed';
a.roundmode = 'ceil';
a.overflowmode = 'saturate';
a.format = [5 4];
q = quantizer(a);
```

Using property name and property value cell arrays pn and pv to set quantizer properties.

```
pn = {'mode',  'roundmode', 'overflowmode', 'format'};
pv = {'fixed', 'ceil', 'saturate', [5 4]};
q = quantizer(pn, pv)
```

Using property name/property value pairs to configure a quantizer.

```
q = quantizer( 'mode', fixed','roundmode','ceil',...
'overflowmode', 'saturate', 'format', [5 4]);
```

See Also

quantize, set

radix

Purpose Return the radix of a quantized FFT

Syntax

```
radix(f)
```

Description `radix(f)` returns the radix of quantized FFT `f`.

Examples After you create a default quantized FFT, the radix function returns 2 as the value of the radix, as shown in this example.

```
F = qfft;
radix(F)
```

returns the default 2.

See Also `qfft`, `qfft/get`, `qfft/set`

Purpose Generate a uniformly distributed, quantized random number

Syntax

```
randquant(q,n)
randquant(q,m,n)
randquant(q,m,n,p,...)
randquant(q,[m,n])
randquant(q,[m,n,p,...])
```

Description

`randquant(q,n)` uses quantizer `q` to generate An `n` by `n` matrix with random entries whose values cover the range of `q` when `q` is a fixed-point quantizer. When `q` is a floating-point quantizer, `randquant` populates the `n` by `n` array with values covering the range -[square root of `realmax(q)`] to [square root of `realmax(q)`].

`randquant(q,m,n)` uses quantizer `q` to generate an `m` by `n` matrix with random entries whose values cover the range of `q` when `q` is a fixed-point quantizer. When `q` is a floating-point quantizer, `randquant` populates the `m` by `n` array with values covering the range -[square root of `realmax(q)`] to [square root of `realmax(q)`].

`randquant(q,m,n,p,...)` uses quantizer q to generate an `m` by `n` by `p` by … matrix with random entries whose values cover the range of q when q is fixed-point quantizer. When q is a floating-point quantizer, randquant populates the matrix with values covering the range -[square root of `realmax(q)`] to [square root of `realmax(q)`].

`randquant(q,[m,n])` uses quantizer `q` to generate an `m` by `n` matrix with random entries whose values cover the range of `q` when `q` is a fixed-point quantizer. When `q` is a floating-point quantizer, `randquant` populates the `m` by `n` array with values covering the range -[square root of `realmax(q)`] to [square root of `realmax(q)`].

`randquant(q,[m,n,p,...])` uses quantizer `q` to generate `p` `m` by `n` matrices containing random entries whose values cover the range of `q` when `q` is a fixed-point quantizer. When `q` is a floating-point quantizer, `randquant` populates the `m` by `n` arrays with values covering the range -[square root of `realmax(q)`] to [square root of `realmax(q)`].

`randquant` produces pseudorandom numbers. The number sequence `randquant` generates during each call is determined by the state of the generator. Since MATLAB resets the random number generator state at start-up, the sequence of random numbers generated by the function remains the same unless you change the state.

`randquant` works like `rand` in most respects, including the generator used, but it does not support the '`state`' and '`seed`' options available in `rand`.

Examples

```
q=quantizer([4 3]);
rand('state',0)
randquant(q,3)

ans =
    0.7500   -0.1250   -0.2500
   -0.6250    0.6250   -1.0000
    0.1250    0.3750    0.5000
```

See Also

`quantizer`, `quantizer/range`, `quantizer/realmax`, `rand`

Purpose

Return the numerical range of quantizers in a quantized FFT, or the range of a quantizer

Syntax

```
range(F)
rtwiddle = range(F)
[rtwiddle, rinput, routput, rproduct, rsum] = range(F)
[r1, r2, ...] = range(F, formattype1, formattype2, ...)
r = range(q)
[a, b] = range(q)
```

Description

`range(F)` displays the numerical range of all the quantizers in quantized FFT `F`.

`rtwiddle = range(F)` returns the numerical range of the twiddle factor quantizer (although twiddle factors always have magnitudes less than 1).

`[rtwiddle, rinput, routput, rproduct, rsum] = range(F)` returns the range of each of the quantizers.

`[r1, r2,...] = range(F, formattype1, formattype2,...)` returns the range of the quantizers specified by strings `formattype` *i*, which may take on the values `'twiddle'`, `'input'`, `'output'`, `'product'`, `'sum'`.

`r = range(q)` returns the two-element row vector r = $[a\ b]$ such that for all real x, `y = quantize(q,x)` returns y in the range $a \le y \le b$.

`[a, b] = range(q)` returns the minimum and maximum values of the range in separate output variables.

Examples

```
q = quantizer('float',[6 3]);
r = range(q)
```

returns `r = [ 14, 14]`.

```
q = quantizer('fixed',[4 2],'floor');
[a,b] = range(q)
```

returns `a =  2, b = 1.75 = 2  eps(q)`.

Algorithm

If `q` is a floating-point quantizer, a = `-realmax`(q), b = `realmax`(q).

If `q` is a signed fixed-point quantizer (`mode = 'fixed'`),

$$a = -\mathrm{realmax}(q) - \mathrm{eps}(q) = \frac{-2^{w-1}}{2^f}$$

$$b = \mathrm{realmax}(q) = \frac{2^{w-1} - 1}{2^f}$$

If `q` is an unsigned fixed-point quantizer (`mode = 'ufixed'`),

$a = 0$

$$b = \mathrm{realmax}(q) = \frac{2^w - 1}{2^f}$$

See `realmax` for more information.

Errors

If you use more than two output arguments, MATLAB returns the error message `Too many output arguments` and aborts the function.

See Also

`realmax`, `realmin`, `exponentmin`, `fractionlength`

Purpose

Directly realize a Simulink subsystem block for a direct-form quantized filter

Syntax

```
realizemdl(hq)
realizemdl(hq, propertyname1, propertyvalue1,...)
```

Description

`realizemdl(hq)` generates a model of filter `hq` in a Simulink subsystem block using sum, gain, and delay blocks from the Fixed-Point Blockset. The properties and values of `hq` define the resulting subsystem block parameters.

`realizemdl` requires either the DSP Blockset or the Fixed-Point Blockset. To accurately realize models of quantized filters, use the Fixed-Point Blockset.

`realizemdl(hq,propertyname1,propertyvalue1,...)` generates the model or `hq` with the associated `propertyname/propertyvalue` pairs, and any other values you set in `hq`.

Using the optional propertyname/propertyvalue pairs lets you control more fully the way the block subsystem model gets built, such as where the block goes, what the name is, or how to optimize the block structure. Valid properties and values for `realizemdl` are listed in this table, with the default value noted and descriptions of what the properties do.

Property Name	Property Values	Description
`BlockType`	`'fixed-point blocks'` (default) or `'floating-point blocks'`	Specifies the type of blocks to use to realize the model. `fixed-point blocks` is the default. Models that use `floating-point blocks` may not match the quantized filter `hq` exactly.
`Destination`	`'current'` (default) or `'new'`	Specify whether to add the block to your current Simulink model or create a new model to contain the block.
`Blockname`	`'filter'` (default)	Provides the name for the new subsystem block. By default the block is named '`filter`'. To enter a name for the block, use the `propertyvalue` set to a string '*`blockname`*'.

Property Name	Property Values	Description
`OverwriteBlock`	`'off'` or `'on'`	Specify whether to overwrite an existing block with the same name or create a new block.
`OptimizeZeros`	`'off'` (default) or `'on'`	Specify whether to remove zero-gain blocks.
`OptimizeOnes`	`'off'` (default) or `'on'`	Specify whether to replace unity-gain blocks with direct connections.
`OptimizeNegOnes`	`'off'` (default) or `'on'`	Specify whether to replace negative unity-gain blocks with a sign change at the nearest sum block.
`OptimizeDelayChains`	`'off'` (default) or `'on'`	Specify whether to replace cascaded chains of delay blocks with a single integer delay block to provide an equivalent delay.

Examples

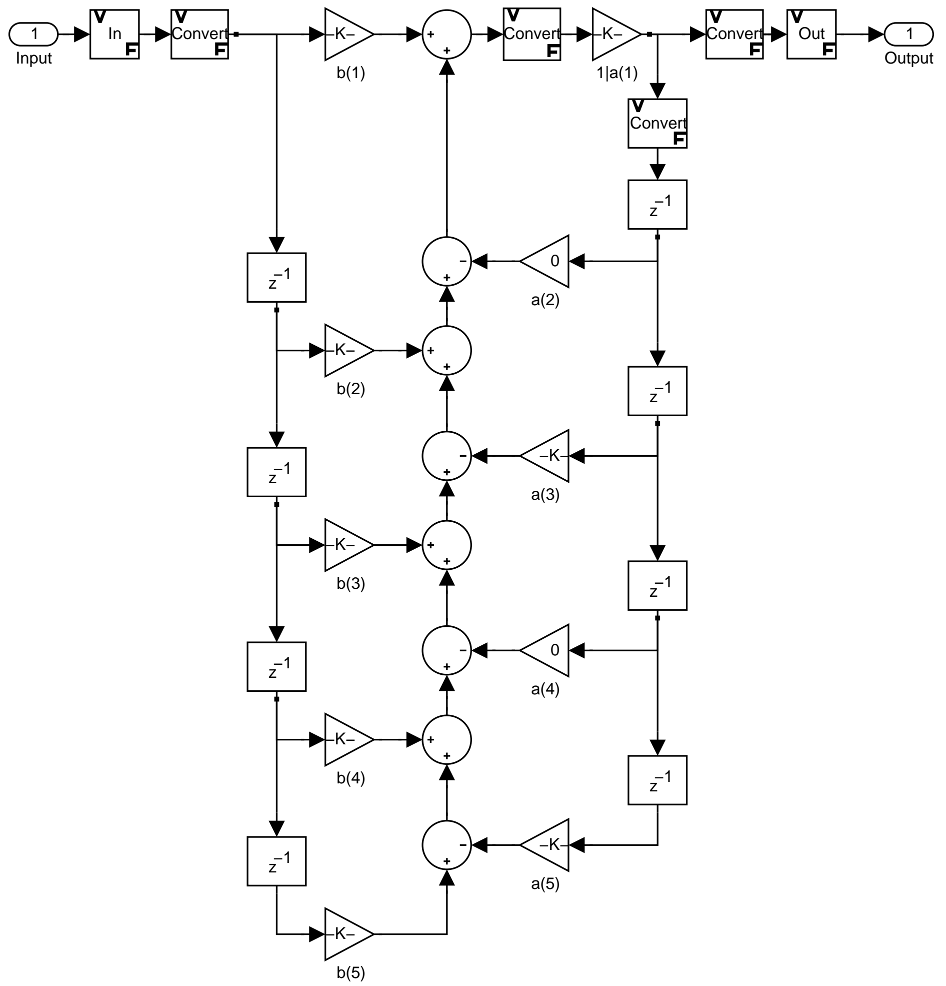
To demonstrate how `realizemdl` works to create models, these two examples show the default and optional syntaxes in use. Both examples begin from a quantized filter designed by `butter` in the Signal Processing Toolbox.

```
[b,a] = butter(4,.5);
hq = qfilt('df1',{b,a});
```

Example 1—Using the default syntax to realize a model of your quantized filter `hq`. When you use this syntax, `realizemdl` uses blocks from the Fixed-Point Blockset to realize the subsystem in your current Simulink model.

```
realizemdl(hq);
```

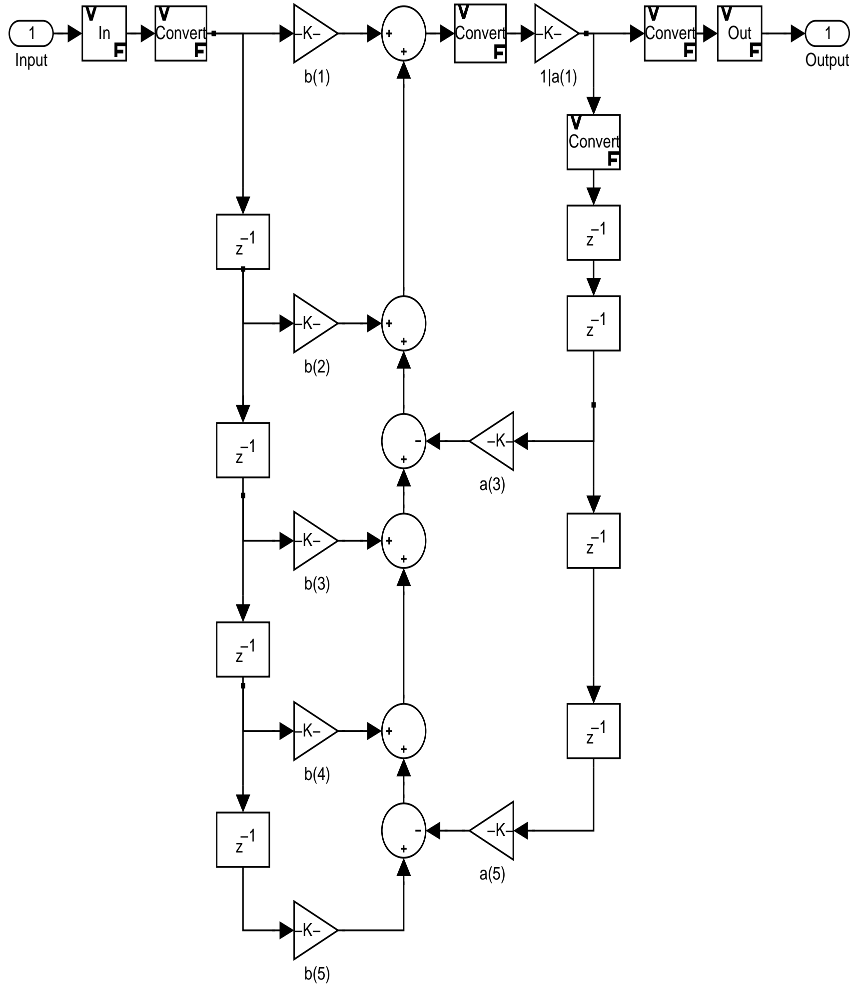
Look at the figure to see the model as realized by `realizemdl`.



Example 2—Using propertyname/propertyvalue pairs to specify the features of the subsystem block model created by `realizemdl`.

```
realizemdl(hq, 'blocktype', 'fixed-point blocks',...
'optimizezeros', 'on','blockname','newfiltermodel');
```

Since this example uses the optional property name `optimizezeros`, set to `'on'`, the resulting block subsystem is slightly different—the zero-gain blocks for coefficients `a(2)` and `a(4)` are not included in the subsystem.



See Also

`realizemdl` under the methods for `dfilt` in the Signal Processing Toolbox

realmax

Purpose

Return the largest positive quantized number

Syntax

```
x = realmax(q)
```

Description

`x = realmax(q)` is the largest quantized number representable where `q` is a quantizer. Anything larger overflows.

Examples

```
q = quantizer('float',[6 3]);
x = realmax(q)
```

returns `x` = 14.

Algorithm

If `q` is a floating-point quantizer, the largest positive number, x, is

$$x = 2^{E_{max}} \cdot (2 - eps(q))$$

If `q` is a signed fixed-point quantizer, the largest positive number, x, is

$$x = \frac{2^{w-1} - 1}{2^f}$$

If `q` is an unsigned fixed-point quantizer (`mode = 'ufixed'`), the largest positive number, x, is

$$x = \frac{2^w - 1}{2^f}$$

See Also

`quantizer`, `realmin`, `exponentmin`, `fractionlength`

realmin

Purpose Return the smallest positive normal quantized number

Syntax

```
x = realmin(q)
```

Description `x = realmin(q)` is the smallest positive normal quantized number where `q` is a quantizer. Anything smaller than `x` underflows or is an IEEE "denormal" number.

Examples

```
q = quantizer('float',[6 3]);
realmin(q)
```

returns the value 0.25.

Algorithm If `q` is a floating-point quantizer, $x = 2^{E_{min}}$ where E_{min} = `exponentmin`(q) is the minimum exponent.

If `q` is a signed or unsigned fixed-point quantizer, $x = 2^{-f} = \varepsilon$ where f is the fraction length.

See Also `exponentmin`, `fractionlength`

Purpose

Reset one or more quantizers, quantized filters, or quantized FFTs to their initial conditions

Syntax

```
reset(q)
reset(q1, q2, ...)
reset(hq)
reset(hq1,hq2,...)
reset(f)
reset(f1,f2,...)
```

Description

`reset(q)` resets quantizer `q` to its initial conditions. Works for quantized filters and quantized FFTs as well by replacing the quantizer with a quantized filter or quantized FFT in the command syntax.

`reset(q1, q2,...)` resets the states of the quantizers `q1`, `q2`,`...`. to the states they were in when you created them — their initial conditions.

The states of a quantizer are

```
'max'          - Maximum value before quantizing.
'min'          - Minimum value before quantizing.
'noverflows'   - Number of overflows.
'nunderflows'  - Number of underflows.
'noperations'  - Number of quantization operations performed.
```

`reset(hq1,hq2,...)` resets the states of quantized filters `hq1`, `hq2`,`...` to the states you set when you created them — their initial conditions.

The states of a quantized filter are

```
   'FilterStructure' - Structure of the filter
       'ScaleValues' - Scale values between filter sections
  'NumberOfSections' - Number of filter sections
  'StatesPerSection' - Number of states in each filter section
 'CoefficientFormat' - quantizer
       'InputFormat' - quantizer
      'OutputFormat' - quantizer
'MultiplicandFormat  - quantizer
     'ProductFormat' - quantizer
         'SumFormat' - quantizer
```

`reset(f1,f2,...)` resets the states of quantized FFTs `f1`, `f2`,`...` to the states you set when you created them — their initial conditions.

The states of a quantized FFT are

```
              'Radix' - Either 2 or 4
             'Length' - Scalar integer, length of the FFT
  'CoefficientFormat' - quantizer
        'InputFormat' - quantizer
       'OutputFormat' - quantizer
 'MultiplicandFormat' - quantizer
      'ProductFormat' - quantizer
          'SumFormat' - quantizer
   'NumberOfSections' - 4
        'ScaleValues' - Vector of the scale values between FFT
                        sections
```

Examples

```
q1 = quantizer('fixed','ceil','saturate',[4 3])
q2 = quantizer('double')
y1 = quantize(q1, -1.2:.1:1.2 )
y2 = quantize(q2, -1.2:.1:1.2 )
q1, q2
reset(q1, q2)
q1, q2
```

See Also

`quantizer`, `set`

Purpose Return the scalevalues property of a quantized filter

Syntax

```
s = scalevalues(hq)
```

Description

s = scalevalues(hq) returns the scale values of the quantized filter hq. The scale values for the filter scale the input to each filter section. The value of the scalevalues property must be a scalar, or a vector of length numberofsections(hq). For efficient computation, set the scale values to be powers of 2.

If s is a scalar, the input to the first section of the quantized filter is scaled by s. When s is a vector, the input to the k-th section of the filter is scaled by s(k), the k-th element of vector s.

Examples

```
Hq = qfilt;
scalevalues(Hq)
ans =

      1
```

See Also qfilt, get, set

Purpose

Set or display property values for quantized filters, quantizers, and quantized FFTs

Syntax

```
set(Hq)
set(Hq,'prop',value)
set(Hq,'prop1',value1, 'prop2',value2,...)
s = set(Hq)
set(Hq,struct)
set(Hq,{'prop1','prop2',...},{value1,value2,...})
set(q, PropertyValue1, PropertyValue2, ... )
set(q,a)
set(q,pn,pv)
set(q, PropertyName1 ,PropertyValue1, PropertyName2 ,
  PropertyValue2,...)
q.PropertyName = Value
set(q)
s = set(q)
set(F,'PropertyName',PropertyValue)
set(F,'PropertyName1',PropertyValue1,'PropertyName2',
  PropertyValue2,...)
set(F,a)
set(F,pn,pv)
F.PropertyName = Value
set(F)
s = set(F)
```

Description

`set(Hq)` displays all of the property names and their possible values for a given quantized filter `Hq`. The display indicates the default values for properties in braces. When the default values for a property cannot be represented by a finite list, `set(Hq)` does not display the property's default values.

`set(Hq,'prop',value)` sets the values for the property '*prop*' of a quantized filter `Hq`. You specify the property name by the string `'prop'`, and the associated value in `value`. `'prop'` can be any of the properties listed in Table 12-3, Quick Guide to Quantized Filter Properties, on page 12-10. `value` can be a string, a numerical value, or a cell array containing numerical values. The possible values for each property are described in detail in "Quantized Filter Properties Reference" on page 12-11.

`set(Hq,'prop1',value1,'prop2',value2,...)` lets you set multiple properties in one command.

`s = set(Hq)` returns all property names and their possible values for a quantized filter `Hq`. `s` is a MATLAB structure whose field names are the property names of `Hq` and whose values are cell arrays of possible property values, except when the possible values for the property cannot be described with a finite list. In this case the values are empty cell arrays.

`set(Hq,struct)` sets the properties of the quantized filter `Hq` according to the values associated with the field names of the MATLAB structure `struct`. All field names for the structure `s` must be valid quantized filter properties. See Table 12-3, Quick Guide to Quantized Filter Properties, on page 12-10 for a list of all property names.

`set(Hq,{'prop1','prop2',...},{value1,value2,...})` sets the listed properties specified in the cell array of a vector of strings `{'prop1','prop2',...}` to the corresponding values listed in the cell array `{value1,value2,...}` for quantized filter object `Hq`. The two cell array input arguments must be the same size, and the values must be valid for the corresponding properties.

`set(q, PropertyValue1, PropertyValue2,...)` sets the properties of quantizer `q`. If two property values conflict, the last value in the list is the one that is set.

`set(q,a)` where `a` is a structure whose field names are object property names, sets the properties named in each field name with the values contained in the structure.

`set(q,pn,pv)` sets the named properties specified in the cell array of strings `pn` to the corresponding values in the cell array `pv`.

`set(q, PropertyName1 ,PropertyValue1, PropertyName2 , PropertyValue2,...)` sets multiple property values with a single statement. Note that you can use property name/property value string pairs, structures, and property name/property value cell array pairs in the same call to `set`.

`q.PropertyName = Value` uses the dot notation to set property `PropertyName` to `Value`.

`set(q)` displays the possible values for all properties of quantizer `q`.

s = set(q) returns a structure containing the possible values for the properties of quantizer q.

The states are cleared when you set any value other than WarnIfOverflow.

For a quantizer, these are the possible property values, sorted by property name.

Property Name	Property Value	Description
Mode	'double'	Double-precision mode. Override all other parameters.
	'float'	Custom-precision floating-point mode.
	'fixed'	Signed fixed-point mode.
	'single'	Single-precision mode. Override all other parameters.
	'ufixed'	Unsigned fixed-point mode.
Roundmode	'ceil'	Round towards negative infinity.
	'convergent'	Convergent rounding.
	'fix'	Round towards zero.
	'floor'	Round towards positive infinity.
	'round'	Round towards nearest.
Overflowmode (fixed-point only)	'saturate'	Saturate at max value on overflow.
	'wrap'	Wrap on overflow.
Format	[wordlength exponentlength]	The format for fixed or ufixed mode.
	[wordlength exponentlength]	The format for float mode.

Property Name (Continued)	Property Value	Description
`Max`		Maximum value before quantize.
`Min`		Minimum value before quantize.
`NOverflows`		Number of overflows.
`NUnderflows`		Number of underflows.

`set(F,'PropertyName',PropertyValue)` sets the value of the specified property for the quantized FFT `F`.

`set(F,'PropertyName1',PropertyValue1,'PropertyName2',PropertyValue2,...)` sets multiple property values with a single statement. Note that you can use property name/property value string pairs, structures, and property name/property value cell array pairs in the same call to `set`.

`set(F,a)` where `a` is a structure whose field names are object property names, sets the properties named in each field name with the values contained in the structure.

`set(F,pn,pv)` sets the named properties specified in the cell array of strings `pn` to the corresponding values in the cell array `pv` for all objects specified in `H`.

`F.PropertyName = Value` uses the dot notation to set property `PropertyName` to `Value`.

`set(F)` displays the possible values.

`s = set(F)` returns a structure with the possible values.

Remarks

- Property names are not case sensitive.
- You can abbreviate property names to the shortest uniquely identifying string.
- You can use direct property referencing to set properties with a structure-like syntax. The following two statements are equivalent:
  - `set(Hq,'roundm','convergent');`
  - `Hq.round = 'convergent';`

Examples

Create a quantized filter and change the values for the `ReferenceCoefficients` and `InputFormat` properties.

```
Hq = qfilt;
set(Hq,'ref',{[1 .5] [1 .7 .89]},'inp',[16,14])
Hq

Hq =
Quantized direct-form II transposed filter
Numerator
     QuantizedCoefficients{1}    ReferenceCoefficients{1}
+ (1)       0.999969482421875  1.000000000000000000
  (2)       0.500000000000000  0.500000000000000000
Denominator
     QuantizedCoefficients{2}    ReferenceCoefficients{2}
+ (1)       0.999969482421875  1.000000000000000000
  (2)       0.700012207031250  0.699999999999999960
  (3)       0.890014648437500  0.890000000000000010

   FilterStructure = df2t
       ScaleValues = [1]
  NumberOfSections = 1
  StatesPerSection = [2]
 CoefficientFormat = quantizer('fixed', 'round', 'saturate', [16  15])
       InputFormat = quantizer('fixed', 'floor', 'saturate', [16  14])
      OutputFormat = quantizer('fixed', 'floor', 'saturate', [16  15])
MultiplicandFormat = quantizer('fixed', 'floor', 'saturate', [16  15])
     ProductFormat = quantizer('fixed', 'floor', 'saturate', [32  30])
         SumFormat = quantizer('fixed', 'floor', 'saturate', [32  30])
Warning: 2 overflows in coefficients.
```

You can create a structure to assign the same data format property values to a set of filters.

```
s.InputFor = [16,14];
s.Coefficient = [16,14];
s.SumF = [17,15];
s.Prod = [16,15];
s.output = [24,23];
```

Now assign those property values to the filter in the previous example.

```
set(Hq,s)
Hq

Hq =
```

```
Quantized Direct-form II transposed filter
Numerator
     QuantizedCoefficients{1}    ReferenceCoefficients{1}
  (1)        1.00000000000000  1.000000000000000000
  (2)        0.50000000000000  0.500000000000000000
Denominator
     QuantizedCoefficients{2}    ReferenceCoefficients{2}
  (1)        1.00000000000000  1.000000000000000000
  (2)        0.70001220703125  0.699999999999999960
  (3)        0.89001464843750  0.890000000000000010

   FilterStructure = df2t
       ScaleValues = [1]
  NumberOfSections = 1
  StatesPerSection = [2]
 CoefficientFormat = quantizer('fixed', 'round', 'saturate', [16  14])
       InputFormat = quantizer('fixed', 'floor', 'saturate', [16  14])
      OutputFormat = quantizer('fixed', 'floor', 'saturate', [24  23])
MultiplicandFormat = quantizer('fixed', 'floor', 'saturate', [16  15])
     ProductFormat = quantizer('fixed', 'floor', 'saturate', [16  15])
         SumFormat = quantizer('fixed', 'floor', 'saturate', [17  15])
```

Notice that you do not have to provide complete property names when you create the structure fields.

The next example uses property name/property value pairs to set quantizer properties.

```
q = quantizer;
set(q, 'mode','fixed', ...
       'roundmode','ceil', ...
       'overflowmode','wrap', ...
       'format',[24 22]);
```

O you might use dot notation to enter the new property values:

```
q = quantizer;
q.mode = 'fixed';
q.roundmode = 'ceil';
q.overflowmode = 'wrap';
q.format = [24 22];
```

With no output arguments and one input argument, `set` displays the defaults for the quantizer, quantized filter, or quantized FFT.

```
q = quantizer;
set(q)
```

```
        Mode: [double | float | {fixed} | single | ufixed]
   RoundMode: [ceil | convergent | fix | {floor} | round]
OverflowMode: [{saturate} | wrap]
      Format: [wordlength  fractionlength] - In 'fixed', 'ufixed' mode.
              [wordlength  exponentlength] - In 'float' mode.
              [16 15] = default.
         Max: Maximum value before quantize.
         Min: Minimum value before quantize.
  NOverflows: Number of overflows.
 NUnderflows: Number of underflows.
```

With one output argument and one input argument, `set` returns a structure.

```
q = quantizer;
s = set(q)
```

returns

```
s =
            Mode: {'double'  'float'  'fixed'  'single'  'ufixed'}
       RoundMode: {'ceil'  'convergent'  'fix'  'floor'  'round'}
    OverflowMode: {'saturate'  'wrap'}
          Format: {}
             Max: {}
             Min: {}
       Overflows: {}
     NUnderflows: {}
```

See Also

`get`, `qfilt`, `setbits`, `sos2cell`, `sos`

Purpose

Set all data format property values for quantized filters and quantized FFTs

Syntax

```
setbits(Hq,format)
setbits(F,fmt)
```

Description

When `Hq` is a floating-point quantized filter, `setbits(Hq,format)` sets all data format properties for the quantized filter `Hq` to the values specified by `format`. In this case, `format` is a two-element vector of integers whose entries are described as follows:

- The first entry in `format` sets the word length in bits.
- The second entry in `format` sets the exponent length in bits.

When `Hq` is a fixed-point quantized filter, `setbits(Hq,format)` sets the properties `CoefficientFormat`, `InputFormat`, and `OutputFormat` to the value specified by `format`, whereas the property values `SumFormat` and `ProductFormat` are specified by `2*format`. In this case, `format` is a two-element vector of integers whose entries are described as follows:

- The first entry in `format` sets the word length in bits.
- The second entry in `format` sets the fraction length (the number of bits after the radix point).

Note When `2*format` exceeds the maximum values for the `SumFormat` and `ProductFormat` properties, their maximum values are used instead.

`setbits(F,fmt)` sets all data format property values for quantized FFT `F`.

When `F` is a fixed-point quantized FFT, `fmt = [w, f]` where `w` is the word length and `f` is the fraction length. The twiddle, input, and output formats are set to `[w, f]`. The sum and product formats are set to `[2w, 2f]`.

When `F` is a floating-point quantized FFT, `fmt = [w, e]` where `w` is the word length and `e` is the exponent length. All formats are set to `[w, e]`.

If the specified formats exceed the maximum allowed, they are set to the maximum.

Examples

Create a quantized filter with default data format property values. Set the `CoefficientFormat`, `InputFormat`, and `OutputFormat` property values for a 24-bit word length, and a 23-bit fraction length, while setting the `SumFormat` and `ProductFormat` property values to a 48-bit word length and a 46-bit fraction length.

```
Hq = qfilt;
setbits(Hq,[24 23])

get(Hq)

Quantized Direct form II transposed filter
Numerator
     QuantizedCoefficients{1}    ReferenceCoefficients{1}
+ (1)      0.9999998807907105  1.000000000000000000
  (2)      0.5000000000000000  0.500000000000000000
Denominator
     QuantizedCoefficients{2}    ReferenceCoefficients{2}
+ (1)      0.9999998807907105  1.000000000000000000
  (2)      0.7000000476837158  0.699999999999999960
  (3)      0.8899999856948853  0.890000000000000010

   FilterStructure = df2t
       ScaleValues = [1]
  NumberOfSections = 1
  StatesPerSection = [2]
 CoefficientFormat = quantizer('fixed', 'round', 'saturate', [24  23])
       InputFormat = quantizer('fixed', 'floor', 'saturate', [24  23])
      OutputFormat = quantizer('fixed', 'floor', 'saturate', [24  23])
MultiplicandFormat = quantizer('fixed', 'floor', 'saturate', [24  23])
     ProductFormat = quantizer('fixed', 'floor', 'saturate', [48  46])
         SumFormat = quantizer('fixed', 'floor', 'saturate', [48  46])
```

See Also

`get`, `qfilt`, `set`

Purpose

Convert a quantized filter to second-order section form, order, and scale.

Syntax

```
Hq2 = sos(Hq)
Hq2 = sos(Hq, order)
Hq2 = sos(Hq, order, scale)
```

Description

`Hq2 = sos(Hq)` returns a quantized filter `Hq2` that has second-order sections and the `dft2` structure. Use the same optional arguments used in `tf2sos`.

`Hq2 = sos(Hq, order)` specifies the order of the sections in `Hq2`, where `order` is either of the following strings:

- `'down'` — to order the sections so the first section of `Hq2` contains the poles closest to the unit circle (L_∞ norm scaling)
- `'up'` — to order the sections so the first section of `Hq2` contains the poles farthest from the unit circle (L_2 norm scaling and the default)

`Hq2 = sos(Hq, order, scale)` also specifies the desired scaling of the gain and numerator coefficients of all second-order sections, where `scale` is one of the following strings:

- `'none'` — to apply no scaling (default)
- `'inf'` — to apply infinity-norm scaling
- `'two'` — to apply 2-norm scaling

Use infinity-norm scaling in conjunction with up-ordering to minimize the probability of overflow in the filter realization. Consider using 2-norm scaling in conjunction with down-ordering to minimize the peak round-off noise.

When `Hq` is a fixed-point filter, the filter coefficients are normalized so that the magnitude of the maximum coefficient in each section is 1. The gain of the filter is applied to the first scale value of `Hq2`.

`sos` uses the direct form II transposed (`dft2`) structure to implement second-order section filters.

Examples

```
[A,B,C,D]=butter(8,.5);
Hq = qfilt('StateSpace',{A,B,C,D},'mode','single');
Hq1 = sos(Hq)
```

See Also

qfilt, qfilt2tf

tf2sos in your Signal Processing Toolbox documentation

Purpose

Transfer function to coupled allpass conversion

Syntax

```
[d1,d2] = tf2ca(b,a)
[d1,d2] = tf2ca(b,a)
[d1,d2,beta] = tf2ca(b,a)
```

Description

`[d1,d2] = tf2ca(b,a)` where `b` is a real, symmetric vector of numerator coefficients and `a` is a real vector of denominator coefficients, corresponding to a stable digital filter, returns real vectors `d1` and `d2` containing the denominator coefficients of the allpass filters *H1(z)* and *H2(z)* such that

$$H(z) = \frac{B(z)}{A(z)} = \frac{1}{2[H1(z) + H2(z)]}$$

representing a coupled allpass decomposition.

`[d1,d2] = tf2ca(b,a)` where `b` is a real, antisymmetric vector of numerator coefficients and `a` is a real vector of denominator coefficients, corresponding to a stable digital filter, returns real vectors `d1` and `d2` containing the denominator coefficients of the allpass filters *H1(z)* and *H2(z)* such that

$$H(z) = \frac{B(z)}{A(z)} = \left(\frac{1}{2}\right)[H1(z) - H2(z)]$$

In some cases, the decomposition is not possible with real *H1(z)* and *H2(z)*. In those cases a generalized coupled allpass decomposition may be possible, whose syntax is

```
[d1,d2,beta] = tf2ca(b,a)
```

to return complex vectors `d1` and `d2` containing the denominator coefficients of the allpass filters *H1(z)* and *H2(z)*, and a complex scalar `beta`, satisfying `|beta| = 1`, such that

$$H(z) = \frac{B(z)}{A(z)} = \left(\frac{1}{2}\right)[\bar{\beta} \bullet H1(z) + \beta \bullet H2(z)]$$

representing the generalized allpass decomposition.

In the above equations, *H1(z)* and *H2(z)* are real or complex allpass IIR filters given by

$$H1(z) = \frac{fliplr((\overline{D1(z)}))}{D1(z)}, H2(1)(z) = \frac{fliplr((\overline{D2(1)(z)}))}{D2(1)(z)}$$

where *D1(z)* and *D2(z)* are polynomials whose coefficients are given by `d1` and `d2`.

Note A coupled allpass decomposition is not always possible. Nevertheless, Butterworth, Chebyshev, and Elliptic IIR filters, among others, can be factored in this manner. For details, refer to *Signal Processing Toolbox User's Guide*.

Examples

```
[b,a]=cheby1(9,.5,.4);
[d1,d2]=tf2ca(b,a); % TF2CA returns denominators of the allpass.
num = 0.5*conv(fliplr(d1),d2)+0.5*conv(fliplr(d2),d1);
den = conv(d1,d2); % Reconstruct numerator and denonimator.
max([max(b-num),max(a-den)]) % Compare original and reconstructed
                             % numerator and denominators.
```

See Also

`ca2tf`, `cl2tf`, `iirpowcomp`, `latc2tf`, `tf2latc`

Purpose

Transfer function to coupled allpass lattice conversion

Syntax

```
[k1,k2] = tf2cl(b,a)
[k1,k2] = tf2cl(b,a)
```

Description

`[k1,k2] = tf2cl(b,a)` where `b` is a real, symmetric vector of numerator coefficients and `a` is a real vector of denominator coefficients, corresponding to a stable digital filter, will perform the coupled allpass decomposition

$$H(z) = \frac{B(z)}{A(z)} = \frac{1}{2[H1(z) + H2(z)]}$$

of a stable IIR filter *H(z)* and convert the allpass transfer functions *H1(z)* and H2(z) to a coupled lattice allpass structure with coefficients given in vectors `k1` and `k2`.

`[k1,k2] = tf2cl(b,a)` where `b` is a real, antisymmetric vector of numerator coefficients and `a` is a real vector of denominator coefficients, corresponding to a stable digital filter, performs the coupled allpass decomposition

$$H(z) = \frac{B(z)}{A(z)} = \left(\frac{1}{2}\right)[H1(z) - H2(z)]$$

of a stable IIR filter *H(z)* and converts the allpass transfer functions *H1(z)* and *H2(z)* to a coupled lattice allpass structure with coefficients given in vectors `k1` and `k2`.

In some cases, the decomposition is not possible with real *H1(z)* and *H2(z)*. In those cases, a generalized coupled allpass decomposition may be possible, using the command syntax

```
[k1,k2,beta] = tf2cl(b,a)
```

to perform the generalized allpass decomposition of a stable IIR filter *H(z)* and convert the complex allpass transfer functions *H1(z)* and *H2(z)* to corresponding lattice allpass filters

$$H(z) = \frac{B(z)}{A(z)} = \left(\frac{1}{2}\right)[\bar{\beta} \bullet H1(z) + \beta \bullet H2(z)]$$

where `beta` is a complex scalar of magnitude equal to 1.

Note Coupled allpass decomposition is not always possible. Nevertheless, Butterworth, Chebyshev, and Elliptic IIR filters, among others, can be factored in this manner. For details, refer to *Signal Processing Toolbox User's Guide*.

Examples

```
[b,a]=cheby1(9,.5,.4);
[k1,k2]=tf2cl(b,a); % Get the reflection coeffs. for the lattices.
[num1,den1]=latc2tf(k1,'allpass'); % Convert each allpass lattice
[num2,den2]=latc2tf(k2,'allpass'); % back to transfer function.
num = 0.5*conv(num1,den2)+0.5*conv(num2,den1);
den = conv(den1,den2); % Reconstruct numerator and denonimator.
max([max(b-num),max(a-den)]) % Compare original and reconstructed
                             % numerator and denominators.
```

See Also

`ca2tf`, `cl2tf`, `iirpowcomp`, `latc2tf`, `tf2ca`, `tf2latc`

Purpose Convert a quantizer, unitquantizer, or quantized FFT to a string

Syntax

```
s = tostring(q)
s = tostring(q)
s = tostring(f)
```

Description

`s = tostring(q)` converts quantizer `q` to a string `s`. After converting `q` to a string, the function `eval(s)` can use `s` to create a quantizer with the same properties as `q`.

`s = tostring(q)` converts unitquantizer `q` to a string `s`. After converting `q` to a string, the function `eval(s)` can use `s` to create a quantizer with the same properties as `q`.

`s = tostring(q)` converts quantized FFT `f` to a string `s`. After converting `f` to a string, the function `eval(s)` can use `f` to create a quantized FFT with the same properties as `f`.

Examples

When you use `tostring` with a quantizer or unitquantizer, you see the following response.

```
q = quantizer
q =
Mode = fixed
       RoundMode = floor
    OverflowMode = saturate
          Format = [16  15]


             Max = reset
             Min = reset
      NOverflows = 0
     NUnderflows = 0
     NOperations = 0

s = tostring(q)
s =
quantizer('fixed', 'floor', 'saturate', [16  15])

eval(s)
```

```
ans =

            Mode = fixed
       RoundMode = floor
    OverflowMode = saturate
          Format = [16  15]

             Max = reset
             Min = reset
      NOverflows = 0
     NUnderflows = 0
     NOperations = 0
```

and s is the same as q.

For a quantized FFT, the result is the same.

```
f = qfft
f =

Radix = 2
           Length = 16
 CoefficientFormat = quantizer('fixed', 'round', 'saturate', [16  15])
       InputFormat = quantizer('fixed', 'floor', 'saturate', [16  15])
      OutputFormat = quantizer('fixed', 'floor', 'saturate', [16  15])
MultiplicandFormat = quantizer('fixed', 'floor', 'saturate', [16  15])
     ProductFormat = quantizer('fixed', 'floor', 'saturate', [32  30])
         SumFormat = quantizer('fixed', 'floor', 'saturate', [32  30])
  NumberOfSections = 4
       ScaleValues = [1]
s=tostring(f)

eval(s)

ans =

             Radix = 2
            Length = 16
 CoefficientFormat = quantizer('fixed', 'round', 'saturate', [16  15])
       InputFormat = quantizer('fixed', 'floor', 'saturate', [16  15])
      OutputFormat = quantizer('fixed', 'floor', 'saturate', [16  15])
MultiplicandFormat = quantizer('fixed', 'floor', 'saturate', [16  15])
     ProductFormat = quantizer('fixed', 'floor', 'saturate', [32  30])
         SumFormat = quantizer('fixed', 'floor', 'saturate', [32  30])
```

```
NumberOfSections = 4
     ScaleValues = [1]
```

See Also

`quantizer`, `qfft`, `unitquantizer`

twiddles

Purpose Return the quantized twiddle factors for quantized FFTs

Syntax

```
w = twiddles(F)
```

Description `w = twiddles(F)` returns a vector of the quantized FFT coefficients specified by quantized FFT `F`. FFT coefficients are also called *twiddle factors*.

Examples

```
f = qfft;
w = twiddles(f)
Warning: 4 overflows.
w =

   1.0000
   1.0000
        0 - 1.0000i
   1.0000
   0.7071 - 0.7071i
        0 - 1.0000i
  -0.7071 - 0.7071i
   1.0000
   0.9239 - 0.3827i
   0.7071 - 0.7071i
   0.3827 - 0.9239i
        0 - 1.0000i
  -0.3827 - 0.9239i
  -0.7071 - 0.7071i
  -0.9239 - 0.3827i
```

See Also `qfft`

Purpose Quantize all numbers in a data set except numbers within `eps` of 1

Syntax `unitquantize(q,...)`

Description `unitquantize(q,...)` works the same as `quantize` except that numbers within `eps(q)` of 1 are made exactly equal to 1 (see `quantize` for a description of the parameters).

This function is especially useful for quantizing fixed-point coefficients.

Examples

```
[b,a] = ellip(4,3,20,.6);
m = tf2sos(b,a)
m =

    0.2758    0.3883    0.2758    1.0000   -0.0548    0.4731
    1.0000    0.7197    1.0000    1.0000    0.5884    0.9574

m==1

ans =

     0     0     0     1     0     0
     1     0     0     1     0     0

q=quantizer;
m > realmax(q)

ans =

     0     0     0     1     0     0
     1     0     1     1     0     0
```

It appears that there are four elements that are exactly equal to 1. In fact, there are only three. Element `m`(2,3) is greater than `realmax(q)`, but less than 1. Ordinarily, `m`(2,3) would be counted as an overflow and be set to `realmax(q)`. However, the desired behavior would be to force `m`(2,3) to be equal to 1 without recording an overflow. This is what `unitquantize` does, as shown in the following example.

```
m = unitquantize(q,m)
m =

    0.2758    0.3882    0.2758    1.0000   -0.0548    0.4731
    1.0000    0.7197    1.0000    1.0000    0.5884    0.9574
m==1

ans =

       0      0      0      1      0      0
       1      0      1      1      0      0
```

By forcing values between eps and 1 to be equal to 1, signal processing algorithms can avoid multiplication operations that involve these numbers, saving processing steps and time.

See Also `qfft`, `quantize`

unitquantizer

Purpose

Construct a unit quantizer

For help on this function, enter `help unitquantizer` at the MATLAB prompt.

Syntax

```
q = unitquantizer(...)
```

`q = unitquantizer(...)` constructs a unitquantizer, which is identical to a quantizer in all respects except that its `quantize` method quantizes numbers within `eps(q)` of 1 to be equal to 1. Refer to `quantizer` for arguments and parameters for the `unitquantizer` function.

Examples

```
u = unitquantizer([4 3]);
quantize(u,1)
ans =

     1

q = quantizer([4 3]);
quantize(q,1)
Warning: 1 overflow.

ans =

    0.8750
```

See Also

`quantizer`, `unitquantize`

wordlength

Purpose Return the word length for a quantizer

Syntax `wordlength(q)`

Description `wordlength(q)` returns the word length of quantizer `q`.

Examples

```
q = quantizer([16 15]);
wordlength(q)
```

returns 16.

Even though the word length can be read in two stages,

```
q = quantizer([16 15]);
fmt = q.format;
w = fmt(1);
```

it is handy to have it available for use in equations. For example, the algorithm for `realmax(q)` when `q` is a signed fixed-point quantizer (`q.mode = 'fixed'`) is

$$r = 2^{w-f-1} - \varepsilon$$

which can be coded as

```
q = quantizer('fixed',[8 4]);
r = pow2(wordlength(q) - fractionlength(q) - 1) - eps(q)
```

See Also `fractionlength`, `exponentlength`

zpkbpc2bpc

Purpose

Zero-pole-gain complex bandpass frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpkbpc2bpc(Z,P,K,Wo,Wt)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpkbpc2bpc(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the complex bandpass prototype by applying a first-order complex bandpass to complex bandpass frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The original lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

This transformation effectively places two features of an original filter, located at frequencies W_{o1} and W_{o2}, at the required target frequency locations, W_{t1}, and W_{t2} respectively. It is assumed that W_{t2} is greater than W_{t1}. In most of the cases the features selected for the transformation are the band edges of the filter passbands. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

This transformation can also be used for transforming other types of filters; e.g., complex notch filters or resonators can be repositioned at two distinct desired frequencies at any place around the unit circle; e.g., in the adaptive system.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
```

Create a complex passband from 0.25 to 0.75:

```
[b, a] = iirlp2bpc(b,a,0.5,[0.25,0.75]);
z = roots(b);
p = roots(a);
k = b(1);
[z2,p2,k2] = zpkbpc2bpc(z, p, k, [0.25, 0.75], [-0.75, -0.25]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Arguments

`Z`
Zeros of the prototype lowpass filter

`P`
Poles of the prototype lowpass filter

`K`
Gain factor of the prototype lowpass filter

`Wo`
Frequency value to be transformed from the prototype filter

`Wt`
Desired frequency location in the transformed target filter

`Z2`
Zeros of the target filter

`P2`
Poles of the target filter

`K2`
Gain factor of the target filter

`AllpassNum`
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

See Also

`zpkftransf`, `allpassbpc2bpc`, `iirbpc2bpc`

Purpose

Zero-pole-gain frequency transformation of the digital filter

Syntax

```
[Z2,P2,K2] = zpkftransf(Z,P,K,AllpassNum,AllpassDen)
```

Description

`[Z2,P2,K2] = zpkftransf(Z,P,K,AllpassNum,AllpassDen)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the transformed lowpass digital filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`. If `AllpassDen` is not specified it will default to 1. If neither `AllpassNum` nor `AllpassDen` is specified, then the function returns the input filter.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
[AlpNum, AlpDen] = allpasslp2lp(0.5, 0.25);
[z2, p2, k2] = zpkftransf(roots(b),roots(a),b(1),AlpNum,AlpDen);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Arguments

`Z`
Zeros of the prototype lowpass filter

`P`
Poles of the prototype lowpass filter

`K`
Gain factor of the prototype lowpass filter

`FTFNum`
Numerator of the mapping filter

`FTFDen`
Denominator of the mapping filter

`Z2`
Zeros of the target filter

`P2`
Poles of the target filter

`K2`
Gain factor of the target filter

See Also

iirftransf

Purpose Zero-pole-gain lowpass to bandpass frequency transformation

Syntax `[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bp(Z,P,K,Wo,Wt)`

Description `[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bp(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the real lowpass prototype by applying a second-order real lowpass to real bandpass frequency mapping.

It also returns the numerator, `AllpassNum`, and the denominator `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

This transformation effectively places one feature of an original filter, located at frequency -W_o, at the required target frequency location, W_{t1}, and the second feature, originally at +W_o, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}. This transformation implements the “DC Mobility,” which means that the Nyquist feature stays at Nyquist, but the DC feature moves to a location dependent on the selection of W_t.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandpass transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Real lowpass to bandpass transformation can also be used for transforming other types of filters; e.g., real notch filters or resonators can be easily doubled and positioned at two distinct, desired frequencies.

Examples Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
```

```
[z2,p2,k2] = zpklp2bp(z, p, k, 0.5, [0.2 0.3]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Arguments

`Z`
Zeros of the prototype lowpass filter

`P`
Poles of the prototype lowpass filter

`K`
Gain factor of the prototype lowpass filter

`Wo`
Frequency value to be transformed from the prototype filter

`Wt`
Desired frequency location in the transformed target filter

`Z2`
Zeros of the target filter

`P2`
Poles of the target filter

`K2`
Gain factor of the target filter

`AllpassNum`
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`zpkftransf`, `allpasslp2bp`, `iirlp2bp`

References

[1] Constantinides, A.G., "Spectral transformations for digital filters," *IEE 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] Nowrouzian, B. and L.T. Bruton, “Closed-form solutions for discrete-time elliptic transfer functions,” *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

[4] Constantinides, A.G., “Design of bandpass digital filters,” *IEEE Proceedings*, vol. 1, pp. 1129-1231, June 1969.

zpklp2bpc

Purpose

Zero-pole-gain lowpass to complex bandpass frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bpc(Z,P,K,Wo,Wt)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bpc(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the real lowpass prototype by applying a first-order real lowpass to complex bandpass frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

This transformation effectively places one feature of an original filter, located at frequency $-W_o$, at the required target frequency location, W_{t1}, and the second feature, originally at $+W_o$, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandpass transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Lowpass to bandpass transformation can also be used for transforming other types of filters; e.g., real notch filters or resonators can be doubled and positioned at two distinct desired frequencies at any place around the unit circle forming a pair of complex notches/resonators. This transformation can be used for designing bandpass filters for radio receivers from the high-quality prototype lowpass filter.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
```

```
k = b(1);
[z2,p2,k2] = zpklp2bpc(z, p, k, 0.5, [0.2 0.3]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Arguments

`Z`
Zeros of the prototype lowpass filter

`P`
Poles of the prototype lowpass filter

`K`
Gain factor of the prototype lowpass filter

`Wo`
Frequency value to be transformed from the prototype filter. It should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

`Wt`
Desired frequency locations in the transformed target filter. They should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

`Z2`
Zeros of the target filter

`P2`
Poles of the target filter

`K2`
Gain factor of the target filter

`AllpassNum`
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

See Also

`zpkftransf`, `allpasslp2bpc`, `iirlp2bpc`

zpklp2bs

Purpose

Zero-pole-gain lowpass to bandstop frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bs(Z,P,K,Wo,Wt)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bs(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the real lowpass prototype by applying a second-order real lowpass to real bandstop frequency mapping.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

This transformation effectively places one feature of an original filter, located at frequency -W_o, at the required target frequency location, W_{t1}, and the second feature, originally at +W_o, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}. This transformation implements the "Nyquist Mobility," which means that the DC feature stays at DC, but the Nyquist feature moves to a location dependent on the selection of W_o and W_ts.

Relative positions of other features of an original filter change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. After the transformation feature F_2 will precede F_1 in the target filter. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandstop transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z2,p2,k2] = zpklp2bs(z, p, k, 0.5, [0.2 0.3]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Arguments

`Z`
Zeros of the prototype lowpass filter

`P`
Poles of the prototype lowpass filter

`K`
Gain factor of the prototype lowpass filter

`Wo`
Frequency value to be transformed from the prototype filter

`Wt`
Desired frequency location in the transformed target filter

`Z2`
Zeros of the target filter

`P2`
Poles of the target filter

`K2`
Gain factor of the target filter

`AllpassNum`
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`zpkftransf`, `allpasslp2bs`, `iirlp2bs`

References

[1] Constantinides, A.G., “Spectral transformations for digital filters,” *IEE 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] Nowrouzian, B. and L.T. Bruton, “Closed-form solutions for discrete-time elliptic transfer functions,” *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

[4] Constantinides, A.G., “Design of bandpass digital filters,” *IEEE Proceedings*, vol. 1, pp. 1129-1231, June 1969.

Purpose

Zero-pole-gain lowpass to complex bandstop frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bsc(Z,P,K,Wo,Wt)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bsc(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the real lowpass prototype by applying a first-order real lowpass to complex bandstop frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

This transformation effectively places one feature of an original filter, located at frequency $-W_o$, at the required target frequency location, W_{t1}, and the second feature, originally at $+W_o$, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}. Additionally the transformation swaps passbands with stopbands in the target filter.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandstop transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Lowpass to bandpass transformation can also be used for transforming other types of filters; e.g., real notch filters or resonators can be doubled and positioned at two distinct desired frequencies at any place around the unit circle forming a pair of complex notches/resonators.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
```

```
[z2,p2,k2] = zpklp2bsc(z, p, k, 0.5, [0.2, 0.3]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Arguments

`Z`
Zeros of the prototype lowpass filter

`P`
Poles of the prototype lowpass filter

`K`
Gain factor of the prototype lowpass filter

`Wo`
Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

`Wt`
Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

`Z2`
Zeros of the target filter

`P2`
Poles of the target filter

`K2`
Gain factor of the target filter

`AllpassNum`
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

See Also

`zpkftransf`, `allpasslp2bsc`, `iirlp2bsc`

Purpose

Zero-pole-gain lowpass to highpass frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2hp(Z,P,K,Wo,Wt)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2hp(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the real lowpass prototype by applying a first-order real lowpass to real highpass frequency mapping. This transformation effectively places one feature of an original filter, located at frequency W_o, at the required target frequency location, W_t, at the same time rotating the whole frequency response by half of the sampling frequency. Result is that the DC and Nyquist features swap places.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and the gain factor, `K`.

Relative positions of other features of an original filter change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. After the transformation feature F_2 will precede F_1 in the target filter. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to highpass transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, or the deep minimum in the stopband, or other ones.

Lowpass to highpass transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can change their position in a simple way without designing them again.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z2,p2,k2] = zpklp2hp(z, p, k, 0.5, 0.25);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Arguments

`Z`
Zeros of the prototype lowpass filter

`P`
Poles of the prototype lowpass filter

`K`
Gain factor of the prototype lowpass filter

`Wo`
Frequency value to be transformed from the prototype filter

`Wt`
Desired frequency location in the transformed target filter

`Z2`
Zeros of the target filter

`P2`
Poles of the target filter

`K2`
Gain factor of the target filter

`AllpassNum`
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`zpkftransf`, `allpasslp2hp`, `iirlp2hp`

References

[1] Constantinides, A.G., “Spectral transformations for digital filters,” *IEE 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] Nowrouzian, B. and L.T. Bruton, “Closed-form solutions for discrete-time elliptic transfer functions,” *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

[4] Constantinides, A.G., “Frequency transformations for digital filters,” *Electronics Letters*, vol. 3, no. 11, pp. 487-489, November 1967.

zpklp2lp

Purpose

Zero-pole-gain lowpass to lowpass frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2lp(Z,P,K,Wo,Wt)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2lp(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the real lowpass prototype by applying a first-order real lowpass to real lowpass frequency mapping. This transformation effectively places one feature of an original filter, located at frequency W_o, at the required target frequency location, W_t.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to lowpass transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Lowpass to lowpass transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can change their position in a simple way without designing them again.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z2,p2,k2] = zpklp2lp(z, p, k, 0.5, 0.25);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Arguments

`Z`
Zeros of the prototype lowpass filter

`P`
Poles of the prototype lowpass filter

`K`
Gain factor of the prototype lowpass filter

`Wo`
Frequency value to be transformed from the prototype filter

`Wt`
Desired frequency location in the transformed target filter

`Z2`
Zeros of the target filter

`P2`
Poles of the target filter

`K2`
Gain factor of the target filter

`AllpassNum`
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`zpkftransf`, `allpasslp2lp`, `iirlp2lp`

References

[1] Constantinides, A.G., “Spectral transformations for digital filters,” *IEE 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] Nowrouzian, B. and L.T. Bruton, “Closed-form solutions for discrete-time elliptic transfer functions,” *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

[4] Constantinides, A.G., “Frequency transformations for digital filters,” *Electronics Letters*, vol. 3, no. 11, pp. 487-489, November 1967.

Purpose Zero-pole-gain lowpass to M-band frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2mb(Z,P,K,Wo,Wt)
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2mb(Z,P,K,Wo,Wt,Pass)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2mb(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the real lowpass prototype by applying an `M`th-order real lowpass to real multibandpass frequency mapping. By default the DC feature is kept at its original location.

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2mb(Z,P,K,Wo,Wt,Pass)` allows you to specify an additional parameter, `Pass`, which chooses between using the "DC Mobility" and the "Nyquist Mobility". In the first case the Nyquist feature stays at its original location and the DC feature is free to move. In the second case the DC feature is kept at an original frequency and the Nyquist feature is allowed to move.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

This transformation effectively places one feature of an original filter, located at frequency W_o, at the required target frequency locations, W_{t1},...,W_{tM}.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z1,p1,k1] = zpklp2mb(z, p, k, 0.5, [2 4 6 8]/10, 'pass');
[z2,p2,k2] = zpklp2mb(z, p, k, 0.5, [2 4 6 8]/10, 'stop');
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k1*poly(z1), poly(p1), k2*poly(z2), poly(p2));
```

Arguments

`Z`
Zeros of the prototype lowpass filter

`P`
Poles of the prototype lowpass filter

`K`
Gain factor of the prototype lowpass filter

`Wo`
Frequency value to be transformed from the prototype filter

`Wt`
Desired frequency location in the transformed target filter

`Pass`
Choice (`'pass'`/`'stop'`) of passband/stopband at DC, `pass` being the default

`Z2`
Zeros of the target filter

`P2`
Poles of the target filter

`K2`
Gain factor of the target filter

`AllpassNum`
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`zpkftransf`, `allpasslp2mb`, `iirlp2mb`

References

[1] Franchitti, J.C., “All-pass filter interpolation and frequency transformation problems,” MSc Thesis, Dept. of Electrical and Computer Engineering, University of Colorado, 1985.

[2] Feyh, G., J.C. Franchitti and C.T. Mullis, “All-pass filter interpolation and frequency transformation problem,” *Proceedings 20th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, California, pp. 164-168, November 1986.

[3] Mullis, C.T. and R.A. Roberts, *Digital Signal Processing*, section 6.7, Reading, Massachussetts, Addison-Wesley, 1987.

[4] Feyh, G., W.B. Jones and C.T. Mullis, *An extension of the Schur Algorithm for frequency transformations, Linear Circuits, Systems and Signal Processing: Theory and Application*, C. J. Byrnes et al Eds, Amsterdam: Elsevier, 1988.

zpklp2mbc

Purpose

Zero-pole-gain lowpass to complex M-band frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklpmbc(Z,P,K,Wo,Wt)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklpmbc(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the real lowpass prototype by applying an `Mth`-order real lowpass to complex multibandpass frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

This transformation effectively places one feature of an original filter, located at frequency W_o, at the required target frequency locations, W_{t1},...,W_{tM}.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature, for example, the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

This transformation can also be used for transforming other types of filters; e.g., to replicate notch filters and resonators at any required location.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z1,p1,k1] = zpklp2mbc(z, p, k, 0.5, [2 4 6 8]/10, 'pass');
[z2,p2,k2] = zpklp2mbc(z, p, k, 0.5, [2 4 6 8]/10, 'stop');
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k1*poly(z1), poly(p1), k2*poly(z2), poly(p2));
```

Arguments

`Z`
Zeros of the prototype lowpass filter

`P`
Poles of the prototype lowpass filter

`K`
Gain factor of the prototype lowpass filter

`Wo`
Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

`Wt`
Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

`Z2`
Zeros of the target filter

`P2`
Poles of the target filter

`K2`
Gain factor of the target filter

`AllpassNum`
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

See Also

`zpkftransf`, `allpasslp2mbc`, `iirlp2mbc`

zpklp2xc

Purpose

Zero-pole-gain lowpass to complex N-point frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2xc(Z,P,K,Wo,Wt)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2xc(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the real lowpass prototype by applying an `N`th-order real lowpass to complex multipoint frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

Parameter `N` also specifies the number of replicas of the prototype filter created around the unit circle after the transformation. This transformation effectively places `N` features of an original filter, located at frequencies W_{o1},...,W_{oN}, at the required target frequency locations, W_{t1},...,W_{tM}.

Relative positions of other features of an original filter are the same in the target filter for the Nyquist mobility and are reversed for the DC mobility. For the Nyquist mobility this means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation. For DC mobility feature F_2 will precede F_1 after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones. The only condition is that the features must be selected in such a way that when creating `N` bands around the unit circle, there will be no band overlap.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
```

```
z = roots(b);
p = roots(a);
k = b(1);
[z2,p2,k2] = zpklp2xc(z, p, k, [-0.5 0.5], [-0.25 0.25], 'pass');
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Arguments

`Z`
Zeros of the prototype lowpass filter

`P`
Poles of the prototype lowpass filter

`K`
Gain factor of the prototype lowpass filter

`Wo`
Frequency values to be transformed from the prototype filter. They should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

`Wt`
Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

`Z2`
Zeros of the target filter

`P2`
Poles of the target filter

`K2`
Gain factor of the target filter

`AllpassNum`
Numerator of the mapping filter

`AllpassDen`

Denominator of the mapping filter

See Also

`zpkftransf`, `allpasslp2xc`, `iirlp2xc`

Purpose

Zero-pole-gain lowpass to N-point frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2xn(Z,P,K,Wo,Wt)
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2xn(Z,P,K,Wo,Wt,Pass)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2xn(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the real lowpass prototype by applying an `N`th-order real lowpass to real multipoint frequency transformation, where `N` is the number of features being mapped. By default the DC feature is kept at its original location.

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2xn(Z,P,K,Wo,Wt,Pass)` allows you to specify an additional parameter, `Pass`, which chooses between using the "DC Mobility" and the "Nyquist Mobility". In the first case the Nyquist feature stays at its original location and the DC feature is free to move. In the second case the DC feature is kept at an original frequency and the Nyquist feature is allowed to move.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

Parameter `N` also specifies the number of replicas of the prototype filter created around the unit circle after the transformation. This transformation effectively places `N` features of an original filter, located at frequencies W_{o1},...,W_{oN}, at the required target frequency locations, W_{t1},...,W_{tM}.

Relative positions of other features of an original filter are the same in the target filter for the Nyquist mobility and are reversed for the DC mobility. For the Nyquist mobility this means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation. For DC mobility feature F_2 will precede F_1 after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones. The only condition is that the features must be

selected in such a way that when creating N bands around the unit circle, there will be no band overlap.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z2,p2,k2] = zpklp2xn(z, p, k, [-0.5 0.5], [-0.25 0.25], 'pass');
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Arguments

Z
Zeros of the prototype lowpass filter

P
Poles of the prototype lowpass filter

K
Gain factor of the prototype lowpass filter

Wo
Frequency value to be transformed from the prototype filter

Wt
Desired frequency location in the transformed target filter

Pass
Choice ('pass'/'stop') of passband/stopband at DC, pass being the default

Z2
Zeros of the target filter

P2
Poles of the target filter

`K2`
Gain factor of the target filter

`AllpassDen`
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`zpkftransf`, `allpasslp2xn`, `iirlp2xn`

References

[1] Cain, G.D. , A. Krukowski and I. Kale, “High Order Transformations for Flexible IIR Filter Design,” *VII European Signal Processing Conference (EUSIPCO'94)*, vol. 3, pp. 1582-1585, Edinburgh, United Kingdom, September 1994.

[2] 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.

Purpose

Zero-pole-gain complex bandpass frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpkrateup(Z,P,K,N)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpkrateup(Z,P,K,N)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter being transformed from any prototype by applying an `N`th-order rateup frequency transformation, where `N` is the upsample ratio. Transformation creates `N` equal replicas of the prototype filter frequency response.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The original lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
```

Upsample the prototype filter four times:

```
[z2,p2,k2] = zpkrateup(z, p, k, 4);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Arguments

`Z`
Zeros of the prototype lowpass filter

`P`
Poles of the prototype lowpass filter

`K`
Gain factor of the prototype lowpass filter

N
Integer upsampling ratio

Z2
Zeros of the target filter

P2
Poles of the target filter

K2
Gain factor of the target filter

AllpassNum
Numerator of the mapping filter

AllpassDen
Denominator of the mapping filter

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

See Also

zpkrateup, allpassrateup, iirrateup

Purpose

Zero-pole-gain real shift frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpkshift(Z,P,K,Wo,Wt)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpkshift(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the real lowpass prototype by applying a second-order real shift frequency mapping.

It also returns the numerator, `AllpassNum`, and the denominator of the allpass mapping filter, `AllpassDen`. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

This transformation places one selected feature of an original filter, located at frequency W_o, at the required target frequency location, W_t. This transformation implements the "DC Mobility," which means that the Nyquist feature stays at Nyquist, but the DC feature moves to a location dependent on the selection of W_o and W_t.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the real shift transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can change their position in a simple way without the need to design them again.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z2,p2,k2] = zpkshift(z, p, k, 0.5, 0.25);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Arguments

`Z`
Zeros of the prototype lowpass filter

`P`
Poles of the prototype lowpass filter

`K`
Gain factor of the prototype lowpass filter

`Wo`
Frequency value to be transformed from the prototype filter

`Wt`
Desired frequency location in the transformed target filter

`Z2`
Zeros of the target filter

`P2`
Poles of the target filter

K2

Gain factor of the target filter

`AllpassNum`
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`zpkftransf`, `allpassshift`, `iirshift`

Purpose

Zero-pole-gain complex shift frequency transformation

Syntax

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpkshiftc(Z,P,K,Wo,Wt)`

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpkshiftc(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the real lowpass prototype by applying a first-order complex frequency shift transformation. This transformation rotates all the features of an original filter by the same amount specified by the location of the selected feature of the prototype filter, originally at W_o, placed at W_t in the target filter.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and the gain factor, `K`.

`[Num,Den,AllpassNum,AllpassDen] = zpkshiftc(Z,P,K,0,0.5)` performs the Hilbert transformation, i.e. a 90 degree counterclockwise rotation of an original filter in the frequency domain.

`[Num,Den,AllpassNum,AllpassDen] = zpkshiftc(Z,P,K,0,-0.5)` performs the inverse Hilbert transformation, i.e. a 90 degree clockwise rotation of an original filter in the frequency domain.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
```

Example 1: Rotation by -0.25:

```
[z2,p2,k2] = zpkshiftc(z, p, k, 0.5, 0.25);
fvtool(b, a, k2*poly(z2), poly(p2));
```

Example 2: Hilbert transform:

```
[z2,p2,k2] = zpkshiftc(z, p, k, 0, 0.5);
fvtool(b, a, k2*poly(z2), poly(p2));
```

Example 3: Inverse Hilbert transform:

```
[z2,p2,k2] = zpkshiftc(z, p, k, 0, -0.5);
fvtool(b, a, k2*poly(z2), poly(p2));
```

Arguments

`Z`
Zeros of the prototype lowpass filter

`P`
Poles of the prototype lowpass filter

`K`
Gain factor of the prototype lowpass filter

`Wo`
Frequency value to be transformed from the prototype filter

`Wt`
Desired frequency location in the transformed target filter

`Z2`
Zeros of the target filter

`P2`
Poles of the target filter

`K2`
Gain factor of the target filter

`AllpassDen`
Numerator of the mapping filter

`AllpassDen`
Denominator of the mapping filter

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

See Also

`zpkftransf`, `allpassshiftc`, `iirshiftc`

References

[1] Oppenheim, A.V., R.W. Schafer and J.R. Buck, *Discrete-Time Signal Processing*, Prentice-Hall International Inc., 1989.

[2] Dutta-Roy, S.C. and B. Kumar, “On digital differentiators, Hilbert transformers, and half-band low-pass filters,” *IEEE Transactions on Education*, vol. 32, pp. 314-318, August 1989.

Purpose Compute a zero-pole plot for quantized filters

Syntax

```
zplane(Hq)
zplane(Hq,'plotoption')
zplane(Hq,'plotoption','plotoption2')
[zq,pq,kq] = zplane(Hq)
[zq,pq,kq,zr,pr,kr] = zplane(Hq)
```

Description This function displays the poles and zeros of quantized filters, as well as the poles and zeros of the associated unquantized reference filter.

`zplane(Hq)` plots the zeros and poles of a quantized filter `Hq` in the current figure window. The poles and zeros of the quantized and unquantized filters are plotted by default. The symbol `o` represents a zero of the unquantized reference filter, and the symbol `x` represents a pole of that filter. The symbols □ and `+` are used to plot the zeros and poles of the quantized filter `Hq`. The plot includes the unit circle for reference.

`zplane(Hq,'`*`plotoption`*`')` plots the poles and zeros associated with the quantized filter `Hq` according to one specified plot option. The string `'`*`plotoption`*`'` can be either of the following reference filter display options:

- `'`**`on`**`'` to display the poles and zeros of both the quantized filter and the associated reference filter (default)
- `'`**`off`**`'` to display the poles and zeros of only the quantized filter

`zplane(Hq,'`*`plotoption`*`','`*`plotoption2`*`')` plots the poles and zeros associated with the quantized filter `Hq` according to two specified plot options. The string `'`*`plotoption`*`'` can be selected from the reference filter display options listed in the previous syntax. The string `'`*`plotoption2`*`'` can be selected from the section-by-section plotting style options described below:

- `'`**`individual`**`'` to display the poles and zeros of each section of the filter in a separate figure window
- `'`**`overlay`**`'` to display the poles and zeros of all sections of the filter on the same plot
- `'`**`tile`**`'` to display the poles and zeros of each section of the filter in a separate plot in the same figure window

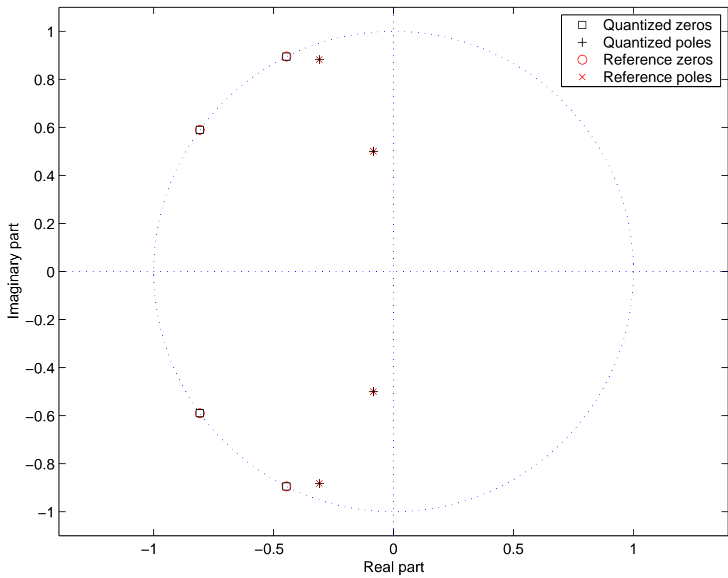
`[zq,pq,kq] = zplane(Hq)` returns the vectors of zeros `zq`, poles `pq`, and gains `kq`. If `Hq` has n sections, `zq`, `pq`, and `kq` are returned as 1-by-n cell arrays. If there are no zeros (or no poles), `zq` (or `pq`) is set to the empty matrix `[]`.

`[zq,pq,kq,zr,pr,kr] = zplane(Hq)` returns the vectors of zeros `zr`, poles `pr`, and gains `kr` of the reference filter associated with the quantized filter `Hq`, and returns the vectors of zeros `zq`, poles `pq`, and gains `kq` for the quantized filter `Hq`.

Examples

Create a quantized filter `Hq` from a fourth-order digital filter with cutoff frequency of 0.6. Scale the transfer function parameters to avoid overflows due to coefficient quantization. Plot the quantized and unquantized poles and zeros associated with this quantized filter.

```
[b,a] = ellip(4,.5,20,.6);
Hq = qfilt('df2',{b/2 a/2});
zplane(Hq);
```



See Also

`freqz`, `impz`

14

Bibliography

Advanced Filters (p. 14-2)	Suggested reading and sources for advanced filter design topics
Adaptive Filters (p. 14-2)	Suggested reading and sources for adaptive filters topics
Frequency Transformations (p. 14-3)	Suggested reading and sources for information about filter frequency transformations

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., *Mulitrate Digital Signal Processing*, John Wiley and Sons, 1994.

[4] Jackson, L., *Digital Filtering and Signal Processing with MATLAB Exercises*, Third edition, Kluwer Academic Publishers, 1996.

[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. Shafer, 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* `http://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., J. Sorensen, "Time-Area Efficient Multipier-Free Recursive Filter Architectures for FPGA Implementation," IEEE International Conference on Acoustics, Speech, and Signal Processing, 1996, pp. 3269-3272

Adaptive Filters

[11] Hayes, M.H., *Statistical Digital Signal Processing and Modeling*, John Wiley and Sons, 1996

[12] Haykin, S., *Adaptive Filter Theory*, Third Edition, Prentice-Hall, Inc., 1996.

Frequency Transformations

[13] Constantinides, A.G., “Spectral Transformations for Digital Filters,” *IEE Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

[14] 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.

[15] 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.

[16] 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.

Index

A

abbreviating property names 6-6
accessing properties 6-5
addition, format for
  quantized FFTs 12-55
  quantized filters 12-50
advanced FIR filter design 2-7
advanced IIR filter design 2-42
algorithm,`gremez` 2-7
`antisymmetricfir` 12-16
arithmetic
  quantized filtering, effects on 10-23

B

basic filter properties
  quantized filters 8-6, 9-6
  quantizers 7-4
bias 5-20
bibliography 14-2
binary
  coding 5-3
  data types 5-3
binary point 5-17
bits
  definition 5-16
  setting, quantized FFTs 9-8
  setting, quantized filters 8-11
brackets, indicating closed interval xxi

C

cell arrays
  indexing into cell arrays of cell arrays 6-15
  indexing into cell arrays of matrices 6-14
cell arrays, quantized filter coefficients 12-11
changing quantized filter properties in FDATool 11-11
coefficient overflow indicator 11-25
coefficient quantization, controlling 11-20
coefficient underflow indicator 11-25
`CoefficientFormat` property
  setting 12-4
command line help 6-12
constructing objects 6-3
context-sensitive help 11-49
controls, FDATool 11-5
convert structure dialog 11-17
converting filter structures in FDATool 11-16
copying objects 6-4
custom floating-point 5-22

D

data formats
  operands, quantized FFTs 12-53
  operands, quantized filters 12-38
  outputs, quantized FFTs 12-54
  outputs, quantized filters 12-39
  properties 13-337
  quantized FFTs, setting all 9-8
  quantized filters 5-5, 8-11, 9-8
  quantized filters, setting all 8-10
  setting 12-4
data types
  binary 5-3
  quantized filters 8-9, 9-7
denormalized numbers 5-24
designing advanced FIR filters 2-7
designing advanced IIR filters 2-42
`df1` 12-18
`df1t` 12-20

`df2` 12-22
`df2t` 12-24
digital filters
  fixed-point 5-3
  floating-point 5-4
digital frequency xxi
direct form I 12-18
  transposed 12-20
direct form II 12-22
  transposed 12-24
direct property referencing 13-333
dot notation 13-333
double-precision 5-21
DSP processors 5-4
dynamic range
  fixed-point 5-17
  floating-point 5-22

E

ellipses, in syntax xxi
entering transfer function scale values in FDATool 11-26
envelope delay. *See* group delay
equiripple filters 2-6
`errmean` 13-5
error, Lp norm 2-4
errors, quantization 2-64
`errpdf` 13-5
`errvar` 13-5
exceptional arithmetic 5-24
exponents 5-3
  length 5-19, 12-4
exporting filters 11-31
exporting quantized filters in FDATool 11-31

F

FDATool
  about importing and exporting filters 11-29
  apply option 11-5
  changing quantized filter properties 11-11
  convert structure dialog 11-17
  convert structure option 11-16
  converting filter structures 11-16
  entering transfer function scale values 11-26
  exporting quantized filters 11-31
  getting help 11-49
  import filter dialog 11-30
  importable filter structures 11-29
  importing filters 11-30
  quantized filter properties 11-7
  quantizer property lists 11-6
  quantizing filters 11-7
  quantizing reference filters 11-10
  scaling transfer function coefficients 11-24
  scaling transfer function coefficients manually 11-26
  set quantization mode 11-5
  set quantization parameters dialog 11-7
  setting properties 11-7
  to transform filters,transform filters in FDATool 11-40
  turn quantization on option 11-5
  user options 11-5
  using input/output scaling 11-26
  viewing filter structure schematics 11-17
fdatool
  frequency point to transform 11-38
  original filter type 11-35
  specify desired frequency location 11-39
  transformed filter type 11-39
`fdatool`
  about 11-2

about quantization mode 11-4
context-sensitive help 11-49
switching to quantization mode 11-4
FFTs
computing quantized 9-10
`filter` 13-141
filter banks
quantized 10-17
filter conversions 12-47
Filter Design and Analysis Tool. *See* `fdatool`
filter design GUI
context-sensitive help 11-49
help on 11-49
filter design methods
`firlpnorm` 2-5
`gremez` 2-7
`gremez` design examples 2-8
IIR filter design examples 2-43
`iirgrpdelay` 2-42
`iirlpnorm` 2-42
`iirlpnorm` design examples 2-45
`iirlpnormc` 2-42
`iirlpnormc` design examples 2-50
filter design, advanced FIR 2-7
filter design, advanced IIR 2-42
filter design, minimax 2-4
filter design, optimal 2-2
filter sections
specifying 12-47
filter structures 8-8
direct form FIR 12-26
direct form I 12-18
direct form I transposed 12-20
direct form II 12-22
direct form II transposed 12-24
direct form symmetric FIR 12-36
FIR transposed 12-27
lattice allpass 12-28
lattice AR 12-32
lattice ARMA 12-34
lattice coupled-allpass 12-42
lattice coupled-allpass power complementary 12-42
lattice MA minimum phase 12-33
lattice moving average maximum phase 12-29
state-space 12-35
filtering data
function for 13-141
logs of overflows 13-144
logs of underflows 13-144
obtaining states 13-144
filters
about equiripple 2-6
direct form 12-12
estimating frequency response with `nlm` 13-3
export to workspace 11-31
exporting as MAT-file 11-32
exporting as text file 11-32
exporting from FDATool 11-31
FIR 12-12
getting filter coefficients after exporting 11-32
importing and exporting 11-29
lattice 12-12
state-space 12-12
test if filter coefficients are real 13-2
testing for allpass structure 13-3
testing for FIR structure 13-3
testing for limitcycles in quantized 13-3
testing for linear phase sections 13-3
testing for maximum phase design 13-3
testing for minimum phase design 13-3
testing for purely real coefficients 13-3
testing for second-order sections 13-3
testing for stability 13-3

filters, importing into FDATool 11-30
filters, low-sensitivity 2-64
filters, robust 2-64
`FilterStructure` property 12-12
finite impulse response
  antisymmetric 12-16
  symmetric 12-36
`fir` 12-26
FIR filters 12-12
`firlpnorm` design method 2-5
`firt` 12-27
fixed-point 5-16
  data formats 5-5
  filters 5-4
  fraction length 5-3
  ranges 5-3
  sign bit 5-16
  word length 5-3
fixed-point numbers
  scaling 5-18
floating-point 5-19
  bias 5-20
  custom 5-22
  double precision 5-21
  dynamic range 5-22
  exponents 5-20
  filters 5-4
  fractions 5-20
  IEEE format 5-20
  mantissa 5-3
  precision 5-23
  ranges 5-3
  sign bits 5-20
  single precision 5-21
  word length 5-19
fractions 5-20
  determining length 5-3
  limitations on length 12-4
frequency
  digital xxi
  Nyquist xxi
frequency point to transform 11-38
frequency response 13-164
  noise loading method 5-13
frequency response plots 5-12
`freqz` 13-164
function for opening FDATool 11-4
functions, overloading 6-11

G
`get` 13-168
getting filter coefficients after exporting 11-32
getting properties 6-8
  command for 13-168
getting started 1-15
getting started example 1-15
`gremez` 2-7
`gremez` algorithm 2-7
`gremez` design examples 2-8
group delay, about 2-56
group delay, prescribed 2-42

H
help
  command line 6-12

I
IEEE
  format 5-20
  nonstandard format 5-22
`iirgrpdelay` 2-42

iirgrpdelay design examples 2-56
iirlpnorm 2-42
iirlpnorm design examples 2-43
iirlpnorm filter design examples 2-43
iirlpnormc 2-42
iirlpnormc design examples 2-43
import filter dialog in FDATool 11-30
import filter dialog options 11-30
  frequency units 11-30
  quantized filter 11-30
import/export filters in FDATool 11-29
importing filters 11-30
importing quantized filters in FDATool 11-30
impulse response 13-232
impulse response plots 5-11
impz 13-232
indexing
  cell arrays of cell arrays 6-15
  cell arrays of matrices 6-14
  vectors xxi
indicator, overflow 11-25
indicator, underflow 11-25
InputFormat property 12-4
interval notation xxi
inverse FFTs
  computing quantized 9-10
isallpass 13-3
isfir 13-3
isfixed,quantizers
  testing for fixed point 13-5
isfloat,quantizers
  testing for floating point 13-5
islinphase 13-3
ismaxphase 13-3
isminphase 13-3
isnone,quantizers
  testing for none 13-5
isreal 13-2, 13-264
issos 13-3
isstable 13-3

L

latcallpass 12-28
latcmax 12-29
lattice filters
  allpass 12-28
  AR 12-32
  ARMA 12-34
  autoregressive 12-32
  coupled-allpass 12-30
  coupled-allpass power complementary 12-31
  MA 12-33
  moving average maximum phase 12-29
  moving average minimum phase 12-33
latticear 12-32
latticearma 12-34
latticeca 12-28, 12-29, 12-30
latticecapc 12-31
latticema 12-33
leading denominator coefficient not equal to 1, about 12-14
least significant bit 5-17
limit cycles in quantized filters 5-14
limitcycle 13-3
low-sensitivity filters 2-64
Lp norm 2-4
LSB 5-17

M

mantissa 5-3
minimax filter designs 2-4
Mode property 12-5

most significant bit 5-17
MSB 5-17
multiple sections
  specifying 12-47
`MultiplicandFormat` property
  quantized FFTs 12-53
  quantized filter 12-38

N

new users, tips for xvi
`nlm` 5-13, 13-3
noise loading method 5-13, 13-3
nonstandard IEEE format 5-22
`NOperations` property 12-6
normalize 5-18
`normalize` 13-279
normalize coefficients 11-21
normalizing quantized filters 8-12
`NumberOfSections` property
  quantized filters 12-38
`NumberOfStages` property
  quantized FFTs 12-53
`NUnderflows` property 12-7
Nyquist frequency xxi

O

object properties 1-13
objects
  constructing 6-3
  copying 6-4
objects in this toolbox 1-13
opening FDATool, function for 11-4
optimal filter design
  problem statement 2-2
  solutions 2-5
  theory 2-2
options, FDATool 11-5
original filter type 11-35
`OutputFormat` property
  quantized FFTs 12-54
  quantized filters 12-38, 12-39
  setting 12-4
overflow 5-23
overflow indicator 11-25
overflow mode property
  `saturate` 11-10
  `wrap` 11-10
overflow, checking for 11-25
`OverflowMode` property 12-6
overflows
  addressing, function for 13-279
overloading 6-11

P

parentheses, indicating open interval xxi
Parks-McClellan method 2-6
plots
  frequency response 5-12
  impulse response 5-11
  impulse response, command for 13-232
  noise loading method 5-13
  pole/zero 5-10
  zero-pole, command for 13-389
pole/zero plots 5-10
pole-zero plots 13-389
precision
  fixed-point 5-17
  floating-point 5-23
prescribed group delay 2-42
properties 1-13
  abbreviating names 6-6

accessing, command for 13-168
data formats
quantized filters 8-11, 9-8
setting 13-337
FilterStructure 12-12
Mode 12-5
MultiplicandFormat, quantized FFT 12-53
MultiplicandFormat, quantized filter 12-38
NumberOfSections, quantized filters 12-38
NumberOfStages, quantized FFTs 12-53
OutputFormat, quantized FFTs 12-54
OutputFormat, quantized filters 12-39
OverflowMode 12-6
QuantizedCoefficients 12-40
Radix 12-54
ReferenceCoefficients 8-7, 12-40
referencing directly 6-9
retrieving 6-5
function for 6-8
retrieving by direct property referencing 6-9
RoundMode 12-8
ScaleValues 12-48
setting 6-5
setting, function for 13-330
StatesPerSection 12-50
SumFormat, quantized FFTs 12-55
SumFormat, quantized filters 12-50
property values
abbreviating 6-8
quantized FFTs 9-6
quantized filters 8-6
quantizers 7-4

Q

QFFT objects 9-2
qfilt 13-299
Qfilt objects 1-13
See also quantized filters
quantization
precision, quantized FFTs 9-8
precision, quantized filters 8-11
quantization errors 2-64
quantization level 5-24
quantization mode in FDATool 11-4
quantization optimization
controlling coefficient quantization 11-20
denominators 11-21
normalize coeffients 11-21
numerators 11-21
quantization, errors during 2-64
quantized FFT properties
CoefficientFormat 12-52
InputFormat 12-52
MultiplicandFormat 12-53
NumberOfStages 12-53
OutputFormat 12-54
ProductFormat 12-54
Radix 12-54
ScaleValues 12-54
SumFormat 12-55
quantized FFTs 9-2
addition 12-55
basic properties 9-6
computing 9-10
constructing 9-3
data formats 9-7
input formats 12-52
multiplicand formats 12-53
output formats 12-54
product formats 12-54
properties 12-52
property values 9-6
scaling 12-54

stages, number of 12-53
quantized filter formats
inputs 12-38
operands 12-38
outputs 12-39
products 12-40
sums 12-50, 12-55
quantized filter properties
`CoefficientFormat` 12-11
`FilterStructure` 12-12
`InputFormat` 12-38
`NumberOfSections` 12-38
`OperandFormat` 12-38
`OutputFormat` 12-39
`ProductFormat` 12-40
`QuantizedCoefficients` 12-40
`ReferenceCoefficients` 12-40
setting 6-9
setting, command for 13-330
quantized filter properties, changing in FDATool 11-11
quantized filters
accessing properties 13-168
addition 12-50
analysis with 10-1
applications 10-1
architecture 12-12
arithmetic effects 10-23
basic properties 8-6
cascaded sections 12-44
coefficients, accessing for multiple sections 6-15
coefficients, accessing for single section 6-14
coefficients, overflows 13-279
coefficients, quantized 12-40
coefficients, reference 12-40
constructing 8-3
function for 13-299
data formats 8-9, 8-11, 9-8
data formats, setting all 8-10, 9-8
defining 6-3
direct form FIR 12-26
direct form FIR transposed 12-27
direct form symmetric FIR 12-36
examples 6-14
exponent length 12-4
filter banks 10-17
filter types 5-7
filtering data 8-14, 13-141
finite impulse response 12-26, 12-27
floating point 12-4
fraction length 12-4
frequency response 13-164
noise loading method 5-13
getting properties 6-8
impulse response 13-232
lattice allpass 12-28
lattice AR 12-32
lattice ARMA 12-34
lattice coupled-allpass 12-28, 12-30
lattice coupled-allpass power complementary 12-31
lattice MA maximum phase 12-29
lattice MA minimum phase 12-33
limit cycles 5-14
multiple sections, specifying coefficients 12-47
table 12-44
normalizing 13-279
objects 6-3
overflow handling 12-6
overflows, logging 8-14
precision, setting 13-337
property values 8-6
Qfilt objects 1-13

real coefficients 13-264
reference coefficients 8-7
reference filter 12-40
rounding, property for 12-8
scaling 12-48
second-order sections 8-4
sections, number of 12-38
setting data formats 13-337
specifying 12-40
state vectors 13-144
states 12-50
structures 12-12
symmetric FIR 12-16
topology 8-8
word length 12-4
zero-pole plots 13-389
quantized filters properties
getting 6-9
ScaleValues 12-48
specifying, command for 13-299
StatesPerSection 12-50
SumFormat 12-50
quantized inverse FFTs
computing 9-10
QuantizedCoefficients property 12-40
quantizers
calculating pdf 13-5
constructing 7-3
construction
shortcuts 7-5
data types
property for 12-5
properties
Format 12-3
Max 12-5
Min 12-5
Mode 12-5
NOperations 12-6
NOverflows 12-6
NUnderflows 12-7
OverflowMode 12-7
property names, leaving out 7-5
RoundMode 12-8
settable 7-4
property values 7-4
testing accuracy 13-5
testing error variance 13-5
unit 7-2
unity 7-3
quantizing filters in FDATool 11-10

R

Radix 12-54
radix point 5-3
interpretation 5-17
range
fixed-point 5-17
floating-point 5-22
range notation xxi
reference coefficients
specifying 12-40
reference filters
quantized filters, specifying from 8-4
specifying 8-7
ReferenceCoefficients property 12-40
Remez exchange algorithm 2-6
robust filters 2-64
rounding
property for 12-8
RoundMode property 12-8

S

`saturate` property value 11-10
`ScaleValues` property 12-48
  interpreting 12-49
scaling
  2 norm 2-4
  implementing for quantized filters 12-49
  infinity norm 2-4
  `Lp` norm 2-4
  quantized filters 12-48
scientific notation 5-19
second-order sections
  normalizing 12-48
`set` 13-330
set quantization parameters dialog 11-7
`setbits` 13-337
setting filter properties in FDATool 11-7
setting properties 6-5
  `set` function 13-330
  dot notation 13-333
sign bits 5-20
single precision 5-21
solution, minimax 2-4
specify desired frequency location 11-39
starting FDATool 11-4
state vectors 13-144
state-space filters 12-35
`StatesPerSection` property 12-50
structure-like referencing 6-9
`SumFormat` property
  quantized FFTs 12-55
  quantized filters 12-50
sums, data format for
  quantized FFTs 12-55
  quantized filters 12-50
`symmetricfir` 12-36
syntax, ellipses (. . .) xxi

T

toolbox
  getting started 1-15
topology 8-8
transform filter
  frequency point to transform 11-38
  original filter type 11-35
  specify desired frequency location 11-39
  transformed filter type 11-39
transformed filter type 11-39
twiddle factors 12-52
two’s complement arithmetic 5-16
typographical conventions (table) xxii

U

underflow indicator 11-25
underflow, checking for 11-25
underflows 5-23
unit quantizers 7-2
unity quantizers 7-3

V

vectors, indexing of xxi

W

word length
  fixed-point 5-3
  floating-point 5-19
  limitations 12-4
  setting 12-4
    all formats 13-337
`wrap` property value 11-10

Z

zero-pole plots 13-389
`zplane` 13-389
    plotting options 13-389