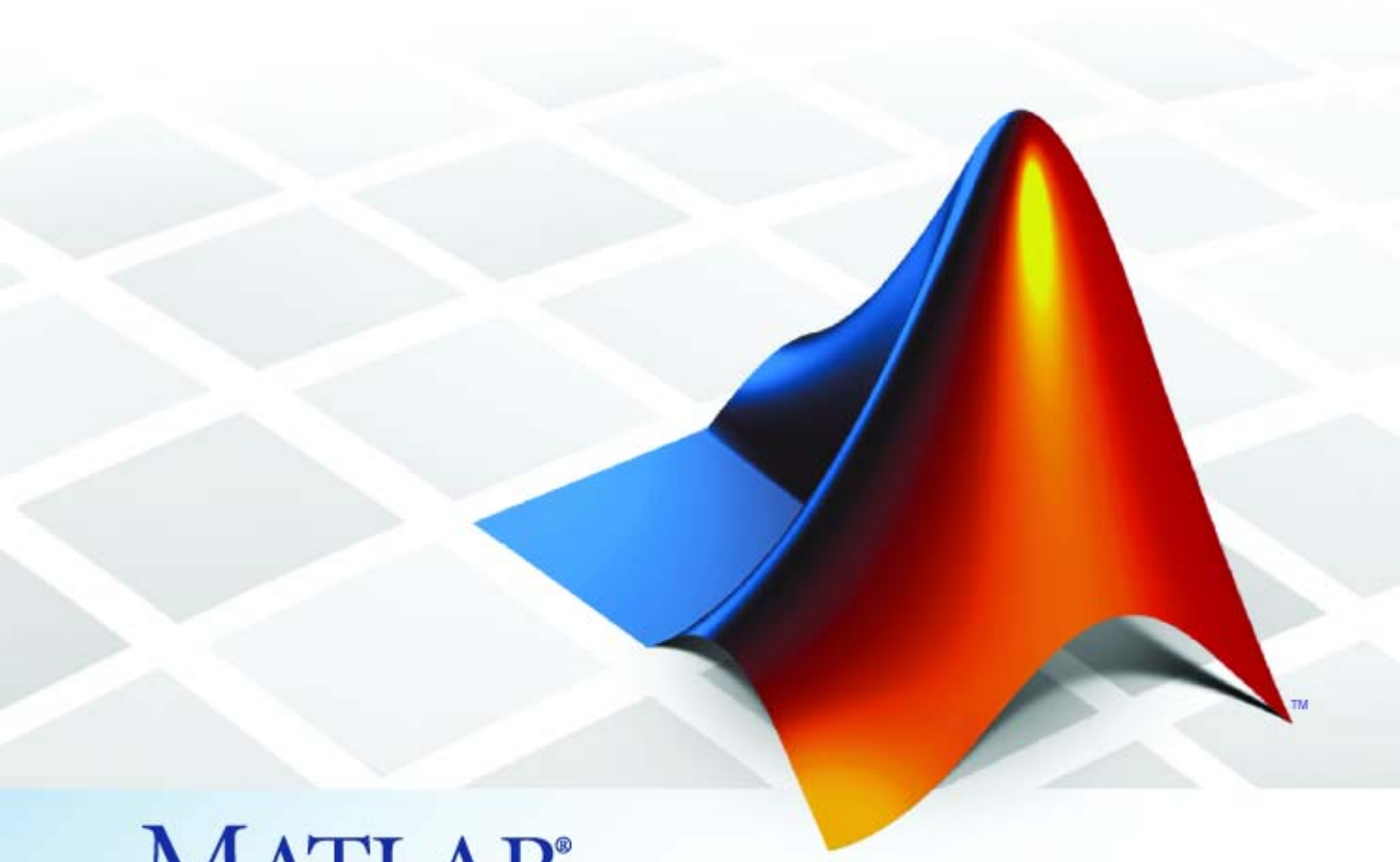
Signal Processing Toolbox™ 6

User's Guide



MATLAB®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	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.
3 Apple Hill Drive
Natick, MA 01760-2098

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

Signal Processing Toolbox™ User's Guide

Trademarks

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

Patents

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

Revision History

1988	First printing	New
November 1997	Second printing	Revised
January 1998	Third printing	Revised
September 2000	Fourth printing	Revised for Version 5.0 (Release 12)
July 2002	Fifth printing	Revised for Version 6.0 (Release 13)
December 2002	Online only	Revised for Version 6.1 (Release 13+)
June 2004	Online only	Revised for Version 6.2 (Release 14)
October 2004	Online only	Revised for Version 6.2.1 (Release 14SP1)
March 2005	Online only	Revised for Version 6.2.1 (Release 14SP2)
September 2005	Online only	Revised for Version 6.4 (Release 14SP3)
March 2006	Sixth printing	Revised for Version 6.5 (Release 2006a)
September 2006	Online only	Revised for Version 6.6 (Release 2006b)
March 2007	Online only	Revised for Version 6.7 (Release 2007a)
September 2007	Online only	Revised for Version 6.8 (Release 2007b)
March 2008	Online only	Revised for Version 6.9 (Release 2008a)
October 2008	Online only	Revised for Version 6.10 (Release 2008b)
March 2009	Online only	Revised for Version 6.11 (Release 2009a)

Contents

1 Filtering, Linear Systems and Transforms Overview

2 Filter Design and Implementation

3 Designing a Filter in Fdesign — Process Overview

4 Designing a Filter in the Filterbuilder GUI

5 FDATool: A Filter Design and Analysis GUI

6 Statistical Signal Processing

7 Special Topics

8 SPTool: A Signal Processing GUI Suite

9 Function Reference

Index

1

Filtering, Linear Systems and Transforms Overview

Filter Implementation and Analysis

In this section...

Filtering Overview

This section describes how to filter discrete signals using the MATLAB® `filter` function and other Signal Processing Toolbox™ functions. It also discusses how to use the toolbox functions to analyze filter characteristics, including impulse response, magnitude and phase response, group delay, and zero-pole locations.

Convolution and Filtering

The mathematical foundation of filtering is convolution. The MATLAB `conv` function performs standard one-dimensional convolution, convolving one vector with another:

```
conv([1 1 1],[1 1 1])
ans =
     1     2     3     2     1
```

Note Convolve rectangular matrices for two-dimensional signal processing using the `conv2` function.

A digital filter's output $y(k)$ is related to its input $x(k)$ by convolution with its impulse response $h(k)$.

$$y(k) = h(k) * x(k) = \sum_{l=-\infty}^{\infty} h(k-l)x(l)$$

If a digital filter's impulse response $h(k)$ is finite length, and the input $x(k)$ is also finite length, you can implement the filter using `conv`. Store $x(k)$ in a vector `x`, $h(k)$ in a vector `h`, and convolve the two:

```
x = randn(5,1);     % A random vector of length 5
h = [1 1 1 1]/4;    % Length 4 averaging filter
y = conv(h,x);
```

Filters and Transfer Functions

In general, the z-transform $Y(z)$ of a digital filter's output $y(n)$ is related to the z-transform $X(z)$ of the input by

$$Y(z) = H(z)X(z) = \frac{b(1) + b(2)z^{-1} + \ldots + b(n+1)z^{-n}}{a(1) + a(2)z^{-1} + \ldots + a(m+1)z^{-m}} X(z)$$

where $H(z)$ is the filter's *transfer function*. Here, the constants $b(i)$ and $a(i)$ are the filter coefficients and the order of the filter is the maximum of n and m.

Note The filter coefficients start with subscript 1, rather than 0. This reflects the standard indexing scheme used for MATLAB vectors.

MATLAB filter functions store the coefficients in two vectors, one for the numerator and one for the denominator. By convention, it uses row vectors for filter coefficients.

Filter Coefficients and Filter Names

Many standard names for filters reflect the number of `a` and `b` coefficients present:

- When `n` = 0 (that is, `b` is a scalar), the filter is an Infinite Impulse Response (IIR), all-pole, recursive, or autoregressive (AR) filter.
- When `m` = 0 (that is, `a` is a scalar), the filter is a Finite Impulse Response (FIR), all-zero, nonrecursive, or moving-average (MA) filter.
- If both `n` and `m` are greater than zero, the filter is an IIR, pole-zero, recursive, or autoregressive moving-average (ARMA) filter.

The acronyms AR, MA, and ARMA are usually applied to filters associated with filtered stochastic processes.

Filtering with the filter Function

It is simple to work back to a difference equation from the z-transform relation shown earlier. Assume that $a(1) = 1$. Move the denominator to the left-hand side and take the inverse z-transform.

$$y(k) + a_2 y(k-1) + \cdots + a_{m+1} y(k-m) = b_1 x(k) + b_2 x(k-1) + \cdots + b_{n+1} x(k-m)$$

In terms of current and past inputs, and past outputs, $y(n)$ is

$$y(k) = b_1 x(k) + b_2 x(k-1) + \cdots + b_{n+1} x(k-n) - a_2 y(k-1) - \cdots - a_{m+1} y(k-n)$$

This is the standard time-domain representation of a digital filter, computed starting with $y(1)$ and assuming zero initial conditions. This representation's progression is

$$\begin{aligned} y(1) &= b_1 x(1) \\ y(2) &= b_1 x(2) + b_2 x(1) - a_2 y(1) \\ y(3) &= b_1 x(3) + b_2 x(2) + b_3 x(1) - a_2 y(2) - a_3 y(1) \\ \vdots &= \vdots \end{aligned}$$

A filter in this form is easy to implement with the `filter` function. For example, a simple single-pole filter (lowpass) is

```
b = 1;          % Numerator
a = [1 -0.9];   % Denominator
```

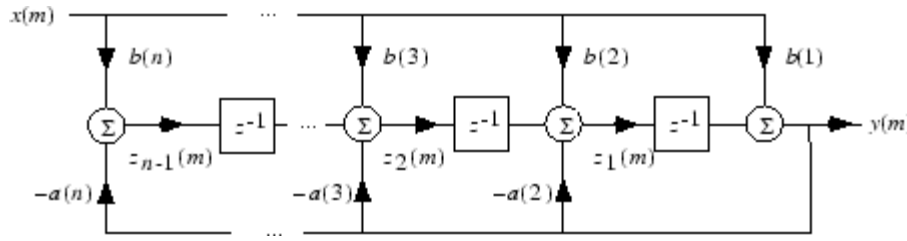
where the vectors `b` and `a` represent the coefficients of a filter in transfer function form. To apply this filter to your data, use

```
y = filter(b,a,x);
```

`filter` gives you as many output samples as there are input samples, that is, the length of `y` is the same as the length of `x`. If the first element of `a` is not 1, `filter` divides the coefficients by `a(1)` before implementing the difference equation.

The filter Function

`filter` is implemented as the transposed direct-form II structure, where n-1 is the filter order. This is a canonical form that has the minimum number of delay elements.



At sample m, `filter` computes the difference equations

$$
\begin{aligned}
y(m) &= b(1)x(m) + z_1(m-1) \\
z_1(m) &= b(2)x(m) + z_2(m-1) - a(2)y(m) \\
\vdots &= \vdots \\
z_{n-2}(m) &= b(n-1)x(m) + z_{n-1}(m-1) - a(n-1)y(m) \\
z_{n-1}(m) &= b(n)x(m) - a(n)y(m)
\end{aligned}
$$

In its most basic form, `filter` initializes the delay outputs $z_i(1)$, i = 1, ..., n-1 to 0. This is equivalent to assuming both past inputs and outputs are zero. Set the initial delay outputs using a fourth input parameter to `filter`, or access the final delay outputs using a second output parameter:

```
[y,zf] = filter(b,a,x,zi)
```

Access to initial and final conditions is useful for filtering data in sections, especially if memory limitations are a consideration. Suppose you have collected data in two segments of 5000 points each:

```
x1 = randn(5000,1);  % Generate two random data sequences.
x2 = randn(5000,1);
```

Perhaps the first sequence, `x1`, corresponds to the first 10 minutes of data and the second, `x2`, to an additional 10 minutes. The whole sequence is

`x = [x1;x2]`. If there is not sufficient memory to hold the combined sequence, filter the subsequences `x1` and `x2` one at a time. To ensure continuity of the filtered sequences, use the final conditions from `x1` as initial conditions to filter `x2`:

```
[y1,zf] = filter(b,a,x1);
y2 = filter(b,a,x2,zf);
```

The `filtic` function generates initial conditions for `filter`. `filtic` computes the delay vector to make the behavior of the filter reflect past inputs and outputs that you specify. To obtain the same output delay values `zf` as above using `filtic`, use

```
zf = filtic(b,a,flipud(y1),flipud(x1));
```

This can be useful when filtering short data sequences, as appropriate initial conditions help reduce transient startup effects.

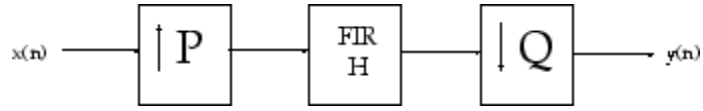
Other Functions for Filtering

In this section...

Multirate Filter Bank Implementation

The `upfirdn` function alters the sampling rate of a signal by an integer ratio P/Q. It computes the result of a cascade of three systems that performs the following tasks:

- Upsampling (zero insertion) by integer factor `p`
- Filtering by FIR filter `h`
- Downsampling by integer factor `q`



For example, to change the sample rate of a signal from 44.1 kHz to 48 kHz, we first find the smallest integer conversion ratio `p/q`. Set

```
d = gcd(48000,44100);
p = 48000/d;
q = 44100/d;
```

In this example, `p = 160` and `q = 147`. Sample rate conversion is then accomplished by typing

```
y = upfirdn(x,h,p,q)
```

This cascade of operations is implemented in an efficient manner using polyphase filtering techniques, and it is a central concept of multirate filtering. Note that the quality of the resampling result relies on the quality of the FIR filter `h`.

Filter banks may be implemented using `upfirdn` by allowing the filter `h` to be a matrix, with one FIR filter per column. A signal vector is passed independently through each FIR filter, resulting in a matrix of output signals.

Other functions that perform multirate filtering (with fixed filter) include `resample`, `interp`, and `decimate`.

Anti-Causal, Zero-Phase Filter Implementation

In the case of FIR filters, it is possible to design linear phase filters that, when applied to data (using `filter` or `conv`), simply delay the output by a fixed number of samples. For IIR filters, however, the phase distortion is usually highly nonlinear. The `filtfilt` function uses the information in the signal at points before and after the current point, in essence "looking into the future," to eliminate phase distortion.

To see how `filtfilt` does this, recall that if the z-transform of a real sequence $x(n)$ is $X(z)$, the z-transform of the time reversed sequence $x(n)$ is $X(1/z)$. Consider the processing scheme.

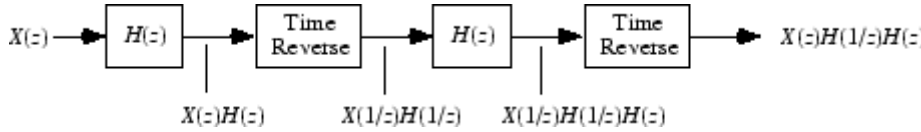


Image of Anti Causal Zero Phase Filter

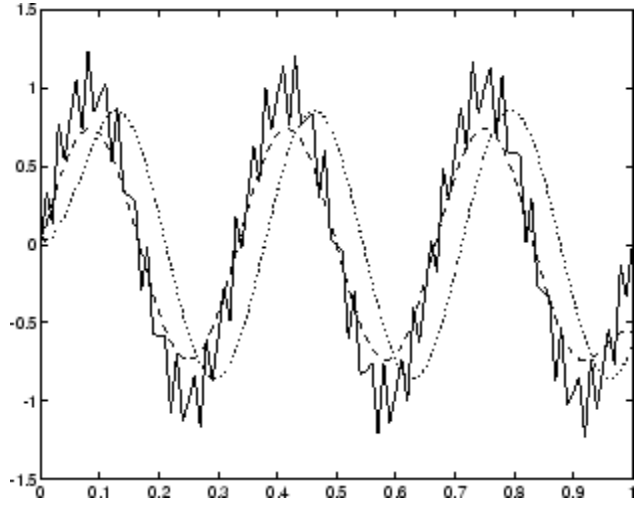
When $|z| = 1$, that is $z = e^{j\omega}$, the output reduces to $X(e^{j\omega})|H(e^{j\omega})|^2$. Given all the samples of the sequence $x(n)$, a doubly filtered version of x that has zero-phase distortion is possible.

For example, a 1-second duration signal sampled at 100 Hz, composed of two sinusoidal components at 3 Hz and 40 Hz, is

```
fs = 100;
t = 0:1/fs:1;
x = sin(2*pi*t*3)+.25*sin(2*pi*t*40);
```

Now create a 10-point averaging FIR filter, and filter `x` using both `filter` and `filtfilt` for comparison:

```
b = ones(1,10)/10;              % 10 point averaging filter
y = filtfilt(b,1,x);            % Noncausal filtering
yy = filter(b,1,x);             % Normal filtering
plot(t,x,t,y,'--',t,yy,':')
```



Both filtered versions eliminate the 40 Hz sinusoid evident in the original, solid line. The plot also shows how `filter` and `filtfilt` differ; the dashed (`filtfilt`) line is in phase with the original 3 Hz sinusoid, while the dotted (`filter`) line is delayed by about five samples. Also, the amplitude of the dashed line is smaller due to the magnitude squared effects of `filtfilt`.

`filtfilt` reduces filter startup transients by carefully choosing initial conditions, and by prepending onto the input sequence a short, reflected piece of the input sequence. For best results, make sure the sequence you are filtering has length at least three times the filter order and tapers to zero on both edges.

Frequency Domain Filter Implementation

Duality between the time domain and the frequency domain makes it possible to perform any operation in either domain. Usually one domain or the other is more convenient for a particular operation, but you can always accomplish a given operation in either domain.

To implement general IIR filtering in the frequency domain, multiply the discrete Fourier transform (DFT) of the input sequence with the quotient of the DFT of the filter:

```
n = length(x);
y = ifft(fft(x).*fft(b,n)./fft(a,n));
```

This computes results that are identical to `filter`, but with different startup transients (edge effects). For long sequences, this computation is very inefficient because of the large zero-padded FFT operations on the filter coefficients, and because the FFT algorithm becomes less efficient as the number of points `n` increases.

For FIR filters, however, it is possible to break longer sequences into shorter, computationally efficient FFT lengths. The function

```
y = fftfilt(b,x)
```

uses the overlap add method to filter a long sequence with multiple medium-length FFTs. Its output is equivalent to `filter(b,1,x)`.

Impulse Response

The impulse response of a digital filter is the output arising from the unit impulse input sequence defined as

$$x(n) = \begin{cases} 1, & n = 1 \\ 0, & n \neq 1 \end{cases}$$

You can generate an impulse sequence a number of ways; one straightforward way is

```
imp = [1; zeros(49,1)];
```

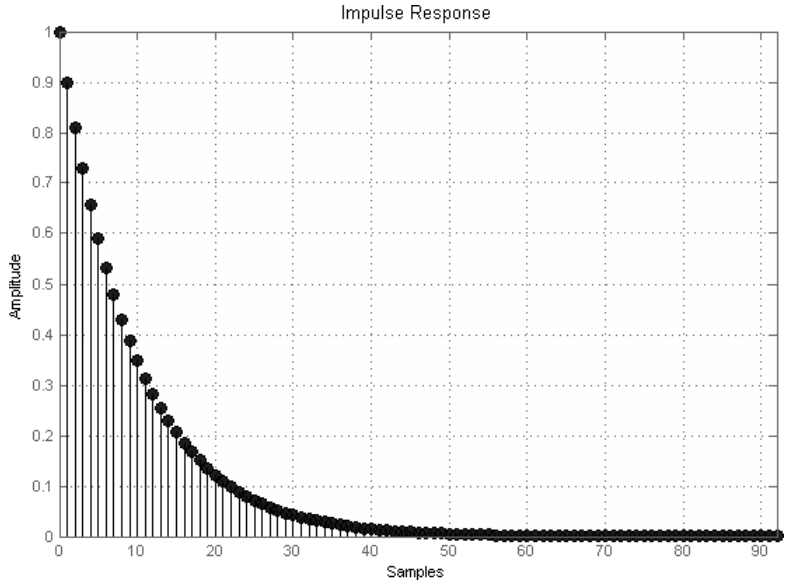
The impulse response of the simple filter `b = 1` and `a = [1 -0.9]` is

```
h = filter(b,a,imp);
```

A simple way to display the impulse response is with the Filter Visualization Tool (`fvtool`):

```
fvtool(b,a)
```

Then click the **Impulse Response** button  on the toolbar or select **Analysis > Impulse Response**. This plot shows the exponential decay `h(n) = 0.9n` of the single pole system:


Impulse Response
Amplitude
Samples
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
10
20
30
40
50
60
70
80
90

Frequency Response

In this section...

Digital Domain

`freqz` uses an FFT-based algorithm to calculate the *z*-transform frequency response of a digital filter. Specifically, the statement

```
[h,w] = freqz(b,a,p)
```

returns the *p*-point complex frequency response, $H(e^{j\omega})$, of the digital filter.

$$H(e^{jw}) = \frac{b(1) + b(2)e^{-jw} + \ldots + b(n+1)e^{-jw(n)}}{a(1) + a(2)e^{-jw} + \ldots + a(m+1)e^{-jw(m)}}$$

In its simplest form, `freqz` accepts the filter coefficient vectors `b` and `a`, and an integer `p` specifying the number of points at which to calculate the frequency response. `freqz` returns the complex frequency response in vector `h`, and the actual frequency points in vector `w` in rad/s.

`freqz` can accept other parameters, such as a sampling frequency or a vector of arbitrary frequency points. The example below finds the 256-point frequency response for a 12th-order Chebyshev Type I filter. The call to `freqz` specifies a sampling frequency `fs` of 1000 Hz:

```
[b,a] = cheby1(12,0.5,200/500);
[h,f] = freqz(b,a,256,1000);
```

Because the parameter list includes a sampling frequency, `freqz` returns a vector `f` that contains the 256 frequency points between 0 and `fs/2` used in the frequency response calculation.

Note This toolbox uses the convention that unit frequency is the Nyquist frequency, defined as half the sampling frequency. The cutoff frequency parameter for all basic filter design functions is normalized by the Nyquist frequency. For a system with a 1000 Hz sampling frequency, for example, 300 Hz is 300/500 = 0.6. To convert normalized frequency to angular frequency around the unit circle, multiply by π. To convert normalized frequency back to hertz, multiply by half the sample frequency.

If you call `freqz` with no output arguments, it plots both magnitude versus frequency and phase versus frequency. For example, a ninth-order Butterworth lowpass filter with a cutoff frequency of 400 Hz, based on a 2000 Hz sampling frequency, is

```
[b,a] = butter(9,400/1000);
```

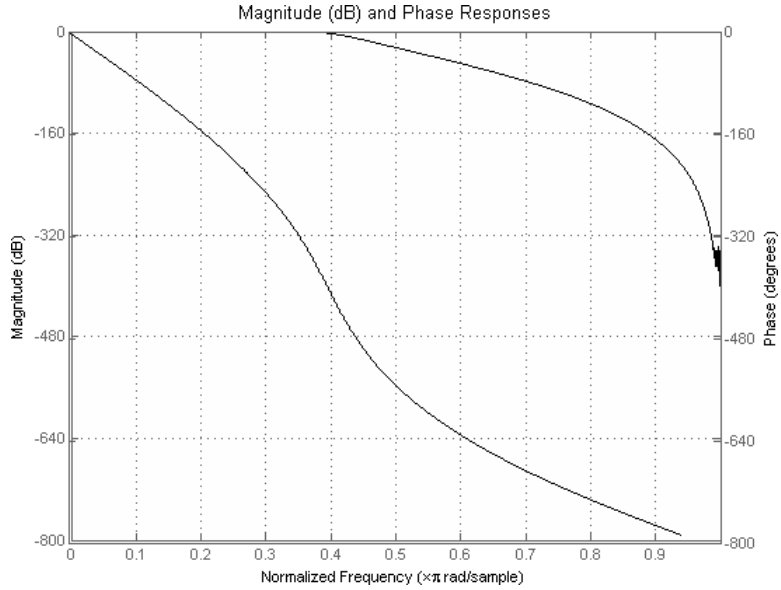
To calculate the 256-point complex frequency response for this filter, and plot the magnitude and phase with `freqz`, use

```
freqz(b,a,256,2000)
```

or to display the magnitude and phase responses in `fvtool`, which provides additional analysis tools, use

```
fvtool(b,a)
```

and click the **Magnitude and Phase Response** button  on the toolbar or select **Analysis > Magnitude and Phase Response**.



`freqz` can also accept a vector of arbitrary frequency points for use in the frequency response calculation. For example,

```
w = linspace(0,pi);
h = freqz(b,a,w);
```

calculates the complex frequency response at the frequency points in `w` for the filter defined by vectors `b` and `a`. The frequency points can range from 0 to 2π. To specify a frequency vector that ranges from zero to your sampling frequency, include both the frequency vector and the sampling frequency value in the parameter list.

Analog Domain

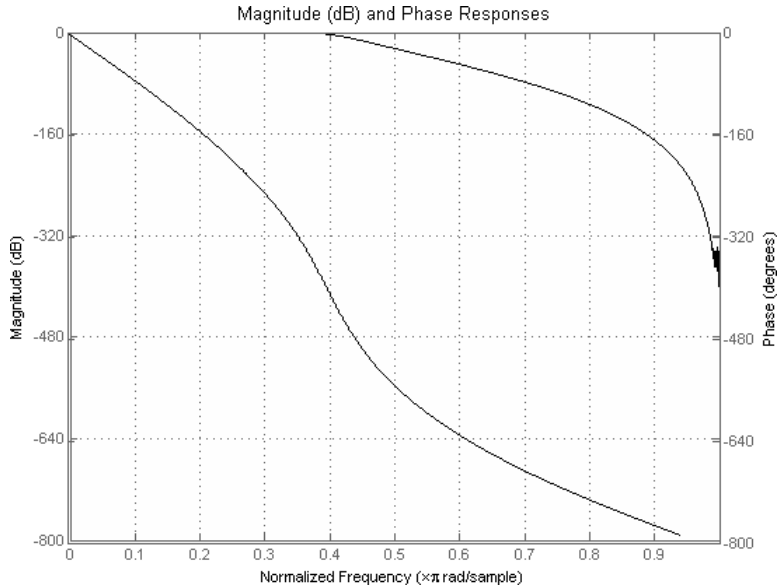
`freqs` evaluates frequency response for an analog filter defined by two input coefficient vectors, `b` and `a`. Its operation is similar to that of `freqz`; you can specify a number of frequency points to use, supply a vector of arbitrary frequency points, and plot the magnitude and phase response of the filter.

Magnitude and Phase

MATLAB functions are available to extract magnitude and phase from a frequency response vector `h`. The function `abs` returns the magnitude of the response; `angle` returns the phase angle in radians. To extract the magnitude and phase of a Butterworth filter:

```
[b,a] = butter(9,400/1000);
fvtool(b,a)
```

and click the **Magnitude and Phase Response** button on the toolbar or select **Analysis > Magnitude and Phase Response** to display the plot.

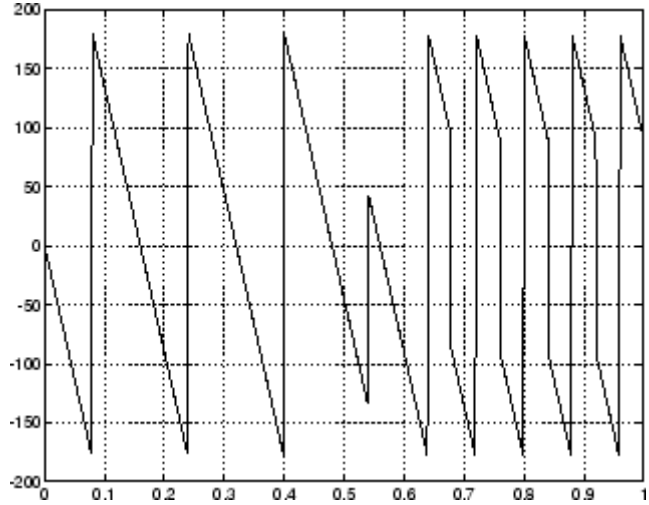


The `unwrap` function is also useful in frequency analysis. `unwrap` unwraps the phase to make it continuous across 360º phase discontinuities by adding multiples of ±360°, as needed. To see how `unwrap` is useful, design a 25th-order lowpass FIR filter:

```
h = fir1(25,0.4);
```

Obtain the filter's frequency response with `freqz`, and plot the phase in degrees:

```
[H,f] = freqz(h,1,512,2);
plot(f,angle(H)*180/pi); grid
```



It is difficult to distinguish the 360° jumps (an artifact of the arctangent function inside `angle`) from the 180° jumps that signify zeros in the frequency response.

`unwrap` eliminates the 360° jumps:

```
plot(f,unwrap(angle(H))*180/pi);
```

or you can use `phasez` to see the unwrapped phase.

Delay

The *group delay* of a filter is a measure of the average time delay of the filter as a function of frequency. It is defined as the negative first derivative of a filter's phase response. If the complex frequency response of a filter is $H(e^{j\omega})$, then the group delay is

$$\tau_g(\omega) = -\frac{d\theta(\omega)}{d\omega}$$

where θ is the phase angle of $H(e^{j\omega})$. Compute group delay with

```
[gd,w] = grpdelay(b,a,n)
```

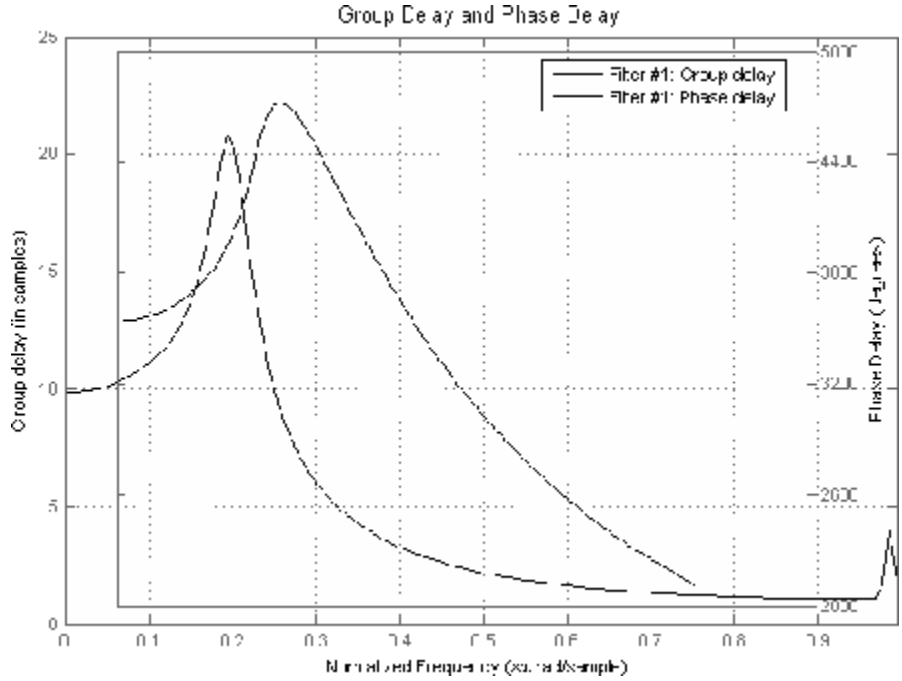
which returns the `n`-point group delay, $\tau_g(\omega)$, of the digital filter specified by `b` and `a`, evaluated at the frequencies in vector `w`.

The *phase delay* of a filter is the negative of phase divided by frequency:

$$\tau_p(\omega) = -\frac{\theta(\omega)}{\omega}$$

To plot both the group and phase delays of a system on the same FVTool graph, type

```
[b,a] = butter(10,200/1000);
hFVT = fvtool(b,a,'Analysis','grpdelay');
set(hFVT,'NumberofPoints',128,'OverlayedAnalysis','phasedelay')
;
legend(hFVT)
```


Group Delay and Phase Delay
Filter #1: Group delay
Filter #1: Phase delay
Group delay (in samples)
Phase Delay (rad)
Normalized Frequency (x rad/sample)
0
5
10
15
20
25
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9

Zero-Pole Analysis

The `zplane` function plots poles and zeros of a linear system. For example, a simple filter with a zero at -1/2 and a complex pole pair at $0.9e^{j2\pi(0.3)}$ and $0.9e^{-j2\pi(0.3)}$ is

```
zer = -0.5;
pol = 0.9*exp(j*2*pi*[-0.3 0.3]');
```

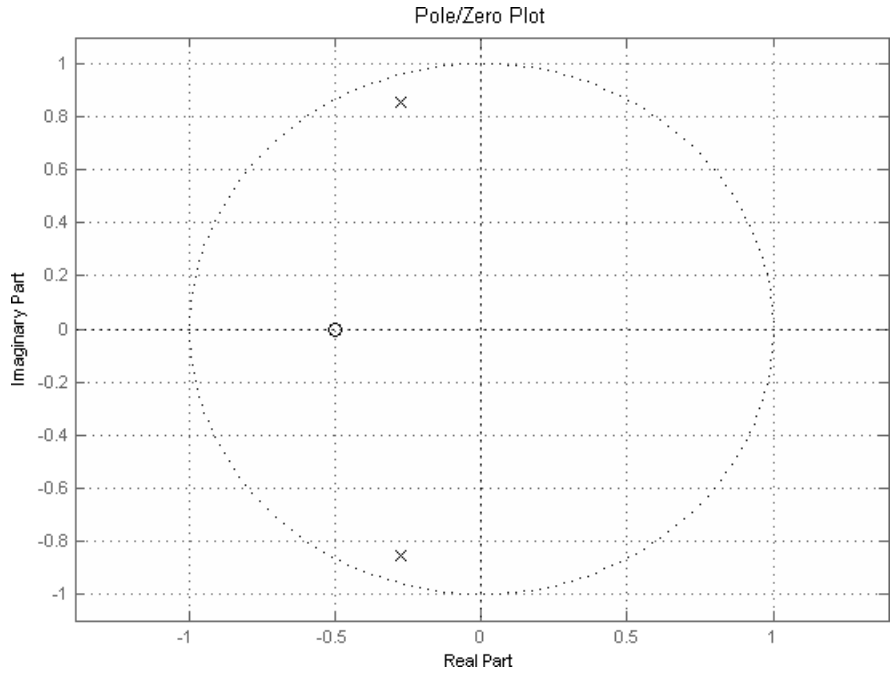
To view the pole-zero plot for this filter you can use

```
zplane(zer,pol)
```

or, for access to additional tools, use `fvtool`. First convert the poles and zeros to transfer function form, then call `fvtool`,

```
[b,a] = zp2tf(zer,pol,1);
fvtool(b,a)
```

and click the **Pole/Zero Plot** toolbar button  on the toolbar or select **Analysis > Pole/Zero Plot** to see the plot.



For a system in zero-pole form, supply column vector arguments `z` and `p` to `zplane`:

```
zplane(z,p)
```

For a system in transfer function form, supply row vectors `b` and `a` as arguments to `zplane`:

```
zplane(b,a)
```

In this case `zplane` finds the roots of `b` and `a` using the `roots` function and plots the resulting zeros and poles.

See “Linear System Models” on page 1-22 for details on zero-pole and transfer function representation of systems.

Linear System Models

In this section...

Available Models

Several Signal Processing Toolbox models are provided for representing linear time-invariant systems. This flexibility lets you choose the representational scheme that best suits your application and, within the bounds of numeric stability, convert freely to and from most other models. This section provides a brief overview of supported linear system models and describes how to work with these models in the MATLAB technical computing environment.

Discrete-Time System Models

The discrete-time system models are representational schemes for digital filters. The MATLAB technical computing environment supports several discrete-time system models, which are described in the following sections:

Transfer Function

The *transfer function* is a basic z-domain representation of a digital filter, expressing the filter as a ratio of two polynomials. It is the principal

discrete-time model for this toolbox. The transfer function model description for the z-transform of a digital filter's difference equation is

$$Y(z) = \frac{b(1) + b(2)z^{-1} + \cdots + b(n+1)z^{-n}}{a(1) + a(2)z^{-1} + \cdots + a(m+1)z^{-m}} X(z)$$

Here, the constants $b(i)$ and $a(i)$ are the filter coefficients, and the order of the filter is the maximum of n and m. In the MATLAB environment, you store these coefficients in two vectors (row vectors by convention), one row vector for the numerator and one for the denominator. See "Filters and Transfer Functions" on page 1-3 for more details on the transfer function form.

Zero-Pole-Gain

The factored or *zero-pole-gain* form of a transfer function is

$$H(z) = \frac{q(z)}{p(z)} = k\frac{(z-q(1))(z-q(2))...(z-q(n))}{(z-p(1))(z-p(2))...(z-p(n))}$$

By convention, polynomial coefficients are stored in row vectors and polynomial roots in column vectors. In zero-pole-gain form, therefore, the zero and pole locations for the numerator and denominator of a transfer function reside in column vectors. The factored transfer function gain k is a MATLAB scalar.

The `poly` and `roots` functions convert between polynomial and zero-pole-gain representations. For example, a simple IIR filter is

```
b = [2 3 4];
a = [1 3 3 1];
```

The zeros and poles of this filter are

```
q = roots(b)
q =
   -0.7500 + 1.1990i
   -0.7500 - 1.1990i
p = roots(a)
p =
   -1.0000
```

```
   -1.0000 + 0.0000i
   -1.0000 - 0.0000i
k = b(1)/a(1)
k =
    2
```

Returning to the original polynomials,

```
bb = k*poly(q)
bb =
    2.0000    3.0000    4.0000
aa = poly(p)
aa =
    1.0000    3.0000    3.0000    1.0000
```

Note that `b` and `a` in this case represent the transfer function:

$$H(z) = \frac{2 + 3z^{-1} + 4z^{-2}}{1 + 3z^{-1} + 3z^{-2} + z^{-3}} = \frac{2z^3 + 3z^2 + 4z}{z^3 + 3z^2 + 3z + 1}$$

For `b = [2 3 4]`, the `roots` function misses the zero for z equal to 0. In fact, it misses poles and zeros for z equal to 0 whenever the input transfer function has more poles than zeros, or vice versa. This is acceptable in most cases. To circumvent the problem, however, simply append zeros to make the vectors the same length before using the `roots` function; for example, `b = [b 0]`.

State-Space

It is always possible to represent a digital filter, or a system of difference equations, as a set of first-order difference equations. In matrix or *state-space* form, you can write the equations as

$$\begin{aligned} x(n+1) &= Ax(n) + Bu(n) \\ y(n) &= Cx(n) + Du(n) \end{aligned}$$

where `u` is the input, `x` is the state vector, and `y` is the output. For single-channel systems, `A` is an `m`-by-`m` matrix where `m` is the order of the filter, `B` is a column vector, `C` is a row vector, and `D` is a scalar. State-space notation is especially convenient for multichannel systems where input `u` and output `y` become vectors, and `B`, `C`, and `D` become matrices.

State-space representation extends easily to the MATLAB environment.`A`, `B`, `C`, and `D` are rectangular arrays; MATLAB functions treat them as individual variables.

Taking the z-transform of the state-space equations and combining them shows the equivalence of state-space and transfer function forms:

$$Y(z) = H(z)U(z), \text{ where } H(z) = C(zI - A)^{-1}B + D$$

Don't be concerned if you are not familiar with the state-space representation of linear systems. Some of the filter design algorithms use state-space form internally but do not require any knowledge of state-space concepts to use them successfully. If your applications use state-space based signal processing extensively, however, see the Control System Toolbox™ product for a comprehensive library of state-space tools.

Partial Fraction Expansion (Residue Form)

Each transfer function also has a corresponding *partial fraction expansion* or *residue* form representation, given by

$$\frac{b(z)}{a(z)} = \frac{r(1)}{1 - p(1)z^{-1}} + \ldots + \frac{r(n)}{1 - p(n)z^{-1}} + k(1) + k(2)z^{-1} + \ldots + k(m - n + 1)z^{-(m-n)}$$

provided $H(z)$ has no repeated poles. Here, n is the degree of the denominator polynomial of the rational transfer function $b(z)/a(z)$. If r is a pole of multiplicity s_r, then $H(z)$ has terms of the form:

$$\frac{r(j)}{1 - p(j)z^{-1}} + \frac{r(j+1)}{(1 - p(j)z^{-1})^2} \ldots + \frac{r(j + s_r - 1)}{(1 - p(j)z^{-1})^{s_r}}$$

The Signal Processing Toolbox `residuez` function in converts transfer functions to and from the partial fraction expansion form. The "`z`" on the end of `residuez` stands for z-domain, or discrete domain. `residuez` returns the poles in a column vector `p`, the residues corresponding to the poles in a column vector `r`, and any improper part of the original transfer function in a row vector `k`. `residuez` determines that two poles are the same if the magnitude of their difference is smaller than 0.1 percent of either of the poles' magnitudes.

Partial fraction expansion arises in signal processing as one method of finding the inverse z-transform of a transfer function. For example, the partial fraction expansion of

$$H(z) = \frac{-4 + 8z^{-1}}{1 + 6z^{-1} + 8z^{-2}}$$

is

```
b = [-4 8];
a = [1 6 8];
[r,p,k] = residuez(b,a)
r =
    -12
     8
p =
    -4
    -2
k =
    []
```

which corresponds to

$$H(z) = \frac{-12}{1 + 4z^{-1}} + \frac{8}{1 + 2z^{-1}}$$

To find the inverse z-transform of $H(z)$, find the sum of the inverse z-transforms of the two addends of $H(z)$, giving the causal impulse response:

$$h(n) = -12(-4)^n + 8(-2)^n, \quad n = 0, 1, 2, \ldots$$

To verify this in the MATLAB environment, type

```
imp = [1 0 0 0 0];
resptf = filter(b,a,imp)
resptf =
        -4    32    -160    704    -2944
respres = filter(r(1),[1 -p(1)],imp)+...
  filter(r(2),[1 -p(2)],imp)
respres =
```

```
    -4     32     -160     704     -2944
```

Second-Order Sections (SOS)

Any transfer function $H(z)$ has a second-order sections representation

$$H(z) = \prod_{k=1}^{L} H_k(z) = \prod_{k=1}^{L} \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2}}{a_{0k} + a_{1k}z^{-1} + a_{2k}z^{-2}}$$

where L is the number of second-order sections that describe the system. The MATLAB environment represents the second-order section form of a discrete-time system as an L-by-6 array `sos`. Each row of `sos` contains a single second-order section, where the row elements are the three numerator and three denominator coefficients that describe the second-order section.

$$sos = \begin{bmatrix} b_{01} & b_{11} & b_{21} & a_{01} & a_{11} & a_{21} \\ b_{02} & b_{12} & b_{22} & a_{02} & a_{12} & a_{22} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{0L} & b_{1L} & b_{2L} & a_{0L} & a_{1L} & a_{2L} \end{bmatrix}$$

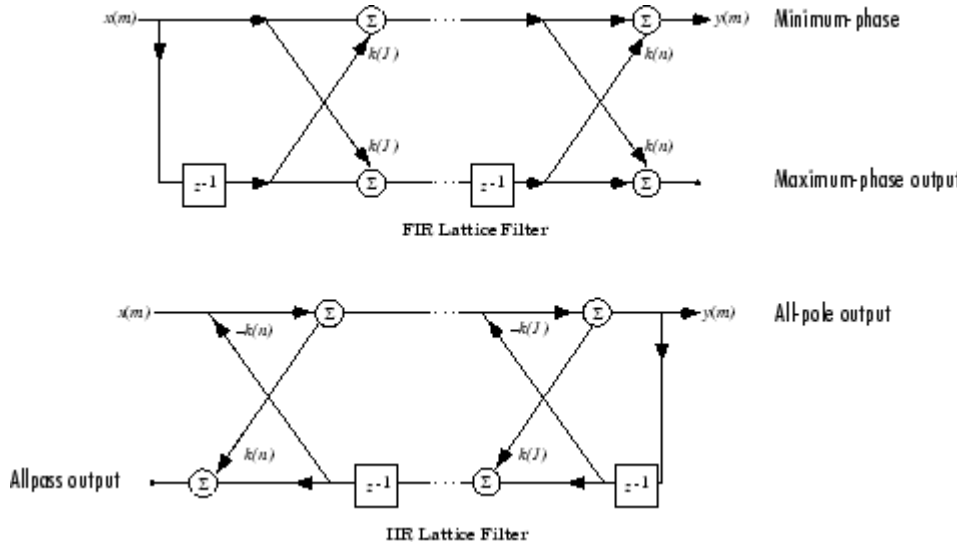
There are many ways to represent a filter in second-order section form. Through careful pairing of the pole and zero pairs, ordering of the sections in the cascade, and multiplicative scaling of the sections, it is possible to reduce quantization noise gain and avoid overflow in some fixed-point filter implementations. The functions `zp2sos` and `ss2sos`, described in "Linear System Transformations" on page 1-32, perform pole-zero pairing, section scaling, and section ordering.

Note All Signal Processing Toolbox second-order section transformations apply only to digital filters.

Lattice Structure

For a discrete Nth order all-pole or all-zero filter described by the polynomial coefficients $a(n)$, $n = 1, 2, ..., N+1$, there are N corresponding lattice structure

coefficients $k(n)$, $n = 1, 2, ..., N$. The parameters $k(n)$ are also called the *reflection coefficients* of the filter. Given these reflection coefficients, you can implement a discrete filter as shown below.



FIR and IIR Lattice Filter structure diagrams

For a general pole-zero IIR filter described by polynomial coefficients a and b, there are both lattice coefficients $k(n)$ for the denominator a and ladder coefficients $v(n)$ for the numerator b. The lattice/ladder filter may be implemented as

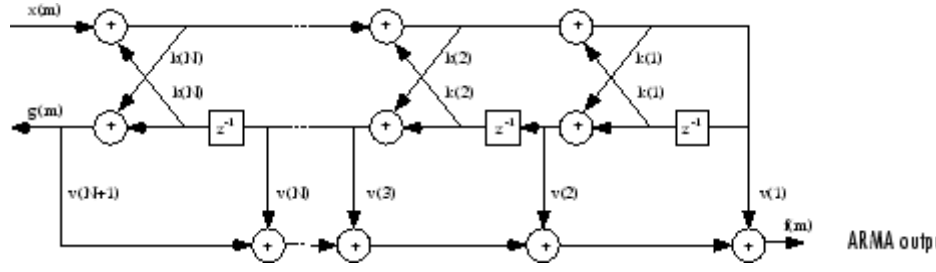


Diagram of lattice/ladder filter

The toolbox function `tf2latc` accepts an FIR or IIR filter in polynomial form and returns the corresponding reflection coefficients. An example FIR filter in polynomial form is

```
b = [1.0000   0.6149   0.9899   0.0000   0.0031  -0.0082];
```

This filter's lattice (reflection coefficient) representation is

```
k = tf2latc(b)
k =
    0.3090
    0.9801
    0.0031
    0.0081
   -0.0082
```

For IIR filters, the magnitude of the reflection coefficients provides an easy stability check. If all the reflection coefficients corresponding to a polynomial have magnitude less than 1, all of that polynomial's roots are inside the unit circle. For example, consider an IIR filter with numerator polynomial `b` from above and denominator polynomial:

```
a = [1 1/2 1/3];
```

The filter's lattice representation is

```
[k,v] = tf2latc(b,a)
k =
    0.3750
    0.3333
         0
         0
         0
v =
    0.6252
    0.1212
    0.9879
   -0.0009
    0.0072
   -0.0082
```

Because `abs(k) < 1` for all reflection coefficients in `k`, the filter is stable.

The function `latc2tf` calculates the polynomial coefficients for a filter from its lattice (reflection) coefficients. Given the reflection coefficient vector `k` (above), the corresponding polynomial form is

```
b = latc2tf(k)
b =
    1.0000    0.6149    0.9899 -0.0000    0.0031   -0.0082
```

The lattice or lattice/ladder coefficients can be used to implement the filter using the function `latcfilt`.

Convolution Matrix

In signal processing, convolving two vectors or matrices is equivalent to filtering one of the input operands by the other. This relationship permits the representation of a digital filter as a *convolution matrix*.

Given any vector, the toolbox function `convmtx` generates a matrix whose inner product with another vector is equivalent to the convolution of the two vectors. The generated matrix represents a digital filter that you can apply to any vector of appropriate length; the inner dimension of the operands must agree to compute the inner product.

The convolution matrix for a vector `b`, representing the numerator coefficients for a digital filter, is

```
b = [1 2 3]; x = randn(3,1);
C = convmtx(b',3)
C =
    1    0    0
    2    1    0
    3    2    1
    0    3    2
    0    0    3
```

Two equivalent ways to convolve `b` with `x` are as follows.

```
y1 = C*x;
y2 = conv(b,x);
```

Continuous-Time System Models

The continuous-time system models are representational schemes for analog filters. Many of the discrete-time system models described earlier are also appropriate for the representation of continuous-time systems:

- State-space form
- Partial fraction expansion
- Transfer function
- Zero-pole-gain form

It is possible to represent any system of linear time-invariant differential equations as a set of first-order differential equations. In matrix or *state-space* form, you can express the equations as

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

where u is a vector of nu inputs, x is an nx-element state vector, and y is a vector of ny outputs. In the MATLAB environment, `A`, `B`, `C`, and `D` are stored in separate rectangular arrays.

An equivalent representation of the state-space system is the Laplace transform transfer function description

$$Y(s) = H(s)U(s)$$

where

$$H(s) = C(sI - A)^{-1}B + D$$

For single-input, single-output systems, this form is given by

$$H(s) = \frac{b(s)}{a(s)} = \frac{b(1)s^n + b(2)s^{n-1} + \cdots + b(n+1)}{a(1)s^m + a(2)s^{m-1} + \cdots + a(m+1)}$$

Given the coefficients of a Laplace transform transfer function, `residue` determines the partial fraction expansion of the system. See the description of `residue` in the MATLAB documentation for details.

The factored zero-pole-gain form is

$$H(s) = \frac{z(s)}{p(s)} = k\frac{(s-z(1))(s-z(2))\cdots(s-z(n))}{(s-p(1))(s-p(2))\cdots(s-p(n))}$$

As in the discrete-time case, the MATLAB environment stores polynomial coefficients in row vectors in descending powers of *s*. It stores polynomial roots, or zeros and poles, in column vectors.

Linear System Transformations

A number of Signal Processing Toolbox functions are provided to convert between the various linear system models; see Chapter 10, "Functions — Alphabetical List" for a complete description of each. You can use the following chart to find an appropriate transfer function: find the row of the model to convert *from* on the left side of the chart and the column of the model to convert *to* on the top of the chart and read the function name(s) at the intersection of the row and column. Note that some cells of this table are empty.

	Transfer Function	**State-Space**	**Zero-Pole-Gain**	**Partial Fraction**	**Lattice Filter**	**Second-Order Sections**	**Convolution Matrix**
Transfer Function		`tf2ss`	`tf2zp` `roots`	`residuez`	`tf2latc`	none	`convmtx`
State-Space	`ss2tf`		`ss2zp`	none	none	`ss2sos`	none
Zero-Pole-Gain	`zp2tf` `poly`	`zp2ss`		none	none	`zp2sos`	none
Partial Fraction	`residuez`	none	none		none	none	none
Lattice Filter	`latc2tf`	none	none	none		none	none
SOS	`sos2tf`	`sos2ss`	`sos2zp`	none	none		none

Note Converting from one filter structure or model to another may produce a result with different characteristics than the original. This is due to the computer's finite-precision arithmetic and the variations in the conversion's round-off computations.

Many of the toolbox filter design functions use these functions internally. For example, the `zp2ss` function converts the poles and zeros of an analog prototype into the state-space form required for creation of a Butterworth, Chebyshev, or elliptic filter. Once in state-space form, the filter design function performs any required frequency transformation, that is, it transforms the initial lowpass design into a bandpass, highpass, or bandstop filter, or a lowpass filter with the desired cutoff frequency. See the descriptions of the individual filter design functions in Chapter 10, "Functions — Alphabetical List" for more details.

Note All Signal Processing Toolbox second-order section transformations apply only to digital filters.

Discrete Fourier Transform

The discrete Fourier transform, or DFT, is the primary tool of digital signal processing. The foundation of Signal Processing Toolbox product is the fast Fourier transform (FFT), a method for computing the DFT with reduced execution time. Many of the toolbox functions (including *z*-domain frequency response, spectrum and cepstrum analysis, and some filter design and implementation functions) incorporate the FFT.

The MATLAB environment provides the functions `fft` and `ifft` to compute the discrete Fourier transform and its inverse, respectively. For the input sequence *x* and its transformed version *X* (the discrete-time Fourier transform at equally spaced frequencies around the unit circle), the two functions implement the relationships

$$X(k+1) = \sum_{n=0}^{N-1} x(n+1) W_N^{kn}$$

$$x(n+1) = \frac{1}{N} \sum_{k=0}^{N-1} X(k+1) W_N^{-kn}$$

In these equations, the series subscripts begin with 1 instead of 0 because of the MATLAB vector indexing scheme, and

$$W_N = e^{-j\left(\frac{2\pi}{N}\right)}$$

Note The MATLAB convention is to use a negative *j* for the `fft` function. This is an engineering convention; physics and pure mathematics typically use a positive *j*.

`fft`, with a single input argument `x`, computes the DFT of the input vector or matrix. If `x` is a vector, `fft` computes the DFT of the vector; if `x` is a rectangular array, `fft` computes the DFT of each array column.

For example, create a time vector and signal:

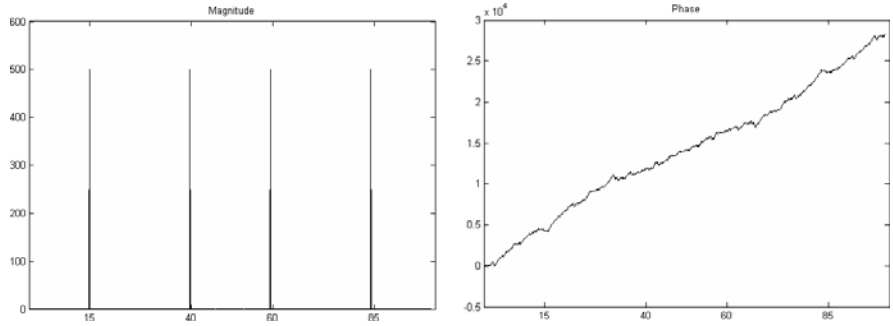
```
t = (0:1/100:10-1/100);                     % Time vector
x = sin(2*pi*15*t) + sin(2*pi*40*t);        % Signal
```

The DFT of the signal, and the magnitude and phase of the transformed sequence, are then

```
y = fft(x);        % Compute DFT of x
m = abs(y); p = unwrap(angle(y));  % Magnitude and phase
```

To plot the magnitude and phase, type the following commands:

```
f = (0:length(y)-1)*99/length(y);  % Frequency vector
plot(f,m); title('Magnitude');
set(gca,'XTick',[15 40 60 85]);
figure; plot(f,p*180/pi); title('Phase');
set(gca,'XTick',[15 40 60 85]);
```



A second argument to `fft` specifies a number of points `n` for the transform, representing DFT length:

```
y = fft(x,n);
```

In this case, `fft` pads the input sequence with zeros if it is shorter than `n`, or truncates the sequence if it is longer than `n`. If `n` is not specified, it defaults to the length of the input sequence. Execution time for `fft` depends on the length, `n`, of the DFT it performs; see the `fft` reference page in the MATLAB documentation for details about the algorithm.

Note The resulting FFT amplitude is A*n/2, where A is the original amplitude and n is the number of FFT points. This is true only if the number of FFT points is greater than or equal to the number of data samples. If the number of FFT points is less, the FFT amplitude is lower than the original amplitude by the above amount.

The inverse discrete Fourier transform function `ifft` also accepts an input sequence and, optionally, the number of desired points for the transform. Try the example below; the original sequence `x` and the reconstructed sequence are identical (within rounding error).

```
t = (0:1/255:1);
x = sin(2*pi*120*t);
y = real(ifft(fft(x)));
```

This toolbox also includes functions for the two-dimensional FFT and its inverse, `fft2` and `ifft2`. These functions are useful for two-dimensional signal or image processing. The `goertzel` function, which is another algorithm to compute the DFT, also is included in the toolbox. This function is efficient for computing the DFT of a portion of a long signal.

It is sometimes convenient to rearrange the output of the `fft` or `fft2` function so the zero frequency component is at the center of the sequence. The MATLAB function `fftshift` moves the zero frequency component to the center of a vector or matrix.

2

Filter Design and Implementation

Filter Requirements and Specification

Filter design is the process of creating the filter coefficients to meet specific filtering requirements. Filter implementation involves choosing and applying a particular filter structure to those coefficients. Only after both design and implementation have been performed can data be filtered. The following chapter describes filter design and implementation in Signal Processing Toolbox software.

The goal of filter design is to perform frequency dependent alteration of a data sequence. A possible requirement might be to remove noise above 30 Hz from a data sequence sampled at 100 Hz. A more rigorous specification might call for a specific amount of passband ripple, stopband attenuation, or transition width. A very precise specification could ask to achieve the performance goals with the minimum filter order, or it could call for an arbitrary magnitude shape, or it might require an FIR filter.

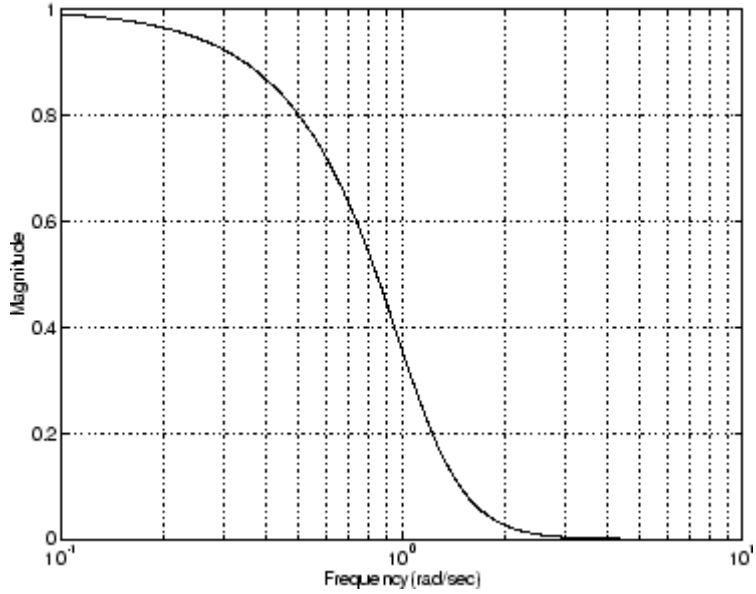
Filter design methods differ primarily in how performance is specified. For "loosely specified" requirements, as in the first case above, a Butterworth IIR filter is often sufficient. To design a fifth-order 30 Hz lowpass Butterworth filter and apply it to the data in vector `x`:

```
Fs=100; %Sampling Frequency
d=fdesign.lowpass('N,F3dB',5,30,Fs); %lowpass filter specification object
Hd=design(d,'butter')
y=filter(Hd,x);
```

The second input argument to `butter` specifies the cutoff frequency, normalized to half the sampling frequency (the Nyquist frequency).

All of the filter design functions operate with normalized frequencies, so they do not require the system sampling rate as an extra input argument. This toolbox uses the convention that unit frequency is the Nyquist frequency, defined as half the sampling frequency. The normalized frequency, therefore, is always in the interval $0 \le f \le 1$. For a system with a 1000 Hz sampling frequency, 300 Hz is 300/500 = 0.6. To convert normalized frequency to angular frequency around the unit circle, multiply by π. To convert normalized frequency back to hertz, multiply by half the sample frequency.

More rigorous filter requirements traditionally include passband ripple (`Rp`, in decibels), stopband attenuation (`Rs`, in decibels), and transition width (`Ws-Wp`, in hertz).



You can design Butterworth, Chebyshev Type I, Chebyshev Type II, and elliptic filters that meet this type of performance specification. The toolbox order selection functions estimate the minimum filter order that meets a given set of requirements.

To meet specifications with more rigid constraints like linear phase or arbitrary filter shape, use the FIR and direct IIR filter design routines.

IIR Filter Design

In this section...

IIR vs. FIR Filters

The primary advantage of IIR filters over FIR filters is that they typically meet a given set of specifications with a much lower filter order than a corresponding FIR filter. Although IIR filters have nonlinear phase, data processing within MATLAB software is commonly performed "offline," that is, the entire data sequence is available prior to filtering. This allows for a noncausal, zero-phase filtering approach (via the `filtfilt` function), which eliminates the nonlinear phase distortion of an IIR filter.

Classical IIR Filters

The classical IIR filters, Butterworth, Chebyshev Types I and II, elliptic, and Bessel, all approximate the ideal "brick wall" filter in different ways.

This toolbox provides functions to create all these types of classical IIR filters in both the analog and digital domains (except Bessel, for which only the analog case is supported), and in lowpass, highpass, bandpass, and bandstop configurations. For most filter types, you can also find the lowest filter order that fits a given filter specification in terms of passband and stopband attenuation, and transition width(s).

Other IIR Filters

The direct filter design function `yulewalk` finds a filter with magnitude response approximating a desired function. This is one way to create a multiband bandpass filter.

You can also use the parametric modeling or system identification functions to design IIR filters. These functions are discussed in "Parametric Modeling" on page 7-15.

The generalized Butterworth design function `maxflat` is discussed in the section "Generalized Butterworth Filter Design" on page 2-15.

IIR Filter Method Summary

The following table summarizes the various filter methods in the toolbox and lists the functions available to implement these methods.

Toolbox Filters Methods and Available Functions

Filter Method	Description	Filter Functions

Analog Prototyping	Using the poles and zeros of a classical lowpass prototype filter in the continuous (Laplace) domain, obtain a digital filter through frequency transformation and filter discretization.	Complete design functions:
		`besself`, `butter`, `cheby1`, `cheby2`, `ellip`
		Order estimation functions:
		
		`buttord`, `cheb1ord`, `cheb2ord`, `ellipord`
		Lowpass analog prototype functions:
		
		`besselap`, `buttap`, `cheb1ap`, `cheb2ap`, `ellipap`
		Frequency transformation functions:
		
		`lp2bp`, `lp2bs`, `lp2hp`, `lp2lp`
		Filter discretization functions:
		
		`bilinear`, `impinvar`

Toolbox Filters Methods and Available Functions (Continued)

Filter Method	Description	Filter Functions
Direct Design	Design digital filter directly in the discrete time-domain by approximating a piecewise linear magnitude response.	`yulewalk`
Generalized Butterworth Design	Design lowpass Butterworth filters with more zeros than poles.	`maxflat`

Parametric Modeling	Find a digital filter that approximates a prescribed time or frequency domain response. (See System Identification Toolbox™ documentation for an extensive collection of parametric modeling tools.)	Time-domain modeling functions:
		
		`lpc`, `prony`, `stmcb`
		Frequency-domain modeling functions:
		
		`invfreqs`, `invfreqz`

Classical IIR Filter Design Using Analog Prototyping

The principal IIR digital filter design technique this toolbox provides is based on the conversion of classical lowpass analog filters to their digital equivalents. The following sections describe how to design filters and summarize the characteristics of the supported filter types. See "Special Topics in IIR Filter Design" on page 2-43 for detailed steps on the filter design process.

Complete Classical IIR Filter Design

You can easily create a filter of any order with a lowpass, highpass, bandpass, or bandstop configuration using the filter design functions.

Filter Design Functions

Filter Type	Design Function

Bessel (analog only)	`[b,a] = besself(n,Wn,`*`options`*`)`
	`[z,p,k] = besself(n,Wn,`*`options`*`)`
	`[A,B,C,D] = besself(n,Wn,`*`options`*`)`

Filter Design Functions (Continued)

Filter Type	Design Function

Butterworth	`[b,a] = butter(n,Wn,options)`
	`[z,p,k] = butter(n,Wn,options)`
	`[A,B,C,D] = butter(n,Wn,options)`

Chebyshev Type I	`[b,a] = cheby1(n,Rp,Wn,options)`
	`[z,p,k] = cheby1(n,Rp,Wn,options)`
	`[A,B,C,D] = cheby1(n,Rp,Wn,options)`

Chebyshev Type II	`[b,a] = cheby2(n,Rs,Wn,options)`
	`[z,p,k] = cheby2(n,Rs,Wn,options)`
	`[A,B,C,D] = cheby2(n,Rs,Wn,options)`

Elliptic	`[b,a] = ellip(n,Rp,Rs,Wn,options)`
	`[z,p,k] = ellip(n,Rp,Rs,Wn,options)`
	`[A,B,C,D] = ellip(n,Rp,Rs,Wn,options)`

By default, each of these functions returns a lowpass filter; you need only specify the desired cutoff frequency `Wn` in normalized frequency (Nyquist frequency = 1 Hz). For a highpass filter, append the string `'high'` to the function's parameter list. For a bandpass or bandstop filter, specify `Wn` as a two-element vector containing the passband edge frequencies, appending the string `'stop'` for the bandstop configuration.

Here are some example digital filters:

```
[b,a] = butter(5,0.4);                    % Lowpass Butterworth
[b,a] = cheby1(4,1,[0.4 0.7]);            % Bandpass Chebyshev Type I
[b,a] = cheby2(6,60,0.8,'high');          % Highpass Chebyshev Type II
[b,a] = ellip(3,1,60,[0.4 0.7],'stop');   % Bandstop elliptic
```

To design an analog filter, perhaps for simulation, use a trailing `'s'` and specify cutoff frequencies in rad/s:

```
[b,a] = butter(5,.4,'s');      % Analog Butterworth filter
```

All filter design functions return a filter in the transfer function, zero-pole-gain, or state-space linear system model representation, depending on how many output arguments are present. In general, you should avoid using the transfer function form because numerical problems caused by roundoff errors can occur. Instead, use the zero-pole-gain form which you can convert to a second-order section (SOS) form using `zp2sos` and then use the SOS form with `dfilt` to analyze or implement your filter.

Note All classical IIR lowpass filters are ill-conditioned for extremely low cutoff frequencies. Therefore, instead of designing a lowpass IIR filter with a very narrow passband, it can be better to design a wider passband and decimate the input signal.

Designing IIR Filters to Frequency Domain Specifications

This toolbox provides order selection functions that calculate the minimum filter order that meets a given set of requirements.

Filter Type	Order Estimation Function
Butterworth	`[n,Wn] = buttord(Wp,Ws,Rp,Rs)`
Chebyshev Type I	`[n,Wn] = cheb1ord(Wp, Ws, Rp, Rs)`
Chebyshev Type II	`[n,Wn] = cheb2ord(Wp, Ws, Rp, Rs)`
Elliptic	`[n,Wn] = ellipord(Wp, Ws, Rp, Rs)`

These are useful in conjunction with the filter design functions. Suppose you want a bandpass filter with a passband from 1000 to 2000 Hz, stopbands starting 500 Hz away on either side, a 10 kHz sampling frequency, at most 1 dB of passband ripple, and at least 60 dB of stopband attenuation. You can meet these specifications by using the `butter` function as follows.

```
[n,Wn] = buttord([1000 2000]/5000,[500 2500]/5000,1,60)
n =
    12
Wn =
    0.1951    0.4080
```

```
[b,a] = butter(n,Wn);
```

An elliptic filter that meets the same requirements is given by

```
[n,Wn] = ellipord([1000 2000]/5000,[500 2500]/5000,1,60)
n =
    5
Wn =
    0.2000    0.4000
[b,a] = ellip(n,1,60,Wn);
```

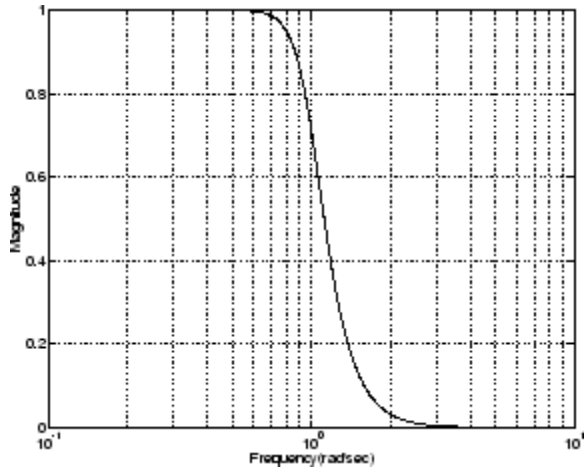
These functions also work with the other standard band configurations, as well as for analog filters; see Chapter 10, "Functions — Alphabetical List" for details.

Comparison of Classical IIR Filter Types

The toolbox provides five different types of classical IIR filters, each optimal in some way. This section shows the basic analog prototype form for each and summarizes major characteristics.

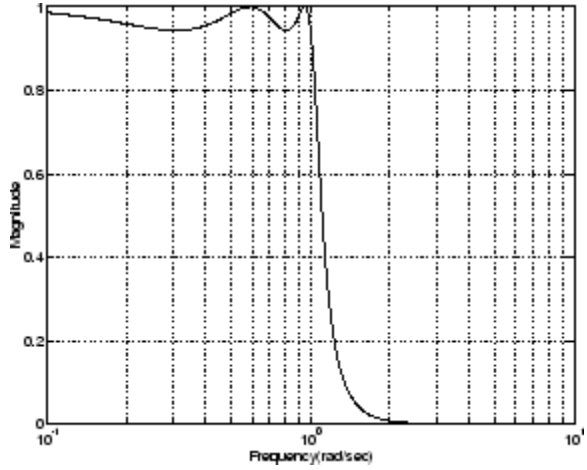
Butterworth Filter

The Butterworth filter provides the best Taylor Series approximation to the ideal lowpass filter response at analog frequencies $\Omega = 0$ and $\Omega = \infty$; for any order N, the magnitude squared response has $2N$-1 zero derivatives at these locations (*maximally flat* at $\Omega = 0$ and $\Omega = \infty$). Response is monotonic overall, decreasing smoothly from $\Omega = 0$ to $\Omega = \infty$. $|H(j\Omega)| = \sqrt{1/2}$ at $\Omega = 1$.



Chebyshev Type I Filter

The Chebyshev Type I filter minimizes the absolute difference between the ideal and actual frequency response over the entire passband by incorporating an equal ripple of Rp dB in the passband. Stopband response is maximally flat. The transition from passband to stopband is more rapid than for the Butterworth filter. $|H(j\Omega)| = 10^{-Rp/20}$ at $\Omega = 1$.

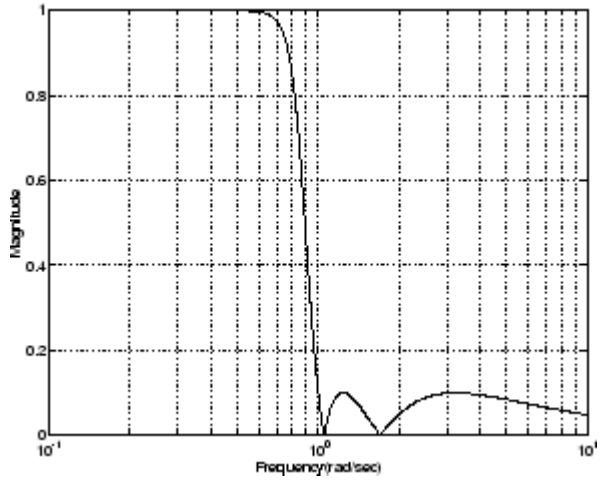


Chebyshev Type II Filter

The Chebyshev Type II filter minimizes the absolute difference between the ideal and actual frequency response over the entire stopband by incorporating an equal ripple of Rs dB in the stopband. Passband response is maximally flat.

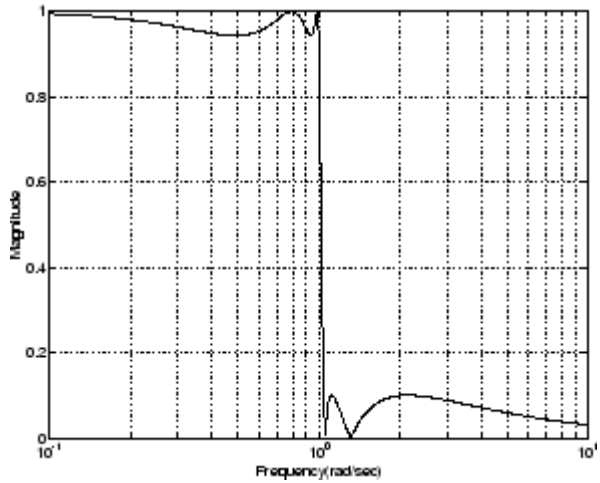
The stopband does not approach zero as quickly as the type I filter (and does not approach zero at all for even-valued filter order n). The absence of ripple in the passband, however, is often an important advantage.

$|H(j\Omega)| = 10^{-Rs/20}$ at $\Omega = 1$.



Elliptic Filter

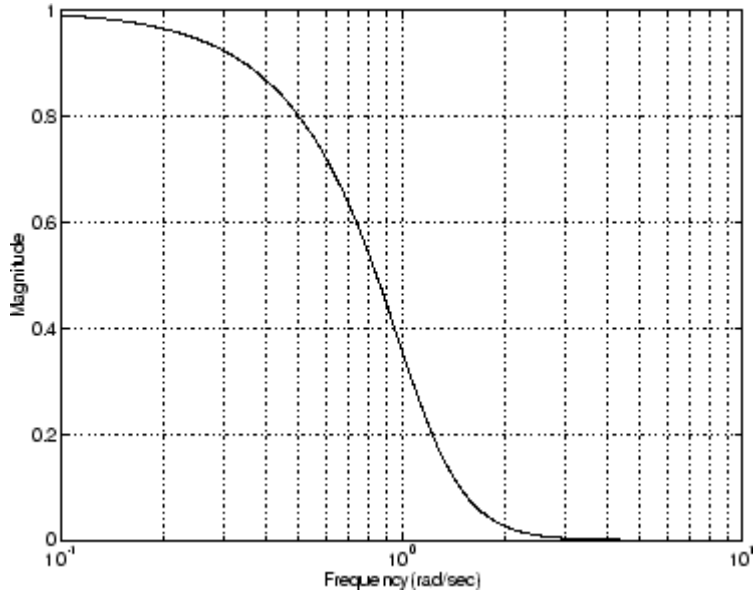
Elliptic filters are equiripple in both the passband and stopband. They generally meet filter requirements with the lowest order of any supported filter type. Given a filter order n, passband ripple Rp in decibels, and stopband ripple Rs in decibels, elliptic filters minimize transition width. $|H(j\Omega)| = 10^{-Rp/20}$ at $\Omega = 1$.



Bessel Filter

Analog Bessel lowpass filters have maximally flat group delay at zero frequency and retain nearly constant group delay across the entire passband. Filtered signals therefore maintain their waveshapes in the passband frequency range. Frequency mapped and digital Bessel filters, however, do not have this maximally flat property; this toolbox supports only the analog case for the complete Bessel filter design function.

Bessel filters generally require a higher filter order than other filters for satisfactory stopband attenuation. $|H(j\Omega)| < 1/\sqrt{2}$ at $\Omega = 1$ and decreases as filter order n increases.



Note The lowpass filters shown above were created with the analog prototype functions `besselap`, `buttap`, `cheb1ap`, `cheb2ap`, and `ellipap`. These functions find the zeros, poles, and gain of an order `n` analog filter of the appropriate type with cutoff frequency of 1 rad/s. The complete filter design functions (`besself`, `butter`, `cheby1`, `cheby2`, and `ellip`) call the prototyping functions as a first step in the design process. See "Special Topics in IIR Filter Design" on page 2-43 for details.

To create similar plots, use `n = 5` and, as needed, `Rp = 0.5` and `Rs = 20`. For example, to create the elliptic filter plot:

```
[z,p,k] = ellipap(5,0.5,20);
w = logspace(-1,1,1000);
h = freqs(k*poly(z),poly(p),w);
semilogx(w,abs(h)), grid
```

Direct IIR Filter Design

This toolbox uses the term *direct methods* to describe techniques for IIR design that find a filter based on specifications in the discrete domain. Unlike the analog prototyping method, direct design methods are not constrained to the standard lowpass, highpass, bandpass, or bandstop configurations. Rather, these functions design filters with an arbitrary, perhaps multiband, frequency response. This section discusses the `yulewalk` function, which is intended specifically for filter design; "Parametric Modeling" on page 7-15 discusses other methods that may also be considered direct, such as Prony's method, Linear Prediction, the Steiglitz-McBride method, and inverse frequency design.

The `yulewalk` function designs recursive IIR digital filters by fitting a specified frequency response. `yulewalk`'s name reflects its method for finding the filter's denominator coefficients: it finds the inverse FFT of the ideal desired magnitude-squared response and solves the modified Yule-Walker equations using the resulting autocorrelation function samples. The statement

```
[b,a] = yulewalk(n,f,m)
```

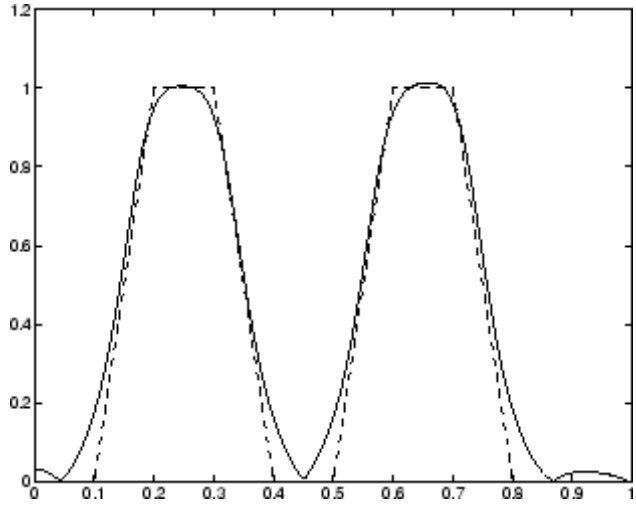
returns row vectors `b` and `a` containing the `n+1` numerator and denominator coefficients of the order `n` IIR filter whose frequency-magnitude characteristics approximate those given in vectors `f` and `m`. `f` is a vector of frequency points ranging from 0 to 1, where 1 represents the Nyquist frequency. `m` is a vector containing the desired magnitude response at the points in `f`. `f` and `m` can describe any piecewise linear shape magnitude response, including a multiband response. The FIR counterpart of this function is `fir2`, which also designs a filter based on an arbitrary piecewise linear magnitude response. See "FIR Filter Design" on page 2-17 for details.

Note that `yulewalk` does not accept phase information, and no statements are made about the optimality of the resulting filter.

Design a multiband filter with `yulewalk`, and plot the desired and actual frequency response:

```
m = [0   0   1   1   0   0   1   1   0 0];
f = [0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 1];
[b,a] = yulewalk(10,f,m);
[h,w] = freqz(b,a,128)
```

```
plot(f,m,w/pi,abs(h))
```



Generalized Butterworth Filter Design

The toolbox function `maxflat` enables you to design generalized Butterworth filters, that is, Butterworth filters with differing numbers of zeros and poles. This is desirable in some implementations where poles are more expensive computationally than zeros. `maxflat` is just like the `butter` function, except that it you can specify *two* orders (one for the numerator and one for the denominator) instead of just one. These filters are *maximally flat*. This means that the resulting filter is optimal for any numerator and denominator orders, with the maximum number of derivatives at 0 and the Nyquist frequency $\omega = \pi$ both set to 0.

For example, when the two orders are the same, `maxflat` is the same as `butter`:

```
[b,a] = maxflat(3,3,0.25)
b =
    0.0317    0.0951    0.0951    0.0317
a =
    1.0000   -1.4590    0.9104   -0.1978
[b,a] = butter(3,0.25)
b =
```

```
    0.0317    0.0951    0.0951    0.0317
a =
    1.0000   -1.4590    0.9104   -0.1978
```

However, `maxflat` is more versatile because it allows you to design a filter with more zeros than poles:

```
[b,a] = maxflat(3,1,0.25)
b =
    0.0950    0.2849    0.2849    0.0950
a =
    1.0000   -0.2402
```

The third input to maxflat is the *half-power frequency*, a frequency between 0 and 1 with a desired magnitude response of $1/\sqrt{2}$.

You can also design linear phase filters that have the maximally flat property using the `'sym'` option:

```
maxflat(4,'sym',0.3)
ans =
    0.0331    0.2500    0.4337    0.2500    0.0331
```

For complete details of the `maxflat` algorithm, see Selesnick and Burrus [2].

FIR Filter Design

In this section...

FIR vs. IIR Filters

Digital filters with finite-duration impulse response (all-zero, or FIR filters) have both advantages and disadvantages compared to infinite-duration impulse response (IIR) filters.

FIR filters have the following primary advantages:

- They can have exactly linear phase.
- They are always stable.
- The design methods are generally linear.
- They can be realized efficiently in hardware.
- The filter startup transients have finite duration.

The primary disadvantage of FIR filters is that they often require a much higher filter order than IIR filters to achieve a given level of performance. Correspondingly, the delay of these filters is often much greater than for an equal performance IIR filter.

FIR Filter Summary

FIR Filters

Filter Design Method	Description	Filter Functions
Windowing	Apply window to truncated inverse Fourier transform of desired "brick wall" filter	`fir1`, `fir2`, `kaiserord`
Multiband with Transition Bands	Equiripple or least squares approach over sub-bands of the frequency range	`firls`, `firpm`, `firpmord`
Constrained Least Squares	Minimize squared integral error over entire frequency range subject to maximum error constraints	`fircls`, `fircls1`
Arbitrary Response	Arbitrary responses, including nonlinear phase and complex filters	`cfirpm`
Raised Cosine	Lowpass response with smooth, sinusoidal transition	`firrcos`

Linear Phase Filters

Except for `cfirpm`, all of the FIR filter design functions design linear phase filters only. The filter coefficients, or "taps," of such filters obey either an even or odd symmetry relation. Depending on this symmetry, and on whether the order n of the filter is even or odd, a linear phase filter (stored in length n+1 vector `b`) has certain inherent restrictions on its frequency response.

Linear Phase Filter Type	Filter Order	Symmetry of Coefficients	Response H(f), f = 0	Response H(f), f = 1 (Nyquist)
Type I	Even	even: $b(k) = b(n+2-k), \quad k = 1, \ldots, n+1$	No restriction	No restriction

Linear Phase Filter Type	Filter Order	Symmetry of Coefficients	Response H(f), f = 0	Response H(f), f = 1 (Nyquist)
Type II	Odd	even: $b(k) = b(n+2-k), \quad k = 1, \ldots, n+1$	No restriction	$H(1) = 0$
Type III	Even	odd: $b(k) = -b(n+2-k), \quad k = 1, \ldots, n+1$	$H(0) = 0$	$H(1) = 0$
Type IV	Odd	odd: $b(k) = -b(n+2-k), \quad k = 1, \ldots, n+1$	$H(0) = 0$	No restriction

The phase delay and group delay of linear phase FIR filters are equal and constant over the frequency band. For an order n linear phase FIR filter, the group delay is $n/2$, and the filtered signal is simply delayed by $n/2$ time steps (and the magnitude of its Fourier transform is scaled by the filter's magnitude response). This property preserves the wave shape of signals in the passband; that is, there is no phase distortion.

The functions `fir1`, `fir2`, `firls`, `firpm`, `fircls`, `fircls1`, and `firrcos` all design type I and II linear phase FIR filters by default. Both `firls` and `firpm` design type III and IV linear phase FIR filters given a `'hilbert'` or `'differentiator'` flag. `cfirpm` can design any type of linear phase filter, and nonlinear phase filters as well.

Note Because the frequency response of a type II filter is zero at the Nyquist frequency ("high" frequency), `fir1` does not design type II highpass and bandstop filters. For odd-valued `n` in these cases, `fir1` adds 1 to the order and returns a type I filter.

Windowing Method

Consider the ideal, or "brick wall," digital lowpass filter with a cutoff frequency of ω_0 rad/s. This filter has magnitude 1 at all frequencies with

magnitude less than ω_0, and magnitude 0 at frequencies with magnitude between ω_0 and π. Its impulse response sequence $h(n)$ is

$$h(n) = \frac{1}{2\pi}\int_{-\pi}^{\pi} H(\omega)e^{j\omega n}d\omega = \frac{1}{2\pi}\int_{-\omega_0}^{\omega_0} e^{j\omega n}d\omega = \frac{\omega_0}{\pi}\mathrm{sinc}(\frac{\omega_0}{\pi}n)$$

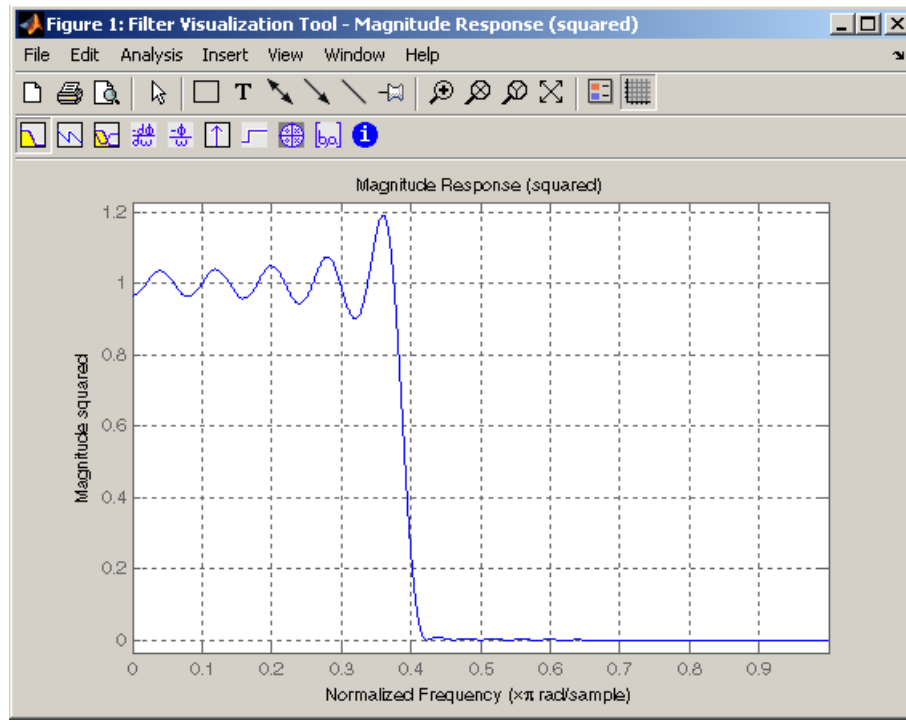
This filter is not implementable since its impulse response is infinite and noncausal. To create a finite-duration impulse response, truncate it by applying a window. By retaining the central section of impulse response in this truncation, you obtain a linear phase FIR filter. For example, a length 51 filter with a lowpass cutoff frequency ω_0 of 0.4π rad/s is

```
b = 0.4*sinc(0.4*(-25:25));
```

The window applied here is a simple rectangular window. By Parseval's theorem, this is the length 51 filter that best approximates the ideal lowpass filter, in the integrated least squares sense. The following command displays the filter's frequency response in FVTool:

```
fvtool(b,1)
```

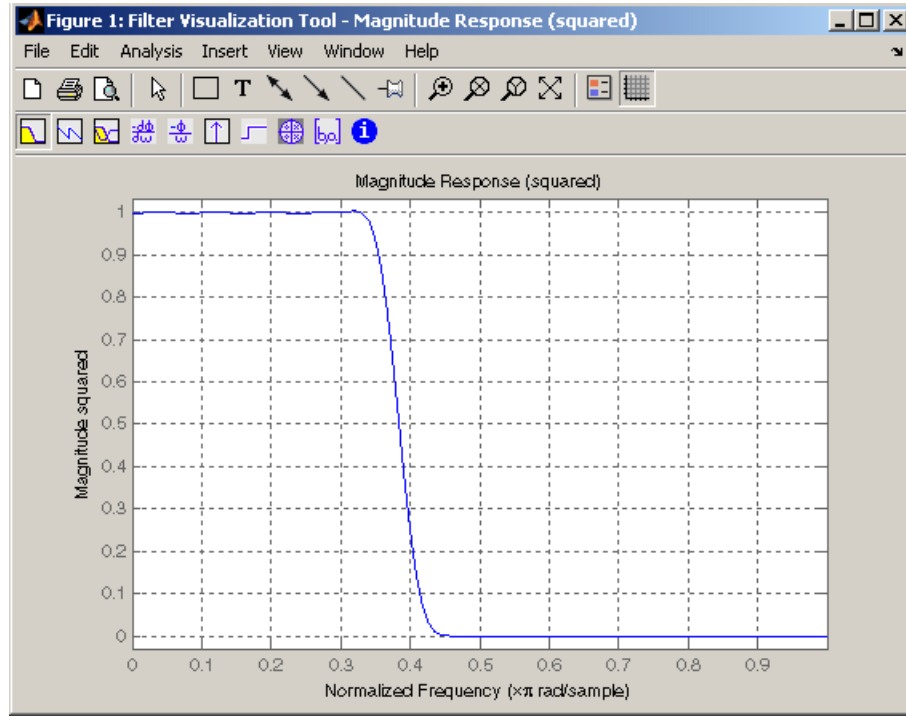
Note that the *y*-axis shown in the figure below is in Magnitude Squared. You can set this by right-clicking on the axis label and selecting **Magnitude Squared** from the menu.



Ringing and ripples occur in the response, especially near the band edge. This “Gibbs effect” does not vanish as the filter length increases, but a nonrectangular window reduces its magnitude. Multiplication by a window in the time domain causes a convolution or smoothing in the frequency domain. Apply a length 51 Hamming window to the filter and display the result using FVTool:

```
b = 0.4*sinc(0.4*(-25:25));
b = b.*hamming(51)';
fvtool(b,1)
```

Note that the *y*-axis shown in the figure below is in Magnitude Squared. You can set this by right-clicking on the axis label and selecting **Magnitude Squared** from the menu.



Using a Hamming window greatly reduces the ringing. This improvement is at the expense of transition width (the windowed version takes longer to ramp from passband to stopband) and optimality (the windowed version does not minimize the integrated squared error).

The functions `fir1` and `fir2` are based on this windowing process. Given a filter order and description of an ideal desired filter, these functions return a windowed inverse Fourier transform of that ideal filter. Both use a Hamming window by default, but they accept any window function. See “Windows” on page 7-2 for an overview of windows and their properties.

Standard Band FIR Filter Design: fir1

`fir1` implements the classical method of windowed linear phase FIR digital filter design. It resembles the IIR filter design functions in that it is formulated to design filters in standard band configurations: lowpass, bandpass, highpass, and bandstop.

The statements

```
n = 50;
Wn = 0.4;
b = fir1(n,Wn);
```

create row vector `b` containing the coefficients of the order `n` Hamming-windowed filter. This is a lowpass, linear phase FIR filter with cutoff frequency `Wn`. `Wn` is a number between 0 and 1, where 1 corresponds to the Nyquist frequency, half the sampling frequency. (Unlike other methods, here `Wn` corresponds to the 6 dB point.) For a highpass filter, simply append the string `'high'` to the function's parameter list. For a bandpass or bandstop filter, specify `Wn` as a two-element vector containing the passband edge frequencies; append the string `'stop'` for the bandstop configuration.

`b = fir1(n,Wn,window)` uses the window specified in column vector `window` for the design. The vector `window` must be `n+1` elements long. If you do not specify a window, `fir1` applies a Hamming window.

Kaiser Window Order Estimation. The `kaiserord` function estimates the filter order, cutoff frequency, and Kaiser window beta parameter needed to meet a given set of specifications. Given a vector of frequency band edges and a corresponding vector of magnitudes, as well as maximum allowable ripple, `kaiserord` returns appropriate input parameters for the `fir1` function.

Multiband FIR Filter Design: fir2

The `fir2` function also designs windowed FIR filters, but with an arbitrarily shaped piecewise linear frequency response. This is in contrast to `fir1`, which only designs filters in standard lowpass, highpass, bandpass, and bandstop configurations.

The commands

```
n = 50;
f = [0 .4 .5 1];
m = [1  1  0 0];
b = fir2(n,f,m);
```

return row vector `b` containing the `n+1` coefficients of the order `n` FIR filter whose frequency-magnitude characteristics match those given by vectors `f` and `m`. `f` is a vector of frequency points ranging from 0 to 1, where 1 represents the Nyquist frequency. `m` is a vector containing the desired magnitude response at the points specified in `f`. (The IIR counterpart of this function is `yulewalk`, which also designs filters based on arbitrary piecewise linear magnitude responses. See "IIR Filter Design" on page 2-4 for details.)

Multiband FIR Filter Design with Transition Bands

The `firls` and `firpm` functions provide a more general means of specifying the ideal desired filter than the `fir1` and `fir2` functions. These functions design Hilbert transformers, differentiators, and other filters with odd symmetric coefficients (type III and type IV linear phase). They also let you include transition or "don't care" regions in which the error is not minimized, and perform band dependent weighting of the minimization.

The `firls` function is an extension of the `fir1` and `fir2` functions in that it minimizes the integral of the square of the error between the desired frequency response and the actual frequency response.

The `firpm` function implements the Parks-McClellan algorithm, which uses the Remez exchange algorithm and Chebyshev approximation theory to design filters with optimal fits between the desired and actual frequency responses. The filters are optimal in the sense that they minimize the maximum error between the desired frequency response and the actual frequency response; they are sometimes called *minimax* filters. Filters designed in this way exhibit an equiripple behavior in their frequency response, and hence are also known as *equiripple* filters. The Parks-McClellan FIR filter design algorithm is perhaps the most popular and widely used FIR filter design methodology.

The syntax for `firls` and `firpm` is the same; the only difference is their minimization schemes. The next example shows how filters designed with `firls` and `firpm` reflect these different schemes.

Basic Configurations

The default mode of operation of `firls` and `firpm` is to design type I or type II linear phase filters, depending on whether the order you desire is even or odd, respectively. A lowpass example with approximate amplitude 1 from 0 to 0.4 Hz, and approximate amplitude 0 from 0.5 to 1.0 Hz is

```
n = 20;                    % Filter order
f = [0 0.4 0.5 1];         % Frequency band edges
a = [1  1  0 0];           % Desired amplitudes
b = firpm(n,f,a);
```

From 0.4 to 0.5 Hz, `firpm` performs no error minimization; this is a transition band or "don't care" region. A transition band minimizes the error more in the bands that you do care about, at the expense of a slower transition rate. In this way, these types of filters have an inherent trade-off similar to FIR design by windowing.

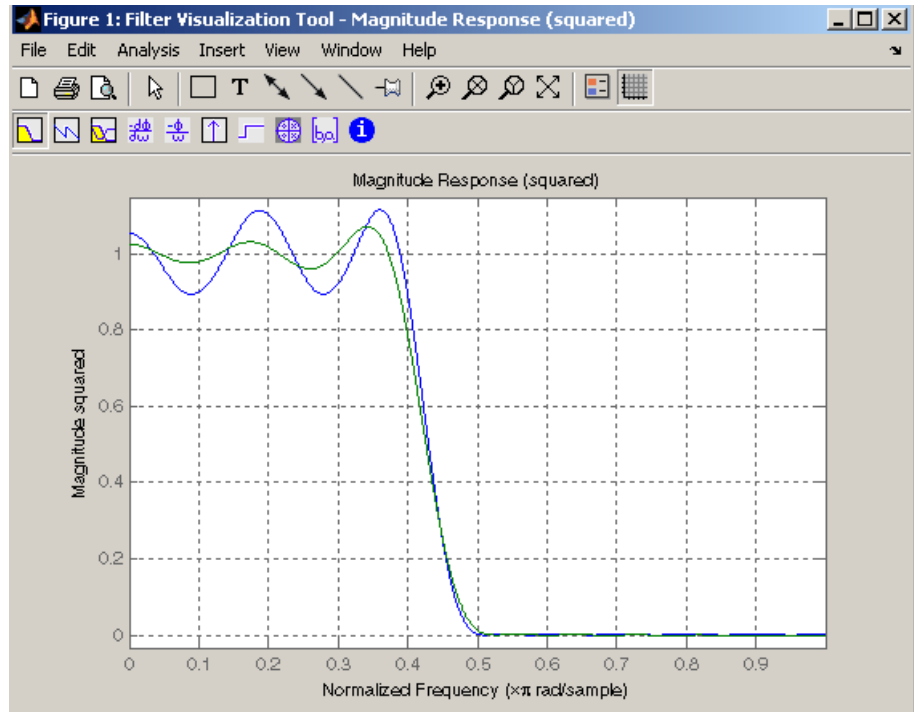
To compare least squares to equiripple filter design, use `firls` to create a similar filter. Type

```
bb = firls(n,f,a);
```

and compare their frequency responses using FVTool:

```
fvtool(b,1,bb,1)
```

Note that the *y*-axis shown in the figure below is in Magnitude Squared. You can set this by right-clicking on the axis label and selecting **Magnitude Squared** from the menu.



The filter designed with `firpm` exhibits equiripple behavior. Also note that the `firls` filter has a better response over most of the passband and stopband, but at the band edges (`f = 0.4` and `f = 0.5`), the response is further away from the ideal than the `firpm` filter. This shows that the `firpm` filter's *maximum* error over the passband and stopband is smaller and, in fact, it is the smallest possible for this band edge configuration and filter length.

Think of frequency bands as lines over short frequency intervals. `firpm` and `firls` use this scheme to represent any piecewise linear desired function with any transition bands. `firls` and `firpm` design lowpass, highpass, bandpass, and bandstop filters; a bandpass example is

```
f = [0 0.3  0.4  0.7  0.8  1];   % Band edges in pairs
```

```
a = [0  0    1    1    0   0];   % Bandpass filter amplitude
```

Technically, these `f` and `a` vectors define five bands:

- Two stopbands, from 0.0 to 0.3 and from 0.8 to 1.0
- A passband from 0.4 to 0.7
- Two transition bands, from 0.3 to 0.4 and from 0.7 to 0.8

Example highpass and bandstop filters are

```
f = [0 0.7  0.8  1];            % Band edges in pairs
a = [0  0    1   1];            % Highpass filter amplitude
f = [0 0.3  0.4  0.5  0.8  1];  % Band edges in pairs
a = [1  1    0    0    1   1];  % Bandstop filter amplitude
```

An example multiband bandpass filter is

```
f = [0 0.1 0.15 0.25 0.3 0.4 0.45 0.55 0.6 0.7 0.75 0.85 0.9 1];
a = [1  1   0    0    1   1   0    0    1   1   0    0    1  1];
```

Another possibility is a filter that has as a transition region the line connecting the passband with the stopband; this can help control "runaway" magnitude response in wide transition regions:

```
f = [0 0.4 0.42 0.48 0.5  1];
a = [1 1 0.8 0.2 0 0];   % Passband, linear transition,
                         %  stopband
```

The Weight Vector

Both `firls` and `firpm` allow you to place more or less emphasis on minimizing the error in certain frequency bands relative to others. To do this, specify a weight vector following the frequency and amplitude vectors. An example lowpass equiripple filter with 10 times less ripple in the stopband than the passband is

```
n = 20;              % Filter order
f = [0 0.4 0.5 1];   % Frequency band edges
a = [1  1   0  0];   % Desired amplitudes
```

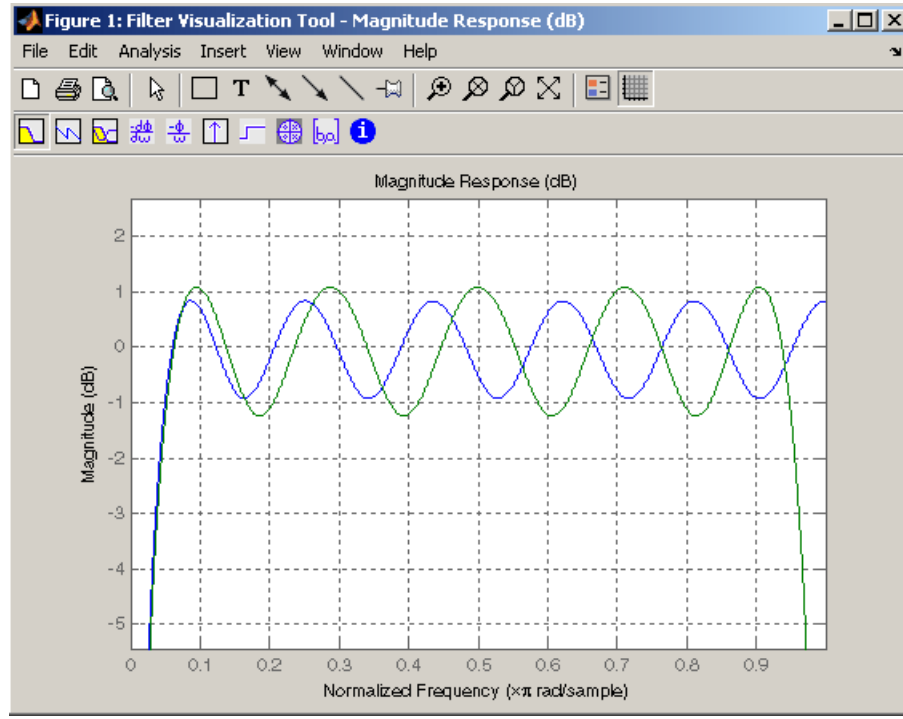
```
w = [1 10];           % Weight vector
b = firpm(n,f,a,w);
```

A legal weight vector is always half the length of the `f` and `a` vectors; there must be exactly one weight per band.

Anti-Symmetric Filters / Hilbert Transformers

When called with a trailing `'h'` or `'Hilbert'` option, `firpm` and `firls` design FIR filters with odd symmetry, that is, type III (for even order) or type IV (for odd order) linear phase filters. An ideal Hilbert transformer has this anti-symmetry property and an amplitude of 1 across the entire frequency range. Try the following approximate Hilbert transformers and plot them using FVTool:

```
b = firpm(21,[0.05 1],[1 1],'h');       % Highpass Hilbert
bb = firpm(20,[0.05 0.95],[1 1],'h');   % Bandpass Hilbert
fvtool(b,1,bb,1)
```



You can find the delayed Hilbert transform of a signal `x` by passing it through these filters.

```
fs = 1000;              % Sampling frequency
t = (0:1/fs:2)';        % Two second time vector
x = sin(2*pi*300*t);    % 300 Hz sine wave example signal
xh = filter(bb,1,x);    % Hilbert transform of x
```

The analytic signal corresponding to `x` is the complex signal that has `x` as its real part and the Hilbert transform of `x` as its imaginary part. For this FIR method (an alternative to the `hilbert` function), you must delay `x` by half the filter order to create the analytic signal:

```
xd = [zeros(10,1); x(1:length(x)-10)];     % Delay 10 samples
xa = xd + j*xh;                           % Analytic signal
```

This method does not work directly for filters of odd order, which require a noninteger delay. In this case, the `hilbert` function, described in "Specialized Transforms" on page 7-40, estimates the analytic signal. Alternatively, use the `resample` function to delay the signal by a noninteger number of samples.

Differentiators

Differentiation of a signal in the time domain is equivalent to multiplication of the signal's Fourier transform by an imaginary ramp function. That is, to differentiate a signal, pass it through a filter that has a response $H(\omega) = j\omega$. Approximate the ideal differentiator (with a delay) using `firpm` or `firls` with a `'d'` or `'differentiator'` option:

```
b = firpm(21,[0 1],[0 pi],'d');
```

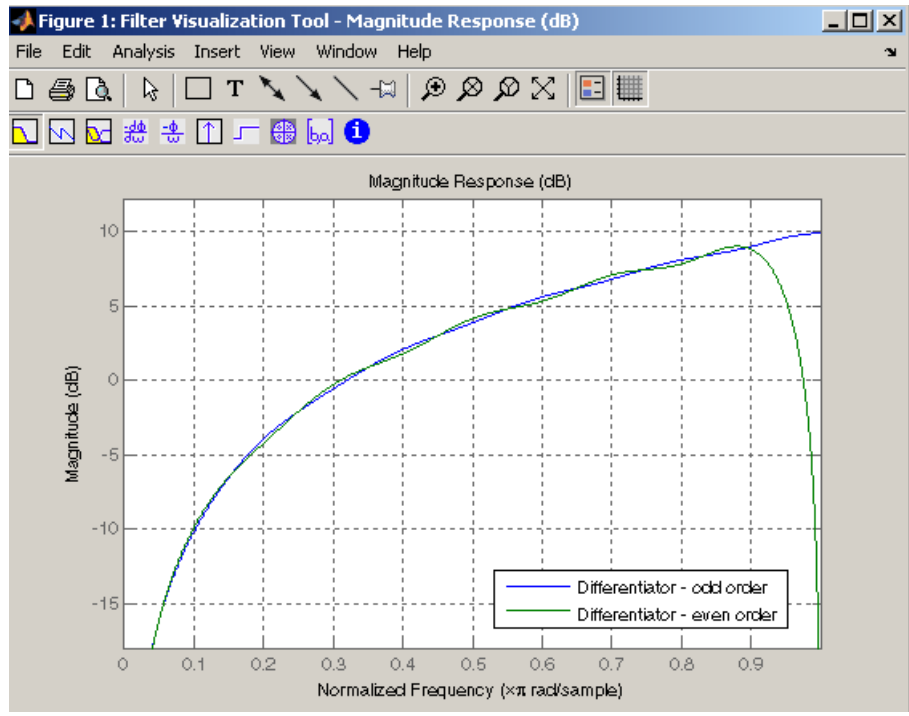
For a type III filter, the differentiation band should stop short of the Nyquist frequency, and the amplitude vector must reflect that change to ensure the correct slope:

```
bb = firpm(20,[0 0.9],[0 0.9*pi],'d');
```

In the `'d'` mode, `firpm` weights the error by $1/\omega$ in nonzero amplitude bands to minimize the maximum *relative* error. `firls` weights the error by $(1/\omega)^2$ in nonzero amplitude bands in the `'d'` mode.

The following plots show the magnitude responses for the differentiators above.

```
fvtool(b,1,bb,1)
```



Constrained Least Squares FIR Filter Design

The Constrained Least Squares (CLS) FIR filter design functions implement a technique that enables you to design FIR filters without explicitly defining the transition bands for the magnitude response. The ability to omit the specification of transition bands is useful in several situations. For example, it may not be clear where a rigidly defined transition band should appear if noise and signal information appear together in the same frequency band. Similarly, it may make sense to omit the specification of transition bands if they appear only to control the results of Gibbs phenomena that appear in the filter's response. See Selesnick, Lang, and Burrus [2] for discussion of this method.

Instead of defining passbands, stopbands, and transition regions, the CLS method accepts a cutoff frequency (for the highpass, lowpass, bandpass, or bandstop cases), or passband and stopband edges (for multiband cases), for the desired response. In this way, the CLS method defines transition regions implicitly, rather than explicitly.

The key feature of the CLS method is that it enables you to define upper and lower thresholds that contain the maximum allowable ripple in the magnitude response. Given this constraint, the technique applies the least square error minimization technique over the frequency range of the filter's response, instead of over specific bands. The error minimization includes any areas of discontinuity in the ideal, "brick wall" response. An additional benefit is that the technique enables you to specify arbitrarily small peaks resulting from Gibbs' phenomena.

There are two toolbox functions that implement this design technique.

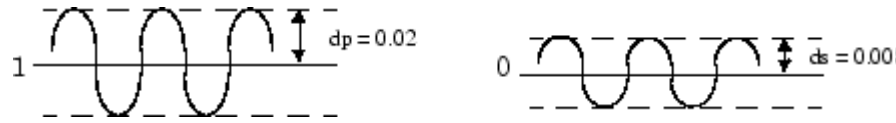
Description	Function
Constrained least square multiband FIR filter design	`fircls`
Constrained least square filter design for lowpass and highpass linear phase filters	`fircls1`

For details on the calling syntax for these functions, see their reference descriptions in the Function Reference.

Basic Lowpass and Highpass CLS Filter Design

The most basic of the CLS design functions, `fircls1`, uses this technique to design lowpass and highpass FIR filters. As an example, consider designing a filter with order 61 impulse response and cutoff frequency of 0.3 (normalized). Further, define the upper and lower bounds that constrain the design process as:

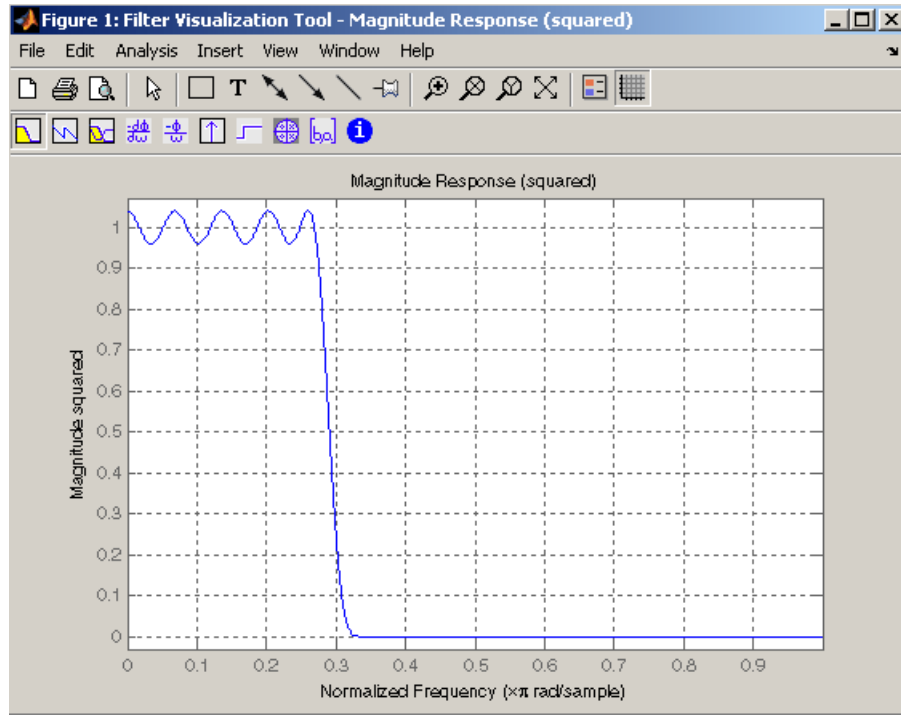
- Maximum passband deviation from 1 (passband ripple) of 0.02.
- Maximum stopband deviation from 0 (stopband ripple) of 0.008.



To approach this design problem using `fircls1`, use the following commands:

```
n = 61;
wo = 0.3;
dp = 0.02;
ds = 0.008;
h = fircls1(n,wo,dp,ds);
fvtool(h,1)
```

Note that the *y*-axis shown below is in Magnitude Squared. You can set this by right-clicking on the axis label and selecting **Magnitude Squared** from the menu.



Multiband CLS Filter Design

`fircls` uses the same technique to design FIR filters with a desired piecewise constant magnitude response. In this case, you can specify a vector of band edges and a corresponding vector of band amplitudes. In addition, you can specify the maximum amount of ripple for each band.

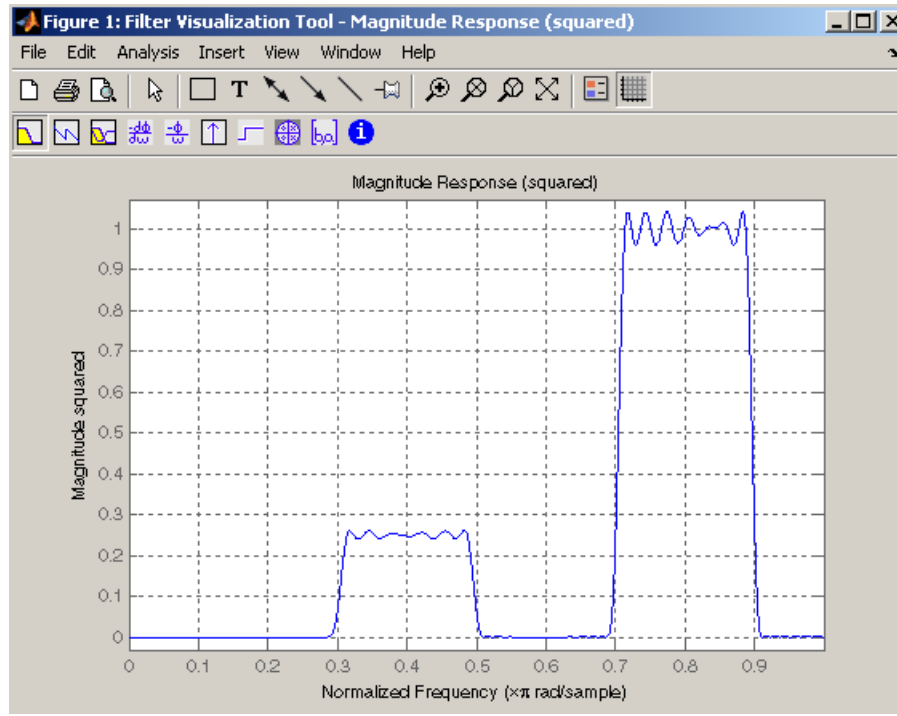
For example, assume the specifications for a filter call for:

- From 0 to 0.3 (normalized): amplitude 0, upper bound 0.005, lower bound -0.005
- From 0.3 to 0.5: amplitude 0.5, upper bound 0.51, lower bound 0.49
- From 0.5 to 0.7: amplitude 0, upper bound 0.03, lower bound -0.03
- From 0.7 to 0.9: amplitude 1, upper bound 1.02, lower bound 0.98
- From 0.9 to 1: amplitude 0, upper bound 0.05, lower bound -0.05

Design a CLS filter with impulse response order 129 that meets these specifications:

```
n = 129;
f = [0 0.3 0.5 0.7 0.9 1];
a = [0 0.5 0 1 0];
up = [0.005 0.51 0.03 1.02 0.05];
lo = [-0.005 0.49 -0.03 0.98 -0.05];
h = fircls(n,f,a,up,lo);
fvtool(h,1)
```

Note that the *y*-axis shown below is in Magnitude Squared. You can set this by right-clicking on the axis label and selecting **Magnitude Squared** from the menu.



Weighted CLS Filter Design

Weighted CLS filter design lets you design lowpass or highpass FIR filters with relative weighting of the error minimization in each band. The `fircls1` function enables you to specify the passband and stopband edges for the least squares weighting function, as well as a constant `k` that specifies the ratio of the stopband to passband weighting.

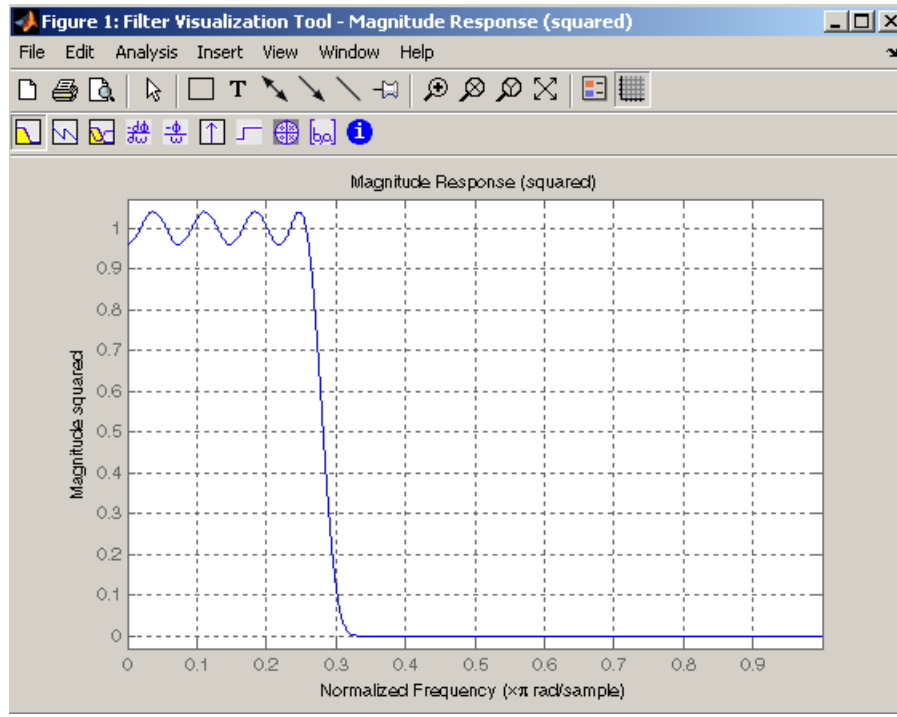
For example, consider specifications that call for an FIR filter with impulse response order of 55 and cutoff frequency of 0.3 (normalized). Also assume maximum allowable passband ripple of 0.02 and maximum allowable stopband ripple of 0.004. In addition, add weighting requirements:

- Passband edge for the weight function of 0.28 (normalized)
- Stopband edge for the weight function of 0.32
- Weight error minimization 10 times as much in the stopband as in the passband

To approach this using `fircls1`, type

```
n = 55;
wo = 0.3;
dp = 0.02;
ds = 0.004;
wp = 0.28;
ws = 0.32;
k = 10;
h = fircls1(n,wo,dp,ds,wp,ws,k);
fvtool(h,1)
```

Note that the *y*-axis shown below is in Magnitude Squared. You can set this by right-clicking on the axis label and selecting **Magnitude Squared** from the menu.



Arbitrary-Response Filter Design

The `cfirpm` filter design function provides a tool for designing FIR filters with arbitrary complex responses. It differs from the other filter design functions in how the frequency response of the filter is specified: it accepts the name of a function which returns the filter response calculated over a grid of frequencies. This capability makes `cfirpm` a highly versatile and powerful technique for filter design.

This design technique may be used to produce nonlinear-phase FIR filters, asymmetric frequency-response filters (with complex coefficients), or more symmetric filters with custom frequency responses.

The design algorithm optimizes the Chebyshev (or minimax) error using an extended Remez-exchange algorithm for an initial estimate. If this exchange method fails to obtain the optimal filter, the algorithm switches to an ascent-descent algorithm that takes over to finish the convergence to the optimal solution.

Multiband Filter Design

Consider a multiband filter with the following special frequency-domain characteristics.

Band	Amplitude	Optimization Weighting
[-1 -0.5]	[5 1]	1
[-0.4 +0.3]	[2 2]	10
[+0.4 +0.8]	[2 1]	5

A linear-phase multiband filter may be designed using the predefined frequency-response function `multiband`, as follows:

```
b = cfirpm(38, [-1 -0.5 -0.4 0.3 0.4 0.8], ...
               {'multiband', [5 1 2 2 2 1]}, [1 10 5]);
```

For the specific case of a multiband filter, we can use a shorthand filter design notation similar to the syntax for `firpm`:

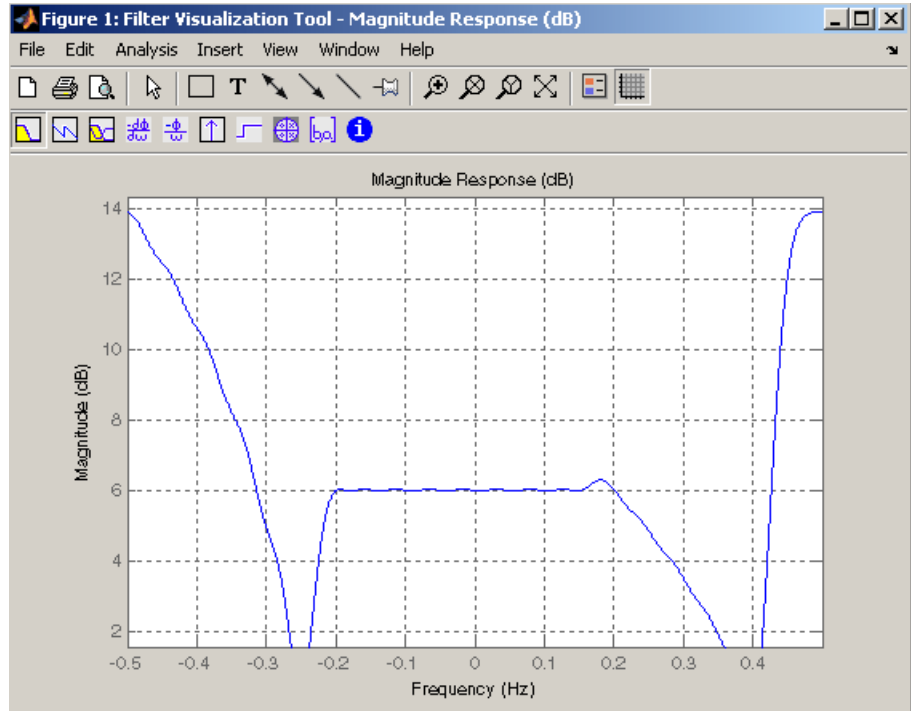
```
b = cfirpm(38,[-1 -0.5 -0.4 0.3 0.4 0.8], ...
              [5 1 2 2 2 1], [1 10 5]);
```

As with `firpm`, a vector of band edges is passed to `cfirpm`. This vector defines the frequency bands over which optimization is performed; note that there are two transition bands, from -0.5 to -0.4 and from 0.3 to 0.4.

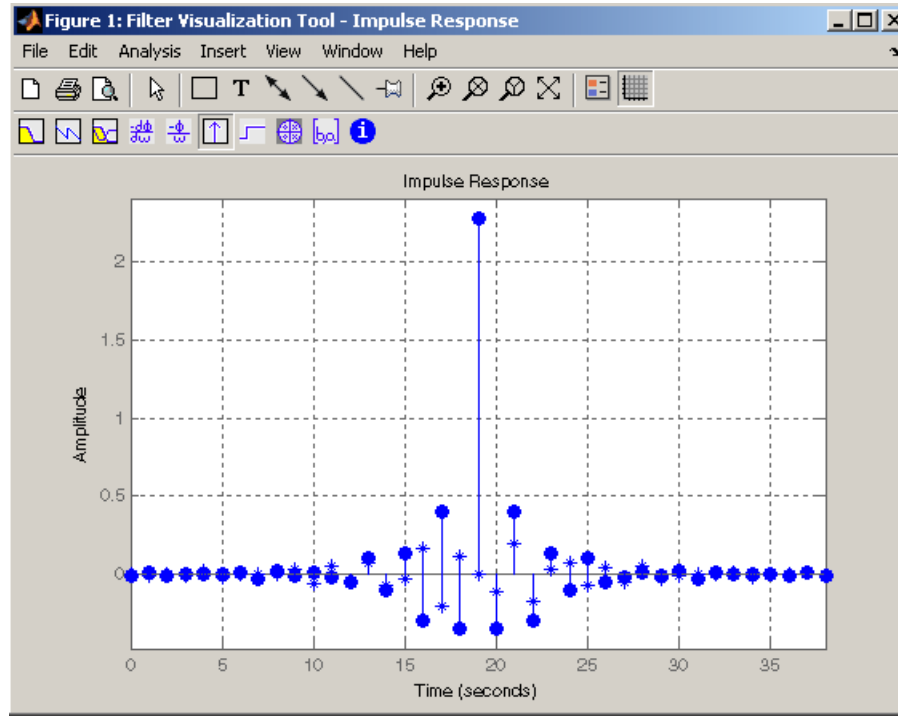
In either case, the frequency response is obtained and plotted using linear scale in FVTool:

```
fvtool(b,1)
```

Note that the range of data shown below is `(-Fs/2,Fs/2)`. You can set this range by changing the *x*-axis units to **Frequency (Fs = 1 Hz)**.



The filter response for this multiband filter is complex, which is expected because of the asymmetry in the frequency domain. The impulse response, which you can select from the FVTool toolbar, is shown below.



Filter Design with Reduced Delay

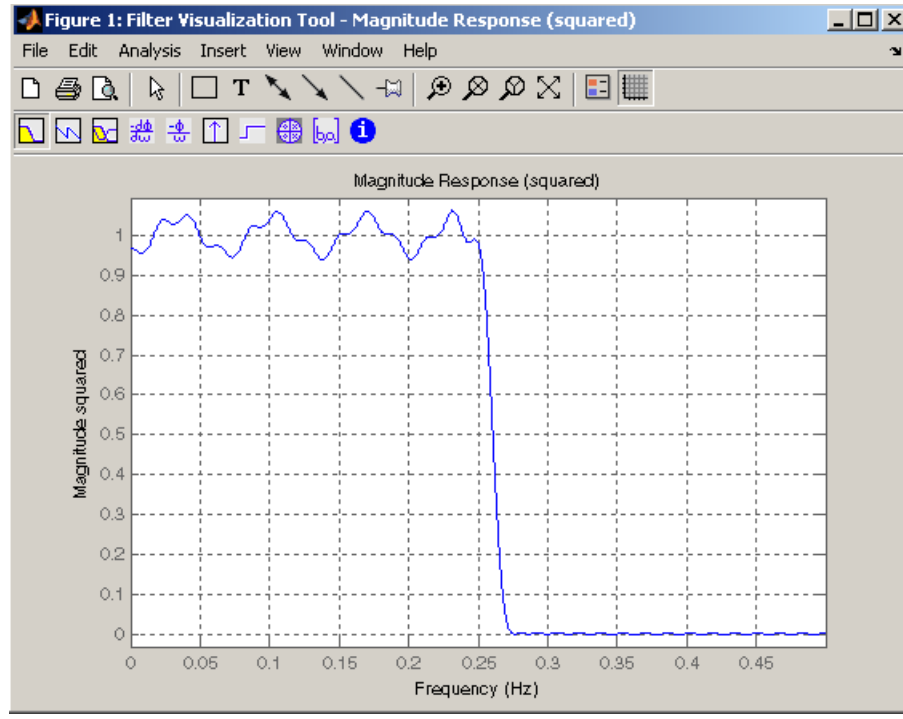
Consider the design of a 62-tap lowpass filter with a half-Nyquist cutoff. If we specify a negative offset value to the `lowpass` filter design function, the group delay offset for the design is significantly less than that obtained for a standard linear-phase design. This filter design may be computed as follows:

```
b = cfirpm(61,[0 0.5 0.55 1],{'lowpass',-16});
```

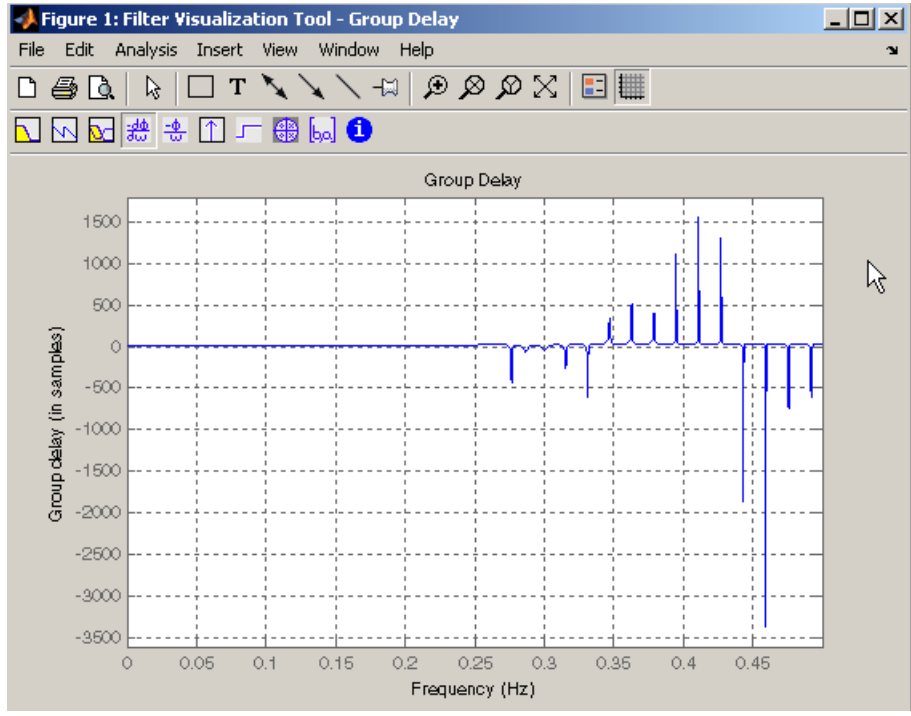
The resulting magnitude response is

```
fvtool(b,1)
```

Note that the range of data in this plot is `(-Fs/2,Fs/2)`, which you can set changing the *x*-axis units to **Frequency**. The *y*-axis is in Magnitude Squared, which you can set by right-clicking on the axis label and selecting **Magnitude Squared** from the menu.



The group delay of the filter reveals that the offset has been reduced from `N/2` to `N/2-16` (i.e., from `30.5` to `14.5`). Now, however, the group delay is no longer flat in the passband region. To create this plot, click the **Group Delay** button on the toolbar.



If we compare this nonlinear-phase filter to a linear-phase filter that has exactly 14.5 samples of group delay, the resulting filter is of order 2*14.5, or 29. Using `b = cfirpm(29,[0 0.5 0.55 1],'lowpass')`, the passband and stopband ripple is much greater for the order 29 filter. These comparisons can assist you in deciding which filter is more appropriate for a specific application.

Special Topics in IIR Filter Design

In this section...

Classic IIR Filter Design

The classic IIR filter design technique includes the following steps.

1. Find an analog lowpass filter with cutoff frequency of 1 and translate this prototype filter to the desired band configuration
2. Transform the filter to the digital domain.
3. Discretize the filter.

The toolbox provides functions for each of these steps.

Design Task	Available functions
Analog lowpass prototype	`buttap`, `cheb1ap`, `besselap`, `ellipap`, `cheb2ap`
Frequency transformation	`lp2lp`, `lp2hp`, `lp2bp`, `lp2bs`
Discretization	`bilinear`, `impinvar`

Alternatively, the `butter`, `cheby1`, `cheb2ord`, `ellip`, and `besself` functions perform all steps of the filter design and the `buttord`, `cheb1ord`, `cheb2ord`, and `ellipord` functions provide minimum order computation for IIR filters. These functions are sufficient for many design problems, and the lower level functions are generally not needed. But if you do have an application where you need to transform the band edges of an analog filter, or discretize a rational transfer function, this section describes the tools with which to do so.

Analog Prototype Design

This toolbox provides a number of functions to create lowpass analog prototype filters with cutoff frequency of 1, the first step in the classical approach to IIR filter design.

The table below summarizes the analog prototype design functions for each supported filter type; plots for each type are shown in "IIR Filter Design" on page 2-4.

Filter Type	Analog Prototype Function
Bessel	`[z,p,k] = besselap(n)`
Butterworth	`[z,p,k] = buttap(n)`
Chebyshev Type I	`[z,p,k] = cheb1ap(n,Rp)`
Chebyshev Type II	`[z,p,k] = cheb2ap(n,Rs)`
Elliptic	`[z,p,k] = ellipap(n,Rp,Rs)`

Frequency Transformation

The second step in the analog prototyping design technique is the frequency transformation of a lowpass prototype. The toolbox provides a set of functions to transform analog lowpass prototypes (with cutoff frequency of 1 rad/s) into bandpass, highpass, bandstop, and lowpass filters of the desired cutoff frequency.

Frequency Transformation	Transformation Function
Lowpass to lowpass $s' = s/\omega_0$	`[numt,dent]   = lp2lp (num,den,Wo)` `[At,Bt,Ct,Dt] = lp2lp (A,B,C,D,Wo)`
Lowpass to highpass $s' = \frac{\omega_0}{s}$	`[numt,dent]   = lp2hp (num,den,Wo)` `[At,Bt,Ct,Dt] = lp2hp (A,B,C,D,Wo)`

Frequency Transformation	Transformation Function
Lowpass to bandpass $s' = \frac{\omega_0}{B_\omega}\frac{(s/\omega_0)^2+1}{s/\omega_0}$	`[numt,dent] = lp2bp (num,den,Wo,Bw)` `[At,Bt,Ct,Dt] = lp2bp (A,B,C,D,Wo,Bw)`
Lowpass to bandstop $s' = \frac{B_\omega}{\omega_0}\frac{s/\omega_0}{(s/\omega_0)^2+1}$	`[numt,dent] = lp2bs (num,den,Wo,Bw)` `[At,Bt,Ct,Dt] = lp2bs( A,B,C,D,Wo,Bw)`

As shown, all of the frequency transformation functions can accept two linear system models: transfer function and state-space form. For the bandpass and bandstop cases

$$\omega_0 = \sqrt{\omega_1\omega_2}$$

and

$$B_\omega = \omega_2 - \omega_1$$

where ω_1 is the lower band edge and ω_2 is the upper band edge.

The frequency transformation functions perform frequency variable substitution. In the case of `lp2bp` and `lp2bs`, this is a second-order substitution, so the output filter is twice the order of the input. For `lp2lp` and `lp2hp`, the output filter is the same order as the input.

To begin designing an order 10 bandpass Chebyshev Type I filter with a value of 3 dB for passband ripple, enter

```
[z,p,k] = cheb1ap(5,3);
```

Outputs `z`, `p`, and `k` contain the zeros, poles, and gain of a lowpass analog filter with cutoff frequency Ω_c equal to 1 rad/s. Use the `lp2bp` function to transform this lowpass prototype to a bandpass analog filter with band edges

$\Omega_1 = \pi/5$ and $\Omega_2 = \pi$. First, convert the filter to state-space form so the `lp2bp` function can accept it:

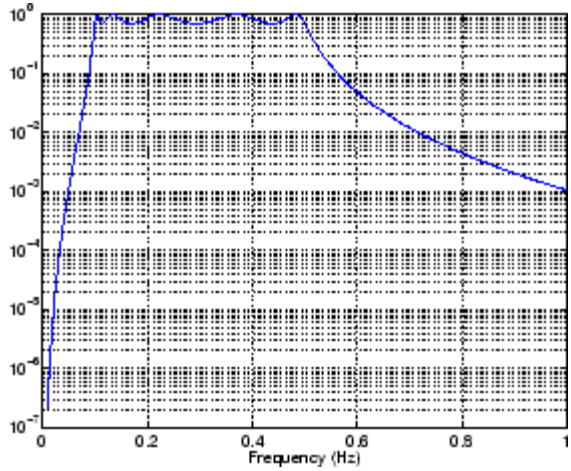
```
[A,B,C,D] = zp2ss(z,p,k);   % Convert to state-space form.
```

Now, find the bandwidth and center frequency, and call `lp2bp`:

```
u1 = 0.1*2*pi;  u2 = 0.5*2*pi;   % In radians per second
Bw = u2-u1;
Wo = sqrt(u1*u2);
[At,Bt,Ct,Dt] = lp2bp(A,B,C,D,Wo,Bw);
```

Finally, calculate the frequency response and plot its magnitude:

```
[b,a] = ss2tf(At,Bt,Ct,Dt);        % Convert to TF form
w = linspace(0.01,1,500)*2*pi;     % Generate frequency vector
h = freqs(b,a,w);                  % Compute frequency response
semilogy(w/2/pi,abs(h)), grid      % Plot log magnitude vs. freq
xlabel('Frequency (Hz)');
```



Filter Discretization

The third step in the analog prototyping technique is the transformation of the filter to the discrete-time domain. The toolbox provides two methods for this: the impulse invariant and bilinear transformations. The filter design

functions `butter`, `cheby1`, `cheby2`, and `ellip` use the bilinear transformation for discretization in this step.

Analog to Digital Transformation	Transformation Function
Impulse invariance	`[numd,dend] = impinvar (num,den,fs)`

Bilinear transform	`[zd,pd,kd] = bilinear (z,p,k,fs,Fp)`
	`[numd,dend] = bilinear (num,den,fs,Fp)`
	`[Ad,Bd,Cd,Dd] = bilinear (At,Bt,Ct,Dt,fs,Fp)`

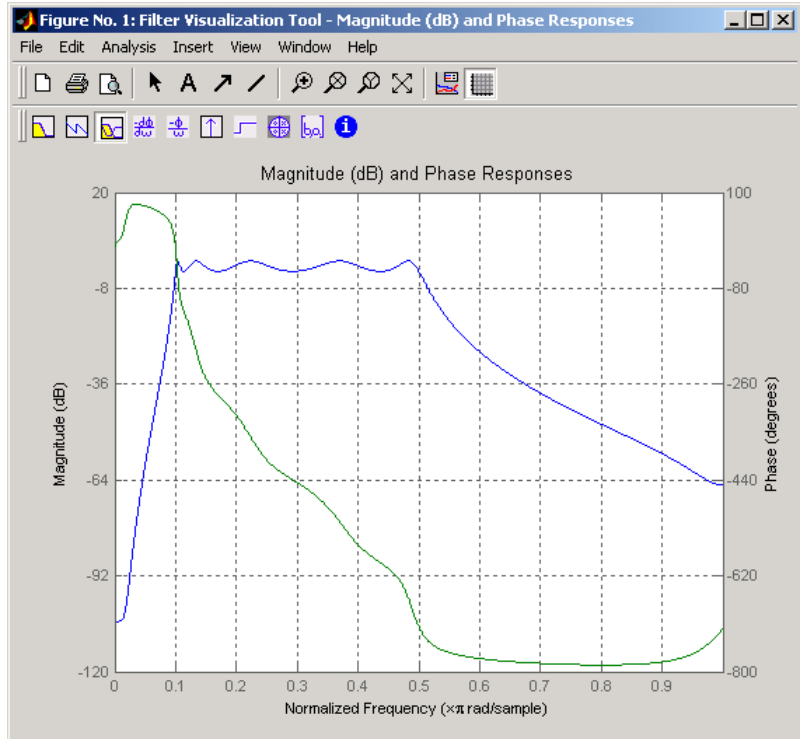
Impulse Invariance

The toolbox function `impinvar` creates a digital filter whose impulse response is the samples of the continuous impulse response of an analog filter. This function works only on filters in transfer function form. For best results, the analog filter should have negligible frequency content above half the sampling frequency, because such high frequency content is aliased into lower bands upon sampling. Impulse invariance works for some lowpass and bandpass filters, but is not appropriate for highpass and bandstop filters.

Design a Chebyshev Type I filter and plot its frequency and phase response using FVTool:

```
[bz,az] = impinvar(b,a,2);
fvtool(bz,az)
```

Click the **Magnitude and Phase Response** toolbar button.



Impulse invariance retains the cutoff frequencies of 0.1 Hz and 0.5 Hz.

Bilinear Transformation

The bilinear transformation is a nonlinear mapping of the continuous domain to the discrete domain; it maps the *s*-plane into the *z*-plane by

$$H(z) = H(s)\Big|_{s = k\frac{z-1}{z+1}}$$

Bilinear transformation maps the $j\Omega$-axis of the continuous domain to the unit circle of the discrete domain according to

$$\omega = 2\tan^{-1}\left(\frac{\Omega}{k}\right)$$

The toolbox function `bilinear` implements this operation, where the frequency warping constant k is equal to twice the sampling frequency (`2*fs`)

by default, and equal to $2\pi f_p / \tan(\pi f_p / f_s)$ if you give `bilinear` a trailing argument that represents a "match" frequency `Fp`. If a match frequency `Fp` (in hertz) is present, `bilinear` maps the frequency $\Omega = 2\pi f_p$ (in rad/s) to the same frequency in the discrete domain, normalized to the sampling rate: $\omega = 2\pi f_p / f_s$ (in rad/sample).

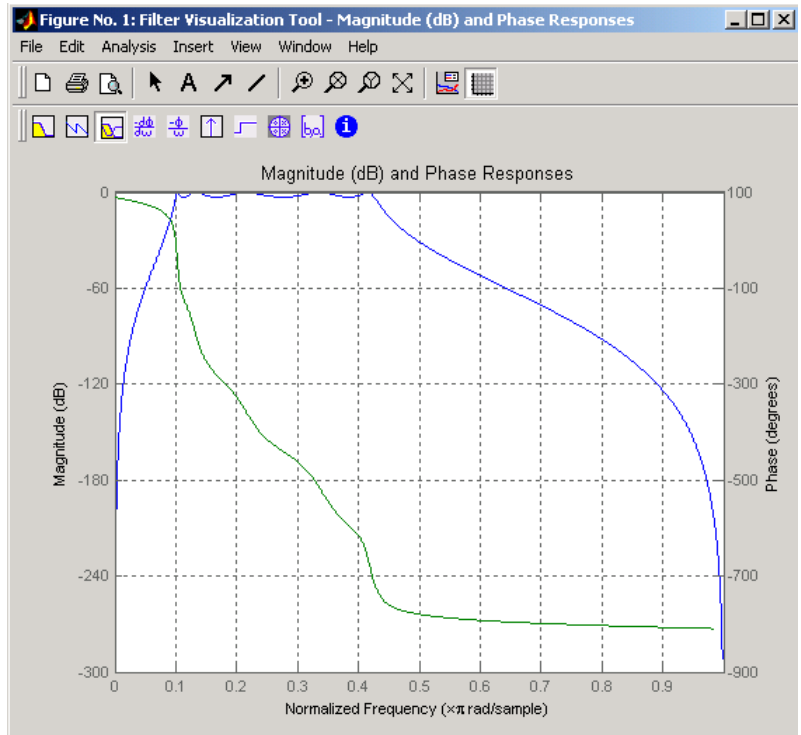
The `bilinear` function can perform this transformation on three different linear system representations: zero-pole-gain, transfer function, and state-space form. Try calling `bilinear` with the state-space matrices that describe the Chebyshev Type I filter from the previous section, using a sampling frequency of 2 Hz, and retaining the lower band edge of 0.1 Hz:

```
[Ad,Bd,Cd,Dd] = bilinear(At,Bt,Ct,Dt,2,0.1);
```

The frequency response of the resulting digital filter is

```
[bz,az] = ss2tf(Ad,Bd,Cd,Dd);       % Convert to TF
fvtool(bz,az)
```

Click the **Magnitude and Phase Response** toolbar button.



The lower band edge is at 0.1 Hz as expected. Notice, however, that the upper band edge is slightly less than 0.5 Hz, although in the analog domain it was exactly 0.5 Hz. This illustrates the nonlinear nature of the bilinear transformation. To counteract this nonlinearity, it is necessary to create analog domain filters with "prewarped" band edges, which map to the correct locations upon bilinear transformation. Here the prewarped frequencies `u1` and `u2` generate `Bw` and `Wo` for the `lp2bp` function:

```
fs = 2;                              % Sampling frequency (hertz)
u1 = 2*fs*tan(0.1*(2*pi/fs)/2);      % Lower band edge (rad/s)
u2 = 2*fs*tan(0.5*(2*pi/fs)/2);      % Upper band edge (rad/s)
Bw = u2 - u1;                        % Bandwidth
Wo = sqrt(u1*u2);                    % Center frequency
[At,Bt,Ct,Dt] = lp2bp(A,B,C,D,Wo,Bw);
```

A digital bandpass filter with correct band edges 0.1 and 0.5 times the Nyquist frequency is

```
[Ad,Bd,Cd,Dd] = bilinear(At,Bt,Ct,Dt,fs);
```

The example bandpass filters from the last two sections could also be created in one statement using the complete IIR design function `cheby1`. For instance, an analog version of the example Chebyshev filter is

```
[b,a] = cheby1(5,3,[0.1 0.5]*2*pi,'s');
```

Note that the band edges are in rad/s for analog filters, whereas for the digital case, frequency is normalized:

```
[bz,az] = cheby1(5,3,[0.1 0.5]);
```

All of the complete design functions call `bilinear` internally. They prewarp the band edges as needed to obtain the correct digital filter.

Selected Bibliography

[1] Karam, L.J., and J.H. McClellan. "Complex Chebyshev Approximation for FIR Filter Design." *IEEE Trans. on Circuits and Systems II*. March 1995.

[2] Selesnick, I.W., and C.S. Burrus. "Generalized Digital Butterworth Filter Design." *Proceedings of the IEEE Int. Conf. Acoust., Speech, Signal Processing*. Vol. 3 (May 1996).

[3] Selesnick, I.W., M. Lang, and C.S. Burrus. "Constrained Least Square Design of FIR Filters without Specified Transition Bands." *Proceedings of the IEEE Int. Conf. Acoust., Speech, Signal Processing*. Vol. 2 (May 1995). Pgs. 1260-1263.

3

Designing a Filter in Fdesign — Process Overview

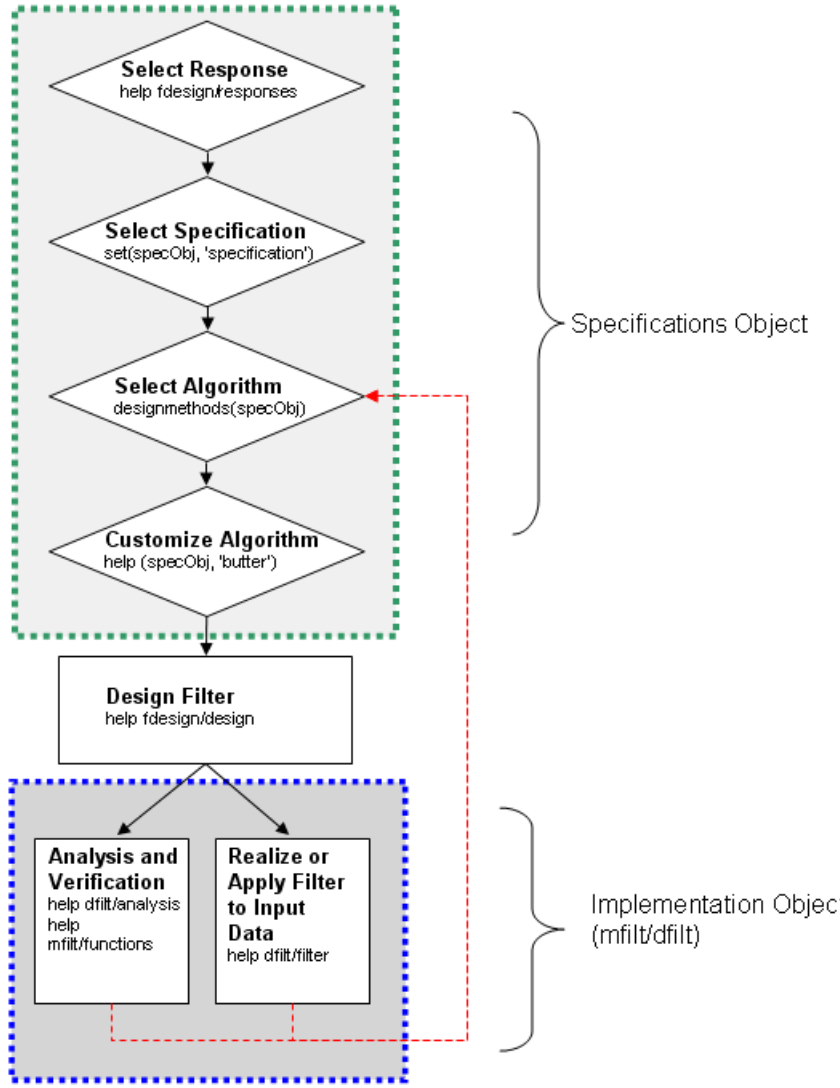
Process Flow Diagram and Filter Design Methodology

In this section...

Note You must minimally have the Signal Processing Toolbox installed to use `fdesign` and `design`. Some of the features described below may be unavailable if your installation does not additionally include the Filter Design Toolbox™ license. The Filter Design Toolbox significantly expands the functionality available for the specification, design, and analysis of filters. You can verify the presence of both toolboxes by typing `ver` at the command prompt.

Exploring the Process Flow Diagram

The process flow diagram shown in the following figure lists the steps and shows the order of the filter design process.



The first four steps of the filter design process relate to the filter Specifications Object, while the last two steps involve the filter Implementation Object. Both of these objects are discussed in more detail in the following sections. Step 5 - the design of the filter, is the transition step from the filter Specifications Object to the Implementation object. The analysis and verification step is

completely optional. It provides methods for the filter designer to ensure that the filter complies with all design criteria. Depending on the results of this verification, you can loop back to steps 3 and 4, to either choose a different algorithm, or to customize the current one. You may also wish to go back to steps 3 or 4 after you filter the input data with the designed filter (step 7), and find that you wish to tweak the filter or change it further.

The diagram shows the help command for each step. Enter the help line at the MATLAB command prompt to receive instructions and further documentation links for the particular step. Not all of the steps have to be executed explicitly. For example, you could go from step 1 directly to step 5, and the interim three steps are done for you by the software.

The following are the details for each of the steps shown above.

Selecting a Response

If you type:

```
help fdesign/responses
```

at the MATLAB command prompt, you see a list of all available filter responses. The responses marked with an asterisk require the Filter Design Toolbox.

You must select a response to initiate the filter. In this example, a bandpass filter Specifications Object is created by typing the following:

```
d = fdesign.bandpass
```

Selecting a Specification

A *specification* is an array of design parameters for a given filter. The specification is a property of the Specifications Object.

Note A specification is not the same as the Specifications Object. A Specifications Object contains a specification as one of its properties.

When you select a filter response, there are a number of different specifications available. Each one contains a different combination of design parameters. After you create a filter Specifications Object, you can query the available specifications for that response. Specifications marked with an asterisk require the Filter Design Toolbox.

```
>> d = fdesign.bandpass; % step 1 - choose the response
>> set (d, 'specification')

ans =

    'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2'
    'N,F3dB1,F3dB2'
    'N,F3dB1,F3dB2,Ap'
    'N,F3dB1,F3dB2,Ast'
    'N,F3dB1,F3dB2,Ast1,Ap,Ast2'
    'N,F3dB1,F3dB2,BWp'
    'N,F3dB1,F3dB2,BWst'
    'N,Fc1,Fc2'
    'N,Fp1,Fp2,Ap'
    'N,Fp1,Fp2,Ast1,Ap,Ast2'
    'N,Fst1,Fp1,Fp2,Fst2'
    'N,Fst1,Fp1,Fp2,Fst2,Ap'
    'N,Fst1,Fst2,Ast'
    'Nb,Na,Fst1,Fp1,Fp2,Fst2'

>> d=fdesign.arbmag;
>> set(d,'specification')

ans =

    'N,F,A'
    'N,B,F,A'
```

The `set` command can be used to select one of the available specifications as follows:

```
>> d = fdesign.lowpass; % step 1
>> % step 2: get a list of available specifications
>> set (d, 'specification')
```

```
ans =

    'Fp,Fst,Ap,Ast'
    'N,F3dB'
    'N,F3dB,Ap'
    'N,F3dB,Ap,Ast'
    'N,F3dB,Ast'
    'N,F3dB,Fst'
    'N,Fc'
    'N,Fc,Ap,Ast'
    'N,Fp,Ap'
    'N,Fp,Ap,Ast'
    'N,Fp,F3dB'
    'N,Fp,Fst'
    'N,Fp,Fst,Ap'
    'N,Fp,Fst,Ast'
    'N,Fst,Ap,Ast'
    'N,Fst,Ast'
    'Nb,Na,Fp,Fst'

>> %step 2: set the required specification
>> set (d, 'specification', 'N,Fc')
```

If you do not perform this step explicitly, `fdesign` returns the default specification for the response you chose in "Select a Response" on page 4-3, and provides default values for all design parameters included in the specification.

Selecting an Algorithm

The availability of algorithms depends the chosen filter response, the design parameters, and the availability of the Filter Design Toolbox. In other words, for the same lowpass filter, changing the specification string also changes the available algorithms. In the following example, for a lowpass filter and a specification of `'N, Fc'`, only one algorithm is available—`window`.

```
>> %step 2: set the required specification
>> set (d, 'specification', 'N,Fc')
>> designmethods (d) %step3: get available algorithms
```

```
Design Methods for class fdesign.lowpass (N,Fc):


window
```

However, for a specification of `'Fp,Fst,Ap,Ast'`, a number of algorithms are available. If the user has only the Signal Processing Toolbox installed, the following algorithms are available:

```
>>set (d, 'specification', 'Fp,Fst,Ap,Ast')
>>designmethods(d)

Design Methods for class fdesign.lowpass (Fp,Fst,Ap,Ast):


butter
cheby1
cheby2
ellip
equiripple
kaiserwin
```

If the user additionally has the Filter Design Toolbox installed, the number of available algorithms for this response and specification string increases:

```
>>set(d,'specification','Fp,Fst,Ap,Ast')
>>designmethods(d)


Design Methods for class fdesign.lowpass (Fp,Fst,Ap,Ast):


butter
cheby1
cheby2
ellip
equiripple
ifir
kaiserwin
```

```
multistage
```

The user chooses a particular algorithm and implements the filter with the `design` function.

```
>>Hd=design(d,'butter');
```

The preceding code actually creates the filter, where `Hd` is the filter Implementation Object. This concept is discussed further in the next step.

If you do not perform this step explicitly, `design` automatically selects the optimum algorithm for the chosen response and specification.

Customizing the Algorithm

The customization options available for any given algorithm depend not only on the algorithm itself, selected in "Selecting an Algorithm" on page 3-6, but also on the specification selected in "Selecting a Specification" on page 3-4. To explore all the available options, type the following at the MATLAB command prompt:

```
help (d, 'algorithm-name')
```

where `d` is the Filter Specification Object, and `algorithm-name` is the name of the algorithm in single quotes, such as `'butter'` or `'cheby1'`.

The application of these customization options takes place while "Designing the Filter" on page 3-8, because these options are the properties of the filter Implementation Object, not the Specification Object.

If you do not perform this step explicitly, the optimum algorithm structure is selected.

Designing the Filter

This next task introduces a new object, the Filter Object, or `dfilt`. To create a filter, use the `design` command:

```
>> % design filter w/o specifying the algorithm
>> Hd = design(d);
```

where `Hd` is the Filter Object and `d` is the Specifications Object. This code creates a filter without specifying the algorithm. When the algorithm is not specified, the software selects the best available one.

To apply the algorithm chosen in "Selecting an Algorithm" on page 3-6, use the same `design` command, but specify the Butterworth algorithm as follows:

```
>> Hd = design(d, 'butter');
```

where `Hd` is the new Filter Object, and `d` is the Specifications Object.

To obtain help and see all the available options, type:

```
>> help fdesign/design
```

This help command describes not only the options for the `design` command itself, but also options that pertain to the method or the algorithm. If you are customizing the algorithm, you apply these options in this step. In the following example, you design a bandpass filter, and then modify the filter structure:

```
>> Hd = design(d, 'butter', 'filterstructure', 'df2sos')

f =

  FilterStructure: 'Direct-Form II, Second-Order Sections'
       Arithmetic: 'double'
        sosMatrix: [7x6 double]
     ScaleValues: [8x1 double]
PersistentMemory: false
```

The filter design step, just like the first task of choosing a response, must be performed explicitly. A Filter Object is created only when `design` is called.

Design Analysis

After the filter is designed you may wish to analyze it to determine if the filter satisfies the design criteria. Filter analysis is broken into three main sections:

- Frequency domain analysis — Includes the magnitude response, group delay, and pole-zero plots.

- Time domain analysis — Includes impulse and step response
- Implementation analysis — Includes quantization noise and cost

To display help for analysis of a discrete-time filter, type:

```
>> help dfilt/analysis
```

To display help for analysis of a multirate filter, type:

```
>> help mfilt/functions
```

To display help for analysis of a farrow filter, type:

```
>> help farrow/functions
```

To analyze your filter, you must explicitly perform this step.

Realize or Apply the Filter to Input Data

After the filter is designed and optimized, it can be used to filter actual input data. The basic filter command takes input data `x`, filters it through the Filter Object, and produces output `y`:

```
>> y = filter (FilterObj, x)
```

This step is never automatically performed for you. To filter your data, you must explicitly execute this step. To understand how the filtering commands work, type:

```
>> help dfilt/filter
```

Note If you have Simulink®, you have the option of exporting this filter to a Simulink block using the `realizemdl` command. To get help on this command, type:

```
>> help realizemdl
```

4

Designing a Filter in the Filterbuilder GUI

The Graphical Interface to Fdesign

In this section...

Introduction to Filterbuilder

The `filterbuilder` function provides a graphical interface to the `fdesign` object-object oriented filter design paradigm and is intended to reduce development time during the filter design process. `filterbuilder` uses a specification-centered approach to find the best algorithm for the desired response.

Note `filterbuilder` requires the Signal Processing Toolbox. The functionality of `filterbuilder` is greatly expanded by the Filter Design Toolbox. Many of the features described or displayed below are only available if the Filter Design Toolbox is installed. You may verify your installation by typing `ver` at the command prompt.

Filterbuilder Design Process

The design process when using `filterbuilder` is similar to the process outlined in the section titled Chapter 3, "Designing a Filter in Fdesign — Process Overview"in the Getting Started guide. The idea is to choose the constraints and specifications of the filter, and to use those as a starting point in the design. Postponing the choice of algorithm for the filter allows the best design method to be determined automatically, based upon the desired

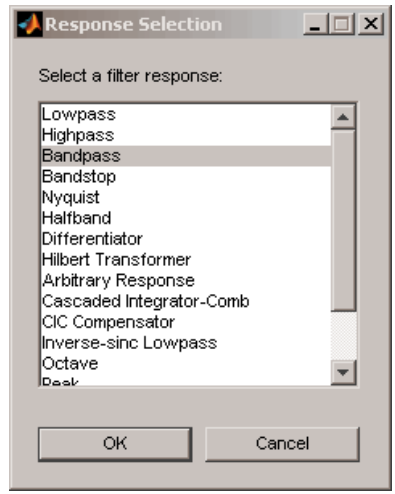
performance criteria. The following are the details of each of the steps for designing a filter with `filterbuilder`.

Select a Response

When you open the `filterbuilder` tool by typing:

```
filterbuilder
```

at the MATLAB command prompt, the **Response Selection** dialog box appears, listing all possible filter responses available in Filter Design Toolbox software.

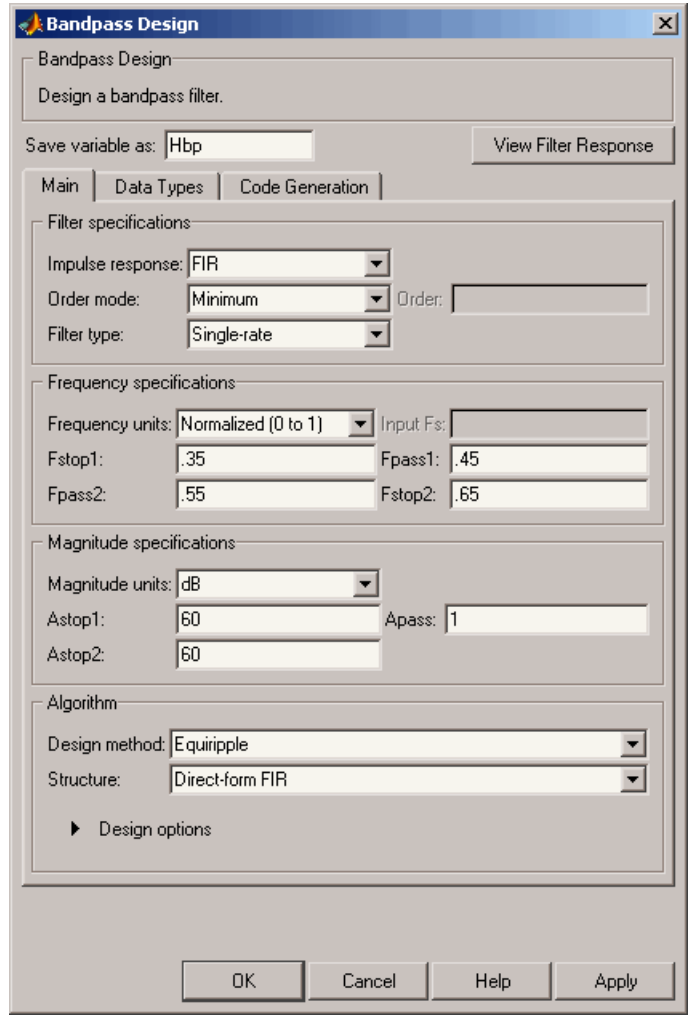


Note This step cannot be skipped because it is not automatically completed for you by the software. You must select a response to initiate the filter design process.

After you choose a response, say bandpass, you start the design of the Specifications Object, and the Bandpass Design dialog box appears. This dialog box contains a **Main** pane, a **Data Types** pane and a **Code Generation** pane. The specifications of your filter are generally set in the **Main** pane of the dialog box.

The **Data Types** pane provides settings for precision and data types, and the **Code Generation** pane contains options for various implementations of the completed filter design.

For the initial design of your filter, you will mostly use the **Main** pane.



The **Bandpass Design** dialog box contains all the parameters you need to determine the specifications of a bandpass filter. The parameters listed in the **Main** pane depend upon the type of filter you are designing. However, no matter what type of filter you have chosen in the **Response Selection** dialog box, the filter design dialog box contains the **Main**, **Data Types**, and **Code Generation** panes.

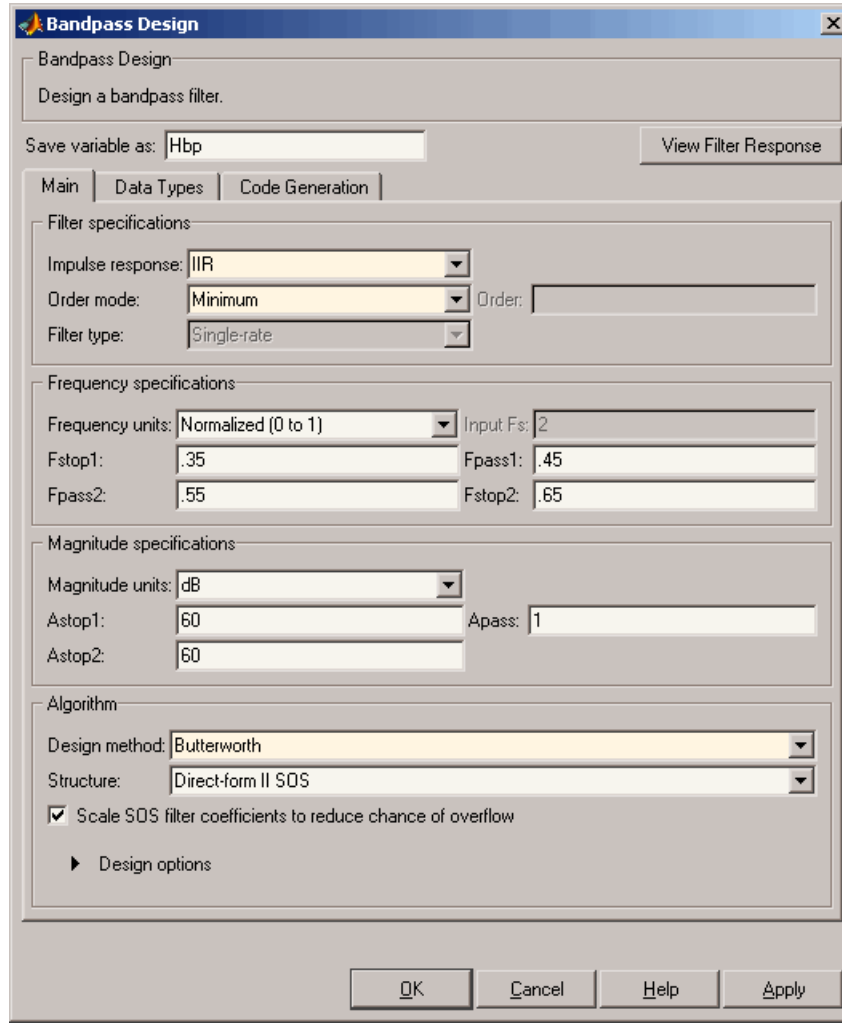
Select a Specification

To choose the specification for the bandpass filter, you can begin by selecting an **Impulse Response**, **Order Mode**, and **Filter Type** in the **Filter Specifications** frame of the **Main Pane**. You can further specify the response of your filter by setting frequency and magnitude specifications in the appropriate frames on the **Main Pane**.

Note **Frequency**, **Magnitude**, and **Algorithm** specifications are interdependent and may change based upon your **Filter Specifications** selections. When choosing specifications for your filter, select your Filter Specifications first and work your way down the dialog box- this approach ensures that the best settings for dependent specifications display as available in the dialog box.

Select an Algorithm

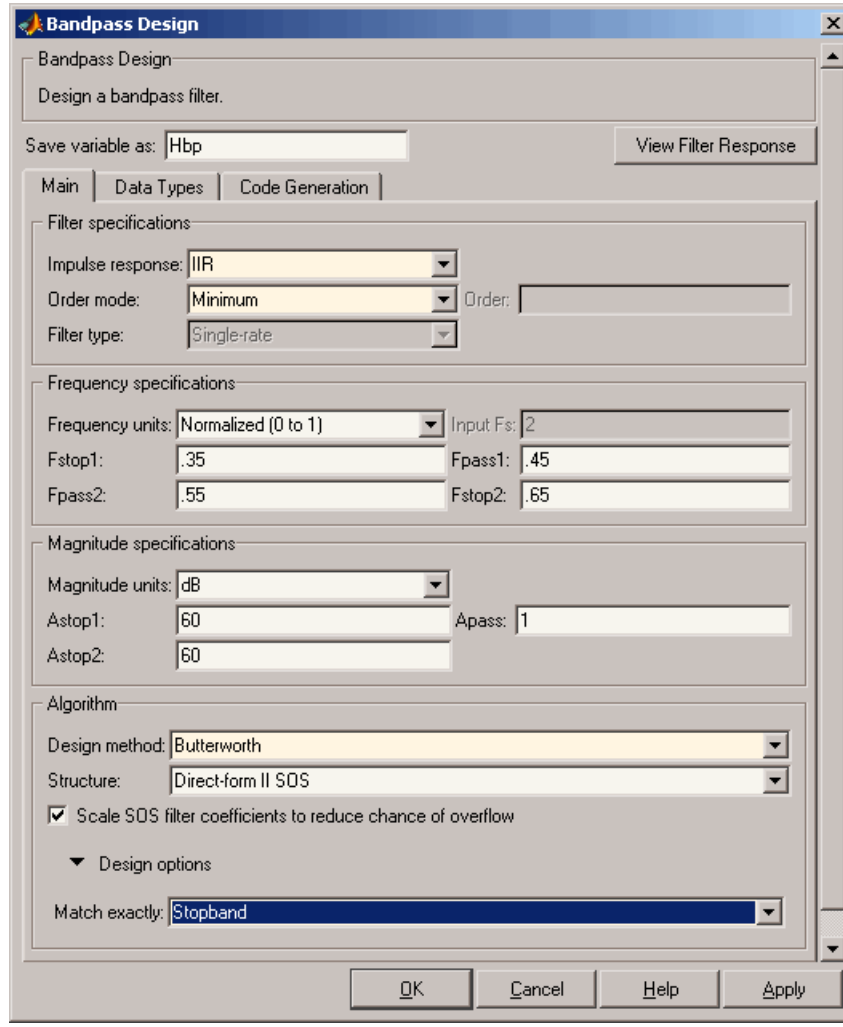
The algorithms available for your filter depend upon the filter response and design parameters you have selected in the previous steps. For example, in the case of a bandpass filter, if the impulse response selected is IIR and the **Order Mode** field is set to`Minimum`, the design methods available is `Butterworth`, `Chebyshev` type I or II, or `Elliptic`, whereas if the **Order Mode** field is set to `Specify`, the design method available is `IIR least p-norm`.



Customize the Algorithm

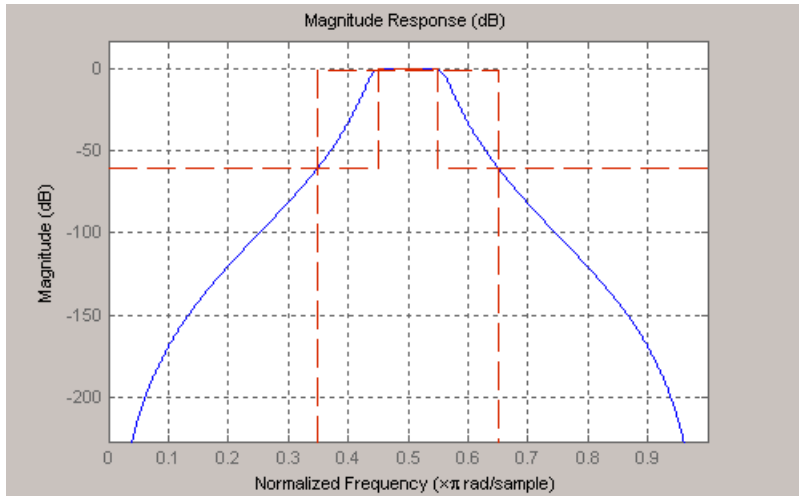
By expanding the **Design options** section of the **Algorithm** frame, you can further customize the algorithm specified. The options available will depend upon the algorithm and settings that have already been selected in the dialog box. In the case of a bandpass IIR filter using the `Butterworth`

method, design options such as **Match Exactly** are available, as shown in the following figure.



Analyze the Design

To analyze the filter response, click on the View Filter Response button. The Filter Visualization Tool opens displaying the magnitude plot of the filter response.



Realize or Apply the Filter to Input Data

When you have achieved the desired filter response through design iterations and analysis using the **Filter Visualization Tool**, apply the filter to the input data. Again, this step is never automatically performed for you by the software. To filter your data, you must explicitly execute this step. In the **Filter Visualization Tool**, click OK and Filter Design Toolbox software creates the filter object with the name specified in the **Save variable as** field and exports it to the MATLAB workspace.

The filter is then ready to be used to filter actual input data. The basic filter command takes input data `x`, filters it through the Filter Object, and produces output `y`:

```
>> y = filter (FilterObj, x)
```

To understand how the filtering commands work, type:

```
>> help dfilt/filter
```

Tip If you have Simulink, you have the option of exporting this filter to a Simulink block using the `realizemdl` command. To get help on this command, type:

```
>> help realizemdl
```

Designing a FIR Filter Using filterbuilder

FIR Filter Design

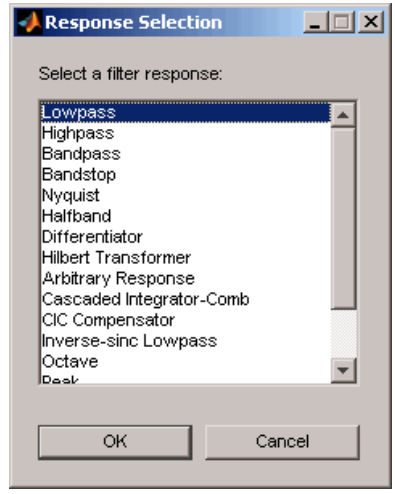
Example – Using Filterbuilder to Design a Finite Impulse Response (FIR) Filter

To design a lowpass FIR filter using `filterbuilder`:

1. Open the Filterbuilder GUI by typing the following at the MATLAB prompt:

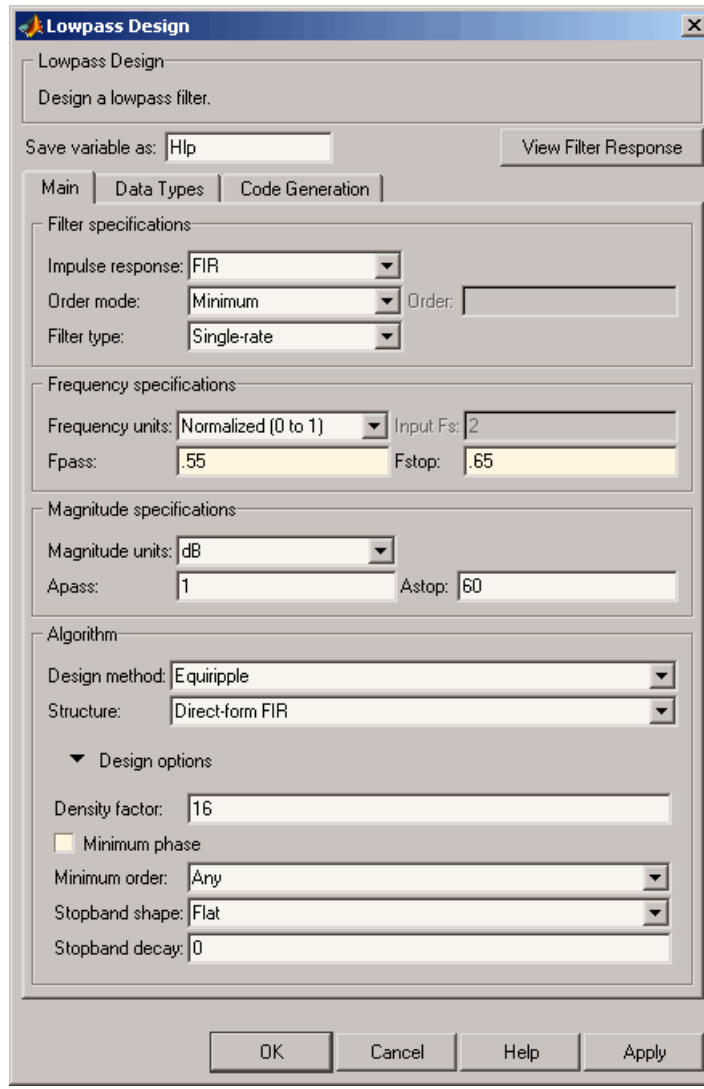
   ```
   filterbuilder
   ```

   The **Response Selection** dialog box appears. In this dialog box, you can select from a list of filter response types. Select `Lowpass` in the list box.



2. Hit the **OK** button. The **Lowpass Design** dialog box opens. Here you can specify the writable parameters of the Lowpass filter object. The components of the **Main** frame of this dialog box are described in the section titled *Lowpass Filter Design Dialog Box — Main Pane*. In the dialog box, make the following changes:

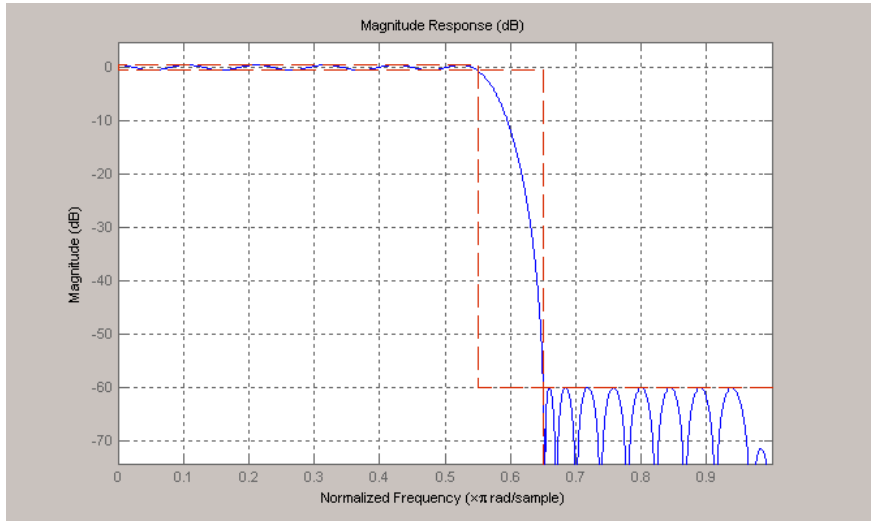
   - Enter a Fpass value of 0.55.
   - Enter a Fstop value of 0.65.



3 Click **Apply**, and the following message appears at the MATLAB prompt:

```
The variable 'Hlp' has been exported to the command window.
```

4 To check your design, click **View Filter Response**. The Filter Visualization tool appears, showing a plot of the magnitude response of the filter.



You can change the design and click **Apply**, followed by **View Filter Response**, as many times as needed until your design specifications are met.

5

FDATool: A Filter Design and Analysis GUI

Overview

In this section...

Introduction to FDA Tool

The Filter Design and Analysis Tool (FDATool) is a powerful user interface for designing and analyzing filters quickly. FDATool enables you to design digital FIR or IIR filters by setting filter specifications, by importing filters from your MATLAB workspace, or by adding, moving or deleting poles and zeros. FDATool also provides tools for analyzing filters, such as magnitude and phase response and pole-zero plots.

Integrated Products

FDATool seamlessly integrates additional functionality from other MathWorks™ products as described in the following table.

Product	Added Features
Target Support Package™ TC6	Download code to TI's C2000™ DSP target board
Filter Design HDL Coder™	Generate synthesizable VHDL or Verilog for fixed-point filters

Product	Added Features
Filter Design Toolbox	• Advanced FIR and IIR design techniques (see "Advanced Filter Design Methods" on page 5-4) • Filter transformations • Multirate filters • Fixed-point filters (available only with Simulink® Fixed Point™ product)
Embedded IDE Link™ CC	Export code usable by Code Composer Studio™ software
"Signal Processing Blockset"	Generate equivalent Signal Processing Blockset™ block for the filter
Simulink	Generate filters from atomic Simulink blocks

Filter Design Methods

FDATool gives you access to the following Signal Processing Toolbox filter design methods.

Design Method	Function
Butterworth	`butter`
Chebyshev Type I	`cheby1`
Chebyshev Type II	`cheby2`
Elliptic	`ellip`
Maximally Flat	`maxflat`
Equiripple	`firpm`
Least-squares	`firls`
Constrained least-squares	`fircls`

Design Method	Function
Complex equiripple	`cfirpm`
Window	`fir1`

When using the `window` method in FDATool, all Signal Processing Toolbox window functions are available, and you can specify a user-defined window by entering its function name and input parameter.

Advanced Filter Design Methods

The following advanced filter design methods are available if you have Filter Design Toolbox product installed.

Design Method	Function
Constrained equiripple FIR	`firceqrip`
Constrained-band equiripple FIR	`fircband`
Generalized remez FIR	`firgr`
Equripple halfband FIR	`firhalfband`
Least P-norm optimal FIR	`firlpnorm`
Equiripple Nyquist FIR	`firnyquist`
Interpolated FIR	`ifir`
IIR comb notching or peaking	`iircomb`
Allpass filter (given group delay)	`iirgrpdelay`
Least P-norm optimal IIR	`iirlpnorm`
Constrained least P-norm IIR	`iirlpnormc`
Second-order IIR notch	`iirnotch`
Second-order IIR peaking (resonator)	`iirpeak`

Using the Filter Design and Analysis Tool

There are different ways that you can design filters using the Filter Design and Analysis Tool. For example:

- You can first choose a response type, such as bandpass, and then choose from the available FIR or IIR filter design methods.
- You can specify the filter by its type alone, along with certain frequency- or time-domain specifications such as passband frequencies and stopband frequencies. The filter you design is then computed using the default filter design method and filter order.

Analyzing Filter Responses

Once you have designed your filter, you can display the filter coefficients and detailed filter information, export the coefficients to the MATLAB workspace, and create a C header file containing the coefficients, and analyze different filter responses in FDATool or in a separate Filter Visualization Tool (`fvtool`). The following filter responses are available:

- Magnitude response (`freqz`)
- Phase response (`phasez`)
- Group delay (`grpdelay`)
- Phase delay (`phasedelay`)
- Impulse response (`impz`)
- Step response (`stepz`)
- Pole-zero plots (`zplane`)
- Zero-phase response (`zerophase`)

Filter Design and Analysis Tool Panels

The Filter Design and Analysis Tool has sidebar buttons that display particular panels in the lower half of the tool. The panels are

- Design Filter. See “Choosing a Filter Design Method” on page 5-9 for more information. You use this panel to
  - Design filters from scratch.
  - Modify existing filters designed in FDATool.
  - Analyze filters.

- Import filter. See “Importing a Filter Design” on page 5-30 for more information. You use this panel to
  - Import previously saved filters or filter coefficients that you have stored in the MATLAB workspace.
  - Analyze imported filters.
- Pole/Zero Editor. See “Editing the Filter Using the Pole/Zero Editor” on page 5-23. You use this panel to add, delete, and move poles and zeros in your filter design.

If you also have Filter Design Toolbox product installed, additional panels are available:

- Set quantization parameters — Use this panel to quantize double-precision filters that you design in FDATool, quantize double-precision filters that you import into FDATool, and analyze quantized filters.
- Transform filter — Use this panel to change a filter from one response type to another.
- Multirate filter design — Use this panel to create a multirate filter from your existing FIR design, create CIC filters, and linear and hold interpolators.

If you have Simulink installed, this panel is available:

- Realize Model — Use this panel to create a Simulink block containing the filter structure. See “Exporting to a Simulink Model” on page 5-38 for more information.

Getting Help

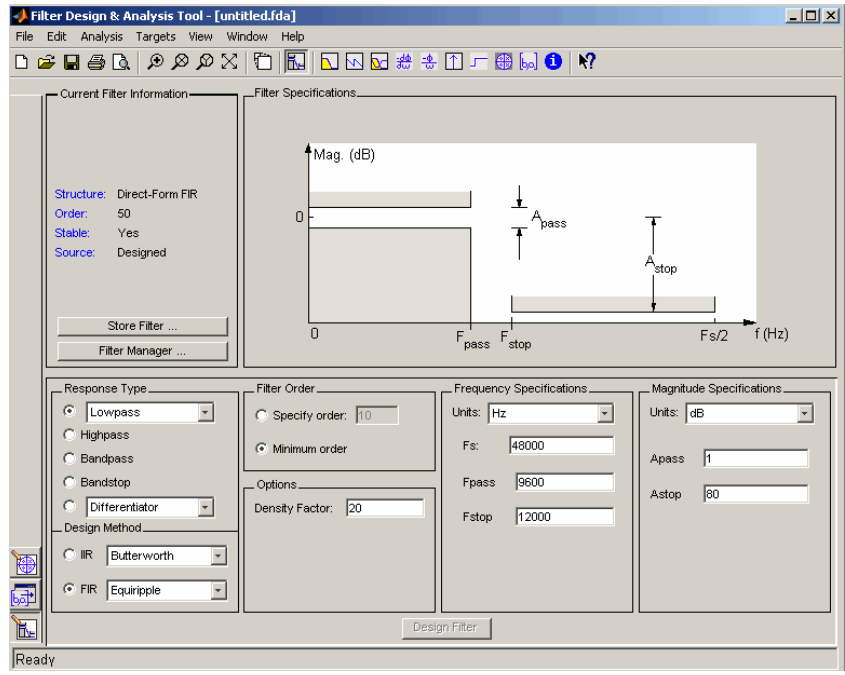
At any time, you can right-click or click the **What's this?** button,  , to get information on the different parts of the tool. You can also use the **Help** menu to see complete Help information.

Opening FDATool

To open the Filter Design and Analysis Tool (FDATool), type

```
fdatool
```

The Filter Design and Analysis Tool opens with the Design Filter panel displayed.



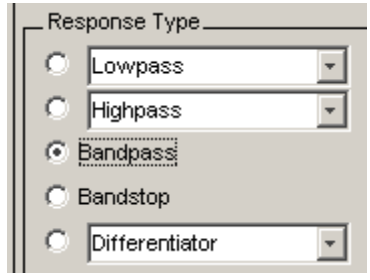
Choosing a Response Type

You can choose from several response types:

- Lowpass
- Raised cosine
- Highpass
- Bandpass
- Bandstop
- Differentiator
- Multiband
- Hilbert transformer
- Arbitrary magnitude

Additional response types are available if you have Filter Design Toolbox product installed.

To design a bandpass filter, select the radio button next to **Bandpass** in the Response Type region of the GUI.

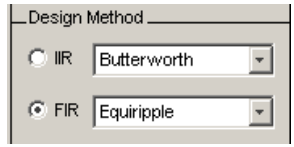


Note Not all filter design methods are available for all response types. Once you choose your response type, this may restrict the filter design methods available to you. Filter design methods that are not available for a selected response type are removed from the Design Method region of the GUI.

Choosing a Filter Design Method

You can use the default filter design method for the response type that you've selected, or you can select a filter design method from the available FIR and IIR methods listed in the GUI.

To select the Remez algorithm to compute FIR filter coefficients, select the **FIR** radio button and choose `Equiripple` from the list of methods.



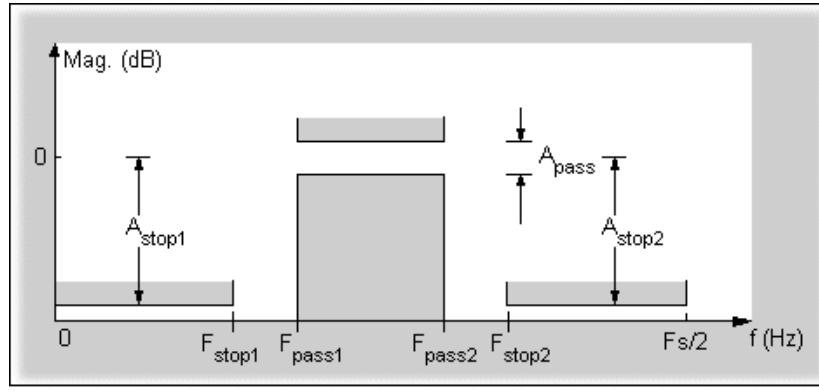
Setting the Filter Design Specifications

In this section...

Viewing Filter Specifications

The filter design specifications that you can set vary according to response type and design method. The display region illustrates filter specifications when you select **Analysis > Filter Specifications** or when you click the **Filter Specifications** toolbar button.

You can also view the filter specifications on the Magnitude plot of a designed filter by selecting **View > Specification Mask**.

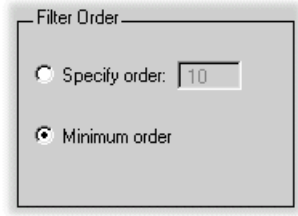


Filter Order

You have two mutually exclusive options for determining the filter order when you design an equiripple filter:

- **Specify order**: You enter the filter order in a text box.
- **Minimum order**: The filter design method determines the minimum order filter.

Select the **Minimum order** radio button for this example.



Note that filter order specification options depend on the filter design method you choose. Some filter methods may not have both options available.

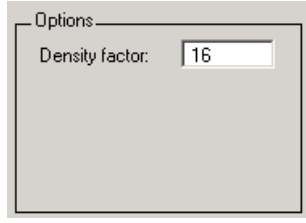
Options

The available options depend on the selected filter design method. Only the FIR Equiripple and FIR Window design methods have settable options. For FIR Equiripple, the option is a **Density Factor**. See `firpm` for more information. For FIR Window the options are **Scale Passband**, **Window** selection, and for the following windows, a settable parameter:

Window	Parameter
Chebyshev (`chebwin`)	Sidelobe attenuation
Gaussian (`gausswin`)	Alpha
Kaiser (`kaiser`)	Beta
Taylor (`taylorwin`)	Nbar and Sidelobe level
Tukey (`tukeywin`)	Alpha
User Defined	Function Name, Parameter

You can view the window in the Window Visualization Tool (`wvtool`) by clicking the **View** button.

For this example, set the **Density factor** to 16.



Bandpass Filter Frequency Specifications

For a bandpass filter, you can set

- Units of frequency:
  - Hz
  - kHz
  - MHz
  - Normalized (0 to 1)
- Sampling frequency
- Passband frequencies
- Stopband frequencies

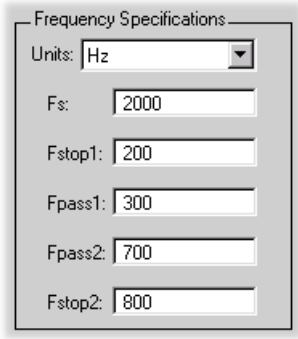
You specify the passband with two frequencies. The first frequency determines the lower edge of the passband, and the second frequency determines the upper edge of the passband.

Similarly, you specify the stopband with two frequencies. The first frequency determines the upper edge of the first stopband, and the second frequency determines the lower edge of the second stopband.

For this example:

- Keep the units in **Hz** (default).
- Set the sampling frequency (**Fs**) to 2000 Hz.

- Set the end of the first stopband (**Fstop1**) to 200 Hz.
- Set the beginning of the passband (**Fpass1**) to 300 Hz.
- Set the end of the passband (**Fpass2**) to 700 Hz.
- Set the beginning of the second stopband (**Fstop2**) to 800 Hz.



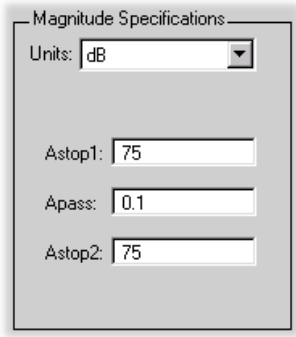
Bandpass Filter Magnitude Specifications

For a bandpass filter, you can specify the following magnitude response characteristics:

- Units for the magnitude response (dB or linear)
- Passband ripple
- Stopband attenuation

For this example:

- Keep **Units** in `dB` (default).
- Set the passband ripple (**Apass**) to `0.1` dB.
- Set the stopband attenuation for both stopbands (**Astop1**, **Astop2**) to 75 dB.


Magnitude Specifications
Units: dB
Astop1: 75
Apass: 0.1
Astop2: 75

Computing the Filter Coefficients

Now that you've specified the filter design, click the **Design Filter** button to compute the filter coefficients.

Notice that the Design Filter button is disabled once you've computed the coefficients for your filter design. This button is enabled again once you make any changes to the filter specifications.

Analyzing the Filter

In this section...

Displaying Filter Responses

You can view the following filter response characteristics in the display region or in a separate window (see "Displaying Filter Responses" on page 5-16):

- Magnitude response
- Phase response
- Magnitude and Phase responses
- Group delay response
- Phase delay response
- Impulse response
- Step response
- Pole-zero plot
- Zero-phase response — available from the *y*-axis context menu in a Magnitude or Magnitude and Phase response plot.

If you have Filter Design Toolbox product installed, two other analyses are available: magnitude response estimate and round-off noise power. These two analyses are the only ones that use filter internals.

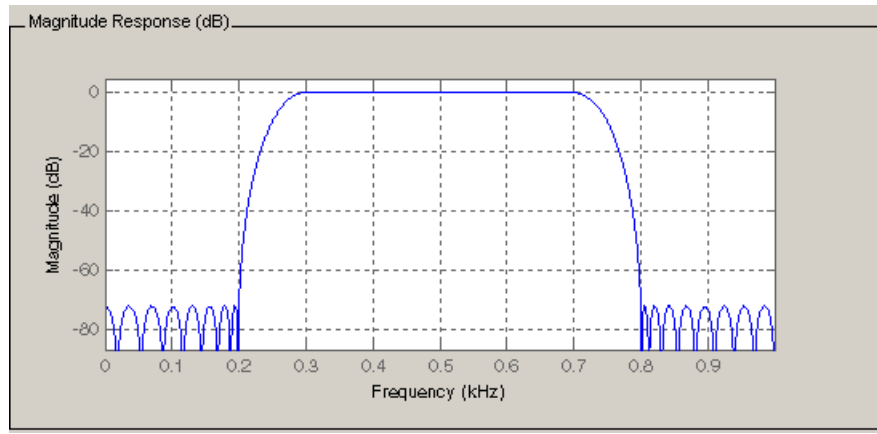
For descriptions of the above responses and their associated toolbar buttons and other FDATool toolbar buttons, see `fvtool`.

You can display two responses in the same plot by selecting **Analysis > Overlay Analysis** and selecting an available response. A second *y*-axis is added to the right side of the response plot. (Note that not all responses can be overlaid on each other.)

You can also display the filter coefficients and detailed filter information in this region.

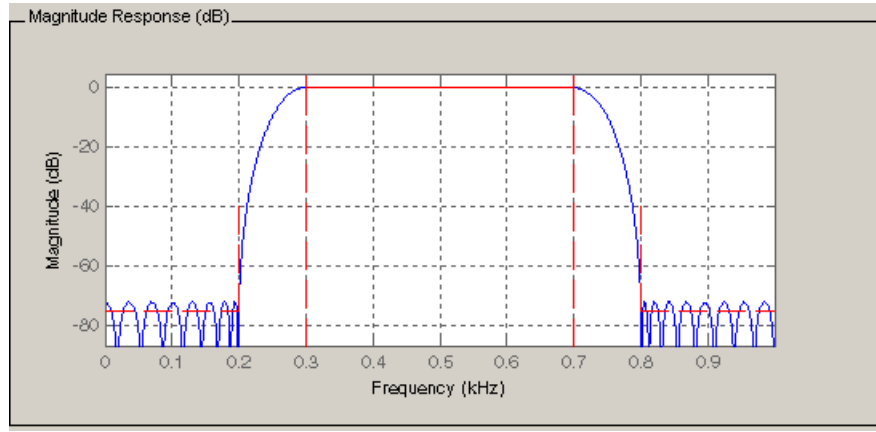
For all the analysis methods, except zero-phase response, you can access them from the **Analysis** menu, the Analysis Parameters dialog box from the context menu, or by using the toolbar buttons. For zero-phase, right-click the *y*-axis of the plot and select **Zero-phase** from the context menu.

For example, to look at the filter's magnitude response, select the **Magnitude Response** button on the toolbar.



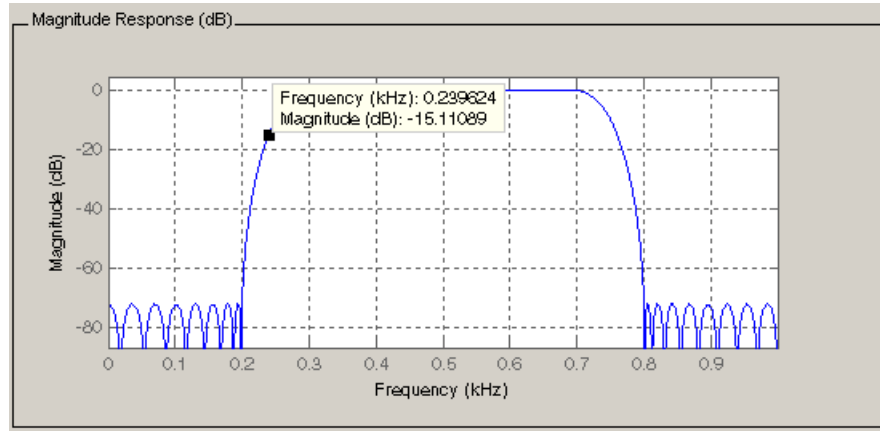
You can also overlay the filter specifications on the Magnitude plot by selecting **View > Specification Mask**.

Note You can use specification masks in FVTool only if FVTool was launched from FDATool.



Using Data Tips

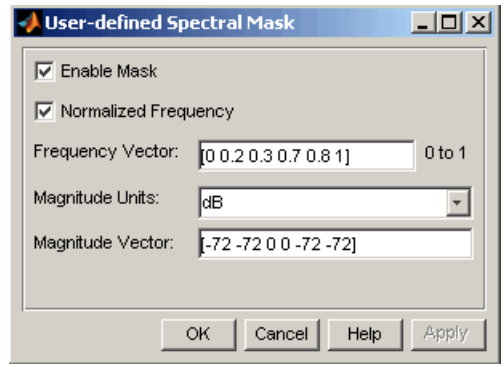
You can click the response to add plot data tips that display information about particular points on the response.



For information on using data tips, see "Data Cursor — Displaying Data Values Interactively" in the MATLAB documentation.

Drawing Spectral Masks

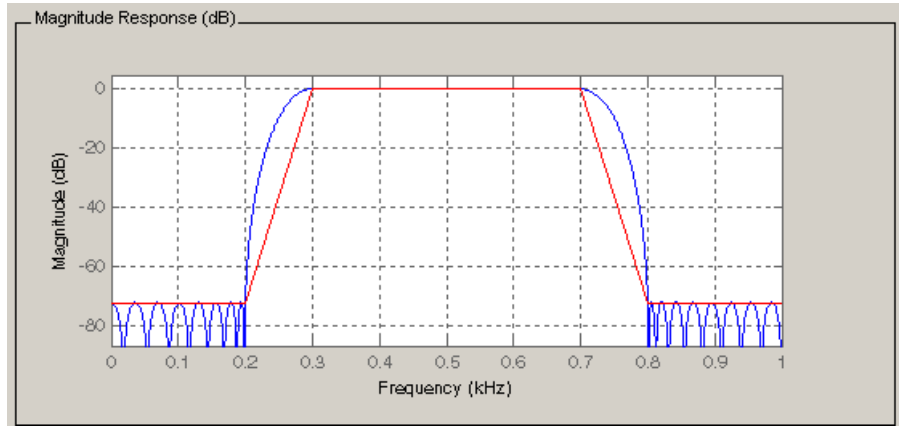
To add spectral masks or rejection area lines to your magnitude plot, click **View > User-defined Spectral Mask**.



The mask is defined by a frequency vector and a magnitude vector. These vectors must be the same length.

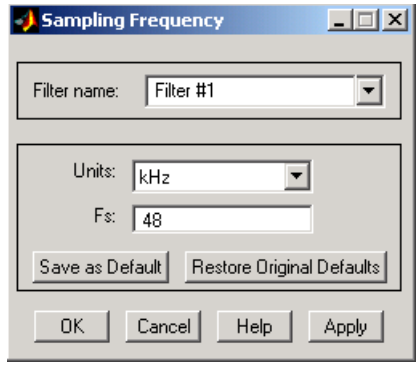
- `Enable Mask` — Select to turn on the mask display.
- `Normalized Frequency` — Select to normalize the frequency between 0 and 1 across the displayed frequency range.
- `Frequency Vector` — Enter a vector of *x*-axis frequency values.
- `Magnitude Units` — Select the desired magnitude units. These units should match the units used in the magnitude plot.
- `Magnitude Vector` — Enter a vector of *y*-axis magnitude values.

The magnitude response below shows a spectral mask.



Changing the Sampling Frequency

To change the sampling frequency of your filter, right-click any filter response plot and select **Sampling Frequency** from the context menu.



To change the filter name, type the new name in **Filter name**. (In `fvtool`, if you have multiple filters, select the desired filter and then enter the new name.)

To change the sampling frequency, select the desired unit from **Units** and enter the sampling frequency in **Fs**. (For each filter in `fvtool`, you can specify a different sampling frequency or you can apply the sampling frequency to all filters.)

To save the displayed parameters as the default values to use when FDATool or FVTool is opened, click **Save as Default**.

To restore the default values, click **Restore Original Defaults**.

Displaying the Response in FVTool

To display the filter response characteristics in a separate window, select **View > Filter Visualization Tool** (available if any analysis, except the filter specifications, is in the display region) or click the **Full View Analysis** button: 

This launches the Filter Visualization Tool (`fvtool`).

Note If Filter Specifications are shown in the display region, clicking the **Full View Analysis** toolbar button launches a MATLAB figure window instead of FVTool. The associated menu item is **Print to figure**, which is enabled only if the filter specifications are displayed.

You can use this tool to annotate your design, view other filter characteristics, and print your filter response. You can link FDATool and FVTool so that changes made in FDATool are immediately reflected in FVTool. See `fvtool` for more information.

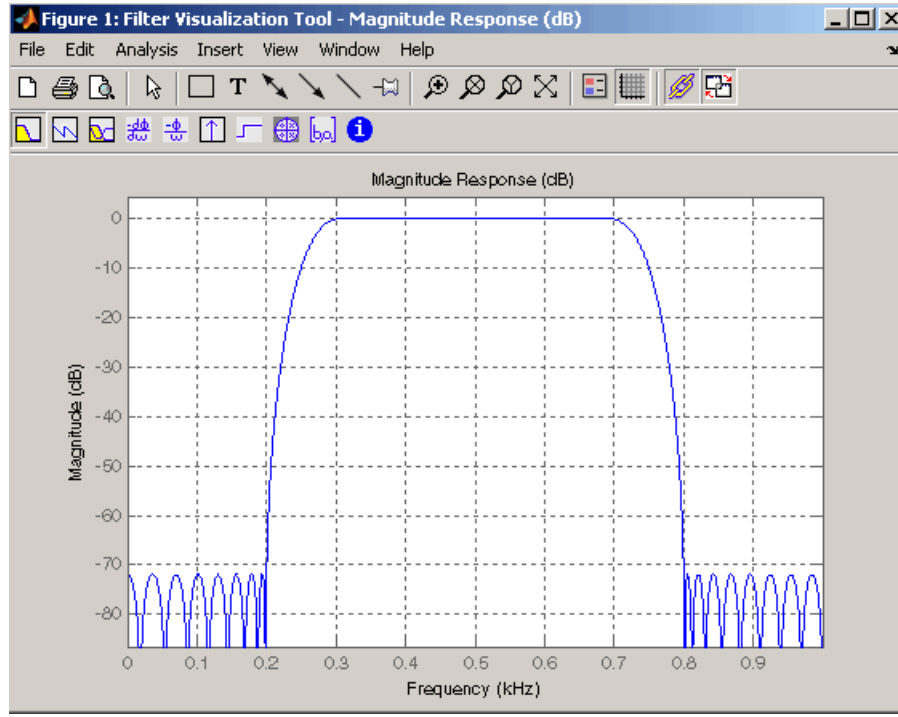


Figure 1: Filter Visualization Tool - Magnitude Response (dB)
File Edit Analysis Insert View Window Help
Magnitude Response (dB)
Magnitude (dB)
0
-10
-20
-30
-40
-50
-60
-70
-80
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
Frequency (kHz)

Editing the Filter Using the Pole/Zero Editor

In this section...

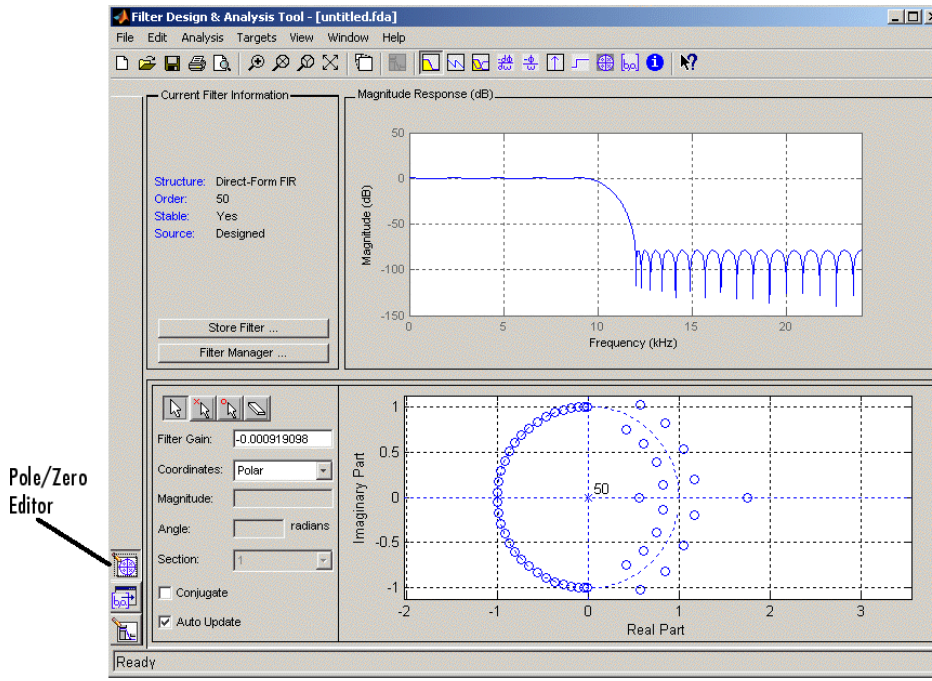
Displaying the Pole-Zero Plot

You can edit a designed or imported filter's coefficients by moving, deleting, or adding poles and/or zeros using the Pole/Zero Editor panel.

Note You cannot generate an M-file (**File > Generate M-file**) if your filter was designed or edited with the Pole/Zero Editor.

You cannot move quantized poles and zeros. You can only move the reference poles and zeros.

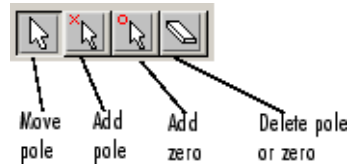
Click the **Pole/Zero Editor** button in the sidebar or select **Edit > Pole/Zero Editor** to display this panel.



Poles are shown using x symbols and zeros are shown using o symbols.

Changing the Pole-Zero Plot

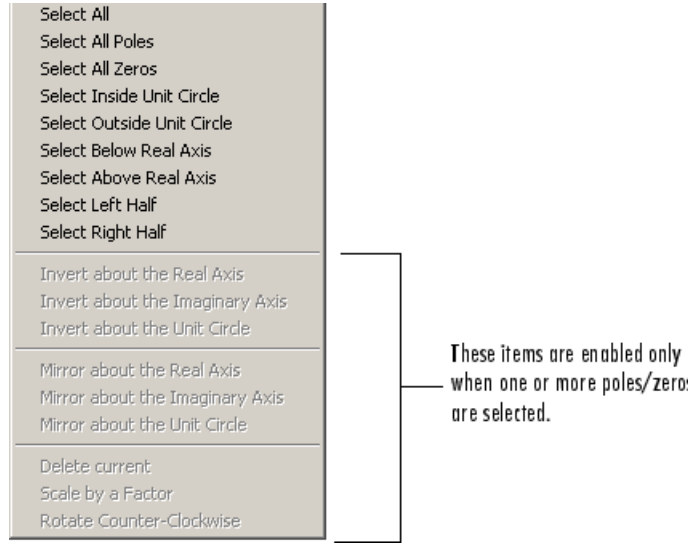
Plot mode buttons are located to the left of the pole/zero plot. Select one of the buttons to change the mode of the pole/zero plot. The Pole/Zero Editor has these buttons from left to right: move pole, add pole, add zero, and delete pole or zero.



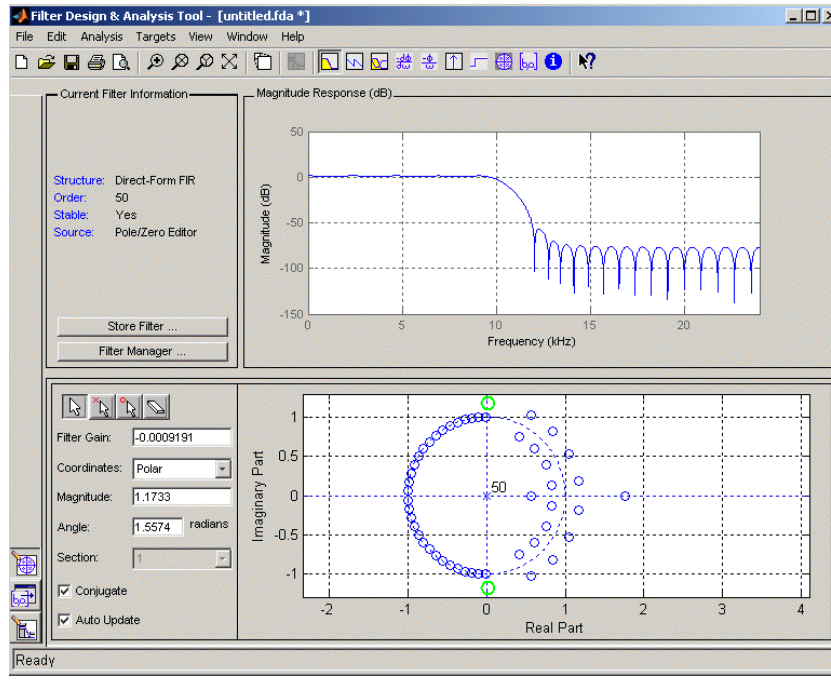
The following plot parameters and controls are located to the left of the pole/zero plot and below the plot mode buttons.

- **Filter gain** — factor to compensate for the filter's pole(s) and zero(s) gains
- **Coordinates** — units (`Polar` or `Rectangular`) of the selected pole or zero
- **Magnitude** — if polar coordinates is selected, magnitude of the selected pole or zero
- **Angle** — if polar coordinates is selected, angle of selected pole(s) or zero(s)
- **Real** — if rectangular coordinates is selected, real component of selected pole(s) or zero(s)
- **Imaginary** — if rectangular coordinates is selected, imaginary component of selected pole or zero
- **Section** — for multisection filters, number of the current section
- **Conjugate** — creates a corresponding conjugate pole or zero or automatically selects the conjugate pole or zero if it already exists.
- **Auto update** — immediately updates the displayed magnitude response when poles or zeros are added, moved, or deleted.

The **Edit > Pole/Zero Editor** has items for selecting multiple poles/zeros, for inverting and mirroring poles/zeros, and for deleting, scaling and rotating poles/zeros.



Moving one of the zeros on the vertical axis produces the following result:



- The selected zero pair is shown in green.
- When you select one of the zeros from a conjugate pair, the Conjugate check box and the conjugate are automatically selected.
- The Magnitude Response plot updates immediately because **Auto update** is active.

Converting the Filter Structure

In this section...

Converting to a New Structure

You can use **Edit > Convert Structure** to convert the current filter to a new structure. All filters can be converted to the following representations:

- Direct-form I
- Direct-form II
- Direct-form I transposed
- Direct-form II transposed
- Lattice ARMA

Note If you have Filter Design Toolbox product installed, you will see additional structures in the Convert structure dialog box.

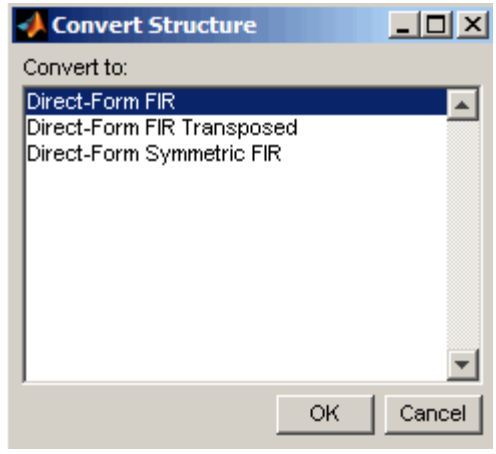
In addition, the following conversions are available for particular classes of filters:

- Minimum phase FIR filters can be converted to Lattice minimum phase
- Maximum phase FIR filters can be converted to Lattice maximum phase
- Allpass filters can be converted to Lattice allpass
- IIR filters can be converted to Lattice ARMA

Note Converting from one filter structure to another may produce a result with different characteristics than the original. This is due to the computer's finite-precision arithmetic and the variations in the conversion's roundoff computations.

For example:

- Select **Edit > Convert Structure** to open the Convert structure dialog box.
- Select `Direct-form I` in the list of filter structures.



Converting to Second-Order Sections

You can use **Edit > Convert to Second-Order Sections** to store the converted filter structure as a collection of second-order sections rather than as a monolithic higher-order structure.

Note The following options are also used for **Edit > Reorder and Scale Scale Second-Order Sections**, which you use to modify an SOS filter structure.

The following **Scale** options are available when converting a direct-form II structure only:

- `None` (default)
- `L-2` (L^2 norm)
- `L-infinity` (L^∞ norm)

The **Direction** (`Up` or `Down`) determines the ordering of the second-order sections. The optimal ordering changes depending on the **Scale** option selected.

For example:

- Select **Edit > Convert to Second-Order Sections** to open the Convert to SOS dialog box.
- Select `L-infinity` from the **Scale** menu for L^∞ norm scaling.
- Leave `Up` as the **Direction** option.

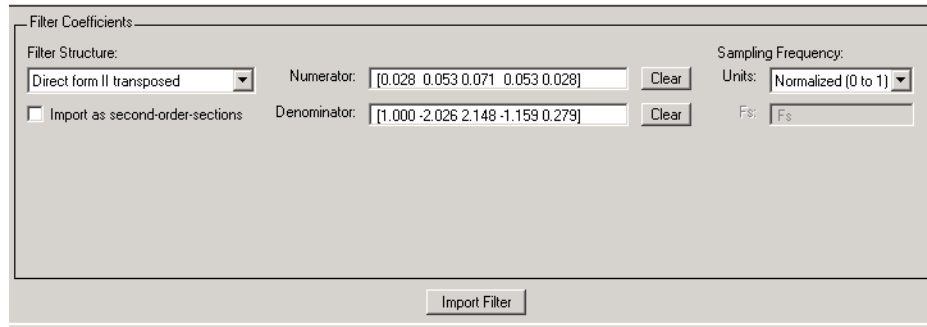
Note To convert from second-order sections back to a single section, use **Edit > Convert to Single Section**.

Importing a Filter Design

In this section...

Import Filter Panel

The Import Filter panel allows you to import a filter. You can access this region by clicking the **Import Filter** button in the sidebar.



The imported filter can be in any of the representations listed in the **Filter Structure** pull-down menu and described in "Filter Structures" on page 5-32. You can import a filter as second-order sections by selecting the check box.

Specify the filter coefficients in **Numerator** and **Denominator**, either by entering them explicitly or by referring to variables in the MATLAB workspace.

Select the frequency units from the following options in the **Units** menu, and for any frequency unit other than Normalized, specify the value or MATLAB workspace variable of the sampling frequency in the **Fs** field.

To import the filter, click the **Import Filter** button. The display region is automatically updated when the new filter has been imported.

You can edit the imported filter using the Pole/Zero Editor panel (see “Editing the Filter Using the Pole/Zero Editor” on page 5-23).

Filter Structures

The available filter structures are:

- "Direct-form" on page 5-32 , which includes direct-form I, direct-form II, direct-form I transposed, direct-form II transposed, and direct-form FIR
- "Lattice" on page 5-33, which includes lattice allpass, lattice MA min phase, lattice MA max phase, and lattice ARMA
- "Discrete-time Filter (dfilt object)" on page 5-33(`dfilt` object)

The structure that you choose determines the type of coefficients that you need to specify in the text fields to the right.

Direct-form

For direct-form I, direct-form II, direct-form I transposed, and direct-form II transposed, specify the filter by its transfer function representation

$$H(z) = \frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \cdots + b(m+1)z^{-m}}{a(1) + a(2)z^{-1} + \cdots + a(n+1)z^{-n}}$$

- The **Numerator** field specifies a variable name or value for the numerator coefficient vector `b`, which contains `m+1` coefficients in descending powers of *z*.
- The **Denominator** field specifies a variable name or value for the denominator coefficient vector `a`, which contains `n+1` coefficients in descending powers of *z*. For FIR filters, the **Denominator** is `1`.

Filters in transfer function form can be produced by all of the Signal Processing Toolbox filter design functions (such as `fir1`, `fir2`, `firpm`, `butter`, `yulewalk`). See "Transfer Function" on page 1-22 for more information.

Importing as second-order sections. For all direct-form structures, except direct-form FIR, you an import the filter in its second-order section representation:

$$H(z) = g\prod_{k=1}^{L} H_k(z) = g\prod_{k=1}^{L} \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2}}{1 + a_{1k}z^{-1} + a_{2k}z^{-2}}$$

The **Gain** field specifies a variable name or a value for the gain *g*, and the **SOS Matrix** field specifies a variable name or a value for the *L*-by-6 SOS matrix

$$SOS = \begin{bmatrix} b_{01} & b_{11} & b_{21} & 1 & a_{11} & a_{21} \\ b_{02} & b_{12} & b_{22} & 1 & a_{12} & a_{22} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{0L} & b_{1L} & b_{2L} & 1 & a_{1L} & a_{2L} \end{bmatrix}$$

whose rows contain the numerator and denominator coefficients b_{ik} and a_{ik} of the second-order sections of $H(z)$.

Filters in second-order section form can be produced by functions such as `tf2sos`, `zp2sos`, `ss2sos`, and `sosfilt`. See "Second-Order Sections (SOS)" on page 1-27for more information.

Lattice

For lattice allpass, lattice minimum and maximum phase, and lattice ARMA filters, specify the filter by its lattice representation:

- For lattice allpass, the **Lattice coeff** field specifies the lattice (reflection) coefficients, `k(1)` to `k(N)`, where `N` is the filter order.
- For lattice MA (minimum or maximum phase), the **Lattice coeff** field specifies the lattice (reflection) coefficients, `k(1)` to `k(N)`, where `N` is the filter order.
- For lattice ARMA, the **Lattice coeff** field specifies the lattice (reflection) coefficients, `k(1)` to `k(N)`, and the **Ladder coeff** field specifies the ladder coefficients, `v(1)` to `v(N+1)`, where `N` is the filter order.

Filters in lattice form can be produced by `tf2latc`. See "Lattice Structure" on page 1-27for more information.

Discrete-time Filter (dfilt object)

For Discrete-time filter, specify the name of the `dfilt` object. See `dfilt` for more information.

Multirate Filter (mfilt object)

For Multirate filter, specify the name of the `mfilt` object. See `mfilt` in the Filter Design Toolbox product for more information.

Exporting a Filter Design

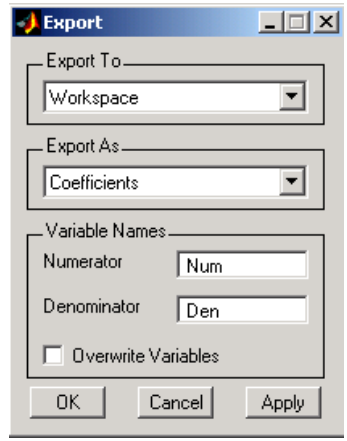
In this section...

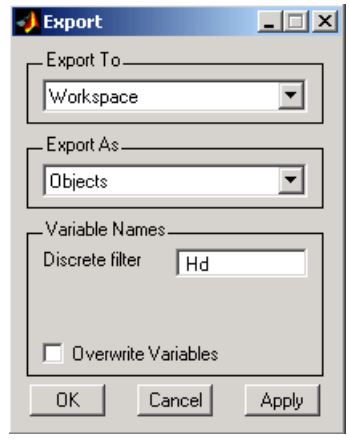
Exporting Coefficients or Objects to the Workspace

You can save the filter either as filter coefficients variables or as a `dfilt` or `mfilt` filter object variable. (Note that you must have Filter Design Toolbox product installed to save as an `mfilt`.) To save the filter to the MATLAB workspace:

1. Select **File > Export**. The Export dialog box appears.
2. Select `Workspace` from the **Export To** menu.
3. Select `Coefficients` from the **Export As** menu to save the filter coefficients or select `Objects` to save the filter in a filter object.
4. For coefficients, assign variable names using the **Numerator** (for FIR filters) or **Numerator** and **Denominator** (for IIR filters), or **SOS Matrix** and **Scale Values** (for IIR filters in second-order section form) text boxes in the Variable Names region.

   For objects, assign the variable name in the **Discrete Filter (**or **Quantized Filter)** text box. If you have variables with the same names in your workspace and you want to overwrite them, select the **Overwrite Variables** check box.
5. Click the **Export** button.





Exporting Coefficients to an ASCII File

To save filter coefficients to a text file,

1. Select **File > Export**. The Export dialog box appears.
2. Select `Coefficients File (ASCII)` from the **Export To** menu.
3. Click the **Export** button. The Export Filter Coefficients to .FCF File dialog box appears.
4. Choose or enter a filename and click the **Save** button.

The coefficients are saved in the text file that you specified, and the MATLAB Editor opens to display the file. The text file also contains comments with the MATLAB version number, the Signal Processing Toolbox version number, and filter information.

Exporting Coefficients or Objects to a MAT-File

To save filter coefficients or a filter object as variables in a MAT-file:

1. Select **File > Export**. The Export dialog box appears.
2. Select `MAT-file` from the **Export To** menu.
3. Select `Coefficients` from the **Export As** menu to save the filter coefficients or select `Objects` to save the filter in a filter object.
4. For coefficients, assign variable names using the **Numerator** (for FIR filters) or **Numerator** and **Denominator** (for IIR filters), or **SOS Matrix** and **Scale Values** (for IIR filters in second-order section form) text boxes in the Variable Names region.

   For objects, assign the variable name in the **Discrete Filter (**or **Quantized Filter)** text box. If you have variables with the same names in your workspace and you want to overwrite them, select the **Overwrite Variables** check box.
5. Click the **Export** button. The Export to a MAT-File dialog box appears.
6. Choose or enter a filename and click the **Save** button.

See also “Saving and Opening Filter Design Sessions” on page 5-47.

Exporting to SPTool

You may want to use your designed filter in SPTool to do signal processing and analysis.

Note The magnitude response you see in SPTool will differ from the one in FDATool because the sampling frequency is preset at `Fs = 2` when a filter is exported from FDATool to SPTool.

1. Select **File > Export**. The Export dialog box appears.
2. Select `SPTool` from the **Export To** menu.
3. Assign the variable name in the **Discrete Filter (**or **Quantized Filter)** text box. If you have variables with the same names in your workspace and you want to overwrite them, select the **Overwrite Variables** check box.
4. Click the **Export** button.

   SPTool opens and the current FDATool filter appears in the Filter area list as the specified variable name followed by `(Imported)`.

   Note If you are using the Filter Design Toolbox product and export a quantized filter, only the values of its quantized coefficients are exported. The reference coefficients are not exported. SPTool does not restrict the coefficient values, so if you edit them in SPTool by moving poles or zeros, the filter will no longer be in quantized form.

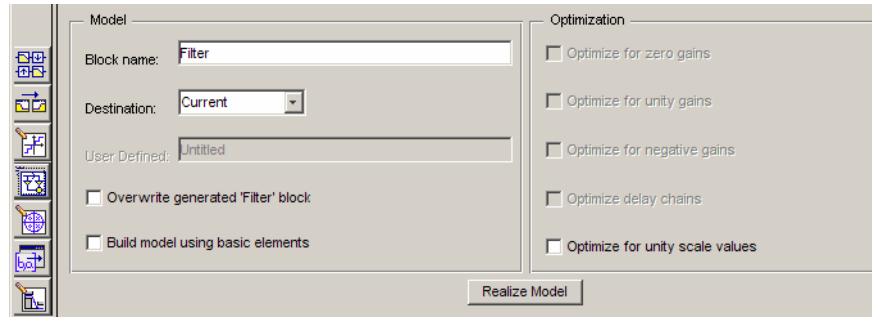
Exporting to a Simulink Model

If you have the Simulink product installed, you can export a Simulink block of your filter design and insert it into a new or existing Simulink model.

You can export a filter designed using any filter design method available in FDATool.

Note If you have Filter Design Toolbox product installed, you can export a CIC filter to a Simulink model, if you also have this toolbox and blockset installed: Fixed-Point Toolbox™ and Signal Processing Blockset.

1 After designing your filter, click the **Realize Model** sidebar button or select **File > Export to Simulink Model**. The Realize Model panel is displayed.



2 Specify the name to use for your block in **Block name**.

3 Select the **Destination** — `Current` to insert the block into the current (most recently selected) Simulink model or `New` to open a new model.

4 If you want to overwrite a block previously created from this panel, check **Overwrite generated 'Filter' block**.

Note If you have Signal Processing Blockset product installed, a **Build model using basic elements** check box is included. If you deselect it, a Digital Filter block is created instead of a subsystem block, which uses separate subelements. See the Filter Realization Wizard and Choosing Between Filter Design Blocks in the Signal Processing Blockset documentation for information.

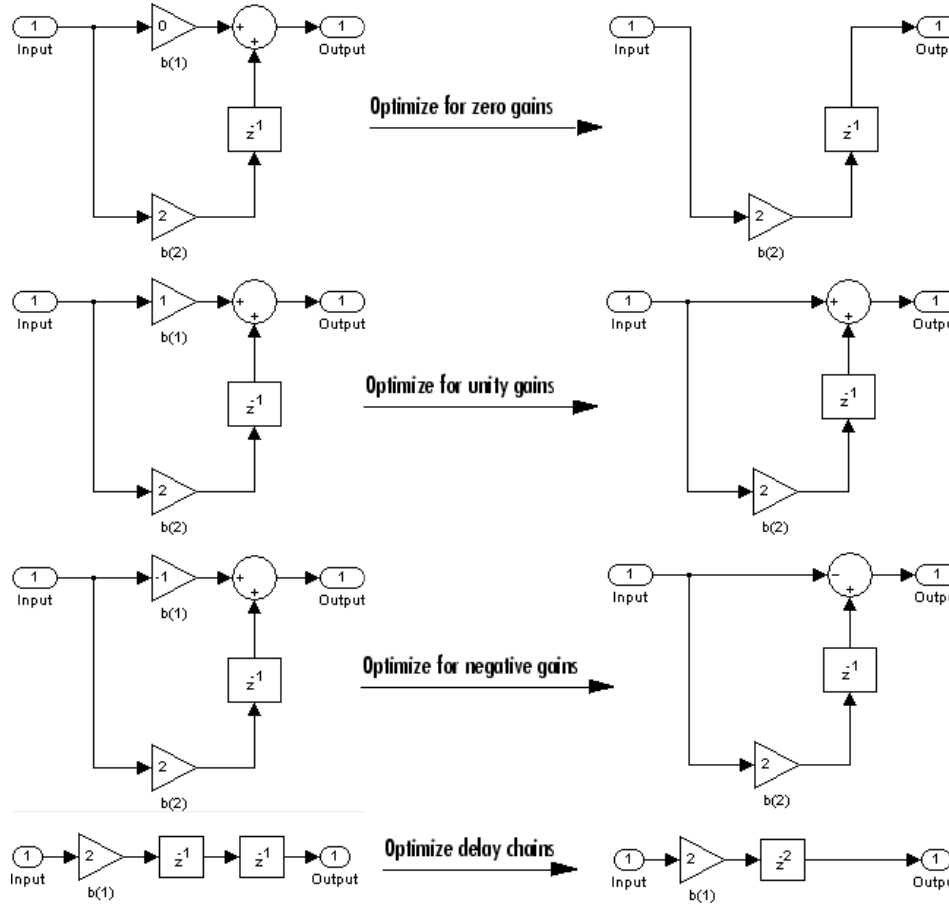
---

5 If you select **Build model using basic elements**, you can select the desired optimization(s) for your block:

- `Optimize for zero gains` — Removes zero-valued gain paths from the filter structure.
- `Optimize for unity gains` — Substitutes a wire (short circuit) for gains equal to 1 in the filter structure.

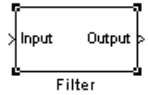
- `Optimize for negative gains` — Substitutes a wire (short circuit) for gains equal to -1 and changes corresponding additions to subtractions in the filter structure.
- `Optimize delay chains` — Substitutes delay chains composed of `n` unit delays with a single delay of `n`.
- `Optimize for unity scale values` — Removes multiplications for scale values equal to 1 from the filter sturcture.

The following illustration shows the effects of some of the optimizations:



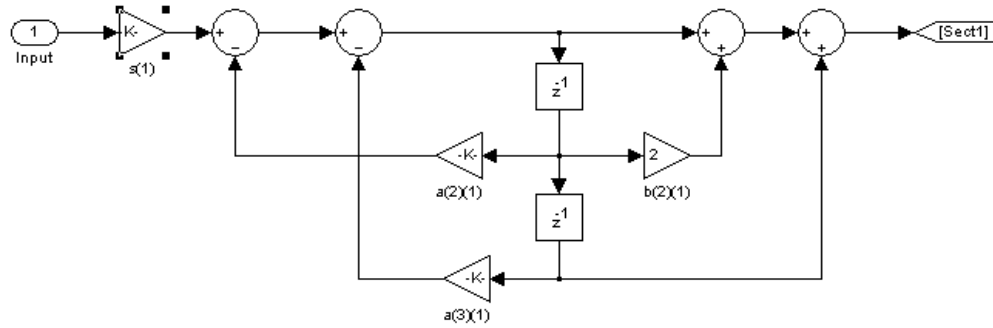
Optimization Effects

6 Click the **Realize Model** button to create the filter block. The filter is implemented as a subsystem block using Sum, Gain, and Integer Delay blocks.



Note Note that if your filter is implemented using basic elements (Sum, Gain, and Delay blocks), inputs to the filter must be sample based.

If you double-click the Simulink Filter block, the filter structure is displayed. The following figure shows the first section of the default four-section, direct form II filter.



Other Ways to Export a Filter

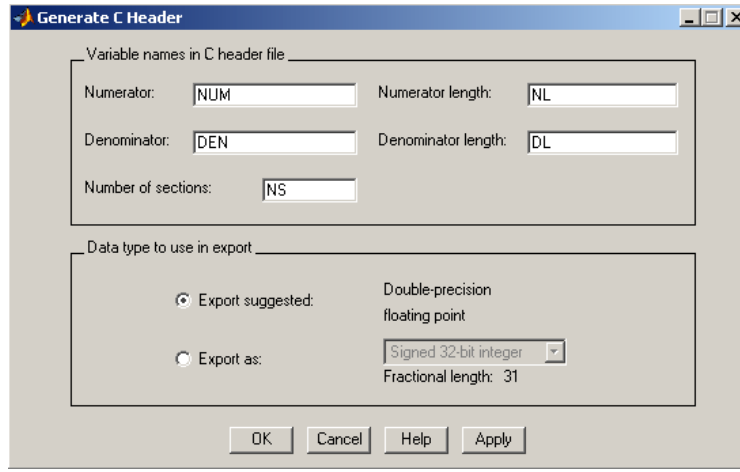
You can also send your filter to a C header file or generate an M-file. The M-file contains code that replicates the filter you designed. See the following sections:

- "Generating a C Header File" on page 5-42
- "Generating an M-File" on page 5-44

Generating a C Header File

You may want to include filter information in an external C program. To create a C header file with variables that contain filter parameter data, follow this procedure:

1 Select **Targets > Generate C Header**. The Generate C Header dialog box appears.



2 Enter the variable names to be used in the C header file. The particular filter structure determines the variables that are created in the file

Filter Structure	Variable Parameter
Direct-form I Direct-form II Direct-form I transposed Direct-form II transposed	Numerator, Numerator length*, Denominator, Denominator length*, and Number of sections (inactive if filter has only one section)
Lattice ARMA	Lattice coeffs, Lattice coeffs length*, Ladder coeffs, Ladder coeffs length*, Number of sections (inactive if filter has only one section)

Filter Structure	Variable Parameter
Lattice MA	Lattice coeffs, Lattice coeffs length*, and Number of sections (inactive if filter has only one section)
Direct-form FIR Direct-form FIR transposed	Numerator, Numerator length*, and Number of sections (inactive if filter has only one section)

***length** variables contain the total number of coefficients of that type.

Note Variable names cannot be C language reserved words, such as “for.”

3 Select **Export Suggested** to use the suggested data type or select **Export As** and select the desired data type from the pull-down.

Note If you do not have Filter Design Toolbox product installed, selecting any data type other than double-precision floating point results in a filter that does not exactly match the one you designed in the FDATool. This is due to rounding and truncating differences.

4 Click **OK** to save the file and close the dialog box or click **Apply** to save the file, but leave the dialog box open for additional C header file definitions.

Generating an M-File

You can generate an M-file that contains all the code used to create the filter you designed in FDATool. Select **File > Generate M-file** and specify the filename in the Generate M-file dialog box.

Note You cannot generate an M-file (**File > Generate M-file**) if your filter was designed or edited with the Pole/Zero Editor.

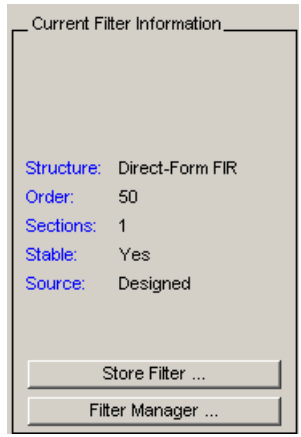
The following is a sample generated M-file of the default FDATool filter.

```
function
Hd = untitled
%UNTITLED Returns a discrete-time filter object
% % M-file generated by MATLAB(R) 6.5 and Signal Processing
% Toolbox 6.0.
% % Generated on: 24-Oct-2002 09:46:59
% % Remez FIR Lowpass filter designed using the firpm function.
% All frequency values are in Hz. Fs = 48000;
% Sampling Frequency  Fpass = 9600;
% Passband Frequency Fstop = 12000;
% Stopband Frequency Dpass = 0.057501127785;
% Passband Ripple Dstop = 0.0001;
% Stopband Attenuation dens  = 16;
% Density Factor

% Calculate the order from the parameters using firpmord.
[N, Fo, Ao, W] = firpmord([Fpass, Fstop]/(Fs/2), [1 0], ... [Dpass, Dstop]);

% Calculate the coefficients using the firpm function.
b  = firpm(N, Fo, Ao, W, {dens});
Hd = dfilt.dffir(b);
% [EOF]
```

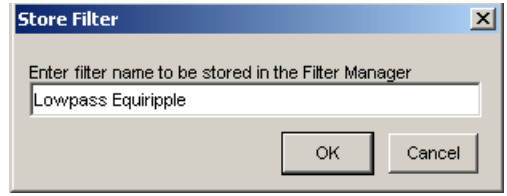
Managing Filters in the Current Session

You can store filters designed in the current FDATool session for cascading together, exporting to FVTool or for recalling later in the same or future FDATool sessions.

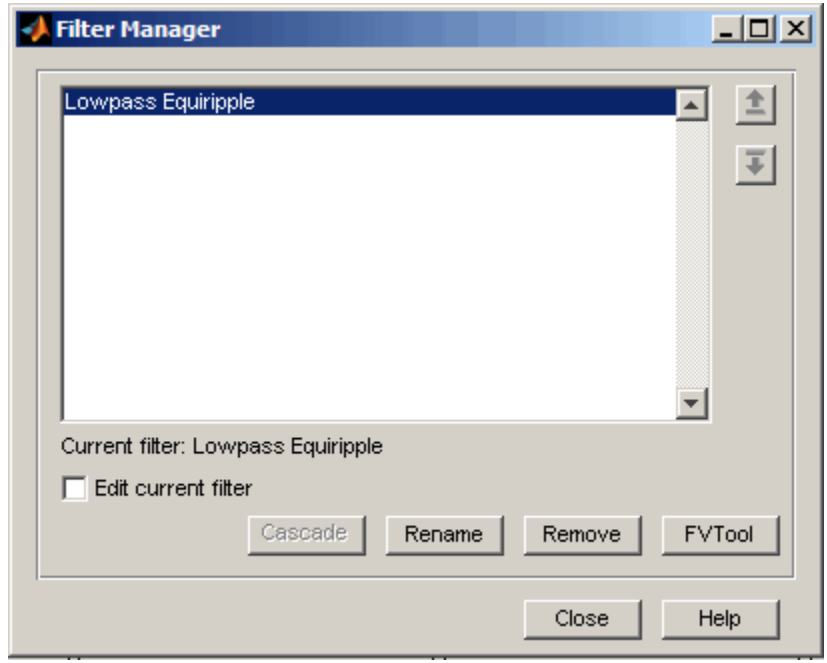
You store and access saved filters with the **Store filter** and **Filter Manager** buttons, respectively, in the Current Filter Information pane.



Store Filter — Displays the Store Filter dialog box in which you specify the filter name to use when storing the filter in the Filter Manager. The default name is the type of the filter.



Filter Manager — Opens the Filter Manager.



The current filter is listed below the listbox. To change the current filter, highlight the desired filter. If you select **Edit current filter**, FDATool displays the currently selected filter specifications. If you make any changes to the specifications, the stored filter is updated immediately.

To cascade two or more filters, highlight the desired filters and press **Cascade**. A new cascaded filter is added to the Filter Manager.

To change the name of a stored filter, press **Rename**. The Rename filter dialog box is displayed.

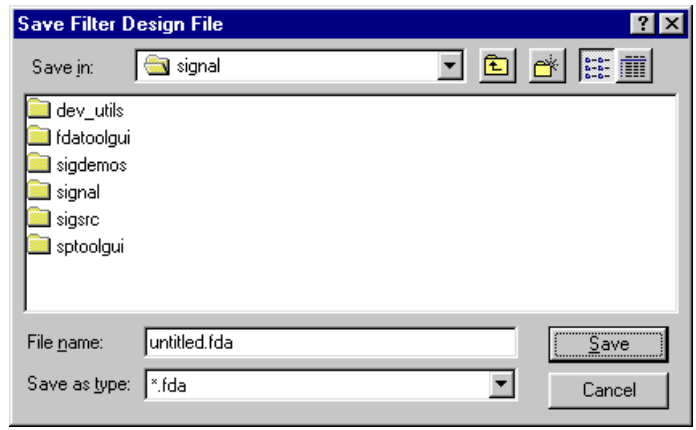
To remove a stored filter from the Filter Manager, press **Delete**.

To export one or more filters to FVTool, highlight the filter(s) and press **FVTool**.

Saving and Opening Filter Design Sessions

You can save your filter design session as a MAT-file and return to the same session another time.

Select the **Save session** button  to save your session as a MAT-file. The first time you save a session, a Save Filter Design File browser opens, prompting you for a session name.



For example, save this design session as `TestFilter.fda` in your current working directory by typing `TestFilter` in the **File name** field.

The `.fda` extension is added automatically to all filter design sessions you save.

Note You can also use the **File > Save session** and **File > Save session as** to save a session.

You can load existing sessions into the Filter Design and Analysis Tool by selecting the **Open session** button,  or **File > Open session** . A Load Filter Design File browser opens that allows you to select from your previously saved filter design sessions.

6

Statistical Signal Processing

The following chapter discusses statistical signal processing tools and applications, including correlations, covariance, and spectral estimation.

Correlation and Covariance

In this section...

Background Information

The true cross-correlation sequence is a statistical quantity defined as

$$R_{xy}(m) = E\{x_{n+m}y^*_n\} = E\{x_n y^*_{n-m}\}$$

where x_n and y_n are stationary random processes, $-\infty < n < \infty$, and $E\{\}$ is the expected value operator. The covariance sequence is the mean-removed cross-correlation sequence

$$C_{xy}(m) = E\{(x_{n+m} - \mu_x)(y_n - \mu_y)^*\}$$

or, in terms of the cross-correlation,

$$C_{xy}(m) = R_{xy}(m) - \mu_x\mu^*_y$$

In practice, you must estimate these sequences, because it is possible to access only a finite segment of the infinite-length random process. A common estimate based on N samples of x_n and y_n is the deterministic cross-correlation sequence (also called the time-ambiguity function)

$$\hat{R}_{xy}(m) = \begin{cases} \sum_{n=0}^{N-m-1} x_{n+m}y^*_n & m \geq 0 \\ \hat{R}^*_{yx}(-m) & m < 0 \end{cases}$$

where we assume for this discussion that x_n and y_n are indexed from 0 to N-1, and $\hat{R}_{xy}(m)$ from -(N-1) to N-1.

Using xcorr and xcov Functions

The functions `xcorr` and `xcov` estimate the cross-correlation and cross-covariance sequences of random processes. They also handle autocorrelation and autocovariance as special cases. The `xcorr` function evaluates the sum shown above with an efficient FFT-based algorithm, given inputs x_n and y_n stored in length N vectors `x` and `y`. Its operation is equivalent to convolution with one of the two subsequences reversed in time.

For example:

```
x = [1 1 1 1 1]';
y = x;
xyc = xcorr(x,y)
xyc =
    1.0000
    2.0000
    3.0000
    4.0000
    5.0000
    4.0000
    3.0000
    2.0000
    1.0000
```

Notice that the resulting sequence length is one less than twice the length of the input sequence. Thus, the Nth element is the correlation at lag 0. Also notice the triangular pulse of the output that results when convolving two square pulses.

The `xcov` function estimates autocovariance and cross-covariance sequences. This function has the same options and evaluates the same sum as `xcorr`, but first removes the means of `x` and `y`.

Bias and Normalization

An estimate of a quantity is *biased* if its expected value is not equal to the quantity it estimates. The expected value of the output of `xcorr` is

$$E\{\hat{R}_{xy}(m)\} = \sum_{n=0}^{N-|m|-1} E\{x_{n+m}y^*_n\} = (N-|m|)R_{xy}(m)$$

`xcorr` provides the unbiased estimate, dividing by $N-|m|$, when you specify an `'unbiased'` flag after the input sequences.

```
xcorr(x,y,'unbiased')
```

Although this estimate is unbiased, the end points (near -(N-1) and N-1) suffer from large variance because `xcorr` computes them using only a few data points. A possible trade-off is to simply divide by N using the `'biased'` flag:

```
xcorr(x,y,'biased')
```

With this scheme, only the sample of the correlation at zero lag (the Nth output element) is unbiased. This estimate is often more desirable than the unbiased one because it avoids random large variations at the end points of the correlation sequence.

`xcorr` provides one other normalization scheme. The syntax

```
xcorr(x,y,'coeff')
```

divides the output by `norm(x)*norm(y)` so that, for autocorrelations, the sample at zero lag is 1.

Multiple Channels

For a multichannel signal, `xcorr` and `xcov` estimate the autocorrelation and cross-correlation and covariance sequences for all of the channels at once. If `S` is an M-by-N signal matrix representing N channels in its columns, `xcorr(S)` returns a (2M-1)-by-N^2 matrix with the autocorrelations and cross-correlations of the channels of `S` in its N^2 columns. If `S` is a three-channel signal

```
S = [s1 s2 s3]
```

then the result of `xcorr(S)` is organized as

```
R = [Rs1s1 Rs1s2 Rs1s3 Rs2s1 Rs2s2 Rs2s3 Rs3s1 Rs3s2 Rs3s3]
```

Two related functions, `cov` and `corrcoef`, are available in the standard MATLAB environment. They estimate covariance and normalized covariance respectively between the different channels at lag 0 and arrange them in a square matrix.

Spectral Analysis

In this section...

Background Information

The goal of *spectral estimation* is to describe the distribution (over frequency) of the power contained in a signal, based on a finite set of data. Estimation of power spectra is useful in a variety of applications, including the detection of signals buried in wide-band noise.

The *power spectral density* (PSD) of a stationary random process x_n is mathematically related to the correlation sequence by the discrete-time Fourier transform. In terms of normalized frequency, this is given by

$$P_{xx}(\omega) = \frac{1}{2\pi} \sum_{m=-\infty}^{\infty} R_{xx}(m) e^{-j\omega m}$$

This can be written as a function of physical frequency f (e.g., in hertz) by using the relation $\omega = 2\pi f/f_s$, where f_s is the sampling frequency.

$$P_{xx}(f) = \frac{1}{f_s} \sum_{m=-\infty}^{\infty} R_{xx}(m) e^{-2\pi j f m / f_s}$$

The correlation sequence can be derived from the PSD by use of the inverse discrete-time Fourier transform:

$$R_{xx}(m) = \int_{-\pi}^{\pi} (P_{xx}(\omega)e^{jwm})dw = \int_{-f_s/2}^{f_s/2} (P_{xx}(f)e^{2\pi jfm/f_s})df$$

The average power of the sequence x_n over the entire Nyquist interval is represented by

$$R_{xx}(0) = \int_{-\pi}^{\pi} P_{xx}(w)dw = \int_{-f_s/2}^{f_s/2} P_{xx}(f)df$$

The average power of a signal over a particular frequency band $[\omega_1, \omega_2]$, $0 \le \omega_1 < \omega_2 \le \pi$, can be found by integrating the PSD over that band:

$$\overline{P}_{[\omega_1, \omega_2]} = \int_{\omega_1}^{\omega_2} P_{xx}(\omega)d\omega + \int_{-\omega_2}^{-\omega_1} P_{xx}(\omega)d\omega$$

You can see from the above expression that $P_{xx}(\omega)$ represents the power content of a signal in an *infinitesimal* frequency band, which is why it is called the power spectral *density*.

The units of the PSD are power (e.g., watts) per unit of frequency. In the case of $P_{xx}(\omega)$, this is watts/radian/sample or simply watts/radian. In the case of $P_{xx}(f)$, the units are watts/hertz. Integration of the PSD with respect to frequency yields units of watts, as expected for the average power $\overline{P}_{[\omega_1, \omega_2]}$.

For real signals, the PSD is symmetric about DC, and thus $P_{xx}(\omega)$ for $0 \le \omega < \pi$ is sufficient to completely characterize the PSD. However, to obtain the average power over the entire Nyquist interval, it is necessary to introduce the concept of the *one-sided* PSD.

The one-sided PSD is given by

$$P_{onesided}(\omega) = \begin{cases} 0, & -\pi \le \omega < 0 \\ 2P_{xx}(\omega), & 0 \le \omega < \pi \end{cases}$$

The average power of a signal over the frequency band $[\omega_1,\omega_2]$, $0 \le \omega_1 < \omega_2 \le \pi$, can be computed using the one-sided PSD as

$$\bar{P}_{[\omega_1,\omega_2]} = \int_{\omega_1}^{\omega_2} P_{onesided}(\omega)d\omega$$

Spectral Estimation Method

The various methods of spectrum estimation available in the toolbox are categorized as follows:

- Nonparametric methods
- Parametric methods
- Subspace methods

Nonparametric methods are those in which the PSD is estimated directly from the signal itself. The simplest such method is the *periodogram*. An improved version of the periodogram is *Welch's method* [8]. A more modern nonparametric technique is the *multitaper method* (*MTM*).

Parametric methods are those in which the PSD is estimated from a signal that is assumed to be output of a linear system driven by white noise. Examples are the *Yule-Walker autoregressive* (*AR*) *method* and the *Burg method*. These methods estimate the PSD by first estimating the parameters (coefficients) of the linear system that hypothetically generates the signal. They tend to produce better results than classical nonparametric methods when the data length of the available signal is relatively short.

Subspace methods, also known as *high-resolution methods* or *super-resolution methods*, generate frequency component estimates for a signal based on an eigenanalysis or eigendecomposition of the correlation matrix. Examples are the *multiple signal classification* (*MUSIC*) *method* or the *eigenvector* (*EV*) *method*. These methods are best suited for line spectra — that is, spectra of sinusoidal signals — and are effective in the detection of sinusoids buried in noise, especially when the signal to noise ratios are low.

All three categories of methods are listed in the table below with the corresponding toolbox function and spectrum object names. More information

about each function is on the corresponding function reference page. See "Parametric Modeling" on page 7-15 for details about `lpc` and other parametric estimation functions.

Spectral Estimation Methods/Functions

Method	Description	Functions
Periodogram	Power spectral density estimate	`spectrum.periodogram`, `periodogram`
Welch	Averaged periodograms of overlapped, windowed signal sections	`spectrum.welch`, `pwelch`, `cpsd`, `tfestimate`, `mscohere`
Multitaper	Spectral estimate from combination of multiple orthogonal windows (or "tapers")	`spectrum.mtm`, `pmtm`
Yule-Walker AR	Autoregressive (AR) spectral estimate of a time-series from its estimated autocorrelation function	`spectrum.yulear`, `pyulear`
Burg	Autoregressive (AR) spectral estimation of a time-series by minimization of linear prediction errors	`spectrum.burg`, `pburg`
Covariance	Autoregressive (AR) spectral estimation of a time-series by minimization of the forward prediction errors	`spectrum.cov`, `pcov`

Spectral Estimation Methods/Functions (Continued)

Method	Description	Functions
Modified Covariance	Autoregressive (AR) spectral estimation of a time-series by minimization of the forward and backward prediction errors	`spectrum.mcov`, `pmcov`
MUSIC	Multiple signal classification	`spectrum.music`, `pmusic`
Eigenvector	Pseudospectrum estimate	`spectrum.eigenvector`, `peig`

Nonparametric Methods

The following sections discuss the periodogram, modified periodogram, Welch, and multitaper methods of nonparametric estimation, along with the related CPSD function, transfer function estimate, and coherence function.

Periodogram

In general terms, one way of estimating the PSD of a process is to simply find the discrete-time Fourier transform of the samples of the process (usually done on a grid with an FFT) and take the magnitude squared of the result. This estimate is called the *periodogram*.

The periodogram estimate of the PSD of a length-L signal $x_L[n]$ is

$$\hat{P}_{xx}(f) = \frac{\left|X_L(f)\right|^2}{f_s L}$$

where

$$X_L(f) = \sum_{n=0}^{L-1} x_L[n] e^{-2\pi j f n / f_s}$$

The actual computation of $X_L(f)$ can be performed only at a finite number of frequency points, N, and usually employs the FFT. In practice, most implementations of the periodogram method compute the N-point PSD estimate

$$\hat{P}_{xx}[f_k] = \frac{\left|X_L[f_k]\right|^2}{f_s L}, \qquad f_k = \frac{kf_s}{N}, \qquad k = 0, 1, \ldots, N-1$$

where

$$X_L[f_k] = \sum_{n=0}^{N-1} x_L[n] e^{-2\pi jkn/N}$$

It is wise to choose $N > L$ so that N is the next power of two larger than L. To evaluate $X_L[f_k]$, we simply pad $x_L[n]$ with zeros to length N. If $L > N$, we must wrap $x_L[n]$ modulo-N prior to computing $X_L[f_k]$.

As an example, consider the following 1001-element signal `xn`, which consists of two sinusoids plus noise:

```
randn;
fs = 1000;                % Sampling frequency
t = (0:fs)/fs;            % One second worth of samples
A = [1 2];                % Sinusoid amplitudes (row vector)
f = [150;140];            % Sinusoid frequencies (column vector)
xn = A*sin(2*pi*f*t) + 0.1*randn(size(t));
```

Note The three last lines illustrate a convenient and general way to express the sum of sinusoids.

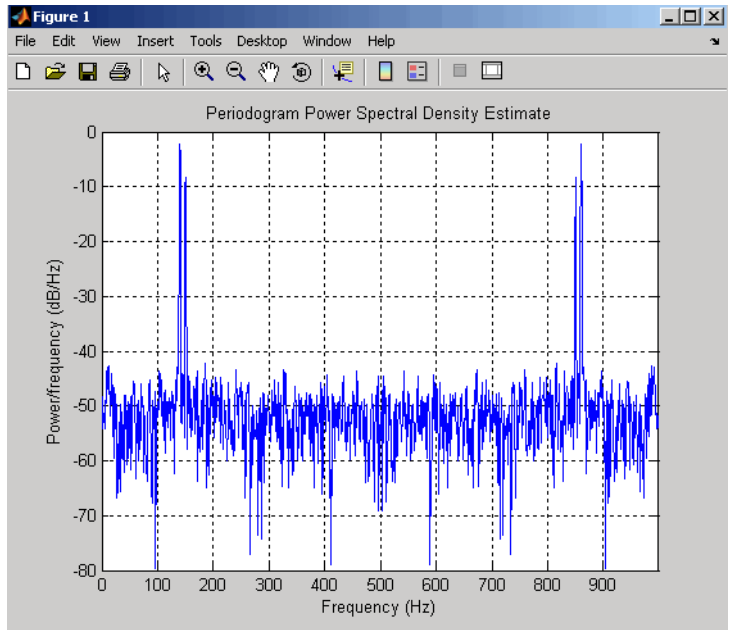
Together they are equivalent to `xn = sin(2*pi*150*t) + 2*sin(2*pi*140*t) + 0.1*randn(size(t));`

The periodogram estimate of the PSD can be computed by creating a periodogram object

```
Hs = spectrum.periodogram('Hamming');
```

and a plot of the estimate can be displayed with the `psd` method:

```
psd(Hs,xn,'Fs',fs,'NFFT',1024,'SpectrumType','twosided')
```



The average power can be computed by approximating the integral with the following sum:

```
Hdsp2= psd(Hs,xn,'Fs',fs,'NFFT',1024,'SpectrumType','twosided');
Pow = avgpower(Hdsp2)

Pow =
    2.5059
```

You can also compute the average power from the one-sided PSD estimate:

```
Hdsp3= psd(Hs,xn,'Fs',fs,'NFFT',1024,'SpectrumType','onesided');
Pow = avgpower(Hdsp3)

Pow =
```

```
2.5059
```

Algorithm. Spectrum.periodogram computes and scales the output of the FFT to produce the power vs. frequency plot as follows. For a detailed example, see Tech Note 1702 on the MathWorks website.

1. For rapid and efficient processing, make the input signal length (`nfft`) equal to a power of 2 by padding it with zeroes.
2. The `nfft` is now even, the magnitude of the resulting FFT is symmetric, and the first (1+`nfft`/2) points are unique. Calculate the number of unique points and keep only those unique points.
3. Calculate the magnitude of the FFT.
4. Scale the FFT so it is not a function of the input length.
5. Calculate the square of the magnitude of the scaled FFT (and multiply by 2 since in Step 2, half of the points were dropped).
6. Create a frequency vector from the number of unique points, the `nfft` and the sampling frequency.
7. Plot the resulting magnitude squared FFT vs. the frequency.

Performance of the Periodogram

The following sections discuss the performance of the periodogram with regard to the issues of leakage, resolution, bias, and variance.

Spectral Leakage. Consider the PSD of a finite-length signal $x_L[n]$, as discussed in the "Periodogram" on page 6-10 section. It is frequently useful to interpret $x_L[n]$ as the result of multiplying an infinite signal, $x[n]$, by a finite-length rectangular window, $w_R[n]$:

$$x_L[n] = x[n] \cdot w_R[n]$$

Because multiplication in the time domain corresponds to convolution in the frequency domain, the Fourier transform of the expression above is

$$X_L(f) = \frac{1}{f_s}\int_{-f_s/2}^{f_s/2} X(\rho)W_R(f-\rho)d\rho$$

The expression developed earlier for the periodogram,

$$\hat{P}_{xx}(f) = \frac{|X_L(f)|^2}{f_s L}$$

illustrates that the periodogram is also influenced by this convolution.

The effect of the convolution is best understood for sinusoidal data. Suppose that $x[n]$ is composed of a sum of M complex sinusoids:

$$x[n] = \sum_{k=1}^{M} A_k e^{j\omega_k n}$$

Its spectrum is

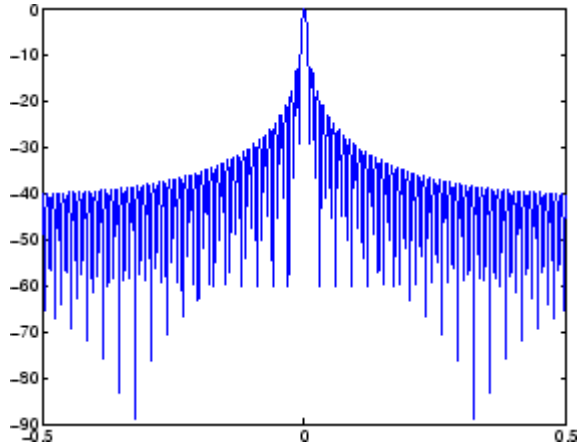
$$X(f) = f_s \sum_{k=1}^{M} A_k \delta(f-f_k$$

which for a finite-length sequence becomes

$$X_L(f) = \int_{-f_s/2}^{f_s/2}\left(\sum_{k=1}^{M} A_k\delta(\rho-f_k)\right)W_R(f-\rho)d\rho = \sum_{k=1}^{M} A_k W_R(f-f_k)$$

So in the spectrum of the finite-length signal, the Dirac deltas have been replaced by terms of the form $W_R(f-f_k)$, which corresponds to the frequency response of a rectangular window centered on the frequency f_k.

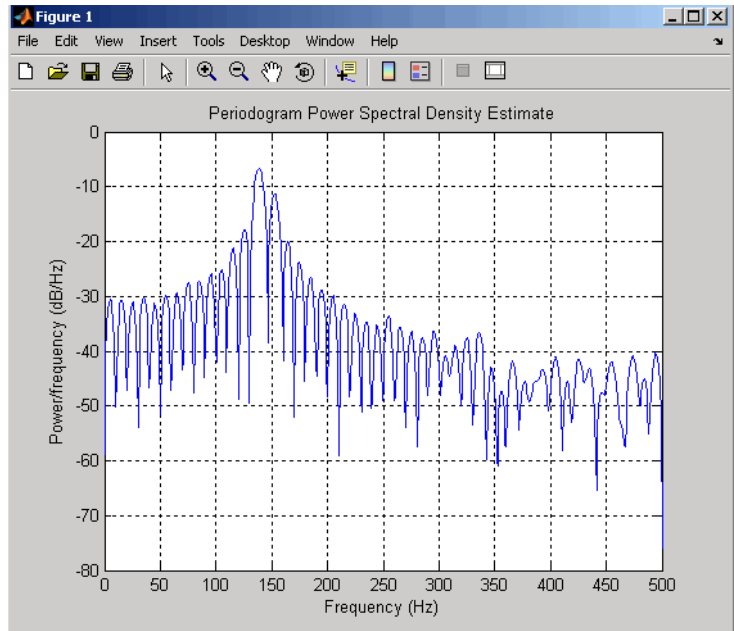
The frequency response of a rectangular window has the shape of a sinc signal, as shown below.



The plot displays a main lobe and several side lobes, the largest of which is approximately 13.5 dB below the mainlobe peak. These lobes account for the effect known as *spectral leakage*. While the infinite-length signal has its power concentrated exactly at the discrete frequency points f_k, the windowed (or truncated) signal has a continuum of power "leaked" around the discrete frequency points f_k.

Because the frequency response of a short rectangular window is a much poorer approximation to the Dirac delta function than that of a longer window, spectral leakage is especially evident when data records are short. Consider the following sequence of 100 samples:

```
randn;
fs = 1000;                  % Sampling frequency
t = (0:fs/10)/fs;           % One-tenth second worth of samples
A = [1 2];                  % Sinusoid amplitudes
f = [150;140];              % Sinusoid frequencies
xn = A*sin(2*pi*f*t) + 0.1*randn(size(t));
Hs = spectrum.periodogram;
psd(Hs,xn,'Fs',fs,'NFFT',1024)
```



It is important to note that the effect of spectral leakage is contingent solely on the length of the data record. It is *not* a consequence of the fact that the periodogram is computed at a finite number of frequency samples.

Resolution. *Resolution* refers to the ability to discriminate spectral features, and is a key concept on the analysis of spectral estimator performance.

In order to resolve two sinusoids that are relatively close together in frequency, it is necessary for the difference between the two frequencies to be greater than the width of the mainlobe of the leaked spectra for either one of these sinusoids. The mainlobe width is defined to be the width of the mainlobe at the point where the power is half the peak mainlobe power (i.e., 3 dB width). This width is approximately equal to f_s / L.

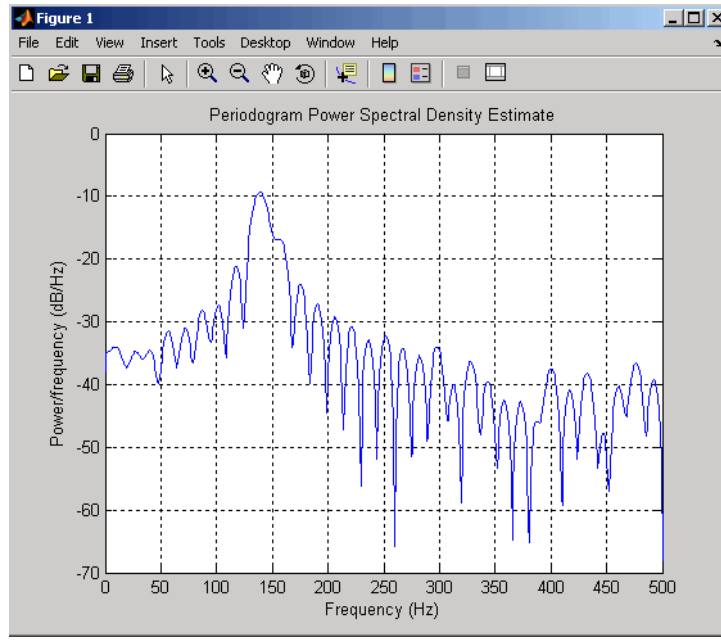
In other words, for two sinusoids of frequencies f_1 and f_2, the resolvability condition requires that

$$\Delta f = (f_1 - f_2) > \frac{f_s}{L}$$

In the example above, where two sinusoids are separated by only 10 Hz, the data record must be greater than 100 samples to allow resolution of two distinct sinusoids by a periodogram.

Consider a case where this criterion is not met, as for the sequence of 67 samples below:

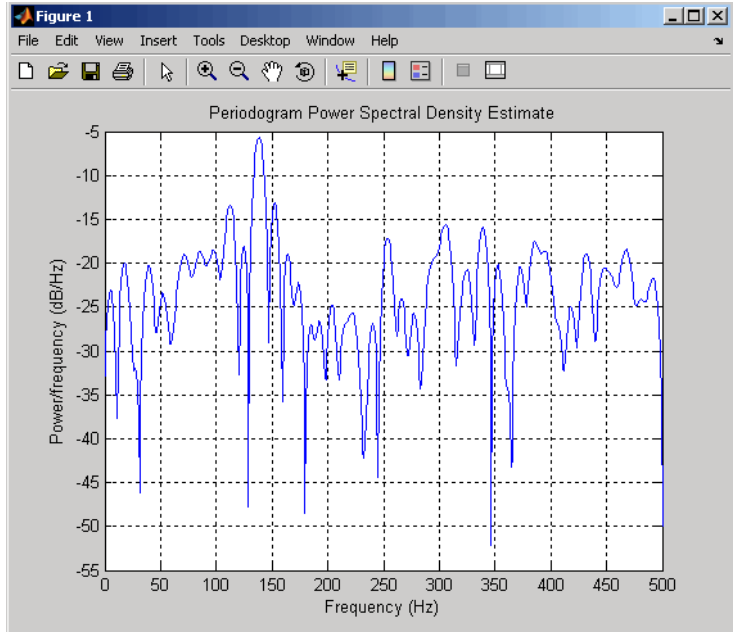
```
randn;
fs = 1000;                    % Sampling frequency
t = (0:fs/15)./fs;            % 67 samples
A = [1 2];                    % Sinusoid amplitudes
f = [150;140];                % Sinusoid frequencies
xn = A*sin(2*pi*f*t) + 0.1*randn(size(t));
Hs=spectrum.periodogram;
psd(Hs,xn,'Fs',fs,'NFFT',1024)
```



The above discussion about resolution did not consider the effects of noise since the signal-to-noise ratio (SNR) has been relatively high thus far. When the SNR is low, true spectral features are much harder to distinguish, and

noise artifacts appear in spectral estimates based on the periodogram. The example below illustrates this:

```
randn;
fs = 1000;                  % Sampling frequency
t = (0:fs/10)./fs;          % One-tenth second worth of samples
A = [1 2];                  % Sinusoid amplitudes
f = [150;140];              % Sinusoid frequencies
xn = A*sin(2*pi*f*t) + 2*randn(size(t));
Hs=spectrum.periodogram;
psd(Hs,xn,'Fs',fs,'NFFT',1024)
```



Bias of the Periodogram. The periodogram is a biased estimator of the PSD. Its expected value can be shown to be

$$E\left\{\frac{\left|X_L(f)\right|^2}{f_s L}\right\} = \frac{1}{f_s L}\int_{-f_s/2}^{f_s/2} P_{xx}(\rho)\left|W_R(f-\rho)\right|^2 d\rho$$

which is similar to the first expression for $X_L(f)$ in "Spectral Leakage" on page 6-13, except that the expression here is in terms of average power rather than magnitude. This suggests that the estimates produced by the periodogram correspond to a *leaky* PSD rather than the true PSD.

Note that $\left|W_R(f-\rho)\right|^2$ essentially yields a triangular Bartlett window (which is apparent from the fact that the convolution of two rectangular pulses is a triangular pulse). This results in a height for the largest sidelobes of the leaky power spectra that is about 27 dB below the mainlobe peak; i.e., about twice the frequency separation relative to the non-squared rectangular window.

The periodogram is asymptotically unbiased, which is evident from the earlier observation that as the data record length tends to infinity, the frequency response of the rectangular window more closely approximates the Dirac delta function (also true for a Bartlett window). However, in some cases the periodogram is a poor estimator of the PSD even when the data record is long. This is due to the variance of the periodogram, as explained below.

Variance of the Periodogram. The variance of the periodogram can be shown to be approximately

$$var\left\{\frac{\left|X_L(f)\right|^2}{f_s L}\right\} = P_{xx}^2(f)\left[1+\left(\frac{\sin(2\pi L f/f_s)}{L\sin(2\pi f/f_s)}\right)^2\right]$$

which indicates that the variance does not tend to zero as the data length L tends to infinity. In statistical terms, the periodogram is not a consistent estimator of the PSD. Nevertheless, the periodogram can be a useful tool for spectral estimation in situations where the SNR is high, and especially if the data record is long.

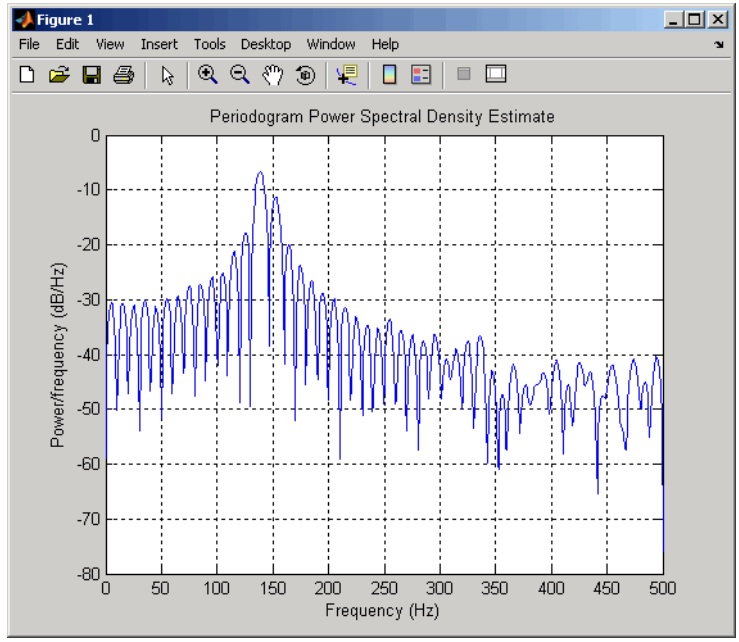
The Modified Periodogram

The *modified periodogram* windows the time-domain signal prior to computing the FFT in order to smooth the edges of the signal. This has the effect of reducing the height of the sidelobes or spectral leakage. This phenomenon gives rise to the interpretation of sidelobes as spurious frequencies introduced into the signal by the abrupt truncation that occurs when a rectangular window is used. For nonrectangular windows, the end points of the truncated signal are attenuated smoothly, and hence the spurious frequencies

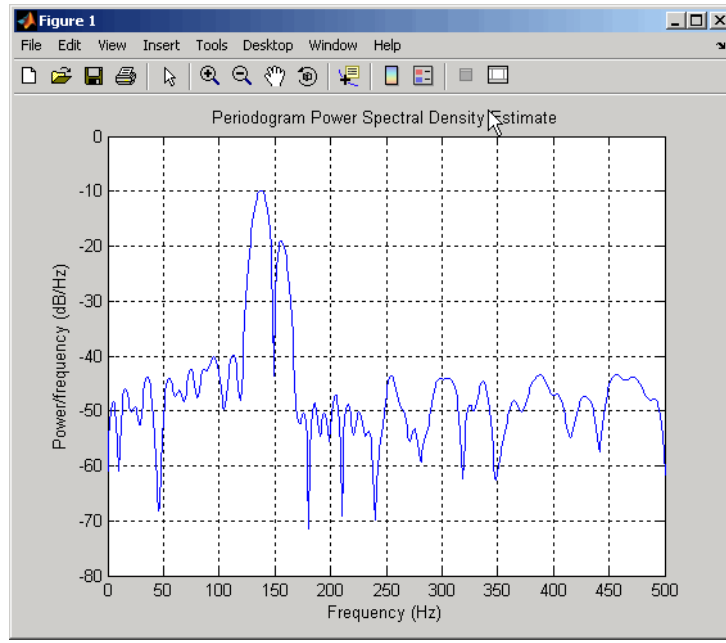
introduced are much less severe. On the other hand, nonrectangular windows also broaden the mainlobe, which results in a net reduction of resolution.

The `periodogram` function allows you to compute a modified periodogram by specifying the window to be used on the data. For example, compare a default rectangular window and a Hamming window:

```
randn;
fs = 1000;                      % Sampling frequency
t = (0:fs/10)./fs;              % One-tenth second worth of samples
A = [1 2];                      % Sinusoid amplitudes
f = [150;140];                  % Sinusoid frequencies
xn = A*sin(2*pi*f*t) + 0.1*randn(size(t));
Hrect = spectrum.periodogram;
psd(Hrect,xn,'Fs',fs,'NFFT',1024);
```



```
Hhamm = spectrum.periodogram('Hamming');
psd(Hhamm,xn,'Fs',fs,'NFFT',1024);
```



You can verify that although the sidelobes are much less evident in the Hamming-windowed periodogram, the two main peaks are wider. In fact, the 3 dB width of the mainlobe corresponding to a Hamming window is approximately twice that of a rectangular window. Hence, for a fixed data length, the PSD resolution attainable with a Hamming window is approximately half that attainable with a rectangular window. The competing interests of mainlobe width and sidelobe height can be resolved to some extent by using variable windows such as the Kaiser window.

Nonrectangular windowing affects the average power of a signal because some of the time samples are attenuated when multiplied by the window. To compensate for this, the `periodogram` function and the Welch PSD normalize the window to have an average power of unity. This ensures that the measured average power is generally independent of window choice. If the frequency components are not well resolved by the PSD estimators, the window choice does affect the average power.

The modified periodogram estimate of the PSD is

$$\hat{P}_{xx}(f) = \frac{\left|X_L(f)\right|^2}{f_s L U}$$

where U is the window normalization constant.

$$U = \frac{1}{L}\sum_{n=0}^{L-1}\left|w(n)\right|^2$$

. For large values of L, U tends to become independent of window length. The addition of U as a normalization constant ensures that the modified periodogram is asymptotically unbiased.

Welch's Method

An improved estimator of the PSD is the one proposed by Welch [8]. The method consists of dividing the time series data into (possibly overlapping) segments, computing a modified periodogram of each segment, and then averaging the PSD estimates. The result is Welch's PSD estimate.

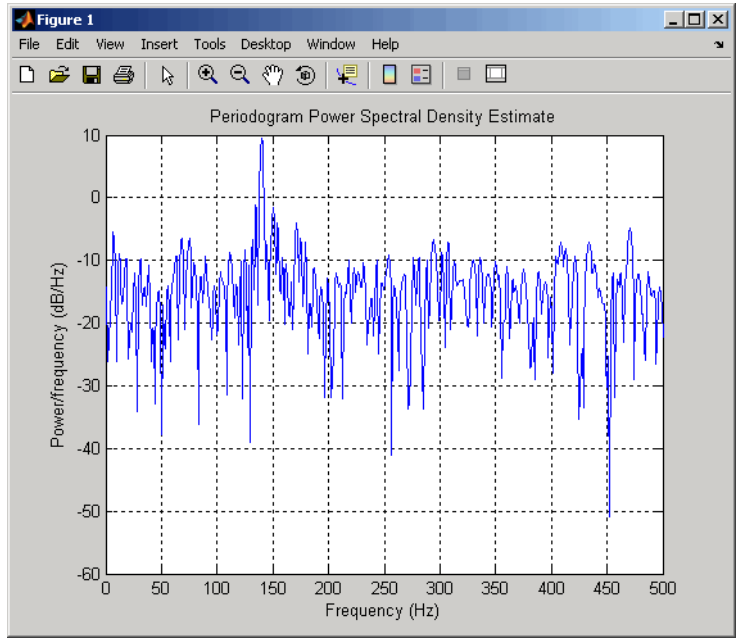
Welch's method is implemented in the toolbox by the `spectrum.welch` object or `pwelch` function. By default, the data is divided into eight segments with 50% overlap between them. A Hamming window is used to compute the modified periodogram of each segment.

The averaging of modified periodograms tends to decrease the variance of the estimate relative to a single periodogram estimate of the entire data record. Although overlap between segments tends to introduce redundant information, this effect is diminished by the use of a nonrectangular window, which reduces the importance or *weight* given to the end samples of segments (the samples that overlap).

However, as mentioned above, the combined use of short data records and nonrectangular windows results in reduced resolution of the estimator. In summary, there is a trade-off between variance reduction and resolution. One can manipulate the parameters in Welch's method to obtain improved estimates relative to the periodogram, especially when the SNR is low. This is illustrated in the following example.

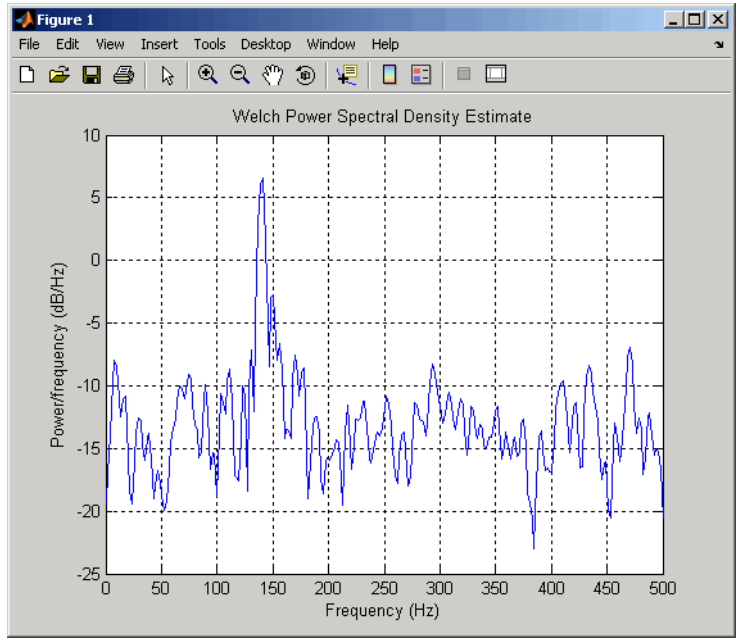
Consider an original signal consisting of 301 samples:

```
randn;
fs = 1000;              % Sampling frequency
t = (0:0.3*fs)./fs;     % 301 samples
A = [2 8];              % Sinusoid amplitudes (row vector)
f = [150;140];          % Sinusoid frequencies (column vector)
xn = A*sin(2*pi*f*t) + 5*randn(size(t));
Hs = spectrum.periodogram('rectangular');
psd(Hs,xn,'Fs',fs,'NFFT',1024);
```



We can obtain Welch's spectral estimate for 3 segments with 50% overlap with

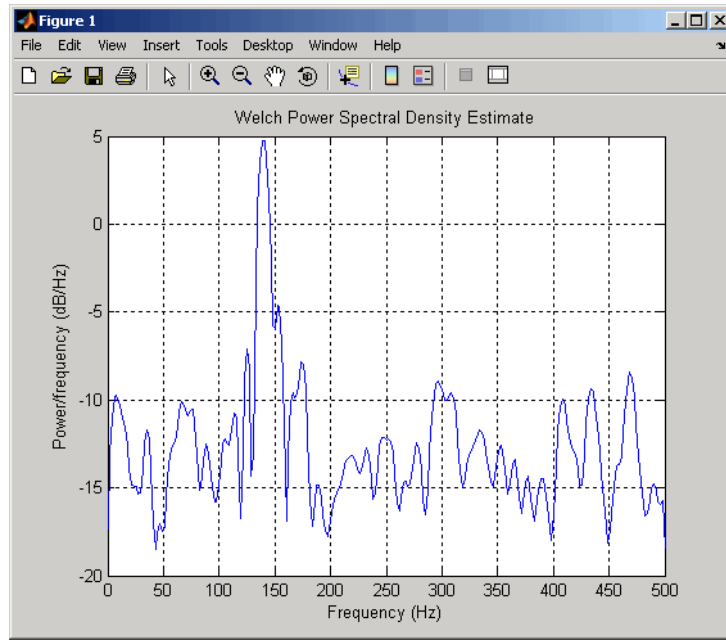
```
Hs = spectrum.welch('rectangular',150,50);
psd(Hs,xn,'Fs',fs,'NFFT',512);
```



In the periodogram above, noise and the leakage make one of the sinusoids essentially indistinguishable from the artificial peaks. In contrast, although the PSD produced by Welch's method has wider peaks, you can still distinguish the two sinusoids, which stand out from the "noise floor."

However, if we try to reduce the variance further, the loss of resolution causes one of the sinusoids to be lost altogether:

```
Hs = spectrum.welch('rectangular',100,75);
psd(Hs,xn,'Fs',fs,'NFFT',512);
```



For a more detailed discussion of Welch's method of PSD estimation, see Kay [2] and Welch [8].

Bias and Normalization in Welch's Method

Welch's method yields a biased estimator of the PSD. The expected value can be found to be

$$E\{\hat{P}_{welch}\} = \frac{1}{f_s L_s U} \int_{-f_s/2}^{f_s/2} P_{xx}(\rho)|W(f-\rho)|^2 d\rho$$

where L_s is the length of the data segments and U is the same normalization constant present in the definition of the modified periodogram. As is the case for all periodograms, Welch's estimator is asymptotically unbiased. For

a fixed length data record, the bias of Welch's estimate is larger than that of the periodogram because $L_s < L$.

The variance of Welch's estimator is difficult to compute because it depends on both the window used and the amount of overlap between segments. Basically, the variance is inversely proportional to the number of segments whose modified periodograms are being averaged.

Multitaper Method

The periodogram can be interpreted as filtering a length L signal, $x_L[n]$, through a filter bank (a set of filters in parallel) of L FIR bandpass filters. The 3 dB bandwidth of each of these bandpass filters can be shown to be approximately equal to f_s / L. The magnitude response of each one of these bandpass filters resembles that of the rectangular window discussed in "Spectral Leakage" on page 6-13. The periodogram can thus be viewed as a computation of the power of each filtered signal (i.e., the output of each bandpass filter) that uses just one sample of each filtered signal and assumes that the PSD of $x_L[n]$ is constant over the bandwidth of each bandpass filter.

As the length of the signal increases, the bandwidth of each bandpass filter decreases, making it a more selective filter, and improving the approximation of constant PSD over the bandwidth of the filter. This provides another interpretation of why the PSD estimate of the periodogram improves as the length of the signal increases. However, there are two factors apparent from this standpoint that compromise the accuracy of the periodogram estimate. First, the rectangular window yields a poor bandpass filter. Second, the computation of the power at the output of each bandpass filter relies on a single sample of the output signal, producing a very crude approximation.

Welch's method can be given a similar interpretation in terms of a filter bank. In Welch's implementation, several samples are used to compute the output power, resulting in reduced variance of the estimate. On the other hand, the bandwidth of each bandpass filter is larger than that corresponding to the periodogram method, which results in a loss of resolution. The filter bank model thus provides a new interpretation of the compromise between variance and resolution.

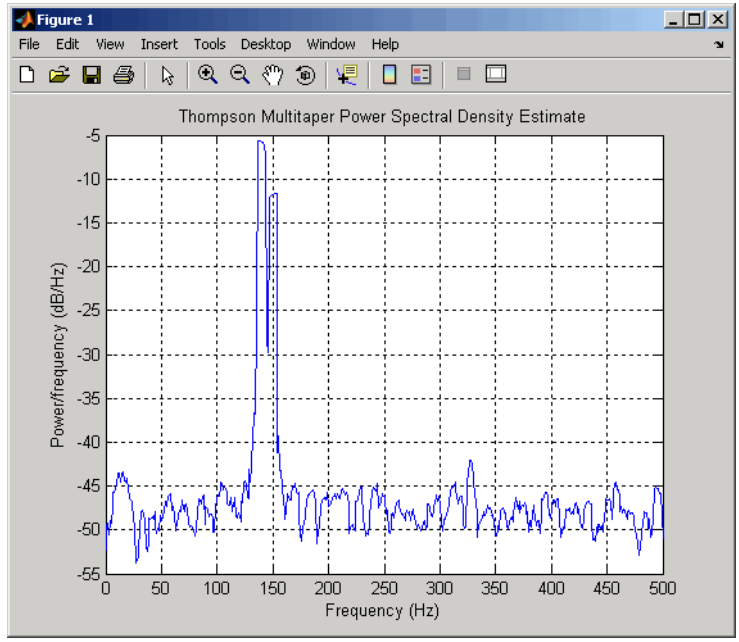
Thompson's *multitaper method* (MTM) builds on these results to provide an improved PSD estimate. Instead of using bandpass filters that are essentially

rectangular windows (as in the periodogram method), the MTM method uses a bank of optimal bandpass filters to compute the estimate. These optimal FIR filters are derived from a set of sequences known as *discrete prolate spheroidal sequences* (DPSSs, also known as *Slepian sequences*).

In addition, the MTM method provides a time-bandwidth parameter with which to balance the variance and resolution. This parameter is given by the time-bandwidth product, *NW* and it is directly related to the number of tapers used to compute the spectrum. There are always 2**NW*-1 tapers used to form the estimate. This means that, as *NW* increases, there are more estimates of the power spectrum, and the variance of the estimate decreases. However, the bandwidth of each taper is also proportional to *NW*, so as *NW* increases, each estimate exhibits more spectral leakage (i.e., wider peaks) and the overall spectral estimate is more biased. For each data set, there is usually a value for *NW* that allows an optimal trade-off between bias and variance.

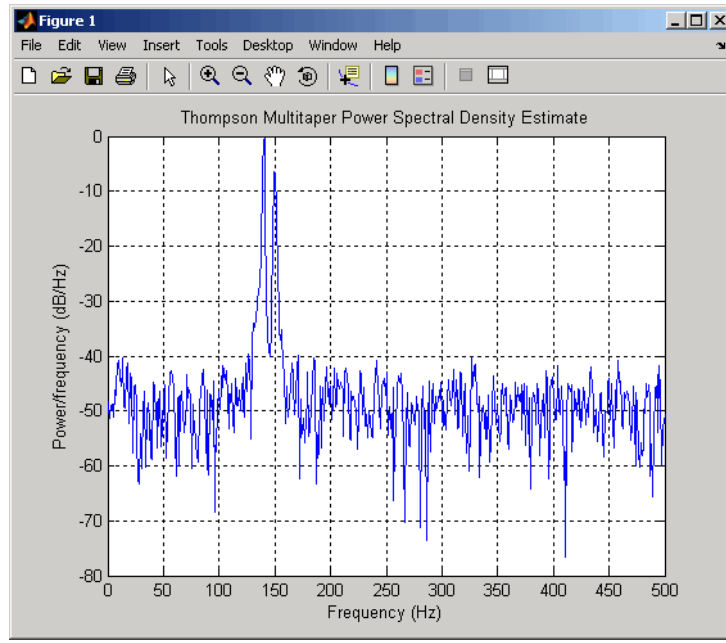
The Signal Processing Toolbox function that implements the MTM method is `pmtm` and the object that implements it is `spectrum.mtm`. Use `spectrum.mtm` to compute the PSD of `xn` from the previous examples:

```
randn;
fs = 1000;                % Sampling frequency
t = (0:fs)/fs;            % One second worth of samples
A = [1 2];                % Sinusoid amplitudes
f = [150;140];            % Sinusoid frequencies
xn = A*sin(2*pi*f*t) + 0.1*randn(size(t));
Hs1 = spectrum.mtm(4,'adapt');
psd(Hs1,xn,'Fs',fs,'NFFT',1024)
```



By lowering the time-bandwidth product, you can increase the resolution at the expense of larger variance:

```
Hs2 = spectrum.mtm(3/2,'adapt');
psd(Hs2,xn,'Fs',fs,'NFFT',1024)
```



Note that the average power is conserved in both cases:

```
Hs1p = psd(Hs1,xn,'Fs',fs,'NFFT',1024);
Pow1 = avgpower(Hs1p)

Pow1 =
    2.4926
Hs2p = psd(Hs2,xn,'Fs',fs,'NFFT',1024);
Pow2 = avgpower(Hs2p)

Pow2 =
    2.4927
```

This method is more computationally expensive than Welch's method due to the cost of computing the discrete prolate spheroidal sequences. For long data series (10,000 points or more), it is useful to compute the DPSSs once and save

them in a MAT-file. The M-files `dpsssave`, `dpssload`, `dpssdir`, and `dpssclear` are provided to keep a database of saved DPSSs in the MAT-file `dpss.mat`.

Cross-Spectral Density Function

The PSD is a special case of the *cross spectral density* (*CPSD*) function, defined between two signals x_n and y_n as

$$P_{xy}(\omega) = \frac{1}{2\pi} \sum_{m=-\infty}^{\infty} R_{xy}(m) e^{-j\omega m}$$

As is the case for the correlation and covariance sequences, the toolbox *estimates* the PSD and CPSD because signal lengths are finite.

To estimate the cross-spectral density of two equal length signals `x` and `y` using Welch's method, the `cpsd` function forms the periodogram as the product of the FFT of `x` and the conjugate of the FFT of `y`. Unlike the real-valued PSD, the CPSD is a complex function. `cpsd` handles the sectioning and windowing of `x` and `y` in the same way as the `pwelch` function:

```
Sxy = cpsd(x,y,nwin,noverlap,nfft,fs)
```

Transfer Function Estimate

One application of Welch's method is nonparametric system identification. Assume that H is a linear, time invariant system, and $x(n)$ and $y(n)$ are the input to and output of H, respectively. Then the power spectrum of $x(n)$ is related to the CPSD of $x(n)$ and $y(n)$ by

$$P_{yx}(\omega) = H(\omega) P_{xx}(\omega)$$

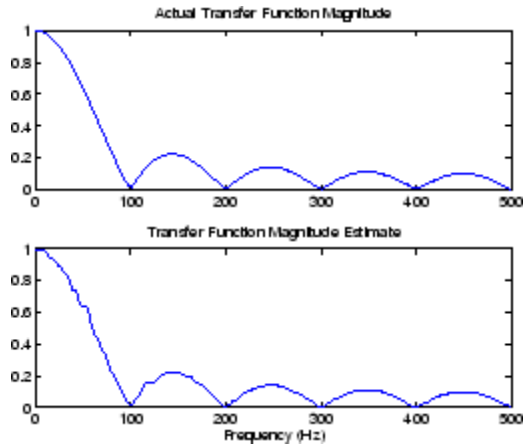
An estimate of the transfer function between $x(n)$ and $y(n)$ is

$$\hat{H}(\omega) = \frac{\hat{P}_{yx}(\omega)}{\hat{P}_{xx}(\omega)}$$

This method estimates both magnitude and phase information. The `tfestimate` function uses Welch's method to compute the CPSD and power spectrum, and then forms their quotient for the transfer function estimate. Use `tfestimate` the same way that you use the `cpsd` function.

Filter the signal `xn` with an FIR filter, then plot the actual magnitude response and the estimated response:

```
h = ones(1,10)/10;              % Moving-average filter
yn = filter(h,1,xn);
[HEST,f] = tfestimate(xn,yn,256,128,256,fs);
H = freqz(h,1,f,fs);
subplot(2,1,1); plot(f,abs(H));
title('Actual Transfer Function Magnitude');
subplot(2,1,2); plot(f,abs(HEST));
title('Transfer Function Magnitude Estimate');
xlabel('Frequency (Hz)');
```



Coherence Function

The magnitude-squared coherence between two signals $x(n)$ and $y(n)$ is

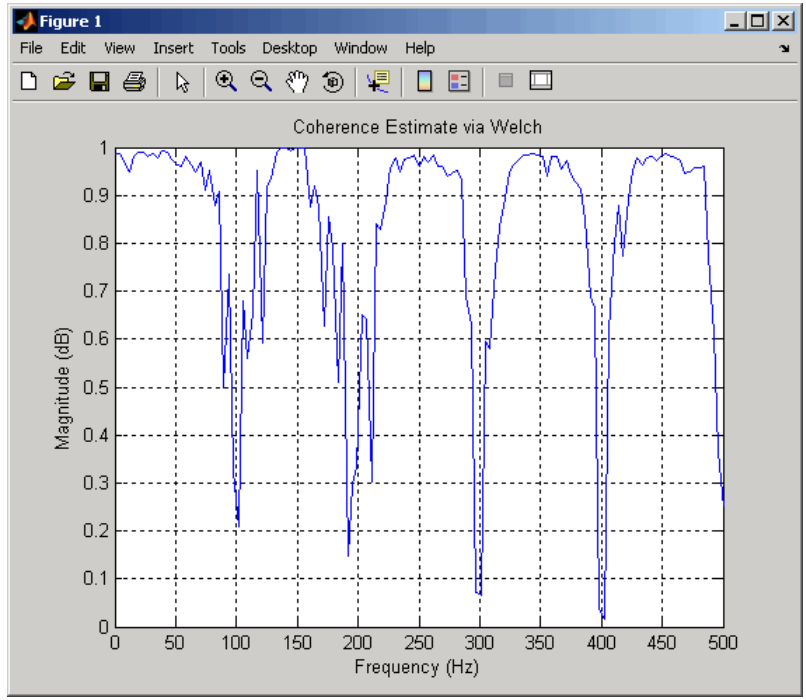
$$C_{xy}(\omega) = \frac{\left|P_{xy}(\omega)\right|^2}{P_{xx}(\omega)P_{yy}(\omega)}$$

This quotient is a real number between 0 and 1 that measures the correlation between $x(n)$ and $y(n)$ at the frequency ω.

The `mscohere` function takes sequences `x` and `y`, computes their power spectra and CPSD, and returns the quotient of the magnitude squared of the CPSD and the product of the power spectra. Its options and operation are similar to the `cpsd` and `tfestimate` functions.

The coherence function of `xn` and the filter output `yn` versus frequency is

```
mscohere(xn,yn,256,128,256,fs)
```



If the input sequence length `nfft`, window length `window`, and the number of overlapping data points in a window `numoverlap`, are such that `mscohere` operates on only a single record, the function returns all ones. This is because the coherence function for linearly dependent data is one.

Parametric Methods

Parametric methods can yield higher resolutions than nonparametric methods in cases when the signal length is short. These methods use a different approach to spectral estimation; instead of trying to estimate the PSD directly from the data, they *model* the data as the output of a linear system driven by white noise, and then attempt to estimate the parameters of that linear system.

The most commonly used linear system model is the *all-pole model*, a filter with all of its zeroes at the origin in the *z*-plane. The output of such a filter for white noise input is an autoregressive (AR) process. For this reason, these methods are sometimes referred to as *AR methods* of spectral estimation.

The AR methods tend to adequately describe spectra of data that is "peaky," that is, data whose PSD is large at certain frequencies. The data in many practical applications (such as speech) tends to have "peaky spectra" so that AR models are often useful. In addition, the AR models lead to a system of linear equations which is relatively simple to solve.

Signal Processing Toolbox AR methods for spectral estimation include:

- Yule-Walker AR method (autocorrelation method)
- Burg method
- Covariance method
- Modified covariance method

All AR methods yield a PSD estimate given by

$$\hat{P}_{AR}(f) = \frac{1}{f_s} \frac{\varepsilon_p}{\left| 1 + \sum_{k=1}^{p} \hat{a}_p(k) e^{-2\pi j k f / f_s} \right|^2}$$

The different AR methods estimate the AR parameters $a_p(k)$ slightly differently, yielding different PSD estimates. The following table provides a summary of the different AR methods.

AR Methods

	Burg	Covariance	Modified Covariance	Yule-Walker
Characteristics	Does not apply window to data	Does not apply window to data	Does not apply window to data	Applies window to data
	Minimizes the forward and backward prediction errors in the least squares sense, with the AR coefficients constrained to satisfy the L-D recursion	Minimizes the forward prediction error in the least squares sense	Minimizes the forward and backward prediction errors in the least squares sense	Minimizes the forward prediction error in the least squares sense (also called "Autocorrelation method")
Advantages	High resolution for short data records	Better resolution than Y-W for short data records (more accurate estimates)	High resolution for short data records	Performs as well as other methods for large data records
	Always produces a stable model	Able to extract frequencies from data consisting of p or more pure sinusoids	Able to extract frequencies from data consisting of p or more pure sinusoids	Always produces a stable model
			Does not suffer spectral line-splitting	

AR Methods (Continued)

	Burg	Covariance	Modified Covariance	Yule-Walker
Disadvantages	Peak locations highly dependent on initial phase	May produce unstable models	May produce unstable models	Performs relatively poorly for short data records
	May suffer spectral line-splitting for sinusoids in noise, or when order is very large	Frequency bias for estimates of sinusoids in noise	Peak locations slightly dependent on initial phase	Frequency bias for estimates of sinusoids in noise
	Frequency bias for estimates of sinusoids in noise		Minor frequency bias for estimates of sinusoids in noise	
Conditions for Nonsingularity		Order must be less than or equal to half the input frame size	Order must be less than or equal to 2/3 the input frame size	Because of the biased estimate, the autocorrelation matrix is guaranteed to positive-definite, hence nonsingular

Yule-Walker AR Method

The *Yule-Walker AR method* of spectral estimation computes the AR parameters by forming a biased estimate of the signal's autocorrelation function, and solving the least squares minimization of the forward prediction error. This results in the Yule-Walker equations.

$$\begin{bmatrix} r(1) & r(2)^* & \cdots & r(p)^* \\ r(2) & r(1) & \cdots & r(p-1)^* \\ \vdots & \ddots & \ddots & \vdots \\ r(p) & \cdots & r(2) & r(1) \end{bmatrix} \begin{bmatrix} a(2) \\ a(3) \\ \vdots \\ a(p+1) \end{bmatrix} = \begin{bmatrix} -r(2) \\ -r(3) \\ \vdots \\ -r(p+1) \end{bmatrix}$$

The Yule-Walker AR method produces the same results as a maximum entropy estimator. For more information, see page 155 of item [2] in the "Selected Bibliography" on page 6-49.

The use of a biased estimate of the autocorrelation function ensures that the autocorrelation matrix above is positive definite. Hence, the matrix is invertible and a solution is guaranteed to exist. Moreover, the AR parameters thus computed always result in a stable all-pole model. The Yule-Walker equations can be solved efficiently via Levinson's algorithm, which takes advantage of the Toeplitz structure of the autocorrelation matrix.

The toolbox object `spectrum.yulear` and function `pyulear` implement the Yule-Walker AR method.

For example, compare the spectrum of a speech signal using Welch's method and the Yule-Walker AR method:

```
load mtlb
Hwelch = spectrum.welch('hamming',256,50);
psd(Hwelch,mtlb,'Fs',Fs,'NFFT',1024)
```

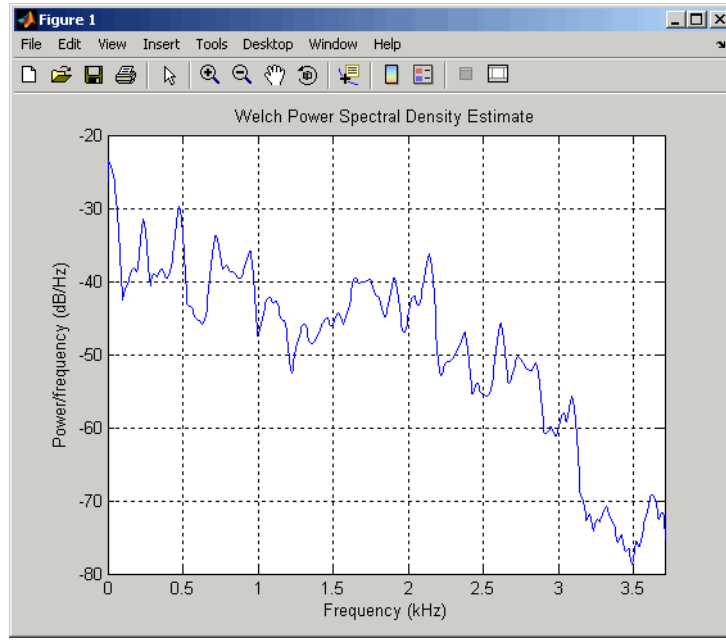
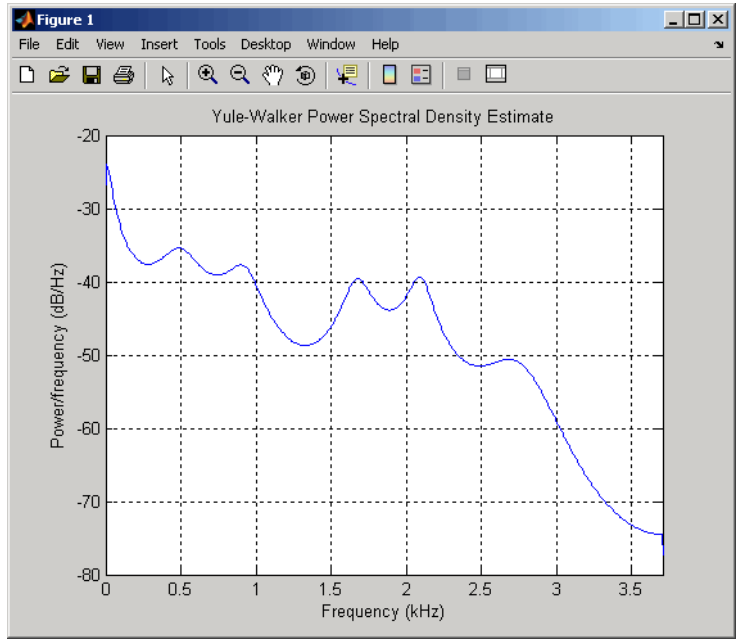


Figure 1
File Edit View Insert Tools Desktop Window Help
Welch Power Spectral Density Estimate
Power/frequency (dB/Hz)
-20
-30
-40
-50
-60
-70
-80
0
0.5
1
1.5
2
2.5
3
3.5
Frequency (kHz)

```
Hyulear = spectrum.yulear(14);
psd(Hyulear,mtlb,'Fs',Fs,'NFFT',1024)
```



The Yule-Walker AR spectrum is smoother than the periodogram because of the simple underlying all-pole model.

Burg Method

The Burg method for AR spectral estimation is based on minimizing the forward and backward prediction errors while satisfying the Levinson-Durbin recursion (see Marple [3], Chapter 7, and Proakis [6], Section 12.3.3). In contrast to other AR estimation methods, the Burg method avoids calculating the autocorrelation function, and instead estimates the reflection coefficients directly.

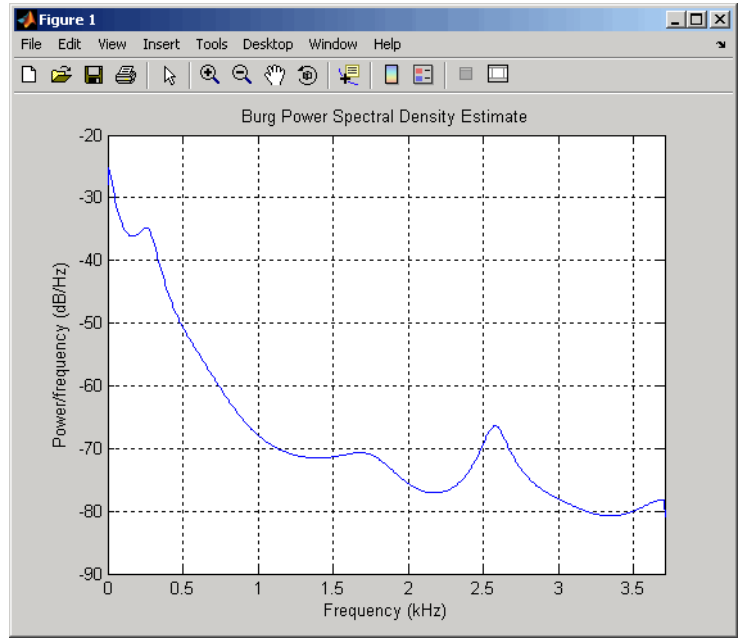
The primary advantages of the Burg method are resolving closely spaced sinusoids in signals with low noise levels, and estimating short data records, in which case the AR power spectral density estimates are very close to the

true values. In addition, the Burg method ensures a stable AR model and is computationally efficient.

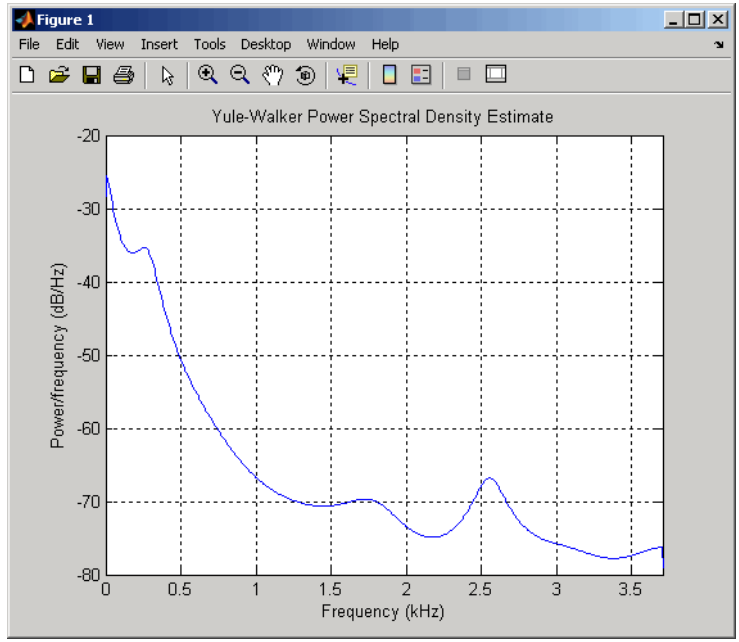
The accuracy of the Burg method is lower for high-order models, long data records, and high signal-to-noise ratios (which can cause *line splitting*, or the generation of extraneous peaks in the spectrum estimate). The spectral density estimate computed by the Burg method is also susceptible to frequency shifts (relative to the true frequency) resulting from the initial phase of noisy sinusoidal signals. This effect is magnified when analyzing short data sequences.

The toolbox object `spectrum.burg` and function `pburg` implement the Burg method. Compare the spectrum of the speech signal generated by both the Burg method and the Yule-Walker AR method. They are very similar for large signal lengths:

```
load mtlb
Hburg = spectrum.burg(14);      % 14th order model
psd(Hburg,mtlb(1:512),'Fs',Fs,'NFFT',1024)
```



```
Hyulear = spectrum.yulear(14);       % 14th order model
psd(Hyulear,mtlb(1:512),'Fs',Fs,'NFFT',1024)
```



Compare the spectrum of a noisy signal computed using the Burg method and the Welch method:

```
randn;
fs = 1000;                % Sampling frequency
t = (0:fs)/fs;            % One second worth of samples
A = [1 2];                % Sinusoid amplitudes
f = [150;140];            % Sinusoid frequencies
xn = A*sin(2*pi*f*t) + 0.1*randn(size(t));
Hwelch = spectrum.welch('hamming',256,50);
psd(Hwelch,xn,'Fs',fs,'NFFT',1024)
```

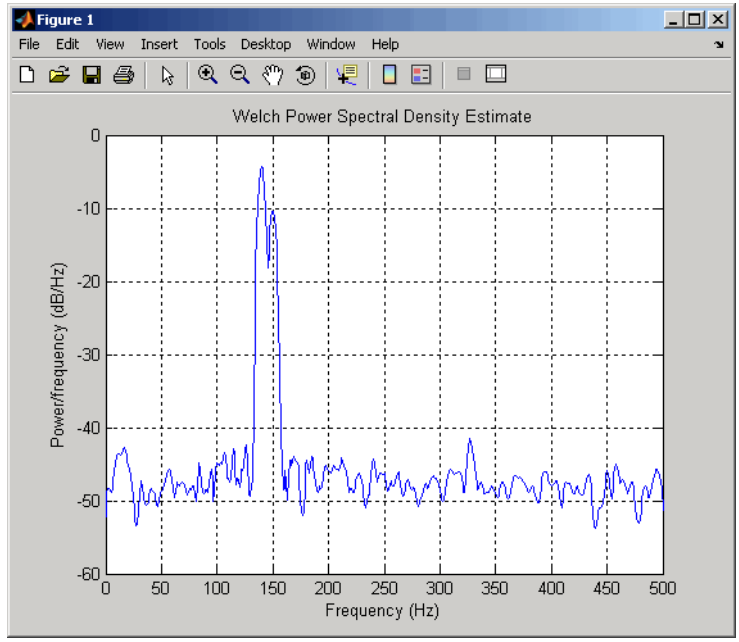
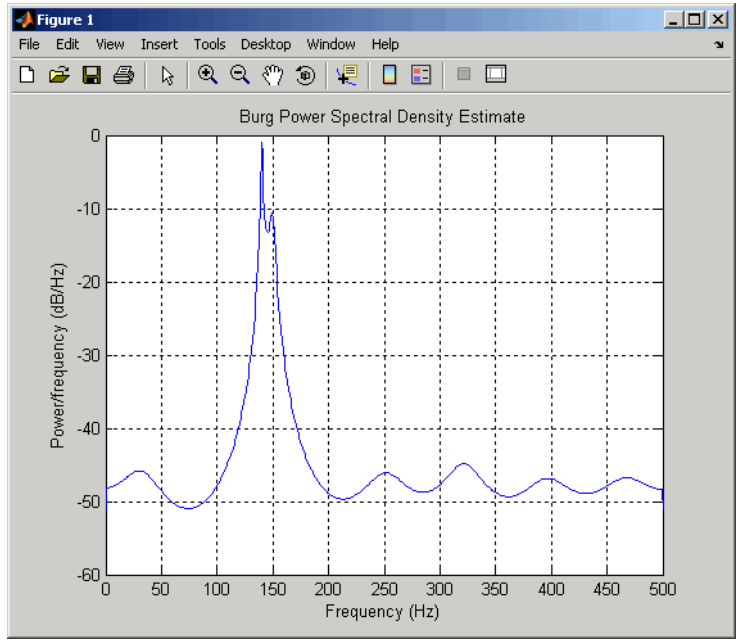


Figure 1
File Edit View Insert Tools Desktop Window Help
Welch Power Spectral Density Estimate
Power/frequency (dB/Hz)
Frequency (Hz)
0
-10
-20
-30
-40
-50
-60
0
50
100
150
200
250
300
350
400
450
500

```
Hburg = spectrum.burg(14);
psd(Hburg,xn,'Fs',fs,'NFFT',1024)
```



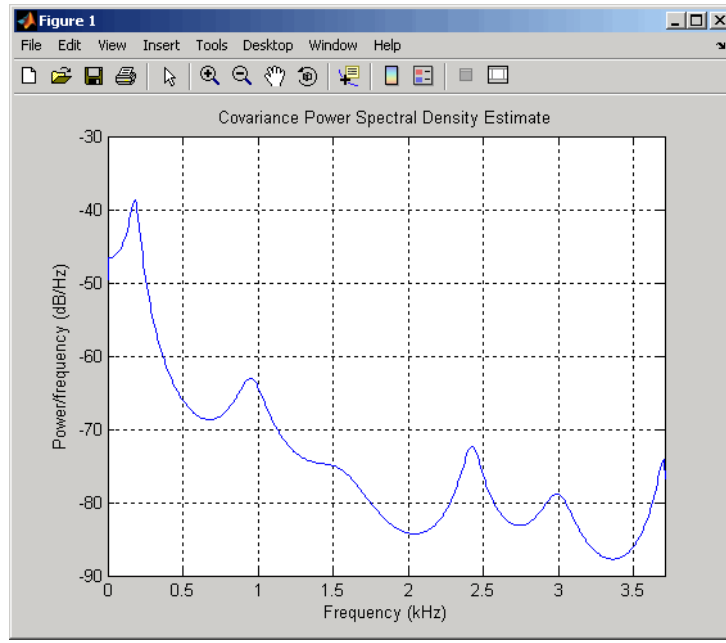
Note that, as the model order for the Burg method is reduced, a frequency shift due to the initial phase of the sinusoids will become apparent.

Covariance and Modified Covariance Methods

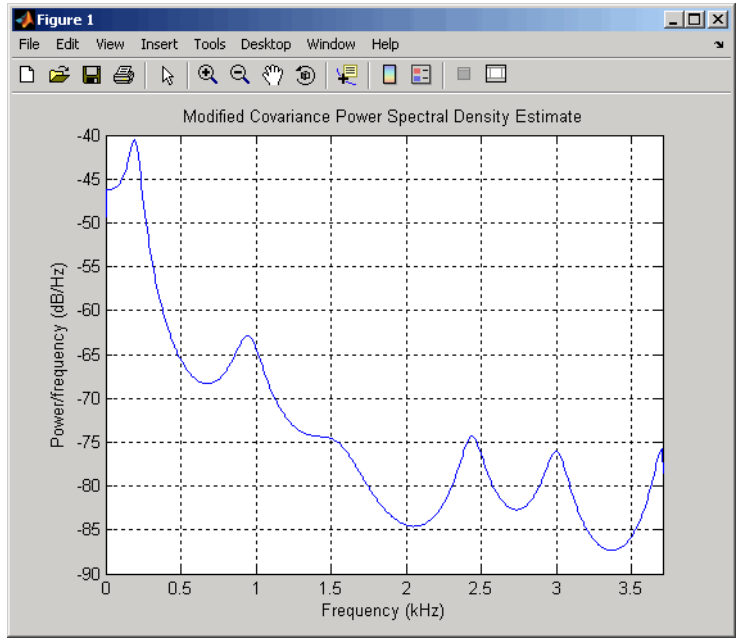
The covariance method for AR spectral estimation is based on minimizing the forward prediction error. The modified covariance method is based on minimizing the forward and backward prediction errors. The toolbox object `spectrum.cov` and function `pcov`, and object `spectrum.mcov` and function `pmcov` implement the respective methods.

Compare the spectrum of the speech signal generated by both the covariance method and the modified covariance method. They are nearly identical, even for a short signal length:

```
load mtlb
Hcov = spectrum.cov(14);       % 14th order model
psd(Hcov,mtlb(1:64),'Fs',Fs,'NFFT',1024)
```




```
Hmcov = spectrum.mcov(14);                % 14th order model
psd(Hmcov,mtlb(1:64),'Fs',Fs,'NFFT',1024)
```



MUSIC and Eigenvector Analysis Methods

The `spectrum.music` object and `pmusic` function, and `spectrum.eigenvector` object and `peig` function provide two related spectral analysis methods:

- `spectrum.music` and `pmusic` provide the multiple signal classification (MUSIC) method developed by Schmidt
- `spectrum.eigenvector` and `peig` provides the eigenvector (EV) method developed by Johnson

See Marple [3] (pgs. 373-378) for a summary of these methods.

Both of these methods are frequency estimator techniques based on eigenanalysis of the autocorrelation matrix. This type of spectral analysis categorizes the information in a correlation or data matrix, assigning information to either a signal subspace or a noise subspace.

Eigenanalysis Overview

Consider a number of complex sinusoids embedded in white noise. You can write the autocorrelation matrix R for this system as the sum of the signal autocorrelation matrix (S) and the noise autocorrelation matrix (W):

$$R = S + W$$

There is a close relationship between the eigenvectors of the signal autocorrelation matrix and the signal and noise subspaces. The eigenvectors v of S span the same signal subspace as the signal vectors. If the system contains M complex sinusoids and the order of the autocorrelation matrix is p, eigenvectors v_{M+1} through v_{p+1} span the noise subspace of the autocorrelation matrix.

Frequency Estimator Functions. To generate their frequency estimates, eigenanalysis methods calculate functions of the vectors in the signal and noise subspaces. Both the MUSIC and EV techniques choose a function that goes to infinity (denominator goes to zero) at one of the sinusoidal frequencies in the input signal. Using digital technology, the resulting estimate has sharp peaks at the frequencies of interest; this means that there might not be infinity values in the vectors.

The MUSIC estimate is given by the formula

$$P_{music}(f) = \frac{1}{\mathbf{e}^H(f)\left(\sum_{k=p+1}^{N} \mathbf{v}_k \mathbf{v}_k^H\right)\mathbf{e}(f)} = \frac{1}{\sum_{k=p+1}^{N} \left|\mathbf{v}_k^H \mathbf{e}(f)\right|^2}$$

where N is the size of the eigenvectors and $\mathbf{e}(f)$ is a vector of complex sinusoids.

$$\mathbf{e}(f) = [1 \;\; \exp(j2\pi f) \; \exp(j2\pi f \cdot 2) \; \exp(j2\pi f \cdot 4) \; \ldots \; \exp(j2\pi f \cdot (n-1))]^H$$

$\mathbf{v}$ represents the eigenvectors of the input signal's correlation matrix; $\mathbf{v}_k$ is the k^{th} eigenvector. H is the conjugate transpose operator. The eigenvectors used in the sum correspond to the smallest eigenvalues and span the noise subspace (p is the size of the signal subspace).

The expression $\mathbf{v}_k^H \mathbf{e}(f)$ is equivalent to a Fourier transform (the vector e(*f*) consists of complex exponentials). This form is useful for numeric computation because the FFT can be computed for each v_k and then the squared magnitudes can be summed.

The EV method weights the summation by the eigenvalues of the correlation matrix:

$$P_{ev}(f) = \frac{1}{\left(\sum_{k=p+1}^{N} \left| \mathbf{v}_k^H \mathbf{e}(f) \right|^2 \right) / \lambda_k}$$

The `pmusic` and `peig` functions in this toolbox interpret their first input either as a signal matrix or as a correlation matrix (if the `'corr'` input flag is set). In the former case, the singular value decomposition of the signal matrix is used to determine the signal and noise subspaces. In the latter case, the eigenvalue decomposition of the correlation matrix is used to determine the signal and noise subspaces.

Using FFT to Obtain Simple Spectral Analysis Plots

In general, you use `spectrum` objects to perform spectral analysis, but if you want to create spectral analysis plots manually starting with an FFT, follow this example. Assume you have a 200 Hz sinusoid signal and a sampling frequency of 1024.

```
Fs = 1024;            % Sampling frequency
t = 0:1/Fs:1;         % Time vector
x = sin(2*pi*t*200);  % 200 Hz sinusoid signal
```

Now, you take the FFT of the signal. To optimize the FFT, you pad the signal with enough zeros so that its length is a power of 2. The `fft` automatically does this if you provide an input argument specifying the length of the FFT.

```
nfft = 2^(nextpow2(length(x)));   % Find next power of 2
fftx = fft(x,nfft);
```

`nfft` is even, so the magnitude of the FFT will be symmetric, in that the first (1+`nfft`/2) points are unique, and the rest are symmetrically redundant. The DC component of `x` is `fftx(1)` and the Nyquist frequency is at

`fftx(1+nfft/2)`. Note that if `nfft` is odd, the Nyquist frequency component is not evaluated, and the number of unique points is (`nfft`+1)/2 . You can use the `ceil` function to determine the number of unique points. In this example, you know the FFT is symmetric, so you keep only the unique points. Then, calculate the magnitude of the FFT.

```
NumUniquePts = ceil((nfft+1)/2);
fftx = fftx(1:NumUniquePts);
mx = abs(fftx);
```

MATLAB does not scale the output of the FFT by the length of the input, so you scale the it so that it is not a function of the length of x. Then, square the magnitude and since you dropped half of the FFT, multiply by 2 to retain the same amount of energy, but do not multiply the DC or Nyquist components by 2. You then obtain an evenly spaced frequency vector with `NumUniquePts` points.

```
mx = mx/length(x);
mx = mx.^2;
if rem(nfft, 2)   % Odd nfft excludes Nyquist
  mx(2:end) = mx(2:end)*2;
else
  mx(2:end -1) = mx(2:end -1)*2;
end

f = (0:NumUniquePts-1)*Fs/nfft;
```

Finally, plot the resulting spectrum.

```
plot(f,mx);
title('Power Spectrum of a 200Hz Sine Wave');
xlabel('Frequency (Hz)');
ylabel('Power');
```

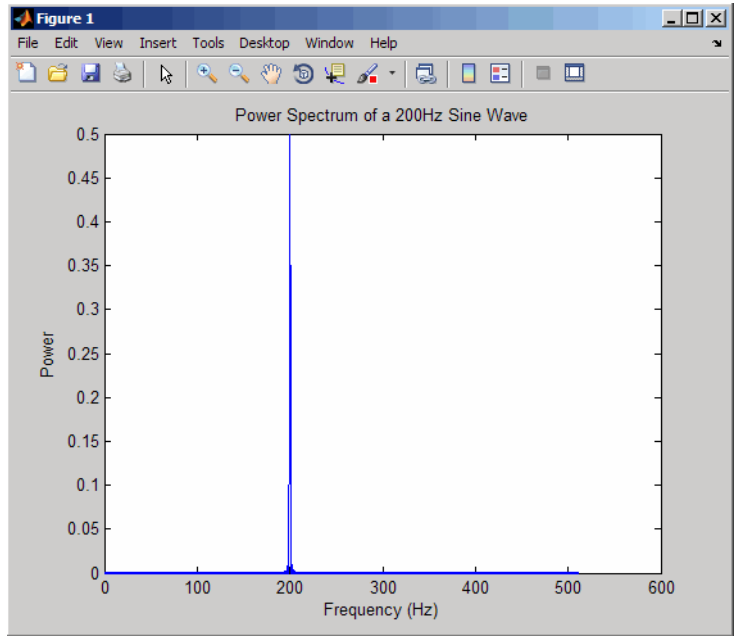


Figure 1
File Edit View Insert Tools Desktop Window Help
Power Spectrum of a 200Hz Sine Wave
Power
Frequency (Hz)
0.5
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
0
100
200
300
400
500
600

Selected Bibliography

[1] Hayes, M.H. *Statistical Digital Signal Processing and Modeling*. New York: John Wiley & Sons, 1996.

[2] Kay, S.M. *Modern Spectral Estimation*. Englewood Cliffs, NJ: Prentice Hall, 1988.

[3] Marple, S.L. *Digital Spectral Analysis*. Englewood Cliffs, NJ: Prentice Hall, 1987.

[4] Orfanidis, S.J. *Introduction to Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1996.

[5] Percival, D.B., and A.T. Walden. *Spectral Analysis for Physical Applications: Multitaper and Conventional Univariate Techniques*. Cambridge: Cambridge University Press, 1993.

[6] Proakis, J.G., and D.G. Manolakis. *Digital Signal Processing: Principles, Algorithms, and Applications*. Englewood Cliffs, NJ: Prentice Hall, 1996.

[7] Stoica, P., and R. Moses. *Introduction to Spectral Analysis*. Upper Saddle River, NJ: Prentice Hall, 1997.

[8] Welch, P.D. "The Use of Fast Fourier Transform for the Estimation of Power Spectra: A Method Based on Time Averaging Over Short, Modified Periodograms." *IEEE Trans. Audio Electroacoust*. Vol. AU-15 (June 1967). Pgs. 70-73.

7

Special Topics

Windows

In this section...

Why Use Windows?

In both digital filter design and spectral estimation, the choice of a windowing function can play an important role in determining the quality of overall results. The main role of the window is to damp out the effects of the Gibbs phenomenon that results from truncation of an infinite series.

Available Window Functions

Window	Function
Bartlett-Hann window	`barthannwin`
Bartlett window	`bartlett`
Blackman window	`blackman`
Blackman-Harris window	`blackmanharris`
Bohman window	`bohmanwin`
Chebyshev window	`chebwin`
Flat Top window	`flattopwin`
Gaussian window	`gausswin`
Hamming window	`hamming`

Window	Function
Hann window	`hann`
Kaiser window	`kaiser`
Nuttall's Blackman-Harris window	`nuttallwin`
Parzen (de la Valle-Poussin) window	`parzenwin`
Rectangular window	`rectwin`
Tapered cosine window	`tukeywin`
Triangular window	`triang`

Graphical User Interface Tools

Two graphical user interface tools are provided for working with windows in the Signal Processing Toolbox product:

- Window Design and Analysis Tool (`wintool`)
- Window Visualization Tool (`wvtool`)

Refer to the reference pages for these tools for detailed information.

Basic Shapes

The basic window is the *rectangular window*, a vector of ones of the appropriate length. A rectangular window of length 50 is

```
n = 50;
w = rectwin(n);
```

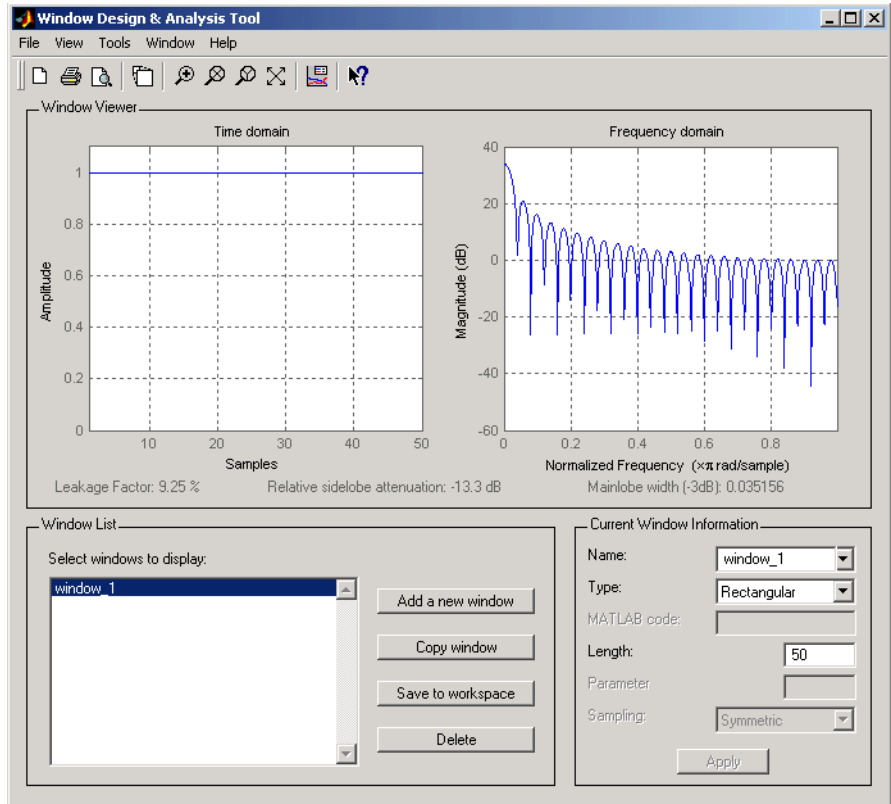
This toolbox stores windows in column vectors by convention, so an equivalent expression is

```
w = ones(50,1);
```

To use the Window Design and Analysis Tool to create this window, type

```
wintool
```

`wintool` opens with a default Hamming window. In the Current Window Information panel, set **Type = Rectangular** and **Length = 50** and then press **Apply**.



The *Bartlett* (or triangular) *window* is the convolution of two rectangular windows. The functions `bartlett` and `triang` compute similar triangular windows, with three important differences. The `bartlett` function always returns a window with two zeros on the ends of the sequence, so that for `n` odd, the center section of `bartlett(n+2)` is equivalent to `triang(n)`:

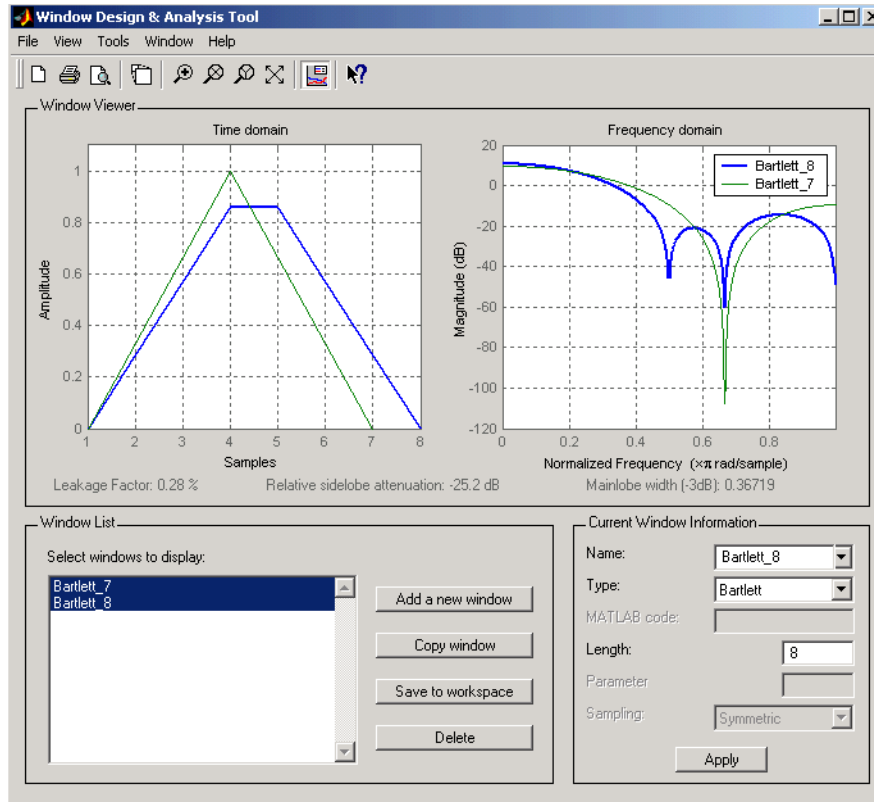
```
bartlett(7)
ans =
         0
    0.3333
```

```
    0.6667
    1.0000
    0.6667
    0.3333
         0
triang(5)
ans =
    0.3333
    0.6667
    1.0000
    0.6667
    0.3333
```

For `n` even, `bartlett` is still the convolution of two rectangular sequences. There is no standard definition for the triangular window for `n` even; the slopes of the line segments of the `triang` result are slightly steeper than those of `bartlett` in this case:

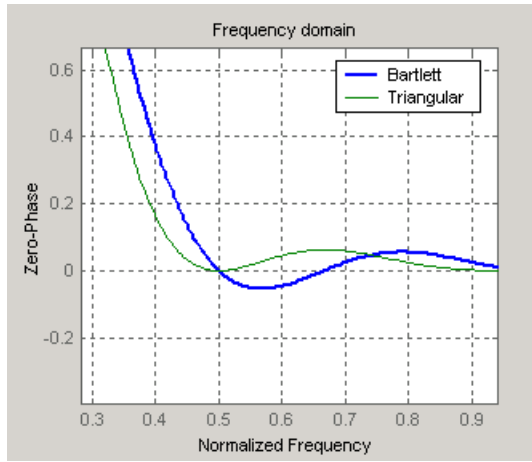
```
w = bartlett(8);
[w(2:7)  triang(6)]
ans =
    0.2857    0.1667
    0.5714    0.5000
    0.8571    0.8333
    0.8571    0.8333
    0.5714    0.5000
    0.2857    0.1667
```

You can see the difference between odd and even Bartlett windows in WinTool.



The final difference between the Bartlett and triangular windows is evident in the Fourier transforms of these functions. The Fourier transform of a Bartlett window is negative for n even. The Fourier transform of a triangular window, however, is always nonnegative.

The following figure, which is a zoomed version of the Frequency domain plot of 8-point Bartlett and Triangular windows with the *y*-axis set to Zerophase, illustrates this difference.



This difference can be important when choosing a window for some spectral estimation techniques, such as the Blackman-Tukey method. Blackman-Tukey forms the spectral estimate by calculating the Fourier transform of the autocorrelation sequence. The resulting estimate might be negative at some frequencies if the window's Fourier transform is negative (see Kay [1], pg. 80).

Generalized Cosine Windows

Blackman, Flat Top, Hamming, Hann, and rectangular windows are all special cases of the *generalized cosine window*. These windows are combinations of sinusoidal sequences with frequencies 0, $2\pi/(N-1)$, and $4\pi/(N-1)$, where N is the window length. One way to generate them is

```
ind = (0:n-1)'*2*pi/(n-1);
w = A - B*cos(ind) + C*cos(2*ind);
```

where `A`, `B`, and `C` are constants you define. The concept behind these windows is that by summing the individual terms to form the window, the low frequency peaks in the frequency domain combine in such a way as to decrease sidelobe height. This has the side effect of increasing the mainlobe width.

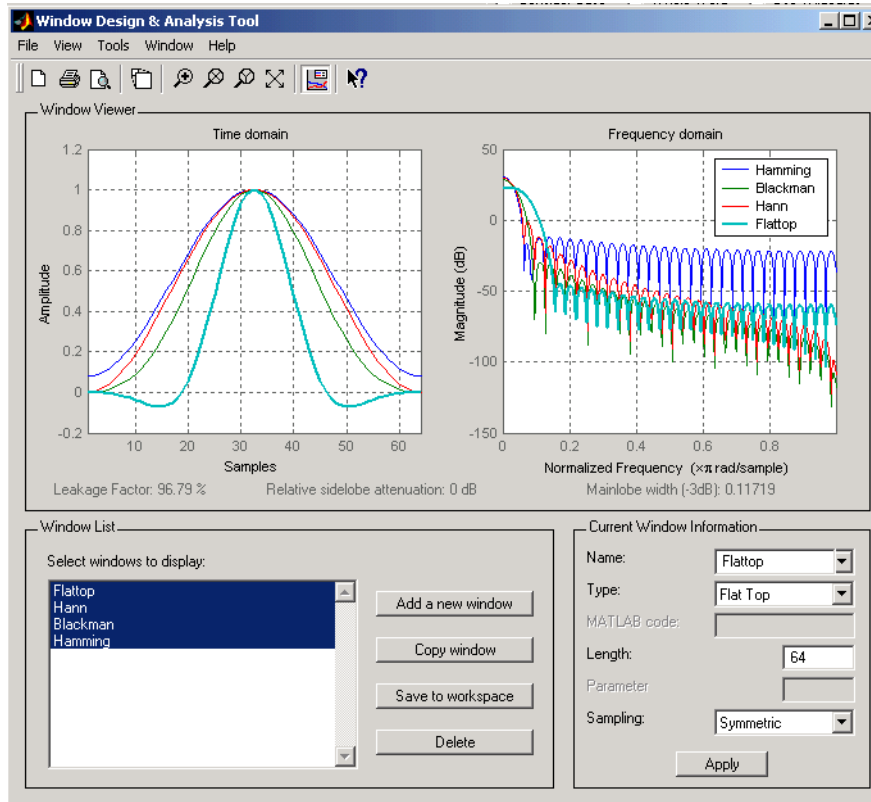
The Hamming and Hann windows are two-term generalized cosine windows, given by `A = 0.54`, `B = 0.46` for Hamming and `A = 0.5`, `B = 0.5` for Hann (`C = 0` in both cases). The `hamming` and `hann` functions, respectively, compute these windows.

Note that the definition of the generalized cosine window shown in the earlier MATLAB code yields zeros at samples 1 and `n` for `A = 0.5` and `B = 0.5`.

The Blackman window is a popular three-term window, given by `A = 0.42`, `B = 0.5`, `C = 0.08`. The `blackman` function computes this window.

The Flat Top window is a five-term window and is used for calibration. It is given by `A = 1`, `B = 1.93`, `C = 1.29`, `D =0.388`, and `E = 0.322`.

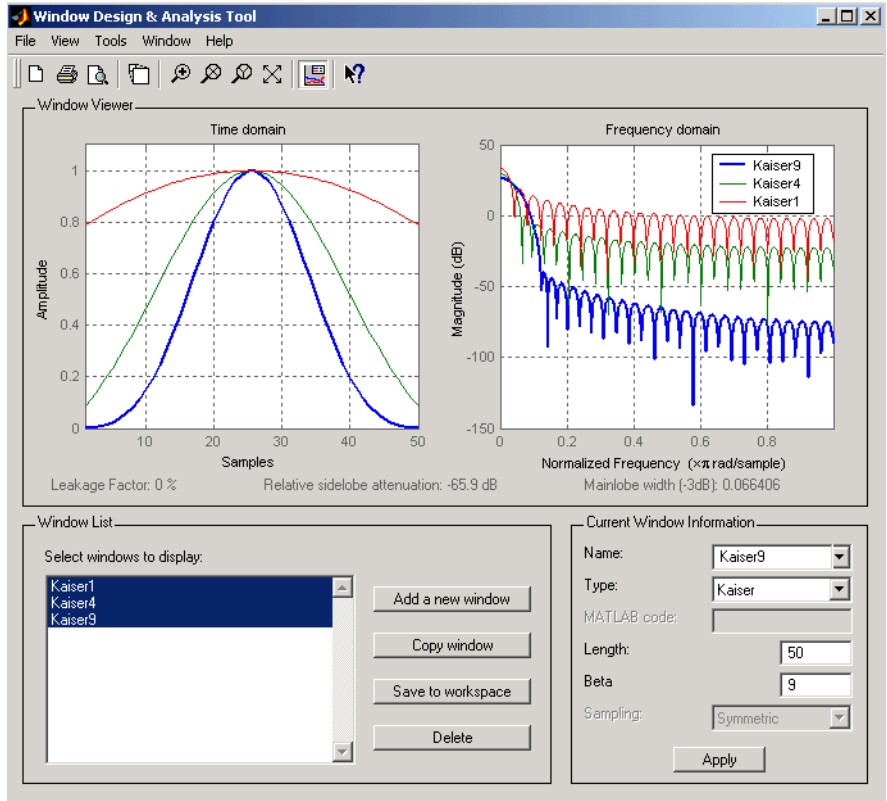
This WinTool compares Blackman, Hamming, Hann, and Flat Top windows.



Kaiser Window

The *Kaiser window* is an approximation to the prolate-spheroidal window, for which the ratio of the mainlobe energy to the sidelobe energy is maximized. For a Kaiser window of a particular length, the parameter ß controls the sidelobe height. For a given ß, the sidelobe height is fixed with respect to window length. The statement `kaiser(n,beta)` computes a length `n` Kaiser window with parameter `beta`.

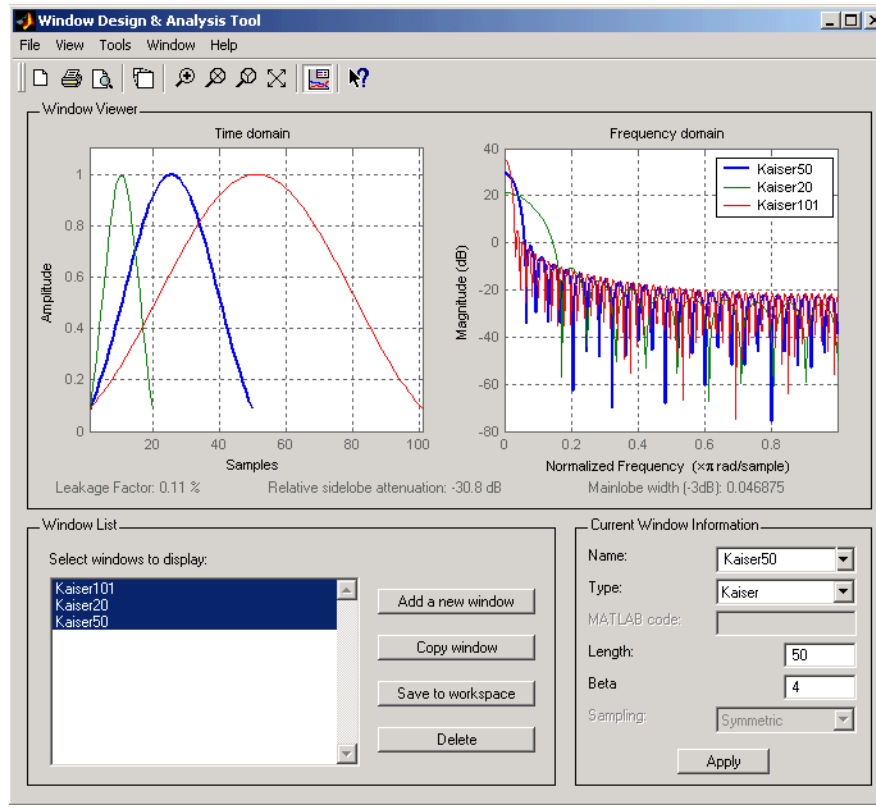
Examples of Kaiser windows with length 50 and `beta` parameters of 1, 4, and 9 are shown in this `wintool` example.



To create these Kaiser windows using the MATLAB command line,

```
n = 50;
w1 = kaiser(n,1);
w2 = kaiser(n,4);
w3 = kaiser(n,9);
[W1,f] = freqz(w1/sum(w1),1,512,2);
[W2,f] = freqz(w2/sum(w2),1,512,2);
[W3,f] = freqz(w3/sum(w3),1,512,2);
plot(f,20*log10(abs([W1 W2 W3]))); grid;
legend('beta = 1','beta = 4','beta = 9',3)
```

As ß increases, the sidelobe height decreases and the mainlobe width increases. This wintool shows how the sidelobe height stays the same for a fixed ß parameter as the length is varied.



To create these Kaiser windows using the MATLAB command line:

```
w1 = kaiser(50,4);
w2 = kaiser(20,4);
w3 = kaiser(101,4);
[W1,f] = freqz(w1/sum(w1),1,512,2);
[W2,f] = freqz(w2/sum(w2),1,512,2);
[W3,f] = freqz(w3/sum(w3),1,512,2);
plot(f,20*log10(abs([W1 W2 W3]))); grid;
```

```
legend('length = 50','length = 20','length = 101')
```

Kaiser Windows in FIR Design

There are two design formulas that can help you design FIR filters to meet a set of filter specifications using a Kaiser window. To achieve a sidelobe height of -α dB, the `beta` parameter is

$$\beta = \begin{cases} 0.1102(\alpha - 8.7), & \alpha > 50 \\ 0.5842(\alpha - 21)^{0.4} + 0.07886(\alpha - 21), & 50 \geq \alpha \geq 21 \\ 0, & \alpha < 21 \end{cases}$$

For a transition width of $\Delta\omega$ rad/s, use the length

$$n = \frac{\alpha - 8}{2.285\Delta\omega} + 1$$

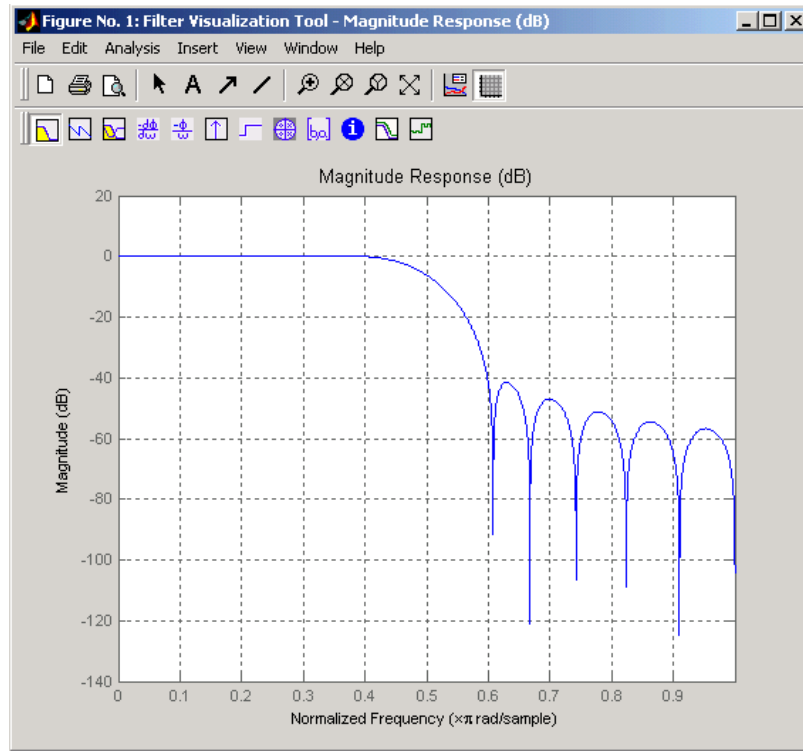
Filters designed using these heuristics will meet the specifications approximately, but you should verify this. To design a lowpass filter with cutoff frequency 0.5π rad/s, transition width 0.2π rad/s, and 40 dB of attenuation in the stopband, try

```
[n,wn,beta] = kaiserord([0.4 0.6]*pi,[1 0],[0.01 0.01],2*pi);
h = fir1(n,wn,kaiser(n+1,beta),'noscale');
```

The `kaiserord` function estimates the filter order, cutoff frequency, and Kaiser window beta parameter needed to meet a given set of frequency domain specifications.

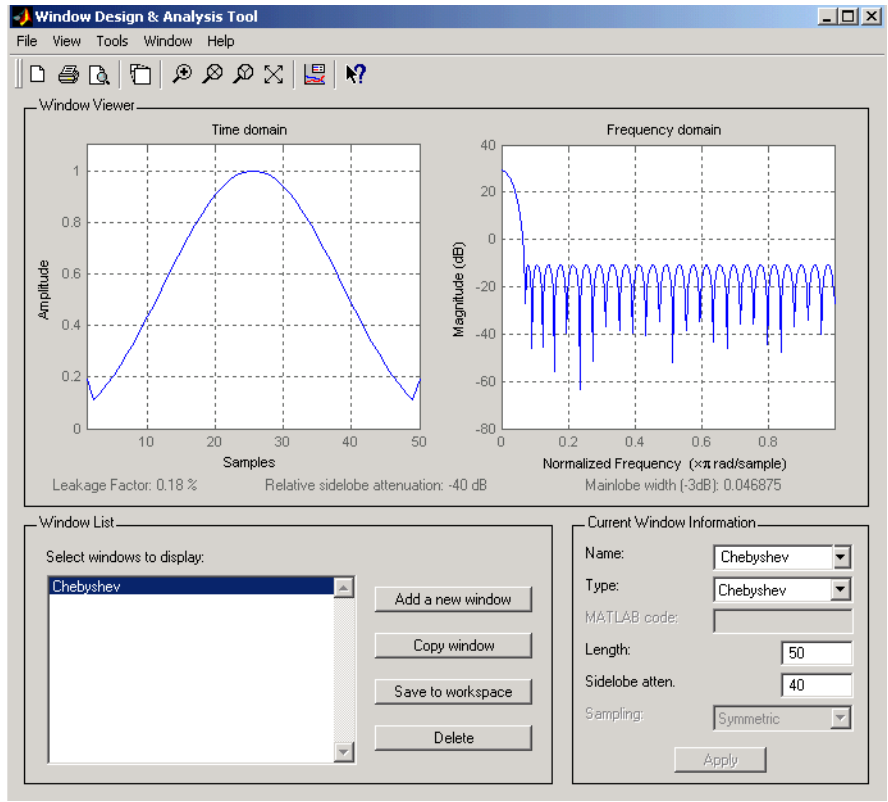
The ripple in the passband is roughly the same as the ripple in the stopband. As you can see from the frequency response, this filter nearly meets the specifications:

```
fvtool(h,1);
```



Chebyshev Window

The Chebyshev window minimizes the mainlobe width, given a particular sidelobe height. It is characterized by an equiripple behavior, that is, its sidelobes all have the same height.



As shown in the Time Domain plot, the Chebyshev window has large spikes at its outer samples.

For a detailed discussion of the characteristics and applications of the various window types, see Oppenheim and Schafer [3], pgs. 444-462, and Parks and Burrus [4], pgs. 71-73.

Parametric Modeling

In this section...

What is Parametric Modeling

Parametric modeling techniques find the parameters for a mathematical model describing a signal, system, or process. These techniques use known information about the system to determine the model. Applications for parametric modeling include speech and music synthesis, data compression, high-resolution spectral estimation, communications, manufacturing, and simulation.

Note Since `yulewalk` is geared explicitly toward ARMA filter design, it is discussed in Chapter 2, "Filter Design and Implementation".

`pburg` and `pyulear` are discussed in Chapter 6, "Statistical Signal Processing", with the other (nonparametric) spectral estimation methods.

Available Parametric Modeling Functions

The toolbox parametric modeling functions operate with the rational transfer function model. Given appropriate information about an unknown system (impulse or frequency response data, or input and output sequences), these functions find the coefficients of a linear system that models the system.

One important application of the parametric modeling functions is in the design of filters that have a prescribed time or frequency response. These functions provide a data-oriented alternative to the IIR and FIR filter design functions discussed in Chapter 2, "Filter Design and Implementation".

Here is a summary of the parametric modeling functions in this toolbox. Note that "System Identification Toolbox" software provides a more extensive collection of parametric modeling functions.

Domain	Functions	Description
Time	`arburg`	Generate all-pole filter coefficients that model an input data sequence using the Levinson-Durbin algorithm.
	`arcov`	Generate all-pole filter coefficients that model an input data sequence by minimizing the forward prediction error.
	`armcov`	Generate all-pole filter coefficients that model an input data sequence by minimizing the forward and backward prediction errors.
	`aryule`	Generate all-pole filter coefficients that model an input data sequence using an estimate of the autocorrelation function.
	`lpc`, `levinson`	Linear Predictive Coding. Generate all-pole recursive filter whose impulse response matches a given sequence.
	`prony`	Generate IIR filter whose impulse response matches a given sequence.
	`stmcb`	Find IIR filter whose output, given a specified input sequence, matches a given output sequence.
Frequency	`invfreqz`, `invfreqs`	Generate digital or analog filter coefficients given complex frequency response data.

Time-Domain Based Modeling

The `lpc`, `prony`, and `stmcb` functions find the coefficients of a digital rational transfer function that approximates a given time-domain impulse response. The algorithms differ in complexity and accuracy of the resulting model.

Linear Prediction

Linear prediction modeling assumes that each output sample of a signal, `x(k)`, is a linear combination of the past `n` outputs (that is, it can be linearly predicted from these outputs), and that the coefficients are constant from sample to sample:

$$x(k) = -a(2)x(k-1) - a(3)x(k-2) - \ldots - a(n+1)x(k-n)$$

An `n`th-order all-pole model of a signal `x` is

```
a = lpc(x,n)
```

To illustrate `lpc`, create a sample signal that is the impulse response of an all-pole filter with additive white noise:

```
randn('state',0);
x = impz(1,[1 0.1 0.1 0.1 0.1],10) + randn(10,1)/10;
```

The coefficients for a fourth-order all-pole filter that models the system are

```
a = lpc(x,4)
a =
    1.0000    0.2574    0.1666    0.1203    0.2598
```

`lpc` first calls `xcorr` to find a biased estimate of the correlation function of `x`, and then uses the Levinson-Durbin recursion, implemented in the `levinson` function, to find the model coefficients `a`. The Levinson-Durbin recursion is a fast algorithm for solving a system of symmetric Toeplitz linear equations. `lpc`'s entire algorithm for `n = 4` is

```
r = xcorr(x);
r(1:length(x)-1) = [];      % Remove corr. at negative lags
a = levinson(r,4)
a =
    1.0000    0.2574    0.1666    0.1203    0.2598
```

You could form the linear prediction coefficients with other assumptions by passing a different correlation estimate to `levinson`, such as the biased correlation estimate:

```
r = xcorr(x,'biased');
```

```
r(1:length(x)-1) = [];      % Remove corr. at negative lags
a = levinson(r,4)
a =
    1.0000    0.2574    0.1666    0.1203    0.2598
```

Prony's Method (ARMA Modeling)

The `prony` function models a signal using a specified number of poles and zeros. Given a sequence `x` and numerator and denominator orders `n` and `m`, respectively, the statement

```
[b,a] = prony(x,n,m)
```

finds the numerator and denominator coefficients of an IIR filter whose impulse response approximates the sequence `x`.

The `prony` function implements the method described in [4] Parks and Burrus (pgs. 226-228). This method uses a variation of the covariance method of AR modeling to find the denominator coefficients `a`, and then finds the numerator coefficients `b` for which the resulting filter's impulse response matches exactly the first `n + 1` samples of `x`. The filter is not necessarily stable, but it can potentially recover the coefficients exactly if the data sequence is truly an autoregressive moving-average (ARMA) process of the correct order.

Note The functions `prony` and `stmcb` (described next) are more accurately described as ARX models in system identification terminology. ARMA modeling assumes noise only at the inputs, while ARX assumes an external input. `prony` and `stmcb` know the input signal: it is an impulse for `prony` and is arbitrary for `stmcb`.

A model for the test sequence `x` (from the earlier `lpc` example) using a third-order IIR filter is

```
[b,a] = prony(x,3,3)
b =
    0.9567   -0.3351    0.1866   -0.3782
a =
    1.0000   -0.0716    0.2560   -0.2752
```

The `impz` command shows how well this filter's impulse response matches the original sequence:

```
format long
[x impz(b,a,10)]

ans =
   0.95674351884718   0.95674351884718
  -0.26655843782381  -0.26655843782381
  -0.07746676935252  -0.07746676935252
  -0.05223235796415  -0.05223235796415
  -0.18754713506815  -0.05726777015121
   0.15348154656430  -0.01204969926150
   0.13986742016521  -0.00057632797226
   0.00609257234067  -0.01271681570687
   0.03349954614087  -0.00407967053863
   0.01086719328209   0.00280486049427
```

Notice that the first four samples match exactly. For an example of exact recovery, recover the coefficients of a Butterworth filter from its impulse response:

```
[b,a] = butter(4,.2);
h = impz(b,a,26);
[bb,aa] = prony(h,4,4);
```

Try this example; you'll see that `bb` and `aa` match the original filter coefficients to within a tolerance of 10^{-13}.

Steiglitz-McBride Method (ARMA Modeling)

The `stmcb` function determines the coefficients for the system $b(z)/a(z)$ given an approximate impulse response `x`, as well as the desired number of zeros and poles. This function identifies an unknown system based on both input and output sequences that describe the system's behavior, or just the impulse response of the system. In its default mode, `stmcb` works like `prony`.

```
[b,a] = stmcb(x,3,3)
b =
    0.9567   -0.5181    0.5702   -0.5471
```

```
a =
    1.0000   -0.2384    0.5234   -0.3065
```

`stmcb` also finds systems that match given input and output sequences:

```
y = filter(1,[1 1],x);     % Create an output signal.
[b,a] = stmcb(y,x,0,1)
b =
     1.0000
a =
     1     1
```

In this example, `stmcb` correctly identifies the system used to create `y` from `x`.

The Steiglitz-McBride method is a fast iterative algorithm that solves for the numerator and denominator coefficients simultaneously in an attempt to minimize the signal error between the filter output and the given output signal. This algorithm usually converges rapidly, but might not converge if the model order is too large. As for `prony`, `stmcb`'s resulting filter is not necessarily stable due to its exact modeling approach.

`stmcb` provides control over several important algorithmic parameters; modify these parameters if you are having trouble modeling the data. To change the number of iterations from the default of five and provide an initial estimate for the denominator coefficients:

```
n = 10;             % Number of iterations
a = lpc(x,3);       % Initial estimates for denominator
[b,a] = stmcb(x,3,3,n,a);
```

The function uses an all-pole model created with `prony` as an initial estimate when you do not provide one of your own.

To compare the functions `lpc`, `prony`, and `stmcb`, compute the signal error in each case:

```
a1 = lpc(x,3);
[b2,a2] = prony(x,3,3);
[b3,a3] = stmcb(x,3,3);
[x-impz(1,a1,10)  x-impz(b2,a2,10)  x-impz(b3,a3,10)]
```

```
ans =
   -0.0433         0   -0.0000
   -0.0240         0    0.0234
   -0.0040         0   -0.0778
   -0.0448   -0.0000    0.0498
   -0.2130   -0.1303   -0.0742
    0.1545    0.1655    0.1270
    0.1426    0.1404    0.1055
    0.0068    0.0188    0.0465
    0.0329    0.0376    0.0530
    0.0108    0.0081   -0.0162
sum(ans.^2)
ans =
    0.0953    0.0659    0.0471
```

In comparing modeling capabilities for a given order IIR model, the last result shows that for this example, `stmcb` performs best, followed by `prony`, then `lpc`. This relative performance is typical of the modeling functions.

Frequency-Domain Based Modeling

The `invfreqs` and `invfreqz` functions implement the inverse operations of `freqs` and `freqz`; they find an analog or digital transfer function of a specified order that matches a given complex frequency response. Though the following examples demonstrate `invfreqz`, the discussion also applies to `invfreqs`.

To recover the original filter coefficients from the frequency response of a simple digital filter:

```
[b,a] = butter(4,0.4)         % Design Butterworth lowpass
b =
    0.0466    0.1863    0.2795    0.1863    0.0466
a =
    1.0000   -0.7821    0.6800   -0.1827    0.0301

[h,w] = freqz(b,a,64);        % Compute frequency response
[b4,a4] = invfreqz(h,w,4,4)   % Model: n = 4, m = 4
b4 =
    0.0466    0.1863    0.2795    0.1863    0.0466
```

```
a4 =
    1.0000   -0.7821    0.6800   -0.1827    0.0301
```

The vector of frequencies `w` has the units in rad/sample, and the frequencies need not be equally spaced. `invfreqz` finds a filter of any order to fit the frequency data; a third-order example is

```
[b4,a4] = invfreqz(h,w,3,3)   % Find third-order IIR
b4 =
    0.0464    0.1785    0.2446    0.1276
a4 =
    1.0000   -0.9502    0.7382   -0.2006
```

Both `invfreqs` and `invfreqz` design filters with real coefficients; for a data point at positive frequency `f`, the functions fit the frequency response at both `f` and `-f`.

By default `invfreqz` uses an equation error method to identify the best model from the data. This finds b and a in

$$\min_{b,a} \sum_{k=1}^{n} wt(k)\left|h(k)A(w(k)) - B(w(k))\right|^2$$

by creating a system of linear equations and solving them with the MATLAB `\` operator. Here $A(w(k))$ and $B(w(k))$ are the Fourier transforms of the polynomials `a` and `b` respectively at the frequency $w(k)$, and n is the number of frequency points (the length of `h` and `w`). $wt(k)$ weights the error relative to the error at different frequencies. The syntax

```
invfreqz(h,w,n,m,wt)
```

includes a weighting vector. In this mode, the filter resulting from `invfreqz` is not guaranteed to be stable.

`invfreqz` provides a superior ("output-error") algorithm that solves the direct problem of minimizing the weighted sum of the squared error between the actual frequency response points and the desired response

$$\min_{b,a} \sum_{k=1}^{n} wt(k)\left|h(k) - \frac{B(w(k))}{A(w(k))}\right|^2$$

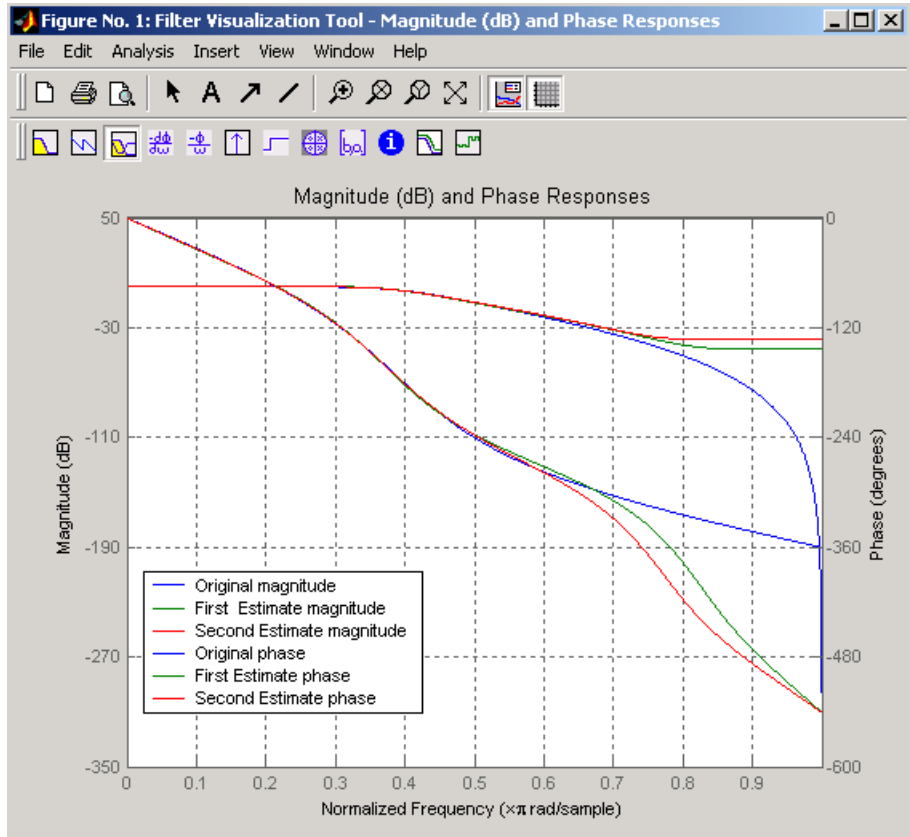
To use this algorithm, specify a parameter for the iteration count after the weight vector parameter:

```
wt = ones(size(w));   % Create unity weighting vector
[b30,a30] = invfreqz(h,w,3,3,wt,30)  % 30 iterations
b30 =
    0.0464    0.1829    0.2572    0.1549
a30 =
    1.0000   -0.8664    0.6630   -0.1614
```

The resulting filter is always stable.

Graphically compare the results of the first and second algorithms to the original Butterworth filter with FVTool (and select the Magnitude and Phase Responses):

```
fvtool(b,a,b4,a4,b30,a30)
```



To verify the superiority of the fit numerically, type

```
sum(abs(h-freqz(b4,a4,w)).^2)    % Total error, algorithm 1
ans =
    0.0200
sum(abs(h-freqz(b30,a30,w)).^2)  % Total error, algorithm 2
ans =
    0.0096
```

Resampling

In this section...

Available Resampling Functions

The toolbox provides a number of functions that resample a signal at a higher or lower rate.

Operation	Function
Apply FIR filter with resampling	`upfirdn`
Cubic spline interpolation	`spline`
Decimation	`decimate`
Interpolation	`interp`
Other 1-D interpolation	`interp1`
Resample at new rate	`resample`

resample Function

The `resample` function changes the sampling rate for a sequence to any rate that is a ratio of two integers. The basic syntax for `resample` is

```
y = resample(x,p,q)
```

where the function resamples the sequence `x` at `p/q` times the original sampling rate. The length of the result `y` is `p/q` times the length of `x`.

One resampling application is the conversion of digitized audio signals from one sampling rate to another, such as from 48 kHz (the digital audio tape standard) to 44.1 kHz (the compact disc standard).

The example file contains a length 4001 vector of speech sampled at 7418 Hz:

```
clear
load mtlb
whos

Name         Size         Bytes        Class
  Fs            1x1           8        double array
  mtlb       4001x1       32008        double array

Grand total is 4002 elements using 32016 bytes
Fs
Fs =
        7418
```

To play this speech signal on a workstation that can only play sound at 8192 Hz, use the `rat` function to find integers `p` and `q` that yield the correct resampling factor:

```
[p,q] = rat(8192/Fs,0.0001)
p =
   127
q =
   115
```

Since `p/q*Fs = 8192.05` Hz, the tolerance of 0.0001 is acceptable; to resample the signal at very close to 8192 Hz:

```
y = resample(mtlb,p,q);
```

`resample` applies a lowpass filter to the input sequence to prevent aliasing during resampling. It designs this filter using the `firls` function with a Kaiser window. The syntax

```
resample(x,p,q,l,beta)
```

controls the filter's length and the beta parameter of the Kaiser window. Alternatively, use the function `intfilt` to design an interpolation filter `b` and use it with

```
resample(x,p,q,b)
```

decimate and interp Functions

The `decimate` and `interp` functions do the same thing as `resample` with `p = 1` and `q = 1`, respectively. These functions provide different anti-alias filtering options, and they incur a slight signal delay due to filtering. The `interp` function is significantly less efficient than the `resample` function with `q = 1`.

upfirdn Function

The toolbox also contains a function, `upfirdn`, that applies an FIR filter to an input sequence and outputs the filtered sequence at a sample rate different than its original. See "Multirate Filter Bank Implementation" on page 1-7.

spline Function

The standard MATLAB environment contains a function, `spline`, that works with irregularly spaced data. The MATLAB function `interp1` performs interpolation, or table lookup, using various methods including linear and cubic interpolation.

Cepstrum Analysis

In this section...

What Is a Cepstrum?

Cepstrum analysis is a nonlinear signal processing technique with a variety of applications in areas such as speech and image processing.

The complex cepstrum for a sequence x is calculated by finding the complex natural logarithm of the Fourier transform of x, then the inverse Fourier transform of the resulting sequence.

$$\hat{x} = \frac{1}{2\pi}\int_{-\pi}^{\pi} \log[X(e^{j\omega})]e^{j\omega n}d\omega$$

The toolbox function `cceps` performs this operation, estimating the complex cepstrum for an input sequence. It returns a real sequence the same size as the input sequence:

```
xhat = cceps(x)
```

For sequences that have roots on the unit circle, cepstrum analysis has numerical problems. See Oppenheim and Schafer [2] for information.

The complex cepstrum transformation is central to the theory and application of *homomorphic systems*, that is, systems that obey certain general rules of superposition. See Oppenheim and Schafer [3] for a discussion of the complex cepstrum and homomorphic transformations, with details on speech processing applications.

Try using `cceps` in an echo detection application. First, create a 45 Hz sine wave sampled at 100 Hz:

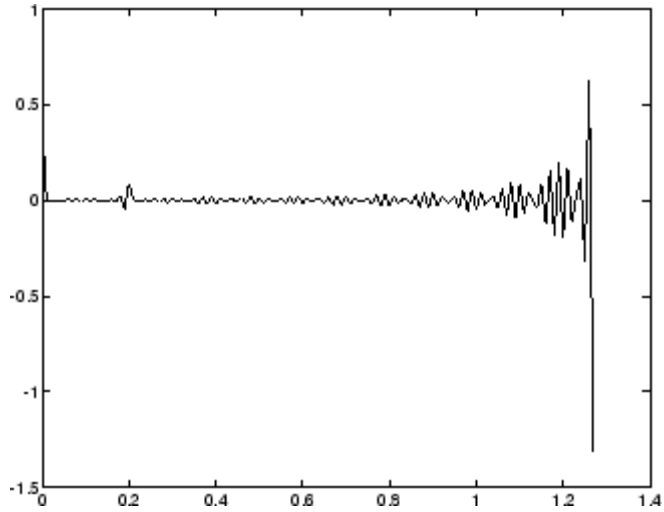
```
t = 0:0.01:1.27;
s1 = sin(2*pi*45*t);
```

Add an echo of the signal, with half the amplitude, 0.2 seconds after the beginning of the signal.

```
s2 = s1 + 0.5*[zeros(1,20) s1(1:108)];
```

The complex cepstrum of this new signal is

```
c = cceps(s2);
plot(t,c)
```



Note that the complex cepstrum shows a peak at 0.2 seconds, indicating the echo.

The *real cepstrum* of a signal *x*, sometimes called simply the cepstrum, is calculated by determining the natural logarithm of magnitude of the Fourier transform of *x*, then obtaining the inverse Fourier transform of the resulting sequence.

$$c_x = \frac{1}{2\pi}\int_{-\pi}^{\pi} \log\left|X(e^{j\omega})\right| e^{j\omega n} d\omega$$

The toolbox function `rceps` performs this operation, returning the real cepstrum for a sequence `x`. The returned sequence is a real-valued vector the same size as the input vector:

```
y = rceps(x)
```

By definition, you cannot reconstruct the original sequence from its real cepstrum transformation, as the real cepstrum is based only on the magnitude of the Fourier transform for the sequence (see Oppenheim and Schafer [3]). The `rceps` function also returns a unique minimum-phase sequence that has the same real cepstrum as `x`. To obtain both the real cepstrum and the minimum phase reconstruction for a sequence, use

```
[y,ym] = rceps(x)
```

where `y` is the real cepstrum and `ym` is the minimum phase reconstruction of `x`. The following example shows that one output of rceps is a unique minimum-phase sequence with the same real cepstrum as `x`.

```
y = [4 1 5];                % Non-minimum phase sequence
[xhat,yhat] = rceps(y);
xhat2= rceps(yhat);
[xhat' xhat2']

ans =
    1.6225    1.6225
    0.3400    0.3400
    0.3400    0.3400
```

Summary of Cepstrum Functions

The Signal Processing Toolbox product provides three functions for cepstrum analysis:

Operation	Function
Complex cepstrum	`cceps`
Inverse complex cepstrum	`icceps`
Real cepstrum	`rceps`

Inverse Complex Cepstrum

To invert the complex cepstrum, use the `icceps` function. Inversion is complicated by the fact that the `cceps` function performs a data dependent phase modification so that the unwrapped phase of its input is continuous at zero frequency. The phase modification is equivalent to an integer delay. This delay term is returned by `cceps` if you ask for a second output. For example:

```
x = 1:10;
[xhat,delay] = cceps(x)
xhat =
  Columns 1 through 4
    2.2428   -0.0420   -0.0210    0.0045
  Columns 5 through 8
    0.0366    0.0788    0.1386    0.2327
  Columns 9 through 10
    0.4114    0.9249
delay =
     1
```

To invert the complex cepstrum, use `icceps` with the original delay parameter:

```
icc = icceps(xhat,2)
ans =
  Columns 1 through 4
    2.0000    3.0000    4.0000    5.0000
  Columns 5 through 8
    6.0000    7.0000    8.0000    9.0000
  Columns 9 through 10
    10.0000   1.0000
```

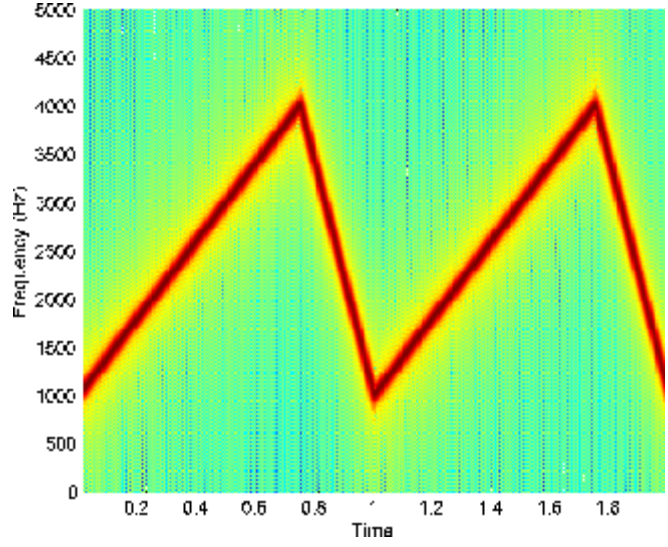
As shown in the above example, with any modification of the complex cepstrum, the original delay term may no longer be valid. You will not be able to invert the complex cepstrum exactly.

FFT-Based Time-Frequency Analysis

The Signal Processing Toolbox product provides a function, `spectrogram`, that returns the time-dependent Fourier transform for a sequence, or displays this information as a spectrogram. The toolbox also includes a spectrogram demo. The *time-dependent Fourier transform* is the discrete-time Fourier transform for a sequence, computed using a sliding window. This form of the Fourier transform, also known as the short-time Fourier transform (STFT), has numerous applications in speech, sonar, and radar processing. The *spectrogram* of a sequence is the magnitude of the time-dependent Fourier transform versus time.

To display the spectrogram of a linear FM signal:

```
fs = 10000;
t = 0:1/fs:2;
x = vco(sawtooth(2*pi*t,.75),[0.1 0.4]*fs,fs);
spectrogram(x,kaiser(256,5),220,512,fs,'yaxis')
```



Median Filtering

The function `medfilt1` implements one-dimensional median filtering, a nonlinear technique that applies a sliding window to a sequence. The median filter replaces the center value in the window with the median value of all the points within the window [5]. In computing this median, `medfilt1` assumes zeros beyond the input points.

When the number of elements `n` in the window is even, `medfilt1` sorts the numbers, then takes the average of the `n/2` and `n/2 + 1` elements.

Two simple examples with fourth- and third-order median filters are

```
medfilt1([4 3 5 2 8 9 1],4)
ans =
    1.500 3.500 3.500 4.000 6.500 5.000 4.500

medfilt1([4 3 5 2 8 9 1],3)
ans =
     3     4     3     5     8     8     1
```

See the `medfilt2` function in the Image Processing Toolbox™ documentation for information on two-dimensional median filtering.

Communications Applications

In this section...

Modulation

Modulation varies the amplitude, phase, or frequency of a *carrier signal* with reference to a *message signal*. The `modulate` function modulates a message signal with a specified modulation method.

The basic syntax for the `modulate` function is

```
y = modulate(x,fc,fs,'method',opt)
```

where:

- `x` is the message signal.
- `fc` is the carrier frequency.
- `fs` is the sampling frequency.
- *`method`* is a flag for the desired modulation method.
- `opt` is any additional argument that the method requires. (Not all modulation methods require an option argument.)

The table below summarizes the modulation methods provided; see the documentation for `modulate`, `demod`, and `vco` for complete details on each.

Method	Description
`amdsb-sc` or `am`	Amplitude modulation, double sideband, suppressed carrier
`amdsb-tc`	Amplitude modulation, double sideband, transmitted carrier

Method	Description
amssb	Amplitude modulation, single sideband
fm	Frequency modulation
pm	Phase modulation
ppm	Pulse position modulation
pwm	Pulse width modulation
qam	Quadrature amplitude modulation

If the input x is an array rather than a vector, modulate modulates each column of the array.

To obtain the time vector that modulate uses to compute the modulated signal, specify a second output parameter:

```
[y,t] = modulate(x,fc,fs,'method',opt)
```

Demodulation

The demod function performs *demodulation*, that is, it obtains the original message signal from the modulated signal:

The syntax for demod is

```
x = demod(y,fc,fs,'method',opt)
```

demod uses any of the methods shown for modulate, but the syntax for quadrature amplitude demodulation requires two output parameters:

```
[X1,X2] = demod(y,fc,fs,'qam')
```

If the input y is an array, demod demodulates all columns.

Try modulating and demodulating a signal. A 50 Hz sine wave sampled at 1000 Hz is

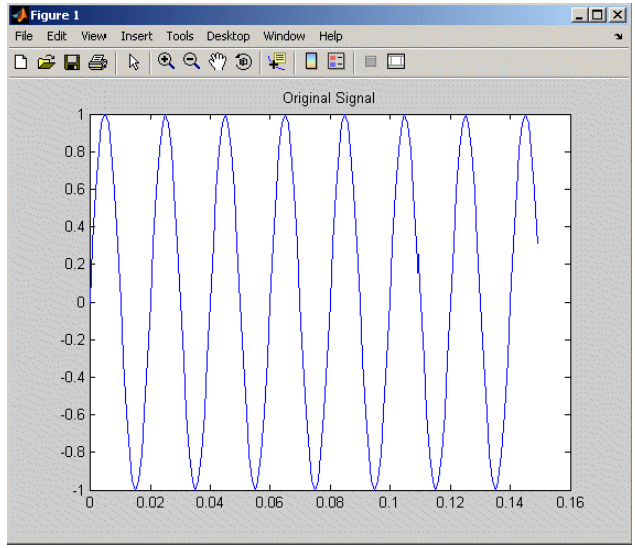
```
t = (0:1/1000:2);
x = sin(2*pi*50*t);
```

With a carrier frequency of 200 Hz, the modulated and demodulated versions of this signal are

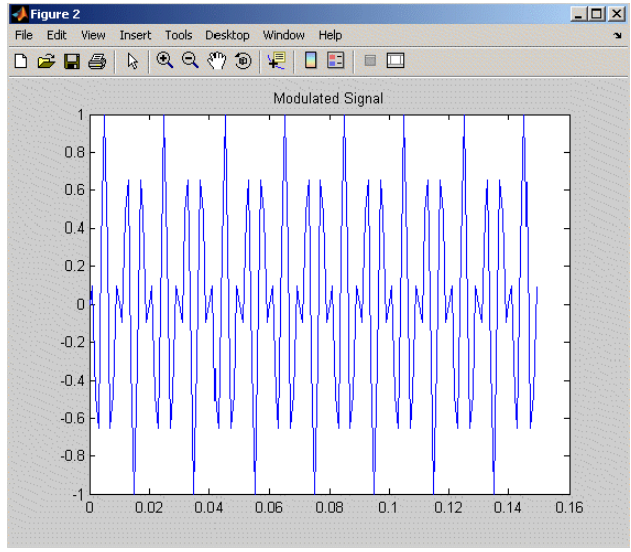
```
y = modulate(x,200,1000,'am');
z = demod(y,200,1000,'am');
```

To plot portions of the original, modulated, and demodulated signal:

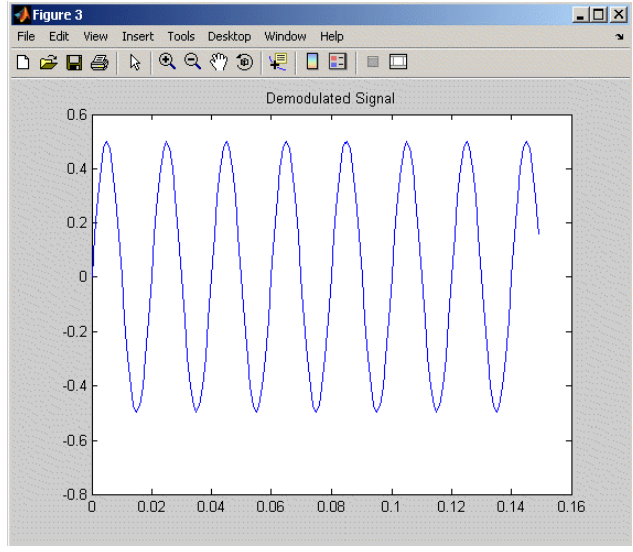
```
figure; plot(t(1:150),x(1:150)); title('Original Signal');
figure; plot(t(1:150),y(1:150)); title('Modulated Signal');
figure; plot(t(1:150),z(1:150)); title('Demodulated Signal');
```



Original Signal



Modulated Signal



Demodulated Signal

Note The demodulated signal is attenuated because demodulation includes two steps: multiplication and lowpass filtering. The multiplication produces a component with frequency centered at 0 Hz and a component with frequency at twice the carrier frequency. The filtering removes the higher frequency component of the signal, producing the attenuated result.

Voltage Controlled Oscillator

The voltage controlled oscillator function `vco` creates a signal that oscillates at a frequency determined by the input vector. The basic syntax for `vco` is

```
y = vco(x,fc,fs)
```

where `fc` is the carrier frequency and `fs` is the sampling frequency.

To scale the frequency modulation range, use

```
y = vco(x,[Fmin Fmax],fs)
```

In this case, `vco` scales the frequency modulation range so values of `x` on the interval `[-1 1]` map to oscillations of frequency on `[Fmin Fmax]`.

If the input `x` is an array, `vco` produces an array whose columns oscillate according to the columns of `x`.

See “FFT-Based Time-Frequency Analysis” on page 7-32 for an example using the `vco` function.

Deconvolution

Deconvolution, or polynomial division, is the inverse operation of convolution. Deconvolution is useful in recovering the input to a known filter, given the filtered output. This method is very sensitive to noise in the coefficients, however, so use caution in applying it.

The syntax for `deconv` is

```
[q,r] = deconv(b,a)
```

where `b` is the polynomial dividend, `a` is the divisor, `q` is the quotient, and `r` is the remainder.

To try `deconv`, first convolve two simple vectors `a` and `b` (see Chapter 1, "Filtering, Linear Systems and Transforms Overview" for a description of the convolution function):

```
a = [1 2 3];
b = [4 5 6];
c = conv(a,b)
c =
    4   13   28   27   18
```

Now use `deconv` to deconvolve `b` from `c`:

```
[q,r] = deconv(c,a)
q =
    4    5    6
r =
    0    0    0    0    0
```

Specialized Transforms

In this section...

Chirp z-Transform

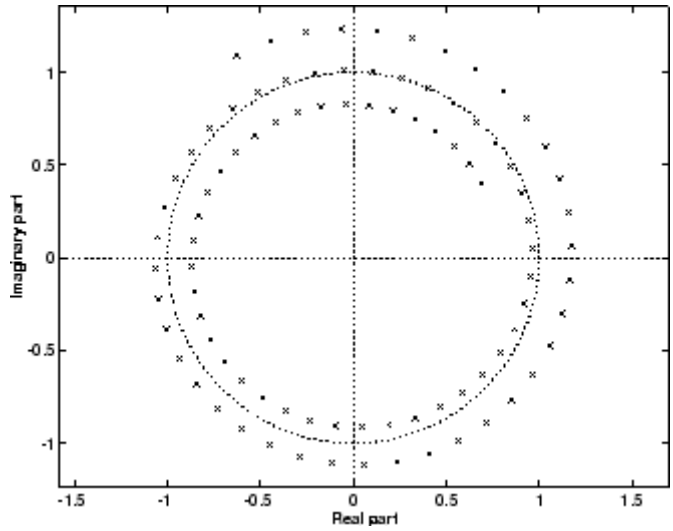
The chirp *z*-transform (CZT), useful in evaluating the *z*-transform along contours other than the unit circle. The chirp *z*-transform is also more efficient than the DFT algorithm for the computation of prime-length transforms, and it is useful in computing a subset of the DFT for a sequence. The chirp *z*-transform, or CZT, computes the *z*-transform along spiral contours in the *z*-plane for an input sequence. Unlike the DFT, the CZT is not constrained to operate along the unit circle, but can evaluate the *z*-transform along contours described by

$$z_l = AW^{-l},\ l = 0, \ldots, M-1$$

where A is the complex starting point, W is a complex scalar describing the complex ratio between points on the contour, and M is the length of the transform.

One possible spiral is

```
A = 0.8*exp(j*pi/6);
W = 0.995*exp(-j*pi*.05);
M = 91;
z = A*(W.^(-(0:M-1)));
zplane([],z.')
```



`czt(x,M,W,A)` computes the *z*-transform of `x` on these points.

An interesting and useful spiral set is `m` evenly spaced samples around the unit circle, parameterized by `A = 1` and `W = exp(-j*pi/M)`. The *z*-transform on this contour is simply the DFT, obtained by

```
y = czt(x)
```

`czt` may be faster than the `fft` function for computing the DFT of sequences with certain odd lengths, particularly long prime-length sequences.

Discrete Cosine Transform

The discrete cosine transform (DCT), closely related to the DFT. The DCT's energy compaction properties are useful for applications like signal coding. The toolbox function `dct` computes the unitary discrete cosine transform, or DCT, for an input vector or matrix. Mathematically, the unitary DCT of an input sequence x is

$$y(k) = w(k) \sum_{n=1}^{N} x(n) \cos \frac{\pi(2n-1)(k-1)}{2N}, \qquad k = 1, \ldots, N$$

where

$$w(k) = \begin{cases} \frac{1}{\sqrt{N}}, & k = 1 \\ \sqrt{\frac{2}{N}}, & 2 \le k \le N \end{cases}$$

The DCT is closely related to the discrete Fourier transform; the DFT is actually one step in the computation of the DCT for a sequence. The DCT, however, has better *energy compaction* properties, with just a few of the transform coefficients representing the majority of the energy in the sequence. The energy compaction properties of the DCT make it useful in applications such as data communications.

The function `idct` computes the inverse DCT for an input sequence, reconstructing a signal from a complete or partial set of DCT coefficients. The inverse discrete cosine transform is

$$x(n) = \sum_{k=1}^{N} w(k)y(k)\cos\frac{\pi(2n-1)(k-1)}{2N}, \qquad n = 1, \ldots, N$$

where

$$w(n) = \begin{cases} \frac{1}{\sqrt{N}}, & n = 1 \\ \sqrt{\frac{2}{N}}, & 2 \le n \le N \end{cases}$$

Because of the energy compaction mentioned above, it is possible to reconstruct a signal from only a fraction of its DCT coefficients. For example, generate a 25 Hz sinusoidal sequence, sampled at 1000 Hz:

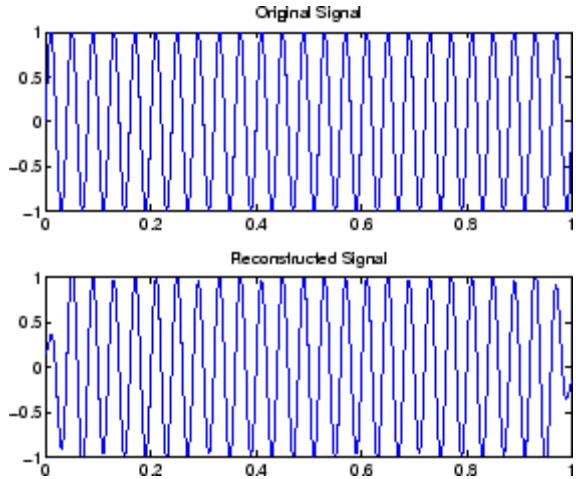
```
t = (0:1/999:1);
x = sin(2*pi*25*t);
```

Compute the DCT of this sequence and reconstruct the signal using only those components with value greater than 0.1 (64 of the original 1000 DCT coefficients):

```
y = dct(x)                    % Compute DCT
y2 = find(abs(y) < 0.9);      % Use 17 coefficients
y(y2) = zeros(size(y2));       % Zero out points < 0.9
z = idct(y);                  % Reconstruct signal w/inverse DCT
```

Plot the original and reconstructed sequences:

```
subplot(2,1,1); plot(t,x);
title('Original Signal')
subplot(2,1,2); plot(t,z), axis([0 1 -1 1])
title('Reconstructed Signal')
```



One measure of the accuracy of the reconstruction is

```
norm(x-z)/norm(x)
```

that is, the norm of the difference between the original and reconstructed signals, divided by the norm of the original signal. In this case, the relative error of reconstruction is 0.1443. The reconstructed signal retains approximately 85% of the energy in the original signal.

Hilbert Transform

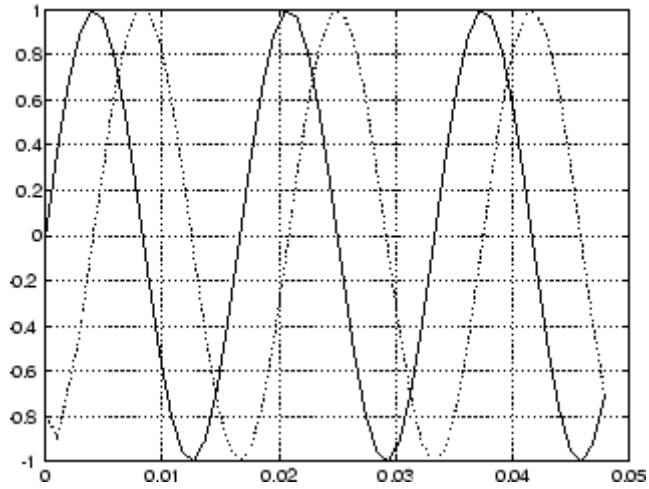
The Hilbert transform facilitates the formation of the analytic signal. The analytic signal is useful in the area of communications, particularly in bandpass signal processing. The toolbox function `hilbert` computes the Hilbert transform for a real input sequence `x` and returns a complex result of the same length

```
y = hilbert(x)
```

where the real part of `y` is the original real data and the imaginary part is the actual Hilbert transform. `y` is sometimes called the *analytic signal*, in reference to the continuous-time analytic signal. A key property of the discrete-time analytic signal is that its *z*-transform is 0 on the lower half of the unit circle. Many applications of the analytic signal are related to this property; for example, the analytic signal is useful in avoiding aliasing effects for bandpass sampling operations. The magnitude of the analytic signal is the complex envelope of the original signal.

The Hilbert transform is related to the actual data by a 90° phase shift; sines become cosines and vice versa. To plot a portion of data (solid line) and its Hilbert transform (dotted line):

```
t = (0:1/1023:1);
x = sin(2*pi*60*t);
y = hilbert(x);
plot(t(1:50),real(y(1:50))), hold on
plot(t(1:50),imag(y(1:50)),':'), hold off
```



The analytic signal is useful in calculating *instantaneous attributes* of a time series, the attributes of the series at any point in time. The instantaneous amplitude of the input sequence is the amplitude of the analytic signal. The instantaneous phase angle of the input sequence is the (unwrapped) `angle` of the analytic signal; the instantaneous frequency is the time rate of change of the instantaneous phase angle. You can calculate the instantaneous frequency using the MATLAB function, `diff`.

Walsh–Hadamard Transform

The Walsh–Hadamard transform is a non-sinusoidal, orthogonal transformation technique that decomposes a signal into a set of basis functions. These basis functions are Walsh functions, which are rectangular or square waves with values of +1 or –1. Walsh–Hadamard transforms are also known as Hadamard (see the `hadamard` function in the MATLAB software), Walsh, or Walsh-Fourier transforms.

The first eight Walsh functions have these values:

Index	Walsh Function Values
0	1 1 1 1 1 1 1 1
1	1 1 1 1 -1 -1 -1 -1

Index	Walsh Function Values
2	1 1 -1 -1 -1 -1 1 1
3	1 1 -1 -1 1 1 -1 -1
4	1 -1 -1 1 1 -1 -1 1
5	1 -1 -1 1 -1 1 1 -1
6	1 -1 1 -1 -1 1 -1 1
7	1 -1 1 -1 1 -1 1 -1

The Walsh–Hadamard transform returns sequency values. Sequency is a more generalized notion of frequency and is defined as one half of the average number of zero-crossings per unit time interval. Each Walsh function has a unique sequency value. You can use the returned sequency values to estimate the signal frequencies in the original signal.

Three different ordering schemes are used to store Walsh functions: sequency, Hadamard, and dyadic. Sequency ordering, which is used in signal processing applications, has the Walsh functions in the order shown in the table above. Hadamard ordering, which is used in controls applications, arranges them as 0, 4, 6, 2, 3, 7, 5, 1. Dyadic or gray code ordering, which is used in mathematics, arranges them as 0, 1, 3, 2, 6, 7, 5, 4.

The Walsh–Hadamard transform is used in a number of applications, such as image processing, speech processing, filtering, and power spectrum analysis. It is very useful for reducing bandwidth storage requirements and spread-spectrum analysis. Like the FFT, the Walsh–Hadamard transform has a fast version, the fast Walsh–Hadamard transform (`fwht`). Compared to the FFT, the FWHT requires less storage space and is faster to calculate because it uses only real additions and subtractions, while the FFT requires complex values. The FWHT is able to represent signals with sharp discontinuities more accurately using fewer coefficients than the FFT. Both the FWHT and the inverse FWHT (`ifwht`) are symmetric and thus, use identical calculation processes. The FWHT and IFWHT for a signal x(t) of length N are defined as:

$$y_n = \frac{1}{N}\sum_{i=0}^{N-1} x_i WAL(n,i)$$

$$x_i = \sum_{i=0}^{N-1} y_n WAL(n,i)$$

where i = 0,1,..., N-1 and *WAL(n,i)* are Walsh functions. Similar to the Cooley-Tukey algorithm for the FFT, the N elements are decomposed into two sets of $N/2$ elements, which are then combined using a butterfly structure to form the FWHT. For images, where the input is typically a 2–D signal, the FWHT coefficients are calculated by first evaluating across the rows and then evaluating down the columns.

For the following simple signal, the resulting FWHT shows that `x` was created using Walsh functions with sequency values of 0, 1, 3, and 6, which are the non-zero indices of the transformed `x`. The inverse FWHT recreates the original signal.

```
x = [4 2 2 0 0 2 -2 0]
y = fwht(x)
y
     1     1     0     1     0     0     1     0

x1 = ifwht(y)
x1
     4     2     2     0     0     2    -2     0
```

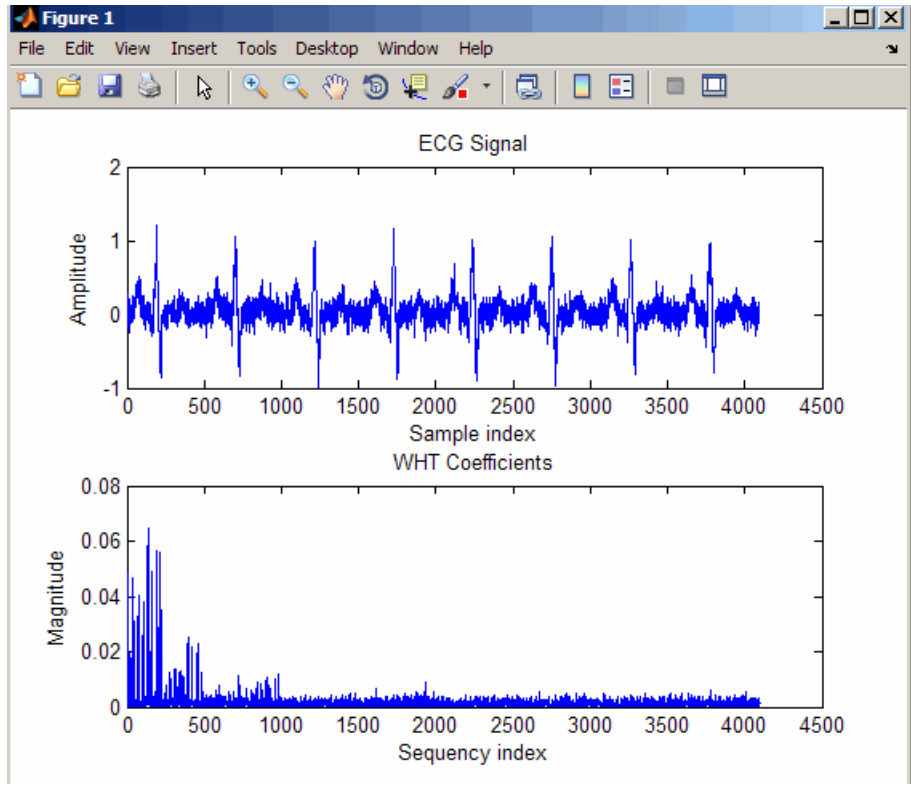
Using Walsh-Hadamard Transform for Spectral Analysis and Compression of ECG Signals

The following example uses an electrocardiogram (ECG) signal to illustrate working with the Walsh-Hadamard transform. ECG signals typically are very large and need to be stored for analysis and retrieval at a future time. Walsh-Hadamard transforms are particularly well-suited to this application because they provide compression and thus, require less storage space, plus they also provide rapid signal reconstruction.

Start with an ECG signal. For this example, we replicate it to create a longer signal and insert some additional random noise. Then, transform the signal

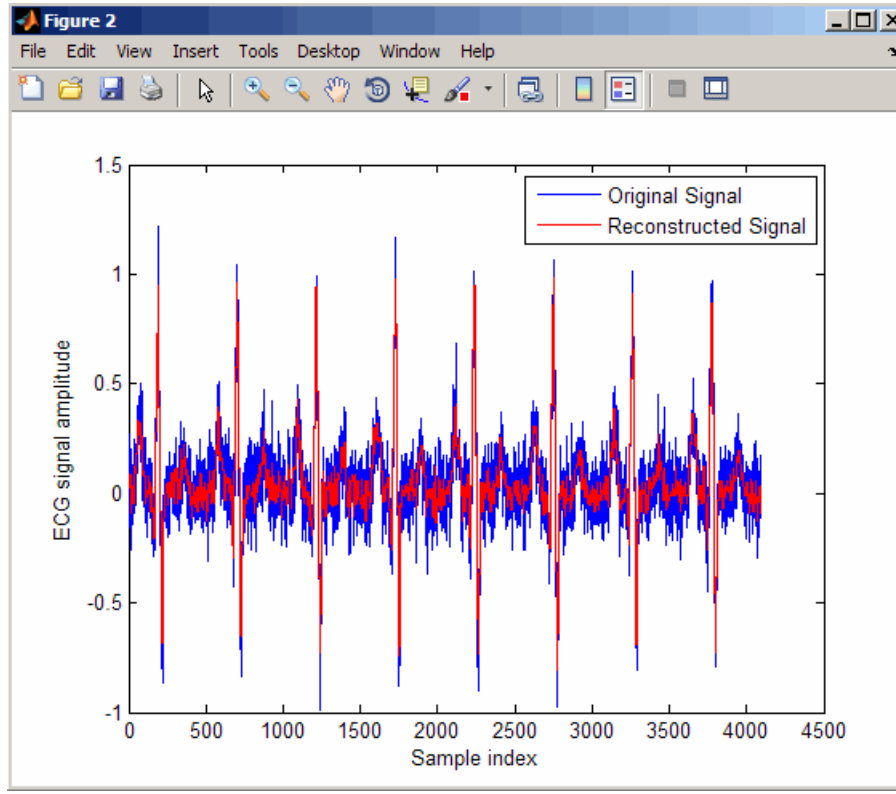
using the fast Walsh-Hadamard transform and plot the original signal and the transformed signal.

```
xe = ecg(512);   % Single ecg wave
xr = repmat(xe,1,8); % Replicate it to create more data
x = xr + 0.1.*randn(1,length(xr));  % Add noise

% Fast Walsh-Hadamard transform. Use default values
% for the number of points to use for the transform and
% for the ordering - sequency -- why???

y = fwht(x);
figure('color','white');
subplot(2,1,1);
plot(x);       % Plot original noisy signal
xlabel('Sample index');
ylabel('Amplitude');
title('ECG Signal');
subplot(2,1,2);
plot(abs(y))      % Plot magnitude of transformed signal
xlabel('Sequency index');
ylabel('Magnitude');
title('WHT Coefficients');
```



The plot shows that most of the signal energy is in the lower sequency values below approximately 1100. We will store only the first 1024 coefficients and see if the signal can be accurately reconstructed from only these stored coefficients.

```
y(1025:length(x)) = 0;            % Zeroing out the higher coefficients
xHat = ifwht(y);                  % Signal reconstruction using inverse WHT
figure('color','white');
plot(x);
hold on
plot(xHat,'r');
xlabel('Sample index');
ylabel('ECG signal amplitude');
legend('Original Signal','Reconstructed Signal');
```



The reproduced signal is very close to the original signal.

To reconstruct the original signal, we stored only the first 1024 coefficients and the ECG signal length. This represents a compression ratio of approximately 4:1. Storing more coefficients is a tradeoff between increased resolution and increased noise, while storing fewer coefficients may cause loss of peaks.

```
req = [length(x) y(1:1024)];
whos x req
```

```
  Name      Size              Bytes  Class     Attributes

  req       1x1025             8200  double
  x         1x4096            32768  double
```

Selected Bibliography

[1] Kay, S.M. *Modern Spectral Estimation*. Englewood Cliffs, NJ: Prentice Hall, 1988.

[2] Oppenheim, A.V., and R.W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

[3] Oppenheim, A.V., and R.W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1975, Section 10.5.3.

[4] Parks, T.W., and C.S. Burrus. *Digital Filter Design*. New York: John Wiley & Sons, 1987.

[5] Pratt,W.K. *Digital Image Processing*. New York: John Wiley & Sons, 1991.

8

SPTool: A Signal Processing GUI Suite

SPTool: An Interactive Signal Processing Environment

In this section...

SPTool Overview

SPTool is an interactive GUI for digital signal processing that can be used to

- Analyze signals
- Design filters
- Analyze (view) filters
- Filter signals
- Analyze signal spectra

You can accomplish these tasks using four GUIs that you access from within SPTool:

- The "Signal Browser" on page 8-7 is for analyzing signals. You can also play portions of signals using your computer's audio hardware.
- The Filter Design and Analysis Tool (FDATool) is available for designing or editing FIR and IIR digital filters. Most Signal Processing Toolbox filter design methods available at the command line are also available in FDATool. Additionally, you can use FDATool to design a filter by using the "Pole/Zero Editor" on page 8-49 to graphically place poles and zeros on the *z*-plane.
- The "Filter Visualization Tool" on page 8-12 (FVTool) is for analyzing filter characteristics.
- The "Spectrum Viewer" on page 8-14 is for spectral analysis. You can use Signal Processing Toolbox spectral estimation methods to estimate the power spectral density of a signal.

SPTool Data Structures

You can use SPTool to analyze signals, filters, or spectra that you create at the MATLAB command line.

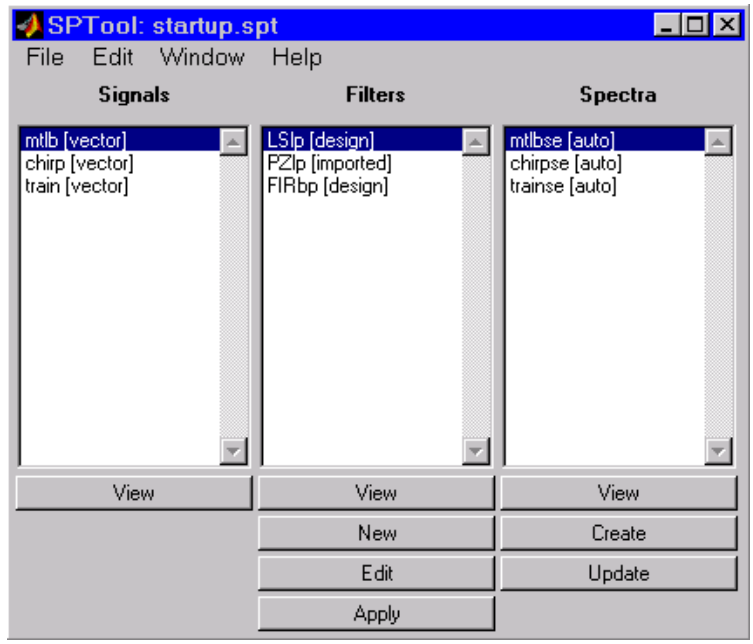
You can bring signals, filters, or spectra from the MATLAB workspace into the SPTool workspace using **File > Import**. For more information, see "Importing Filters and Spectra" on page 8-33. Signals, filters, or spectra that you create in (or import into) the SPTool workspace exist as MATLAB structures. See the MATLAB documentation for more information on MATLAB structures.

When you use **File > Export** to save signals, filters, and spectra that you create or modify in SPTool, these are also saved as MATLAB structures. For more information on exporting, see "Exporting Signals, Filters, and Spectra" on page 8-28.

Opening SPTool

To open SPTool, type

```
sptool
```



When you first open SPTool, it contains a collection of default signals, filters, and spectra. To specify your own preferences for what signals, filters, and spectra to see when SPTool opens see “Setting Preferences” on page 8-44.

You can access these three GUIs from SPTool by selecting a signal, filter, or spectrum and clicking the appropriate **View** button:

- Signal Browser
- Filter Visualization Tool
- Spectrum Viewer

You can access FDATool by clicking **New** to create a new filter or `Edit` to edit a selected filter. Clicking **Apply** applies a selected filter to a selected signal.

Create opens the Spectrum Viewer and creates the power spectral density of the selected signal. **Update** opens the Spectrum Viewer for the selected spectrum.

Getting Context-Sensitive Help

To find information on a particular region of the Signal Browser, Filter Designer, or Spectrum Viewer:

1. Click **What's this?**  .
2. Click on the region of the GUI you want information on.

You can also use **Help > What's This?** to launch context-sensitive help.

Signal Browser

In this section...

Overview of the Signal Browser

You can use the Signal Browser to display and analyze signals listed in the **Signals** list box in SPTool.

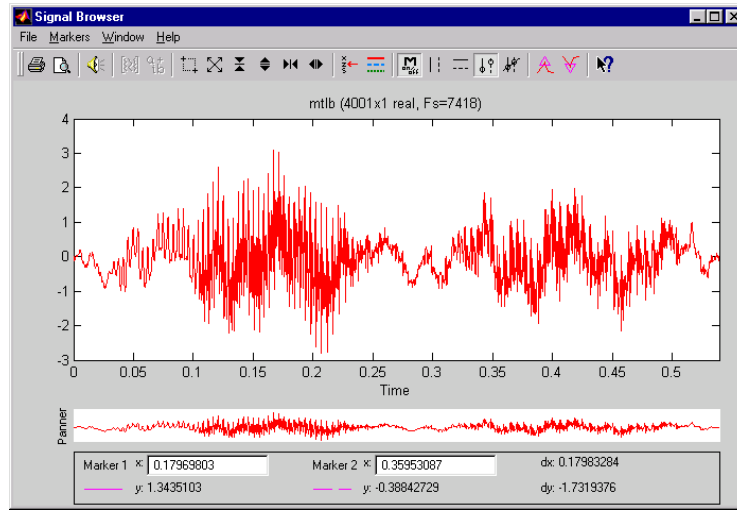
Using the Signal Browser you can

- Analyze and compare vector or array (matrix) signals.
- Zoom in on portions of signal data.
- Measure a variety of characteristics of signal data.
- Compare multiple signals.
- Play portions of signal data on audio hardware.
- Print signal plots.

Opening the Signal Browser

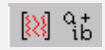
To open the Signal Browser from SPTool:

1. Select one or more signals in the **Signals** list in SPTool.
2. Click **View** under the **Signals** list.



The Signal Browser has the following components:

- A display region for analyzing signals, including markers for measuring, comparing, or playing signals
- A "panner" that displays the entire signal length, highlighting the portion currently active in the display region
- A marker measurements area
- A toolbar with buttons for convenient access to frequently used functions

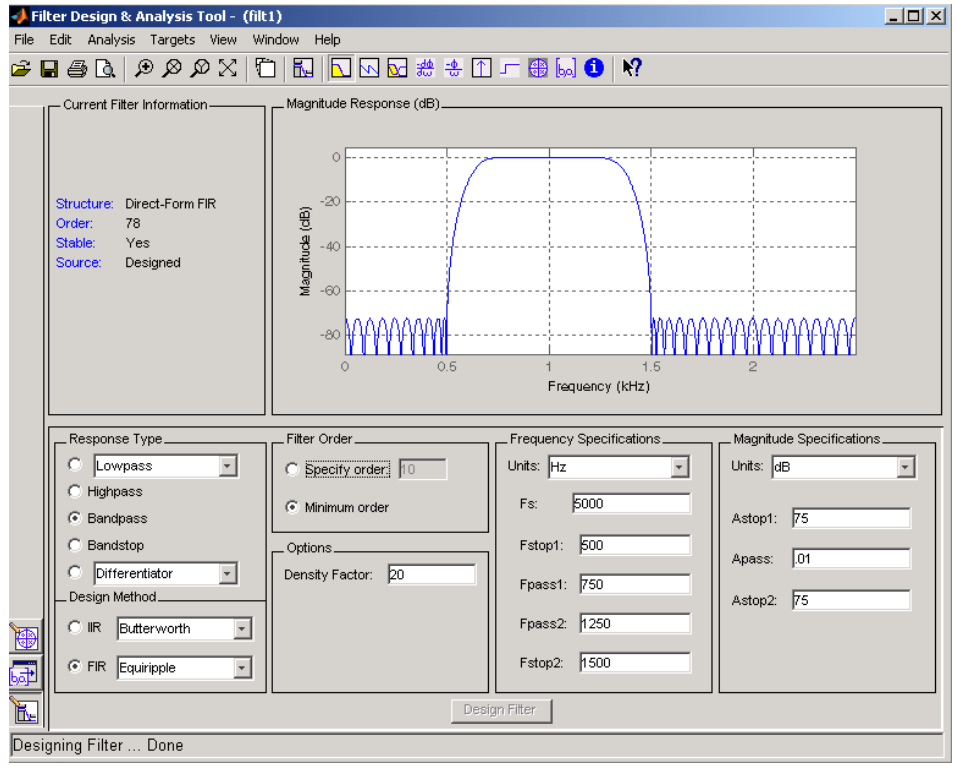
Icon	Description
	Print and print preview
	Play an audio signal
	Display array and complex signals
	Zoom the signal in and out
	Select one of several loaded signals

Icon	Description
	Set the display color and line style of a signal
	Toggle the markers on and off
	Set marker types (See “Making Signal Measurements with Markers” on page 8-42)
	Turn on the What’s This help

FDATool

You can use a reduced version of the Filter Design and Analysis Tool (`fdatool`) to design and edit filters.

To open FDATool from SPTool, click **New** under the **Filters** list to create a new filter or select one of the filters in the **Filters** list in SPTool and click **Edit** to edit that filter.



For details about FDATool, see Chapter 5, “FDATool: A Filter Design and Analysis GUI”.

Note When you open FDATool from SPTool, a reduced version of FDATool that is compatible with SPTool opens.

Filter Visualization Tool

In this section...

Connection between FVTool and SPTool

You can use the Filter Visualization Tool to analyze response characteristics of the selected filter(s). See `fvtool` for detailed information about FVTool.

If you start FVTool by clicking the SPTool **Filter View** button, that FVTool is linked to SPTool. Any changes made in SPTool to the filter are immediately reflected in FVTool. The FVTool title bar includes "SPTool" to indicate the link.

If you start an FVTool by clicking the **New** button or by selecting **File > New** from within FVTool, that FVTool is a standalone version and is not linked to SPTool.

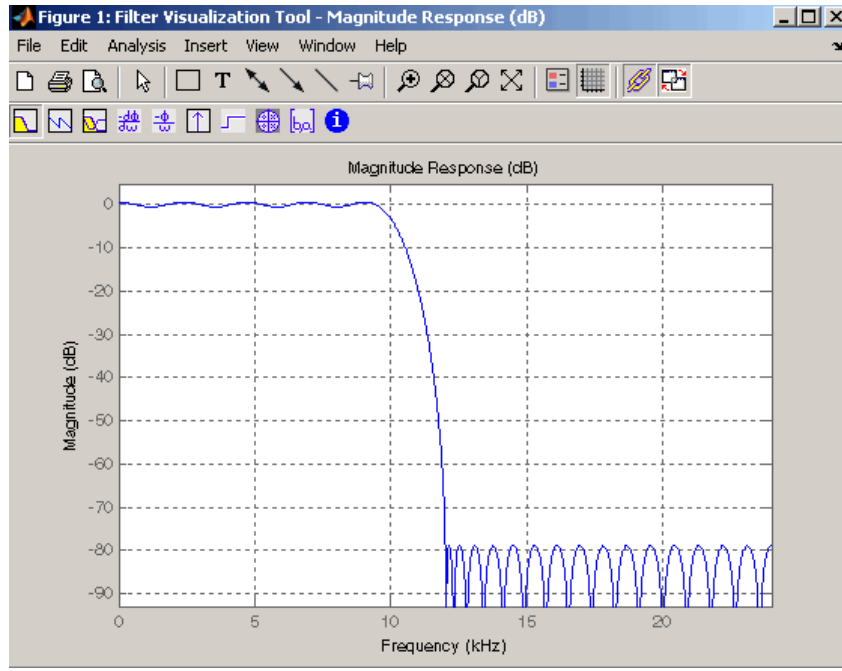
Note Every time you click the **Filter View** button a new, linked FVTool starts. This allows you to view multiple analyses simultaneously.

Opening the Filter Visualization Tool

You open FVTool from SPTool as follows.

1. Select one or more filters in the **Filters** list in SPTool.
2. Click the **View** button under the **Filters** list.

When you first open FVTool, it displays the selected filter's magnitude plot.



Analysis Parameters

In the plot area of any filter response plot, right-click and select **Analysis Parameters** to display details about the displayed plot. See "Analysis Parameters" in the FDATool online help for more information.

You can change any parameter in a linked FVTool, except the sampling frequency. You can only change the sampling frequency using the SPTool **Edit > Sampling Frequency** or the SPTool **Filters Edit** button.

Spectrum Viewer

In this section...

Spectrum Viewer Overview

You can use the Spectrum Viewer for estimating and analyzing a signal's power spectral density (PSD). You can use the PSD estimates to understand a signal's frequency content.

The Spectrum Viewer provides the following functionality.

- Analyze and compare spectral density plots.
- Use different spectral estimation methods to create spectra:
  - Burg (`pburg`)
  - Covariance (`pcov`)
  - FFT (`fft`)
  - Modified covariance (`pmcov`)
  - MTM (multitaper method) (`pmtm`)
  - MUSIC (`pmusic`)
  - Welch (`pwelch`)
  - Yule-Walker AR (`pyulear`)
- Modify power spectral density parameters such as FFT length, window type, and sample frequency.
- Print spectral plots.

Opening the Spectrum Viewer

To open the Spectrum Viewer and create a PSD estimate from SPTool:

1. Select a signal from the **Signal** list box in SPTool.

2 Click **Create** in the **Spectra** list.

3 Click **Apply** in the Spectrum Viewer.

To open the Spectrum Viewer with a PSD estimate already listed in SPTool:

1 Select a PSD estimate from the **Spectra** list box in SPTool.

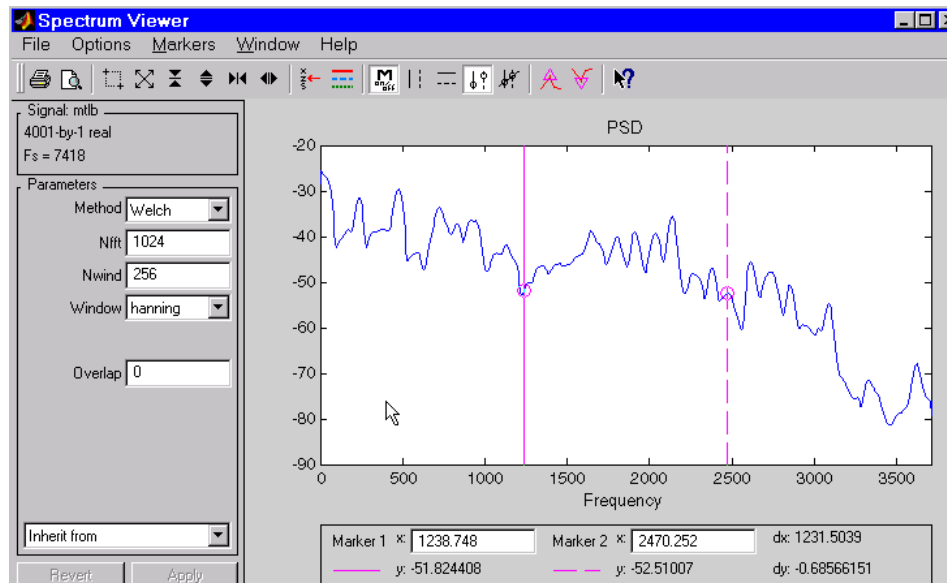
2 Click **View** in the Spectra list.

For example:

1 Select `mtlb` in the default **Signals** list in SPTool.

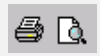
2 Click **Create** in SPTool to open the Spectrum Viewer.

3 Click **Apply** in the Spectrum Viewer to plot the spectrum.



The Spectrum Viewer has the following components:

- A signal identification region that provides information about the signal whose power spectral density estimate is displayed
- A Parameters region for modifying the PSD parameters
- A display region for analyzing spectra and an **Options** menu for modifying display characteristics
- Spectrum management controls
  - **Inherit from** menu to inherit PSD specifications from another PSD object listed in the menu
  - Revert button to revert to the named PSD's original specifications
  - **Apply** button for creating or updating PSD estimates
- A toolbar with buttons for convenient access to frequently used functions

Icon	Description
	Print and print preview
	Zoom the signal in and out
	Select one of several loaded signals
	Set the display color and line style of a signal
	Toggle the markers on and off
	Set marker types
	Turn on the What's This help

Filtering and Analysis of Noise

In this section...

Overview

The following sections provide an example of using the GUI-based interactive tools to:

- Design and implement an FIR bandpass digital filter
- Apply the filter to a noisy signal
- Analyze signals and their spectra

The steps include:

1. Creating a noisy signal in the MATLAB workspace and importing it into SPTool
2. Designing a bandpass filter using FDATool
3. Applying the filter to the original noise signal to create a bandlimited noise signal
4. Comparing the time domain information of the original and filtered signals using the Signal Browser
5. Comparing the spectra of both signals using the Spectrum Viewer

Step 1: Importing a Signal into SPTool

To import a signal into SPTool from the workspace or disk, the signal must be either:

- A special MATLAB signal structure, such as that saved from a previous SPTool session
- A signal created as a variable (vector or matrix) in the MATLAB workspace

For this example, create a new signal at the command line and then import it as a structure into SPTool:

1. Create a random signal in the MATLAB workspace by typing

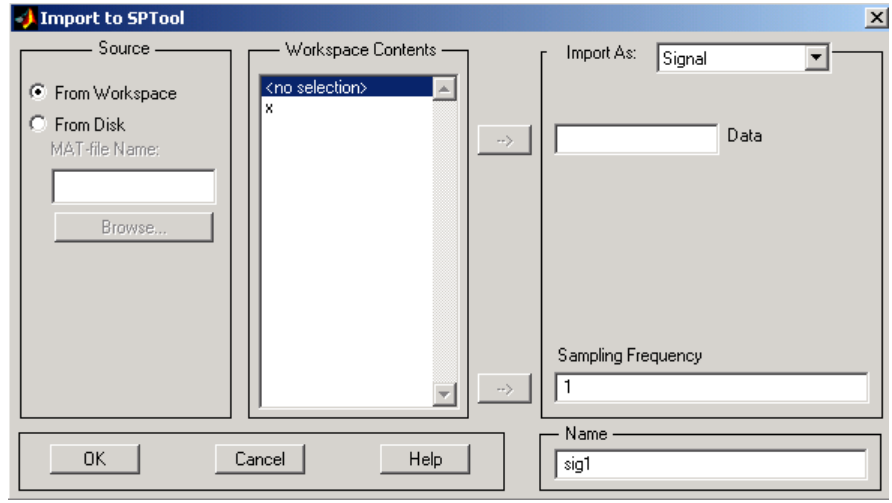
```
randn('state',0);
x = randn(5000,1);
```

2. If SPTool is not already open, open SPTool by typing

```
sptool
```

   The SPTool window is displayed.

3. Select **File > Import**. The Import to SPTool dialog opens.



The variable `x` is displayed in the **Workspace Contents** list. (If it is not, select the **From Workspace** radio button to display the contents of the workspace.)

4. Select the signal and import it into the **Data** field:
   a. Select the signal variable `x` in the **Workspace Contents** list.
   b. Make sure that `Signal` is selected in the **Import As** pull-down menu.
   c. Click on the arrow to the left of the **Data** field or type `x` in the **Data** field.
   d. Type `5000` in the **Sampling Frequency** field.
   e. Name the signal by typing `noise` in the **Name** field.
   f. Click **OK**.

   The signal `noise[vector]` appears and is selected in SPTool's **Signals** list.

Note You can import filters and spectra into SPTool in much the same way as you import signals. See "Importing Filters and Spectra" on page 8-33 for specific details.

You can also import signals from MAT-files on your disk, rather than from the workspace. See "Loading Variables from the Disk" on page 8-37 for more information.

Type `help sptool` for information about importing from the command line.

Step 2: Designing a Filter

You can import an existing filter into SPTool, or you can design and edit a new filter using FDATool.

In this example, you

1. Open a default filter in FDATool.
2. Specify an equiripple bandpass FIR filter.

Opening FDATool

To open FDATool, click **New** in SPTool. FDATool opens with a default filter named `filt1`.

Specifying the Bandpass Filter

Design an equiripple bandpass FIR filter with the following characteristics:

- Sampling frequency of 5000 Hz
- Stopband frequency ranges of [0 500] Hz and [1500 2500] Hz
- Passband frequency range of [750 1250] Hz
- Ripple in the passband of 0.01 dB
- Stopband attenuation of 75 dB

To modify the filter in FDATool to meet these specifications, you need to

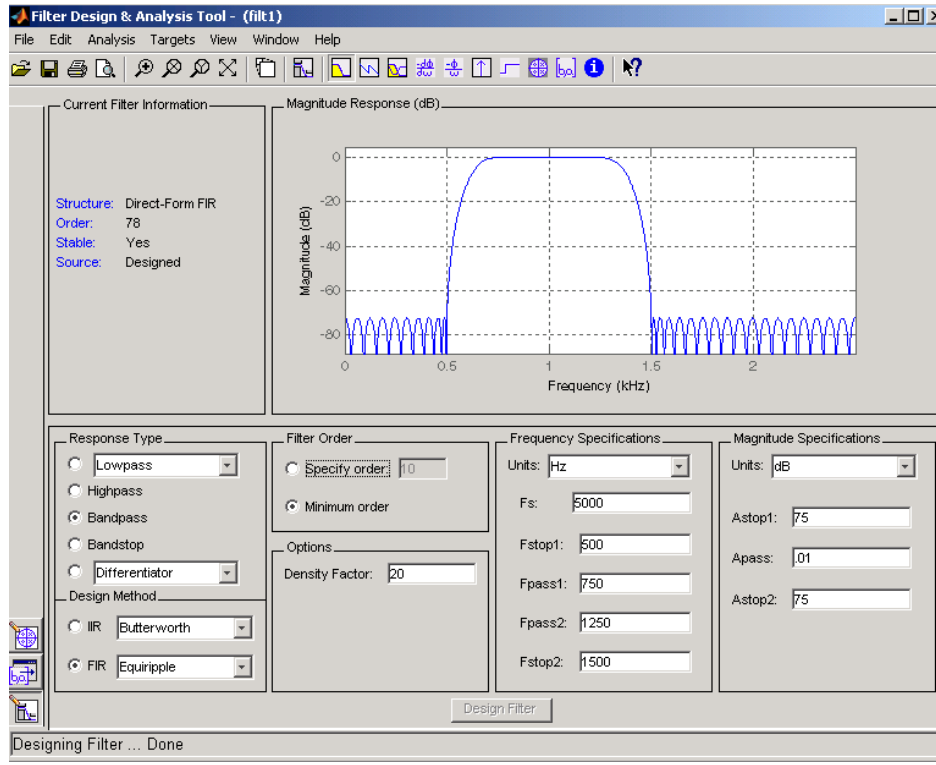
1. Select `Bandpass` from the **Response Type** list.
2. Verify that **FIR Equiripple** is selected as the **Design Method**.
3. Verify that **Minimum order** is selected as the **Filter Order** and that the **Density Factor** is set to `20`.
4. Under **Frequency Specifications**, set the sampling frequency (**Fs**) and the passband (**Fpass1**, **Fpass2**) and stopband (**Fstop1**, **Fstop2**) edges:

Units	Hz
Fs	5000
Fstop1	500
Fpass1	750
Fpass2	1250
Fstop2	1500

5. Under **Magnitude Specifications**, set the stopband attenuation (**Astop1**, **Astop2**) and the maximum passband ripple (**Apass**):

Units	dB
Astop1	75
Apass	0.01
Astop2	75

6 Click **Design Filter** to design the new filter. When the new filter is designed, the magnitude response of the filter is displayed.



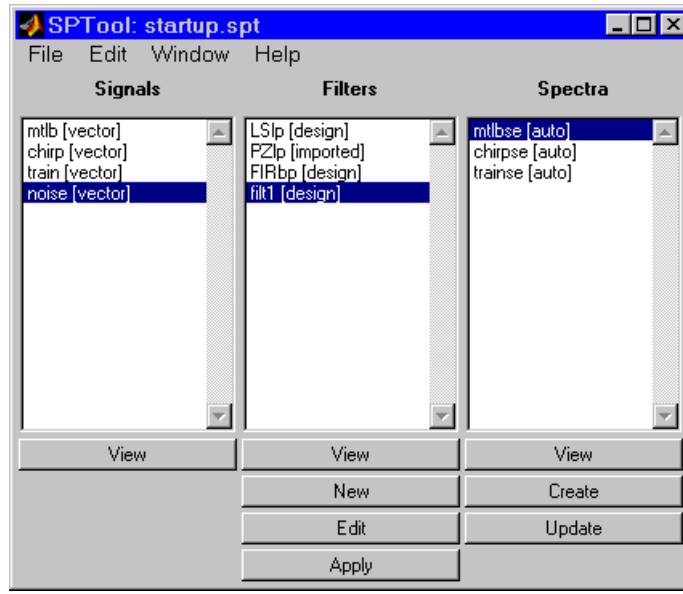
The resulting filter is an order-78 bandpass equiripple filter.

Step 3: Applying a Filter to a Signal

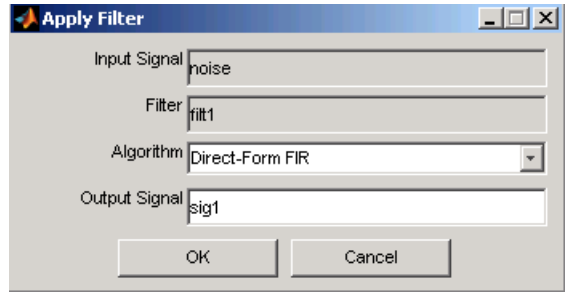
When you apply a filter to a signal, you create a new signal in SPTool representing the filtered signal.

To apply the filter `filt1` you just created to the signal `noise`,

1. In SPTool, select the signal `noise[vector]` from the **Signals** list and select the filter (named `filt1[design]`) from the **Filters** list.



2. Click **Apply** under the **Filters** list.



3. Leave the **Algorithm** as `Direct-Form FIR`.

Note You can apply one of two filtering algorithms to FIR filters. The default algorithm is specific to the filter structure, which is shown in the FDATool Current Filter Info frame. Alternately for FIR filters, `FFT based FIR (fftfilt)` uses the algorithm described in `fftfilt`.

For IIR filters, the alternate algorithm is a zero-phase IIR that uses the algorithm described in `filtfilt`.

4. Enter `blnoise` as the **Output Signal** name.

5. Click **OK** to close the Apply Filter dialog box.

   The filter is applied to the selected signal, and the filtered signal `blnoise[vector]` is listed in the **Signals** list in SPTool.

Step 4: Analyzing a Signal

You can analyze and print signals using the Signal Browser. You can also play the signals if your computer has audio output capabilities.

For example, compare the signal `noise` to the filtered signal `blnoise`:

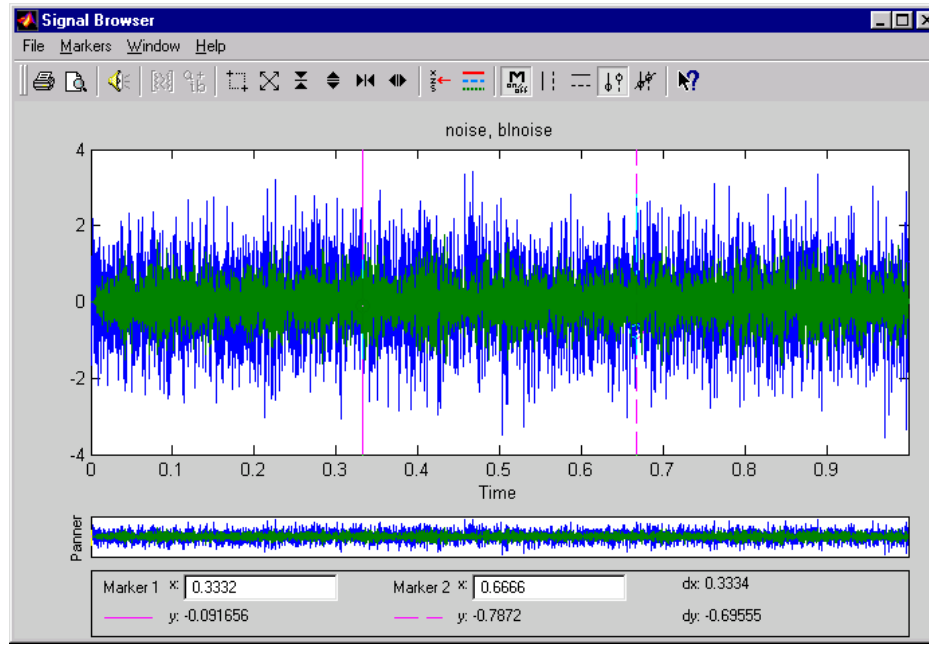
1. **Shift**+click on the `noise` and `blnoise` signals in the **Signals** list of SPTool to select both signals.

2. Click **View** under the **Signals** list.

   The Signal Browser is activated, and both signals are displayed in the display region. (The names of both signals are shown above the display region.) Initially, the original `noise` signal covers up the bandlimited `blnoise` signal.

3. Push the selection button  on the toolbar to select the `blnoise` signal.

   The display area is updated. Now you can see the `blnoise` signal superimposed on top of the `noise` signal. The signals are displayed in different colors in both the display region and the panner. You can change the color of the selected signal using the *Line Properties* button on the toolbar,  .



Playing a Signal

When you click **Play** in the Signal Browser toolbar,  , the active signal is played on the computer's audio hardware:

1. To hear a portion of the active (selected) signal
   a. Use the vertical markers to select a portion of the signal you want to play. Vertical markers are enabled by the  and  buttons.
   b. Click **Play**.
2. To hear the other signal
   a. Select the signal as in step above. You can also select the signal directly in the display region.
   b. Click **Play** again.

Printing a Signal

You can print from the Signal Browser using the **Print** button,  .

You can use the line display buttons to maximize the visual contrast between the signals by setting the line color for noise to gray and the line color for blnoise to white. Do this before printing two signals together.

Note You can follow the same rules to print spectra, but you can't print filter responses directly from SPTool.

Use the Signal Browser region in the Preferences dialog box in SPTool to suppress printing of both the panner and the marker settings.

To print both signals, click **Print** in the Signal Browser toolbar.

Step 5: Spectral Analysis in the Spectrum Viewer

You can analyze the frequency content of a signal using the Spectrum Viewer, which estimates and displays a signal's power spectral density.

For example, to analyze and compare the spectra of `noise` and `blnoise:`

1. Create a power spectral density (PSD) object, `spect1`, that is associated with the signal `noise`, and a second PSD object, spect2, that is associated with the signal `blnoise`.
2. Open the Spectrum Viewer to analyze both of these spectra.
3. Print both spectra.

Creating a PSD Object From a Signal

1. Click on SPTool, or select **Window > SPTool** in any active open GUI. SPTool is now the active window.
2. Select the `noise[vector]` signal in the **Signals** list of SPTool.

3 Click **Create** in the **Spectra** list.

The Spectrum Viewer is activated, and a PSD (`spect1`) corresponding to the `noise` signal is created in the **Spectra** list. The PSD is not computed or displayed yet.

4 Click **Apply** in the Spectrum Viewer to compute and display the PSD estimate `spect1` using the default parameters.

The PSD of the `noise` signal is displayed in the display region. The identifying information for the PSD's associated signal (`noise`) is displayed above the Parameters region.

The PSD estimate `spect1` is within 2 or 3 dB of 0, so the noise has a fairly "flat" power spectral density.

5 Follow steps 1 through 4 for the bandlimited noise signal blnoise to create a second PSD estimate `spect2`.

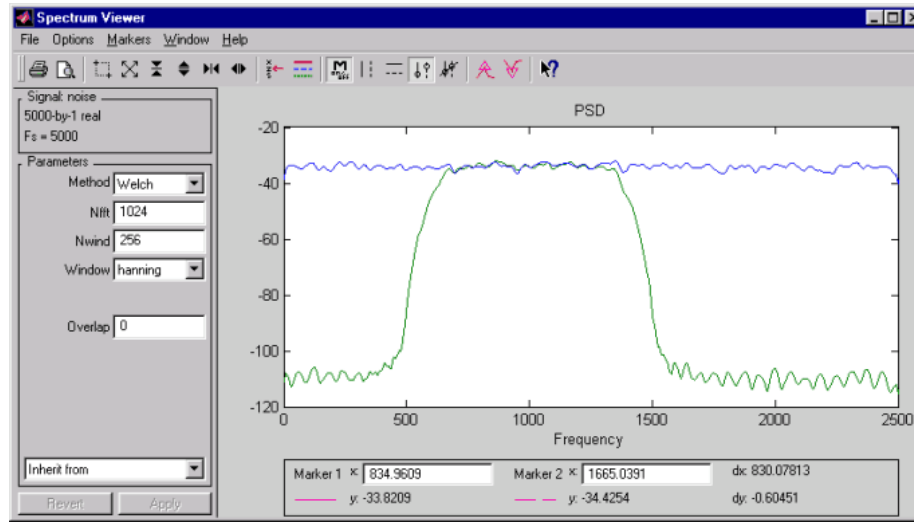
The PSD estimate `spect2` is flat between 750 and 1250 Hz and has 75 dB less power in the stopband regions of `filt1`.

Opening the Spectrum Viewer with Two Spectra

1 Reactivate SPTool again, as in step 1 above.

2 **Shift+click** on `spect1` and `spect2` in the **Spectra** list to select them both.

3 Click **View** in the **Spectra** list to reactivate the Spectrum Viewer and display both spectra together.



Printing the Spectra

Before printing the two spectra together, use the color and line style selection button,  , to differentiate the two plots by line style, rather than by color.

To print both spectra:

1. Click **Print Preview**  in the toolbar on the Spectrum Viewer.
2. From the Spectrum Viewer Print Preview window, drag the legend out of the display region so that it doesn't obscure part of the plot.
3. Click **Print** in the Spectrum Viewer Print Preview window.

Exporting Signals, Filters, and Spectra

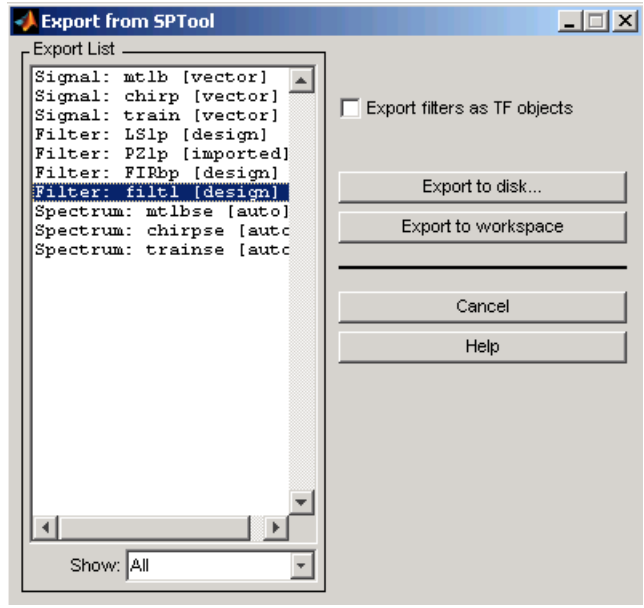
In this section...

Opening the Export Dialog Box

To save the filter `filt1` you just created in this example, open the Export dialog box with `filt1` preselected:

1. Select `filt1` in the SPTool **Filters** list.
2. Select **File > Export**.

The Export dialog box opens with `filt1` preselected.



Exporting a Filter to the MATLAB Workspace

To export the filter filt1 to the MATLAB workspace:

1. Select `filt1` from the **Export List** and deselect all other items using **Ctrl+click**.
2. Click **Export to Workspace**.

Accessing Filter Parameters

In this section...

Accessing Filter Parameters in a Saved Filter

The MATLAB structures created by SPTool have several associated fields, many of which are also MATLAB structures. See the MATLAB documentation for general information about MATLAB structures.

For example, after exporting a filter filt1 to the MATLAB workspace, type

```
filt1
```

to display the fields of the MATLAB filter structure. The `tf` field of the structure contains information that describes the filter.

The tf Field: Accessing Filter Coefficients

The `tf` field is a structure containing the transfer function representation of the filter. Use this field to obtain the filter coefficients;

- `filt1.tf.num` contains the numerator coefficients.
- `filt1.tf.den` contains the denominator coefficients.

The vectors contained in these structures represent polynomials in descending powers of z. The numerator and denominator polynomials are used to specify the transfer function

$$H(z) = \frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \cdots + b(nb+1)z^{-m}}{a(1) + a(2)z^{-1} + \cdots + a(na+1)z^{-n}}$$

where:

- b is a vector containing the coefficients from the `tf.num` field.
- a is a vector containing the coefficients from the `tf.den` field.

- m is the numerator order.
- n is the denominator order.

You can change the filter representation from the default transfer function to another form by using the `tf2ss` or `tf2zp` functions.

Note The `FDAspecs` field of your filter contains internal information about FDATool and should not be changed.

Accessing Parameters in a Saved Spectrum

The following structure fields describe the spectra saved by SPTool.

Field	Description
`P`	The spectral power vector.
`f`	The spectral frequency vector.
`confid`	A structure containing the confidence intervals data • The `confid.level` field contains the chosen confidence level. • The `confid.Pc` field contains the spectral power data for the confidence intervals. • The `confid.enable` field contains a `1` if confidence levels are enabled for the power spectral density.
`signalLabel`	The name of the signal from which the power spectral density was generated.
`Fs`	The associated signal's sample rate.

You can access the information in these fields as you do with every MATLAB structure.

For example, if you export an SPTool PSD estimate `spect1` to the workspace, type

```
spect1.P
```

to obtain the vector of associated power values.

Importing Filters and Spectra

In this section...

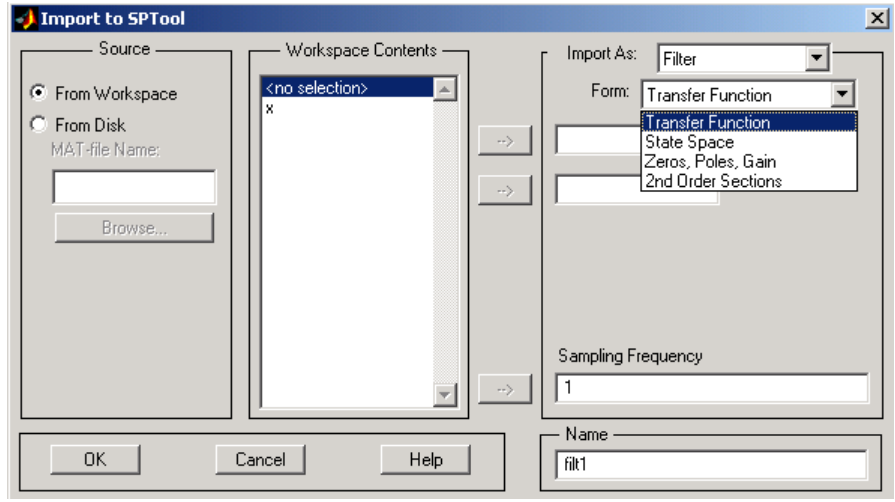
Similarities to Other Procedures

The procedures are very similar to those explained in

- "Step 1: Importing a Signal into SPTool" on page 8-18 for loading variables from the workspace
- "Loading Variables from the Disk" on page 8-37 for loading variables from your disk

Importing Filters

When you import filters, first select the appropriate filter form from the **Form** list.



For every filter you specify a variable name or a value for the filter's sampling frequency in the **Sampling Frequency** field. Each filter form requires different variables.

Transfer Function

For `Transfer Function`, you specify the filter by its transfer function representation:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \cdots + b(m+1)z^{-m}}{a(1) + a(2)z^{-1} + \cdots + a(n+1)z^{-n}}$$

- The **Numerator** field specifies a variable name or value for the numerator coefficient vector b, which contains m+1 coefficients in descending powers of z.
- The **Denominator** field specifies a variable name or value for the denominator coefficient vector a, which contains n+1 coefficients in descending powers of z.

State Space

For `State Space`, you specify the filter by its state-space representation:

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

The **A-Matrix**, **B-Matrix**, **C-Matrix**, and **D-Matrix** fields specify a variable name or a value for each matrix in this system.

Zeros, Poles, Gain

For `Zeros, Poles, Gain`, you specify the filter by its zero-pole-gain representation:

$$H(z) = \frac{Z(z)}{P(z)} = k\frac{(z - z(1))(z - z(2))\cdots(z - z(m))}{(z - p(1))(z - p(2))\cdots(z - p(n))}$$

- The **Zeros** field specifies a variable name or value for the zeros vector z, which contains the locations of m zeros.

- The **Poles** field specifies a variable name or value for the zeros vector p, which contains the locations of n poles.
- The **Gain** field specifies a variable name or value for the gain k.

Second Order Sections

For `2nd Order Sections` you specify the filter by its second-order section representation:

$$H(z) = \prod_{k=1}^{L} H_k(z) = \prod_{k=1}^{L} \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2}}{1 + a_{1k}z^{-1} + a_{2k}z^{-2}}$$

The **SOS Matrix** field specifies a variable name or a value for the L-by-6 SOS matrix

$$sos = \begin{bmatrix} b_{01} & b_{11} & b_{21} & 1 & a_{11} & a_{21} \\ b_{02} & b_{12} & b_{22} & 1 & a_{12} & a_{22} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{0L} & b_{1L} & b_{2L} & 1 & a_{1L} & a_{2L} \end{bmatrix}$$

whose rows contain the numerator and denominator coefficients b_{ik} and a_{ik} of the second-order sections of $H(z)$.

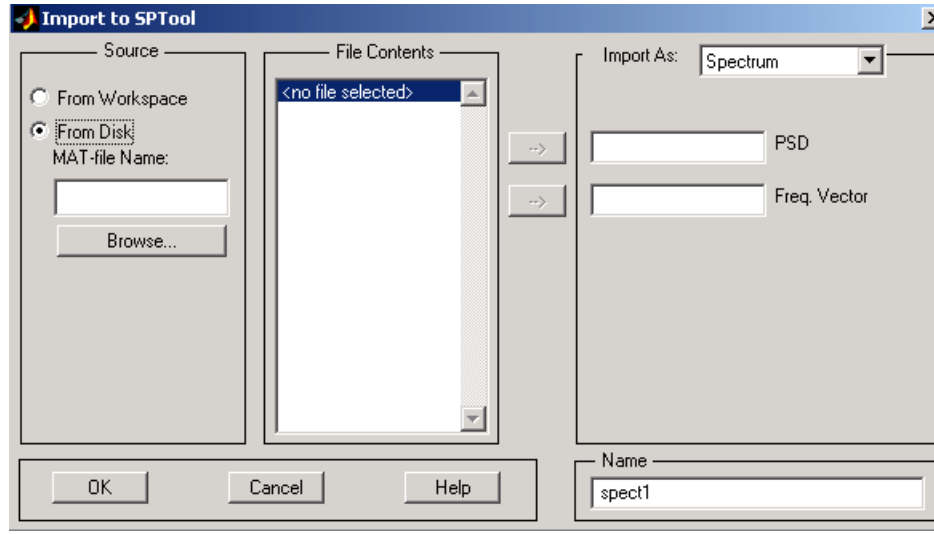
Note If you import a filter that was not created in SPTool, you can only edit that filter using the Pole/Zero Editor.

Importing Spectra

When you import a power spectral density (PSD), you specify:

- A variable name or a value for the PSD vector in the **PSD** field
- A variable name or a value for the frequency vector in the **Freq. Vector** field

The PSD values in the **PSD** vector correspond to the frequencies contained in the **Freq. Vector** vector; the two vectors must have the same length.



Loading Variables from the Disk

To import variables representing signals, filters, or spectra from a MAT-file on your disk;

1. Select the **From Disk** radio button and do either of the following:
   - Type the name of the file you want to import into the **MAT-file Name** field and press either the **Tab** or the **Enter** key on your keyboard.
   - Select **Browse**, and then find and select the file you want to import using **Select > File to Open**. Click **OK** to close that dialog.

   In either case, all variables in the MAT-file you selected are displayed in the **File Contents** list.
2. Select the variables to be imported into SPTool.

You can now import one or more variables from the **File Contents** list into SPTool, as long as these variables are scalars, vectors, or matrices.

Saving and Loading Sessions

In this section...

SPTool Sessions

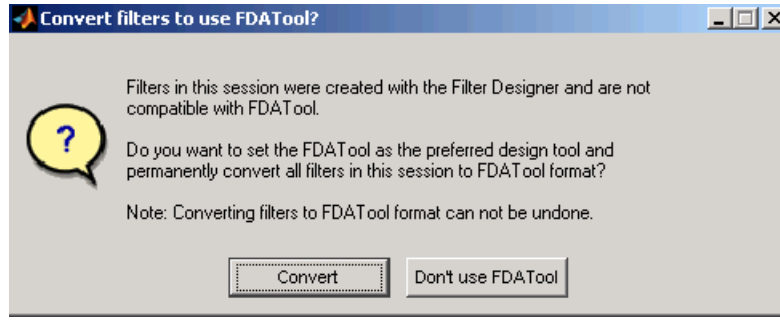
When you start SPTool, the default `startup.spt` session is loaded. To save your work in the startup SPTool session, use **File > Save Session** or to specify a session name, use **File > Save Session As**.

To recall a previously saved session, use **File > Open Session**.

Filter Formats

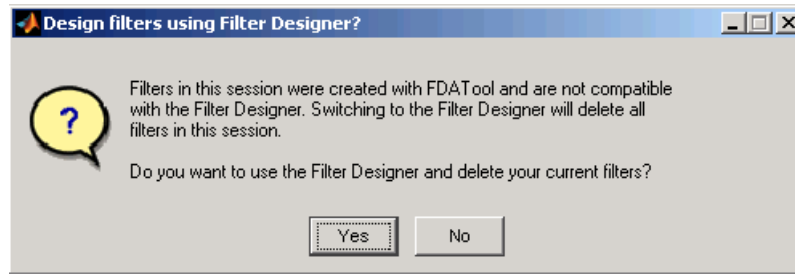
When you start SPTool or open a session, the current filter design format preference is compared to the filter formats in the session. See "Setting Preferences" on page 8-44.

- If the formats match, the session opens.
- If the filter preference is FDATool, but the session contains Filter Designer filters, this warning displays:



Click **Convert** to convert the filters to FDATool format. Click **Don't Use FDATool** to leave the filters in Filter Designer format and change the preference to **Use Filter Designer**.

- If the filter preference is **Use Filter Designer**, but the session contains FDATool filters, this warning displays:



Click **Yes** to remove the current filters. Click **NSelecting data** to leave the filters in FDATool and change the preference to **Use FDATool**.

Selecting Signals, Filters, and Spectra

All signals, filters, or spectra listed in SPTool exist as special MATLAB structures. You can bring data representing signals, filters, or spectra into SPTool from the MATLAB workspace. In general, you can select one or several items in a given list box. An item is selected when it is highlighted.

The **Signals** list shows all vector and array signals in the current SPTool session.

The **Filters** list shows all designed and imported filters in the current SPTool session.

The **Spectra** list shows all spectra in the current SPTool session.

You can select a single data object in a list, a range of data objects in a list, or multiple separate data objects in a list. You can also have data objects simultaneously selected in different lists:

- To select a single item, click it. All other items in that list box become deselected.
- To add or remove a range of items, **Shift+click** on the items at the top and bottom of the section of the list that you want to add. You can also drag your mouse pointer to select these items.
- To add a single data object to a selection or remove a single data object from a multiple selection, **Ctrl+click** on the object.

Editing Signals, Filters, or Spectra

You can edit selected items in SPTool by

1. Selecting the names of the signals, filters, or spectra you want to edit.
2. Selecting the appropriate **Edit** menu item:
   - **Duplicate** to copy an item in an SPTool list
   - **Clear** to delete an item in an SPTool list
   - **Name** to rename an item in an SPTool list
   - **Sampling Frequency** to modify the sampling frequency associated with either a signal (and its associated spectra) or filter in an SPTool list

   The pull-down menu next to each menu item shows the names of all selected items.

You can also edit the following signal characteristics by right-clicking in the display region of the Signal Browser, the Filter Visualization Tool, or the Spectrum Viewer:

- The signal name
- The sampling frequency
- The line style properties

Note If you modify the sampling frequency associated with a signal's spectrum using the right-click menu on the Spectrum Viewer display region, the sampling frequency of the associated signal is automatically updated.

Making Signal Measurements with Markers

You can use the markers on the Signal Browser or the Spectrum Viewer to make measurements on either of the following:

- A signal in the Signal Browser
- A power spectral density plotted in the Spectrum Viewer

The following marker buttons are included

Icon	Description
	Toggle markers on/off
	Vertical markers
	Horizontal markers
	Vertical markers with tracking
	Vertical markers with tracking and slope
	Display peaks (local maxima)
	Display valleys (local minima)

To make a measurement:

1. Select a line to measure (or play, if you are in the Signal Browser).
2. Select one of the marker buttons to apply a marker to the displayed signal.
3. Position a marker in the main display area by grabbing it with your mouse and dragging:

a Select a marker setting. If you choose the **Vertical**, **Track**, or **Slope** buttons, you can drag a marker to the right or left. If you choose the **Horizontal** button, you can drag a marker up or down.

b Move the mouse over the marker (1 or 2) that you want to drag.

The hand cursor with the marker number inside it is displayed when your mouse passes over a marker.

c Drag the marker to where you want it on the signal

As you drag a marker, the bottom of the Signal Browser shows the current position of both markers. Depending on which marker setting you select, some or all of the following fields are displayed — **x1**, **y1**, **x2**, **y2**, **dx**, **dy**, **m**. These fields are also displayed when you print from the Signal Browser, unless you suppress them.

You can also position a marker by typing its **x1** and **x2** or **y1** and **y2** values in the region at the bottom.

Setting Preferences

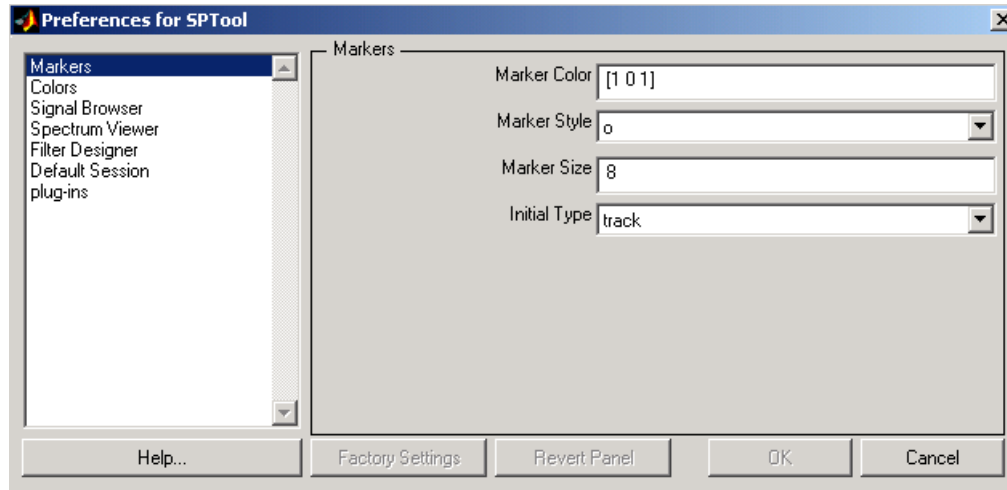
In this section...

Overview of Setting Preferences

Use **File > Preferences** to customize displays and certain parameters for SPTool and its four component GUIs. If you change any preferences, a dialog box displays when you close SPTool asking if you want to save those changes. If you click **Yes**, the new settings are saved on disk and are used when you restart SPTool from the MATLAB workspace.

Note You can set MATLAB preferences that affect the Filter Visualization Tool only from within FVTool by selecting **File > Preferences**. You can set FVTool-specific preferences using **Analysis > Analysis Parameters**.

When you first select **Preferences**, the Preferences dialog box opens with **Markers** selected by default.



Change any marker settings, if desired. To change settings for another category, click its name in the category list to display its settings. Most of the fields are self-explanatory. Details of the Filter Design options are described below.

Summary of Settable Preferences

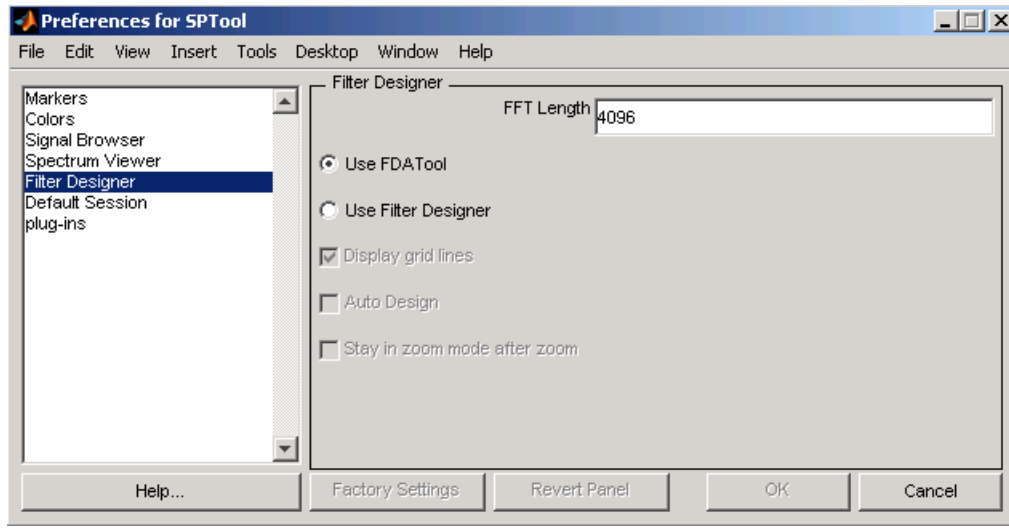
In the Preferences regions, you can

- Select colors and markers for all displays.
- Select colors and line styles for displayed signals.
- Configure labels, and enable/disable markers, panner, and zoom in the Signal Browser.
- Configure display parameters, and enable/disable markers and zoom in the Spectrum Viewer.
- Select whether to use the default FDATool or the Filter Designer to design filters. FDATool is the recommended designer.
- Enable/disable use of a default session file.

- Export filters for use with Control System Toolbox software.
- Enable/disable search for plug-ins at startup.

Setting the Filter Design Tool

The Filter Designer options include radio buttons to select the filter design tool.



FDATool is the default and recommended tool. You cannot change this preference if either FDATool or the Filter Designer is open.

Note Filters in any one SPTool session must be in the same format — either FDATool format or Filter Designer format. You can convert filters from the Filter Designer format to FDATool format, but you cannot convert FDATool filters to Filter Designer format.

When you change the preference from **Use FDATool** to **Use Filter Designer,** a warning message appears indicating that switching will delete current filters. See “Saving and Loading Sessions” on page 8-38 for more information.

When you change the preference from **Use Filter Designer** to **Use FDATool**, a confirmation message appears indicating that switching will convert your filters to FDATool format. See “Saving and Loading Sessions” on page 8-38 for information on this message.

Changes to Filter Designer format are saved only if you save the session.

Using the Filter Designer

In this section...

Why Use the Filter Designer?

Although "FDATool" on page 8-10 is the recommended filter design tool, the following information is provided for users of the Filter Designer, which provides an interactive graphical environment for the design of digital IIR and FIR filters based on specifications that you enter on a magnitude or pole-zero plot.

Filter Types

You can design filters of the following types using the Filter Designer:

- Bandpass
- Lowpass
- Bandstop
- Highpass

FIR Filter Methods

You can use the following filter methods to design FIR filters:

- Equiripple
- Least squares
- Window

IIR Filter Methods

You can use the following filter methods to design IIR filters:

- Butterworth
- Chebyshev Type I
- Chebyshev Type II
- Elliptic

Pole/Zero Editor

You can use the Pole/Zero Editor to design arbitrary FIR and IIR filters by placing and moving poles and zeros on the complex *z*-plane.

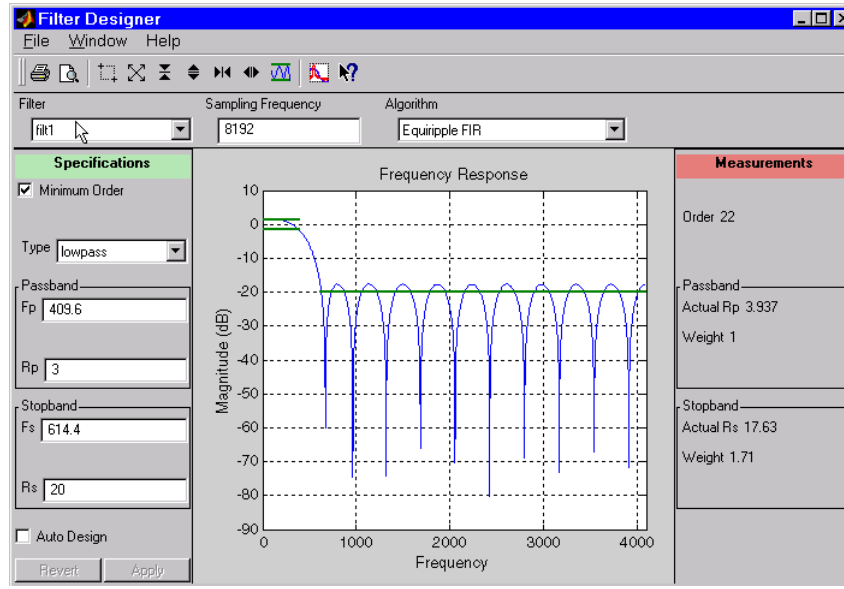
Spectral Overlay Feature

You can also superimpose spectra on a filter's magnitude response to see if the filtering requirements are met.

Opening the Filter Designer

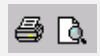
Open the Filter Designer from SPTool by either:

- Clicking **New** in the **Filters** list in SPTool
- Selecting a filter you want to edit from the **Filters** list in SPTool, and then clicking **Edit**



The Filter Designer has the following components:

- A pull-down **Filter** menu for selecting a filter from the list in SPTool
- A **Sampling Frequency** text box
- A pull-down **Algorithm** menu for selecting a filter design method or a pole-zero plot display
- A Specifications area for viewing or modifying a filter's design parameters or pole-zero locations
- A plot display region for graphically adjusting filter magnitude responses or the pole-zero locations
- A Measurements area for viewing the response characteristics and stability of the current filter
- A toolbar with the following buttons

Icon	Description
	Print and print preview

Icon	Description
	Zoom in and out
	Passband view
	Overlay spectrum
	Turn on the What's This help

Accessing Filter Parameters in a Saved Filter

The MATLAB structures created by SPTool have several associated fields, many of which are also MATLAB structures. See the MATLAB documentation for general information about MATLAB structures.

For example, after exporting a filter filt1 to the MATLAB workspace, type

```
filt1
```

to display the fields of the MATLAB filter structure. The `tf`, `Fs`, and `specs` fields of the structure contain the information that describes the filter.

The tf Field: Accessing Filter Coefficients

The `tf` field is a structure containing the transfer function representation of the filter. Use this field to obtain the filter coefficients;

- `filt1.tf.num` contains the numerator coefficients.
- `filt1.tf.den` contains the denominator coefficients.

The vectors contained in these structures represent polynomials in descending powers of *z*. The numerator and denominator polynomials are used to specify the transfer function

$$H(z) = \frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \cdots + b(nb+1)z^{-m}}{a(1) + a(2)z^{-1} + \cdots + a(na+1)z^{-n}}$$

where:

- b is a vector containing the coefficients from the `tf.num` field.
- a is a vector containing the coefficients from the `tf.den` field.
- m is the numerator order.
- n is the denominator order.

You can change the filter representation from the default transfer function to another form by using the `tf2ss` or `tf2zp` functions.

The Fs Field: Accessing Filter Sample Frequency

The `Fs` field contains the sampling frequency of the filter in hertz.

The specs Field: Accessing other Filter Parameters

The `specs` field is a structure containing parameters that you specified for the filter design. The first field, `specs.currentModule`, contains a string representing the most recent design method selected from the Filter Designer's **Algorithm** list before you exported the filter. The possible contents of the `currentModule` field and the corresponding design methods are shown below.

Contents of the currentModule field	Design Method
`fdbutter`	Butterworth IIR
`fdcheby1`	Chebyshev Type I IIR
`fdcheby2`	Chebyshev Type II IIR
`fdellip`	Elliptic IIR
`fdfirls`	Least Squares FIR
`fdkaiser`	Kaiser Window FIR
`fdremez`	Equiripple FIR

Following the `specs.currentModule` field, there may be up to seven additional fields, with labels such as `specs.fdremez`, `specs.fdfirls`, etc. The design specifications for the most recently exported filter are contained

in the field whose label matches the `currentModule` string. For example, if the `specs` structure is

```
filt1.specs

ans
  currentModule: 'fdremez'
  fdremez: [1x1 struct]
```

the filter specifications are contained in the `fdremez` field, which is itself a data structure.

The specifications include the parameter values from the Specifications region of the Filter Designer, such as band edges and filter order. For example, the filter above has the following specifications stored in `filt1.specs.fdremez`:

```
filt1.specs.fdremez

ans =
    setOrderFlag: 0
            type: 3
               f: [0 0.2000 0.3000 0.5000 0.6000 1]
               m: [6x1 double]
              Rp: 0.0100
              Rs: 75
              wt: [3.2371 1 3.2371]
           order: 78
```

Because certain filter parameters are unique to a particular design, this structure has a different set of fields for each filter design.

The table below describes the possible fields associated with the filter design specification field (the `specs` field) that can appear in the exported structure.

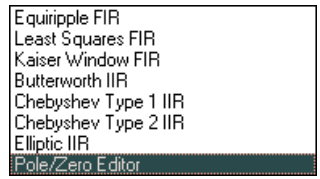
Parameter	Description
`Beta`	Kaiser window ß parameter.
`f`	Contains a vector of band-edge frequencies, normalized so that 1 Hz corresponds to half the sample frequency.

Parameter	Description
`Fpass`	Passband cutoff frequencies. Scalar for lowpass and highpass designs, two-element vector for bandpass and bandstop designs.
`Fstop`	Stopband cutoff frequencies. Scalar for lowpass and highpass designs, two-element vector for bandpass and bandstop designs.
`m`	The response magnitudes corresponding to the band-edge frequencies in `f`.
`order`	Filter order.
`Rp`	Passband ripple (dB)
`Rs`	Stopband attenuation (dB)
`setOrderFlag`	Contains `1` if the filter order was specified manually (i.e., the **Minimum Order** box in the Specifications region was not selected). Contains `0` if the filter order was computed automatically.
`type`	Contains `1` for lowpass, `2` for highpass, `3` for bandpass, or `4` for bandstop.
`w3db`	-3 dB frequency for Butterworth IIR designs.
`wind`	Vector of Kaiser window coefficients.
`Wn`	Cutoff frequency for the Kaiser window FIR filter when `setOrderFlag = 1`.
`wt`	Vector of weights, one weight per frequency band.

Designing a Filter with the Pole/Zero Editor

To design a filter transfer function using the Filter Designer Pole/Zero Editor:

1. Select the `Pole/Zero Editor` option from the **Algorithm** list to open the Pole/Zero Editor in the Filter Designer display.



2. Enter the desired filter gain in the **Gain** edit box.

3. Select a pole or zero (or conjugate pair) by selecting one of the × (pole) or ○ (zero) symbols on the plot.

4. Choose the coordinates to work in by specifying `Polar` or `Rectangular` from the **Coordinates** list.

5. Specify the new location(s) of the selected pole, zero, or conjugate pair by typing values into the **Mag** and **Angle** fields (for angular coordinates) or **X** and **Y** (for rectangular coordinates) fields. Alternatively, position the poles and zeros by dragging the × and ○ symbols.

6. Use the **Conjugate pair** check box to create a conjugate pair from a lone pole or zero, or to break a conjugate pair into two individual poles or zeros.

Design a new filter or edit an existing filter in the same way.

Note Keep the Filter Visualization Tool (FVTool) open while designing a filter with the Pole/Zero Editor. Any changes that you make to the filter transfer function in the Pole/Zero Editor are then simultaneously reflected in the response plots of FVTool.

Positioning Poles and Zeros

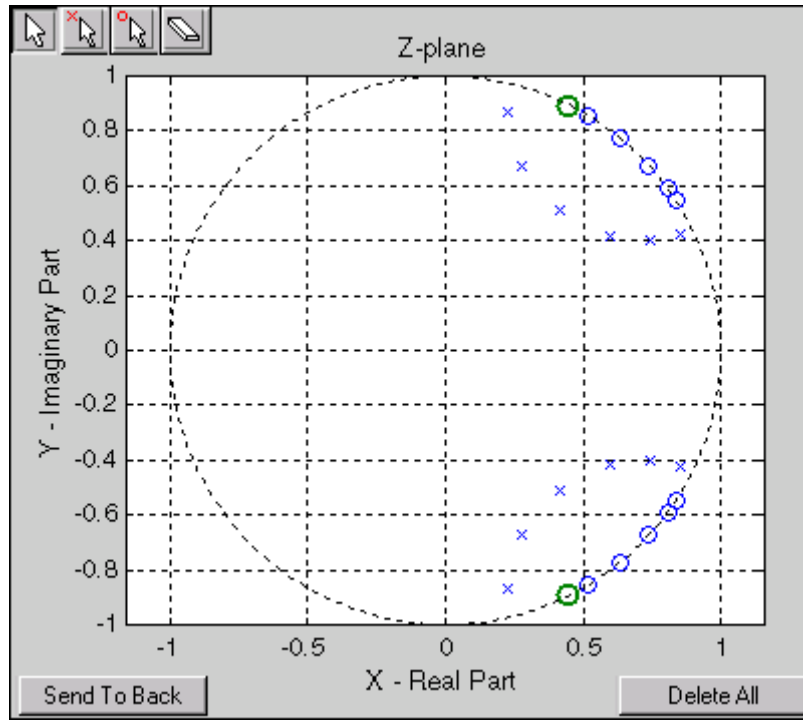
You can use your mouse to move poles and zeros around the pole/zero plot and modify your filter design.

Icon	Description
	Enable moving poles or zeros by dragging on the plot

Icon	Description
	Add pole
	Add zero
	Erase poles or zeros

You can move both members of a conjugate pair simultaneously by manipulating just one of the poles or zeros.

To ungroup conjugates, select the desired pair and clear **Conjugate pair** in the Specifications region on the Filter Designer.



When you place two or more poles (or two or more zeros) directly on top of each other, a number is displayed next to the symbols (on the left for poles, and on

the right for zeros) indicating the number of poles or zeros at that location (e.g.,  for three zeros). This number makes it easy to keep track of all the poles and zeros in the plot area, even when several are superimposed on each other and are not visually differentiable. Note, however, that this number *does not* indicate the *multiplicity* of the poles or zeros to which it is attached.

To detect whether or not a set of poles or zeros are truly multiples, use the zoom tools to magnify the region around the poles or zeros in question. Because numerical limitations usually prevent any set of poles or zeros from sharing *exactly* the same value, at a high enough zoom level even truly multiple poles or zeros appear distinct from each other.

A common way to assess whether a particular group of poles or zeros contains multiples is by comparing the mutual proximity of the group members against a selected threshold value. As an example, the `residuez` function defines a pole or zero as being a multiple of another pole or zero if the absolute distance separating them is less than 0.1% of the larger pole or zero's magnitude.

Redesigning a Filter Using the Magnitude Plot

After designing a filter in the Filter Designer, you can redesign it by dragging the specification lines on the magnitude plot. Use the specification lines to change passband ripple, stopband attenuation, and edge frequencies.

In the following example, create a Chebyshev filter and modify it by dragging the specification lines:

1. Select `Chebyshev Type I IIR` from the **Algorithm** menu.
2. Select `highpass` from the **Type** menu.
3. Type `2000` in the **Sampling Frequency** field.
4. Set the following parameters:
   - **Fp** = `800`
   - **Fs** = `700`
   - **Rp** = `2.5`
   - **Rs** = `35`

5 Select **Minimum Order** so the Filter Designer can calculate the lowest filter order that produces the desired characteristics.

6 Click **Apply** to compute the filter and update the response plot.

7 Position the cursor over the horizontal filter specification line for the stopband. This is the first (leftmost) horizontal specification line you see.

The cursor changes to the up/down drag indicator.

8 Drag the line until the **Rs** (stopband attenuation) field reads 100.

Note The **Order** value in the Measurements region changes because a higher filter order is needed to meet the new specifications.

9

Function Reference

Specialized Operations (p. 9-17)	Plotting, vector manipulation, uniform encoding/decoding, etc.
GUIs (p. 9-18)	GUIs for data visualization, spectral analysis, filter design, and window design

Signal Processing Functions in MATLAB

`conv`	Convolution and polynomial multiplication
`conv2`	2-D convolution
`corrcoef`	Correlation coefficients
`cov`	Covariance matrix
`cplxpair`	Sort complex numbers into complex conjugate pairs
`deconv`	Deconvolution and polynomial division
`fft`	Discrete Fourier transform
`fft2`	2-D discrete Fourier transform
`fftshift`	Shift zero-frequency component to center of spectrum
`filter2`	2-D digital filter
`freqspace`	Frequency spacing for frequency response
`ifft`	Inverse discrete Fourier transform
`ifft2`	2-D inverse discrete Fourier transform
`unwrap`	Correct phase angles to produce smoother phase plots

Digital Filters

FIR Filter Design (p. 9-4)	Design functions for FIR filters
Communications Filters (p. 9-5)	Design functions used in communications
IIR Digital Filter Design (p. 9-5)	Design functions for IIR filters
IIR Filter Order Estimation (p. 9-5)	Estimation functions for IIR filters
Filter Analysis (p. 9-5)	Functions for analyzing filters
Filter Implementation (p. 9-6)	Functions for implementing filters

Filter Specification Objects – Response Types (p. 9-7)	Functions for creating filter specification objects
Filter Specification Objects – Design Methods (p. 9-7)	Functions for designing filter specification objects

FIR Filter Design

`cfirpm`	Complex and nonlinear-phase equiripple FIR filter design
`fir1`	Window-based finite impulse response filter design
`fir2`	Frequency sampling-based finite impulse response filter design
`fircls`	Constrained least square, FIR multiband filter design
`fircls1`	Constrained least square, lowpass and highpass, linear phase, FIR filter design
`firls`	Least square linear-phase FIR filter design
`firpm`	Parks-McClellan optimal FIR filter design
`firpmord`	Parks-McClellan optimal FIR filter order estimation
`intfilt`	Interpolation FIR filter design
`kaiserord`	Kaiser window FIR filter design estimation parameters
`sgolay`	Savitzky-Golay filter design

Communications Filters

`firrcos`	Raised cosine FIR filter design
`gaussfir`	Gaussian FIR pulse-shaping filter

IIR Digital Filter Design

`butter`	Butterworth filter design
`cheby1`	Chebyshev Type I filter design (passband ripple)
`cheby2`	Chebyshev Type II filter design (stopband ripple)
`ellip`	Elliptic filter design
`maxflat`	Generalized digital Butterworth filter design
`yulewalk`	Recursive digital filter design

IIR Filter Order Estimation

`buttord`	Butterworth filter order and cutoff frequency
`cheb1ord`	Chebyshev Type I filter order
`cheb2ord`	Chebyshev Type II filter order
`ellipord`	Minimum order for elliptic filters

Filter Analysis

`abs`	Absolute value (magnitude)
`angle`	Phase angle
`filternorm`	2-norm or infinity-norm of digital filter

freqz	Frequency response of digital filter
fvtool	Open Filter Visualization Tool
grpdelay	Average filter delay (group delay)
impz	Impulse response of digital filter
phasedelay	Phase delay of digital filter
phasez	Phase response of digital filter
stepz	Step response of digital filter
zerophase	Zero-phase response of digital filter
zplane	Zero-pole plot

Filter Implementation

cconv	Modulo-N circular convolution
convmtx	Convolution matrix
fftfilt	FFT-based FIR filtering using overlap-add method
filter	Filter data with recursive (IIR) or nonrecursive (FIR) filter
filtfilt	Zero-phase digital filtering
filtic	Initial conditions for transposed direct-form II filter implementation
latcfilt	Lattice and lattice-ladder filter implementation
medfilt1	1-D median filtering
sgolayfilt	Savitzky-Golay filtering
sosfilt	Second-order (biquadratic) IIR digital filtering
upfirdn	Upsample, apply FIR filter, and downsample

Filter Specification Objects – Response Types

`fdesign`	Filter specification object
`fdesign.arbmag`	Arbitrary response magnitude filter specification object
`fdesign.bandpass`	Bandpass filter specification object
`fdesign.bandstop`	Bandstop filter specification object
`fdesign.differentiator`	Differentiator filter specification object
`fdesign.highpass`	Highpass filter specification object
`fdesign.hilbert`	Hilbert filter specification object
`fdesign.lowpass`	Lowpass filter specification
`fdesign.pulseshaping`	Pulse-shaping filter specification object

Filter Specification Objects – Design Methods

`design`	Apply design method to specification object
`designmethods`	Methods available for designing filter from specification object
`equiripple`	Equiripple single-rate FIR filter from specification object
`kaiserwin`	Kaiser window filter from specification object
`window (filter design method)`	FIR filter using windowed impulse response

Analog Filters

Analog Lowpass Filter Prototypes

`besselap`	Bessel analog lowpass filter prototype
`buttap`	Butterworth filter prototype
`cheb1ap`	Chebyshev Type I analog lowpass filter prototype
`cheb2ap`	Chebyshev Type II analog lowpass filter prototype
`ellipap`	Elliptic analog lowpass filter prototype

Analog Filter Design

`besself`	Bessel analog filter design
`butter`	Butterworth filter design
`cheby1`	Chebyshev Type I filter design (passband ripple)

`cheby2`	Chebyshev Type II filter design (stopband ripple)
`ellip`	Elliptic filter design

Filter Analysis

`abs`	Absolute value (magnitude)
`freqs`	Frequency response of analog filters

Analog Filter Transformation

`lp2bp`	Transform lowpass analog filters to bandpass
`lp2bs`	Transform lowpass analog filters to bandstop
`lp2hp`	Transform lowpass analog filters to highpass
`lp2lp`	Change cutoff frequency for lowpass analog filter

Filter Discretization

`bilinear`	Bilinear transformation method for analog-to-digital filter conversion
`impinvar`	Impulse invariance method for analog-to-digital filter conversion

Linear Systems

`latc2tf`	Convert lattice filter parameters to transfer function form
`polyscale`	Scale roots of polynomial
`polystab`	Stabilize polynomial
`residuez`	*z*-transform partial-fraction expansion
`sos2ss`	Convert digital filter second-order section parameters to state-space form
`sos2tf`	Convert digital filter second-order section data to transfer function form
`sos2zp`	Convert digital filter second-order section parameters to zero-pole-gain form
`ss2sos`	Convert digital filter state-space parameters to second-order sections form
`ss2tf`	Convert state-space filter parameters to transfer function form
`ss2zp`	Convert state-space filter parameters to zero-pole-gain form
`tf2latc`	Convert transfer function filter parameters to lattice filter form
`tf2sos`	Convert digital filter transfer function data to second-order sections form
`tf2ss`	Convert transfer function filter parameters to state-space form
`tf2zp`	Convert transfer function filter parameters to zero-pole-gain form

`tf2zpk`	Convert transfer function filter parameters to zero-pole-gain form
`zp2sos`	Convert zero-pole-gain filter parameters to second-order sections form
`zp2ss`	Convert zero-pole-gain filter parameters to state-space form
`zp2tf`	Convert zero-pole-gain filter parameters to transfer function form

Windows

`barthannwin`	Modified Bartlett-Hann window
`bartlett`	Bartlett window
`blackman`	Blackman window
`blackmanharris`	Minimum 4-term Blackman-Harris window
`bohmanwin`	Bohman window
`chebwin`	Chebyshev window
`dpss`	Discrete prolate spheroidal sequences (Slepian sequences)
`dpssclear`	Remove discrete prolate spheroidal sequences from database
`dpssdir`	Discrete prolate spheroidal sequences database directory
`dpssload`	Load discrete prolate spheroidal sequences from database
`dpsssave`	Save discrete prolate spheroidal sequences in database
`flattopwin`	Flat Top weighted window

`gausswin`	Gaussian window
`hamming`	Hamming window
`hann`	Hann (Hanning) window
`kaiser`	Kaiser window
`nuttallwin`	Nuttall-defined minimum 4-term Blackman-Harris window
`parzenwin`	Parzen (de la Valle-Poussin) window
`rectwin`	Rectangular window
`taylorwin`	Taylor window
`triang`	Triangular window
`tukeywin`	Tukey (tapered cosine) window
`window`	Window function gateway
`wvtool`	Open Window Visualization Tool

Transforms

`bitrevorder`	Permute data into bit-reversed order
`czt`	Chirp *z*-transform
`dct`	Discrete cosine transform (DCT)
`dftmtx`	Discrete Fourier transform matrix
`digitrevorder`	Permute input into digit-reversed order
`fwht`	Fast Walsh–Hadamard transform
`goertzel`	Discrete Fourier transform using second-order Goertzel algorithm
`hilbert`	Discrete-time analytic signal using Hilbert transform

`idct`	Inverse discrete cosine transform
`ifwht`	Inverse Fast Walsh–Hadamard transform

Cepstral Analysis

`cceps`	Complex cepstral analysis
`icceps`	Inverse complex cepstrum
`rceps`	Real cepstrum and minimum phase reconstruction

Statistical Signal Processing

`corrmtx`	Data matrix for autocorrelation matrix estimation
`cpsd`	Cross power spectral density
`dspdata`	DSP data parameter information
`mscohere`	Magnitude squared coherence
`pburg`	PSD using Burg method
`pcov`	PSD using covariance method
`peig`	Pseudospectrum using eigenvector method
`periodogram`	PSD using periodogram
`pmcov`	PSD using modified covariance method
`pmtm`	PSD using multitaper method (MTM)

pmusic	Pseudospectrum using MUSIC algorithm
pwelch	PSD using Welch's method
pyulear	PSD using Yule-Walker AR method
rooteig	Frequency and power content using eigenvector method
rootmusic	Frequency and power content using root MUSIC algorithm
spectrogram	Spectrogram using short-time Fourier transform
spectrum	Spectral estimation
tfestimate	Transfer function estimate
xcorr	Cross-correlation
xcorr2	2–D cross-correlation
xcov	Cross-covariance

Parametric Modeling

arburg	Estimate AR model parameters using Burg method
arcov	Estimate AR model parameters using covariance method
armcov	Estimate AR model parameters using modified covariance method
aryule	Estimate AR model parameters using Yule-Walker method
invfreqs	Identify continuous-time filter parameters from frequency response data

`invfreqz`	Identify discrete-time filter parameters from frequency response data
`prony`	Prony's method for time domain IIR filter design
`stmcb`	Compute linear model using Steiglitz-McBride iteration

Linear Prediction

`ac2poly`	Convert autocorrelation sequence to prediction polynomial
`ac2rc`	Convert autocorrelation sequence to reflection coefficients
`is2rc`	Convert inverse sine parameters to reflection coefficients
`lar2rc`	Convert log area ratio parameters to reflection coefficients
`levinson`	Levinson-Durbin recursion
`lpc`	Linear prediction filter coefficients
`lsf2poly`	Convert line spectral frequencies to prediction filter coefficients
`poly2ac`	Convert prediction filter polynomial to autocorrelation sequence
`poly2lsf`	Convert prediction filter coefficients to line spectral frequencies
`poly2rc`	Convert prediction filter polynomial to reflection coefficients
`rc2ac`	Convert reflection coefficients to autocorrelation sequence

`rc2is`	Convert reflection coefficients to inverse sine parameters
`rc2lar`	Convert reflection coefficients to log area ratio parameters
`rc2poly`	Convert reflection coefficients to prediction filter polynomial
`rlevinson`	Reverse Levinson-Durbin recursion
`schurrc`	Compute reflection coefficients from autocorrelation sequence

Multirate Signal Processing

`decimate`	Decimation — decrease sampling rate
`downsample`	Decrease sampling rate by integer factor
`interp`	Interpolation — increase sampling rate by integer factor
`resample`	Change sampling rate by rational factor
`upfirdn`	Upsample, apply FIR filter, and downsample
`upsample`	Increase sampling rate by integer factor

Waveform Generation

`chirp`	Swept-frequency cosine
`diric`	Dirichlet or periodic sinc function

`gauspuls`	Gaussian-modulated sinusoidal pulse
`gmonopuls`	Gaussian monopulse
`pulstran`	Pulse train
`rectpuls`	Sampled aperiodic rectangle
`sawtooth`	Sawtooth or triangle wave
`sinc`	Sinc
`square`	Square wave
`tripuls`	Sampled aperiodic triangle
`vco`	Voltage controlled oscillator

Specialized Operations

`buffer`	Buffer signal vector into matrix of data frames
`cell2sos`	Convert second-order sections cell array to matrix
`db2mag`	Convert decibels (dB) to magnitude
`db2pow`	Convert decibels (dB) to power
`demod`	Demodulation for communications simulation
`eqtflength`	Equalize lengths of transfer function's numerator and denominator
`findpeaks`	Find local maxima
`mag2db`	Convert magnitude to decibels (dB)
`marcumq`	Generalized Marcum Q function
`modulate`	Modulation for communications simulation

`pow2db`	Convert power to decibels (dB)
`seqperiod`	Compute period of sequence
`sos2cell`	Convert second-order sections matrix to cell array
`strips`	Strip plot
`udecode`	Decode 2-level quantized integer inputs to floating-point outputs
`uencode`	Quantize and encode floating-point inputs to integer outputs

GUIs

`fdatool`	Open Filter Design and Analysis Tool
`fvtool`	Open Filter Visualization Tool
`sptool`	Open interactive digital signal processing tool
`wintool`	Open Window Design and Analysis Tool
`wvtool`	Open Window Visualization Tool

10

Functions — Alphabetical List

abs

Purpose

Absolute value (magnitude)

Description

`abs` is a MATLAB function.

Signal-Specific Example

Calculate the magnitude of the FFT of a sequence.

```
t = (0:99)/100;                          % Time vector
x = sin(2*pi*15*t) + sin(2*pi*40*t);     % Signal
y = fft(x);                              % Compute DFT of x
m = abs(y);                              % Magnitude
```

Plot the magnitude:

```
f = (0:length(y)-1)'/length(y)*100;      % Frequency vector
plot(f,m)
```

Purpose

Convert autocorrelation sequence to prediction polynomial

Syntax

```
a = ac2poly(r)
[a,efinal] = ac2poly(r)
```

Description

`a = ac2poly(r)` finds the linear prediction, FIR filter polynomial `a` corresponding to the autocorrelation sequence `r`. `a` is the same length as `r`, and `a(1) = 1`. The prediction filter polynomial represents the coefficients of the prediction filter whose output produces a signal whose autocorrelation sequence is approximately the same as the given autocorrelation sequence `r`.

`[a,efinal] = ac2poly(r)` returns the final prediction error `efinal`, determined by running the filter for `length(r)` steps.

Remarks

You can apply this function to real or complex data.

Examples

Consider the autocorrelation sequence:

```
r = [5.0000 -1.5450 -3.9547 3.9331 1.4681 -4.7500];
```

The corresponding prediction filter polynomial is

```
[a,efinal] = ac2poly(r)
a =
    1.0000  0.6147  0.9898  0.0004  0.0034 -0.0077
efinal =
    0.1791
```

References

[1] Kay, S.M. *Modern Spectral Estimation.* Englewood Cliffs, NJ: Prentice-Hall, 1988.

See Also

`ac2rc`, `poly2ac`, `rc2poly`

ac2rc

Purpose

Convert autocorrelation sequence to reflection coefficients

Syntax

```
[k,r0] = ac2rc(r)
```

Description

`[k,r0] = ac2rc(r)` finds the reflection coefficients `k` corresponding to the autocorrelation sequence `r`. `r0` contains the zero-lag autocorrelation. If `r` is a matrix where the columns are separate channels of autocorrelation sequences, `r0` contains the zero-lag autocorrelation coefficient for each channel. These reflection coefficients can be used to specify the lattice prediction filter that produces a sequence with approximately the same autocorrelation sequence as the given sequence `r`.

Remarks

You can apply this function to real or complex data.

References

[1] Kay, S.M. *Modern Spectral Estimation.* Englewood Cliffs, NJ: Prentice-Hall, 1988.

See Also

`ac2poly`, `poly2rc`, `rc2ac`

Purpose

Phase angle

Description

`angle` is a MATLAB function.

Signal-specific Example

Calculate the phase of the FFT of a sequence.

```
t = (0:99)/100;                          % Time vector
x = sin(2*pi*15*t) + sin(2*pi*40*t);     % Signal
y = fft(x);                              % Compute DFT of x
p = unwrap(angle(y));                    % Phase
```

Plot the phase:

```
f = (0:length(y)-1)'/length(y)*100;      % Frequency vector
plot(f,p)
```

arburg

Purpose

Estimate AR model parameters using Burg method

Syntax

```
a = arburg(x,p)
[a,e] = arburg(x,p)
[a,e,k] = arburg(x,p)
```

Description

`a = arburg(x,p)` uses the Burg method to fit a `p`th order autoregressive (AR) model to the input signal, `x`, by minimizing (least squares) the forward and backward prediction errors while constraining the AR parameters to satisfy the Levinson-Durbin recursion. `x` is assumed to be the output of an AR system driven by white noise. Vector `a` contains the normalized estimate of the AR system parameters, $A(z)$, in descending powers of z.

$$H(z) = \frac{\sqrt{e}}{A(z)} = \frac{\sqrt{e}}{1 + a_2 z^{-1} + \ldots + a_{(p+1)} z^{-p}}$$

Since the method characterizes the input data using an all-pole model, the correct choice of the model order `p` is important.

`[a,e] = arburg(x,p)` returns the variance estimate, `e`, of the white noise input to the AR model.

`[a,e,k] = arburg(x,p)` returns a vector, `k`, of reflection coefficients.

See Also

`arcov`, `armcov`, `aryule`, `lpc`, `pburg`, `prony`

Purpose

Estimate AR model parameters using covariance method

Syntax

```
a = arcov(x,p)
[a,e] = arcov(x,p)
```

Description

`a = arcov(x,p)` uses the covariance method to fit a `p`th order autoregressive (AR) model to the input signal, `x`, which is assumed to be the output of an AR system driven by white noise. This method minimizes the forward prediction error in the least-squares sense. Vector `a` contains the normalized estimate of the AR system parameters, $A(z)$, in descending powers of z.

$$H(z) = \frac{\sqrt{e}}{A(z)} = \frac{\sqrt{e}}{1 + a_2 z^{-1} + \ldots + a_{(p+1)} z^{-p}}$$

Because the method characterizes the input data using an all-pole model, the correct choice of the model order `p` is important.

`[a,e] = arcov(x,p)` returns the variance estimate, `e`, of the white noise input to the AR model.

See Also

`arburg`, `armcov`, `aryule`, `lpc`, `pcov`, `prony`

Purpose

Estimate AR model parameters using modified covariance method

Syntax

```
a = armcov(x,p)
[a,e] = armcov(x,p)
```

Description

`a = armcov(x,p)` uses the modified covariance method to fit a `p`th order autoregressive (AR) model to the input signal, `x`, which is assumed to be the output of an AR system driven by white noise. This method minimizes the forward and backward prediction errors in the least-squares sense. Vector `a` contains the normalized estimate of the AR system parameters, $A(z)$, in descending powers of z.

$$H(z) = \frac{\sqrt{e}}{A(z)} = \frac{\sqrt{e}}{1 + a_2 z^{-1} + \ldots + a_{(p+1)} z^{-p}}$$

Because the method characterizes the input data using an all-pole model, the correct choice of the model order `p` is important.

`[a,e] = armcov(x,p)` returns the variance estimate, `e`, of the white noise input to the AR model.

See Also

`arburg`, `arcov`, `aryule`, `lpc`, `pmcov`, `prony`

Purpose

Estimate AR model parameters using Yule-Walker method

Syntax

```
a = aryule(x,p)
[a,e] = aryule(x,p)
[a,e,k] = aryule(x,p)
```

Description

`a = aryule(x,p)` uses the Yule-Walker method, also called the autocorrelation method, to fit a `p`th order autoregressive (AR) model to the windowed input signal, `x`, by minimizing the forward prediction error in the least-squares sense. This formulation leads to the Yule-Walker equations, which are solved by the Levinson-Durbin recursion. `x` is assumed to be the output of an AR system driven by white noise. Vector `a` contains the normalized estimate of the AR system parameters, $A(z)$, in descending powers of z.

$$H(z) = \frac{\sqrt{e}}{A(z)} = \frac{\sqrt{e}}{1 + a_2 z^{-1} + \ldots + a_{(p+1)} z^{-p}}$$

Because the method characterizes the input data using an all-pole model, the correct choice of the model order `p` is important.

`[a,e] = aryule(x,p)` returns the variance estimate, `e`, of the white noise input to the AR model.

`[a,e,k] = aryule(x,p)` returns a vector, `k`, of reflection coefficients.

See Also

`arburg`, `arcov`, `armcov`, `lpc`, `prony`, `pyulear`

barthannwin

Purpose

Modified Bartlett-Hann window

Syntax

```
w = barthannwin(L)
```

Description

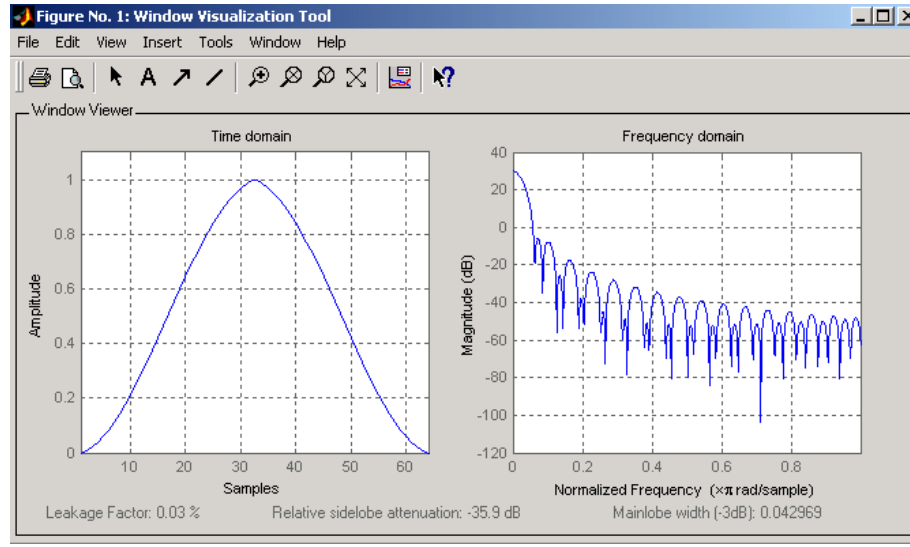
`w = barthannwin(L)` returns an `L`-point modified Bartlett-Hann window in the column vector `w`. Like Bartlett, Hann, and Hamming windows, this window has a mainlobe at the origin and asymptotically decaying sidelobes on both sides. It is a linear combination of weighted Bartlett and Hann windows with near sidelobes lower than both Bartlett and Hann and with far sidelobes lower than both Bartlett and Hamming windows. The mainlobe width of the modified Bartlett-Hann window is not increased relative to either Bartlett or Hann window mainlobes.

Note The Hann window is also called the Hanning window.

Examples

Create a 64-point Bartlett-Hann window and display the result using WVTool:

```
L=64;
wvtool(barthannwin(L))
```



Algorithm

The equation for computing the coefficients of a Modified Bartlett-Hanning window is

$$w(n) = 0.62 - 0.48\left|\left(\frac{n}{N} - 0.5\right)\right| + 0.38\cos\left(2\pi\left(\frac{n}{N} - 0.5\right)\right)$$

where $0 \le n \le N$ and the window length is $L = N + 1$.

References

[1] Ha, Y.H., and J.A. Pearce. "A New Window and Comparison to Standard Windows." *IEEE® Transactions on Acoustics, Speech, and Signal Processing.* Vol. 37, No. 2, (February 1999). pp. 298-301.

[2] Oppenheim, A.V., and R.W. Schafer. *Discrete-Time Signal Processing.* Upper Saddle River, NJ: Prentice-Hall, 1999, p. 468.

See Also

`bartlett`, `blackmanharris`, `bohmanwin`, `nuttallwin`, `parzenwin`, `rectwin`, `triang`, `window`, `wintool`, `wvtool`

bartlett

Purpose

Bartlett window

Syntax

```
w = bartlett(L)
```

Description

`w = bartlett(L)` returns an `L`-point Bartlett window in the column vector `w`, where `L` must be a positive integer. The coefficients of a Bartlett window are computed as follows:

$$w(n) = \begin{cases} \dfrac{2n}{N}, & 0 \le n \le \dfrac{N}{2} \\ 2 - \dfrac{2n}{N}, & \dfrac{N}{2} \le n \le N \end{cases}$$

The window length $L = N + 1$.

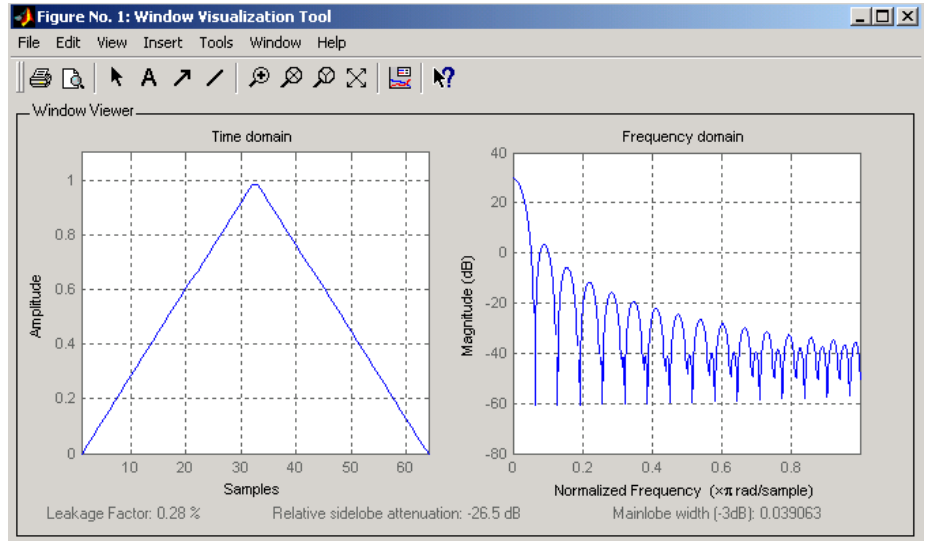
The Bartlett window is very similar to a triangular window as returned by the `triang` function. The Bartlett window always ends with zeros at samples 1 and `n`, however, while the triangular window is nonzero at those points. For `L` odd, the center `L-2` points of `bartlett(L)` are equivalent to `triang(L-2)`.

Note If you specify a one-point window (set `L=1`), the value `1` is returned.

Examples

Create a 64-point Bartlett window and display the result using WVTool:

```
L=64;
wvtool(bartlett(L))
```



References

[1] Oppenheim, A.V., and R.W. Schafer. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1999, pp. 468-471.

See Also

`barthannwin`, `blackmanharris`, `bohmanwin`, `nuttallwin`, `parzenwin`, `rectwin`, `triang`, `window`, `wintool`, `wvtool`

besselap

Purpose

Bessel analog lowpass filter prototype

Syntax

```
[z,p,k] = besselap(n)
```

Description

`[z,p,k] = besselap(n)` returns the poles and gain of an order `n` Bessel analog lowpass filter prototype. `n` must be less than or equal to 25. The function returns the poles in the length `n` column vector `p` and the gain in scalar `k`. `z` is an empty matrix because there are no zeros. The transfer function is

$$H(s) = \frac{k}{(s-p(1))(s-p(2))\cdots(s-p(n))}$$

`besselap` normalizes the poles and gain so that at low frequency and high frequency the Bessel prototype is asymptotically equivalent to the Butterworth prototype of the same order [1]. The magnitude of the filter is less than $\sqrt{1/2}$ at the unity cutoff frequency $\Omega_c = 1$.

Analog Bessel filters are characterized by a group delay that is maximally flat at zero frequency and almost constant throughout the passband. The group delay at zero frequency is

$$\left(\frac{(2n)!}{2^n n!}\right)^{1/n}$$

Algorithm

`besselap` finds the filter roots from a lookup table constructed using Symbolic Math Toolbox™ software.

References

[1] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing.* Englewood Cliffs, NJ: Prentice-Hall, 1975. Pgs. 228-230.

See Also

`besself`, `buttap`, `cheb1ap`, `cheb2ap`, `ellipap`

Also see the Symbolic Math Toolbox documentation.

Purpose

Bessel analog filter design

Syntax

```
[b,a] = besself(n,Wo)
[z,p,k] = besself(...)
[A,B,C,D] = besself(...)
```

Description

`besself` designs lowpass, analog Bessel filters, which are characterized by almost constant group delay across the entire passband, thus preserving the wave shape of filtered signals in the passband. `besself` does not support the design of digital Bessel filters.

`[b,a] = besself(n,Wo)` designs an order `n` lowpass analog Bessel filter, where Wo is the frequency up to which the filter's group delay is approximately constant. Larger values of the filter order (`n`) produce a group delay that better approximates a constant up to frequency `Wo`.

`besself` returns the filter coefficients in the length `n+1` row vectors `b` and `a`, with coefficients in descending powers of *s,* derived from this transfer function:

$$H(s) = \frac{B(s)}{A(s)} = \frac{b(1)s^n + b(2)s^{n-1} + \cdots + b(n+1)}{s^n + a(2)s^{n-1} + \cdots + a(n+1)}$$

`[z,p,k] = besself(...)` returns the zeros and poles in length `n` or `2*n` column vectors `z` and `p` and the gain in the scalar `k`.

`[A,B,C,D] = besself(...)` returns the filter design in state-space form, where `A`, `B`, `C`, and `D` are

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

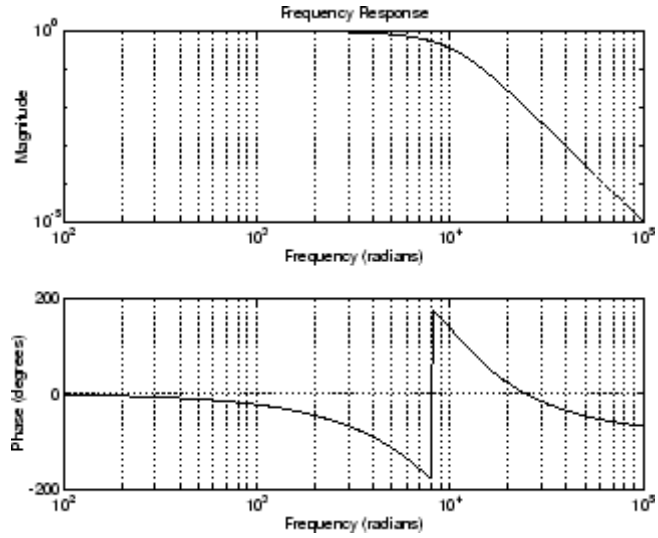
and u is the input, x is the state vector, and y is the output.

Examples

Design a fifth-order analog lowpass Bessel filter with an approximate constant group delay up to 10,000 rad/s and plot the frequency response of the filter using `freqs`:

```
[b,a] = besself(5,10000);
```

```
freqs(b,a)                    % Plot frequency response
```



Limitations

Lowpass Bessel filters have a monotonically decreasing magnitude response, as do lowpass Butterworth filters. Compared to the Butterworth, Chebyshev, and elliptic filters, the Bessel filter has the slowest rolloff and requires the highest order to meet an attenuation specification.

For high order filters, the state-space form is the most numerically accurate, followed by the zero-pole-gain form. The transfer function coefficient form is the least accurate; numerical problems can arise for filter orders as low as 15.

Algorithm

`besself` performs a four-step algorithm:

1. It finds lowpass analog prototype poles, zeros, and gain using the `besselap` function.
2. It converts the poles, zeros, and gain into state-space form.

3 It transforms the lowpass prototype into a lowpass filter that meets the design specifications.

4 It converts the state-space filter back to transfer function or zero-pole-gain form, as required.

See Also

`besselap`, `butter`, `cheby1`, `cheby2`, `ellip`

Purpose

Bilinear transformation method for analog-to-digital filter conversion

Syntax

```
[zd,pd,kd] = bilinear(z,p,k,fs)
[zd,pd,kd] = bilinear(z,p,k,fs,fp)
[numd,dend] = bilinear(num,den,fs)
[numd,dend] = bilinear(num,den,fs,fp)
[Ad,Bd,Cd,Dd] = bilinear(A,B,C,D,fs)
[Ad,Bd,Cd,Dd] = bilinear(A,B,C,D,fs,fp)
```

Description

The *bilinear transformation* is a mathematical mapping of variables. In digital filtering, it is a standard method of mapping the *s* or analog plane into the *z* or digital plane. It transforms analog filters, designed using classical filter design techniques, into their discrete equivalents.

The bilinear transformation maps the *s*-plane into the *z*-plane by

$$H(z) = H(s)\Big|_{s = 2f_s\frac{z-1}{z+1}}$$

This transformation maps the $j\Omega$ axis (from $\Omega = -\infty$ to $+\infty$) repeatedly around the unit circle ($e^{j\omega}$, from $\omega = -\pi$ to π) by

$$\omega = 2\tan^{-1}\left(\frac{\Omega}{2f_s}\right)$$

`bilinear` can accept an optional parameter `Fp` that specifies prewarping. `fp`, in hertz, indicates a "match" frequency, that is, a frequency for which the frequency responses before and after mapping match exactly. In prewarped mode, the bilinear transformation maps the *s*-plane into the *z*-plane with

$$H(z) = H(s)\Big|_{s = \frac{2\pi f_p}{\tan\left(\pi\frac{f_p}{f_s}\right)}\frac{(z-1)}{(z+1)}}$$

With the prewarping option, `bilinear` maps the $j\Omega$ axis (from $\Omega = -\infty$ to $+\infty$) repeatedly around the unit circle ($e^{j\omega}$, from $\omega = -\pi$ to π) by

$$\omega = 2\tan^{-1}\left(\frac{\Omega\tan\left(\pi\frac{f_p}{f_s}\right)}{2\pi f_p}\right)$$

In prewarped mode, `bilinear` matches the frequency $2\pi f_p$ (in radians per second) in the s-plane to the normalized frequency $2\pi f_p/f_s$ (in radians per second) in the z-plane.

The `bilinear` function works with three different linear system representations: zero-pole-gain, transfer function, and state-space form.

Zero-Pole-Gain

`[zd,pd,kd] = bilinear(z,p,k,fs)` and

`[zd,pd,kd] = bilinear(z,p,k,fs,fp)` convert the s-domain transfer function specified by `z`, `p`, and `k` to a discrete equivalent. Inputs `z` and `p` are column vectors containing the zeros and poles, `k` is a scalar gain, and `fs` is the sampling frequency in hertz. `bilinear` returns the discrete equivalent in column vectors `zd` and `pd` and scalar `kd`. The optional match frequency, `fp` is in hertz and is used for prewarping.

Transfer Function

`[numd,dend] = bilinear(num,den,fs)` and

`[numd,dend] = bilinear(num,den,fs,fp)` convert an s-domain transfer function given by `num` and `den` to a discrete equivalent. Row vectors `num` and `den` specify the coefficients of the numerator and denominator, respectively, in descending powers of s.

$$\frac{num(s)}{den(s)} = \frac{num(1)s^n + \cdots + num(n)s + num(n+1)}{den(1)s^m + \cdots + den(m)s + den(m+1)}$$

`fs` is the sampling frequency in hertz. `bilinear` returns the discrete equivalent in row vectors `numd` and `dend` in descending powers of z (ascending powers of z^{-1}). `fp` is the optional match frequency, in hertz, for prewarping.

State-Space

`[Ad,Bd,Cd,Dd] = bilinear(A,B,C,D,fs)` and

`[Ad,Bd,Cd,Dd] = bilinear(A,B,C,D,fs,fp)` convert the continuous-time state-space system in matrices `A`, `B`, `C`, `D`

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

to the discrete-time system:

$$x[n+1] = A_d x[n] + B_d u[n]$$
$$y[n] = C_d x[n] + D_d u[n]$$

`fs` is the sampling frequency in hertz. `bilinear` returns the discrete equivalent in matrices `Ad`, `Bd`, `Cd`, `Dd`. The optional match frequency, `fp` is in hertz and is used for prewarping.

Algorithm

`bilinear` uses one of two algorithms depending on the format of the input linear system you supply. One algorithm works on the zero-pole-gain format and the other on the state-space format. For transfer function representations, `bilinear` converts to state-space form, performs the transformation, and converts the resulting state-space system back to transfer function form.

Zero-Pole-Gain Algorithm

For a system in zero-pole-gain form, `bilinear` performs four steps:

1. If `fp` is present, it prewarps:

   ```
   fp = 2*pi*fp;
   fs = fp/tan(fp/fs/2)
   ```

   otherwise, `fs = 2*fs`.

2. It strips any zeros at $\pm\infty$ using

```
z = z(finite(z));
```

3 It transforms the zeros, poles, and gain using

```
pd = (1+p/fs)./(1-p/fs);    % Do bilinear transformation
zd = (1+z/fs)./(1-z/fs);
kd = real(k*prod(fs-z)./prod(fs-p));
```

4 It adds extra zeros at -1 so the resulting system has equivalent numerator and denominator order.

State-Space Algorithm

For a system in state-space form, `bilinear` performs two steps:

1 If `fp` is present, `k = 2*pi*fp/tan(pi*fp/fs)`; else `k = 2*fs`.

2 It computes `Ad`, `Bd`, `Cd`, and `Dd` in terms of `A`, `B`, `C`, and `D` using

$$A_d = \left(I + \left(\frac{1}{k}\right)A\right)\left(I - \left(\frac{1}{k}\right)A\right)^{-1}$$

$$B_d = \frac{2k}{r}\left(I - \left(\frac{1}{k}\right)A\right)^{-1}B$$

$$C_d = rC\left(I - \left(\frac{1}{k}\right)A\right)^{-1}$$

$$D_d = \left(\frac{1}{k}\right)C\left(I - \left(\frac{1}{k}\right)A\right)^{-1}B + D$$

`bilinear` implements these relations using conventional MATLAB statements. The scalar r is arbitrary; `bilinear` uses $r = \sqrt{2/k}$ to ensure good quantization noise properties in the resulting system.

Diagnostics

`bilinear` requires that the numerator order be no greater than the denominator order. If this is not the case, `bilinear` displays

```
Numerator cannot be higher order than denominator.
```

For `bilinear` to distinguish between the zero-pole-gain and transfer function linear system formats, the first two input parameters must be vectors with the same orientation in these cases. If this is not the case, `bilinear` displays

```
First two arguments must have the same orientation.
```

References

[1] Parks, T.W., and C.S. Burrus. *Digital Filter Design.* New York: John Wiley & Sons, 1987. Pgs. 209-213.

[2] Oppenheim, A.V., and R.W. Schafer. *Discrete-Time Signal Processing.* Upper Saddle River, NJ: Prentice-Hall, 1999, pp. 450-454.

See Also

`impinvar`, `lp2bp`, `lp2bs`, `lp2hp`, `lp2lp`

Purpose

Permute data into bit-reversed order

Syntax

```
y = bitrevorder(x)
[y,i] = bitrevorder(x)
```

Description

`bitrevorder` is useful for pre-arranging filter coefficients so that bit-reversed ordering does not have to be performed as part of an `fft` or inverse FFT (`ifft`) computation. This can improve run-time efficiency for external applications or for Simulink blockset models. Both MATLAB `fft` and `ifft` functions process linear input and output.

Note Using `bitrevorder` is equivalent to using `digitrevorder` with radix base 2.

`y = bitrevorder(x)` returns the input data in bit-reversed order in vector or matrix `y`. The length of `x` must be an integer power of 2. If `x` is a matrix, the bit-reversal occurs on the first dimension of `x` with size greater than 1. `y` is the same size as `x`.

`[y,i] = bitrevorder(x)` returns the bit-reversed vector or matrix `y` and the bit-reversed indices `i`, such that `y = x(i)`. Recall that MATLAB matrices use 1-based indexing, so the first index of `y` will be 1, not 0.

The following table shows the numbers 0 through 7, the corresponding bits and the bit-reversed numbers.

Linear Index	Bits	Bit- Reversed	Bit-Reversed Index
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1

Linear Index	Bits	Bit- Reversed	Bit-Reversed Index
5	101	101	5
6	110	011	3
7	111	111	7

Examples

Obtain the bit-reversed ordered output of a vector:

```
x=[0:7]';                    % Create a column vector
[x,bitrevorder(x)]
ans =
     0     0
     1     4
     2     2
     3     6
     4     1
     5     5
     6     3
     7     7
```

See Also

`fft`, `digitrevorder`, `ifft`

Purpose

Blackman window

Syntax

```
w = blackman(L)
w = blackman(L,'sflag')
```

Description

`w = blackman(L)` returns the `L`-point symmetric Blackman window in the column vector `w`, where `L` is a positive integer.

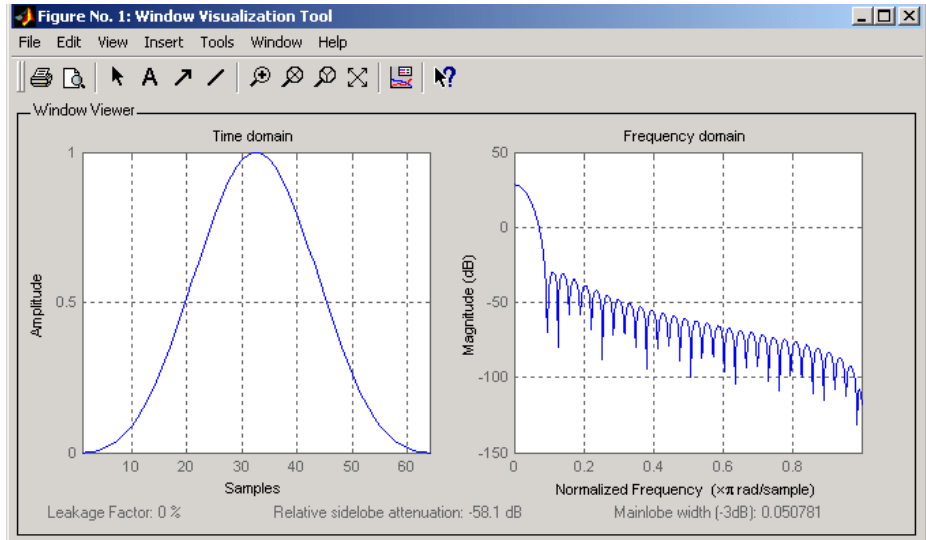
`w = blackman(L,'sflag')` returns an `L`-point Blackman window using the window sampling specified by `'sflag'`, which can be either `'periodic'` or `'symmetric'` (the default). The `'periodic'` flag is useful for DFT/FFT purposes, such as in spectral analysis. The DFT/FFT contains an implicit periodic extension and the periodic flag enables a signal windowed with a periodic window to have perfect periodic extension. When `'periodic'` is specified, `blackman` computes a length `L+1` window and returns the first `L` points. When using windows for filter design, the `'symmetric'` flag should be used.

Note If you specify a one-point window (set `L=1`), the value `1` is returned.

Examples

Create a 64-point Blackman window and display the result using WVTool:

```
L=64;
wvtool(blackman(L))
```



Algorithm

The equation for computing the coefficients of a Blackman window is

$$w(n) = 0.42 - 0.5\cos\left(2\pi\frac{n}{N}\right) + 0.08\cos\left(4\pi\frac{n}{N}\right), \quad 0 \le n \le N$$

The window length $L = N + 1$.

Blackman windows have slightly wider central lobes and less sideband leakage than equivalent length Hamming and Hann windows.

See Also

`flattopwin`, `hamming`, `hann`, `window`, `wintool`, `wvtool`

References

[1] Oppenheim, A.V., and R.W. Schafer. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1999, pp. 468-471.

Purpose

Minimum 4-term Blackman-Harris window

Syntax

```
w = blackmanharris(L)
```

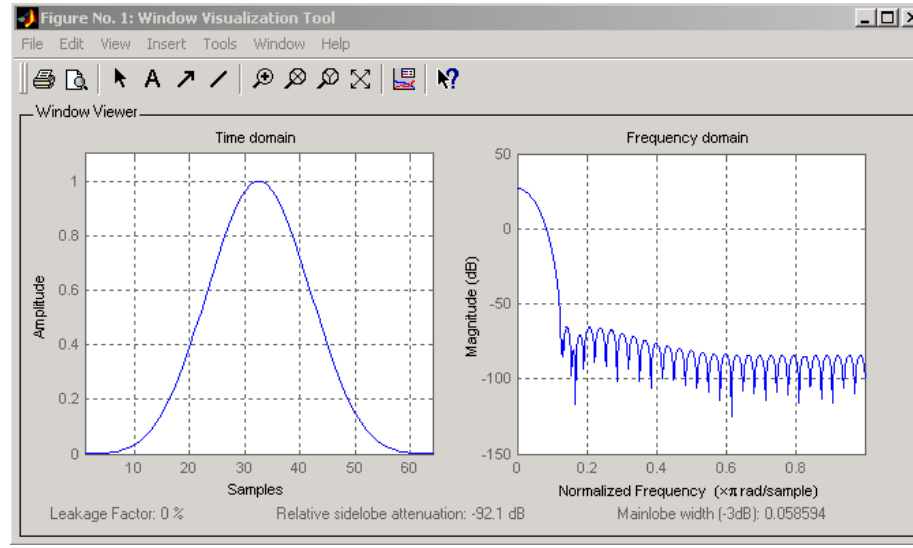
Description

`w = blackmanharris(L)` returns an `L`-point, minimum , 4-term Blackman-Harris window in the column vector `w`. The window is minimum in the sense that its maximum sidelobes are minimized.

Examples

Create a 32-point Blackman-Harris window and display the result using WVTool:

```
L=32;
wvtool(blackmanharris(L))
```



Algorithm

The equation for computing the coefficients of a minimum 4-term Blackman-harris window is

$$w(n) = a_0 - a_1 \cos\left(\frac{2\pi}{N}n\right) + a_2 \cos\left(\frac{2\pi}{N}2n\right) - a_3 \cos\left(\frac{2\pi}{N}3n\right)$$

where $-\frac{N}{2} \le n \le \frac{N}{2}$ and the window length is $L = N + 1$.

The coefficients for this window are

$a_0 = 0.35875$

$a_1 = 0.48829$

$a_2 = 0.14128$

$a_3 = 0.01168$

References

[1] Harris, F. J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform." *Proceedings of the IEEE*. Vol. 66 (January 1978). pp. 51-84.

See Also

`barthannwin`, `bartlett`, `bohmanwin`, `nuttallwin`, `parzenwin`, `rectwin`, `triang`, `window`, `wintool`, `wvtool`

Purpose

Bohman window

Syntax

```
w = bohmanwin(L)
```

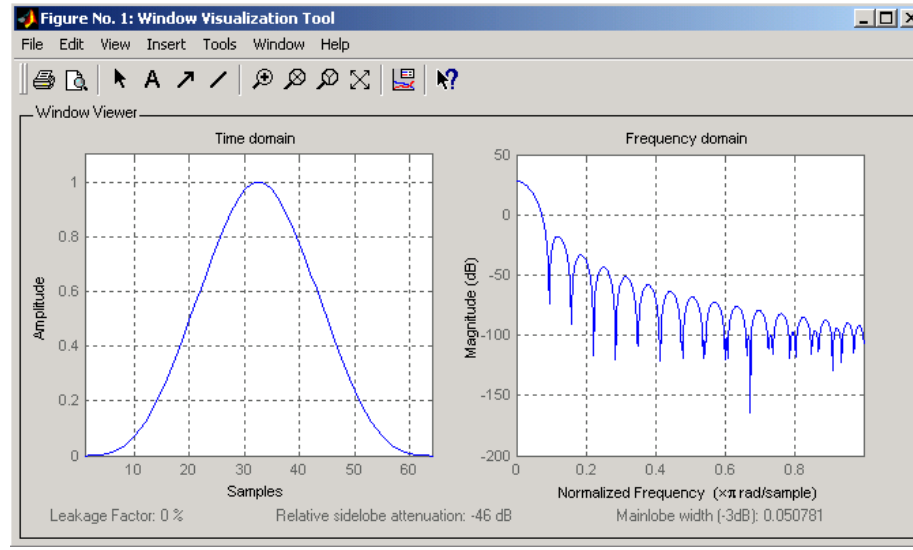
Description

`w = bohmanwin(L)` returns an `L`-point Bohman window in column vector `w`. A Bohman window is the convolution of two half-duration cosine lobes. In the time domain, it is the product of a triangular window and a single cycle of a cosine with a term added to set the first derivative to zero at the boundary. Bohman windows fall off as $1/w^4$.

Examples

Compute a 64-point Bohman window and display the result using WVTool:

```
L=64;
wvtool(bohmanwin(L))
```



Algorithm

The equation for computing the coefficients of a Bohman window is

$$w(n) = \left(1.0 - \frac{|n|}{N/2}\right)\cos\left(\pi\frac{|n|}{N/2}\right) + \frac{1}{n}\sin\left(\pi\frac{|n|}{N/2}\right)$$

where $0 \le |n| \le \frac{N}{2}$ and the window length is $L = N + 1$.

References

[1] Harris, F. J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform." *Proceedings of the IEEE*. Vol. 66 (January 1978). p. 67.

See Also

`barthannwin`, `bartlett`, `blackmanharris`, `nuttallwin`, `parzenwin`, `rectwin`, `triang`, `window`, `wintool`, `wvtool`

Purpose

Buffer signal vector into matrix of data frames

Syntax

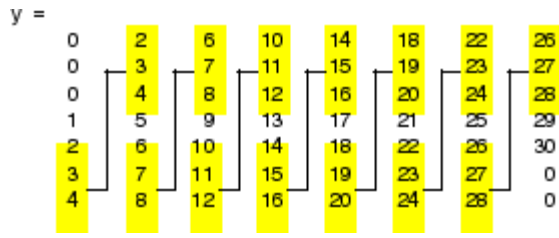
```
y = buffer(x,n)
y = buffer(x,n,p)
y = buffer(x,n,p,opt)
[y,z] = buffer(...)
[y,z,opt] = buffer(...)
```

Description

`y = buffer(x,n)` partitions a length-`L` signal vector `x` into nonoverlapping data segments (frames) of length `n`. Each data frame occupies one column of matrix output `y`, which has `n` rows and `ceil(L/n)` columns. If `L` is not evenly divisible by `n`, the last column is zero-padded to length `n`.

`y = buffer(x,n,p)` overlaps or underlaps successive frames in the output matrix by `p` samples:

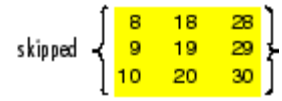
- For `0 < p < n` (overlap), `buffer` repeats the final `p` samples of each frame at the beginning of the following frame. For example, if `x = 1:30` and `n = 7`, an overlap of `p = 3` looks like this.



  The first frame starts with `p` zeros (the default initial condition), and the number of columns in `y` is `ceil(L/(n-p))`.

- For `p < 0` (underlap), `buffer` skips `p` samples between consecutive frames. For example, if `x = 1:30` and `n = 7`, a buffer with underlap of `p = -3` looks like this.

```
y =
     1    11    21
     2    12    22
     3    13    23
     4    14    24
     5    15    25
     6    16    26
     7    17    27
```



The number of columns in `y` is `ceil(L/(n-p))`.

`y = buffer(x,n,p,opt)` specifies a vector of samples to precede `x(1)` in an overlapping buffer, or the number of initial samples to skip in an underlapping buffer:

- For `0 < p < n` (overlap), `opt` specifies a length-`p` vector to insert before `x(1)` in the buffer. This vector can be considered an *initial condition*, which is needed when the current buffering operation is one in a sequence of consecutive buffering operations. To maintain the desired frame overlap from one buffer to the next, `opt` should contain the final `p` samples of the previous buffer in the sequence. See "Continuous Buffering" on page 10-35 below.

  By default, `opt` is `zeros(p,1)` for an overlapping buffer. Set `opt` to `'nodelay'` to skip the initial condition and begin filling the buffer immediately with `x(1)`. In this case, `L` must be `length(p)` or longer. For example, if `x = 1:30` and `n = 7`, a buffer with overlap of `p = 3` looks like this.

```
y =
     1     5     9    13    17    21    25
     2     6    10    14    18    22    26
     3     7    11    15    19    23    27
     4     8    12    16    20    24    28
     5     9    13    17    21    25    29
     6    10    14    18    22    26    30
     7    11    15    19    23    27     0
```

- For `p < 0` (underlap), `opt` is an integer value in the range `[0,-p]` specifying the number of initial input samples, `x(1:opt)`, to skip before adding samples to the buffer. The first value in the buffer

is therefore `x(opt+1)`. By default, `opt` is zero for an underlapping buffer.

This option is especially useful when the current buffering operation is one in a sequence of consecutive buffering operations. To maintain the desired frame underlap from one buffer to the next, `opt` should equal the difference between the total number of points to skip between frames (`p`) and the number of points that were *available* to be skipped in the previous input to `buffer`. If the previous input had fewer than `p` points that could be skipped after filling the final frame of that buffer, the remaining `opt` points need to be removed from the first frame of the current buffer. See "Continuous Buffering" on page 10-35 for an example of how this works in practice.

`[y,z] = buffer(...)` partitions the length-`L` signal vector `x` into frames of length `n`, and outputs only the *full* frames in `y`. If `y` is an overlapping buffer, it has `n` rows and `m` columns, where

```
m = floor(L/(n-p))                % When length(opt) = p
```

or

```
m = floor((L-n)/(n-p))+1           % When opt = 'nodelay'
```

If `y` is an underlapping buffer, it has `n` rows and `m` columns, where

```
m = floor((L-opt)/(n-p)) + (rem((L-opt),(n-p)) >= n)
```

If the number of samples in the input vector (after the appropriate overlapping or underlapping operations) exceeds the number of places available in the `n`-by-`m` buffer, the remaining samples in `x` are output in vector `z`, which for an overlapping buffer has length

```
length(z) = L - m*(n-p)           % When length(opt) = p
```

or

```
length(z) = L - ((m-1)*(n-p)+n)   % When opt = 'nodelay'
```

and for an underlapping buffer has length

```
length(z) = (L-opt) - m*(n-p)
```

Output `z` shares the same orientation (row or column) as `x`. If there are no remaining samples in the input after the buffer with the specified overlap or underlap is filled, `z` is an empty vector.

`[y,z,opt] = buffer(...)` returns the last `p` samples of a overlapping buffer in output `opt`. In an underlapping buffer, `opt` is the difference between the total number of points to skip between frames (`-p`) and the number of points in `x` that were *available* to be skipped after filling the last frame:

- For `0 < p < n` (overlap), `opt` (as an output) contains the final `p` samples in the last frame of the buffer. This vector can be used as the *initial condition* for a subsequent buffering operation in a sequence of consecutive buffering operations. This allows the desired frame overlap to be maintained from one buffer to the next. See "Continuous Buffering" on page 10-35 below.
- For `p < 0` (underlap), `opt` (as an output) is the difference between the total number of points to skip between frames (`-p`) and the number of points in `x` that were *available* to be skipped after filling the last frame.

  ```
  opt = m*(n-p) + opt - L        % z is the empty vector.
  ```

  where `opt` on the right-hand side is the input argument to `buffer`, and `opt` on the left-hand side is the output argument. Here `m` is the number of columns in the buffer, which is

  ```
  m = floor((L-opt)/(n-p)) + (rem((L-opt),(n-p))>=n)
  ```

  Note that for an underlapping buffer output `opt` is always zero when output `z` contains data.

The `opt` output for an underlapping buffer is especially useful when the current buffering operation is one in a sequence of consecutive buffering operations. The `opt` output from each buffering operation specifies the number of samples that need to be skipped at the start of the next buffering operation to maintain the desired frame underlap from one buffer to the next. If fewer than `p` points were available to be skipped after filling the final frame of the current buffer, the remaining `opt` points need to be removed from the first frame of the next buffer.

In a sequence of buffering operations, the `opt` output from each operation should be used as the `opt` input to the subsequent buffering operation. This ensures that the desired frame overlap or underlap is maintained from buffer to buffer, as well as from frame to frame within the same buffer. See "Continuous Buffering" on page 10-35 below for an example of how this works in practice.

Continuous Buffering

In a continuous buffering operation, the vector input to the `buffer` function represents one frame in a sequence of frames that make up a discrete signal. These signal frames can originate in a frame-based data acquisition process, or within a frame-based algorithm like the FFT.

As an example, you might acquire data from an A/D card in frames of 64 samples. In the simplest case, you could rebuffer the data into frames of 16 samples; `buffer` with `n = 16` creates a buffer of four frames from each 64-element input frame. The result is that the signal of frame size 64 has been converted to a signal of frame size 16; no samples were added or removed.

In the general case where the original signal frame size, `L`, is not equally divisible by the new frame size, `n`, the overflow from the last frame needs to be captured and recycled into the following buffer. You can do this by iteratively calling `buffer` on input `x` with the two-output-argument syntax:

```
[y,z] = buffer([z;x],n)     % x is a column vector.
[y,z] = buffer([z,x],n)     % x is a row vector.
```

This simply captures any buffer overflow in z, and prepends the data to the subsequent input in the next call to buffer. Again, the input signal, x, of frame size L, has been converted to a signal of frame size n without any insertion or deletion of samples.

Note that continuous buffering cannot be done with the single-output syntax y = buffer(...), because the last frame of y in this case is zero padded, which adds new samples to the signal.

Continuous buffering in the presence of overlap and underlap is handled with the opt parameter, which is used as both an input and output to buffer. The following two examples demonstrate how the opt parameter should be used.

Examples

Example 1: Continuous Overlapping Buffers

First create a buffer containing 100 frames, each with 11 samples:

```
data = buffer(1:1100,11);      % 11 samples per frame
```

Imagine that the frames (columns) in the matrix called data are the sequential outputs of a data acquisition board sampling a physical signal: data(:,1) is the first D/A output, containing the first 11 signal samples; data(:,2) is the second output, containing the next 11 signal samples, and so on.

You want to rebuffer this signal from the acquired frame size of 11 to a frame size of 4 with an overlap of 1. To do this, you will repeatedly call buffer to operate on each successive input frame, using the opt parameter to maintain consistency in the overlap from one buffer to the next.

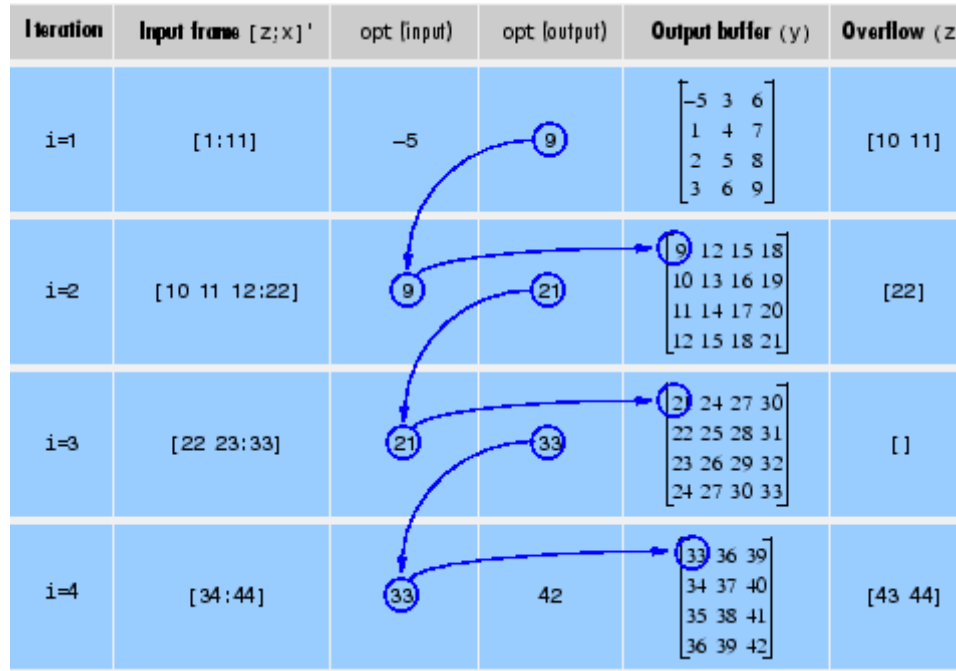
Set the buffer parameters:

```
n = 4;         % New frame size
p = 1;         % Overlap
opt = -5;      % Value of y(1)
z = [];        % Initialize the carry-over vector.
```

Now repeatedly call `buffer`, each time passing in a new signal frame from `data`. Note that overflow samples (returned in `z`) are carried over and prepended to the input in the subsequent call to `buffer`:

```
for i=1:size(data,2),       % Loop over each source
                            %    frame (column)
   x = data(:,i);           % Single frame of D/A output
   [y,z,opt] = buffer([z;x],n,p,opt);
   disp(y);                 % Display the buffer of data.
   pause
end
```

Here's what happens during the first four iterations.



Note that the size of the output matrix, `y`, can vary by a single column from one iteration to the next. This is typical for buffering operations with overlap or underlap.

Example 2: Continuous Underlapping Buffers

Again create a buffer containing 100 frames, each with 11 samples:

```
data = buffer(1:1100,11);  % 11 samples per frame
```

Again, imagine that `data(:,1)` is the first D/A output, containing the first 11 signal samples; `data(:,2)` is the second output, containing the next 11 signal samples, and so on.

You want to rebuffer this signal from the acquired frame size of 11 to a frame size of 4 with an underlap of 2. To do this, you will repeatedly call `buffer` to operate on each successive input frame, using the `opt` parameter to maintain consistency in the underlap from one buffer to the next.

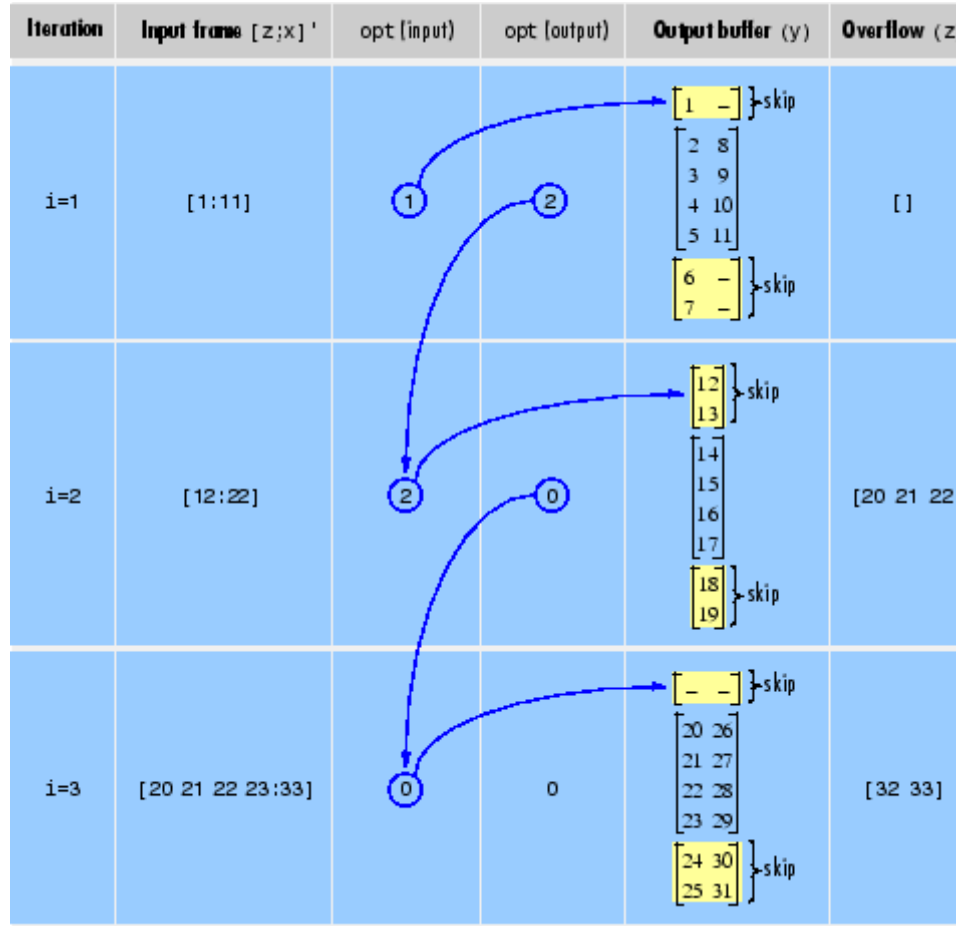
Set the buffer parameters:

```
n = 4;     % New frame size
p = -2;    % Underlap
opt = 1;   % Skip the first input element, x(1).
z = [];    % Initialize the carry-over vector.
```

Now repeatedly call `buffer`, each time passing in a new signal frame from `data`. Note that overflow samples (returned in `z`) are carried over and prepended to the input in the subsequent call to `buffer`:

```
for i=1:size(data,2),      % Loop over each source
                           %    frame (column)
   x = data(:,i);          % Single frame of D/A output
   [y,z,opt] = buffer([z;x],n,p,opt);
   disp(y);                % Display the buffer of data
   pause
end
```

Here's what happens during the first three iterations.



Diagnostics

Error messages are displayed when `p` ≥`n` or `length(opt)`≠`length(p)` in an overlapping buffer case:

```
Frame overlap P must be less than the buffer size N.
Initial conditions must be specified as a length-P vector.
```

See Also

reshape

Purpose

Butterworth filter prototype

Syntax

```
[z,p,k] = buttap(n)
```

Description

`[z,p,k] = buttap(n)` returns the poles and gain of an order `n` Butterworth analog lowpass filter prototype. The function returns the poles in the length `n` column vector `p` and the gain in scalar `k`. `z` is an empty matrix because there are no zeros. The transfer function is

$$H(s) = \frac{z(s)}{p(s)} = \frac{k}{(s-p(1))(s-p(2))\cdots(s-p(n))}$$

Butterworth filters are characterized by a magnitude response that is maximally flat in the passband and monotonic overall. In the lowpass case, the first 2`n`-1 derivatives of the squared magnitude response are zero at $\omega = 0$. The squared magnitude response function is

$$|H(\omega)|^2 = \frac{1}{1+(\omega/\omega_0)^{2n}}$$

corresponding to a transfer function with poles equally spaced around a circle in the left half plane. The magnitude response at the cutoff angular frequency ω_0 is always $1/\sqrt{2}$ regardless of the filter order. `buttap` sets ω_0 to 1 for a normalized result.

Algorithm

```
z = [];
p = exp(sqrt(-1)*(pi*(1:2:2*n-1)/(2*n)+pi/2)).';
k = real(prod(-p));
```

References

[1] Parks, T.W., and C.S. Burrus. *Digital Filter Design.* New York: John Wiley & Sons, 1987. Chapter 7.

See Also

`besselap`, `butter`, `cheb1ap`, `cheb2ap`, `ellipap`

Purpose

Butterworth filter design

Syntax

```
[z,p,k] = butter(n,Wn)
[z,p,k] = butter(n,Wn,'ftype')
[b,a] = butter(n,Wn)
[b,a] = butter(n,Wn,'ftype')
[A,B,C,D] = butter(n,Wn)
[A,B,C,D] = butter(n,Wn,'ftype')
[z,p,k] = butter(n,Wn,'s')
[z,p,k] = butter(n,Wn,'ftype','s')
[b,a] = butter(n,Wn,'s')
[b,a] = butter(n,Wn,'ftype','s')
[A,B,C,D] = butter(n,Wn,'s')
[A,B,C,D] = butter(n,Wn,'ftype','s')
```

Description

`butter` designs lowpass, bandpass, highpass, and bandstop digital and analog Butterworth filters. Butterworth filters are characterized by a magnitude response that is maximally flat in the passband and monotonic overall.

Butterworth filters sacrifice rolloff steepness for monotonicity in the pass- and stopbands. Unless the smoothness of the Butterworth filter is needed, an elliptic or Chebyshev filter can generally provide steeper rolloff characteristics with a lower filter order.

Digital Domain

`[z,p,k] = butter(n,Wn)` designs an order `n` lowpass digital Butterworth filter with normalized cutoff frequency `Wn`. It returns the zeros and poles in length `n` column vectors `z` and `p`, and the gain in the scalar `k`.

`[z,p,k] = butter(n,Wn,'ftype')` designs a highpass, lowpass, or bandstop filter, where the string `'ftype'` is one of the following:

- `'high'` for a highpass digital filter with normalized cutoff frequency `Wn`
- `'low'` for a lowpass digital filter with normalized cutoff frequency `Wn`

- `'stop'` for an order `2*n` bandstop digital filter if `Wn` is a two-element vector, `Wn = [w1 w2]`. The stopband is `w1 < ω < w2`.

Cutoff frequency is that frequency where the magnitude response of the filter is $\sqrt{1/2}$. For `butter`, the normalized cutoff frequency `Wn` must be a number between 0 and 1, where 1 corresponds to the Nyquist frequency, π radians per sample.

If `Wn` is a two-element vector, `Wn = [w1 w2]`, `butter` returns an order `2*n` digital bandpass filter with passband `w1 < ω < w2`.

With different numbers of output arguments, `butter` directly obtains other realizations of the filter. To obtain the transfer function form, use two output arguments as shown below.

Note See "Limitations" on page 10-45 below for information about numerical issues that affect forming the transfer function.

`[b,a] = butter(n,Wn)` designs an order `n` lowpass digital Butterworth filter with normalized cutoff frequency `Wn`. It returns the filter coefficients in length `n+1` row vectors `b` and `a`, with coefficients in descending powers of *z*.

$$H(z) = \frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \cdots + b(n+1)z^{-n}}{1 + a(2)z^{-1} + \cdots + a(n+1)z^{-n}}$$

`[b,a] = butter(n,Wn,'ftype')` designs a highpass, lowpass, or bandstop filter, where the string `'ftype'` is `'high'`, `'low'`, or `'stop'`, as described above.

To obtain state-space form, use four output arguments as shown below:

`[A,B,C,D] = butter(n,Wn)` or

`[A,B,C,D] = butter(n,Wn,'ftype')` where `A`, `B`, `C`, and `D` are

$$x[n+1] = Ax[n] + Bu[n]$$
$$y[n] = Cx[n] + Du[n]$$

and u is the input, x is the state vector, and y is the output.

Analog Domain

`[z,p,k] = butter(n,Wn,'s')` designs an order `n` lowpass analog Butterworth filter with angular cutoff frequency `Wn` rad/s. It returns the zeros and poles in length `n` or `2*n` column vectors `z` and `p` and the gain in the scalar `k`. `butter`'s angular cutoff frequency `Wn` must be greater than 0 rad/s.

If `Wn` is a two-element vector with `w1 < w2`, `butter(n,Wn,'s')` returns an order `2*n` bandpass analog filter with passband `w1` < ω < `w2`.

`[z,p,k] = butter(n,Wn,'ftype','s')` designs a highpass, lowpass, or bandstop filter using the *`ftype`* values described above.

With different numbers of output arguments, `butter` directly obtains other realizations of the analog filter. To obtain the transfer function form, use two output arguments as shown below:

`[b,a] = butter(n,Wn,'s')` designs an order `n` lowpass analog Butterworth filter with angular cutoff frequency `Wn` rad/s. It returns the filter coefficients in the length `n+1` row vectors `b` and `a`, in descending powers of *s,* derived from this transfer function:

$$H(s) = \frac{B(s)}{A(s)} = \frac{b(1)s^n + b(2)s^{n-1} + \cdots + b(n+1)}{s^n + a(2)s^{n-1} + \cdots + a(n+1)}$$

`[b,a] = butter(n,Wn,'ftype','s')` designs a highpass, lowpass, or bandstop filter using the *`ftype`* values described above.

To obtain state-space form, use four output arguments as shown below:

`[A,B,C,D] = butter(n,Wn,'s')` or

`[A,B,C,D] = butter(n,Wn,'ftype','s')` where `A`, `B`, `C`, and `D` are

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

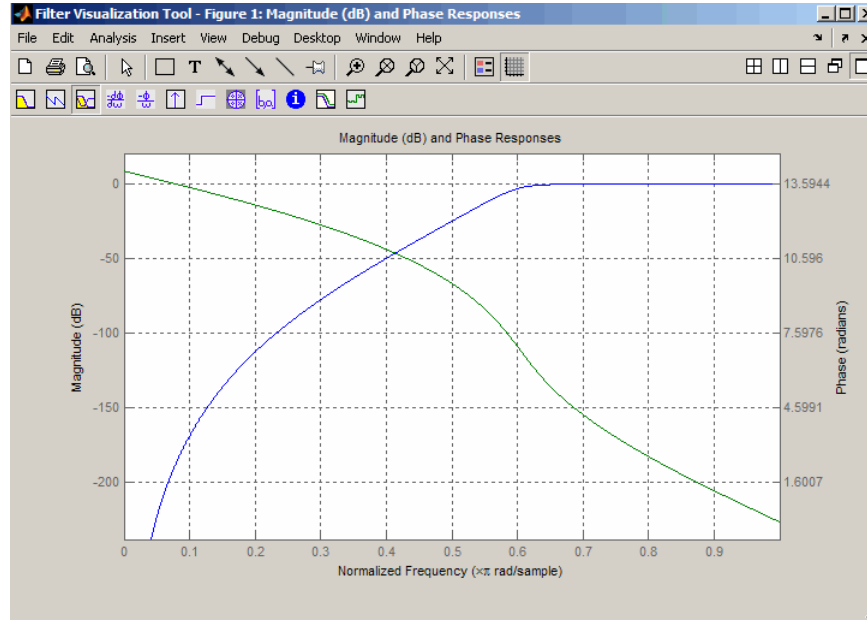
and u is the input, x is the state vector, and y is the output.

Examples

Highpass Filter

For data sampled at 1000 Hz, design a 9th-order highpass Butterworth filter with cutoff frequency of 300 Hz, which corresponds to a normalized value of 0.6:

```
[z,p,k] = butter(9,300/500,'high');
[sos,g] = zp2sos(z,p,k);      % Convert to SOS form
Hd = dfilt.df2tsos(sos,g);    % Create a dfilt object
h = fvtool(Hd);               % Plot magnitude response
set(h,'Analysis','freq')      % Display frequency response
```

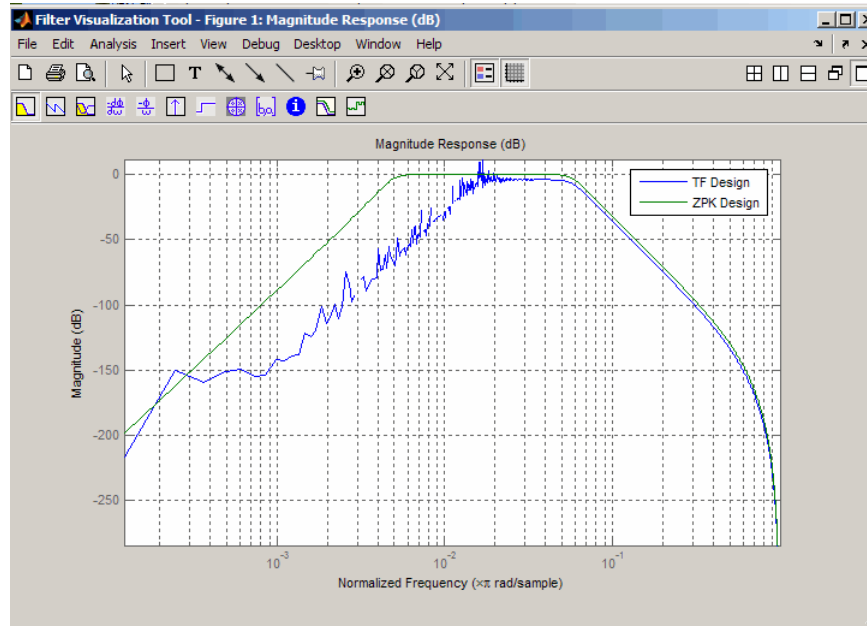


Limitations

In general, you should use the `[z,p,k]` syntax to design IIR filters. To analyze or implement your filter, you can then use the `[z,p,k]` output with `zp2sos` and an sos `dfilt` structure. For higher order filters

(possibly starting as low as order 8), numerical problems due to roundoff errors may occur when forming the transfer function using the `[b,a]` syntax. The following example illustrates this limitation:

```
n = 6; Wn = [2.5e6 29e6]/500e6;
ftype = 'bandpass';

% Transfer Function design
[b,a] = butter(n,Wn,ftype);
h1=dfilt.df2(b,a);      % This is an unstable filter.

% Zero-Pole-Gain design
[z, p, k] = butter(n,Wn,ftype);
[sos,g]=zp2sos(z,p,k);
h2=dfilt.df2sos(sos,g);

% Plot and compare the results
hfvt=fvtool(h1,h2,'FrequencyScale','log');
legend(hfvt,'TF Design','ZPK Design')
```



Algorithm

`butter` uses a five-step algorithm:

1. It finds the lowpass analog prototype poles, zeros, and gain using the `buttap` function.
2. It converts the poles, zeros, and gain into state-space form.
3. It transforms the lowpass filter into a bandpass, highpass, or bandstop filter with desired cutoff frequencies, using a state-space transformation.
4. For digital filter design, `butter` uses `bilinear` to convert the analog filter into a digital filter through a bilinear transformation with frequency prewarping. Careful frequency adjustment guarantees that the analog filters and the digital filters will have the same frequency response magnitude at `Wn` or `w1` and `w2`.

5 It converts the state-space filter back to transfer function or zero-pole-gain form, as required.

See Also

`besself`, `buttap`, `buttord`, `cheby1`, `cheby2`, `ellip`, `maxflat`

Purpose

Butterworth filter order and cutoff frequency

Syntax

```
[n,Wn] = buttord(Wp,Ws,Rp,Rs)
[n,Wn] = buttord(Wp,Ws,Rp,Rs,'s')
```

Description

`buttord` calculates the minimum order of a digital or analog Butterworth filter required to meet a set of filter design specifications.

Digital Domain

`[n,Wn] = buttord(Wp,Ws,Rp,Rs)` returns the lowest order, `n`, of the digital Butterworth filter that loses no more than `Rp` dB in the passband and has at least `Rs` dB of attenuation in the stopband. The scalar (or vector) of corresponding cutoff frequencies, `Wn`, is also returned. Use the output arguments `n` and `Wn` in `butter`.

Choose the input arguments to specify the stopband and passband according to the following table.

Description of Stopband and Passband Filter Parameters

Parameter	Description
`Wp`	Passband corner frequency `Wp`, the cutoff frequency, is a scalar or a two-element vector with values between 0 and 1, with 1 corresponding to the normalized Nyquist frequency, π radians per sample.
`Ws`	Stopband corner frequency `Ws`, is a scalar or a two-element vector with values between 0 and 1, with 1 corresponding to the normalized Nyquist frequency.
`Rp`	Passband ripple, in decibels. This value is the maximum permissible passband loss in decibels.
`Rs`	Stopband attenuation, in decibels. This value is the number of decibels the stopband is down from the passband.

Use the following guide to specify filters of different types.

Filter Type Stopband and Passband Specifications

Filter Type	Stopband and Passband Conditions	Stopband	Passband
Lowpass	`Wp` < `Ws`, both scalars	`(Ws,1)`	`(0,Wp)`
Highpass	`Wp` > `Ws`, both scalars	`(0,Ws)`	`(Wp,1)`
Bandpass	The interval specified by `Ws` contains the one specified by `Wp` (`Ws(1) < Wp(1) < Wp(2) < Ws(2)`).	`(0,Ws(1))` and `(Ws(2),1)`	`(Wp(1),Wp(2))`
Bandstop	The interval specified by `Wp` contains the one specified by `Ws` (`Wp(1) < Ws(1) < Ws(2) < Wp(2)`).	`(0,Wp(1))` and `(Wp(2),1)`	`(Ws(1),Ws(2))`

If your filter specifications call for a bandpass or bandstop filter with unequal ripple in each of the passbands or stopbands, design separate lowpass and highpass filters according to the specifications in this table, and cascade the two filters together.

Analog Domain

`[n,Wn] = buttord(Wp,Ws,Rp,Rs,'s')` finds the minimum order `n` and cutoff frequencies `Wn` for an analog Butterworth filter. You specify the frequencies `Wp` and `Ws` similar those described in the Description of Stopband and Passband Filter Parameters on page 10-49 table above, only in this case you specify the frequency in radians per second, and the passband or the stopband can be infinite.

Use `buttord` for lowpass, highpass, bandpass, and bandstop filters as described in the Filter Type Stopband and Passband Specifications on page 10-50 table above.

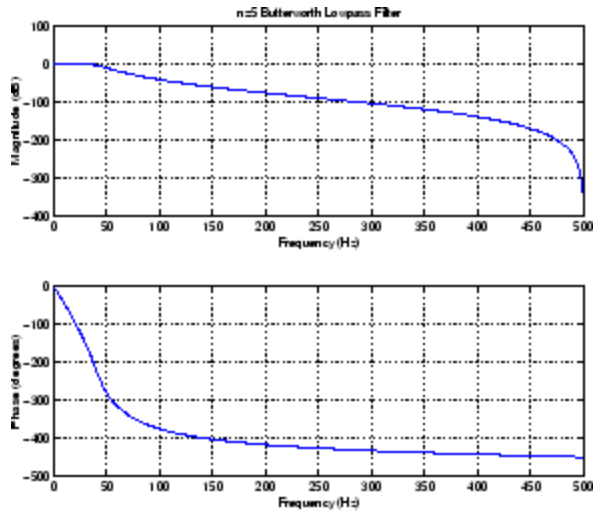
Examples

Example 1

For data sampled at 1000 Hz, design a lowpass filter with less than 3 dB of ripple in the passband, defined from 0 to 40 Hz, and at least 60 dB

of attenuation in the stopband, defined from 150 Hz to the Nyquist frequency (500 Hz). Plot the filter's frequency response:

```
Wp = 40/500; Ws = 150/500;
[n,Wn] = buttord(Wp,Ws,3,60)
n =
     5
Wn =
    0.0810
[b,a] = butter(n,Wn);
freqz(b,a,512,1000);
title('n=5 Butterworth Lowpass Filter')
```

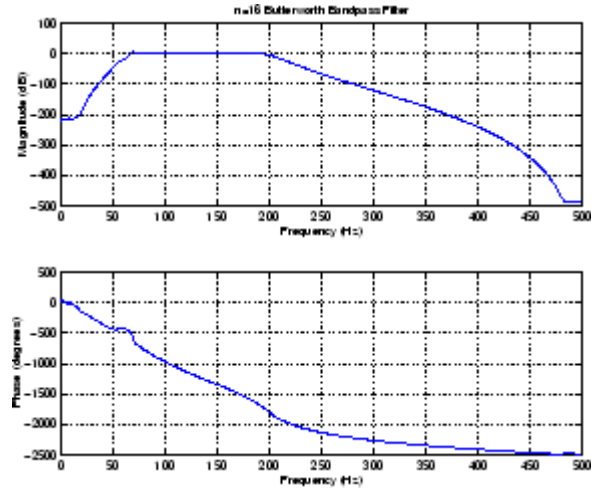


Example 2

Next design a bandpass filter with passband of 60 Hz to 200 Hz, with less than 3 dB of ripple in the passband, and 40 dB attenuation in the stopbands that are 50 Hz wide on both sides of the passband:

```
Wp = [60 200]/500; Ws = [50 250]/500;
Rp = 3; Rs = 40;
[n,Wn] = buttord(Wp,Ws,Rp,Rs)
```

```
n =
    16
Wn =
    0.1198    0.4005
[b,a] = butter(n,Wn);
freqz(b,a,128,1000)
title('n=16 Butterworth Bandpass Filter')
```



Algorithm

`buttord`'s order prediction formula is described in [1]. It operates in the analog domain for both analog and digital cases. For the digital case, it converts the frequency parameters to the *s*-domain before estimating the order and natural frequency, and then converts back to the *z*-domain.

`buttord` initially develops a lowpass filter prototype by transforming the passband frequencies of the desired filter to 1 rad/s (for lowpass and highpass filters) and to -1 and 1 rad/s (for bandpass and bandstop filters). It then computes the minimum order required for a lowpass filter to meet the stopband specification.

References

[1] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975. Pg. 227.

See Also

butter, cheb1ord, cheb2ord, ellipord, kaiserord

cceps

Purpose

Complex cepstral analysis

Syntax

```
xhat = cceps(x)
[xhat,nd] = cceps(x)
[xhat,nd,xhat1] = cceps(x)
[...] = cceps(x,n)
```

Description

Cepstral analysis is a nonlinear signal processing technique that is applied most commonly in speech processing and homomorphic filtering [1].

Note `cceps` only works on real data.

`xhat = cceps(x)` returns the complex cepstrum of the real data sequence `x` using the Fourier transform. The input is altered, by the application of a linear phase term, to have no phase discontinuity at $\pm\pi$ radians. That is, it is circularly shifted (after zero padding) by some samples, if necessary, to have zero phase at π radians.

`[xhat,nd] = cceps(x)` returns the number of samples `nd` of (circular) delay added to `x` prior to finding the complex cepstrum.

`[xhat,nd,xhat1] = cceps(x)` returns a second complex cepstrum `xhat1` computed using an alternative factorization algorithm[1][2]. This method can be applied only to finite duration signals. See the Algorithm section below for a comparison of the Fourier and factorization methods of computing the complex cepstrum.

`[...] = cceps(x,n)` zero pads `x` to length `n` and returns the length `n` complex cepstrum of `x`.

Algorithm

`cceps` is an M-file implementation of algorithm 7.1 in [3]. A lengthy Fortran program reduces to these three lines of MATLAB code, which compose the core of `cceps`:

```
h = fft(x);
```

```
logh = log(abs(h)) + sqrt(-1)*rcunwrap(angle(h));
y = real(ifft(logh));
```

Note `rcunwrap` in the above code segment is a special version of `unwrap` that subtracts a straight line from the phase. `rcunwrap` is a local function within `cceps` and is not available for use from the MATLAB command line.

The following table lists the pros and cons of the Fourier and factorization algorithms.

Algorithm	Pros	Cons
Fourier	Can be used for any signal.	Requires phase unwrapping. Output is aliased.
Factorization	Does not require phase unwrapping. No aliasing	Can be used only for short duration signals. Input signal must have an all-zero Z-transform with no zeros on the unit circle.

In general, you cannot use the results of these two algorithms to verify each other. You can use them to verify each other only when the first element of the input data is positive, the Z-transform of the data sequence has only zeros, all of these zeros are inside the unit circle, and the input data sequence is long (or padded with zeros).

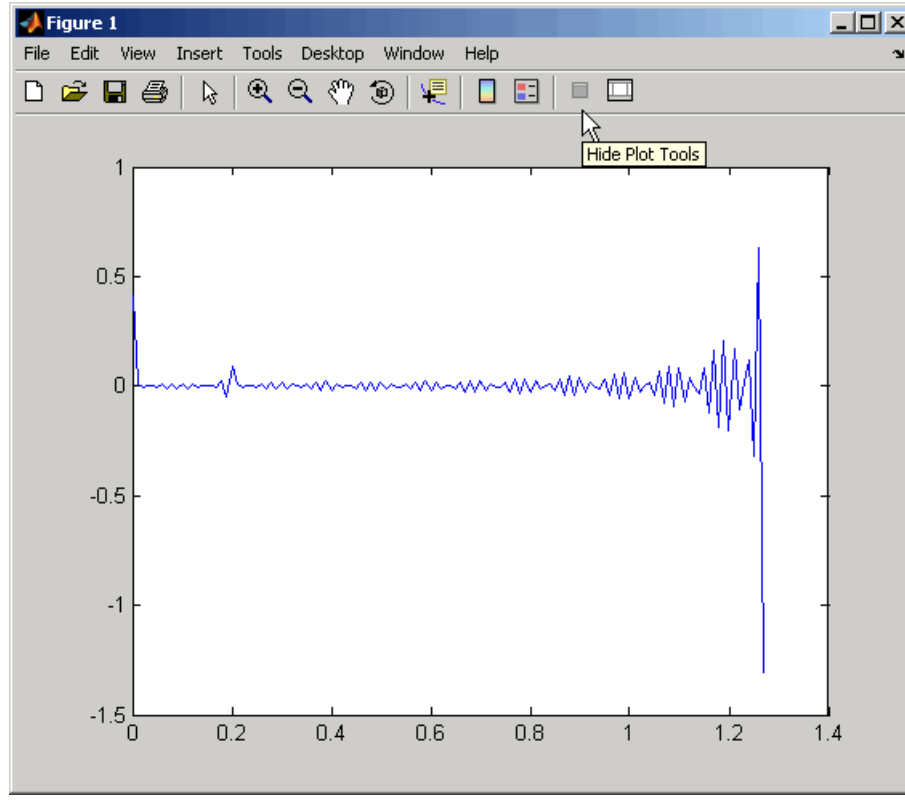
Examples

The following example uses `cceps` to show an echo.

```
Fs = 100;
t = 0:1/Fs:1.27;

% 45Hz sine sampled at 100Hz
s1 = sin(2*pi*45*t);
```

```
% Add an echo with half the amplitude and 0.2 second later
s2 = s1 + 0.5*[zeros(1,20) s1(1:108)];

c = cceps(s2);
plot(t,c)
```

Notice the echo at 0.2 second.



References

[1] Oppenheim, A.V., and R.W. Schafer. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1999, pp. 788-789.

[2] Steiglitz, K., and B. Dickinson. "Computation of the complex cepstrum by factorization of the Z-transform" in *Proc. Int. Conf. ASSP*. 1977, pp. 723–726.

[3] *IEEE Programs for Digital Signal Processing*. IEEE Press. New York: John Wiley & Sons, 1979.

See Also

`icceps`, `hilbert`, `rceps`, `unwrap`

Purpose

Modulo-N circular convolution

Syntax

```
c = cconv(a,b,n)
```

Description

Circular convolution is used to convolve two discrete Fourier transform (DFT) sequences. For very long sequences, circular convolution may be faster than linear convolution.

`c = cconv(a,b,n)` circularly convolves vectors `a` and `b`. `n` is the length of the resulting vector. If you omit `n`, it defaults to `length(a)+length(B)-1`. When `n = length(a)+length(B)-1`, the circular convolution is equivalent to the linear convolution computed with `conv`. You can also use `cconv` to compute the circular cross-correlation of two sequences (see the example below).

Examples

The following example calculates a modulo-4 circular convolution.

```
a = [2 1 2 1];
b = [1 2 3 4];
c = cconv(a,b,4)

c =
    14    16    14    16
```

The following example compares a circular correlation, where `n` uses the default value, and a linear convolution. The resulting norm is a value that is virtually zero, which shows that the two convolutions produce virtually the same result.

```
a = [1 2 -1 1];
b = [1 1 2 1 2 2 1 1];
c = cconv(a,b)      % Circular convolution
cref = conv(a,b)    % Linear convolution
norm(c-cref)

ans =
  9.7422e-016
```

The following example uses cconv to compute the circular cross-correlation of two sequences. The result is compared to the cross-correlation computed using xcorr.

```
a = [1 2 2 1]+i;
b = [1 3 4 1]-2*i;
c = cconv(a,conj(fliplr(b)),7) % Compute using cconv
cref = xcorr(a,b)      % Compute using xcorr

c =

  Columns 1 through 5
  -1.0000 + 3.0000i   2.0000 +11.0000i   7.0000 +18.0000i
   8.0000 +21.0000i   6.0000 +18.0000i

  Columns 6 through 7
   1.0000 +10.0000i  -1.0000 + 3.0000i

cref =

  Columns 1 through 5
  -1.0000 + 3.0000i   2.0000 +11.0000i   7.0000 +18.0000i
   8.0000 +21.0000i   6.0000 +18.0000i

  Columns 6 through 7
   1.0000 +10.0000i  -1.0000 + 3.0000i
```

References

[1] Orfanidis, S.J., *Introduction to Signal Processing*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1996. pp. 524–529.

See Also

conv

cell2sos

Purpose

Convert second-order sections cell array to matrix

Syntax

```
m = cell2sos(c)
```

Description

`m = cell2sos(c)` changes a 1-by-*L* cell array `c` consisting of 1-by-2 cell arrays into an *L*-by-6 second-order section matrix m. Matrix `m` takes the same form as the matrix generated by `tf2sos`. You can use m = cell2sos(c) to invert the results of `c = sos2cell(m)`.

`c` must be a cell array of the form

```
c = { {b1 a1} {b2 a2} ... {bL aL} }
```

where both `bi` and `ai` are row vectors of at most length 3, and *i* = 1, 2, ..., *L*. The resulting matrix `m` is given by

```
m = [b1 a1;b2 a2; ... ;bL aL]
```

See Also

`sos2cell`, `tf2sos`

Purpose

Complex and nonlinear-phase equiripple FIR filter design

Syntax

```
b = cfirpm(n,f,@fresp)
b = cfirpm(n,f,@fresp,w)
b = cfirpm(n,f,a,w)
b = cfirpm(...,'sym')
b = cfirpm(...,'skip_stage2')
b = cfirpm(..., 'debug')
b = cfirpm(...,{lgrid})
[b,delta] = cfirpm(...)
[b,delta,opt] = cfirpm(...)
```

Description

`cfirpm` allows arbitrary frequency-domain constraints to be specified for the design of a possibly complex FIR filter. The Chebyshev (or minimax) filter error is optimized, producing equiripple FIR filter designs.

`b = cfirpm(n,f,@fresp)` returns a length `n+1` FIR filter with the best approximation to the desired frequency response as returned by function *`fresp`*, which is called by its function handle (`@fresp`). `f` is a vector of frequency band edge pairs, specified in the range -1 and 1, where 1 corresponds to the normalized Nyquist frequency. The frequencies must be in increasing order, and `f` must have even length. 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.

Predefined `fresp` frequency response functions are included for a number of common filter designs, as described below. For all of the predefined frequency response functions, the symmetry option *`'sym'`* defaults to `'even'` if no negative frequencies are contained in `f` and `d = 0`; otherwise *`'sym'`* defaults to `'none'`. (See the *`'sym'`* option below for details.) For all of the predefined frequency response functions, `d` specifies a group-delay offset such that the filter response has a group delay of `n/2+d` in units of the sample interval. Negative values create less delay; positive values create more delay. By default `d = 0`:

- `@lowpass`, `@highpass`, `@allpass`, `@bandpass`, `@bandstop`

These functions share a common syntax, exemplified below by the string `'lowpass'`.

`b = cfirpm(n,f,@lowpass,...)` and

`b = cfirpm(n,f,{@lowpass,d},...)` design a linear-phase (`n/2+d` delay) filter.

Note For `@bandpass` filters, the first element in the frequency vector must be less than or equal to zero and the last element must be greater than or equal to zero.

- `@multiband` designs a linear-phase frequency response filter with arbitrary band amplitudes.

  `b = cfirpm(n,f,{@multiband,a},...)` and

  `b = cfirpm(n,f,{@multiband,a,d},...)` specify vector `a` containing the desired amplitudes at the band edges in `f`. The desired amplitude at frequencies between pairs of points `f(k)` and `f(k+1)` for `k` odd is the line segment connecting the points `(f(k),a(k))` and `(f(k+1),a(k+1))`.

- `@differentiator` designs a linear-phase differentiator. For these designs, zero-frequency must be in a transition band, and band weighting is set to be inversely proportional to frequency.

  `b = cfirpm(n,f,{@differentiator,fs},...)` and

  `b = cfirpm(n,f,{@differentiator,fs,d},...)` specify the sample rate `fs` used to determine the slope of the differentiator response. If omitted, `fs` defaults to 1.

- `@hilbfilt` designs a linear-phase Hilbert transform filter response. For Hilbert designs, zero-frequency must be in a transition band.

  `b = cfirpm(n,f,@hilbfilt,...)` and

  `b = cfirpm(N,F,{@hilbfilt,d},...)` design a linear-phase (`n/2+d` delay) Hilbert transform filter.

- `@invsinc` designs a linear-phase inverse-sinc filter response.

  `b = cfirpm(n,f,{@invsinc,a},...)` and

  `b = cfirpm(n,f,{@invsinc,a,d},...)` specify gain `a` for the sinc-function, computed as sinc(`a`*g), where g contains the optimization grid frequencies normalized to the range [-1,1]. By default, `a`=1. The group-delay offset is `d`, such that the filter response will have a group delay of N/2 + `d` in units of the sample interval, where N is the filter order. Negative values create less delay and positive values create more delay. By default, `d`=0.

`b = cfirpm(n,f,@fresp,w)` uses the real, non-negative weights in vector `w` to weight the fit in each frequency band. The length of `w` is half the length of `f`, so there is exactly one weight per band.

`b = cfirpm(n,f,a,w)` is a synonym for
`b = cfirpm(n,f,{@multiband,a},w)`.

`b = cfirpm(...,'sym')` imposes a symmetry constraint on the impulse response of the design, where *`'sym'`* may be one of the following:

- `'none'` indicates no symmetry constraint. This is the default if any negative band edge frequencies are passed, or if *`fresp`* does not supply a default.
- `'even'` indicates a real and even impulse response. This is the default for highpass, lowpass, allpass, bandpass, bandstop, invsinc, and multiband designs.
- `'odd'` indicates a real and odd impulse response. This is the default for Hilbert and differentiator designs.
- `'real'` indicates conjugate symmetry for the frequency response

If any *`'sym'`* option other than `'none'` is specified, the band edges should be specified only over positive frequencies; the negative frequency region is filled in from symmetry. If a *`'sym'`* option is not specified, the *`fresp`* function is queried for a default setting. Any

user-supplied *fresp* function should return a valid *'sym'* string when it is passed the string `'defaults'` as the filter order `N`.

`b = cfirpm(...,'skip_stage2')` disables the second-stage optimization algorithm, which executes only when `cfirpm` determines that an optimal solution has not been reached by the standard firpm error-exchange. Disabling this algorithm may increase the speed of computation, but may incur a reduction in accuracy. By default, the second-stage optimization is enabled.

`b = cfirpm(..., 'debug')` enables the display of intermediate results during the filter design, where *'debug'* may be one of `'trace'`, `'plots'`, `'both'`, or `'off'`. By default it is set to `'off'`.

`b = cfirpm(...,{lgrid})` uses the integer `lgrid` to control the density of the frequency grid, which has roughly `2^nextpow2(lgrid*n)` frequency points. The default value for `lgrid` is `25`. Note that the `{lgrid}` argument must be a 1-by-1 cell array.

Any combination of the *'sym'*, `'skip_stage2'`, *'debug'*, and {lgrid} options may be specified.

`[b,delta] = cfirpm(...)` returns the maximum ripple height `delta`.

`[b,delta,opt] = cfirpm(...)` returns a structure `opt` of optional results computed by `cfirpm` and contains the following fields.

Field	Description
`opt.fgrid`	Frequency grid vector used for the filter design optimization
`opt.des`	Desired frequency response for each point in `opt.fgrid`
`opt.wt`	Weighting for each point in `opt.fgrid`
`opt.H`	Actual frequency response for each point in `opt.fgrid`
`opt.error`	Error at each point in `opt.fgrid`

Field	Description
`opt.iextr`	Vector of indices into `opt.fgrid` for extremal frequencies
`opt.fextr`	Vector of extremal frequencies

User-definable functions may be used, instead of the predefined frequency response functions for *@fresp*. The function is called from within `cfirpm` using the following syntax

```
[dh,dw] = fresp(n,f,gf,w,p1,p2,...)
```

where:

- `n` is the filter order.
- `f` is the vector of frequency band edges that appear monotonically between -1 and 1, where 1 corresponds to the Nyquist frequency.
- `gf` is a vector of grid points that have been linearly interpolated over each specified frequency band by `cfirpm`. `gf` determines the frequency grid at which the response function must be evaluated. This is the same data returned by `cfirpm` in the `fgrid` field of the `opt` structure.
- `w` is a vector of real, positive weights, one per band, used during optimization. `w` is optional in the call to `cfirpm`; if not specified, it is set to unity weighting before being passed to *`fresp`*.
- `dh` and `dw` are the desired complex frequency response and band weight vectors, respectively, evaluated at each frequency in grid `gf`.
- `p1`, `p2`, `...`, are optional parameters that may be passed to *`fresp`*.

Additionally, a preliminary call is made to *`fresp`* to determine the default symmetry property *`'sym'`*. This call is made using the syntax:

```
sym = fresp('defaults',{n,f,[],w,p1,p2,...})
```

The arguments may be used in determining an appropriate symmetry default as necessary. The function `private/lowpass.m` may be useful as a template for generating new frequency response functions.

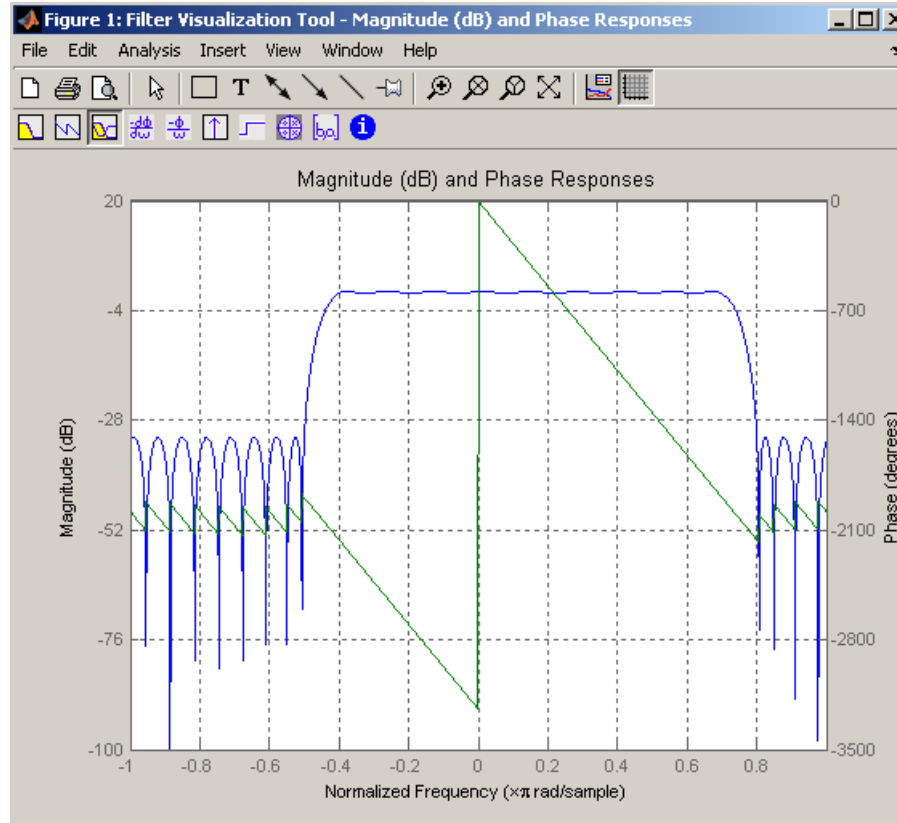
Examples

Example 1

Design a 31-tap, linear-phase, lowpass filter:

```
b = cfirpm(30,[-1 -0.5 -0.4 0.7 0.8 1],@lowpass);
fvtool(b,1)
```

Click the **Magnitude and Phase Response** button.



Example 2

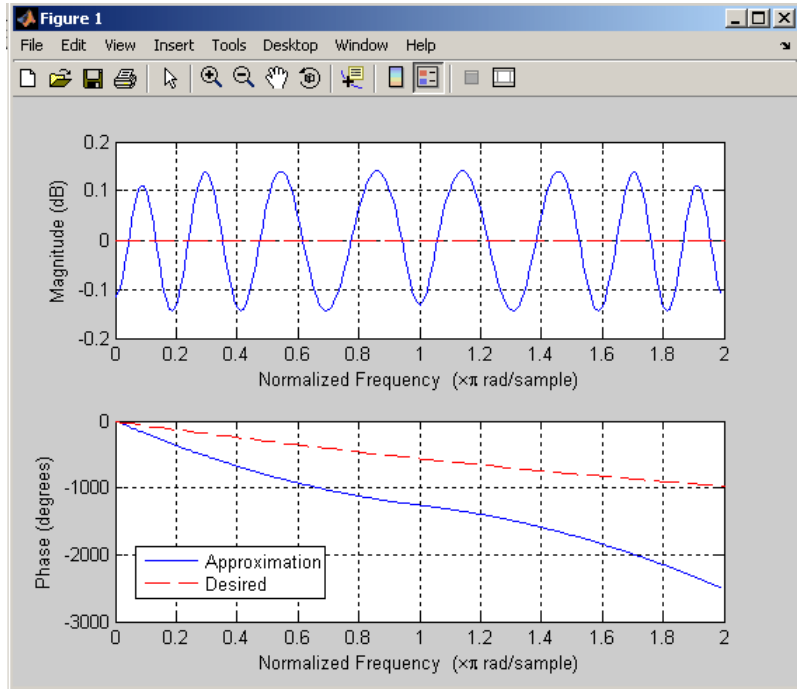
Design a nonlinear-phase allpass FIR filter:

```
n = 22;                         % Filter order
f = [-1 1];                     % Frequency band edges
w = [1 1];                      % Weights for optimization
gf = linspace(-1,1,256);     % Grid of frequency points
d = exp(-1i*pi*gf*n/2 + 1i*pi*pi*sign(gf).*gf.*gf*(4/pi));
                  % Desired frequency response
```

Vector `d` now contains the complex frequency response that we desire for the FIR filter computed by `cfirpm`.

Now compute the FIR filter that best approximates this response:

```
b = cfirpm(n,f,'allpass',w,'real');  % Approximation
freqz(b,1,256,'whole');
subplot(2,1,1); hold on              % Overlay response
plot(pi*(gf+1),20*log10(abs(fftshift(d))),'r--')
subplot(2,1,2); hold on
plot(pi*(gf+1),unwrap(angle(fftshift(d)))*180/pi,'r--')
legend('Approximation','Desired')
```



Algorithm

An extended version of the Remez exchange method is implemented for the complex case. This exchange method obtains the optimal filter when

the equiripple nature of the filter is restricted to have `n+2` extremals. When it does not converge, the algorithm switches to an ascent-descent algorithm that takes over to finish the convergence to the optimal solution. See the references for further details.

References

[1] Karam, L.J., and J.H. McClellan. "Complex Chebyshev Approximation for FIR Filter Design." *IEEE Trans. on Circuits and Systems II,*March 1995. Pgs. 207-216.

[2] Karam, L.J. *Design of Complex Digital FIR Filters in the Chebyshev Sense,* Ph.D. Thesis, Georgia Institute of Technology, March 1995.

[3] Demjanjov, V.F., and V.N. Malozemov. *Introduction to Minimax,* New York: John Wiley & Sons, 1974.

See Also

`fir1`, `fir2`, `firls`, `firpm`, `function_handle`

cheb1ap

Purpose

Chebyshev Type I analog lowpass filter prototype

Syntax

```
[z,p,k] = cheb1ap(n,Rp)
```

Description

`[z,p,k] = cheb1ap(n,Rp)` returns the poles and gain of an order `n` Chebyshev Type I analog lowpass filter prototype with `Rp` dB of ripple in the passband. The function returns the poles in the length `n` column vector `p` and the gain in scalar `k`. `z` is an empty matrix, because there are no zeros. The transfer function is

$$H(s) = \frac{z(s)}{p(s)} = \frac{k}{(s-p(1))(s-p(2))\cdots(s-p(n))}$$

Chebyshev Type I filters are equiripple in the passband and monotonic in the stopband. The poles are evenly spaced about an ellipse in the left half plane. The Chebyshev Type I passband edge angular frequency ω_0 is set to 1.0 for a normalized result. This is the frequency at which the passband ends and the filter has magnitude response of $10^{-Rp/20}$.

References

[1] Parks, T.W., and C.S. Burrus. *Digital Filter Design,*New York: John Wiley & Sons, 1987. Chapter 7.

See Also

`besselap`, `buttap`, `cheby1`, `cheb2ap`, `ellipap`

Purpose

Chebyshev Type I filter order

Syntax

```
[n,Wp] = cheb1ord(Wp,Ws,Rp,Rs)
[n,Wp] = cheb1ord(Wp,Ws,Rp,Rs,'s')
```

Description

`cheb1ord` calculates the minimum order of a digital or analog Chebyshev Type I filter required to meet a set of filter design specifications.

Digital Domain

`[n,Wp] = cheb1ord(Wp,Ws,Rp,Rs)` returns the lowest order `n` of the Chebyshev Type I filter that loses no more than `Rp` dB in the passband and has at least `Rs` dB of attenuation in the stopband. The scalar (or vector) of corresponding cutoff frequencies `Wp`, is also returned. Use the output arguments `n` and `Wp` with the `cheby1` function.

Choose the input arguments to specify the stopband and passband according to the following table.

Description of Stopband and Passband Filter Parameters

Parameter	Description
`Wp`	Passband corner frequency `Wp`, the cutoff frequency, is a scalar or a two-element vector with values between 0 and 1, with 1 corresponding to the normalized Nyquist frequency, π radians per sample.
`Ws`	Stopband corner frequency `Ws`, is a scalar or a two-element vector with values between 0 and 1, with 1 corresponding to the normalized Nyquist frequency.
`Rp`	Passband ripple, in decibels. This value is the maximum permissible passband loss in decibels.
`Rs`	Stopband attenuation, in decibels. This value is the number of decibels the stopband is down from the passband.

Use the following guide to specify filters of different types.

Filter Type Stopband and Passband Specifications

Filter Type	Stopband and Passband Conditions	Stopband	Passband
Lowpass	`Wp` < `Ws`, both scalars	`(Ws,1)`	`(0,Wp)`
Highpass	`Wp` > `Ws`, both scalars	`(0,Ws)`	`(Wp,1)`
Bandpass	The interval specified by `Ws` contains the one specified by `Wp` (`Ws(1) < Wp(1) < Wp(2) < Ws(2)`).	`(0,Ws(1))` and `(Ws(2),1)`	`(Wp(1),Wp(2))`
Bandstop	The interval specified by `Wp` contains the one specified by `Ws` (`Wp(1) < Ws(1) < Ws(2) < Wp(2)`).	`(0,Wp(1))` and `(Wp(2),1)`	`(Ws(1),Ws(2))`

If your filter specifications call for a bandpass or bandstop filter with unequal ripple in each of the passbands or stopbands, design separate lowpass and highpass filters according to the specifications in this table, and cascade the two filters together.

Analog Domain

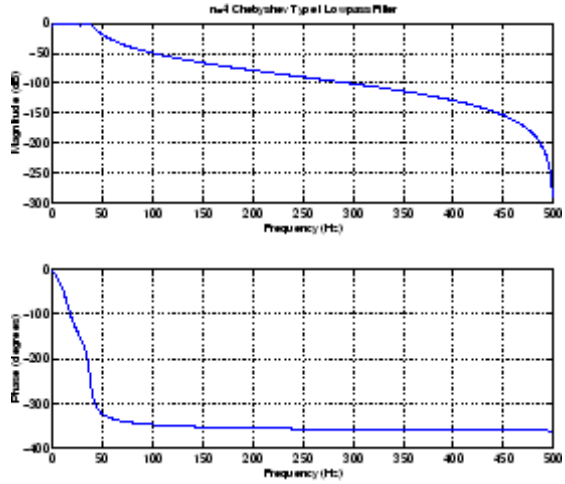
`[n,Wp] = cheb1ord(Wp,Ws,Rp,Rs,'s')` finds the minimum order `n` and cutoff frequencies `Wp` for an analog Chebyshev Type I filter. You specify the frequencies `Wp` and `Ws` similar to those described in the Description of Stopband and Passband Filter Parameters on page 10-71 table above, only in this case you specify the frequency in radians per second, and the passband or the stopband can be infinite.

Use `cheb1ord` for lowpass, highpass, bandpass, and bandstop filters as described in the Filter Type Stopband and Passband Specifications on page 10-72 table above.

Examples

For data sampled at 1000 Hz, design a lowpass filter with less than 3 dB of ripple in the passband defined from 0 to 40 Hz and at least 60 dB of ripple in the stopband defined from 150 Hz to the Nyquist frequency (500 Hz):

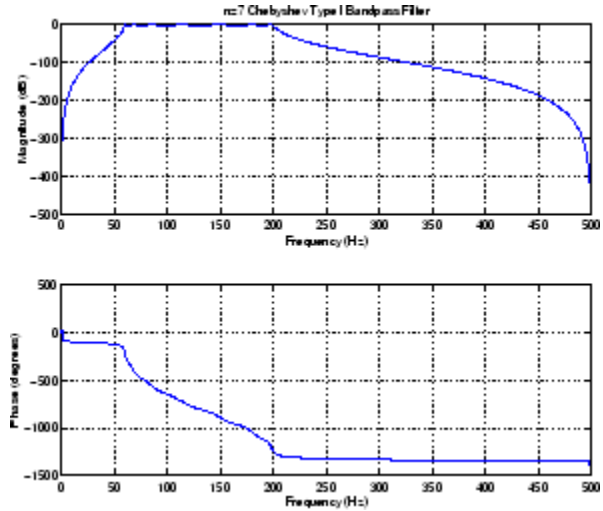
```
Wp = 40/500; Ws = 150/500;
Rp = 3; Rs = 60;
[n,Wp] = cheb1ord(Wp,Ws,Rp,Rs)
n =
     4
Wp =
    0.0800
[b,a] = cheby1(n,Rp,Wp);
freqz(b,a,512,1000);
title('n=4 Chebyshev Type I Lowpass Filter')
```



Next design a bandpass filter with a passband of 60 Hz to 200 Hz, with less than 3 dB of ripple in the passband, and 40 dB attenuation in the stopbands that are 50 Hz wide on both sides of the passband:

```
Wp = [60 200]/500; Ws = [50 250]/500;
Rp = 3; Rs = 40;
```

```
[n,Wp] = cheb1ord(Wp,Ws,Rp,Rs)
n =
     7
Wp =
    0.1200    0.4000
[b,a] = cheby1(n,Rp,Wp);
freqz(b,a,512,1000);
title('n=7 Chebyshev Type I Bandpass Filter')
```



Algorithm

`cheb1ord` uses the Chebyshev lowpass filter order prediction formula described in [1]. The function performs its calculations in the analog domain for both analog and digital cases. For the digital case, it converts the frequency parameters to the *s*-domain before the order and natural frequency estimation process, and then converts them back to the *z*-domain.

`cheb1ord` initially develops a lowpass filter prototype by transforming the passband frequencies of the desired filter to 1 rad/s (for low- or highpass filters) or to -1 and 1 rad/s (for bandpass or bandstop filters).

It then computes the minimum order required for a lowpass filter to meet the stopband specification.

References

[1] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing,* Englewood Cliffs, NJ: Prentice-Hall, 1975. Pg. 241.

See Also

`buttord`, `cheby1`, `cheb2ord`, `ellipord`, `kaiserord`

cheb2ap

Purpose

Chebyshev Type II analog lowpass filter prototype

Syntax

```
[z,p,k] = cheb2ap(n,Rs)
```

Description

`[z,p,k] = cheb2ap(n,Rs)` finds the zeros, poles, and gain of an order `n` Chebyshev Type II analog lowpass filter prototype with stopband ripple `Rs` dB down from the passband peak value. `cheb2ap` returns the zeros and poles in length `n` column vectors `z` and `p` and the gain in scalar `k`. If `n` is odd, `z` is length `n-1`. The transfer function is

$$H(s) = \frac{z(s)}{p(s)} = k\frac{(s-z(1))(s-z(2))\cdots(s-z(n))}{(s-p(1))(s-p(2))\cdots(s-p(n))}$$

Chebyshev Type II filters are monotonic in the passband and equiripple in the stopband. The pole locations are the inverse of the pole locations of `cheb1ap`, whose poles are evenly spaced about an ellipse in the left half plane. The Chebyshev Type II stopband edge angular frequency ω_0 is set to 1 for a normalized result. This is the frequency at which the stopband begins and the filter has magnitude response of $10^{-Rs/20}$.

Algorithm

Chebyshev Type II filters are sometimes called *inverse Chebyshev* filters because of their relationship to Chebyshev Type I filters. The `cheb2ap` function is a modification of the Chebyshev Type I prototype algorithm:

1. `cheb2ap` replaces the frequency variable ω with $1/\omega$, turning the lowpass filter into a highpass filter while preserving the performance at $\omega = 1$.
2. `cheb2ap` subtracts the filter transfer function from unity.

References

[1] Parks, T.W., and C.S. Burrus. *Digital Filter Design,* New York: John Wiley & Sons, 1987. Chapter 7.

See Also

`besselap`, `buttap`, `cheb1ap`, `cheby2`, `ellipap`

Purpose

Chebyshev Type II filter order

Syntax

```
[n,Ws] = cheb2ord(Wp,Ws,Rp,Rs)
[n,Ws] = cheb2ord(Wp,Ws,Rp,Rs,'s')
```

Description

`cheb2ord` calculates the minimum order of a digital or analog Chebyshev Type II filter required to meet a set of filter design specifications.

Digital Domain

`[n,Ws] = cheb2ord(Wp,Ws,Rp,Rs)` returns the lowest order `n` of the Chebyshev Type II filter that loses no more than `Rp` dB in the passband and has at least `Rs` dB of attenuation in the stopband. The scalar (or vector) of corresponding cutoff frequencies `Ws`, is also returned. Use the output arguments `n` and `Ws` in `cheby2`.

Choose the input arguments to specify the stopband and passband according to the following table.

Description of Stopband and Passband Filter Parameters

Parameter	Description
`Wp`	Passband corner frequency `Wp`, the cutoff frequency, is a scalar or a two-element vector with values between 0 and 1, with 1 corresponding to the normalized Nyquist frequency, π radians per sample.
`Ws`	Stopband corner frequency `Ws`, is a scalar or a two-element vector with values between 0 and 1, with 1 corresponding to the normalized Nyquist frequency.
`Rp`	Passband ripple, in decibels. This value is the maximum permissible passband loss in decibels.
`Rs`	Stopband attenuation, in decibels. This value is the number of decibels the stopband is down from the passband.

Use the following guide to specify filters of different types.

Filter Type Stopband and Passband Specifications

Filter Type	Stopband and Passband Conditions	Stopband	Passband
Lowpass	`Wp` < `Ws`, both scalars	`(Ws,1)`	`(0,Wp)`
Highpass	`Wp` > `Ws`, both scalars	`(0,Ws)`	`(Wp,1)`
Bandpass	The interval specified by `Ws` contains the one specified by `Wp` (`Ws(1) < Wp(1) < Wp(2) < Ws(2)`).	`(0,Ws(1))` and `(Ws(2),1)`	`(Wp(1),Wp(2))`
Bandstop	The interval specified by `Wp` contains the one specified by `Ws` (`Wp(1) < Ws(1) < Ws(2) < Wp(2)`).	`(0,Wp(1))` and `(Wp(2),1)`	`(Ws(1),Ws(2))`

If your filter specifications call for a bandpass or bandstop filter with unequal ripple in each of the passbands or stopbands, design separate lowpass and highpass filters according to the specifications in this table, and cascade the two filters together.

Analog Domain

`[n,Ws] = cheb2ord(Wp,Ws,Rp,Rs,'s')` finds the minimum order `n` and cutoff frequencies `Ws` for an analog Chebyshev Type II filter. You specify the frequencies `Wp` and `Ws` similar to those described in the Description of Stopband and Passband Filter Parameters on page 10-77 table above, only in this case you specify the frequency in radians per second, and the passband or the stopband can be infinite.

Use `cheb2ord` for lowpass, highpass, bandpass, and bandstop filters as described in the Filter Type Stopband and Passband Specifications on page 10-78 table above.

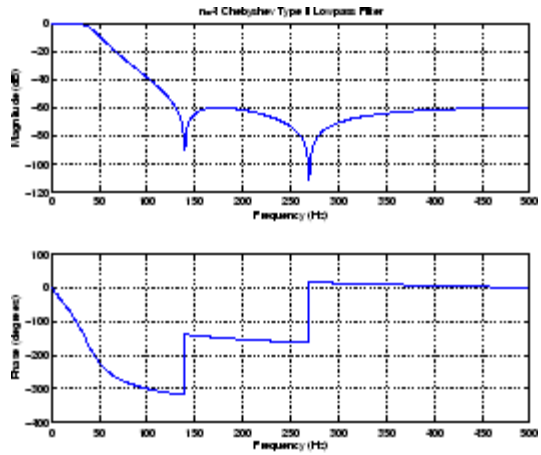
Examples

Example 1

For data sampled at 1000 Hz, design a lowpass filter with less than 3 dB of ripple in the passband defined from 0 to 40 Hz, and at least

60 dB of attenuation in the stopband defined from 150 Hz to the Nyquist frequency (500 Hz):

```
Wp = 40/500; Ws = 150/500;
Rp = 3; Rs = 60;
[n,Ws] = cheb2ord(Wp,Ws,Rp,Rs)
n =
     4
Ws =
    0.3000
[b,a] = cheby2(n,Rs,Ws);
freqz(b,a,512,1000);
title('n=4 Chebyshev Type II Lowpass Filter')
```

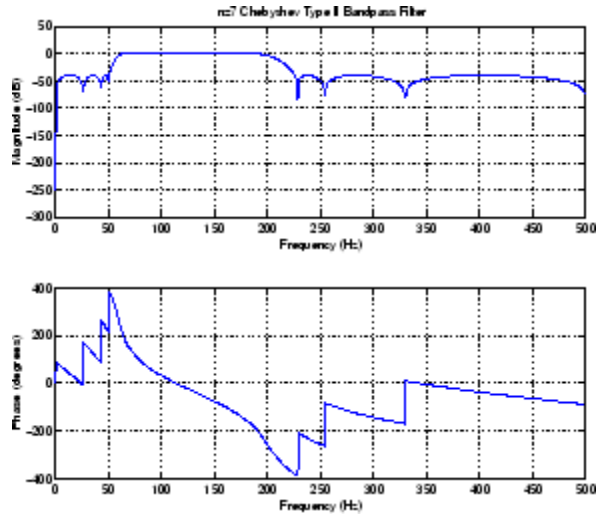


Example 2

Next design a bandpass filter with a passband of 60 Hz to 200 Hz, with less than 3 dB of ripple in the passband, and 40 dB attenuation in the stopbands that are 50 Hz wide on both sides of the passband:

```
Wp = [60 200]/500; Ws = [50 250]/500;
Rp = 3; Rs = 40;
[n,Ws] = cheb2ord(Wp,Ws,Rp,Rs)
```

```
n =
     7
Ws =
    0.1000    0.5000
[b,a] = cheby2(n,Rs,Ws);
freqz(b,a,512,1000)
title('n=7 Chebyshev Type II Bandpass Filter')
```



Algorithm

`cheb2ord` uses the Chebyshev lowpass filter order prediction formula described in [1]. The function performs its calculations in the analog domain for both analog and digital cases. For the digital case, it converts the frequency parameters to the *s*-domain before the order and natural frequency estimation process, and then converts them back to the *z*-domain.

`cheb2ord` initially develops a lowpass filter prototype by transforming the stopband frequencies of the desired filter to 1 rad/s (for low- and highpass filters) and to -1 and 1 rad/s (for bandpass and bandstop filters). It then computes the minimum order required for a lowpass filter to meet the passband specification.

References

[1] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing,* Englewood Cliffs, NJ: Prentice-Hall, 1975. Pg. 241.

See Also

`buttord`, `cheb1ord`, `cheby2`, `ellipord`, `kaiserord`

Purpose

Chebyshev window

Syntax

```
w = chebwin(L,r)
```

Description

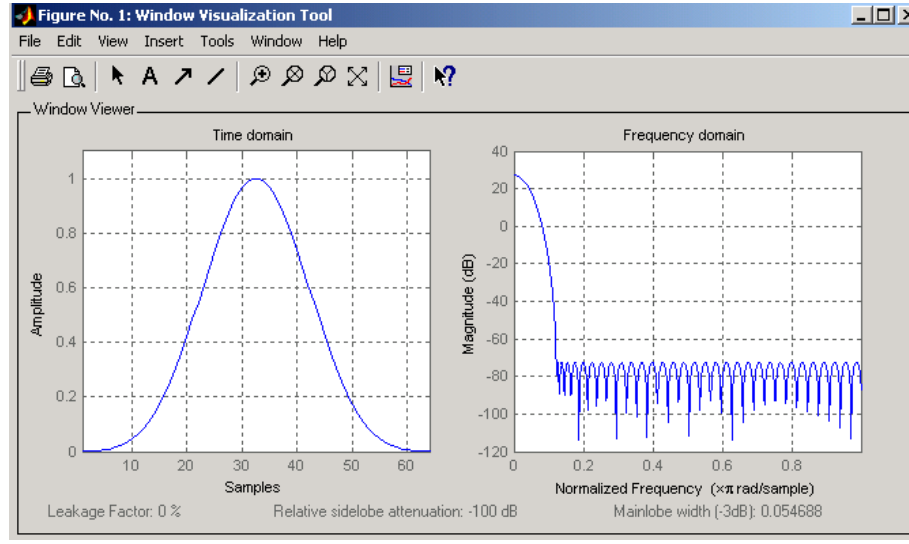
`w = chebwin(L,r)` returns the column vector `w` containing the length `L` Chebyshev window whose Fourier transform sidelobe magnitude is `r` dB below the mainlobe magnitude. The default value for `r` is 100.0 dB.

Note If you specify a one-point window (set `L=1`), the value `1` is returned.

Examples

Create a 64-point Chebyshev window with 100 dB of sidelobe attenuation and display the result using WVTool:

```
L=64;
wvtool(chebwin(L))
```



Algorithm

An artifact of the equiripple design method used in `chebwin` is the presence of impulses at the endpoints of the time-domain response. This is due to the constant-level sidelobes in the frequency domain. The magnitude of the impulses are on the order of the size of the spectral sidelobes. If the sidelobes are large, the effect at the endpoints may be significant. For more information on this effect, see [2].

References

[1] *IEEE Programs for Digital Signal Processing*. IEEE Press. New York: John Wiley & Sons, 1979. Program 5.2.

[2] Harris, Fredric J. *Multirate Signal Processing for Communication Systems,* New Jersey: Prentice Hall PTR, 2004, pp. 60-64.

See Also

`gausswin`, `kaiser`, `tukeywin`, `window`, `wintool`, `wvtool`

Purpose

Chebyshev Type I filter design (passband ripple)

Syntax

```
[z,p,k] = cheby1(n,R,Wp)
[z,p,k] = cheby1(n,R,Wp,'ftype')
[b,a] = cheby1(n,R,Wp)
[b,a] = cheby1(n,R,Wp,'ftype')
[A,B,C,D] = cheby1(n,R,Wp)
[A,B,C,D] = cheby1(n,R,Wp,'ftype')
[z,p,k] = cheby1(n,R,Wp,'s')
[z,p,k] = cheby1(n,R,Wp,'ftype','s')
[b,a] = cheby1(n,R,Wp,'s')
[b,a] = cheby1(n,R,Wp, 'ftype','s')
[A,B,C,D] = cheby1(n,R,Wp,'s')
[A,B,C,D] = cheby1(n,R,Wp,'ftype','s')
```

Description

`cheby1` designs lowpass, bandpass, highpass, and bandstop digital and analog Chebyshev Type I filters. Chebyshev Type I filters are equiripple in the passband and monotonic in the stopband. Type I filters roll off faster than type II filters, but at the expense of greater deviation from unity in the passband.

Digital Domain

`[z,p,k] = cheby1(n,R,Wp)` designs an order `n` `Chebyshev` lowpass digital Chebyshev filter with normalized passband edge frequency `Wp` and `R` dB of peak-to-peak ripple in the passband. It returns the zeros and poles in length `n` column vectors `z` and `p` and the gain in the scalar `k`.

`[z,p,k] = cheby1(n,R,Wp,'ftype')` designs a highpass, lowpass, or bandstop filter, where the string *`'ftype'`* is one of the following:

- `'high'` for a highpass digital filter with normalized passband edge frequency `Wp`
- `'low'` for a lowpass digital filter with normalized passband edge frequency `Wp`
- `'stop'` for an order `2*n` bandstop digital filter if `Wp` is a two-element vector, `Wp = [w1 w2]`. The stopband is $w1 < \omega < w2$.

Normalized passband edge frequency is the frequency at which the magnitude response of the filter is equal to `-R` dB. For `cheby1`, the normalized passband edge frequency `Wp` is a number between 0 and 1, where 1 corresponds to half the sample rate, π radians per sample. Smaller values of passband ripple `R` lead to wider transition widths (shallower rolloff characteristics).

If `Wp` is a two-element vector, `Wp = [w1 w2]`, `cheby1` returns an order `2*n` bandpass filter with passband `w1 < ω < w2`.

With different numbers of output arguments, `cheby1` directly obtains other realizations of the filter. To obtain the transfer function form, use two output arguments as shown below.

Note See "Limitations" on page 10-88 for information about numerical issues that affect forming the transfer function.

`[b,a] = cheby1(n,R,Wp)` designs an order `n` `Chebyshev` lowpass digital Chebyshev filter with normalized passband edge frequency `Wp` and `R` dB of peak-to-peak ripple in the passband. It returns the filter coefficients in the length `n+1` row vectors `b` and `a`, with coefficients in descending powers of *z*.

$$H(z) = \frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \cdots + b(n+1)z^{-n}}{1 + a(2)z^{-1} + \cdots + a(n+1)z^{-n}}$$

`[b,a] = cheby1(n,R,Wp,'ftype')` designs a highpass, lowpass, or bandstop filter, where the string `'ftype'` is `'high'`, `'low'`, or `'stop'`, as described above.

To obtain state-space form, use four output arguments as shown below:

`[A,B,C,D] = cheby1(n,R,Wp)` or

`[A,B,C,D] = cheby1(n,R,Wp,'ftype')` where `A`, `B`, `C`, and `D` are

$$x[n+1] = Ax[n] + Bu[n]$$
$$y[n] = Cx[n] + Du[n]$$

and u is the input, x is the state vector, and y is the output.

Analog Domain

`[z,p,k] = cheby1(n,R,Wp,'s')` designs an order `n` lowpass analog Chebyshev Type I filter with angular passband edge frequency `Wp` rad/s. It returns the zeros and poles in length `n` or `2*n` column vectors `z` and `p` and the gain in the scalar `k`.

Angular passband edge frequency is the frequency at which the magnitude response of the filter is `-R` dB. For `cheby1`, the angular passband edge frequency `Wp` must be greater than 0 rad/s.

If `Wp` is a two-element vector `Wp = [w1 w2]` with `w1 < w2`, then `cheby1(n,R,Wp,'s')` returns an order `2*n` bandpass analog filter with passband `w1 < ω< w2`.

`[z,p,k] = cheby1(n,R,Wp,'ftype','s')` designs a highpass, lowpass, or bandstop filter, where the string *`'ftype'`* is `'high'`, `'low'`, or `'stop'`, as described above.

You can supply different numbers of output arguments for `cheby1` to directly obtain other realizations of the analog filter. To obtain the transfer function form, use two output arguments as shown below.

`[b,a] = cheby1(n,R,Wp,'s')` designs an order `n` lowpass analog Chebyshev Type I filter with angular passband edge frequency `Wp` rad/s. It returns the filter coefficients in length `n+1` row vectors `b` and `a`, in descending powers of s, derived from the transfer function

$$H(s) = \frac{B(s)}{A(s)} = \frac{b(1)s^n + b(2)s^{n-1} + \cdots + b(n+1)}{s^n + a(2)s^{n-1} + \cdots + a(n+1)}$$

`[b,a] = cheby1(n,R,Wp, 'ftype','s')` designs a highpass, lowpass, or bandstop filter, where the string *`'ftype'`* is `'high'`, `'low'`, or `'stop'`, as described above.

To obtain state-space form, use four output arguments as shown below:

`[A,B,C,D] = cheby1(n,R,Wp,'s')` or

`[A,B,C,D] = cheby1(n,R,Wp,'ftype','s')` where `A`, `B`, `C`, and `D` are defined as

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

and u is the input, x is the state vector, and y is the output.

Examples

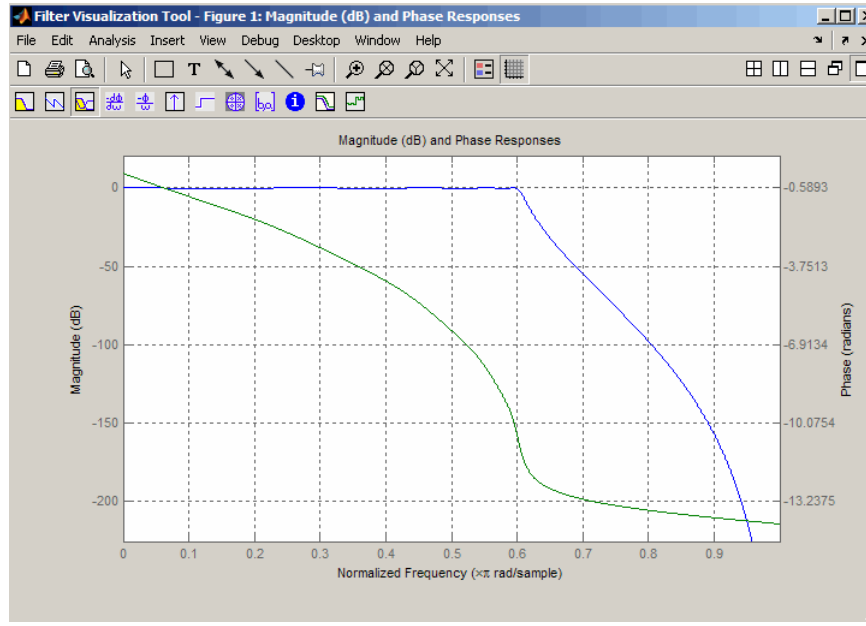
Lowpass Filter

For data sampled at 1000 Hz, design a 9th-order lowpass Chebyshev Type I filter with 0.5 dB of ripple in the passband and a passband edge frequency of 300 Hz, which corresponds to a normalized value of 0.6:

```
[z,p,k] = cheby1(9,0.5,300/500);
[sos,g] = zp2sos(z,p,k);   % Convert to SOS form
Hd = dfilt.df2tsos(sos,g);  % Create a dfilt object
h = fvtool(Hd)      % Plot magnitude response
set(h,'Analysis','freq')   % Display frequency response
```

The frequency response of the filter is

```
freqz(b,a,512,1000)
```

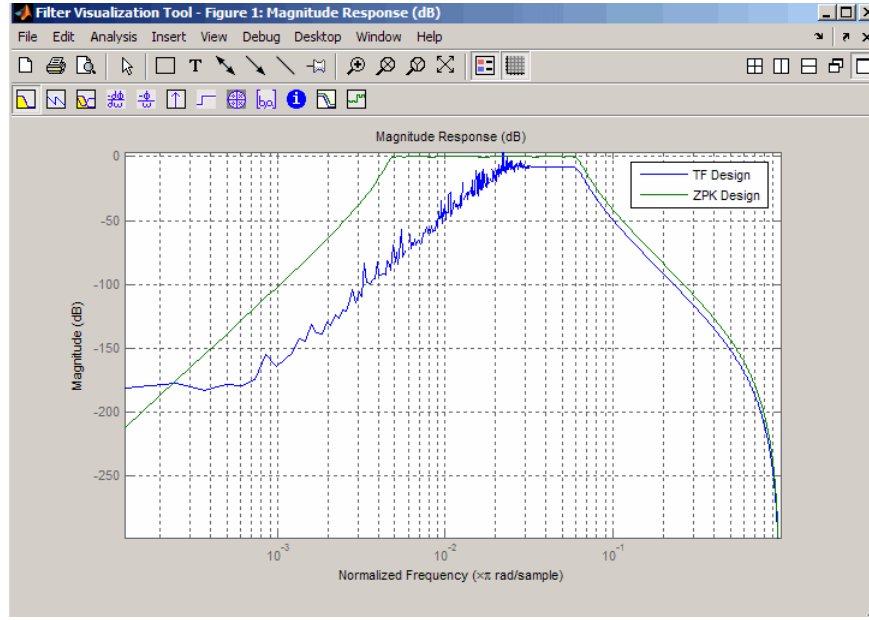


Limitations

In general, you should use the `[z,p,k]` syntax to design IIR filters. To analyze or implement your filter, you can then use the `[z,p,k]` output with `zp2sos` and an sos `dfilt` structure. For higher order filters (possibly starting as low as order 8), numerical problems due to roundoff errors may occur when forming the transfer function using the `[b,a]` syntax. The following example illustrates this limitation:

```
n = 6;
r = 0.1;
Wn = ([2.5e6 29e6]/500e6);
ftype = 'bandpass';

% Transfer Function design
[b,a] = cheby1(n,r,Wn,ftype);
h1=dfilt.df2(b,a);      % This is an unstable filter.
```

```
% Zero-Pole-Gain design
[z, p, k] = cheby1(n,r, Wn,ftype);
[sos,g]=zp2sos(z,p,k);
h2=dfilt.df2sos(sos,g);

% Plot and compare the results
hfvt=fvtool(h1,h2,'FrequencyScale','log');
legend(hfvt,'TF Design','ZPK Design')
```



Algorithm

`cheby1` uses a five-step algorithm:

1. It finds the lowpass analog prototype poles, zeros, and gain using the `cheb1ap` function.

2. It converts the poles, zeros, and gain into state-space form.

3. It transforms the lowpass filter into a bandpass, highpass, or bandstop filter with desired cutoff frequencies, using a state-space transformation.

4. For digital filter design, `cheby1` uses `bilinear` to convert the analog filter into a digital filter through a bilinear transformation with frequency prewarping. Careful frequency adjustment guarantees that the analog filters and the digital filters will have the same frequency response magnitude at `Wp` or `w1` and `w2`.

5. It converts the state-space filter back to transfer function or zero-pole-gain form, as required.

See Also

`besself`, `butter`, `cheb1ap`, `cheb1ord`, `cheby2`, `ellip`

Purpose

Chebyshev Type II filter design (stopband ripple)

Syntax

```
[z,p,k] = cheby2(n,R,Wst)
[z,p,k] = cheby2(n,R,Wst,'ftype')
[b,a] = cheby2(n,R,Wst)
[b,a] = cheby2(n,R,Wst,'ftype')
[A,B,C,D] = cheby2(n,R,Wst)
[A,B,C,D] = cheby2(n,R,Wst,'ftype')
[z,p,k] = cheby2(n,R,Wst,'s')
[z,p,k] = cheby2(n,R,Wst,'ftype','s')
[b,a] = cheby2(n,R,Wst,'s')
[b,a] = cheby2(n,R,Wst,'ftype','s')
[A,B,C,D] = cheby2(n,R,Wst,'s')
[A,B,C,D] = cheby2(n,R,Wst,'ftype','s')
```

Description

`cheby2` designs lowpass, highpass, bandpass, and bandstop digital and analog Chebyshev Type II filters. Chebyshev Type II filters are monotonic in the passband and equiripple in the stopband. Type II filters do not roll off as fast as type I filters, but are free of passband ripple.

Digital Domain

`[z,p,k] = cheby2(n,R,Wst)` designs an order `n` lowpass digital Chebyshev Type II filter with normalized stopband edge frequency `Wst` and stopband ripple `R` dB down from the peak passband value. It returns the zeros and poles in length `n` column vectors `z` and `p` and the gain in the scalar `k`.

Normalized stopband edge frequency is the beginning of the stopband, where the magnitude response of the filter is equal to `-R` dB. For `cheby2`, the normalized stopband edge frequency `Wst` is a number between 0 and 1, where 1 corresponds to half the sample rate. Larger values of stopband attenuation R lead to wider transition widths (shallower rolloff characteristics).

If `Wst` is a two-element vector, `Wst = [w1 w2]`, `cheby2` returns an order `2*n` bandpass filter with passband `w1` < ω < `w2`.

`[z,p,k] = cheby2(n,R,Wst,'ftype')` designs a highpass, lowpass, or bandstop filter, where the string `'ftype'` is one of the following:

- `'high'` for a highpass digital filter with normalized stopband edge frequency `Wst`
- `'low'` for a lowpass digital filter with normalized stopband edge frequency `Wst`
- `'stop'` for an order `2*n` bandstop digital filter if `Wst` is a two-element vector, `Wst = [w1 w2]`. The stopband is `w1` < ω < `w2`.

With different numbers of output arguments, `cheby2` directly obtains other realizations of the filter. To obtain the transfer function form, use two output arguments as shown below.

Note See "Limitations" on page 10-95 below for information about numerical issues that affect forming the transfer function.

`[b,a] = cheby2(n,R,Wst)` designs an order `n` lowpass digital Chebyshev Type II filter with normalized stopband edge frequency `Wst` and stopband ripple `R` dB down from the peak passband value. It returns the filter coefficients in the length `n+1` row vectors `b` and `a`, with coefficients in descending powers of *z*.

$$H(z) = \frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \cdots + b(n+1)z^{-n}}{1 + a(2)z^{-1} + \cdots + a(n+1)z^{-n}}$$

`[b,a] = cheby2(n,R,Wst,'ftype')` designs a highpass, lowpass, or bandstop filter, where the string *'ftype'* is `'high'`, `'low'`, or `'stop'`, as described above.

To obtain state-space form, use four output arguments as shown below.

`[A,B,C,D] = cheby2(n,R,Wst)` or

`[A,B,C,D] = cheby2(n,R,Wst,'ftype')` where `A`, `B`, `C`, and `D` are

$$x[n+1] = Ax[n] + Bu[n]$$
$$y[n] = Cx[n] + Du[n]$$

and u is the input, x is the state vector, and y is the output.

Analog Domain

`[z,p,k] = cheby2(n,R,Wst,'s')` designs an order `n` lowpass analog Chebyshev Type II filter with angular stopband edge frequency `Wst` rad/s.. It returns the zeros and poles in length `n` or `2*n` column vectors `z` and `p` and the gain in the scalar `k`.

Angular stopband edge frequency is the frequency at which the magnitude response of the filter is equal to `-R` dB. For `cheby2`, the angular stopband edge frequency `Wst` must be greater than 0 rad/s.

If `Wst` is a two-element vector `Wst = [w1 w2]` with `w1 < w2`, then `cheby2(n,R,Wst,'s')` returns an order `2*n` bandpass analog filter with passband `w1` < ω< `w2`.

`[z,p,k] = cheby2(n,R,Wst,'ftype','s')` designs a highpass, lowpass, or bandstop filter, where the string *`'ftype'`* is `'high'`, `'low'`, or `'stop'`, as described above.

With different numbers of output arguments, `cheby2` directly obtains other realizations of the analog filter. To obtain the transfer function form, use two output arguments as shown below:

`[b,a] = cheby2(n,R,Wst,'s')` designs an order `n` lowpass analog Chebyshev Type II filter with angular stopband edge frequency `Wst` rad/s.. It returns the filter coefficients in the length `n+1` row vectors `b` and `a`, with coefficients in descending powers of s, derived from the transfer function.

$$H(s) = \frac{B(s)}{A(s)} = \frac{b(1)s^n + b(2)s^{n-1} + \cdots + b(n+1)}{s^n + a(2)s^{n-1} + \cdots + a(n+1)}$$

`[b,a] = cheby2(n,R,Wst,'ftype','s')` designs a highpass , lowpass, or bandstop filter, where the string *`'ftype'`* is `'high'`, `'low'`, or `'stop'`, as described above.

To obtain state-space form, use four output arguments as shown below:

`[A,B,C,D] = cheby2(n,R,Wst,'s')` or

`[A,B,C,D] = cheby2(n,R,Wst,'ftype','s')` where `A`, `B`, `C`, and `D` are

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

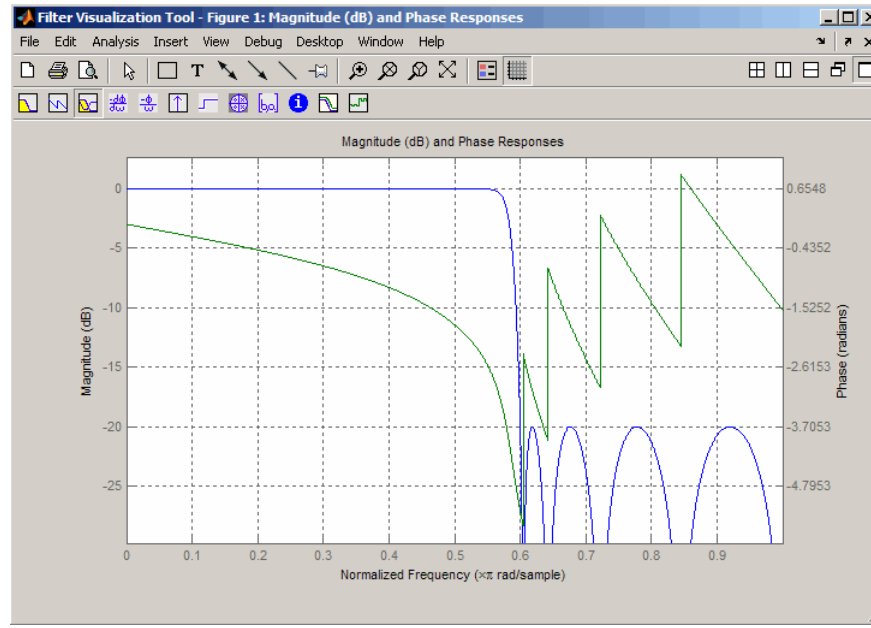
and u is the input, x is the state vector, and y is the output.

Examples

Lowpass Filter

For data sampled at 1000 Hz, design a ninth-order lowpass Chebyshev Type II filter with stopband attenuation 20 dB down from the passband and a stopband edge frequency of 300 Hz, which corresponds to a normalized value of 0.6:

```
[z,p,k] = cheby2(9,20,300/500);
[sos,g] = zp2sos(z,p,k);   % Convert to SOS form
Hd = dfilt.df2tsos(sos,g);  % Create a dfilt object
h = fvtool(Hd);      % Plot magnitude response
set(h,'Analysis','freq')   % Display frequency response
```

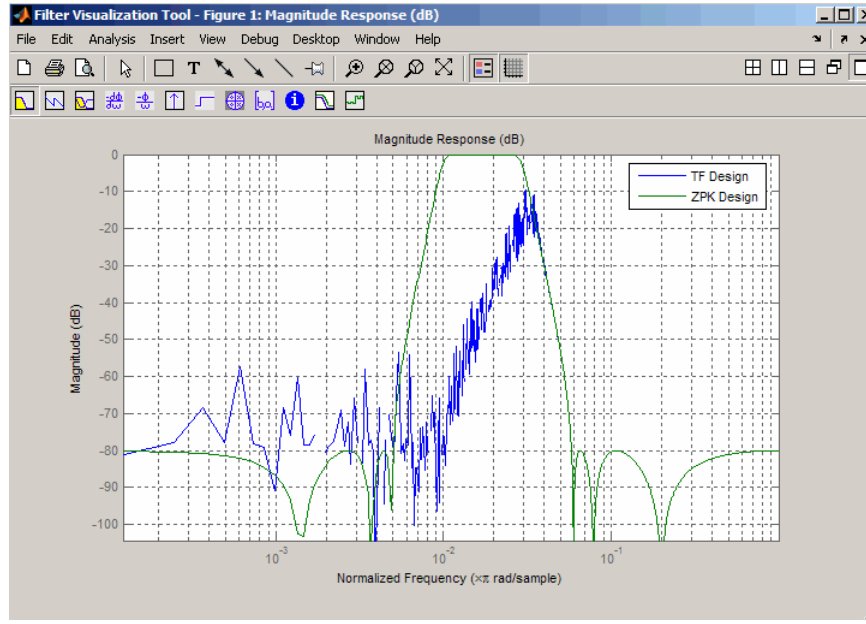


Limitations

In general, you should use the `[z,p,k]` syntax to design IIR filters. To analyze or implement your filter, you can then use the `[z,p,k]` output with `zp2sos` and an sos `dfilt` structure. For higher order filters (possibly starting as low as order 8), numerical problems due to roundoff errors may occur when forming the transfer function using the `[b,a]` syntax. The following example illustrates this limitation:

```
n = 6;
r = 80;
Wn = [2.5e6 29e6]/500e6;
ftype = 'bandpass';

% Transfer Function design
[b,a] = cheby2(n,r,Wn,ftype);
h1=dfilt.df2(b,a);      % This is an unstable filter.
```

```
% Zero-Pole-Gain design
[z, p, k] = cheby2(n,r,Wn,ftype);
[sos,g]=zp2sos(z,p,k);
h2=dfilt.df2sos(sos,g);

% Plot and compare the results
hfvt=fvtool(h1,h2,'FrequencyScale','log');
legend(hfvt,'TF Design','ZPK Design')
```



Algorithm

`cheby2` uses a five-step algorithm:

1. It finds the lowpass analog prototype poles, zeros, and gain using the `cheb2ap` function.

2. It converts poles, zeros, and gain into state-space form.

3 It transforms the lowpass filter into a bandpass, highpass, or bandstop filter with desired cutoff frequencies, using a state-space transformation.

4 For digital filter design, `cheby2` uses `bilinear` to convert the analog filter into a digital filter through a bilinear transformation with frequency prewarping. Careful frequency adjustment guarantees that the analog filters and the digital filters will have the same frequency response magnitude at `Wst` or `w1` and `w2`.

5 It converts the state-space filter back to transfer function or zero-pole-gain form, as required.

See Also

`besself`, `butter`, `cheb2ap`, `cheb1ord`, `cheby1`, `ellip`

Purpose

Swept-frequency cosine

Syntax

```
y = chirp(t,f0,t1,f1)
y = chirp(t,f0,t1,f1,'method')
y = chirp(t,f0,t1,f1,'method',phi)
y = chirp(t,f0,t1,f1,'quadratic',phi,'shape')
```

Description

`y = chirp(t,f0,t1,f1)` generates samples of a linear swept-frequency cosine signal at the time instances defined in array `t`, where `f0` is the instantaneous frequency at time 0, and `f1` is the instantaneous frequency at time `t1`. `f0` and `f1` are both in hertz. If unspecified, `f0` is e^{-6} for logarithmic chirp and 0 for all other methods, `t1` is `1`, and `f1` is `100`.

`y = chirp(t,f0,t1,f1,'method')` specifies alternative sweep method options, where *`method`* can be:

- `linear`, which specifies an instantaneous frequency sweep $f_i(t)$ given by

  $$f_i(t) = f_0 + \beta t$$

  where

  $$\beta = (f_1 - f_0)/t_1$$

  and the default value for f_0 is 0. ß ensures that the desired frequency breakpoint f_1 at time t_1 is maintained.

- `quadratic`, which specifies an instantaneous frequency sweep $f_i(t)$ given by

  $$f_i(t) = f_0 + \beta t^2$$

  where

  $$\beta = (f_1 - f_0)/t_1^2$$

  and the default value for f_0 is 0. If $f_0 > f_1$ (downsweep), the default shape is convex. If $f_0 < f_1$ (upsweep), the default shape is concave.

- `logarithmic` specifies an instantaneous frequency sweep $f_i(t)$ given by

  $$f_i(t) = f_0 \times \beta^t$$

  where

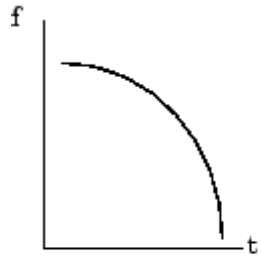
  $$\beta = \left(\frac{f_1}{f_0}\right)^{\frac{1}{t_1}}$$

  and the default value for f_0 is $1e^{-6}$. Both an upsweep ($f_1 > f_0$) and a downsweep ($f_0 > f_1$) of frequency is possible.

Each of the above methods can be entered as `'li'`, `'q'`, and `'lo'`, respectively.

`y = chirp(t,f0,t1,f1,'method',phi)` allows an initial phase `phi` to be specified in degrees. If unspecified, `phi` is `0`. Default values are substituted for empty or omitted trailing input arguments.

`y = chirp(t,f0,t1,f1,'quadratic',phi,'shape')` specifies the `shape` of the quadratic swept-frequency signal's spectrogram. `shape` is either `concave` or `convex`, which describes the shape of the parabola in the positive frequency axis. If `shape` is omitted, the default is convex for downsweep ($f_0 > f_1$) and is concave for upsweep ($f_0 < f_1$).



Convex downsweep shape

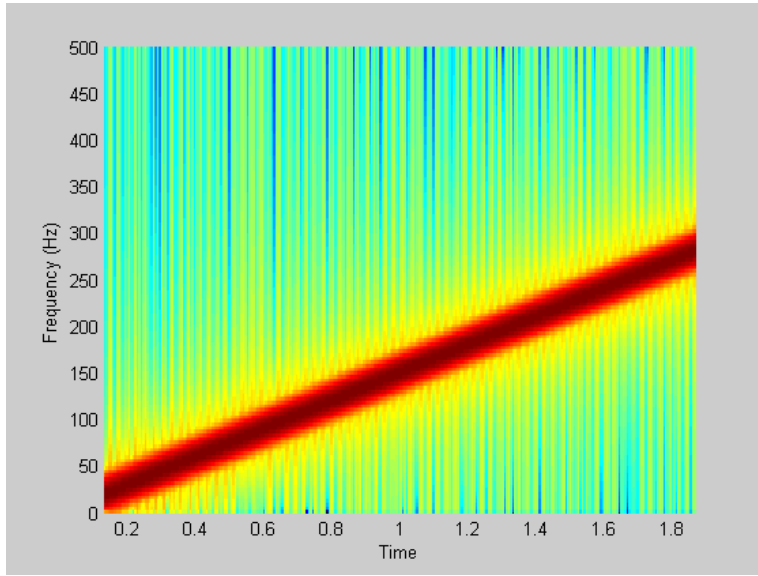
Concave upsweep shape

Examples

Example 1

Compute the spectrogram of a chirp with linear instantaneous frequency deviation:

```
t = 0:0.001:2;            % 2 secs @ 1kHz sample rate
y = chirp(t,0,1,150);     % Start @ DC,
                          %   cross 150Hz at t=1 sec
spectrogram(y,256,250,256,1E3,'yaxis')
```

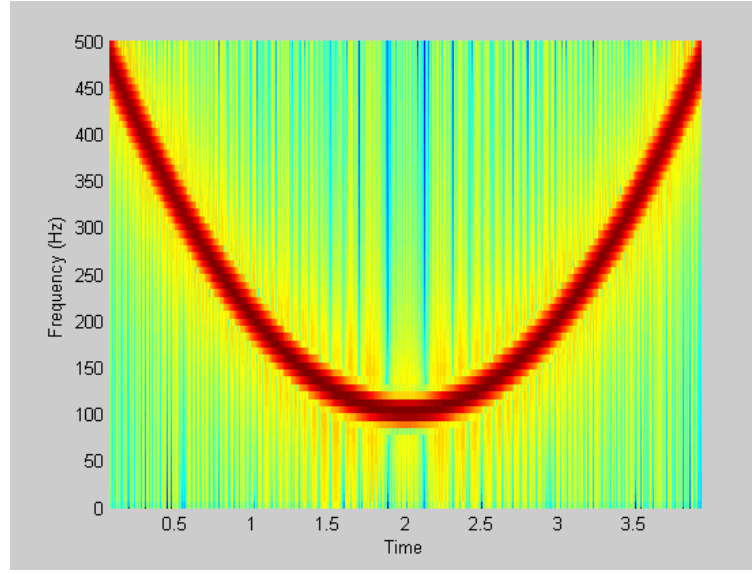


Example 2

Compute the spectrogram of a chirp with quadratic instantaneous frequency deviation:

```
% –2 secs @ 1kHz sample rate
t = -2:0.001:2;

% Start @ 100Hz, cross 200Hz at t=1 sec
y = chirp(t,100,1,200,'quadratic');
```

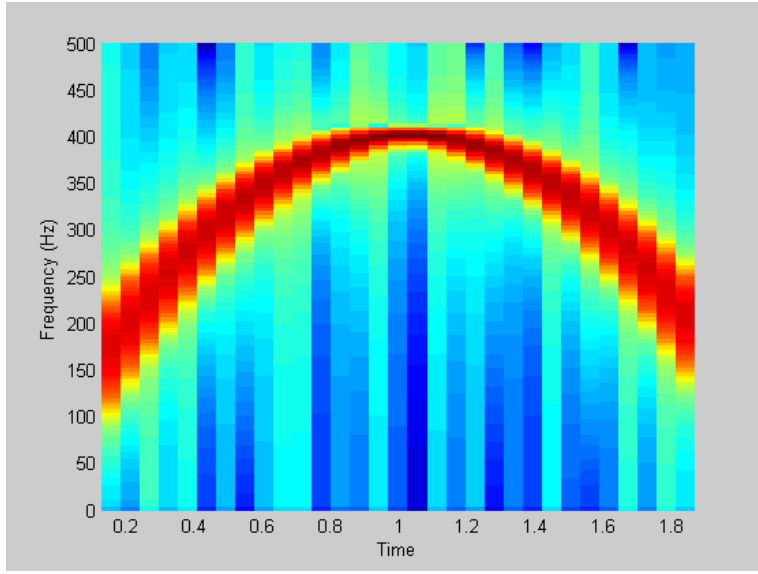
```
spectrogram(y,128,120,128,1E3,'yaxis')
```



Example 3

Compute the spectrogram of a convex quadratic chirp:

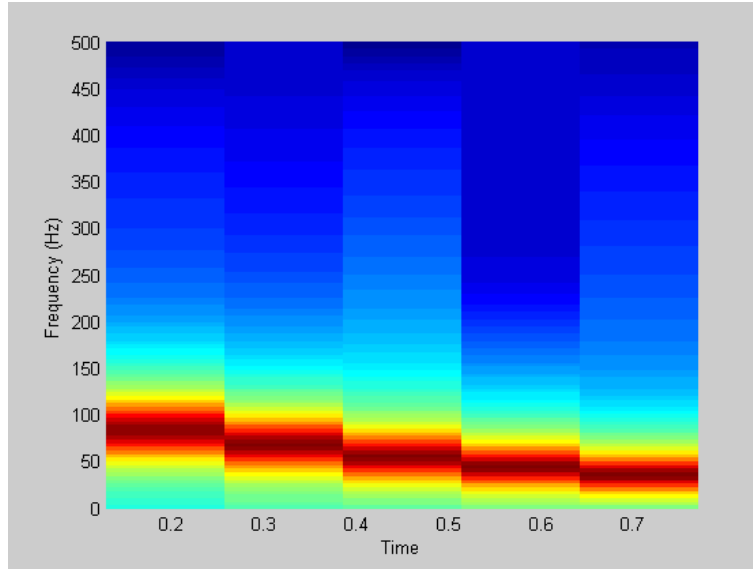
```
t = -1:0.001:1;          % +/-1 second @ 1kHz sample rate
fo = 100; f1 = 400;      % Start at 100Hz, go up to 400Hz
y = chirp(t,fo,1,f1,'q',[],'convex');
spectrogram(y,256,200,256,1000,'yaxis')
```



Example 4

Compute the spectrogram of a concave quadratic chirp:

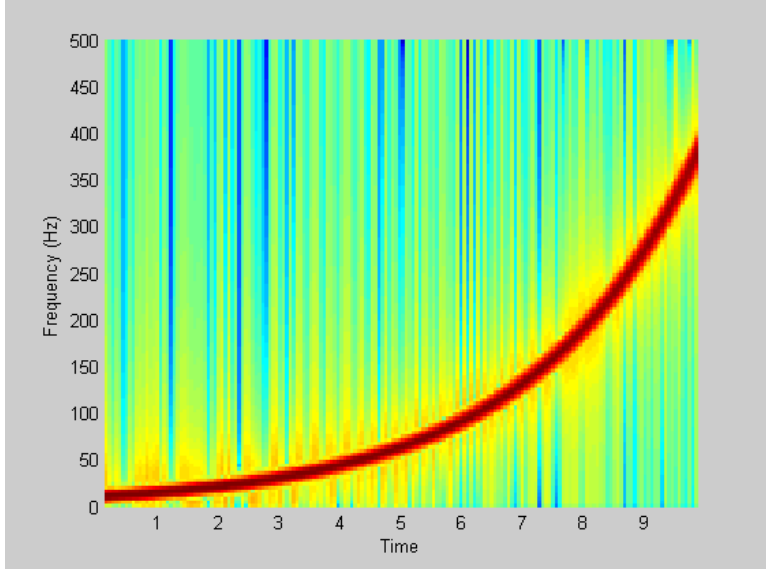
```
t = 0:0.001:1;       % 1 second @ 1kHz sample rate
fo = 100; f1 = 25;   % Start at 100Hz, go down to 25Hz
y = chirp(t,fo,1,f1,'q',[],'concave');
spectrogram(y,hanning(256),128,256,1000,'yaxis')
```



Example 5

Compute the spectrogram of a logarithmic chirp:

```
t = 0:0.001:10;       % 10 seconds @ 1kHz sample rate
fo = 10; f1 = 400;    % Start at 10Hz, go up to 400Hz
y = chirp(t,fo,10,f1,'logarithmic');
spectrogram(y,256,200,256,1000,'yaxis')
```



See Also

cos, diric, gauspuls, pulstran, rectpuls, sawtooth, sin, sinc, square, tripuls

Purpose

Convolution matrix

Syntax

```
A = convmtx(c,n)
A = convmtx(r,n)
```

Description

A *convolution matrix* is a matrix, formed from a vector, whose inner product with another vector is the convolution of the two vectors.

`A = convmtx(c,n)` where `c` is a length `m` column vector returns a matrix `A` of size `(m+n-1)`-by-`n`. The product of `A` and another column vector `x` of length `n` is the convolution of `c` with `x`.

`A = convmtx(r,n)` where `r` is a length `m` row vector returns a matrix `A` of size `n`-by-`(m+n-1)`. The product of `A` and another row vector `x` of length `n` is the convolution of `r` with `x`.

Examples

Generate a simple convolution matrix:

```
h = [1 2 3 2 1];
convmtx(h,7)
ans =
 1    2    3    2    1    0    0    0    0    0    0
 0    1    2    3    2    1    0    0    0    0    0
 0    0    1    2    3    2    1    0    0    0    0
 0    0    0    1    2    3    2    1    0    0    0
 0    0    0    0    1    2    3    2    1    0    0
 0    0    0    0    0    1    2    3    2    1    0
 0    0    0    0    0    0    1    2    3    2    1
```

Note that `convmtx` handles edge conditions by zero padding.

In practice, it is more efficient to compute convolution using

```
y = conv(c,x)
```

than by using a convolution matrix.

```
n = length(x);
y = convmtx(c,n)*x
```

Algorithm

`convmtx` uses the function `toeplitz` to generate the convolution matrix.

See Also

`conv`, `convn`, `conv2`, `dftmtx`

Purpose

Data matrix for autocorrelation matrix estimation

Syntax

```
X = corrmtx(x,m)
X = corrmtx(x,m,'method')
[X,R] = corrmtx(...)
```

Description

`X = corrmtx(x,m)` returns an (*n*+`m`)-by-(`m`+1) rectangular Toeplitz matrix `X`, such that `X'X` is a (biased) estimate of the autocorrelation matrix for the length *n* data vector `x`.

`X = corrmtx(x,m,'method')` computes the matrix `X` according to the method specified by the string '`method`':

- `'autocorrelation'`: (default) `X` is the (*n*+`m`)-by-(`m`+1) rectangular Toeplitz matrix that generates an autocorrelation estimate for the length *n* data vector `x`, derived using *prewindowed* and *postwindowed* data, based on an `m`th order prediction error model.
- `'prewindowed'`: `X` is the *n*-by-(`m`+1) rectangular Toeplitz matrix that generates an autocorrelation estimate for the length *n* data vector `x`, derived using *prewindowed* data, based on an `m`th order prediction error model.
- `'postwindowed'`: `X` is the *n*-by-(`m`+1) rectangular Toeplitz matrix that generates an autocorrelation estimate for the length *n* data vector `x`, derived using *postwindowed* data, based on an `m`th order prediction error model.
- `'covariance'`: `X` is the (*n*-`m`)-by-(`m`+1) rectangular Toeplitz matrix that generates an autocorrelation estimate for the length *n* data vector `x`, derived using *nonwindowed* data, based on an `m`th order prediction error model.
- `'modified'`: `X` is the 2(*n*-`m`)-by-(`m`+1) modified rectangular Toeplitz matrix that generates an autocorrelation estimate for the length *n* data vector `x`, derived using forward and backward prediction error estimates, based on an `m`th order prediction error model.

`[X,R] = corrmtx(...)` also returns the (`m`+1)-by-(`m`+1) autocorrelation matrix estimate `R`, calculated as `X'*X`.

Examples

```
randn('state',1); n=0:99;
s=exp(i*pi/2*n)+2*exp(i*pi/4*n)+exp(i*pi/3*n)+randn(1,100);
X=corrmtx(s,12,'mod');
```

Algorithm

The Toeplitz data matrix computed by `corrmtx` depends on the method you select. The matrix determined by the autocorrelation (default) method is given by the following matrix.

$$X = \begin{bmatrix} x(1) & \cdots & 0 \\ \vdots & \ddots & \vdots \\ x(m+1) & \cdots & x1) \\ \vdots & \ddots & \vdots \\ x(n-m) & \cdots & x(m+1) \\ \vdots & \ddots & \vdots \\ x(n) & \cdots & x(n-m) \\ \vdots & \ddots & \vdots \\ 0 & \cdots & x(n) \end{bmatrix}$$

In this matrix, m is the same as the input argument `m` to `corrmtx`, and n is `length(x)`. Variations of this matrix are used to return the output `X` of `corrmtx` for each method:

- `'autocorrelation'`: (default) `X` = X, above.
- `'prewindowed'`: `X` is the n-by-(m+1) submatrix of X that is given by the portion of X above the lower gray line.
- `'postwindowed'`: `X` is the n-by-(m+1) submatrix of X that is given by the portion of X below the upper gray line.
- ’`covariance`’: `X` is the (n-m)-by-(m+1) submatrix of X that is given by the portion of X between the two gray lines.
- `'modified'`: `X` is the 2(n-m)-by-(m+1) matrix X_{mod} shown below.

$$X_{\text{mod}} = \begin{bmatrix} x(m+1) & \cdots & x1) \\ \vdots & \ddots & \vdots \\ x(n-m) & \cdots & x(m+1) \\ \vdots & \ddots & \vdots \\ x(n) & \cdots & x(n-m) \\ x^*(1) & \cdots & x^*(m+1) \\ \vdots & \iddots & \vdots \\ x^*(m+1) & \cdots & x^*(n-m) \\ \vdots & \iddots & \vdots \\ x^*(n-m) & \cdots & x^*(n) \end{bmatrix}$$

References

[1] Marple, S.L. *Digital Spectral Analysis,*Englewood Cliffs, NJ, Prentice-Hall, 1987, pp. 216-223.

See Also

`peig`, `pmusic`, `rooteig`, `rootmusic`, `xcorr`

Purpose

Cross power spectral density

Syntax

```
Pxy = cpsd(x,y)
Pxy = cpsd(x,y,window)
Pxy = cpsd(x,y,window,noverlap)
[Pxy,W] = cpsd(x,y,window,noverlap,nfft)
[Pxy,F] = cpsd(x,y,window,noverlap,nfft,fs)
[...] = cpsd(...,'twosided')
cpsd(...)
```

Description

`Pxy = cpsd(x,y)` estimates the cross power spectral density `Pxy` of the discrete-time signals `x` and `y` using the Welch's averaged, modified periodogram method of spectral estimation. The cross power spectral density is the distribution of power per unit frequency and is defined as

$$P_{xy}(\omega) = \sum_{m=-\infty}^{\infty} R_{xy}(m) e^{-j\omega m}$$

The cross-correlation sequence is defined as

$$R_{xy}(m) = E\{x_{n+m} y^*_n\} = E\{x_n y^*_{n-m}\}$$

where x_n and y_n are jointly stationary random processes, $-\infty < n < \infty$, and $E\{\cdot\}$ is the expected value operator.

For real `x` and `y`, `cpsd` returns a one-sided CPSD and for complex `x` or `y`, it returns a two-sided CPSD.

`cpsd` uses the following default values:

Parameter	Description	Default Value
`nfft`	FFT length which determines the frequencies at which the PSD is estimated For real `x` and `y`, the length of `Pxy` is (`nfft`/2+1) if `nfft` is even or (`nfft`+1)/2 if `nfft` is odd. For complex `x` or `y`, the length of `Pxy` is `nfft`. If `nfft` is greater than the signal length, the data is zero-padded. If `nfft` is less than the signal length, the segment is wrapped using datawrap so that the length is equal to `nfft`.	Maximum of 256 or the next power of 2 greater than the length of each section of `x` or `y`
`fs`	Sampling frequency	1
`window`	Windowing function and number of samples to use for each section	Periodic Hamming window of length to obtain eight equal sections of `x` and `y`
`noverlap`	Number of samples by which the sections overlap	Value to obtain 50% overlap

Note You can use the empty matrix `[]` to specify the default value for any input argument except `x` or `y`. For example, `Pxy = cpsd(x,y,[],[],128)` uses a Hamming window, default `noverlap` to obtain 50% overlap, and the specified 128 `nfft`.

`Pxy = cpsd(x,y,window)` specifies a windowing function, divides `x` and `y` into overlapping sections of the specified window length, and windows each section using the specified window function. If you supply a scalar for `window`, `Pxy` uses a Hamming window of that length. The x and y vectors are divided into eight equal sections of that length. If the signal cannot be sectioned evenly with 50% overlap, it is truncated.

`Pxy = cpsd(x,y,window,noverlap)` overlaps the sections of `x` by `noverlap` samples. `noverlap` must be an integer smaller than the length of `window`.

`[Pxy,W] = cpsd(x,y,window,noverlap,nfft)` uses the specified FFT length `nfft` in estimating the CPSD. It also returns `W`, which is the vector of normalized frequencies (in rad/sample) at which the CPSD is estimated. For real signals, the range of `W` is [0, pi] when `nfft` is even and [0, pi) when `nfft` is odd. For complex signals, the range of `W` is [0, 2*pi).

`[Pxy,F] = cpsd(x,y,window,noverlap,nfft,fs)` returns `Pxy` as a function of frequency and a vector `F` of frequencies at which the CPSD is estimated. `fs` is the sampling frequency in Hz. For real signals, the range of `F` is [0, `fs`/2] when `nfft` is even and [0, `fs`/2) when `nfft` is odd. For complex signals, the range of `F` is [0, `fs`).

`[...] = cpsd(...,'twosided')` returns the two-sided CPSD of real signals `x` and `y`. The length of the resulting Pxy is nfft and its range is [0, 2*pi) if you do not specify fs. If you specify fs, the range is [0,fs). Entering'onesided'for a real signal produces the default. You can place the 'onesided' or 'twosided' string in any position after the `noverlap` parameter.

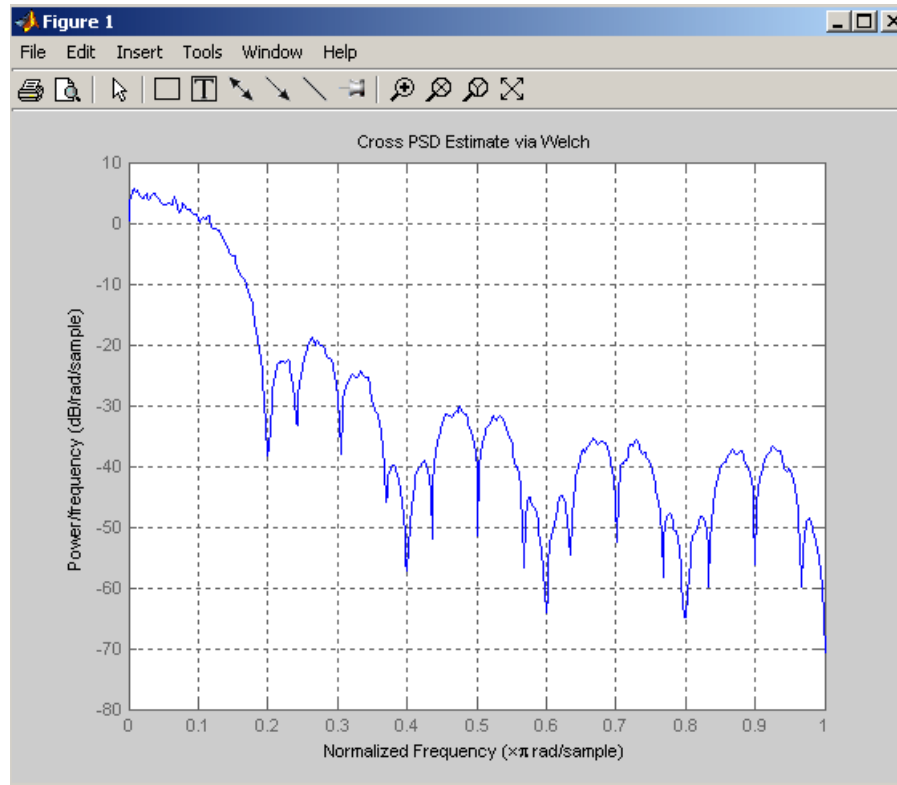
`cpsd(...)` plots the CPSD versus frequency in the current figure window.

Examples

Generate two colored noise signals and plot their CPSD. Specify a length 1024 FFT and a 500 point triangular window with no overlap.

```
randn('state',0);
h = fir1(30,0.2,rectwin(31));
```

```
h1 = ones(1,10)/sqrt(10);
r = randn(16384,1);
x = filter(h1,1,r);
y = filter(h,1,x);
cpsd(x,y,triang(500),250,1024)
```



Algorithm

`cpsd` uses Welch's averaged periodogram method. See the references listed below.

References

[1] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing,* Englewood Cliffs, NJ: Prentice-Hall, 1975. Pgs. 414-419.

[2] Welch, P.D. “The Use of Fast Fourier Transform for the Estimation of Power Spectra: A Method Based on Time Averaging Over Short, Modified Periodograms.” *IEEE Trans. Audio Electroacoust,* Vol. AU-15 (June 1967). Pgs. 70-73.

[3] Oppenheim, A.V., and R.W. Schafer. *Discrete-Time Signal Processing,* Upper Saddle River, NJ: Prentice-Hall, 1999, pp. 737.

See Also

`dspdata`, `mscohere`, `pburg`, `pcov`, `peig`, `periodogram`, `pmcov`, `pmtm`, `pmusic`, `pwelch`, `pyulear`, `spectrum`, `tfestimate`

Purpose

Chirp *z*-transform

Syntax

```
y = czt(x,m,w,a)
y = czt(x)
```

Description

`y = czt(x,m,w,a)` returns the chirp *z*-transform of signal `x`. The chirp *z*-transform is the *z*-transform of `x` along a spiral contour defined by `w` and `a`. `m` is a scalar that specifies the length of the transform, `w` is the ratio between points along the *z*-plane spiral contour of interest, and scalar `a` is the complex starting point on that contour. The contour, a spiral or "chirp" in the *z*-plane, is given by

```
z = a*(w.^-(0:m-1))
```

`y = czt(x)` uses the following default values:

- `m = length(x)`
- `w = exp(-j*2*pi/m)`
- `a = 1`

With these defaults, `czt` returns the *z*-transform of `x` at `m` equally spaced points around the unit circle. This is equivalent to the discrete Fourier transform of `x`, or `fft(x)`. The empty matrix `[]` specifies the default value for a parameter.

If `x` is a matrix, `czt(x,m,w,a)` transforms the columns of `x`.

Examples

Create a random vector `x` of length 1013 and compute its DFT using `czt`:

```
randn('state',0);
x = randn(1013,1);
y = czt(x);
```

Use `czt` to zoom in on a narrow-band section (100 to 150 Hz) of a filter's frequency response. First design the filter:

```
h = fir1(30,125/500,rectwin(31));  % filter
```

Establish frequency and CZT parameters:

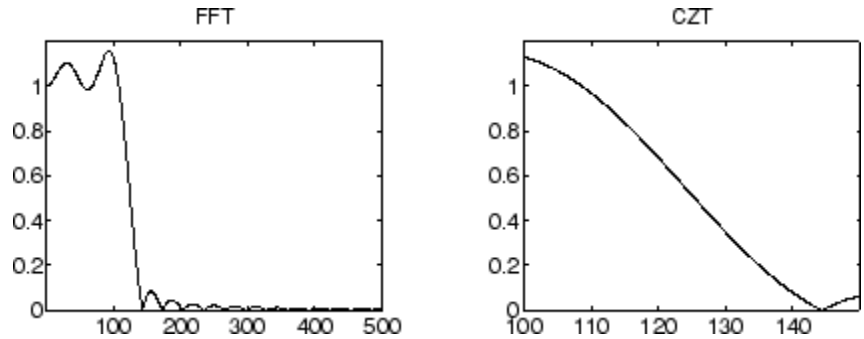
```
fs = 1000; f1 = 100; f2 = 150;     % in hertz
m = 1024;
w = exp(-j*2*pi*(f2-f1)/(m*fs));
a = exp(j*2*pi*f1/fs);
```

Compute both the DFT and CZT of the filter:

```
y = fft(h,1000);
z = czt(h,m,w,a);
```

Create frequency vectors and compare the results:

```
fy = (0:length(y)-1)'*1000/length(y);
fz = ((0:length(z)-1)'*(f2-f1)/length(z)) + f1;
plot(fy(1:500),abs(y(1:500))); axis([1 500 0 1.2])
title('FFT')
figure
plot(fz,abs(z)); axis([f1 f2 0 1.2])
title('CZT')
```



Algorithm

`czt` uses the next power-of-2 length FFT to perform a fast convolution when computing the *z*-transform on a specified chirp contour [1].

Diagnostics

If `m`, `w`, or `a` is not a scalar, `czt` gives the following error message:

```
Inputs M, W, and A must be scalars.
```

References

[1] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing,* Englewood Cliffs, NJ: Prentice-Hall, 1975. Pgs. 393-399.

See Also

`fft`, `freqz`

db2mag

Purpose

Convert decibels (dB) to magnitude

Syntax

```
y = db2mag(ydb)
```

Description

`y = db2mag(ydb)` returns the corresponding magnitude `y` for a given decibel (dB) value `ydb`. The relationship between magnitude and decibels is `ydb` = $20*\log_{10}(y)$.

See Also

`mag2db`

Purpose

Convert decibels (dB) to power

Syntax

```
y = db2pow(ydb)
```

Description

`y = db2pow(ydb)` returns the corresponding power value `y` for a given decibel (dB) value `ydb`. The relationship between power and decibels is `ydb` = $10*\log_{10}(y)$.

See Also

`pow2db`

Purpose

Discrete cosine transform (DCT)

Syntax

```
y = dct(x)
y = dct(x,n)
```

Description

`y = dct(x)` returns the unitary discrete cosine transform of `x`

$$y(k) = w(k) \sum_{n=1}^{N} x(n) \cos \frac{\pi(2n-1)(k-1)}{2N}, \qquad k = 1, \ldots, N$$

where

$$w(k) = \begin{cases} \frac{1}{\sqrt{N}}, & k = 1 \\ \sqrt{\frac{2}{N}}, & 2 \le k \le N \end{cases}$$

N is the length of `x`, and `x` and `y` are the same size. If `x` is a matrix, `dct` transforms its columns. The series is indexed from $n = 1$ and $k = 1$ instead of the usual $n = 0$ and $k = 0$ because MATLAB vectors run from 1 to N instead of from 0 to N- 1.

`y = dct(x,n)` pads or truncates `x` to length `n` before transforming.

The DCT is closely related to the discrete Fourier transform. You can often reconstruct a sequence very accurately from only a few DCT coefficients, a useful property for applications requiring data reduction.

Examples

Find how many DCT coefficients represent 99% of the energy in a sequence:

```
x = (1:100) + 50*cos((1:100)*2*pi/40);
X = dct(x);
[XX,ind] = sort(abs(X)); ind = fliplr(ind);
i = 1;
while (norm([X(ind(1:i)) zeros(1,100-i)])/norm(X)<.99)
   i = i + 1;
```

```
end
i =
      3
```

References

[1] Jain, A.K. *Fundamentals of Digital Image Processing,* Englewood Cliffs, NJ: Prentice-Hall, 1989.

[2] Pennebaker, W.B., and J.L. Mitchell. *JPEG Still Image Data Compression Standard,* New York, NY: Van Nostrand Reinhold, 1993. Chapter 4.

See Also

`fft`, `idct`, `dct2`, `idct2`

decimate

Purpose

Decimation — decrease sampling rate

Syntax

```
y = decimate(x,r)
y = decimate(x,r,n)
y = decimate(x,r,'fir')
y = decimate(x,r,n,'fir')
```

Description

Decimation reduces the original sampling rate for a sequence to a lower rate, the opposite of interpolation. The decimation process filters the input data with a lowpass filter and then resamples the resulting smoothed signal at a lower rate.

`y = decimate(x,r)` reduces the sample rate of `x` by a factor `r`. The decimated vector `y` is `r` times shorter in length than the input vector `x`. By default, `decimate` employs an eighth-order lowpass Chebyshev Type I filter with a cutoff frequency of 0.8*(Fs/2)/`r`. It filters the input sequence in both the forward and reverse directions to remove all phase distortion, effectively doubling the filter order.

`y = decimate(x,r,n)` uses an order `n` Chebyshev filter. Orders above 13 are not recommended because of numerical instability. In this case, a warning is displayed.

Note For better results when `r` is greater than 13, you should break `r` into its factors and call `decimate` several times.

`y = decimate(x,r,'fir')` uses an order 30 FIR filter, instead of the Chebyshev IIR filter. Here `decimate` filters the input sequence in only one direction. This technique conserves memory and is useful for working with long sequences.

`y = decimate(x,r,n,'fir')` uses an order `n` FIR filter.

Examples

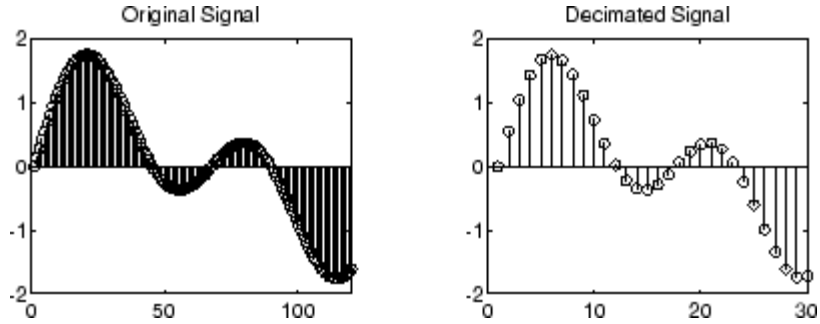
Decimate a signal by a factor of four:

```
t = 0:.00025:1;                       % Time vector
```

```
x = sin(2*pi*30*t) + sin(2*pi*60*t);
y = decimate(x,4);
```

View the original and decimated signals:

```
stem(x(1:120)), axis([0 120 -2 2])   % Original signal
title('Original Signal')
figure
stem(y(1:30))                        % Decimated signal
title('Decimated Signal')
```



Algorithm

`decimate` uses decimation algorithms 8.2 and 8.3 from [1]:

1. It designs a lowpass filter. By default, `decimate` uses a Chebyshev Type I filter with normalized cutoff frequency `0.8/r` and 0.05 dB of passband ripple. For the `fir` option, `decimate` designs a lowpass FIR filter with cutoff frequency `1/r` using `fir1`.

2. For the FIR filter, `decimate` applies the filter to the input vector in one direction. In the IIR case, `decimate` applies the filter in forward and reverse directions with `filtfilt`.

3. `decimate` resamples the filtered data by selecting every `r`th point.

Note Depending on the CPU and operating system of your computer, the decimate function may use a lower filter order. If the specified filter order will produce passband distortion, caused by roundoff errors accumulated from the convolutions needed to create the transfer function, the filter order is automatically reduced.

Diagnostics

If `r` is not an integer, `decimate` gives the following error message:

```
Resampling rate R must be an integer.
```

If `n` specifies an IIR filter with order greater than 13, `decimate` gives the following warning:

```
Warning: IIR filters above order 13 may be unreliable.
```

References

[1] *IEEE Programs for Digital Signal Processing,* IEEE Press. New York: John Wiley & Sons, 1979. Chapter 8.

See Also

`cheby1`, `downsample`, `filtfilt`, `fir1`, `mfilt`, `interp`, `resample`

Purpose

Demodulation for communications simulation

Syntax

```
x = demod(y,fc,fs,'method')
x = demod(y,fc,fs,'method',opt)
x = demod(y,fc,fs,'pwm','centered')
```

Description

`demod` performs demodulation, that is, it obtains the original signal from a modulated version of the signal. `demod` undoes the operation performed by `modulate`.

`x = demod(y,fc,fs,'method')` and

`x = demod(y,fc,fs,'method',opt)` demodulate the real carrier signal `y` with a carrier frequency `fc` and sampling frequency `fs`, using one of the options listed below for *`method`*. (Note that some methods accept an option, `opt`.)

Method	Description

`amdsb-sc`
or
`am`
	Amplitude demodulation, double sideband, suppressed carrier. Multiplies `y` by a sinusoid of frequency `fc` and applies a fifth-order Butterworth lowpass filter using `filtfilt`.
	`x = y.*cos(2*pi*fc*t);`
	`[b,a] = butter(5,fc*2/fs);`
	`x = filtfilt(b,a,x);`

Method	Description

`amdsb-tc`	Amplitude demodulation, double sideband, transmitted carrier. Multiplies `y` by a sinusoid of frequency `fc`, and applies a fifth-order Butterworth lowpass filter using `filtfilt`.
	`x = y.*cos(2*pi*fc*t);`
	`[b,a] = butter(5,fc*2/fs);`
	`x = filtfilt(b,a,x);`
	If you specify `opt`, `demod` subtracts scalar `opt` from `x`. The default value for `opt` is 0.

`amssb`	Amplitude demodulation, single sideband. Multiplies `y` by a sinusoid of frequency `fc` and applies a fifth-order Butterworth lowpass filter using `filtfilt`.
	`x = y.*cos(2*pi*fc*t);`
	`[b,a] = butter(5,fc*2/fs);`
	`x = filtfilt(b,a,x);`

`fm`	Frequency demodulation. Demodulates the FM waveform by modulating the Hilbert transform of `y` by a complex exponential of frequency `-fc` Hz and obtains the instantaneous frequency of the result.
`pm`	Phase demodulation. Demodulates the PM waveform by modulating the Hilbert transform of `y` by a complex exponential of frequency `-fc` Hz and obtains the instantaneous phase of the result.
`ppm`	Pulse-position demodulation. Finds the pulse positions of a pulse-position modulated signal `y`. For correct demodulation, the pulses cannot overlap. `x` is length `length(t)*fc/fs`.

Method	Description
pwm	Pulse-width demodulation. Finds the pulse widths of a pulse-width modulated signal y. demod returns in x a vector whose elements specify the width of each pulse in fractions of a period. The pulses in y should start at the beginning of each carrier period, that is, they should be left justified.

qam	Quadrature amplitude demodulation.
	[x1,x2] = demod(y,fc,fs,'qam') multiplies y by a cosine and a sine of frequency fc and applies a fifth-order Butterworth lowpass filter using filtfilt.
	x1 = y.*cos(2*pi*fc*t);
	x2 = y.*sin(2*pi*fc*t);
	[b,a] = butter(5,fc*2/fs);
	x1 = filtfilt(b,a,x1);
	x2 = filtfilt(b,a,x2);

The default method is 'am'. In all cases except 'ppm' and 'pwm', x is the same size as y.

If y is a matrix, demod demodulates its columns.

x = demod(y,fc,fs,'pwm','centered') finds the pulse widths assuming they are centered at the beginning of each period. x is length length(y)*fc/fs.

See Also

modulate, vco, fskdemod, genqamdemod, mskdemod, pamdemod, pmdemod, qamdemod

design

Purpose

Apply design method to specification object

Syntax

```
h = design(d)
h = design(d,designmethod)
h = design(d,designmethod,specname,specvalue,...)
```

Description

`h = design(d)` uses a filter specification object `d` to generate a filter `h`. When you do not provide a design method as an input argument, `design` chooses the design method to use by following these rules in the order listed.

1. Use`equiripple` if it applies to the object `d`.
2. When `equiripple` does not apply to `d`, use another FIR design method, such as `firls`.
3. If FIR design methods do not apply to `d`, use `ellip`.
4. When `ellip`does not apply to `d`, use another IIR design method, such as `butter` or `cheby1`, that applies to the object `d`.

For more guidance about using `design` to design filters, refer to Chapter 3, “Designing a Filter in Fdesign — Process Overview”. There you find examples that use `design` to design filters and use methods in the toolbox to analyze them. Alternatively, you can type the following at the MATLAB command prompt to obtain more information:

```
help design
```

`h = design(d,designmethod)` lets you specify a valid design method to design the filter as an input string. Note that the filter returned by `design` changes depending on the design method you choose. For more information about the filter that a design method returns, refer to the help for the design method.

The design method you provide as the `designmethod` input argument must be one of the methods returned by

```
designmethods(d)
```

for the specifications object `d`.

Valid entries depend on `d`. This is the complete set of design methods. The methods that apply to a specific specifications object usually represent a subset of this list.

- `butter`
- `cheby1`
- `cheby2`
- `ellip`
- `equiripple`
- `firls`
- `kaiserwin`
- `window`

To help you design filters more quickly, the input argument `designmethod` accepts a variety of special keywords that force `design` to behave in different ways. The following table presents the keywords you can use for `designmethod` and how `design` responds to the keyword.

Designmethod Keyword	Description of the design Response
`fir`	Forces `design` to produce an FIR filter. When no FIR design method exists for object `d`, `design` returns an error.
`iir`	Forces `design` to produce an IIR filter. When no IIR design method exists for object `d`, `design` returns an error.

Designmethod Keyword	Description of the design Response
`allfir`	Produces filters from every applicable FIR design method for the specifications in `d`, one filter for each design method. As a result, `design` returns multiple filters in the output object.
`alliir`	Produces filters from every applicable IIR design method for the specifications in `d`, one filter for each design method. As a result, `design` returns multiple filters in the output object.
`all`	Designs filters using all applicable design methods for the specifications object `d`. As a result, `design` returns multiple filters, one for each design method. `design` uses the design methods in the order that `designmethods(d)` returns them. Refer to Examples to see this in use.

Keywords are not case sensitive and must be enclosed in single quotation marks like any string input.

When design returns multiple filters in the output object, use indexing to see the individual filters. For example, to see the third filter in `h`, enter

```
h(3)
```

at the MATLAB prompt.

`h = design(d,designmethod,specname,specvalue,...)` with this syntax you can specify not only the design method but also values for the filter specifications in the method. Provide the specifications in the order of the name of the specification, such as the `FilterOrder`, followed by the value to assign to the specification. Enter as many `specname`/`specvalue` pairs as you need to define your filter. Any specification you do not define uses the default specification value. To use the `specname`/`specvalue` syntax, you must provide the design method to use in `designmethod`.

Examples

To demonstrate some of the design options, these examples use a few different input arguments and output arguments. For the first example, use `design` to return the default filter based on the default equiripple design method .

```
d=fdesign.lowpass('Fp,Fst,Ap,Ast',0.2,0.22,1,60);
Hd = design(d) % Uses the default equiripple method.

Hd =

     FilterStructure: 'Direct-Form FIR'
           Numerator: [1x202 double]
    PersistentMemory: false
```

In this example, use the **`allfir`** keyword with `design` to return an FIR filter for each valid design method for the specifications in specifications object `d`.

```
designmethods(d)


Design Methods for class fdesign.lowpass (Fp,Fst,Ap,Ast):


butter
cheby1
cheby2
ellip
equiripple
kaiserwin

hallfir=design(d,'allfir')

hallfir =

dfilt.dffir
dfilt.dffir
```

`hallfir` contains filters designed using the `equiripple` and `kaiserwin` design methods, in the order shown by `designmethods(d)`..

To see an individual filter, use an index with the filter object. For example, to see the second filter in `hallfir`, enter `hallfir(2)`

```
hallfir(2)

ans =

     FilterStructure: 'Direct-Form FIR'
           Numerator: [1x365 double]
    PersistentMemory: false
```

This final example uses `equiripple` to design an FIR filter with the density factor set to 20 by using the `specname/specvalue` syntax.

```
hd= design(d,'equiripple','densityfactor',20)

hd =

     FilterStructure: 'Direct-Form FIR'
           Numerator: [1x202 double]
    PersistentMemory: false
```

See Also

`designmethods`, `designopts`, `fdesign`,

Purpose

Methods available for designing filter from specification object

Syntax

```
m = designmethods(d)
m = designmethods(d,'default')
m = designmethods(d,type)
m = designmethods(d,'full')
```

Description

`m = designmethods(d)` returns a list of the design methods available for the filter specification object `d` with its `Specification`. When you change the `Specification` for a filter specification object, the methods available to design filters from the object change.

Here are all the design methods and the filters they produce.

Design Method	Filter Result
`butter`	IIR
`cheby1`	IIR
`cheby2`	IIR
`ellip`	IIR
`equiripple`	FIR
`firls`	FIR
`freqsamp`	FIR
`kaiserwin`	FIR
`window (filter design method)`	FIR

`m = designmethods(d,'default')` returns the default design method for the filter specification object `d` and its current `Specification`.

`m = designmethods(d,type)` returns either the FIR or IIR design methods that apply to d, as specified by the `type` string, either `fir` or `iir`. By default, `designmethods` returns all the valid design methods when you omit the `type` string.

`m = designmethods(d,'full')` returns the full name for each of the available design methods. For example, `designmethods` with the **`full`** argument returns `Butterworth` for the `butter` method.

Examples

Construct a lowpass filter specification object and determine the design methods available to design a filter from the object.

```
d=fdesign.lowpass('n,fc',10,12000,48000)

d =

            Response: 'Lowpass'
       Specification: 'N,Fc'
         Description: {'Filter Order';'Cutoff Frequency'}
 NormalizedFrequency: false
                  Fs: 48000
         FilterOrder: 10
             Fcutoff: 12000

designmethods(d)


Design Methods for class fdesign.lowpass (N,Fc):


window

hd=window(d)

hd =

     FilterStructure: 'Direct-Form FIR'
          Arithmetic: 'double'
           Numerator: [1x11 double]
    PersistentMemory: false
```

Now change the `Specification` string for `d` to `'fp,fst,ap,ast'` and determine the design methods that apply to your modified specifications object.

```
set(d,'specification','fp,fst,ap,ast');
d

d =

               Response: 'Lowpass'
          Specification: 'Fp,Fst,Ap,Ast'
            Description: {4x1 cell}
    NormalizedFrequency: false
                     Fs: 48000
                  Fpass: 10800
                  Fstop: 13200
                  Apass: 1
                  Astop: 60

m2 = designmethods(d);
m3 = designmethods(d, 'iir');
m4 = designmethods(d, 'iir', 'full');

m2

m2 =

    'butter'
    'cheby1'
    'cheby2'
    'ellip'
    'equiripple'
    'kaiserwin'

m3

m3 =
```

```
    'butter'
    'cheby1'
    'cheby2'
    'ellip'

m4

m4 =

    'Butterworth'
    'Chebyshev type I'
    'Chebyshev type II'
    'Elliptic'
```

Now you can get specific help on a particular design method for the specifications object. This example returns the help for the first design method for the `m2` set of methods — `butter`.

```
help(d,m2{1})
```

This is the same as `help(d,'butter')`.

See Also

`design`, `designopts`, `fdesign`.

Purpose

Valid input arguments and values for specification object and method

Syntax

```
options = designopts(d,'designmethod')
```

Description

`options = designopts(d,'designmethod')` returns the structure `options` with the default design parameters used by the design method `designmethod`, specific to the response you defined for `d`. Replace `designmethod` with one of the strings returned by `designmethods`.

Use `help(d,`*`designmethod`*`)` to get a description of the design parameters. For example, to see the help for designing a highpass Chebyshev II filter from a specifications object `d`, enter

```
help(d,'cheby2')
```

at the prompt. MATLAB responds with help for Chebyshev II filter designs that use the specification `Fst`,`Fp`,`Ast`,`Ap`.

```
DESIGN Design a Chebyshev Type II iir filter.
HD = DESIGN(D, 'cheby2') designs a Chebyshev Type II
filter specified by the FDESIGN object H.

HD = DESIGN(..., 'FilterStructure', STRUCTURE) returns a
filter with the structure STRUCTURE.  STRUCTURE is
'df2sos' by default and can be any of the following.

'df1sos'
'df2sos'
'df1tsos'
'df2tsos'

HD = DESIGN(..., 'MatchExactly', MATCH) designs a
Chebyshev Type II filter and matches the frequency and
magnitude specification for the band MATCH exactly.
The other band will exceed the specification.
MATCH can be 'stopband' or 'passband' and is 'passband'
by default.
```

Examples

Design a minimum order, lowpass Butterworth filter. Use `designmethods` to determine the appropriate input arguments. Start by creating a lowpass filter specification object `d`.

```
d = fdesign.lowpass;
```

Because you want information about the input arguments for designing a filter using a design method, use `designmethods(d)` to get the list of valid methods.

```
designmethods(d)

Design Methods for class fdesign.lowpass (Fp,Fst,Ap,Ast):


butter
cheby1
cheby2
ellip
equiripple
kaiserwin
```

Pick one method and determine the design options for that method.

```
options = designopts(d,'butter')

options =

    FilterStructure: 'df2sos'
       MatchExactly: 'stopband'
```

In this example, the filter structure is Direct-Form II with second-order sections, and the design seeks to match the desired stopband performance exactly. As you see by reading the help, `FilterStructure` and `MatchExactly` are input arguments for designing the Butterworth filter.

Get help for designing a filter from `d` using the `butter` design method to see the arguments.

```
help(d,'butter')

 DESIGN Design a Butterworth IIR filter.
 HD = DESIGN(D, 'butter') designs a Butterworth filter specified by the
 FDESIGN object H.

 HD = DESIGN(..., 'FilterStructure', STRUCTURE) returns a filter with the
structure STRUCTURE.  STRUCTURE is 'df2sos' by default and can be any of
the following.

    'df1sos'
    'df2sos'
    'df1tsos'
    'df2tsos'

 HD = DESIGN(..., 'MatchExactly', MATCH) designs a Butterworth filter
 and matches the frequency and magnitude specification for the band
 MATCH exactly.  The other band will exceed the specification.  MATCH
 can be 'stopband' or 'passband' and is 'stopband' by default.
```

See Also

`design`, `designmethods`, `fdesign`, `validstructures`.

dfilt

Purpose

Discrete-time filter

Syntax

```
Hd = dfilt.structure(input1,...)
Hd = [dfilt.structure(input1,...),dfilt.structure(input1,...),
   ...]
```

Description

`Hd = dfilt.structure(input1,...)` returns a discrete-time filter, `Hd`, of type *`structure`*. Each structure takes one or more inputs. If you specify a `dfilt.structure` with no inputs, a default filter is created.

Note You must use a *`structure`* with `dfilt`.

`Hd = [dfilt.structure(input1,...),dfilt.structure(input1,...),...]` returns a vector containing `dfilt` filters.

Structures

Structures for `dfilt` specify the type of filter structure. Available types of structures for `dfilt` are shown below.

dfilt.structure	Description
`dfilt.delay`	Delay
`dfilt.df1`	Direct-form I
`dfilt.df1sos`	Direct-form I, second-order sections
`dfilt.df1t`	Direct-form I transposed
`dfilt.df1tsos`	Direct-form I transposed, second-order sections
`dfilt.df2`	Direct-form II
`dfilt.df2sos`	Direct-form II, second-order sections
`dfilt.df2t`	Direct-form II transposed
`dfilt.df2tsos`	Direct-form II transposed, second-order sections
`dfilt.dffir`	Direct-form FIR

dfilt.structure	Description
`dfilt.dffirt`	Direct-form FIR transposed
`dfilt.dfsymfir`	Direct-form symmetric FIR
`dfilt.dfasymfir`	Direct-form antisymmetric FIR
`dfilt.fftfir`	Overlap-add FIR
`dfilt.latticeallpass`	Lattice allpass
`dfilt.latticear`	Lattice autoregressive (AR)
`dfilt.latticearma`	Lattice autoregressive moving- average (ARMA)
`dfilt.latticemamax`	Lattice moving-average (MA) for maximum phase
`dfilt.latticemamin`	Lattice moving-average (MA) for minimum phase
`dfilt.calattice`	Coupled, allpass lattice (available only with Filter Design Toolbox product)
`dfilt.calatticepc`	Coupled, allpass lattice with power complementary output (available only with Filter Design Toolbox product)
`dfilt.statespace`	State-space
`dfilt.scalar`	Scalar gain object
`dfilt.cascade`	Filters arranged in series
`dfilt.parallel`	Filters arranged in parallel

For more information on each structure, use the syntax `help diflt.`*`structure`* at the MATLAB prompt or refer to its reference page.

Methods

Methods provide ways of performing functions directly on your `dfilt` object without having to specify the filter parameters again. You can apply these methods directly on the variable you assigned to your `dfilt` object.

For example, if you create a `dfilt` object, `Hd`, you can check whether it has linear phase with `islinphase(Hd)`, view its frequency response

plot with `fvtool(Hd)`, or obtain its frequency response values with `h=freqz(Hd)`. You can use all of the methods below in this way.

Note If your variable is a 1-D array of `dfilt` filters, the method is applied to each object in the array. Only `freqz`, `grpdelay`, `impz`, `is*`, `order`, and `stepz` methods can be applied to arrays. The `zplane` method can be applied to an array only if it is used without outputs.

Some of the methods listed below have the same name as Signal Processing Toolbox functions and they behave similarly. This is called *overloading* of functions.

Available methods are:

Method	Description
`addstage`	Adds a stage to a `cascade` or `parallel` object, where a stage is a separate, modular filter. See `dfilt.cascade` and `dfilt.parallel`.
`block`	(Available only with Signal Processing Blockset product) `block(Hd)` creates a Signal Processing Blockset block of the `dfilt` object. The `block` method can specify these properties/values: `'Destination'` indicates where to place the block. `'Current'` places the block in the current Simulink model. `'New'` creates a new model. If you enter the name of an existing subsytem in your model, the block is added to that subsystem. Default value is `'current'`. `'Blockname'` assigns the entered string to the block name. Default name is `'filter'`.

Method	Description
	'OverwriteBlock'indicates whether to overwrite the block generated by the block method ('on') and defined by Blockname. Default is 'off'. 'MapStates' specifies initial conditions in the block ('on'). Default is 'off'. See "Using Filter States" on page 10-151. 'Link2Obj' indicates whether to specify the filter by linking the block and the command line dfilt object ('on') or by inserting the coefficients directly into the block with no further connection between the command line dfilt object and the block ('off'). If you set 'link2obj' to 'on', the dfilt variable name is inserted in the block and changes to the dfilt object via the command line are reflected in the linked block. Default value is 'off'. 'MapCoeffstoPorts' indicates whether to map the filter coefficients to constant blocks connected to the generated block. Default value is 'off'. Setting 'MapCoeffstoPorts' to 'on' turns on the mapping and enables the 'CoeffNames' property, which defines the constant block parameter names. 'CoeffNames' is a cell array of strings. Default values are {'Num'} for Direct form FIR filters, {'K'} for lattice filters, {'Num','Den'} for IIR filters, and {Num','Den','g'} for biquad filters. Variables, defined by 'CoeffNames', are created in the MATLAB workspace and have the same data type as the filter's 'Arithmetic' property. Any existing variable with the same name is overwritten. Note that you can use either

Method	Description
	`'Link2Obj'` or `'MapCoeffstoPorts'`, but not both simultaneously.
`cascade`	Returns the series combination of two `dfilt` objects. See `dfilt.cascade`.
`coeffs`	Returns the filter coefficients in a structure containing fields that use the same property names as those in the original `dfilt`.
`convert`	Converts a `dfilt` object from one filter structure to another filter structure.
`fcfwrite`	Writes a filter coefficient ASCII file. The file can contain a single filter or a vector of objects. If Filter Design Toolboxx product is installed, the file can contain multirate filters (`mfilt`) or adaptive filters (`adaptfilt`). Default filename is `untitled.fcf`. `fcfwrite(Hd,filename)` writes to a disk file named `filename` in the current working directory. The `.fcf` extension is added automatically. `fcfwrite(...,fmt)` writes the coefficients in the format `fmt`, where valid `fmt` strings are: `'hex'` for hexadecimal `'dec'` for decimal `'bin'` for binary representation.
`fftcoeffs`	Returns the frequency-domain coefficients used when filtering with a `dfilt.fftfir`.

Method	Description
`filter`	Performs filtering using the `dfilt` object. `y = filter(Hd,x)` filters `x` using the `Hd` filter and returns the filtered data in `y`. See "Using Filter States" on page 10-151 for information on using initial conditions. If `x` is a matrix, each column is filtered as an independent channel. If `x` is a multidimensional array, `filter` operates on the first nonsingleton dimension. `y = filter(Hd,x,dim)` operates along the dimension `dim`. If `x` is a vector or matrix and `dim` is 1, every column of `x` is a channel. If `dim` is 2, every row is a channel.
`firtype`	Returns the type (1-4) of a linear phase FIR filter.
`freqz`	Plots the frequency response in `fvtool`. Note that unlike the `freqz` function, this `dfilt freqz` method has a default length of 8192.
`grpdelay`	Plots the group delay in `fvtool`.
`impz`	Plots the impulse response in `fvtool`.
`impzlength`	Returns the length of the impulse response.
`info`	Displays brief `dfilt` information, such as filter structure, length, stability, linear phase, and, when appropriate, lattice and ladder length. To display detailed information about the design method, options, etc, use `info(Hd, 'long')`. The default display is `'short'`. For multistage filters (`cascade` and `parallel`), use `info(Hd.Stage(x))`, where `x` is the stage number, to see information about that stage.

Method	Description
`isallpass`	Returns a logical `1` (i.e., true) if the `dfilt` object in an allpass filter or a logical `0` (i.e., false) if it is not.
`iscascade`	Returns a logical `1` if the `dfilt` object is cascaded or a logical `0` if it is not.
`isfir`	Returns a logical `1` if the `dfilt` object has finite impulse response (FIR) or a logical `0` if it does not.
`islinphase`	Returns a logical `1` if the `dfilt` object is linear phase or a logical `0` if it is not.
`ismaxphase`	Returns a logical `1` if the `dfilt` object is maximum-phase or a logical `0` if it is not.
`isminphase`	Returns a logical `1` if the `dfilt` object is minimum-phase or a logical `0` if it is not.
`isparallel`	Returns a logical `1` if the `dfilt` object has parallel stages or a logical `0` if it does not.
`isreal`	Returns a logical `1` if the `dfilt` object has real-valued coefficients or a logical 0 if it does not.
`isscalar`	Returns a logical `1` if the `dfilt` object is a scalar or a logical `0` if it is not scalar.
`issos`	Returns a logical `1` if the `dfilt` object has second-order sections or a logical `0` if it does not.
`isstable`	Returns a logical `1` if the `dfilt` object is stable or a logical `0` if it are not.
`nsections`	Returns the number of sections in a second-order sections filter. If a multistage filter contains stages with multiple sections, using `nsections` returns the total number of sections in all the stages (a stage with a single section returns 1).

Method	Description
`nstages`	Returns the number of stages of the filter, where a stage is a separate, modular filter.
`nstates`	Returns the number of states for an object.
`order`	Returns the filter order. If `Hd` is a single-stage filter, the order is given by the number of delays needed for a minimum realization of the filter. If `Hd` has multiple stages, the order is given by the number of delays needed for a minimum realization of the overall filter.
`parallel`	Returns the parallel combination of two `dfilt` filters. See `dfilt.parallel`.
`phasez`	Plots the phase response in `fvtool`.
`realizemdl`	(Available only with Simulink software.) `realizemdl(Hd)` creates a Simulink model containing a subsystem block realization of your `dfilt`. `realizemdl(Hd,p1,v1,p2,v2,...)` creates the block using the properties `p1`, `p2`,... and values `v1`, `v2`,.. specified. The following properties are available: `'Blockname'` specifies the name of the block. The default value is `'Filter'`. `'Destination'` specifies whether to add the block to a current Simulink model, create a new model, or place the block in an existing subsystem in your model. Valid values are `'current'`, `'new'`, or the name of an existing subsytem in your model. Default value is `'current'`.

Method	Description
	'OverwriteBlock' specifies whether to overwrite an existing block that was created by realizemdl or create a new block. Valid values are 'on' and 'off' and the default is 'off'. Note that only blocks created by realizemdl are overwritten. The following properties optimize the block structure. Specifying 'on' turns the optimization on and 'off' creates the block without optimization. The default for each of the following is 'on'. 'OptimizeZeros' removes zero-gain blocks. 'OptimizeOnes' replaces unity-gain blocks with a direct connection. 'OptimizeNegOnes' replaces negative unity-gain blocks with a sign change at the nearest summation block. 'OptimizeDelayChains' replaces cascaded chains of delay block with a single integer delay block set to the appropriate delay.
removestage	Removes a stage from a cascade or parallel dfilt. See dfilt.cascade and dfilt.parallel.
setstage	Overwrites a stage of a cascade or parallel dfilt. See dfilt.cascade and dfilt.parallel.

Method	Description
sos	Converts the dfilt to a second-order sections dfilt. If Hd has a single section, the returned filter has the same class. sos(Hd,flag) specifies the ordering of the second-order sections. If flag='UP', the first row contains the poles closest to the origin, and the last row contains the poles closest to the unit circle. If flag='down', the sections are ordered in the opposite direction. The zeros are always paired with the poles closest to them. sos(Hd,flag,scale) specifies the scaling of the gain and the numerator coefficients of all second-order sections. scale can be 'none', 'inf' (infinity-norm) or 'two' (2-norm). Using infinity-norm scaling with up ordering minimizes the probability of overflow in the realization. Using 2-norm scaling with down ordering minimizes the peak roundoff noise.
ss	Converts the dfilt to state-space. To see the separate A,B,C,D matrices for the state-space model, use [A,B,C,D]=ss(Hd).
stepz	Plots the step response in fvtool. stepz(Hd,n) computes the first n samples of the step response. stepz(Hd,n,Fs) separates the time samples by T = 1/Fs, where Fs is assumed to be in Hz.
tf	Converts the dfilt to a transfer function.
zerophase	Plots the zero-phase response in fvtool.
zpk	Converts the dfilt to zeros-pole-gain form.
zplane	Plots a pole-zero plot in fvtool.

For more information on each method, use the syntax `help dflt/method` at the MATLAB prompt.

Viewing Properties

As with any object, you can use `get` to view a `dfilt` properties. To see a specific property, use

```
get(Hd,'property')
```

To see all properties for an object, use

```
get(Hd)
```

Note If you have Filter Design Toolbox product installed, an `arithmetic` property is displayed. You can change the internal arithmetic of the filter from double- precision to single-precision using: `Hd.arithmetic = 'single'` If you have both Filter Design Toolbox and Fixed-Point Toolbox products installed, you can change the `arithmetic` property to fixed-point using: `Hd.arithmetic = 'fixed'`

Changing Properties

To set specific properties, use

```
set(Hd,'property1',value,'property2',value,...)
```

Note that you must use single quotation marks around the property name.

Copying an Object

To create a copy of an object, use the `copy` method.

```
H2 = copy(Hd)
```

Note Using the syntax `H2 = Hd` copies only the object handle and does not create a new object.

Converting Between Filter Structures

To change the filter structure of a `dfilt` object `Hd`, use

```
Hd2=convert(Hd,'structure_string');
```

where `structure_string` is any valid structure name in single quotation marks. If `Hd` is a `cascade` or `parallel` structure, each of its stages is converted to the new structure.

Using Filter States

Two properties control the filter states:

- `states` — stores the current states of the filter. Before the filter is applied, the states correspond to the initial conditions and after the filter is applied, the states correspond to the final conditions. For `df1`, `df1t`, `df1sos` and `df1tsos` structures, `states` returns a `filtstate` object.
- `PersistentMemory` — controls whether filter `states` are saved. The default value is `'false'`, which causes the initial conditions to be reset to zero before filtering and turns off the display of `states` information. Setting `PersistentMemory` to `'true'` allows the filter to use your initial conditions or to reuse the final conditions of a previous filtering operation as the initial conditions of the next filtering operation. It also displays information about the filter `states`.

Note If you set `states` and want to use them for filtering, you must set `PersistentMemory` to `'true'` before you use the filter.

Examples

Create a direct-form I filter and use a method to see if it is stable.

```
[b,a] = butter(8,0.25);
Hd = dfilt.df1(b,a)

Hd =
         FilterStructure: 'Direct-Form I'
               Numerator: [1x9 double]
             Denominator: [1x9 double]
        PersistentMemory: false

isstable(Hd)
ans =
     1
```

If a `dfilt`'s numerator values do not fit on a single line, a description of the vector is displayed. To see the specific numerator values for this example, use

```
get(Hd,'numerator')

ans =
Columns 1 through 6
    0.0001  0.0009  0.0030  0.0060  0.0076  0.0060
  Columns 7 through 9
    0.0030  0.0009  0.0001
```

Create an array containing two `dfilt` objects, apply a method and verify that the method acts on both objects, and use a method to test whether the objects are FIR objects.

```
b = fir1(5,.5);
Hd = dfilt.dffir(b);        % Create an FIR object
[b,a] = butter(5,.5);
Hd(2) = dfilt.df2t(b,a);    % Create a DF2T object and
               % place it in the second
                            % column of Hd
```

```
[h,w] = freqz(Hd);
size(h)                         % Verify that resulting h is
ans =                           % 2 columns
        8192           2
size(w)                         % Verify that resulting w is
ans =                           % 1 column
        8192           1

test_fir = isfir(Hd)
test_fir =
     1     0                    % Hd(1) is FIR and Hd(2) is not
```

Refer to the reference pages for each structure for more examples.

See Also

dfilt.cascade , dfilt.df1, dfilt.df1t, dfilt.df2, dfilt.df2t, dfilt.dfasymfir, dfilt.dffir, dfilt.dffirt, dfilt.dfsymfir, dfilt.latticeallpass, dfilt.latticear, dfilt.latticearma, dfilt.latticemamax, dfilt.latticemamin, dfilt.parallel, dfilt.statespace, filter, freqz, grpdelay, impz, step, tf, zpk, zplane

Purpose

Cascade of discrete-time filters

Syntax

```
Hd = dfilt.cascade(Hd1,Hd2,...)
```

Description

`Hd = dfilt.cascade(Hd1,Hd2,...)` returns a discrete-time filter, `Hd`, of type `cascade`, which is a serial interconnection of two or more `dfilt` filters, `Hd1`, `Hd2`, etc. Each filter in a cascade is a separate stage.

To add a filter (`Hd3`) to the end of an existing cascade (`Hd`), use

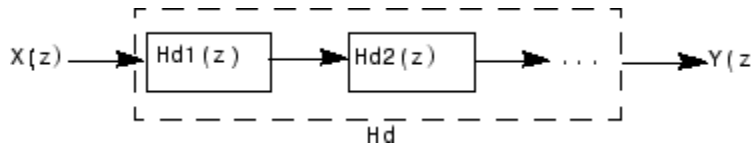
```
Hd = addstage(Hd3)
```

and to reorder the filters in a cascade, use the stage indices to indicate the desired ordering, such as.

```
Hd.stage = Hd.stage([1,3,2]);
```

You can also use the nondot notation format for calling a cascade:

```
cascade(Hd1,Hd2,...)
```



Examples

Cascade a lowpass filter and a highpass filter to produce a bandpass filter:

```
[b1,a1]=butter(8,0.6);          % Lowpass
[b2,a2]=butter(8,0.4,'high');   % Highpass
H1=dfilt.df2t(b1,a1);
H2=dfilt.df2t(b2,a2);
Hcas=dfilt.cascade(H1,H2)       % Bandpass-passband .4-.6
Hcas =
         FilterStructure: Cascade
                Stage(1): Direct-Form II Transposed
                Stage(2): Direct-Form II Transposed
```

```
           PersistentMemory: false
```

To view details of the first stage, use

```
info(Hcas.Stage(1))

Discrete-Time IIR Filter (real)
-------------------------------
Filter Structure    : Direct-Form II Transposed
Numerator Length    : 9
Denominator Length  : 9
Stable              : Yes
Linear Phase        : No
```

To view the states of a stage, use

```
Hcas.stage(1).states
```

You can display states for individual stages only.

See Also

`dfilt`, `dfilt.parallel`, `dfilt.scalar`

dfilt.delay

Purpose

Delay filter

Syntax

```
Hd = dfilt.delay
Hd = dfilt.delay(latency)
```

Description

`Hd = dfilt.delay` returns a discrete-time filter, `Hd`, of type `delay`, which adds a single delay to any signal filtered with `Hd`. The filtered signal has its values shifted by one sample.

`Hd = dfilt.delay(latency)` returns a discrete-time filter, `Hd`, of type `delay`, which adds the number of delay units specified in `latency` to any signal filtered with `Hd`. The filtered signal has its values shifted by the `latency` number of samples. The values that appear before the shifted signal are the filter states.

Examples

Create a `delay` filter with a `latency` of 4 and filter a simple signal to view the impact of applying a delay.

```
h = dfilt.delay(4)
h =
     FilterStructure: 'Delay'
             Latency: 4
    PersistentMemory: false

sig = 1:7      % Create some simple signal data
sig =
     1     2     3     4     5     6     7

states = h.states    % Filter states before filtering
states =
     0
     0
     0
     0

filter(h,sig)       % Filter using the delay filter
ans =
```

```
     0     0     0     0     1     2     3

states=h.states     % Filter states after filtering
states =
     4
     5
     6
     7
```

See Also

dfilt

dfilt.df1

Purpose

Discrete-time, direct-form I filter

Syntax

```
Hd = dfilt.df1(b,a)
Hd = dfilt.df1
```

Description

`Hd = dfilt.df1(b,a)` returns a discrete-time, direct-form I filter, `Hd`, with numerator coefficients `b` and denominator coefficients `a`. The filter states for this object are stored in a `filtstates` object.

`Hd = dfilt.df1` returns a default, discrete-time, direct-form I filter, `Hd`, with `b=1` and `a=1`. This filter passes the input through to the output unchanged.

Note The leading coefficient of the denominator `a(1)` cannot be 0.

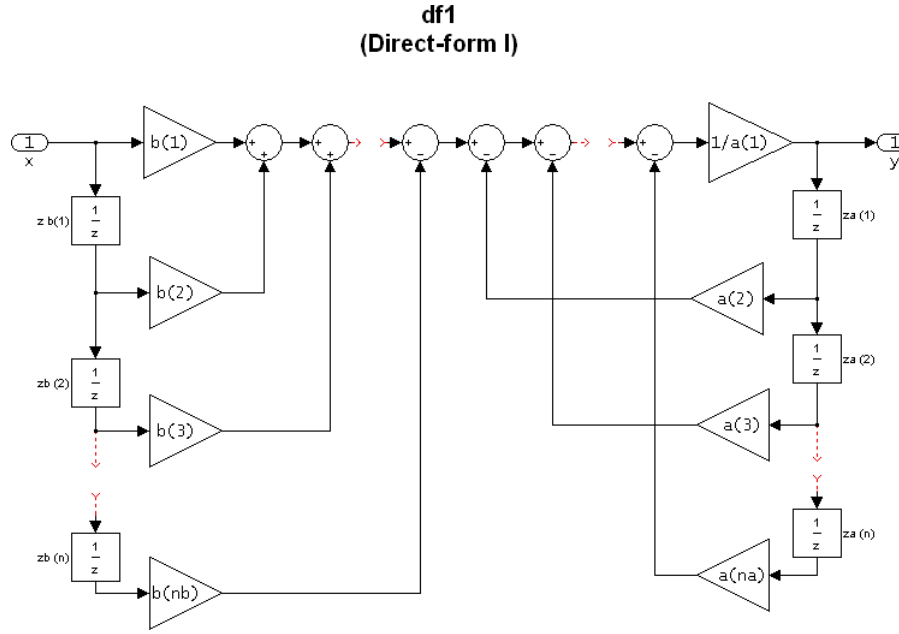


Image of direct form one filter diagram

To display the filter states, use this code to access the `filtstates` object.

```
Hs = Hd.states        % Where Hd is the dfilt.df1 object and
double (Hs)           % Hs is the filtstates object
```

The vector is

$$\begin{bmatrix} zb(1) \\ zb(2) \\ \dots \\ zb(n) \\ za(1) \\ za(2) \\ \dots \\ za(n) \end{bmatrix}$$

Examples

Create a direct-form I discrete-time filter with coefficients from a fourth-order lowpass Butterworth design

```
[b,a] = butter(4,.5);
Hd = dfilt.df1(b,a)

  FilterStructure: 'Direct-Form I'
     Numerator: [0.0940 0.3759 0.5639 0.3759 0.0940]
   Denominator: [1 -3.6082e-016 0.4860 3.6545e-017 0.0177]
PersistentMemory: false
```

See Also

`dfilt`, `dfilt.df1t`, `dfilt.df2`, `dfilt.df2t`

Purpose

Discrete-time, second-order section, direct-form I filter

Syntax

```
Hd = dfilt.df1sos(s)
Hd = dfilt.df1sos(b1,a1,b2,a2,...)
Hd = dfilt.df1sos(...,g)
Hd = dfilt.df1sos
```

Description

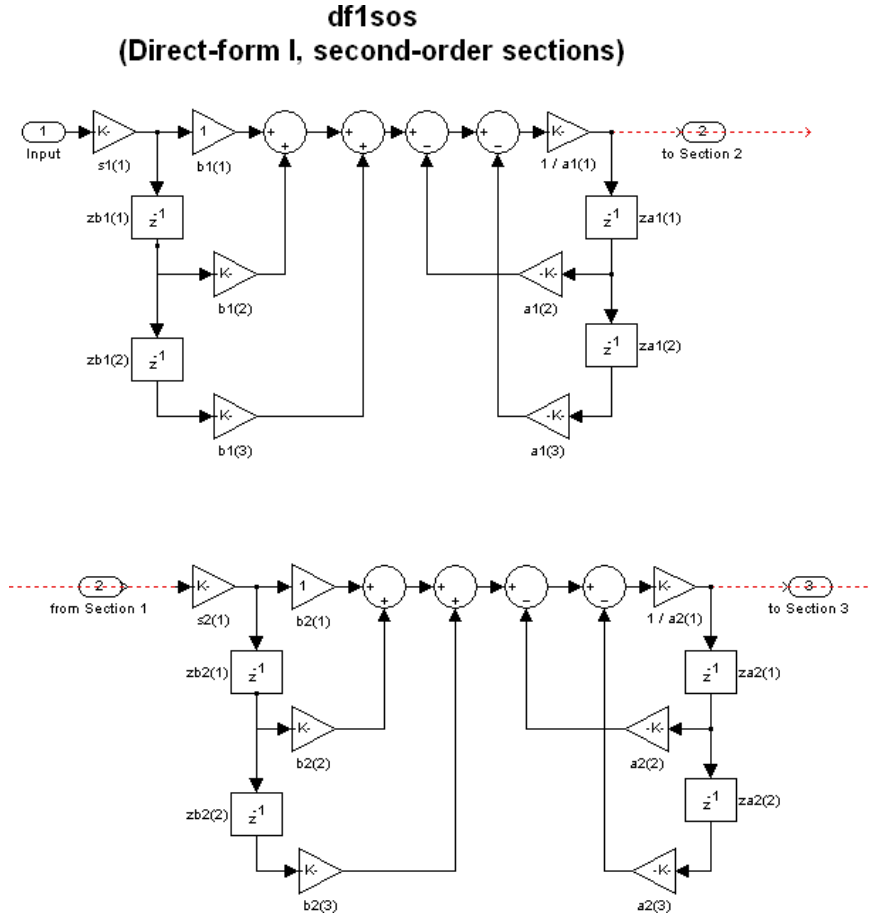
`Hd = dfilt.df1sos(s)` returns a discrete-time, second-order section, direct-form I filter, `Hd`, with coefficients given in the `s` matrix. The filter states for this object are stored in a `filtstates` object.

`Hd = dfilt.df1sos(b1,a1,b2,a2,...)` returns a discrete-time, second-order section, direct-form I filter, `Hd`, with coefficients for the first section given in the `b1` and `a1` vectors, for the second section given in the `b2` and `a2` vectors, etc.

`Hd = dfilt.df1sos(...,g)` includes a gain vector `g`. The elements of `g` are the gains for each section. The maximum length of `g` is the number of sections plus one. If `g` is not specified, all gains default to one.

`Hd = dfilt.df1sos` returns a default, discrete-time, second-order section, direct-form I filter, `Hd`. This filter passes the input through to the output unchanged.

Note The leading coefficient of the denominator `a(1)` cannot be 0.



To display the filter states, use this code to access the `filtstates` object.

```
Hs = Hd.states      % Where Hd is the dfilt.df1 object and
double (Hs)         % Hs is the filtstates object
```

The vector is

$$\begin{pmatrix} zb1(1) & zb2(1) \\ zb1(2) & zb2(2) \\ za1(1) & za2(1) \\ za1(2) & za2(2) \end{pmatrix}$$

For filters with more than one section, each section is a separate column in the matrix.

Examples

Specify a second-order sections, direct-form I discrete-time filter with coefficients from a sixth order, lowpass, elliptical filter using the following code. The resulting filter has three sections.

```
[z,p,k] = ellip(6,1,60,.4);  % Obtain filter coefficients
[s,g] = zp2sos(z,p,k);       % Convert to SOS
Hd = dfilt.df1sos(s,g)
Hd =
    FilterStructure: 'Direct-Form I, Second-Order Sections'
          sosMatrix: [3x6 double]
        ScaleValues: [0.0153280112138154;1;1;1]
OptimizeScaleValues: true
PersistentMemory: false
```

See Also

`dfilt`, `dfilt.df1tsos`, `dfilt.df2sos`, `dfilt.df2tsos`

Purpose

Discrete-time, direct-form I transposed filter

Syntax

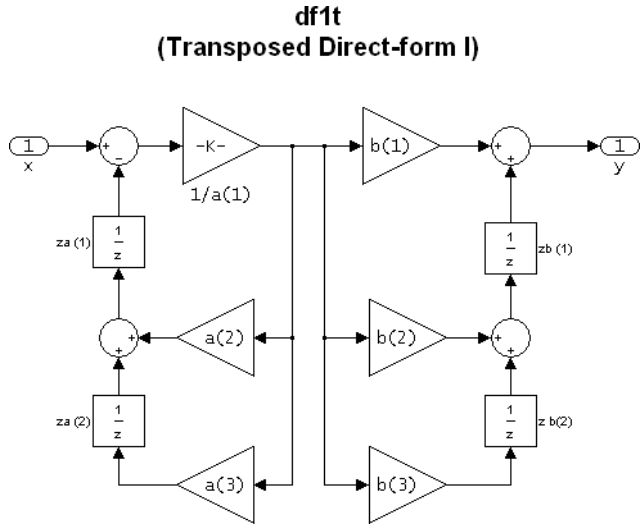
```
Hd = dfilt.df1t(b,a)
Hd = dfilt.df1t
```

Description

`Hd = dfilt.df1t(b,a)` returns a discrete-time, direct-form I transposed filter, `Hd`, with numerator coefficients `b` and denominator coefficients `a`. The filter states for this object are stored in a `filtstates` object.

`Hd = dfilt.df1t` returns a default, discrete-time, direct-form I transposed filter, `Hd`, with `b`=1 and `a`=1. This filter passes the input through to the output unchanged.

Note The leading coefficient of the denominator `a(1)` cannot be 0.



To display the filter states, use this code to access the `filtstates` object.

```
Hs = Hd.states     % Where Hd is the dfilt.df1 object and
```

```
double (Hs)        % Hs is the filtstates object
```

The vector is

$$\begin{pmatrix} zb1(1) & zb2(1) \\ zb1(2) & zb2(2) \\ za1(1) & za2(1) \\ za1(2) & za2(2) \end{pmatrix}$$

Examples

Create a direct-form I transposed discrete-time filter with coefficients from a fourth-order lowpass Butterworth design:

```
[b,a] = butter(4,.5);
Hd = dfilt.df1t(b,a)
Hd =
FilterStructure: 'Direct-Form I Transposed'
      Numerator: [0.0940 0.3759 0.5639 0.3759 0.0940]
    Denominator: [1 -3.6082e-016 0.4860 3.6545e-017 0.0177]
  PersistentMemory: false
```

See Also

`dfilt`, `dfilt.df1`, `dfilt.df2`, `dfilt.df2t`

dfilt.df1tsos

Purpose

Discrete-time, second-order section, direct-form I transposed filter

Syntax

```
Hd = dfilt.df1tsos(s)
Hd = dfilt.df1tsos(b1,a1,b2,a2,...)
Hd = dfilt.df1tsos(...,g)
Hd = dfilt.df1tsos
```

Description

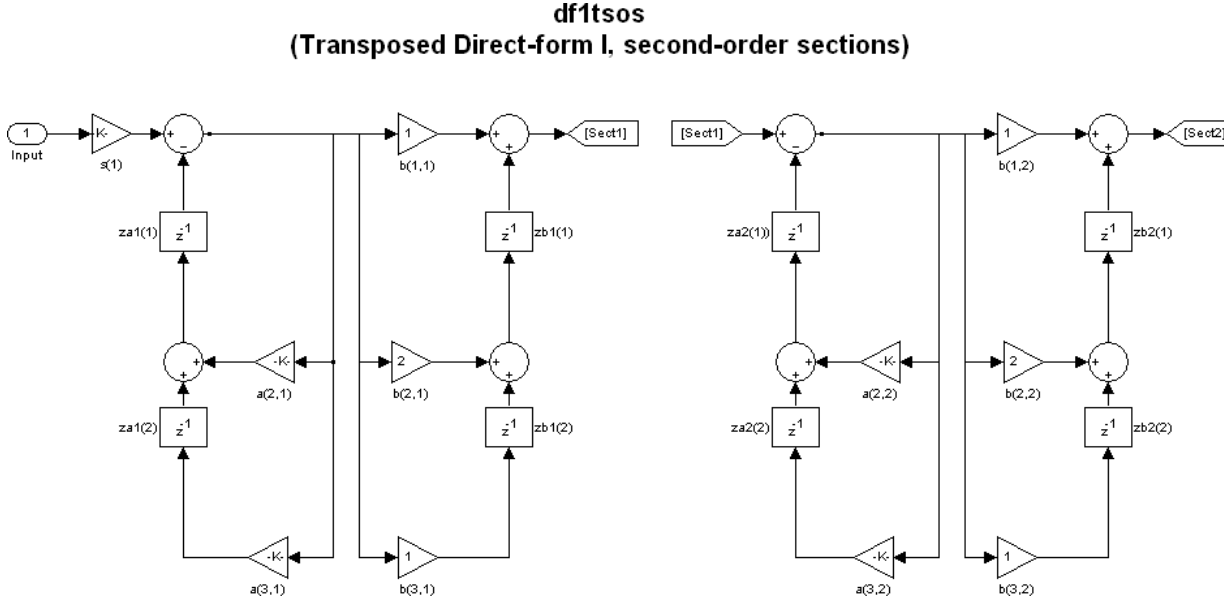
`Hd = dfilt.df1tsos(s)` returns a discrete-time, second-order section, direct-form I, transposed filter, `Hd`, with coefficients given in the `s` matrix. The filter states for this object are stored in a `filtstates` object.

`Hd = dfilt.df1tsos(b1,a1,b2,a2,...)` returns a discrete-time, second-order section, direct-form I, tranposed filter, `Hd`, with coefficients for the first section given in the `b1` and `a1` vectors, for the second section given in the `b2` and `a2` vectors, etc.

`Hd = dfilt.df1tsos(...,g)` includes a gain vector `g`. The elements of `g` are the gains for each section. The maximum length of `g` is the number of sections plus one. If `g` is not specified, all gains default to one.

`Hd = dfilt.df1tsos` returns a default, discrete-time, second-order section, direct-form I, transposed filter, `Hd`. This filter passes the input through to the output unchanged.

Note The leading coefficient of the denominator `a(1)` cannot be 0.



To display the filter states, use this code to access the `filtstates` object.

```
Hs = Hd.states      % Where Hd is the dfilt.df1 object and
double (Hs)         % Hs is the filtstates object
```

The matrix is

$$\begin{pmatrix} zb1(1) & zb2(1) \\ zb1(2) & zb2(2) \\ za1(1) & za2(1) \\ za1(2) & za2(2) \end{pmatrix}$$

Examples

Specify a second-order sections, direct-form I, transposed discrete-time filter with coefficients from a sixth order, lowpass, elliptical filter using the following code:

```
[z,p,k] = ellip(6,1,60,.4);    % Obtain filter coefficients
```

```
[s,g] = zp2sos(z,p,k);         % Convert to SOS
Hd = dfilt.df1tsos(s,g)
Hd =
     FilterStructure: [1x47 char]
           sosMatrix: [3x6 double]
         ScaleValues: [0.0153280112138154;1;1;1]
 OptimizeScaleValues: true
    PersistentMemory: false

Hd.FilterStructure     % Display FilterStructure string
ans =

Direct-Form I Transposed, Second-Order Sections
```

See Also

dfilt, dfilt.df1sos, dfilt.df2sos, dfilt.df2tsos

Purpose

Discrete-time, direct-form II filter

Syntax

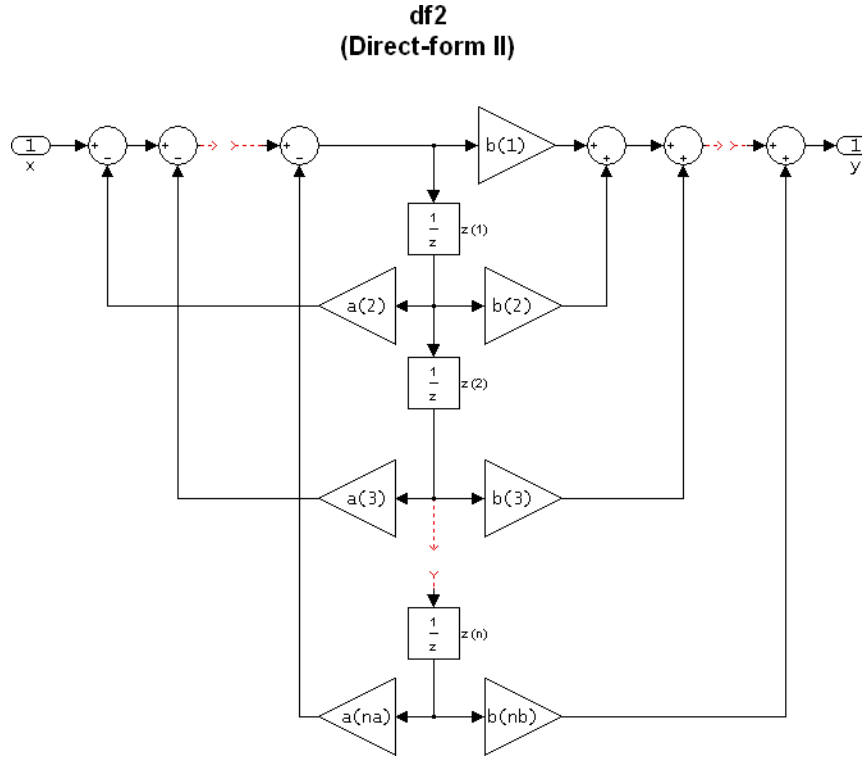
```
Hd = dfilt.df2(b,a)
Hd = dfilt.df2
```

Description

`Hd = dfilt.df2(b,a)` returns a discrete-time, direct-form II filter, `Hd`, with numerator coefficients `b` and denominator coefficients `a`.

`Hd = dfilt.df2` returns a default, discrete-time, direct-form II filter, `Hd`, with `b`=1 and `a`=1. This filter passes the input through to the output unchanged.

Note The leading coefficient of the denominator `a(1)` cannot be 0.



The resulting filter states column vector is

$$\begin{bmatrix} z(1) \\ z(2) \\ \dots \\ z(n) \end{bmatrix}$$

Examples

Create a direct-form II discrete-time filter with coefficients from a fourth-order lowpass Butterworth design:

```
[b,a] = butter(4,.5);
Hd = dfilt.df2(b,a)
```

```
Hd =

 FilterStructure: 'Direct-Form II'
    Numerator: [0.0940 0.3759 0.5639 0.3759 0.0940]
  Denominator: [1 -3.6082e-016 0.4860 3.6545e-017 0.0177]
 PersistentMemory: false
```

See Also

dfilt, dfilt.df1, dfilt.df1t, dfilt.df2t

dfilt.df2sos

Purpose

Discrete-time, second-order section, direct-form II filter

Syntax

```
Hd = dfilt.df2sos(s)
Hd = dfilt.df2sos(b1,a1,b2,a2,...)
Hd = dfilt.df2sos(...,g)
Hd = dfilt.df2sos
```

Description

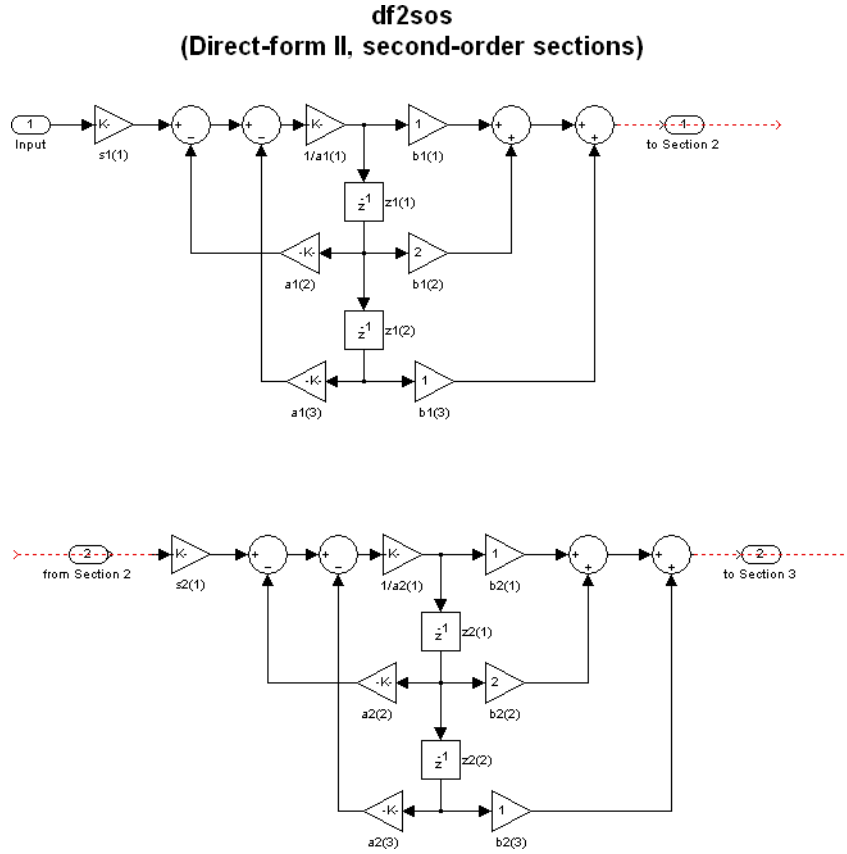
`Hd = dfilt.df2sos(s)` returns a discrete-time, second-order section, direct-form II filter, `Hd`, with coefficients given in the `s` matrix.

`Hd = dfilt.df2sos(b1,a1,b2,a2,...)` returns a discrete-time, second-order section, direct-form II object, `Hd`, with coefficients for the first section given in the `b1` and `a1` vectors, for the second section given in the `b2` and `a2` vectors, etc.

`Hd = dfilt.df2sos(...,g)` includes a gain vector `g`. The elements of `g` are the gains for each section. The maximum length of `g` is the number of sections plus one. If `g` is not specified, all gains default to one.

`Hd = dfilt.df2sos` returns a default, discrete-time, second-order section, direct-form II filter, `Hd`. This filter passes the input through to the output unchanged.

Note The leading coefficient of the denominator `a(1)` cannot be 0.



The resulting filter states column vector is

$$\begin{pmatrix} z1(1) & z2(1) \\ z1(2) & z2(2) \end{pmatrix}$$

For filters with more than one section, each section is a separate column in the vector.

Examples

Specify a second-order sections, direct-form II discrete-time filter with coefficients from a sixth order, lowpass, elliptical filter using the following code:

```
[z,p,k] = ellip(6,1,60,.4);   % Obtain filter coefficients
[s,g] = zp2sos(z,p,k);        % Convert to SOS
Hd = dfilt.df2sos(s,g)
Hd =
         FilterStructure: [1x37 char]
               sosMatrix: [3x6 double]
             ScaleValues: [0.0153280112138154;1;1;1]
     OptimizeScaleValues: true
        PersistentMemory: false

Hd.FilterStructure     % Display FilterStructure string
ans =
Direct-Form II Transposed, Second-Order Sections
```

See Also

`dfilt`, `dfilt.df1sos`, `dfilt.df1tsos`, `dfilt.df2tsos`

Purpose

Discrete-time, direct-form II transposed filter

Syntax

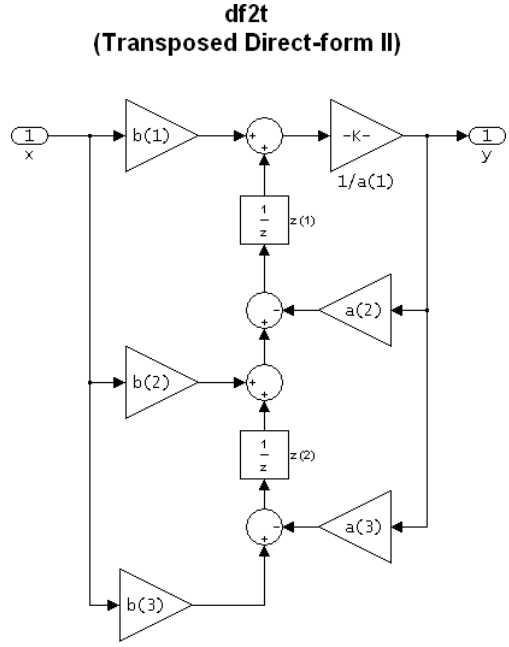
```
Hd = dfilt.df2t(b,a)
Hd = dfilt.df2t
```

Description

`Hd = dfilt.df2t(b,a)` returns a discrete-time, direct-form II transposed filter, `Hd`, with numerator coefficients `b` and denominator coefficients `a`.

`Hd = dfilt.df2t` returns a default, discrete-time, direct-form II transposed filter, `Hd`, with `b`=1 and `a`=1. This filter passes the input through to the output unchanged.

Note The leading coefficient of the denominator `a(1)` cannot be 0.



The resulting filter states column vector is

$$\begin{pmatrix} z1(1) & z2(1) \\ z1(2) & z2(2) \end{pmatrix}$$

Examples

Create a direct-form II transposed discrete-time filter with coefficients from a fourth-order lowpass Butterworth design:

```
[b,a] = butter(4,.5);
Hd = dfilt.df2t(b,a)
Hd =
 FilterStructure: 'Direct-Form II Transposed'
    Numerator: [0.0940 0.3759 0.5639 0.3759 0.0940]
  Denominator: [1 -3.6082e-016 0.4860 3.6545e-017 0.0177]
 PersistentMemory: false
```

See Also

`dfilt`, `dfilt.df1`, `dfilt.df1t`, `dfilt.df2`

Purpose

Discrete-time, second-order section, direct-form II transposed filter

Syntax

```
Hd = dfilt.df2sos(s)
Hd = dfilt.df2tsos(b1,a1,b2,a2,...)
Hd = dfilt.df2tsos(...,g)
Hd = dfilt.df2tso
```

Description

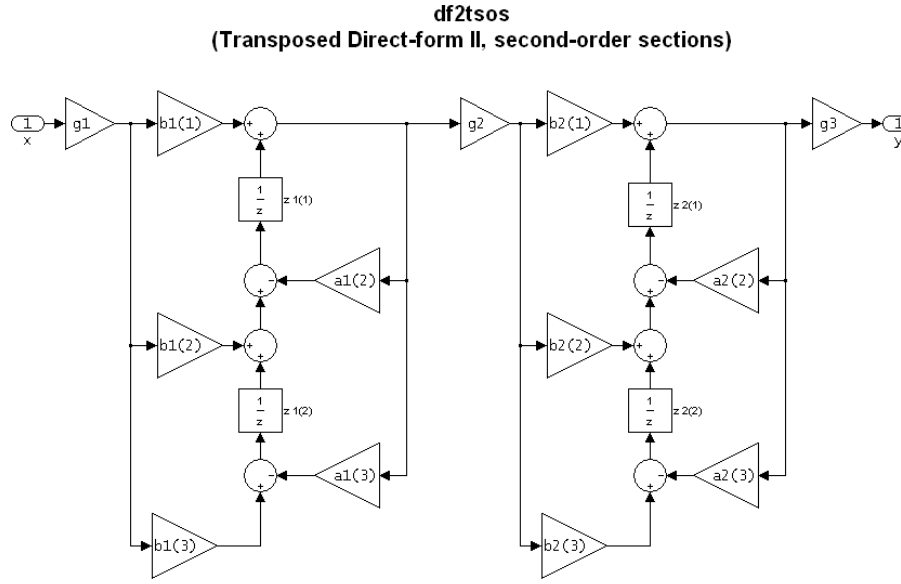
`Hd = dfilt.df2sos(s)` returns a discrete-time, second-order section, direct-form II, transposed filter, `Hd`, with coefficients given in the `s` matrix.

`Hd = dfilt.df2tsos(b1,a1,b2,a2,...)` returns a discrete-time, second-order section, direct-form II, tranposed filter, `Hd`, with coefficients for the first section given in the `b1` and `a1` vectors, for the second section given in the `b2` and `a2` vectors, etc.

`Hd = dfilt.df2tsos(...,g)` includes a gain vector `g`. The elements of `g` are the gains for each section. The maximum length of `g` is the number of sections plus one. If `g` is not specified, all gains default to one.

`Hd = dfilt.df2tso` returns a default, discrete-time, second-order section, direct-form II, transposed filter, `Hd`. This filter passes the input through to the output unchanged.

Note The leading coefficient of the denominator `a(1)` cannot be 0.



The resulting filter states column vector is

$$\begin{pmatrix} z1(1) & z2(1) \\ z1(2) & z2(2) \end{pmatrix}$$

Examples

Specify a second-order sections, direct-form II, transposed discrete-time filter with coefficients from a sixth order, lowpass, elliptical filter using the following code:

```
[z,p,k] = ellip(6,1,60,.4);  % Obtain filter coefficients
[s,g] = zp2sos(z,p,k);       % Convert to SOS
Hd = dfilt.df2tsos(s,g)
Hd =
         FilterStructure: [1x48 char]
               sosMatrix: [3x6 double]
            ScaleValues: [0.0153280112138154;1;1;1]
     OptimizeScaleValues: true
        PersistentMemory: false
```

See Also

dfilt, dfilt.df1sos, dfilt.df1tsos, dfilt.df2sos

dfilt.dfasymfir

Purpose

Discrete-time, direct-form antisymmetric FIR filter

Syntax

```
Hd = dfilt.dfasymfir(b)
Hd = dfilt.dfasymfir
```

Description

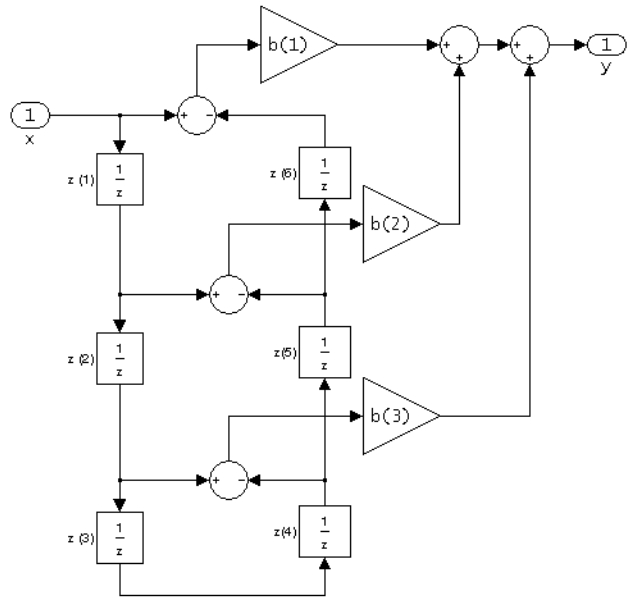
`Hd = dfilt.dfasymfir(b)` returns a discrete-time, direct-form, antisymmetric FIR filter, `Hd`, with numerator coefficients `b`.

`Hd = dfilt.dfasymfir` returns a default, discrete-time, direct-form, antisymmetric FIR filter, `Hd`, with `b`=1. This filter passes the input through to the output unchanged.

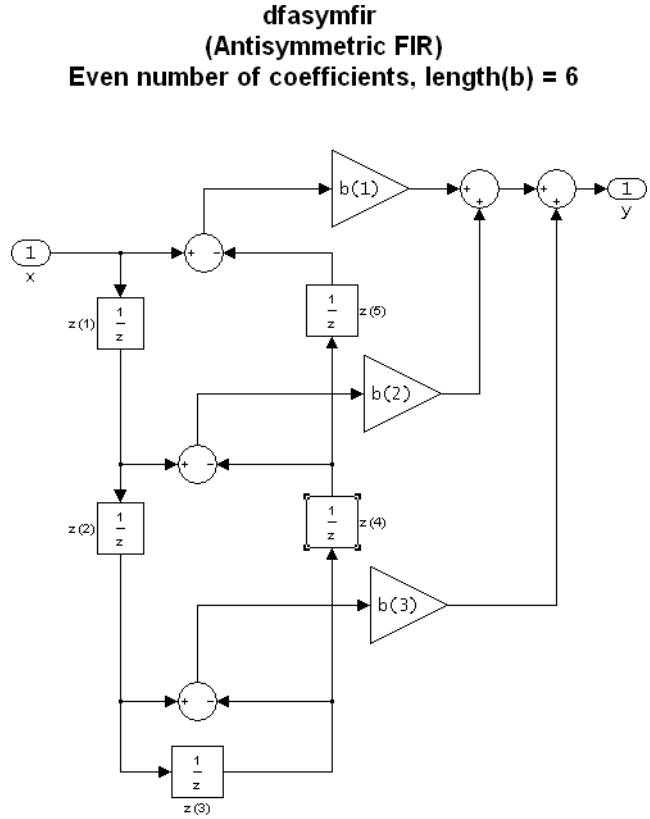
Note Only the first half of vector `b` is used because the second half is assumed to be antisymmetric. In the figure below for an odd number of coefficients, b(3) = 0, b(4) = -b(2) and b(5) = -b(1), and in the next figure for an even number of coefficients, b(4) = -b(3), b(5) = -b(2), and b(6) = -b(1).

---

dfasymfir
(Antisymmetric FIR)
Even order
Odd number of coefficients, length(b) = 7



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



b(i) == -b(end - i + 1)

The resulting filter states column vector for the odd number of coefficients example above is

$$\begin{bmatrix} z(1) \\ z(2) \\ z(3) \\ z(4) \\ z(5) \\ z(6) \end{bmatrix}$$

Examples

Odd Order

Create a Type 4 25th order highpass direct-form antisymmetric FIR filter structure for a `dfilt` object, `Hd`, with the following code:

```
Hd = firpm(25,[0 .4 .5 1],[0 0 1 1],'h');
```

Even Order

Create a 44th order lowpass direct-form antisymmetric FIR differentiator filter structure for a `dfilt` object, `Hd`, with the following code:

```
h=firpm(44,[0 .3 .4 1],[0 .2 0 0],'differentiator');
```

See Also

`dfilt`, `dfilt.dffir`, `dfilt.dffirt`, `dfilt.dfsymfir`

dfilt.dffir

Purpose

Discrete-time, direct-form, FIR filter

Syntax

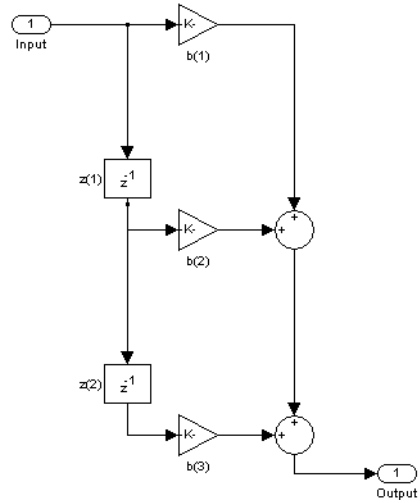
```
Hd = dfilt.dffir(b)
Hd = dfilt.dffir
```

Description

`Hd = dfilt.dffir(b)` returns a discrete-time, direct-form finite impulse response (FIR) filter, `Hd`, with numerator coefficients, `b`.

`Hd = dfilt.dffir` returns a default, discrete-time, direct-form FIR filter, `Hd`, with `b=1`. This filter passes the input through to the output unchanged.

dffir
(Direct-form FIR = Tapped delay line)



The resulting filter states column vector is

$$\begin{pmatrix} z1(1) & z2(1) \\ z1(2) & z2(2) \end{pmatrix}$$

Examples

Create a direct-form FIR discrete-time filter with coefficients from a 30^{th} order lowpass equiripple design:

```
b = firpm(30,[0 .1 .2 .5]*2,[1 1 0 0]);
Hd = dfilt.dffir(b)
Hd =
          FilterStructure: 'Direct-Form FIR'
                Numerator: [1x31 double]
         PersistentMemory: false
```

See Also

`dfilt`, `dfilt.dfasymfir`, `dfilt.dffirt`, `dfilt.dfsymfir`

Purpose

Discrete-time, direct-form FIR transposed filter

Syntax

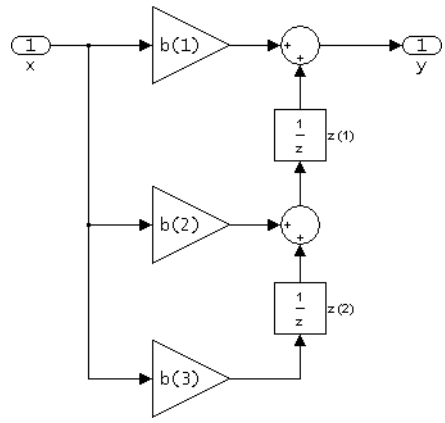
```
Hd = dfilt.dffirt(b)
Hd = dfilt.dffirt
```

Description

`Hd = dfilt.dffirt(b)` returns a discrete-time, direct-form FIR transposed filter, `Hd`, with numerator coefficients `b`.

`Hd = dfilt.dffirt` returns a default, discrete-time, direct-form FIR transposed filter, `Hd`, with `b=1`. This filter passes the input through to the output unchanged.

dffirt
(Transposed Direct-form FIR)



The resulting filter states column vector is

$$\begin{pmatrix} z1(1) & z2(1) \\ z1(2) & z2(2) \end{pmatrix}$$

Examples

Create a direct-form FIR transposed discrete-time filter with coefficients from a 30th order lowpass equiripple design:

```
b = firpm(30,[0 .1 .2 .5]*2,[1 1 0 0]);
Hd = dfilt.dffirt(b)
Hd =
          FilterStructure: 'Direct-Form FIR Transposed'
                Numerator: [1x31 double]
         PersistentMemory: false
```

See Also

dfilt, dfilt.dffir, dfilt.dfasymfir, dfilt.dfsymfir

dfilt.dfsymfir

Purpose

Discrete-time, direct-form symmetric FIR filter

Syntax

```
Hd = dfilt.dfsymfir(b)
Hd = dfilt.dfsymfir
```

Description

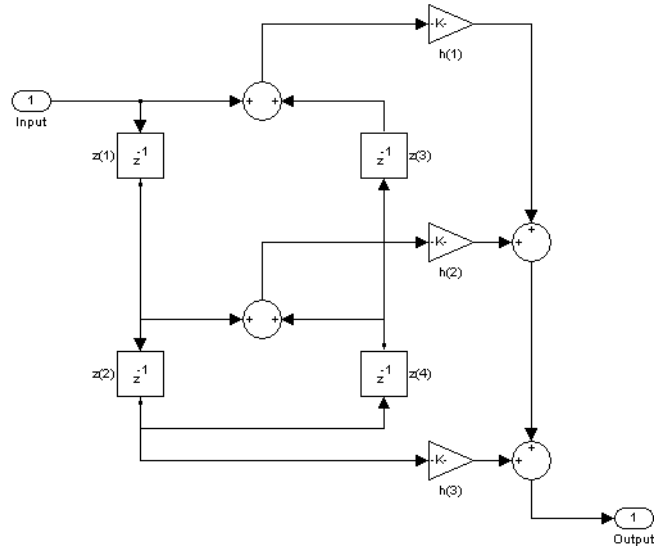
`Hd = dfilt.dfsymfir(b)` returns a discrete-time, direct-form symmetric FIR filter, `Hd`, with numerator coefficients `b`.

`Hd = dfilt.dfsymfir` returns a default, discrete-time, direct-form symmetric FIR filter, `Hd`, with `b=1`. This filter passes the input through to the output unchanged.

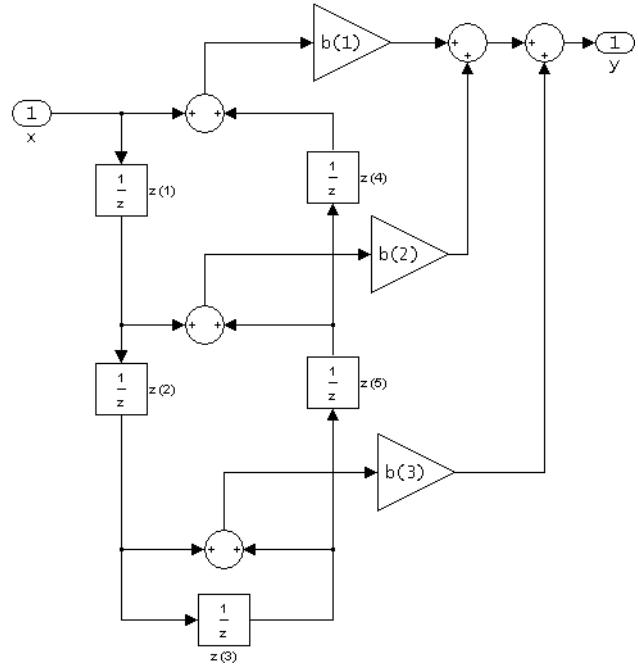
Note Only the first half of vector `b` is used because the second half is assumed to be symmetric. In the figure below for an odd number of coefficients, b(3) = 0, b(4) = b(2) and b(5) = b(1), and in the next figure for an even number of coefficients, b(4) = b(3), b(5) = b(2), and b(6) = b(1).

---

dfsymfir
(Symmetric FIR)
Even order
Odd number of coefficients, length(b) = 5
b(i) == b(end - i + 1)



dfsymfir
(Symmetric FIR)
Odd order
Even number of coefficients, length(b) = 6
b(i) == b(end - i + 1)



The resulting filter states column vector for the odd number of coefficients example above is

$$\begin{bmatrix} z(1) \\ z(2) \\ z(3) \\ z(4) \end{bmatrix}$$

Examples

Odd Order

Specify a fifth-order direct-form symmetric FIR filter structure for a `dfilt` object, `Hd`, with the following code:

```
b = [-0.008 0.06 0.44 0.44 0.06 -0.008];
Hd = dfilt.dfsymfir(b)
Hd =
FilterStructure: 'Direct-Form Symmetric FIR'
 Numerator: [-0.0080 0.0600 0.4400 0.4400 0.0600 -0.0080]
PersistentMemory: false
```

Even Order

Specify a fourth-order direct-form symmetric FIR filter structure for a `dfilt` object, `Hd`, with the following code:

```
b = [-0.01 0.1 0.8 0.1 -0.01];
Hd = dfilt.dfsymfir(b)
Hd =
FilterStructure: 'Direct-Form Symmetric FIR'
  Numerator: [-0.0100 0.1000 0.8000 0.1000 -0.0100]
PersistentMemory: false
```

See Also

`dfilt`, `dfilt.dfasymfir`, `dfilt.dffir`, `dfilt.dffirt`

dfilt.fftfir

Purpose

Discrete-time, overlap-add, FIR filter

Syntax

```
Hd = dfilt.fftfir(b,len)
Hd = dfilt.fftfir(b)
Hd = dfilt.fftfir
```

Description

This object uses the overlap-add method of block FIR filtering, which is very efficient for streaming data.

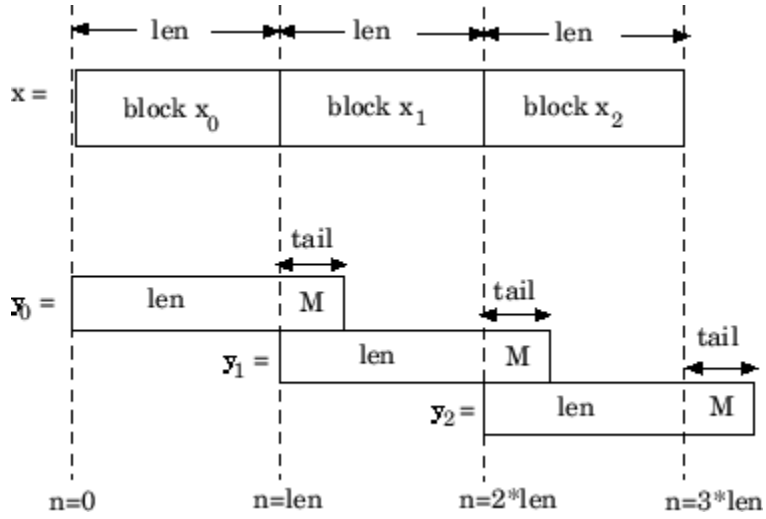
`Hd = dfilt.fftfir(b,len)` returns a discrete-time, FFT, FIR filter, `Hd`, with numerator coefficients, `b` and block length, `len`. The block length is the number of input points to use for each overlap-add computation.

`Hd = dfilt.fftfir(b)` returns a discrete-time, FFT, FIR filter, `Hd`, with numerator coefficients, `b` and block length, `len`=100.

`Hd = dfilt.fftfir` returns a default, discrete-time, FFT, FIR filter, `Hd`, with the numerator `b=1` and block length, `len`=100. This filter passes the input through to the output unchanged.

Note When you use a `dfilt.fftfir` object to filter, the input signal length must be an integer multiple of the object's block length, `len`. The resulting number of FFT points = (filter length + the block length - 1). The filter is most efficient if the number of FFT points is a power of 2.

The `fftfir` uses an overlap-add block processing algorithm, which is represented as follows,



where `len` is the block length and `M` is the length of the numerator-1, (`length(b)-1`), which is also the number of states. The output of each convolution is a block that is longer than the input block by a tail of (`length(b)-1`) samples. These tails overlap the next block and are added to it. The states reported by `dfilt.fftfir` are the tails of the final convolution.

Examples

Create an FFT FIR discrete-time filter with coefficients from a 30th order lowpass equiripple design:

```
b = firpm(30,[0 .1 .2 .5]*2,[1 1 0 0]);
Hd = dfilt.fftfir(b)
Hd =
        FilterStructure: 'Overlap-Add FIR'
              Numerator: [1x31 double]
            BlockLength: 100
    NonProcessedSamples: []
       PersistentMemory: false
```

To view the frequency domain coefficients used in the filtering, use the following command.

```
fftcoeffs(Hd)
```

See Also

`dfilt`, `dfilt.dffir`, `dfilt.dfasymfir`, `dfilt.dffirt`, `dfilt.dfsymfir`

Purpose

Discrete-time, lattice allpass filter

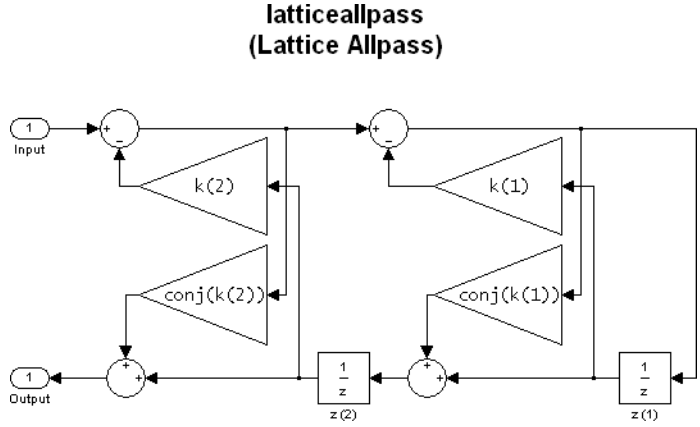
Syntax

```
Hd = dfilt.latticeallpass(k)
Hd = dfilt.latticeallpass
```

Description

`Hd = dfilt.latticeallpass(k)` returns a discrete-time, lattice allpass filter, `Hd`, with lattice coefficients, `k`.

`Hd = dfilt.latticeallpass` returns a default, discrete-time, lattice allpass filter, `Hd`, with `k=[ ]`. This filter passes the input through to the output unchanged.



The resulting filter states column vector is

$$\begin{pmatrix} z1(1) & z2(1) \\ z1(2) & z2(2) \end{pmatrix}$$

Examples

Form a third-order lattice allpass filter structure for a `dfilt` object, `Hd`, using the following lattice coefficients:

```
k = [.66 .7 .44];
Hd = dfilt.latticeallpass(k)
Hd =
```

```
                FilterStructure: 'Lattice Allpass'
                        Lattice: [0.6600 0.7000 0.4400]
               PersistentMemory: false
```

See Also

dfilt, dfilt.latticear, dfilt.latticearma, dfilt.latticemamax, dfilt.latticemamin

Purpose

Discrete-time, lattice, autoregressive filter

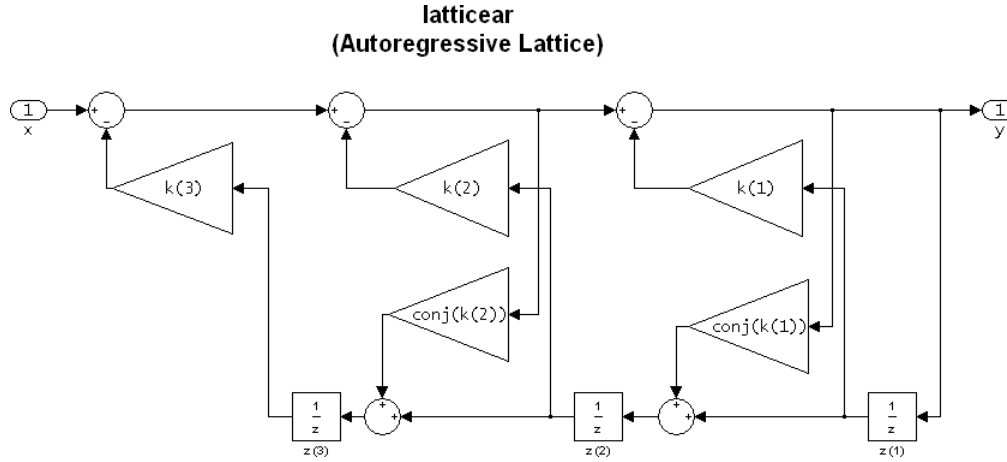
Syntax

```
Hd = dfilt.latticear(k)
Hd = dfilt.latticear
```

Description

`Hd = dfilt.latticear(k)` returns a discrete-time, lattice autoregressive filter, `Hd`, with lattice coefficients, `k`.

`Hd = dfilt.latticear` returns a default, discrete-time, lattice autoregressive filter, `Hd`, with `k=[ ]`. This filter passes the input through to the output unchanged.



The resulting filter states column vector is

$$\begin{bmatrix} z(1) \\ z(2) \\ z(3) \end{bmatrix}$$

Examples

Form a third-order lattice autoregressive filter structure for a `dfilt` object, `Hd`, using the following lattice coefficients:

```
k = [.66 .7 .44];
```

```
Hd = dfilt.latticear(k)
Hd =
             FilterStructure: 'Lattice Autoregressive (AR)'
                     Lattice: [0.6600 0.7000 0.4400]
                PersistentMemory: false
```

See Also

dfilt, dfilt.latticeallpass, dfilt.latticearma, dfilt.latticemamax, dfilt.latticemamin

Purpose

Discrete-time, lattice, autoregressive, moving-average filter

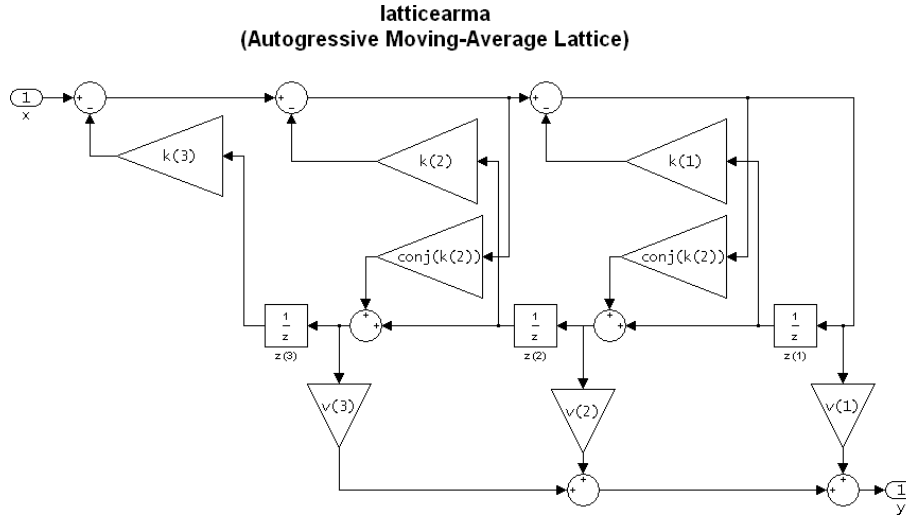
Syntax

```
Hd = dfilt.latticearma(k,v)
Hd = dfilt.latticearma
```

Description

`Hd = dfilt.latticearma(k,v)` returns a discrete-time, lattice autoregressive, moving-average filter, `Hd`, with lattice coefficients, `k` and ladder coefficients `v`.

`Hd = dfilt.latticearma` returns a default, discrete-time, lattice autoregressive, moving-average filter, `Hd`, with `k=[ ]` and `v=1`. This filter passes the input through to the output unchanged.



The resulting filter states column vector is

$$\begin{bmatrix} z(1) \\ z(2) \\ z(3) \end{bmatrix}$$

Examples

Form a third-order lattice autoregressive, moving-average filter structure for a `dfilt` object, `Hd`, using the following lattice coefficients:

```
k = [.66 .7 .44];
Hd = dfilt.latticearma(k)

Hd =
  FilterStructure: 'Lattice Autoregressive Moving-
                    Average (ARMA)'
          Lattice: [0.6600 0.7000 0.4400]
           Ladder: 1
 PersistentMemory: false
```

See Also

`dfilt`, `dfilt.latticeallpass`, `dfilt.latticear`, `dfilt.latticemamax`, `dfilt.latticemamin`

Purpose

Discrete-time, lattice, moving-average filter

Syntax

```
Hd = dfilt.latticemamax(k)
Hd = dfilt.latticemamax
```

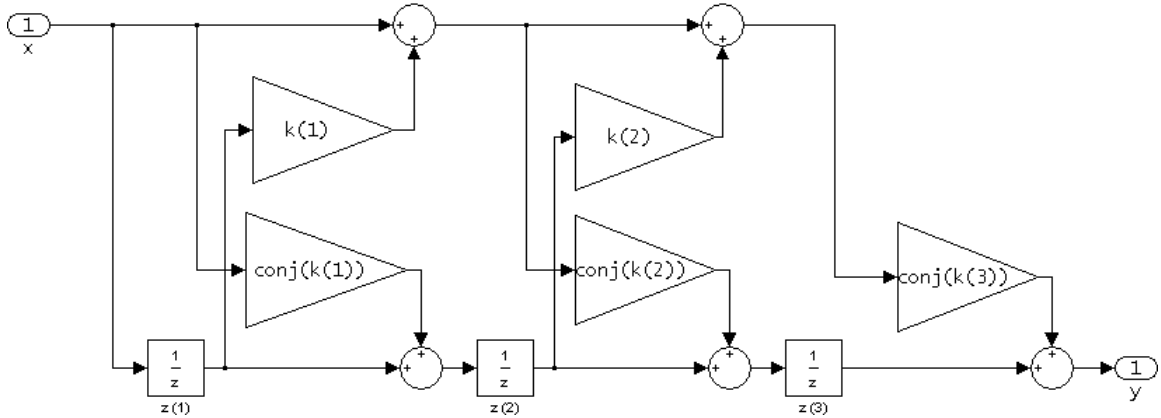
Description

`Hd = dfilt.latticemamax(k)` returns a discrete-time, lattice, moving-average filter, `Hd`, with lattice coefficients `k`.

Note If the `k` coefficients define a maximum phase filter, the resulting filter in this structure is maximum phase. If your coefficients do not define a maximum phase filter, placing them in this structure does not produce a maximum phase filter.

`Hd = dfilt.latticemamax` returns a default discrete-time, lattice, moving-average filter, `Hd`, with `k=[ ]`. This filter passes the input through to the output unchanged.

latticemamax
(Moving-Average, Maximum Phase Lattice)



The resulting filter states column vector is

$$\begin{bmatrix} z(1) \\ z(2) \\ z(3) \end{bmatrix}$$

Examples

Form a fourth-order lattice, moving-average, maximum phase filter structure for a `dfilt` object, `Hd`, using the following lattice coefficients:

```
k = [.66 .7 .44 .33];
Hd = dfilt.latticemamax(k)
Hd =
  FilterStructure: 'Lattice Moving-Average (MA)
                    For Maximum Phase'
       Arithmetic: 'double'
          Lattice: [0.6600 0.7000 0.4400 0.3300]
 PersistentMemory: false
```

See Also

`dfilt`, `dfilt.latticeallpass`, `dfilt.latticear`, `dfilt.latticearma`, `dfilt.latticemamin`

Purpose

Discrete-time, lattice, moving-average filter

Syntax

```
Hd = dfilt.latticemamin(k)
Hd = dfilt.latticemamin
```

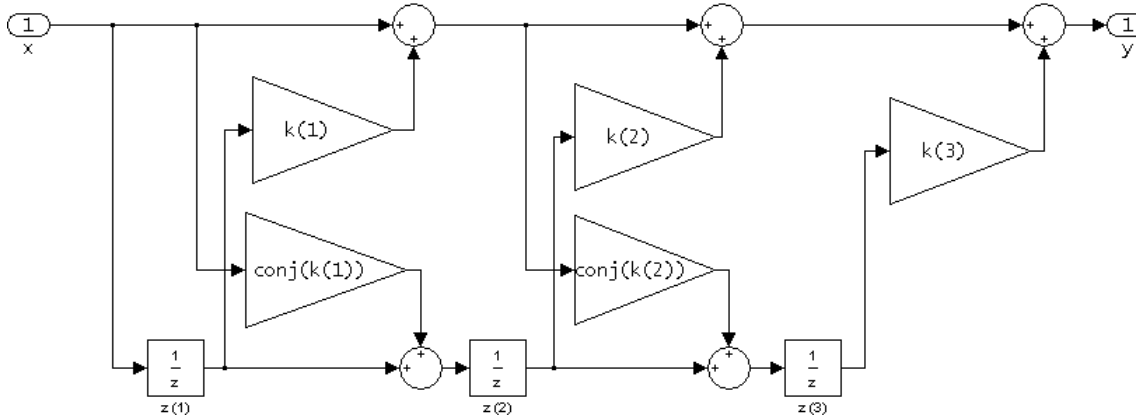
Description

`Hd = dfilt.latticemamin(k)` returns a discrete-time, lattice, moving-average, minimum phase, filter, `Hd`, with lattice coefficients `k`.

Note If the `k` coefficients define a minimum phase filter, the resulting filter in this structure is minimum phase. If your coefficients do not define a minimum phase filter, placing them in this structure does not produce a minimum phase filter.

`Hd = dfilt.latticemamin` returns a default discrete-time, lattice, moving-average, minimum phase, filter, `Hd`, with `k=[ ]`. This filter passes the input through to the output unchanged.

latticemamin
(Moving-Average, Minimum Phase Lattice)



The resulting filter states column vector is

$$\begin{bmatrix} z(1) \\ z(2) \\ z(3) \end{bmatrix}$$

Examples

Form a third-order lattice, moving-average, minimup phase, filter structure for a `dfilt` object, `Hd`, using the following lattice coefficients.

```
k = [.66 .7 .44];
Hd = dfilt.latticemamin(k)
Hd =
  FilterStructure: 'Lattice Moving-Average (MA)
                    For Minimum Phase'
          Lattice: [0.6600 0.7000 0.4400]
PersistentMemory: false
```

See Also

`dfilt`, `dfilt.latticeallpass`, `dfilt.latticear`, `dfilt.latticearma`, `dfilt.latticemamax`

Purpose

Discrete-time, parallel structure filter

Syntax

```
Hd = dfilt.parallel(Hd1,Hd2,...)
```

Description

`Hd = dfilt.parallel(Hd1,Hd2,...)` returns a discrete-time filter, `Hd`, which is a structure of two or more `dfilt` filters, `Hd1`, `Hd2`, etc. arranged in parallel. Each filter in a parallel structure is a separate stage. You can display states for individual stages only. To view the states of a stage use

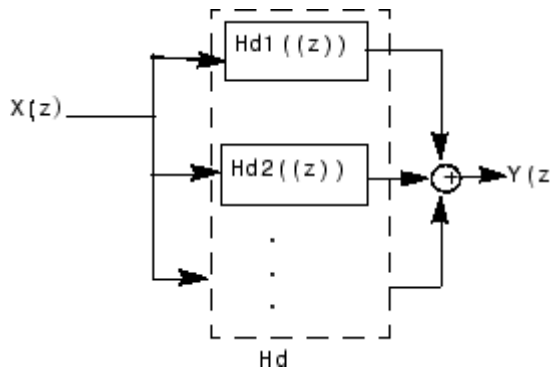
```
Hd.stage(1).states
```

To append a filter (`Hd3`) onto an existing parallel filter (`Hd`), use

```
Hd = addstage(Hd3)
```

You can also use the nondot notation format for calling a parallel structure.

```
parallel(Hd1,Hd2,...)
```



Examples

Using a parallel structure, create a coupled-allpass decomposition of a 7th order lowpass digital, elliptic filter with a normalized cutoff frequency of 0.5, 1 decibel of peak-to-peak ripple and a minimum stopband attenuation of 40 decibels.

```
k1 = [-0.0154    0.9846   -0.3048    0.5601];
Hd1 = dfilt.latticeallpass(k1);
k2 = [-0.1294    0.8341   -0.4165];
Hd2 = dfilt.latticeallpass(k2);
Hpar = parallel(Hd1 ,Hd2);
gain = dfilt.scalar(0.5);    % Normalize passband gain
Hcas = cascade(gain,Hpar);
```

For details on the stages of this filter, use

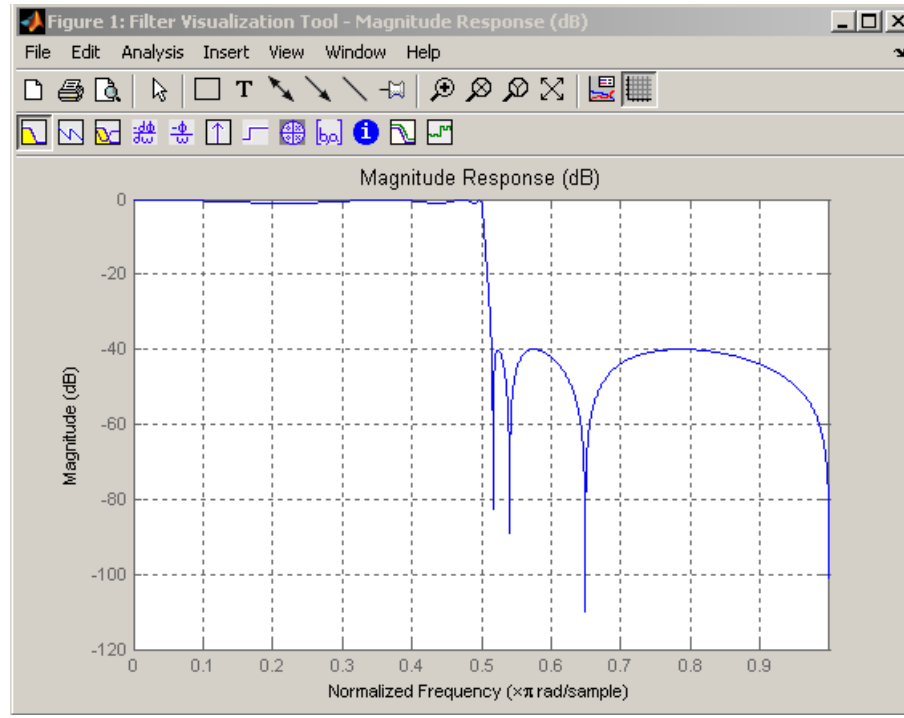
```
info(Hcas.Stage(1))
```

and

```
info(Hcas.Stage(2))
```

To view this filter, use

```
fvtool(Hcas)
```



See Also

`dfilt`, `dfilt.cascade`

dfilt.scalar

Purpose

Discrete-time, scalar filter

Syntax

```
Hd = dfilt.scalar(g)
Hd = dfilt.scalar
```

Description

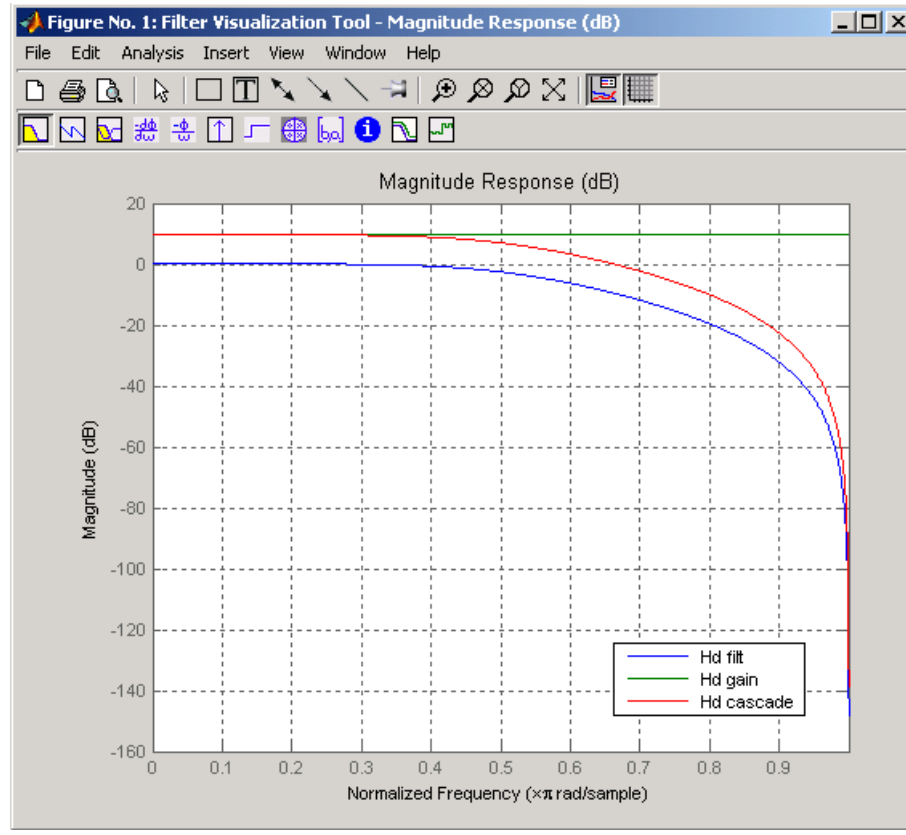
`Hd = dfilt.scalar(g)` returns a discrete-time, scalar filter, `Hd`, with gain `g`, where `g` is a scalar.

`Hd = dfilt.scalar` returns a default, discrete-time scalar gain filter, `Hd`, with gain `1`.

Example

Create a direct-form I filter and a scalar object with a gain of 3 and cascade them together.

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
Hd_filt = dfilt.df1(b,a)
Hd_gain = dfilt.scalar(3)
Hd=cascade(Hd_gain,Hd_filt)
fvtool(Hd_filt,Hd_gain,Hd)
Hd_filt =
         FilterStructure: 'Direct-Form I'
               Numerator: [0.3000 0.6000 0.3000]
             Denominator: [1 0 0.2000]
        PersistentMemory: false

Hd_gain =
         FilterStructure: 'Scalar'
                    Gain: 3
        PersistentMemory: false

Hd =
         FilterStructure: Cascade
                Stage(1): Scalar
                Stage(2): Direct-Form I
        PersistentMemory: false
```



To view the stages of the cascaded filter, use

```
Hd.stage(1)

ans =
          FilterStructure: 'Scalar'
                     Gain: 3
        PersistentMemory: false
```

and

```
Hd.stage(2)

ans =
          FilterStructure: 'Direct-Form I'
                Numerator: [0.3 0.6 0.3]
              Denominator: [1 0 0.2]
         PersistentMemory: false
```

See Also

dfilt, dfilt.cascade

Purpose

Discrete-time, state-space filter

Syntax

```
Hd = dfilt.statespace(A,B,C,D)
Hd = dfilt.statespace
```

Description

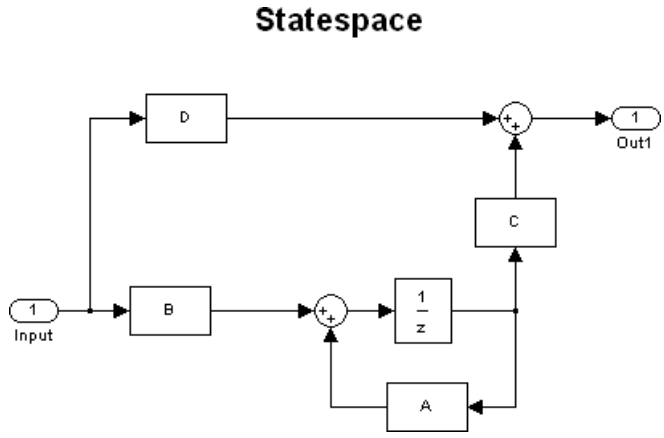
`Hd = dfilt.statespace(A,B,C,D)` returns a discrete-time state-space filter, `Hd`, with rectangular arrays `A`, `B`, `C`, and `D`.

`A`, `B`, `C`, and `D` are from the matrix or state-space form of a filter's difference equations

$$x(n+1) = Ax(n) + Bu(n)$$
$$y(n) = Cx(n) + Du(n)$$

where *x(n)* is the vector states at time *n*, *u(n)* is the input at time *n*, `y` is the output at time *n*, `A` is the state-transition matrix, `B` is the input-to-state transmission matrix, `C` is the state-to-output transmission matrix, and `D` is the input-to-ouput transmission matrix. For single-channel systems, `A` is an `m`-by-`m` matrix where `m` is the order of the filter, `B` is a column vector, `C` is a row vector, and `D` is a scalar.

`Hd = dfilt.statespace` returns a default, discrete-time state-space filter, `Hd`, with `A=[ ]`, `B=[ ]`, `C=[ ]`, and `D=1`. This filter passes the input through to the output unchanged.



The resulting filter states column vector has the same number of rows as the number of rows of `A` or `B`.

Examples

Create a second-order, state-space filter structure from a second-order, lowpass Butterworth design.

```
[A,B,C,D] = butter(2,0.5);
Hd = dfilt.statespace(A,B,C,D)
Hd =
             FilterStructure: 'State-space'
                           A: [2x2 double]
                           B: [0.8284;0.8284]
                           C: [0.2071 0.5]
                           D: 0.2929
                PersistentMemory: false
```

See Also

`dfilt`

Purpose

Discrete Fourier transform matrix

Syntax

```
A = dftmtx(n)
```

Description

A *discrete Fourier transform matrix* is a complex matrix of values around the unit circle, whose matrix product with a vector computes the discrete Fourier transform of the vector.

`A = dftmtx(n)` returns the `n`-by-`n` complex matrix `A` that, when multiplied into a length `n` column vector `x`.

```
y = A*x
```

computes the discrete Fourier transform of `x`.

The inverse discrete Fourier transform matrix is

```
Ai = conj(dftmtx(n))/n
```

Examples

In practice, the discrete Fourier transform is computed more efficiently and uses less memory with an FFT algorithm

```
x = 1:256;
y1 = fft(x);
```

than by using the Fourier transform matrix.

```
n = length(x);
y2 = x*dftmtx(n);
norm(y1-y2)
ans =
    1.8297e-009
```

Algorithm

`dftmtx` takes the FFT of the identity matrix to generate the transform matrix.

See Also

`convmtx`, `fft`

digitrevorder

Purpose

Permute input into digit-reversed order

Syntax

```
y = digitrevorder(x,r)
[y,i] = digitrevorder(x,r)
```

Description

`digitrevorder` is useful for pre-ordering a vector of filter coefficients for use in frequency-domain filtering algorithms, in which the `fft` and `ifft` transforms are computed without digit-reversed ordering for improved run-time efficiency.

`y = digitrevorder(x,r)` returns the input data in digit-reversed order in vector or matrix `y`. The digit-reversal is computed using the number system base (radix base) `r`, which can be any integer from 2 to 36. The length of `x` must be an integer power of `r`. If `x` is a matrix, the digit reversal occurs on the first dimension of `x` with size greater than 1. `y` is the same size as `x`.

`[y,i] = digitrevorder(x,r)` returns the digit-reversed vector or matrix `y` and the digit-reversed indices `i`, such that `y = x(i)`. Recall that MATLAB matrices use 1-based indexing, so the first index of `y` will be 1, not 0.

The following table shows the numbers 0 through 15, the corresponding digits and the digit-reversed numbers using radix base-4. The corresponding radix base-2 bits and bit-reversed indices are also shown.

Linear Index	Base-4 Digits	Digit-Reversed	Digit-Reversed Index	Base-2 Bits	Base-2 Reversed (bitrevorder)	Bit-Reversed Index
0	00	00	0	0000	0000	0
1	01	10	4	0001	1000	8
2	02	20	8	0010	0100	4
3	03	30	12	0011	1100	12
4	10	01	1	0100	0010	2
5	11	11	5	0101	1010	10

Linear Index	Base-4 Digits	Digit-Reversed	Digit-Reversed Index	Base-2 Bits	Base-2 Reversed (bitrevorder)	Bit-Reversed Index
6	12	21	9	0110	0110	6
7	13	31	13	0111	1110	14
8	20	02	2	1000	0001	1
9	21	12	6	1001	1001	9
10	22	22	10	1010	0101	5
11	23	32	14	1011	1101	13
12	30	03	3	1100	0011	3
13	31	13	7	1101	1011	11
14	32	23	11	1110	0111	7
15	33	33	15	1111	1111	15

Examples

Obtain the digit-reversed, radix base-3 ordered output of a vector containing 9 values:

```
x=[0:8]';                    % Create a column vector
[x,digitrevorder(x,3)]
ans =

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

See Also

`bitrevorder`, `fft`, `ifft`

diric

Purpose

Dirichlet or periodic sinc function

Syntax

```
y = diric(x,n)
```

Description

`y = diric(x,n)` returns a vector or array `y` the same size as `x`. The elements of `y` are the Dirichlet function of the elements of `x`. `n` must be a positive integer.

The Dirichlet function, or periodic sinc function, is

$$\text{diric}(x,n) = \begin{cases} -1^{\frac{x}{2\pi}(n-1)} & x = 0, \pm 2\pi, \pm 4\pi, \ldots \\ \dfrac{\sin(nx/2)}{n\sin(x/2)} & \text{else} \end{cases}$$

for any nonzero integer `n`. This function has period 2π for `n` odd and period 4π for `n` even. Its peak value is 1, and its minimum value is -1 for `n` even. The magnitude of this function is `(1/n)` times the magnitude of the discrete-time Fourier transform of the `n`-point rectangular window.

Diagnostics

If `n` is not a positive integer, `diric` gives the following error message:

```
Requires n to be a positive integer.
```

See Also

`cos`, `gauspuls`, `pulstran`, `rectpuls`, `sawtooth`, `sin`, `sinc`, `square`, `tripuls`

Purpose

Decrease sampling rate by integer factor

Syntax

```
y = downsample(x,n)
y = downsample(x,n,phase)
```

Description

`y = downsample(x,n)` decreases the sampling rate of `x` by keeping every n^{th} sample starting with the first sample. `x` can be a vector or a matrix. If `x` is a matrix, each column is considered a separate sequence.

`y = downsample(x,n,phase)` specifies the number of samples by which to offset the downsampled sequence. `phase` must be an integer from 0 to `n`-1.

Examples

Decrease the sampling rate of a sequence by 3:

```
x = [1 2 3 4 5 6 7 8 9 10];
y = downsample(x,3)
y =
    1    4    7    10
```

Decrease the sampling rate of the sequence by 3 and add a phase offset of 2:

```
y = downsample(x,3,2)
y =
    3    6    9
```

Decrease the sampling rate of a matrix by 3:

```
x = [1 2 3; 4 5 6; 7 8 9; 10 11 12];
y = downsample(x,3);
x,y
x =
    1    2    3
    4    5    6
    7    8    9
   10   11   12
y =
```

```
     1     2     3
    10    11    12
```

See Also

decimate, interp, interp1, resample, spline, upfirdn, upsample

Purpose

Discrete prolate spheroidal sequences (Slepian sequences)

Syntax

```
[e,v] = dpss(n,nw)
[e,v] = dpss(n,nw,k)
[e,v] = dpss(n,nw,[k1 k2])
[e,v] = dpss(n,nw,'int')
[e,v] = dpss(n,nw,'int',Ni)
[e,v] = dpss(...,'trace')
```

Description

`[e,v] = dpss(n,nw)` generates the first `2*nw` *discrete prolate spheroidal sequences* (DPSS) of length `n` in the columns of `e`, and their corresponding concentrations in vector `v`. They are also generated in the DPSS MAT-file database `dpss.mat`. `nw` must be less than `n/2`.

`[e,v] = dpss(n,nw,k)` returns the `k` most band-limited discrete prolate spheroidal sequences. `k` must be an integer such that $1 \le k \le n$.

`[e,v] = dpss(n,nw,[k1 k2])` returns the `k1`st through the `k2`nd discrete prolate spheroidal sequences, where $1 \le k1 \le k2 \le n$.

For all of the above forms:

- The Slepian sequences are calculated directly.
- The sequences are generated in the frequency band $|\omega| \le (2\pi W)$, where `W = nw/n` is the half-bandwidth and ω is in rad/sample.
- `e(:,1)` is the length `n` signal most concentrated in the frequency band $|\omega| \le (2\pi W)$ radians, `e(:,2)` is the signal orthogonal to `e(:,1)` that is most concentrated in this band, `e(:,3)` is the signal orthogonal to both `e(:,1)` and `e(:,2)` that is most concentrated in this band, etc.
- For multitaper spectral analysis, typical choices for `nw` are 2, 5/2, 3, 7/2, or 4.

`[e,v] = dpss(n,nw,'int')` uses the interpolation method specified by the string `'int'` to compute `e` and `v` from the sequences in `dpss.mat` with length closest to `n`. The string `'int'` can be either:

- `'spline'`: Use spline interpolation.

- `'linear'`: Use linear interpolation. This is much faster but less accurate than spline interpolation.

`[e,v] = dpss(n,nw,'int',Ni)` interpolates from existing length `Ni` sequences. The interpolation method `'linear'` requires `Ni > n`.

`[e,v] = dpss(...,'trace')` uses the trailing string `'trace'` to display which interpolation method DPSS uses. If you don't specify the interpolation method, the display indicates that you are using the *direct method*.

Examples

Example 1: Using dpss, dpssave, and dpssdir

Create a catalogue of 16 DPSS functions with `nw = 4`, and use spline interpolation on 10 of these functions while displaying the interpolation method you use. You can do this using `dpss`, `dpsssave`, and `dpssdir`:

```
% Create the catalogue of functions.
[e,v] = dpss(16,4);
% Save e and v in a MAT-file.
dpsssave(4,e,v);
% Find nw = 4. First create a structure called index.
index = dpssdir;
index.wlists
ans =
     NW: 4
    key: 1
% Use spline interpolation on 10 of the DPSS functions.
[e1,v1] = dpss(10,4,'spline',size(e,1),'trace');
```

Example 2: Using dpss and dpssload

Create a set of DPSS functions using `dpss`, and use the spline method on a subset of these functions. Use `dpssload` to load the MAT-file created by `dpss`:

```
% Create the catalogue of functions.
[e,v] = dpss(16,4);
```

```
% Load dpss.mat, where e and v are saved.
[e1,v1] = dpssload(16,4);
% Use spline interpolation on 10 of the DPSS functions.
[e1,v1] = dpss(10,4,'spline');
```

References

[1] Percival, D.B., and A.T. Walden. *Spectral Analysis for Physical Applications: Multitaper and Conventional Univariate Techniques,* Cambridge: Cambridge University Press, 1993.

See Also

`dpssclear`, `dpssdir`, `dpssload`, `dpsssave`, `pmtm`

dpssclear

Purpose

Remove discrete prolate spheroidal sequences from database

Syntax

```
dpssclear(n,nw)
```

Description

`dpssclear(n,nw)` removes sequences with length `n` and time-bandwidth product `nw` from the DPSS MAT-file database `dpss.mat`.

See Also

`dpss`, `dpssdir`, `dpssload`, `dpsssave`

Purpose

Discrete prolate spheroidal sequences database directory

Syntax

```
dpssdir
dpssdir(n)
dpssdir(nw,'nw')
dpssdir(n,nw)
index = dpssdir
```

Description

`dpssdir` manages the database directory that contains the generated DPSS samples in the DPSS MAT-file database `dpss.mat`.

`dpssdir` lists the directory of saved sequences in `dpss.mat`.

`dpssdir(n)` lists the sequences saved with length `n`.

`dpssdir(nw,'nw')` lists the sequences saved with time-bandwidth product `nw`.

`dpssdir(n,nw)` lists the sequences saved with length `n` and time-bandwidth product `nw`.

`index = dpssdir` is a structure array describing the DPSS database. Pass `n` and `nw` options as for the no output case to get a filtered `index`.

Examples

See “Example 1: Using dpss, dpssave, and dpssdir” on page 10-220.

See Also

`dpss`, `dpssclear`, `dpssload`, `dpsssave`

dpssload

Purpose

Load discrete prolate spheroidal sequences from database

Syntax

```
[e,v] = dpssload(n,nw)
```

Description

`[e,v] = dpssload(n,nw)` loads all sequences with length `n` and time-bandwidth product `nw` in the columns of `e` and their corresponding concentrations in vector `v` from the DPSS MAT-file database `dpss.mat`.

Examples

See “Example 2: Using dpss and dpssload” on page 10-220.

See Also

`dpss`, `dpssclear`, `dpssdir`, `dpsssave`

Purpose

Save discrete prolate spheroidal sequences in database

Syntax

```
dpsssave(nw,e,v)
status = dpsssave(nw,e,v)
```

Description

`dpsssave(nw,e,v)` saves the sequences in the columns of `e` and their corresponding concentrations in vector `v` in the DPSS MAT-file database `dpss.mat`. It is not necessary to specify sequence length, because the length of the sequence is determined by the number of rows of `e`.

`nw` is the *time-bandwidth product* that was specified when the sequence was created using `dpss`.

`status = dpsssave(nw,e,v)` returns `0` if the save was successful and `1` if there was an error.

Examples

See "Example 1: Using dpss, dpssave, and dpssdir" on page 10-220.

See Also

`dpss`, `dpssclear`, `dpssdir`, `dpssload`

dspdata

Purpose

DSP data parameter information

Syntax

```
Hs = dspdata.dataobj(input1,...)
```

Description

`Hs = dspdata.dataobj(input1,...)` returns a `dspdata` object `Hs` of type *`dataobj`*. This object contains all the parameter information needed for the specified type of *`dataobj`*. Each *`dataobj`* takes one or more inputs, which are described on the individual reference pages. If you do not specify any input values, the returned object has default property values appropriate for the particular *`dataobj`* type.

Note You must use a *`dataobj`* with `dspdata`.

Data Objects

A data object (*`dataobj`*) for `dspdata` specifies the type of data stored in the object. Available *`dataobj`* types for `dspdata` are shown below.

dspdata.dataobj	Description
`dspdata.msspectrum`	Mean-square spectrum data (power)
`dspdata.psd`	Power spectral density data (power/frequency)
`dspdata.pseudospectrum`	Pseudospectrum data (power)

For more information on each *`dataobj`* type, use the syntax `help dspdata.dataobj` at the MATLAB prompt or refer to its reference page.

Methods

Methods provide ways of performing functions directly on your `dspdata` object. You can apply these methods directly on the variable you assigned to your `dspdata` object.

Method	Description
`avgpower`	Note that this method applies only to `dspdata.psd` objects. `avgpower(Hs)` computes the average power in a given frequency band. The technique uses a rectangle approximation of the integral of the `Hs` signal's power spectral density (PSD). If the signal is a matrix, the computation is done on each column. The average power is the total signal power and the `SpectrumType` property determines whether the total average power is contained in the one-sided or two-sided spectrum. For aa one-sided spectrum, the range is [0,pi] for even number of frequency points and [0,pi) for odd. For a two-sided spectrum the range is [0,2pi). `avgpower(Hs,freqrange)` specifies the frequency range over which to calculate the average power. `freqrange` is a two-element vector of the frequencies between which to calculate. If a frequency value does not match exactly the frequency in `Hs`, the next closest value is used. Note that the first frequency value in `freqrange` is included in the calculation and the second value is excluded.
`centerdc`	`centerdc(Hs)` or `centerdc(Hs,true)` shifts the data and frequency values so that the DC component is at the center of the spectrum. If the `SpectrumType` property is `'onesided'`, it is changed to `'twosided'` and then the DC component is centered. `centerdc(Hs,'false')` shifts the data and frequency values so that the DC component is at the left edge of the spectrum.

Method	Description
`findpeaks`	`findpeaks(Hs)` finds local maxima or peaks. If no peaks are found, `findpeaks` returns an empty vector. `[pks,frqs] = findpeaks(x)` returns peaks values (`pks`) and the frequencies (`frqs`) at which the peaks occur. `findpeaks(x,'minpeakheight',mph)` returns only peaks greater than the minimum peak height `mph`, where `mph` is a real scalar. Default is `-Inf`. `findpeaks(x,'minpeakdistance',mpd)` returns only peaks separated by the minimum frequency units distance `mpd`, which is a positive integer. Setting the minimum peak distance ignores smaller peaks that may occur close to larger local peaks. Default is 1. `findpeaks(x,'threshold',th)` returns only peaks greater than their neighbors by at least the threshold `th`, which is a real, scalar value greater than or equal to 0. Default is 0. `findpeaks(x,'npeaks',np)` returns a maximum of `np` number of peaks. When `np` peaks are found, the search stops. Default is to return all peaks. `findpeaks(x,'sortstr',str)` specifies the sorting order, where `str` is `'ascend'`, `'descend'` or `'none'`. For `'ascend'` , the peaks are returned in order from smallest to largest, and vice versa for `'descend'`. For `'none'`, the peaks are returned in the order in which they occur.

Method	Description
`halfrange`	`halfrange(Hs)` converts the `Hs` spectrum to a spectrum calculated over half the Nyquist interval. All associated properties affected by the new frequency range are adjusted automatically. This method is used for `dspdata.pseudospectrum` objects. Note that the spectrum is assumed to be from a real signal (that is, `halfrange` uses half the data points regardless of whether the data is symmetric).
`normalizefreq`	`normalizefreq(Hs)` or `normalizefreq(Hs,true)` normalizes the frequency specifications in the `Hs` object to `Fs` so the frequencies are between 0 and 1. It also sets the `NormalizedFrequency` property to `true`. `normalizefreq(Hs,false)` converts the frequencies to linear frequencies. `normalizefreq(Hs,false,Fs)` sets a new sampling frequency `Fs`. This can be used only with `false`.
`onesided`	`onesided(Hs)` converts the `Hs` spectrum to a spectrum calculated over half the Nyquist interval and containing the total signal power. All associated properties affected by the new frequency range are adjusted automatically. This method is used for `dspdata.psd` and `dspdata.msspectrum` objects. Note that the spectrum is assumed to be from a real signal (that is, `onesided` uses half the data points regardless of whether the data is symmetric).

Method	Description
`plot`	Displays the data graphically in the current figure window. For a `dspdata.psd` object, it displays the power spectral density in dB/Hz. For a `dspdata.msspectrum` object, it displays the mean–square in dB. For a `dspdata.pseudospectrum` object, it displays the pseudospectrum in dB.
`sfdr`	This method applies only to `dspdata.msspectrum` objects. `sfdr(Hs)` computes the spurious-free dynamic range (SFDR) in dB of a mean square spectrum object `Hs`. SFDR is the usable range before spurious noise interferes with the signal. `[sfd,spur,frq] = sfdr(Hs)` returns the magnitude of the highest `spur` and the frequency `frq` at which it occurs. `sfdr(Hs,'minspurlevel',msl)` ignores spurs below the minimum spur level `msl`, which is a real scalar in dB. `sfdr(Hs,'minspurdistance',msd)` includes spurs only if they are separated by at least the minimum spur distance `msd`, which is a real, positive scalar in frequency units.

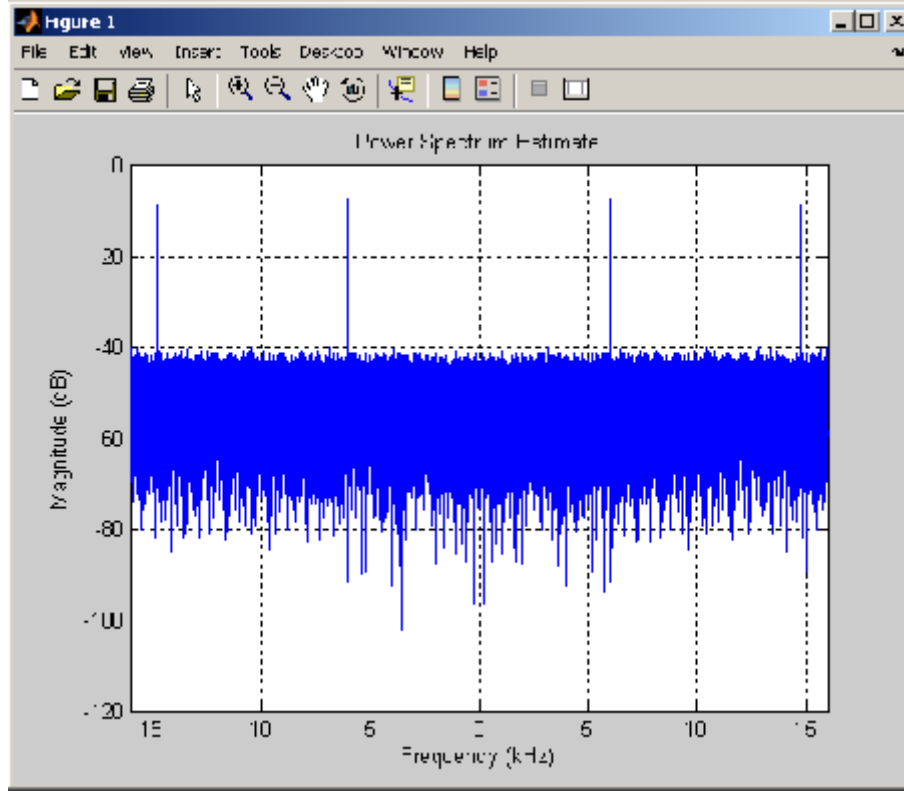
Method	Description
`twosided`	`twosided(Hs)` converts the `Hs` spectrum to a spectrum calculated over the whole Nyquist interval. All associated properties affected by the new frequency range are adjusted automatically. This method is used for `dspdata.psd` and `dspdata.msspectrum` objects. Note that if your data is nonuniformly sampled, converting from `onesided` to `twosided` may produce incorrect results.
`wholerange`	`wholerange(Hs)` converts the `Hs` spectrum to a spectrum calculated over the whole Nyquist interval. All associated properties affected by the new frequency range are adjusted automatically. This method is used for `dspdata.pseudospectrum` objects. Note that if your data is nonuniformly sampled, converting from `half` to `wholerange` may produce incorrect results.

For more information on each method, use the syntax `help dspdata/`*`method`* at the MATLAB prompt.

Plotting a dspdata Object

The plot method displays the `dspdata` object spectrum in a separate figure window.

```
plot(Hs)              % Plots an existing Hs object
```



Modifying a dspdata Object

After you create a `dspdata` object, you can use any of the methods in the table above to modify the object properties.

For example, to change the object from two-sided to one-sided, use

```
onesided(Hs)
```

The `Hs` object is modified.

Examples

See the `msspectrum`, `psd`, or `pseudospectrum` reference pages for specific examples.

See Also

`dspdata.msspectrum`, `dspdata.psd`, `dspdata.pseudospectrum`

dspdata.msspectrum

Purpose

Mean-square (power) spectrum

Syntax

```
Hmss = dspdata.msspectrum(Data)
Hmss = dspdata.msspectrum(Data,Frequencies)
Hmss = dspdata.msspectrum(...,'Fs',Fs)
Hmss = dspdata.msspectrum(...,'SpectrumType',SpectrumType)
Hmss = dspdata.msspectrum(...,'CenterDC',flag)
```

Description

The mean-squared spectrum (MSS) is intended for discrete spectra. Unlike the power spectral density (PSD), the peaks in the MSS reflect the power in the signal at a given frequency. The MSS of a signal is the Fourier transform of that signal's autocorrelation.

`Hmss = dspdata.msspectrum(Data)` uses the mean-square (power) spectrum data contained in `Data`, which can be in the form of a vector or a matrix, where each column is a separate set of data. Default values for other properties of the object are as follows:

Property	Default Value	Description
Name	`'Mean-square Spectrum'`	Read-only string
`Frequencies`	`[]` type `double`	Vector of frequencies at which the spectrum is evaluated. The range of this vector depends on the `SpectrumType` value. For a one-sided spectrum, the default range is [0, pi) or [0, `Fs`/2) for odd length, and [0, pi] or [0, `Fs`/2] for even length, if `Fs` is specified. For a two-sided spectrum, it is [0, 2pi) or [0, `Fs`). The length of the `Frequencies` vector must match the length of the columns of `Data`. If you do not specify `Frequencies`, a default vector is created. If one-sided is selected, then the whole number of FFT points (nFFT) for this vector is assumed to be even. If `onesided` is selected and you specify `Frequencies`, the last frequency point is compared to the next-to-last point and to pi (or `Fs`/2, if `Fs` is specified). If the last point is closer to pi (or `Fs`/2) than it is to the previous point, nFFT is assumed to be even. If it is closer to the previous point, nFFT is assumed to be odd.
`Fs`	`'Normalized'`	Sampling frequency, which is `'Normalized'` if `NormalizedFrequency` is `true`. If `NormalizedFrequency` is `false` `Fs` defaults to `1` Hz.

Property	Default Value	Description
SpectrumType	'Onesided'	Nyquist interval over which the spectral density is calculated. Valid values are 'Onesided' and 'Twosided'. See the onesided and twosided methods in dspdata for information on changing this property. The interval for Onesided is [0 pi) or [0 pi] depending on the number of FFT points, and for Twosided the interval is [0 2pi).
NormalizedFrequency	true	Whether the frequency is normalized (true) or not (false). This property is set automatically at construction time based on Fs. If Fs is specified, NormalizedFrequency is set to false. See the normalizefreq method in dspdata for information on changing this property.

`Hmss = dspdata.msspectrum(Data,Frequencies)` uses the mean–square spectrum data contained in `Data` and `Frequencies` vectors.

`Hmss = dspdata.msspectrum(...,'Fs',Fs)` uses the sampling frequency `Fs`. Specifying `Fs` uses a default set of linear frequencies (in `Hz`) based on `Fs` and sets `NormalizedFrequency` to `false`.

`Hmss = dspdata.msspectrum(...,'SpectrumType',SpectrumType)` uses the `SpectrumType` string to specify the interval over which the mean–square spectrum was calculated. For data that ranges from [0 pi) or [0 pi], set the `SpectrumType` to `onesided`; for data that ranges from [0 2pi), set the the `SpectrumType` to `twosided`.

`Hmss = dspdata.msspectrum(...,'CenterDC',flag)` uses the value of `flag` to indicate whether the zero-frequency (DC) component is centered. If `flag` is `true`, it indicates that the DC component is in

the center of the two-sided spectrum. Set the `flag` to `false` if the DC component is on the left edge of the spectrum.

Methods

Methods provide ways of performing functions directly on your `dspdata` object without having to specify the parameters again. You can apply a method directly on the variable you assigned to your `dspdata.msspectrum` object. You can use the following methods with a `dspdata.msspectrum` object.

- `centerdc`
- `normalizefreq`
- `onesided`
- `plot`
- `sfdr`
- `twosided`

For example, to normalize the frequency and set the `NormalizedFrequency` parameter to true, use

```
Hmss = normalizefreq(Hs)
```

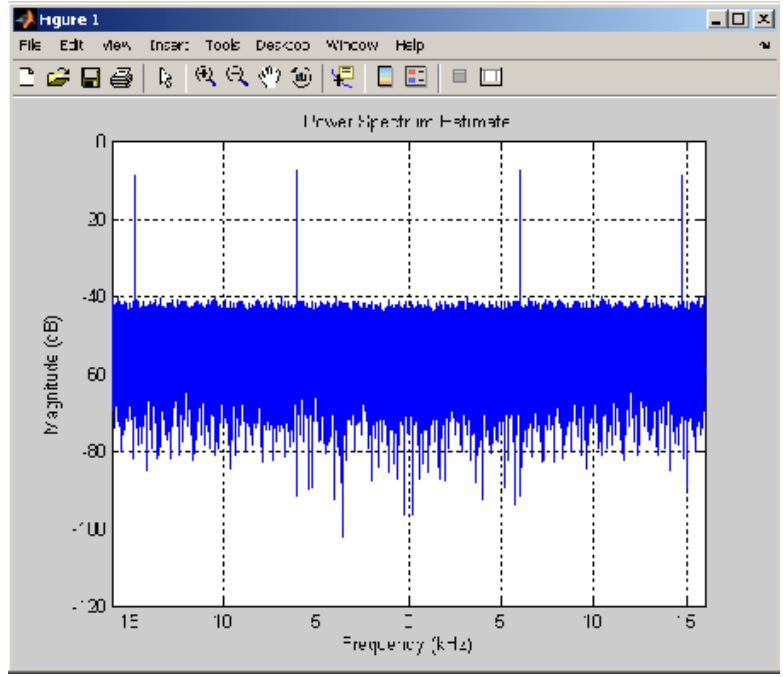
For detailed information on using the methods and plotting the spectrum, see the `dspdata` reference page.

Examples

This example shows how to view the spectral content of two sinusoids with random noise.

```
Fs = 32e3;
t = 0:1/Fs:2.96;
x = cos(2*pi*t*1.24e3)+cos(2*pi*t*10e3)+randn(size(t));
X = fft(x);
P = (abs(X)/length(x)).^2;     % Compute the mean-square.

% Create data object.
```

```
Hmss = dspdata.msspectrum(P,'Fs',Fs,'centerdc',true);
plot(Hmss);          % Plot the mean-square spectrum.
```



See Also

`dspdata.psd`, `dspdata.pseudospectrum`, `spectrum`

Purpose

Power spectral density

Syntax

```
Hpsd = dspdata.psd(Data)
Hpsd = dspdata.psd(Data,Frequencies)
Hpsd = dspdata.psd(...,'Fs',Fs)
Hpsd = dspdata.psd(...,'SpectrumType',SpectrumType)
Hpsd = dspdata.psd(...,'CenterDC',flag)
```

Description

The power spectral density (PSD) is intended for continuous spectra. The integral of the PSD over a given frequency band computes the average power in the signal over that frequency band. In contrast to the mean-squared spectrum, the peaks in this spectra do not reflect the power at a given frequency. See the `avgpower` method of `dspdata` for more information.

A one-sided PSD contains the total power of the signal in the frequency interval from DC to half of the Nyquist rate. A two-sided PSD contains the total power in the frequency interval from DC to the Nyquist rate.

`Hpsd = dspdata.psd(Data)` uses the power spectral density data contained in `Data`, which can be in the form of a vector or a matrix, where each column is a separate set of data. Default values for other properties of the object are shown below:

Property	Default Value	Description
Name	`'Power Spectral Density'`	Read-only string

Property	Default Value	Description
`Frequencies`	`[]` type `double`	Vector of frequencies at which the power spectral density is evaluated. The range of this vector depends on the `SpectrumType` value. For one-sided, the default range is [0, pi) or [0, `Fs`/2) for odd length, and [0, pi] or [0, `Fs`/2] for even length, if `Fs` is specified. For two-sided, it is [0, 2pi) or [0, `Fs`). If you do not specify `Frequencies`, a default vector is created. If one-sided is selected, then the whole number of FFT points (nFFT) for this vector is assumed to be even. If `onesided` is selected and you specify `Frequencies`, the last frequency point is compared to the next-to-last point and to pi (or `Fs`/2, if `Fs` is specified). If the last point is closer to pi (or `Fs`/2) than it is to the previous point, nFFT is assumed to be even. If it is closer to the previous point, nFFT is assumed to be odd. The length of the `Frequencies` vector must match the length of the columns of `Data`.
`Fs`	`'Normalized'`	Sampling frequency, which is `'Normalized'` if `NormalizedFrequency` is `true`. If `NormalizedFrequency` is `false` `Fs` defaults to `1`.

Property	Default Value	Description
`SpectrumType`	`'Onesided'`	Nyquist interval over which the power spectral density is calculated. Valid values are `'Onesided'` and `'Twosided'`. A one-sided PSD contains the total signal power in half the Nyquist interval. See the `onesided` and `twosided` methods in `dspdata` for information on changing this property. The range for half the Nyquist interval is [0 pi) or [0 pi] depending on the number of FFT points. For the whole Nyquist interval, the range is [0 2pi).
`NormalizedFrequency`	`true`	Whether the frequency is normalized (`true`) or not (`false`). This property is set automatically at construction time based on `Fs`. If `Fs` is specified, `NormalizedFrequency` is set to `false`. See the `normalizefreq` method in `dspdata` for information on changing this property.

`Hpsd = dspdata.psd(Data,Frequencies)` uses the power spectral density estimation data contained in `Data` and `Frequencies` vectors.

`Hpsd = dspdata.psd(...,'Fs',Fs)` uses the sampling frequency `Fs`. Specifying `Fs` uses a default set of linear frequencies (in `Hz`) based on `Fs` and sets `NormalizedFrequency` to `false`.

`Hpsd = dspdata.psd(...,'SpectrumType',SpectrumType)` uses the `SpectrumType` string to specify the interval over which the power spectral density was calculated. For data that ranges from [0 pi) or [0 pi], set the `SpectrumType` to `onesided`; for data that ranges from [0 2pi), set the `SpectrumType` to `twosided`.

`Hpsd = dspdata.psd(...,'CenterDC',flag)` uses the value of `flag` to indicate whether the zero-frequency (DC) component is centered. If

`flag` is `true`, it indicates that the DC component is in the center of the two-sided spectrum. Set the `flag` to `false` if the DC component is on the left edge of the spectrum.

Methods

Methods provide ways of performing functions directly on your `dspdata` object. You can apply a method directly on the variable you assigned to your `dspdata.psd` object. You can use the following methods with a `dspdata.psd` object.

- `avgpower`
- `centerdc`
- `normalizefreq`
- `onesided`
- `plot`
- `twosided`

For example, to normalize the frequency and set the `NormalizedFrequency` parameter to true, use

```
Hpsd = normalizefreq(Hpsd)
```

For detailed information on using the methods and plotting the spectrum, see the `dspdata` reference page.

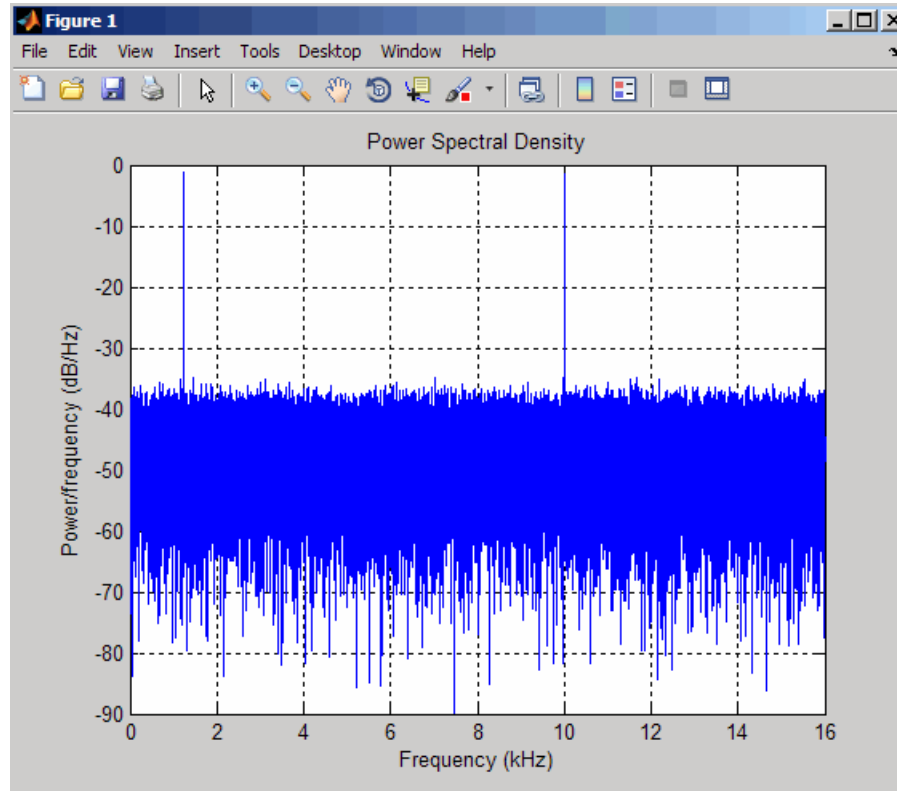
Examples

Resolving Signal Components

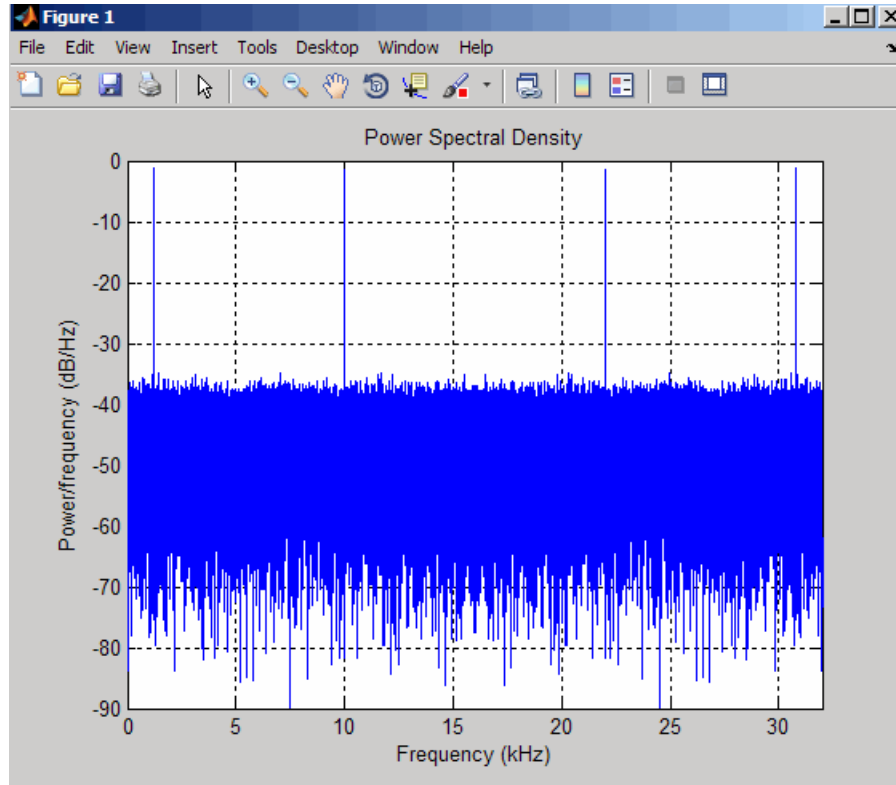
Estimate the power spectral density of a noisy sinusoidal signal with two frequency components and then store the results in a PSD data object and plot it.

```
Fs = 32e3;
t = 0:1/Fs:2.96;
x = cos(2*pi*t*1.24e3)+ cos(2*pi*t*10e3)+ randn(size(t));
nfft = 2^nextpow2(length(x));
```

```
Pxx = abs(fft(x,nfft)).^2/length(x)/Fs;

% Create a single-sided spectrum
Hpsd = dspdata.psd(Pxx(1:length(Pxx)/2),'Fs',Fs);
plot(Hpsd);
```




```
% Create a double-sided spectrum
Hpsd = dspdata.psd(Pxx,'Fs',Fs,'SpectrumType','twosided');
plot(Hpsd)
```



See Also

`dspdata.msspectrum`, `dspdata.pseudospectrum`, `spectrum`

Purpose

Pseudospectrum `dspdata` object

Syntax

```
Hps = dspdata.pseudospectrum(Data)
Hps = dspdata.pseudospectrum(Data,Frequencies)
Hps = dspdata.pseudospectrum(...,'Fs',Fs)
Hps = dspdata.pseudospectrum...,'SpectrumRange',SpectrumRange)
Hps = dspdata.pseudospectrum(...,'CenterDC',flag)
```

Description

A pseudospectrum is an indicator of the presence of sinusoidal components in a signal.

`Hps = dspdata.pseudospectrum(Data)` uses the pseudospsectrum data contained in `Data`, which can be in the form of a vector or a matrix, where each column is a separate set of data. Default values for other properties of the object are:

Property	Default Value	Description
Name	`'Pseudospectrum'`	Read-only string
`Frequencies`	`[]` type `double`	Vector of frequencies at which the power spectral density is evaluated. The range of this vector depends on the `SpectrumRange` value. For half, the default range is [0, pi) or [0, `Fs`/2) for odd length, and [0, pi] or [0, `Fs`/2] for even length, if `Fs` is specified. For whole, it is [0, 2pi) or [0, `Fs`). If you do not specify `Frequencies`, a default vector is created. If half the Nyquist range is selected, then the whole number of FFT points (nFFT) for this vector is assumed to be even. If `half` the Nyquist range is selected and you specify `Frequencies`, the last frequency point is compared to the next-to-last point and to pi (or `Fs`/2, if `Fs` is specified). If the last point is closer to pi (or `Fs`/2) than it is to the previous point, nFFT is assumed to be even. If it is closer to the previous point, nFFT is assumed to be odd. The length of the `Frequencies` vector must match the length of the columns of `Data`.
`Fs`	`'Normalized'`	Sampling frequency, which is `'Normalized'` if `NormalizedFrequency` is `true`. If `NormalizedFrequency` is `false` `Fs` defaults to `1`.

Property	Default Value	Description
SpectrumRange	'Half'	Nyquist interval over which the pseudospectrum is calculated. Valid values are 'Half' and 'Whole'. See the half and whole methods in dspdata for information on changing this property. The interval for Half is [0 pi) or [0 pi] depending on the number of FFT points, and for Whole the interval is [0 2pi).
NormalizedFrequency	true	Whether the frequency is normalized (true) or not (false). This property is set automatically at construction time based on Fs. If Fs is specified, NormalizedFrequency is set to false. See the normalizefreq method in dspdata for information on changing this property.

Hps = dspdata.pseudospectrum(Data,Frequencies) uses the pseudospectrum estimation data contained in the Data and Frequencies vectors.

Hps = dspdata.pseudospectrum(...,'Fs',Fs) uses the sampling frequency Fs. Specifying Fs uses a default set of linear frequencies (in Hz) based on Fs and sets NormalizedFrequency to false.

Hps = dspdata.pseudospectrum...,'SpectrumRange',SpectrumRange) uses the SpectrumRange string to specify the interval over which the pseudospectrum was calculated. For data that ranges from [0 pi) or [0 pi], set the SpectrumRange to half; for data that ranges from [0 2pi), set the SpectrumRange to whole.

Hps = dspdata.pseudospectrum(...,'CenterDC',flag) uses the value of flag to indicate whether the zero-frequency (DC) component is centered. If flag is true, it indicates that the DC component is in the center of the whole Nyquist range spectrum. Set the flag to false if the DC component is on the left edge of the spectrum.

Methods

Methods provide ways of performing functions directly on your `dspdata` object. You can apply a method directly on the variable you assigned to your `dspdata.pseudospectrum` object. You can use the following methods with a `dspdata.pseudospectrum` object.

- `centerdc`
- `halfrange`
- `normalizefreq`
- `plot`
- `wholerange`

For example, to normalize the frequency and set the `NormalizedFrequency` parameter to true, use

```
Hps = normalizefreq(Hps)
```

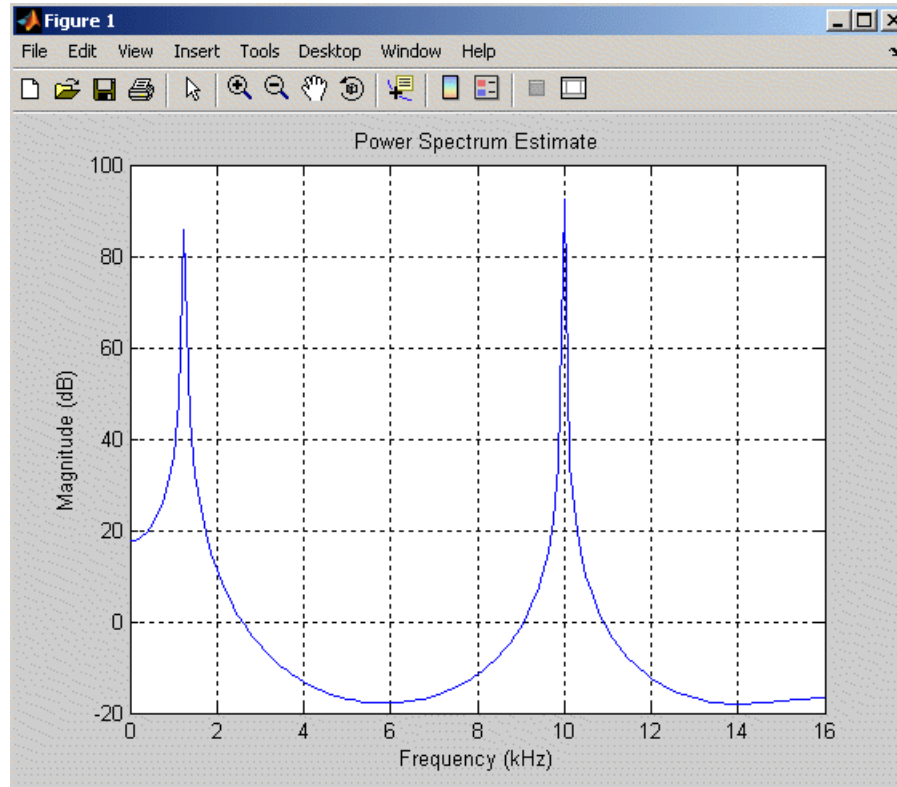
For detailed information on using the methods and plotting the pseudospectrum, see the `dspdata` reference page.

Examples

Storing and Plotting Pseudospectrum Data

Use eigenanalysis to estimate the pseudospectrum of a noisy sinusoidal signal with two frequency components. Then store the results in a `pseudospectrum` data object and plot it.

```
Fs = 32e3;
t  = 0:1/Fs:2.96;
x = cos(2*pi*t*1.24e3) + cos(2*pi*t*10e3) + randn(size(t));
P = pmusic(x,4);
% Create data object
hps = dspdata.pseudospectrum(P,'Fs',Fs);
% Plot the pseudospectrum
plot(hps);
```



See Also

`dspdata.msspectrum`, `dspdata.psd`, `spectrum`

dspfwiz

Purpose

Open FDATool Realize Model panel to create Simulink filter block

Syntax

```
dspfwiz
```

Description

Note You must have the Simulink product installed to use this function.

`dspfwiz` opens FDATool with the Realize Model panel displayed. See "Exporting to a Simulink Model" on page 5-38 for information on using this panel.

Use other panels in FDATool to design your filter and then use the Realize Model panel to create your filter as a subsystem block, which is a combination of Sum, Gain, and Integer Delay blocks, in a Simulink model.

If you also have Signal Processing Blockset product installed, you can create a Digital Filter block instead of a subsystem block, by deselecting the **Build model using basic elements** check box. For more information on the differences between these types of blocks, see "Choosing Between Filter Design Blocks" in the Signal Processing Blockset documentation.

See Also

`fdatool`, `realizemdl` on the `dfilt` reference page

Purpose

Elliptic filter design

Syntax

```
[z,p,k] = ellip(n,Rp,Rs,Wp)
[z,p,k] = ellip(n,Rp,Rs,Wp,'ftype')
[b,a] = ellip(n,Rp,Rs,Wp)
[b,a] = ellip(n,Rp,Rs,Wp,'ftype')
[A,B,C,D] = ellip(n,Rp,Rs,Wp)
[A,B,C,D] = ellip(n,Rp,Rs,Wp,'ftype')
[z,p,k] = ellip(n,Rp,Rs,Wp,'s')
[z,p,k] = ellip(n,Rp,Rs,Wp,'ftype','s')
[b,a] = ellip(n,Rp,Rs,Wp,'s')
[b,a] = ellip(n,Rp,Rs,Wp,'ftype','s')
[A,B,C,D] = ellip(n,Rp,Rs,Wp,'s')
[A,B,C,D] = ellip(n,Rp,Rs,Wp,'ftype','s')
```

Description

`ellip` designs lowpass, bandpass, highpass, and bandstop digital and analog elliptic filters. Elliptic filters offer steeper rolloff characteristics than Butterworth or Chebyshev filters, but are equiripple in both the pass- and stopbands. In general, elliptic filters meet given performance specifications with the lowest order of any filter type.

Digital Domain

`[z,p,k] = ellip(n,Rp,Rs,Wp)` designs an order `n` lowpass digital elliptic filter with normalized passband edge frequency `Wp`, `Rp` dB of ripple in the passband, and a stopband `Rs` dB down from the peak value in the passband. It returns the zeros and poles in length `n` column vectors `z` and `p` and the gain in the scalar `k`.

The *normalized passband edge frequency* is the edge of the passband, at which the magnitude response of the filter is `-Rp` dB. For `ellip`, the normalized cutoff frequency `Wp` is a number between 0 and 1, where 1 corresponds to half the sampling frequency (Nyquist frequency). Smaller values of passband ripple `Rp` and larger values of stopband attenuation `Rs` both lead to wider transition widths (shallower rolloff characteristics).

If `Wp` is a two-element vector, `Wp = [w1 w2]`, `ellip` returns an order `2*n` bandpass filter with passband $w1 < \omega < w2$.

`[z,p,k] = ellip(n,Rp,Rs,Wp,'ftype')` designs a highpass, lowpass, or bandstop filter, where the string *'ftype'* is one of the following:

- `'high'` for a highpass digital filter with normalized passband edge frequency `Wp`
- `'low'` for a lowpass digital filter with normalized passband edge frequency `Wp`
- `'stop'` for an order `2*n` bandstop digital filter if `Wp` is a two-element vector, `Wp = [w1 w2]`. The stopband is `w1` < ω < `w2`.

With different numbers of output arguments, `ellip` directly obtains other realizations of the filter. To obtain the transfer function form, use two output arguments as shown below.

Note See "Limitations" on page 10-255 for information about numerical issues that affect forming the transfer function.

`[b,a] = ellip(n,Rp,Rs,Wp)` designs an order `n` lowpass digital elliptic filter with normalized passband edge frequency `Wp`, `Rp` dB of ripple in the passband, and a stopband `Rs` dB down from the peak value in the passband. It returns the filter coefficients in the length `n+1` row vectors `b` and `a`, with coefficients in descending powers of *z*.

$$H(z) = \frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \cdots + b(n+1)z^{-n}}{1 + a(2)z^{-1} + \cdots + a(n+1)z^{-n}}$$

`[b,a] = ellip(n,Rp,Rs,Wp,'ftype')` designs a highpass, lowpass, or bandstop filter, where the string *'ftype'* is `'high'`, `'low'`, or `'stop'`, as described above.

To obtain state-space form, use four output arguments as shown below:

`[A,B,C,D] = ellip(n,Rp,Rs,Wp)` or

`[A,B,C,D] = ellip(n,Rp,Rs,Wp,'ftype')` where `A`, `B`, `C`, and `D` are

$$x[n+1] = Ax[n] + Bu[n]$$
$$y[n] = Cx[n] + Du[n]$$

and u is the input, x is the state vector, and y is the output.

Analog Domain

`[z,p,k] = ellip(n,Rp,Rs,Wp,'s')` designs an order `n` lowpass analog elliptic filter with angular passband edge frequency `Wp` rad/s and returns the zeros and poles in length `n` or `2*n` column vectors `z` and `p` and the gain in the scalar `k`.

The *angular passband edge frequency* is the edge of the passband, at which the magnitude response of the filter is `-Rp` dB. For `ellip`, the angular passband edge frequency `Wp` must be greater than 0 rad/s.

If `Wp` is a two-element vector with `w1 < w2`, then ellip(n,Rp,`Rs,Wp,'s'`) returns an order `2*n` bandpass analog filter with passband `w1` < ω< `w2`.

`[z,p,k] = ellip(n,Rp,Rs,Wp,'ftype','s')` designs a highpass, lowpass, or bandstop filter, where the string *`'ftype'`* is `'high'`, `'low'`, or `'stop'`, as described above.

With different numbers of output arguments, ellip directly obtains other realizations of the analog filter. To obtain the transfer function form, use two output arguments as shown below:

`[b,a] = ellip(n,Rp,Rs,Wp,'s')` designs an order `n` lowpass analog elliptic filter with angular passband edge frequency `Wp` rad/s and returns the filter coefficients in the length `n+1` row vectors `b` and `a`, in descending powers of s, derived from this transfer function:

$$H(s) = \frac{B(s)}{A(s)} = \frac{b(1)s^n + b(2)s^{n-1} + \cdots + b(n+1)}{s^n + a(2)s^{n-1} + \cdots + a(n+1)}$$

`[b,a] = ellip(n,Rp,Rs,Wp,'ftype','s')` designs a highpass, lowpass, or bandstop filter, where the string *`'ftype'`* is `'high'`, `'low'`, or `'stop'`, as described above.

To obtain state-space form, use four output arguments as shown below:

`[A,B,C,D] = ellip(n,Rp,Rs,Wp,'s')` or

`[A,B,C,D] = ellip(n,Rp,Rs,Wp,'ftype','s')` where `A`, `B`, `C`, and `D` are

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

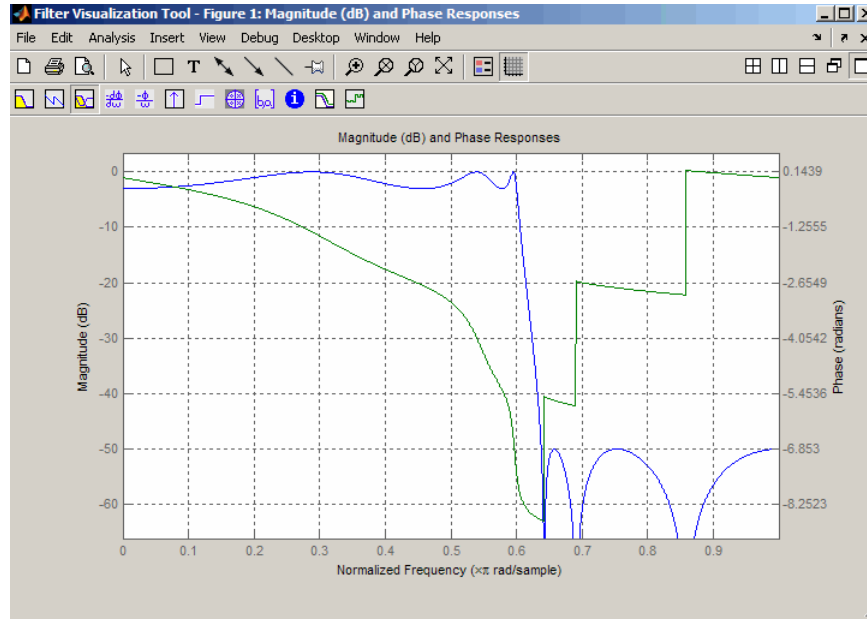
and u is the input, x is the state vector, and y is the output.

Examples

Lowpass Filter

For data sampled at 1000 Hz, design a sixth-order lowpass elliptic filter with a passband edge frequency of 300 Hz, which corresponds to a normalized value of 0.6, 3 dB of ripple in the passband, and 50 dB of attenuation in the stopband:

```
[z,p,k] = ellip(6,3,50,300/500);
[sos,g] = zp2sos(z,p,k);       % Convert to SOS form
Hd = dfilt.df2tsos(sos,g);    % Create a dfilt object
h = fvtool(Hd)                % Plot magnitude response
set(h,'Analysis','freq')      % Display frequency response
```

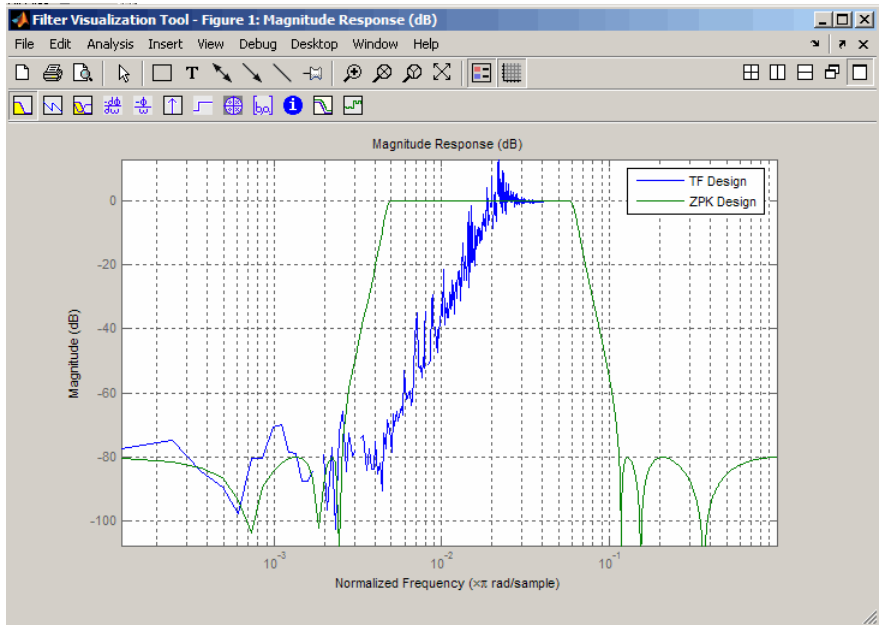


Limitations

In general, you should use the `[z,p,k]` syntax to design IIR filters. To analyze or implement your filter, you can then use the `[z,p,k]` output with `zp2sos` and an sos `dfilt` structure. For higher order filters (possibly starting as low as order 8), numerical problems due to roundoff errors may occur when forming the transfer function using the `[b,a]` syntax. The following example illustrates this limitation:

```
n = 6;
Rp = .1; Rs = 80;
Wn = [2.5e6 29e6]/500e6;
ftype = 'bandpass';

% Transfer Function design
[b,a] = ellip(n,Rp,Rs,Wn,ftype);
h1=dfilt.df2(b,a);    % This is an unstable filter.

% Zero-Pole-Gain design
```

```
[z, p, k] = ellip(n,Rp,Rs,Wn,ftype);
[sos,g]=zp2sos(z,p,k);
h2=dfilt.df2sos(sos,g);

% Plot and compare the results
hfvt=fvtool(h1,h2,'FrequencyScale','log');
legend(hfvt,'TF Design','ZPK Design')
```



Algorithm

The design of elliptic filters is the most difficult and computationally intensive of the Butterworth, Chebyshev Type I and II, and elliptic designs. `ellip` uses a five-step algorithm:

1. It finds the lowpass analog prototype poles, zeros, and gain using the `ellipap` function.

2. It converts the poles, zeros, and gain into state-space form.

3 It transforms the lowpass filter to a bandpass, highpass, or bandstop filter with the desired cutoff frequencies using a state-space transformation.

4 For digital filter design, `ellip` uses `bilinear` to convert the analog filter into a digital filter through a bilinear transformation with frequency prewarping. Careful frequency adjustment guarantees that the analog filters and the digital filters will have the same frequency response magnitude at `Wp` or `w1` and `w2`.

5 It converts the state-space filter back to transfer function or zero-pole-gain form, as required.

See Also

`besself`, `butter`, `cheby1`, `cheby2`, `ellipap`, `ellipord`

ellipap

Purpose

Elliptic analog lowpass filter prototype

Syntax

```
[z,p,k] = ellipap(n,Rp,Rs)
```

Description

`[z,p,k] = ellipap(n,Rp,Rs)` returns the zeros, poles, and gain of an order `n` elliptic analog lowpass filter prototype, with `Rp` dB of ripple in the passband, and a stopband `Rs` dB down from the peak value in the passband. The zeros and poles are returned in length `n` column vectors `z` and `p` and the gain in scalar `k`. If `n` is odd, `z` is length `n - 1`. The transfer function is

$$H(s) = \frac{z(s)}{p(s)} = k\frac{(s-z(1))(s-z(2))\cdots(s-z(n))}{(s-p(1))(s-p(2))\cdots(s-p(n))}$$

Elliptic filters offer steeper rolloff characteristics than Butterworth and Chebyshev filters, but they are equiripple in both the passband and the stopband. Of the four classical filter types, elliptic filters usually meet a given set of filter performance specifications with the lowest filter order.

`ellip` sets the passband edge angular frequency ω_0 of the elliptic filter to 1 for a normalized result. The *passband edge angular frequency* is the frequency at which the passband ends and the filter has a magnitude response of $10^{-Rp/20}$.

Algorithm

`ellipap` uses the algorithm outlined in [1]. It employs the M-file `ellipk` to calculate the complete elliptic integral of the first kind and the M-file `ellipj` to calculate Jacobi elliptic functions.

References

[1] Parks, T.W., and C.S. Burrus. *Digital Filter Design,* New York: John Wiley & Sons, 1987. Chapter 7.

See Also

`besselap`, `buttap`, `cheb1ap`, `cheb2ap`, `ellip`

Purpose

Minimum order for elliptic filters

Syntax

```
[n,Wp] = ellipord(Wp,Ws,Rp,Rs)
[n,Wp] = ellipord(Wp,Ws,Rp,Rs,'s')
```

Description

`ellipord` calculates the minimum order of a digital or analog elliptic filter required to meet a set of filter design specifications.

Digital Domain

`[n,Wp] = ellipord(Wp,Ws,Rp,Rs)` returns the lowest order `n` of the elliptic filter that loses no more than `Rp` dB in the passband and has at least `Rs` dB of attenuation in the stopband. The scalar (or vector) of corresponding cutoff frequencies `Wp`, is also returned. Use the output arguments `n` and `Wp` in `ellip`.

Choose the input arguments to specify the stopband and passband according to the following table.

Description of Stopband and Passband Filter Parameters

Parameter	Description
`Wp`	Passband corner frequency `Wp`, the cutoff frequency, is a scalar or a two-element vector with values between 0 and 1, with 1 corresponding to the normalized Nyquist frequency, π radians per sample.
`Ws`	Stopband corner frequency `Ws`, is a scalar or a two-element vector with values between 0 and 1, with 1 corresponding to the normalized Nyquist frequency.
`Rp`	Passband ripple, in decibels. This value is the maximum permissible passband loss in decibels.
`Rs`	Stopband attenuation, in decibels. This value is the number of decibels the stopband is attenuated with respect to the passband response.

Use the following guide to specify filters of different types.

Filter Type Stopband and Passband Specifications

Filter Type	Stopband and Passband Conditions	Stopband	Passband
Lowpass	`Wp` < `Ws`, both scalars	`(Ws,1)`	`(0,Wp)`
Highpass	`Wp` > `Ws`, both scalars	`(0,Ws)`	`(Wp,1)`
Bandpass	The interval specified by `Ws` contains the one specified by `Wp` (`Ws(1) < Wp(1) < Wp(2) < Ws(2)`).	`(0,Ws(1))` and `(Ws(2),1)`	`(Wp(1),Wp(2))`
Bandstop	The interval specified by `Wp` contains the one specified by `Ws` (`Wp(1) < Ws(1) < Ws(2) < Wp(2)`).	`(0,Wp(1))` and `(Wp(2),1)`	`(Ws(1),Ws(2))`

If your filter specifications call for a bandpass or bandstop filter with unequal ripple in each of the passbands or stopbands, design separate lowpass and highpass filters according to the specifications in this table, and cascade the two filters together.

Analog Domain

`[n,Wp] = ellipord(Wp,Ws,Rp,Rs,'s')` finds the minimum order `n` and cutoff frequencies `Wp` for an analog filter. You specify the frequencies `Wp` and `Ws` similar to those described in the Description of Stopband and Passband Filter Parameters on page 10-259 table above, only in this case you specify the frequency in radians per second, and the passband or the stopband can be infinite.

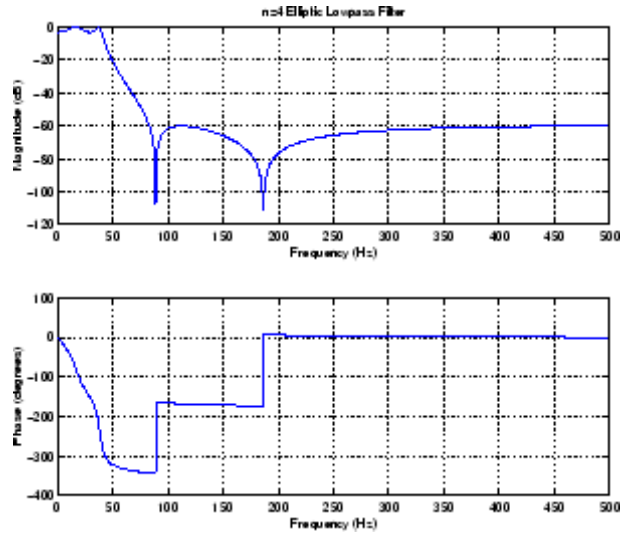
Use `ellipord` for lowpass, highpass, bandpass, and bandstop filters as described in the Filter Type Stopband and Passband Specifications on page 10-260 table above.

Examples

Example 1

For 1000 Hz data, design a lowpass filter with less than 3 dB of ripple in the passband defined from 0 to 40 Hz and at least 60 dB of ripple in the stopband defined from 150 Hz to the Nyquist frequency (500 Hz):

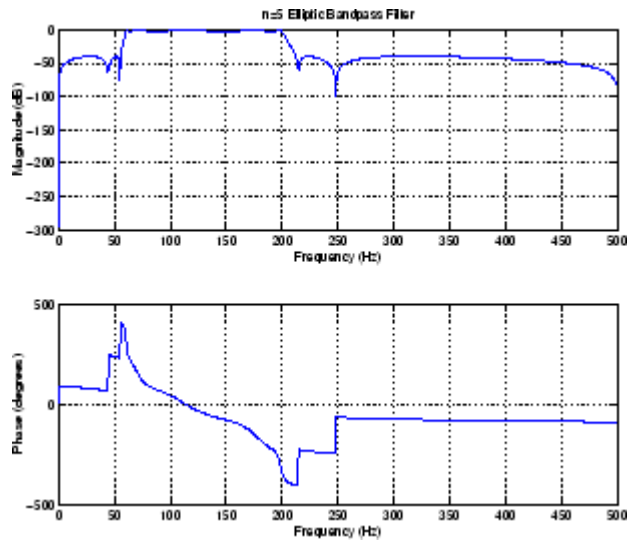
```
Wp = 40/500; Ws = 150/500;
Rp = 3; Rs = 60;
[n,Wp] = ellipord(Wp,Ws,Rp,Rs)
n =
     4
Wp =
    0.0800
[b,a] = ellip(n,Rp,Rs,Wp);
freqz(b,a,512,1000);
title('n=4 Elliptic Lowpass Filter')
```



Example 2

Now design a bandpass filter with a passband from 60 Hz to 200 Hz, with less than 3 dB of ripple in the passband, and 40 dB attenuation in the stopbands that are 50 Hz wide on both sides of the passband:

```
Wp = [60 200]/500; Ws = [50 250]/500;
Rp = 3; Rs = 40;
[n,Wp] = ellipord(Wp,Ws,Rp,Rs)
n =
     5
Wp =
    0.1200    0.4000
[b,a] = ellip(n,Rp,Rs,Wp);
freqz(b,a,512,1000);
title('n=5 Elliptic Bandpass Filter')
```



Algorithm

`ellipord` uses the elliptic lowpass filter order prediction formula described in [1]. The function performs its calculations in the analog domain for both the analog and digital cases. For the digital case, it

converts the frequency parameters to the *s*-domain before estimating the order and natural frequencies, and then converts them back to the *z*-domain.

`ellipord` initially develops a lowpass filter prototype by transforming the passband frequencies of the desired filter to 1 rad/s (for low- and highpass filters) and to -1 and 1 rad/s (for bandpass and bandstop filters). It then computes the minimum order required for a lowpass filter to meet the stopband specification.

References

[1] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing,* Englewood Cliffs, NJ: Prentice-Hall, 1975. Pg. 241.

See Also

`buttord`, `cheb1ord`, `cheb2ord`, `ellip`

equiripple

Purpose

Equiripple single-rate FIR filter from specification object

Syntax

```
hd = design(d,'equiripple')
hd = design(d,'equiripple',designoption,value,designoption,
...value,...)
```

Description

`hd = design(d,'equiripple')` designs an equiripple FIR digital filter using the specifications supplied in the object `d`. Equiripple filter designs minimize the maximum ripple in the passbands and stopbands. `hd` is a `dfilt` object

`hd = design(d,'equiripple',designoption,value,designoption, ...value,...)` returns an equiripple FIR filter where you specify design options as input arguments.

To determine the available design options, use `designopts` with the specification object and the design method as input arguments as shown.

```
designopts(d,'method')
```

For complete help about using `equiripple`, refer to the command line help system. For example, to get specific information about using `equiripple` with `d`, the specification object, enter the following at the MATLAB prompt.

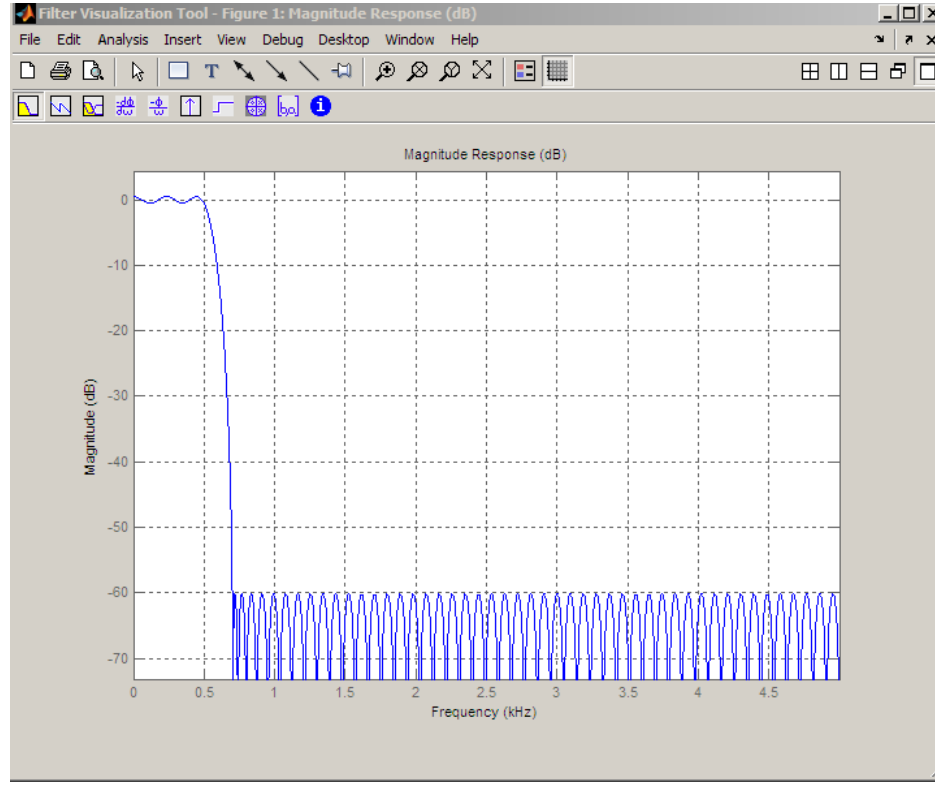
```
help(d,'equiripple')
```

Examples

First create a lowpass equiripple filter. Assume the data is sampled at 10,000 Hertz. The passband frequency is 500 Hertz with a stopband frequency of 700 Hz. The desired passband ripple is 1 dB with 60 dB of stopband attenuation.

```
Fs=10000;
d=fdesign.lowpass('Fp,Fst,Ap,Ast',500,700,1,60,10000);
Hd=design(d,'equiripple');
```

Displaying the filter in FVTool shows the equiripple nature of the filter.



The next example designs a highpass equiripple filter with a direct-form transposed structure by specifying the **`filterstructure`** argument. To set the design options for the filter, use the `designopts` method and options object `opts`.

```
d=fdesign.highpass;
opts=designopts(d,'equiripple')

opts =

    FilterStructure: 'dffir'
      DensityFactor: 16
```

```
opts.FilterStructure='dffirt'

opts =

    FilterStructure: 'dffirt'
      DensityFactor: 16
opts.densityfactor=20

opts =

    FilterStructure: 'dffirt'
      DensityFactor: 16
      densityfactor: 20
Hd=design(d,'equiripple',opts)

Hd =

     FilterStructure: 'Direct-Form FIR Transposed'
           Numerator: [1x45 double]
    PersistentMemory: false
```

See Also

design, designmethods.

Purpose

Equalize lengths of transfer function's numerator and denominator

Syntax

```
[b,a] = eqtflength(num,den)
[b,a,n,m] = eqtflength(num,den)
```

Description

`[b,a] = eqtflength(num,den)` modifies the vector `num` and/or the vector `den`, so that the resulting output vectors `b` and `a` have the same length. The input vectors `num` and `den` may have different lengths. The vector `num` represents the numerator polynomial of a given discrete-time transfer function, and the vector `den` represents its denominator. The resulting numerator `b` and denominator `a` represent the same discrete-time transfer function, but these vectors have the same length.

`[b,a,n,m] = eqtflength(num,den)` modifies the vectors as above and also returns the numerator order `n` and the denominator `m`, not including any trailing zeros.

Use `eqtflength` to obtain a numerator and denominator of equal length before applying transfer function conversion functions such as `tf2ss` and `tf2zp` to discrete-time models.

Examples

```
num = [1 0.5];
den = [1 0.75 0.6 0];
[b,a,n,m] = eqtflength(num,den)
b =
    1.0000    0.5000         0
a =
    1.0000    0.7500    0.6000
n =
     1
m =
     2
```

Algorithm

`eqtflength(num,den)` appends zeros to either `num` or `den` as necessary. If both `num` and `den` have trailing zeros in common, these are removed.

See Also

`tf2ss`, `tf2zp`

Purpose

Open Filter Design and Analysis Tool

Syntax

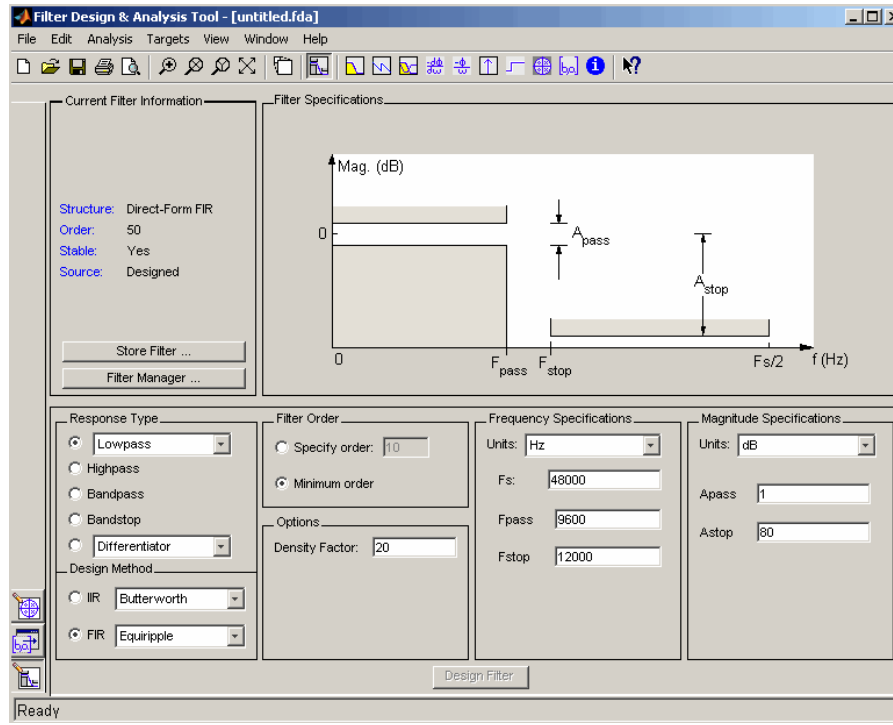
```
fdatool
```

Description

`fdatool` opens the Filter Design and Analysis Tool (FDATool). Use this tool to

- Design filters
- Analyze filters
- Modify existing filter designs

See Chapter 5, “FDATool: A Filter Design and Analysis GUI”.



Remarks

The Filter Design and Analysis Tool provides more design methods than the SPTool Filter Designer. It also integrates advanced filter design methods from the Filter Design Toolbox product.

Note The Filter Design and Analysis Tool requires a screen resolution greater than 640 x 480.

See Also

`fvtool`, `sptool`, `wvtool`

fdesign

Purpose

Filter specification object

Syntax

```
d = fdesign.response
d = fdesign.response(spec)
d = fdesign.response(...,fs)
d = fdesign.response(...,magunits)
```

Description

Filter Specification Objects

`d = fdesign.response` returns a filter specification object `d`, of filter response *`response`*. To create filters from `d`, use one of the design methods listed in Chapter 3, "Designing a Filter in Fdesign — Process Overview"

Note Several of the filter response types described below are only available if your installation includes the Filter Design Toolbox. The Filter Design Toolbox significantly expands the functionality available for the specification, design, and analysis of filters.

Here is how you design filters using `fdesign`.

1. Use `fdesign.response` to construct a filter specification object.
2. Use `designmethods` to determine which filter design methods work for your new filter specification object.
3. Use `design` to apply your filter design method from step 2 to your filter specification object to construct a filter object.
4. Use FVTool to inspect and analyze your filter object.

Note `fdesign` does not create filters. `fdesign` returns a filter specification object that contains the specifications for a filter, such as the passband cutoff or attenuation in the stopband. To design a filter `hd` from a filter specification object `d`, use `d` with a filter design method such as `butter` —`hd = design(d,'butter')`.

For more guidance about using `fdesign`, refer to the examples in Chapter 3, "Designing a Filter in Fdesign — Process Overview". Alternatively, type the following at the MATLAB prompt for more information:

```
help fdesign
```

`response` can be one of the entries in the following table that specify the filter response desired, such as a bandstop filter or an interpolator.

fdesign Response String	Description
`arbmag`	`fdesign.arbmag` creates an object to specify IIR filters that have arbitrary magnitude responses defined by the input arguments.
`bandpass`	`fdesign.bandpass` creates an object to specify bandpass filters.
`bandstop`	`fdesign.bandstop` creates an object to specify bandstop filters.
`differentiator`	`fdesign.differentiator` creates an object to specify differentiators.
`highpass`	`fdesign.highpass` creates an object to specify highpass filters.
`hilbert`	`fdesign.hilbert` creates an object to specify Hilbert filters.

fdesign Response String	Description
`lowpass`	`fdesign.lowpass` creates an object to specify lowpass filters.
`pulseshaping`	`fdesign.pulseshaping` creates an object to specify pulse-shaping filters.

Use the `doc fdesign.`*`response`* syntax at the MATLAB prompt to get help on a specific structure. Using `doc` in a syntax like

```
doc fdesign.lowpass
doc fdesign.bandstop
```

gets more information about the lowpass or bandstop structure objects.

Each `response` has a property `Specification` that defines the specifications to use to design your filter. You can use defaults or specify the `Specification` property when you construct the specifications object.

With the strings for the `Specification` property, you provide filter constraints such as the filter order or the passband attenuation to use when you construct your filter from the specification object.

Properties

`fdesign` returns a filter specification object. Every filter specification object has the following properties.

Property Name	Default Value	Description
`Response`	Depends on the chosen type	Defines the type of filter to design, such as an interpolator or bandpass filter. This is a read-only value.

Property Name	Default Value	Description
`Specification`	Depends on the chosen type	Defines the filter characteristics used to define the desired filter performance, such as the cutoff frequency `Fstop` or the filter order `N`.
`Description`	Depends on the filter type you choose	Contains descriptions of the filter specifications used to define the object, and the filter specifications you use when you create a filter from the object. This is a read-only value.
`NormalizedFrequency`	Logical `true`	Determines whether the filter calculation uses normalized frequency from 0 to 1, or the frequency band from 0 to Fs/2, the sampling frequency. Accepts either `true` or `false` without single quotation marks.

In addition to these properties, filter specification objects may have other properties as well, depending on whether they design `dfilt` objects or `mfilt` objects.

Added Properties for mfilt Objects	Description
`DecimationFactor`	Specifies the amount to decrease the sampling rate. Always a positive integer.

Added Properties for mfilt Objects	Description
`InterpolationFactor`	Specifies the amount to increase the sampling rate. Always a positive integer.
`PolyphaseLength`	Polyphase length is the length of each polyphase subfilter that composes the decimator or interpolator or rate-change factor filters. Total filter length is the product of `pl` and the rate change factors. `pl` must be an even integer.

`d = fdesign.`*`response`*`(spec)`. In `spec`, you specify the variables to use that define your filter design, such as the passband frequency or the stopband attenuation. These variables are applied to the filter design method you choose to design your filter.

For example, when you create a default lowpass filter specification object `d`, `fdesign` sets the passband frequency `Fpass`, the stopband frequency `Fstop`, the stopband attenuation `Astop`, and the passband attenuation `Apass` (ripple in the passband) for `d`:

```
d = fdesign.lowpass

d =

               Response: 'Lowpass'
          Specification: 'Fp,Fst,Ap,Ast'
            Description: {4x1 cell}
    NormalizedFrequency: true
                  Fpass: 0.45
                  Fstop: 0.55
                  Apass: 1
                  Astop: 60
```

However, lowpass design syntax accepts any one of the following `Spec` strings (among others) to define the filter response:

Spec String	Description
`Fp,Fst,Ap,Ast`	Define the filter by specifying the passband cutoff, the stopband cutoff, the ripple in the passband, and the attenuation in the stopband. This is the default string for a lowpass filter.
`N,Fc`	Set the filter order and the cutoff frequency to define the filter.
`N,Fp,Ap`	Set the filter order, passband cutoff frequency, and passband ripple.
`N,Fst,Ast`	Define the filter by setting the order, stopband frequency, and stopband attenuation.
`N,Fp,Ap,Ast`	Set the order, passband cutoff frequency, passband ripple, and stopband attenuation.
`N,Fp,Fst,Ap`	Set the filter order, passband cutoff frequency, stopband frequency, and passband ripple.

Other filter object types, such as Nyquist or highpass, accept a different set of strings for `Spec`. Refer to the Help system for details about the strings for each filter type.

One important note is that the `Spec` string you choose controls which design method works for the specifications object.

For the lowpass filter specification object `d` from earlier, you can use `butter`, `cheby1`, `cheby2`, or `equiripple` (to name a few) to design a filter. However, if the `Spec` string had been `'n,fp,fst,ap'`, you could only use the `ellip` design method to design your filter.

When you implement this lowpass filter `hd` using a filter design method such as Butterworth (the `butter` design function), the constraints in `fp`, `fst`, `ap`, and `ast` (the default string and filter specification) define the response of the final minimum-order lowpass filter:

```
hd = design(d,'butter')

hd =
```

```
     FilterStructure: 'Direct-Form II, Second-Order Sections'
          Arithmetic: 'double'
           sosMatrix: [13x6 double]
         ScaleValues: [14x1 double]
    PersistentMemory: false
```

FVTool shows that `hd` is a lowpass filter that meets the design specification.

`d = fdesign.`*`response`*`(...,fs)` adds the argument `fs`, specified in Hz to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

`d = fdesign.`*`response`*`(...,magunits)` specifies the units for any magnitude specification you provide in the input arguments. `magunits` can be one of

- `linear` — specify the magnitude in linear units
- `dB` — specify the magnitude in decibels
- `squared` — specify the magnitude in power units

When you omit the `magunits` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Using Filter Design Methods with Specification Objects

After you create a filter specification object, you use a filter design method to implement your filter with a selected algorithm. The following methods are available for filter specification objects, but all methods do not apply to all object types. Also, the specification string you use to define the object changes the algorithms available to design a filter. Enter `doc butter`, for example, to get more information about using the Butterworth design method with your filter specification object.

Design Function	Description
`butter`	Implement a Butterworth filter resulting in an SOS filter with direct-form II structure
`cheby1`	Implement a Chebyshev Type I filter, resulting in a direct-form II second-order filter
`cheby2`	Implement a Chebyshev Type II filter, resulting in an SOS filter with direct-form II structure
`ellip`	Implement an elliptic filter resulting in an SOS filter with direct-form II structure
`equiripple`	Implement an equiripple filter
`firls`	Implement a least-squares filter
`kaiserwin`	Implement a filter that uses a Kaiser window

When you use any of the design methods without providing an output argument, the resulting filter design appears in FVTool by default.

Along with filter design methods, `fdesign` works with supporting methods that help you create filter specification objects or determine which design methods work for a given specifications object.

Supporting Function	Description
`designmethods`	Return the design methods.
`designopts`	Return the input arguments and default values that apply to a specifications object and method

You can set filter specification values by passing them after the `Specification` argument, or by passing the values without the `Specification` string.

When the first input to `fdesign` is not a valid `Specification` string like '`n,fc`', `fdesign` assumes that the input argument is a filter specification and applies it using the default `Specification` string —`fp,fst,ap,ast` for a lowpass object, for example.

Examples

The following examples require only the Signal Processing Toolbox.

Example 1–Bandstop Filter

A bandstop filter specification object for data sampled at 8,000 Hertz. The stopband between 2000 and 2400 Hertz is attenuated at least 80 dB.

```
d = fdesign.bandstop('Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2',...
           1600,2000,2400,2800,1,80,1,8000);
```

Example 2–Lowpass Filter

A lowpass filter specification object for data sampled at 10,000 Hertz. The passband frequency is 500 Hz and the stopband frequency is 750 Hz. The passband ripple is set to 1 dB and the required attenuation in the stopband is 80 dB.

```
d=fdesign.lowpass('Fp,Fst,Ap,Ast',500,750,1,80,10000);
```

Example 3–Highpass Filter

A default highpass filter specification object.

```
d=fdesign.highpass % Creates specifications object.

d =

               Response: 'Minimum-order highpass'
      Specification    : 'Fst,Fp,Ast,Ap'
            Description: {4x1 cell}
    NormalizedFrequency: true
                     Fs: 'Normalized'
                  Fstop: 0.4500
                  Fpass: 0.5500
                  Astop: 60
                  Apass: 1

d.Description

ans =
```

```
'Stopband Frequency'
'Passband Frequency'
'Stopband Attenuation (dB)'
'Passband Ripple (dB)'
```

Notice the correspondence between the properties `Specification` and `Description` — in `Description` you see in words the definitions of the variables shown in `Specification`.

Example 4–Filter Specification and Design

Lowpass Butterworth filter specification object

Use a filter specification object to construct a lowpass Butterworth filter with default `Specification` `fp,fst,ap,ast` — the edge frequencies of the passband and stopband, the attenuation in the passband, and the attenuation in the stopband. Start by creating the specifications object `d` and providing the filter order and cutoff frequency values.

```
d = fdesign.lowpass(0.4,0.5,1,80);
```

Determine which design methods apply to `d`. With only the Signal Processing Toolbox installed, you can choose among the following algorithms:

```
>>designmethods(d)


Design Methods for class fdesign.lowpass:


butter
cheby1
cheby2
ellip
```

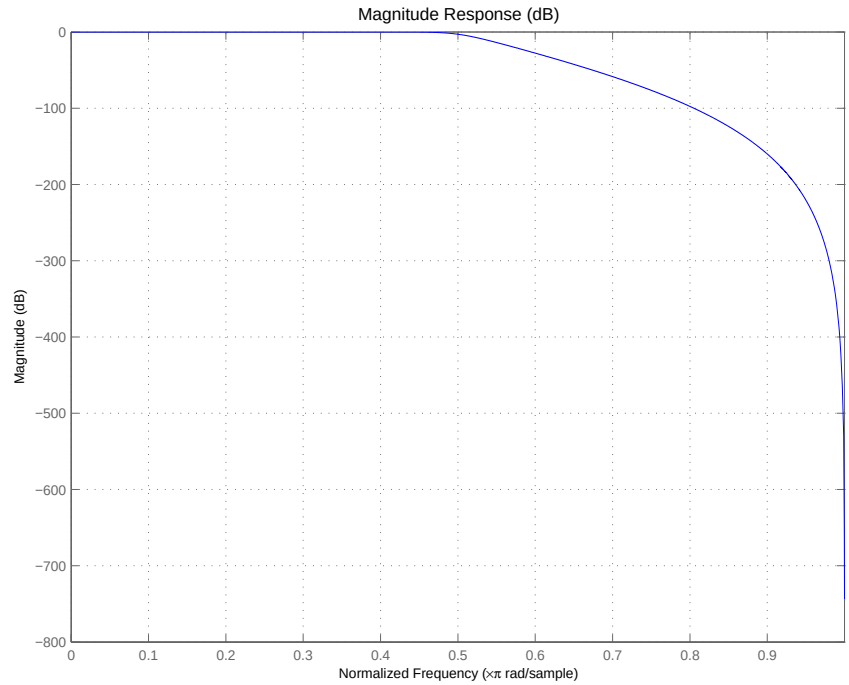
With the Filter Design Toolbox installed, you have additional algorithms available.

```
>>designmethods(d)

Design Methods for class fdesign.lowpass (Fp,Fst,Ap,Ast):

butter
cheby1
cheby2
ellip
equiripple
ifir
kaiserwin
multistage
```

You can use `d` and the `butter` design method to design a Butterworth filter.

```
hd = design(d,'butter','matchexactly','passband');
fvtool(hd);
```

The resulting filter magnitude response shown by FVTool appears in the following figure.



If you had a default Nyquist filter specification object `d`

```
d = fdesign.nyquist
```

you could find out which design methods apply to `d` by entering

```
designmethods(d)

Design methods for class fdesign.nyquist:

kaiserwin
```

See Also

`butter`, `cheby1`, `cheby2` , `design`, `designmethods`, `designopts`, `equiripple`.

Purpose

Arbitrary response magnitude filter specification object

Syntax

```
d = fdesign.arbmag
d = fdesign.arbmag(specification)
d = fdesign.arbmag(specification,specvalue1,specvalue2,...)
d = fdesign.arbmag(specvalue1,specvalue2,specvalue3)
d = fdesign.arbmag(...,fs)
```

Description

`d = fdesign.arbmag` constructs an arbitrary magnitude filter designer `d`.

`d = fdesign.arbmag(specification)` initializes the `Specification` property for specifications object `d` to the string in `specification`. The input argument `specification` must be one of the strings shown in the following table. Specification strings are not case sensitive.

Specification String	Description of Resulting Filter
`n,f,a`	Single band design (default). FIR and IIR (`n` is the order for both numerator and denominator)
`n,b,f,a`	Multiband design where `b` defines the number of bands
`nb,na,f,a`	IIR single band design
`nb,na,b,f,a`	IIR multiband design where `b` defines the number of bands

The following table describes the arguments in the `specification` strings.

Argument	Description
`a`	Amplitude vector. Values in `a` define the filter amplitude at frequency points you specify in `f`, the frequency vector. If you use `a`, you must use `f` as well. Amplitude values must be real. For complex values designs, use `fdesign.arbmagnphase`.
`b`	Number of bands in the multiband filter.
`f`	Frequency vector. Frequency values in specified in `f` indicate locations where you provide specific filter response amplitudes. When you provide `f` you must also provide `a`.
`n`	Filter order for FIR filters and the numerator and denominator orders for IIR filters.
`nb`	Numerator order for IIR filters.
`na`	Denominator order for IIR filter designs.

By default, this method assumes that all frequency specifications are supplied in normalized frequency.

Specifying f and a

`f` and `a` are the input arguments you use to define the filter response desired. Each frequency value you specify in `f` must have a corresponding response value in `a`. The following example creates a filter with two passbands (`b = 4`) and shows how `f` and `a` are related. This example is for illustration only. It is not an actual filter.

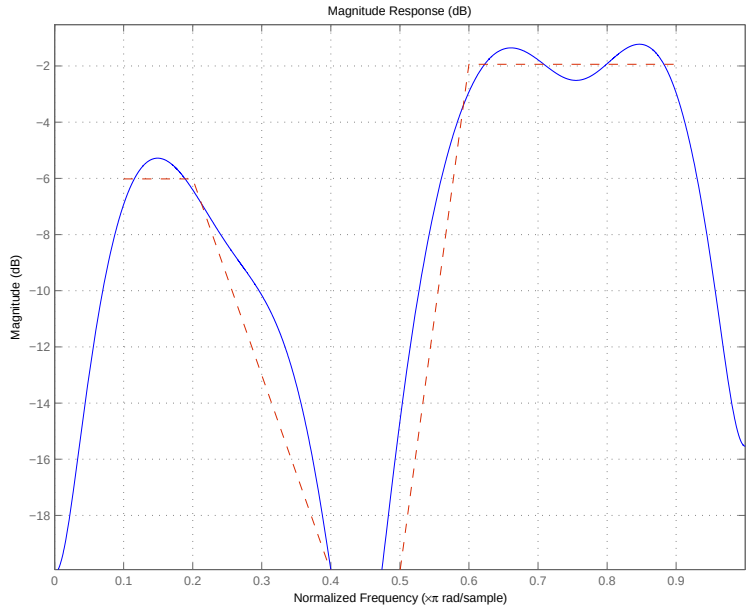
Define the frequency vector `f` as `[0 0.1 0.2 0.4 0.5 0.6 0.9 1.0]`

Define the response vector `a` as `[0 0.5 0.5 0.1 0.1 0.8 0.8 0]`

These specifications connect `f` and `a` as shown in the following table.

f (Normalized Frequency)	a (Response Desired at f)
0	0
0.1	0.5
0.2	0.5
0.4	0.1
0.5	0.1
0.6	0.8
0.9	0.8
1.0	0.0

A response with two passband—one roughly between 0.1 and 0.2 and the second between 0.6 and 0.9—results from the mapping between `f` and `a`. A filter that used `f` and `a` might look .



Different specification types often have different design methods available. Use `designmethods(d)` to get a list of design methods available for a given specification string and specifications object.

`d = fdesign.arbmag(specification,specvalue1,specvalue2,...)` initializes the filter specification object specifications with `specvalue1`, `specvalue2`, and so on. Use `get(d,'description')` for descriptions of the various specifications `specvalue1`, `specvalue2`, ... spec*n*.

`d = fdesign.arbmag(specvalue1,specvalue2,specvalue3)` uses the default specification string `n,f,a`, setting the filter order, filter frequency vector, and the amplitude vector to the values `specvalue1`, `specvalue2`, and `specvalue3`.

`d = fdesign.arbmag(...,fs)` specifies the sampling frequency in Hz. All other frequency specifications are also assumed to be in Hz when you specify `fs`.

Examples

These three examples introduce designing filters that have arbitrary filter response shapes. In this first example, use `fdesign.arbmag` to design a single-band, arbitrary-magnitude FIR filter. The design process uses the default design method for the `n,f,a` specification, as shown in the following code:

```
n = 120;
f = linspace(0,1,100);  % 100 frequency points.
as = ones(1,100)-f*0.2;
absorb = [ones(1,30),(1-0.6*bohmanwin(10))',...
ones(1,5), (1-0.5*bohmanwin(8))',ones(1,47)];
a = as.*absorb; % Optical absorption of atomic Rubidium 87 vapor.
d = fdesign.arbmag(n,f,a);
hd1 = design(d,'freqsamp');
```

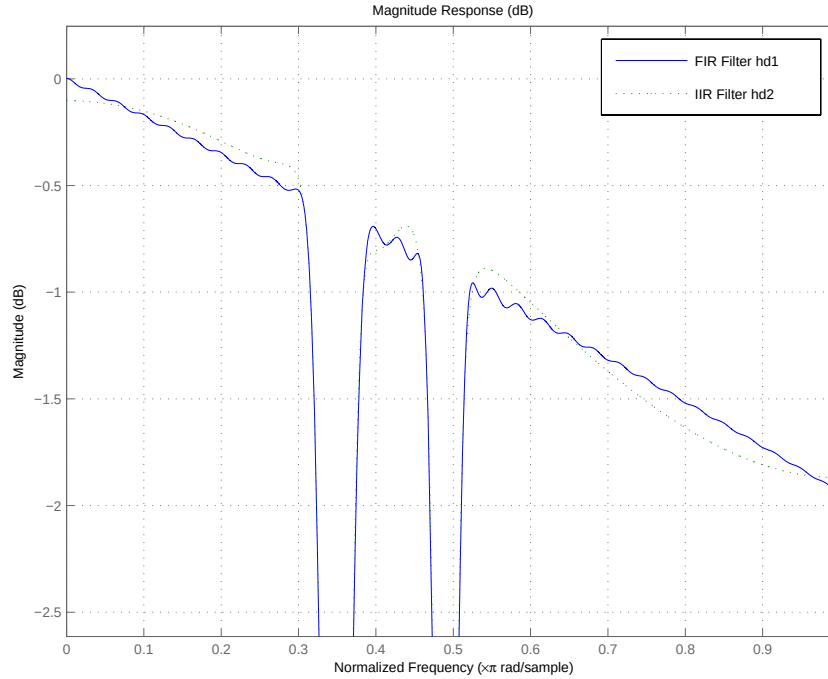
Next, design a single-band, arbitrary-magnitude IIR filter and display the magnitude response in FVTool. Use `f` and `a` from the previous example as input arguments for this case. Display the response from the previous example in FVTool as well, because the FIR and IIR filters are similar.

To demonstrate that the same specification generates both FIR and IIR filters, use the same specifications object `d`, but change the design method to `iirlpnorm`.

```
d.filterorder=10

d =

                Response: 'Arbitrary Magnitude'
           Specification: 'N,F,A'
             Description: {'Filter Order';'Frequency Vector';'
                            Amplitude Vector'}
     NormalizedFrequency: true
             FilterOrder: 10
             Frequencies: [1x100 double]
              Amplitudes: [1x100 double]
```

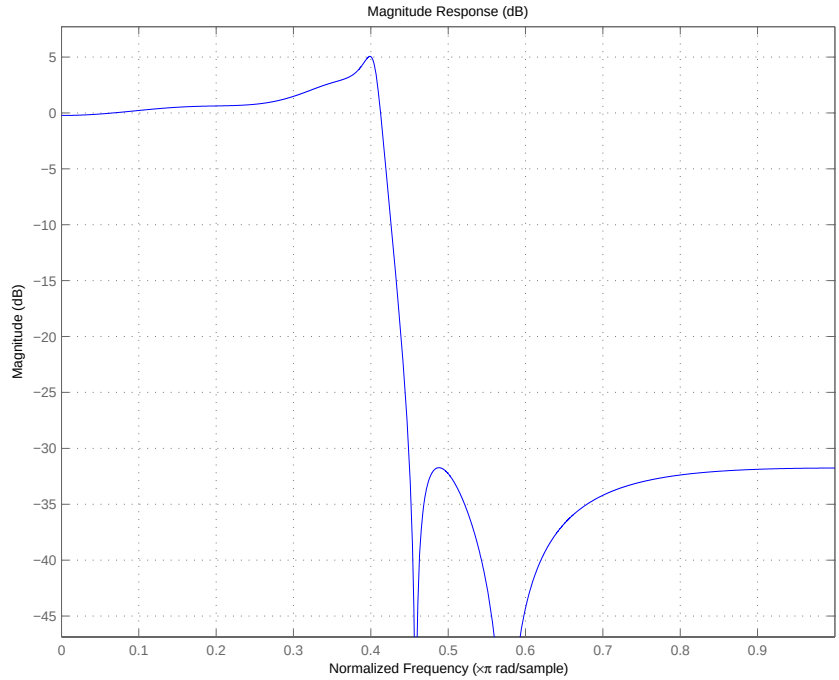
```
hd2=design(d,'iirlpnorm') % Design an IIR filter from the same object.

hd2 =

         FilterStructure: 'Direct-Form II, Second-Order Sections'
              Arithmetic: 'double'
               sosMatrix: [5x6 double]
             ScaleValues: [0.85714867585342;1;1;1;1;1]
        PersistentMemory: false

fvtool(hd1,hd2)
```

FVTool returns the following plot for the filters.



For the third example, design a multiband filter for noise shaping when you are simulating the Rayleigh fading phenomenon in a wireless communications channel. This example uses the default design method for `fdesign.arbmag` specifications objects with the `nb,na,nbands` specification—`iirlpnorm`.

```
nb = 4;     % Numerator order.
na = 6;     % Denominator order.
nbands = 2; % Number of filter bands.
f1 = 0:0.01:0.4; % Frequency vector values.
a1 = 1.0 ./ (1 - (f1./0.42).^2).^0.25; % Amplitude values.
f2 = [.45 1];
a2 = [0 0];
d = fdesign.arbmag('nb,na,b,f,a',nb,na,nbands,f1,a1,f2,a2);
design(d); % Starts FVTool to display the filter response.
```



The filter response shows the characteristic shape for noise shaping—increasing gain with increasing frequency in the passband, and a narrow transition region.

See Also

`design`, `designmethods`,

Purpose

Bandpass filter specification object

Syntax

```
d = fdesign.bandpass
d = fdesign.bandpass(spec)
d = fdesign.bandpass(spec,specvalue1,specvalue2,...)
d = fdesign.bandpass(specvalue1,specvalue2,specvalue3,
specvalue4,...specvalue4,specvalue5,specvalue6)
d = fdesign.bandpass(...,fs)
d = fdesign.bandpass(...,magunits)
```

Description

`d = fdesign.bandpass` constructs a bandpass filter specification object `d`, applying default values for the properties `Fstop1`, `Fpass1`, `Fpass2`, `Fstop2`, `Astop1`, `Apass`, and `Astop2` — one possible set of values you use to specify a bandpass filter.

Using `fdesign.bandpass` with a design method generates a `dfilt` object.

`d = fdesign.bandpass(spec)` constructs object `d` and sets its `Specification` property to `spec`. Entries in the `spec` string represent various filter response features, such as the filter order, that govern the filter design. Valid entries for *`spec`* are shown below and used to define the bandpass filter. The strings are not case sensitive.

- `fst1,fp1,fp2,fst2,ast1,ap,ast2` (default `spec`)
- `n,f3dB1,f3dB2`
- `n,f3dB1,f3dB2,ap`
- `n,f3dB1,f3dB2,ast`
- `n,f3dB1,f3dB2,ast1,ap,ast2`
- `n,f3dB1,f3dB2,bwp`
- `n,f3dB1,f3dB2,bwst`
- `n,fc1,fc2`
- `n,fc1,fc2,ast1,ap,ast2`

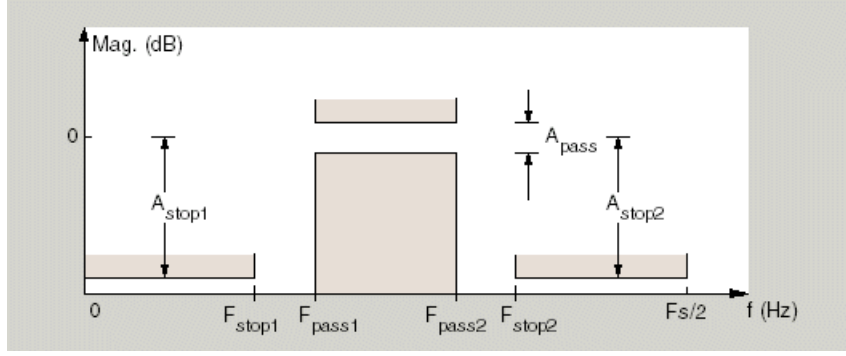
- `n,fp1,fp2,ap`
- `n,fp1,fp2,ast1,ap,ast2`
- `n,fst1,fp1,fp2,fst2`
- `n,fst1,fp1,fp2,fst2,ap`
- `n,fst1,fst2,ast`
- `nb,na,fst1,fp1,fp2,fst2`

The string entries are defined as follows:

- `ap` — amount of ripple allowed in the pass band. Also called Apass.
- `ast1` — attenuation in the first stop band in decibels (the default units). Also called Astop1.
- `ast2` — attenuation in the second stop band in decibels (the default units). Also called Astop2.
- `bwp` — bandwidth of the filter passband. Specified in normalized frequency units.
- `bwst` — bandwidth of the filter stopband. Specified in normalized frequency units.
- `f3dB1` — cutoff frequency for the point 3 dB point below the passband value for the first cutoff. Specified in normalized frequency units. (IIR filters)
- `f3dB2` — cutoff frequency for the point 3 dB point below the passband value for the second cutoff. Specified in normalized frequency units. (IIR filters)
- `fc1` — cutoff frequency for the point 3 dB point below the passband value for the first cutoff. Specified in normalized frequency units. (FIR filters)
- `fc2` — cutoff frequency for the point 3 dB point below the passband value for the second cutoff. Specified in normalized frequency units. (FIR filters)

- `fp1` — frequency at the edge of the start of the pass band. Specified in normalized frequency units. Also called Fpass1.
- `fp2` — frequency at the edge of the end of the pass band. Specified in normalized frequency units. Also called Fpass2.
- `fst1` — frequency at the edge of the start of the first stop band. Specified in normalized frequency units. Also called Fstop1.
- `fst2` — frequency at the edge of the start of the second stop band. Specified in normalized frequency units. Also called Fstop2.
- `n` — filter order for FIR filters. Or both the numerator and denominator orders for IIR filters when na and nb are not provided.
- `na` — denominator order for IIR filters
- `nb` — numerator order for IIR filters

Graphically, the filter specifications look similar to those shown in the following figure.



Regions between specification values like `fst1` and `fp1` are transition regions where the filter response is not explicitly defined.

The filter design methods that apply to a bandpass filter specification object change depending on the `Specification` string. Use `designmethods` to determine which design method applies to an object and its specification string.

`d = fdesign.bandpass(spec,specvalue1,specvalue2,...)` constructs an object `d` and sets its specifications at construction time.

`d = fdesign.bandpass(specvalue1,specvalue2,specvalue3, specvalue4,...specvalue4,specvalue5,specvalue6)` constructs `d`, an object with the default `Specification` property string, using the values you provide as input arguments for `specvalue1,specvalue2,specvalue3,specvalue4,specvalue4,specvalue5, specvalue6` and `specvalue7`.

`d = fdesign.bandpass(...,fs)` adds the argument `fs`, specified in Hz to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

`d = fdesign.bandpass(...,magunits)` specifies the units for any magnitude specification you provide in the input arguments. `magunits` can be one of

- `linear` — specify the magnitude in linear units
- `dB` — specify the magnitude in dB (decibels)
- `squared` — specify the magnitude in power units

When you omit the `magunits` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Examples

These examples show how to construct a bandpass filter specification object. First, create a default specifications object without using input arguments.

```
d = fdesign.bandpass
d =

                Response: 'Minimum-order bandpass'
           Specification: 'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2'
             Description: {7x1 cell}
```

```
    NormalizedFrequency: true
                Fstop1: 0.3500
                Fpass1: 0.4500
                Fpass2: 0.5500
                Fstop2: 0.6500
                Astop1: 60
                 Apass: 1
                Astop2: 60
```

Now, pass the filter specifications that correspond to the default `Specification` — `fst1,fp1,fp2,fst2,ast1,ap,ast2` — without specifying the `Specification` string. This example adds `fs` as the final input argument to specify the sampling frequency of 48 Hz.

```
d = fdesign.bandpass(10, 12, 14, 16, 80, .5, 60, 48)
d =

               Response: 'Minimum-order bandpass'
          Specification: 'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2'
            Description: {7x1 cell}
    NormalizedFrequency: false
                     Fs: 48
                Fstop1: 10
                Fpass1: 12
                Fpass2: 14
                Fstop2: 16
                Astop1: 80
                 Apass: 0.5000
                Astop2: 60
```

Next create a specifications object by passing a specification type string '`n,fc1,fc2`' — the resulting object uses default values for `n`, `fc1`, and `fc2`.

```
d = fdesign.bandpass('n,fc1,fc2')
d =

               Response: 'Bandpass with cutoff'
```

```
            Specification: 'N,Fc1,Fc2'
              Description: {3x1 cell}
        NormalizedFrequency: true
                FilterOrder: 10
                   Fcutoff1: 0.4000
                   Fcutoff2: 0.6000
```

Create the same filter, passing the specification values to the object rather than accepting the default values for `n`, `fc1`, and `fc2`. You can include the sampling frequency `fs` as the final input argument, and that you specify the cutoff frequencies in Hz since `fs` is in Hz.

```
d = fdesign.bandpass('n,fc1,fc2', 10, 9600, 14400, 48000)
d =

                   Response: 'Bandpass with cutoff'
            Specification: 'N,Fc1,Fc2'
              Description: {3x1 cell}
        NormalizedFrequency: false
                         Fs: 48000
                FilterOrder: 10
                   Fcutoff1: 9600
                   Fcutoff2: 14400
```

The following topics include examples of fdesign.bandpass:

"Basic Filter Design Process"

"Floating-Point to Fixed-Point Conversion"

"Process Flow Diagram and Filter Design Methodology"

See Also

`fdesign`, `fdesign.bandstop`, `fdesign.highpass`, `fdesign.lowpass`

Purpose

Bandstop filter specification object

Syntax

```
d = fdesign.bandstop
d = fdesign.bandstop(spec)
d = fdesign.bandstop(spec,specvalue1,specvalue2,...)
d = fdesign.bandstop(specvalue1,specvalue2,specvalue3,specvalue4,...
specvalue5,specvalue6,specvalue7)
d = fdesign.bandstop(...,fs)
d = fdesign.bandstop(...,magunits)
```

Description

`d = fdesign.bandstop` constructs a bandstop filter specification object `d`, applying default values for the properties `Fpass1`, `Fstop1`, `Fstop2`, `Fpass2`, `Apass1`, `Astop1` and `Apass2`.

Using `fdesign.bandstop` with a design method generates a `dfilt` object.

`d = fdesign.bandstop(spec)` constructs object `d` and sets its '`Specification`' to *`spec`*. Entries in the `spec` string represent various filter response features, such as the filter order, that govern the filter design. Valid entries for *`spec`* are shown below. The strings are not case sensitive.

- `fp1,fst1,fst2,fp2,ap1,ast,ap2 (defaultspec)`
- `n,f3dB1,f3dB2`
- `n,f3dB1,f3dB2,ap`
- `n,f3dB1,f3dB2,ap,ast`
- `n,f3dB1,f3dB2,ast`
- `n,f3dB1,f3dB2,bwp`
- `n,f3dB1,f3dB2,bwst`
- `n,fc1,fc2`
- `n,fc1,fc2,ap1,ast,ap2`
- `n,fp1,fp2,ap`

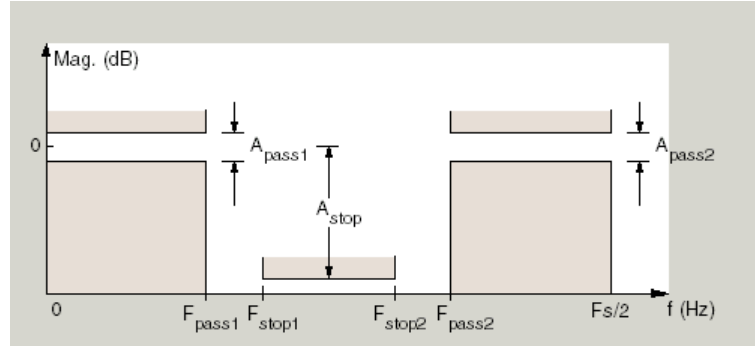
- `n,fp1,fp2,ap,ast`
- `n,fp1,fst1,fst2,fp2`
- `n,fp1,fst1,fst2,fp2,ap`
- `n,fst1,fst2,ast`
- `nb,na,fp1,fst1,fst2,fp2`

The string entries are defined as follows:

- `ap` — amount of ripple allowed in the passband in decibels (the default units). Also called Apass.
- `ap1` — amount of ripple allowed in the pass band in decibels (the default units). Also called Apass1.
- `ap2` — amount of ripple allowed in the pass band in decibels (the default units). Also called Apass2.
- `ast` — attenuation in the first stopband in decibels (the default units). Also called Astop1.
- `bwp` — bandwidth of the filter passband. Specified in normalized frequency units.
- `bwst` — bandwidth of the filter stopband. Specified in normalized frequency units.
- `f3dB1` — cutoff frequency for the point 3 dB point below the passband value for the first cutoff. Specified in normalized frequency units.
- `f3dB2` — cutoff frequency for the point 3 dB point below the passband value for the second cutoff. Specified in normalized frequency units.
- `fc1` — cutoff frequency for the point 3 dB point below the passband value for the first cutoff. Specified in normalized frequency units. (FIR filters)
- `fc2` — cutoff frequency for the point 3 dB point below the passband value for the second cutoff. Specified in normalized frequency units. (FIR filters)

- `fp1` — frequency at the start of the pass band. Specified in normalized frequency units. Also called Fpass1.
- `fp2` — frequency at the end of the pass band. Specified in normalized frequency units. Also called Fpass2.
- `fst1` — frequency at the end of the first stop band. Specified in normalized frequency units. Also called Fstop1.
- `fst2` — frequency at the start of the second stop band. Specified in normalized frequency units. Also called Fstop2.
- `n` — filter order.
- `na` — denominator order for IIR filters.
- `nb` — numerator order for IIR filters.

Graphically, the filter specifications look similar to those shown in the following figure.



Regions between specification values like `fp1` and `fst1` are transition regions where the filter response is not explicitly defined.

The filter design methods that apply to a bandstop filter specification object change depending on the `Specification` string. Use `designmethods` to determine which design method applies to an object and its specification string.

`d = fdesign.bandstop(spec,specvalue1,specvalue2,...)` constructs an object `d` and sets its specifications at construction time.

`d = fdesign.bandstop(specvalue1,specvalue2,specvalue3,specvalue4,...specvalue5,specvalue6,specvalue7)` constructs an object `d` with the default `Specification` property string `fpass1,fstop1,fstop2,fpass2,apass1,astop,apass2`, using the values you provide in `specvalue1,specvalue2,specvalue3,specvalue4,specvalue5,specvalue6` and `specvalue7`.

`d = fdesign.bandstop(...,fs)` adds the argument `fs`, specified in Hz to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

`d = fdesign.bandstop(...,magunits)` specifies the units for any magnitude specification you provide in the input arguments. `magunits` can be one of

- `linear` — specify the magnitude in linear units
- `dB` — specify the magnitude in dB (decibels)
- `squared` — specify the magnitude in power units

When you omit the `magunits` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Examples

These examples show how to construct a bandpass filter specification object. First, create a default specifications object without using input arguments.

```
d = fdesign.bandstop
d =

                Response: 'Minimum-order bandstop'
```

```
              Description: {7x1 cell}
            Specification: 'Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2'
        NormalizedFrequency: true
                    Fpass1: 0.3500
                    Fstop1: 0.4500
                    Fstop2: 0.5500
                    Fpass2: 0.6500
                    Apass1: 1
                     Astop: 60
                    Apass2: 1
```

Now create an object by passing a specification type string 'n,fc1,fc2' — the resulting object uses default values for n, fc1, and fc2.

```
d=fdesign.bandstop('n,f3dB1,f3dB2')

d =

                  Response: 'Bandstop with cutoff'
            Specification: 'N,F3dB1,F3dB2'
              Description: {3x1 cell}
        NormalizedFrequency: true
                FilterOrder: 10
                   Fcutoff1: 0.4000
                   Fcutoff2: 0.6000

designmethods(d)

Design Methods for class fdesign.bandstop:

butter
cheby1
cheby2
ellip
```

Create another bandstop filter, passing the specification values to the object rather than accepting the default values for n, f3db1, and fc2.

You can add `fs` as the final input argument to specify the sampling frequency of 48 kHz.

```
d = fdesign.bandstop('n,f3db1,f3db2', 10, 9600, ...
  14400, 48000)

d =

               Response: 'Bandstop with cutoff'
          Specification: 'N,F3dB1,F3dB2'
            Description: {3x1 cell}
    NormalizedFrequency: false
                     Fs: 48000
            FilterOrder: 10
               Fcutoff1: 9600
               Fcutoff2: 14400
```

For this bandstop filter, pass the filter specifications that correspond to the default `Specification` — `fp1`,`fst1`,`fst2`,`fp2`,`ap1`,`ast`,`ap2`.

```
d = fdesign.bandstop(0.3,0.4,0.6,0.7,0.5,60,1)

d =

               Response: 'Minimum-order bandstop'
          Specification: 'Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2'
            Description: {7x1 cell}
    NormalizedFrequency: true
                 Fpass1: 0.3000
                 Fstop1: 0.4000
                 Fstop2: 0.6000
                 Fpass2: 0.7000
                 Apass1: 0.5000
                  Astop: 60
                 Apass2: 1
```

And for the final example, pass the magnitude specifications in squared units, using the `magunits` option `squared`.

```
d = fdesign.bandstop(0.4,0.5,0.6,0.7,0.98,...
0.01,0.99,'squared')
d =

                Response: 'Minimum-order bandstop'
           Specification: 'Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2'
             Description: {7x1 cell}
      NormalizedFrequency: true
                  Fpass1: 0.4000
                  Fstop1: 0.5000
                  Fstop2: 0.6000
                  Fpass2: 0.7000
                  Apass1: 0.0877
                   Astop: 20
                  Apass2: 0.0436
```

See Also

fdesign, fdesign.bandpass, fdesign.highpass, fdesign.lowpass

fdesign.differentiator

Purpose

Differentiator filter specification object

Syntax

```
d = fdesign.differentiator
d = fdesign.differentiator(spec)
d = fdesign.differentiator(spec,specvalue1,specvalue2, ...)
d = fdesign.differentiator(specvalue1)
d = fdesign.differentiator(...,fs)
d = fdesign.differentiator(...,magunits)
```

Description

`d = fdesign.differentiator` constructs a default differentiator filter designer `d` with the filter order set to 31.

`d = fdesign.differentiator(spec)` initializes the filter designer `Specification` property to `spec`. You provide one of the following strings as input to replace `spec`. The string you provide is not case sensitive:

- `n` — full band differentiator (default).
- `n,fp,fst` — partial band differentiator.
- `ap` — minimum-order full band differentiator.
- `fp,fst,ap,ast` — minimum-order partial band differentiator.

The string entries are defined as follows:

- `ap` — amount of ripple allowed in the pass band in decibels (the default units). Also called Apass.
- `ast` — attenuation in the stop band in decibels (the default units). Also called Astop.
- `fp` — frequency at the start of the pass band. Specified in normalized frequency units. Also called Fpass.
- `fst` — frequency at the end of the stop band. Specified in normalized frequency units. Also called Fstop.
- `n` — filter order.

By default, `fdesign.differentiator` assumes that all frequency specifications are provided in normalized frequency units. Also, decibels is the default for all magnitude specifications.

Different specification strings may have different design methods available. Use `designmethods(d)` to get a list of the design methods available for a given specification string.

`d = fdesign.differentiator(spec,specvalue1,specvalue2, ...)` initializes the filter designer specifications in `spec` with `specvalue1`, `specvalue2`, and so on. To get a description of the specifications `specvalue1`, `specvalue2`, and more, enter

```
get(d,'description')
```

at the Command prompt.

`d = fdesign.differentiator(specvalue1)` assumes the default specification string `n`, setting the filter order to the value you provide.

`d = fdesign.differentiator(...,fs)` adds the argument `fs`, specified in Hz to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

`d = fdesign.differentiator(...,magunits)` specifies the units for any magnitude specification you provide in the input arguments. `magunits` can be one of

- `linear` — specify the magnitude in linear units
- `dB` — specify the magnitude in dB (decibels)
- `squared` — specify the magnitude in power units

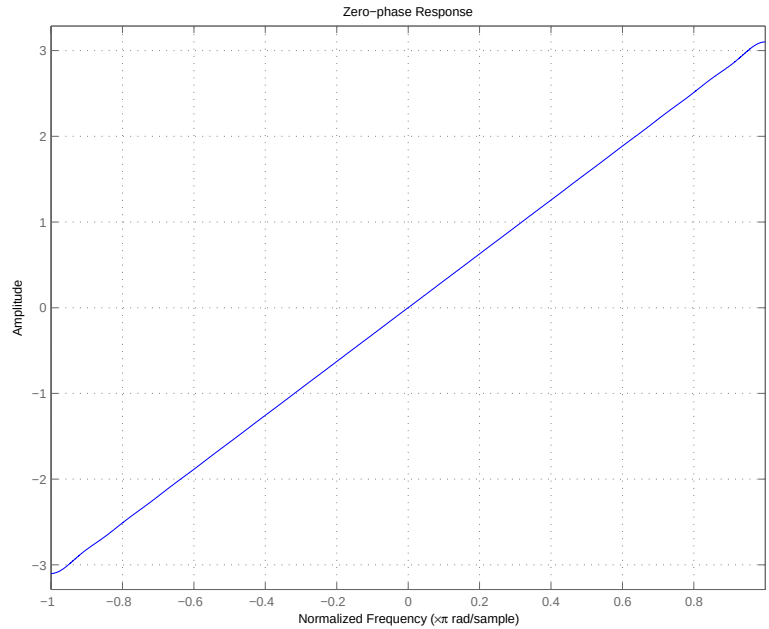
When you omit the `magunits` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Examples

The toolbox lets you design a range of differentiators. These examples present a few possible designs. The first example designs a 33rd-order

full band differentiator. The FVTool plot following the code shows the resulting 33rd-order filter.

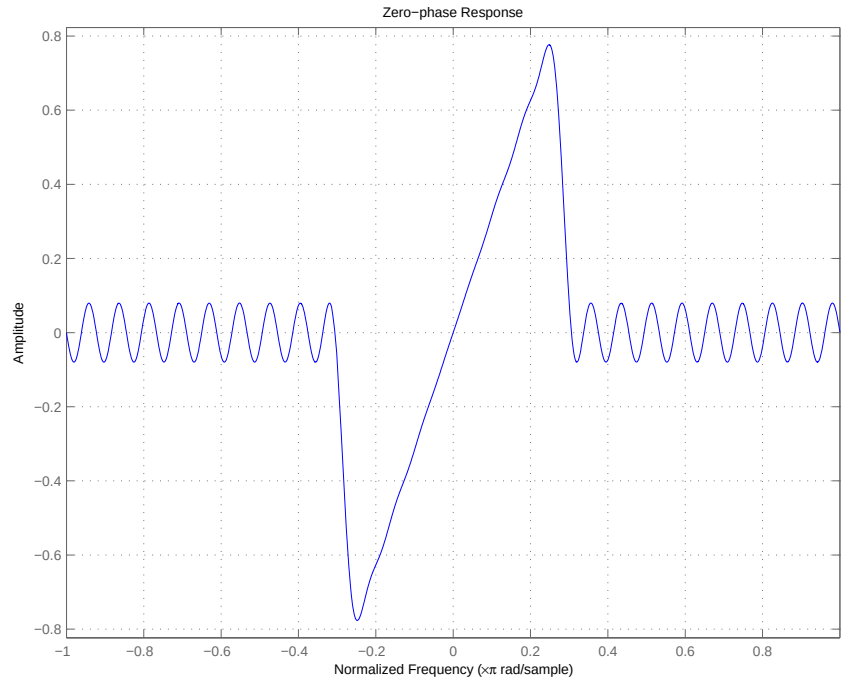
```
d = fdesign.differentiator(33); % Filter order is 33.
designmethods(d);

hd = design(d,'firls');
fvtool(hd,'magnitudedisplay','zero-phase',...
'frequencyrange','[-pi, pi)')

Design Methods for class fdesign.differentiator (N):

equiripple
firls
```



For the second example, design a narrow band differentiator. Differentiate the first 25 percent of the frequencies in the Nyquist range and filter the higher frequencies.

```
d = fdesign.differentiator('n,fp,fst',54,.25,.3);
designmethods(d);
hd = design(d,'equiripple');
fvtool(hd,'magnitudedisplay','zero-phase');
set(hf,'frequencyrange','[-fs/2, fs/2)')
```

Here is the view from FVTool.

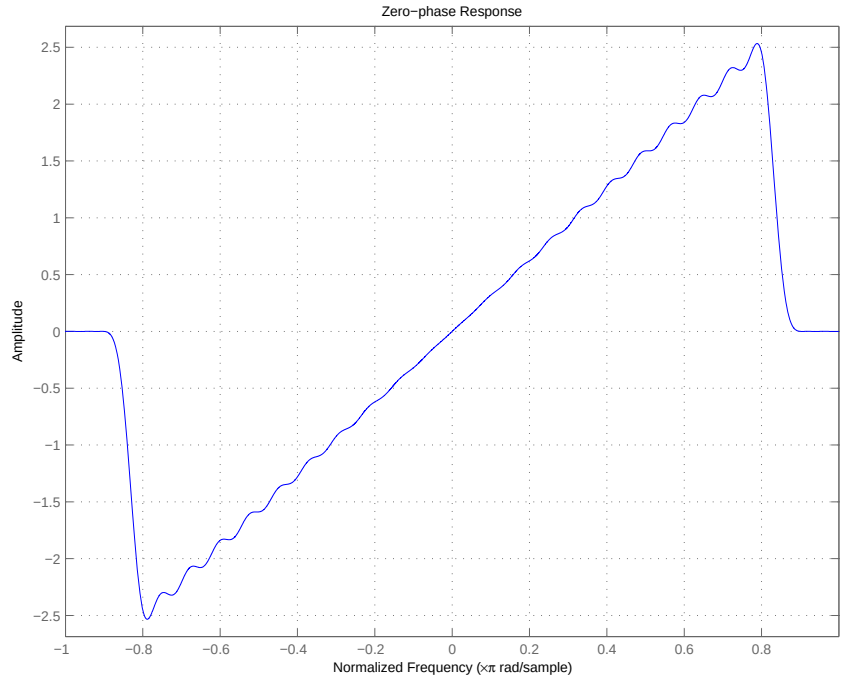


Finally, design a minimum-order, wide-band differentiator.

```
d = fdesign.differentiator('fp,fst,ap,ast',.8,.9,1,80);
```

```
designmethods(d);
hd = design(d,'equiripple');
fvtool(hd,'magnitudedisplay','zero-phase','frequencyrange')
```

FVTool returns this plot.



See Also

`design`, `fdesign`, `setspecs`

Purpose

Highpass filter specification object

Syntax

```
d = fdesign.highpass
d = fdesign.highpass(spec)
d = fdesign.highpass(spec,specvalue1,specvalue2,...)
d = fdesign.highpass(specvalue1,specvalue2,specvalue3,
specvalue4)
d = fdesign.highpass(...,fs)
d = fdesign.highpass(...,magunits)
```

Description

`d = fdesign.highpass` constructs a highpass filter specification object `d`, applying default values for the properties `fst`, `fp`, `ast` and `ap`.

Using `fdesign.highpass` with a design method generates a `dfilt` object.

`d = fdesign.highpass(spec)` constructs object `d` and sets its '`Specification`' to *`spec`*. Entries in the `spec` string represent various filter response features, such as the filter order, that govern the filter design. Valid entries for *`spec`* are shown below. The strings are not case sensitive.

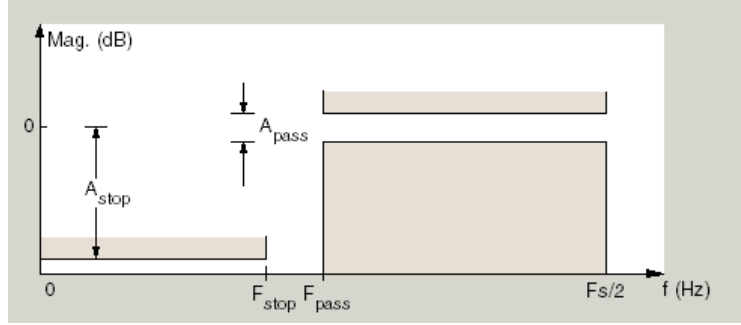
- `fst,fp,ast,ap` (default `spec`)
- `n,f3db`
- `n,f3db,ap`
- `n,f3db,ast`
- `n,f3db,ast,ap`
- `n,f3db,fp`
- `n,fc`
- `n,fc,ast,ap`
- `n,fp,ap`
- `n,fp,ast,ap`
- `n,fst,ast`

- `n,fst,ast,ap`
- `n,fst,f3db`
- `n,fst,fp`
- `n,fst,fp,ap`
- `n,fst,fp,ast`
- `nb,na,fst,fp`

The string entries are defined as follows:

- `ap` — amount of ripple allowed in the pass band in decibels (the default units). Also called Apass.
- `ast` — attenuation in the stop band in decibels (the default units). Also called Astop.
- `f3db` — cutoff frequency for the point 3 dB point below the passband value. Specified in normalized frequency units.
- `fc` — cutoff frequency for the point 3 dB point below the passband value. Specified in normalized frequency units.
- `fp` — frequency at the start of the pass band. Specified in normalized frequency units. Also called Fpass.
- `fst` — frequency at the end of the stop band. Specified in normalized frequency units. Also called Fstop.
- `n` — filter order.
- `na` and `nb` are the order of the denominator and numerator.

Graphically, the filter specifications look similar to those shown in the following figure.



Regions between specification values like `fst1` and `fp` are transition regions where the filter response is not explicitly defined.

The filter design methods that apply to a highpass filter specification object change depending on the `Specification` string. Use `designmethods` to determine which design method applies to an object and its specification string.

`d = fdesign.highpass(spec,specvalue1,specvalue2,...)` constructs an object `d` and sets its specification values at construction time.

`d = fdesign.highpass(specvalue1,specvalue2,specvalue3, specvalue4)` constructs an object `d` with the values for the default `Specification` property string, using the specifications you provide as input arguments specvalue1,specvalue2,specvalue3,specvalue4.

`d = fdesign.highpass(...,fs)` adds the argument `fs`, specified in Hz to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

`d = fdesign.highpass(...,magunits)` specifies the units for any magnitude specification you provide in the input arguments. `magunits` can be one of

- `linear` — specify the magnitude in linear units
- `dB` — specify the magnitude in dB (decibels)
- `squared` — specify the magnitude in power units

When you omit the `magunits` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Examples

These examples show how to construct a highpass filter specification object. First, create a default specifications object without using input arguments.

```
d=fdesign.highpass

d =

               Response: 'Minimum-order highpass'
          Specification: 'Fst,Fp,Ast,Ap'
            Description: {4x1 cell}
    NormalizedFrequency: true
                  Fstop: 0.4500
                  Fpass: 0.5500
                  Astop: 60
                  Apass: 1
```

This time, pass the specifications that correspond to the default `Specification` string.

```
hs = fdesign.highpass(.4,.5,80,1);

hs =

               Response: 'Minimum-order highpass'
          Specification: 'Fst,Fp,Ast,Ap'
            Description: {4x1 cell}
    NormalizedFrequency: true
                  Fstop: 0.4000
                  Fpass: 0.5000
                  Astop: 80
                  Apass: 1
```

Now create an object by passing a specification type string `'n,fc'` — the resulting object uses default values for `n` and `fc`.

```
d=fdesign.highpass('n,fc')

d =

               Response: 'Highpass with cutoff'
          Specification: 'N,Fc'
            Description: {2x1 cell}
    NormalizedFrequency: true
            FilterOrder: 10
                Fcutoff: 0.5000
```

Create the same filter, passing the values for `n` and `fc` rather than accepting the default values. You can add include the sampling frequency `fs` as the final input argument. Adding `fs` puts all the frequency specifications into linear frequency format, rather than normalized frequency.

```
d=fdesign.highpass('n,fc',10,9600,48000)

d =

               Response: 'Highpass with cutoff'
          Specification: 'N,Fc'
            Description: {2x1 cell}
    NormalizedFrequency: false
                     Fs: 48000
            FilterOrder: 10
                Fcutoff: 9600
```

Finally, pass values for the filter specifications that match the default `Specification` string — `fp` = `10`, `fst` = `12`, `ast` = `80` and `ap` = `0.5`. Add the sampling frequency on the end.

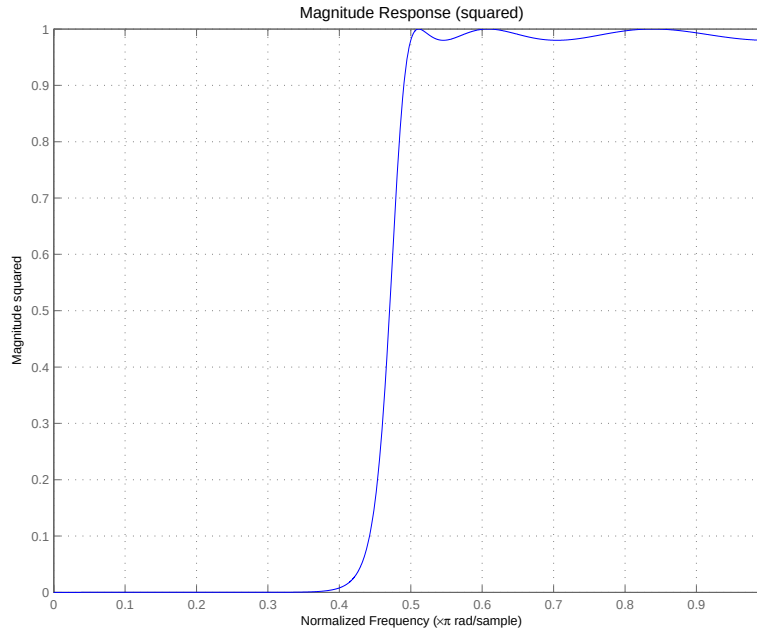
```
d=fdesign.highpass(10,12,80,0.5,48000)
```

```
d =

               Response: 'Minimum-order highpass'
          Specification: 'Fst,Fp,Ast,Ap'
            Description: {4x1 cell}
    NormalizedFrequency: false
                     Fs: 48000
                  Fstop: 10
                  Fpass: 12
                  Astop: 80
```

To demonstrate the `magunits` input option, pass the magnitude specifications in squared units and include the `squared` input argument for `magunits`.

```
hs = fdesign.highpass(.4, .5, .02, .98, 'squared');
hd = cheby1(hs);
fvtool(hd,'MagnitudeDisplay','Magnitude Squared');
```

The following figure shows the filter response.



See Also

`fdesign`, `fdesign.bandpass`, `fdesign.bandstop`, `fdesign.lowpass`

fdesign.hilbert

Purpose

Hilbert filter specification object

Syntax

```
d = fdesign.hilbert
d = fdesign.hilbert(specvalue1,specvalue2)
d = fdesign.hilbert(spec)
d = fdesign.hilbert(spec,specvalue1,specvalue2)
d = fdesign.hilbert(...,fs)
d = fdesign.hilbert(...,magunits)
```

Description

`d = fdesign.hilbert` constructs a default Hilbert filter designer `d` with `n`, the filter order, set to 31.

`d = fdesign.hilbert(specvalue1,specvalue2)` constructs a Hilbert filter designer `d` assuming the default specification string `n,tw`. You input `specvalue1` and `specvalue2` for `n` and `tw`.

`d = fdesign.hilbert(spec)` initializes the filter designer `Specification` property to `spec`. You provide one of the following strings as input to replace `spec`. The string you provide is not case sensitive:

- `n,tw` — default spec string.
- `tw,ap` — minimum-order Hilbert filter.

The string entries are defined as follows:

- `ap` — amount of ripple allowed in the pass band in decibels (the default units). Also called Apass.
- `n` — filter order.
- `tw` — width of the transition region between the pass and stop bands.

By default, `fdesign.hilbert` assumes that all frequency specifications are provided in normalized frequency units. Also, decibels is the default for all magnitude specifications.

Different specification strings may have different design methods available. Use `designmethods(d)` to get a list of the design methods available for a given specification string.

`d = fdesign.hilbert(`*spec*`,specvalue1,specvalue2)` initializes the filter designer specifications in `spec` with `specvalue1`, `specvalue2`, and so on. To get a description of the specifications `specvalue1` and `specvalue2`, enter

```
get(d,'description')
```

at the Command prompt.

`d = fdesign.hilbert(...,fs)` adds the argument `fs`, specified in Hz to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

`d = fdesign.hilbert(...,magunits)` specifies the units for any magnitude specification you provide in the input arguments. `magunits` can be one of

- `linear` — specify the magnitude in linear units
- `dB` — specify the magnitude in dB (decibels)
- `squared` — specify the magnitude in power units

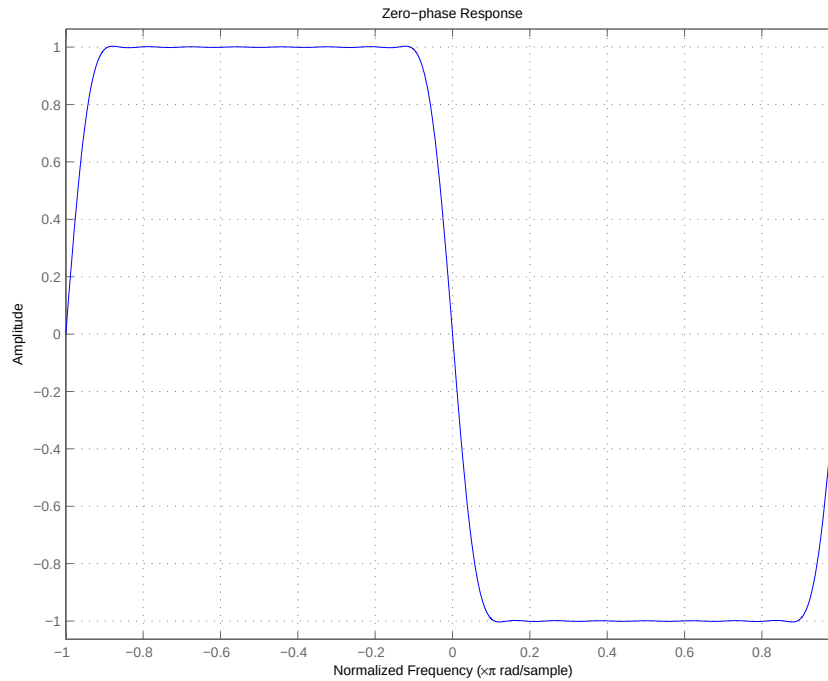
When you omit the `magunits` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Examples

The toolbox lets you design a range of Hilbert filters. These examples present a few possible designs. The first example designs a 30th-order type III Hilbert transformer filter. The FVTool plot following the code shows the resulting filter.

```
d = fdesign.hilbert(30,0.2); % n,tw specification string.
designmethods(d);
```

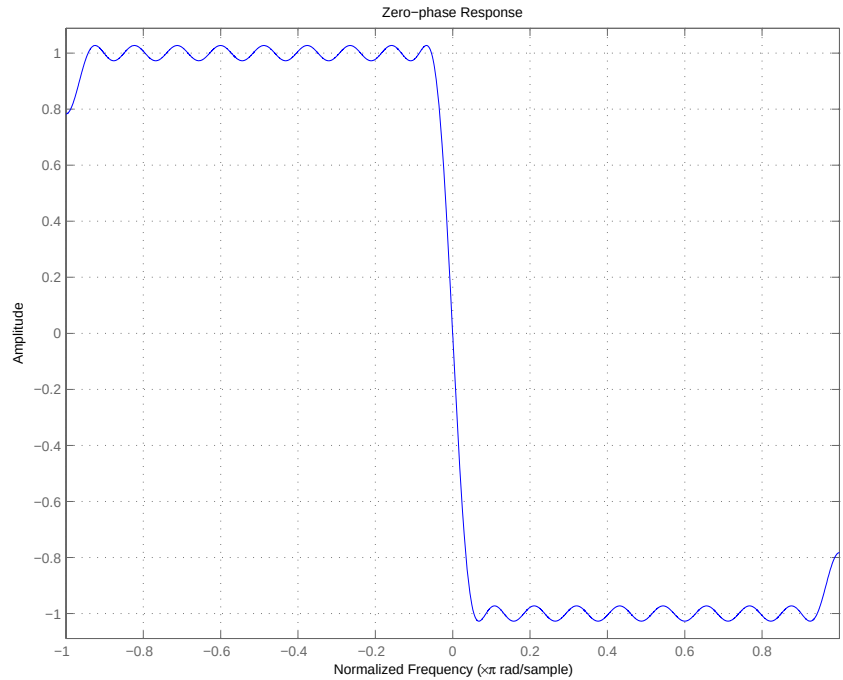
```
hd = design(d,'firls');
fvtool(hd,'magnitudedisplay','zero-phase',...
'frequencyrange','[-pi, pi)')


Design Methods for class fdesign.hilbert (N,TW):

ellip
iirlinphase
equiripple
firls
```



For the second example, design a 35th-order type IV Hilbert transformer.

```
d = fdesign.hilbert('n,tw',35,0.1);
designmethods(d);
hd = design(d,'equiripple');
hf = fvtool(hd,'magnitudedisplay','zero-phase',...
'frequencyrange')
set(hf,'frequencyrange','[-fs/2, fs/2)')
```

Here is the view from FVTool.

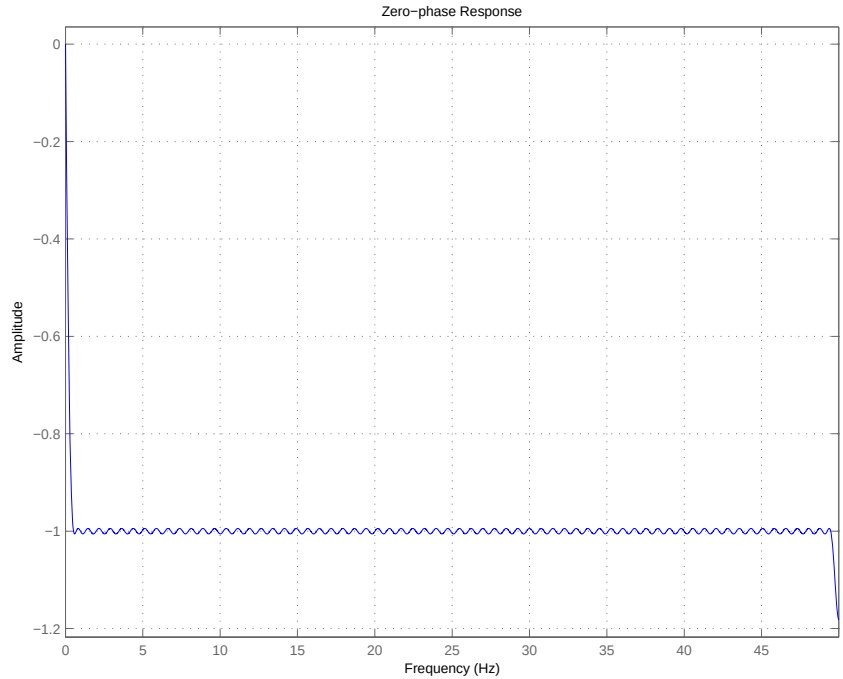


Finally, design a minimum-order transformer that has a sampling frequency of 100 Hz — add `Fs` as an input argument in Hz.

```
d = fdesign.hilbert('tw,ap',1,0.1,100); % Fs = 100 Hz.
designmethods(d);
hd = design(d,'equiripple');
```

```
fvtool(hd,'magnitudedisplay','zero-phase');
set(hf,'frequencyrange','[-fs/2, fs/2)')
```

FVTool returns this plot.



See Also

`design`, `fdesign`, `setspecs`

Purpose

Lowpass filter specification

Syntax

```
d = fdesign.lowpass
d = fdesign.lowpass(spec)
d = fdesign.lowpass(spec,specvalue1,specvalue2,...)
d = fdesign.lowpass(specvalue1,specvalue2,specvalue3,
   specvalue4)
d = fdesign.lowpass(...,fs)
d = fdesign.lowpass(...,magunits)
```

Description

`d = fdesign.lowpass` constructs a lowpass filter specification object `d`, applying default values for the properties `fp`, `fst`, `ap`, and `ast`.

Using the `fdesign.lowpass` specification object with a design method generates a `dfilt` object.

`d = fdesign.lowpass(spec)` constructs object `d` and sets its '`Specification`' property to the string in *`spec`*. Entries in the `spec` string represent various filter response features, such as the filter order, that govern the filter design. Valid entries for *`spec`* are shown below. The strings are not case sensitive.

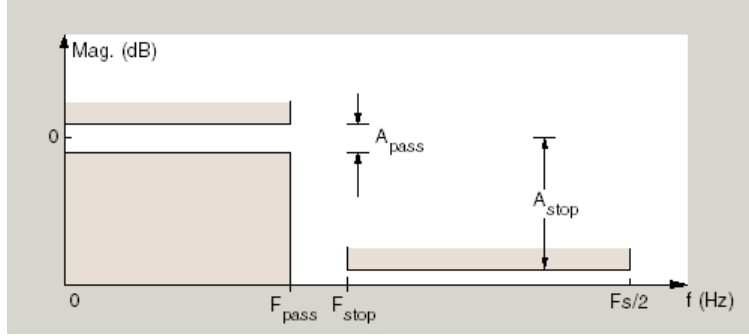
- `fp,fst,ap,ast` (default `spec`)
- `n,f3db`
- `n,f3db,ap`
- `n,f3db,ap,ast`
- `n,f3db,ast`
- `n,f3db,fst`
- `n,fc`
- `n,fc,ap,ast`
- `n,fp,ap`
- `n,fp,ap,ast`
- `n,fp,fst,ap`

- `n,fp,f3db`
- `n,fp,fst`
- `n,fp,fst,ap`
- `n,fp,fst,ast`
- `n,fst,ap,ast`
- `n,fst,ast`
- `nb,na,fp,fst`

The string entries are defined as follows:

- `ap` — amount of ripple allowed in the pass band in decibels (the default units). Also called Apass.
- `ast` — attenuation in the stop band in decibels (the default units). Also called Astop.
- `f3db` — cutoff frequency for the point 3 dB point below the passband value. Specified in normalized frequency units.
- `fc` — cutoff frequency for the point 3 dB point below the passband value. Specified in normalized frequency units.
- `fp` — frequency at the start of the pass band. Specified in normalized frequency units. Also called Fpass.
- `fst` — frequency at the end of the stop band. Specified in normalized frequency units. Also called Fstop.
- `n` — filter order.
- `na` and `nb` are the order of the denominator and numerator.

Graphically, the filter specifications look similar to those shown in the following figure.



Regions between specification values like `fp` and `fst` are transition regions where the filter response is not explicitly defined.

`d = fdesign.lowpass(`*`spec`*`,specvalue1,specvalue2,...)` constructs an object `d` and sets its specification values at construction time using `specvalue1`, `specvalue2`, and so on for all of the specification variables in *`spec`*.

`d = fdesign.lowpass(specvalue1,specvalue2,specvalue3,specvalue4)` constructs an object `d` with values for the default `Specification` property string `fp,fst,ap,ast` using the specifications you provide as input arguments `specvalue1,specvalue2,specvalue3,specvalue4.`

`d = fdesign.lowpass(...,fs)` adds the argument `fs`, specified in Hz to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

`d = fdesign.lowpass(...,magunits)` specifies the units for any magnitude specification you provide in the input arguments. `magunits` can be one of

- `linear` — specify the magnitude in linear units
- `dB` — specify the magnitude in dB (decibels)
- `squared` — specify the magnitude in power units

When you omit the `magunits` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Examples

These examples how to construct a lowpass filter specification object. First, create a default lowpass filter object without using input arguments.

```
d=fdesign.lowpass

d =

               Response: 'Minimum-order lowpass'
          Specification: 'Fp,Fst,Ap,Ast'
            Description: {4x1 cell}
    NormalizedFrequency: true
                  Fpass: 0.4500
                  Fstop: 0.5500
                  Apass: 1
                  Astop: 60
```

Now create an object by passing specifications for the passband and stopband edge frequencies and the passband and stopband attenuations — the resulting object uses the input values for `fp`, `fst`, `ap`, and `ast`.

```
hs = fdesign.lowpass(.4,.5,1,80);
hs

hs =

               Response: 'Minimum-order lowpass'
          Specification: 'Fp,Fst,Ap,Ast'
            Description: {4x1 cell}
    NormalizedFrequency: true
                  Fpass: 0.4000
                  Fstop: 0.5000
```

```
                    Apass: 1
                    Astop: 80
```

Create another filter object, passing the values for `n` and `fc` rather than accepting the default values. You can add include the sampling frequency `fs` as the final input argument.

```
d=fdesign.lowpass('n,fc',10, 9600,48000)

d =

                Response: 'Lowpass with cutoff'
           Specification: 'N,Fc'
             Description: {2x1 cell}
     NormalizedFrequency: false
                      Fs: 48000
             FilterOrder: 10
                 Fcutoff: 9600
```

Finally, pass values for the filter specifications that match the default `Specification` string entries — `fp = 0.4`, `fst = 0.5`, `ast = 80` and `ap = 1.0`. Add the sampling frequency on the end.

```
hs = fdesign.lowpass(.4,.5,1,80)

hs =

                Response: 'Minimum-order lowpass'
           Specification: 'Fp,Fst,Ap,Ast'
             Description: {4x1 cell}
     NormalizedFrequency: true
                   Fpass: 0.4000
                   Fstop: 0.5000
                   Apass: 1
                   Astop: 80
```

Finally, the next examples add the sampling frequency specification in Hz, and then the `magunits` option.

```
hs = fdesign.lowpass('N,Fp,Ap', 10, 9600, .5, 48000);
```

and

```
hsmag = fdesign.lowpass(.4, .5, .98, .02, 'squared');
```

Using the last example filter object, create a highpass filter.

```
hd = design(hsmag,'cheby1');
```

See Also

`fdesign`, `fdesign.bandpass`, `fdesign.bandstop`, `fdesign.highpass`

Purpose

Pulse-shaping filter specification object

Syntax

```
d = fdesign.pulseshaping
d = fdesign.pulseshaping(sps)
d = fdesign.pulseshaping(sps,shape)
d = fdesign.pulseshaping(sps,shape,spec,value1,value2,...)
d = fdesign.pulseshaping(...,fs)
d = fdesign.pulseshaping(...,magunits)
```

Description

`d = fdesign.pulseshaping` constructs a specification object `d`, which can be used to design a minimum-order raised cosine filter object with a stop band attenuation of 60dB and a rolloff factor of 0.25.

```
d =

                Response: 'Pulse Shaping'
               PulseShape: 'Raised Cosine'
         SamplesPerSymbol: 8
            Specification: 'Ast,Beta'
              Description: {'Stopband Attenuation (dB)';'Rolloff Factor'}
      NormalizedFrequency: true
                    Astop: 60
            RolloffFactor: 0.25
```

`d = fdesign.pulseshaping(sps)` constructs a minimum-order raised cosine filter specification object `d` with a positive integer-valued oversampling factor, `SamplesPerSymbol` .

`d = fdesign.pulseshaping(sps,shape)` constructs `d` where `shape` specifies the `PulseShape` property. Valid entries for `shape` are:

- `'Raised Cosine'`
- `'Square Root Raised Cosine'`
- `'Gaussian'`

`d = fdesign.pulseshaping(sps,shape,spec,value1,value2,...)` constructs `d` where `spec` defines the `Specification` properties. The

string entries for `spec` specify various properties of the filter, including the order and frequency response. Valid entries for `spec` depend upon the `shape` property. For `'Raised Cosine'` and `'Square Root Raised Cosine'` filters, the valid entries for `spec` are:

- `'Ast,Beta'` (minimum order; default)
- `'Nsym,Beta'`
- `'N,Beta'`

The string entries are defined as follows:

- `Ast` —stopband attenuation (in dB). The default stopband attenuation for a raised cosine filter is 60 dB. The default stopband attenuation for a square root raised cosine filter is 30 dB. If `Ast` is specified, the minimum-order filter is returned.
- `Beta` —rolloff factor expressed as a real-valued scalar ranging from 0 to 1. Smaller rolloff factors result in steeper transitions between the passband and stopband of the filter.
- `Nsym` —filter order in symbols. The length of the impulse response is given by `Nsym*SamplesPerSymbol+1`. The product `Nsym*SamplesPerSymbol` must be even.
- `N` —filter order (must be even). The length of the impulse response is `N+1`.

If the `shape` property is specified as `'Gaussian'`, the valid entries for `spec` are:

- `'Nsym,BT'` (default)

The string entries are defined as follows:

- `Nsym`—filter order in symbols. `Nsym` defaults to 6. The length of the filter impulse response is `Nsym*SamplesPerSymbol+1`. The product `Nsym*SamplesPerSymbol` must be even.

- `BT` —the 3–dB bandwidth-symbol time product. `BT` is a positive real-valued scalar, which defaults to 0.3. Larger values of `BT` produce a narrower pulse width in time with poorer concentration of energy in the frequency domain.

`d = fdesign.pulseshaping(...,fs)` specifies the sampling frequency of the signal to be filtered. `fs` must be specified as a scalar trailing the other numerical values provided. For this case, `fs` is assumed to be in Hz and is used for analysis and visualization.

`d = fdesign.pulseshaping(...,magunits)` specifies the units for any magnitude specification you provide in the input arguments. Valid entries for `magunits` are:

- `linear` — specify the magnitude in linear units
- `dB` — specify the magnitude in dB (decibels)
- `squared` — specify the magnitude in power units

When you omit the `magunits` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

After creating the specification object `d`, you can use the `design` function to create a filter object such as `h` in the following example:

```
d = fdesign.pulseshaping(8,'Raised Cosine','Nsym,Beta',6,0.25);
h = design(d);
```

Normally, the `Specification` property of the specification object also determines which design methods you can use when you create the filter object. Currently, regardless of the `Specification` property, the `design` function uses the `window` design method with all `fdesign.pulseshaping` specification objects. The `window` method creates an FIR filter with a windowed impulse response.

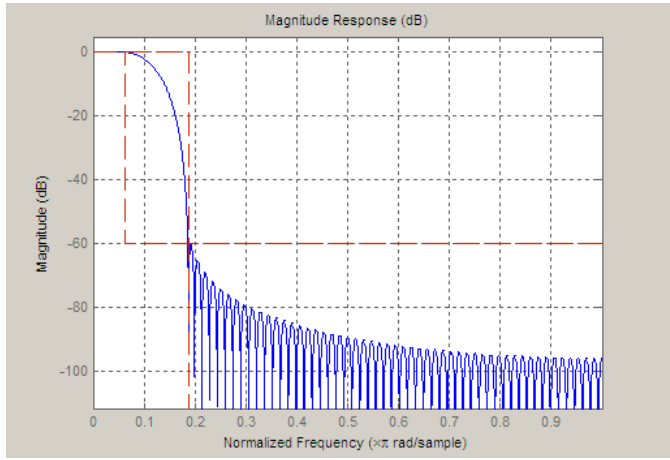
Examples

Pulse-shaping can be used to change the waveform of transmitted pulses so the signal bandwidth matches that of the communication channel. This helps to reduce distortion and intersymbol interference (ISI).

This example shows how to design a minimum-order raised cosine filter that provides a stop band attenuation of 60 dB, rolloff factor of 0.50, and 8 samples per symbol.

```
h  = fdesign.pulseshaping(8,'Raised Cosine','Ast,Beta',60,0.50);
    Hd = design(h);
    fvtool(Hd)
```

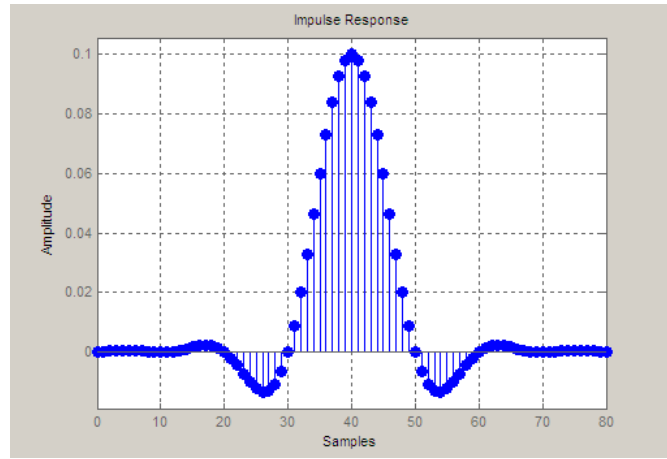
This code generates the following figure.



This example shows how to design a raised cosine filter that spans 8 symbol durations (i.e., of order 8 symbols), has a rolloff factor of 0.50, and oversampling factor of 10.

```
h  = fdesign.pulseshaping(10,'Raised Cosine','Nsym,Beta',8,0.50);
    Hd = design(h);
    fvtool(Hd, 'impulse')
```

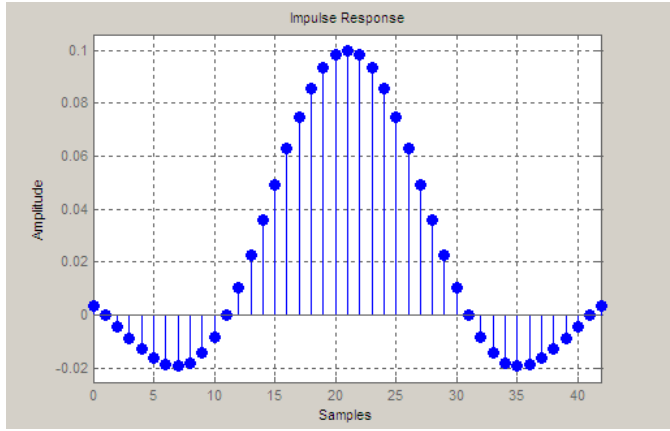
This code generates the following figure.



This example shows how to design a square root raised cosine filter of order 42, rolloff factor of 0.25, and 10 samples per symbol.

```
h  = fdesign.pulseshaping(10,'Square Root Raised Cosine','N,Beta',42);
    Hd = design(h);
    fvtool(Hd, 'impulse')
```

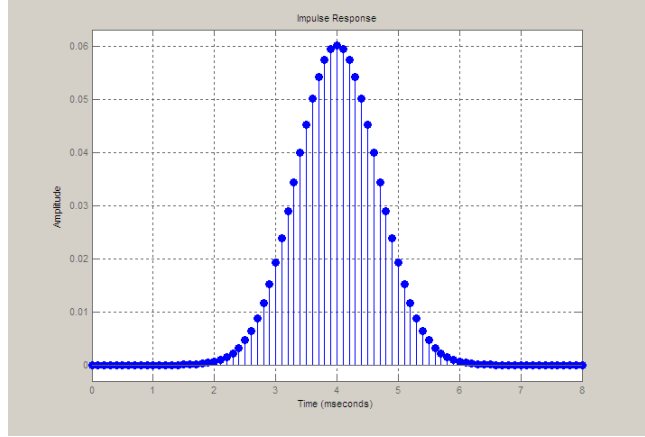
This code generates the following figure.



The following example demonstrates how to create a Gaussian pulse-shaping filter with an oversampling factor (`sps`) of 10, a bandwidth-time symbol product of 0.2, and 8 symbol periods. The sampling frequency is specified as 10 kHz.

```
d=fdesign.pulseshaping(10,'gaussian','nsym,bt',8,0.2,10000);
Hd=design(d); %note the length of d.Numerator is 8*10+1
fvtool(Hd,'impulse')
```

The above code generates the following figure. Note the time axis in milliseconds.



For more information, see the “Pulse Shaping Filter Design” demo, pulseshapingfilterdemo

Purpose

FFT-based FIR filtering using overlap-add method

Syntax

```
y = fftfilt(b,x)
y = fftfilt(b,x,n)
```

Description

`fftfilt` filters data using the efficient FFT-based method of *overlap-add*, a frequency domain filtering technique that works only for FIR filters.

`y = fftfilt(b,x)` filters the data in vector `x` with the filter described by coefficient vector `b`. It returns the data vector `y`. The operation performed by `fftfilt` is described in the *time domain* by the difference equation:

$$y(n) = b(1)x(n) + b(2)x(n-1) + \cdots + b(nb+1)x(n-nb)$$

An equivalent representation is the *z*-transform or *frequency domain* description:

$$Y(z) = (b(1) + b(2)z^{-1} + \cdots + b(nb+1)z^{-nb})X(z)$$

By default, `fftfilt` chooses an FFT length and data block length that guarantee efficient execution time.

If `x` is a matrix, `fftfilt` filters its columns. If `b` is a matrix, `fftfilt` applies the filter in each column of `b` to the signal vector `x`. If `b` and `x` are both matrices with the same number of columns, the i-th column of `b` is used to filter the i-th column of `x`.

`y = fftfilt(b,x,n)` uses n to determine the length of the FFT. See "Algorithm" on page 10-335 for information.

`fftfilt` works for both real and complex inputs.

Comparison to filter function

When the input signal is relatively large, it is advantageous to use `fftfilt` instead of `filter`, which performs N multiplications for each sample in `x`, where N is the filter length. `fftfilt` performs 2 FFT

operations — the FFT of the signal block of length L plus the inverse FT of the product of the FFTs — at the cost of

1/2*L*log2(L)

where L is the block length. It then performs L pointwise multiplications for a total cost of

L+L*log2(L) = L*(1+log2(L))

multiplications. The cost ratio is therefore

L*(1+log2(L))/(N*L) = (1+log2(L))/N

which is approximately log2(L)/N.

Therefore, `fftfilt` becomes advantageous when log2(L) is less than N.

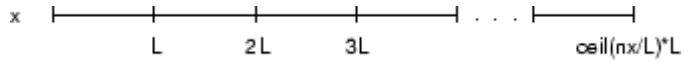
Examples

Show that the results from `fftfilt` and `filter` are identical:

```
b = [1 2 3 4];
x = [1 zeros(1,99)]';
norm(fftfilt(b,x) - filter(b,1,x))

ans =
    9.5914e-15
```

Algorithm

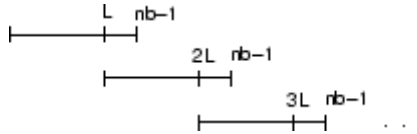
`fftfilt` uses `fft` to implement the *overlap-add method* [1], a technique that combines successive frequency domain filtered blocks of an input sequence. `fftfilt` breaks an input sequence `x` into length `L` data blocks, where L must be greater than the filter length N.



and convolves each block with the filter `b` by

```
y = ifft(fft(x(i:i+L-1),nfft).*fft(b,nfft));
```

where `nfft` is the FFT length. `fftfilt` overlaps successive output sections by `n-1` points, where `n` is the length of the filter, and sums them.



`fftfilt` chooses the key parameters `L` and `nfft` in different ways, depending on whether you supply an FFT length `n` and on the lengths of the filter and signal. If you do not specify a value for `n` (which determines FFT length), `fftfilt` chooses these key parameters automatically:

- If `length(x)` is greater than `length(b)`, `fftfilt` chooses values that minimize the number of blocks times the number of flops per FFT.
- If `length(b)` is greater than or equal to `length(x)`, `fftfilt` uses a single FFT of length

  ```
  2^nextpow2(length(b) + length(x) - 1)
  ```

  This essentially computes

  ```
  y = ifft(fft(B,nfft).*fft(X,nfft))
  ```

If you supply a value for `n`, `fftfilt` chooses an FFT length, `nfft`, of `2^nextpow2(n)` and a data block length of `nfft - length(b) + 1`. If `n` is less than `length(b)`, `fftfilt` sets `n` to `length(b)`.

References

[1] Oppenheim, A.V., and R.W. Schafer. *Discrete-Time Signal Processing,* Prentice-Hall, 1989.

See Also

`conv`, `dfilt.fftfir`, `filter`, `filtfilt`

Purpose

Filter data with recursive (IIR) or nonrecursive (FIR) filter

Description

`filter` is a MATLAB function.

Signal-Specific Information

Filter Method of DFILT

Filter is also an overloaded method of the discrete-time filter object (`dfilt`). You can pass an object handle, data, and optionally, the dimension into the filter method.

The MATLAB `filter` function describes a `zi` input for initial conditions. Note that the recommended way of passing initial conditions into a `dfilt` is by using the `states` property. For more information, see the `dfilt` reference page.

Filter Normalization

Using the `filter` function on `b` and `a` coefficients normalizes the filter by forcing the a_0 coefficient to be equal to 1.

Using the `filter` method on a `dfilt` object does not normalize the a_0 coefficient.

FIR Filters

The denominator of FIR filters is, by definition, equal to 1. To use the `filter` function with the `b` coefficients from an FIR function, use `y = filter(b,1,x)` .

filterbuilder

Purpose

GUI-based filter design

Syntax

```
filterbuilder(h)
filterbuilder('response')
```

Description

`filterbuilder` starts a GUI-based tool for building filters. It relies on the `fdesign` object-object oriented filter design paradigm, and is intended to reduce development time during the filter design process. `filterbuilder` uses a specification-centered approach to find the best algorithm for the desired response.

Note You must have the Signal Processing Toolbox installed to use `fdesign` and `filterbuilder`. Some of the features described below may be unavailable if your installation does not additionally include the Filter Design Toolbox. You can verify the presence of both toolboxes by typing `ver` at the command prompt.

The `filterbuilder` GUI contains many features not available in FDATool. For more information on how to use `filterbuilder`, see Chapter 4, "Designing a Filter in the Filterbuilder GUI".

To use `filterbuilder`, enter `filterbuilder` at the MATLAB command line using one of three approaches:

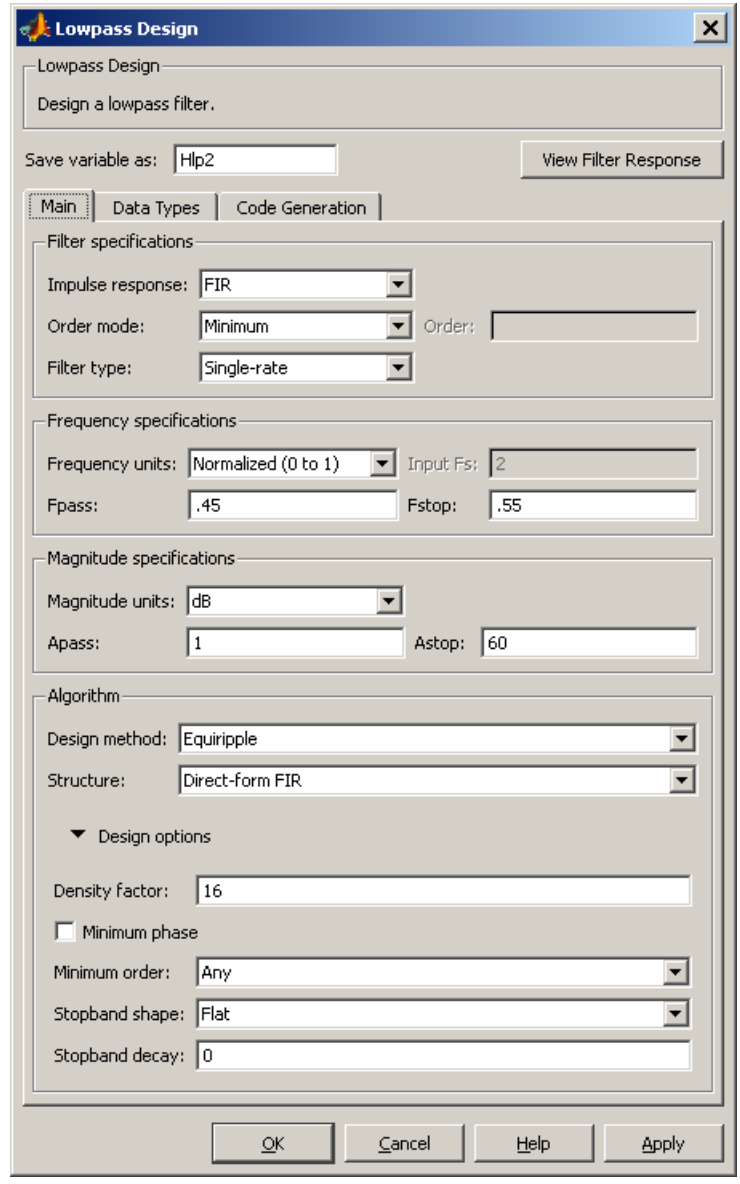
- Simply enter `filterbuilder`. MATLAB opens a dialog for you to select a filter response type and then launches the filter design dialog box.
- Enter `filterbuilder(h)`, where h is an existing filter object. For example, if `h` is a bandpass filter, `filterbuilder(h)` opens the bandpass filter design dialog box. (The `h` object must have been created using `filterbuilder` or must be a `dfilt` or `mfilt` object.)
- Enter `filterbuilder('response')`, replacing *`response`* with a response string from the following table. MATLAB opens a filter design dialog that corresponds to the response string.

Response String	Description of Resulting Filter Design
`arbmag`	Arbitrary magnitude and phase filter
`bandpass` or `bp`	Bandpass filter
`bandstop` or `bs`	Bandstop filter
`cic`	CIC filter
`ciccomp`	CIC compensator
`diff`	Differentiator filter
`fracdelay`	Fractional delay filter
`halfband` or `hb`	Halfband filter
`highpass` or `hp`	Highpass filter
`hilb`	Hilbert filter
`isinclp`	Inverse sinc lowpass filter
`lowpass` or `lp`	Lowpass filter (default)
`notch`	Notch filter
`nyquist`	Nyquist filter
`octave`	Octave filter
`parameq`	Parametric equalizer filter
`peak`	Peak filter
`pulseshaping`	Pulse-shaping filter

Note Because they do not change the filter structure, the magnitude specifications and design method are tunable when using `filterbuilder`.

Filterbuilder Dialog Box

Although the main pane of the filterbuilder dialog box varies depending on the filter response type, the basic structure is the same. The following figure shows the basic layout of the dialog box.


Lowpass Design
Lowpass Design
Design a lowpass filter.
Save variable as:
Hlp2
View Filter Response
Main
Data Types
Code Generation
Filter specifications
Impulse response:
FIR
Order mode:
Minimum
Order:
Filter type:
Single-rate
Frequency specifications
Frequency units:
Normalized (0 to 1)
Input Fs:
2
Fpass:
.45
Fstop:
.55
Magnitude specifications
Magnitude units:
dB
Apass:
1
Astop:
60
Algorithm
Design method:
Equiripple
Structure:
Direct-form FIR
Design options
Density factor:
16
Minimum phase
Minimum order:
Any
Stopband shape:
Flat
Stopband decay:
0
OK
Cancel
Help
Apply

As you choose the response for the filter, the available options and design parameters displayed in the dialog box change. This display allows you to focus only on parameters that make sense in the context of your filter design.

Every filter design dialog box includes the options displayed at the top of the dialog box, shown in the following figure.

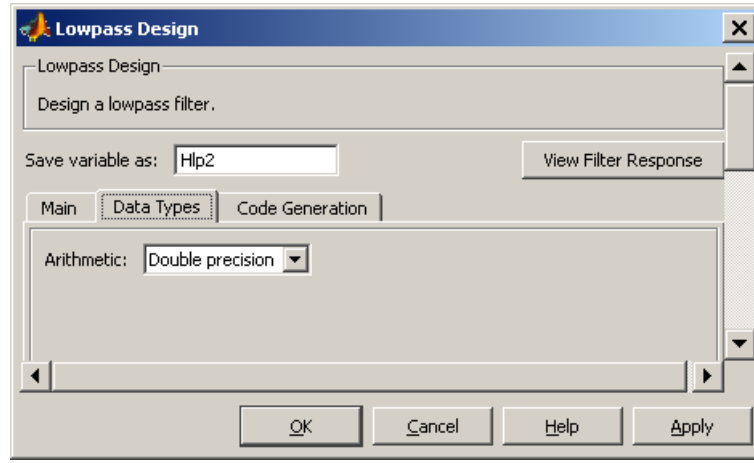
- **Save variable as** — When you click **Apply** to apply your changes or **OK** to close this dialog box, `filterbuilder` saves the current filter to your MATLAB workspace as a filter object with the name you enter.
- **View Filter Response** — Displays the magnitude response for the current filter specifications and design method by opening the Filter Visualization Tool (`fvtool`) from Signal Processing Toolbox software. For more information about FVTool, refer to Signal Processing Toolbox documentation.

Note The `filterbuilder` dialog box includes an **Apply** option. Each time you click **Apply**, `filterbuilder` writes the modified filter to your MATLAB workspace. This modified filter has the variable name you assign in **Save variable as**. To apply changes without overwriting the variable in you workspace, change the variable name in **Save variable as** before you click **Apply**.

There are three tabs in the Filterbuilder dialog box, containing three panes: **Main**, **Data Types**, and **Code Generation**. The first pane changes according to the filter being designed. The last two panes are the same for all filters. These panes are discussed in the following sections.

Data Types Pane

The second tab in the Filterbuilder dialog box is shown in the following figure.

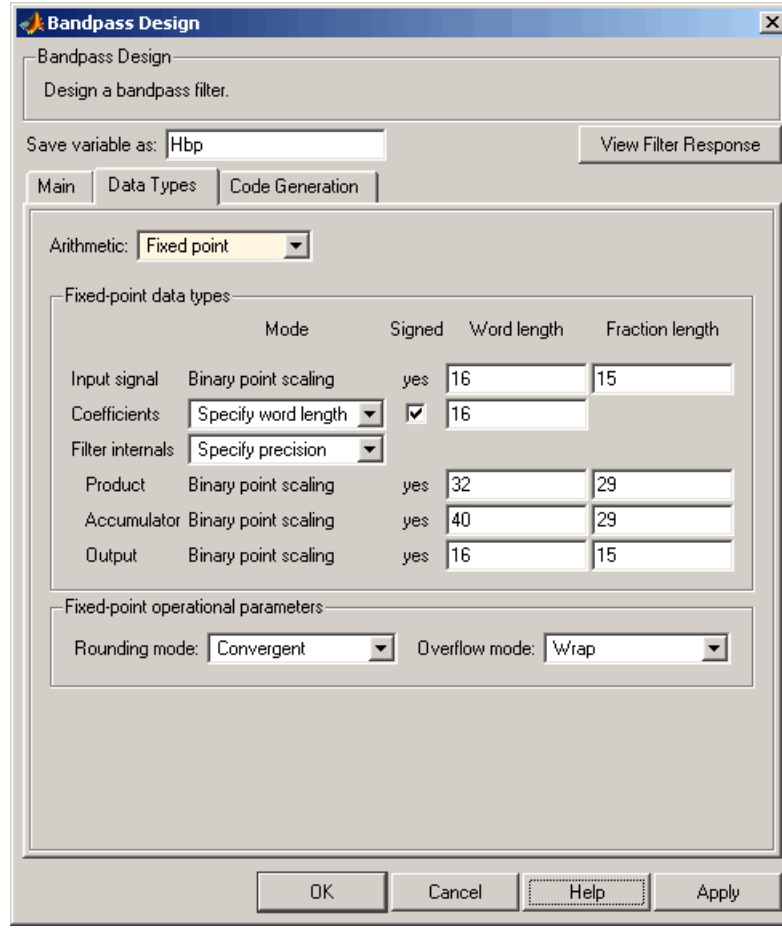


The **Arithmetic** drop down box allows the choice of `Double precision`, `Single precision`, or `Fixed point`. Some of these options may be unavailable depending on the filter parameters. The following table describes these options.

Arithmetic List Entry	Effect on the Filter
`Double precision`	All filtering operations and coefficients use double-precision, floating-point representations and math. When you use `filterbuilder` to create a filter, `double precision` is the default value for the Arithmetic property.

Arithmetic List Entry	Effect on the Filter
`Single-precision`	All filtering operations and coefficients use single-precision floating-point representations and math.
`Fixed point`	This string applies selected default values, typically used on many digital processors, for the properties in the fixed-point filter. These properties include coefficient word lengths, fraction lengths, and various operating modes. This setting allows signed fixed data types only. Fixed-point filter design with `filterbuilder` is available only when you install Fixed-Point Toolbox software along with Filter Design Toolbox software.

The following figure shows the **Data Types** pane after you select `Fixed point` for **Arithmetic**.



Not all parameters described in the following section apply to all filters. For example, FIR filters do not have the **Section input** and **Section output** parameters.

Input signal

Specify the format the filter applies to data to be filtered. For all cases, `filterbuilder` implements filters that use binary point

scaling and signed input. You set the word length and fraction length as needed.

Coefficients

Choose how you specify the word length and the fraction length of the filter numerator and denominator coefficients:

- `Specify word length` enables you to enter the word length of the coefficients in bits. In this mode, `filterbuilder` automatically sets the fraction length of the coefficients to the binary-point only scaling that provides the best possible precision for the value and word length of the coefficients.
- `Binary point scaling` enables you to enter the word length and the fraction length of the coefficients in bits. If applicable, enter separate fraction lengths for the numerator and denominator coefficients.
- The filter coefficients do not obey the **Rounding mode** and **Overflow mode** parameters that are available when you select `Specify precision` from the Filter internals list. Coefficients are always saturated and rounded to `Nearest`.

Section Input

Choose how you specify the word length and the fraction length of the fixed-point data type going into each section of an SOS filter. This parameter is visible only when the selected filter structure is IIR and SOS.

- `Binary point scaling` enables you to enter the word and fraction lengths of the section input in bits.
- `Specify word length` enables you to enter the word lengths in bits.

Section Output

Choose how you specify the word length and the fraction length of the fixed-point data type coming out of each section of an SOS filter. This parameter is visible only when the selected filter structure is IIR and SOS.

- `Binary point scaling` enables you to enter the word and fraction lengths of the section output in bits.
- `Specify word length` enables you to enter the output word lengths in bits.

State

Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. Use this parameter to specify how to designate the state word and fraction lengths. This parameter is not visible for direct form and direct form I filter structures because `filterbuilder` deduces the state directly from the input format. States always use signed representation:

- `Binary point scaling` enables you to enter the word length and the fraction length of the accumulator in bits.
- `Specify precision` enables you to enter the word length and fraction length in bits (if available).

Product

Determines how the filter handles the output of product operations. Choose from the following options:

- `Full precision` — Maintain full precision in the result.
- `Keep LSB` — Keep the least significant bit in the result when you need to shorten the data words.
- `Specify Precision` — Enables you to set the precision (the fraction length) used by the output from the multiplies.

Filter internals

Specify how the fixed-point filter performs arithmetic operations within the filter. The affected filter portions are filter products, sums, states, and output. Select one of these options:

- `Full precision` — Specifies that the filter maintains full precision in all calculations for products, output, and in the accumulator.

- `Specify precision` — Set the word and fraction lengths applied to the results of product operations, the filter output, and the accumulator. Selecting this option enables the word and fraction length controls.

Signed

Selecting this option directs the filter to use signed representations for the filter coefficients.

Word length

Sets the word length for the associated filter parameter in bits.

Fraction length

Sets the fraction length for the associate filter parameter in bits.

Accum

Use this parameter to specify how you would like to designate the accumulator word and fraction lengths.

Determines how the accumulator outputs stored values. Choose from the following options:

- `Full precision` — Maintain full precision in the accumulator.
- `Keep MSB` — Keep the most significant bit in the accumulator.
- `Keep LSB` — Keep the least significant bit in the accumulator when you need to shorten the data words.
- `Specify Precision` — Enables you to set the precision (the fraction length) used by the accumulator.

Output

Sets the mode the filter uses to scale the output data after filtering. You have the following choices:

- `Avoid Overflow` — Set the output data fraction length to avoid causing the data to overflow. `Avoid overflow` is considered the conservative setting because it is independent of the input data values and range.

- `Best Precision` — Set the output data fraction length to maximize the precision in the output data.
- `Specify Precision` — Set the fraction length used by the filtered data.

Fixed-point operational parameters

Parameters in this group control how the filter rounds fixed-point values and how it treats values that overflow.

Rounding mode

Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths).

- `ceil` - Round toward positive infinity.
- `convergent` - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.
- `zero/fix` - Round toward zero.
- `floor` - Round toward negative infinity.
- `nearest` - Round toward nearest. Ties round toward positive infinity.
- `round` - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.

The choice you make affects everything except coefficient values and input data which always round. In most cases, products do not overflow—they maintain full precision.

Overflow mode

Sets the mode the filter uses to respond to overflow conditions in fixed-point arithmetic. Choose from the following options:

- `Saturate` — Limit the output to the largest positive or negative representable value.

- `Wrap` — Set overflowing values to the nearest representable value using modular arithmetic.

The choice you make affects everything except coefficient values and input data which always round. In most cases, products do not overflow—they maintain full precision.

Cast before sum

Specifies whether to cast numeric data to the appropriate accumulator format before performing sum operations. Selecting **Cast before sum** ensures that the results of the affected sum operations match most closely the results found on most digital signal processors. Performing the cast operation before the summation adds one or two additional quantization operations that can add error sources to your filter results.

If you clear **Cast before sum**, the filter prevents the addends from being cast to the sum format before the addition operation. Choose this setting to get the most accurate results from summations without considering the hardware your filter might use. The input format referenced by **Cast before sum** depends on the filter structure you are using.

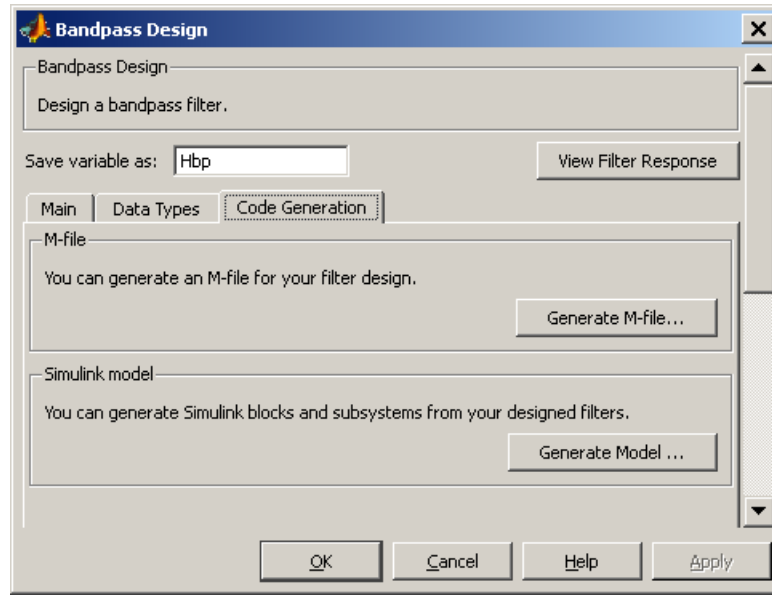
The effect of clearing or selecting **Cast before sum** is as follows:

- Cleared — Configures filter summation operations to retain the addends in the format carried from the previous operation.
- Selected — Configures filter summation operations to convert the input format of the addends to match the summation output format before performing the summation operation. Usually, selecting **Cast before sum** generates results from the summation that more closely match those found from digital signal processors.

Code Generation Pane

The code generation pane contains options for various implementations of the completed filter design. You can generate VHDL and Verilog

code from the designed filter. You can generated M-Code. You can also choose to create or update a Simulink model from the designed filter. The following section explains these options.



HDL

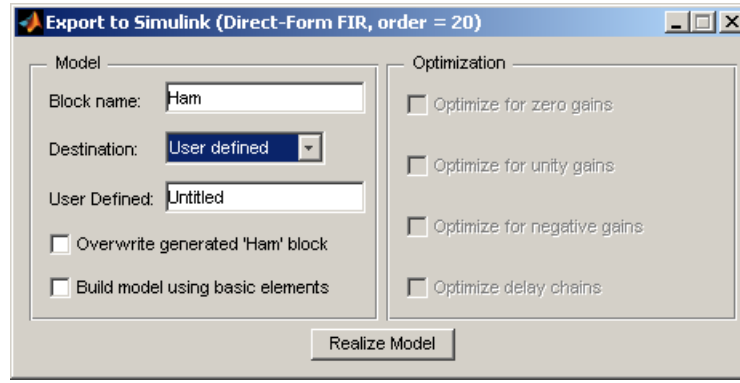
For more information on this option, see "Opening the Generate HDL Dialog Box from the filterbuilder GUI" documentation, where all the parameters on the sub dialog box are explained in detail.

M-Code

Clicking on the **Generate M-Code** button, brings up a Save File dialog. Specify the file name and location, and save. The filter is now contained in an editable M-file.

Simulink Model

Clicking on the **Generate Model** button brings up the **Export to Simulink** dialog box, as shown in the following figure.



You can set the following parameters in this dialog box:

- **Block Name** — The name for the new subsystem block, set to **Filter** by default.
- **Destination** — **Current** saves the generated model to the current Simulink model; **New** creates a new model to contain the generated block; **User Defined** creates a new model or subsystem to the user-specified location enumerated in the **User Defined** text box.
- **Overwrite generated 'Filter' block** — When this check box is selected, Filter Design Toolbox software overwrites an existing block with the name specified in **Block Name**; when cleared, creates a new block with the same name.
- **Build model using basic elements** — When this check box is selected, Filter Design Toolbox software builds the model using only basic blocks.
- **Optimize for zero gains** — When this check box is selected, Filter Design Toolbox software removes all zero gain blocks from the model.
- **Optimize for unity gains** — When this check box is selected, Filter Design Toolbox software replaces all unity gains with direct connections.

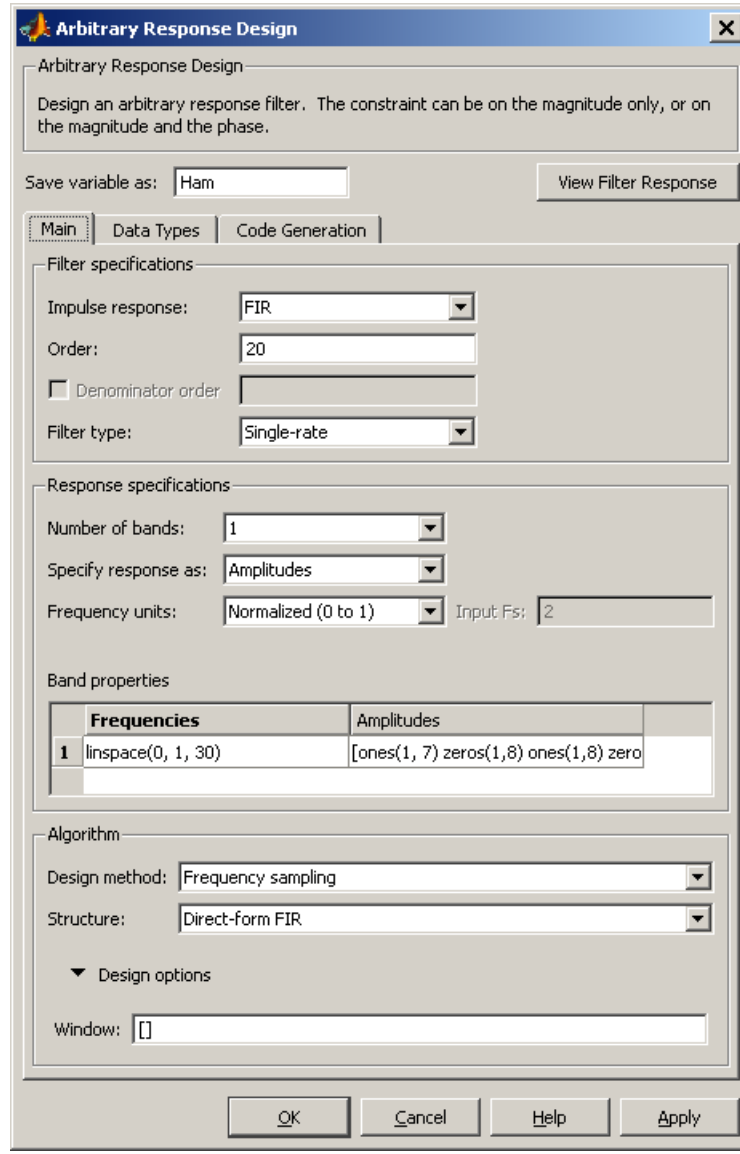
- **Optimize for negative gains** — When this check box is selected, Filter Design Toolbox software removes all negative unity gain blocks, and changes sign at the nearest summation block.
- **Optimize delay chains** — When this check box is selected, Filter Design Toolbox software replaces cascaded delay blocks with a single integer delay block with an equivalent total delay.
- **Realize Model** — Filter Design Toolbox software builds the model with the set parameters.

Main Pane

Most of this pane contains parameters specific to the filter type. These are described in detail in the following sections:

- "Arbitrary Response Design Dialog Box — Main Pane" on page 10-355
- "Bandpass Filter Design Dialog Box — Main Pane" on page 10-359
- "Bandstop Filter Design Dialog Box — Main Pane" on page 10-367
- "CIC Filter Design Dialog Box — Main Pane" on page 10-375
- "CIC Compensator Filter Design Dialog Box — Main Pane" on page 10-378
- "Differentiator Filter Design Dialog Box — Main Pane" on page 10-384
- "Fractional Delay Filter Design Dialog Box — Main Pane" on page 10-391
- "Halfband Filter Design Dialog Box — Main Pane" on page 10-393
- "Highpass Filter Design Dialog Box — Main Pane" on page 10-400
- "Hilbert Filter Design Dialog Box — Main Pane" on page 10-408
- "Inverse Sinc Filter Design Dialog Box — Main Pane" on page 10-414
- "Lowpass Filter Design Dialog Box — Main Pane" on page 10-422
- "Nyquist Filter Design Dialog Box — Main Pane" on page 10-430

Arbitrary Response Design Dialog Box — Main Pane



Filter Specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Impulse response

Select either `FIR` or `IIR` from the drop down list, where `FIR` is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

Order

Enter the order for FIR filter, or the order of the numerator for the IIR filter.

Denominator order

Select the check box and enter the denominator order. This option is enabled only if `IIR` is selected for **Impulse response**.

Filter type

This option is available for FIR filters only. Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, `filterbuilder` specifies single-rate filters.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting `Sample-rate converter` activates both factors.

When you design either a decimator or interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default factor value is 2.

Response Specification

Number of Bands

Select the number of bands in the filter. Multiband design is available for both FIR and IIR filters.

Specify response as:

Specify the response as `Amplitudes`, `Magnitudes and phase`, or `Frequency response`.

Frequency units

Specify frequency units as either `Normalized`, which means normalized by the input sampling frequency, or select from `Hz`, `kHz`, `MHz`, or `GHz`.

Input Fs

Enter the input sampling frequency in the units specified in the **Frequency units** drop-down box. This option is enabled when the frequency units are selected.

Band Properties

These properties are modified automatically depending on the response chosen in the **Specify response as** drop-down box. Two or three columns are presented for input. The first column is always Frequencies. The other columns are either Amplitudes, Magnitudes, Phases, or Frequency Response. Enter the corresponding vectors of values for each column.

- **Frequencies** and **Amplitudes** — These columns are presented for input if the response chosen in the **Specify response as** drop-down box is `Amplitudes`.
- **Frequencies**, **Magnitudes**, and **Phases** — These columns are presented for input if the response chosen in the **Specify response as** drop-down box is `Magnitudes and phases`.

- **Frequencies** and **Frequency response** —These columns are presented for input if the response chosen in the **Specify response as** drop-down box is `Frequency response`.

Algorithm

Design Method

Select the design method for the filter. Different methods are enabled depending on the defining parameters entered in the previous sections.

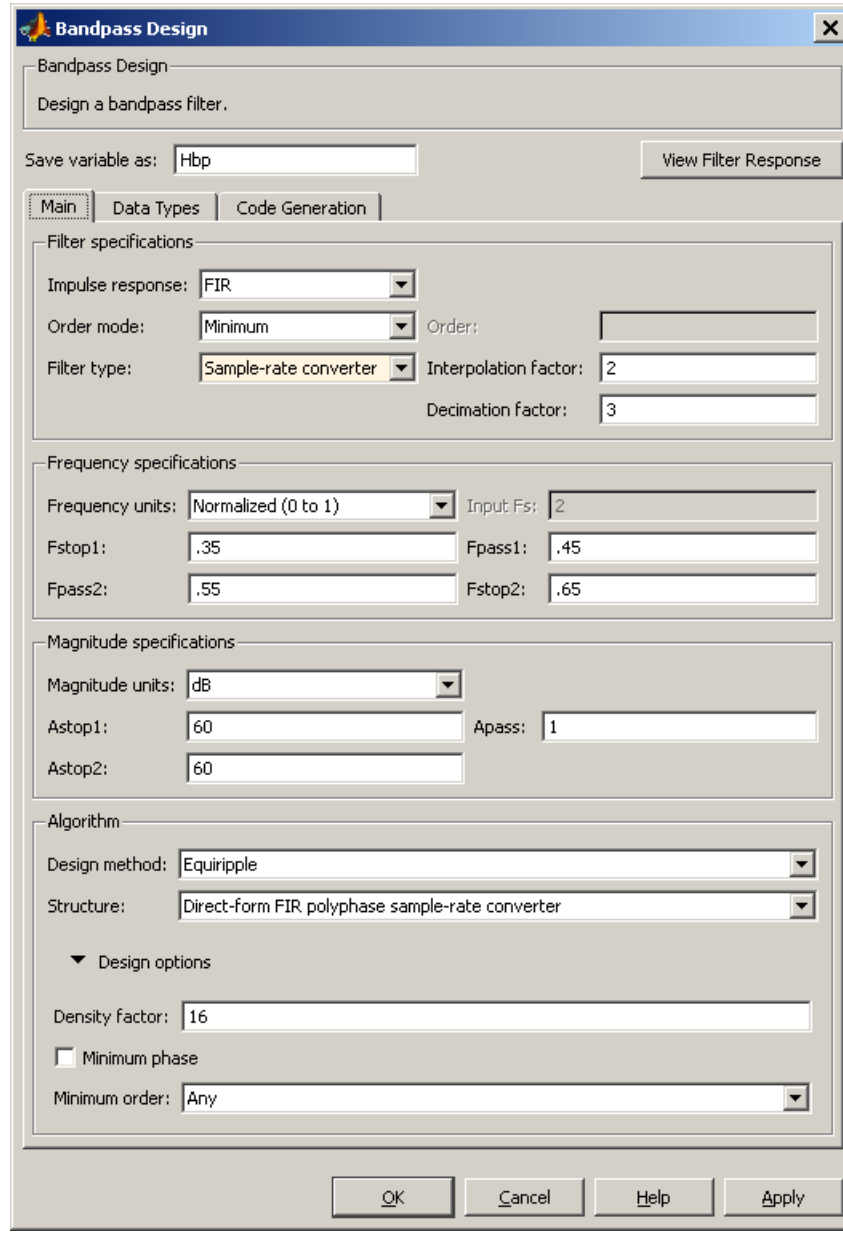
Structure

Select the structure for the filter, available for the design method selected in the previous box.

Design Options

- **Window** — replace the square brackets with the name of a `window` function or function handle. For example, “hamming” or “@hamming”. If the window function takes parameters other than the length, use a cell array. For example, {‘kaiser’,3.5} or {@chebwin,60}

Bandpass Filter Design Dialog Box — Main Pane



Filter Specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Impulse response

Select either `FIR` or `IIR` from the drop-down list, where `FIR` is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

Note The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Filter order mode

Select either `Minimum` (the default) or `Specify` from the drop-down box. Selecting `Specify` enables the **Order** option (explained in the following descriptions) so you can enter the filter order.

Filter type

Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, `filterbuilder` specifies single-rate filters.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting `Sample-rate converter` activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

Order

Enter the filter order. This option is enabled only if `Specify` was selected for **Filter order mode**.

Decimation Factor

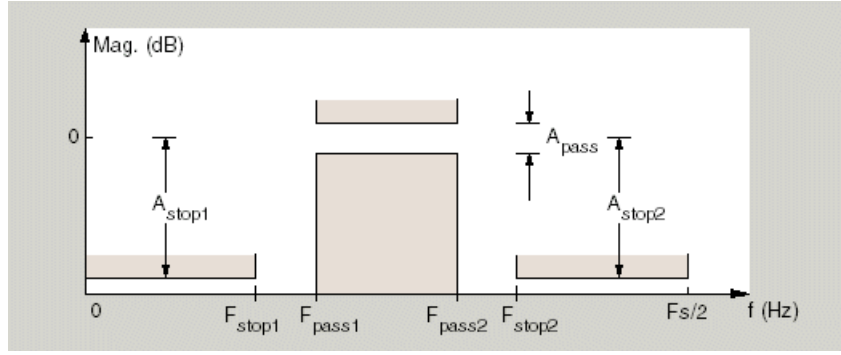
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default factor value is 2.

Frequency Specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to those shown in the following figure.



In the figure, regions between specification values such as Fstop1 and Fpass1 represent transition regions where the filter response is not explicitly defined.

Frequency constraints

Select the filter features to use to define the frequency response characteristics. The list contains the following options, when available for the filter specifications.

- `Passband and stopband edges` — Define the filter by specifying the frequencies for the edges for the stop- and passbands.
- `Passband edges` — Define the filter by specifying frequencies for the edges of the passband.
- `Stopband edges` — Define the filter by specifying frequencies for the edges of the stopbands.
- `3 dB points` — Define the filter response by specifying the locations of the 3 dB points. The 3 dB point is the frequency for the point 3 dB point below the passband value.
- `3 dB points and passband width` — Define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the passband.
- `3 dB points and stopband widths` — Define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the stopband.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Fstop1

Enter the frequency at the edge of the end of the first stopband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Fpass1

Enter the frequency at the edge of the start of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Fpass2

Enter the frequency at the edge of the end of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Fstop2

Enter the frequency at the edge of the start of the second stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude Specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- `Linear` — Specify the magnitude in linear units.
- `dB` — Specify the magnitude in dB (decibels). This is the default setting.
- `Squared` — Specify the magnitude in squared units.

Astop1

Enter the filter attenuation in the first stopband in the units you choose for **Magnitude units**, either linear or decibels.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Astop2

Enter the filter attenuation in the second stopband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterbuilder` uses to implement your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select

different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 20 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Minimum phase

To design a filter that is minimum phase, select **Minimum phase**. Clearing the **Minimum phase** option removes the phase constraint—the resulting design is not minimum phase.

Minimum order

When you select this parameter, the design method determines and design the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select `Any`, `Even`, or `Odd` from the drop-down list to direct the design to be any minimum order, or minimum even order, or minimum odd order.

Note Generally, **Minimum order** designs are not available for IIR filters.

Match Exactly

Specifies that the resulting filter design matches either the passband or stopband or both bands when you select `passband` or `stopband` or `both` from the drop-down list.

Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

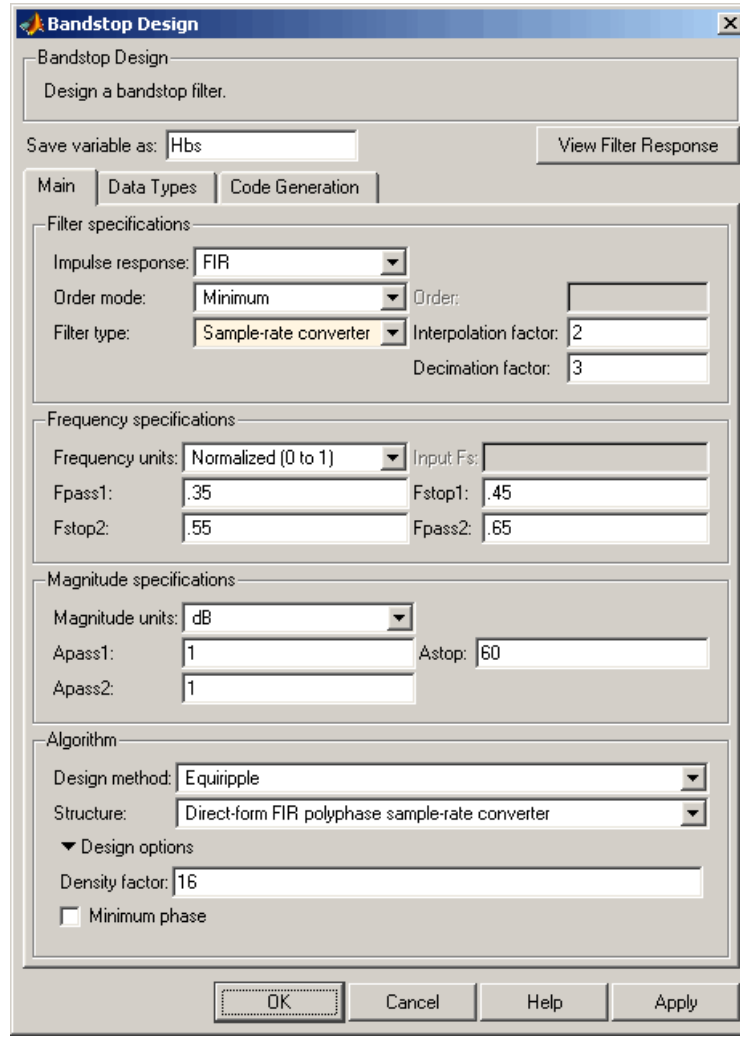
- `Flat` — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- `Linear` — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.
- `1/f` — Specifies that the stopband attenuation changes exponentially as the frequency increases, where `f` is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to `Flat`, **Stopband decay** has no affect on the stopband.
- When you set **Stopband shape** to `Linear`, enter the slope of the stopband in units of dB/rad/s. `filterbuilder` applies that slope to the stopband.
- When you set **Stopband shape** to `1/f`, enter a value for the exponent n in the relation $(1/f)^n$ to define the stopband decay. `filterbuilder` applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

Bandstop Filter Design Dialog Box — Main Pane



Filter Specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Impulse response

Select either `FIR` or `IIR` from the drop-down list, where `FIR` is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

Note The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Filter order mode

Select either `Minimum` (the default) or `Specify` from the drop-down list. Selecting `Specify` enables the **Order** option (see the following sections) so you can enter the filter order.

Filter type

Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, `filterbuilder` specifies single-rate filters.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting `Sample-rate converter` activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

Order

Enter the filter order. This option is enabled only if `Specify` was selected for **Filter order mode**.

Decimation Factor

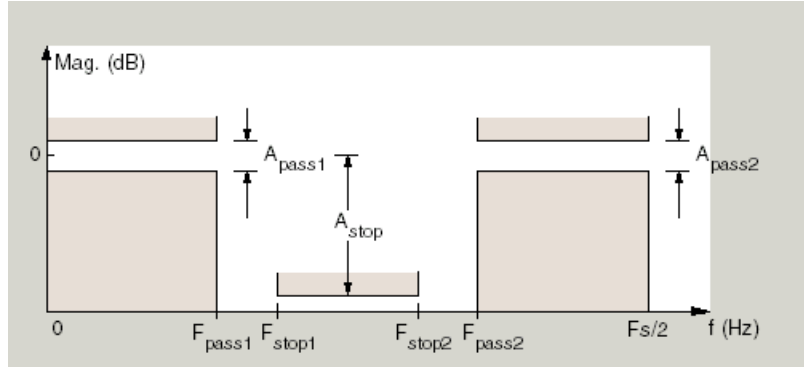
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default factor value is 2.

Frequency Specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to those shown in the following figure.



Frequency constraints

Select the filter features to use to define the frequency response characteristics. The list contains the following options, when available for the filter specifications.

- `Passband and stopband edges` — Define the filter by specifying the frequencies for the edges for the stop- and passbands.

- `Passband edges` — Define the filter by specifying frequencies for the edges of the passband.
- `Stopband edges` — Define the filter by specifying frequencies for the edges of the stopbands.
- `3 dB points` — Define the filter response by specifying the locations of the 3 dB points. The 3 dB point is the frequency for the point 3 dB point below the passband value.
- `3 dB points and passband width` — Define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the passband.
- `3 dB points and stopband widths` — Define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the stopband.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Output Fs

When you design an interpolator, Fs represents the sampling frequency at the filter output rather than the filter input. This option is available only when you set **Filter type** is `interpolator`.

Fpass1

Enter the frequency at the edge of the end of the first passband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Fstop1

Enter the frequency at the edge of the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Fstop2

Enter the frequency at the edge of the end of the stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Fpass2

Enter the frequency at the edge of the start of the second passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude Specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- `Linear` — Specify the magnitude in linear units.
- `dB` — Specify the magnitude in decibels (default).
- `Squared` — Specify the magnitude in squared units.

Apass1

Enter the filter ripple allowed in the first passband in the units you choose for **Magnitude units**, either linear or decibels.

Astop
Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels

Apass2
Enter the filter ripple allowed in the second passband in the units you choose for **Magnitude units**, either linear or decibels

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterbuilder` uses to implement your filter.

Design Method
Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

Structure
For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Scale SOS filter coefficients to reduce chance of overflow
Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select

different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 20 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Minimum phase

To design a filter that is minimum phase, select **Minimum phase**. Clearing the **Minimum phase** option removes the phase constraint—the resulting design is not minimum phase.

Minimum order

When you select this parameter, the design method determines and design the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select `Any`, `Even`, or `Odd` from the drop-down list to direct the design to be any minimum order, or minimum even order, or minimum odd order.

Note Generally, **Minimum order** designs are not available for IIR filters.

Match Exactly

Specifies that the resulting filter design matches either the passband or stopband or both bands when you select `passband` or `stopband` or `both` from the drop-down list.

Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

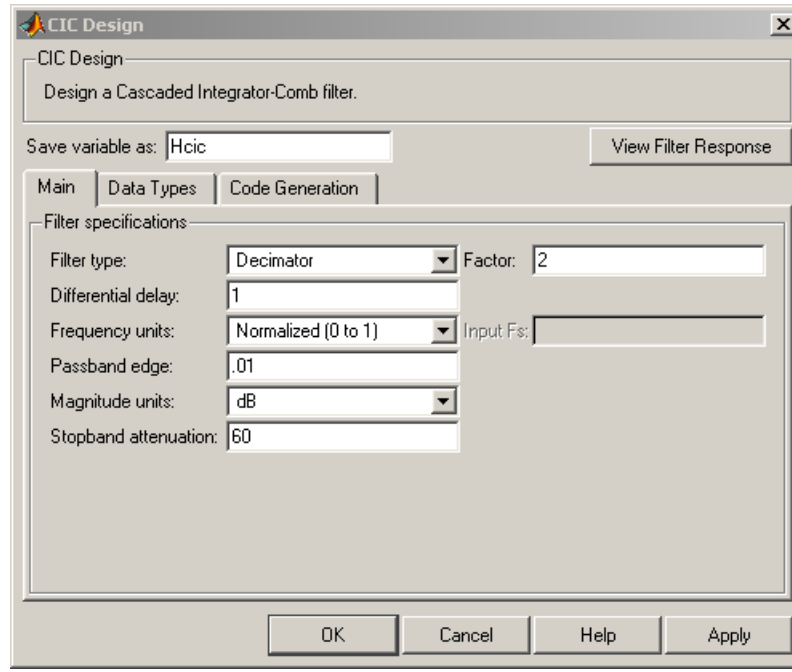
- `Flat` — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- `Linear` — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.
- `1/f` — Specifies that the stopband attenuation changes exponentially as the frequency increases, where `f` is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to `Flat`, **Stopband decay** has no affect on the stopband.
- When you set **Stopband shape** to `Linear`, enter the slope of the stopband in units of dB/rad/s. `filterbuilder` applies that slope to the stopband.
- When you set **Stopband shape** to `1/f`, enter a value for the exponent n in the relation $(1/f)^n$ to define the stopband decay. `filterbuilder` applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

CIC Filter Design Dialog Box — Main Pane



Filter Specifications

Parameters in this group enable you to specify your CIC filter format, such as the filter type and the differential delay.

Filter type

Select whether your filter will be a `decimator` or an `interpolator`. Your choice determines the type of filter and the design methods and structures that are available to implement your filter. Selecting `decimator` or `interpolator` activates the **Factor** option. When you design an interpolator, you enable the **Output Fs** parameter.

When you design either a decimator or interpolator, the resulting filter is a CIC filter that decimates or interpolates your input signal.

Differential Delay

Specify the differential delay of your CIC filter. The default value is `1`. Most CIC filters use 1 or 2. Differential delay changes both the shape and number of nulls in the filter response. The delay value also affects the null locations. Increasing the delay increases the number and sharpness of the nulls and response between nulls. Generally, 1 or 2 work best as values for the delay.

Factor

When you select `decimator` or `interpolator` for **Filter type**, enter the decimation or interpolation factor for your filter in this field. You must enter a positive integer for the factor. The default factor value is 2.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Output Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter output. When you provide an output sampling frequency, all frequencies in the

specifications are in the selected units as well. This parameter is available only when you design interpolators.

Fpass

Enter the frequency at the end of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude units

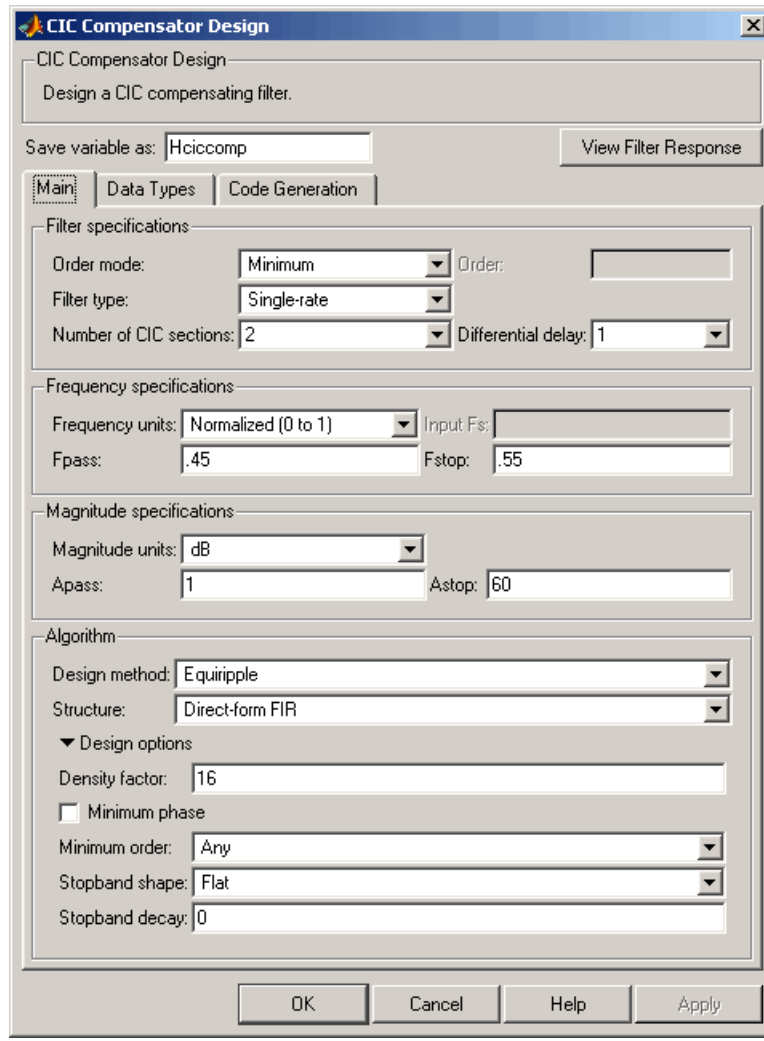
Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- `Linear` — Specify the magnitude in linear units.
- `dB` — Specify the magnitude in decibels (default).
- `Squared` — Specify the magnitude in squared units.

Astop

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

CIC Compensator Filter Design Dialog Box — Main Pane



Filter Specifications

Parameters in this group enable you to specify your filter format, such as the filter order mode and the filter type.

Filter order mode

Select either `Minimum` (the default) or `Specify` from the drop-down list. Selecting `Specify` enables the **Order** option (see the following sections) so you can enter the filter order.

Filter type

Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, `filterbuilder` specifies single-rate filters.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting `Sample-rate converter` activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

Order

Enter the filter order. This option is enabled only if `Specify` was selected for **Filter order mode**.

Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default factor value is 2.

Number of CIC sections

Specify the number of sections in the CIC filter for which you are designing this compensator. Select the number of sections from the drop-down list or enter the number.

Differential Delay

Specify the differential delay of your target CIC filter. The default value is `1`. Most CIC filters use 1 or 2.

Frequency Specifications

The parameters in this group allow you to specify your filter response curve.

Frequency Specifications

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Output Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter output. When you provide an output sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available only when you design interpolators.

Fpass

Enter the frequency at the end of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Fstop

Enter the frequency at the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude Specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- `Linear` — Specify the magnitude in linear units.
- `dB` — Specify the magnitude in decibels (default).
- `Squared` — Specify the magnitude in squared units.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterbuilder` uses to implement your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 20 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Minimum phase

To design a filter that is minimum phase, select **Minimum phase**. Clearing the **Minimum phase** option removes the phase constraint—the resulting design is not minimum phase.

Minimum order

When you select this parameter, the design method determines and design the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select `Any`, `Even`, or `Odd` from the drop-down list to direct the design to be any minimum order, or minimum even order, or minimum odd order.

Note Generally, **Minimum order** designs are not available for IIR filters.

Match Exactly

Specifies that the resulting filter design matches either the passband or stopband or both bands when you select `passband` or `stopband` or `both` from the drop-down list.

Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- `Flat` — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- `Linear` — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.
- `1/f` — Specifies that the stopband attenuation changes exponentially as the frequency increases, where `f` is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

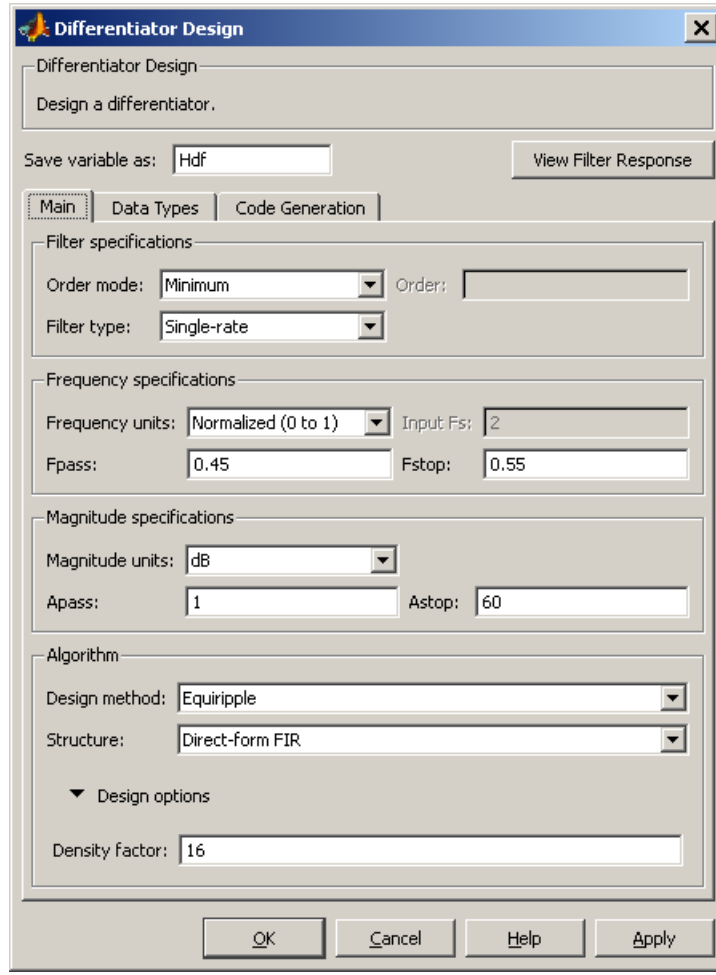
Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to `Flat`, **Stopband decay** has no affect on the stopband.
- When you set **Stopband shape** to `Linear`, enter the slope of the stopband in units of dB/rad/s. `filterbuilder` applies that slope to the stopband.
- When you set **Stopband shape** to `1/f`, enter a value for the exponent *n* in the relation $(1/f)^n$ to define the stopband decay.

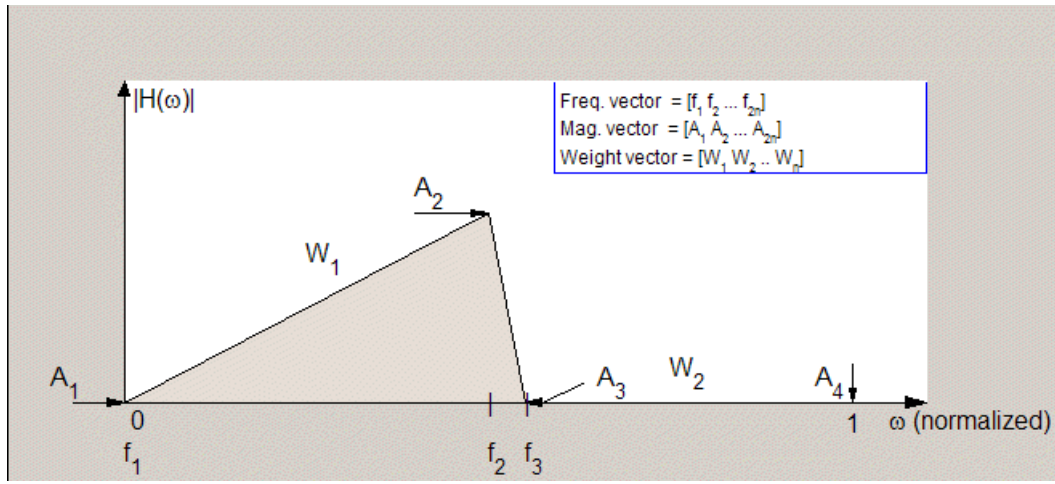
`filterbuilder` applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

Differentiator Filter Design Dialog Box — Main Pane



Filter Specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order. Graphically, the filter specifications look similar to those shown in the following figure.



In the figure, regions between specification values such as **Fpass** (f_1) and **Fstop** (f_3) represent transition regions where the filter response is not explicitly defined.

Filter order mode

Select either `Minimum` (the default) or `Specify` from the drop-down list. Selecting `Specify` enables the **Order** option (see the following sections) so you can enter the filter order.

Filter type

Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, `filterbuilder` specifies single-rate filters.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting `Sample-rate converter` activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

Order

Enter the filter order. This option is enabled only if `Specify` was selected for **Filter order mode**.

Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default factor value is 2.

Frequency Specifications

The parameters in this group allow you to specify your filter response curve.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you

provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Fpass

Enter the frequency at the end of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Fstop

Enter the frequency at the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude Specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- `Linear` — Specify the magnitude in linear units.
- `dB` — Specify the magnitude in decibels (default).
- `Squared` — Specify the magnitude in squared units.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Astop2

Enter the filter attenuation in the second stopband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterbuilder` uses to implement your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 20 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Minimum phase

To design a filter that is minimum phase, select **Minimum phase**. Clearing the **Minimum phase** option removes the phase constraint—the resulting design is not minimum phase.

Minimum order

When you select this parameter, the design method determines and design the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select `Any`, `Even`, or `Odd` from the drop-down list to direct the design to be any minimum order, or minimum even order, or minimum odd order.

Note Generally, **Minimum order** designs are not available for IIR filters.

Match Exactly

Specifies that the resulting filter design matches either the passband or stopband or both bands when you select `passband` or `stopband` or `both` from the drop-down list.

Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- `Flat` — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- `Linear` — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.
- `1/f` — Specifies that the stopband attenuation changes exponentially as the frequency increases, where `f` is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

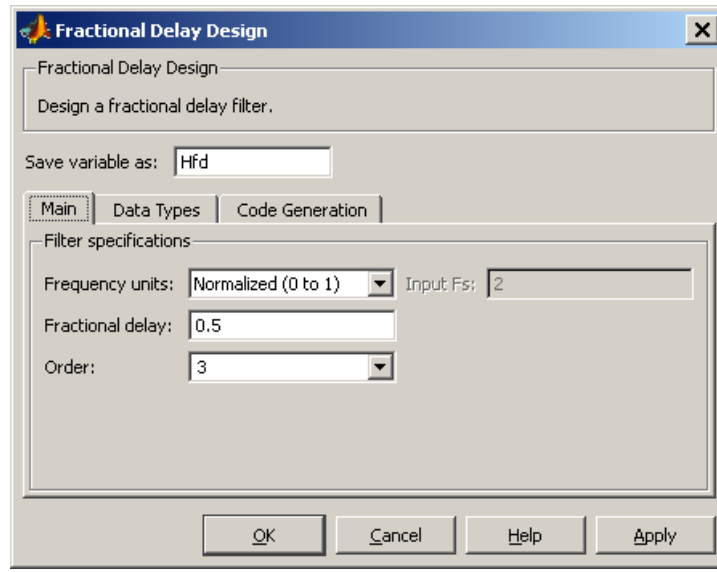
Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to `Flat`, **Stopband decay** has no affect on the stopband.
- When you set **Stopband shape** to `Linear`, enter the slope of the stopband in units of dB/rad/s. `filterbuilder` applies that slope to the stopband.
- When you set **Stopband shape** to `1/f`, enter a value for the exponent n in the relation $(1/f)^n$ to define the stopband decay.

`filterbuilder` applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

Fractional Delay Filter Design Dialog Box — Main Pane



Frequency Specifications

Parameters in this group enable you to specify your filter format, such as the fractional delay and the filter order.

Order

If you choose `Specify` for **Filter order mode**, enter your filter order in this field, or select the order from the drop-down list.`filterbuilder` designs a filter with the order you specify.

Fractional delay

Specify a value between 0 and 1 samples for the filter fractional delay. The default value is `0.5` samples.

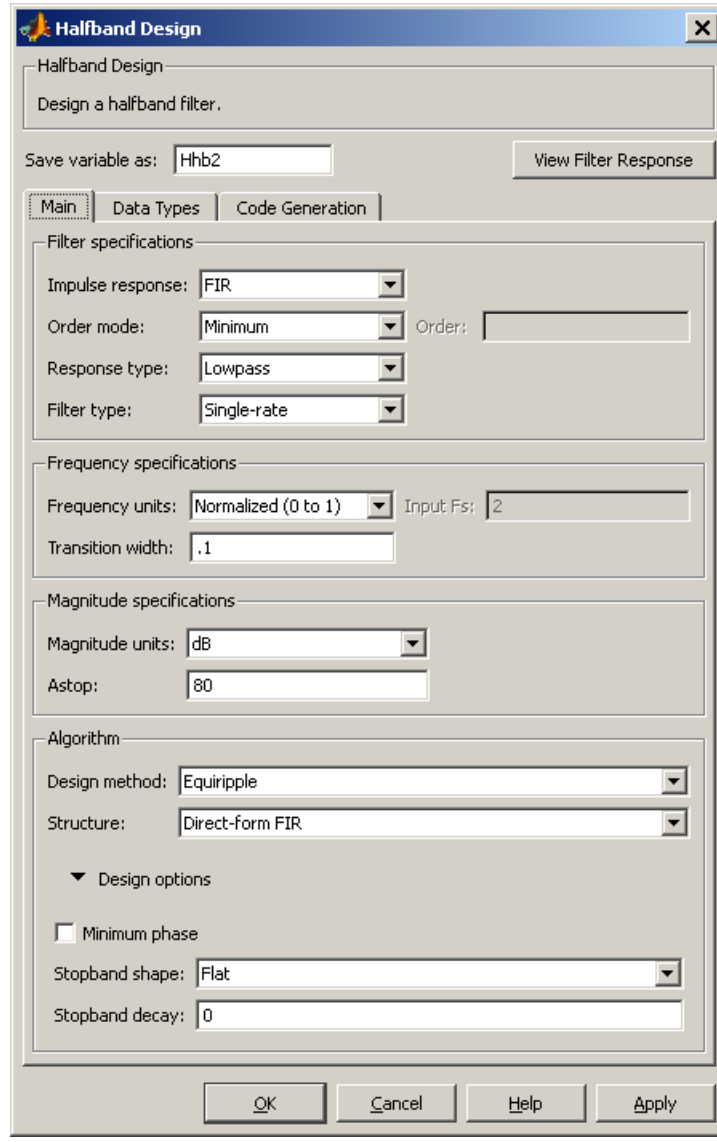
Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Halfband Filter Design Dialog Box — Main Pane



Filter Specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Impulse response

Select either `FIR` or `IIR` from the drop-down list, where `FIR` is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

Note The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Filter order mode

Select either `Minimum` (the default) or `Specify` from the drop-down list. Selecting `Specify` enables the **Order** option (see the following sections) so you can enter the filter order.

Filter type

Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, `filterbuilder` specifies single-rate filters.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting `Sample-rate converter` activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

Order

Enter the filter order. This option is enabled only if `Specify` was selected for **Filter order mode**.

Decimation Factor

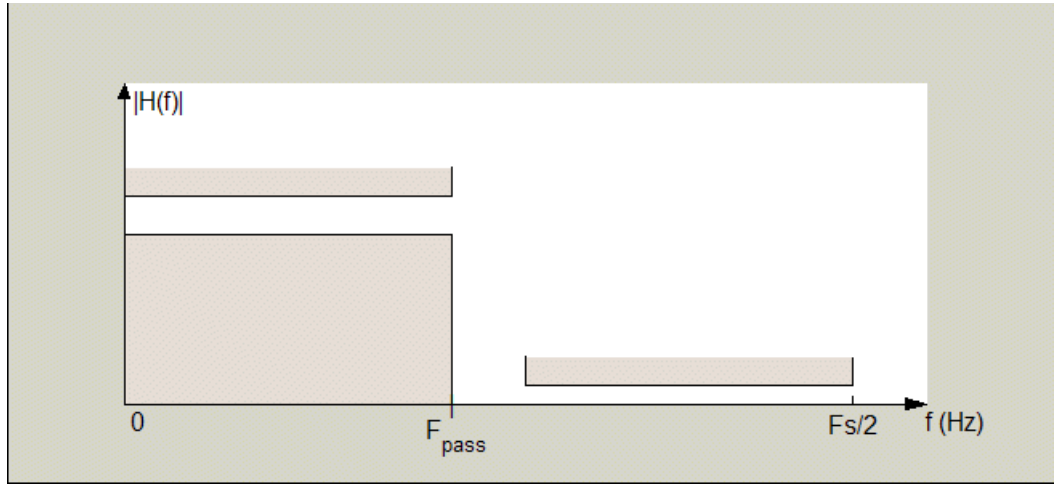
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default factor value is 2.

Frequency Specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications for a halfband lowpass filter look similar to those shown in the following figure.



In the figure, the transition region lies between the end of the passband and the start of the stopband. The width is defined explicitly by the value of **Transition width**.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)`

to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Transition width

Specify the width of the transition between the end of the passband and the edge of the stopband. Specify the value in normalized frequency units or the absolute units you select in **Frequency units**.

Magnitude Specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- `Linear` — Specify the magnitude in linear units.
- `dB` — Specify the magnitude in decibels (default).
- `Squared` — Specify the magnitude in squared units.

Astop

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterbuilder` uses to implement your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 20 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Minimum phase

To design a filter that is minimum phase, select **Minimum phase**. Clearing the **Minimum phase** option removes the phase constraint—the resulting design is not minimum phase.

Minimum order

When you select this parameter, the design method determines and design the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select `Any`, `Even`, or `Odd` from the drop-down list to direct the design to be any minimum order, or minimum even order, or minimum odd order.

Note Generally, **Minimum order** designs are not available for IIR filters.

Match Exactly

Specifies that the resulting filter design matches either the passband or stopband or both bands when you select `passband` or `stopband` or `both` from the drop-down list.

Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- `Flat` — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- `Linear` — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.

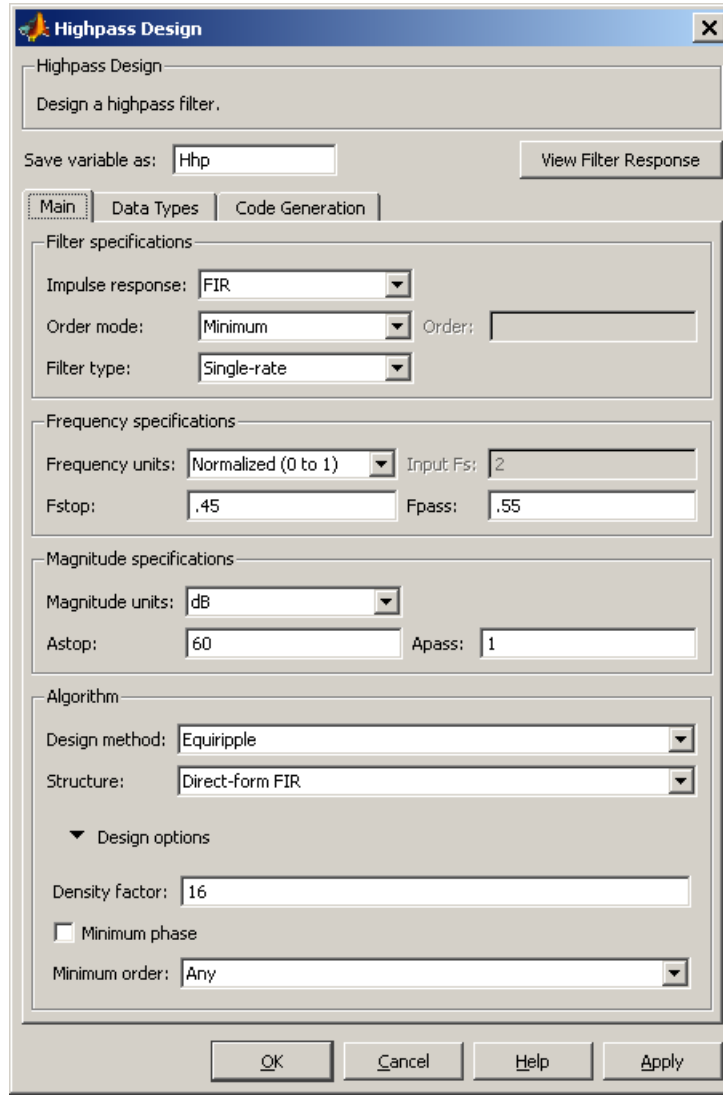
- `1/f` — Specifies that the stopband attenuation changes exponentially as the frequency increases, where `f` is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to `Flat`, **Stopband decay** has no affect on the stopband.
- When you set **Stopband shape** to `Linear`, enter the slope of the stopband in units of dB/rad/s. `filterbuilder` applies that slope to the stopband.
- When you set **Stopband shape** to `1/f`, enter a value for the exponent n in the relation $(1/f)^n$ to define the stopband decay. `filterbuilder` applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

Highpass Filter Design Dialog Box — Main Pane



Filter Specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Impulse response

Select either `FIR` or `IIR` from the drop-down list, where `FIR` is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

Note The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Filter order mode

Select either `Minimum` (the default) or `Specify` from the drop-down list. Selecting `Specify` enables the **Order** option (see the following sections) so you can enter the filter order.

Filter type

Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, `filterbuilder` specifies single-rate filters.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting `Sample-rate converter` activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

Order

Enter the filter order. This option is enabled only if `Specify` was selected for **Filter order mode**.

Decimation Factor

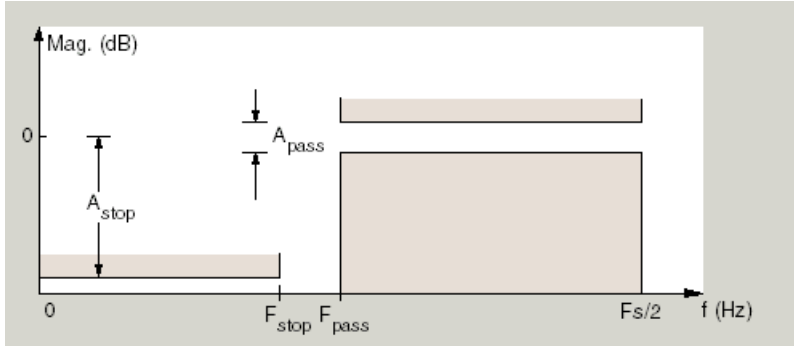
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default factor value is 2.

Frequency Specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to those shown in the following figure.



In the figure, the region between specification values Fstop and Fpass represents the transition region where the filter response is not explicitly defined.

Frequency constraints

Select the filter features to use to define the frequency response characteristics. The list contains the following options, when available for the filter specifications.

- `Passband and stopband edges` — Define the filter by specifying the frequencies for the edges for the stop- and passbands.
- `Passband edges` — Define the filter by specifying frequencies for the edges of the passband.
- `Stopband edges` — Define the filter by specifying frequencies for the edges of the stopbands.
- `3 dB points` — Define the filter response by specifying the locations of the 3 dB points. The 3 dB point is the frequency for the point 3 dB point below the passband value.
- `3 dB points and passband width` — Define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the passband.
- `3 dB points and stopband widths` — Define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the stopband.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Fstop

Enter the frequency at the edge of the end of the stopband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Fpass

Enter the frequency at the edge of the start of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude Specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- `Linear` — Specify the magnitude in linear units.
- `dB` — Specify the magnitude in decibels (default).
- `Squared` — Specify the magnitude in squared units.

Astop

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterbuilder` uses to implement your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the

specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 20 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Minimum phase

To design a filter that is minimum phase, select **Minimum phase**. Clearing the **Minimum phase** option removes the phase constraint—the resulting design is not minimum phase.

Minimum order

When you select this parameter, the design method determines and design the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select `Any`, `Even`, or `Odd` from the drop-down list to direct the design to be any minimum order, or minimum even order, or minimum odd order.

Note Generally, **Minimum order** designs are not available for IIR filters.

Match Exactly

Specifies that the resulting filter design matches either the passband or stopband or both bands when you select `passband` or `stopband` or `both` from the drop-down list.

Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

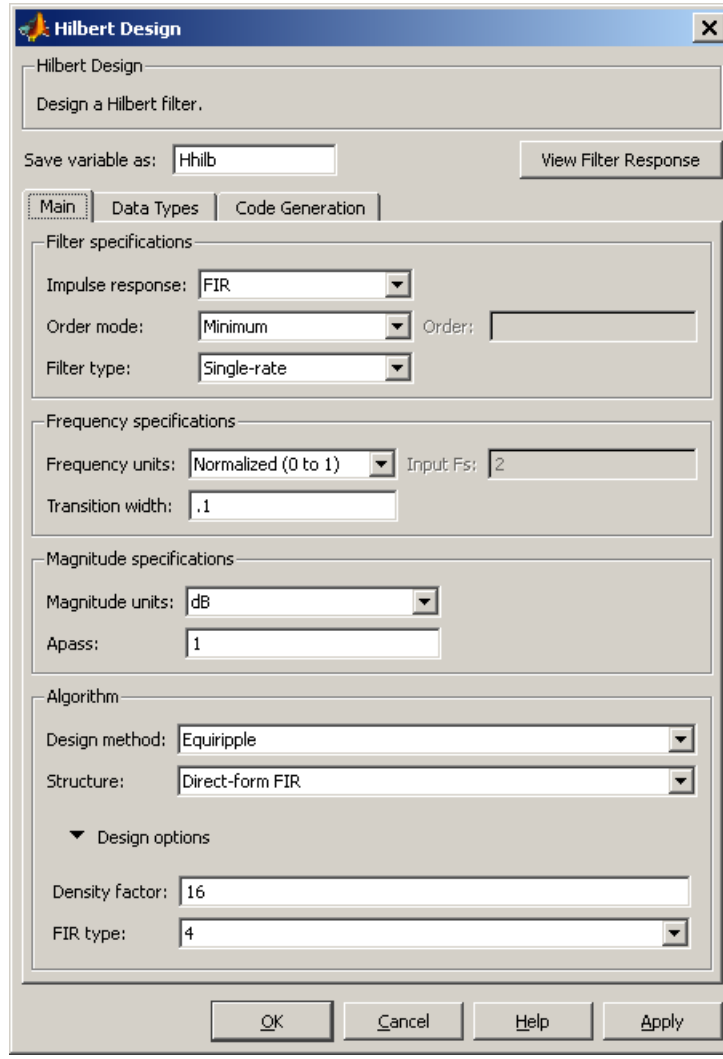
- `Flat` — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- `Linear` — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.
- `1/f` — Specifies that the stopband attenuation changes exponentially as the frequency increases, where `f` is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to `Flat`, **Stopband decay** has no affect on the stopband.
- When you set **Stopband shape** to `Linear`, enter the slope of the stopband in units of dB/rad/s. `filterbuilder` applies that slope to the stopband.
- When you set **Stopband shape** to `1/f`, enter a value for the exponent n in the relation $(1/f)^n$ to define the stopband decay. `filterbuilder` applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

Hilbert Filter Design Dialog Box — Main Pane



Filter Specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Impulse response

Select either `FIR` or `IIR` from the drop-down list, where `FIR` is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

Note The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Filter order mode

Select either `Minimum` (the default) or `Specify` from the drop-down list. Selecting `Specify` enables the **Order** option (see the following sections) so you can enter the filter order.

Filter type

Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, `filterbuilder` specifies single-rate filters.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting `Sample-rate converter` activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

Order

Enter the filter order. This option is enabled only if `Specify` was selected for **Filter order mode**.

Decimation Factor

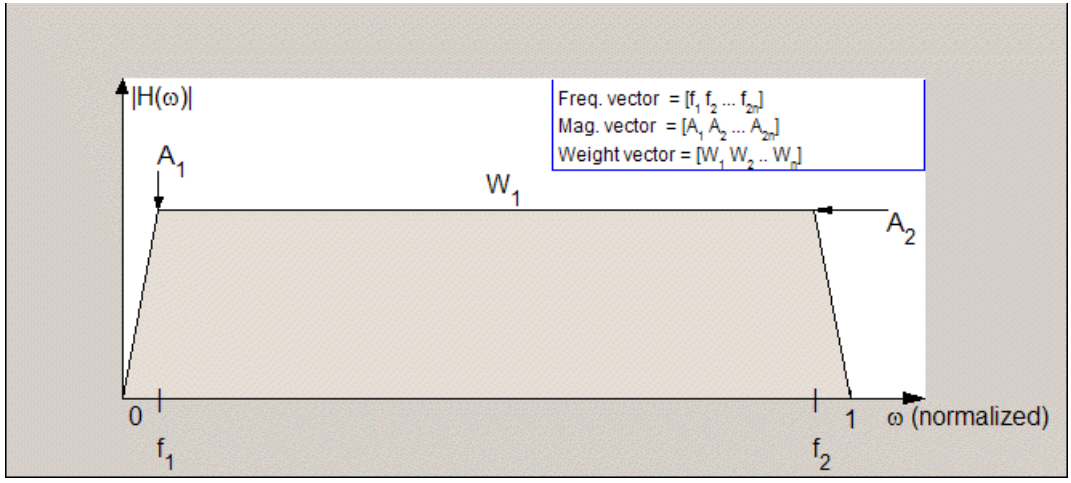
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default factor value is 2.

Frequency Specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to those shown in the following figure.



In the figure, the regions between 0 and f_1 and between f_2 and 1 represent the transition regions where the filter response is explicitly defined by the transition width.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)`

to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Transition width

Specify the width of the transitions at the ends of the passband. Specify the value in normalized frequency units or the absolute units you select in **Frequency units**.

Magnitude Specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- `Linear` — Specify the magnitude in linear units.
- `dB` — Specify the magnitude in decibels (default)
- `Squared` — Specify the magnitude in squared units.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterbuilder` uses to implement your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 20 represents a

reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

FIR Type

Specify whether to design a type 3 or a type 4 FIR filter. The filter type is defined as follows:

- Type 3 — FIR filter with even order antisymmetric coefficients
- Type 4 — FIR filter with odd order antisymmetric coefficients

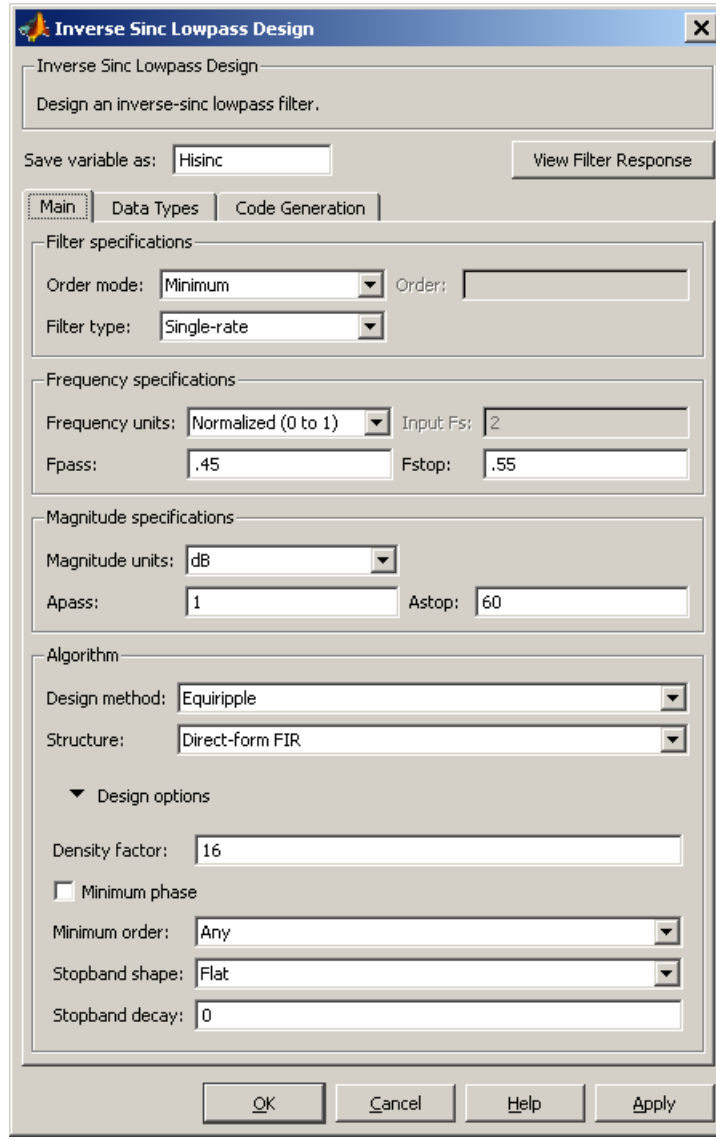
Select either `3` or `4` from the drop-down list.

Minimum order

When you select this parameter, the design method determines and design the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select `Any`, `Even`, or `Odd` from the drop-down list to direct the design to be any minimum order, or minimum even order, or minimum odd order.

Note Generally, **Minimum order** designs are not available for IIR filters.

Inverse Sinc Filter Design Dialog Box — Main Pane



Filter Specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Filter order mode

Select either `Minimum` (the default) or `Specify` from the drop-down list. Selecting `Specify` enables the **Order** option (see the following sections) so you can enter the filter order.

Filter type

Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, `filterbuilder` specifies single-rate filters.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting `Sample-rate converter` activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

Order

Enter the filter order. This option is enabled only if `Specify` was selected for **Filter order mode**.

Decimation Factor

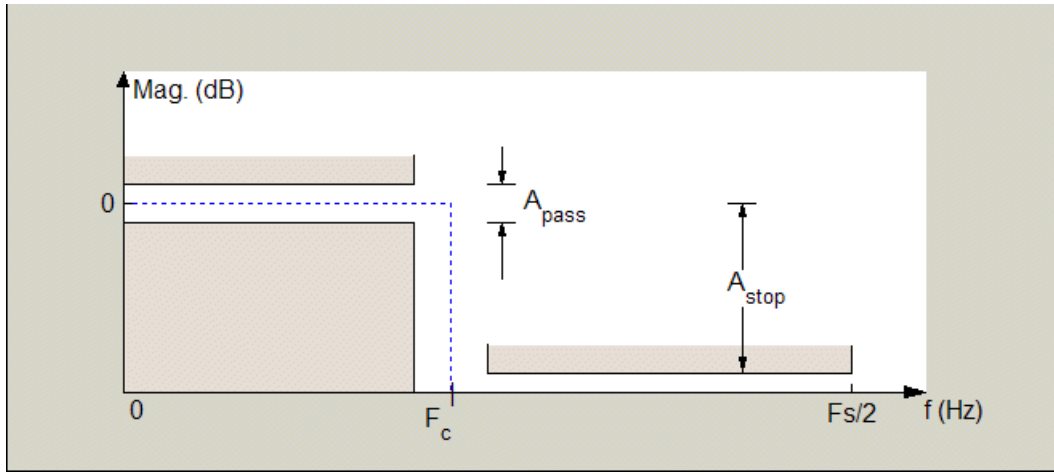
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default factor value is 2.

Frequency Specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to those shown in the following figure.



Regions between specification values such as Fpass and Fstop represent transition regions where the filter response is not explicitly defined.

Frequency constraints

Select the filter features to use to define the frequency response characteristics. The list contains the following options, when available for the filter specifications.

- `Passband and stopband edges` — Define the filter by specifying the frequencies for the edges for the stop- and passbands.
- `Passband edges` — Define the filter by specifying frequencies for the edges of the passband.
- `Stopband edges` — Define the filter by specifying frequencies for the edges of the stopbands.

- `3 dB points` — Define the filter response by specifying the locations of the 3 dB points. The 3 dB point is the frequency for the point 3 dB point below the passband value.
- `3 dB points and passband width` — Define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the passband.
- `3 dB points and stopband widths` — Define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the stopband.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Fpass

Enter the frequency at the end of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Fstop

Enter the frequency at the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude Specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- `Linear` — Specify the magnitude in linear units.
- `dB` — Specify the magnitude in decibels (default)
- `Squared` — Specify the magnitude in squared units.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Astop

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterbuilder` uses to implement your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 20 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Minimum phase

To design a filter that is minimum phase, select **Minimum phase**. Clearing the **Minimum phase** option removes the phase constraint—the resulting design is not minimum phase.

Minimum order

When you select this parameter, the design method determines and design the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select `Any`, `Even`, or `Odd` from the drop-down list to direct the design to be any minimum order, or minimum even order, or minimum odd order.

Note Generally, **Minimum order** designs are not available for IIR filters.

Match Exactly

Specifies that the resulting filter design matches either the passband or stopband or both bands when you select `passband` or `stopband` or `both` from the drop-down list.

Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options;

- `Flat` — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- `Linear` — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.
- `1/f` — Specifies that the stopband attenuation changes exponentially as the frequency increases, where `f` is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

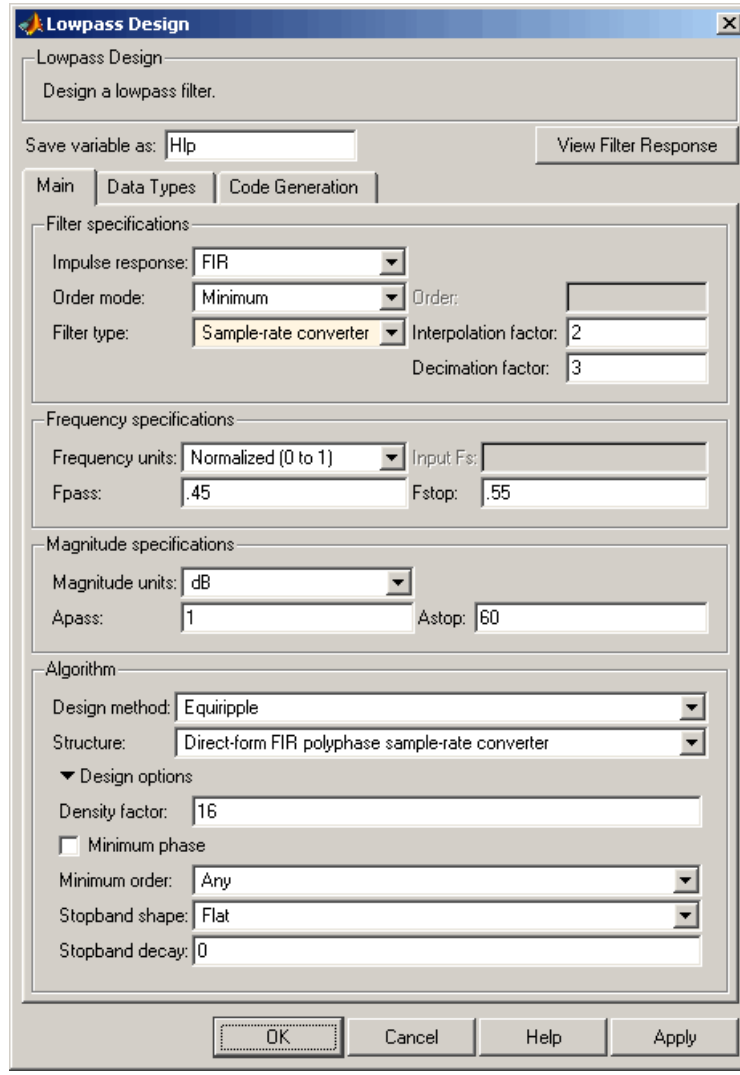
Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to `Flat`, **Stopband decay** has no affect on the stopband.
- When you set **Stopband shape** to `Linear`, enter the slope of the stopband in units of dB/rad/s. `filterbuilder` applies that slope to the stopband.
- When you set **Stopband shape** to `1/f`, enter a value for the exponent n in the relation $(1/f)^n$ to define the stopband decay.

`filterbuilder` applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

Lowpass Filter Design Dialog Box — Main Pane



Filter Specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Impulse response

Select either `FIR` or `IIR` from the drop-down list, where `FIR` is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

Note The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Filter order mode

Select either `Minimum` (the default) or `Specify` from the drop-down list. Selecting `Specify` enables the **Order** option (see the following sections) so you can enter the filter order.

Filter type

Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, `filterbuilder` specifies single-rate filters.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting `Sample-rate converter` activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

Order

Enter the filter order. This option is enabled only if `Specify` was selected for **Filter order mode**.

Decimation Factor

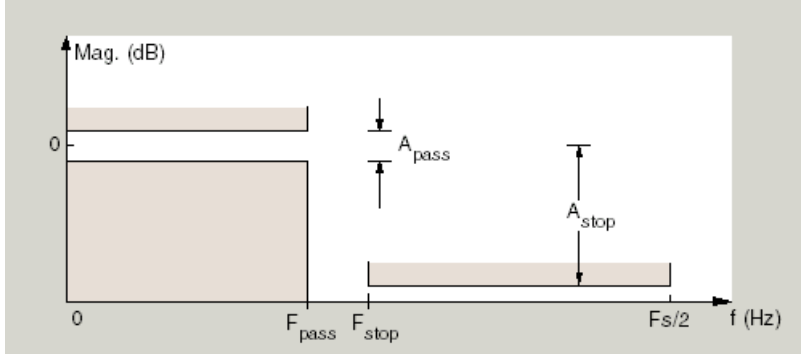
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default factor value is 2.

Frequency Specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to the one shown in the following figure.



In the figure, regions between specification values such as F_{pass} and F_{stop} represent transition regions where the filter response is not explicitly defined.

Frequency constraints

Select the filter features to use to define the frequency response characteristics. The list contains the following options, when available for the filter specifications.

- `Passband and stopband edges` — Define the filter by specifying the frequencies for the edges for the stopbands and passbands.
- `Passband edges` — Define the filter by specifying frequencies for the edges of the passband.
- `Stopband edges` — Define the filter by specifying frequencies for the edges of the stopbands.
- `3 dB points` — Define the filter response by specifying the locations of the 3 dB points. The 3 dB point is the frequency for the point 3 dB point below the passband value.
- `3 dB points and passband width` — Define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the passband.
- `3 dB points and stopband widths` — Define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the stopband.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Fpass

Enter the frequency at the of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Fstop

Enter the frequency at the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude Specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- `Linear` — Specify the magnitude in linear units.
- `dB` — Specify the magnitude in decibels (default)
- `Squared` — Specify the magnitude in squared units.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Astop

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterbuilder` uses to implement your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the

specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 20 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Minimum phase

To design a filter that is minimum phase, select **Minimum phase**. Clearing the **Minimum phase** option removes the phase constraint—the resulting design is not minimum phase.

Minimum order

When you select this parameter, the design method determines and design the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select `Any`, `Even`, or `Odd` from the drop-down list to direct the design to be any minimum order, or minimum even order, or minimum odd order.

Note Generally, **Minimum order** designs are not available for IIR filters.

Match Exactly

Specifies that the resulting filter design matches either the passband or stopband or both bands when you select `passband` or `stopband` or `both` from the drop-down list.

Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

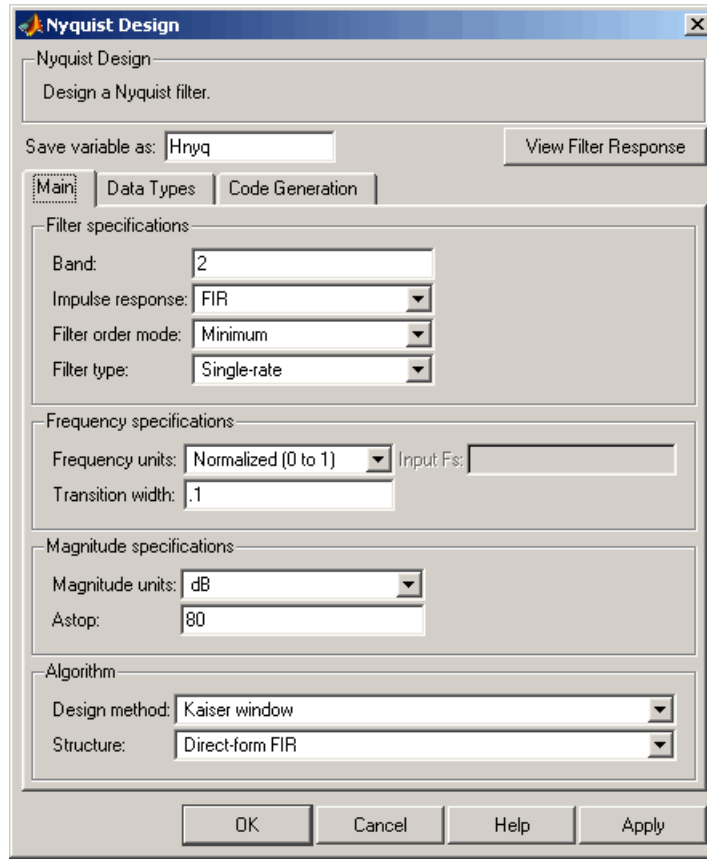
- `Flat` — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- `Linear` — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.
- `1/f` — Specifies that the stopband attenuation changes exponentially as the frequency increases, where `f` is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to `Flat`, **Stopband decay** has no affect on the stopband.
- When you set **Stopband shape** to `Linear`, enter the slope of the stopband in units of dB/rad/s. `filterbuilder` applies that slope to the stopband.
- When you set **Stopband shape** to `1/f`, enter a value for the exponent *n* in the relation $(1/f)^n$ to define the stopband decay. `filterbuilder` applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

Nyquist Filter Design Dialog Box — Main Pane



Filter Specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Band

Specifies the location of the center of the transition region between the passband and the stopband. The center of the transition region, bw, is calculated using the value for Band:

bw = Fs/(2*Band).

Impulse response

Select either `FIR` or `IIR` from the drop-down list, where `FIR` is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

Note The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Filter order mode

Select either `Minimum` (the default) or `Specify` from the drop-down list. Selecting `Specify` enables the **Order** option (see the following sections) so you can enter the filter order.

Filter type

Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, `filterbuilder` specifies single-rate filters.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting `Sample-rate converter` activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

Order

Enter the filter order. This option is enabled only if `Specify` was selected for **Filter order mode**.

Decimation Factor

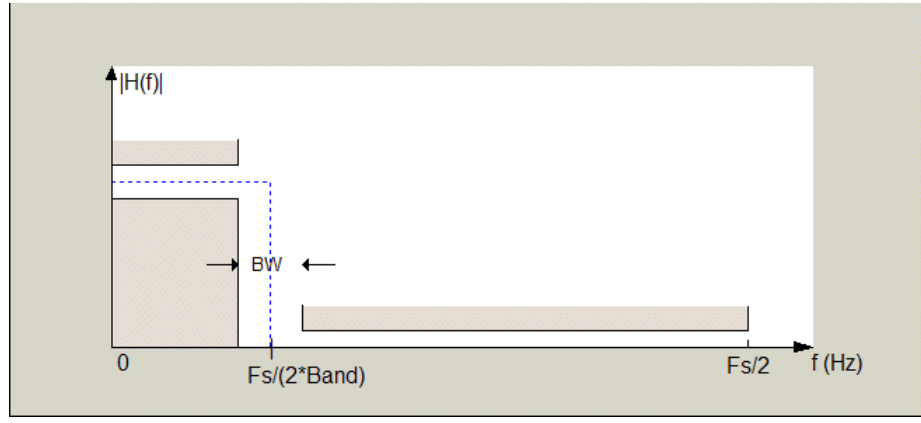
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default factor value is 2.

Frequency Specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to those shown in the following figure.



In the figure, BW is the width of the transition region and **Band** determines the location of the center of the region.

Frequency constraints

Select the filter features to use to define the frequency response characteristics. The list contains the following options, when available for the filter specifications.

- `Passband and stopband edges` — Define the filter by specifying the frequencies for the edges for the stopbands and passbands.
- `Passband edges` — Define the filter by specifying frequencies for the edges of the passband.
- `Stopband edges` — Define the filter by specifying frequencies for the edges of the stopbands.
- `3 dB points` — Define the filter response by specifying the locations of the 3 dB points. The 3 dB point is the frequency for the point 3 dB point below the passband value.
- `3 dB points and passband width` — Define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the passband.
- `3 dB points and stopband widths` — Define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the stopband.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is

available when you select one of the frequency options from the **Frequency units** list.

Transition width

Specify the width of the transition between the end of the passband and the edge of the stopband. Specify the value in normalized frequency units or the absolute units you select in **Frequency units**.

Magnitude Specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- `Linear` — Specify the magnitude in linear units.
- `dB` — Specify the magnitude in decibels (default)
- `Squared` — Specify the magnitude in squared units.

Astop

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterbuilder` uses to implement your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 20 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Minimum phase

To design a filter that is minimum phase, select **Minimum phase**. Clearing the **Minimum phase** option removes the phase constraint—the resulting design is not minimum phase.

Minimum order

When you select this parameter, the design method determines and design the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select `Any`, `Even`, or `Odd` from the drop-down list to direct the design to be any minimum order, or minimum even order, or minimum odd order.

Note Generally, **Minimum order** designs are not available for IIR filters.

Match Exactly

Specifies that the resulting filter design matches either the passband or stopband or both bands when you select `passband` or `stopband` or `both` from the drop-down list.

Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- `Flat` — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- `Linear` — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.

- `1/f` — Specifies that the stopband attenuation changes exponentially as the frequency increases, where `f` is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

Stopband Decay

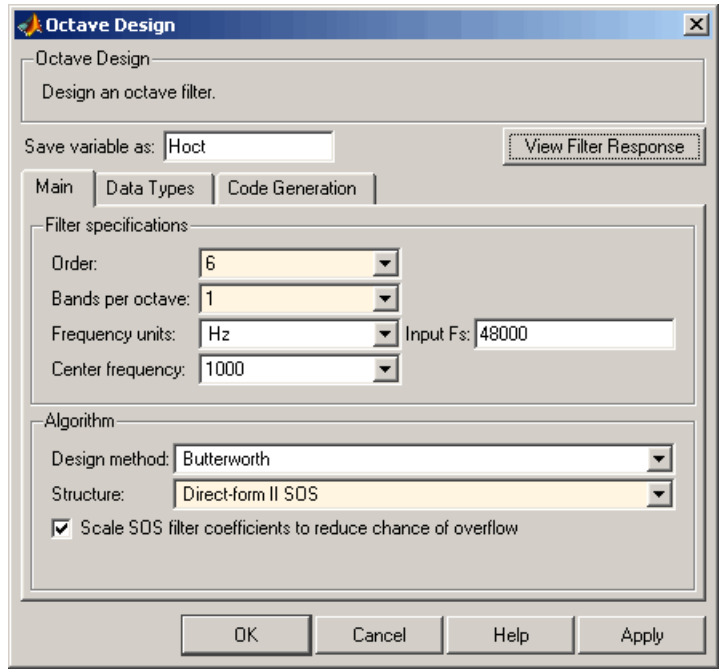
When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to `Flat`, **Stopband decay** has no affect on the stopband.
- When you set **Stopband shape** to `Linear`, enter the slope of the stopband in units of dB/rad/s. `filterbuilder` applies that slope to the stopband.
- When you set **Stopband shape** to `1/f`, enter a value for the exponent n in the relation $(1/f)^n$ to define the stopband decay. `filterbuilder` applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

Notch

See “Peak/Notch Filter Design Dialog Box — Main Pane” on page 10-445.

Octave Filter Design Dialog Box — Main Pane



Filter Specifications

Order

Specify filter order. Possible values are: `4, 6, 8, 10`.

Bands per octave

Specify the number of bands per octave. Possible values are: `1, 3, 6, 12, 24`.

Frequency units

Specify frequency units as `Hz` or `kHz`.

Input Fs

Specify the input sampling frequency in the frequency units specified previously.

Center Frequency

Select from the drop-down list of available center frequency values.

Algorithm

Design Method

Butterworth is the design method used for this type of filter.

Structure

Specify filter structure. Choose from:

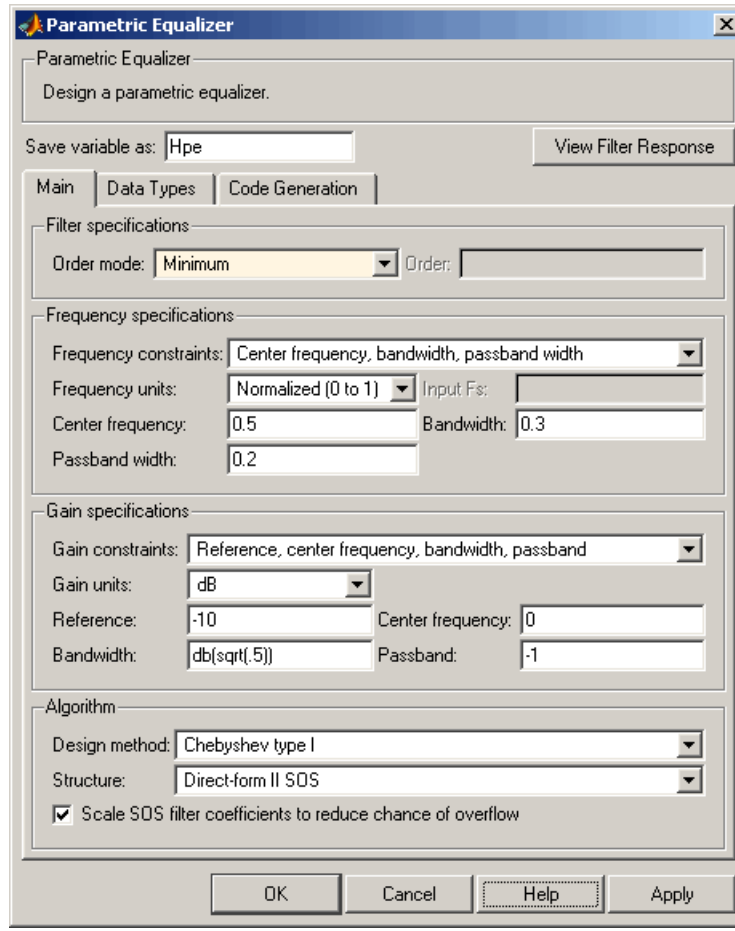
- Direct-form I SOS
- Direct-form II SOS
- Direct-form I transposed SOS
- Direct-form II transposed SOS

Scale SOS filter coefficients to reduce chance of overflow

Select the check box to scale the filter coefficients.

Parametric Equalizer Filter Design Dialog Box — Main Pane

Filter Specifications



Filter Specifications

Order mode

Select `Minimum` for minimum filter order, or `Specify` to enter a specific filter order. The order mode also affects the possible

frequency constraints, which in turn limit the gain specifications. For example, if you specify a `Minimum` order filter, then the available frequency constraints are: `Center frequency, bandwidth, passband width` and `Center frequency, bandwidth, stopband width`. However, if you select a specific filter order, then the list of frequency constraints consists of: `Center frequency, bandwidth` and `Low frequency, high frequency`.

Order

This parameter is enabled only if the**Order mode** is set to `Specify`. Enter the filter order in this text box.

Frequency specifications

Depending on the filter order, the possible frequency constraints change. Also, once you choose the frequency constraints the input boxes in this area change to reflect the selection.

Frequency constraints

Select the specification array to represent frequency constraints. The following options are available:

- `Center frequency, bandwidth, passband width` (available for minimum order only)
- `Center frequency, bandwidth, stopband width` (available for minimum order only)
- `Center frequency, bandwidth` (available for a custom specified order only)
- `Low frequency, high frequency` (available for a custom specified order only)

Frequency units

Select the frequency units from the available drop down list (`Normalized, Hz, kHz, MHz, GHz`). If `Normalized` is selected, then the **Input Fs** box is disabled for input.

Input Fs

Enter the input sampling frequency. This input box is disabled for input if `Normalized` is selected in the **Frequency units** input box.

Center frequency

Enter the center frequency, either normalized, or in the units selected previously.

Bandwidth

Enter the bandwidth.

Passband width

Enter the passband width. This option is enabled only if the filter is of minimum order, and the frequency constraint selected is `Center frequency, bandwidth, passband width`.

Stopband width

Enter the stopband width. This option is enabled only if the filter is of minimum order, and the frequency constraint selected is `Center frequency, bandwidth, stopband width`.

Low frequency

Enter the low frequency. This option is enabled only if the filter order is user specified and the frequency constraint selected is `Low frequency, high frequency`

High frequency

Enter the high frequency. This option is enabled only if the filter order is user specified and the frequency constraint selected is `Low frequency, high frequency`

Gain Specifications

Depending on the filter order and frequency constraints, the possible gain constraints change. Also, once you choose the gain constraints the input boxes in this area change to reflect the selection.

Gain constraints

Select the specification array to represent gain constraints, and remember that not all of these options are available for all configurations. The following is a list of all available options:

- `Reference, center frequency, bandwidth, passband`
- `Reference, center frequency, bandwidth, stopband`
- `Reference, center frequency, bandwidth, passband, stopband`
- `Reference, center frequency, bandwidth`

Gain units

Specify the gain units either `dB` or `squared`. These units are used for all gain specifications in the dialog box.

Reference

Enter the reference gain.

Bandwidth

Enter the bandwidth.

Center frequency

Enter the center frequency (in the units specified in **Frequency units** input box)

Passband

Specify the passband gain. This input is enabled only for specific configurations.

Stopband

Specify the stopband gain. This input is enabled only for specific configurations.

Algorithm

Design method

Select the design method from the drop-down list. Different methods are available depending on the chosen filter constraints.

Structure

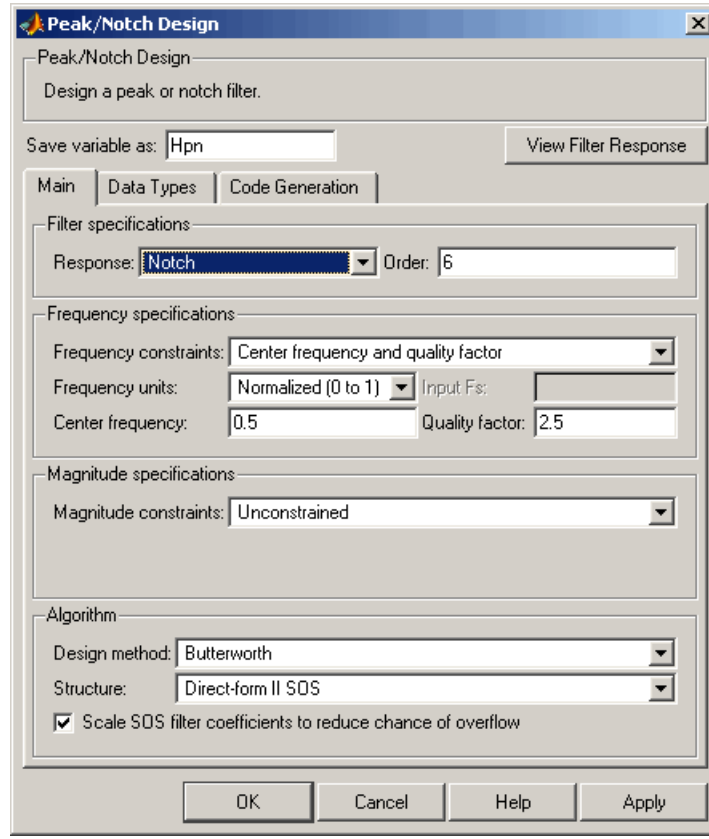
Select filter structure. The possible choices are:

- Direct-form I SOS
- Direct-form II SOS
- Direct-form I transposed SOS
- Direct-form II transposed SOS

Scale SOS filter coefficients to reduce chance of overflow

Select the check box to scale the filter coefficients.

Peak/Notch Filter Design Dialog Box — Main Pane



Filter Specifications

In this area you can specify whether you want to design a peaking filter or a notching filter, as well as the order of the filter.

Response

Select `Peak` or `Notch` from the drop-down box. The rest of the parameters that specify are equivalent for either filter type.

Order

Enter the filter order. The order must be even.

Frequency Specifications

This group of parameters allows you to specify frequency constraints and units.

Frequency Constraints

Select the frequency constraints for filter specification. There are two choices as follows:

- `Center frequency and quality factor`
- `Center frequency and bandwidth`

Frequency units

The frequency units are normalized by default. If you specify units other than normalized, `filterbuilder` assumes that you wish to specify an input sampling frequency, and enables this input box. The choice of frequency units are: `Normalized (0 to 1), Hz, kHz, MHz, GHz`.

Input Fs

This input box is enabled if **Frequency units** other than `Normalized (0 to 1)` are specified. Enter the input sampling frequency.

Center frequency

Enter the center frequency in the units specified previously.

Quality Factor

This input box is enabled only when `Center frequency and quality factor` is chosen for the **Frequency Constraints**. Enter the quality factor.

Bandwidth

This input box is enabled only when `Center frequency and bandwidth` is chosen for the **Frequency Constraints**. Enter the bandwidth.

Magnitude Specifications

This group of parameters allows you to specify the magnitude constraints, as well as their values and units.

Magnitude Constraints

Depending on the choice of constraints, the other input boxes are enabled or disabled. Select from four magnitude constraints available:

- `Unconstrained`
- `Passband ripple`
- `Stopband attenuation`
- `Passband ripple and stopband attenuation`

Magnitude units

Select the magnitude units: either `dB` or `squared`.

Apass

This input box is enabled if the magnitude constraints selected are `Passband ripple` or `Passband ripple and stopband attenuation`. Enter the passband ripple.

Astop

This input box is enabled if the magnitude constraints selected are `Stopband attenuation` or `Passband ripple and stopband attenuation`. Enter the stopband attenuation.

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterbuilder` uses to implement your filter.

Design Method

Lists all design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter the methods available to design filters changes as well.

Structure

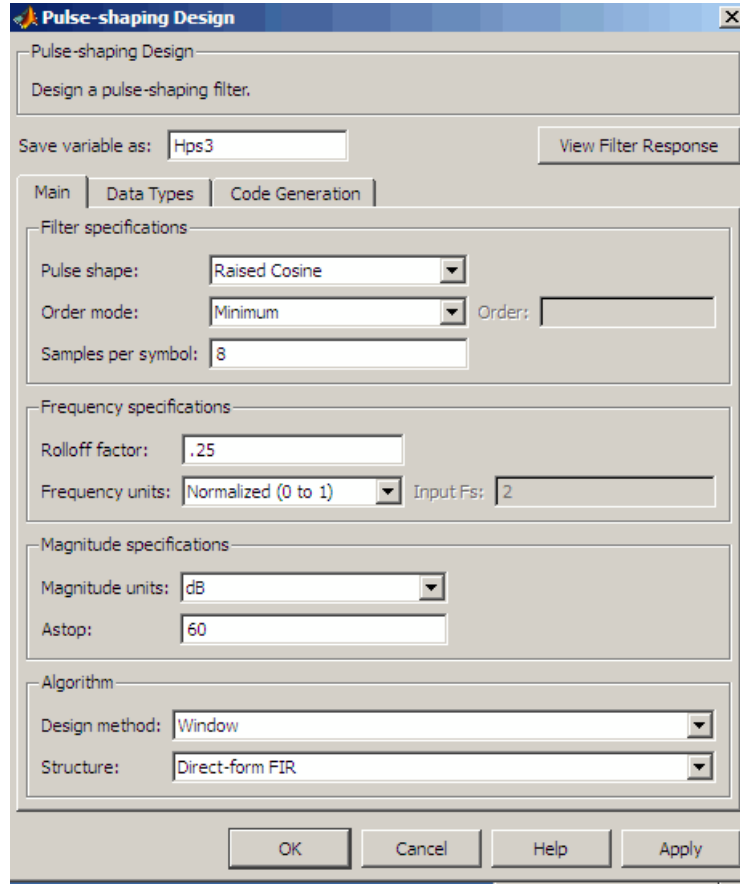
Lists all available filter structures for the filter specifications and design method you select. The typical options are:

- Direct-form I SOS
- Direct-form II SOS
- Direct-form I transposed SOS
- Direct-form II transposed SOS

Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Pulse-shaping Filter Design Dialog Box—Main Pane



Filter Specifications

Parameters in this group enable you to specify the shape and length of the filter.

Pulse shape

Select the shape of the impulse response from the following options:

- `Raised Cosine`
- `Square Root Raised Cosine`
- `Gaussian`

Order mode

This specification is only available for raised cosine and square root raised cosine filters. For these filters, select one of the following options:

- `Minimum`— This option will result in the minimum-length filter satisfying the user-specified **Frequency specifications**.
- `Specify order`—This option allows the user to construct a raised cosine or square root cosine filter of a specified order by entering an even number in the **Order** input box. The length of the impulse response will be `Order+1` .
- `Specify symbols`—This option enables the user to specify the length of the impulse response in an alternative manner. If `Specify symbols` is chosen, the **Order** input box changes to the **Number of symbols** input box.

Samples per symbol

Specify the oversampling factor. Increasing the oversampling factor guards against aliasing and improves the FIR filter approximation to the ideal frequency response. If **Order** is specified in **Number of symbols**, the filter length will be **Number of symbols*Samples per symbol**+1. The product **Number of symbols*Samples per symbol** must be an even number.

If a Gaussian filter is specified, the filter length must be specified in **Number of symbols** and **Samples per symbol**. The product **Number of symbols*Samples per symbol** must be an even number. The filter length will be **Number of symbols*Samples per symbol**+1.

Frequency specifications

Parameters in this group enable you to specify the frequency response of the filter. For raised cosine and square root raised cosine filters, the frequency specifications include:

`Rolloff factor`
: The rolloff factor takes values in the range [0,1]. The smaller the rolloff factor, the steeper the transition in the stopband.

`Frequency units`
: The frequency units are normalized by default. If you specify units other than normalized, `filterbuilder` assumes that you wish to specify an input sampling frequency, and enables this input box. The choice of frequency units are: `Normalized (0 to 1), Hz, kHz, MHz, GHz`

For a Gaussian pulse shape, the available frequency specifications are:

`Bandwidth-time product`
: This option allows the user to specify the width of the Gaussian filter. Note that this is independent of the length of the filter. The bandwidth-time product (`BT`) must be a positive real number. Smaller values of the bandwidth-time product result in larger pulse widths in time and steeper stopband transitions in the frequency response.

`Frequency units`
: The frequency units are normalized by default. If you specify units other than normalized, `filterbuilder` assumes that you wish to specify an input sampling frequency, and enables this input box. The choice of frequency units are: `Normalized (0 to 1), Hz, kHz, MHz, GHz`

Magnitude specifications

If the **Order mode** is specified as `minimum`, the magnitude units may be selected from:

- `dB`—Specify the magnitude in decibels (default).

- `Linear`—Specify the magnitude in linear units.

Algorithm

The only design method available for FIR pulse-shaping filters is the window method.

Purpose

2-norm or infinity-norm of digital filter

Syntax

```
filternorm(b,a)
filternorm(b,a,pnorm)
filternorm(b,a,2,tol)
```

Description

A typical use for filter norms is in digital filter scaling to reduce quantization effects. Scaling often improves the signal-to-noise ratio of the filter without resulting in data overflow. You, also, can use the 2-norm to compute the energy of the impulse response of a filter.

`filternorm(b,a)` computes the 2-norm of the digital filter defined by the numerator coefficients in `b` and denominator coefficients in `a`.

`filternorm(b,a,pnorm)` computes the 2- or infinity-norm (inf-norm) of the digital filter, where `pnorm` is either 2 or `inf`.

`filternorm(b,a,2,tol)` computes the 2-norm of an IIR filter with the specified tolerance, `tol`. The tolerance can be specified only for IIR 2-norm computations. `pnorm` in this case must be `2`. If `tol` is not specified, it defaults to `1e-8`.

Examples

Compute the 2-norm with a tolerance of 1e-10 of an IIR filter:

```
[b,a]=butter(5,.5);
L2=filternorm(b,a,2,1e-10)

L2 =

    0.7071
```

Compute the inf-norm of an FIR filter:

```
b=firpm(30,[.1 .9],[1 1],'Hilbert');
Linf=filternorm(b,1,inf)

Linf =
```

```
1.0028
```

Algorithm

Given a filter H(z) with frequency reponse $H(e^{j\omega})$, the L_p-norm is given by

$$\|H\|_p \equiv \left[\frac{1}{2\pi}\int_{-\pi}^{\pi}\left|H(e^{j\omega})\right|^p d\omega\right]^{\frac{1}{p}}$$

For the case $p = \infty$, the L_∞ norm simplifies to

$$\|H\|_\infty = \max_{-\pi \le \omega \le \pi}\left|H(e^{j\omega})\right|$$

For the case $p = 2$, Parseval's theorem states that

$$\|H\|_2 = \left[\frac{1}{2\pi}\int_{-\pi}^{\pi}\left|H(e^{j\omega})\right|^2 d\omega\right]^{\frac{1}{2}} = \left[\sum_{n=-\infty}^{\infty}|h(n)|^2\right]^{\frac{1}{2}}$$

where h(n) is the impulse response of the filter. The energy of the impulse response, then, is $\|H\|_2^2$.

Reference

[1] Jackson, L.B., *Digital Filters and Signal Processing, Third Edition,* Kluwer Academic Publishers, 1996, Chapter 11.

See Also

`zp2sos`, `norm`

Purpose

Zero-phase digital filtering

Syntax

```
y = filtfilt(b,a,x)
```

Description

`y = filtfilt(b,a,x)` performs zero-phase digital filtering by processing the input data in both the forward and reverse directions (see problem 5.39 in [1]). After filtering in the forward direction, it reverses the filtered sequence and runs it back through the filter. The result has:

- Precisely zero-phase distortion
- Magnitude that is the square of the filter's magnitude response
- Filter order that is double the order of the filter specified by `b` and `a`

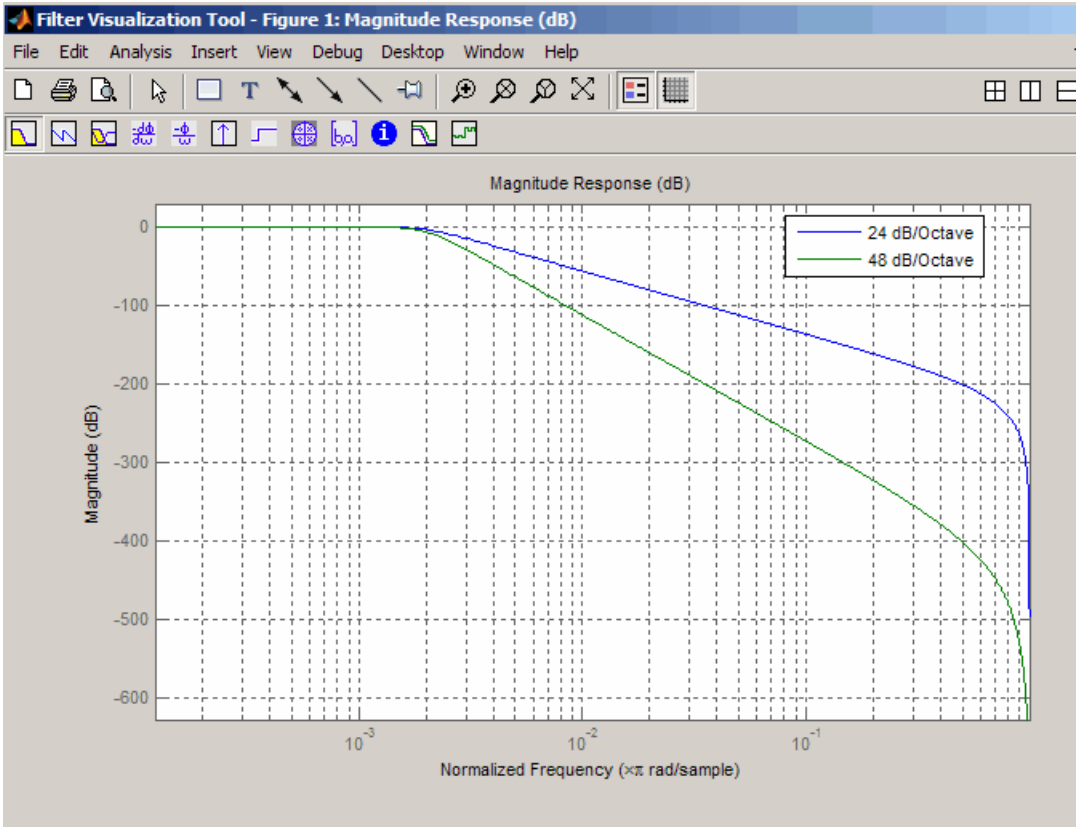
`filtfilt` minimizes start-up and ending transients by matching initial conditions, and works for both real and complex inputs. Note that `filtfilt` should not be used with differentiator and Hilbert FIR filters, since the operation of these filters depends heavily on their phase response.

Note The length of the input `x` must be more than three times the filter order, which is defined `max(length(b)-1,length(a)-1)`. The input `x` should be large enough so that the impulse is correctly represented. For example, for a fifth order filter, if the input sequence is a delta sequence, the 1 value should appear within the first 15 samples.

Examples

This example demonstrates the `filtfilt` squared magnitude response. (Note that `dfilt.df2` uses `filtfilt`.)

```
[b,a] = butter(4,.002);
h = dfilt.df2(b,a);
hc = cascade(h,h); % hc magnitude = h magnitude squared
hfvt = fvtool(h,hc,'FrequencyScale','log');
legend(hfvt,'24 dB/Octave','48 dB/Octave')
```



Algorithm

`filtfilt` is an M-file that uses the `filter` function. In addition to the forward-reverse filtering, it attempts to minimize startup transients by adjusting initial conditions to match the DC component of the signal and by prepending several filter lengths of a flipped, reflected copy of the input signal.

References

[1] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing,* Prentice-Hall, 1989, pp. 284—285.

[2] Mitra, S.K., *Digital Signal Processing, 2nd ed.,* McGraw-Hill, 2001, Sections 4.4.2 and 8.2.5.

[3] Gustafsson, F., Determining the initial states in forward-backward filtering, *IEEE Transactions on Signal Processing*, April 1996, Volume 44, Issue 4, pp. 988–992.

See Also

`fftfilt`, `filter`, `filter2`

Purpose

Initial conditions for transposed direct-form II filter implementation

Syntax

```
z = filtic(b,a,y,x)
z = filtic(b,a,y)
```

Description

`z = filtic(b,a,y,x)` finds the initial conditions, `z`, for the delays in the *transposed direct-form II* filter implementation given past outputs `y` and inputs `x`. The vectors `b` and `a` represent the numerator and denominator coefficients, respectively, of the filter's transfer function.

The vectors `x` and `y` contain the most recent input or output first, and oldest input or output last.

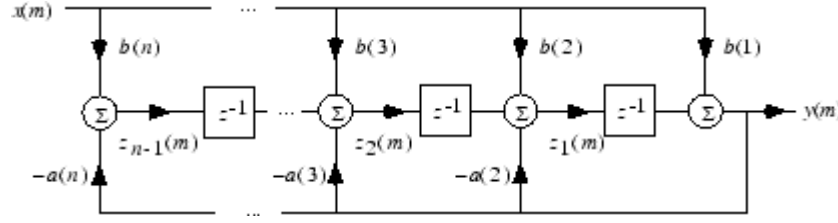
$$x = [x(-1), x(-2), x(-3), \ldots, x(-n), \ldots]$$
$$y = [y(-1), y(-2), y(-3), \ldots, y(-m), \ldots]$$

where `n` is `length(b)-1` (the numerator order) and `m` is `length(a)-1` (the denominator order). If `length(x)` is less than `n`, `filtic` pads it with zeros to length `n`; if `length(y)` is less than `m`, `filtic` pads it with zeros to length `m`. Elements of `x` beyond `x(n-1)` and elements of `y` beyond `y(m-1)` are unnecessary so `filtic` ignores them.

Output `z` is a column vector of length equal to the larger of n and m. `z` describes the state of the delays given past inputs `x` and past outputs `y`.

`z = filtic(b,a,y)` assumes that the input `x` is 0 in the past.

The transposed direct-form II structure is shown in the following illustration.



n-1 is the filter order.

`filtic` works for both real and complex inputs.

Algorithm

`filtic` performs a reverse difference equation to obtain the delay states `z`.

Diagnostics

If any of the input arguments `y`, `x`, `b`, or `a` is not a vector (that is, if any argument is a scalar or array), `filtic` gives the following error message:

```
Requires vector inputs.
```

References

[1] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, pp. 296, 301-302.

See Also

`filter`, `filtfilt`

filtstates

Purpose

Filter states

Syntax

```
Hs = filtstates.structure(input1,...)
```

Description

`Hs = filtstates.structure(input1,...)` returns a filter states object `Hs`, which contains the filter states.

You can extract a `filtstates` object from the states property of an object with

```
Hd = dfilt.df1
Hs = Hd.states
```

or, for an `mfilt` object in the Filter Design Toolbox product, with

```
Hm = mfilt.cicdecim
Hs = Hm.states
```

Structures

Structures for `filtstates` specify the type of filter structure. Available types of structures for `filtstates` are shown below.

filtstates.structure	Description
`filtstates.dfiir`	filtstates for IIR direct-form I filters (`dfilt.df1`, `dfilt.df1t`, `dfilt.df1sos`, and `dfilt.df1tsos`)
`filtstates.cic`	filtstates for cascaded integrator comb filters. (Available only with Filter Design Toolbox and Fixed-Point Toolbox products.)

Refer to the particular `filtstates.structure` reference page or use the syntax `help filtstates.structure` at the MATLAB prompt for more information.

See Also

`filtstates.dfiir`, `dfilt`, `dfilt.df1`, `dfilt.df1t`, `dfilt.df1sos`, `dfilt.df1tsos`

filtstates.dfiir

Purpose

IIR direct-form filter states

Syntax

```
Hs = filtstates.dfiir(numstates,denstates)
```

Description

`Hs = filtstates.dfiir(numstates,denstates)` returns an IIR direct-form filter states object `Hs` with two properties — `Numerator` and `Denominator`, which contain the filter states. These two properties are column vectors with each column representing a separate channel of filter states. The number of states is always one less than the number of filter numerator or denominator coefficients.

You can extract a filtstates object from the states property of an IIR direct-form I object with

```
Hd = dfilt.df1
Hs = Hd.states
```

Methods

You can use the following methods on a `filtstates.dfiir` object.

Method	Description
`double`	Converts a `filtstates` object to a double-precision vector containing the values of the numerator and denominator states. The numerator states are listed first in this vector, followed by the denominator states.
`single`	Converts a `filtstates` object to a single-precision vector containing the values of the numerator and denominator states. (This method is used with the Filter Design Toolbox product.)

Examples

This example demonstrates the interaction of `filtstates` with a `dfilt.df1` object.

```
[b,a] = butter(4,0.5);    % Design butterworth filter
```

```
Hd = dfilt.df1(b,a);      % Create dfilt object
Hs = Hd.states            % Extract filter states object
                          %  from dfilt states property
Hs.Numerator = [1,1,1,1]  % Modify numerator states
Hd.states = Hs            % Set modified states back to
                          %  original object

Dbl = double(Hs)          % Create double vector from
                          %  states
```

See Also

filtstates, dfilt, dfilt.df1, dfilt.df1t, dfilt.df1sos, dfilt.df1tsos

findpeaks

Purpose

Find local maxima

Syntax

```
pks = findpeaks(x)
[pks,locs] = findpeaks(x)
[...] = findpeaks(x,'minpeakheight',mph)
[...] = findpeaks(x,'minpeakdistance',mpd)
[...] = findpeaks(x,'threshold',th)
[...] = findpeaks(x,'npeaks',np)
[...] = findpeaks(x,'sortstr',str)
```

Description

`pks = findpeaks(x)` finds local maxima or peaks in `x`, which must contain at least three samples. Each value of `x` is compared to its neighboring values, and if it is larger than both of its neighbors, it is a local peak and is returned in the vector `pks`. If no peaks are found, `findpeaks` returns an empty vector.

`[pks,locs] = findpeaks(x)` returns, in the `locs` vector, the location index of each peak in `x`.

`[...] = findpeaks(x,'minpeakheight',mph)` returns only peaks that are greater than the minimum peak height `mph`, where `mph` is a real, scalar value. Default for `mph` is `-Inf`. Setting the minimum peak height may reduce the number of peaks returned and the overall processing time.

`[...] = findpeaks(x,'minpeakdistance',mpd)` returns only peaks that are separated by the minimum peak distance `mpd`. The minimum peak distance is a positive integer that specifies the number of data values to ignore around a peak in `x`. Setting the minimum peak distance ignores smaller peaks that may occur close to larger local peaks. For example, if a large local peak occurs at *N*, any other peaks in (*N*-`mpd`, *N*+`mpd`) are ignored. Default for `mpd` is 1.

`[...] = findpeaks(x,'threshold',th)` returns only peaks that are greater than their neighbors by at least the threshold `th`, which is a real, scalar value and is greater than or equal to 0. Default for `th` is 0.

`[...] = findpeaks(x,'npeaks',np)` returns a maximum of `np` number of peaks. When `np` peaks are found, the search stops. Default is to return all peaks.

`[...] = findpeaks(x,'sortstr',str)` specifies the sorting order, where `str` is `'ascend'`, `'descend'` or `'none'`. For `'ascend'`, the peaks are returned in order from smallest to largest, and vice versa for `'descend'`. For `'none'`, the peaks are returned in the order in which they occur in `x`.

Examples

```
x = [2 12 4 6 9 4 3 1 19 7];
pks = findpeaks(x)

pks =
    12     9    19

pks = findpeaks(x,'threshold',6)

pks =
    12    19

pks = findpeaks(x,'threshold',6,'sortstr','descend')
pks =
    19    12
```

Purpose

Window-based finite impulse response filter design

Syntax

```
b = fir1(n,Wn)
b = fir1(n,Wn,'ftype')
b = fir1(n,Wn,window)
b = fir1(n,Wn,'ftype',window)
b = fir1(...,'normalization')
```

Description

`fir1` implements the classical method of windowed linear-phase FIR digital filter design [1]. It designs filters in standard lowpass, highpass, bandpass, and bandstop configurations. By default the filter is normalized so that the magnitude response of the filter at the center frequency of the passband is 0 dB.

Note Use `fir2` for windowed filters with arbitrary frequency response.

`b = fir1(n,Wn)` returns row vector `b` containing the `n+1` coefficients of an order `n` lowpass FIR filter. This is a Hamming-window based, linear-phase filter with normalized cutoff frequency `Wn`. The output filter coefficients, `b`, are ordered in descending powers of z.

$$B(z) = b(1) + b(2)z^{-1} + \cdots + b(n+1)z^{-n}$$

`Wn` is a number between 0 and 1, where 1 corresponds to the Nyquist frequency.

If `Wn` is a two-element vector, `Wn = [w1 w2]`, `fir1` returns a bandpass filter with passband `w1` < ω< `w2`.

If `Wn` is a multi-element vector, `Wn = [w1 w2 w3 w4 w5 ... wn]`, `fir1` returns an order `n` multiband filter with bands `0` < ω< `w1`, `w1` < ω< `w2`, ..., `wn` < ω< `1`.

By default, the filter is scaled so that the center of the first passband has a magnitude of exactly 1 after windowing.

`b = fir1(n,Wn,'ftype')` specifies a filter type, where `'ftype'` is:

- `'high'` for a highpass filter with cutoff frequency `Wn`.
- `'stop'` for a bandstop filter, if `Wn = [w1 w2]`. The stopband frequency range is specified by this interval.
- `'DC-1'` to make the first band of a multiband filter a passband.
- `'DC-0'` to make the first band of a multiband filter a stopband.

`fir1` always uses an even filter order for the highpass and bandstop configurations. This is because for odd orders, the frequency response at the Nyquist frequency is 0, which is inappropriate for highpass and bandstop filters. If you specify an odd-valued `n`, `fir1` increments it by 1.

`b = fir1(n,Wn,window)` uses the window specified in column vector `window` for the design. The vector `window` must be `n+1` elements long. If no window is specified, `fir1` uses a Hamming window (see `hamming`) of length `n+1`.

`b = fir1(n,Wn,'ftype',window)` accepts both *`'ftype'`* and `window` parameters.

`b = fir1(...,'normalization')` specifies whether or not the filter magnitude is normalized. The string *`'normalization'`* can be the following:

- `'scale'` (default): Normalize the filter so that the magnitude response of the filter at the center frequency of the passband is 0 dB.
- `'noscale'`: Do not normalize the filter.

The group delay of the FIR filter designed by `fir1` is `n/2`.

Algorithm

`fir1` uses the window method of FIR filter design [1]. If $w(n)$ denotes a window, where $1 \le n \le N$, and the impulse response of the ideal filter is $h(n)$, where $h(n)$ is the inverse Fourier transform of the ideal frequency response, then the windowed digital filter coefficients are given by

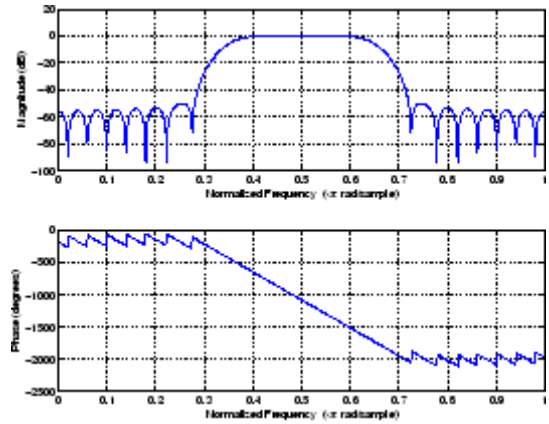
$$b(n) = w(n)h(n), \quad 1 \le n \le N$$

Examples

Example 1

Design a 48th-order FIR bandpass filter with passband $0.35 \le \omega \le 0.65$:

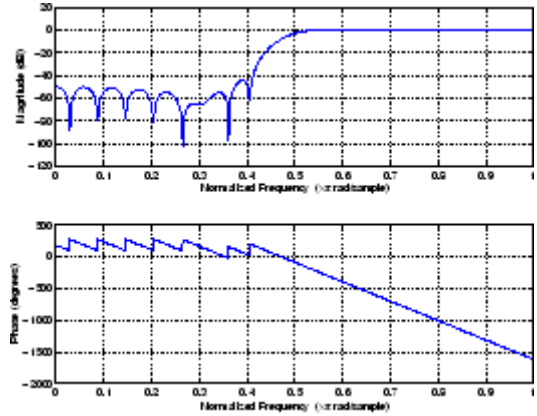
```
b = fir1(48,[0.35 0.65]);
freqz(b,1,512)
```



Example 2

The `chirp.mat` file contains a signal, `y`, that has most of its power above `fs/4`, or half the Nyquist frequency. Design a 34th-order FIR highpass filter to attenuate the components of the signal below `fs/4`. Use a cutoff frequency of 0.48 and a Chebyshev window with 30 dB of ripple:

```
load chirp          % Load y and fs.
b = fir1(34,0.48,'high',chebwin(35,30));
freqz(b,1,512)
```



References

[1] *Programs for Digital Signal Processing*, IEEE Press, New York, 1979. Algorithm 5.2.

See Also

`cfirpm`, `filter`, `fir2`, `fircls`, `fircls1`, `firls`, `freqz`, `kaiserord`, `firpm`, `window`

Purpose

Frequency sampling-based finite impulse response filter design

Syntax

```
b = fir2(n,f,m)
b = fir2(n,f,m,window)
b = fir2(n,f,m,npt)
b = fir2(n,f,m,npt,window)
b = fir2(n,f,m,npt,lap)
b = fir2(n,f,m,npt,lap,window)
```

Description

`fir2` designs frequency sampling-based digital FIR filters with arbitrarily shaped frequency response.

Note Use `fir1` for windows-based standard lowpass, bandpass, highpass, and bandstop configurations.

`b = fir2(n,f,m)` returns row vector `b` containing the `n+1` coefficients of an order `n` FIR filter. The frequency-magnitude characteristics of this filter match those given by vectors `f` and `m`:

- `f` is a vector of frequency points in the range from 0 to 1, where 1 corresponds to the Nyquist frequency. The first point of `f` must be `0` and the last point `1`. The frequency points must be in increasing order.
- `m` is a vector containing the desired magnitude response at the points specified in `f`.
- `f` and `m` must be the same length.
- Duplicate frequency points are allowed, corresponding to steps in the frequency response.

Use `plot(f,m)` to view the filter shape.

The output filter coefficients, `b`, are ordered in descending powers of z.

$$b(z) = b(1) + b(2)z^{-1} + \cdots + b(n+1)z^{-n}$$

`fir2` always uses an even filter order for configurations with a passband at the Nyquist frequency. This is because for odd orders, the frequency response at the Nyquist frequency is necessarily 0. If you specify an odd-valued `n`, `fir2` increments it by 1.

`b = fir2(n,f,m,window)` uses the window specified in the column vector `window`. The vector `window` must be `n+1` elements long. If no window is specified, `fir2` uses a Hamming window (see `hamming`) of length `n+1`.

`b = fir2(n,f,m,npt)` or

`b = fir2(n,f,m,npt,window)` specifies the number of points, `npt`, for the grid onto which `fir2` linearly interpolates the frequency response, with or without `window` specification. If desired, you can interpolate `f` and `m` before passing them to `fir2`.

`b = fir2(n,f,m,npt,lap)` and

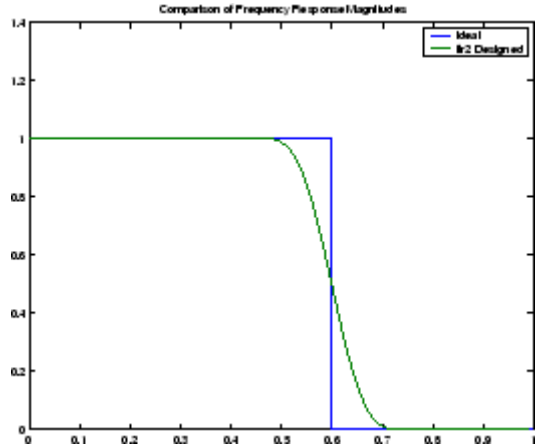
`b = fir2(n,f,m,npt,lap,window)` specify the size of the region, `lap`, that `fir2` inserts around duplicate frequency points, with or without a `window` specification.

See “Algorithm” on page 10-472 for more on `npt` and `lap`.

Examples

Design a 30th-order lowpass filter and overplot the desired frequency response with the actual frequency response:

```
f = [0 0.6 0.6 1]; m = [1 1 0 0];
b = fir2(30,f,m);
[h,w] = freqz(b,1,128);
plot(f,m,w/pi,abs(h))
legend('Ideal','fir2 Designed')
title('Comparison of Frequency Response Magnitudes')
```



Algorithm

The desired frequency response is linearly interpolated onto a dense, evenly spaced grid of length `npt`. `npt` is 512 by default. If two successive values of `f` are the same, a region of `lap` points is set up around this frequency to provide a smooth but steep transition in the requested frequency response. By default, `lap` is 25. The filter coefficients are obtained by applying an inverse fast Fourier transform to the grid and multiplying by a window; by default, this is a Hamming window.

References

[1] Mitra, S.K., *Digital Signal Processing A Computer Based Approach, First Edition*, McGraw-Hill, New York, 1998, pp. 462-468.

[2] Jackson, L.B., *Digital Filters and Signal Processing, Third Edition*, Kluwer Academic Publishers, Boston, 1996, pp. 301-307.

See Also

`butter`, `cheby1`, `cheby2`, `ellip`, `fir1`, `maxflat`, `firpm`, `yulewalk`

Purpose

Constrained least square, FIR multiband filter design

Syntax

```
b = fircls(n,f,amp,up,lo)
fircls(n,f,amp,up,lo,'design_flag')
```

Description

`b = fircls(n,f,amp,up,lo)` generates a length `n+1` linear phase FIR filter `b`. The frequency-magnitude characteristics of this filter match those given by vectors `f` and `amp`:

- `f` is a vector of transition frequencies in the range from 0 to 1, where 1 corresponds to the Nyquist frequency. The first point of `f` must be `0` and the last point `1`. The frequency points must be in increasing order.
- `amp` is a vector describing the piecewise constant desired amplitude of the frequency response. The length of `amp` is equal to the number of bands in the response and should be equal to `length(f)-1`.
- `up` and `lo` are vectors with the same length as `amp`. They define the upper and lower bounds for the frequency response in each band.

`fircls` always uses an even filter order for configurations with a passband at the Nyquist frequency (that is, highpass and bandstop filters). This is because for odd orders, the frequency response at the Nyquist frequency is necessarily 0. If you specify an odd-valued `n`, `fircls` increments it by 1.

`fircls(n,f,amp,up,lo,'design_flag')` enables you to monitor the filter design, where '*design_flag*' can be

- `'trace'`, for a textual display of the design error at each iteration step.
- `'plots'`, for a collection of plots showing the filter's full-band magnitude response and a zoomed view of the magnitude response in each sub-band. All plots are updated at each iteration step. The O's on the plot are the estimated extremals of the new iteration and the X's are the estimated extremals of the previous iteration, where the extremals are the peaks (maximum and minimum) of the filter

ripples. Only ripples that have a corresponding O and X are made equal.

- `'both'`, for both the textual display and plots.

Examples

Design an order 150 lowpass filter:

```
n=150;
f=[0 0.4 1];
a=[1 0];
up=[1.02 0.01];
lo =[0.98 -0.01];
b = fircls(n,f,a,up,lo,'both');   % Display plots of bands
  Bound Violation = 0.0788344298966
  Bound Violation = 0.0096137744998
  Bound Violation = 0.0005681345753
  Bound Violation = 0.0000051519942
  Bound Violation = 0.0000000348656
  Bound Violation = 0.0000000006231
%  The above Bound Violations indicate iterations as
%  the design converges.
fvtool(b)                         % Display magnitude plot
```

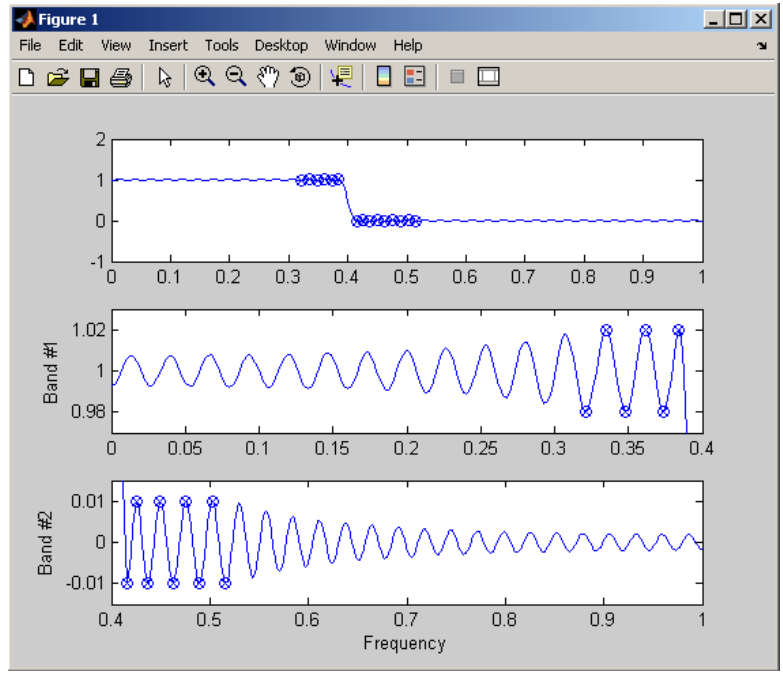
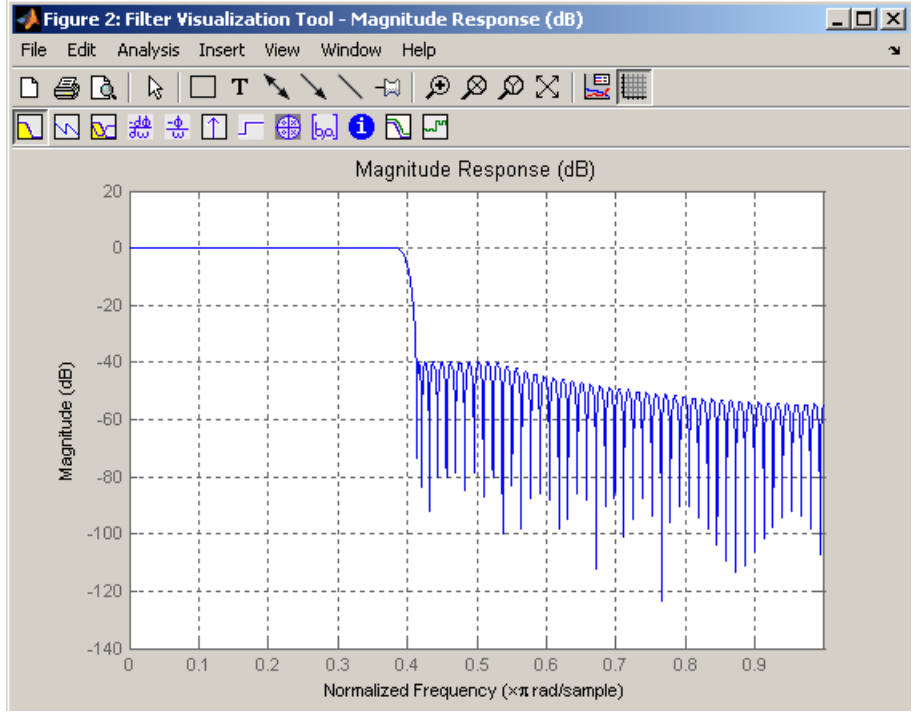


Figure 1
File Edit View Insert Tools Desktop Window Help
2
1
0
-1
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
Band #1
1.02
1
0.98
0 0.05 0.1 0.15 0.2 0.25 0.3 0.35 0.4
Band #2
0.01
0
-0.01
0.4 0.5 0.6 0.7 0.8 0.9 1
Frequency



Note Normally, the lower value in the stopband will be specified as negative. By setting `lo` equal to `0` in the stopbands, a nonnegative frequency response amplitude can be obtained. Such filters can be spectrally factored to obtain minimum phase filters.

Algorithm

`fircls` uses an iterative least-squares algorithm to obtain an equiripple response. The algorithm is a multiple exchange algorithm that uses Lagrange multipliers and Kuhn-Tucker conditions on each iteration.

References

[1] Selesnick, I.W., M. Lang, and C.S. Burrus, “Constrained Least Square Design of FIR Filters without Specified Transition

Bands," *Proceedings of the IEEE Int. Conf. Acoust., Speech, Signal Processing, Vol. 2* (May 1995), pp. 1260-1263.

[2] Selesnick, I.W., M. Lang, and C.S. Burrus. "Constrained Least Square Design of FIR Filters without Specified Transition Bands." *IEEE Transactions on Signal Processing, Vol. 44*, No. 8 (August 1996).

See Also

`fircls1`, `firls`, `firpm`

Purpose

Constrained least square, lowpass and highpass, linear phase, FIR filter design

Syntax

```
b = fircls1(n,wo,dp,ds)
b = fircls1(n,wo,dp,ds,'high')
b = fircls1(n,wo,dp,ds,wt)
b = fircls1(n,wo,dp,ds,wt,'high')
b = fircls1(n,wo,dp,ds,wp,ws,k)
b = fircls1(n,wo,dp,ds,wp,ws,k,'high')
b = fircls1(n,wo,dp,ds,...,'design_flag')
```

Description

`b = fircls1(n,wo,dp,ds)` generates a lowpass FIR filter `b`, where `n+1` is the filter length, `wo` is the normalized cutoff frequency in the range between 0 and 1 (where 1 corresponds to the Nyquist frequency), `dp` is the maximum passband deviation from 1 (passband ripple), and `ds` is the maximum stopband deviation from 0 (stopband ripple).

`b = fircls1(n,wo,dp,ds,'high')` generates a highpass FIR filter `b`. `fircls1` always uses an even filter order for the highpass configuration. This is because for odd orders, the frequency response at the Nyquist frequency is necessarily 0. If you specify an odd-valued `n`, `fircls1` increments it by 1.

`b = fircls1(n,wo,dp,ds,wt)` and

`b = fircls1(n,wo,dp,ds,wt,'high')` specifies a frequency `wt` above which (for `wt` > `wo`) or below which (for `wt` < `wo`) the filter is guaranteed to meet the given band criterion. This will help you design a filter that meets a passband or stopband edge requirement. There are four cases:

- Lowpass:
  - `0` < `wt` < `wo` < `1`: the amplitude of the filter is within `dp` of 1 over the frequency range `0` < ω < `wt`.
  - `0` < `wo` < `wt` < `1`: the amplitude of the filter is within `ds` of 0 over the frequency range `wt` < ω < `1`.
- Highpass:

- `0 < wt < wo < 1`: the amplitude of the filter is within `ds` of 0 over the frequency range `0 < ω < wt`.
- `0 < wo < wt < 1`: the amplitude of the filter is within `dp` of 1 over the frequency range `wt < ω < 1`.

`b = fircls1(n,wo,dp,ds,wp,ws,k)` generates a lowpass FIR filter `b` with a weighted function, where `n+1` is the filter length, `wo` is the normalized cutoff frequency, `dp` is the maximum passband deviation from 1 (passband ripple), and `ds` is the maximum stopband deviation from 0 (stopband ripple). `wp` is the passband edge of the L2 weight function and `ws` is the stopband edge of the L2 weight function, where `wp < wo < ws`. `k` is the ratio (passband L2 error)/(stopband L2 error)

$$k = \frac{\int_0^{\omega_p} |A(\omega) - D(\omega)|^2 d\omega}{\int_{\omega_s}^{\pi} |A(\omega) - D(\omega)|^2 d\omega}$$

`b = fircls1(n,wo,dp,ds,wp,ws,k,'high')` generates a highpass FIR filter `b` with a weighted function, where `ws < wo < wp`.

`b = fircls1(n,wo,dp,ds,...,'design_flag')` enables you to monitor the filter design, where `'design_flag'` can be

- `'trace'`, for a textual display of the design table used in the design
- `'plots'`, for plots of the filter's magnitude, group delay, and zeros and poles. All plots are updated at each iteration step. The O's on the plot are the estimated extremals of the new iteration and the X's are the estimated extremals of the previous iteration, where the extremals are the peaks (maximum and minimum) of the filter ripples. Only ripples that have a corresponding O and X are made equal.
- `'both'`, for both the textual display and plots

Note In the design of very narrow band filters with small `dp` and `ds`, there may not exist a filter of the given length that meets the specifications.

Examples

Design an order 55 lowpass filter with a cutoff frequency at 0.3:

```
n = 55;     wo = 0.3;
dp = 0.02;  ds = 0.008;
b = fircls1(n,wo,dp,ds,'both');    % Display plots of bands
    Bound Violation = 0.0870385343920
    Bound Violation = 0.0149343456540
    Bound Violation = 0.0056513587932
    Bound Violation = 0.0001056264205
    Bound Violation = 0.0000967624352
    Bound Violation = 0.0000000226538
    Bound Violation = 0.0000000000038
%  The above Bound Violations indicate iterations as
%  the design converges.
fvtool(b)            % Display magnitude plot
```

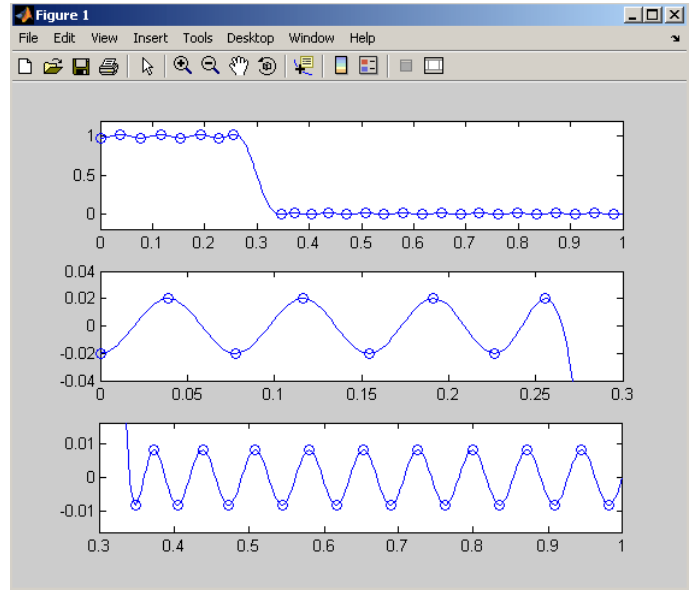


Figure 1
File Edit View Insert Tools Desktop Window Help
1
0.5
0
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
0.04
0.02
0
-0.02
-0.04
0 0.05 0.1 0.15 0.2 0.25 0.3
0.01
0
-0.01
0.3 0.4 0.5 0.6 0.7 0.8 0.9 1

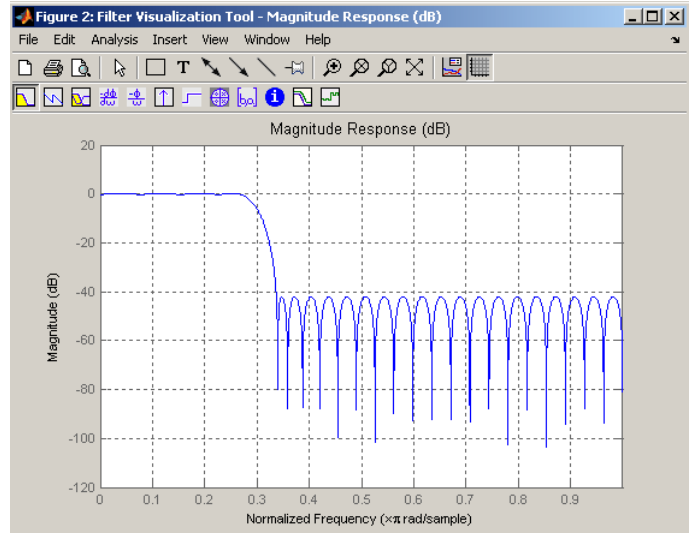


Figure 2: Filter Visualization Tool - Magnitude Response (dB)
File Edit Analysis Insert View Window Help
Magnitude Response (dB)
20
0
-20
-40
-60
-80
-100
-120
Magnitude (dB)
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
Normalized Frequency (×π rad/sample)

Algorithm

`fircls1` uses an iterative least-squares algorithm to obtain an equiripple response. The algorithm is a multiple exchange algorithm that uses Lagrange multipliers and Kuhn-Tucker conditions on each iteration.

References

[1] Selesnick, I.W., M. Lang, and C.S. Burrus, "Constrained Least Square Design of FIR Filters without Specified Transition Bands," *Proceedings of the IEEE Int. Conf. Acoust., Speech, Signal Processing, Vol. 2* (May 1995), pp.1260-1263.

[2] Selesnick, I.W., M. Lang, and C.S. Burrus, "Constrained Least Square Design of FIR Filters without Specified Transition Bands," *IEEE Transactions on Signal Processing, Vol. 44*, No. 8 (August 1996).

See Also

`fircls`, `firls`, `firpm`

Purpose

Least square linear-phase FIR filter design

Syntax

```
b = firls(n,f,a)
b = firls(n,f,a,w)
b = firls(n,f,a,'ftype')
b = firls(n,f,a,w,'ftype')
```

Description

`firls` designs a linear-phase FIR filter that minimizes the weighted, integrated squared error between an ideal piecewise linear function and the magnitude response of the filter over a set of desired frequency bands.

`b = firls(n,f,a)` returns row vector `b` containing the `n+1` coefficients of the order `n` FIR filter whose frequency-amplitude characteristics approximately match those given by vectors `f` and `a`. The output filter coefficients, or "taps," in `b` obey the symmetry relation.

$$b(k) = b(n+2-k), \qquad k = 1, \ldots, n+1$$

These are type I (`n` odd) and type II (`n` even) linear-phase filters. Vectors `f` and `a` specify the frequency-amplitude characteristics of the filter:

- `f` is a vector of pairs of frequency points, specified in the range between 0 and 1, where 1 corresponds to the Nyquist frequency. The frequencies must be in increasing order. Duplicate frequency points are allowed and, in fact, can be used to design a filter exactly the same as those returned by the `fir1` and `fir2` functions with a rectangular (`rectwin`) window.
- `a` is a vector containing the desired amplitude at the points specified in `f`.

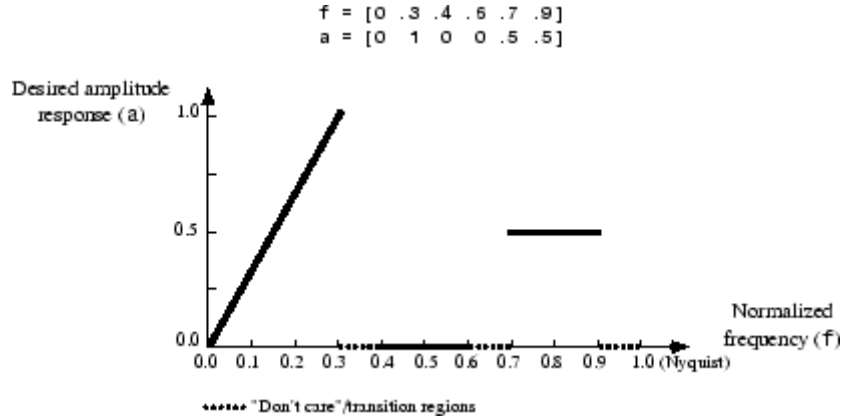
  The desired amplitude function at frequencies between pairs of points ($f(k)$, $f(k+1)$) for k odd is the line segment connecting the points ($f(k)$, $a(k)$) and ($f(k+1)$, $a(k+1)$).

  The desired amplitude function at frequencies between pairs of points ($f(k)$, $f(k+1)$) for k even is unspecified. These are transition or "don't care" regions.

- `f` and `a` are the same length. This length must be an even number.

`firls` always uses an even filter order for configurations with a passband at the Nyquist frequency. This is because for odd orders, the frequency response at the Nyquist frequency is necessarily 0. If you specify an odd-valued `n`, `firls` increments it by 1.

The figure below illustrates the relationship between the `f` and `a` vectors in defining a desired amplitude response.



`b = firls(n,f,a,w)` uses the weights in vector `w` to weight the fit in each frequency band. The length of `w` is half the length of `f` and `a`, so there is exactly one weight per band.

`b = firls(n,f,a,'ftype')` and

`b = firls(n,f,a,w,'ftype')` specify a filter type, where `'ftype'` is:

- `'hilbert'` for linear-phase filters with odd symmetry (type III and type IV). The output coefficients in `b` obey the relation

  $$b(k) = -b(n+2-k),\ k = 1, \ldots, n + 1$$

  . This class of filters includes the Hilbert transformer, which has a desired amplitude of 1 across the entire band.

- `'differentiator'` for type III and type IV filters, using a special weighting technique. For nonzero amplitude bands, the integrated squared error has a weight of $(1/f)^2$ so that the error at low frequencies is much smaller than at high frequencies. For FIR differentiators, which have an amplitude characteristic proportional to frequency, the filters minimize the relative integrated squared error (the integral of the square of the ratio of the error to the desired amplitude).

Examples

Example 1

Design an order 255 lowpass filter with transition band:

```
b = firls(255,[0 0.25 0.3 1],[1 1 0 0]);
```

Example 2

Design a 31 coefficient differentiator:

```
b = firls(30,[0 0.9],[0 0.9*pi],'differentiator');
```

An ideal differentiator has the response

D(w) = jw

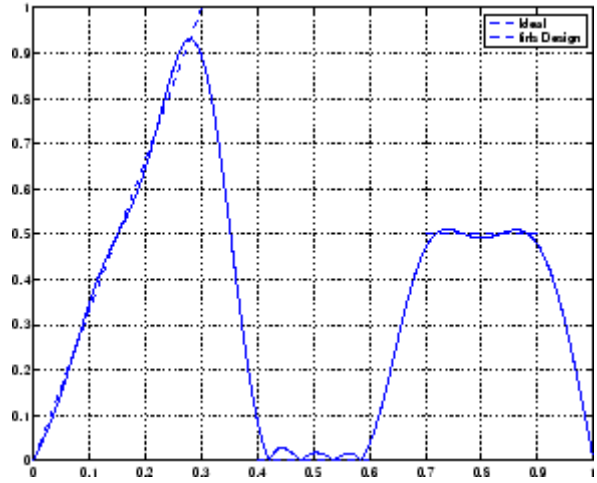
The amplitudes include a pi multiplier because the frequencies are normalized by pi.

Example 3

Design a 24th-order anti-symmetric filter with piecewise linear passbands and plot the desired and actual frequency response:

```
F = [0 0.3  0.4 0.6  0.7 0.9];
A = [0  1   0  0  0.5 0.5];
b = firls(24,F,A,'hilbert');
for i=1:2:6,
   plot([F(i) F(i+1)],[A(i) A(i+1)],'--'), hold on
end
[H,f] = freqz(b,1,512,2);
plot(f,abs(H)), grid on, hold off
```

```
legend('Ideal','firls Design')
```



Algorithm

Reference [1] describes the theoretical approach behind `firls`. The function solves a system of linear equations involving an inner product matrix of size roughly `n/2` using the MATLAB \ operator.

This function designs type I, II, III, and IV linear-phase filters. Type I and II are the defaults for n even and odd respectively, while the `'hilbert'` and `'differentiator'` flags produce type III (n even) and IV (n odd) filters. The various filter types have different symmetries and constraints on their frequency responses (see [2] for details).

Linear Phase Filter Type	Filter Order	Symmetry of Coefficients	Response H(f), f = 0	Response H(f), f = 1 (Nyquist)
Type I	Even	b(k) = b(n+2-k), k=1,..., n+1	No restriction	No restriction

Linear Phase Filter Type	Filter Order	Symmetry of Coefficients	Response H(f), f = 0	Response H(f), f = 1 (Nyquist)
Type II	Even	b(k) = b(n+2-k), k=1,..., n+1	No restriction	$H(1) = 0$
Type III	Odd	b(k) = –b(n+2-k), k=1,..., n+1	$H(0) = 0$	$H(1) = 0$
Type IV	Odd	b(k) = –b(n+2-k), k=1,..., n+1	$H(0) = 0$	No restriction

Diagnostics

One of the following diagnostic messages is displayed when an incorrect argument is used:

```
F must be even length.
F and A must be equal lengths.
Requires symmetry to be 'hilbert' or 'differentiator'.
Requires one weight per band.
Frequencies in F must be nondecreasing.
Frequencies in F must be in range [0,1].
```

A more serious warning message is

```
Warning: Matrix is close to singular or badly scaled.
```

This tends to happen when the product of the filter length and transition width grows large. In this case, the filter coefficients b might not represent the desired filter. You can check the filter by looking at its frequency response.

See Also

`fir1`, `fir2`, `firrcos`, `firpm`

References

[1] Parks, T.W., and C.S. Burrus, *Digital Filter Design*, John Wiley & Sons, 1987, pp. 54-83.

[2] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, pp. 256-266.

Purpose

Parks-McClellan optimal FIR filter design

Syntax

```
b = firpm(n,f,a)
b = firpm(n,f,a,w)
b = firpm(n,f,a, 'ftype')
b = firpm(n,f,a,w, 'ftype')
b = firpm(...,{lgrid})
[b,err] = firpm(...)
[b,err,res] = firpm(...)
b = firpm(n,f,@fresp,w)
b = firpm(n,f,@fresp,w,'ftype')
```

Description

`firpm` designs a linear-phase FIR filter using the Parks-McClellan algorithm [1]. The Parks-McClellan algorithm uses the Remez exchange algorithm and Chebyshev approximation theory to design filters with an optimal fit between the desired and actual frequency responses. The filters are optimal in the sense that the maximum error between the desired frequency response and the actual frequency response is minimized. Filters designed this way exhibit an equiripple behavior in their frequency responses and are sometimes called *equiripple* filters. `firpm` exhibits discontinuities at the head and tail of its impulse response due to this equiripple nature.

`b = firpm(n,f,a)` returns row vector `b` containing the `n+1` coefficients of the order `n` FIR filter whose frequency-amplitude characteristics match those given by vectors `f` and `a`.

The output filter coefficients (taps) in `b` obey the symmetry relation:

$$b(k) = b(n+2-k), \qquad k = 1, \ldots, n+1$$

Vectors `f` and `a` specify the frequency-magnitude characteristics of the filter:

- `f` is a vector of pairs of normalized frequency points, specified in the range between 0 and 1, where 1 corresponds to the Nyquist frequency. The frequencies must be in increasing order.

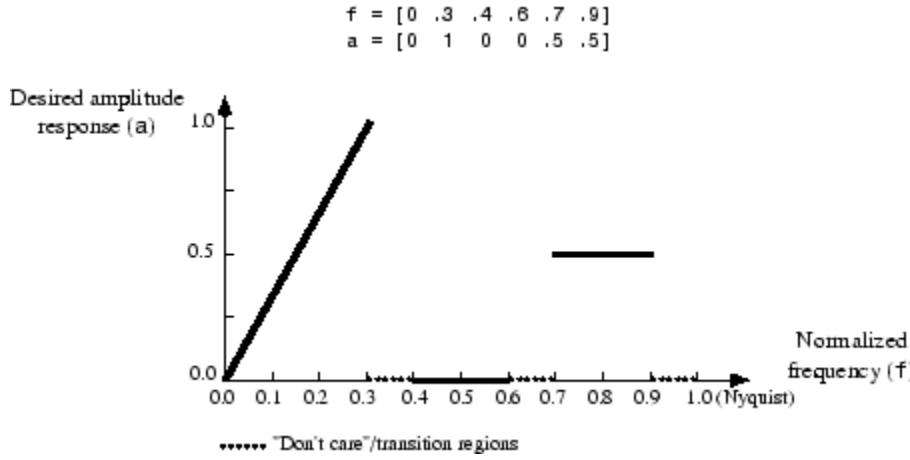
- `a` is a vector containing the desired amplitudes at the points specified in f.

  The desired amplitude at frequencies between pairs of points ($f(k)$, $f(k+1)$) for k odd is the line segment connecting the points ($f(k)$, $a(k)$) and ($f(k+1)$, $a(k+1)$).

  The desired amplitude at frequencies between pairs of points ($f(k)$, $f(k+1)$) for k even is unspecified. The areas between such points are transition or "don't care" regions.

- `f` and `a` must be the same length. The length must be an even number.

The relationship between the `f` and `a` vectors in defining a desired frequency response is shown in the illustration below.



`firpm` always uses an even filter order for configurations with a passband at the Nyquist frequency. This is because for odd orders, the frequency response at the Nyquist frequency is necessarily 0. If you specify an odd-valued `n`, `firpm` increments it by 1.

`b = firpm(n,f,a,w)` uses the weights in vector `w` to weight the fit in each frequency band. The length of `w` is half the length of `f` and `a`, so there is exactly one weight per band.

Note `b = firpm(n,f,a,w)` is a synonym for `b = firpm(n,f,{@firpmfrf,a},w)`, where, `@firpmfrf` is the predefined frequency response function handle for `firpm`. If desired, you can write your own response function. Use `help private/firpmfrf` for information.

`b = firpm(n,f,a, 'ftype')` and

`b = firpm(n,f,a,w, 'ftype')` specify a filter type, where `'ftype'` is

- `'hilbert'`, for linear-phase filters with odd symmetry (type III and type IV)

  The output coefficients in `b` obey the relation $b(k) = -b(n+2-k)$, $k = 1, \ldots, n + 1$. This class of filters includes the Hilbert transformer, which has a desired amplitude of 1 across the entire band.

  For example,

  ```
  h = firpm(30,[0.1 0.9],[1 1],'hilbert');
  ```

  designs an approximate FIR Hilbert transformer of length 31.

- `'differentiator'`, for type III and type IV filters, using a special weighting technique

  For nonzero amplitude bands, it weights the error by a factor of $1/f$ so that the error at low frequencies is much smaller than at high frequencies. For FIR differentiators, which have an amplitude characteristic proportional to frequency, these filters minimize the maximum relative error (the maximum of the ratio of the error to the desired amplitude).

`b = firpm(...,{lgrid})` uses the integer `lgrid` to control the density of the frequency grid, which has roughly `(lgrid*n)/(2*bw)` frequency points, where `bw` is the fraction of the total frequency band interval [0,1] covered by `f`. Increasing `lgrid` often results in filters that more exactly match an equiripple filter, but that take longer to compute. The default value of `16` is the minimum value that should be specified for `lgrid`. Note that the `{lgrid}` argument must be a 1-by-1 cell array.

`[b,err] = firpm(...)` returns the maximum ripple height in err.

`[b,err,res] = firpm(...)` returns a structure `res` with the following fields.

`res.fgrid`	Frequency grid vector used for the filter design optimization
`res.des`	Desired frequency response for each point in `res.fgrid`
`res.wt`	Weighting for each point in `opt.fgrid`
`res.H`	Actual frequency response for each point in `res.fgrid`
`res.error`	Error at each point in `res.fgrid` (`res.des-res.H`)
`res.iextr`	Vector of indices into `res.fgrid` for extremal frequencies
`res.fextr`	Vector of extremal frequencies

You can also use `firpm` to write a function that defines the desired frequency response. The predefined frequency response function handle for `firpm` is `@firpmfrf`, which designs a linear-phase FIR filter.

`b = firpm(n,f,@fresp,w)` returns row vector `b` containing the `n+1` coefficients of the order `n` FIR filter whose frequency-amplitude characteristics best approximate the response returned by function handle `@fresp`. The function is called from within `firpm` with the following syntax.

```
[dh,dw] = fresp(n,f,gf,w)
```

The arguments are similar to those for `firpm`:

- `n` is the filter order.
- `f` is the vector of normalized frequency band edges that appear monotonically between 0 and 1, where 1 is the Nyquist frequency.
- `gf` is a vector of grid points that have been linearly interpolated over each specified frequency band by `firpm`. `gf` determines the frequency grid at which the response function must be evaluated, and contains the same data returned by `cfirpm` in the `fgrid` field of the `opt` structure.
- `w` is a vector of real, positive weights, one per band, used during optimization. `w` is optional in the call to `firpm`; if not specified, it is set to unity weighting before being passed to `fresp`.
- `dh` and `dw` are the desired complex frequency response and band weight vectors, respectively, evaluated at each frequency in grid `gf`.

`b = firpm(n,f,@`*`fresp`*`,w,`*`'ftype'`*`)` designs antisymmetric (odd) filters, where *`'ftype'`* is either `'d'` for a differentiator or `'h'` for a Hilbert transformer. If you do not specify an *`ftype`*, a call is made to *`fresp`* to determine the default symmetry property `sym`. This call is made using the syntax.

```
sym = fresp('defaults',{n,f,[],w,p1,p2,...})
```

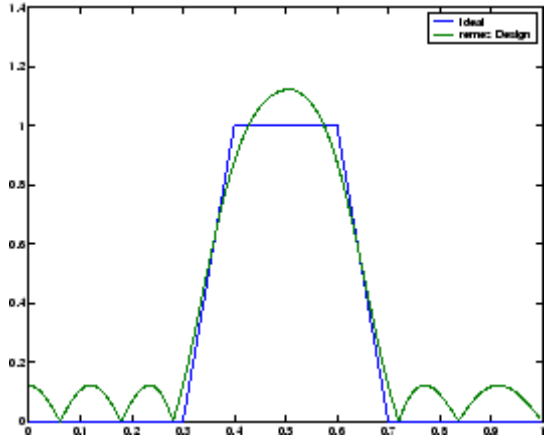
The arguments `n`, `f`, `w`, etc., may be used as necessary in determining an appropriate value for `sym`, which `firpm` expects to be either `'even'` or `'odd'`. If *`fresp`* does not support this calling syntax, `firpm` defaults to even symmetry.

Examples

Graph the desired and actual frequency responses of a 17th-order Parks-McClellan bandpass filter:

```
f = [0 0.3 0.4 0.6 0.7 1]; a = [0 0 1 1 0 0];
b = firpm(17,f,a);
```

```
[h,w] = freqz(b,1,512);
plot(f,a,w/pi,abs(h))
legend('Ideal','firpm Design')
```



Algorithm

`firpm` is a MEX-file version of the original Fortran code from [1], altered to design arbitrarily long filters with arbitrarily many linear bands.

`firpm` designs type I, II, III, and IV linear-phase filters. Type I and type II are the defaults for `n` even and `n` odd, respectively, while type III (`n` even) and type IV (`n` odd) are obtained with the `'hilbert'` and `'differentiator'` flags. The different types of filters have different symmetries and certain constraints on their frequency responses (see [5] for more details).

Linear Phase Filter Type	Filter Order	Symmetry of Coefficients	Response H(f), f = 0	Response H(f), f = 1 (Nyquist)
Type I	Even	even: $b(k) = b(n+2-k),\ k = 1,\ldots,n+1$	No restriction	No restriction

Linear Phase Filter Type	Filter Order	Symmetry of Coefficients	Response H(f), f = 0	Response H(f), f = 1 (Nyquist)
Type II	Odd		No restriction	
Type III	Even	odd: $b(k) = -b(n+2-k), \quad k = 1, \ldots, n+1$	$H(0) = 0$	$H(1) = 0$
Type IV	Odd		$H(0) = 0$	No restriction

Diagnostics

If you get the following warning message,

```
-- Failure to Converge --
Probable cause is machine rounding error.
```

it is possible that the filter design may still be correct. Verify the design by checking its frequency response.

References

[1] *Programs for Digital Signal Processing*, IEEE Press, New York, 1979, Algorithm 5.1.

[2] *Selected Papers in Digital Signal Processing*, II, IEEE Press, New York, 1979.

[3] Parks, T.W., and C.S. Burrus, *Digital Filter Design*, John Wiley & Sons, New York:, 1987, p. 83.

[4] Rabiner, L.R., J.H. McClellan, and T.W. Parks, "FIR Digital Filter Design Techniques Using Weighted Chebyshev Approximations," Proc. IEEE 63 (1975).

[5] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1989, pp. 256-266.

See Also

butter, cheby1, cheby2, cfirpm, ellip, fir1, fir2, fircls, fircls1, firls, firrcos, firpmord, function_handle, yulewalk

Purpose

Parks-McClellan optimal FIR filter order estimation

Syntax

```
[n,fo,ao,w] = firpmord(f,a,dev)
[n,fo,ao,w] = firpmord(f,a,dev,fs)
c = firpmord(f,a,dev,fs,'cell')
```

Description

`[n,fo,ao,w] = firpmord(f,a,dev)` finds the approximate order, normalized frequency band edges, frequency band amplitudes, and weights that meet input specifications `f`, `a`, and `dev`.

- `f` is a vector of frequency band edges (between 0 and $F._s/2$, where f_s is the sampling frequency), and `a` is a vector specifying the desired amplitude on the bands defined by `f`. The length of `f` is two less than twice the length of `a`. The desired function is piecewise constant.
- `dev` is a vector the same size as `a` that specifies the maximum allowable deviation or ripples between the frequency response and the desired amplitude of the output filter for each band.

Use `firpm` with the resulting order `n`, frequency vector `fo`, amplitude response vector `ao`, and weights `w` to design the filter `b` which approximately meets the specifications given by `firpmord` input parameters `f`, `a`, and `dev`.

```
b = firpm(n,fo,ao,w)
```

`[n,fo,ao,w] = firpmord(f,a,dev,fs)` specifies a sampling frequency `fs`. `fs` defaults to 2 Hz, implying a Nyquist frequency of 1 Hz. You can therefore specify band edges scaled to a particular application's sampling frequency.

In some cases `firpmord` underestimates the order `n`. If the filter does not meet the specifications, try a higher order such as `n+1` or `n+2`.

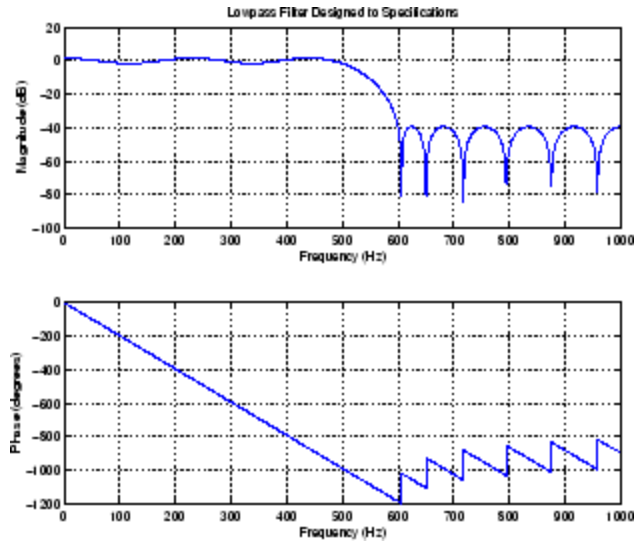
`c = firpmord(f,a,dev,fs,'cell')` generates a cell-array whose elements are the parameters to `firpm`.

Examples

Example 1

Design a minimum-order lowpass filter with a 500 Hz passband cutoff frequency and 600 Hz stopband cutoff frequency, with a sampling frequency of 2000 Hz, at least 40 dB attenuation in the stopband, and less than 3 dB of ripple in the passband:

```
rp = 3;           % Passband ripple
rs = 40;          % Stopband ripple
fs = 2000;        % Sampling frequency
f = [500 600];    % Cutoff frequencies
a = [1 0];        % Desired amplitudes
% Compute deviations
dev = [(10^(rp/20)-1)/(10^(rp/20)+1)  10^(-rs/20)];
[n,fo,ao,w] = firpmord(f,a,dev,fs);
b = firpm(n,fo,ao,w);
freqz(b,1,1024,fs);
title('Lowpass Filter Designed to Specifications');
```



Note that the filter falls slightly short of meeting the stopband attenuation and passband ripple specifications. Using `n+1` in the call to `firpm` instead of `n` achieves the desired amplitude characteristics.

Example 2

Design a lowpass filter with a 1500 Hz passband cutoff frequency and 2000 Hz stopband cutoff frequency, with a sampling frequency of 8000 Hz, a maximum stopband amplitude of 0.1, and a maximum passband error (ripple) of 0.01:

```
[n,fo,ao,w] = firpmord([1500 2000],[1 0],[0.01 0.1],8000 );
b = firpm(n,fo,ao,w);
```

This is equivalent to

```
c = firpmord( [1500 2000],[1 0],[0.01 0.1],8000,'cell');
b = firpm(c{:});
```

Note In some cases, `firpmord` underestimates or overestimates the order `n`. If the filter does not meet the specifications, try a higher order such as `n+1` or `n+2`.

Results are inaccurate if the cutoff frequencies are near 0 or the Nyquist frequency.

Algorithm

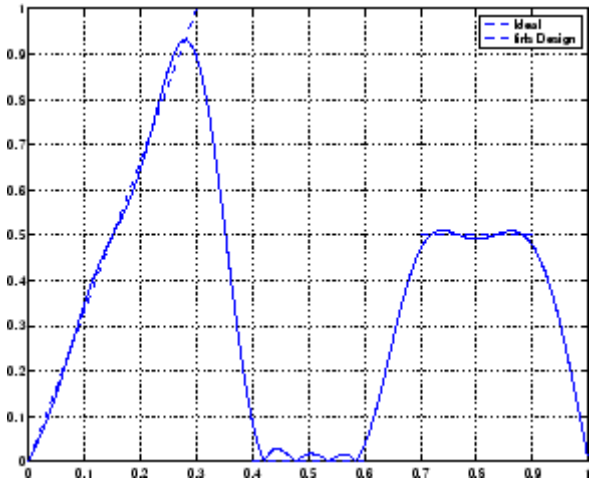
`firpmord` uses the algorithm suggested in [1]. This method is inaccurate for band edges close to either 0 or the Nyquist frequency (`fs/2`).

References

[1] Rabiner, L.R., and O. Herrmann, "The Predictability of Certain Optimum Finite Impulse Response Digital Filters," *IEEE Trans. on Circuit Theory, Vol. CT-20*, No. 4 (July 1973), pp. 401-408.

[2] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing. Englewood Cliffs*, NJ: Prentice-Hall, 1975, pp. 156-157.

See Also



`buttord`, `cheb1ord`, `cheb2ord`, `ellipord`, `kaiserord`, `firpm`

Purpose

Raised cosine FIR filter design

Syntax

```
b = firrcos(n,F0,df,fs)
b = firrcos(n,F0,df,fs,'bandwidth')
b = firrcos(n,F0,df)
b = firrcos(n,F0,r,fs,'rolloff')
b = firrcos(...,'type')
b = firrcos(...,'type',delay)
b = firrcos(...,'type',delay,window)
[b,a] = firrcos(...)
```

Description

`b = firrcos(n,F0,df,fs)` or, equivalently,

`b = firrcos(n,F0,df,fs,'bandwidth')` returns an order `n` lowpass linear-phase FIR filter with a raised cosine transition band. The order `n` must be even. The filter has cutoff frequency `F0`, transition bandwidth `df`, and sampling frequency `fs`, all in hertz. `df` must be small enough so that `F0` ± `df/2` is between `0` and `fs/2`. The coefficients in `b` are normalized so that the nominal passband gain is always equal to 1. Specify `fs` as the empty vector `[]` to use the default value `fs = 2`.

`b = firrcos(n,F0,df)` uses a default sampling frequency of `fs = 2`.

`b = firrcos(n,F0,r,fs,'rolloff')` interprets the third argument, `r`, as the rolloff factor instead of the transition bandwidth, `df`. `r` must be in the range `[0,1]`.

`b = firrcos(...,'type')` designs either a normal raised cosine filter or a square root raised cosine filter according to how you specify of the string *`'type'`*. Specify *`'type'`* as:

- `'normal'`, for a regular raised cosine filter. This is the default, and is also in effect when the *`'type'`* argument is left empty, `[]`.
- `'sqrt'`, for a square root raised cosine filter.

`b = firrcos(...,'type',delay)` specifies an integer delay in the range `[0,n+1]`. The default is `n/2` for all `n`.

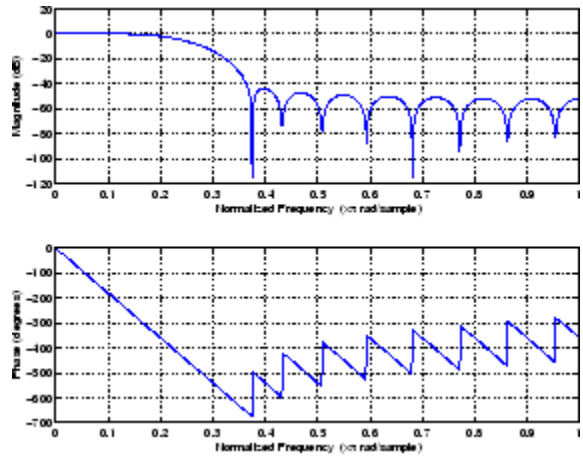
`b = firrcos(...,'type',delay,window)` applies a length `n+1` window to the designed filter to reduce the ripple in the frequency response. `window` must be a length `n+1` column vector. If no window is specified, a rectangular (`rectwin`) window is used. Care must be exercised when using a window with a delay other than the default.

`[b,a] = firrcos(...)` always returns `a = 1`.

Examples

Design an order 20 raised cosine FIR filter with cutoff frequency 0.25 of the Nyquist frequency and a transition bandwidth of 0.25:

```
h = firrcos(20,0.25,0.25);
freqz(h,1)
```



See Also

`fir1`, `fir2`, `firls`, `firpm`

Purpose

Flat Top weighted window

Syntax

```
w = flattopwin(L)
w = flattopwin(L,sflag)
```

Description

Flat Top windows have very low passband ripple (< 0.01 dB) and are used primarily for calibration purposes. Their bandwidth is approximately 2.5 times wider than a Hann window.

`w = flattopwin(L)` returns the `L`-point symmetric flat top window in column vector `w`.

`w = flattopwin(L,sflag)` returns the `L`-point symmetric flat top window using `sflag` window sampling, where `sflag` is either `'symmetric'` or `'periodic'`. The `'periodic'` flag is useful for DFT/FFT purposes, such as in spectral analysis. The DFT/FFT contains an implicit periodic extension and the periodic flag enables a signal windowed with a periodic window to have perfect periodic extension. When `'periodic'` is specified, `flattopwin` computes a length `L+1` window and returns the first `L` points. When using windows for filter design, the `'symmetric'` flag should be used.

Algorithm

Flat top windows are summations of cosines. The coefficients of a flat top window are computed from the following equation

$$w(n) = a_0 - a_1\cos\left(\frac{2\pi n}{N}\right) + a_2\cos\left(\frac{4\pi n}{N}\right) - a_3\cos\left(\frac{6\pi n}{N}\right) + a_4\cos\left(\frac{8\pi n}{N}\right)$$

where $0 \le n \le N$ and $w(n) = 0$ elsewhere and the window length is L = N +1. The coefficient values are

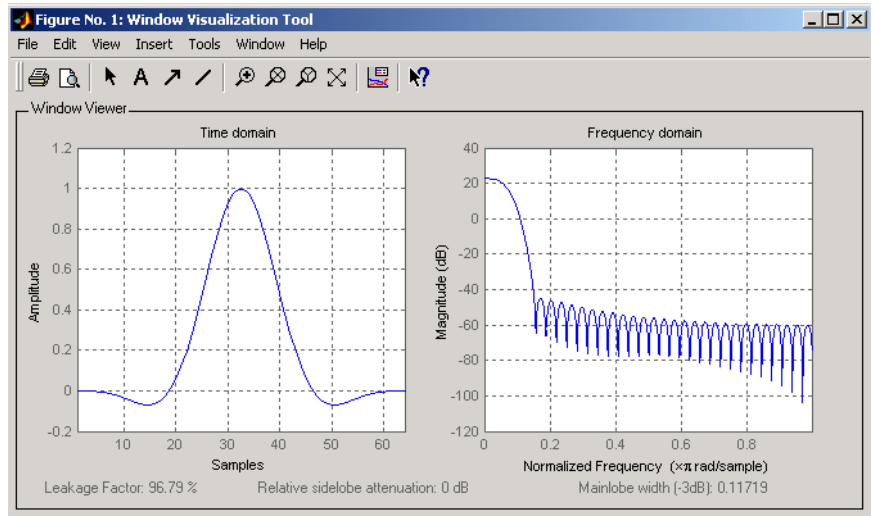
Coefficient	Value
a_0	0.21557895
a_1	0.41663158

Coefficient	Value
a_2	0.277263158
a_3	0.083578947
a_4	0.006947368

Examples

Create a 64-point, symmetric Flat Top window and view the window using WVTool:

```
w = flattopwin(64);
wvtool(w);
```



Reference

[1] D'Antona, Gabriele. and A. Ferrero, *Digital Signal Processing for Measurement Systems*, New York: Springer Media, Inc., 2006, pp. 70–72.

[2] Gade, Svend and H. Herlufsen, "Use of Weighting Functions in DFT/FFT Analysis (Part I)," Brüel & Kjær, *Windows to FFT Analysis (Part I) Technical Review, No. 3*, 1987, pp. 19-21.

See Also

`blackman`, `hamming`, `hann`

freqs

Purpose

Frequency response of analog filters

Syntax

```
h = freqs(b,a,w)
[h,w] = freqs(b,a,n)
freqs
```

Description

`freqs` returns the complex frequency response $H(j\omega)$ (Laplace transform) of an analog filter

$$H(s) = \frac{B(s)}{A(s)} = \frac{b(1)s^n + b(2)s^{n-1} + \cdots + b(n+1)}{a(1)s^m + a(2)s^{m-1} + \cdots + a(m+1)}$$

given the numerator and denominator coefficients in vectors `b` and `a`.

`h = freqs(b,a,w)` returns the complex frequency response of the analog filter specified by coefficient vectors `b` and `a`. `freqs` evaluates the frequency response along the imaginary axis in the complex plane at the angular frequencies in rad/sec specified in real vector `w`, where `w` is a vector containing more than one frequency.

`[h,w] = freqs(b,a,n)` uses `n` frequency points to compute the frequency response `h`, where `n` is a real, scalar value. The frequency vector `w` is auto-generated and has length `n`. If you omit `n` as an input, 200 frequency points are used. If you do not need the generated frequency vector returned, you can use the form `h = freqs(b,a,n)` to return only the frequency response `h`.

`freqs` with no output arguments plots the magnitude and phase response versus frequency in the current figure window.

`freqs` works only for real input systems and positive frequencies.

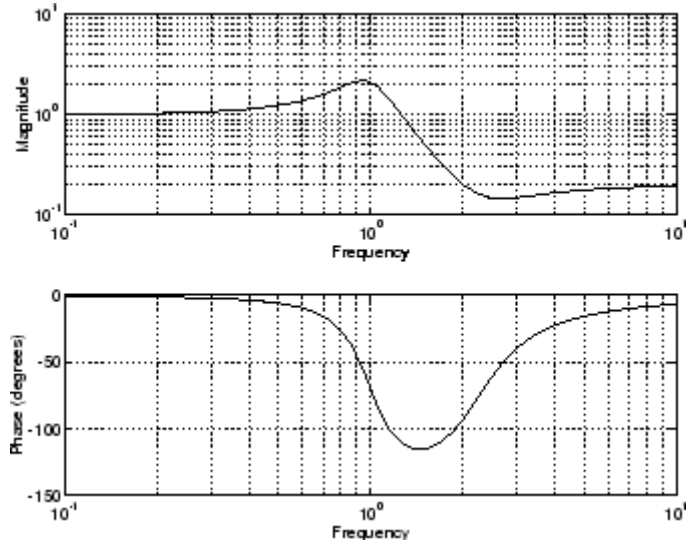
Examples

Find and graph the frequency response of the transfer function given by:

$$H(s) = \frac{0.2s^2 + 0.3s + 1}{s^2 + 0.4s + 1}$$

```
a = [1 0.4 1];
b = [0.2 0.3 1];
```

```
w = logspace(-1,1);
freqs(b,a,w)
```



You can also create the plot with

```
h = freqs(b,a,w);
mag = abs(h);
phase = angle(h);
subplot(2,1,1), loglog(w,mag)
subplot(2,1,2), semilogx(w,phase)
```

To convert to hertz, degrees, and decibels, use

```
f = w/(2*pi);
mag = 20*log10(mag);
phase = phase*180/pi;
```

Algorithm

`freqs` evaluates the polynomials at each frequency point, then divides the numerator response by the denominator response:

```
s = i*w;
h = polyval(b,s)./polyval(a,s);
```

See Also

abs, angle, freqz, invfreqs, logspace, polyval

Purpose

Real or complex frequency-sampled FIR filter from specification object

Syntax

```
hd = design(d,'freqsamp')
hd = design(...,'filterstructure',structure)
hd = design(...,'window',window)
```

Description

`hd = design(d,'freqsamp')` designs a frequency-sampled filter specified by the `fspecifications object` `h`.

`hd = design(...,'filterstructure',structure)` returns a filter with the filter structure you specify by the `structure` input argument. `structure` is `dffir` by default and can be any one of the following filter structures.

Structure String	Description of Resulting Filter Structure
`dffir`	Direct-form FIR filter
`dffirt`	Transposed direct-form FIR filter
`dfsymfir`	Symmetrical direct-form FIR filter
`dfasymfir`	Asymmetrical direct-form FIR filter
`fftfir`	Fast Fourier transform FIR filter

`hd = design(...,'window',window)` designs filters using the window specified by the string in `window`. Provide the input argument `window` as

- A string for the window type. For example, use `bartlett` or chebwin, or hamming. Click `window` for the full list of windows available in the *Signal Processing Toolbox User's Guide*.
- A function handle that references the `window` function. When the `window` function requires more than one input, use a cell array to hold the required arguments. The final example shows a cell array input argument.
- The window vector itself.

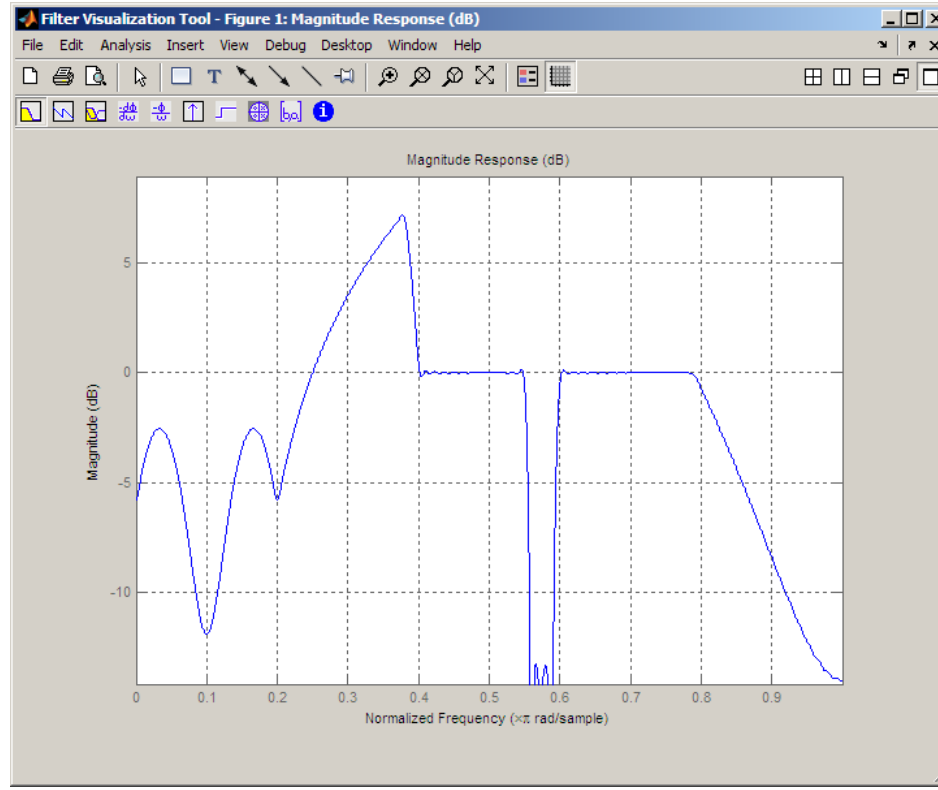
Examples

These examples design FIR filters that have arbitrary magnitude responses. In the first filter, the response has three distinct sections and the resulting filter is real.

The second example creates a complex filter.

```
b1 = 0:0.01:0.18;
b2 = [.2 .38 .4 .55 .562 .585 .6 .78];
b3 = [0.79:0.01:1];
a1 = .5+sin(2*pi*7.5*b1)/4;    % Sinusoidal response section.
a2 = [.5 2.3 1 1 -.2 -.2 1 1]; % Piecewise linear response section.
a3 = .2+18*(1-b3).^2;          % Quadratic response section.
f = [b1 b2 b3];
a = [a1 a2 a3];
n = 300;
d = fdesign.arbmag('n,f,a',n,f,a); % First specifications object.
hd = design(d,'freqsamp','window',{@kaiser,.5}); % Filter.
fvtool(hd)
```

The plot from FVTool shows the response for `hd`.

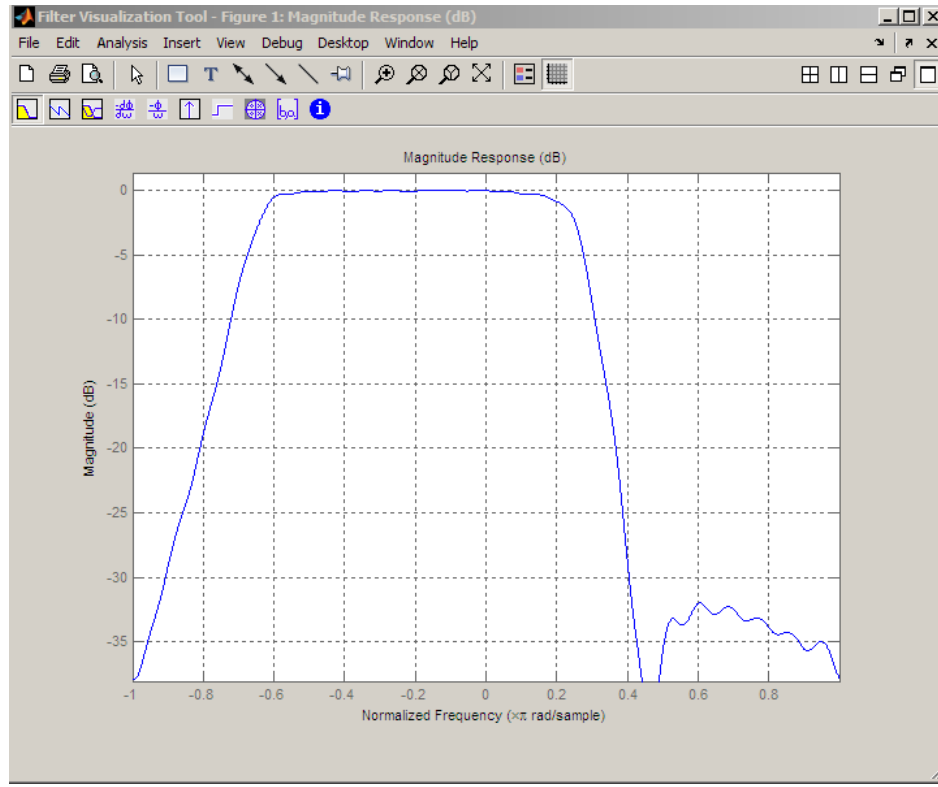


Now design the arbitrary-magnitude complex FIR filter. Recall that vector `f` contains frequency locations and vector `a` contains the desired filter response values at the locations specified in `f`.

```
f = [-1 -.93443 -.86885 -.80328 -.7377 -.67213 -.60656 -.54098 ...
-.47541,-.40984 -.34426 -.27869 -.21311 -.14754 -.081967 ...
-.016393 .04918 .11475,.18033 .2459 .31148 .37705 .44262 ...
 .5082 .57377 .63934 .70492 .77049,.83607 .90164 1];
a = [.0095848 .021972 .047249 .099869 .23119 .57569 .94032 ...
.98084 .99707,.99565 .9958 .99899 .99402 .99978 .99995 .99733 ...
.99731 .96979 .94936,.8196 .28502 .065469 .0044517 .018164 ...
.023305 .02397 .023141 .021341,.019364 .017379 .016061];
```

```
n = 48;
d = fdesign.arbmag('n,f,a',n,f,a); % Second spec. object.
hdc = design(d,'freqsamp','window','rectwin'); % Filter.
fvtool(hdc)
```

FVTool shows you the response for `hdc` from -1 to 1 in normalized frequency because the filter's transfer function is not symmetric around 0. Since the Fourier transform of the filter does not exhibit conjugate symmetry, `design(d,...)` returns a complex—valued filter for `hdc` .



See Also

`design`, `designmethods`, `fdesign.arbmag`.

Purpose

Frequency response of digital filter

Syntax

```
[h,w] = freqz(b,a,l)
h = freqz(b,a,w)
[h,w] = freqz(b,a,l,'whole')
[h,f] = freqz(b,a,l,fs)
h = freqz(b,a,f,fs)
[h,f] = freqz(b,a,l,'whole',fs)
freqz(b,a,...)
freqz(Hd)
```

Description

`[h,w] = freqz(b,a,l)` returns the frequency response vector `h` and the corresponding angular frequency vector `w` for the digital filter whose transfer function is determined by the (real or complex) numerator and denominator polynomials represented in the vectors `b` and `a`, respectively. The vectors `h` and `w` are both of length `l`. The angular frequency vector w has values ranging from 0 to π radians per sample. When you don't specify the integer `l`, or you specify it as the empty vector `[]`, the frequency response is calculated using the default value of 512 samples.

`h = freqz(b,a,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(b,a,l,'whole')` uses `n` sample points around the entire unit circle to calculate the frequency response. The frequency vector w has length `l` and has values ranging from 0 to 2π radians per sample.

`[h,f] = freqz(b,a,l,fs)` returns the frequency response vector `h` and the corresponding frequency vector `f` for the digital filter whose transfer function is determined by the (real or complex) numerator and denominator polynomials represented in the vectors `b` and `a`, respectively. The vectors `h` and f are both of length `l`. For this syntax, the frequency response is calculated using the sampling frequency specified by the scalar `fs` (in hertz). The frequency vector f is calculated in units of hertz (Hz). The frequency vector f has values ranging from `0` to `fs/2`Hz.

`h = freqz(b,a,f,fs)` returns the frequency response vector `h` calculated at the frequencies (in Hz) supplied in the vector `f`. The vector `f` can be any length.

`[h,f] = freqz(b,a,l,'whole',fs)` uses `n` points around the entire unit circle to calculate the frequency response. The frequency vector f has length `l` and has values ranging from `0` to `fs`Hz.

`freqz(b,a,...)` plots the magnitude and unwrapped phase of the frequency response of the filter. The plot is displayed in the current figure window.

`freqz(Hd)` plots the magnitude and unwrapped phase of the frequency response of the filter. The plot is displayed in `fvtool`. The input `Hd` is a `dfilt` filter object or an array of `dfilt` filter objects.

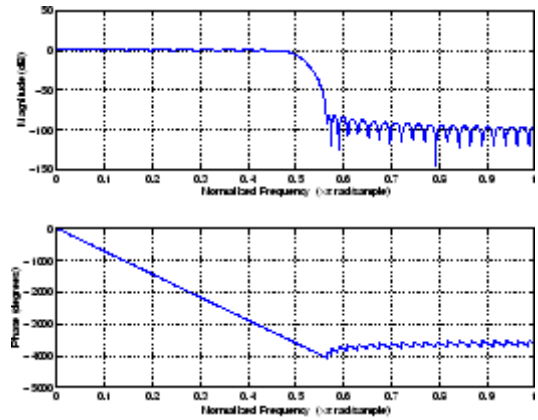
Remarks

It is best to choose a power of 2 for the third input argument `n`, because `freqz` uses an FFT algorithm to calculate the frequency response. See the reference description of `fft` for more information.

Examples

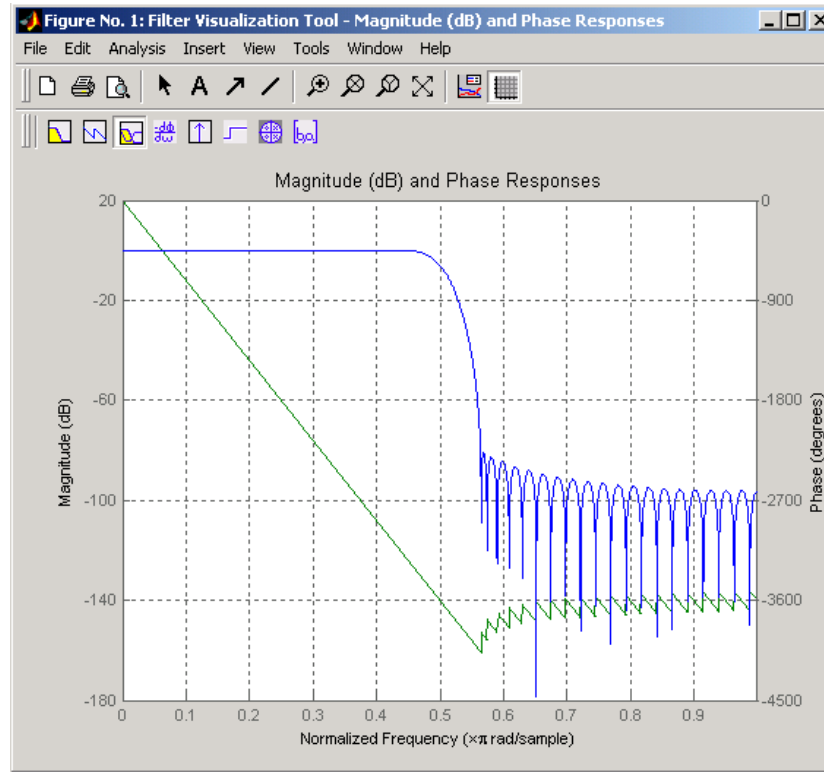
Plot the magnitude and phase response of an FIR filter:

```
b = fir1(80,0.5,kaiser(81,8));
freqz(b,1);
```



The same example using a `dfilt` object and displaying the result in the Filter Visualization Tool (`fvtool`) is

```
Hd = dfilt.dffir(b);
freqz(Hd);
```



Algorithm

The frequency response [1] of a digital filter can be interpreted as the transfer function evaluated at $z = e^{j\omega}$. You can always write a rational transfer function in the following form.

$$H(z) = \frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \cdots + b(n+1)z^{-n}}{a(1) + a(2)z^{-1} + \cdots + a(m+1)z^{-m}}$$

`freqz` determines the transfer function from the (real or complex) numerator and denominator polynomials you specify, and returns the complex frequency response $H(e^{j\omega})$ of a digital filter. The frequency response is evaluated at sample points determined by the syntax that you use.

`freqz` generally uses an FFT algorithm to compute the frequency response whenever you don't supply a vector of frequencies as an input argument. It computes the frequency response as the ratio of the transformed numerator and denominator coefficients, padded with zeros to the desired length.

When you do supply a vector of frequencies as an input argument, then `freqz` evaluates the polynomials at each frequency point using Horner's method of nested polynomial evaluation [1], dividing the numerator response by the denominator response.

References

[1] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing, Prentice-Hall*, 1989, pp. 203-205.

See Also

`abs`, `angle`, `fft`, `filter`, `freqs`, `impz`, `invfreqs`, `logspace`

Purpose

Open Filter Visualization Tool

Syntax

```
fvtool(b,a)
fvtool(b1,a1,b2,a2,...bn,an)
fvtool(Hd1,Hd2,...)
h = fvtool(...)
```

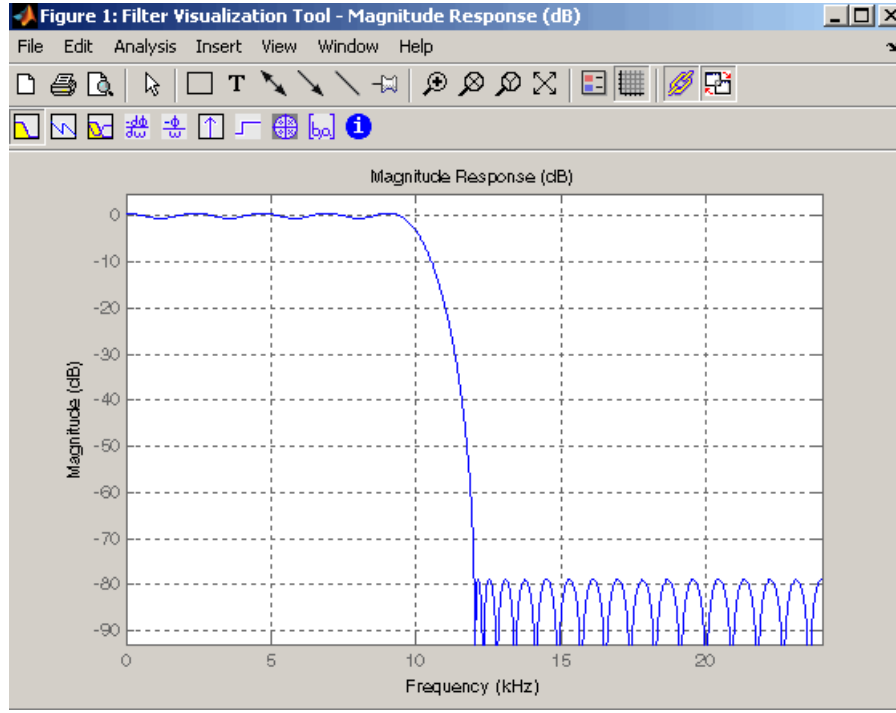
Description

`fvtool(b,a)` opens FVTool and computes the magnitude response of the digital filter defined with numerator, `b` and denominator, `a`. Using FVTool you can display the phase response, group delay, impulse response, step response, pole-zero plot, and coefficients of the filter. You can export the displayed response to a file with **File > Export**.

`fvtool(b1,a1,b2,a2,...bn,an)` opens FVTool and computes the magnitude responses of multiple filters defined with numerators, $b_1 \ldots b_n$ and denominators, $a_1 \ldots a_n$.

`fvtool(Hd1,Hd2,...)` opens FVTool and computes the magnitude responses of the filters in the `dfilt` objects `Hd1`, `Hd2`, etc. If you have the Filter Design Toolbox product installed, you can also use `fvtool(H1,H2,...)` to analyze quantized filter objects (`dfilt` with arithmetic set to `'single'` or `'fixed'`), multirate filter (`mfilt`) objects, and adaptive filter (`adaptfilt`) objects.

`h = fvtool(...)` returns a figure handle `h`. You can use this handle to interact with FVTool from the command line. See "Controlling FVTool from the MATLAB Command Line" on page 10-524 below.



FVTool has two toolbars.

- An extended version of the MATLAB plot editing toolbar. The following table shows the toolbar icons specific to FVTool.

Icon	Description
	Restore default view. This view displays buffer regions around the data and shows only significant data. To see the response using standard MATLAB plotting, which shows all data values, use **View > Full View**.
	Toggle legend

Icon	Description
	Toggle grid
	Link to FDATool (appears only if FVTool was started from FDATool)
	Toggle Add mode/Replace mode (appears only if FVTool was launched from FDATool)

- Analysis toolbar with the following icons

	Magnitude response of the current filter. See `freqz` and `zerophase` for more information. To see the zero-phase response, right-click the *y*-axis label of the Magnitude plot and select **Zero-phase** from the context menu.
	Phase response of the current filter. See `phasez` for more information.
	Superimposes the magnitude response and the phase response of the current filter. See `freqz` for more information.
	Shows the group delay of the current filter. Group delay is the average delay of the filter as a function of frequency. See `grpdelay` for more information.
	Shows the phase delay of the current filter. Phase delay is the time delay the filter imposes on each component of the input signal. See `phasedelay` for more information.
	Impulse response of the current filter. The impulse response is the response of the filter to a impulse input. See `impz` for more information.

	Step response of the current filter. The step response is the response of the filter to a step input. See `stepz` for more information.
	Pole-zero plot, which shows the pole and zero locations of the current filter on the *z*-plane. See `zplane` for more information.
	Filter coefficients of the current filter, which depend on the filter structure (e.g., direct-form, lattice, etc.) in a text box. For SOS filters, each section is displayed as a separate filter.
	Detailed filter information.

Linking to FDATool

In `fdatool`, selecting **View > Filter Visualization Tool** or the **Full View Analysis** toolbar button  when an analysis is displayed starts FVTool for the current filter. You can synchronize FDATool and FVTool with the **FDAToolLink** toolbar button  . Any changes made to the filter in FDATool are immediately reflected in FVTool.

Two FDATool link modes are provided via the **Set Link Mode** toolbar button:

- Replace  — removes the filter currently displayed in FVTool and inserts the new filter.
- Add  — retains the filter currently displayed in FVTool and adds the new filter to the display.

Modifying the Axes

You can change the *x*- or *y*-axis units by right-clicking the mouse on the axis label or by right-clicking on the plot and selecting **Analysis Parameters**. Available options for the axes units are as follows.

Plot	X-Axis Units	Y-Axis Units
Magnitude	Normalized Frequency Linear Frequency	Magnitude Magnitude(dB) Magnitude squared Zero-Phase
Phase	Normalized Frequency Linear Frequency	Phase Continuous Phase Degrees Radians
Magnitude and Phase	Normalized Frequency Linear Frequency	(*y*-axis on left side) Magnitude Magnitude(dB) Magnitude squared Zero-Phase (*y*-axis on right side) Phase Continuous Phase Degrees Radians
Group Delay	Normalized Frequency Linear Frequency	Samples Time
Phase Delay	Normalized Frequency Linear Frequency	Degrees Radians
Impulse Response	Samples Time	Amplitude

Plot	X-Axis Units	Y-Axis Units
Step Response	Samples Time	Amplitude
Pole-Zero	Real Part	Imaginary Part

Modifying the Plot

You can use any of the plot editing toolbar buttons to change the properties of your plot.

Analysis Parameters are parameters that apply to the displayed analyses. To display them, right-click in the plot area and select **Analysis Parameters** from the menu. (Note that you can access the menu only if the **Edit Plot** button is inactive.) The following analysis parameters are displayed. (If more than one response is displayed, parameters applicable to each plot are displayed.) Not all of these analysis fields are displayed for all types of plots:

- **Normalized Frequency** — if checked, frequency is normalized between 0 and 1, or if not checked, frequency is in Hz
- **Frequency Scale** — *y*-axis scale (`Linear` or `Log`)
- **Frequency Range** — range of the frequency axis or `Specify freq. vector`
- **Number of Points** — number of samples used to compute the response
- **Frequency Vector** — vector to use for plotting, if `Specify freq. vector` is selected in **Frequency Range**.
- **Magnitude Display** — *y*-axis units (`Magnitude`, `Magnitude (dB)`, `Magnitude squared`, or `Zero-Phase`)
- **Phase Units** — *y*-axis units (`Degrees` or `Radians`)
- **Phase Display** — type of phase plot (`Phase` or `Continuous Phase`)
- **Group Delay Units** — *y*-axis units (`Samples` or `Time`)

- **Specify Length** — length type of impulse or step response (`Default` or `Specified`)
- **Length** — number of points to use for the impulse or step response

In addition to the above analysis parameters, you can change the plot type for Impulse and Step Response plots by right-clicking and selecting **Line with Marker**, **Stem** or **Line** from the context menu. You can change the *x*-axis units by right-clicking the *x*-axis label and selecting `Samples` or `Time`.

To save the displayed parameters as the default values to use when FDATool or FVTool is opened, click **Save as default**.

To restore the default values, click **Restore original defaults**.

Data tips display information about a particular point in the plot. See "Data Cursor — Displaying Data Values Interactively" in the MATLAB documentation for information on data tips.

When FVTool is started from FDATool, you can use Specification Masks to display filter specifications on a Magnitude plot. You can also draw your own specification masks. See "Analyzing the Filter" on page 5-16.

Note To use **View > Passband zoom**, your filter must have been designed using `fdesign` or FDATool. Passband zoom is not provided for cascaded integrator-comb (CIC) filters because CICs do not have conventional passbands.

Overlaying a Response

You can overlay a second response on the plot by selecting **Analysis > Overlay Analysis** and selecting an available response. A second *y*-axis is added to the right side of the response plot. The Analysis Parameters dialog box shows parameters for the *x*-axis and both *y*-axes. See "Example 2" on page 10-528 for a sample Analysis Parameters dialog box.

Controlling FVTool from the MATLAB Command Line

After you obtain the handle for FVTool, you can control some aspects of FVTool from the command line. In addition to the standard Handle Graphics® properties (see Handle Graphics in the MATLAB documentation), FVTool has the following properties:

- `'Filters'` — returns a cell array of the filters in FVTool.
- `'Analysis'` — displays the specified type of analysis plot. The following table lists the analyses and corresponding analysis strings. Note that the only analyses that use filter internals are magnitude response estimate and round-off noise power, which are available only with the Filter Design Toolbox product.

Analysis Type	Analysis String
Magnitude plot	`'magnitude'`
Phase plot	`'phase'`
Magnitude and phase plot	`` `freq' ``
Group delay plot	`'grpdelay'`
Phase delay plot	`` `phasedelay' ``
Impulse response plot	`'impulse'`
Step response plot	`'step'`
Pole-zero plot	`'polezero'`
Filter coefficients	`'coefficients'`
Filter information	`'info'`

Analysis Type	Analysis String
Magnitude response estimate (available only with the Filter Design Toolbox product, see `freqrespest` for more information)	`'magestimate'`
Round-off noise power (available only with the Filter Design Toolbox product, see `noisepsd` for more information)	`'noisepower'`

- `'Grid'` — controls whether the grid is `'on'` or `'off'`
- `'Legend'` — controls whether the legend is `'on'` or `'off'`
- `'Fs'` — controls the sampling frequency of filters in FVTool. The sampling frequency vector must be of the same length as the number of filters or a scalar value. If it is a vector, each value is applied to its corresponding filter. If it is a scalar, the same value is applied to all filters.
- `SosViewSettings` — (This option is available only if you have the Filter Design Toolbox product.) For second-order sections filters, this controls how the filter is displayed. The `SOSViewSettings` property contains an object so you must use this syntax to set it: `set(h.SOSViewSettings,'View',`*`viewtype`*`)`, where *`viewtype`* is one of the following:
  - `'Complete'` — Displays the complete response of the overall filter
  - `'Individual'` — Displays the response of each section separately
  - 'Cumulative' — Displays the response for each section accumulated with each prior section. If your filter has three sections, the first plot shows section one, the second plot shows the accumulation of sections one and two, and the third plot show the accumulation of all three sections.

You can also define whether to use `SecondaryScaling`, which determines where the sections should be split. The secondary scaling points are the scaling locations between the recursive and the nonrecursive parts of the section. The default value is `false`, which does not use secondary scaling. To turn on secondary scaling, use this syntax: `set(h.SOSViewSettings,'View','Cumulative',true)`

- `'UserDefined'` — Allows you to define which sections to display and the order in which to display them. Enter a cell array where each section is represented by its index. If you enter one index, only that section is plotted. If you enter a range of indices, the combined response of that range of sections is plotted. For example, if your filter has four sections, entering `{1:4}` plots the combined response for all four sections, and entering `{1,2,3,4}` plots the response for each section individually.

Note You can change other properties of FVTool from the command line using the `set` function. Use `get(h)` to view property tags and current property settings.

You can use the following methods with the FVTool handle.

`addfilter(h,filtobj)` adds a new filter to FVTool. The new filter, `filtobj`, must be a `dfilt` filter object. You can specify the sampling frequency of the new filter with `addfilter(h,filtobj,'Fs',10)`.

`setfilter(h,filtobj)` replaces the filter in FVTool with the filter specified in `filtobj`. You can set the sampling frequency as described above.

`deletefilter(h, index)` deletes the filter at the FVTool cell array `index` location.

`legend(h,str1,str2,...)` creates a legend in FVTool by associating `str1` with filter 1, `str2` with filter 2, etc. See `legend` in the MATLAB documentation for information.

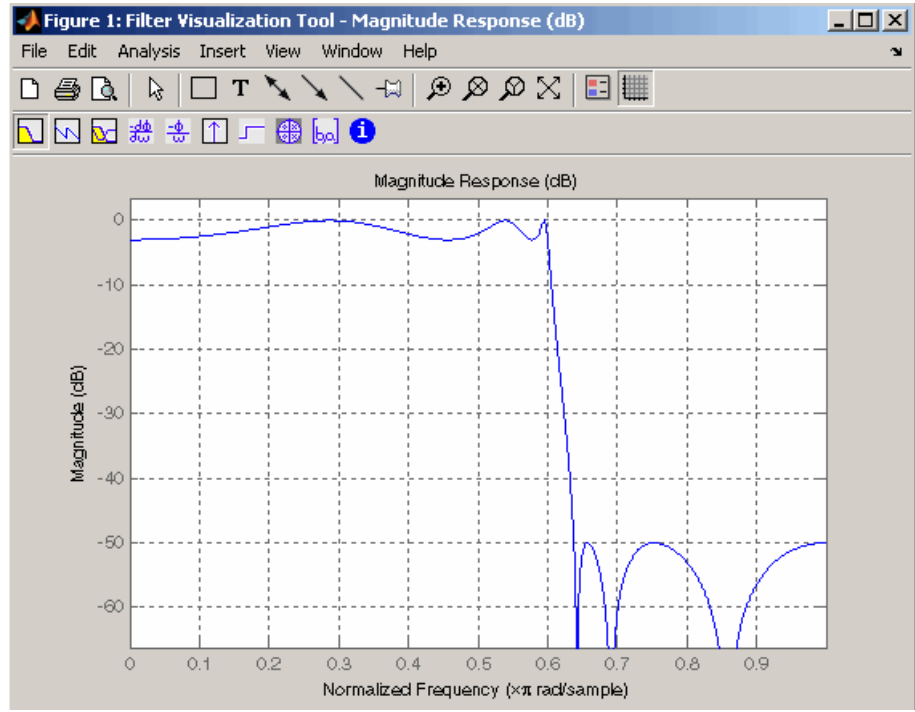
For more information on using FVTool from the command line, see the demo `fvtooldemo`.

Examples

Example 1

Display the magnitude response of an elliptic filter, starting FVTool from the command line:

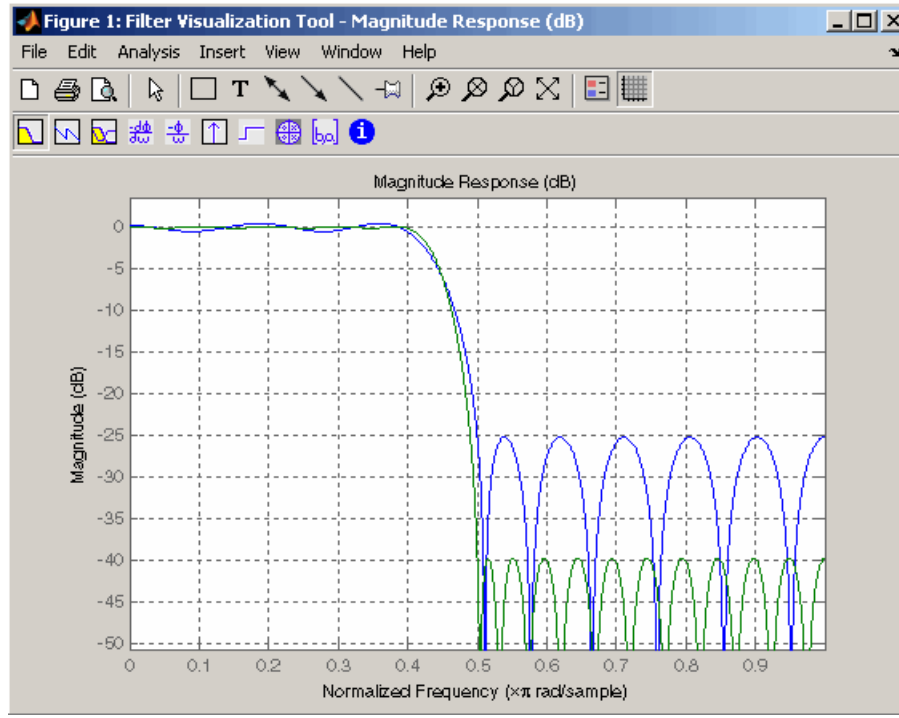
```
[b,a]=ellip(6,3,50,300/500);
fvtool(b,a);
```

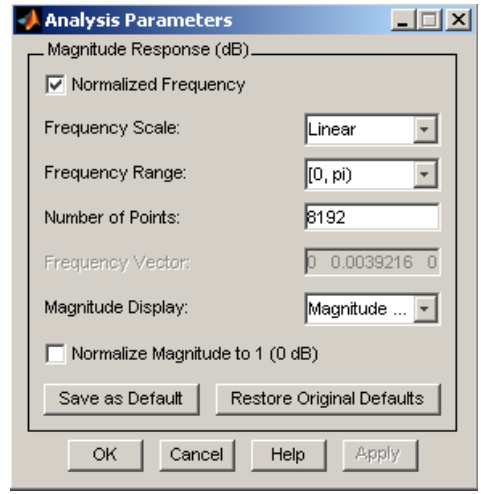


Example 2

Display and analyze multiple FIR filters, starting FVTool from the command line. Then, display the associated analysis parameters for the magnitude:

```
b1 = firpm(20,[0 0.4 0.5 1],[1 1 0 0]);
b2 = firpm(40,[0 0.4 0.5 1],[1 1 0 0]);
fvtool(b1,1,b2,1);
```





Example 3

Create a lowpass, equiripple filter of order 20 in FDATool and display it in FVTool.

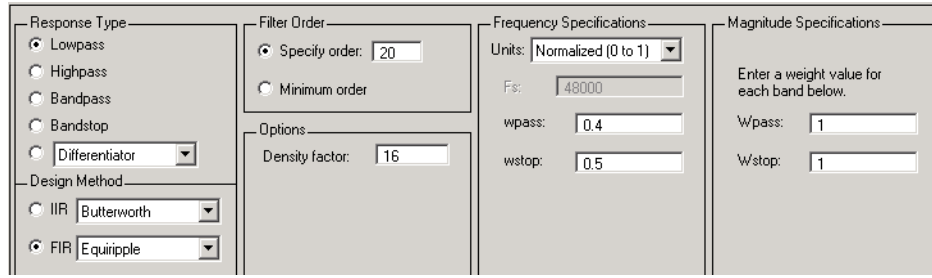
```
fdatool          % Start FDATool
```

Set these parameters in `fdatool`:

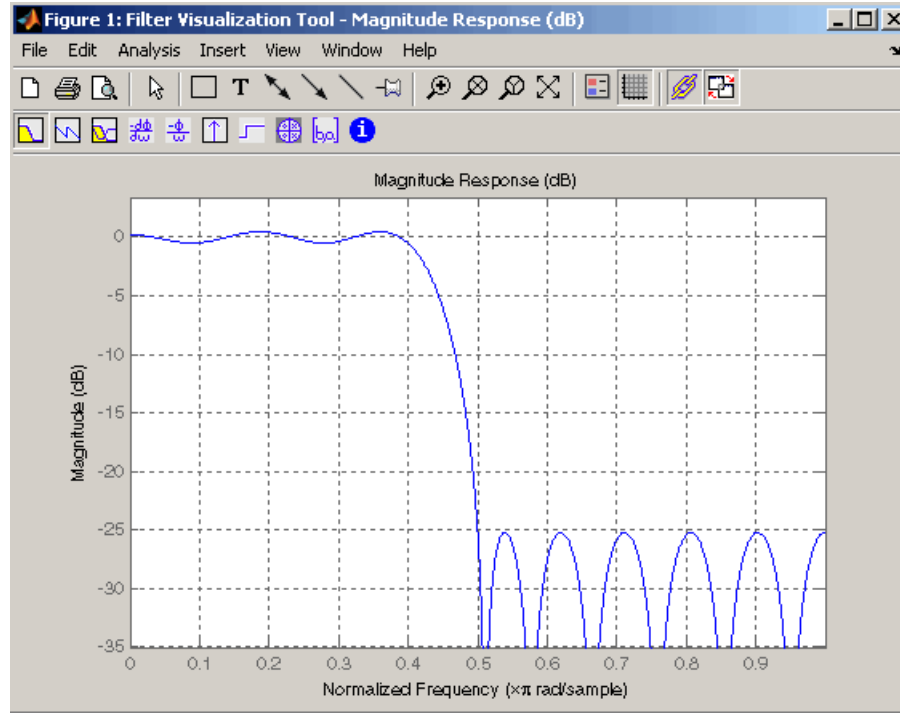
Parameter	Setting
Response Type	Lowpass
Design Method	FIR Equiripple
Filter Order	Specify order: 20
Density factor	16
Frequency specifications -- units	Normalized (0 to 1)
Wpass	0.4

Parameter	Setting
`Wstop`	`0.5`
`Magnitude specifications -- Wpass and Wstop`	`1`

and then click the **Design Filter** button.



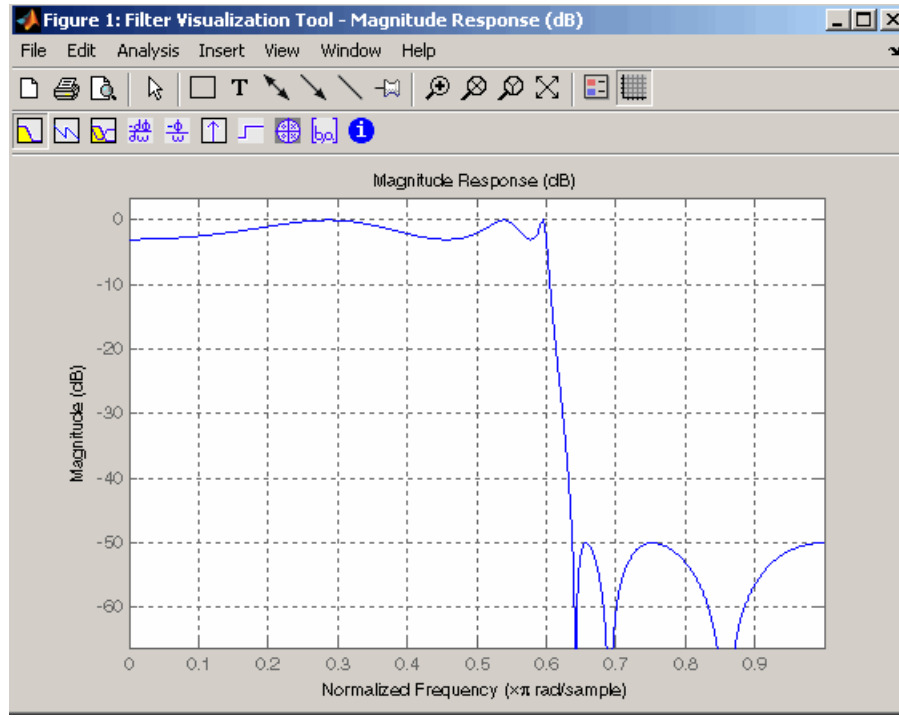
Click the **Full View Analysis** button to start FVTool.



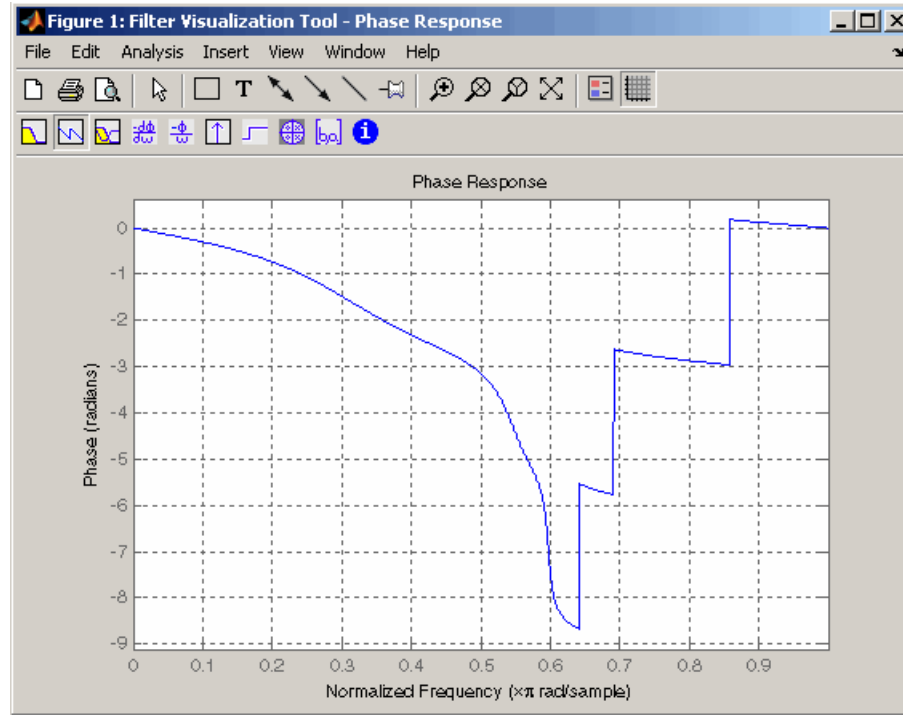
Example 4

Create an elliptic filter and use some of FVTool's figure handle commands:

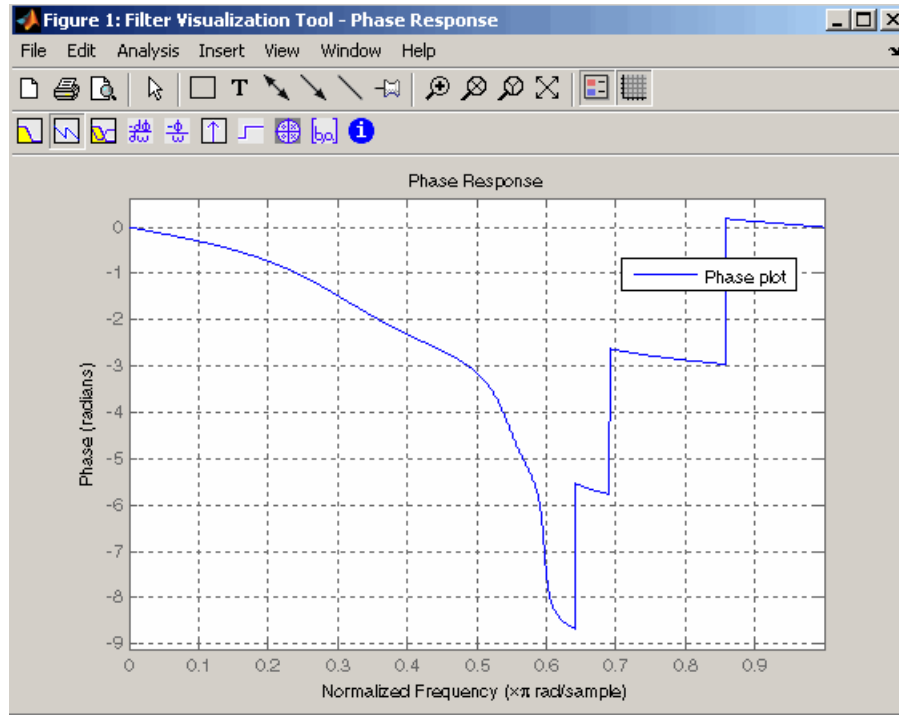
```
[b,a]=ellip(6,3,50,300/500);
h = fvtool(b,a);    % Create handle, h and start FVTool
                    % with magnitude plot
```




```
set(h,'Analysis','phase')   % Change display to phase plot
```




```
set(h,'Legend','on')          % Turn legend on
legend(h,'Phase plot')        % Add legend text
```




```
get(h)              % View all properties
                    % FVTool-specific properties are
                    %  at the end of this list.

            AlphaMap: [1x64 double]
     CloseRequestFcn: 'closereq'
               Color: [0.8314 0.8157 0.7843]
            ColorMap: [64x3 double]
         CurrentAxes: 208.0084
    CurrentCharacter: ''
       CurrentObject: []
        CurrentPoint: [0 0]
        DockControls: 'on'
        DoubleBuffer: 'on'
```

```
                FileName: ''
              FixedColors: [11x3 double]
            IntegerHandle: 'on'
           InvertHardcopy: 'on'
              KeyPressFcn: ''
                  MenuBar: 'none'
              MinColormap: 64
                     Name: 'Filter Visualization Tool - Phase Response'
                 NextPlot: 'new'
              NumberTitle: 'on'
               PaperUnits: 'inches'
          PaperOrientation: 'portrait'
             PaperPosition: [0.2500 2.5000 8 6]
        PaperPositionMode: 'manual'
                PaperSize: [8.5000 11]
                PaperType: 'usletter'
                  Pointer: 'arrow'
        PointerShapeCData: [16x16 double]
      PointerShapeHotSpot: [1 1]
                 Position: [360 292 560 345]
                 Renderer: 'painters'
           RendererMode: 'auto'
                 Resize: 'on'
               ResizeFcn: ''
           SelectionType: 'normal'
                 Toolbar: 'auto'
                   Units: 'pixels'
    WindowButtonDownFcn: ''
  WindowButtonMotionFcn: ''
      WindowButtonUpFcn: ''
            WindowStyle: 'normal'
           BeingDeleted: 'off'
          ButtonDownFcn: ''
               Children: [15x1 double]
               Clipping: 'on'
              CreateFcn: ''
              DeleteFcn: ''
```

```
                BusyAction: 'queue'
          HandleVisibility: 'on'
                   HitTest: 'on'
              Interruptible: 'on'
                     Parent: 0
                   Selected: 'off'
        SelectionHighlight: 'on'
                       Tag: 'filtervisualizationtool'
            UIContextMenu: []
                 UserData: []
                  Visible: 'on'
           AnalysisToolbar: 'on'
             FigureToolbar: 'on'
                   Filters: {[1x1 dfilt.df2t]}
                      Grid: 'on'
                    Legend: 'on'
                DesignMask: 'off'
                        Fs: 1
           SOSViewSettings: [1x1 dspopts.sosview]
                  Analysis: 'phase'
        OverlayedAnalysis: ''
            ShowReference: 'on'
             PolyphaseView: 'off'
    NormalizedFrequency: 'on'
         FrequencyScale: 'Linear'
         FrequencyRange: '[0, pi)'
         NumberofPoints: 8192
        FrequencyVector: [1x256 double]
             PhaseUnits: 'Radians'
           PhaseDisplay: 'Phase'
```

See Also

fdatool, sptool

Purpose

Fast Walsh–Hadamard transform

Syntax

```
y = fwht(x)
y = fwht(x,n)
y = fwht(x,n,ordering)
```

Description

`y = fwht(x)` returns the coefficients of the discrete Walsh–Hadamard transform of the input `x`. If `x` is a matrix, the FWHT is calculated on each column of `x`. The FWHT operates only on signals with length equal to a power of 2. If the length of `x` is less than a power of 2, its length is padded with zeros to the next greater power of two before processing.

`y = fwht(x,n)` returns the `n`-point discrete Walsh–Hadamard transform, where `n` must be a power of 2. `x` and `n` must be the same length. If `x` is longer than `n`, `x` is truncated; if `x` is shorter than `n`, `x` is padded with zeros.

`y = fwht(x,n,ordering)` specifies the ordering to use for the returned Walsh–Hadamard transform coefficients. To specify ordering, you must enter a value for the length `n` or, to use the default behavior, specify an empty vector `[]` for `n`. Valid values for ordering are the following strings:

Ordering	Description
`'sequency'`	Coefficients in order of increasing sequency value, where each row has an additional zero crossing. This is the default ordering.
`'hadamard'`	Coefficients in normal Hadamard order.
`'dyadic'`	Coefficients in Gray code order, where a single bit change occurs from one coefficient to the next.

For more information on the Walsh functions and ordering, see “Walsh–Hadamard Transform” on page 7-45.

Examples

This example shows a simple input signal and the resulting transformed signal.

```
x = [19 -1 11 -9 -7 13 -15 5];
```

```
y = fwht(x)
y =
     2     3     0     4     0     0    10     0
```

y contains nonzero values at these locations: 0, 1, 3, and 6. By forming the Walsh functions with the sequency values of 0, 1, 3, and 6, we can recreate x, as follows.

```
w0 = [1 1 1 1 1 1 1 1];
w1 = [1 1 1 1 -1 -1 -1 -1];
w3 = [1 1 -1 -1 1 1 -1 -1];
w6 = [1 -1 1 -1 -1 1 -1 1];

w = 2*w0 + 3*w1 + 4*w3 + 10*w6;
y1=fwht(w)
y1 =
     2     3     0     4     0     0    10     0

x1 = ifwht(y)
x1 =
    19    -1    11    -9    -7    13   -15     5
```

Algorithm

The fast Walsh-Hadamard tranform algorithm is similar to the Cooley-Tukey algorithm used for the FFT. Both use a butterfly structure to determine the transform coefficients. See the references for details.

References

[1] Beauchamp, K.G., *Applications of Walsh and Related Functions*, Academic Press, 1984.

[2] Beer, T., *Walsh Transforms*, American Journal of Physics, Volume 49, Issue 5, May 1981.

See Also

`ifwht`, `dct`, `idct`, `dwt`, `idwt`, `fft`, `ifft`

Purpose

Gaussian-modulated sinusoidal pulse

Syntax

```
yi = gauspuls(t,fc,bw)
yi = gauspuls(t,fc,bw,bwr)
[yi,yq] = gauspuls(...)
[yi,yq,ye] = gauspuls(...)
tc = gauspuls('cutoff',fc,bw,bwr,tpe)
```

Description

`gauspuls` generates Gaussian-modulated sinusoidal pulses.

`yi = gauspuls(t,fc,bw)` returns a unity-amplitude Gaussian RF pulse at the times indicated in array `t`, with a center frequency `fc` in hertz and a fractional bandwidth `bw`, which must be greater than 0. The default value for `fc` is 1000 Hz and for `bw` is 0.5.

`yi = gauspuls(t,fc,bw,bwr)` returns a unity-amplitude Gaussian RF pulse with a fractional bandwidth of `bw` as measured at a level of `bwr` dB with respect to the normalized signal peak. The fractional bandwidth reference level `bwr` must be less than 0, because it indicates a reference level less than the peak (unity) envelope amplitude. The default value for `bwr` is -6 dB.

`[yi,yq] = gauspuls(...)` returns both the in-phase and quadrature pulses.

`[yi,yq,ye] = gauspuls(...)` returns the RF signal envelope.

`tc = gauspuls('cutoff',fc,bw,bwr,tpe)` returns the cutoff time `tc` (greater than or equal to 0) at which the trailing pulse envelope falls below `tpe` dB with respect to the peak envelope amplitude. The trailing pulse envelope level `tpe` must be less than 0, because it indicates a reference level less than the peak (unity) envelope amplitude. The default value for `tpe` is -60 dB.

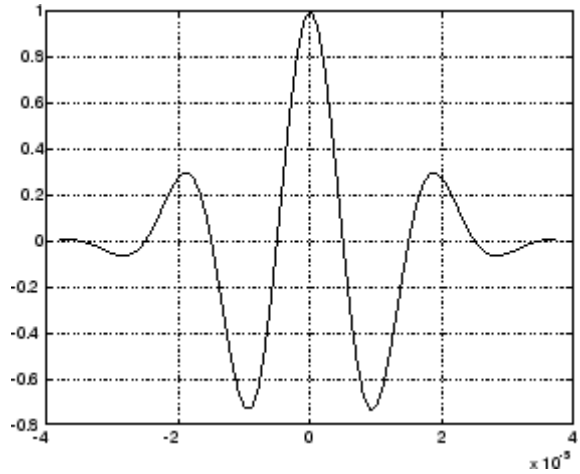
Remarks

Default values are substituted for empty or omitted trailing input arguments.

Examples

Plot a 50 kHz Gaussian RF pulse with 60% bandwidth, sampled at a rate of 1 MHz. Truncate the pulse where the envelope falls 40 dB below the peak:

```
tc = gauspuls('cutoff',50e3,0.6,[],-40);
t = -tc : 1e-6 : tc;
yi = gauspuls(t,50e3,0.6);
plot(t,yi)
```



See Also

chirp, cos, diric, pulstran, rectpuls, sawtooth, sin, sinc, square, tripuls

Purpose

Gaussian FIR pulse-shaping filter

Syntax

```
h = gaussfir(bt)
h = gaussfir(bt,n)
h = gaussfir(bt,n,o)
```

Description

This filter is used primarily in Gaussian minimum shift keying (GMSK) communications applications.

`h = gaussfir(bt)` designs a low pass FIR Gaussian pulse-shaping filter and returns the filter coefficients in the `h` vector. `bt` is the 3-dB bandwidth-symbol time product where `b` is the one-sided bandwidth in hertz and `t` is in seconds. Smaller `bt` products produce larger pulse widths. The number of symbol periods (`n`) defaults to 3 and the oversampling factor (`o`) defaults to 2.

The length of the impulse response of the filter is given by `2*o*n+1`. The coefficients `h` are normalized so that the nominal passband gain is always equal to 1.

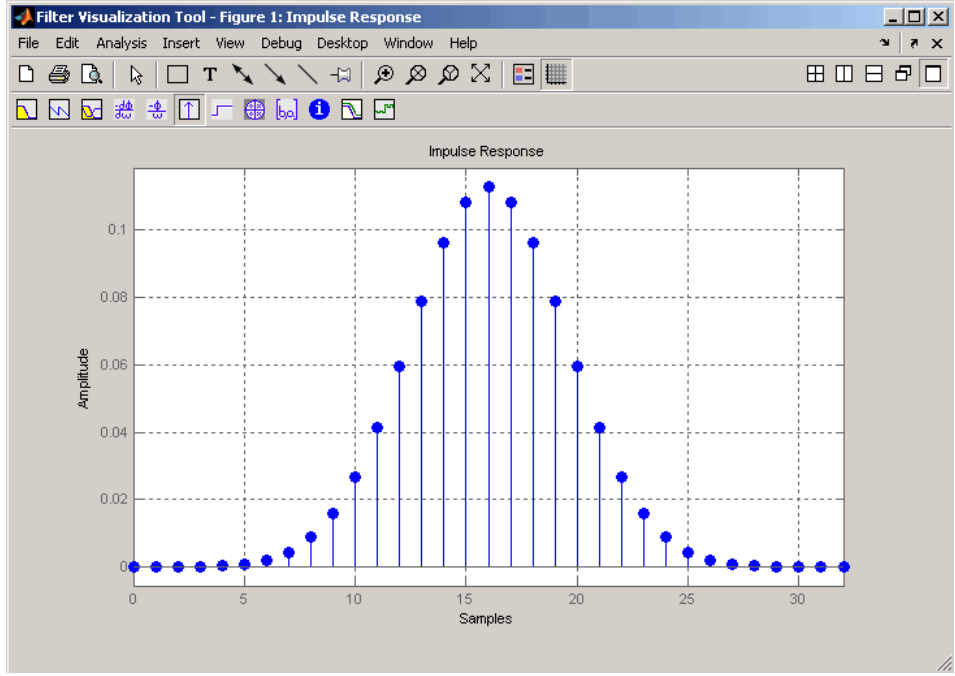
`h = gaussfir(bt,n)` uses `n` number of symbol periods between the start of the filter impulse response and its peak.

`h = gaussfir(bt,n,o)` uses an oversampling factor of `o`, which is the number of samples per symbol.

Examples

Design a Gaussian filter to be used in a Global System for Mobile (GSM) communications GMSK scheme.

```
bt = .3;          % 3-dB bandwidth-symbol time
o = 8;            % Oversampling factor
n = 2;            % 2 symbol periods to the filters peak
h = gaussfir(bt,n,o);
hfvt = fvtool(h,'impulse');
```



References

[1] Rappaport T.S., Wireless Communications Principles and Practice, 2nd Edition, Prentice Hall, 2001.

[2] Krishnapura N., Pavan S., Mathiazhagan C., Ramamurthi B., "A Baseband Pulse Shaping Filter for Gaussian Minimum Shift Keying," *Proceedings of the 1998 IEEE International Symposium on Circuits and Systems*, 1998.

See Also

`firrcos`, `rcosfir`

Purpose

Gaussian window

Syntax

```
w = gausswin(L)
w = gausswin(L,α)
```

Description

`w = gausswin(L)` returns an `L`-point Gaussian window in the column vector `w`. `L` is a positive integer. The coefficients of a Gaussian window are computed from the following equation.

$$w(n) = e^{-\frac{1}{2}\left(\alpha\frac{n}{N/2}\right)^2}$$

where $-\frac{N}{2} \le n \le \frac{N}{2}$, $\alpha \ge 2$ and the window length is $L = N + 1$

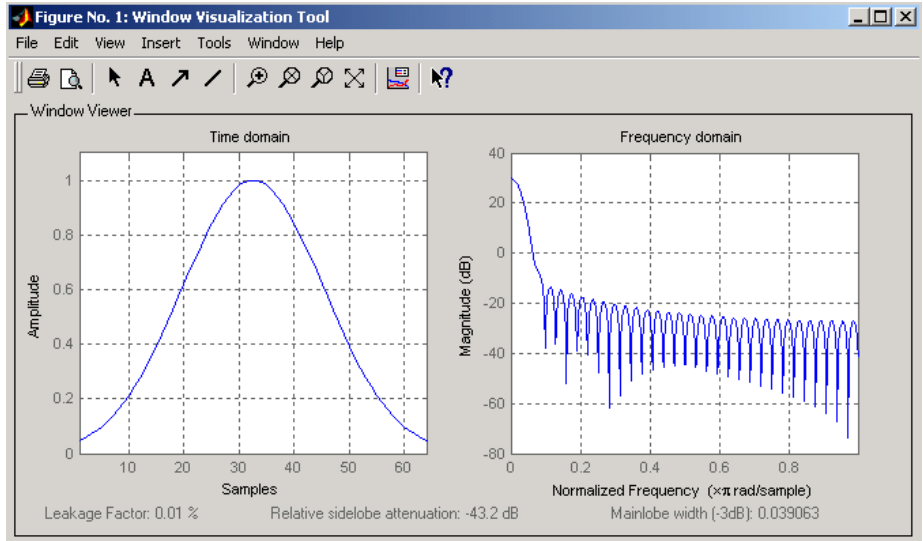
`w = gausswin(L,α)` returns an `L`-point Gaussian window where α is the reciprocal of the standard deviation. The width of the window is inversely related to the value of α; a larger value of α produces a more narrow window. If α is omitted, it defaults to 2.5.

Note If the window appears to be clipped, increase the number of points (`L`) used for `gausswin(n)`.

Examples

Create a 64-point Gaussian window and display the result in WVTool:

```
L=64;
wvtool(gausswin(L))
```



Note The shape of this window is similar in the frequency domain because the Fourier transform of a Gaussian is also a Gaussian.

References

[1] Harris, F.J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform." *Proceedings of the IEEE. Vol. 66*, No. 1 (January 1978).

[2] Roberts, Richard A., and C.T. Mullis. *Digital Signal Processing*. Reading, MA: Addison-Wesley, 1987, pp. 135-136.

See Also

`chebwin`, `kaiser`, `tukeywin`, `window`, `wintool`, `wvtool`

Purpose

Gaussian monopulse

Syntax

```
y = gmonopuls(t,fc)
tc = gmonopuls('cutoff',fc)
```

Description

`y = gmonopuls(t,fc)` returns samples of the unity-amplitude Gaussian monopulse with center frequency `fc` (in hertz) at the times indicated in array `t`. By default, `fc` = 1000 Hz.

`tc = gmonopuls('cutoff',fc)` returns the time duration between the maximum and minimum amplitudes of the pulse.

Remarks

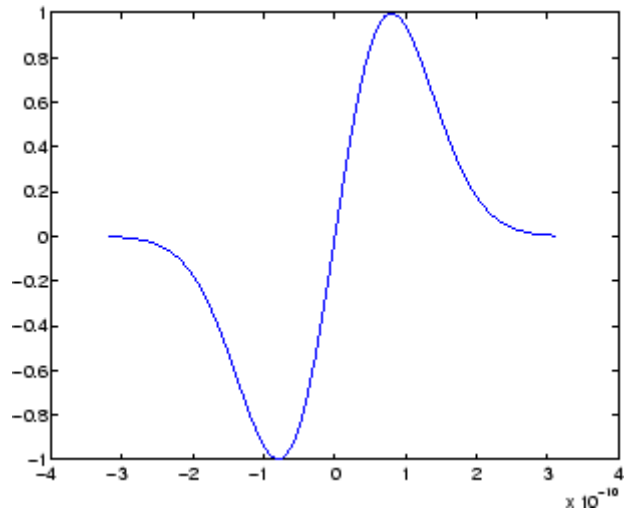
Default values are substituted for empty or omitted trailing input arguments.

Examples

Example 1

Plot a 2 GHz Gaussian monopulse sampled at a rate of 100 GHz:

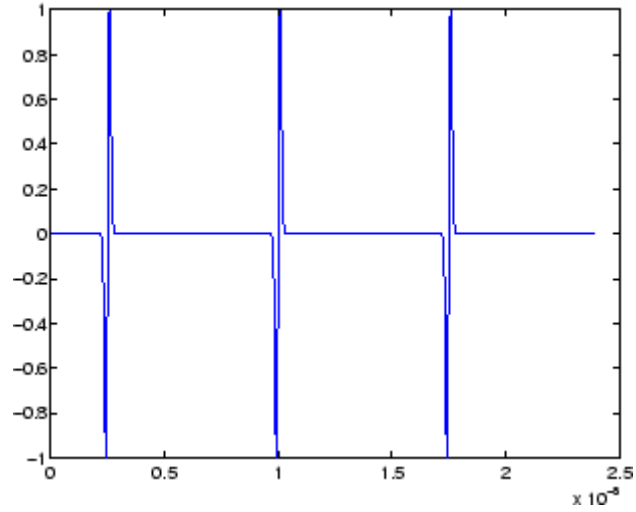
```
fc = 2E9; fs=100E9;
tc = gmonopuls('cutoff',fc);
t  = -2*tc : 1/fs : 2*tc;
y = gmonopuls(t,fc); plot(t,y)
```



Example 2

Construct a pulse train from the monopulse of Example 1 using a spacing of 7.5 ns:

```
fc = 2E9;  fs=100E9;              % Center freq, sample freq
D = [2.5 10 17.5]' * 1e-9;        % Pulse delay times
tc = gmonopuls('cutoff',fc);      % Width of each pulse
t  = 0 : 1/fs : 150*tc;           % Signal evaluation time
yp = pulstran(t,D,@gmonopuls,fc);
plot(t,yp)
```



See Also

chirp, gauspuls, pulstran, rectpuls, tripuls

goertzel

Purpose

Discrete Fourier transform using second-order Goertzel algorithm

Syntax

```
y = goertzel(x,i)
y = goertzel(x,i,dim)
```

Description

`goertzel` computes the discrete Fourier transform (DFT) of specific indices in a vector or matrix.

`y = goertzel(x,i)` returns the DFT of vector `x` at the indices in vector `i`, computed using the second-order Goertzel algorithm. If `x` is a matrix, `goertzel` computes each column separately. The indices in vector `i` must be integer values from 1 to N, where N is the length of the first matrix dimension of `x` that is greater than 1. The resulting `y` has the same dimensions as `x`. If `i` is omitted, it is assumed to be [1:N], which results in a full DFT computation.

`y = goertzel(x,i,dim)` returns the discrete Fourier transform (DFT) of matrix `x` at the indices in vector `i`, computed along the dimension `dim` of `x`.

Note `fft` computes all DFT values at all indices, while `goertzel` computes DFT values at a specified subset of indices (i.e., a portion of the signal's frequency range). If less than $\log_2(N)$ points are required, `goertzel` is more efficient than the Fast Fourier Transform (`fft`).

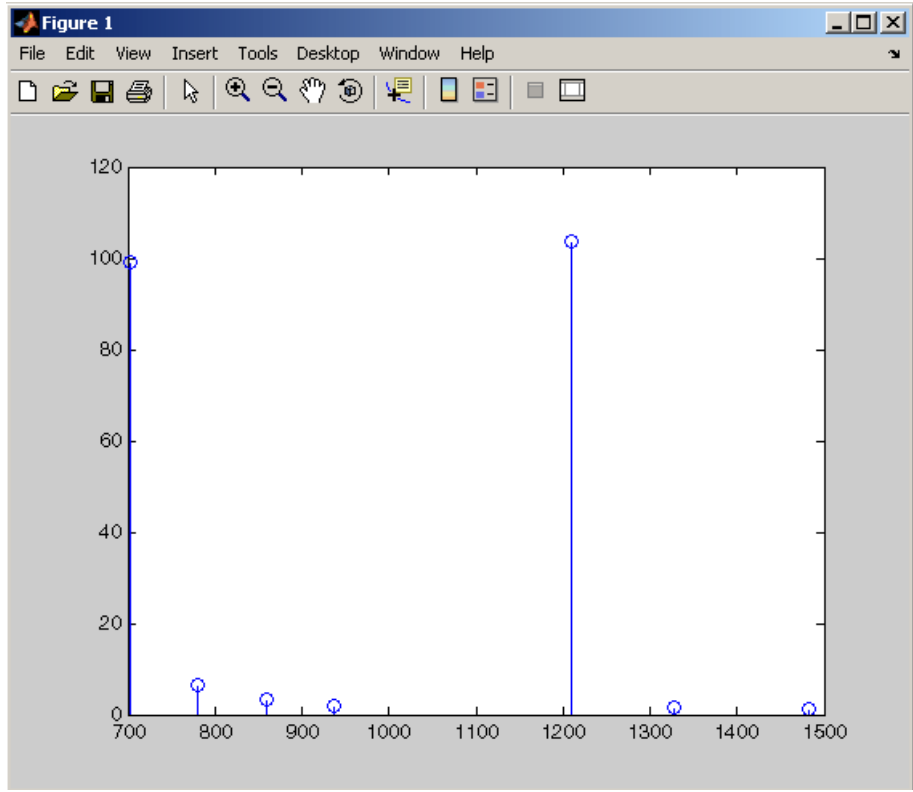
Two examples where `goertzel` can be useful are spectral analysis of very large signals and dual-tone multi-frequency (DTMF) signal detection.

Examples

Estimate the frequency of the two tones generated by the "1" button on a telephone keypad.

```
% Frequency tones of the telephone pad (Hz)
f = [697 770 852 941 1209 1336 1477];
Fs = 8000;
N = 205;
```

```
% Tones of 25.6 ms
tones = sum(sin(2*pi*[697;1209]*(0:N-1)/Fs));

% Indices of the DFT
k = round(f/Fs*N);

% DC is represented by the value 1
ydft = goertzel(tones,k+1);

% Frequencies at which DFT estimated
estim_f = round(k*Fs/N);

% Peaks detected around 697 & 1209 Hz
stem(estim_f,abs(ydft))
```



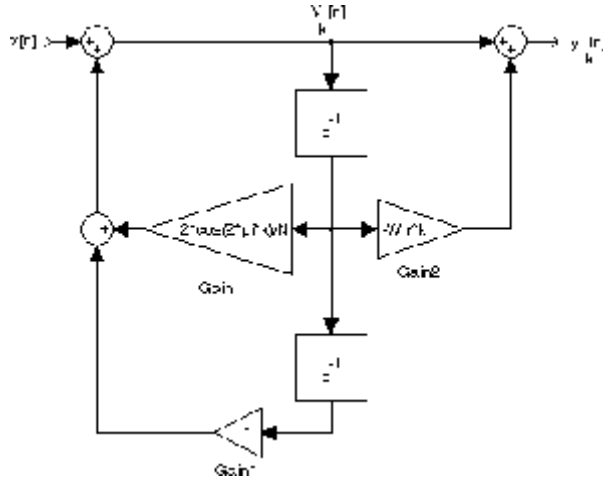
Algorithm

`goertzel` implements this transfer function

$$H_k(z) = \frac{1 - W_N^k z^{-1}}{1 - 2\cos\left(\frac{2\pi}{N}k\right)z^{-1} + z^{-2}}$$

where N is the length of the signal and k is the index of the computed DFT. k is related to the indices in vector `i` above as k = `i` - 1.

The signal flow graph for this transfer function is



and it is implemented as

$$v_k[n] = x_e[n] + 2\cos\left(\frac{2\pi k}{N}\right)v_k[n-1] - v_k[n-2]$$

where

$$x_e(n) = \begin{cases} x(n), 0 \le n \le N-1 \\ 0, n < 0, n \ge N \end{cases}$$

and

$$X[k] = y_k[N] = v_k[N] - W_N^k v_k[N-1]$$

To compute X[k] for a particular k, the Goertzel algorithm requires 4N real multiplications and 4N real additions. Although this is less efficient than computing the DFT by the direct method, Goertzel uses recursion to compute

W_N^k and $\cos\left(\frac{2\pi k}{N}\right)$

which are evaluated only at $n = N$. The direct DFT does not use recursion and must compute each complex term separately.

References

[1] Burrus, C.S. and T.W. Parks. *DFT/FFT and Convolution Algorithms*. John Wiley & Sons, 1985, pp. 32-26.

[2] Mitra, Sanjit K. *Digital Signal Processing: A Computer-Based Approach*. New York, NY: McGraw-Hill, 1998, pp. 520-523.

See Also

`fft`, `fft2`

Purpose

Average filter delay (group delay)

Syntax

```
grpdelay(b,a)
[gd,w] = grpdelay(b,a,l)
[gd,f] = grpdelay(b,a,n,fs)
[gd,w] = grpdelay(b,a,n,'whole')
[gd,f] = grpdelay(b,a,n,'whole', fs)
gd = grpdelay(b,a,w)
gd = grpdelay(b,a,f,fs)
grpdelay(Hd)
```

Description

The *group delay* of a filter is a measure of the average delay of the filter as a function of frequency. It is the negative first derivative of the phase response of the filter. If the complex frequency response of a filter is $H(e^{j\omega})$, then the group delay is

$$\tau_g(\omega) = -\frac{d\theta(\omega)}{d\omega}$$

where ω is frequency and [[THETA]] is the phase angle of $H(e^{j\omega})$.

`grpdelay(b,a)` with no output arguments plots the group delay versus frequency in the current figure window.

`[gd,w] = grpdelay(b,a,l)` returns the `l`-point group delay, $\tau_g(\omega)$, of the digital filter

$$H(z) = \frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \cdots + b(n+1)z^{-n}}{a(1) + a(2)z^{-1} + \cdots + a(m+1)z^{-m}}$$

given the numerator and denominator coefficients in vectors `b` and `a`. `grpdelay` returns both `gd`, the group delay, which has units of samples, and `w`, a vector containing the `n` frequency points in radians. `grpdelay` evaluates the group delay at `n` points equally spaced around the upper half of the unit circle, so `w` contains `n` points between 0 and π.

`[gd,f] = grpdelay(b,a,n,fs)` specifies a positive sampling frequency `fs` in hertz. It returns a length `n` vector `f` containing the actual

frequency points at which the group delay is calculated, also in hertz. `f` contains `n` points between 0 and `fs/2`.

`[gd,w] = grpdelay(b,a,n,'whole')` and

`[gd,f] = grpdelay(b,a,n,'whole', fs)` use `n` points around the whole unit circle (from 0 to 2π, or from 0 to `fs`).

`gd = grpdelay(b,a,w)` and

`gd = grpdelay(b,a,f,fs)` return the group delay evaluated at the points in `w` (in radians) or `f` (in hertz), respectively, where `fs` is the sampling frequency in hertz.

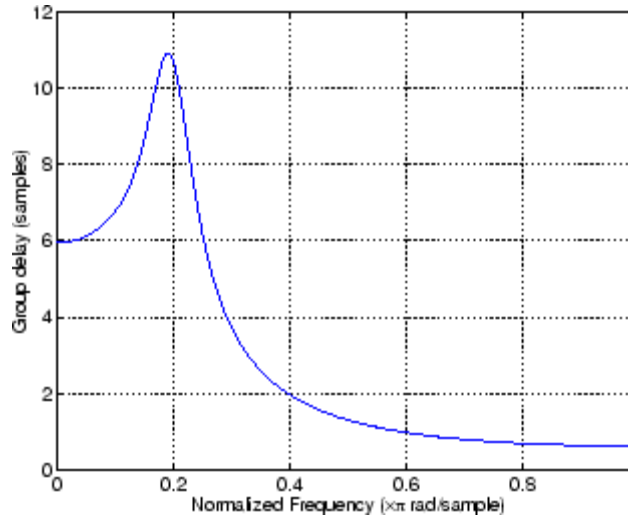
`grpdelay(Hd)` plots the group delay and displays the plot in `fvtool`. The input `Hd` is a `dfilt` filter object or an array of `dfilt` filter objects.

`grpdelay` works for both real and complex filters.

Examples

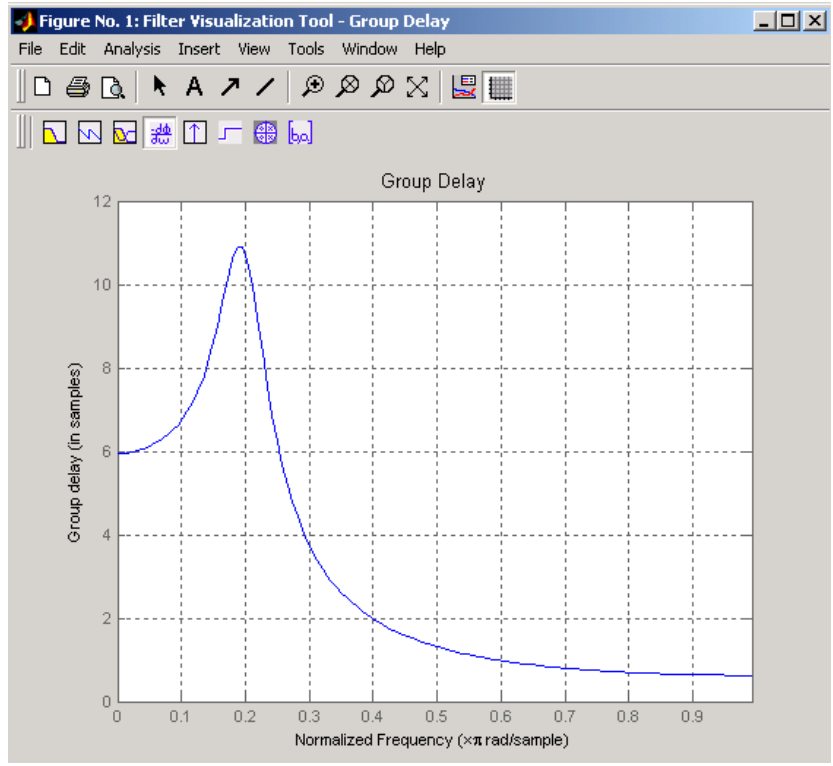
Plot the group delay of Butterworth filter `b(z)/a(z)`:

```
[b,a] = butter(6,0.2);
grpdelay(b,a,128)
```



The same example using a `dfilt` object and displaying the result in the Filter Visualization Tool (`fvtool`) is

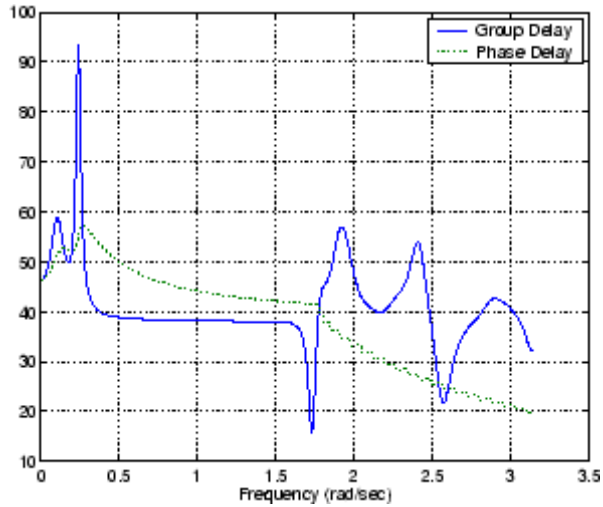
```
[b,a] = butter(6,0.2);
Hd=dfilt.df1(b,a);
grpdelay(Hd,128)
```



Plot both the group and phase delays of a system on the same graph:

```
gd = grpdelay(b,a,512);
gd(1) = [];       % Avoid NaNs
[h,w] = freqz(b,a,512); h(1) = []; w(1) = [];
pd = -unwrap(angle(h))./w;
```

```
plot(w,gd,w,pd,':')
xlabel('Frequency (rad/sec)'); grid;
legend('Group Delay','Phase Delay');
```



Algorithm

`grpdelay` multiplies the filter coefficients by a unit ramp. After Fourier transformation, this process corresponds to differentiation.

See Also

`cceps`, `fft`, `freqz`, `fvtool`, `hilbert`, `icceps`, `rceps`

Purpose

Hamming window

Syntax

```
w = hamming(L)
w = hamming(L,'sflag')
```

Description

`w = hamming(L)` returns an `L`-point symmetric Hamming window in the column vector `w`. `L` should be a positive integer. The coefficients of a Hamming window are computed from the following equation.

$$w(n) = 0.54 - 0.46\cos\left(2\pi\frac{n}{N}\right), \quad 0 \le n \le N$$

The window length is $L = N + 1$.

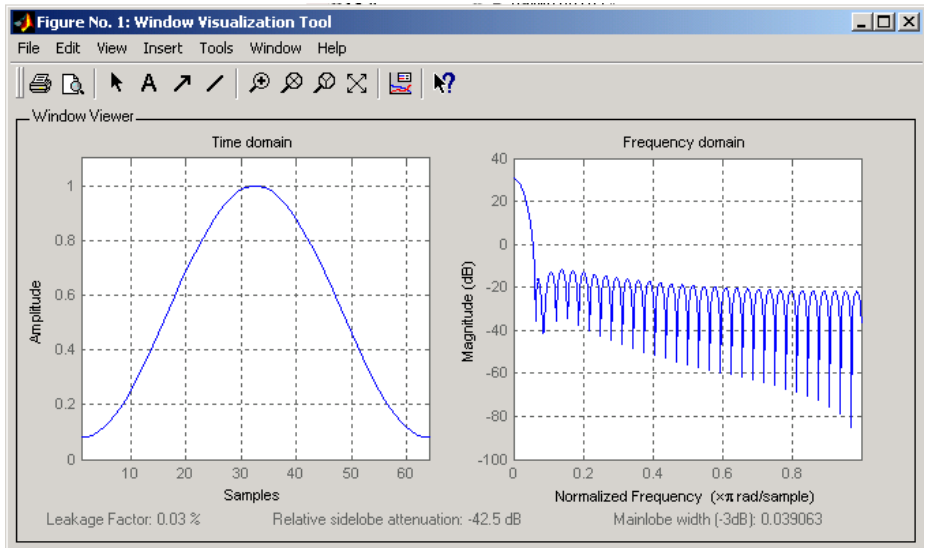
`w = hamming(L,'sflag')` returns an `L`-point Hamming window using the window sampling specified by `'sflag'`, which can be either `'periodic'` or `'symmetric'` (the default). The `'periodic'` flag is useful for DFT/FFT purposes, such as in spectral analysis. The DFT/FFT contains an implicit periodic extension and the periodic flag enables a signal windowed with a periodic window to have perfect periodic extension. When `'periodic'` is specified, `hamming` computes a length `L+1` window and returns the first `L` points. When using windows for filter design, the `'symmetric'` flag should be used.

Note If you specify a one-point window (`L=1`), the value `1` is returned.

Examples

Create a 64-point Hamming window and display the result in WVTool:

```
L=64;
wvtool(hamming(L))
```



References

[1] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, pp. 447-448.

See Also

`blackman`, `flattopwin`, `hann`, `window`, `wintool`, `wvtool`

Purpose

Hann (Hanning) window

Syntax

```
w = hann(L)
w = hann(L,'sflag')
```

Description

`w = hann(L)` returns an `L`-point symmetric Hann window in the column vector `w`. `L` must be a positive integer. The coefficients of a Hann window are computed from the following equation.

$$w(n) = 0.5\left(1 - \cos\left(2\pi\frac{n}{N}\right)\right), \quad 0 \le n \le N$$

The window length is $L = N + 1$.

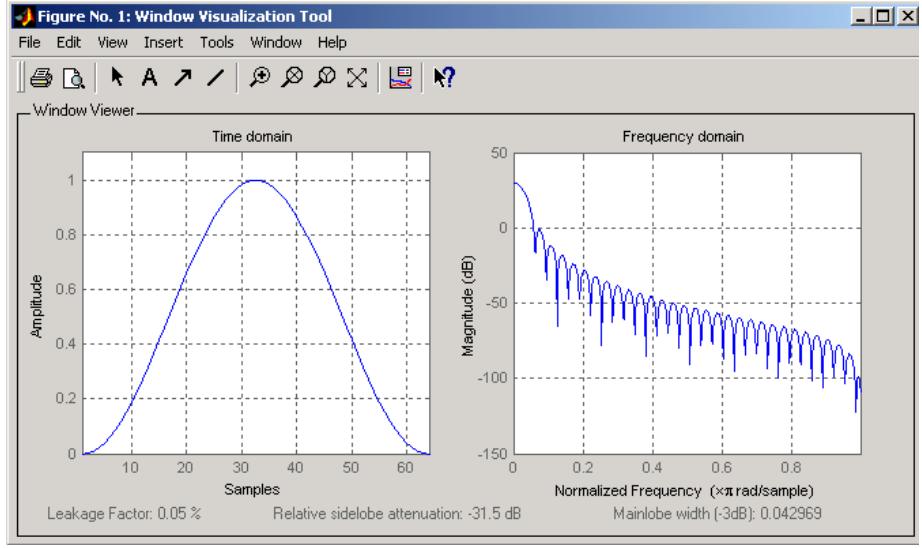
`w = hann(L,'sflag')` returns an `L`-point Hann window using the window sampling specified by *`'sflag'`*, which can be either `'periodic'` or `'symmetric'` (the default). The `'periodic'` flag is useful for DFT/FFT purposes, such as in spectral analysis. The DFT/FFT contains an implicit periodic extension and the periodic flag enables a signal windowed with a periodic window to have perfect periodic extension. When `'periodic'` is specified, `hann` computes a length `L+1` window and returns the first `L` points. When using windows for filter design, the `'symmetric'` flag should be used.

Note If you specify a one-point window (`L=1`), the value `1` is returned.

Examples

Create a 64-point Hann window and display the result in WVTool:

```
L=64;
wvtool(hann(L))
```



References

[1] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, pp. 447-448.

See Also

`blackman`, `flattopwin`, `hamming`, `window`, `wintool`, `wvtool`

Purpose

Discrete-time analytic signal using Hilbert transform

Syntax

```
x = hilbert(xr)
x = hilbert(xr,n)
```

Description

`x = hilbert(xr)` returns a complex helical sequence, sometimes called the *analytic signal*, from a real data sequence. The analytic signal `x = xr + i*xi` has a real part, `xr`, which is the original data, and an imaginary part, `xi`, which contains the Hilbert transform. The imaginary part is a version of the original real sequence with a 90° phase shift. Sines are therefore transformed to cosines and vice versa. The Hilbert transformed series has the same amplitude and frequency content as the original real data and includes phase information that depends on the phase of the original data.

If `xr` is a matrix, `x = hilbert(xr)` operates columnwise on the matrix, finding the Hilbert transform of each column.

`x = hilbert(xr,n)` uses an `n` point FFT to compute the Hilbert transform. The input data `xr` is zero-padded or truncated to length `n`, as appropriate.

The Hilbert transform is useful in calculating instantaneous attributes of a time series, especially the amplitude and frequency. The instantaneous amplitude is the amplitude of the complex Hilbert transform; the instantaneous frequency is the time rate of change of the instantaneous phase angle. For a pure sinusoid, the instantaneous amplitude and frequency are constant. The instantaneous phase, however, is a sawtooth, reflecting the way in which the local phase angle varies linearly over a single cycle. For mixtures of sinusoids, the attributes are short term, or local, averages spanning no more than two or three points.

Reference [1] describes the Kolmogorov method for minimum phase reconstruction, which involves taking the Hilbert transform of the logarithm of the spectral density of a time series. The toolbox function `rceps` performs this reconstruction.

For a discrete-time analytic signal `x`, the last half of `fft(x)` is zero, and the first (DC) and center (Nyquist) elements of `fft(x)` are purely real.

Examples

```
xr = [1 2 3 4];
x = hilbert(xr)
x =
1.0000+1.0000i 2.0000-1.0000i 3.0000-1.0000i 4.0000+1.0000i
```

You can see that the imaginary part, `imag(x)` = `[1 -1 -1 1]`, is the Hilbert transform of `xr`, and the real part, `real(x)` = `[1 2 3 4]`, is simply `xr` itself. Note that the last half of `fft(x)` = `[10 -4+4i -2 0]` is zero (in this example, the last half is just the last element), and that the DC and Nyquist elements of `fft(x)`, 10 and -2 respectively, are purely real.

Algorithm

The analytic signal for a sequence `x` has a *one-sided Fourier transform*, that is, negative frequencies are 0. To approximate the analytic signal, `hilbert` calculates the FFT of the input sequence, replaces those FFT coefficients that correspond to negative frequencies with zeros, and calculates the inverse FFT of the result.

In detail, `hilbert` uses a four-step algorithm:

1. It calculates the FFT of the input sequence, storing the result in a vector `x`.
2. It creates a vector `h` whose elements `h(i)` have the values:
   - 1 for `i = 1, (n/2)+1`
   - 2 for `i = 2, 3, ... , (n/2)`
   - 0 for `i = (n/2)+2, ... , n`
3. It calculates the element-wise product of `x` and `h`.
4. It calculates the inverse FFT of the sequence obtained in step 3 and returns the first `n` elements of the result.

If the input data `xr` is a matrix, `hilbert` operates in a similar manner, extending each step above to handle the matrix case.

References

[1] Claerbout, J.F., *Fundamentals of Geophysical Data Processing*, McGraw-Hill, 1976, pp.59-62.

[2] Marple, S.L., "Computing the discrete-time analytic signal via FFT," IEEE Transactions on Signal Processing, Vol. 47, No. 9 (September 1999), pp. 2600-2603.

[3] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, 2nd ed., Prentice-Hall, 1998.

See Also

`fft`, `ifft`, `rceps`

icceps

Purpose

Inverse complex cepstrum

Syntax

```
x = icceps(xhat,nd)
```

Description

Note `icceps` only works on real data.

`x = icceps(xhat,nd)` returns the inverse complex cepstrum of the real data sequence `xhat`, removing `nd` samples of delay. If `xhat` was obtained with `cceps(x)`, then the amount of delay that was added to `x` was the element of `round(unwrap(angle(fft(x)))/pi)` corresponding to π radians.

References

[1] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989.

See Also

`cceps`, `hilbert`, `rceps`, `unwrap`

Purpose

Inverse discrete cosine transform

Syntax

```
x = idct(y)
x = idct(y,n)
```

Description

The inverse discrete cosine transform reconstructs a sequence from its discrete cosine transform (DCT) coefficients. The `idct` function is the inverse of the `dct` function.

`x = idct(y)` returns the inverse discrete cosine transform of `y`

$$x(n) = \sum_{k=1}^{N} w(k)\,y(k)\cos\frac{\pi(2n-1)(k-1)}{2N}, \qquad n = 1, \ldots, N$$

where

$$w(k) = \begin{cases} \frac{1}{\sqrt{N}}, & k = 1 \\ \sqrt{\frac{2}{N}}, & 2 \le k \le N \end{cases}$$

and N = `length(x)`, which is the same as `length(y)`. The series is indexed from $n = 1$ and $k = 1$ instead of the usual $n = 0$ and $k = 0$ because MATLAB vectors run from 1 to N instead of from 0 to N-1.

`x = idct(y,n)` appends zeros or truncates the vector `y` to length `n` before transforming.

If `y` is a matrix, `idct` transforms its columns.

References

[1] Jain, A.K., *Fundamentals of Digital Image Processing*, Prentice-Hall, 1989.

[2] Pennebaker, W.B., and J.L. Mitchell, *JPEG Still Image Data Compression Standard*, Van Nostrand Reinhold, 1993, Chapter 4.

See Also

`dct`, `dct2`, `idct2`, `ifft`

Purpose

Inverse Fast Walsh–Hadamard transform

Syntax

```
y = iwht(x)
y = ifwht(x,n)
y = ifwht(x,n,ordering)
```

Description

`y = iwht(x)` returns the coefficients of the inverse discrete fast Walsh–Hadamard transform of the input `x`. If `x` is a matrix, the inverse fast Walsh-Hadamard tranform is calculated on each column of `x`. The inverse fast Walsh-Hadamard tranform operates only on signals with length equal to a power of 2. If the length of `x` is less than a power of 2, its length is padded with zeros to the next greater power of two before processing.

`y = ifwht(x,n)` returns the `n`-point inverse discrete Walsh–Hadamard transform, where `n` must be a power of 2.

`y = ifwht(x,n,ordering)` specifies the ordering to use for the returned inverse Walsh–Hadamard transform coefficients. To specify ordering, you must enter a value for the length `n` or, to use the default behavior, specify an empty vector `[]` for `n`. Valid values for ordering are the following strings:

Ordering	Description
`'sequency'`	Coefficients in order of ascending sequency value, where each row has an additional zero crossing. This is the default ordering.
`'hadamard'`	Coefficients in normal Hadamard order.
`'dyadic'`	Coefficients in Gray code order, where a single bit change occurs from one coefficient to the next.

Algorithm

The inverse fast Walsh-Hadamard tranform algorithm is similar to the Cooley-Tukey algorithm used for the inverse FFT. Both use a butterfly structure to determine the transform coefficients. See the references below for details.

References

[1] Beauchamp, K.G., *Applications of Walsh and Related Functions*, Academic Press, 1984.

[2] Beer, T., *Walsh Transforms*, American Journal of Physics, Volume 49, Issue 5, May 1981.

See Also

`fwht`, `dct`, `idct`, `dwt`, `idwt`, `fft`, `ifft`

Purpose

Impulse invariance method for analog-to-digital filter conversion

Syntax

```
[bz,az] = impinvar(b,a,fs)
[bz,az] = impinvar(b,a,fs,tol)
```

Description

`[bz,az] = impinvar(b,a,fs)` creates a digital filter with numerator and denominator coefficients `bz` and `az`, respectively, whose impulse response is equal to the impulse response of the analog filter with coefficients `b` and `a`, scaled by `1/fs`. If you leave out the argument `fs`, or specify `fs` as the empty vector `[]`, it takes the default value of 1 Hz.

`[bz,az] = impinvar(b,a,fs,tol)` uses the tolerance specified by `tol` to determine whether poles are repeated. A larger tolerance increases the likelihood that `impinvar` interprets closely located poles as multiplicities (repeated ones). The default is 0.001, or 0.1% of a pole's magnitude. Note that the accuracy of the pole values is still limited to the accuracy obtainable by the `roots` function.

Examples

Example 1

Convert an analog lowpass filter to a digital filter using `impinvar` with a sampling frequency of 10 Hz:

```
[b,a] = butter(4,0.3,'s');
[bz,az] = impinvar(b,a,10)
bz =
  1.0e-006 *
   -0.0000    0.1324    0.5192    0.1273         0
az =
    1.0000   -3.9216    5.7679   -3.7709    0.9246
```

Example 2

Illustrate the relationship between analog and digital impulse responses [2].

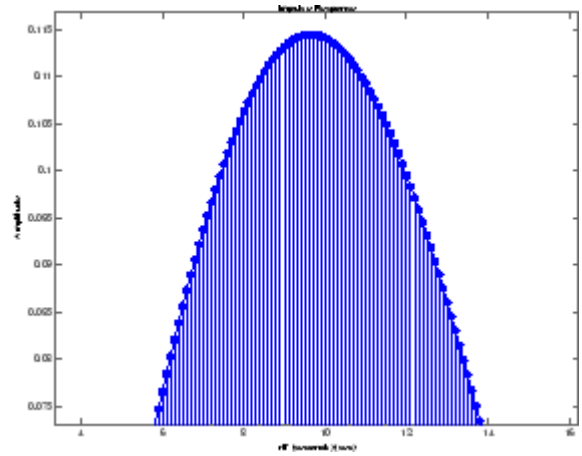
Note This example requires the `impulse` function from Control System Toolbox software.

The steps used in this example are:

1. Create an analog Butterworth filter
2. Use `impinvar` with a sampling frequency `Fs` of 10 Hz to scale the coefficients by 1/`Fs`. This compensates for the gain that will be introduced in Step 4 below.
3. Use Control System Toolbox `impulse` function to plot the continuous-time unit impulse response of an LTI system.
4. Plot the digital impulse response, multiplying the numerator by a constant (`Fs`) to compensate for the 1/`Fs` gain introduced in the impulse response of the derived digital filter.

```
[b,a] = butter(4,0.3,'s');
[bz,az] = impinvar(b,a,10);
sys = tf(b,a);
impulse(sys);
hold on;
impz(10*bz,az,[],10);
```

Zooming the resulting plot shows that the analog and digital impulse responses are the same.



Algorithm

`impinvar` performs the impulse-invariant method of analog-to-digital transfer function conversion discussed in reference [1]:

1. It finds the partial fraction expansion of the system represented by `b` and `a`.
2. It replaces the poles `p` by the poles `exp(p/fs)`.
3. It finds the transfer function coefficients of the system from the residues from step 1 and the poles from step 2.

References

[1] Parks, T.W., and C.S. Burrus, *Digital Filter Design*, John Wiley & Sons, 1987, pp.206-209.

[2] Antoniou, Andreas, *Digital Filters*, McGraw Hill, Inc, 1993, pp.221-224.

See Also

`bilinear`, `lp2bp`, `lp2bs`, `lp2hp`, `lp2lp`

Purpose

Impulse response of digital filter

Syntax

```
[h,t] = impz(b,a)
[h,t] = impz(b,a,n)
[h,t] = impz(b,a,n,fs)
impz(b,a)
impz(Hd)
```

Description

`[h,t] = impz(b,a)` computes the impulse response of the filter with numerator coefficients `b` and denominator coefficients `a`. `impz` chooses the number of samples and returns the response in the column vector `h` and sample times in the column vector `t` (where `t = [0:n-1]'`, and `n = length(t)` is computed automatically).

`[h,t] = impz(b,a,n)` computes n samples of the impulse response when n is an integer (`t = [0:n-1]'`). If 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 `[]` for the second argument, the number of samples is computed automatically by default.

`[h,t] = impz(b,a,n,fs)` computes n samples and produces a vector `t` of length `n` so that the samples are spaced `1/fs` units apart.

`impz(b,a)` with no output arguments plots the impulse response and displays the response in the current figure window.

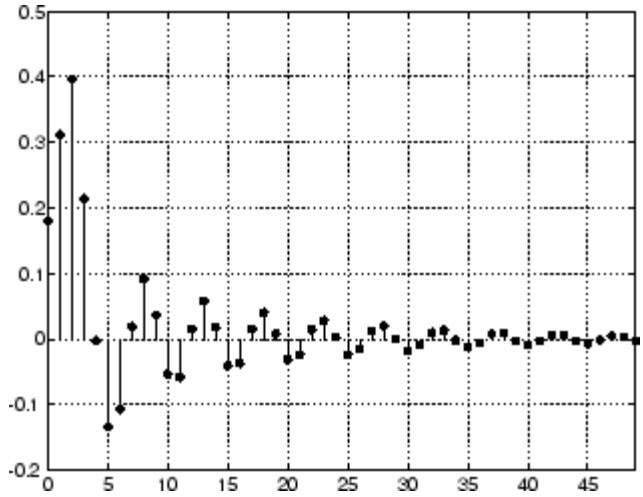
`impz(Hd)` plots the impulse response of the filter and displays the plot in `fvtool`. The input `Hd` is a `dfilt` filter object or an array of `dfilt` filter objects.

Note `impz` works for both real and complex input systems.

Examples

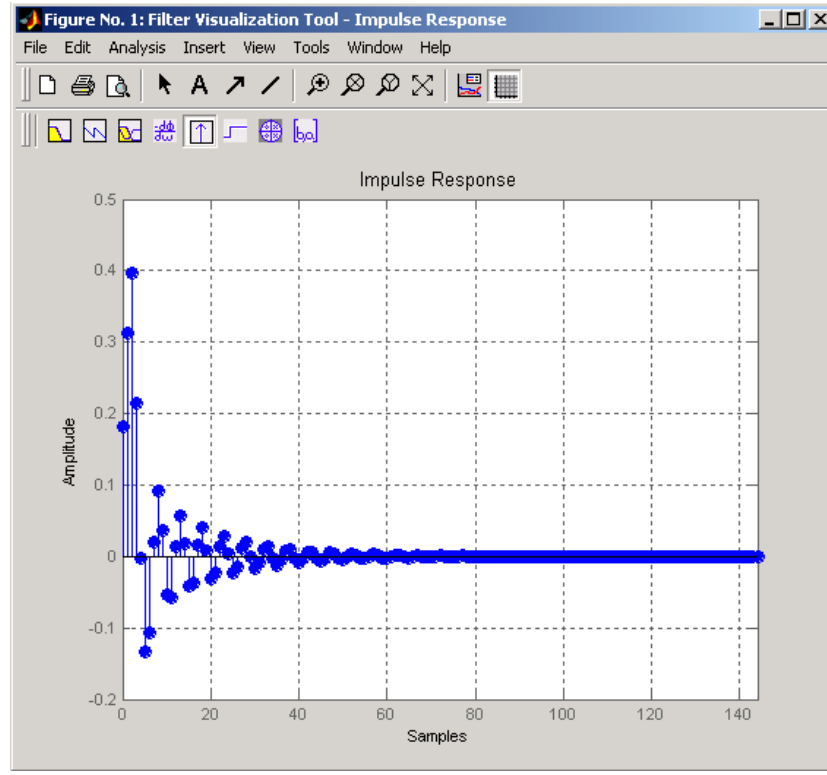
Plot the first 50 samples of the impulse response of a fourth-order lowpass elliptic filter with cutoff frequency of 0.4 times the Nyquist frequency:

```
[b,a] = ellip(4,0.5,20,0.4);
impz(b,a,50)
```



The same example using a `dfilt` object and displaying the result in the Filter Visualization Tool (`fvtool`) is

```
[b,a] = ellip(4,0.5,20,0.4);
Hd = dfilt.df1(b,a)
impz(Hd,50)
```



Algorithm

`impz` filters a length `n` impulse sequence using

```
filter(b,a,[1 zeros(1,n-1)])
```

and plots the results using `stem`.

To compute `n` in the auto-length case, `impz` either uses `n = length(b)` for the FIR case or first finds the poles using `p = roots(a)`, if `length(a)` is greater than `1`.

If the filter is unstable, `n` is chosen to be the point at which the term from the largest pole reaches `10^6` times its original value.

If the filter is stable, `n` is chosen to be the point at which the term due to the largest amplitude pole is `5*10^-5` of its original amplitude.

If the filter is oscillatory (poles on the unit circle only), `impz` computes five periods of the slowest oscillation.

If the filter has both oscillatory and damped terms, `n` is chosen to equal five periods of the slowest oscillation or the point at which the term due to the largest (nonunity) amplitude pole is `5*10^-5` of its original amplitude, whichever is greater.

`impz` also allows for delays in the numerator polynomial. The number of delays is incorporated into the computation for the number of samples.

See Also

`impulse`, `stem`

Purpose

Interpolation — increase sampling rate by integer factor

Syntax

```
y = interp(x,r)
y = interp(x,r,l,alpha)
[y,b] = interp(x,r,l,alpha)
```

Description

Interpolation increases the original sampling rate for a sequence to a higher rate. `interp` performs lowpass interpolation by inserting zeros into the original sequence and then applying a special lowpass filter. The filter returned by `intfilt` is identical to the filter used by `interp`.

`y = interp(x,r)` increases the sampling rate of `x` by a factor of `r`. The interpolated vector `y` is `r` times longer than the original input `x`.

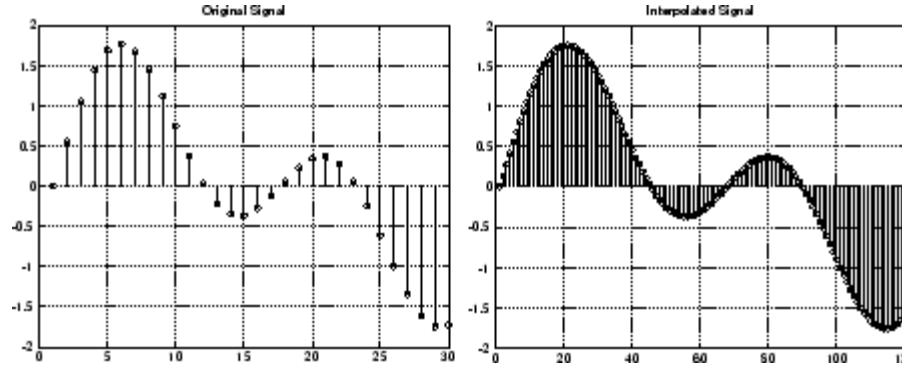
`y = interp(x,r,l,alpha)` specifies `l` (filter length) and `alpha` (cut-off frequency). The default value for `l` is 4 and the default value for `alpha` is 0.5.

`[y,b] = interp(x,r,l,alpha)` returns vector `b` containing the filter coefficients used for the interpolation.

Examples

Interpolate a signal by a factor of four:

```
t = 0:0.001:1;     % Time vector
x = sin(2*pi*30*t) + sin(2*pi*60*t);
y = interp(x,4);
stem(x(1:30));
title('Original Signal');
figure
stem(y(1:120));
title('Interpolated Signal');
```



Algorithm

`interp` uses the lowpass interpolation Algorithm 8.1 described in [1]:

1. It expands the input vector to the correct length by inserting zeros between the original data values.
2. It designs a special symmetric FIR filter that allows the original data to pass through unchanged and interpolates between so that the mean-square errors between the interpolated points and their ideal values are minimized.
3. It applies the filter to the input vector to produce the interpolated output vector.

The length of the FIR lowpass interpolating filter is `2*l*r+1`. The number of original sample values used for interpolation is `2*l`. Ordinarily, `l` should be less than or equal to 10. The original signal is assumed to be band limited with normalized cutoff frequency $0 \le$ `alpha` ≤ 1, where 1 is half the original sampling frequency (the Nyquist frequency). The default value for `l` is 4 and the default value for `alpha` is 0.5.

Diagnostics

If `r` is not an integer, `interp` gives the following error message:

```
Resampling rate R must be an integer.
```

References

[1] *Programs for Digital Signal Processing*, IEEE Press, New York, 1979, Algorithm 8.1.

See Also

`decimate`, `downsample`, `interp1`, `intfiltresample`, `spline`, `upfirdn`, `upsample`

Purpose

Interpolation FIR filter design

Syntax

```
b = intfilt(l,p,alpha)
b = intfilt(l,n,'Lagrange')
```

Description

`b = intfilt(l,p,alpha)` designs a linear phase FIR filter that performs ideal bandlimited interpolation using the nearest `2*p` nonzero samples, when used on a sequence interleaved with `l-1` consecutive zeros every `l` samples. It assumes an original bandlimitedness of alpha times the Nyquist frequency. The returned filter is identical to that used by `interp`. `b` is length 2*l*p-1

alpha is inversely proportional to the transition bandwidth of the filter and it also affects the bandwith of the don't-care regions in the stopband. Specifying alpha allows you to specify how much of the Nyquist interval your input signal occupies. This is beneficial, particularly for signals to be interpolated, because it allows you to increase the transition bandwidth without affecting the interpolation and results in better stopband attenuation for a given `l` and `p`. If you set alpha to 1, your signal is assumed to occupy the entire Nyquist interval. Setting alpha to less than one allows for don't-care regions in the stopband. For example, if your input occupies half the Nyquist interval, you could set alpha to 0.5.

`b = intfilt(l,n,'Lagrange')` designs an FIR filter that performs `n`th-order Lagrange polynomial interpolation on a sequence interleaved with `l-1` consecutive zeros every `r` samples. `b` has length `(n + 1)*l` for `n` even, and length `(n + 1)*l-1` for `n` odd. If both `n` and `l` are even, the filter designed is not linear phase.

Both types of filters are basically lowpass and have a gain of `l` in the passband..

Examples

Design a digital interpolation filter to upsample a signal by four, using the bandlimited method:

```
alpha = 0.5;                % "Bandlimitedness" factor
h1 = intfilt(4,2,alpha);    % Bandlimited interpolation
```

The filter `h1` works best when the original signal is bandlimited to alpha times the Nyquist frequency. Create a bandlimited noise signal:

```
randn('state',0)
x = filter(fir1(40,0.5),1,randn(200,1));   % Bandlimit
```

Now zero pad the signal with three zeros between every sample. The resulting sequence is four times the length of `x`:

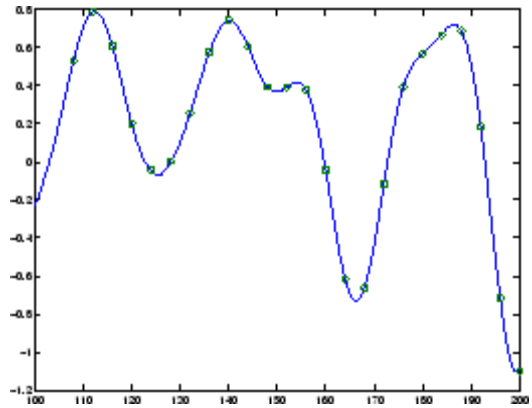
```
xr = reshape([x zeros(length(x),3)]',4*length(x),1);
```

Interpolate using the `filter` command:

```
y = filter(h1,1,xr);
```

`y` is an interpolated version of `x`, delayed by seven samples (the group-delay of the filter). Zoom in on a section of one hundred samples to see this:

```
plot(100:200,y(100:200),7+(101:4:196),x(26:49),'o')
```



`intfilt` also performs Lagrange polynomial interpolation of the original signal. For example, first-order polynomial interpolation is just linear interpolation, which is accomplished with a triangular filter:

```
h2 = intfilt(4,1,'l')       % Lagrange interpolation
h2 =
   0.2500  0.5000  0.7500  1.0000  0.7500  0.5000 0.2500
```

Algorithm

The bandlimited method uses `firls` to design an interpolation FIR equivalent to that presented in [1]. The polynomial method uses Lagrange's polynomial interpolation formula on equally spaced samples to construct the appropriate filter.

References

[1] Oetken, Parks, and Schüßler, "New Results in the Design of Digital Interpolators," *IEEE Trans. Acoust., Speech, Signal Processing*, Vol. ASSP-23 (June 1975), pp. 301-309.

See Also

`decimate`, `downsample`, `interp`, `resample`, `upsample`

Purpose

Identify continuous-time filter parameters from frequency response data

Syntax

```
[b,a] = invfreqs(h,w,n,m)
[b,a] = invfreqs(h,w,n,m,wt)
[b,a] = invfreqs(h,w,n,m,wt,iter)
[b,a] = invfreqs(h,w,n,m,wt,iter,tol)
[b,a] = invfreqs(h,w,n,m,wt,iter,tol,'trace')
[b,a] = invfreqs(h,w,'complex',n,m,...)
```

Description

`invfreqs` is the inverse operation of `freqs`. It finds a continuous-time transfer function that corresponds to a given complex frequency response. From a laboratory analysis standpoint, `invfreqs` is useful in converting magnitude and phase data into transfer functions.

`[b,a] = invfreqs(h,w,n,m)` returns the real numerator and denominator coefficient vectors `b` and `a` of the transfer function

$$H(s) = \frac{B(s)}{A(s)} = \frac{b(1)s^n + b(2)s^{n-1} + \cdots + b(n+1)}{a(1)s^m + a(2)s^{m-1} + \cdots + a(m+1)}$$

whose complex frequency response is given in vector `h` at the frequency points specified in vector `w`. Scalars `n` and `m` specify the desired orders of the numerator and denominator polynomials.

The length of `h` must be the same as the length of `w`. `invfreqs` uses `conj(h)` at `-w` to ensure the proper frequency domain symmetry for a real filter.

`[b,a] = invfreqs(h,w,n,m,wt)` weights the fit-errors versus frequency, where `wt` is a vector of weighting factors the same length as `w`.

`[b,a] = invfreqs(h,w,n,m,wt,iter)` and

`[b,a] = invfreqs(h,w,n,m,wt,iter,tol)` provide a superior algorithm that guarantees stability of the resulting linear system and searches for the best fit using a numerical, iterative scheme. The `iter` parameter tells `invfreqs` to end the iteration when the solution has converged, or after `iter` iterations, whichever comes first. `invfreqs` defines convergence as occurring when the norm of the (modified) gradient

vector is less than `tol`, where `tol` is an optional parameter that defaults to 0.01. To obtain a weight vector of all ones, use

```
invfreqs(h,w,n,m,[],iter,tol)
```

`[b,a] = invfreqs(h,w,n,m,wt,iter,tol,'trace')` displays a textual progress report of the iteration.

`[b,a] = invfreqs(h,w,'complex',n,m,...)` creates a complex filter. In this case no symmetry is enforced, and the frequency is specified in radians between -π and π.

Remarks

When building higher order models using high frequencies, it is important to scale the frequencies, dividing by a factor such as half the highest frequency present in `w`, so as to obtain well conditioned values of `a` and `b`. This corresponds to a rescaling of time.

Examples

Example 1

Convert a simple transfer function to frequency response data and then back to the original filter coefficients:

```
a = [1 2 3 2 1 4]; b = [1 2 3 2 3];
[h,w] = freqs(b,a,64);
[bb,aa] = invfreqs(h,w,4,5)
bb =
    1.0000   2.0000   3.0000   2.0000   3.0000
aa =
    1.0000   2.0000   3.0000   2.0000   1.0000   4.0000
```

Notice that `bb` and `aa` are equivalent to `b` and `a`, respectively. However, `aa` has poles in the right half-plane and thus the system is unstable. Use `invfreqs`'s iterative algorithm to find a stable approximation to the system:

```
[bbb,aaa] = invfreqs(h,w,4,5,[],30)
bbb =
    0.6816   2.1015   2.6694   0.9113  -0.1218
```

```
aaa =
    1.0000    3.4676    7.4060    6.2102    2.5413    0.0001
```

Example 2

Suppose you have two vectors, `mag` and `phase`, that contain magnitude and phase data gathered in a laboratory, and a third vector `w` of frequencies. You can convert the data into a continuous-time transfer function using `invfreqs`:

```
[b,a] = invfreqs(mag.*exp(j*phase),w,2,3);
```

Algorithm

By default, `invfreqs` uses an equation error method to identify the best model from the data. This finds `b` and `a` in

$$\min_{b,a} \sum_{k=1}^{n} wt(k)\left|h(k)A(w(k)) - B(w(k))\right|^2$$

by creating a system of linear equations and solving them with the MATLAB `\` operator. Here $A(w(k))$ and $B(w(k))$ are the Fourier transforms of the polynomials `a` and `b`, respectively, at the frequency $w(k)$, and n is the number of frequency points (the length of `h` and `w`). This algorithm is based on Levi [1]. Several variants have been suggested in the literature, where the weighting function `wt` gives less attention to high frequencies.

The superior ("output-error") algorithm uses the damped Gauss-Newton method for iterative search [2], with the output of the first algorithm as the initial estimate. This solves the direct problem of minimizing the weighted sum of the squared error between the actual and the desired frequency response points.

$$\min_{b,a} \sum_{k=1}^{n} wt(k)\left|h(k) - \frac{B(w(k))}{A(w(k))}\right|^2$$

References

[1] Levi, E.C., "Complex-Curve Fitting," *IRE Trans. on Automatic Control*, Vol.AC-4 (1959), pp.37-44.

[2] Dennis, J.E., Jr., and R.B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Englewood Cliffs, NJ: Prentice-Hall, 1983.

See Also

`freqs`, `freqz`, `invfreqz`, `prony`

Purpose

Identify discrete-time filter parameters from frequency response data

Syntax

```
[b,a] = invfreqz(h,w,n,m)
[b,a] = invfreqz(h,w,n,m,wt)
[b,a] = invfreqz(h,w,n,m,wt,iter)
[b,a] = invfreqz(h,w,n,m,wt,iter,tol)
[b,a] = invfreqz(h,w,n,m,wt,iter,tol,'trace')
[b,a] = invfreqz(h,w,'complex',n,m,...)
```

Description

`invfreqz` is the inverse operation of `freqz`; it finds a discrete-time transfer function that corresponds to a given complex frequency response. From a laboratory analysis standpoint, `invfreqz` can be used to convert magnitude and phase data into transfer functions.

`[b,a] = invfreqz(h,w,n,m)` returns the real numerator and denominator coefficients in vectors `b` and `a` of the transfer function

$$H(z) = \frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \cdots + b(n+1)z^{-n}}{a(1) + a(2)z^{-1} + \cdots + a(m+1)z^{-m}}$$

whose complex frequency response is given in vector `h` at the frequency points specified in vector `w`. Scalars `n` and `m` specify the desired orders of the numerator and denominator polynomials.

Frequency is specified in radians between 0 and π, and the length of `h` must be the same as the length of `w`. `invfreqz` uses `conj(h)` at `-w` to ensure the proper frequency domain symmetry for a real filter.

`[b,a] = invfreqz(h,w,n,m,wt)` weights the fit-errors versus frequency, where `wt` is a vector of weighting factors the same length as `w`.

`[b,a] = invfreqz(h,w,n,m,wt,iter)` and

`[b,a] = invfreqz(h,w,n,m,wt,iter,tol)` provide a superior algorithm that guarantees stability of the resulting linear system and searches for the best fit using a numerical, iterative scheme. The `iter` parameter tells `invfreqz` to end the iteration when the solution has converged, or after `iter` iterations, whichever comes first. `invfreqz` defines convergence as occurring when the norm of the (modified)

gradient vector is less than `tol`, where `tol` is an optional parameter that defaults to 0.01. To obtain a weight vector of all ones, use

```
invfreqz(h,w,n,m,[],iter,tol)
```

`[b,a] = invfreqz(h,w,n,m,wt,iter,tol,'trace')` displays a textual progress report of the iteration.

`[b,a] = invfreqz(h,w,'complex',n,m,...)` creates a complex filter. In this case no symmetry is enforced, and the frequency is specified in radians between -π and π.

Examples

Convert a simple transfer function to frequency response data and then back to the original filter coefficients:

```
a = [1 2 3 2 1 4]; b = [1 2 3 2 3];
[h,w] = freqz(b,a,64);
[bb,aa] = invfreqz(h,w,4,5)
bb =
    1.0000   2.0000   3.0000   2.0000   3.0000
aa =
    1.0000   2.0000   3.0000   2.0000   1.0000   4.0000
```

Notice that `bb` and `aa` are equivalent to `b` and `a`, respectively. However, `aa` has poles outside the unit circle and thus the system is unstable. Use `invfreqz`'s iterative algorithm to find a stable approximation to the system:

```
[bbb,aaa] = invfreqz(h,w,4,5,[],30)
bbb =
    0.2427   0.2788   0.0069   0.0971   0.1980
aaa =
    1.0000  -0.8944   0.6954   0.9997  -0.8933   0.6949
```

Algorithm

By default, `invfreqz` uses an equation error method to identify the best model from the data. This finds `b` and `a` in

$$\min_{b,a} \sum_{k=1}^{n} wt(k)\left|h(k)A(\omega(k)) - B(\omega(k))\right|^2$$

by creating a system of linear equations and solving them with the MATLAB \ operator. Here $A(\omega(k))$ and $B(\omega(k))$ are the Fourier transforms of the polynomials `a` and `b`, respectively, at the frequency $\omega(k)$, and n is the number of frequency points (the length of `h` and `w`). This algorithm is a based on Levi [1].

The superior ("output-error") algorithm uses the damped Gauss-Newton method for iterative search [2], with the output of the first algorithm as the initial estimate. This solves the direct problem of minimizing the weighted sum of the squared error between the actual and the desired frequency response points.

$$\min_{b,a} \sum_{k=1}^{n} wt(k)\left|h(k) - \frac{B(\omega(k))}{A(\omega(k))}\right|^2$$

References

[1] Levi, E.C., "Complex-Curve Fitting," IRE Trans. on Automatic Control, Vol. AC-4 (1959), pp. 37-44.

[2] Dennis, J.E., Jr., and R.B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice-Hall, 1983.

See Also

`freqs`, `freqz`, `invfreqz`, `prony`

is2rc

Purpose

Convert inverse sine parameters to reflection coefficients

Syntax

```
k = is2rc(isin)
```

Description

`k = is2rc(isin)` returns a vector of reflection coefficients `k` from a vector of inverse sine parameters `isin`.

Examples

```
isin = [0.2000 0.8727 0.0020 0.0052 -0.0052];
k = is2rc(isin)

k =
    0.3090    0.9801    0.0031    0.0082   -0.0082
```

References

[1] Deller, J.R., J.G. Proakis, and J.H.L. Hansen, *Discrete-Time Processing of Speech Signals*, Prentice-Hall, 1993.

See Also

`ac2rc`, `lar2rc`, `poly2rc`, `rc2is`

Purpose

Kaiser window

Syntax

```
w = kaiser(L,beta)
```

Description

`w = kaiser(L,beta)` returns an `L`-point Kaiser (I_0I_0 - sinh) window in the column vector `w`. `beta` is the Kaiser window ß parameter that affects the sidelobe attenuation of the Fourier transform of the window. The default value for `beta` is 0.5.

To obtain a Kaiser window that designs an FIR filter with sidelobe attenuation of α dB, use the following ß.

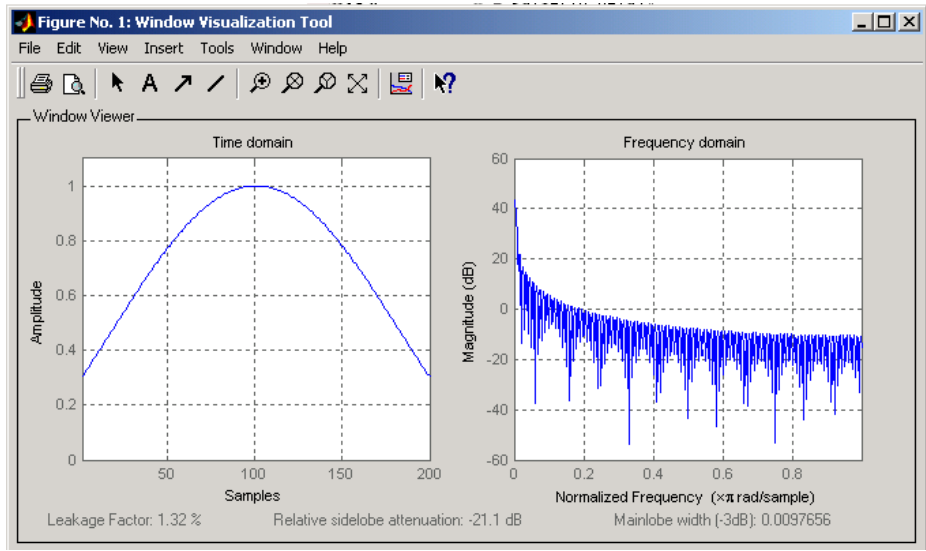
$$\beta = \begin{cases} 0.1102(\alpha - 8.7), & \alpha > 50 \\ 0.5842(\alpha - 21)^{0.4} + 0.07886(\alpha - 21), & 50 \geq \alpha \geq 21 \\ 0, & \alpha < 21 \end{cases}$$

Increasing `beta` widens the main lobe and decreases the amplitude of the sidelobes (i.e., increases the attenuation).

Examples

Create a 200-point Kaiser window with a beta of 2.5 and display the result using WVTool:

```
w = kaiser(200,2.5);
wvtool(w)
```



References

[1] Kaiser, J.F., "Nonrecursive Digital Filter Design Using the I0- sinh Window Function," Proc. 1974 *IEEE Symp. Circuits and Systems*, (April 1974), pp. 20-23.

[2] *Selected Papers in Digital Signal Processing II*, IEEE Press, New York, 1975.

[3] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, p. 453.

See Also

`chebwin`, `gausswin`, `kaiserord`, `tukeywin`, `window`, `wintool`, `wvtool`

Purpose

Kaiser window FIR filter design estimation parameters

Syntax

```
[n,Wn,beta,ftype] = kaiserord(f,a,dev)
[n,Wn,beta,ftype] = kaiserord(f,a,dev,fs)
c = kaiserord(f,a,dev,fs,'cell')
```

Description

`kaiserord` returns a filter order `n` and `beta` parameter to specify a Kaiser window for use with the `fir1` function. Given a set of specifications in the frequency domain, `kaiserord` estimates the minimum FIR filter order that will approximately meet the specifications. `kaiserord` converts the given filter specifications into passband and stopband ripples and converts cutoff frequencies into the form needed for windowed FIR filter design.

`[n,Wn,beta,ftype] = kaiserord(f,a,dev)` finds the approximate order `n`, normalized frequency band edges `Wn`, and weights that meet input specifications `f`, `a`, and `dev`. `f` is a vector of band edges and `a` is a vector specifying the desired amplitude on the bands defined by `f`. The length of `f` is twice the length of `a`, minus 2. Together, `f` and `a` define a desired piecewise constant response function. `dev` is a vector the same size as `a` that specifies the maximum allowable error or deviation between the frequency response of the output filter and its desired amplitude, for each band. The entries in `dev` specify the passband ripple and the stopband attenuation. You specify each entry in `dev` as a positive number, representing absolute filter gain (not in decibels).

Note If, in the vector `dev`, you specify unequal deviations across bands, the minimum specified deviation is used, since the Kaiser window method is constrained to produce filters with minimum deviation in all of the bands.

`fir1` can use the resulting order `n`, frequency vector `Wn`, multiband magnitude type `ftype`, and the Kaiser window parameter `beta`. The `ftype` string is intended for use with `fir1`; it is equal to `'high'` for a highpass filter and `'stop'` for a bandstop filter. For multiband filters, it

can be equal to `'dc-0'` when the first band is a stopband (starting at $f = 0$) or `'dc-1'` when the first band is a passband.

To design an FIR filter `b` that approximately meets the specifications given by `kaiser` parameters `f`, `a`, and `dev`, use the following command.

```
b = fir1(n,Wn,kaiser(n+1,beta),ftype,'noscale')
```

`[n,Wn,beta,ftype] = kaiserord(f,a,dev,fs)` uses a sampling frequency `fs` in Hz. If you don't specify the argument `fs`, or if you specify it as the empty vector `[]`, it defaults to 2 Hz, and the Nyquist frequency is 1 Hz. You can use this syntax to specify band edges scaled to a particular application's sampling frequency. The frequency band edges in `f` must be from 0 to `fs`/2.

`c = kaiserord(f,a,dev,fs,'cell')` is a cell-array whose elements are the parameters to `fir1`.

Note In some cases, `kaiserord` underestimates or overestimates the order `n`. If the filter does not meet the specifications, try a higher order such as `n+1`, `n+2`, and so on, or a try lower order.

Results are inaccurate if the cutoff frequencies are near 0 or the Nyquist frequency, or if `dev` is large (greater than 10%).

Remarks

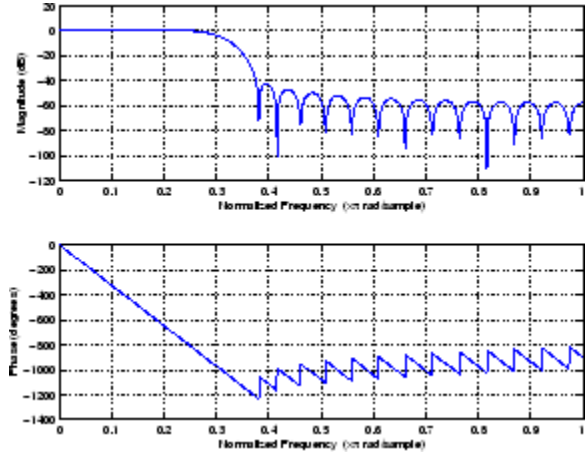
Be careful to distinguish between the meanings of filter length and filter order. The filter *length* is the number of impulse response samples in the FIR filter. Generally, the impulse response is indexed from $n = 0$ to $n = L\text{-}1$, where L is the filter length. The filter *order* is the highest power in a z-transform representation of the filter. For an FIR transfer function, this representation is a polynomial in z, where the highest power is z^{L-1} and the lowest power is z^0. The filter order is one less than the length (L-1) and is also equal to the number of zeros of the z polynomial.

Examples

Example 1

Design a lowpass filter with passband defined from 0 to 1 kHz and stopband defined from 1500 Hz to 4 kHz. Specify a passband ripple of 5% and a stopband attenuation of 40 dB:

```
fsamp = 8000;
fcuts = [1000 1500];
mags = [1 0];
devs = [0.05 0.01];
[n,Wn,beta,ftype] = kaiserord(fcuts,mags,devs,fsamp);
hh = fir1(n,Wn,ftype,kaiser(n+1,beta),'noscale');
freqz(hh)
```

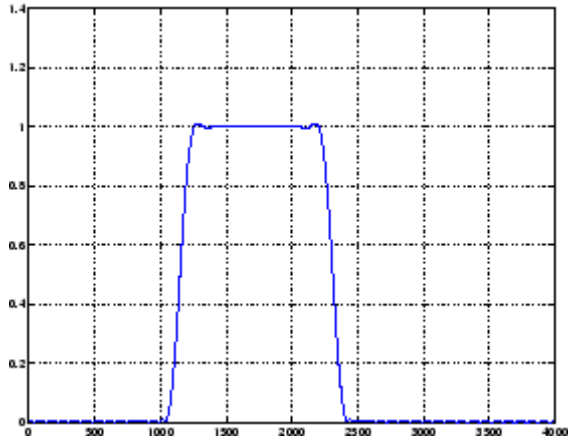


Example 2

Design an odd-length bandpass filter (note that odd length means even order, so the input to `fir1` must be an even integer):

```
fsamp = 8000;
fcuts = [1000 1300 2210 2410];
mags = [0 1 0];
devs = [0.01 0.05 0.01];
[n,Wn,beta,ftype] = kaiserord(fcuts,mags,devs,fsamp);
```

```
n = n + rem(n,2);
hh = fir1(n,Wn,ftype,kaiser(n+1,beta),'noscale');
[H,f] = freqz(hh,1,1024,fsamp);
plot(f,abs(H)), grid on
```



Example 3

Design a lowpass filter with a passband cutoff of 1500 Hz, a stopband cutoff of 2000 Hz, passband ripple of 0.01, stopband ripple of 0.1, and a sampling frequency of 8000 Hz:

```
[n,Wn,beta,ftype] = kaiserord([1500 2000],[1 0],...
                                [0.01 0.1],8000);
b = fir1(n,Wn,ftype,kaiser(n+1,beta),'noscale');
```

This is equivalent to

```
c = kaiserord([1500 2000],[1 0],[0.01 0.1],8000,'cell');
b = fir1(c{:});
```

Algorithm

`kaiserord` uses empirically derived formulas for estimating the orders of lowpass filters, as well as differentiators and Hilbert transformers. Estimates for multiband filters (such as bandpass filters) are derived from the lowpass design formulas.

The design formulas that underlie the Kaiser window and its application to FIR filter design are

$$\beta = \begin{cases} 0.1102(\alpha - 8.7), & \alpha > 50 \\ 0.5842(\alpha - 21)^{0.4} + 0.07886(\alpha - 21), & 50 \geq \alpha \geq 21 \\ 0, & \alpha < 21 \end{cases}$$

where $\alpha = -20\log_{10}\delta$ is the stopband attenuation expressed in decibels (recall that $\delta_p = \delta_s$ is required).

The design formula is

$$n = \frac{\alpha - 7.95}{2.285(\Delta\omega)}$$

where n is the filter order and $\Delta\omega$ is the width of the smallest transition region.

References

[1] Kaiser, J.F., "Nonrecursive Digital Filter Design Using the - sinh Window Function," Proc. 1974 IEEE Symp. *Circuits and Systems*, (April 1974), pp. 20-23.

[2] *Selected Papers in Digital Signal Processing II*, IEEE Press, New York, 1975, pp. 123-126.

[3] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, pp. 458-562.

See Also

`fir1`, `kaiser`, `firpmord`

kaiserwin

Purpose

Kaiser window filter from specification object

Syntax

```
h = design(d,'kaiserwin')
h = design(d,'kaiserwin',designoption,value,designoption,...
value,...)
```

Description

`h = design(d,'kaiserwin')` designs a digital filter `hd`, or a multirate filter `hm` that uses a Kaiser window. For `kaiserwin` to work properly, the filter order in the specifications object must be even. In addition, higher order filters (filter order greater than 120) tend to be more accurate for smaller transition widths. `kaiserwin` returns a warning when your filter order may be too low to design your filter accurately.

`h = design(d,'kaiserwin',designoption,value,designoption,... value,...)` returns a filter where you specify design options as input arguments and the design process uses the Kaiser window technique.

To determine the available design options, use `designmethods` with the specification object and the design method as input arguments as shown.

```
designopts(d,'method')
```

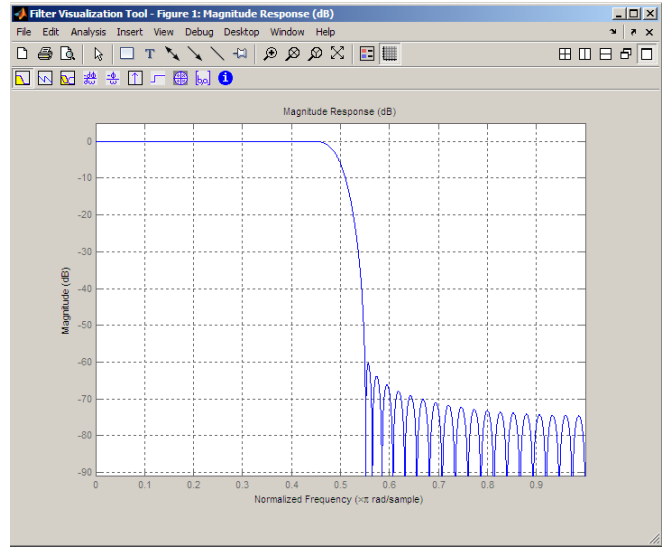
For complete help about using `kaiserwin`, refer to the command line help system. For example, to get specific information about using `kaiserwin` with `d`, the specification object, enter the following at the MATLAB prompt.

```
help(d,'kaiserwin')
```

Examples

This example designs a direct form FIR filter from a lowpass filter specification object.

```
d=fdesign.lowpass;
Hd=design(d,'kaiserwin');
fvtool(Hd)
```



See Also

`design`, `designmethods`, `fdesign`.

lar2rc

Purpose

Convert log area ratio parameters to reflection coefficients

Syntax

```
k = lar2rc(g)
```

Description

`k = lar2rc(g)` returns a vector of reflection coefficients `k` from a vector of log area ratio parameters `g`.

Examples

```
g = [0.6389    4.5989    0.0063    0.0163   -0.0163];
k = lar2rc(g)
k =
    0.3090    0.9801    0.0031    0.0081   -0.0081
```

References

[1] Deller, J.R., J.G. Proakis, and J.H.L. Hansen, *Discrete-Time Processing of Speech Signals*, Prentice-Hall, 1993.

See Also

`ac2rc`, `is2rc`, `poly2rc`, `rc2lar`

Purpose

Convert lattice filter parameters to transfer function form

Syntax

```
[num,den] = latc2tf(k,v)
[num,den] = latc2tf(k,'iiroption')
num = latc2tf(k,'firoption')
```

Description

`[num,den] = latc2tf(k,v)` finds the transfer function numerator `num` and denominator `den` from the IIR lattice coefficients `k` and ladder coefficients `v`.

`[num,den] = latc2tf(k,'iiroption')` produces an IIR filter transfer function according to the value of the string `'iiroption'`:

- `'allpole'`: Produces an all-pole filter transfer function from the associated all-pole IIR lattice filter coefficients `k`.
- `'allpass'`: Produces an allpass filter transfer function from the associated allpass IIR lattice filter coefficients `k`.

`num = latc2tf(k,'firoption')` produces an FIR filter according to the value of the string `'firoption'`:

- `'min'`: Produces a minimum-phase FIR filter numerator from the associated minimum-phase FIR lattice filter coefficients `k`.
- `'max'`: Produces a maximum-phase FIR filter numerator from the associated maximum-phase FIR lattice filter coefficients `k`.
- `'FIR'`: Produces a general FIR filter numerator from the lattice filter coefficients `k` (default, if you leave off the string altogether).

See Also

`latcfilt`, `tf2latc`

latcfilt

Purpose

Lattice and lattice-ladder filter implementation

Syntax

```
[f,g] = latcfilt(k,x)
[f,g] = latcfilt(k,v,x)
[f,g] = latcfilt(k,1,x)
[f,g,zf] = latcfilt(...,'ic',zi)
[f,g,zf] = latcfilt(...,dim)
```

Description

When filtering data, lattice coefficients can be used to represent

- FIR filters
- All-pole IIR filters
- Allpass IIR filters
- General IIR filters

`[f,g] = latcfilt(k,x)` filters `x` with the FIR lattice coefficients in the vector `k`. The forward lattice filter result is `f` and `g` is the backward

filter result. If $|k| \le 1$, `f` corresponds to the minimum-phase output, and `g` corresponds to the maximum-phase output.

If `k` and `x` are vectors, the result is a (signal) vector. Matrix arguments are permitted under the following rules:

- If `x` is a matrix and `k` is a vector, each column of `x` is processed through the lattice filter specified by `k`.
- If `x` is a vector and `k` is a matrix, each column of `k` is used to filter `x`, and a signal matrix is returned.
- If `x` and `k` are both matrices with the same number of columns, then the *i*th column of `k` is used to filter the *i*th column of `x`. A signal matrix is returned.

`[f,g] = latcfilt(k,v,x)` filters `x` with the IIR lattice coefficients `k` and ladder coefficients `v`. Both `k` and `v` must be vectors, while `x` can be a signal matrix.

`[f,g] = latcfilt(k,1,x)` filters `x` with the IIR lattice specified by `k`, where `k` and `x` can be vectors or matrices. `f` is the all-pole lattice filter result and `g` is the allpass filter result.

`[f,g,zf] = latcfilt(...,'ic',zi)` accepts a length-`k` vector `zi` specifying the initial condition of the lattice states. Output `zf` is a length-`k` vector specifying the final condition of the lattice states.

`[f,g,zf] = latcfilt(...,dim)` filters `x` along the dimension `dim`. To specify a `dim` value, the FIR lattice coefficients `k` must be a vector and you must specify all previous input parameters in order. Use the empty vector [ ] for any parameters you do not want to specify. `zf` returns the final conditions in columns, regardless of the shape of `x`.

See Also

`filter`, `latc2tf`, `tf2latc`

levinson

Purpose

Levinson-Durbin recursion

Syntax

```
a = levinson(r,n)
[a,e] = levinson(r,n)
[a,e,k] = levinson(r,n)
```

Description

The Levinson-Durbin recursion is an algorithm for finding an all-pole IIR filter with a prescribed deterministic autocorrelation sequence. It has applications in filter design, coding, and spectral estimation. The filter that `levinson` produces is minimum phase.

`a = levinson(r,n)` finds the coefficients of an `n`th-order autoregressive linear process which has `r` as its autocorrelation sequence.`r` is a real or complex deterministic autocorrelation sequence. If `r` is a matrix, `levinson` finds the coefficients for each column of `r` and returns them in the rows of `a`. `n` is the order of denominator polynomial $A(z)$; that is, `a = [1 a(2) ... a(n+1)]`. The filter coefficients are ordered in descending powers of z.

$$H(z) = \frac{1}{A(z)} = \frac{1}{1 + a(2)z^{-1} + \cdots + a(n+1)z^{-n}}$$

`[a,e] = levinson(r,n)` returns the prediction error, e, of order n.

`[a,e,k] = levinson(r,n)` returns the reflection coefficients `k` as a column vector of length `n`.

Note `k` is computed internally while computing the `a` coefficients, so returning `k` simultaneously is more efficient than converting `a` to `k` with `tf2latc`.

Algorithm

`levinson` solves the symmetric Toeplitz system of linear equations

$$\begin{bmatrix} r(1) & r(2)^* & \cdots & r(n)^* \\ r(2) & r(1) & \cdots & r(n-1)^* \\ \vdots & \ddots & \ddots & \vdots \\ r(n) & \cdots & r(2) & r(1) \end{bmatrix} \begin{bmatrix} a(2) \\ a(3) \\ \vdots \\ a(n+1) \end{bmatrix} = \begin{bmatrix} -r(2) \\ -r(3) \\ \vdots \\ -r(n+1) \end{bmatrix}$$

where `r = [`$r(1)$ ... $r(n+1)$`]` is the input autocorrelation vector, and $r(i)^*$ denotes the complex conjugate of $r(i)$. The input `r` is typically a vector of autocorrelation coefficients where lag 0 is the first element $r(1)$. The algorithm requires $O(n^2)$ flops and is thus much more efficient than the MATLAB `\` command for large `n`. However, the `levinson` function uses `\` for low orders to provide the fastest possible execution.

References

[1] Ljung, L., *System Identification: Theory for the User*, Prentice-Hall, 1987, pp. 278-280.

See Also

`lpc`, `prony`, `rlevinson`, `schurrc`, `stmcb`

lp2bp

Purpose

Transform lowpass analog filters to bandpass

Syntax

```
[bt,at] = lp2bp(b,a,Wo,Bw)
[At,Bt,Ct,Dt] = lp2bp(A,B,C,D,Wo,Bw)
```

Description

`lp2bp` transforms analog lowpass filter prototypes with a cutoff angular frequency of 1 rad/s into bandpass filters with desired bandwidth and center frequency. The transformation is one step in the digital filter design process for the `butter`, `cheby1`, `cheby2`, and `ellip` functions.

`lp2bp` can perform the transformation on two different linear system representations: transfer function form and state-space form. In both cases, the input system must be an analog filter prototype.

Transfer Function Form (Polynomial)

`[bt,at] = lp2bp(b,a,Wo,Bw)` transforms an analog lowpass filter prototype given by polynomial coefficients into a bandpass filter with center frequency `Wo` and bandwidth `Bw`. Row vectors `b` and `a` specify the coefficients of the numerator and denominator of the prototype in descending powers of *s*.

$$\frac{b(s)}{a(s)} = \frac{b(1)s^n + \cdots + b(n)s + b(n+1)}{a(1)s^m + \cdots + a(m)s + a(m+1)}$$

Scalars `Wo` and `Bw` specify the center frequency and bandwidth in units of rad/s. For a filter with lower band edge `w1` and upper band edge `w2`, use `Wo = sqrt(w1*w2)` and `Bw = w2-w1`.

`lp2bp` returns the frequency transformed filter in row vectors `bt` and `at`.

State-Space Form

`[At,Bt,Ct,Dt] = lp2bp(A,B,C,D,Wo,Bw)` converts the continuous-time state-space lowpass filter prototype in matrices `A`, `B`, `C`, `D` shown below

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

into a bandpass filter with center frequency `Wo` and bandwidth `Bw`. For a filter with lower band edge `w1` and upper band edge `w2`, use `Wo = sqrt(w1*w2)` and `Bw = w2-w1`.

The bandpass filter is returned in matrices `At`, `Bt`, `Ct`, `Dt`.

Algorithm

`lp2bp` is a highly accurate state-space formulation of the classic analog filter frequency transformation. Consider the state-space system

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

where u is the input, x is the state vector, and y is the output. The Laplace transform of the first equation (assuming zero initial conditions) is

$$sX(s) = AX(s) + BU(s)$$

Now if a bandpass filter is to have center frequency ω_0 and bandwidth B_w, the standard s-domain transformation is

$$s = Q(p^2 + 1)/p$$

where $Q = \omega_0/B_w$ and $p = s/\omega_0$. Substituting this for s in the Laplace transformed state-space equation, and considering the operator p as d/dt results in

$$Q\ddot{x} + Qx = \dot{A}x + B\dot{u}$$

or

$$Q\ddot{x} - \dot{A}x - B\dot{u} = -Qx$$

Now define

$$Q\dot{\omega} = -Qx$$

which, when substituted, leads to

$$Q\dot{x} = Ax + Q\omega + Bu$$

The last two equations give equations of state. Write them in standard form and multiply the differential equations by ω_0 to recover the time/frequency scaling represented by p and find state matrices for the bandpass filter:

```
Q = Wo/Bw; [ma,m] = size(A);
At = Wo*[A/Q eye(ma,m);-eye(ma,m) zeros(ma,m)];
Bt = Wo*[B/Q; zeros(ma,n)];
Ct = [C zeros(mc,ma)];
Dt = d;
```

If the input to `lp2bp` is in transfer function form, the function transforms it into state-space form before applying this algorithm.

See Also

`bilinear`, `impinvar`, `lp2bs`, `lp2hp`, `lp2lp`

Purpose

Transform lowpass analog filters to bandstop

Syntax

```
[bt,at] = lp2bs(b,a,Wo,Bw)
[At,Bt,Ct,Dt] = lp2bs(A,B,C,D,Wo,Bw)
```

Description

`lp2bs` transforms analog lowpass filter prototypes with a cutoff angular frequency of 1 rad/s into bandstop filters with desired bandwidth and center frequency. The transformation is one step in the digital filter design process for the `butter`, `cheby1`, `cheby2`, and `ellip` functions.

`lp2bs` can perform the transformation on two different linear system representations: transfer function form and state-space form. In both cases, the input system must be an analog filter prototype.

Transfer Function Form (Polynomial)

`[bt,at] = lp2bs(b,a,Wo,Bw)` transforms an analog lowpass filter prototype given by polynomial coefficients into a bandstop filter with center frequency `Wo` and bandwidth `Bw`. Row vectors `b` and `a` specify the coefficients of the numerator and denominator of the prototype in descending powers of *s*.

$$\frac{b(s)}{a(s)} = \frac{b(1)s^n + \cdots + b(n)s + b(n+1)}{a(1)s^m + \cdots + a(m)s + a(m+1)}$$

Scalars `Wo` and `Bw` specify the center frequency and bandwidth in units of radians/second. For a filter with lower band edge `w1` and upper band edge `w2`, use `Wo = sqrt(w1*w2)` and `Bw = w2-w1`.

`lp2bs` returns the frequency transformed filter in row vectors `bt` and `at`.

State-Space Form

`[At,Bt,Ct,Dt] = lp2bs(A,B,C,D,Wo,Bw)` converts the continuous-time state-space lowpass filter prototype in matrices `A`, `B`, `C`, `D` shown below

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

into a bandstop filter with center frequency `Wo` and bandwidth `Bw`. For a filter with lower band edge `w1` and upper band edge `w2`, use `Wo = sqrt(w1*w2)` and `Bw = w2-w1`.

The bandstop filter is returned in matrices `At`, `Bt`, `Ct`, `Dt`.

Algorithm

`lp2bs` is a highly accurate state-space formulation of the classic analog filter frequency transformation. If a bandstop filter is to have center frequency ω_0 and bandwidth B_w, the standard s-domain transformation is

$$s = \frac{p}{Q(p^2 + 1)}$$

where $Q = \omega_0/B_w$ and $p = s/\omega_0$. The state-space version of this transformation is

```
Q = Wo/Bw;
At = [Wo/Q*inv(A) Wo*eye(ma);-Wo*eye(ma) zeros(ma)];
Bt = -[Wo/Q*(A B); zeros(ma,n)];
Ct = [C/A zeros(mc,ma)];
Dt = D - C/A*B;
```

See `lp2bp` for a derivation of the bandpass version of this transformation.

See Also

`bilinear`, `impinvar`, `lp2bp`, `lp2hp`, `lp2lp`

Purpose

Transform lowpass analog filters to highpass

Syntax

```
[bt,at] = lp2hp(b,a,Wo)
[At,Bt,Ct,Dt] = lp2hp(A,B,C,D,Wo)
```

Description

`lp2hp` transforms analog lowpass filter prototypes with a cutoff angular frequency of 1 rad/s into highpass filters with desired cutoff angular frequency. The transformation is one step in the digital filter design process for the `butter`, `cheby1`, `cheby2`, and `ellip` functions.

The `lp2hp` function can perform the transformation on two different linear system representations: transfer function form and state-space form. In both cases, the input system must be an analog filter prototype.

Transfer Function Form (Polynomial)

`[bt,at] = lp2hp(b,a,Wo)` transforms an analog lowpass filter prototype given by polynomial coefficients into a highpass filter with cutoff angular frequency `Wo`. Row vectors `b` and `a` specify the coefficients of the numerator and denominator of the prototype in descending powers of *s*.

$$\frac{b(s)}{a(s)} = \frac{b(1)s^n + \cdots + b(n)s + b(n+1)}{a(1)s^m + \cdots + a(m)s + a(m+1)}$$

Scalar `Wo` specifies the cutoff angular frequency in units of radians/second. The frequency transformed filter is returned in row vectors `bt` and `at`.

State-Space Form

`[At,Bt,Ct,Dt] = lp2hp(A,B,C,D,Wo)` converts the continuous-time state-space lowpass filter prototype in matrices `A`, `B`, `C`, `D` below

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

into a highpass filter with cutoff angular frequency `Wo`. The highpass filter is returned in matrices `At`, `Bt`, `Ct`, `Dt`.

Algorithm

`lp2hp` is a highly accurate state-space formulation of the classic analog filter frequency transformation. If a highpass filter is to have cutoff angular frequency ω_0, the standard s-domain transformation is

$$s = \frac{\omega_0}{p}$$

The state-space version of this transformation is

```
At = Wo*inv(A);
Bt = -Wo*(A\B);
Ct = C/A;
Dt = D - C/A*B;
```

See `lp2bp` for a derivation of the bandpass version of this transformation.

See Also

`bilinear`, `impinvar`, `lp2bp`, `lp2bs`, `lp2lp`

Purpose

Change cutoff frequency for lowpass analog filter

Syntax

```
[bt,at] = lp2lp(b,a,Wo)
[At,Bt,Ct,Dt] = lp2lp(A,B,C,D,Wo)
```

Description

`lp2lp` transforms an analog lowpass filter prototype with a cutoff angular frequency of 1 rad/s into a lowpass filter with any specified cutoff angular frequency. The transformation is one step in the digital filter design process for the `butter`, `cheby1`, `cheby2`, and `ellip` functions.

The `lp2lp` function can perform the transformation on two different linear system representations: transfer function form and state-space form. In both cases, the input system must be an analog filter prototype.

Transfer Function Form (Polynomial)

`[bt,at] = lp2lp(b,a,Wo)` transforms an analog lowpass filter prototype given by polynomial coefficients into a lowpass filter with cutoff angular frequency `Wo`. Row vectors `b` and `a` specify the coefficients of the numerator and denominator of the prototype in descending powers of *s*.

$$\frac{b(s)}{a(s)} = \frac{b(1)s^n + \cdots + b(n)s + b(n+1)}{a(1)s^m + \cdots + a(m)s + a(m+1)}$$

Scalar `Wo` specifies the cutoff angular frequency in units of radians/second. `lp2lp` returns the frequency transformed filter in row vectors `bt` and `at`.

State-Space Form

`[At,Bt,Ct,Dt] = lp2lp(A,B,C,D,Wo)` converts the continuous-time state-space lowpass filter prototype in matrices `A`, `B`, `C`, `D` below

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

into a lowpass filter with cutoff angular frequency `Wo`. `lp2lp` returns the lowpass filter in matrices `At`, `Bt`, `Ct`, `Dt`.

Algorithm

`lp2lp` is a highly accurate state-space formulation of the classic analog filter frequency transformation. If a lowpass filter is to have cutoff angular frequency ω_0, the standard s-domain transformation is

$$s = p/\omega_0$$

The state-space version of this transformation is

```
At = Wo*A;
Bt = Wo*B;
Ct = C;
Dt = D;
```

See `lp2bp` for a derivation of the bandpass version of this transformation.

See Also

`bilinear`, `impinvar`, `lp2bp`, `lp2bs`, `lp2hp`

Purpose

Linear prediction filter coefficients

Syntax

```
[a,g] = lpc(x,p)
```

Description

`lpc` determines the coefficients of a forward linear predictor by minimizing the prediction error in the least squares sense. It has applications in filter design and speech coding.

`[a,g] = lpc(x,p)` finds the coefficients of a `p`th-order linear predictor (FIR filter) that predicts the current value of the real-valued time series `x` based on past samples.

$$\hat{x}(n) = -a(2)x(n-1) - a(3)x(n-2) - \cdots - a(p+1)x(n-p)$$

`p` is the order of the prediction filter polynomial, `a = [1 a(2) ... a(p+1)]`. If `p` is unspecified, `lpc` uses as a default `p = length(x)-1`. If `x` is a matrix containing a separate signal in each column, `lpc` returns a model estimate for each column in the rows of matrix `a` and a column vector of prediction error variances `g`. The length of `p` must be less than or equal to the length of `x`.

Examples

Estimate a data series using a third-order forward predictor, and compare to the original signal.

First, create the signal data as the output of an autoregressive process driven by white noise. Use the last 4096 samples of the AR process output to avoid start-up transients:

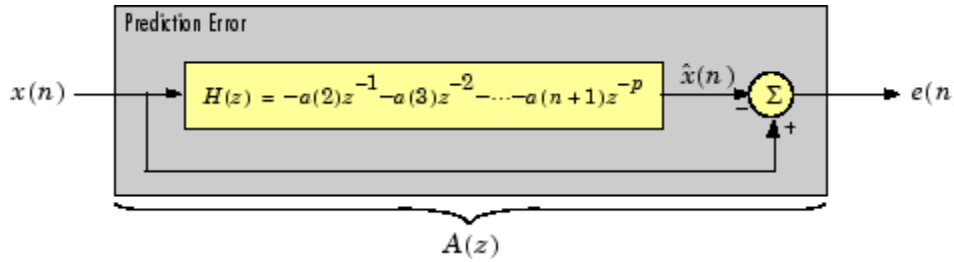
```
randn('state',0);
noise = randn(50000,1);  % Normalized white Gaussian noise
x = filter(1,[1 1/2 1/3 1/4],noise);
x = x(45904:50000);
```

Compute the predictor coefficients, estimated signal, prediction error, and autocorrelation sequence of the prediction error:

```
a = lpc(x,3);
est_x = filter([0 -a(2:end)],1,x);    % Estimated signal
```

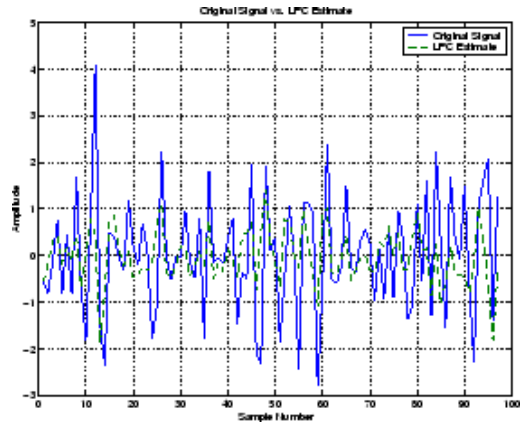
```
e = x - est_x;                          % Prediction error
[acs,lags] = xcorr(e,'coeff');   % ACS of prediction error
```

The prediction error, $e(n)$, can be viewed as the output of the prediction error filter $A(z)$ shown below, where $H(z)$ is the optimal linear predictor, $x(n)$ is the input signal, and $\hat{x}(n)$ is the predicted signal.



Compare the predicted signal to the original signal:

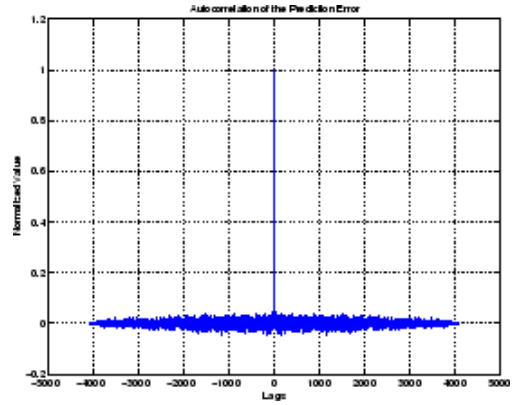
```
plot(1:97,x(4001:4097),1:97,est_x(4001:4097),'--');
title('Original Signal vs. LPC Estimate');
xlabel('Sample Number'); ylabel('Amplitude'); grid;
legend('Original Signal','LPC Estimate')
```



Look at the autocorrelation of the prediction error:

```
plot(lags,acs);
title('Autocorrelation of the Prediction Error');
xlabel('Lags'); ylabel('Normalized Value'); grid;
```

The prediction error is approximately white Gaussian noise, as expected for a third-order AR input process.



Algorithm

`lpc` uses the autocorrelation method of autoregressive (AR) modeling to find the filter coefficients. The generated filter might not model the process exactly even if the data sequence is truly an AR process of the correct order. This is because the autocorrelation method implicitly windows the data, that is, it assumes that signal samples beyond the length of `x` are 0.

`lpc` computes the least squares solution to

$$Xa = b$$

where

$$X = \begin{bmatrix} x(1) & 0 & \cdots & 0 \\ x(2) & x(1) & \ddots & \vdots \\ \vdots & x(2) & \ddots & 0 \\ x(m) & \vdots & \ddots & x(1) \\ 0 & x(m) & \ddots & x(2) \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & x(m) \end{bmatrix}, \quad a = \begin{bmatrix} 1 \\ a(2) \\ \vdots \\ a(p+1) \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

and `m` is the length of `x`. Solving the least squares problem via the normal equations

$$X^H X a = X^H b$$

leads to the Yule-Walker equations

$$\begin{bmatrix} r(1) & r(2)^* & \cdots & r(p)^* \\ r(2) & r(1) & \ddots & \vdots \\ \vdots & \ddots & \ddots & r(2)^* \\ r(p) & \cdots & r(2) & r(1) \end{bmatrix} \begin{bmatrix} a(2) \\ a(3) \\ \vdots \\ a(p+1) \end{bmatrix} = \begin{bmatrix} -r(2) \\ -r(3) \\ \vdots \\ -r(p+1) \end{bmatrix}$$

where r = `[`$r(1)$ $r(2)$ ... $r(p+1)$`]` is an autocorrelation estimate for `x` computed using `xcorr`. The Yule-Walker equations are solved in $O(p^2)$ flops by the Levinson-Durbin algorithm (see `levinson`).

References

[1] Jackson, L.B., *Digital Filters and Signal Processing, Second Edition*, Kluwer Academic Publishers, 1989. pp. 255-257.

See Also

`aryule`, `levinson`, `prony`, `pyulear`, `stmcb`

Purpose

Convert line spectral frequencies to prediction filter coefficients

Syntax

```
a = lsf2poly(lsf)
```

Description

`a = lsf2poly(lsf)` returns a vector `a` containing the prediction filter coefficients from the vector `lsf` of line spectral frequencies. If `lsf` is a matrix of size MxN with separate channels of line spectral frequencies in each column, the returned `a` matrix has the resulting prediction filter coefficients in rows and is of size Nx(M+1).

Examples

```
lsf = [0.7842    1.5605    1.8776    1.8984    2.3593];
a = lsf2poly(lsf)
a =
    1.0000    0.6148    0.9899    0.0001    0.0031   -0.0081
```

References

[1] Deller, J.R., J.G. Proakis, and J.H.L. Hansen, *Discrete-Time Processing of Speech Signals*, Prentice-Hall, 1993.

[2] Rabiner, L.R., and R.W. Schafer, *Digital Processing of Speech Signals*, Prentice-Hall, 1978.

See Also

`ac2poly`, `poly2lsf`, `rc2poly`

mag2db

Purpose

Convert magnitude to decibels (dB)

Syntax

```
ydb = mag2db(y)
```

Description

`ydb = mag2db(y)` returns the corresponding decibel (dB) value `ydb` for a given magnitude `y`. The relationship between magnitude and decibels is `ydb` = $20*\log_{10}$(`y`).

See Also

`db2mag`

Purpose

Generalized Marcum Q function

Syntax

```
Q = marcumq(a,b)
Q = marcumq(a,b,m)
```

Description

`Q = marcumq(a,b)` computes the Marcum Q function of `a` and `b`, defined by

$$Q(a,b) = \int_b^\infty x \exp\left(-\frac{x^2+a^2}{2}\right) I_0(ax)dx$$

where `a` and `b` are nonnegative real numbers. In this expression, I_0 is the modified Bessel function of the first kind of zero order.

`Q = marcumq(a,b,m)` computes the generalized Marcum Q, defined by

$$Q\ (a,b) = \frac{1}{a^{m-1}} \int_b^\infty x^m \exp\left(-\frac{x^2+a^2}{2}\right) I_{m-1}(ax)dx$$

where `a` and `b` are nonnegative real numbers, and `m` is a positive integer. In this expression, I_{m-1} is the modified Bessel function of the first kind of order m-1.

If any of the inputs is a scalar, it is expanded to the size of the other inputs.

See Also

`besseli`

References

[1] Cantrell, P. E., and A. K. Ojha, "Comparison of Generalized Q-Function Algorithms," *IEEE Transactions on Information Theory*, Vol. IT-33, July, 1987, pp. 591–596.

[2] Marcum, J. I., "A Statistical Theory of Target Detection by Pulsed Radar: Mathematical Appendix," RAND Corporation, Santa Monica,

CA, Research Memorandum RM-753, July 1, 1948. Reprinted in *IRE Transactions on Information Theory*, Vol. IT-6, April, 1960, pp. 59–267.

[3] Shnidman, D. A., "The Calculation of the Probability of Detection and the Generalized Marcum Q-Function," *IEEE Transactions on Information Theory*, Vol. IT-35, March, 1989, pp. 389–400.

Purpose

Generalized digital Butterworth filter design

Syntax

```
[b,a] = maxflat(n,m,Wn)
b = maxflat(n,'sym',Wn)
[b,a,b1,b2] = maxflat(n,m,Wn)
[b,a,b1,b2,sos,g] = maxflat(n,m,Wn)
[...] = maxflat(n,m,Wn,'design_flag')
```

Description

`[b,a] = maxflat(n,m,Wn)` is a lowpass Butterworth filter with numerator and denominator coefficients `b` and `a` of orders `n` and `m` respectively. `Wn` is the normalized cutoff frequency at which the magnitude response of the filter is equal to $1/\sqrt{2}$ (approx. -3 dB). `Wn` must be between 0 and 1, where 1 corresponds to the Nyquist frequency.

`b = maxflat(n,'sym',Wn)` is a symmetric FIR Butterworth filter. `n` must be even, and `Wn` is restricted to a subinterval of [0,1]. The function raises an error if `Wn` is specified outside of this subinterval.

`[b,a,b1,b2] = maxflat(n,m,Wn)` returns two polynomials `b1` and `b2` whose product is equal to the numerator polynomial `b` (that is, `b = conv(b1,b2)`). `b1` contains all the zeros at `z = -1`, and `b2` contains all the other zeros.

`[b,a,b1,b2,sos,g] = maxflat(n,m,Wn)` returns the second-order sections representation of the filter as the filter matrix `sos` and the gain `g`.

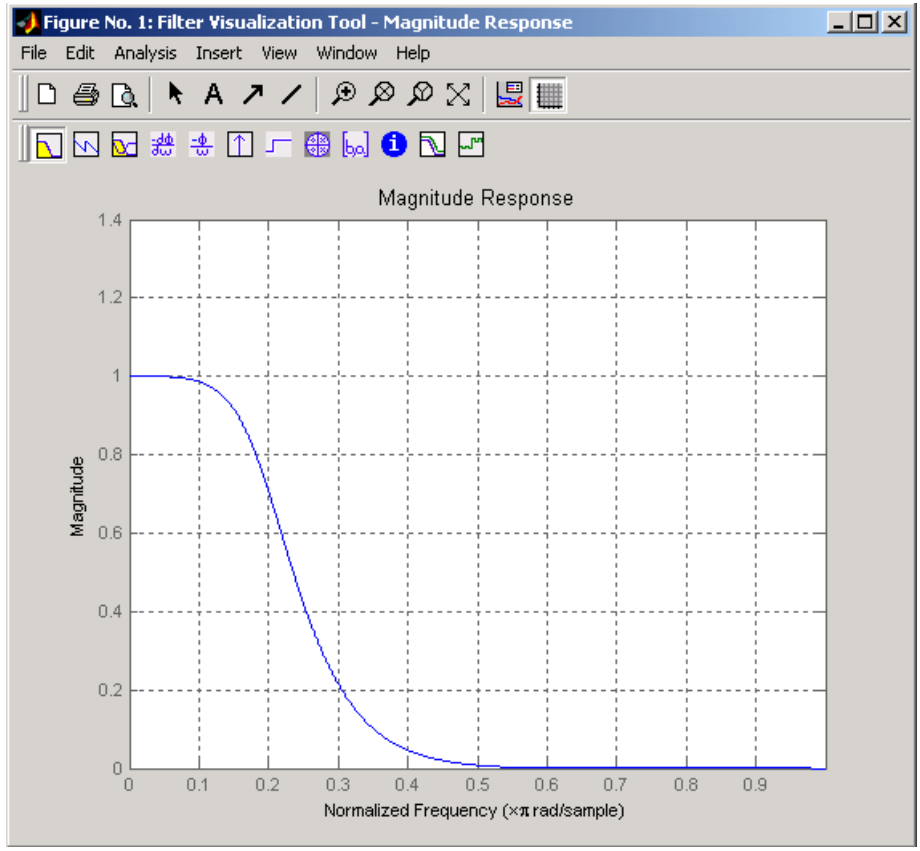
`[...] = maxflat(n,m,Wn,'design_flag')` enables you to monitor the filter design, where *`'design_flag'`* is

- `'trace'` for a textual display of the design table used in the design
- `'plots'` for plots of the filter's magnitude, group delay, and zeros and poles
- `'both'` for both the textual display and plots

Examples

```
n = 10; m = 2; Wn = 0.2;
[b,a] = maxflat(n,m,Wn)
```

```
fvtool(b,a)                    % Display the magnitude plot
```



Algorithm

The method consists of the use of formulae, polynomial root finding, and a transformation of polynomial roots.

References

[1] Selesnick, I.W., and C.S. Burrus, "Generalized Digital Butterworth Filter Design," *Proceedings of the IEEE Int. Conf. Acoust., Speech, Signal Processing, Vol. 3* (May 1996).

See Also

butter, filter, freqz

medfilt1

Purpose

1-D median filtering

Syntax

```
y = medfilt1(x,n)
y = medfilt1(x,n,blksz)
y = medfilt1(x,n,blksz,dim)
```

Description

`y = medfilt1(x,n)` applies an order `n` one-dimensional median filter to vector `x`; the function considers the signal to be 0 beyond the end points. Output `y` has the same length as `x`.

For n odd, `y(k)` is the median of `x(k-(n-1)/2:k+(n-1)/2)`.

For `n` even, `y(k)` is the median of `x(k-n/2)`, `x(k-(n/2)+1)`, ..., `x(k+(n/2)-1)`. In this case, `medfilt1` sorts the numbers, then takes the average of the `n/2` and `(n/2)+1` elements.

The default for n is 3.

`y = medfilt1(x,n,blksz)` uses a `for`-loop to compute `blksz` (block size) output samples at a time. Use `blksz` << `length(x)` if you are low on memory, since `medfilt1` uses a working matrix of size n-by-`blksz`. By default, `blksz = length(x)`; this provides the fastest execution if you have sufficient memory.

If `x` is a matrix, `medfilt1` median filters its columns using

```
y(:,i) = medfilt1(x(:,i),n,blksz)
```

in a loop over the columns of `x`.

`y = medfilt1(x,n,blksz,dim)` specifies the dimension, `dim`, along which the filter operates.

References

[1] Pratt, W.K., *Digital Image Processing*, John Wiley & Sons, 1978, pp. 330-333.

See Also

`filter`, `medfilt2`, `median`

Purpose

Modulation for communications simulation

Syntax

```
y = modulate(x,fc,fs,'method')
y = modulate(x,fc,fs,'method',opt)
[y,t] = modulate(x,fc,fs)
```

Description

`y = modulate(x,fc,fs,'method')` and

`y = modulate(x,fc,fs,'method',opt)` modulate the real message signal `x` with a carrier frequency `fc` and sampling frequency `fs`, using one of the options listed below for *`'method'`*. Note that some methods accept an option, `opt`.

Method	Description

`amdsb-sc`
or
`am`
	Amplitude modulation, double sideband, suppressed carrier. Multiplies `x` by a sinusoid of frequency `fc`.
	`y = x.*cos(2*pi*fc*t)`

`amdsb-tc`	Amplitude modulation, double sideband, transmitted carrier. Subtracts scalar `opt` from `x` and multiplies the result by a sinusoid of frequency `fc`. `y = (x-opt).*cos(2*pi*fc*t)` If the `opt` parameter is not present, `modulate` uses a default of `min(min(x))` so that the message signal `(x-opt)` is entirely nonnegative and has a minimum value of 0.

Method	Description
amssb	Amplitude modulation, single sideband. Multiplies x by a sinusoid of frequency fc and adds the result to the Hilbert transform of x multiplied by a phase shifted sinusoid of frequency fc. `y = x.*cos(2*pi*fc*t)+imag(hilbert(x)).*sin(2*pi*fc*t)` This effectively removes the upper sideband.

fm	Frequency modulation. Creates a sinusoid with instantaneous frequency that varies with the message signal x.
	
	`y = cos(2*pi*fc*t + opt*cumsum(x))`
	
	cumsum is a rectangular approximation to the integral of x. modulate uses opt as the constant of frequency modulation. If opt is not present, modulate uses a default of
	
	`opt = (fc/fs)*2*pi/(max(max(x)))`
	
	so the maximum frequency excursion from fc is fc Hz.

pm	Phase modulation. Creates a sinusoid of frequency fc whose phase varies with the message signal x.
	
	`y = cos(2*pi*fc*t + opt*x)`
	
	modulate uses opt as the constant of phase modulation. If opt is not present, modulate uses a default of
	
	`opt = pi/(max(max(x)))`
	
	so the maximum phase excursion is π radians.

Method	Description
`pwm`	Pulse-width modulation. Creates a pulse-width modulated signal from the pulse widths in `x`. The elements of `x` must be between 0 and 1, specifying the width of each pulse in fractions of a period. The pulses start at the beginning of each period, that is, they are left justified. `modulate(x,fc,fs,'pwm','centered')` yields pulses centered at the beginning of each period. `y` is length `length(x)*fs/fc`.
`ppm`	Pulse-position modulation. Creates a pulse-position modulated signal from the pulse positions in `x`. The elements of `x` must be between 0 and 1, specifying the left edge of each pulse in fractions of a period. `opt` is a scalar between 0 and 1 that specifies the length of each pulse in fractions of a period. The default for `opt` is `0.1`. `y` is length `length(x)*fs/fc`.
`qam`	Quadrature amplitude modulation. Creates a quadrature amplitude modulated signal from signals `x` and `opt`. `y = x.*cos(2*pi*fc*t) + opt.*sin(2*pi*fc*t)` `opt` must be the same size as `x`.

If you do not specify *`'method'`*, then `modulate` assumes `am`. Except for the `pwm` and `ptm` cases, `y` is the same size as `x`.

If `x` is an array, `modulate` modulates its columns.

`[y,t] = modulate(x,fc,fs)` returns the internal time vector `t` that `modulate` uses in its computations.

See Also

`demod`, `vco`, `fskdemod`, `genqamdemod`, `mskdemod`, `pamdemod`, `pmdemod`, `qamdemod`

Purpose

Magnitude squared coherence

Syntax

```
Cxy = mscohere(x,y)
Cxy = mscohere(x,y,window)
Cxy = mscohere(x,y,window,noverlap)
[Pxy,W] = mscohere(x,y,window,noverlap,nfft)
[Cxy,F] = mscohere(x,y,window,noverlap,nfft,fs)
[...] = mscohere(x,y,...,'whole')
mscohere(...)
```

Description

`Cxy = mscohere(x,y)` finds the magnitude squared coherence estimate Cxy of the input signals `x` and `y` using Welch's averaged, modified periodogram method. The magnitude squared coherence estimate is a function of frequency with values between 0 and 1 that indicates how well `x` corresponds to `y` at each frequency. The coherence is a function of the power spectral density (*Pxx* and *Pyy*) of `x` and `y` and the cross power spectral density (*Pxy*) of `x` and `y`.

$$C_{xy}(f) = \frac{|P_{xy}(f)|^2}{P_{xx}(f)P_{yy}(f}$$

`x` and `y` must be the same length. For real `x` and `y`, `mscohere` returns a one-sided coherence estimate and for complex `x` or `y`, it returns a two-sided estimate.

`mscohere` uses the following default values:

Parameter	Description	Default Value
`nfft`	FFT length which determines the frequencies at which the coherence is estimated For real `x` and `y`, the length of `Cxy` is (`nfft`/2+1) if `nfft` is even or (`nfft`+1)/2 if `nfft` is odd. For complex `x` or `y`, the length of `Cxy` is `nfft`. If `nfft` is greater than the signal length, the data is zero-padded. If `nfft` is less than the signal length, the segment is wrapped using datawrap so that the length is equal to `nfft`.	Maximum of 256 or the next power of 2 greater than the length of each section of `x` or `y`
`fs`	Sampling frequency	1
`window`	Windowing function and number of samples to use for each section	Periodic Hamming window of length to obtain eight equal sections of x and `y`
`noverlap`	Number of samples by which the sections overlap	Value to obtain 50% overlap

Note You can use the empty matrix `[]` to specify the default value for any input argument except `x` or `y`. For example, `Pxy = mschoere(x,y,[],[],128)` uses a Hamming window, default `noverlap` to obtain 50% overlap, and the specified 128 `nfft`.

`Cxy = mscohere(x,y,window)` specifies a windowing function, divides `x` and `y` into equal overlapping sections of the specified window length, and windows each section using the specified window function. If you supply a scalar for `window`, `Cxy` uses a Hamming window of that length. `mscohere` zero pads the sections if the window length exceeds `nfft`.

`Cxy = mscohere(x,y,window,noverlap)` overlaps the sections of `x` by `noverlap` samples. `noverlap` must be an integer smaller than the length of `window`.

`[Pxy,W] = mscohere(x,y,window,noverlap,nfft)` uses the specified FFT length `nfft` to calculate the coherence estimate. It also returns `W`, which is the vector of normalized frequencies (in rad/sample) at which the coherence is estimated. For real `x` and `y`, `Cxy` length is (`nfft`/2 +1) if `nfft` is even and if `nfft` is odd, the length is (`nfft`+1)/2. For complex `x` or `y`, the length of `Cxy` is `nfft`. For real signals, the range of `W` is [0, pi] when `nfft` is even and [0, pi) when `nfft` is odd. For complex signals, the range of `W` is [0, 2*pi).

`[Cxy,F] = mscohere(x,y,window,noverlap,nfft,fs)` returns `Cxy` as a function of frequency and a vector `F` of frequencies at which the coherence is estimated. `fs` is the sampling frequency in Hz. For real signals, the range of `F` is [0, `fs`/2] when `nfft` is even and [0, `fs`/2) when `nfft` is odd. For complex signals, the range of `F` is [0, `fs`).

`[...] = mscohere(x,y,...,'whole')` returns a coherence estimate with frequencies that range over the whole Nyquist interval. Specifying `'half'` uses half the Nyquist interval.

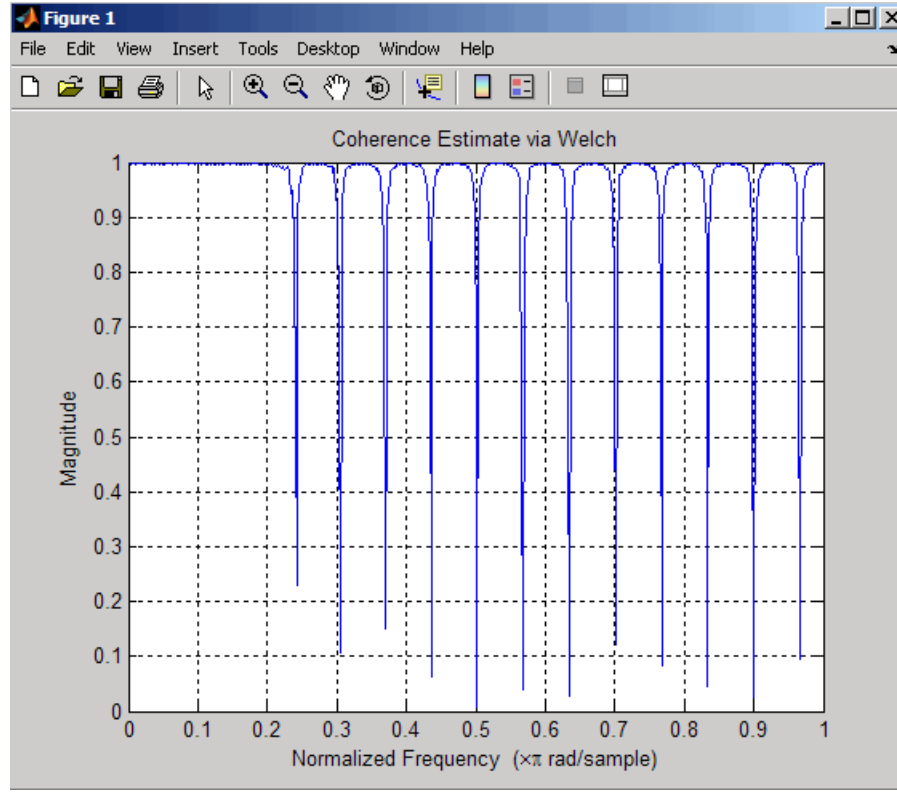
`mscohere(...)` plots the magnitude squared coherence versus frequency in the current figure window.

Note If you use `mscohere` on two linearly related signals [1]with a single, non-overlapping window, the output for all frequencies is `Cxy` = 1.

Examples

Compute and plot the coherence estimate between two colored noise sequences `x` and `y`:

```
randn('state',0);
h = fir1(30,0.2,rectwin(31));
h1 = ones(1,10)/sqrt(10);
r = randn(16384,1);
x = filter(h1,1,r);
y = filter(h,1,x);
mscohere(x,y,hanning(1024),512,1024)
```



Algorithm

`mscohere` estimates the magnitude squared coherence function [2] using Welch's averaged periodogram method (see references [3] and [4]).

References

[1] Stoica, P., and R. Moses. *Introduction to Spectral Analysis.* Upper Saddle River, NJ: Prentice-Hall, 1997. Pgs. 61-64.

[2] Kay, S.M. *Modern Spectral Estimation.* Englewood Cliffs, NJ: Prentice-Hall, 1988. Pg. 454.

[3] Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing.* Englewood Cliffs, NJ: Prentice-Hall, 1975.

[4] Welch, P.D. "The Use of Fast Fourier Transform for the Estimation of Power Spectra: A Method Based on Time Averaging Over Short, Modified Periodograms." *IEEE Trans. Audio Electroacoust. Vol. AU-15 (June 1967).* Pgs. 70-73.

See Also

`cpsd`, `periodogram`, `pwelch`, `spectrum`, `tfestimate`

Purpose

Nuttall-defined minimum 4-term Blackman-Harris window

Syntax

```
w = nuttallwin(L)
```

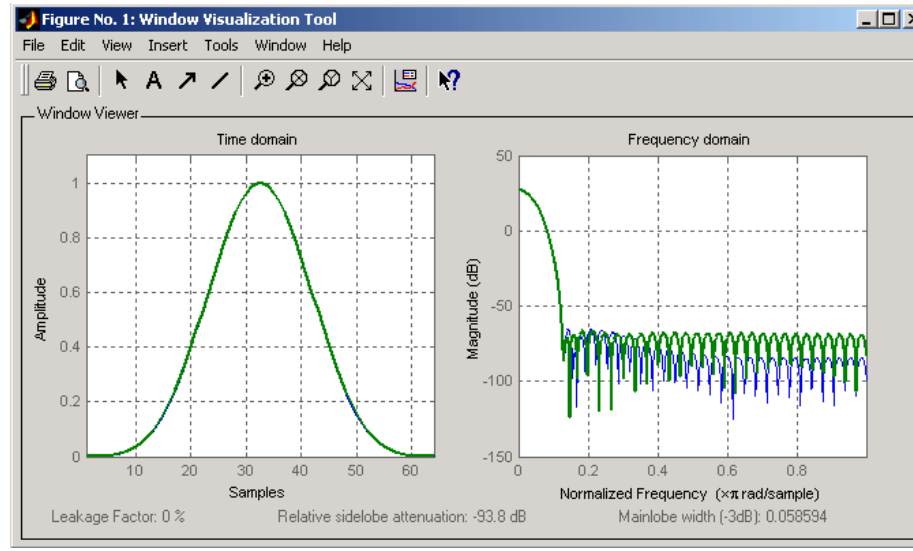
Description

`w = nuttallwin(L)` returns a minimum, `L`-point, 4-term Blackman-Harris window in the column vector `w`. The window is minimum in the sense that its maximum sidelobes are minimized. The coefficients for this window differ from the Blackman-Harris window coefficients computed with `blackmanharris` and produce slightly lower sidelobes.

Examples

Compare 64-point Blackman-Harris and Nuttall's Blackman-Harris windows and plot them using WVTool:

```
L = 64;
w = blackmanharris(L);
y = nuttallwin(L);
wvtool(w,y)
```



The maximum difference between the two windows is

```
max(abs(y-w))

ans =

    0.0099
```

Algorithm

The equation for computing the coefficients of a minimum 4-term Blackman-Harris window, according to Nuttall, is

$$w(n) = a_0 - a_1 \cos\left(2\pi\frac{n}{N}\right) + a_2 \cos\left(4\pi\frac{n}{N}\right) - a_3 \cos\left(6\pi\frac{n}{N}\right)$$

where $0 \le n \le N$ and the window length is $L = N + 1$.

The coefficients for this window are

$a_0 = 0.3635819$

$a_1 = 0.4891775$

$a_2 = 0.1365995$

$a_3 = .0106411$

References

[1] Nuttall, Albert H. “Some Windows with Very Good Sidelobe Behavior.” *IEEE Transactions on Acoustics, Speech, and Signal Processing*. Vol. ASSP-29 (February 1981). pp. 84-91.

See Also

`barthannwin`, `bartlett`, `blackmanharris`, `bohmanwin`, `parzenwin`, `rectwin`, `triang`, `window`, `wintool`, `wvtool`

Purpose

Parzen (de la Valle-Poussin) window

Syntax

```
w = parzenwin(L)
```

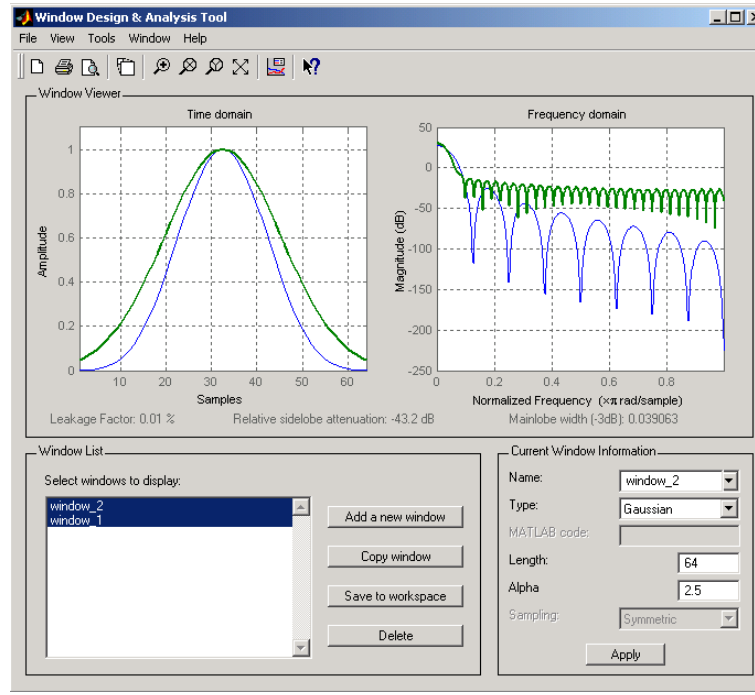
Description

`w = parzenwin(L)` returns the `L`-point Parzen (de la Valle-Poussin) window in column vector `w`. Parzen windows are piecewise cubic approximations of Gaussian windows. Parzen window sidelobes fall off as $1/\omega^4$.

Examples

Compare 64-point Parzen and Gaussian windows and display the result using `sigwin` window objects and `wintool`:

```
wintool(sigwin.parzenwin(64),sigwin.gausswin(64))
```



Algorithm

The Parzen window is defined as

$$w(n) = \begin{cases} 1.0 - 6\left(\frac{n}{N/2}\right)^2\left(1.0 - \frac{|n|}{N/2}\right), & 0 \le |n| \le \frac{N}{4} \\ 2\left(1.0 - \frac{|n|}{N/2}\right)^3, & \frac{N}{4} \le |n| \le \frac{N}{2} \end{cases}$$

The window length is $L = N + 1$.

References

[1] Harris, F.J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform." *Proceedings of the IEEE.* Vol. 66, No. 1 (January 1978).

See Also

`barthannwin`, `bartlett`, `blackmanharris`, `bohmanwin`, `nuttallwin`, `rectwin`, `triang`, `window`, `wintool`, `wvtool`

Purpose

PSD using Burg method

Syntax

```
Pxx = pburg(x,p)
Pxx = pburg(x,p,nfft)
[Pxx,w] = pburg(...)
[Pxx,w] = pburg(x,p,w)
Pxx = pburg(x,p,nfft,fs)
Pxx = pburg(x,p,f,fs)
[Pxx,f] = pburg(x,p,nfft,fs)
[Pxx,f] = pburg(x,p,f,fs)
[Pxx,f] = pburg(x,p,nfft,fs,'range')
[Pxx,w] = pburg(x,p,nfft,'range')
pburg(...)
```

Description

`Pxx = pburg(x,p)` implements the Burg algorithm, a parametric spectral estimation method, and returns `Pxx`, an estimate of the power spectral density (PSD) of the vector `x`. The entries of `x` represent samples of a discrete-time signal, and `p` is the integer specifying the order of an autoregressive (AR) prediction model for the signal, used in estimating the PSD.

The power spectral density is calculated in units of power per radians per sample. Real-valued inputs produce full power one-sided (in frequency) PSDs (by default), while complex-valued inputs produce two-sided PSDs.

In general, the length of the FFT and the values of the input `x` determine the length of `Pxx` and the range of the corresponding normalized frequencies. For this syntax, the (default) FFT length is 256. The following table indicates the length of `Pxx` and the range of the corresponding normalized frequencies for this syntax.

PSD Vector Characteristics for an FFT Length of 256 (Default)

Real/Complex Input Data	Length of Pxx	Range of the Corresponding Normalized Frequencies
Real-valued	129	[0, π]
Complex-valued	256	[0, 2π)

`Pxx = pburg(x,p,nfft)` uses the integer FFT length `nfft` to calculate the PSD vector `Pxx`.

`[Pxx,w] = pburg(...)` also returns `w`, a vector of normalized angular frequencies at which the two-sided PSD is estimated. `Pxx` and `w` have the same length. The units for `w` are rad/sample.

The length of `Pxx` and the frequency range for `w` depend on `nfft` and the values of the input `x`. The following table indicates the length of `Pxx` and the frequency range for `w` in this syntax.

PSD and Frequency Vector Characteristics

Real/Complex Input Data	nfft Even/Odd	Length of Pxx	Range of w
Real-valued	Even	`(nfft/2 + 1)`	[0, π]
Real-valued	Odd	`(nfft + 1)/2`	[0, π)
Complex-valued	Even or odd	`nfft`	[0, 2π)

`[Pxx,w] = pburg(x,p,w)` uses a vector of normalized frequencies `w` with two or more elements to compute the PSD at those frequencies and returns a two-sided PSD.

```
Pxx = pburg(x,p,nfft,fs)
```

or

`Pxx = pburg(x,p,f,fs)` uses the integer FFT length `nfft` to calculate the PSD vector `Pxx` or uses the vector of frequencies `f` in Hz and the sampling frequency `fs` to compute the two-sided PSD vector `Pxx` at those frequencies. If you specify `nfft` as the empty vector `[]`, it uses the default value of 256. If you specify `fs` as the empty vector `[]`, the sampling frequency `fs` defaults to 1 Hz. The spectral density produced is calculated in units of power per Hz.

`[Pxx,f] = pburg(x,p,nfft,fs)`

or

`[Pxx,f] = pburg(x,p,f,fs)` returns the frequency vector f. In this case, the units for the frequency vector are in Hz. The frequency range for `f` depends on `nfft`, `fs`, and the values of the input `x`. The length of `Pxx` is the same as in the table above. The following table indicates the frequency range for `f` for this syntax.

PSD and Frequency Vector Characteristics with fs Specified

Real/Complex Input Data	nfft Even/Odd	Range of f
Real-valued	Even	`[0,fs/2]`
Real-valued	Odd	`[0,fs/2)`
Complex-valued	Even or odd	`[0,fs)`

`[Pxx,f] = pburg(x,p,nfft,fs,'range')` or

`[Pxx,w] = pburg(x,p,nfft,'range')` specifies the range of frequency values to include in `f` or `w`. This syntax is useful when `x` is real. `'range'` can be either:

- `'twosided'`: Compute the two-sided PSD over the frequency range `[0,fs)`. This is the default for determining the frequency range for complex-valued `x`.
  - If you specify `fs` as the empty vector, `[]`, the frequency range is `[0,1)`.

  - If you don't specify `fs`, the frequency range is [0, 2π).
- `'onesided'`: Compute the one-sided PSD over the frequency ranges specified for real `x`. This is the default for determining the frequency range for real-valued `x`. Note that `'onesided'` is not valid if you pass in a vector of frequencies (`f` or `w`).

Note You can put the string argument *`'range'`* anywhere in the input argument list after `p`.

`pburg(...)` with no outputs plots the PSD in the current figure window. The frequency range on the plot is the same as the range of output `w` (or `f`) for a given set of parameters.

Remarks

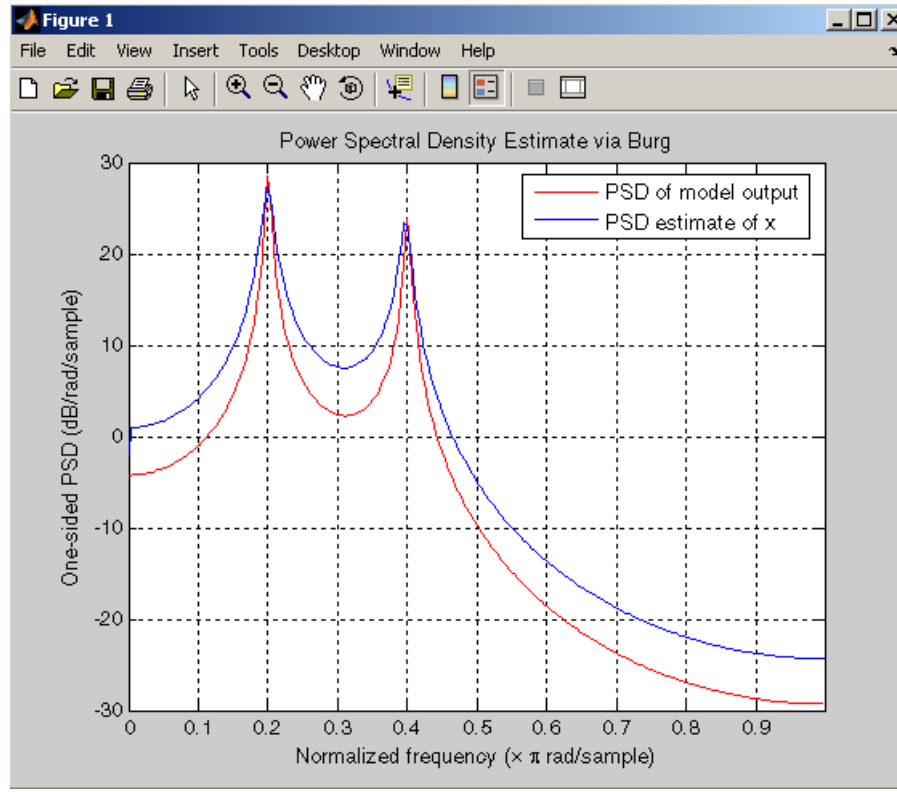
The power spectral density is computed as the distribution of power per unit frequency. This algorithm depends on your selecting an appropriate model order for your signal.

Examples

The Burg method estimates the spectral density by fitting an AR prediction model of a given order to the signal, so first generate a signal from an AR (all-pole) model of a given order. Use `freqz` to check the magnitude of the frequency response of your AR filter. Then, generate the input signal `x` by filtering white noise through the AR filter. Estimate the PSD of `x` based on a fourth-order AR prediction model because in this case we know that the original AR system model `a` has order 4:

```
% Define AR filter coefficients
a = [1 -2.2137 2.9403 -2.1697 0.9606];

[H,w] = freqz(1,a,256);     % AR filter freq response

% Scale to make one-sided PSD
Hp = plot(w/pi,20*log10(2*abs(H)/(2*pi)),'r');
hold on;
randn('state',1);
```

```
x = filter(1,a,randn(256,1));           % AR system output
pburg(x,4,511);
xlabel('Normalized frequency (\times \pi rad/sample)')
ylabel('One-sided PSD (dB/rad/sample)')
legend('PSD of model output','PSD estimate of x')
```



Algorithm

You can use linear prediction filters to model the second-order statistical characteristics of a signal. The prediction filter output can be used to model the signal when the input is white noise.

The Burg method fits an AR linear prediction filter model of the specified order to the input signal by minimizing (using least squares)

the arithmetic mean of the forward and backward prediction errors. The spectral density then is computed from the frequency response of the prediction filter. The AR filter parameters are constrained to satisfy the Levinson-Durbin recursion.

References

[1] Marple, S.L. *Digital Spectral Analysis*, Englewood Cliffs, NJ, Prentice-Hall, 1987, Chapter 7.

[2] Stoica, P., and R.L. Moses, *Introduction to Spectral Analysis*, Prentice-Hall, 1997.

See Also

`arburg`, `lpc`, `pcov`, `peig`, `periodogram`, `pmcov`, `pmtm`, `pmusic`, `pwelch`, `pyulear`

Purpose

PSD using covariance method

Syntax

```
Pxx = pcov(x,p)
Pxx = pcov(x,p,nfft)
[Pxx,w] = pcov(...)
[Pxx,w] = pcov(x,p,w)
Pxx = pcov(x,p,nfft,fs)
Pxx = pcov(x,p,f,fs)
[Pxx,f] = pcov(x,p,nfft,fs)
[Pxx,f] = pcov(x,p,f,fs)
[Pxx,f] = pcov(x,p,nfft,fs,'range')
[Pxx,w] = pcov(x,p,nfft,'range')
pcov(...)
```

Description

`Pxx = pcov(x,p)` implements the covariance algorithm, a parametric spectral estimation method, and returns `Pxx`, an estimate of the power spectral density (PSD) of the vector `x`. The entries of `x` represent samples of a discrete-time signal, and where `p` is the integer specifying the order of an autoregressive (AR) prediction model for the signal, used in estimating the PSD.

The power spectral density is calculated in units of power per radians per sample. Real-valued inputs produce full power one-sided (in frequency) PSDs (by default), while complex-valued inputs produce two-sided PSDs.

In general, the length of the FFT and the values of the input `x` determine the length of `Pxx` and the range of the corresponding normalized frequencies. For this syntax, the (default) FFT length is 256. The following table indicates the length of `Pxx` and the range of the corresponding normalized frequencies for this syntax.

PSD Vector Characteristics for an FFT Length of 256 (Default)

Real/Complex Input Data	Length of Pxx	Range of the Corresponding Normalized Frequencies
Real-valued	129	[0, π]
Complex-valued	256	[0, 2π)

`Pxx = pcov(x,p,nfft)` uses the integer FFT length `nfft` to calculate the PSD vector `Pxx`.

`[Pxx,w] = pcov(...)` also returns `w`, a vector of normalized angular frequencies at which the two-sided PSD is estimated. `Pxx` and `w` have the same length. The units for `w` are rad/sample.

The length of `Pxx` and the frequency range for `w` depend on `nfft` and the values of the input `x`. The following table indicates the length of `Pxx` and the frequency range for `w` in this syntax.

PSD and Frequency Vector Characteristics

Real/Complex Input Data	nfft Even/Odd	Length of Pxx	Range of w
Real-valued	Even	`(nfft/2 + 1)`	[0, π]
Real-valued	Odd	`(nfft + 1)/2`	[0, π)
Complex-valued	Even or odd	`nfft`	[0, 2π)

`[Pxx,w] = pcov(x,p,w)` uses a vector of normalized frequencies `w` with two or more elements to compute the PSD at those frequencies and returns a two-sided PSD.

```
Pxx = pcov(x,p,nfft,fs)
```

or

`Pxx = pcov(x,p,f,fs)` uses the integer FFT length `nfft` to calculate the PSD vector `Pxx` or uses the vector of frequencies `f` in Hz and the sampling frequency `fs` to compute the two-sided PSD vector `Pxx` at those frequencies. If you specify `nfft` as the empty vector `[]`, it uses the default value of 256. If you specify `fs` as the empty vector `[]`, the sampling frequency `fs` defaults to 1 Hz. The spectral density produced is calculated in units of power per Hz.

`[Pxx,f] = pcov(x,p,nfft,fs)`

or

`[Pxx,f] = pcov(x,p,f,fs)` returns the frequency vector f. In this case, the units for the frequency vector are in Hz. The frequency range for `f` depends on `nfft`, `fs`, and the values of the input `x`. The length of `Pxx` is the same as in the table above. The following table indicates the frequency range for `f` for this syntax.

PSD and Frequency Vector Characteristics with fs Specified

Real/Complex Input Data	nfft Even/Odd	Range of f
Real-valued	Even	`[0,fs/2]`
Real-valued	Odd	`[0,fs/2)`
Complex-valued	Even or odd	`[0,fs)`

`[Pxx,f] = pcov(x,p,nfft,fs,'range')` or

`[Pxx,w] = pcov(x,p,nfft,'range')` specifies the range of frequency values to include in `f` or `w`. This syntax is useful when `x` is real. `'range'` can be either:

- `'twosided'`: Compute the two-sided PSD over the frequency range `[0,fs)`. This is the default for determining the frequency range for complex-valued `x`.
  - If you specify `fs` as the empty vector, `[]`, the frequency range is `[0,1)`.

  - If you don't specify `fs`, the frequency range is [0, 2π).
- `'onesided'`: Compute the one-sided PSD over the frequency ranges specified for real `x`. This is the default for determining the frequency range for real-valued `x`. Note that `'onesided'` is not valid if you pass in a vector of frequencies (`f` or `w`).

Note You can put the string argument *`'range'`* anywhere in the input argument list after `p`.

`pcov(...)` with no outputs plots the power spectral density in the current figure window. The frequency range on the plot is the same as the range of output `w` (or `f`) for a given set of parameters.

Remarks

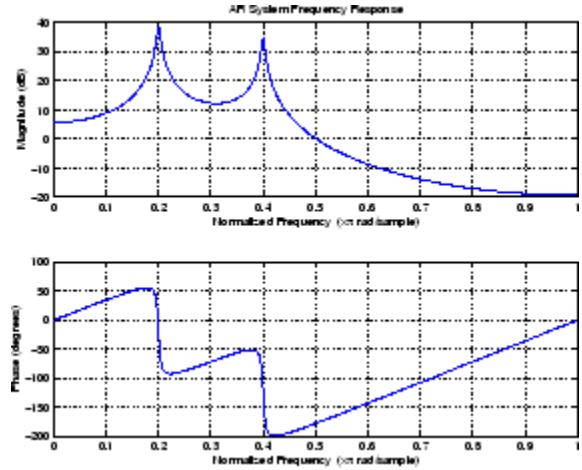
The power spectral density is computed as the distribution of power per unit frequency.

This algorithm depends on your selecting an appropriate model order for your signal.

Examples

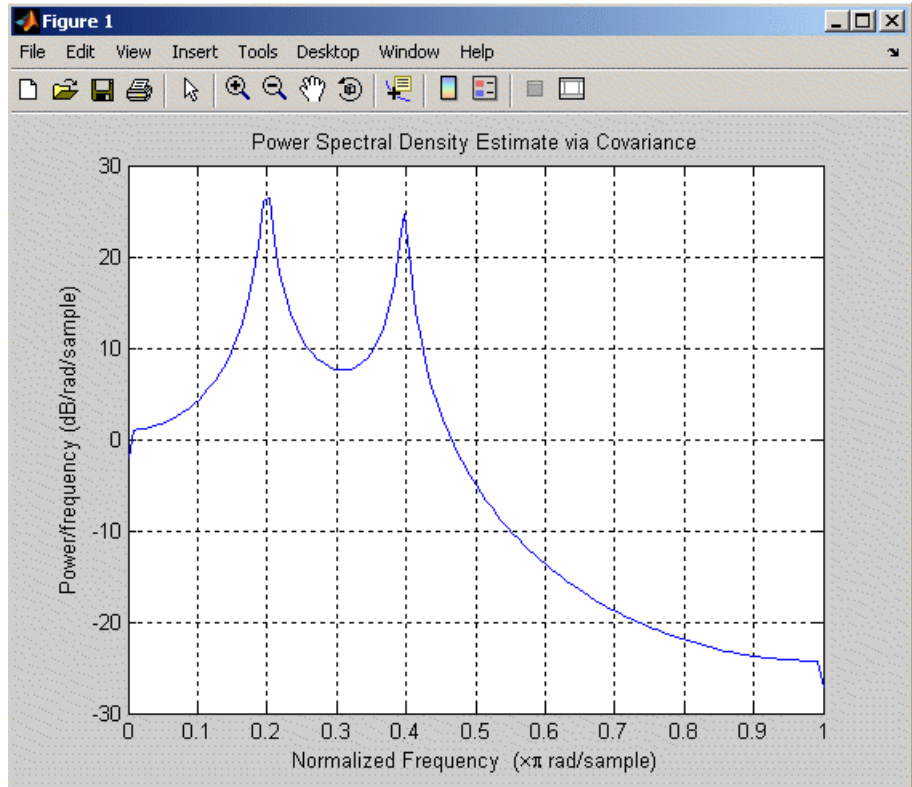
Because the covariance method estimates the spectral density by fitting an AR prediction model of a given order to the signal, first generate a signal from an AR (all-pole) model of a given order. You can use `freqz` to check the magnitude of the frequency response of your AR filter. This will give you an idea of what to expect when you estimate the PSD using `pcov`:

```
a = [1 -2.2137 2.9403 -2.1697 0.9606];    % AR filter coefficients
freqz(1,a)                                % AR filter frequency response
title('AR System Frequency Response')
```



Now generate the input signal `x` by filtering white noise through the AR filter. Estimate the PSD of `x` based on a fourth-order AR prediction model since in this case we know that the original AR system model `a` has order 4:

```
randn('state',1);

% Signal generated from AR filter
x = filter(1,a,randn(256,1));

% Fourth-order estimate
pcov(x,4)
```



Algorithm

Linear prediction filters can be used to model the second-order statistical characteristics of a signal. The prediction filter output can be used to model the signal when the input is white noise.

The covariance method estimates the PSD of a signal using the covariance method. The covariance (or nonwindowed) method fits an AR linear prediction filter model to the signal by minimizing the forward prediction error (based on causal observations of your input signal) in the least squares sense. The spectral estimate returned by `pcov` is the squared magnitude of the frequency response of this AR model.

References

[1] Marple, S.L. *Digital Spectral Analysis*, Englewood Cliffs, NJ, Prentice-Hall, 1987, Chapter 7.

[2] Stoica, P., and R.L. Moses, *Introduction to Spectral Analysis*, Prentice-Hall, 1997.

See Also

`arcov`, `lpc`, `pburg`, `peig`, `periodogram`, `pmcov`, `pmtm`, `pmusic`, `pwelch`, `pyulear`

Purpose

Pseudospectrum using eigenvector method

Syntax

```
[S,w] = peig(x,p)
[S,w] = peig(x,p,w)
[S,w] = peig(...,nfft)
[S,f] = peig(x,p,nfft,fs)
[S,f] = peig(x,p,f,fs)
[S,f] = peig(...,'corr')
[S,f] = peig(x,p,nfft,fs,nwin,noverlap)
[...] = peig(...,'range')
[...,v,e] = peig(...)
peig(...)
```

Description

`[S,w] = peig(x,p)` implements the eigenvector spectral estimation method and returns `S`, the pseudospectrum estimate of the input signal `x`, and `w`, a vector of normalized frequencies (in rad/sample) at which the pseudospectrum is evaluated. The pseudospectrum is calculated using estimates of the eigenvectors of a correlation matrix associated with the input data `x`, where `x` is specified as either:

- A row or column vector representing one observation of the signal
- A rectangular array for which each row of `x` represents a separate observation of the signal (for example, each row is one output of an array of sensors, as in array processing), such that `x'*x` is an estimate of the correlation matrix

---

Note You can use the output of `corrmtx` to generate such an array `x`.

You can specify the second input argument `p` as either:

- A scalar integer. In this case, the signal subspace dimension is `p`.
- A two-element vector. In this case, `p(2)`, the second element of `p`, represents a threshold that is multiplied by λ_{min}, the smallest estimated eigenvalue of the signal's correlation matrix. Eigenvalues

below the threshold λ_{min}*p(2) are assigned to the noise subspace. In this case, p(1) specifies the maximum dimension of the signal subspace.

Note If the inputs to peig are real sinusoids, set the value of p to double the number of input signals. If the inputs are complex sinusoids, set p equal to the number of inputs.

The extra threshold parameter in the second entry in p provides you more flexibility and control in assigning the noise and signal subspaces.

S and w have the same length. In general, the length of the FFT and the values of the input x determine the length of the computed S and the range of the corresponding normalized frequencies. The following table indicates the length of S (and w) and the range of the corresponding normalized frequencies for this syntax.

S Characteristics for an FFT Length of 256 (Default)

Real/Complex Input Data	Length of S and w	Range of the Corresponding Normalized Frequencies
Real-valued	129	[0, π]
Complex-valued	256	[0, 2π)

[S,w] = peig(x,p,w) returns the pseudospectrum in the vector S computed at the normalized frequencies specified in vector w, which has two or more elements

[S,w] = peig(...,nfft) specifies the integer length of the FFT nfft used to estimate the pseudospectrum. The default value for nfft (entered as an empty vector []) is 256.

The following table indicates the length of S and w, and the frequency range for w for this syntax.

S and Frequency Vector Characteristics

Real/Complex Input Data	nfft Even/Odd	Length of S and w	Range of w
Real-valued	Even	`(nfft/2 + 1)`	[0, π]
Real-valued	Odd	`(nfft + 1)/2`	[0, π)
Complex-valued	Even or odd	`nfft`	[0, 2π)

`[S,f] = peig(x,p,nfft,fs)` returns the pseudospectrum in the vector `S` evaluated at the corresponding vector of frequencies `f` (in Hz). You supply the sampling frequency `fs` in Hz. If you specify `fs` with the empty vector `[]`, the sampling frequency defaults to 1 Hz.

The frequency range for `f` depends on `nfft`, `fs`, and the values of the input `x`. The length of `S` (and `f`) is the same as in the S and Frequency Vector Characteristics on page 10-652 above. The following table indicates the frequency range for `f` for this syntax.

S and Frequency Vector Characteristics with fs Specified

Real/Complex Input Data	nfft Even/Odd	Range of f
Real-valued	Even	`[0,fs/2]`
Real-valued	Odd	`[0,fs/2)`
Complex-valued	Even or odd	`[0,fs)`

`[S,f] = peig(x,p,f,fs)` returns the pseudospectrum in the vector `S` computed at the frequencies specified in vector `f`, which has two or more elements

`[S,f] = peig(...,'corr')` forces the input argument `x` to be interpreted as a correlation matrix rather than matrix of signal data. For this syntax `x` must be a square matrix, and all of its eigenvalues must be nonnegative.

`[S,f] = peig(x,p,nfft,fs,nwin,noverlap)` allows you to specify `nwin`, a scalar integer indicating a rectangular window length, or a real-valued vector specifying window coefficients. Use the scalar integer `noverlap` in conjunction with `nwin` to specify the number of input sample points by which successive windows overlap. `noverlap` is not used if `x` is a matrix. The default value for `nwin` is `2*p(1)` and `noverlap` is `nwin-1`.

With this syntax, the input data `x` is segmented and windowed before the matrix used to estimate the correlation matrix eigenvalues is formulated. The segmentation of the data depends on `nwin`, `noverlap`, and the form of `x`. Comments on the resulting windowed segments are described in the following table.

Windowed Data Depending on x and nwin

Input data x	Form of nwin	Windowed Data
Data vector	Scalar	Length is `nwin`
Data vector	Vector of coefficients	Length is `length(nwin)`
Data matrix	Scalar	Data is not windowed.
Data matrix	Vector of coefficients	`length(nwin)` must be the same as the column length of `x`, and `noverlap` is not used.

See the table, Eigenvector Length Depending on Input Data and Syntax on page 10-655, for related information on this syntax.

Note The arguments `nwin` and `noverlap` are ignored when you include the string `'corr'` in the syntax.

`[...] = peig(...,'range')` specifies the range of frequency values to include in `f` or `w`. This syntax is useful when `x` is real. *`'range'`* can be either:

- `'whole'`: Compute the pseudospectrum over the frequency range `[0,fs)`. This is the default for determining the frequency range for complex-valued `x`.
  - If you specify `fs` as the empty vector, `[]`, the frequency range is `[0,1)`.
  - If you don't specify `fs`, the frequency range is [0, 2π).
- `'half'`: Compute the pseudospectrum over the frequency ranges specified for real `x`. This is the default for determining the frequency range for real-valued `x`.

Note You can put the string arguments *`'range'`* or `'corr'` anywhere in the input argument list after `p`.

`[...,v,e] = peig(...)` returns the matrix `v` of noise eigenvectors, along with the associated eigenvalues in the vector `e`. The columns of `v` span the noise subspace of dimension `size(v,2)`. The dimension of the signal subspace is `size(v,1)-size(v,2)`. For this syntax, `e` is a vector of estimated eigenvalues of the correlation matrix.

`peig(...)` with no output arguments plots the pseudospectrum in the current figure window.

Remarks

In the process of estimating the pseudospectrum, `peig` computes the noise and signal subspaces from the estimated eigenvectors v_j and eigenvalues λ_j of the signal's correlation matrix. The smallest of these eigenvalues is used in conjunction with the threshold parameter `p(2)` to affect the dimension of the noise subspace in some cases.

The length n of the eigenvectors computed by `peig` is the sum of the dimensions of the signal and noise subspaces. This eigenvector length

depends on your input (signal data or correlation matrix) and the syntax you use.

The following table summarizes the dependency of the eigenvector length on the input argument.

Eigenvector Length Depending on Input Data and Syntax

Form of Input Data x	Comments on the Syntax	Length n of Eigenvectors
Row or column vector	`nwin` is specified as a scalar integer.	`nwin`
Row or column vector	`nwin` is specified as a vector.	`length(nwin)`
Row or column vector	`nwin` is not specified.	`2*p(1)`
l-by-*m* matrix	If `nwin` is specified as a scalar, it is not used. If `nwin` is specified as a vector, `length(nwin)` must equal *m*.	*m*
m-by-*m* nonnegative definite matrix	The string `'corr'` is specified and `nwin` is not used.	*m*

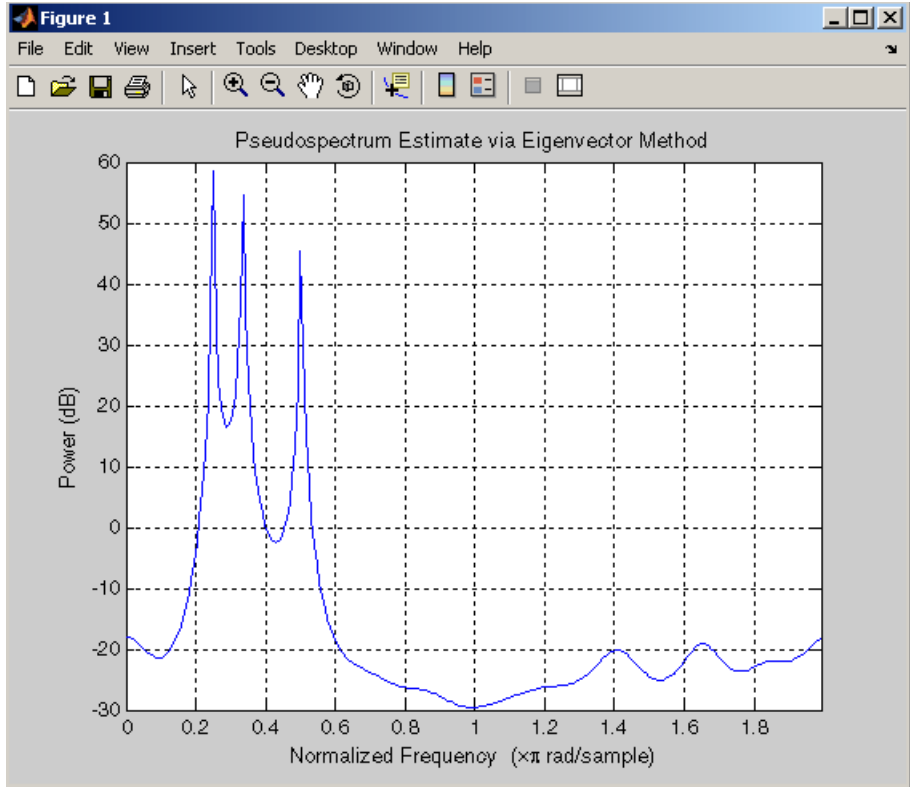
You should specify `nwin` > `p(1)` or `length(nwin)` > `p(1)` if you want `p(2)` > `1` to have any effect.

Examples

Implement the eigenvector method to find the pseudospectrum of the sum of three sinusoids in noise, using the default FFT length of 256. The inputs are complex sinusoids so you set `p` equal to the number of inputs. Use the modified covariance method for the correlation matrix estimate:

```
randn('state',1); n=0:99;
s=exp(i*pi/2*n)+2*exp(i*pi/4*n)+exp(i*pi/3*n)+randn(1,100);
X=corrmtx(s,12,'mod');
```

```
peig(X,3,'whole')              % Uses default NFFT of 256
```



Algorithm

The eigenvector method estimates the pseudospectrum from a signal or a correlation matrix using a weighted version of the MUSIC algorithm derived from Schmidt's eigenspace analysis method [1] [2]. The algorithm performs eigenspace analysis of the signal's correlation matrix in order to estimate the signal's frequency content. The eigenvalues and eigenvectors of the signal's correlation matrix are estimated using `svd` if you don't supply the correlation matrix. This algorithm is particularly suitable for signals that are the sum of sinusoids with additive white Gaussian noise.

The eigenvector method produces a pseudospectrum estimate given by

$$P_{ev}(f) = \frac{1}{\left(\sum_{k=p+1}^{N} \left| \mathbf{v}_k^H \mathbf{e}(f) \right|^2 \right) / \lambda_k}$$

where N is the dimension of the eigenvectors and v_k is the kth eigenvector of the correlation matrix of the input signal. The integer p is the dimension of the signal subspace, so the eigenvectors v_k used in the sum correspond to the smallest eigenvalues λ_k of the correlation matrix. The eigenvectors used span the noise subspace. The vector e(f) consists of complex exponentials, so the inner product

$$\mathbf{v}_k^H \mathbf{e}(f)$$

amounts to a Fourier transform. This is used for computation of the pseudospectrum. The FFT is computed for each v_k and then the squared magnitudes are summed and scaled.

References

[1] Marple, S.L. *Digital Spectral Analysis*, Englewood Cliffs, NJ, Prentice-Hall, 1987, pp. 373-378.

[2] Schmidt, R.O, "Multiple Emitter Location and Signal Parameter Estimation," *IEEE Trans. Antennas Propagation*, Vol. AP-34 (March 1986), pp.276-280.

[3] Stoica, P., and R.L. Moses, *Introduction to Spectral Analysis*, Prentice-Hall, 1997.

See Also

`corrmtx`, `dspdata`, `pburg`, `periodogram`, `pmtm`, `pmusic`, `prony`, `pwelch`, `rooteig`, `rootmusic`, `spectrum`

Purpose

PSD using periodogram

Syntax

```
[Pxx,w] = periodogram(x)
[Pxx,w] = periodogram(x,window)
[Pxx,w] = periodogram(x,window,nfft)
[Pxx,w] = periodogram(x,window,w)
[Pxx,f] = periodogram(x,window,nfft,fs)
[Pxx,f] = periodogram(x,window,f,fs)
[Pxx,f] = periodogram(x,window,nfft,fs,'range')
[Pxx,w] = periodogram(x,window,nfft,'range')
periodogram(...)
```

Description

`[Pxx,w] = periodogram(x)` returns the power spectral density (PSD) estimate `Pxx` of the sequence `x` using a periodogram. The power spectral density is calculated in units of power per radians per sample. The corresponding vector of frequencies `w` is computed in radians per sample, and has the same length as `Pxx`.

A real-valued input vector `x` produces a full power one-sided (in frequency) PSD (by default), while a complex-valued `x` produces a two-sided PSD.

In general, the length N of the FFT and the values of the input `x` determine the length of `Pxx` and the range of the corresponding normalized frequencies. For this syntax, the (default) length N of the FFT is the larger of 256 and the next power of 2 greater than the length of `x`. The following table indicates the length of `Pxx` and the range of the corresponding normalized frequencies for this syntax.

PSD Vector Characteristics for an FFT Length of N (Default)

Real/Complex Input Data	Length of Pxx	Range of the Corresponding Normalized Frequencies
Real-valued	$(N/2) +1$	$[0, \pi]$
Complex-valued	N	$[0, 2\pi)$

`[Pxx,w] = periodogram(x,window)` returns the PSD estimate `Pxx` computed using the modified periodogram method. The vector `window` specifies the coefficients of the window used in computing a modified periodogram of the input signal. Both input arguments must be vectors of the same length. When you don't supply the second argument `window`, or set it to the empty vector `[]`, a rectangular window (`rectwin`) is used by default. In this case the standard periodogram is calculated.

`[Pxx,w] = periodogram(x,window,nfft)` uses the modified periodogram to estimate the PSD while specifying the length of the FFT with the integer `nfft`. If you set `nfft` to the empty vector `[]`, it takes the default value for *N* listed in the previous syntax.

The length of `Pxx` and the frequency range for `w` depend on `nfft` and the values of the input `x`. The following table indicates the length of `Pxx` and the frequency range for `w` for this syntax.

PSD and Frequency Vector Characteristics

Real/Complex Input Data	nfft Even/Odd	Length of Pxx	Range of w
Real-valued	Even	`(nfft/2 + 1)`	[0, π]
Real-valued	Odd	`(nfft + 1)/2`	[0, π)
Complex-valued	Even or odd	`nfft`	[0, 2π)

Note `periodogram` uses an `nfft`-point FFT of the windowed data (`x.*window`) to compute the periodogram. If the value you specify for `nfft` is less than the length of `x`, then `x.*window` is wrapped modulo `nfft`. If the value you specify for `nfft` is greater than the length of `x`, then `x.*window` is zero-padded to compute the FFT.

`[Pxx,w] = periodogram(x,window,w)` estimates the two-sided PSD at the normalized frequencies specified in the vector `w` using the Goertzel algorithm. The frequencies of `w` are rounded to the nearest DFT bin

commensurate with the resolution of the signal. The units of `w` are rad/sample.

`[Pxx,f] = periodogram(x,window,nfft,fs)` uses the sampling frequency `fs` specified as an integer in hertz (Hz) to compute the PSD vector (`Pxx`) and the corresponding vector of frequencies (f). In this case, the units for the frequency vector are in Hz. The spectral density produced is calculated in units of power per Hz. If you specify `fs` as the empty vector `[]`, the sampling frequency defaults to 1 Hz.

The frequency range for `f` depends on `nfft`, `fs`, and the values of the input `x`. The length of `Pxx` is the same as in the table above. The following table indicates the frequency range for `f` for this syntax.

PSD and Frequency Vector Characteristics with fs Specified

Real/Complex Input Data	nfft Even/Odd	Range of f
Real-valued	Even	`[0,fs/2]`
Real-valued	Odd	`[0,fs/2)`
Complex-valued	Even or odd	`[0,fs)`

`[Pxx,f] = periodogram(x,window,f,fs)` uses the vector of frequencies `f` at which the PSD is estimated. The frequencies of `f` are rounded to the nearest DFT bin commensurate with the resolution of the signal.

`[Pxx,f] = periodogram(x,window,nfft,fs,'range')` or

`[Pxx,w] = periodogram(x,window,nfft,'range')` specifies the range of frequency values to include in `f` or `w`. This syntax is useful when `x` is real. `'range'` can be either:

- `'twosided'`: Compute the two-sided PSD over the frequency range `[0,fs)`. This is the default for determining the frequency range for complex-valued `x`.

  - If you specify `fs` as the empty vector, `[]`, the frequency range is `[0,1)`.
  - If you don't specify `fs`, the frequency range is [0, 2π).
- `'onesided'`: Compute the one-sided PSD over the frequency ranges specified for real `x`. This is the default for determining the frequency range for real-valued `x`.

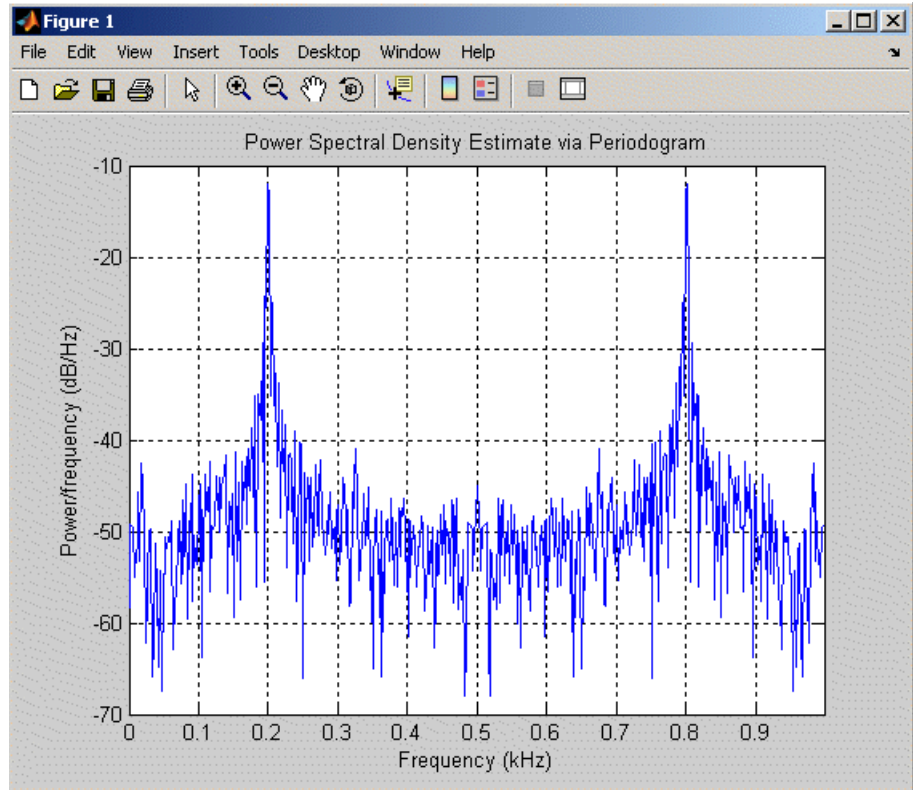
Note You can put the string argument *`'range'`* anywhere in the input argument list after window.

`periodogram(...)` with no outputs plots the power spectral density in dB per unit frequency in the current figure window. The frequency range on the plot is the same as the range of output `w` (or `f`) for the syntax you use.

Examples

Compute the periodogram of a 200 Hz signal embedded in additive noise using the default window:

```
randn('state',0);
Fs = 1000;
t = 0:1/Fs:.3;
x = cos(2*pi*t*200)+0.1*randn(size(t));
periodogram(x,[],'twosided',512,Fs)
```



Algorithm

The periodogram for a sequence $[x_1, \ldots, x_n]$ is given by the following formula:

$$S\left(e^{jw}\right) = \frac{1}{Fn}\left|\sum_{l=1}^{n} w_l e^{-jwl}\right|^2$$

This expression forms an estimate of the PSD of the signal defined by the sequence $[x_1, \ldots, x_n]$.

If you weight your signal sequence by a window [w_1, ... , w_n], then the weighted or modified periodogram is defined as

$$S\left(e^{jw}\right) = \frac{\frac{1}{Fn}\left|\sum_{l=1}^{n} w_l x_l e^{-jwl}\right|^2}{\frac{1}{n}\sum_{l=1}^{n}|w_l|^2}$$

In either case, `periodogram` uses an `nfft`-point FFT to compute the power spectral density as $S\left(e^{jw}\right)$, where F is

- 2π when you do not supply the sampling frequency
- `fs` when you supply the sampling frequency

References

[1] Stoica, P., and R.L. Moses, *Introduction to Spectral Analysis*, Prentice-Hall, 1997, pp. 24-26.

[2] Welch, P.D, "The Use of Fast Fourier Transform for the Estimation of Power Spectra: A Method Based on Time Averaging Over Short, Modified Periodograms," *IEEE Trans. Audio Electroacoustics, Vol. AU-15 (June 1967)*, pp. 70-73.

[3] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, pp.730-742.

See Also

`dspdata.msspectrum`, `pburg`, `pcov`, `peig`, `pmcov`, `pmtm`, `pmusic`, `pwelch`, `pyulear`

phasedelay

Purpose

Phase delay of digital filter

Syntax

```
[phi,w] = phasedelay(b,a,n)
[phi,w] = phasedelay(b,a,n,'whole')
phi = phasedelay(b,a,w)
[phi,f] = phasedelay(b,a,n,fs)
[phi,f] = phasedelay(b,a,n,'whole',fs)
phi = phasedelay(b,a,f,fs)
[phi,w,s] = phasedelay(...)
[phi,f,s] = phasedelay(...)
phasedelay(b,a,...)
```

Description

`[phi,w] = phasedelay(b,a,n)` returns the `n`-point phase delay response vector `phi` and the `n`-point frequency reponse vector `w` (in radians/sample) of the filter defined by numerator coefficients `b` and denominator coefficients `a`. The phase delay response is evaluated at `n` equally spaced points around the upper half of the unit circle. If `n` is omitted, it defaults to 512.

`[phi,w] = phasedelay(b,a,n,'whole')` uses `n` equally spaced points around the whole unit circle.

`phi = phasedelay(b,a,w)` returns the phase delay response at frequencies specified in vector `w` (in radians/sample). The frequencies are normally between 0 and π.

`[phi,f] = phasedelay(b,a,n,fs)` and `[phi,f] = phasedelay(b,a,n,'whole',fs)` return the phase delay vector `f` (in Hz), using the sampling frequency `fs` (in Hz).

`phi = phasedelay(b,a,f,fs)` returns the phase delay response at the frequencies specified in vector `f` (in Hz), using the sampling frequency `fs` (in Hz)..

`[phi,w,s] = phasedelay(...)` and `[phi,f,s] = phasedelay(...)` return plotting information, where `s` is a structure with fields you can change to display different frequency response plots.

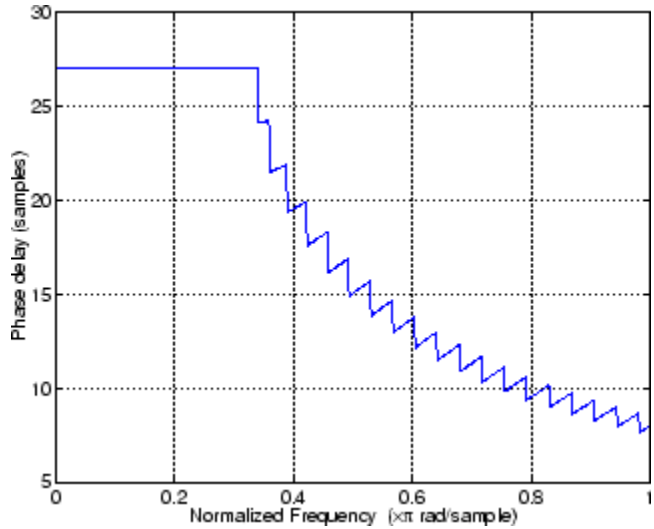
`phasedelay(b,a,...)` with no output arguments, plots the phase delay response of the filter.

Examples

Example 1

Plot the phase delay response of a constrained least squares FIR filter:

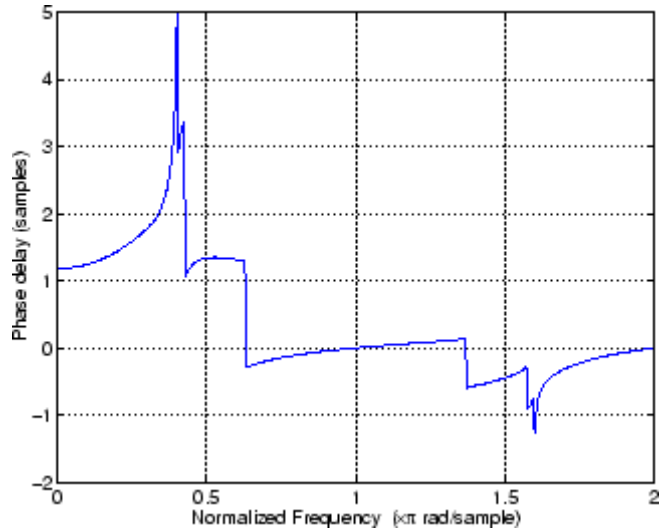
```
b=fircls1(54,.3,.02,.008);
phasedelay(b)
```



Example 2

Plot the phase delay response of an elliptic filter:

```
[b,a] = ellip(10,.5,20,.4);
phasedelay(b,a,512,'whole')
```



See Also

`freqz`, `fvtool`, `phasez`, `grpdelay`

Purpose

Phase response of digital filter

Syntax

```
[phi,w] = phasez(b,a,n)
[phi,w] = phasez(b,a,n,'whole')
phi = phasez(b,a,w)
[phi,f] = phasez(b,a,n,fs)
[phi,f] = phasez(b,a,n,'whole',fs)
phi = phasez(b,a,f,fs)
[phi,w,s] = phasez(...)
[phi,f,s] = phasez(...)
phasez(b,a,...)
phasez(Hd)
```

Description

`[phi,w] = phasez(b,a,n)` returns the `n`-point phase response vector `phi` and the `n`-point frequency reponse vector `w` (in radians/sample) of the filter defined by numerator coefficients `b` and denominator coefficients `a`. The phase response is evaluated at `n` equally spaced points around the upper half of the unit circle. If `n` is omitted, it defaults to 512. `phasez` returns the unwrapped phase response.

`[phi,w] = phasez(b,a,n,'whole')` uses `n` equally spaced points around the whole unit circle.

`phi = phasez(b,a,w)` returns the phase response at frequencies specified in vector `w` (in radians/sample). The frequencies are normally between 0 and π.

`[phi,f] = phasez(b,a,n,fs)` and `[phi,f] = phasez(b,a,n,'whole',fs)` return the phase vector `f` (in Hz), using the sampling frequency `fs` (in Hz).

`phi = phasez(b,a,f,fs)` returns the phase response at the frequencies specified in vector `f` (in Hz), using the sampling frequency `fs` (in Hz)..

`[phi,w,s] = phasez(...)` and `[phi,f,s] = phasez(...)` return plotting information, where `s` is a structure with fields you can change to display different frequency response plots.

`phasez(b,a,...)` with no output arguments, plots the phase response of the filter in the current filter window.

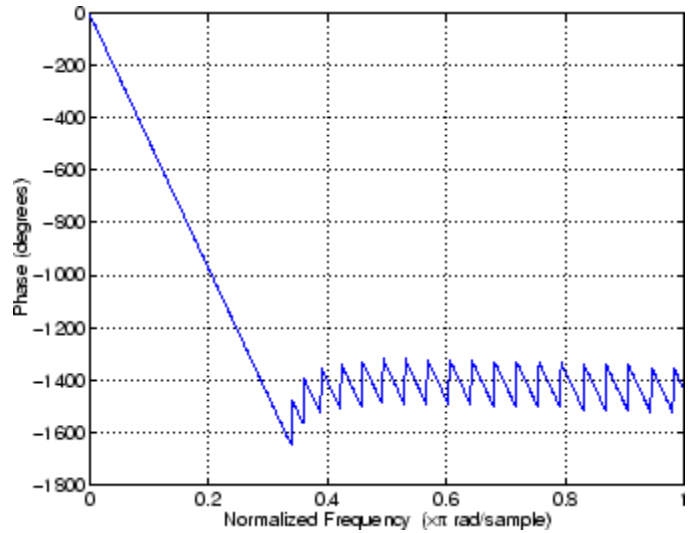
`phasez(Hd)` plots the phase response of the filter and displays the plot in `fvtool`. The input `Hd` is a `dfilt` filter object.

Examples

Example 1

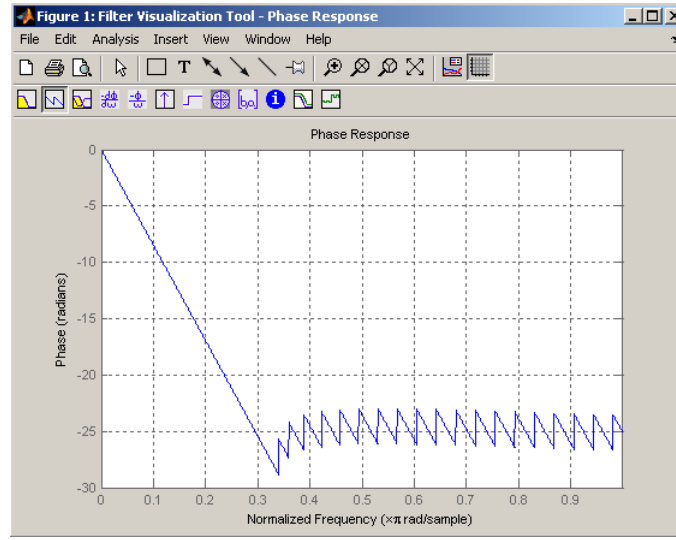
Plot the phase response of a constrained least squares FIR filter:

```
b=fircls1(54,.3,.02,.008);
phasez(b)
```



The same example using a `dfilt` filter object and displaying the result in `fvtool`, where you can perform more analyses, is

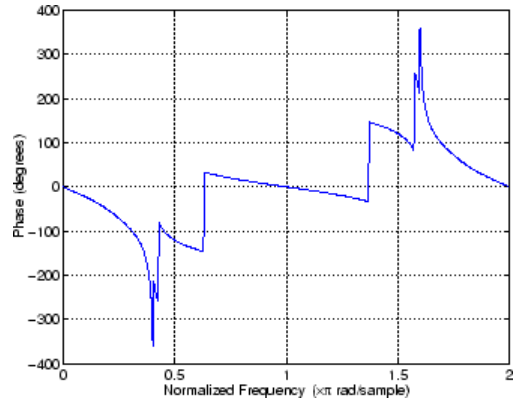
```
b=fircls1(54,.3,.02,.008);
Hd=dfilt.dffir(b);
phasez(Hd)
```



Example 2

Plot the phase response of an elliptic filter:

```
[b,a] = ellip(10,.5,20,.4);
phasez(b,a,512,'whole')
```



See Also

freqz, fvtool, phasedelay, grpdelay

Purpose

PSD using modified covariance method

Syntax

```
Pxx = pmcov(x,p)
Pxx = pmcov(x,p,nfft)
[Pxx,w] = pmcov(...)
[Pxx,w] = pmcov(x,p,w)
Pxx = pmcov(x,p,nfft,fs)
Pxx = pmcov(x,p,f,fs)
[Pxx,f] = pmcov(x,p,nfft,fs)
[Pxx,f] = pmcov(x,p,f,fs)
[Pxx,f] = pmcov(x,p,nfft,fs,'range')
[Pxx,w] = pmcov(x,p,nfft,'range')
pmcov(...)
```

Description

`Pxx = pmcov(x,p)` implements the modified covariance algorithm, a parametric spectral estimation method, and returns `Pxx`, an estimate of the power spectral density (PSD) of the vector `x`. The entries of `x` represent samples of a discrete-time signal, and `p` is the integer specifying the order of an autoregressive (AR) prediction model for the signal, used in estimating the PSD.

The power spectral density is calculated in units of power per radians per sample. Real-valued inputs produce full power one-sided (in frequency) PSDs (by default), while complex-valued inputs produce two-sided PSDs.

In general, the length of the FFT and the values of the input `x` determine the length of `Pxx` and the range of the corresponding normalized frequencies. For this syntax, the (default) FFT length is 256. The following table indicates the length of `Pxx` and the range of the corresponding normalized frequencies for this syntax.

PSD Vector Characteristics for an FFT Length of 256 (Default)

Real/Complex Input Data	Length of Pxx	Range of the Corresponding Normalized Frequencies
Real-valued	129	[0, π]
Complex-valued	256	[0, 2π)

`Pxx = pmcov(x,p,nfft)` uses the integer FFT length `nfft` to calculate the PSD vector `Pxx`.

`[Pxx,w] = pmcov(...)` also returns `w`, a vector of normalized angular frequencies at which the two-sided PSD is estimated. `Pxx` and `w` have the same length. The units for `w` are rad/sample.

The length of `Pxx` and the frequency range for `w` depend on `nfft` and the values of the input `x`. The following table indicates the length of `Pxx` and the frequency range for `w` in this syntax.

PSD and Frequency Vector Characteristics

Real/Complex Input Data	nfft Even/Odd	Length of Pxx	Range of w
Real-valued	Even	`(nfft/2 + 1)`	[0, π]
Real-valued	Odd	`(nfft + 1)/2`	[0, π)
Complex-valued	Even or odd	`nfft`	[0, 2π)

`[Pxx,w] = pmcov(x,p,w)` uses a vector of normalized frequencies `w` with two or more elements to compute the PSD at those frequencies and returns a two-sided PSD.

```
Pxx = pmcov(x,p,nfft,fs)
```

or

`Pxx = pmcov(x,p,f,fs)` uses the integer FFT length `nfft` to calculate the PSD vector `Pxx` or uses the vector of frequencies `f` in Hz and the

sampling frequency `fs` to compute the two-sided PSD vector `Pxx` at those frequencies. If you specify `nfft` as the empty vector `[]`, it uses the default value of 256. If you specify `fs` as the empty vector `[]`, the sampling frequency `fs` defaults to 1 Hz. The spectral density produced is calculated in units of power per Hz.

```
[Pxx,f] = pmcov(x,p,nfft,fs)
```

or

`[Pxx,f] = pmcov(x,p,f,fs)` returns the frequency vector f. In this case, the units for the frequency vector are in Hz. The frequency range for `f` depends on `nfft`, `fs`, and the values of the input `x`. The length of `Pxx` is the same as in the table above. The following table indicates the frequency range for `f` for this syntax.

PSD and Frequency Vector Characteristics with fs Specified

Real/Complex Input Data	nfft Even/Odd	Range of f
Real-valued	Even	`[0,fs/2]`
Real-valued	Odd	`[0,fs/2)`
Complex-valued	Even or odd	`[0,fs)`

`[Pxx,f] = pmcov(x,p,nfft,fs,'range')` or

`[Pxx,w] = pmcov(x,p,nfft,'range')` specifies the range of frequency values to include in `f` or `w`. This syntax is useful when `x` is real. `'range'` can be either:

- `'twosided'`: Compute the two-sided PSD over the frequency range `[0,fs)`. This is the default for determining the frequency range for complex-valued `x`.
  - If you specify `fs` as the empty vector, `[]`, the frequency range is `[0,1)`.
  - If you don't specify `fs`, the frequency range is [0, 2π).

- `'onesided'`: Compute the one-sided PSD over the frequency ranges specified for real `x`. This is the default for determining the frequency range for real-valued `x`. Note that `'onesided'` is not valid if you pass in a vector of frequencies (`f` or `w`).

Note You can put the string argument *`'range'`* anywhere in the input argument list after `p`.

`pmcov(...)` with no outputs plots the power spectral density in the current figure window. The frequency range on the plot is the same as the range of output `w` (or `f`) for a given set of parameters.

Remarks

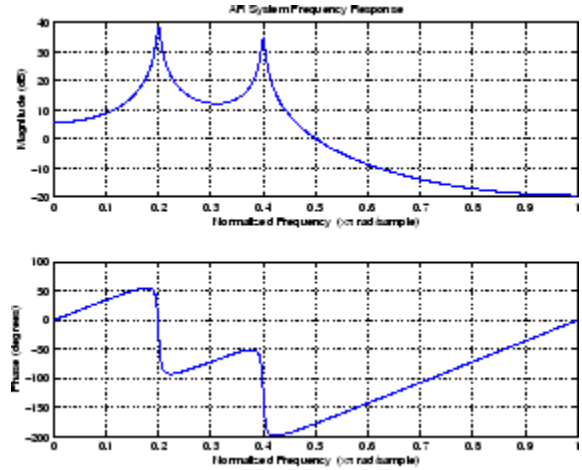
The power spectral density is computed as the distribution of power per unit frequency.

This algorithm depends on your selecting an appropriate model order for your signal.

Examples

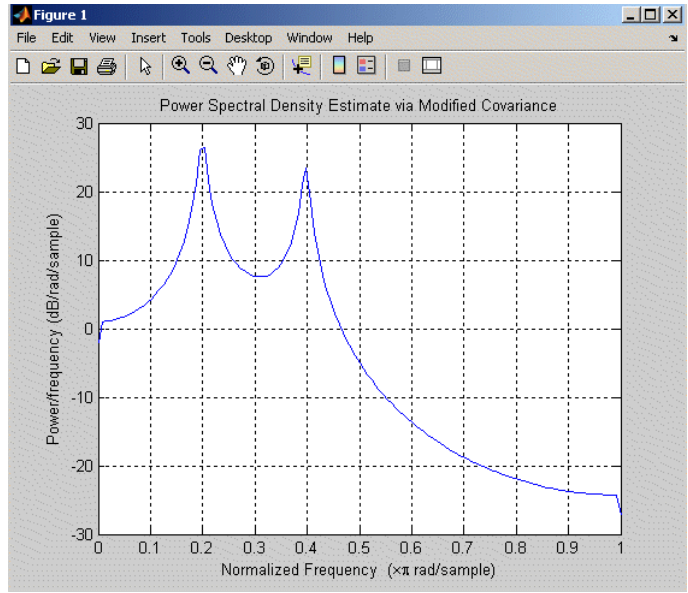
Because the modified covariance method estimates the spectral density by fitting an AR prediction model of a given order to the signal, first generate a signal from an AR (all-pole) model of a given order. You can use `freqz` to check the magnitude of the frequency response of your AR filter. This will give you an idea of what to expect when you estimate the PSD using `pmcov`:

```
a = [1 -2.2137 2.9403 -2.1697 0.9606];    % AR filter coefficients
freqz(1,a)                                % AR filter frequency response
title('AR System Frequency Response')
```



Now generate the input signal `x` by filtering white noise through the AR filter. Estimate the PSD of `x` based on a fourth-order AR prediction model since in this case we know that the original AR system model `a` has order 4:

```
randn('state',1);
x = filter(1,a,randn(256,1));     % AR filter output
pmcov(x,4)                        % Fourth-order estimate
```



Algorithm

Linear prediction filters can be used to model the second-order statistical characteristics of a signal. The prediction filter output can be used to model the signal when the input is white noise.

`pmcov` estimates the PSD of the signal vector using the modified covariance method. This method fits an autoregressive (AR) linear prediction filter model to the signal by simultaneously minimizing the forward and backward prediction errors (based on causal observations of your input signal) in the least squares sense. The spectral estimate returned by `pmcov` is the magnitude squared frequency response of this AR model.

References

[1] Marple, S.L. *Digital Spectral Analysis*, Englewood Cliffs, NJ, Prentice-Hall, 1987, Chapter 7.

[2] Stoica, P., and R.L. Moses, *Introduction to Spectral Analysis*, Prentice-Hall, 1997.

See Also

armcov, lpc, pburg, pcov, peig, periodogram, pmtm, pmusic, pwelch, prony, pyulear

Purpose

PSD using multitaper method (MTM)

Syntax

```
[Pxx,w] = pmtm(x,nw)
[Pxx,w] = pmtm(x,nw,nfft)
[Pxx,w] = pmtm(x,nw,w)
[Pxx,f] = pmtm(x,nw,nfft,fs)
[Pxx,w] = pmtm(x,nw,f,fs)
[Pxx,Pxxc,f] = pmtm(x,nw,nfft,fs)
[Pxx,Pxxc,f] = pmtm(x,nw,nfft,fs,p)
[Pxx,Pxxc,f] = pmtm(x,e,v,nfft,fs,p)
[Pxx,Pxxc,f] = pmtm(x,dpss_params,nfft,fs,p)
[...] = pmtm(...,'DropLastTaper',dropflag)
[...] = pmtm(...,'method')
[...] = pmtm(...,'range')
pmtm(...)
```

Description

`pmtm` estimates the power spectral density (PSD) of the time series x using the multitaper method (MTM) described in [1]. This method uses linear or nonlinear combinations of modified periodograms to estimate the PSD. These periodograms are computed using a sequence of orthogonal tapers (windows in the frequency domain) specified from the discrete prolate spheroidal sequences (see `dpss`).

`[Pxx,w] = pmtm(x,nw)` estimates the PSD Pxx for the input signal `x`, using `2*nw-1` discrete prolate spheroidal sequences as data tapers for the multitaper estimation method. `nw` is the time-bandwidth product for the discrete prolate spheroidal sequences. If you specify `nw` as the empty vector `[]`, a default value of 4 is used. Other typical choices are 2, 5/2, 3, or 7/2. `pmtm` also returns `w`, a vector of frequencies at which the PSD is estimated. `Pxx` and `w` have the same length. The units for frequency are rad/sample.

The power spectral density is calculated in units of power per radians per sample. Real-valued inputs produce (by default) full power one-sided (in frequency) PSDs, while complex-valued inputs produce two-sided PSDs.

In general, the length N of the FFT and the values of the input `x` determine the length of `Pxx` and the range of the corresponding normalized frequencies. For this syntax, the (default) length N of the FFT is the larger of 256 and the next power of 2 greater than the length of the segment. The following table indicates the length of `Pxx` and the range of the corresponding normalized frequencies for this syntax.

PSD Vector Characteristics for an FFT Length of N (Default)

Real/Complex Input Data	Length of Pxx	Range of the Corresponding Normalized Frequencies
Real-valued	$(N/2) +1$	$[0, \pi]$
Complex-valued	N	$[0, 2\pi)$

`[Pxx,w] = pmtm(x,nw,nfft)` uses the multitaper method to estimate the PSD while specifying the length of the FFT with the integer `nfft`. If you specify `nfft` as the empty vector `[]`, it adopts the default value for N described in the previous syntax.

The length of `Pxx` and the frequency range for `w` depend on `nfft` and the values of the input `x`. The following table indicates the length of `Pxx` and the frequency range for `w` for this syntax.

PSD and Frequency Vector Characteristics

Real/Complex Input Data	nfft Even/Odd	Length of Pxx	Range of w
Real-valued	Even	(nfft/2 + 1)	$[0, \pi]$
Real-valued	Odd	(nfft + 1)/2	$[0, \pi)$
Complex-valued	Even or odd	nfft	$[0, 2\pi)$

`[Pxx,w] = pmtm(x,nw,w)` estimates the two-sided PSD at the normalized frequencies specified in the vector `w` using the Goertzel algorithm. The frequencies of `w` are rounded to the nearest DFT bin

commensurate with the resolution of the signal. The units of `w` are rad/sample.

`[Pxx,f] = pmtm(x,nw,nfft,fs)` uses the sampling frequency `fs` specified as an integer in hertz (Hz) to compute the PSD vector (`Pxx`) and the corresponding vector of frequencies (f). In this case, the units for the frequency vector f are in Hz. The spectral density produced is calculated in units of power per Hz. If you specify `fs` as the empty vector `[]`, the sampling frequency defaults to 1 Hz.

The frequency range for `f` depends on `nfft`, `fs`, and the values of the input `x`. The length of `Pxx` is the same as in the table, PSD and Frequency Vector Characteristics on page 10-679 above. The following table indicates the frequency range for `f` for this syntax.

PSD and Frequency Vector Characteristics with fs Specified

Real/Complex Input Data	nfft Even/Odd	Range of f
Real-valued	Even	[0, fs/2]
Real-valued	Odd	[0, fs/2)
Complex-valued	Even or odd	[0, fs)

`[Pxx,w] = pmtm(x,nw,f,fs)` estimates the two-sided PSD at the frequencies specified in the vector `f` using the Goertzel algorithm. The frequencies of `f` are rounded to the nearest DFT bin commensurate with the resolution of the signal. The units of `w` are rad/sample.

`[Pxx,Pxxc,f] = pmtm(x,nw,nfft,fs)` returns `Pxxc`, the 95% confidence interval for `Pxx`. Confidence intervals are computed using a chi-squared approach. `Pxxc` is a two-column matrix with the same number of rows as `Pxx`. `Pxxc(:,1)` is the lower bound of the confidence interval and `Pxxc(:,2)` is the upper bound of the confidence interval.

`[Pxx,Pxxc,f] = pmtm(x,nw,nfft,fs,p)` returns `Pxxc`, the `p*100%` confidence interval for `Pxx`, where `p` is a scalar between 0 and 1. If you don't specify `p`, or if you specify `p` as the empty vector `[]`, the default 95% confidence interval is used.

`[Pxx,Pxxc,f] = pmtm(x,e,v,nfft,fs,p)` returns the PSD estimate `Pxx`, the confidence interval `Pxxc`, and the frequency vector `f` from the data tapers contained in the columns of the matrix `e`, and their concentrations in the vector `v`. The length of `v` is the same as the number of columns in `e`. You can obtain the data to supply as these arguments from the outputs of `dpss`.

`[Pxx,Pxxc,f] = pmtm(x,dpss_params,nfft,fs,p)` uses the cell array `dpss_params` containing the input arguments to `dpss` (listed in order, but excluding the first argument) to compute the data tapers. For example, `pmtm(x,{3.5,'trace'},512,1000)` calculates the prolate spheroidal sequences for `nw` = `3.5`, using `nfft` = 512, and `fs` = 1000, and displays the method that `dpss` uses for this calculation. See `dpss` for other options.

`[...] = pmtm(...,'DropLastTaper',dropflag)` specifies whether the last taper or eigenvector should be dropped during the MTM calculation of the PSD. The last taper is associated with an eigenvalue that is typically much smaller than one and this taper is usually not included in the calculation. The default value of `dropflag` is `true`. Setting `dropflag` to `false` forces the last taper to be included in the calculation.

`[...] = pmtm(...,'`*`method`*`')` specifies the algorithm used for combining the individual spectral estimates. The string *`'method'`* can be one of the following:

- `'adapt'`: Thomson's adaptive nonlinear combination (default)
- `'unity'`: A linear combination of the weighted periodograms with unity weights
- `'eigen'`: A linear combination of the weighted periodograms with eigenvalue weights

`[...] = pmtm(...,`*`'range'`*`)` specifies the range of frequency values to include in `f` or `w`. This syntax is useful when `x` is real. *`'range'`* can be either:

- `'twosided'`: Compute the two-sided PSD over the frequency range `[0,fs)`. This is the default for determining the frequency range for complex-valued `x`.
  - If you specify `fs` as the empty vector, `[]`, the frequency range is `[0,1)`.
  - If you don't specify `fs`, the frequency range is [0, 2π).
- `'onesided'`: Compute the one-sided PSD over the frequency ranges specified for real `x`. This is the default for determining the frequency range for real-valued `x`.

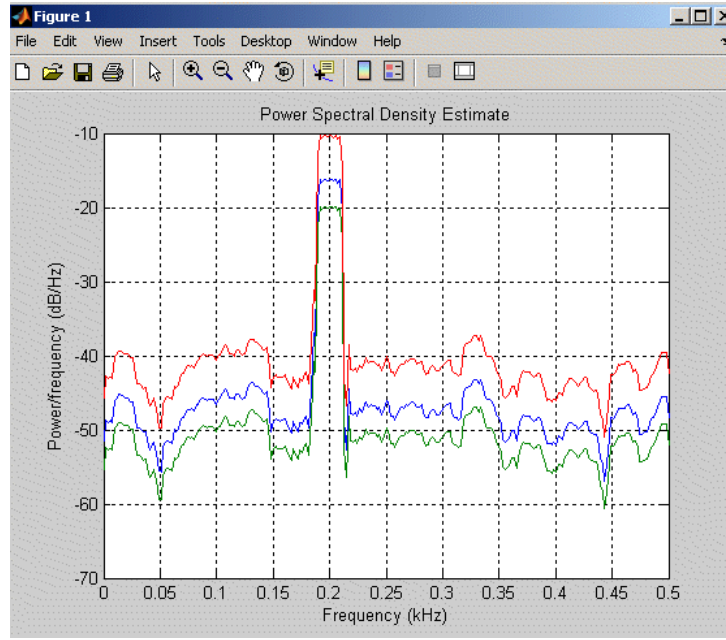
Note You can put the string arguments *`'range'`* or *`'method'`* anywhere after the input argument `nw` or `v`.

`pmtm(...)` with no output arguments plots the PSD estimate and the confidence intervals in the current figure window. If you don't specify `fs`, the 95% confidence interval is plotted. If you do specify `fs`, the confidence intervals plotted depend on the value of `p`.

Examples

This example analyzes a sinusoid in white noise:

```
randn('state',0);
fs = 1000;
t = 0:1/fs:0.3;
x = cos(2*pi*t*200) + 0.1*randn(size(t));
[Pxx,Pxxc,f] = pmtm(x,3.5,512,fs,0.99);
hpsd = dspdata.psd([Pxx Pxxc],'Fs',fs);
plot(hpsd)
```



References

[1] Percival, D.B., and A.T. Walden, *Spectral Analysis for Physical Applications: Multitaper and Conventional Univariate Techniques*, Cambridge University Press, 1993.

[2] Thomson, D.J., "Spectrum estimation and harmonic analysis," *Proceedings of the IEEE, Vol. 70* (1982), pp. 1055-1096.

See Also

`dpss`, `pburg`, `pcov`, `peig`, `periodogram`, `pmcov`, `pmusic`, `pwelch`, `pyulear`

Purpose

Pseudospectrum using MUSIC algorithm

Syntax

```
[S,w] = pmusic(x,p)
[S,w] = pmusic(x,p,w)
[S,w] = pmusic(...,nfft)
[S,f] = pmusic(x,p,nfft,fs)
[S,f] = pmusic(x,p,f,fs)
[S,f] = pmusic(...,'corr')
[S,f] = pmusic(x,p,nfft,fs,nwin,noverlap)
[...] = pmusic(...,'range')
[...,v,e] = pmusic(...)
pmusic(...)
```

Description

`[S,w] = pmusic(x,p)` implements the MUSIC (Multiple Signal Classification) algorithm and returns `S`, the pseudospectrum estimate of the input signal `x`, and a vector `w` of normalized frequencies (in rad/sample) at which the pseudospectrum is evaluated. The pseudospectrum is calculated using estimates of the eigenvectors of a correlation matrix associated with the input data `x`, where `x` is specified as either:

- A row or column vector representing one observation of the signal
- A rectangular array for which each row of `x` represents a separate observation of the signal (for example, each row is one output of an array of sensors, as in array processing), such that `x'*x` is an estimate of the correlation matrix

Note You can use the output of `corrmtx` to generate such an array `x`.

You can specify the second input argument `p` as either:

- A scalar integer. In this case, the signal subspace dimension is `p`.
- A two-element vector. In this case, `p(2)`, the second element of `p`, represents a threshold that is multiplied by λ_{min}, the smallest

estimated eigenvalue of the signal's correlation matrix. Eigenvalues below the threshold λ_{min}`*p(2)` are assigned to the noise subspace. In this case, `p(1)` specifies the maximum dimension of the signal subspace.

Note If the inputs to `pmusic` are real sinusoids, set the value of `p` to double the number of input signals. If the inputs are complex sinusoids, set `p` equal to the number of inputs.

The extra threshold parameter in the second entry in `p` provides you more flexibility and control in assigning the noise and signal subspaces.

`S` and `w` have the same length. In general, the length of the FFT and the values of the input `x` determine the length of the computed `S` and the range of the corresponding normalized frequencies. The following table indicates the length of `S` (and `w`) and the range of the corresponding normalized frequencies for this syntax.

S Characteristics for an FFT Length of 256 (Default)

Real/Complex Input Data	Length of S and w	Range of the Corresponding Normalized Frequencies
Real-valued	129	[0, π]
Complex-valued	256	[0, 2π)

`[S,w] = pmusic(x,p,w)` returns the pseudospectrum in the vector `S` computed at the normalized frequencies specified in vector `w`, which has two or more elements

`[S,w] = pmusic(...,nfft)` specifies the integer length of the FFT `nfft` used to estimate the pseudospectrum. The default value for `nfft` (entered as an empty vector `[]`) is 256.

The following table indicates the length of S and w, and the frequency range for w in this syntax.

S and Frequency Vector Characteristics

Real/Complex Input Data	nfft Even/Odd	Length of S and w	Range of w
Real-valued	Even	(nfft/2 + 1)	[0, π]
Real-valued	Odd	(nfft + 1)/2	[0, π)
Complex-valued	Even or odd	nfft	[0, 2π)

[S,f] = pmusic(x,p,nfft,fs) returns the pseudospectrum in the vector S evaluated at the corresponding vector of frequencies f (in Hz). You supply the sampling frequency fs in Hz. If you specify fs with the empty vector [], the sampling frequency defaults to 1 Hz.

The frequency range for f depends on nfft, fs, and the values of the input x. The length of S (and f) is the same as in the S and Frequency Vector Characteristics on page 10-686 above. The following table indicates the frequency range for f for this syntax.

S and Frequency Vector Characteristics with fs Specified

Real/Complex Input Data	nfft Even/Odd	Range of f
Real-valued	Even	[0,fs/2]
Real-valued	Odd	[0,fs/2)
Complex-valued	Even or odd	[0,fs)

[S,f] = pmusic(x,p,f,fs) returns the pseudospectrum in the vector S computed at the frequencies specified in vector f, which has two or more elements

[S,f] = pmusic(...,'corr') forces the input argument x to be interpreted as a correlation matrix rather than matrix of signal data.

For this syntax `x` must be a square matrix, and all of its eigenvalues must be nonnegative.

`[S,f] = pmusic(x,p,nfft,fs,nwin,noverlap)` allows you to specify `nwin`, a scalar integer indicating a rectangular window length, or a real-valued vector specifying window coefficients. Use the scalar integer `noverlap` in conjunction with `nwin` to specify the number of input sample points by which successive windows overlap. `noverlap` is not used if `x` is a matrix. The default value for `nwin` is `2*p(1)` and `noverlap` is `nwin-1`.

With this syntax, the input data `x` is segmented and windowed before the matrix used to estimate the correlation matrix eigenvalues is formulated. The segmentation of the data depends on `nwin`, `noverlap`, and the form of `x`. Comments on the resulting windowed segments are described in the following table.

Windowed Data Depending on x and nwin

Input data x	Form of nwin	Windowed Data
Data vector	Scalar	Length is `nwin`
Data vector	Vector of coefficients	Length is `length(nwin)`
Data matrix	Scalar	Data is not windowed.
Data matrix	Vector of coefficients	`length(nwin)` must be the same as the column length of `x`, and `noverlap` is not used.

See the Eigenvector Length Depending on Input Data and Syntax on page 10-689 below for related information on this syntax.

Note The arguments `nwin` and `noverlap` are ignored when you include the string `'corr'` in the syntax.

`[...] = pmusic(...,'range')` specifies the range of frequency values to include in `f` or `w`. This syntax is useful when `x` is real. *`'range'`* can be either:

- `'whole'`: Compute the pseudospectrum over the frequency range `[0,fs)`. This is the default for determining the frequency range for complex-valued `x`.
  - If you specify `fs` as the empty vector, `[]`, the frequency range is `[0,1)`.
  - If you don't specify `fs`, the frequency range is [0, 2π).
- `'half'`: Compute the pseudospectrum over the frequency ranges specified for real `x`. This is the default for determining the frequency range for real-valued `x`.

Note You can put the string arguments *`'range'`* or `'corr'` anywhere in the input argument list after `p`.

`[...,v,e] = pmusic(...)` returns the matrix `v` of noise eigenvectors, along with the associated eigenvalues in the vector `e`. The columns of `v` span the noise subspace of dimension `size(v,2)`. The dimension of the signal subspace is `size(v,1)-size(v,2)`. For this syntax, `e` is a vector of estimated eigenvalues of the correlation matrix.

`pmusic(...)` with no output arguments plots the pseudospectrum in the current figure window.

Remarks

In the process of estimating the pseudospectrum, `pmusic` computes the noise and signal subspaces from the estimated eigenvectors v_j and eigenvalues λ_j of the signal's correlation matrix. The smallest of these eigenvalues is used in conjunction with the threshold parameter `p(2)` to affect the dimension of the noise subspace in some cases.

The length n of the eigenvectors computed by `pmusic` is the sum of the dimensions of the signal and noise subspaces. This eigenvector

length depends on your input (signal data or correlation matrix) and the syntax you use.

The following table summarizes the dependency of the eigenvector length on the input argument.

Eigenvector Length Depending on Input Data and Syntax

Form of Input Data x	Comments on the Syntax	Length n of Eigenvectors
Row or column vector	`nwin` is specified as a scalar integer.	`nwin`
Row or column vector	`nwin` is specified as a vector.	`length(nwin)`
Row or column vector	`nwin` is not specified.	`2*p(1)`
l-by-*m* matrix	If `nwin` is specified as a scalar, it is not used. If `nwin` is specified as a vector, `length(nwin)` must equal *m*.	*m*
m-by-*m* nonnegative definite matrix	The string `'corr'` is specified and `nwin` is not used.	*m*

You should specify `nwin > p(1)` or `length(nwin) > p(1)` if you want `p(2) > 1` to have any effect.

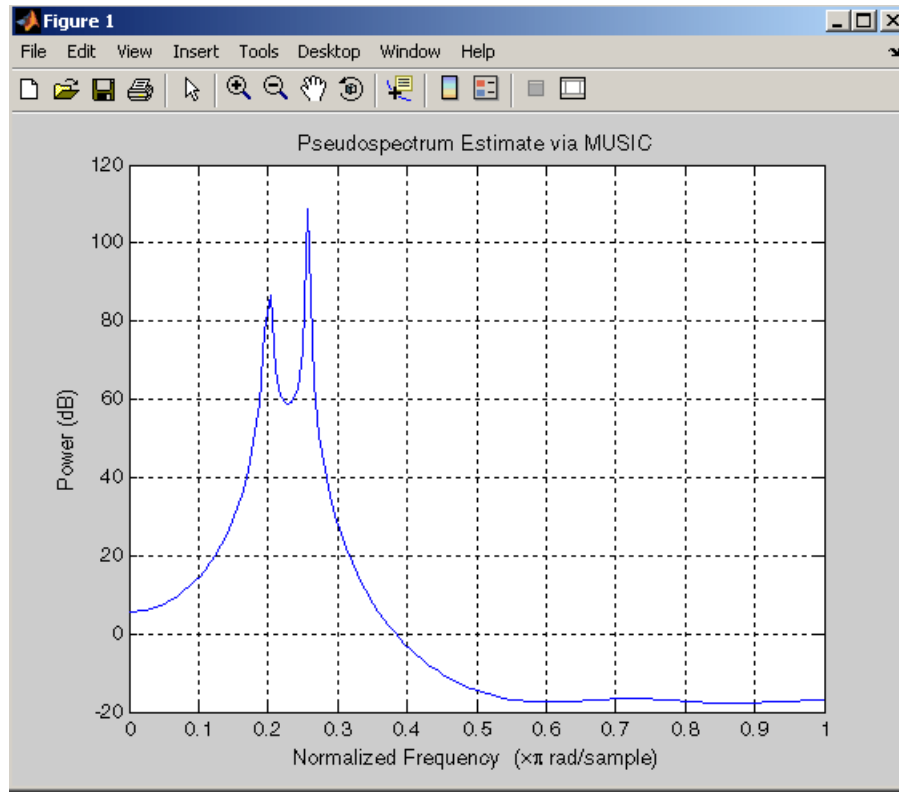
Examples

Example 1: pmusic with no Sampling Specified

This example analyzes a signal vector `x`, assuming that two real sinusoidal components are present in the signal subspace. In this case, the dimension of the signal subspace is 4 because each real sinusoid is the sum of two complex exponentials:

```
randn('state',0);
```

```
n = 0:199;
x = cos(0.257*pi*n) + sin(0.2*pi*n) + 0.01*randn(size(n));
pmusic(x,4)      % Set p to 4 because two real inputs
```



Example 2: Specifying Sampling Frequency and Subspace Dimensions

This example analyzes the same signal vector `x` with an eigenvalue cutoff of 10% above the minimum. Setting `p(1) = Inf` forces the signal/noise subspace decision to be based on the threshold parameter `p(2)`. Specify the eigenvectors of length 7 using the `nwin` argument, and set the sampling frequency `fs` to 8 kHz:

```
randn('state',0);
n = 0:199;
x = cos(0.257*pi*n) + sin(0.2*pi*n) + 0.01*randn(size(n));
[P,f] = pmusic(x,[Inf,1.1],[],8000,7); % Window length = 7
```

Example 3: Entering a Correlation Matrix

Supply a positive definite correlation matrix `R` for estimating the spectral density. Use the default 256 samples:

```
R = toeplitz(cos(0.1*pi*[0:6])) + 0.1*eye(7);
[P,f] = pmusic(R,4,'corr');
```

Example 4: Entering a Signal Data Matrix Generated from corrmtx

Enter a signal data matrix `Xm` generated from data using `corrmtx`:

```
randn('state',0);
n = 0:699;
x = cos(0.257*pi*(n)) + 0.1*randn(size(n));
Xm = corrmtx(x,7,'mod');
[P,w] = pmusic(Xm,2);
```

Example 5: Using Windowing to Create the Effect of a Signal Data Matrix

Use the same signal, but let `pmusic` form the 100-by-7 data matrix using its windowing input arguments. In addition, specify an FFT of length 512:

```
randn('state',0);
n = 0:699;
x = cos(0.257*pi*(n)) + 0.1*randn(size(n));
[PP,ff] = pmusic(x,2,512,[],7,0);
```

Algorithm

The name MUSIC is an acronym for MUltiple SIgnal Classification. The MUSIC algorithm estimates the pseudospectrum from a signal or a correlation matrix using Schmidt's eigenspace analysis method [1]. The algorithm performs eigenspace analysis of the signal's correlation matrix in order to estimate the signal's frequency content. This algorithm is particularly suitable for signals that are the sum of sinusoids with additive white Gaussian noise. The eigenvalues and eigenvectors of the signal's correlation matrix are estimated if you don't supply the correlation matrix.

The MUSIC pseudospectrum estimate is given by

$$P_{music}(f) = \frac{1}{\mathbf{e}^H(f)\left(\sum_{k=p+1}^{N} \mathbf{v}_k \mathbf{v}_k^H\right)\mathbf{e}(f)} = \frac{1}{\sum_{k=p+1}^{N} \left|\mathbf{v}_k^H \mathbf{e}(f)\right|^2}$$

where N is the dimension of the eigenvectors and v_k is the k-th eigenvector of the correlation matrix. The integer p is the dimension of the signal subspace, so the eigenvectors v_k used in the sum correspond to the smallest eigenvalues and also span the noise subspace. The vector e(f) consists of complex exponentials, so the inner product

$$\mathbf{v}_k^H \mathbf{e}(f)$$

amounts to a Fourier transform. This is used for computation of the pseudospectrum estimate. The FFT is computed for each v_k and then the squared magnitudes are summed.

References

[1] Marple, S.L. *Digital Spectral Analysis*, Englewood Cliffs, NJ, Prentice-Hall, 1987, pp. 373-378.

[2] Schmidt, R.O, "Multiple Emitter Location and Signal Parameter Estimation," *IEEE Trans. Antennas Propagation, Vol. AP-34* (March 1986), pp. 276-280.

[3] Stoica, P., and R.L. Moses, *Introduction to Spectral Analysis*, Prentice-Hall, Englewood Cliffs, NJ, 1997.

See Also

corrmtx, dspdata, pburg, peig, periodogram, pmtm, prony, pwelch, rooteig, rootmusic, spectrum

poly2ac

Purpose

Convert prediction filter polynomial to autocorrelation sequence

Syntax

```
r = poly2ac(a,efinal)
```

Description

`r = poly2ac(a,efinal)` finds the autocorrelation vector `r` corresponding to the prediction filter polynomial `a`. The autocorrelation sequence produced is approximately the same as that of the output of the autoregressive prediction filter whose coefficients are determined by `a`. `poly2ac` also produces the final `length(r)` step prediction error `efinal`. If `a(1)` is not equal to `1`, poly2ac normalizes the prediction filter polynomial by `a(1)`. `a(1)` cannot be 0.

Remarks

You can apply this function to both real and complex polynomials.

Examples

```
a = [1.0000 0.6147 0.9898 0.0004 0.0034 -0.0077];
efinal = 0.2;
r = poly2ac(a,efinal)
r =
    5.5917
   -1.7277
   -4.4231
    4.3985
    1.6426
   -5.3126
```

References

[1] Kay, S.M. *Modern Spectral Estimation*, Englewood Cliffs, NJ, Prentice-Hall, 1988

See Also

`ac2poly`, `poly2rc`, `rc2ac`

Purpose

Convert prediction filter coefficients to line spectral frequencies

Syntax

```
lsf = poly2lsf(a)
```

Description

`lsf = poly2lsf(a)` returns a vector `lsf` of line spectral frequencies from a vector `a` of prediction filter coefficients.

Examples

```
a = [1.0000  0.6149  0.9899  0.0000  0.0031 -0.0082];
lsf = poly2lsf(a)
lsf =
    0.7842
    1.5605
    1.8776
    1.8984
    2.3593
```

References

[1] Deller, J.R., J.G. Proakis, and J.H.L. Hansen, *Discrete-Time Processing of Speech Signals*, Prentice-Hall, 1993.

[2] Rabiner, L.R., and R.W. Schafer, *Digital Processing of Speech Signals*, Prentice-Hall, 1978.

See Also

`lsf2poly`

poly2rc

Purpose

Convert prediction filter polynomial to reflection coefficients

Syntax

```
k = poly2rc(a)
[k,r0] = poly2rc(a,efinal)
```

Description

`k = poly2rc(a)` converts the prediction filter polynomial `a` to the reflection coefficients of the corresponding lattice structure. `a` can be real or complex, and `a(1)` cannot be 0. If `a(1)` is not equal to `1`, poly2rc normalizes the prediction filter polynomial by `a(1)`. `k` is a row vector of size `length(a)-1`.

`[k,r0] = poly2rc(a,efinal)` returns the zero-lag autocorrelation, `r0`, based on the final prediction error, `efinal`.

A simple, fast way to check if `a` has all of its roots inside the unit circle is to check if each of the elements of `k` has magnitude less than 1.

```
stable = all(abs(poly2rc(a))<1)
```

Examples

```
a = [1.0000   0.6149   0.9899   0.0000   0.0031  -0.0082];
efinal = 0.2;
[k,r0] = poly2rc(a,efinal)
k =
    0.3090
    0.9801
    0.0031
    0.0081
   -0.0082
r0 =
    5.6032
```

Limitations

If `abs(k(i)) == 1` for any `i`, finding the reflection coefficients is an ill-conditioned problem. `poly2rc` returns some `NaN`s and provide a warning message in this case.

Algorithm

`poly2rc` implements this recursive relationship:

$$k(n) = a_n(n)$$

$$a_{n-1}(m) = \frac{a_n(m) - k(n)a_n(n-m)}{1-k(n)^2}, \quad m = 1, 2, \ldots, n-1$$

This relationship is based on Levinson's recursion [1]. To implement it, `poly2rc` loops through `a` in reverse order after discarding its first element. For each loop iteration `i`, the function:

1. Sets `k(i)` equal to `a(i)`
2. Applies the second relationship above to elements 1 through `i` of the vector `a`.

```
a = (a-k(i)*fliplr(a))/(1-k(i)^2);
```

References

[1] Kay, S.M. *Modern Spectral Estimation*, Englewood Cliffs, NJ, Prentice-Hall, 1988.

See Also

`ac2rc`, `latc2tf`, `latcfilt`, `poly2ac`, `rc2poly`, `tf2latc`

polyscale

Purpose

Scale roots of polynomial

Syntax

```
b = polyscale(a,alpha)
```

Description

`b = polyscale(a,alpha)` scales the roots of a polynomial in the *z*-plane, where `a` is a vector containing the polynomial coefficients and `alpha` is the scaling factor.

If `alpha` is a real value in the range `[0 1]`, then the roots of `a` are radially scaled toward the origin in the *z*-plane. Complex values for `alpha` allow arbitrary changes to the root locations.

Remark

By reducing the radius of the roots in an autoregressive polynomial, the bandwidth of the spectral peaks in the frequency response is expanded (flattened). This operation is often referred to as *bandwidth expansion*.

See Also

`polystab`, `roots`

Purpose

Stabilize polynomial

Syntax

```
b = polystab(a)
```

Description

`polystab` stabilizes a polynomial with respect to the unit circle; it reflects roots with magnitudes greater than 1 inside the unit circle.

`b = polystab(a)` returns a row vector `b` containing the stabilized polynomial, where `a` is a vector of polynomial coefficients, normally in the *z*-domain.

$$a(z) = a(1) + a(2)z^{-1} + \cdots + a(m+1)z^{-m}$$

Examples

`polystab` can convert a linear-phase filter into a minimum-phase filter with the same magnitude response:

```
h = fir1(25,0.4);
hmin = polystab(h) * norm(h)/norm(polystab(h));
```

Algorithm

`polystab` finds the roots of the polynomial and maps those roots found outside the unit circle to the inside of the unit circle:

```
v = roots(a);
vs = 0.5*(sign(abs(v)-1)+1);
v = (1-vs).*v + vs./conj(v);
b = a(1)*poly(v);
```

See Also

`roots`

pow2db

Purpose

Convert power to decibels (dB)

Syntax

```
ydb = pow2db(y)
```

Description

`ydb = pow2db(y)` returns the corresponding decibel (dB) value `ydb` for a given power value `y`. The relationship between power and decibels is `ydb` = $10*\log_{10}$(`y`).

See Also

`db2pow`

Purpose

Prony's method for time domain IIR filter design

Syntax

```
[b,a] = prony(h,n,m)
```

Description

Prony's method is an algorithm for finding an IIR filter with a prescribed time domain impulse response. It has applications in filter design, exponential signal modeling, and system identification (parametric modeling).

`[b,a] = prony(h,n,m)` finds a filter with numerator order `n`, denominator order `m`, and the time domain impulse response in `h`. If the length of `h` is less than the largest order (`n` or `m`), `h` is padded with zeros. `prony` returns the filter coefficients in row vectors `b` and `a`, of length `n + 1` and `m + 1`, respectively. The filter coefficients are in descending powers of z.

$$H(z) = \frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \cdots + b(n+1)z^{-n}}{a(1) + a(2)z^{-1} + \cdots + a(m+1)z^{-m}}$$

Examples

Recover the coefficients of a Butterworth filter from its impulse response:

```
[b,a] = butter(4,0.2)
b =
    0.0048    0.0193    0.0289    0.0193    0.0048
a =
    1.0000   -2.3695    2.3140   -1.0547    0.1874
h = filter(b,a,[1 zeros(1,25)]);
[bb,aa] = prony(h,4,4)
bb =
    0.0048    0.0193    0.0289    0.0193    0.0048
aa =
    1.0000   -2.3695    2.3140   -1.0547    0.1874
```

Algorithm

`prony` implements the method described in [1]. This method uses a variation of the covariance method of AR modeling to find the denominator coefficients `a` and then finds the numerator coefficients `b`

for which the impulse response of the output filter matches exactly the first `n + 1` samples of `x`. The filter is not necessarily stable, but potentially can recover the coefficients exactly if the data sequence is truly an autoregressive moving-average (ARMA) process of the correct order.

References

[1] Parks, T.W., and C.S. Burrus, *Digital Filter Design*, John Wiley & Sons, 1987, pp. 226-228.

See Also

`butter`, `cheby1`, `cheby2`, `ellip`, `invfreqz`, `levinson`, `lpc`, `stmcb`

Purpose

Pulse train

Syntax

```
pulstran
y = pulstran(t,d,'func')
pulstran(t,d,'func',p1,p2,...)
pulstran(t,d,p,fs)
pulstran(t,d,p)
pulstran(...,'func')
```

Description

`pulstran` generates pulse trains from continuous functions or sampled prototype pulses.

`y = pulstran(t,d,'func')` generates a pulse train based on samples of a continuous function, *`'func'`*, where *`'func'`* is

- `'gauspuls'`, for generating a Gaussian-modulated sinusoidal pulse
- `'rectpuls'`, for generating a sampled aperiodic rectangle
- `'tripuls'`, for generating a sampled aperiodic triangle

`pulstran` is evaluated `length(d)` times and returns the sum of the evaluations `y = func(t-d(1)) + func(t-d(2)) + ...`

The function is evaluated over the range of argument values specified in array `t`, after removing a scalar argument offset taken from the vector `d`. Note that *`func`* must be a vectorized function that can take an array `t` as an argument.

An optional gain factor may be applied to each delayed evaluation by specifying `d` as a two-column matrix, with the offset defined in column 1 and associated gain in column 2 of `d`. Note that a row vector will be interpreted as specifying delays only.

`pulstran(t,d,'func',p1,p2,...)` allows additional parameters to be passed to *`'func'`* as necessary. For example:

```
func(t-d(1),p1,p2,...) + func(t-d(2),p1,p2,...) + ...
```

`pulstran(t,d,p,fs)` generates a pulse train that is the sum of multiple delayed interpolations of the prototype pulse in vector `p`, sampled at the rate `fs`, where `p` spans the time interval `[0,(length(p)-1)/fs]`, and its samples are identically 0 outside this interval. By default, linear interpolation is used for generating delays.

`pulstran(t,d,p)` assumes that the sampling rate `fs` is equal to 1 Hz.

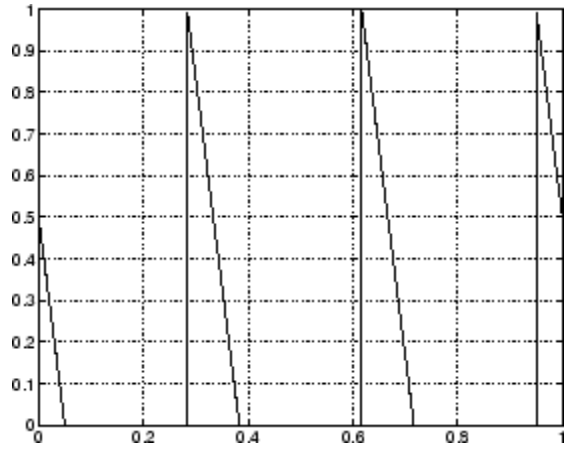
`pulstran(...,'func')` specifies alternative interpolation methods. See `interp1` for a list of available methods.

Examples

Example 1

This example generates an asymmetric sawtooth waveform with a repetition frequency of 3 Hz and a sawtooth width of 0.1s. It has a signal length of 1s and a 1 kHz sample rate:

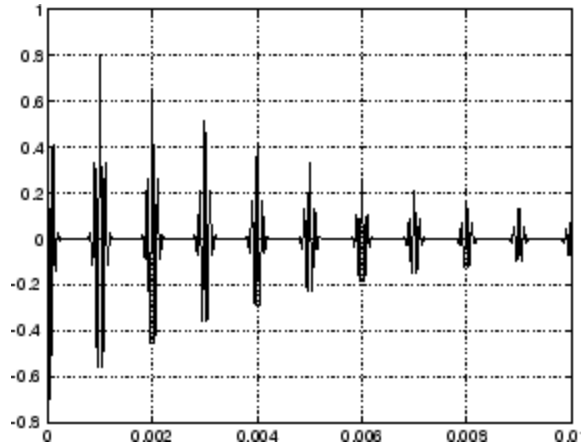
```
t = 0 : 1/1e3 : 1;          % 1 kHz sample freq for 1 sec
d = 0 : 1/3 : 1;            % 3 Hz repetition freq
y = pulstran(t,d,'tripuls',0.1,-1);
plot(t,y)
```



Example 2

This example generates a periodic Gaussian pulse signal at 10 kHz, with 50% bandwidth. The pulse repetition frequency is 1 kHz, sample rate is 50 kHz, and pulse train length is 10 msec. The repetition amplitude should attenuate by 0.8 each time:

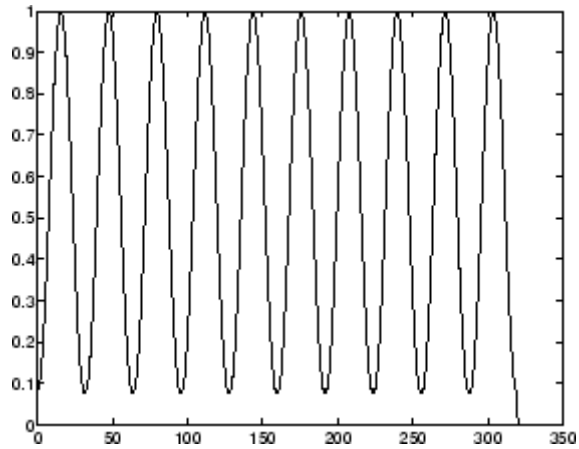
```
t = 0 : 1/50E3 : 10e-3;
d = [0 : 1/1E3 : 10e-3 ; 0.8.^(0:10)]';
y = pulstran(t,d,'gauspuls',10e3,0.5);
plot(t,y)
```



Example 3

This example generates a train of 10 Hamming windows:

```
p = hamming(32);
t = 0:320; d = (0:9)'*32;
y = pulstran(t,d,p);
plot(t,y)
```



See Also

`chirp`, `cos`, `diric`, `gauspuls`, `rectpuls`, `sawtooth`, `sin`, `sinc`, `square`, `tripuls`

Purpose

PSD using Welch's method

Syntax

```
[Pxx,w] = pwelch(x)
[Pxx,w] = pwelch(x,window)
[Pxx,w] = pwelch(x,window,noverlap)
[Pxx,w] = pwelch(x,window,noverlap,nfft)
[Pxx,w] = pwelch(x,window,noverlap,w)
[Pxx,f] = pwelch(x,window,noverlap,nfft,fs)
[Pxx,f] = pwelch(x,window,noverlap,f,fs)
[...] = pwelch(x,window,noverlap,...,'range')
pwelch(x,...)
```

Description

`[Pxx,w] = pwelch(x)` estimates the power spectral density `Pxx` of the input signal vector `x` using Welch's averaged modified periodogram method of spectral estimation. With this syntax:

- The vector `x` is segmented into eight sections of equal length, each with 50% overlap.
- Any remaining (trailing) entries in `x` that cannot be included in the eight segments of equal length are discarded.
- Each segment is windowed with a Hamming window (see `hamming`) that is the same length as the segment.

The power spectral density is calculated in units of power per radians per sample. The corresponding vector of frequencies w is computed in radians per sample, and has the same length as `Pxx`.

A real-valued input vector `x` produces a full power one-sided (in frequency) PSD (by default), while a complex-valued `x` produces a two-sided PSD.

In general, the length N of the FFT and the values of the input `x` determine the length of `Pxx` and the range of the corresponding normalized frequencies. For this syntax, the (default) length N of the FFT is the larger of 256 and the next power of 2 greater than the length of the segment. The following table indicates the length of `Pxx` and the range of the corresponding normalized frequencies for this syntax.

PSD Vector Characteristics for an FFT Length of N (Default)

Real/Complex Input Data	Length of Pxx	Range of the Corresponding Normalized Frequencies
Real-valued	(*N*/2) +1	[0, π]
Complex-valued	*N*	[0, 2π)

`[Pxx,w] = pwelch(x,window)` calculates the modified periodogram using either:

- The window length `window` for the Hamming window when `window` is a positive integer
- The window weights specified in `window` when `window` is a vector

With this syntax, the input vector `x` is divided into an integer number of segments with 50% overlap, and each segment is the same length as the window. Entries in `x` that are left over after it is divided into segments are discarded. If you specify `window` as the empty vector `[]`, then the signal data is divided into eight segments, and a Hamming window is used on each one.

`[Pxx,w] = pwelch(x,window,noverlap)` divides `x` into segments according to `window`, and uses the integer `noverlap` to specify the number of signal samples (elements of `x`) that are common to two adjacent segments. `noverlap` must be less than the length of the window you specify. If you specify `noverlap` as the empty vector `[]`, then `pwelch` determines the segments of `x` so that there is 50% overlap (default).

`[Pxx,w] = pwelch(x,window,noverlap,nfft)` uses Welch's method to estimate the PSD while specifying the length of the FFT with the integer `nfft`. If you set `nfft` to the empty vector `[]`, it uses the default value for *N* listed in the previous syntax. The window size must be greater than or equal to `nfft`.

The length of Pxx and the frequency range for w depend on nfft and the values of the input x. The following table indicates the length of Pxx and the frequency range for w for this syntax.

PSD and Frequency Vector Characteristics

Real/Complex Input Data	nfft Even/Odd	Length of Pxx	Range of w
Real-valued	Even	(nfft/2 + 1)	[0, π]
Real-valued	Odd	(nfft + 1)/2	[0, π)
Complex-valued	Even or odd	nfft	[0, 2π)

[Pxx,w] = pwelch(x,window,noverlap,w) estimates the two-sided PSD at the normalized frequencies specified in the vector w using the Goertzel algorithm. The frequencies of w are rounded to the nearest DFT bin commensurate with the resolution of the signal. The units of w are rad/sample.

[Pxx,f] = pwelch(x,window,noverlap,nfft,fs) uses the sampling frequency fs specified in hertz (Hz) to compute the PSD vector (Pxx) and the corresponding vector of frequencies (f). In this case, the units for the frequency vector are in Hz. The spectral density produced is calculated in units of power per Hz. If you specify fs as the empty vector [], the sampling frequency defaults to 1 Hz.

The frequency range for f depends on nfft, fs, and the values of the input x. The length of Pxx is the same as in the PSD and Frequency Vector Characteristics on page 10-709 above. The following table indicates the frequency range for f for this syntax.

PSD and Frequency Vector Characteristics with fs Specified

Real/Complex Input Data	nfft Even/Odd	Range of f
Real-valued	Even	[0,fs/2]

PSD and Frequency Vector Characteristics with fs Specified (Continued)

Real/Complex Input Data	nfft Even/Odd	Range of f
Real-valued	Odd	`[0,fs/2)`
Complex-valued	Even or odd	`[0,fs)`

`[Pxx,f] = pwelch(x,window,noverlap,f,fs)` estimates the two-sided PSD at the normalized frequencies specified in the vector `f` using the Goertzel algorithm. The `f` vector returned is the same vector as the input `f` vector. The frequencies of `f` are rounded to the nearest DFT bin commensurate with the resolution of the signal.

`[...] = pwelch(x,window,noverlap,...,'range')` specifies the range of frequency values. This syntax is useful when `x` is real. The string *`'range'`* can be either:

- `'twosided'`: Compute the two-sided PSD over the frequency range `[0,fs)`. This is the default for determining the frequency range for complex-valued `x`.
  - If you specify `fs` as the empty vector, `[]`, the frequency range is `[0,1)`.
  - If you don't specify `fs`, the frequency range is [0, 2π).
- `'onesided'`: Compute the one-sided PSD over the frequency ranges specified for real `x`. This is the default for determining the frequency range for real-valued `x`.

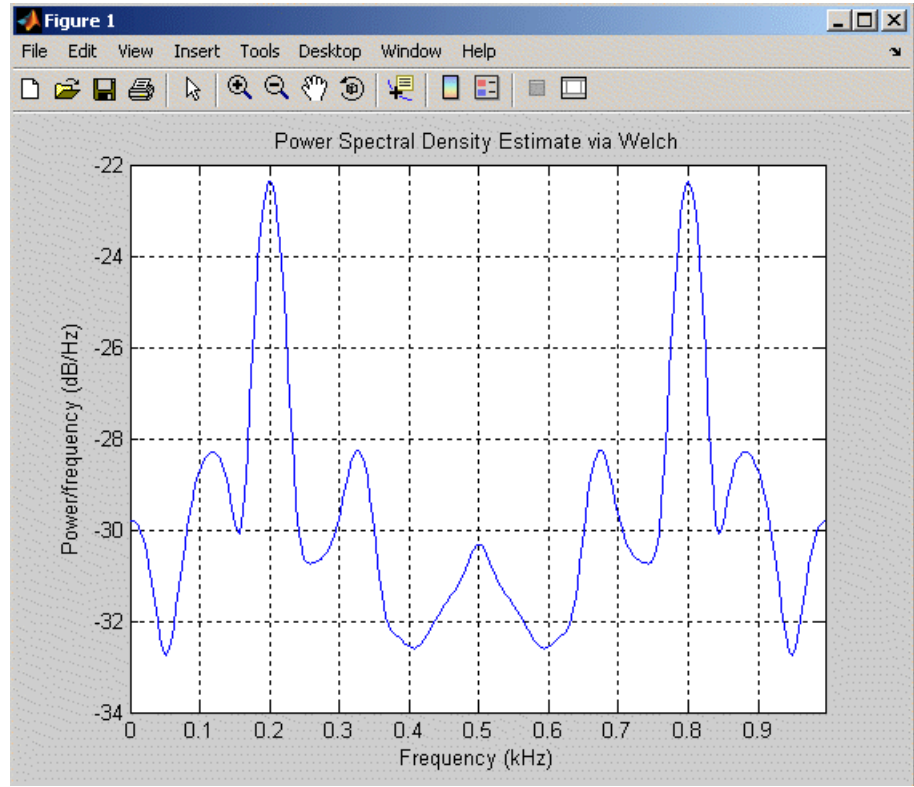
The string *`'range'`* can appear anywhere in the syntax after `noverlap`.

`pwelch(x,...)` with no output arguments plots the PSD estimate in dB per unit frequency in the current figure window.

Examples

Estimate the PSD of a signal composed of a sinusoid plus noise, sampled at 1000 Hz. Use 33-sample windows with 32-sample overlap, and the default FFT length, and display the two-sided PSD estimate:

```
randn('state',0);
Fs = 1000;   t = 0:1/Fs:.3;

% 200Hz cosine + noise
x = cos(2*pi*t*200) + randn(size(t));

pwelch(x,33,32,[],Fs,'twosided')
```



Algorithm

`pwelch` calculates the power spectral density using Welch's method (see References below):

1. The input signal vector `x` is divided into k overlapping segments according to `window` and `noverlap` (or their default values). If the window size is larger than the number of FFT points (`NFFT`) , the signal is divided into `NFFT`–length segments and then, the last segment is padded with zeros.
2. The specified (or default) window is applied to each segment of `x`. (No preprocessing is done before applying the window to each segment.)
3. An `nfft`-point FFT is applied to the windowed data.
4. The (modified) periodogram of each windowed segment is computed.
5. The set of modified periodograms is averaged to form the spectrum estimate $S(e^{j\omega})$.
6. The resulting spectrum estimate is scaled to compute the power spectral density as $S(e^{j\omega})/F$, where F is
   - 2π when you do not supply the sampling frequency
   - `fs` when you supply the sampling frequency

The number of segments k that `x` is divided into is calculated as:

- Eight if you don't specify `window`, or if you specify it as the empty vector `[]`
- $k = \frac{m-o}{l-o}$ if you specify window as a nonempty vector or a scalar

  In this equation, m is the length of the signal vector `x`, o is the number of overlapping samples (`noverlap`), and l is the length of each segment (the window length).

References

[1] Hayes, M., *Statistical Digital Signal Processing and Modeling*, John Wiley & Sons, 1996.

[2] Stoica, P., and R.L. Moses, *Introduction to Spectral Analysis*, Prentice-Hall, Englewood Cliffs, NJ, 1997, pp. 52-54.

[3] Welch, P.D, "The Use of Fast Fourier Transform for the Estimation of Power Spectra: A Method Based on Time Averaging Over Short, Modified Periodograms," *IEEE Trans. Audio Electroacoustics, Vol. AU-15 (June 1967)*, pp. 70-73.

See Also

`dspdata.msspectrum`, `pburg`, `pcov`, `peig`, `periodogram`, `pmcov`, `pmtm`, `pmusic`, `pyulear`

Purpose

PSD using Yule-Walker AR method

Syntax

```
Pxx = pyulear(x,p)
Pxx = pyulear(x,p,nfft)
[Pxx,w] = pyulear(...)
[Pxx,w] = pyulear(x,p,w)
Pxx = pyulear(x,p,nfft,fs)
Pxx = pyulear(x,p,f,fs)
[Pxx,f] = pyulear(x,p,nfft,fs)
[Pxx,f] = pyulear(x,p,f,fs)
[Pxx,f] = pyulear(x,p,nfft,fs,'range')
[Pxx,w] = pyulear(x,p,nfft,'range')
pyulear(...)
```

Description

`Pxx = pyulear(x,p)` implements the Yule-Walker algorithm, a parametric spectral estimation method, and returns `Pxx`, an estimate of the power spectral density (PSD) of the vector `x`. The entries of `x` represent samples of a discrete-time signal. `p` is the integer specifying the order of an autoregressive (AR) prediction model for the signal, used in estimating the PSD. This estimate is also an estimate of the maximum entropy.

The power spectral density is calculated in units of power per radians per sample. Real-valued inputs produce full power one-sided (in frequency) PSDs (by default), while complex-valued inputs produce two-sided PSDs.

In general, the length of the FFT and the values of the input `x` determine the length of `Pxx` and the range of the corresponding normalized frequencies. For this syntax, the (default) FFT length is 256. The following table indicates the length of `Pxx` and the range of the corresponding normalized frequencies for this syntax.

PSD Vector Characteristics for an FFT Length of 256 (Default)

Real/Complex Input Data	Length of Pxx	Range of the Corresponding Normalized Frequencies
Real-valued	129	[0, π]
Complex-valued	256	[0, 2π)

`Pxx = pyulear(x,p,nfft)` uses the integer FFT length `nfft` to calculate the PSD vector `Pxx`.

`[Pxx,w] = pyulear(...)` also returns `w`, a vector of normalized angular frequencies at which the two-sided PSD is estimated. `Pxx` and `w` have the same length. The units for `w` are rad/sample.

The length of `Pxx` and the frequency range for `w` depend on `nfft` and the values of the input `x`. The following table indicates the length of `Pxx` and the frequency range for `w` in this syntax.

PSD and Frequency Vector Characteristics

Real/Complex Input Data	nfft Even/Odd	Length of Pxx	Range of w
Real-valued	Even	`(nfft/2 + 1)`	[0, π]
Real-valued	Odd	`(nfft + 1)/2`	[0, π)
Complex-valued	Even or odd	`nfft`	[0, 2π)

`[Pxx,w] = pyulear(x,p,w)` uses a vector of normalized frequencies `w` with two or more elements to compute the PSD at those frequencies and returns a two-sided PSD.

```
Pxx = pyulear(x,p,nfft,fs)
```

or

`Pxx = pyulear(x,p,f,fs)` uses the integer FFT length `nfft` to calculate the PSD vector `Pxx` or uses the vector of frequencies `f` in Hz and the sampling frequency `fs` to compute the two-sided PSD vector `Pxx` at those frequencies. If you specify `nfft` as the empty vector `[]`, it uses the default value of 256. If you specify `fs` as the empty vector `[]`, the sampling frequency `fs` defaults to 1 Hz. The spectral density produced is calculated in units of power per Hz.

`[Pxx,f] = pyulear(x,p,nfft,fs)`

or

`[Pxx,f] = pyulear(x,p,f,fs)` returns the frequency vector f. In this case, the units for the frequency vector are in Hz. The frequency range for `f` depends on `nfft`, `fs`, and the values of the input `x`. The length of `Pxx` is the same as in the table above. The following table indicates the frequency range for `f` for this syntax.

PSD and Frequency Vector Characteristics with fs Specified

Real/Complex Input Data	nfft Even/Odd	Range of f
Real-valued	Even	`[0,fs/2]`
Real-valued	Odd	`[0,fs/2)`
Complex-valued	Even or odd	`[0,fs)`

`[Pxx,f] = pyulear(x,p,nfft,fs,'range')` or

`[Pxx,w] = pyulear(x,p,nfft,'range')` specifies the range of frequency values to include in `f` or `w`. This syntax is useful when `x` is real. *`'range'`* can be either:

- `'twosided'`: Compute the two-sided PSD over the frequency range `[0,fs)`. This is the default for determining the frequency range for complex-valued `x`.
  - If you specify `fs` as the empty vector, `[]`, the frequency range is `[0,1)`.

  - If you don't specify `fs`, the frequency range is [0, 2π).
- `'onesided'`: Compute the one-sided PSD over the frequency ranges specified for real `x`. This is the default for determining the frequency range for real-valued `x`. Note that `'onesided'` is not valid if you pass in a vector of frequencies (`f` or `w`).

Note You can put the string argument *`'range'`* anywhere in the input argument list after `p`.

`pyulear(...)` plots the power spectral density in the current figure window. The frequency range on the plot is the same as the range of output `w` (or `f`) for a given set of parameters.

Remarks

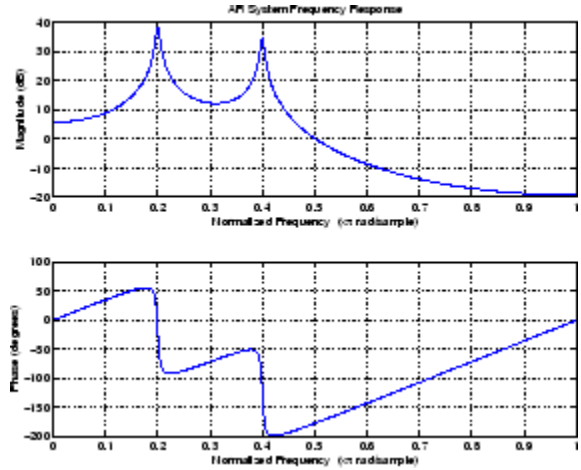
The power spectral density is computed as the distribution of power per unit frequency.

This algorithm depends on your selecting an appropriate model order for your signal.

Examples

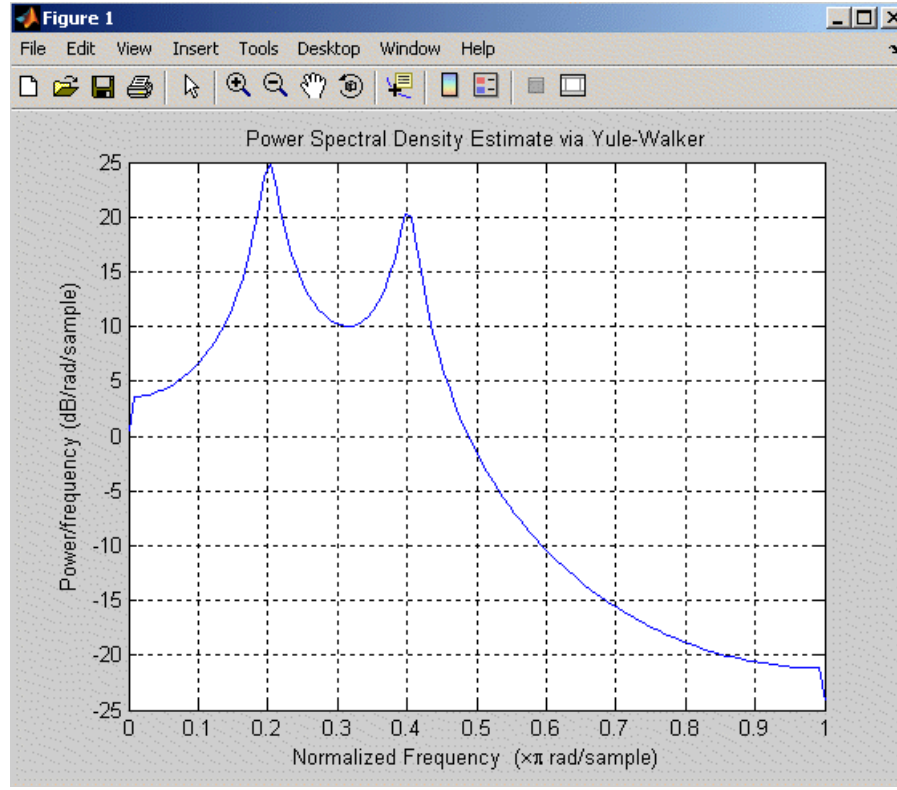
Because the Yule-walker method estimates the spectral density by fitting an AR prediction model of a given order to the signal, first generate a signal from an AR (all-pole) model of a given order. You can use `freqz` to check the magnitude of the frequency response of your AR filter. This will give you an idea of what to expect when you estimate the PSD using `pyulear`:

```
% AR filter coefficients
a = [1 -2.2137 2.9403 -2.1697 0.9606];

% AR filter frequency response
freqz(1,a)
title('AR System Frequency Response')
```



Now generate the input signal `x` by filtering white noise through the AR filter. Estimate the PSD of `x` based on a fourth-order AR prediction model, since in this case, we know that the original AR system model `a` has order 4:

```
randn('state',1);
x = filter(1,a,randn(256,1));    % AR system output
pyulear(x,4)                     % Fourth-order estimate
```



Algorithm

Linear prediction filters can be used to model the second-order statistical characteristics of a signal. The prediction filter output can be used to model the signal when the input is white noise.

`pyulear` estimates the PSD of an input signal vector using the Yule-Walker AR method. This method, also called the autocorrelation or windowed method, fits an autoregressive (AR) linear prediction filter model to the signal by minimizing the forward prediction error (based on all observations of the in put sequence) in the least squares sense. This formulation leads to the Yule-Walker equations, which are solved by the

Levinson-Durbin recursion. The spectral estimate returned by `pyulear` is the squared magnitude of the frequency response of this AR model.

References

[1] Marple, S.L., *Digital Spectral Analysis*, Prentice-Hall, 1987, Chapter 7.

[2] Stoica, P., and R.L. Moses, *Introduction to Spectral Analysis*, Prentice-Hall, 1997.

See Also

`aryule`, `lpc`, `pburg`, `pcov`, `peig`, `periodogram`, `pmcov`, `pmtm`, `pmusic`, `prony`, `pwelch`

Purpose

Convert reflection coefficients to autocorrelation sequence

Syntax

```
r = rc2ac(k,r0)
```

Description

`r = rc2ac(k,r0)` finds the autocorrelation coefficients, `r`, of the output of the discrete-time prediction error filter from the lattice-form reflection coefficients `k` and initial zero-lag autocorrelation `r0`.

Examples

```
k = [0.3090    0.9800    0.0031    0.0082   -0.0082];
r0 = 0.1;
a = rc2ac(k,r0)
a =
    0.1000
   -0.0309
   -0.0791
    0.0787
    0.0294
   -0.0950
```

References

[1] Kay, S.M., *Modern Spectral Estimation*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

See Also

`ac2rc`, `poly2ac`, `rc2poly`

rc2is

Purpose

Convert reflection coefficients to inverse sine parameters

Syntax

```
isin = is2rc(k)
```

Description

`isin = is2rc(k)` returns a vector of inverse sine parameters `isin` from a vector of reflection coefficients `k`.

Examples

```
k = [0.3090 0.9801 0.0031 0.0082 -0.0082];
isin = rc2is(k)
isin =
    0.2000    0.8728    0.0020    0.0052   -0.0052
```

References

[1] Deller, J.R., J.G. Proakis, and J.H.L. Hansen, *Discrete-Time Processing of Speech Signals*, Prentice-Hall, 1993.

See Also

`is2rc`

Purpose

Convert reflection coefficients to log area ratio parameters

Syntax

```
g = rc2lar(k)
```

Description

`g = rc2lar(k)` returns a vector of log area ratio parameters `g` from a vector of reflection coefficients `k`.

Examples

```
k = [0.3090 0.9801 0.0031 0.0082 -0.0082];
g = rc2lar(k)

g =
    0.6389    4.6002    0.0062    0.0164   -0.0164
```

References

[1] Deller, J.R., J.G. Proakis, and J.H.L. Hansen, *Discrete-Time Processing of Speech Signals*, Prentice-Hall, 1993.

See Also

`lar2rc`

rc2poly

Purpose

Convert reflection coefficients to prediction filter polynomial

Syntax

```
a = rc2poly(k)
[a,efinal] = rc2poly(k,r0)
```

Description

`a = rc2poly(k)` converts the reflection coefficients `k` corresponding to the lattice structure to the prediction filter polynomial `a`, with `a(1) = 1`. The output `a` is row vector of length `length(k)+1`.

`[a,efinal] = rc2poly(k,r0)` returns the final prediction error `efinal` based on the zero-lag autocorrelation, `r0`.

Examples

Consider a lattice IIR filter given by reflection coefficients `k`:

```
k = [0.3090    0.9800    0.0031    0.0082   -0.0082];
```

Its equivalent prediction filter representation is given by

```
a = rc2poly(k)
a =
    1.0000    0.6148    0.9899    0.0000    0.0032   -0.0082
```

Algorithm

`rc2poly` computes output `a` using Levinson's recursion [1]. The function

1. Sets the output vector `a` to the first element of `k`.
2. Loops through the remaining elements of `k`.

   For each loop iteration `i`, `a = [a + a(i-1:-1:1)*k(i) k(i)]`.
3. Implements `a = [1 a]`.

References

[1] Kay, S.M., *Modern Spectral Estimation*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

See Also

`ac2poly`, `latc2tf`, `latcfilt`, `poly2rc`, `rc2ac`, `rc2is`, `rc2lar`, `tf2latc`

Purpose

Real cepstrum and minimum phase reconstruction

Syntax

```
rceps(x)
[y,ym] = rceps(x)
```

Description

The *real cepstrum* is the inverse Fourier transform of the real logarithm of the magnitude of the Fourier transform of a sequence.

Note `rceps` only works on real data.

`rceps(x)` returns the real cepstrum of the real sequence `x`. The real cepstrum is a real-valued function.

`[y,ym] = rceps(x)` returns both the real cepstrum `y` and a minimum phase reconstructed version `ym` of the input sequence.

Algorithm

`rceps` is an M-file implementation of algorithm 7.2 in [2], that is,

```
y = real(ifft(log(abs(fft(x)))));
```

Appropriate windowing in the cepstral domain forms the reconstructed minimum phase signal:

```
w = [1;2*ones(n/2-1,1);ones(1-rem(n,2),1);zeros(n/2-1,1)];
ym = real(ifft(exp(fft(w.*y))));
```

References

[1] Oppenheim, A.V., and R.W. Schafer, *Digital Signal Processing*, Englewood Cliffs, NJ, Prentice-Hall, 1975.

[2] *Programs for Digital Signal Processing*, IEEE Press, New York, 1979.

See Also

`cceps`, `fft`, `hilbert`, `icceps`, `unwrap`

rectpuls

Purpose

Sampled aperiodic rectangle

Syntax

```
y = rectpuls(t)
y = rectpuls(t,w)
```

Description

`y = rectpuls(t)` returns a continuous, aperiodic, unity-height rectangular pulse at the sample times indicated in array `t`, centered about `t = 0` and with a default width of 1. Note that the interval of non-zero amplitude is defined to be open on the right, that is, `rectpuls(-0.5) = 1` while `rectpuls(0.5) = 0`.

`y = rectpuls(t,w)` generates a rectangle of width `w`.

`rectpuls` is typically used in conjunction with the pulse train generating function `pulstran`.

See Also

`chirp`, `cos`, `diric`, `gauspuls`, `pulstran`, `sawtooth`, `sin`, `sinc`, `square`, `tripuls`

Purpose

Rectangular window

Syntax

```
w = rectwin(L)
```

Description

`w = rectwin(L)` returns a rectangular window of length `L` in the column vector `w`. This function is provided for completeness; a rectangular window is equivalent to no window at all.

Algorithm

```
w = ones(L,1);
```

References

[1] Oppenheim, A.V., and R.W. Schafer. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1999, pp. 468-471.

See Also

`barthannwin`, `bartlett`, `blackmanharris`, `bohmanwin`, `nuttallwin`, `parzenwin`, `triang`, `window`, `wintool`, `wvtool`

resample

Purpose

Change sampling rate by rational factor

Syntax

```
y = resample(x,p,q)
y = resample(x,p,q,n)
y = resample(x,p,q,n,beta)
y = resample(x,p,q,b)
[y,b] = resample(x,p,q)
```

Description

`y = resample(x,p,q)` resamples the sequence in vector `x` at `p/q` times the original sampling rate, using a polyphase filter implementation. `p` and `q` must be positive integers. The length of `y` is equal to `ceil(length(x)*p/q)`. If `x` is a matrix, `resample` works down the columns of `x`.

`resample` applies an anti-aliasing (lowpass) FIR filter to `x` during the resampling process. It designs the filter using `firls` with a Kaiser window.

`y = resample(x,p,q,n)` uses `n` terms on either side of the current sample, `x(k)`, to perform the resampling. The length of the FIR filter `resample` uses is proportional to `n`; larger values of `n` provide better accuracy at the expense of more computation time. The default for `n` is 10. If you let `n = 0`, resample performs a nearest-neighbor interpolation

```
y(k) = x(round((k-1)*q/p)+1)
```

where `y(k) = 0` if the index to `x` is greater than `length(x)`.

`y = resample(x,p,q,n,beta)` uses beta as the design parameter for the Kaiser window that `resample` employs in designing the lowpass filter. The default for `beta` is 5.

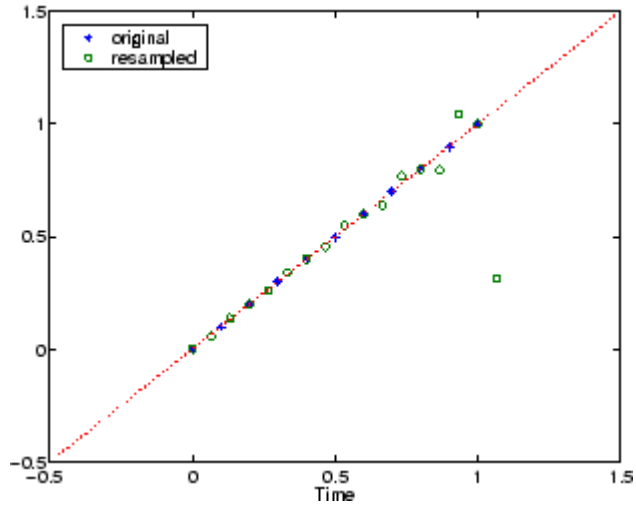
`y = resample(x,p,q,b)` filters `x` using the vector of filter coefficients `b`.

`[y,b] = resample(x,p,q)` returns the vector `b`, which contains the coefficients of the filter applied to `x` during the resampling process.

Examples

Resample a simple linear sequence at 3/2 the original rate:

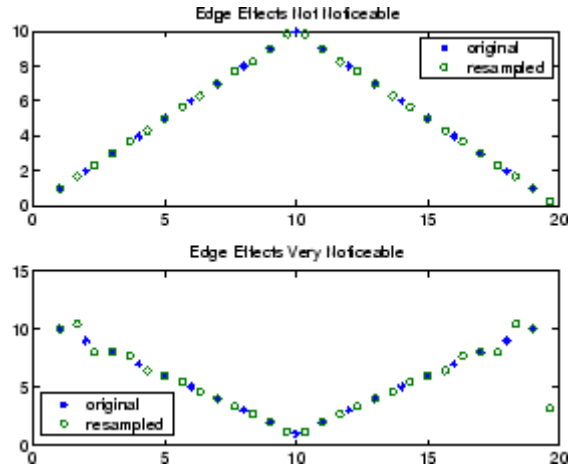
```
fs1 = 10;              % Original sampling frequency in Hz
t1 = 0:1/fs1:1;        % Time vector
x = t1;                % Define a linear sequence
y = resample(x,3,2);   % Now resample it
t2 = (0:(length(y)-1))*2/(3*fs1);  % New time vector
plot(t1,x,'*',t2,y,'o',-0.5:0.01:1.5,-0.5:0.01:1.5,':')
legend('original','resampled'); xlabel('Time')
```



Notice that the last few points of the output `y` are inaccurate. In its filtering process, `resample` assumes the samples at times before and after the given samples in `x` are equal to zero. Thus large deviations from zero at the end points of the sequence `x` can cause inaccuracies in `y` at its end points. The following two plots illustrate this side effect of `resample`:

```
x = [1:10 9:-1:1]; y = resample(x,3,2);
subplot(2,1,1);
plot(1:19,x,'*',(0:28)*2/3 + 1,y,'o');
title('Edge Effects Not Noticeable');
```

```
legend('original','resampled');
x = [10:-1:1 2:10]; y = resample(x,3,2);
subplot(2,1,2);
plot(1:19,x,'*',(0:28)*2/3 + 1,y,'o')
title('Edge Effects Very Noticeable');
legend('original','resampled');
```



Algorithm

`resample` performs an FIR design using `firls`, followed by rate changing implemented with `upfirdn`.

See Also

`decimate`, `downsample`, `firls`, `interp`, `interp1`, `intfilt`, `kaiser`, `mfilt`, `spline`, `upfirdn`, `upsample`

Purpose

z-transform partial-fraction expansion

Syntax

```
[r,p,k] = residuez(b,a)
[b,a] = residuez(r,p,k)
```

Description

`residuez` converts a discrete time system, expressed as the ratio of two polynomials, to partial fraction expansion, or residue, form. It also converts the partial fraction expansion back to the original polynomial coefficients.

Note Numerically, the partial fraction expansion of a ratio of polynomials is an ill-posed problem. If the denominator polynomial is near a polynomial with multiple roots, then small changes in the data, including roundoff errors, can cause arbitrarily large changes in the resulting poles and residues. You should use state-space (or pole-zero representations instead.

`[r,p,k] = residuez(b,a)` finds the residues, poles, and direct terms of a partial fraction expansion of the ratio of two polynomials, $b(z)$ and $a(z)$. Vectors `b` and `a` specify the coefficients of the polynomials of the discrete-time system $b(z)/a(z)$ in descending powers of z.

$$b(z) = b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_m z^{-m}$$

$$a(z) = a_0 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_n z^{-n}$$

If there are no multiple roots and `a > n-1`,

$$\frac{b(z)}{a(z)} = \frac{r(1)}{1 - p(1)z^{-1}} + \cdots + \frac{r(n)}{1 - p(n)z^{-1}} + k(1) + k(2)z^{-1} + \cdots + k(m-n+1)z^{-(m-n)}$$

The returned column vector `r` contains the residues, column vector `p` contains the pole locations, and row vector `k` contains the direct terms. The number of poles is

```
n = length(a)-1 = length(r) = length(p)
```

The direct term coefficient vector `k` is empty if `length(b)` is less than `length(a)`; otherwise:

```
length(k) = length(b) - length(a) + 1
```

If `p(j) = ... = p(j+s-1)` is a pole of multiplicity `s`, then the expansion includes terms of the form

$$\frac{r(j)}{.-p(j)z^{-1}} + \frac{r(j+1)}{(1-p(j)z^{-1})^2} + \cdots + \frac{r(j+s_r-1)}{(1-p(j)z^{-1})^s}$$

`[b,a] = residuez(r,p,k)` with three input arguments and two output arguments, converts the partial fraction expansion back to polynomials with coefficients in row vectors `b` and `a`.

The `residue` function in the standard MATLAB language is very similar to `residuez`. It computes the partial fraction expansion of continuous-time systems in the Laplace domain (see reference [1]), rather than discrete-time systems in the *z*-domain as does `residuez`.

Algorithm

`residuez` applies standard MATLAB functions and partial fraction techniques to find `r`, `p`, and `k` from `b` and `a`. It finds

- The direct terms `a` using `deconv` (polynomial long division) when `length(b)` > `length(a)-1`.
- The poles using `p = roots(a)`.
- Any repeated poles, reordering the poles according to their multiplicities.
- The residue for each nonrepeating pole p_i by multiplying $b(z)/a(z)$ by $1/(1 - p_i z^{-1})$ and evaluating the resulting rational function at $z = p_i$.
- The residues for the repeated poles by solving

  ```
  S2*r2 = h - S1*r1
  ```

for `r2` using `\`. `h` is the impulse response of the reduced $b(z)/a(z)$, `S1` is a matrix whose columns are impulse responses of the first-order systems made up of the nonrepeating roots, and `r1` is a column containing the residues for the nonrepeating roots. Each column of matrix `S2` is an impulse response. For each root p_j of multiplicity s_j, `S2` contains s_j columns representing the impulse responses of each of the following systems.

$$\frac{1}{1-pjz^{-1}}, \frac{1}{(1-pjz^{-1})^2}, \cdots, \frac{1}{(1-pjz^{-1})^{s_j}}$$

The vector `h` and matrices `S1` and `S2` have `n + xtra` rows, where `n` is the total number of roots and the internal parameter `xtra`, set to 1 by default, determines the degree of over-determination of the system of equations.

References

[1] Oppenheim, A.V., and R.W. Schafer, *Digital Signal Processing,* Prentice-Hall, Englewood Cliffs, NJ, 1975, pp. 166-170.

See Also

`convmtx`, `deconv`, `poly`, `prony`, `residue`, `roots`, `ss2tf`, `tf2ss`, `tf2zp`, `tf2zpk`, `zp2ss`

Purpose

Reverse Levinson-Durbin recursion

Syntax

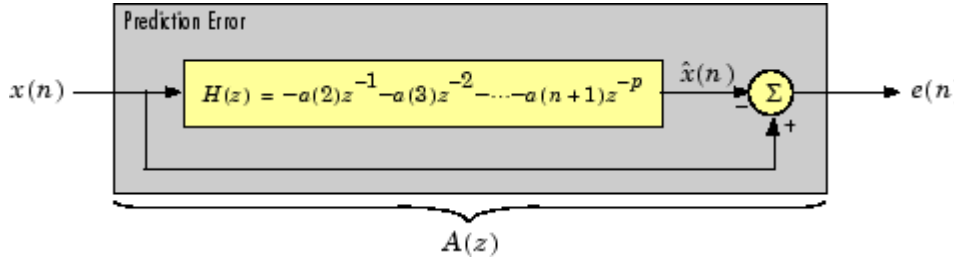
```
r = rlevinson(a,efinal)
[r,u] = rlevinson(a,efinal)
[r,u,k] = rlevinson(a,efinal)
[r,u,k,e] = rlevinson(a,efinal)
```

Description

The reverse Levinson-Durbin recursion implements the step-down algorithm for solving the following symmetric Toeplitz system of linear equations for r, where $r = [r(1)\ \mathrm{L}\ r(p+1)]$ and $r(i)^*$ denotes the complex conjugate of $r(i)$.

$$\begin{bmatrix} r(1) & r(2)^* & \cdots & r(p)^* \\ r(2) & r(1) & \cdots & r(p-1)^* \\ \vdots & \ddots & \ddots & \vdots \\ r(p) & \cdots & r(2) & r(1) \end{bmatrix} \begin{bmatrix} a(2) \\ a(3) \\ \vdots \\ a(p+1) \end{bmatrix} = \begin{bmatrix} -r(2) \\ -r(3) \\ \vdots \\ -r(p+1) \end{bmatrix}$$

`r = rlevinson(a,efinal)` solves the above system of equations for r given vector `a`, where `a = [1` $a(2)\ \mathrm{L}\ a(p+1)$`]`. In linear prediction applications, `r` represents the autocorrelation sequence of the input to the prediction error filter, where `r(1)` is the zero-lag element. The figure below shows the typical filter of this type, where $H(z)$ is the optimal linear predictor, $x(n)$ is the input signal, $\hat{x}(n)$ is the predicted signal, and $e(n)$ is the prediction error.



Input vector `a` represents the polynomial coefficients of this prediction error filter in descending powers of z.

$$A(z) = 1 + a(2)z^{-1} + \cdots + a(n+1)z^{-p}$$

The filter must be minimum phase to generate a valid autocorrelation sequence. `efinal` is the scalar prediction error power, which is equal to the variance of the prediction error signal, $\sigma^2(e)$.

`[r,u] = rlevinson(a,efinal)` returns upper triangular matrix U from the UDU^* decomposition

$$R^{-1} = UE^{-1}U^*$$

where

$$R = \begin{bmatrix} r(1) & r(2)^* & \cdots & r(p)^* \\ r(2) & r(1) & \cdots & r(p-1)^* \\ \vdots & \ddots & \ddots & \vdots \\ r(p) & \cdots & r(2) & r(1) \end{bmatrix}$$

and E is a diagonal matrix with elements returned in output `e` (see below). This decomposition permits the efficient evaluation of the inverse of the autocorrelation matrix, R^{-1}.

Output matrix `u` contains the prediction filter polynomial, `a`, from each iteration of the reverse Levinson-Durbin recursion

$$U = \begin{bmatrix} a_1(1)^* & a_2(2)^* & \cdots & a_{p+1}(p+1)^* \\ 0 & a_2(1)^* & \ddots & a_{p+1}(p)^* \\ 0 & 0 & \ddots & a_{p+1}(p-1)^* \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & a_{p+1}(1)^* \end{bmatrix}$$

where $a_i(j)$ is the jth coefficient of the ith order prediction filter polynomial (i.e., step i in the recursion). For example, the 5th order prediction filter polynomial is

```
a5 = u(5:-1:1,5)'
```

Note that `u(p+1:-1:1,p+1)'` is the input polynomial coefficient vector `a`.

`[r,u,k] = rlevinson(a,efinal)` returns a vector `k` of length (p+1) containing the reflection coefficients. The reflection coefficients are the conjugates of the values in the first row of `u`.

```
k = conj(u(1,2:end))
```

`[r,u,k,e] = rlevinson(a,efinal)` returns a vector of length p+1 containing the prediction errors from each iteration of the reverse Levinson-Durbin recursion: `e(1)` is the prediction error from the first-order model, `e(2)` is the prediction error from the second-order model, and so on.

These prediction error values form the diagonal of the matrix E in the UDU^* decomposition of R^{-1}.

$$R^{-1} = UE^{-1}U^*$$

References

[1] Kay, S.M., *Modern Spectral Estimation: Theory and Application*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

See Also

`levinson`, `lpc`, `prony`, `stmcb`

Purpose

Frequency and power content using eigenvector method

Syntax

```
[w,pow] = rooteig(x,p)
[f,pow] = rooteig(...,fs)
[w,pow] = rooteig(...,'corr')
```

Description

`[w,pow] = rooteig(x,p)` estimates the frequency content in the time samples of a signal `x`, and returns `w`, a vector of frequencies in rad/sample, and the corresponding signal power in the vector `pow` in units of power, such as volts^2. The input signal `x` is specified either as:

- A row or column vector representing one observation of the signal
- A rectangular array for which each row of `x` represents a separate observation of the signal (for example, each row is one output of an array of sensors, as in array processing), such that `x'*x` is an estimate of the correlation matrix

Note You can use the output of `corrmtx` to generate such an array `x`.

You can specify the second input argument `p` as either:

- A scalar integer. In this case, the signal subspace dimension is `p`.
- A two-element vector. In this case, `p(2)`, the second element of `p`, represents a threshold that is multiplied by λ_{min}, the smallest estimated eigenvalue of the signal's correlation matrix. Eigenvalues below the threshold λ_{min}`*p(2)` are assigned to the noise subspace. In this case, `p(1)` specifies the maximum dimension of the signal subspace.

The extra threshold parameter in the second entry in `p` provides you more flexibility and control in assigning the noise and signal subspaces.

The length of the vector `w` is the computed dimension of the signal subspace. For real-valued input data `x`, the length of the corresponding power vector `pow` is given by

```
length(pow) = 0.5*length(w)
```

For complex-valued input data `x`, `pow` and `w` have the same length.

`[f,pow] = rooteig(...,fs)` returns the vector of frequencies `f` calculated in Hz. You supply the sampling frequency `fs` in Hz. If you specify `fs` with the empty vector `[]`, the sampling frequency defaults to 1 Hz.

`[w,pow] = rooteig(...,'corr')` forces the input argument `x` to be interpreted as a correlation matrix rather than a matrix of signal data. For this syntax, you must supply a square matrix for `x`, and all of its eigenvalues must be nonnegative.

Note You can place the string `'corr'` anywhere after `p`.

Examples

Find the frequency content in a signal composed of three complex exponentials in noise. Use the modified covariance method to estimate the correlation matrix used by the eigenvector method:

```
randn('state',1); n=0:99;
s = exp(i*pi/2*n)+2*exp(i*pi/4*n)+...
  exp(i*pi/3*n)+randn(1,100);
% Estimate correlation matrix using
%  modified covariance method.
X=corrmtx(s,12,'mod');
[W,P] = rooteig(X,3)
W =
    0.7811
    1.5767
    1.0554
P =
```

```
3.9971
1.1362
1.4102
```

Algorithm

The eigenvector method used by `rooteig` is the same as that used by `peig`. The algorithm performs eigenspace analysis of the signal's correlation matrix in order to estimate the signal's frequency content.

The difference between `peig` and `rooteig` is:

- `peig` returns the pseudospectrum at all frequency samples.
- `rooteig` returns the estimated discrete frequency spectrum, along with the corresponding signal power estimates.

`rooteig` is most useful for frequency estimation of signals made up of a sum of sinusoids embedded in additive white Gaussian noise.

See Also

`corrmtx`, `peig`, `pmusic`, `powerest` method of `spectrum`, `rootmusic`, `spectrum.eigenvector`

Purpose

Frequency and power content using root MUSIC algorithm

Syntax

```
[w,pow] = rootmusic(x,p)
[f,pow] = rootmusic(...,fs)
[w,pow] = rootmusic(...,'corr')
```

Description

`[w,pow] = rootmusic(x,p)` estimates the frequency content in the time samples of a signal `x`, and returns `w`, a vector of frequencies in rad/sample, and the corresponding signal power in the vector `pow` in dB per rad/sample. The input signal `x` is specified either as:

- A row or column vector representing one observation of the signal
- A rectangular array for which each row of `x` represents a separate observation of the signal (for example, each row is one output of an array of sensors, as in array processing), such that `x'*x` is an estimate of the correlation matrix

Note You can use the output of `corrmtx` to generate such an array `x`.

The second input argument, `p` is the number of complex sinusoids in `x`. You can specify `p` as either:

- A scalar integer. In this case, the signal subspace dimension is `p`.
- A two-element vector. In this case, `p(2)`, the second element of `p`, represents a threshold that is multiplied by λ_{min}, the smallest estimated eigenvalue of the signal's correlation matrix. Eigenvalues below the threshold λ_{min}`*p(2)` are assigned to the noise subspace. In this case, `p(1)` specifies the maximum dimension of the signal subspace.

The extra threshold parameter in the second entry in `p` provides you more flexibility and control in assigning the noise and signal subspaces.

The length of the vector `w` is the computed dimension of the signal subspace. For real-valued input data `x`, the length of the corresponding power vector `pow` is given by

```
length(pow) = 0.5*length(w)
```

For complex-valued input data `x`, `pow` and `w` have the same length.

`[f,pow] = rootmusic(...,fs)` returns the vector of frequencies `f` calculated in Hz. You supply the sampling frequency `fs` in Hz. If you specify `fs` with the empty vector `[]`, the sampling frequency defaults to 1 Hz.

`[w,pow] = rootmusic(...,'corr')` forces the input argument `x` to be interpreted as a correlation matrix rather than a matrix of signal data. For this syntax, you must supply a square matrix for `x`, and all of its eigenvalues must be nonnegative.

Note You can place the string `'corr'` anywhere after `p`.

Examples

Find the frequency content in a signal composed of three complex exponentials in noise. Use the modified covariance method to estimate the correlation matrix used by the MUSIC algorithm:

```
randn('state',1); n=0:99;
s = exp(i*pi/2*n)+2*exp(i*pi/4*n)+exp(i*pi/3*n)...
+randn(1,100);
% Estimate correlation matrix using modified
% covariance method.
X=corrmtx(s,12,'mod');
[W,P] = rootmusic(X,3)
W =
    1.5769
    0.7817
    1.0561
P =
```

```
1.1358
3.9975
1.4102
```

Algorithm

The MUSIC algorithm used by `rootmusic` is the same as that used by `pmusic`. The algorithm performs eigenspace analysis of the signal's correlation matrix in order to estimate the signal's frequency content.

The difference between `pmusic` and `rootmusic` is:

- `pmusic` returns the pseudospectrum at all frequency samples.
- `rootmusic` returns the estimated discrete frequency spectrum, along with the corresponding signal power estimates.

`rootmusic` is most useful for frequency estimation of signals made up of a sum of sinusoids embedded in additive white Gaussian noise.

Diagnostics

If the input signal, `x` is real and an odd number of sinusoids, `p` is specified, this error message is displayed

```
Real signals require an even number p of complex sinusoids.
```

See Also

`corrmtx`, `peig`, `pmusic`, `powerest` method of `spectrum`, `rooteig`, `spectrum.music`

Purpose

Sawtooth or triangle wave

Syntax

```
sawtooth(t)
sawtooth(t,width)
```

Description

`sawtooth(t)` generates a sawtooth wave with period 2π for the elements of time vector `t`. `sawtooth(t)` is similar to `sin(t)`, but creates a sawtooth wave with peaks of -1 and 1 instead of a sine wave. The sawtooth wave is defined to be -1 at multiples of 2π and to increase linearly with time with a slope of 1/π at all other times.

`sawtooth(t,width)` generates a modified triangle wave where `width`, a scalar parameter between 0 and 1, determines the point between 0 and 2π at which the maximum occurs. The function increases from -1 to 1 on the interval 0 to 2π*`width`, then decreases linearly from 1 to -1 on the interval 2π*`width` to 2π. Thus a parameter of 0.5 specifies a standard triangle wave, symmetric about time instant π with peak-to-peak amplitude of 1. `sawtooth(t,1)` is equivalent to `sawtooth(t)`.

See Also

`chirp`, `cos`, `diric`, `gauspuls`, `pulstran`, `rectpuls`, `sin`, `sinc`, `square`, `tripuls`

schurrc

Purpose

Compute reflection coefficients from autocorrelation sequence

Syntax

```
k = schurrc(r)
[k,e] = schurrc(r)
```

Description

`k = schurrc(r)` uses the Schur algorithm to compute a vector `k` of reflection coefficients from a vector `r` representing an autocorrelation sequence. `k` and `r` are the same size. The reflection coefficients represent the lattice parameters of a prediction filter for a signal with the given autocorrelation sequence, `r`. When `r` is a matrix, `schurrc` treats each column of `r` as an independent autocorrelation sequence, and produces a matrix `k`, the same size as `r`. Each column of `k` represents the reflection coefficients for the lattice filter for predicting the process with the corresponding autocorrelation sequence `r`.

`[k,e] = schurrc(r)` also computes the scalar `e`, the prediction error variance. When `r` is a matrix, `e` is a column vector. The number of rows of `e` is the same as the number of columns of `r`.

Examples

Create an autocorrelation sequence from the MATLAB speech signal contained in `mtlb.mat`, and use the Schur algorithm to compute the reflection coefficients of a lattice prediction filter for this autocorrelation sequence:

```
load mtlb
r = xcorr(mtlb(1:5),'unbiased');
k = schurrc(r(5:end))
k =
   -0.7583
    0.1384
    0.7042
   -0.3699
```

References

[1] Proakis, J. and D. Manolakis, *Digital Signal Processing: Principles, Algorithms, and Applications*, Third edition, Prentice-Hall, 1996, pp. 868-873.

See Also

levinson

seqperiod

Purpose

Compute period of sequence

Syntax

```
p = seqperiod(x)
[p,num] = seqperiod(x)
```

Description

`p = seqperiod(x)` returns the integer `p` that corresponds to the period of the sequence in a vector `x`. The period `p` is computed as the minimum length of a subsequence `x(1:p)` of `x` that repeats itself continuously every `p` samples in `x`. The length of `x` does not have to be a multiple of `p`, so that an incomplete repetition is permitted at the end of `x`. If the sequence `x` is not periodic, then `p = length(x)`.

- If `x` is a matrix, then `seqperiod` checks for periodicity along each column of `x`. The resulting output p is a row vector with the same number of columns as `x`.
- If `x` is a multidimensional array, then `seqperiod` checks for periodicity along the first nonsingleton dimension of `x`. In this case:
  - `p` is a multidimensional array of integers with a leading singleton dimension.
  - The lengths of the remaining dimensions of `p` correspond to those of the dimensions of `x` after the first nonsingleton one.

`[p,num] = seqperiod(x)` also returns the number `num` of repetitions of `x(1:p)` in `x`. `num` might not be an integer.

Examples

```
x = [4 0 1 6;
     2 0 2 7;
     4 0 1 5;
     2 0 5 6];
p = seqperiod(x)
p =
     2     1     4     3
```

The result implies:

- The first column of `x` has period 2.
- The second column of `x` has period 1.
- The third column of `x` is not periodic, so `p(3)` is just the number of rows of `x`.
- The fourth column of `x` has period 3, although the last (second) repetition of the periodic sequence is incomplete.

Purpose

Savitzky-Golay filter design

Syntax

```
b = sgolay(k,f)
b = sgolay(k,f,w)
[b,g] = sgolay(...)
```

Description

`b = sgolay(k,f)` designs a Savitzky-Golay FIR smoothing filter `b`. The polynomial order `k` must be less than the frame size, `f`, which must be odd. If `k = f-1`, the designed filter produces no smoothing. The output, `b`, is an `f`-by-`f` matrix whose rows represent the time-varying FIR filter coefficients. In a smoothing filter implementation (for example, `sgolayfilt`), the last `(f-1)/2` rows (each an FIR filter) are applied to the signal during the startup transient, and the first `(f-1)/2` rows are applied to the signal during the terminal transient. The center row is applied to the signal in the steady state.

`b = sgolay(k,f,w)` specifies a weighting vector `w` with length `f`, which contains the real, positive-valued weights to be used during the least-squares minimization.

`[b,g] = sgolay(...)` returns the matrix `g` of differentiation filters. Each column of `g` is a differentiation filter for derivatives of order `p`-1 where `p` is the column index. Given a signal `x` of length `f`, you can find an estimate of the p^{th} order derivative, `xp`, of its middle value from:

```
xp((f+1)/2) = (factorial(p)) * g(:,p+1)' * x
```

Remarks

Savitzky-Golay smoothing filters (also called digital smoothing polynomial filters or least squares smoothing filters) are typically used to “smooth out” a noisy signal whose frequency span (without noise) is large. In this type of application, Savitzky-Golay smoothing filters perform much better than standard averaging FIR filters, which tend to filter out a significant portion of the signal’s high frequency content along with the noise. Although Savitzky-Golay filters are more effective at preserving the pertinent high frequency components of the signal, they are less successful than standard averaging FIR filters at rejecting noise when noise levels are particularly high. The particular formulation of Savitzky-Golay filters preserves various moment orders

better than other smoothing methods, which tend to preserve peak widths and heights better than Savitzky-Golay.

Savitzky-Golay filters are optimal in the sense that they minimize the least-squares error in fitting a polynomial to each frame of noisy data.

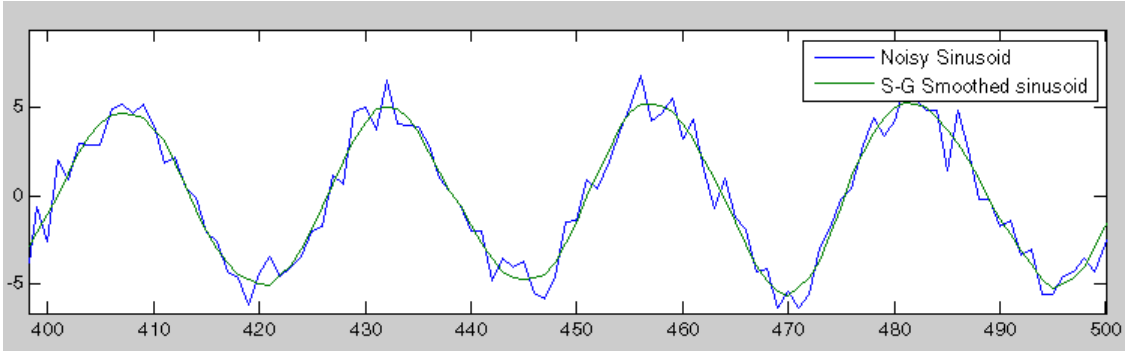
Examples

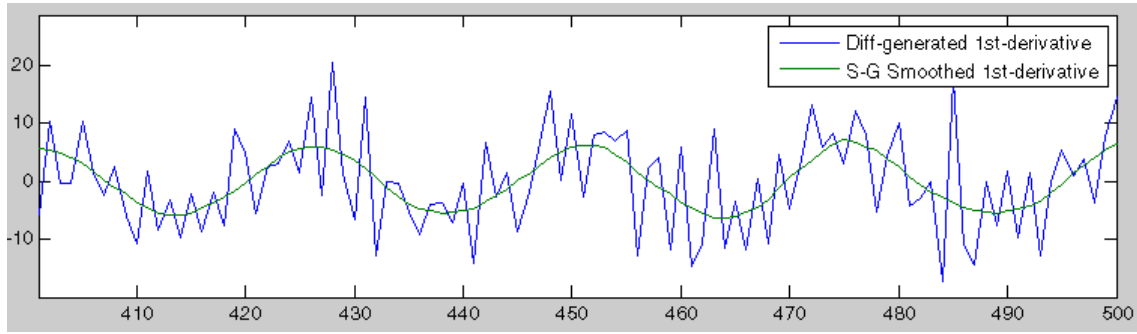
Use `sgolay` to smooth a noisy sinusoid and compare the resulting first and second derivatives to the first and second derivatives computed using `diff`. Notice how using `diff` amplifies the noise and generates useless results.

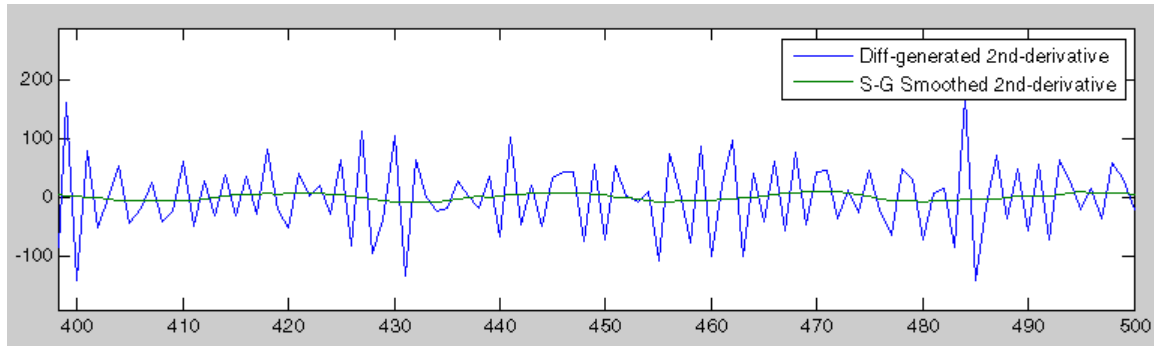
```
N = 4;                  % Order of polynomial fit
F = 21;                 % Window length
[b,g] = sgolay(N,F);    % Calculate S-G coefficients

dx = .2;
xLim = 200;
x = 0:dx:xLim-1;

y = 5*sin(0.4*pi*x)+randn(size(x));  % Sinusoid with noise

HalfWin  = ((F+1)/2) -1;
for n = (F+1)/2:996-(F+1)/2,
  % Zero-th derivative (smoothing only)
  SG0(n) =   dot(g(:,1), y(n - HalfWin: n + HalfWin));

  % 1st differential
  SG1(n) =   dot(g(:,2), y(n - HalfWin: n + HalfWin));

  % 2nd differential
  SG2(n) = 2*dot(g(:,3)', y(n - HalfWin: n + HalfWin))';
end

SG1 = SG1/dx;          % Turn differential into derivative
SG2 = SG2/(dx*dx);     % and into 2nd derivative

% Scale the "diff" results
```

```
DiffD1 = (diff(y(1:length(SG0)+1)))/ dx;
DiffD2 = (diff(diff(y(1:length(SG0)+2)))) / (dx*dx);

subplot(3,1,1);
plot([y(1:length(SG0))', SG0'])
legend('Noisy Sinusoid','S-G Smoothed sinusoid')

subplot(3, 1, 2);
plot([DiffD1',SG1'])
legend('Diff-generated 1st-derivative', ...
'S-G Smoothed 1st-derivative')

subplot(3, 1, 3);
plot([DiffD2',SG2'])
legend('Diff-generated 2nd-derivative',...
'S-G Smoothed 2nd-derivative')
```

Note The figures below are zoomed in each figure window panel to show more detail.







References

[1] Orfanidis, S.J., *Introduction to Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1996.

See Also

`fir1`, `firls`, `filter`, `sgolayfilt`

sgolayfilt

Purpose

Savitzky-Golay filtering

Syntax

```
y = sgolayfilt(x,k,f)
y = sgolayfilt(x,k,f,w)
y = sgolayfilt(x,k,f,w,dim)
```

Description

`y = sgolayfilt(x,k,f)` applies a Savitzky-Golay FIR smoothing filter to the data in vector `x`. If `x` is a matrix, `sgolayfilt` operates on each column. The polynomial order `k` must be less than the frame size, `f`, which must be odd. If `k = f-1`, the filter produces no smoothing.

`y = sgolayfilt(x,k,f,w)` specifies a weighting vector `w` with length `f`, which contains the real, positive-valued weights to be used during the least-squares minimization. If `w` is not specified or if it is specified as empty, `[]`, `w` defaults to an identity matrix.

`y = sgolayfilt(x,k,f,w,dim)` specifies the dimension, `dim`, along which the filter operates. If `dim` is not specified, `sgolayfilt` operates along the first non-singleton dimension; that is, dimension 1 for column vectors and nontrivial matrices, and dimension 2 for row vectors.

Remarks

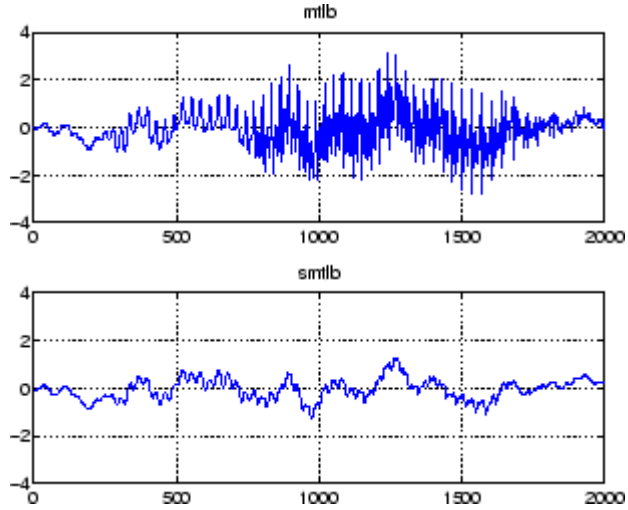
Savitzky-Golay smoothing filters (also called digital smoothing polynomial filters or least-squares smoothing filters) are typically used to "smooth out" a noisy signal whose frequency span (without noise) is large. In this type of application, Savitzky-Golay smoothing filters perform much better than standard averaging FIR filters, which tend to filter out a significant portion of the signal's high frequency content along with the noise. Although Savitzky-Golay filters are more effective at preserving the pertinent high frequency components of the signal, they are less successful than standard averaging FIR filters at rejecting noise.

Savitzky-Golay filters are optimal in the sense that they minimize the least-squares error in fitting a polynomial to frames of noisy data.

Examples

Smooth the `mtlb` signal by applying a cubic Savitzky-Golay filter to data frames of length 41:

```
load mtlb                          % Load data
smtlb = sgolayfilt(mtlb,3,41);     % Apply 3rd-order filter
subplot(2,1,1)
plot([1:2000],mtlb(1:2000)); axis([0 2000 -4 4]);
title('mtlb'); grid;
subplot(2,1,2)
plot([1:2000],smtlb(1:2000)); axis([0 2000 -4 4]);
title('smtlb'); grid;
```



References

[1] Orfanidis, S.J., *Introduction to Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1996.

See Also

`medfilt1`, `filter`, `sgolay`, `sosfilt`

Purpose

Signal processing window

Syntax

```
w = sigwin.window
```

Description

`w = sigwin.window` returns a window object, `w`, of type *`window`*. Each window takes one or more inputs. If you specify a `sigwin.window` with no inputs, a default window of length 64 is created.

Note You must use a *`window`* with `sigwin`.

Constructors

`window` for `sigwin` specifies the type of window. All Signal Processing Toolbox windows are available for use with `sigwin`. For a complete list, see the `window` reference page. To get help on a `sigwin`, use the syntax `help sigwin.window` at the MATLAB prompt.

Methods

Methods provide ways of performing functions directly on your `sigwin` object without having to specify the window parameters again. You can apply this method directly on the variable you assigned to your `sigwin` object.

Method	Description
`generate`	Returns a column vector of values representing the window.

Method	Description
`info`	Returns information about the window object.
`winwrite`	Writes an ASCII file that contains window weights for a single window object or a vector of window objects. Default filename is `untitled.wf`. `winwrite(Hd,filename)` writes to a disk file named `filename` in the current working directory. The `.wf` extension is added automatically.

Viewing Object Parameters

As with any object, you can use `get` to view a `sigwin` object's parameters. To see a specific parameter,

```
get(w,'parameter')
```

or to see all parameters for an object,

```
get(w)
```

Changing Object Parameters

To set specific parameters,

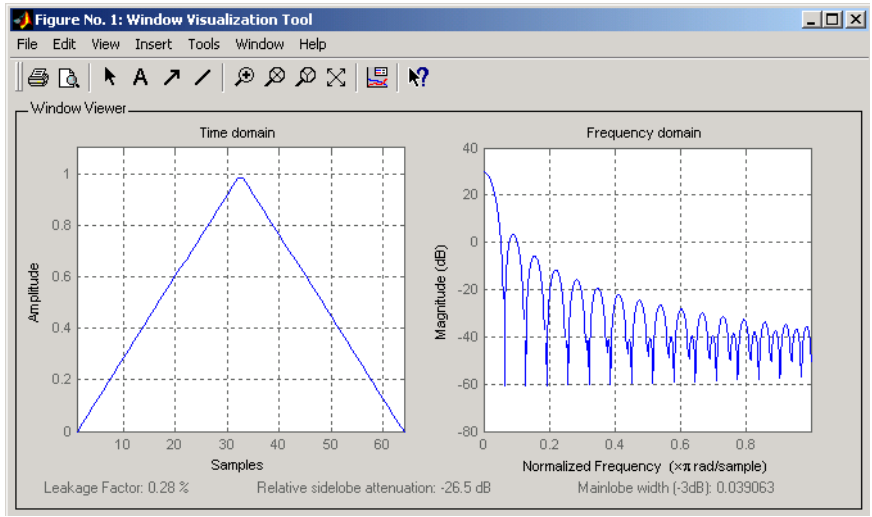
```
set(w,'parameter1',value,'parameter2',value,...)
```

Note that you must use single quotation marks around the parameter name.

Examples

Create a default Bartlett window and view the results in the Window Visualization Tool (`wvtool`). See `bartlett` for information on Bartlett windows:

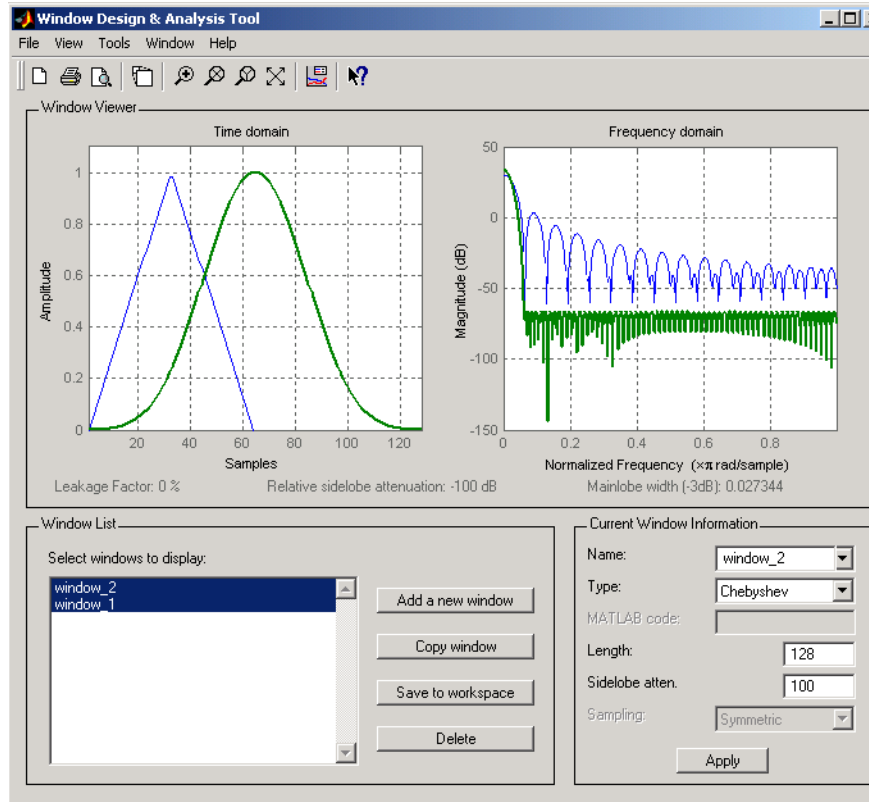
```
w=sigwin.bartlett

w =
    Length: 64
      Name: 'Bartlett'

wvtool(w)
```



Create a 128-point Chebyshev window with 100 dB of sidelobe attenuation. (See `chebwin` for information on Chebyshev windows.) View the results of this and the above Bartlett window in the Window Design and Analysis Tool (`wintool`):

```
w1=sigwin.chebwin(128,100)

w1 =
```

```
                Length: 128
                  Name: 'Chebyshev'
          SidelobeAtten: 100

wintool(w,w1)
```



To save the window values in a vector, use:

```
d = generate(w);
```

See Also

`window`, `wintool`, `wvtool`

Purpose

Sinc

Syntax

```
y = sinc(x)
```

Description

`sinc` computes the sinc function of an input vector or array, where the sinc function is

$$\mathrm{sinc}(t) = \begin{cases} 1, & t = 0 \\ \dfrac{\sin(\pi t)}{\pi t}, & t \neq 0 \end{cases}$$

This function is the continuous inverse Fourier transform of the rectangular pulse of width 2π and height 1.

$$\mathrm{sinc}(t) = \frac{1}{2\pi} \int_{-\pi}^{\pi} e^{j\omega t} d\omega$$

`y = sinc(x)` returns an array `y` the same size as `x`, whose elements are the `sinc` function of the elements of `x`.

The space of functions bandlimited in the frequency range $\omega \in [-\pi,\pi]$ is spanned by the infinite (yet countable) set of sinc functions shifted by integers. Thus any such bandlimited function $g(t)$ can be reconstructed from its samples at integer spacings.

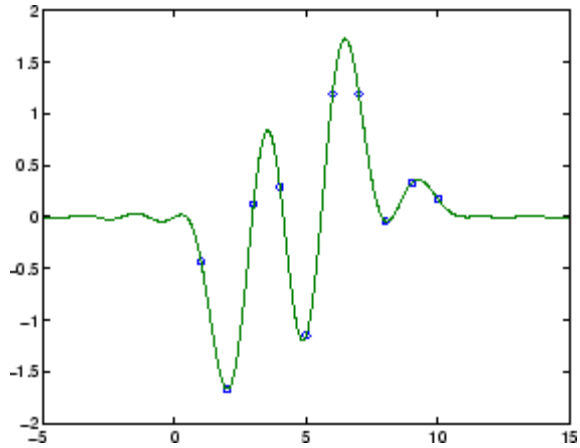
$$g(t) = \sum_{n=-\infty}^{\infty} g(n)\mathrm{sinc}(t-n)$$

Examples

Perform ideal bandlimited interpolation by assuming that the signal to be interpolated is 0 outside of the given time interval and that it has been sampled at exactly the Nyquist frequency:

```
t = (1:10)';                % Column vector of time samples
randn('state',0);
x = randn(size(t));         % Column vector of data
ts = linspace(-5,15,600)';  % Times at which to interpolate
```

```
y = sinc(ts(:,ones(size(t))) - t(:,ones(size(ts)))')*x;
plot(t,x,'o',ts,y)
```



See Also

chirp, cos, diric, gauspuls, pulstran, rectpuls, sawtooth, sin, square, tripuls

sos2cell

Purpose

Convert second-order sections matrix to cell array

Syntax

```
c = sos2cell(m)
c = sos2cell(m,g)
```

Description

`c = sos2cell(m)` changes an L-by-6 second-order section matrix m generated by `tf2sos` into a 1-by-L cell array of 1-by-2 cell arrays `c`. You can use `c` to specify a quantized filter with L cascaded second-order sections.

The matrix `m` should have the form

```
m = [b1 a1;b2 a2; ... ;bL aL]
```

where both `bi` and `ai`, with $i = 1, ..., L$, are 1-by-3 row vectors. The resulting `c` is a 1-by-L cell array of cells of the form

```
c = { {b1 a1} {b2 a2} ... {bL aL} }
```

`c = sos2cell(m,g)` with the optional gain term `g`, prepends the constant value `g` to `c`. When you use the added gain term in the command, `c` is a 1-by-L cell array of cells of the form

```
c = {{g,1} {b1,a1} {b2,a2}...{bL,aL} }
```

Examples

Use `sos2cell` to convert the 2-by-6 second-order section matrix produced by `tf2sos` into a 1-by-2 cell array `c` of cells. Display the second entry in the first cell in `c`:

```
[b,a] = ellip(4,0.5,20,0.6);
m = tf2sos(b,a);
c = sos2cell(m);
c{1}{2}
ans =
    1.0000    0.1677    0.2575
```

See Also

`tf2sos`, `cell2sos`

Purpose

Convert digital filter second-order section parameters to state-space form

Syntax

```
[A,B,C,D] = sos2ss(sos)
[A,B,C,D] = sos2ss(sos,g)
```

Description

`sos2ss` converts a second-order section representation of a given digital filter to an equivalent state-space representation.

`[A,B,C,D] = sos2ss(sos)` converts the system `sos`, in second-order section form, to a single-input, single-output state-space representation.

$$\begin{aligned} x[n+1] &= Ax[n] + Bu[n] \\ y[n] &= Cx[n] + Du[n] \end{aligned}$$

The discrete transfer function in second-order section form is given by

$$H(z) = \prod_{k=1}^{L} H_k(z) = \prod_{k=1}^{L} \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2}}{1 + a_{1k}z^{-1} + a_{2k}z^{-2}}$$

`sos` is a L-by-6 matrix organized as

$$sos = \begin{bmatrix} b_{01} & b_{11} & b_{21} & 1 & a_{11} & a_{21} \\ b_{02} & b_{12} & b_{22} & 1 & a_{12} & a_{22} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{0L} & b_{1L} & b_{2L} & 1 & a_{1L} & a_{2L} \end{bmatrix}$$

The entries of `sos` must be real for proper conversion to state space. The returned matrix `A` is size N-by-N, where $N = 2L$-1, `B` is a length N-1 column vector, `C` is a length N-1 row vector, and `D` is a scalar.

`[A,B,C,D] = sos2ss(sos,g)` converts the system `sos` in second-order section form with gain `g`.

$$H(z) = g \prod_{k=1}^{L} H_k(z)$$

Examples

Compute the state-space representation of a simple second-order section system with a gain of 2:

```
sos = [1  1  1  1  0 -1; -2  3  1  1 10  1];
[A,B,C,D] = sos2ss(sos)
A =
   -10     0    10     1
     1     0     0     0
     0     1     0     0
     0     0     1     0
B =
     1
     0
     0
     0
C =
    21     2   -16    -1
D =
    -2
```

Algorithm

`sos2ss` first converts from second-order sections to transfer function using `sos2tf`, and then from transfer function to state-space using `tf2ss`.

See Also

`sos2tf`, `sos2zp`, `ss2sos`, `tf2ss`, `zp2ss`

Purpose

Convert digital filter second-order section data to transfer function form

Syntax

```
[b,a] = sos2tf(sos)
[b,a] = sos2tf(sos,g)
```

Description

`sos2tf` converts a second-order section representation of a given digital filter to an equivalent transfer function representation.

`[b,a] = sos2tf(sos)` returns the numerator coefficients `b` and denominator coefficients `a` of the transfer function that describes a discrete-time system given by `sos` in second-order section form. The second-order section format of $H(z)$ is given by

$$H(z) = \prod_{k=1}^{L} H_k(z) = \prod_{k=1}^{L} \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2}}{1 + a_{1k}z^{-1} + a_{2k}z^{-2}}$$

`sos` is an L-by-6 matrix that contains the coefficients of each second-order section stored in its rows.

$$sos = \begin{bmatrix} b_{01} & b_{11} & b_{21} & 1 & a_{11} & a_{21} \\ b_{02} & b_{12} & b_{22} & 1 & a_{12} & a_{22} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{0L} & b_{1L} & b_{2L} & 1 & a_{1L} & a_{2L} \end{bmatrix}$$

Row vectors `b` and `a` contain the numerator and denominator coefficients of $H(z)$ stored in descending powers of z.

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_1 + b_2z^{-1} + \cdots + b_{n+1}z^{-n}}{a_1 + a_2z^{-1} + \cdots + a_{m+1}z^{-m}}$$

`[b,a] = sos2tf(sos,g)` returns the transfer function that describes a discrete-time system given by `sos` in second-order section form with gain `g`.

$$H(z) = g \prod_{k=1}^{L} H_k(z)$$

Examples

Compute the transfer function representation of a simple second-order section system:

```
sos = [1  1  1  1  0 -1; -2  3  1  1 10  1];
[b,a] = sos2tf(sos)
b =
   -2     1     2     4     1
a =
    1    10     0   -10    -1
```

Algorithm

`sos2tf` uses the `conv` function to multiply all of the numerator and denominator second-order polynomials together. For higher order filters (possibly starting as low as order 8), numerical problems due to roundoff errors may occur when forming the transfer function.

See Also

`latc2tf`, `sos2ss`, `sos2zp`, `ss2tf`, `tf2sos`, `zp2tf`

Purpose

Convert digital filter second-order section parameters to zero-pole-gain form

Syntax

```
[z,p,k] = sos2zp(sos)
[z,p,k] = sos2zp(sos,g)
```

Description

`sos2zp` converts a second-order section representation of a given digital filter to an equivalent zero-pole-gain representation.

`[z,p,k] = sos2zp(sos)` returns the zeros `z`, poles `p`, and gain `k` of the system given by `sos` in second-order section form. The second-order section format of $H(z)$ is given by

$$H(z) = \prod_{k=1}^{L} H_k(z) = \prod_{k=1}^{L} \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2}}{1 + a_{1k}z^{-1} + a_{2k}z^{-2}}$$

`sos` is an L-by-6 matrix that contains the coefficients of each second-order section in its rows.

$$sos = \begin{bmatrix} b_{01} & b_{11} & b_{21} & 1 & a_{11} & a_{21} \\ b_{02} & b_{12} & b_{22} & 1 & a_{12} & a_{22} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{0L} & b_{1L} & b_{2L} & 1 & a_{1L} & a_{2L} \end{bmatrix}$$

Column vectors `z` and `p` contain the zeros and poles of the transfer function $H(z)$.

$$H(z) = k\frac{(z-z_1)(z-z_2)\cdots(z-z_n)}{(p-p_1)(p-p_2)\cdots(p-p_m)}$$

where the orders n and m are determined by the matrix `sos`.

`[z,p,k] = sos2zp(sos,g)` returns the zeros `z`, poles `p`, and gain `k` of the system given by `sos` in second-order section form with gain `g`.

$$H(z) = g \prod_{k=1}^{L} H_k(z)$$

Examples

Compute the poles, zeros, and gain of a simple system in second-order section form:

```
sos = [1  1  1  1  0 -1; -2  3  1  1 10  1];
[z,p,k] = sos2zp(sos)
z =
  -0.5000 + 0.8660i
  -0.5000 - 0.8660i
   1.7808
  -0.2808
p =
   -1.0000
    1.0000
   -9.8990
   -0.1010
k =
     -2
```

Algorithm

`sos2zp` finds the poles and zeros of each second-order section by repeatedly calling `tf2zp`.

See Also

`sos2ss`, `sos2tf`, `ss2zp`, `tf2zp`, `tf2zpk`, `zp2sos`

Purpose

Second-order (biquadratic) IIR digital filtering

Syntax

```
y = sosfilt(sos,x)
y = sosfilt(sos,x,dim)
```

Description

`y = sosfilt(sos,x)` applies the second-order section digital filter `sos` to the vector `x`. The output, `y`, is the same length as `x`.

`sos` represents the second-order section digital filter $H(z)$

$$H(z) = \prod_{k=1}^{L} H_k(z) = \prod_{k=1}^{L} \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2}}{1 + a_{1k}z^{-1} + a_{2k}z^{-2}}$$

by an L-by-6 matrix containing the coefficients of each second-order section in its rows.

$$sos = \begin{bmatrix} b_{01} & b_{11} & b_{21} & 1 & a_{11} & a_{21} \\ b_{02} & b_{12} & b_{22} & 1 & a_{12} & a_{22} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{0L} & b_{1L} & b_{2L} & 1 & a_{1L} & a_{2L} \end{bmatrix}$$

If `x` is a matrix, `sosfilt` applies the filter to each column of `x` independently. The output `y` is a matrix of the same size, containing the filtered data corresponding to each column of `x`.

If `x` is a multidimensional array, `sosfilt` filters along the first nonsingleton dimension. The output `y` is a multidimensional array of the same size as `x`, containing the filtered data corresponding to each row and column of `x`

`y = sosfilt(sos,x,dim)` operates along the dimension `dim`.

References

[1] Orfanidis, S.J., *Introduction to Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1996.

See Also

`filter`, `medfilt1`, `sgolayfilt`

Purpose

Spectrogram using short-time Fourier transform

Syntax

```
S = spectrogram(x)
S = spectrogram(x,window)
S = spectrogram(x,window,noverlap)
S = spectrogram(x,window,noverlap,nfft)
S = spectrogram(x,window,noverlap,nfft,fs)
[S,F,T] = spectrogram(x,window,noverlap,F)
[S,F,T] = spectrogram(x,window,noverlap,F,fs)
[S,F,T,P] = spectrogram(...)
spectrogram(...)
```

Description

`spectrogram`, when used without any outputs, plots a spectrogram or, when used with an `S` output, returns the short-time Fourier transform of the input signal. To create a spectrogram from the returned short-time Fourier transform data, refer to the `[S,F,T,P]` syntax described below.

`S = spectrogram(x)` returns `S`, the short time Fourier transform of the input signal vector `x`. By default, `x` is divided into eight segments. If `x` cannot be divided exactly into eight segments, it is truncated. These default values are used.

- `window` is a Hamming window of length `nfft`.
- `noverlap` is the number of overlapping segments that produces 50% overlap between segments.
- `nfft` is the FFT length and is the maximum of 256 or the next power of 2 greater than the length of each segment of `x`. (Instead of `nfft`, you can specify a vector of frequencies, `F`. See below for more information.)
- `fs` is the sampling frequency, which defaults to normalized frequency.

Each column of `S` contains an estimate of the short-term, time-localized frequency content of `x`. Time increases across the columns of `S` and frequency increases down the rows.

If x is a length Nx complex signal, S is a complex matrix with nfft rows and k columns, where for a scalar window

```
k = fix((Nx-noverlap)/(window-noverlap))
```

or if window is a vector

```
k = fix((Nx-noverlap)/(length(window)-noverlap))
```

For real x, the output S has (nfft/2+1) rows if nfft is even, and (nfft+1)/2 rows if nfft is odd.

`S = spectrogram(x,window)` uses the window specified. If window is an integer, x is divided into segments equal to that integer value and a Hamming window is used. If window is a vector, x is divided into segments equal to the length of window and then the segments are windowed using the window functions specified in the window vector.

Note To obtain the same results for the removed specgram function, specify a 'Hann' window of length 256.

`S = spectrogram(x,window,noverlap)` overlaps noverlap samples of each segment. noverlap must be an integer smaller than window or if window is a vector, smaller than the length of window.

`S = spectrogram(x,window,noverlap,nfft)` uses the nfft number of sampling points to calculate the discrete Fourier transform. nfft must be a scalar.

`S = spectrogram(x,window,noverlap,nfft,fs)` uses fs sampling frequency in Hz. If fs is specified as empty [], it defaults to 1 Hz.

`[S,F,T] = spectrogram(x,window,noverlap,F)` uses a vector F of frequencies in Hz. F must be a vector with at least two elements. This case computes the spectrogram at the frequencies in F using the Goertzel algorithm. The specified frequencies are rounded to the nearest DFT bin commensurate with the signal's resolution. In all other

syntax cases where `nfft` or a default for `nfft` is used, the short-time Fourier transform is used. The `F` vector returned is a vector of the rounded frequencies. `T` is a vector of times at which the spectrogram is computed. The length of `F` is equal to the number of rows of `S`. The length of `T` is equal to `k`, as defined above and each value corresponds to the center of each segment.

`[S,F,T] = spectrogram(x,window,noverlap,F,fs)` uses a vector `F` of frequencies in Hz as above and uses the `fs` sampling frequency in Hz. If `fs` is specified as empty `[]`, it defaults to 1 Hz.

`[S,F,T,P] = spectrogram(...)` returns a matrix `P` containing the power spectral density (PSD) of each segment. For real `x`, `P` contains the one-sided modified periodogram estimate of the PSD of each segment. For complex `x` and when you specify a vector of frequencies `F`, `P` contains the two-sided PSD.

Note You can convert the returned `S` matrix of short time Fourier transforms into a spectrogram using `10*log10(abs(S.'))`.

`spectrogram(...)` plots the PSD estimate for each segment on a surface in a figure window. The plot is created using `surf(F,T,10*log10(abs(P))`. The `surf` function allows you to rotate the spectrogram.

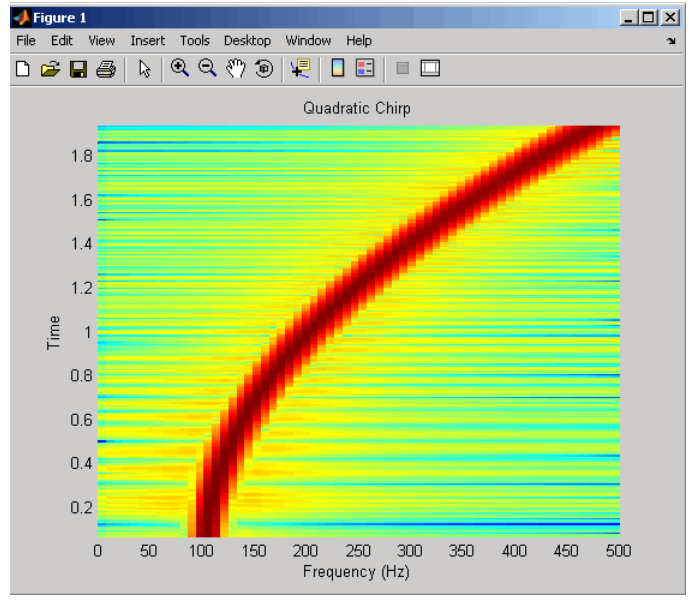
Using `spectrogram(...,'freqloc')` syntax and adding a `'freqloc'` string (either `'xaxis'` or `'yaxis'`) controls where the frequency axis is displayed. Using `'xaxis'` displays the frequency on the *x*-axis. Using `'yaxis'` displays frequency on the *y*-axis and time on the *x*-axis. The default is `'xaxis'`. If you specify both a `'freqloc'` string and output arguments, `'freqloc'` is ignored.

Examples

Compute and display the PSD of each segment of a quadratic chirp, which starts at 100 Hz and crosses 200 Hz at `t` = 1 sec.

```
T = 0:0.001:2;
```

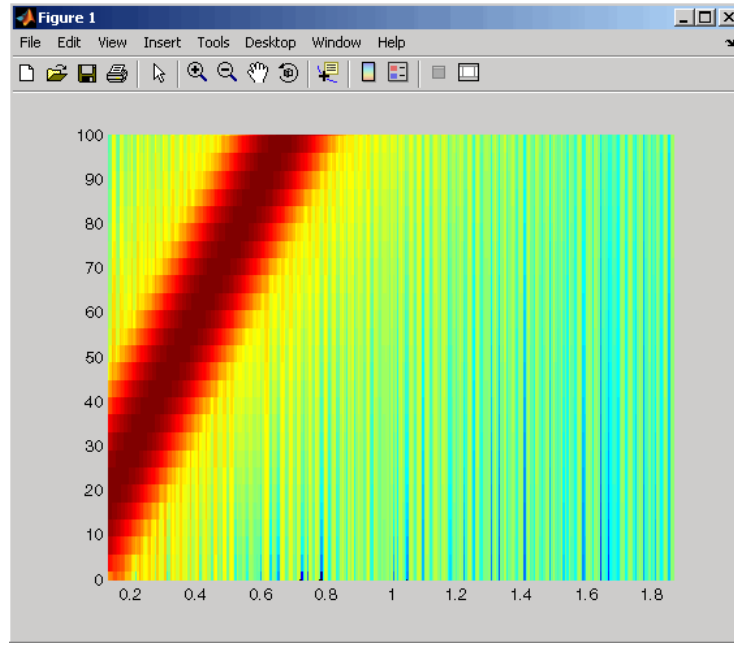
```
X = chirp(T,100,1,200,'q');
spectrogram(X,128,120,128,1E3);
title('Quadratic Chirp');
```



Compute and display the PSD of each segment of a linear chirp, which starts at DC and crosses 150 Hz at `t` = 1 sec. Display the frequency on the *y*-axis.

```
T = 0:0.001:2;
X = chirp(T,0,1,150);
F = 0:.1:100;
[Y,F,T,P] = spectrogram(X,256,250,F,1E3,'yaxis');

% The following code produces the same result as calling
% spectrogram with no outputs:
surf(T,F,10*log10(abs(P)),'EdgeColor','none');
axis xy; axis tight; colormap(jet); view(0,90);
xlabel('Time');
```

```
ylabel('Frequency (Hz)');
```



References

[1] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1989, pp. 713-718.

[2] Rabiner, L.R., and R.W. Schafer, *Digital Processing of Speech Signals*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

See Also

`goertzel`, `periodogram`, `pwelch`, `spectrum.periodogram`, `spectrum.welch`

Purpose

Spectral estimation

Syntax

```
Hs = spectrum.estmethod(input1,...)
```

Description

`Hs = spectrum.estmethod(input1,...)` returns a spectral estimation object `Hs` of type *`estmethod`*. This object contains all the parameter information needed for the specified estimation method. Each estimation method takes one or more inputs, which are described on the individual reference pages.

Estimation Methods

Estimation methods for `spectrum` specify the type of spectral estimation method to use. Available estimation methods for `spectrum` are listed below.

Note You must use a spectral *`estmethod`* with `spectrum`.

Spectrum Estimation Methods

spectrum.estmethod	Description	Corresponding Function
`spectrum.burg`	Burg	`pburg`
`spectrum.cov`	Covariance	`pcov`
`spectrum.eigenvector`	Eigenvector	`peig`
`spectrum.mcov`	Modified covariance	`pmcov`
`spectrum.mtm`	Thompson multitaper	`pmtm`
`spectrum.music`	Multiple Signal Classification	`pmusic`
`spectrum.periodogram`	Periodogram	`periodogram`
`spectrum.welch`	Welch	`pwelch`
`spectrum.yulear`	Yule-Walker	`pyulear`

For more information on each estimation method, use the syntax `help spectrum.estmethod` at the MATLAB prompt or refer to its reference page.

Note For estimation methods that use overlap and window length inputs, you specify the number of overlap samples as a percent overlap and you specify the segment length instead of the window length.

For estimation methods that use windows, if the window uses an additional parameter, a property is dynamically added to the spectrum object for that parameter. You can change that property using `set` (see "Changing Object Properties" on page 10-784).

Methods

Methods provide ways of performing functions directly on your `spectrum` object without having to specify the spectral estimation parameters again. You can apply these methods directly on the variable you assigned to your `spectrum` object. For more information on any of these methods, use the syntax `help spectrum/method` at the MATLAB prompt or refer to the table below.

Spectrum Methods

Method	Description
`msspectrum`	Note that the `msspectrum` method is only available for the `periodogram` and `welch` spectrum estimation objects. The mean-squared spectrum is intended for discrete spectra (from periodic, discrete-time signals). The distribution of the mean square value across frequency is the msspectrum. Unlike the power spectral density (see `psd` below), the peaks in the mean-square spectrum reflect the power in the signal at a given frequency. For the PSD, the power is reflected as the area in a frequency band. The units of the mean-squared spectrum are units of power.

Spectrum Methods (Continued)

Method	Description

	`Hmss = msspectrum(Hs,X)` returns a mean-square spectrum object containing the mean-square (power) estimate of the discrete-time signal `X` using the spectrum object `Hs`. Default for real `X` is the `'onesided'` Nyquist frequency range and for complex `X` the default is the `'twosided'` Nyquist frequency range.
	
	`Hmss` contains a vector of normalized frequencies `W`, at which the mean-square spectrum is estimated. For real signals, the range of `W` is [0,π] if the number of FFT points (`NFFT`) is even, and [0,π) if `NFFT` is odd. For complex signals, the range of `W` is [0,2π). To estimate the spectrum on a vector of specific frequencies, see `FreqPoints` property below.
	
	The `msspectrum` method includes these properties, which you can set using this `msspectrum` method or via the `msspectrumopts` method. These properties are listed here and described in the `msspectrumopts` section below:
	
	`SpectrumType` — `'onesided'` or `'twosided'`
	`NormalizedFrequency` – normalizes frequency between 0 and 1
	`Fs` — sampling frequency in Hz
	`NFFT` — number of FFT points
	`CenterDC` — shifts data and frequencies to center DC component
	`FreqPoints` — `'All'` or `'User Defined'`
	`FrequencyVector` — frequencies at which to compute spectrum
	`ConfLevel` — confidence level to calculate the confidence interval. Value must be from 0 to 1.
	
	For example, `Hmss = msspectrum(Hs,X,'FreqPoints','User Defined', FreqVector,fvect)` returns a mean-square spectrum object where the spectrum is calculated only on the frequency points defined in the frequency vector, `fvect`.
	
	`msspectrum(...)` with no output arguments plots the mean-square spectrum in dB.

Spectrum Methods (Continued)

Method	Description
`msspectrumopts`	`Hopts = msspectrumopts(Hs)` returns an object that contains options for the spectrum object `Hs`. `Hopts = msspectrumopts(Hs,X)` returns an object with data-specific options and defaults. You can pass an `Hopts` options object as an argument to the `msspectrum` method. Any individual option you specify after the `Hopts` object overrides the value in `Hopts`. For example, `Hmss = msspectrum(Hs,X,Hopts, 'SpectrumType', 'twosided')` overrides the default `SpectrumType` value in `Hopts`. The following properties apply to both `msspectrumopts` and `msspectrum` methods. `Hmss = msspectrum (..., 'SpectrumType', 'twosided')` returns the two-sided mean-square spectrum. The spectrum length (NFFT) is computed over [0,2π), or if `Fs` is specified, [0,`Fs`) . Entering `'onesided'` returns the one-sided mean-square spectrum, which contains the total signal power in half the Nyquist range. Default is `'onesided'`. `Hmss = msspectrum(Hs,X,'NormalizedFrequency',true)` returns a mean-square spectrum object with frequency values normalized between 0 and 1. Default is `true`. `Hmss = msspectrum(Hs,X,'Fs',Fs)` returns a mean-square spectrum object computed as a function of frequency, where `Fs` is the sampling frequency in Hz. Note that you can set `Fs` only if `NormalizedFrequency` is set to `false`. `Hmss = msspectrum(...,'NFFT',nfft)` specifies the number of FFT points to use. Valid values are a positive integer, `'Nextpow2'` or `'Auto'`. `'Nextpow2'` uses the next power of 2 greater than the input length or 256, whichever is greater. `'Auto'` uses the input length or 256, whichever is greater. Default is `'Nextpow2'`. Note that

Spectrum Methods (Continued)

Method	Description

	for `spectrum.welch`, `'Nextpow2'` and `'Auto'` are compared to the `SegmentLength` instead of the input length.
	
	`Hmss = msspectrum (..., 'Centerdc', true)` shifts the data and frequency values so that the DC component is at the center of the spectrum. Default is `false`.
	
	To estimate the spectrum on a vector of specific frequencies, first set the number of frequency points to `'User Defined'`, which replaces the `NFFT` property of `msspectrum` with a `FrequencyVector` property.
	`Hopts.FreqPoints = 'User Defined'`
	(Note that the default for `FreqPoints` is `'All'`, which causes `msspectrum` to use the `NFFT` property as described above.)
	
	Then, specify the frequency vector `F` to use.
	`Hopts.FrequencyVector = F`
	(Note that the default value for `FrequencyVector` is `'Auto'`. In this case, the number of frequency points used follows the same rule as described for `NFFT 'Auto'` above.)
	
	`Hmms = msspectrum(...,'ConfLevel',p)` specifies the confidence level `p` for computing the confidence interval, which is an estimate of the error in the calculated mean-squared spectrum. The confidence level (`p`) is between 0 and 1. For example, `Hmss = msspectrum(Hs,X,'ConfLevel',0.95)` returns the 95% confidence interval.

Spectrum Methods (Continued)

Method	Description
psd	Note that music and eigenvector spectrum objects do not support the psd method. See the pseudospectrum method below. The power spectral density (PSD) is intended for continuous spectra. The integral of the PSD over a given frequency band computes the average power in the signal in that frequency band. In contrast to the msspectrum, the peaks in this spectra do not reflect the power at a given frequency. The units of the PSD are power per unit of frequency. See the avgpower method of dspdata for more information. Hpsd = psd (Hs,X) returns a power spectral density object containing the power spectral density estimate of the discrete-time signal X using the spectrum object Hs. The PSD is the distribution of power per unit frequency. Default for real X is 'onesided' and for complex X is 'twosided'. Hpsd contains a vector of normalized frequencies W, at which the PSD is estimated. For real signals, the range of W is [0,π] if the number of FFT points (NFFT) is even, and [0,π) if NFFT is odd. For complex signals, the range of W is [0,2π). The psd method includes these properties, which you can set using this psd method or via the psdopts method. These properties are listed here and described in the psdopts section below: SpectrumType — 'onesided' or 'twosided' NormalizedFrequency — normalizes frequency between 0 and 1 Fs — sampling frequency in Hz NFFT — number of FFT points CenterDC — shifts data and frequencies to center DC component FreqPoints — 'All' or 'User Defined' FrequencyVector – frequencies at which to compute spectrum ConfLevel — confidence level to calculate the confidence interval. Value must be from 0 to 1.

Spectrum Methods (Continued)

Method	Description
	For example, `Hmss = psd(Hs,X,'FreqPoints','User Defined', FreqVector,fvect)` returns a PSD object where the spectrum is calculated only on the frequency points defined in the frequency vector, `fvect`. `psd(...)` with no output arguments plots PSD in dB per unit frequency.
`psdopts`	`Hopts = psdopts(Hs)` returns an object that contains options for the spectrum object `Hs`. `Hopts = psdopts(Hs,X)` returns an object with data-specific options and defaults. You can pass an `Hopts` options object as an argument to the `psd` method. Any individual option you specify after the `Hopts` object overrides the value in `Hopts`. For example, `Hpsd = psd(Hs,X,Hopts,'SpectrumType', 'twosided')` overrides the `SpectrumType` value in `Hopts`. The following properties apply to both `psdmopts` and `psd` methods. `Hpsd = psd (Hs,X,'SpectrumType','twosided')` returns the two-sided power spectral density of `X`. The spectrum length is NFFT and is computed over [0,2π) if `Fs` is not specified or [0,`Fs`) if `Fs` is specified. Entering `'onesided'` returns the one-sided PSD, which contains the total signal power. `Hmss = psd(Hs,X,'NormalizedFrequency',true)` returns a power spectral density object with frequency values normalized between 0 and 1. Default is `true`. `Hpsd = psd (...,'Fs',Fs)` returns a power spectral density object computed as a function of frequency, where `Fs` is the sampling frequency in Hz.

Spectrum Methods (Continued)

Method	Description
	`Hmss = psd(...,'NFFT',nfft)` specifies the number of FFT points to use. Valid values are a positive integer, `'Nextpow2'` or `'Auto'`. `'Nextpow2'` uses the next power of 2 greater than the input length or 256, whichever is greater. `'Auto'` uses the input length or 256, whichever is greater. Default is `'Nextpow2'`. Note that for `spectrum.welch`, `'Nextpow2'` and `'Auto'` are compared to the `SegmentLength` instead of the input length. `Hmss = psd (..., 'Centerdc', true)` shifts the data and frequency values so that the DC component is at the center of the spectrum. Default is `false`. To estimate the spectrum on a vector of specific frequencies, first set the number of frequency points to `'User Defined'`, which replaces the `NFFT` property of `psd` with a `FrequencyVector` property. `Hopts.FreqPoints = 'User Defined'` (Note that the default for `FreqPoints` is `'All'` which causes `psd` to use the `NFFT` property as described above.) `Hmms = msspectrum(...,'ConfLevel',p)` specifies the confidence level `p` for computing the confidence interval, which is an estimate of the error in the calculated PSD. The confidence level (`p`) is between 0 and 1. For example, `Hmss = psd(Hs,X,'ConfLevel',0.95)` returns the 95% confidence interval.

Spectrum Methods (Continued)

Method	Description

`pseudospectrum`	Note that this method is used for only `music` or `eigenvector` spectrum objects.
	
	`Hps = pseudospectrum(Hs,X)` returns an object containing the pseudospectrum estimate of the discrete-time signal `X` using the spectrum object `Hs`. `Hs` must be a `music` or `eigenvector` object. Default for real `X` is `'half'` and for complex `X` is the `'whole'` Nyquist frequency range.
	
	`Hps` contains a vector of normalized frequencies `W`, at which the pseudospectrum is estimated. For real signals, the range of `W` is [0,π] if the number of FFT points (`NFFT`) is even, and [0,π) if `NFFT` is odd. For complex signals, the range of `W` is [0,2π).
	
	The `pseudospectrum` method includes these properties, which you can set using this `pseudospectrum` method or via the `pseudospectrumopts` method. These properties are described below:
	
	`SpectrumRange` — `'half'` or `'whole'`
	`NormalizedFrequency` — normalizes frequency between 0 and 1
	`Fs` — sampling frequency in Hz
	`NFFT` — number of FFT points
	`CenterDC` — shifts data and frequencies to center DC component
	`FreqPoints` — `'All'` or `'User Defined'`
	`FrequencyVector` — frequencies at which to compute spectrum
	
	For example, `Hmss = psd(Hs,X,'FreqPoints','User Defined', FreqVector,fvect)` returns a PSD object where the spectrum is calculated only on the frequency points defined in the frequency vector, `fvect`.
	
	`pseudospectrum(...)` with no output arguments plots the pseudospectrum in dB.

Spectrum Methods (Continued)

Method	Description
`pseudo-spectrumopts`	`Hopts = pseudospectrumopts(Hs)` returns an object that contains options for the spectrum object `Hs`. `Hopts = pseudospectrumopts(Hs,X)` returns an object with data-specific options and defaults. You can pass an `Hopts` options object as an argument to the `pseudospectrum` method. Any individual option you specify after the `Hopts` object overrides the value in `Hopts`. For example, `Hpseudospectrum= pseudospectrum(Hs,X, Hopts,'SpectrumRange', 'whole')` overrides the `SpectrumRange` value in `Hopts`. `Hmps = pseudospectrum (..., 'SpectrumRange', 'whole')` returns the pseudospectrum over the whole Nyquist range. The spectrum length is NFFT and is computed over [0,2π) if `Fs` is not specified or [0,`Fs`) if `Fs` is specified. Entering `'half'` returns the pseudospectrum calculated over half the Nyquist range. `Hmss = pseudospectrum(Hs,X,'NormalizedFrequency',true)` returns a pseudospectrum object with frequency values normalized between 0 and 1. Default is `true`. `Hps = pseudospectrum(Hs,X,'Fs',Fs)` returns a pseudospectrum object computed as a function of frequency, where `Fs` is the sampling frequency in Hz. `Hps = pseudospectrum(...,'NFFT',nfft)` specifies the number of FFT points to use. Valid values are a positive integer, `'Nextpow2'` or `'Auto'`. `'Nextpow2'` uses the next power of 2 greater than the input length or 256, whichever is greater. `'Auto'` uses the input length or 256, whichever is greater. Default is `'Nextpow2'`. `Hps = pseudospectrum(...,'Centerdc',true)` shifts the data and frequency values so that the DC component is at the center of the spectrum. The default value is `false`. To estimate the spectrum on a vector of specific frequencies, first set the number of frequency points to `'User Defined'`, which replaces

Spectrum Methods (Continued)

Method	Description
	the NFFT property of pseudospectrum with a FrequencyVector property. Hopts.FreqPoints = 'User Defined' (Note that the default for FreqPoints is 'All', which causes pseudospectrum to use the NFFT property as described above.)
powerest	Note that powerest is available only for music and eigenvector spectrum objects. POW = powerest(Hs,X) returns a vector POW containing estimates of the powers of the complex sinusoids in X. The input X can be a vector or a matrix. If it is a matrix it can be a data matrix, where $X'^{*}X = R$ or a correlation matrix R. The value the InputType property of Hs determines how X is interpreted. Hs must be a music or eigenvector spectrum object. [POW,W]=powerest(Hs,X) returns POW and a vector W of the frequencies in rad/sample of the sinusoids in X. [POW,F]=powerest(Hs,X,Fs) returns POW and a vector F of the frequencies in Hz of the sinusoids in X. Fs is the sampling frequency.

Viewing Object Properties

As with any object, you can use get to view a spectrum object's properties. To see a specific property, use

```
get(Hs,'property')
```

where 'property' is the specific property name.

To see all properties for an object, use

```
get(Hs)
```

Changing Object Properties

To set specific properties, use

```
set(Hs,'property1',value, 'property2',value,...)
```

where `'property1'`, `'property2'`, etc. are the specific property names.

To view the options for a property use `set` without specifying a value

```
set(Hs,'property')
```

Note that you must use single quotation marks around the property name. For example, to change the order of a Burg `spectrum` object `Hs` to 6, use

```
set(Hs,'order',6)
```

Another example of using `set` to change an object's properties is this example of changing the dynamically created window property of a periodogram `spectrum` object.

```
Hs=spectrum.periodogram       % Create periodogram object

Hs =

    EstimationMethod: 'Periodogram'
          WindowName: 'Rectangular'

set(Hs,'WindowName','Chebyshev')   % Change window type
Hs                                 % View changed object

Hs =

    EstimationMethod: 'Periodogram'
          WindowName: 'Chebyshev'  % Note changed property
       SidelobeAtten: 100

set(Hs,'SidelobeAtten',150)   % Change dynamic property
```

```
Hs                              % View changed object

Hs =

    EstimationMethod: 'Periodogram'
          WindowName: 'Chebyshev'
       SidelobeAtten: 150
```

All `spectrum` object properties can be changed using the `set` command, except for the `EstimationMethod` property.

Another way to change an object's properties is by using the `inspect` command which opens the Property Inspector window where you can edit any property, except dynamic properties, such as those used with windows.

```
inspect(Hs)
```

Copying an Object

To create a copy of an object, use the `copy` method.

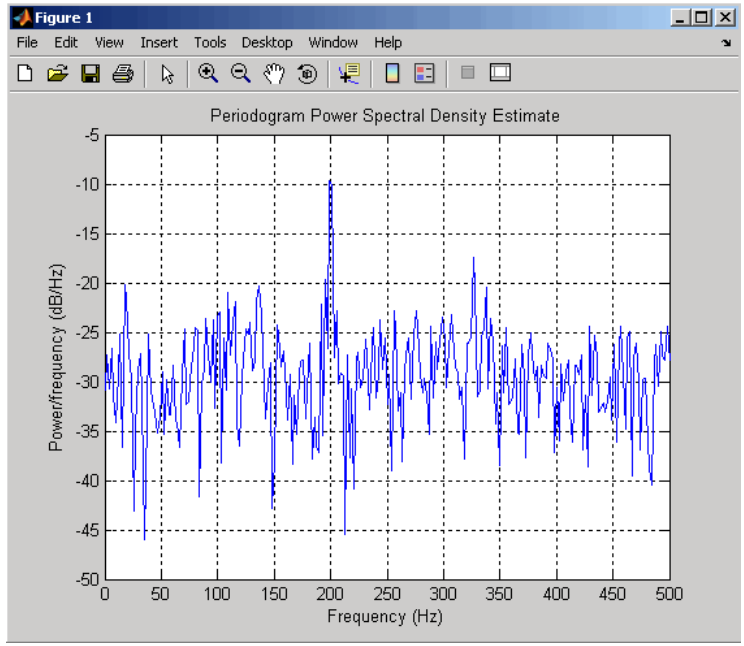
```
H2 = copy(Hs)
```

Note Using the syntax `H2 = Hs` copies only the object handle and does not create a new object.

Examples

Define a cosine of 200 Hz, add some noise and then view its power spectral density estimate generated with the periodogram algorithm.

```
Fs = 1000;
t = 0:1/Fs:.3;
x=cos(2*pi*t*200)+randn(size(t));
Hs=spectrum.periodogram;
psd(Hs,x,'Fs',Fs)
```



Refer to the reference pages for each estimation method for more examples.

See Also

`dspdata`, `spectrum.burg`, `spectrum.cov`, `spectrum.mcov`, `spectrum.yulear`, `spectrum.periodogram`, `spectrum.welch`, `spectrum.mtm`, `spectrum.eigenvector`, `spectrum.music`

Purpose

Burg spectrum

Syntax

```
Hs = spectrum.burg
Hs = spectrum.burg(order)
```

Description

`Hs = spectrum.burg` returns a default Burg spectrum object, `Hs`, that defines the parameters for the Burg parametric spectral estimation algorithm. The Burg algorithm estimates the spectral content by fitting an auto-regressive (AR) linear prediction filter model of a given order to the signal.

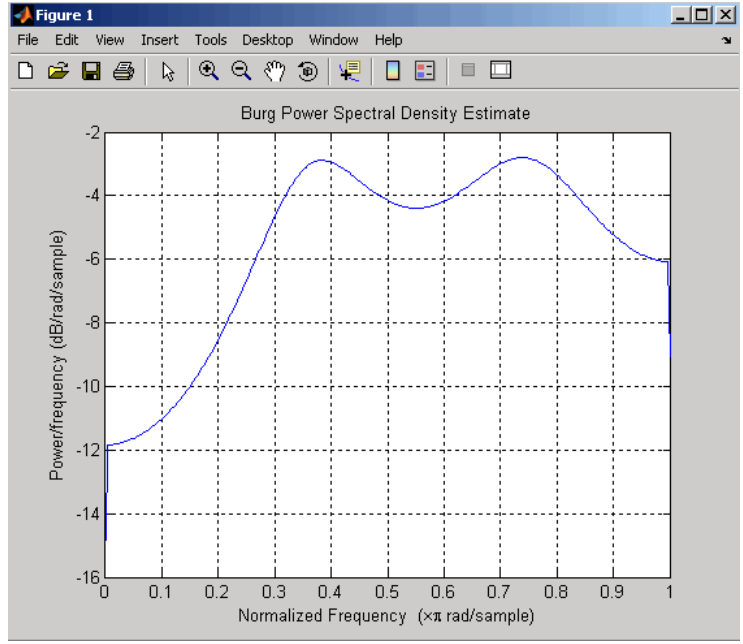
`Hs = spectrum.burg(order)` returns a spectrum object, `Hs` with the specified `order`. The default value for `order` is 4.

Note See `pburg` for more information on the Burg algorithm.

Examples

Define a fourth order auto-regressive model and view its power spectral density using the Burg algorithm.

```
randn('state',1);
x=randn(100,1);
x=filter(1,[1 1/2 1/3 1/4 1/5],x);   % 4th order AR filter
Hs=spectrum.burg;                    % 4th order AR model
psd(Hs,x,'NFFT',512)
```



See Also

dspdata, spectrum, spectrum.cov, spectrum.mcov, spectrum.yulear, spectrum.periodogram, spectrum.welch, spectrum.mtm, spectrum.eigenvector, spectrum.music

Purpose

Covariance spectrum

Syntax

```
Hs = spectrum.cov
Hs = spectrum.cov(order)
```

Description

`Hs = spectrum.cov` returns a default covariance spectrum object, `Hs`, that defines the parameters for the covariance spectral estimation algorithm. The covariance algorithm estimates the spectral content by fitting an auto-regressive (AR) linear prediction model of a given `order` to the signal.

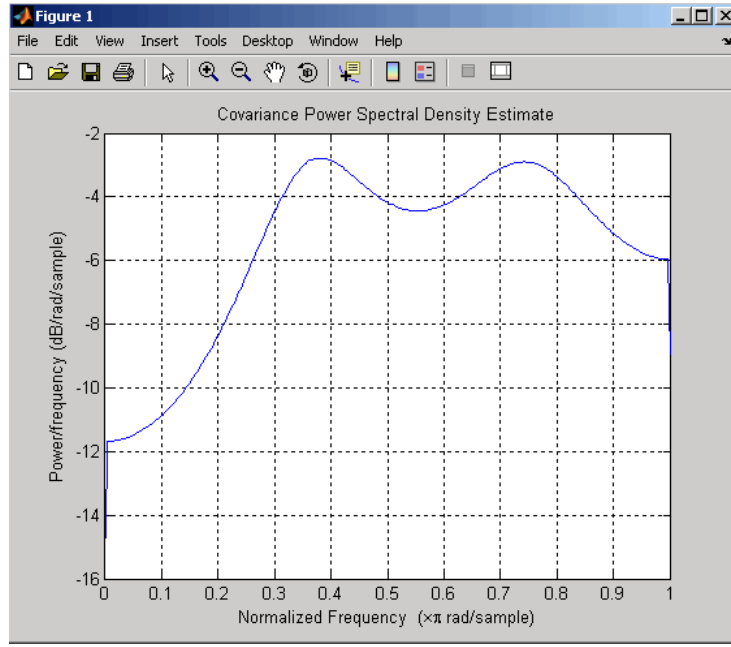
`Hs = spectrum.cov(order)` returns a spectrum object, `Hs` with the specified `order`. The default value for `order` is 4.

Note See `pcov` for more information on the covariance algorithm.

Examples

Define a fourth order auto-regressive model and view its power spectral density using the covariance algorithm.

```
randn('state',1);
x=randn(100,1);
x=filter(1,[1 1/2 1/3 1/4 1/5],x);   % 4th order AR filter
Hs=spectrum.cov;                     % 4th order AR model
psd(Hs,x,'NFFT',512)
```



See Also

dspdata, spectrum, spectrum.burg, spectrum.mcov, spectrum.yulear, spectrum.periodogram, spectrum.welch, spectrum.mtm, spectrum.eigenvector, spectrum.music

Purpose

Eigenvector spectrum

Syntax

```
Hs = spectrum.eigenvector
Hs = spectrum.eigenvector(NSinusoids)
Hs = spectrum.eigenvector(NSinusoids,SegmentLength)
Hs = spectrum.eigenvector(NSinusoids,SegmentLength,...
OverlapPercent)
Hs = spectrum.eigenvector(NSinusoids,SegmentLength,...
OverlapPercent,WindowName)
Hs = spectrum.eigenvector(NSinusoids,SegmentLength,...
OverlapPercent,WindowName,SubspaceThreshold)
Hs = spectrum.eigenvector(NSinusoids,SegmentLength,...
OverlapPercent,WindowName,SubspaceThreshold,InputType)
```

Description

`Hs = spectrum.eigenvector` returns a default eigenvector spectrum object, `Hs`, that defines the parameters for an eigenanalysis spectral estimation method. This object uses the following default values:

Default Values

Property Name	Default Value	Description
`NSinusoids`	`2`	Number of complex sinusoids
`SegmentLength`	`4`	Length of each of the time-based segments into which the input signal is divided.
`OverlapPercent`	`50`	Percent overlap between segments

Default Values (Continued)

Property Name	Default Value	Description
`WindowName`	`'Rectangular'`	Window name string or `'User Defined'` (see `window` for valid window names). For more information on each window, refer to its reference page. This argument can also be a cell array containing the window name string or `'User Defined'` and, if used for the particular window, an optional parameter value. The syntax is `{wname,wparam}`. You can use `set` to change the value of the additional parameter or to define the MATLAB expression and parameters for a user-defined window (see `spectrum` for information on using `set`).
`SubspaceThreshold`	`0`	Threshold is the cutoff for signal and noise separation. The threshold is multiplied by λ_{min} , the smallest estimated eigenvalue of the signal's correlation matrix. Eigenvalues below the threshold (λ_{min}`*threshold`) are assigned to the noise subspace.
`InputType`	`'Vector'`	Type of input that will be used with this spectrum object. Valid values are `'Vector'`, `'DataMatrix'` and `'CorrelationMatrix'`.

`Hs = spectrum.eigenvector(NSinusoids)` returns a spectrum object, `Hs`, with the specified number of sinusoids and default values for all other properties. Refer to the table above for default values.

`Hs = spectrum.eigenvector(NSinusoids,SegmentLength)` returns a spectrum object, `Hs`, with the specified segment length.

`Hs = spectrum.eigenvector(NSinusoids,SegmentLength,...
OverlapPercent)` returns a spectrum object, `Hs`, with the specified overlap between segments.

`Hs = spectrum.eigenvector(NSinusoids,SegmentLength,...
OverlapPercent,WindowName)` returns a spectrum object, `Hs`, with the specified window.

Note Window names must be enclosed in single quotes, such as `spectrum.eigenvector(3,32,50,'chebyshev')` or `spectrum.eigenvector(3,32,50,{'chebyshev',60})`.

`Hs = spectrum.eigenvector(NSinusoids,SegmentLength,...
OverlapPercent,WindowName,SubspaceThreshold)` returns a spectrum object, `Hs`, with the specified subspace threshold.

`Hs = spectrum.eigenvector(NSinusoids,SegmentLength,...
OverlapPercent,WindowName,SubspaceThreshold,InputType)` returns a spectrum object, `Hs`, with the specified input type.

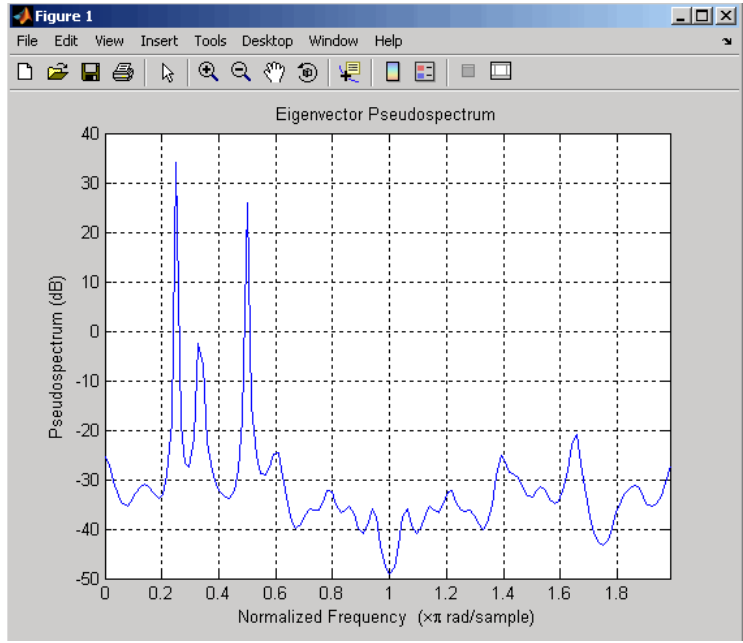
Note See `peig` for more information on the eigenanalysis algorithm.

Examples

Define a complex signal with three sinusoids, add noise, and view its pseudospectrum using eigenanalysis. Set the FFT length to 128.

```
randn('state',1);
n=0:99;
s=exp(i*pi/2*n)+2*exp(i*pi/4*n)+exp(i*pi/3*n)+randn(1,100);
```

```
Hs=spectrum.eigenvector(3,32,95,'rectangular',5);
pseudospectrum(Hs,s,'NFFT',128)
```



References

[1] Harris, F. J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform." *Proceedings of the IEEE*. Vol. 66 (January 1978).

See Also

`dspdata`, `spectrum`, `spectrum.music`, `spectrum.burg`, `spectrum.cov`, `spectrum.mcov`, `spectrum.yulear`, `spectrum.periodogram`, `spectrum.welch`, `spectrum.mtm`

Purpose

Modified covariance spectrum

Syntax

```
Hs = spectrum.mcov
Hs = spectrum.mcov(order)
```

Description

`Hs = spectrum.mcov` returns a default modified covariance spectrum object, `Hs`, that defines the parameters for the modified covariance spectral estimation algorithm. The modified covariance algorithm estimates the spectral content by fitting an auto-regressive (AR) linear prediction filter model of a given order to the signal.

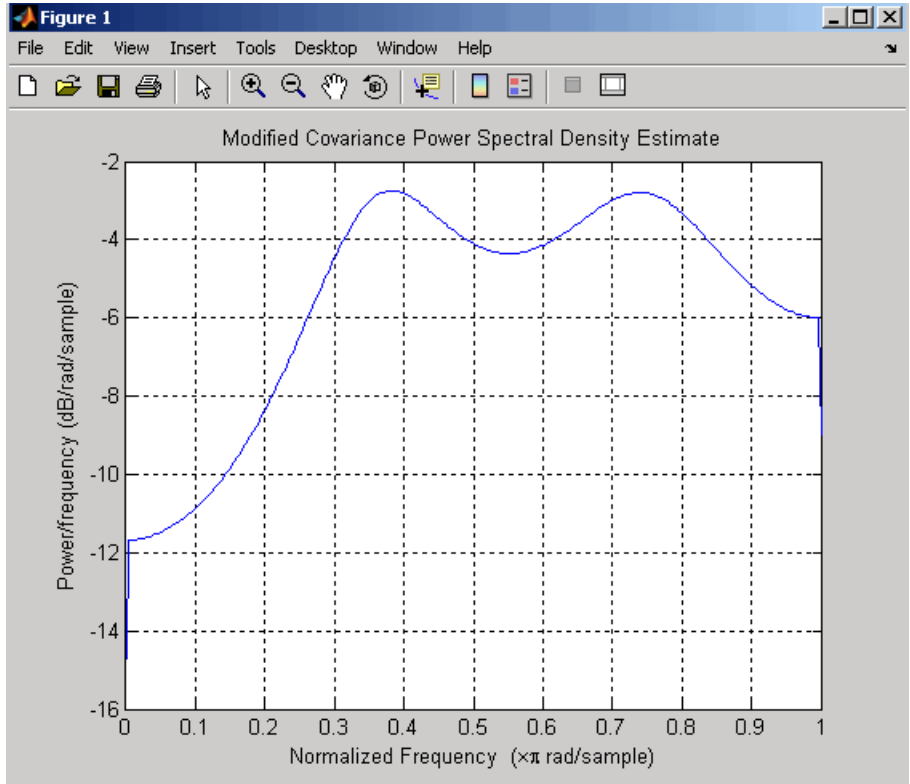
`Hs = spectrum.mcov(order)` returns a spectrum object, `Hs` with the specified `order`. The default value for `order` is 4.

Note See `pmcov` for more information on the modified covariance algorithm.

Examples

Define a fourth order auto-regressive model and view its power spectral density using the modified covariance algorithm.

```
randn('state',1);
x=randn(100,1);
x=filter(1,[1 1/2 1/3 1/4 1/5],x);   % 4th order AR filter
Hs=spectrum.mcov;                    % 4th order AR model
psd(Hs,x,'NFFT',512)
```



See Also

dspdata, spectrum, spectrum.burg, spectrum.cov, spectrum.yulear, spectrum.periodogram, spectrum.welch, spectrum.mtm, spectrum.eigenvector, spectrum.music

Purpose

Thompson multitaper spectrum

Syntax

```
Hs = spectrum.mtm
Hs = spectrum.mtm(TimeBW)
Hs = spectrum.mtm(DPSS,Concentrations)
Hs = spectrum.mtm(...,CombineMethod)
```

Description

`Hs = spectrum.mtm` returns a default Thompson multitaper spectrum object, `Hs` that defines the parameters for the Thompson multitaper spectral estimation algorithm, which uses a linear or nonlinear combination of modified periodograms. The periodograms are computed using a sequence of orthogonal tapers (windows in the frequency domain) specified from discrete prolate spheroidal sequences (`dpss`). This object uses the following default values:

Property Name	Default Value	Description
`TimeBW`	`4`	Product of time and bandwidth for the discrete prolate spheroidal sequences (or Slepian sequences) used as data windows
`CombineMethod`	`'adaptive'`	Algorithm for combining the individual spectral estimates. Valid values are `'adaptive'` — adaptive (nonlinear) `'unity'` — unity weights (linear) `'eigenvector'` — Eigenvalue weights (linear)

`Hs = spectrum.mtm(TimeBW)` returns a spectrum object, `Hs` with the specified time-bandwidth product.

`Hs = spectrum.mtm(DPSS,Concentrations)` returns a spectrum object, `Hs` with the specified dpss data tapers and their concentrations.

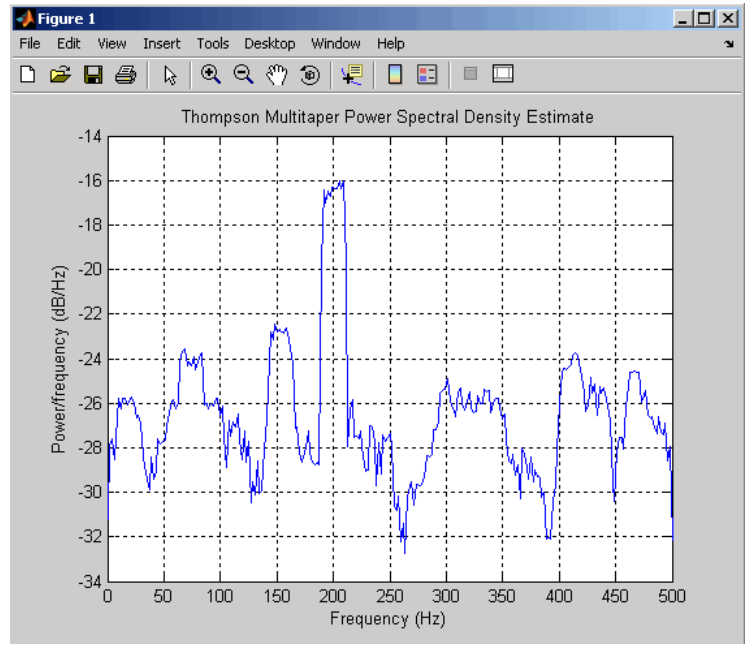
Note You can either specify the time-bandwidth product (`TimeBW`) or the `DPSS` data tapers and their `Concentrations`. See `dpss` and `pmtm` for more information.

`Hs = spectrum.mtm(...,CombineMethod)` returns a spectrum object, `Hs`, with the specified method for combining the spectral estimates. Refer to the table above for valid `CombineMethod` values.

Examples

Define a cosine of 200 Hz, add noise and view its power spectral density using the Thompson multitaper algorithm with a time-bandwidth product of 3.5.

```
Fs=1000;
t=0:1/Fs:.3;
x=cos(2*pi*t*200)+randn(size(t));
Hs=spectrum.mtm(3.5);
psd(Hs,x,'Fs',Fs)
```



The above example could be done by specifying the data tapers and concentrations instead of the time-bandwidth product.

```
Fs=1000;
t=0:1/Fs:.3;
x=cos(2*pi*t*200)+randn(size(t));
[e,v]=dpss(length(x),3.5);
Hs=spectrum.mtm(e,v);
psd(Hs,x,'Fs',Fs)
```

See Also

dspdata, spectrum, spectrum.periodogram, spectrum.welch, spectrum.burg, spectrum.cov, spectrum.mcov, spectrum.yulear, spectrum.eigenvector, spectrum.music

Purpose

Multiple signal classification spectrum

Syntax

```
Hs = spectrum.music
Hs = spectrum.music(NSinusoids)
Hs = spectrum.eigenvector(NSinusoids,SegmentLength)
Hs = spectrum.music(NSinusoids,SegmentLength,...
OverlapPercent)
Hs = spectrum.music(NSinusoids,SegmentLength,...
OverlapPercent,WindowName)
Hs = spectrum.music(NSinusoids,SegmentLength,...
OverlapPercent,WindowName,SubspaceThreshold)
Hs = spectrum.music(NSinusoids,SegmentLength,...
OverlapPercent,WindowName,SubspaceThreshold,InputType)
```

Description

`Hs = spectrum.music` returns a default multiple signal classification (MUSIC) spectrum object, `Hs`, that defines the parameters for the MUSIC spectral estimation algorithm, which uses Schmidt's eigenspace analysis algorithm. This object uses the following default values.

Default Values

Property Name	Default Value	Description
`NSinusoids`	`2`	Number of complex sinusoids
`SegmentLength`	`4`	Length of each of the time-based segments into which the input signal is divided.
`OverlapPercent`	`50`	Percent overlap between segments

Default Values (Continued)

Property Name	Default Value	Description
`WindowName`	`'Rectangular'`	Window name string or `'User Defined'` (see `window` for valid window names). For more information on each window, refer to its reference page). This argument can also be a cell array containing the window name string or `'User Defined'` and, if used for the particular window, an optional parameter value. The syntax is `{wname,wparam}`. You can use `set` to change the value of the additional parameter or to define the MATLAB expression and parameters for a user-defined window (see `spectrum` for information on using `set`).

Default Values (Continued)

Property Name	Default Value	Description
`SubspaceThreshold`	`0`	Threshold is the cutoff for signal and noise separation. The threshold is multiplied by λ_{min} , the smallest estimated eigenvalue of the signal's correlation matrix. Eigenvalues below the threshold (λ_{min}`*threshold`) are assigned to the noise subspace.
`InputType`	`'Vector'`	Type of input that will be used with this spectrum object. Valid values are `'Vector'`, `'DataMatrix'` and `'CorrelationMatrix'`.

`Hs = spectrum.music(NSinusoids)` returns a spectrum object, `Hs`, with the specified number of sinusoids and default values for all other properties. Refer to the table above for default values.

`Hs = spectrum.eigenvector(NSinusoids,SegmentLength)` returns a spectrum object, `Hs`, with the specified segment length.

`Hs = spectrum.music(NSinusoids,SegmentLength,...
OverlapPercent)` returns a spectrum object, `Hs`, with the specified overlap between segments.

`Hs = spectrum.music(NSinusoids,SegmentLength,...
OverlapPercent,WindowName)` returns a spectrum object, `Hs`, with the specified window.

Note Window names must be enclosed in single quotes, such as `spectrum.music(3,32,50,'chebyshev')` or `spectrum.music(3,32,50,{'chebyshev',60})`

`Hs = spectrum.music(NSinusoids,SegmentLength,...
OverlapPercent,WindowName,SubspaceThreshold)` returns a spectrum object, `Hs`, with the specified subspace threshold.

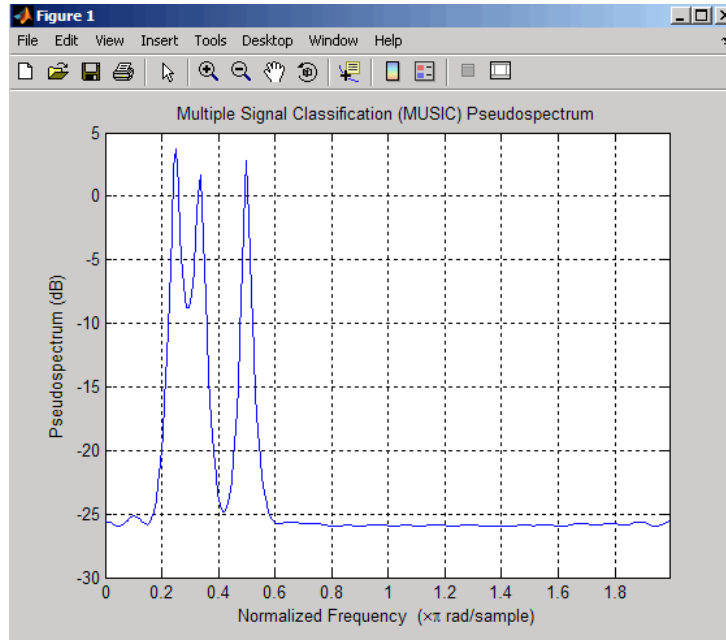
`Hs = spectrum.music(NSinusoids,SegmentLength,...
OverlapPercent,WindowName,SubspaceThreshold,InputType)` returns a spectrum object, `Hs`, with the specified input type.

Note See `pmusic` for more information on the MUSIC algorithm.

Examples

Define a complex signal with three sinusoids, add noise, and estimate its pseudospectrum using the MUSIC algorithm.

```
randn('state',1);
n=0:99;
s=exp(i*pi/2*n)+2*exp(i*pi/4*n)+exp(i*pi/3*n)+randn(1,100);
Hs=spectrum.music(3,20);
pseudospectrum(Hs,s)
```



References

[1] Harris, F. J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform." *Proceedings of the IEEE*. Vol. 66 (January 1978).

See Also

`dspdata`, `spectrum`, `spectrum.eigenvector`, `spectrum.burg`, `spectrum.cov`, `spectrum.mcov`, `spectrum.yulear`, `spectrum.periodogram`, `spectrum.welch`, `spectrum.mtm`

Purpose

Periodogram spectrum

Syntax

```
Hs = spectrum.periodogram
Hs = spectrum.periodogram(winname)
Hs = spectrum.periodogram({winname,winparameter})
```

Description

`Hs = spectrum.periodogram` returns a default periodogram spectrum object, `Hs`, that defines the parameters for the periodogram spectral estimation method. This default object uses a rectangular window and a default FFT length equal to the next power of 2 (`NextPow2`) that is greater than the input length.

`Hs = spectrum.periodogram(winname)` returns a spectrum object, `Hs`, that uses the specified window. If the window uses an optional associated window parameter, it is set to the default value. This object uses the default FFT length.

`Hs = spectrum.periodogram({winname,winparameter})` returns a spectrum object, `Hs`, that uses the specified window and optional associated window parameter, if any. You specify the window and window parameter in a cell array with a `windowname` string and the parameter value. This object uses the default FFT length.

Valid `windowname` strings are:

```
'Bartlett'
'Bartlett-Hanning'
'Blackman'
'Blackman-Harris'
'Bohman'
'Chebyshev'
'Flat Top'
'Gaussian'
'Hamming'
'Hann'
'Kaiser'
'Nuttall'
'Parzen'
```

```
'Rectangular'
'Triangular'
'Tukey'
'User Defined'
```

See `window` and the corresponding window function page for window parameter information.

You can use `set` to change the value of the additional parameter or to define the MATLAB expression and parameters for a user-defined window (see `spectrum` for information on using `set`).

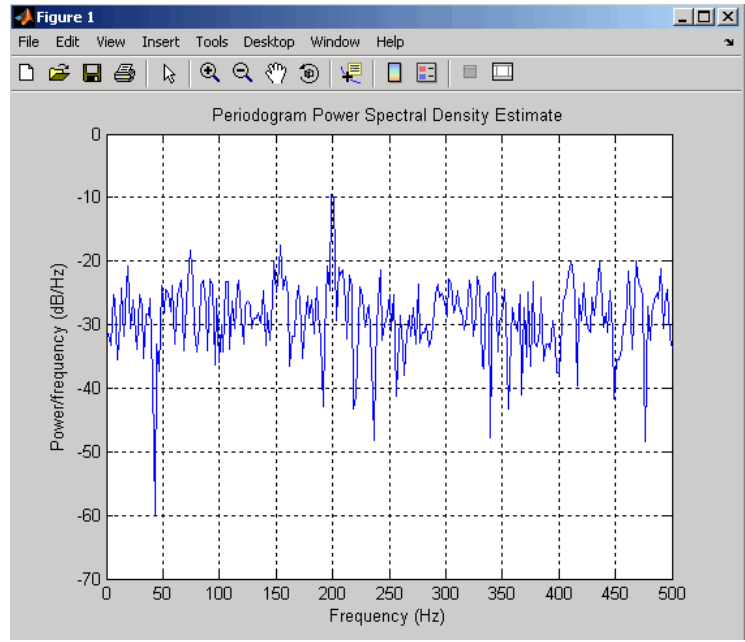
Note Window names must be enclosed in single quotes, such as `spectrum.periodogram('tukey') or spectrum.periodogram({'tukey',0.7})`.

Note See `periodogram` for more information on the periodogram algorithm.

Examples

Define a cosine of 200 Hz, add noise and view its spectral content using the periodogram spectral estimation technique.

```
Fs=1000;
t=0:1/Fs:.3;
x=cos(2*pi*t*200)+randn(size(t));
Hs=spectrum.periodogram;      % Use default values
psd(Hs,x,'Fs',Fs)
```



References

[1] Harris, F. J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform." *Proceedings of the IEEE*. Vol. 66 (January 1978).

See Also

`dspdata`, `spectrum`, `spectrum.welch`, `spectrum.mtm`, `spectrum.burg`, `spectrum.cov`, `spectrum.mcov`, `spectrum.yulear`, `spectrum.eigenvector`, `spectrum.music`

Purpose

Welch spectrum

Syntax

```
Hs = spectrum.welch
Hs = spectrum.welch(WindowName)
Hs = spectrum.welch(WindowName,SegmentLength)
Hs = spectrum.welch(WindowName,SegmentLength,OverlapPercent)
```

Description

`Hs = spectrum.welch` returns a default Welch spectrum object, `Hs`, that defines the parameters for Welch's averaged, modified periodogram spectral estimation method. The object uses these default values.

Property Name	Default Value	Description
`{WindowName, winparam}` Cell array containing `WindowName` and optional window parameter	`'Hamming'`, `SamplingFlag: symmetric`	Cell array containing the window name string or `'User Defined'` and, if used for the particular window, an optional parameter value. (See `window` for valid window names and for more information on each window, refer to its reference page.) You can use `set` to change the value of the additional parameter or to define the MATLAB expression and parameters for a user-defined window. (See `spectrum` for information on using `set`.)

Property Name	Default Value	Description
WindowName	'Hamming', SamplingFlag: symmetric	Valid windowname strings are: 'Bartlett' 'Bartlett-Hanning' 'Blackman' 'Blackman-Harris' 'Bohman' 'Chebyshev' 'Flat Top' 'Gaussian' 'Hamming' 'Hann' 'Kaiser' 'Nuttall' 'Parzen' 'Rectangular' 'Triangular' 'Tukey' 'User Defined' Window names must be enclosed in single quotes, such as spectrum.welch('tukey') or spectrum.welch({'tukey',0.7}). See window and the corresponding window function page for window parameter information. You can use set to change the value of the additional window parameter or to define

Property Name	Default Value	Description
		the MATLAB expression and parameters for a user-defined window (see `spectrum` for information on using `set`).
`SegmentLength`	`64`	Length of each of the time-based segments into which the input signal is divided. A modified periodogram is computed on each segment and the average of the periodograms forms the spectral estimate. Choosing the segment length is a compromise between estimate reliability (shorter segments) and frequency resolution (longer segments). A long segment length producese better resolution while a short segment length produces more averages, and therefore a decrease in the variance.
`OverlapPercent`	`50%`	Percent overlap between segments

`Hs = spectrum.welch(WindowName)` returns a spectrum object, `Hs`, using Welch's method with the specified window and the default values for all other parameters. To specify parameters for a window, use a cell array formatted as `spectrum.welch({WindowName,winparam})`.

`Hs = spectrum.welch(WindowName,SegmentLength)` returns a spectrum object, `Hs` with the specified segment length.

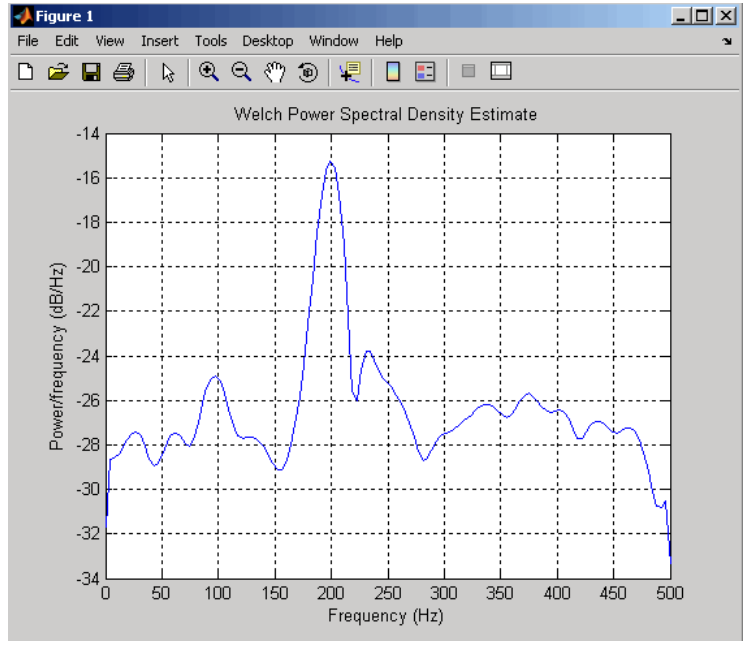
`Hs = spectrum.welch(WindowName,SegmentLength,OverlapPercent)` returns a spectrum object, `Hs` with the specified percentage overlap between segments.

Note See `pwelch` for more information on the Welch algorithm.

Examples

Define a cosine of 200 Hz, add noise and view its spectral content using the Welch algorithm.

```
Fs=1000;
t=0:1/Fs:.3;
x=cos(2*pi*t*200)+randn(size(t));
Hs=spectrum.welch;
psd(Hs,x,'Fs',Fs)
```



The following example produces a result similar to the obsoleted `spectrum` function, which used a Hann window as the default.

```
Fs = 1000;
t = 0:1/Fs:.3;
x=cos(2*pi*t*200)+randn(size(t));
window=33;
noverlap=32;
nfft=4097;
h = spectrum.welch('Hann',window,100*noverlap/window);
hpsd = psd(h,x,'NFFT',nfft,'Fs',Fs);
Pw = hpsd.Data;
Fw = hpsd.Frequencies;
```

References

[1] Harris, F. J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform." *Proceedings of the IEEE*. Vol. 66 (January 1978).

See Also

`dspdata`, `spectrum`, `spectrum.periodogram`, `spectrum.mtm`, `spectrum.burg`, `spectrum.cov`, `spectrum.mcov`, `spectrum.yulear`, `spectrum.eigenvector`, `spectrum.music`

spectrum.yulear

Purpose

Yule-Walker spectrum object

Syntax

```
Hs = spectrum.yulear
Hs = spectrum.yulear(order)
```

Description

`Hs = spectrum.yulear` returns a default Yule-Walker spectrum object, `Hs`, that defines the parameters for the Yule-Walker spectral estimation algorithm. This method is also called the auto-correlation or windowed method. The Yule-Walker algorithm estimates the spectral content by fitting an auto-regressive (AR) linear prediction filter model of a given `order` to the signal. This leads to a set of Yule-Walker equations, which are solved using Levinson-Durbin recursion.

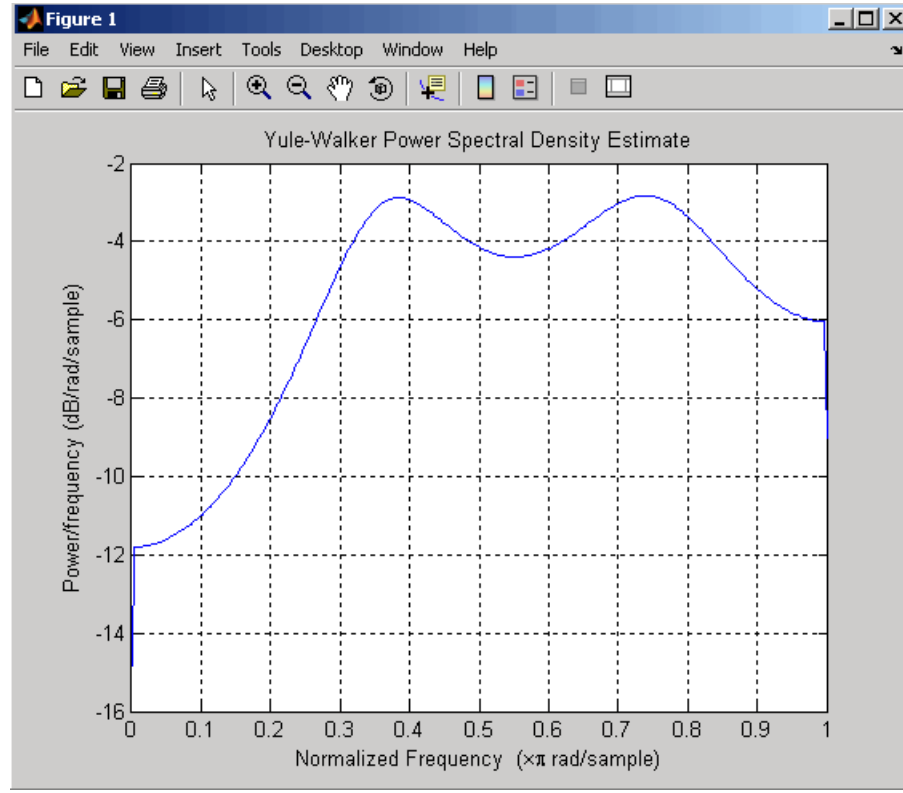
`Hs = spectrum.yulear(order)` returns a spectrum object, `Hs`, with the specified `order`. The default value for `order` is 4.

Note See `pyulear` for more information on the Yule-Walker algorithm.

Examples

Define a fourth order auto-regressive model and view its spectral content using the Yule-Walker algorithm.

```
randn('state',1);
x=randn(100,1);
x=filter(1,[1 1/2 1/3 1/4 1/5],x);   % 4th order AR filter
Hs=spectrum.yulear;                  % 4th order AR model
psd(Hs,x,'NFFT',512)
```



See Also

dspdata, spectrum, spectrum.burg, spectrum.cov, spectrum.mcov, spectrum.periodogram, spectrum.welch, spectrum.mtm, spectrum.eigenvector, spectrum.music

Purpose

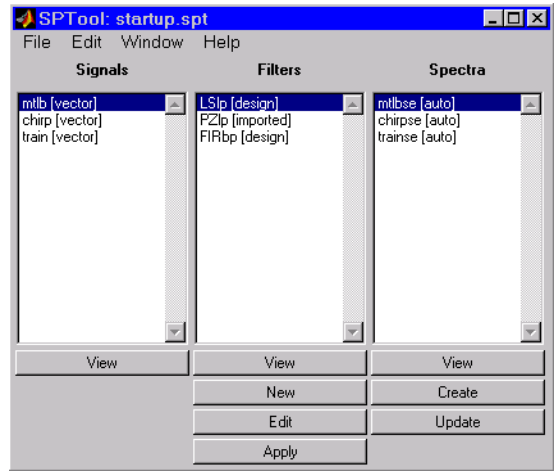
Open interactive digital signal processing tool

Syntax

```
sptool
```

Description

`sptool` opens SPTool, a graphical user interface (GUI) that manages a suite of four other GUIs: Signal Browser, Filter Designer, FVTool, and Spectrum Viewer. These GUIs provide access to many of the signal, filter, and spectral analysis functions in the toolbox. When you type `sptool` at the command line, the SPTool GUI opens.



Using SPTool you can

- Analyze signals listed in the **Signals** list box with the Signal Browser
- Design or edit filters with the Filter Designer (includes a Pole/Zero Editor)
- Analyze filter responses for filters listed in the **Filters** list box with FVTool
- Apply filters in the **Filters** list box to signals in the **Signals** list box
- Create and analyze signal spectra with the Spectrum Viewer

- Print the Signal Browser, Filter Designer, and Spectrum Viewer

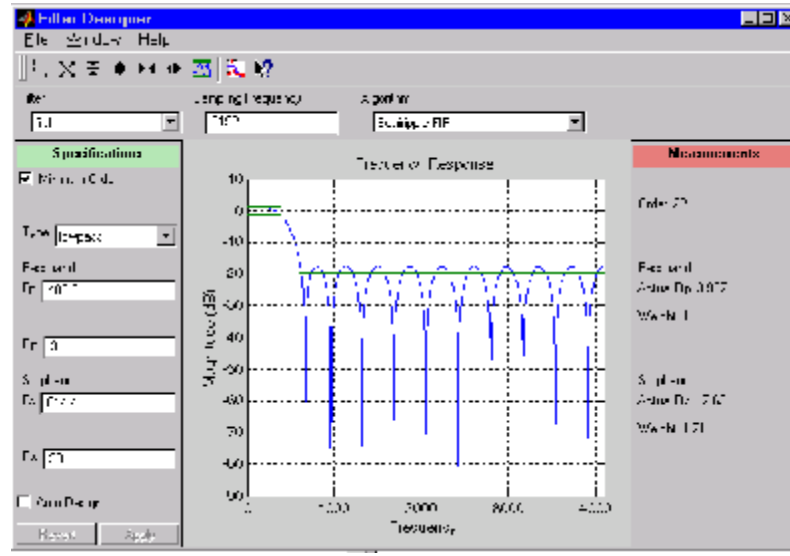
You can activate the four integrated signal processing GUIs from SPTool.

Signal Browser

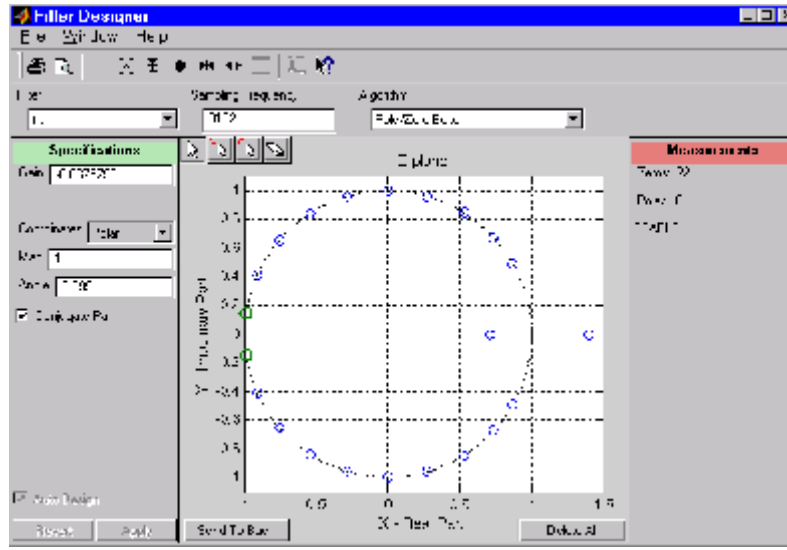
The Signal Browser allows you to view, measure, and analyze the time-domain information of one or more signals. To activate the Signal Browser, press the **View** button under the **Signals** list box in SPTool.

Filter Designer

The Filter Designer allows you to design and edit FIR and IIR filters of various lengths and types, with standard (lowpass, highpass, bandpass, bandstop, and multiband) configurations. To activate the Filter Designer, press either the **New** button or the **Edit** button under the **Filters** list box in SPTool.

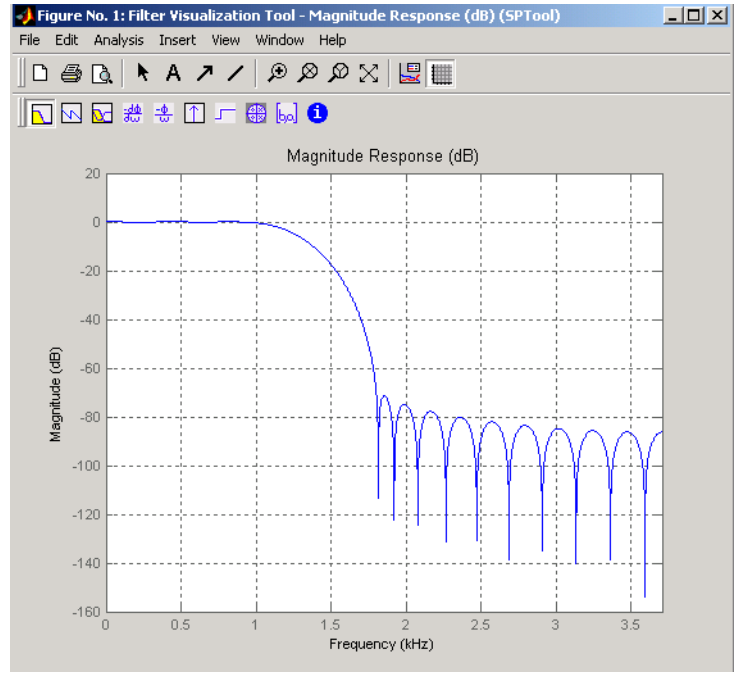


The Filter Designer has a Pole/Zero Editor you can access from the **Algorithms** pulldown.



Filter Visualization Tool

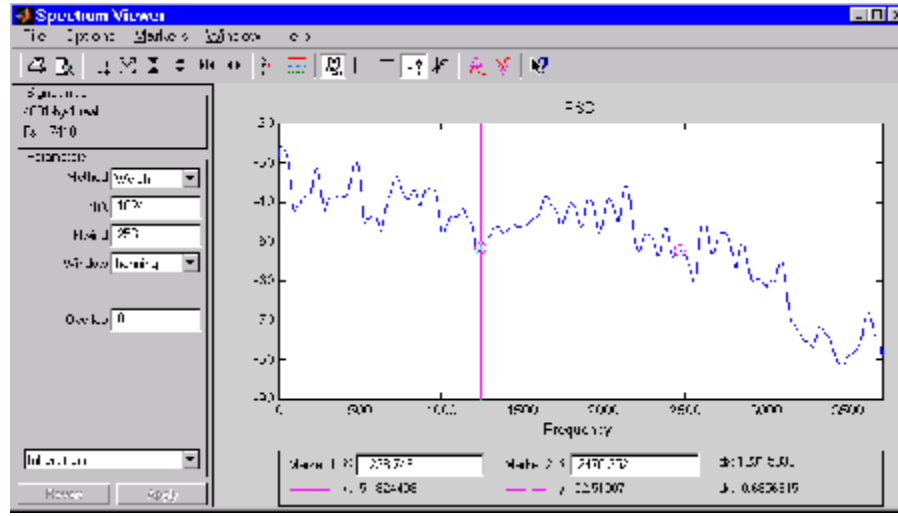
The Filter Visualization Tool (`fvtool`) allows you to view the characteristics of a designed or imported filter, including its magnitude response, phase response, group delay, phase delay, pole-zero plot, impulse response, and step response. To activate FVTool, click the **View** button under the **Filters** list box in SPTool.



Spectrum Viewer

The Spectrum Viewer allows you to analyze frequency-domain data graphically using a variety of methods of spectral density estimation, including the Burg method, the FFT method, the multitaper method, the MUSIC eigenvector method, Welch's method, and the Yule-Walker autoregressive method. To activate the Spectrum Viewer:

- Click the **Create** button under the **Spectra** list box to compute the power spectral density for a signal selected in the **Signals** list box in SPTool. You may need to click **Apply** to view the spectra.
- Click the **View** button to analyze spectra selected under the **Spectra** list box in SPTool.
- Click the **Update** button under the **Spectra** list box in SPTool to modify a selected power spectral density signal.



In addition, you can right-click in any plot display area of the GUIs to modify signal properties.

See Chapter 8, "SPTool: A Signal Processing GUI Suite" for a full discussion of how to use SPTool.

See Also

`fdatool`, `fvtool`

Purpose

Square wave

Syntax

```
x = square(t)
x = square(t,duty)
```

Description

`x = square(t)` generates a square wave with period 2π for the elements of time vector `t`. `square(t)` is similar to `sin(t)`, but creates a square wave with peaks of ±1 instead of a sine wave.

`x = square(t,duty)` generates a square wave with specified duty cycle, `duty`, which is a number between 0 and 100. The *duty cycle* is the percent of the period in which the signal is positive.

See Also

`chirp`, `cos`, `diric`, `gauspuls`, `pulstran`, `rectpuls`, `sawtooth`, `sin`, `tripuls`

Purpose

Convert digital filter state-space parameters to second-order sections form

Syntax

```
[sos,g] = ss2sos(A,B,C,D)
[sos,g] = ss2sos(A,B,C,D,iu)
[sos,g] = ss2sos(A,B,C,D,'order')
[sos,g] = ss2sos(A,B,C,D,iu,'order')
[sos,g] = ss2sos(A,B,C,D,iu,'order','scale')
sos = ss2sos(...)
```

Description

`ss2sos` converts a state-space representation of a given digital filter to an equivalent second-order section representation.

`[sos,g] = ss2sos(A,B,C,D)` finds a matrix `sos` in second-order section form with gain `g` that is equivalent to the state-space system represented by input arguments `A`, `B`, `C`, and `D`. The input system must be single output and real. `sos` is an L-by-6 matrix

$$sos = \begin{bmatrix} b_{01} & b_{11} & b_{21} & 1 & a_{11} & a_{21} \\ b_{02} & b_{12} & b_{22} & 1 & a_{12} & a_{22} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{0L} & b_{1L} & b_{2L} & 1 & a_{1L} & a_{2L} \end{bmatrix}$$

whose rows contain the numerator and denominator coefficients b_{ik} and a_{ik} of the second-order sections of $H(z)$.

$$H(z) = g\prod_{k=1}^{L} H_k(z) = g\prod_{k=1}^{L} \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2}}{1 + a_{1k}z^{-1} + a_{2k}z^{-2}}$$

`[sos,g] = ss2sos(A,B,C,D,iu)` specifies a scalar `iu` that determines which input of the state-space system `A`, `B`, `C`, `D` is used in the conversion. The default for `iu` is 1.

`[sos,g] = ss2sos(A,B,C,D,'order')` and

`[sos,g] = ss2sos(A,B,C,D,iu,'order')` specify the order of the rows in `sos`, where `'order'` is

- `'down'`, to order the sections so the first row of `sos` contains the poles closest to the unit circle
- `'up'`, to order the sections so the first row of `sos` contains the poles farthest from the unit circle (default)

The zeros are always paired with the poles closest to them.

`[sos,g] = ss2sos(A,B,C,D,iu,'order','scale')` specifies the desired scaling of the gain and the numerator coefficients of all second-order sections, where `'scale'` is

- `'none'`, to apply no scaling (default)
- `'inf'`, to apply infinity-norm scaling
- `'two'`, to apply 2-norm scaling

Using infinity-norm scaling in conjunction with `up`-ordering minimizes the probability of overflow in the realization. Using 2-norm scaling in conjunction with `down`-ordering minimizes the peak round-off noise.

Note Infinity-norm and 2-norm scaling are appropriate only for direct-form II implementations.

`sos = ss2sos(...)` embeds the overall system gain, `g`, in the first section, $H_1(z)$, so that

$$H(z) = \prod_{k=1}^{L} H_k(z)$$

Note Embedding the gain in the first section when scaling a direct-form II structure is not recommended and may result in erratic scaling. To avoid embedding the gain, use `ss2sos` with two outputs.

Examples

Find a second-order section form of a Butterworth lowpass filter:

```
[A,B,C,D] = butter(5,0.2);
sos = ss2sos(A,B,C,D)
sos =
    0.0013   0.0013        0   1.0000  -0.5095        0
    1.0000   2.0008   1.0008   1.0000  -1.0966   0.3554
    1.0000   1.9979   0.9979   1.0000  -1.3693   0.6926
```

Algorithm

`ss2sos` uses a four-step algorithm to determine the second-order section representation for an input state-space system:

1. It finds the poles and zeros of the system given by `A`, `B`, `C`, and `D`.
2. It uses the function `zp2sos`, which first groups the zeros and poles into complex conjugate pairs using the `cplxpair` function. `zp2sos` then forms the second-order sections by matching the pole and zero pairs according to the following rules:
   a. Match the poles closest to the unit circle with the zeros closest to those poles.
   b. Match the poles next closest to the unit circle with the zeros closest to those poles.
   c. Continue until all of the poles and zeros are matched.

   `ss2sos` groups real poles into sections with the real poles closest to them in absolute value. The same rule holds for real zeros.
3. It orders the sections according to the proximity of the pole pairs to the unit circle. `ss2sos` normally orders the sections with poles closest

to the unit circle last in the cascade. You can tell `ss2sos` to order the sections in the reverse order by specifying the `'down'` flag.

4 `ss2sos` scales the sections by the norm specified in the `'scale'` argument. For arbitrary $H(\omega)$, the scaling is defined by

$$\|H\|_p = \left[\frac{1}{2\pi}\int_0^{2\pi} |H(\omega)|^p d\omega\right]^{\frac{1}{p}}$$

where p can be either ∞ or 2. See the references for details. This scaling is an attempt to minimize overflow or peak round-off noise in fixed point filter implementations.

Diagnostics

If there is more than one input to the system, `ss2sos` gives the following error message:

```
State-space system must have only one input.
```

References

[1] Jackson, L.B., *Digital Filters and Signal Processing, 3rd ed.*, Kluwer Academic Publishers, Boston, 1996. Chapter 11.

[2] Mitra, S.K., *Digital Signal Processing: A Computer-Based Approach*, McGraw-Hill, New York, 1998. Chapter 9.

[3] Vaidyanathan, P.P., "Robust Digital Filter Structures," *Handbook for Digital Signal Processing*, S.K. Mitra and J.F. Kaiser, ed., John Wiley & Sons, New York, 1993, Chapter 7.

See Also

`cplxpair`, `sos2ss`, `ss2tf`, `ss2zp`, `tf2sos`, `zp2sos`

Purpose

Convert state-space filter parameters to transfer function form

Syntax

```
[b,a] = ss2tf(A,B,C,D,iu)
```

Description

`ss2tf` converts a state-space representation of a given system to an equivalent transfer function representation.

`[b,a] = ss2tf(A,B,C,D,iu)` returns the transfer function

$$H(s) = \frac{B(s)}{A(s)} = C(sI - A)^{-1}B + D$$

of the system

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

from the `iu`-th input. Vector `a` contains the coefficients of the denominator in descending powers of s. The numerator coefficients are returned in array `b` with as many rows as there are outputs y. `ss2tf` also works with systems in discrete time, in which case it returns the z-transform representation.

The `ss2tf` function is part of the standard MATLAB language.

Algorithm

The `ss2tf` function uses `poly` to find the characteristic polynomial det(sI-A) and the equality:

$$H(s) = C(sI - A)^{-1}B = \frac{\det(sI - A + BC) - \det(sI - A)}{\det(sI - A)}$$

See Also

`latc2tf`, `sos2tf`, `ss2sos`, `ss2zp`, `tf2ss`, `zp2tf`

Purpose

Convert state-space filter parameters to zero-pole-gain form

Syntax

```
[z,p,k] = ss2zp(A,B,C,D,i)
```

Description

`ss2zp` converts a state-space representation of a given system to an equivalent zero-pole-gain representation. The zeros, poles, and gains of state-space systems represent the transfer function in factored form.

`[z,p,k] = ss2zp(A,B,C,D,i)` calculates the transfer function in factored form

$$H(s) = \frac{Z(s)}{P(s)} = k\frac{(s-z_1)(s-z_2)\cdots(s-z_n)}{(s-p_1)(s-p_2)\cdots(s-p_n)}$$

of the continuous-time system

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

from the `i`th input (using the `i`th columns of `B` and `D`). The column vector `p` contains the pole locations of the denominator coefficients of the transfer function. The matrix `z` contains the numerator zeros in its columns, with as many columns as there are outputs *y* (rows in `C`). The column vector `k` contains the gains for each numerator transfer function.

`ss2zp` also works for discrete time systems. The input state-space system must be real.

The `ss2zp` function is part of the standard MATLAB language.

Examples

Here are two ways of finding the zeros, poles, and gains of a discrete-time transfer function:

$$H(z) = \frac{2 + 3z^{-1}}{1 + 0.4z^{-1} + z^{-2}}$$

```
b = [2 3 0];
a = [1 0.4 1];
```

```
[z,p,k] = tf2zp(b,a)
z =
    0.0000
   -1.5000
p =
    -0.2000 + 0.9798i
    -0.2000 - 0.9798i
k =
    2
[A,B,C,D] = tf2ss(b,a);
[z,p,k] = ss2zp(A,B,C,D,1)
z =
    0.0000
   -1.5000
p =
    -0.2000 + 0.9798i
    -0.2000 - 0.9798i
k =
    2
```

Algorithm

`ss2zp` finds the poles from the eigenvalues of the `A` array. The zeros are the finite solutions to a generalized eigenvalue problem:

```
z = eig([A B;C D], diag([ones(1,n) 0]);
```

In many situations this algorithm produces spurious large, but finite, zeros. `ss2zp` interprets these large zeros as infinite.

`ss2zp` finds the gains by solving for the first nonzero Markov parameters.

References

[1] Laub, A.J., and B.C. Moore, "Calculation of Transmission Zeros Using QZ Techniques," *Automatica 14* (1978), p. 557.

See Also

`sos2zp`, `ss2sos`, `ss2tf`, `tf2zp`, `tf2zpk`, `zp2ss`

Purpose

Step response of digital filter

Syntax

```
[h,t] = stepz(b,a)
[h,t] = stepz(b,a,n)
[h,t] = stepz(b,a,n,fs)
stepz(b,a)
stepz(Hd)
```

Description

`[h,t] = stepz(b,a)` computes the step response of the filter with numerator coefficients `b` and denominator coefficients `a`. `stepz` chooses the number of samples and returns the response in the column vector `h` and sample times in the column vector `t` (where `t = [0:n-1]'`, and `n = length(t)` is computed automatically).

`[h,t] = stepz(b,a,n)` computes the first n samples of the step response when n is an integer (`t = [0:n-1]'`). I

`[h,t] = stepz(b,a,n,fs)` computes n samples and produces a vector `t` of length `n` so that the samples are spaced `1/fs` units apart. `fs` is assumed to be in Hz.

`stepz(b,a)` with no output arguments plots the step response in the current figure window.

`stepz(Hd)` plots the step response of the filter and displays the plot in `fvtool`. The input `Hd` is a `dfilt` filter object or an array of `dfilt` filter objects. Note that if you have Filter Design Toolbox product installed and are using a `dfilt` object with fixed-point properties, its filter internals are not used when calculating the step response.

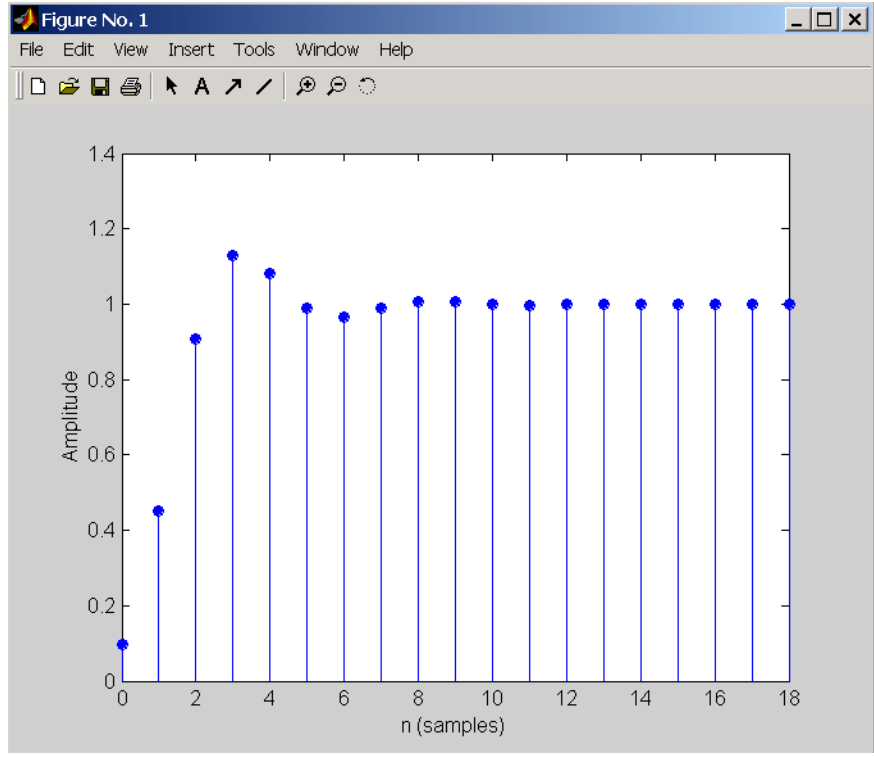
`stepz` works for both real and complex input systems.

Examples

Example 1

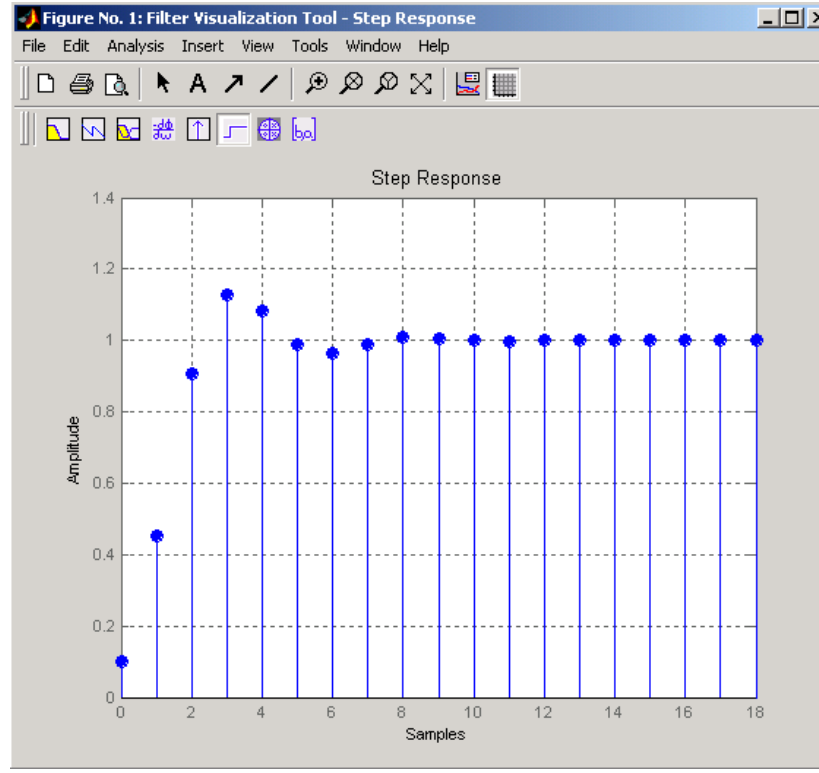
Plot the step response of a Butterworth filter:

```
[b,a] = butter(3,.4);
stepz(b,a)
```



The same example using a `dfilt` object and displaying the result in the Filter Visualization Tool (`fvtool`) is

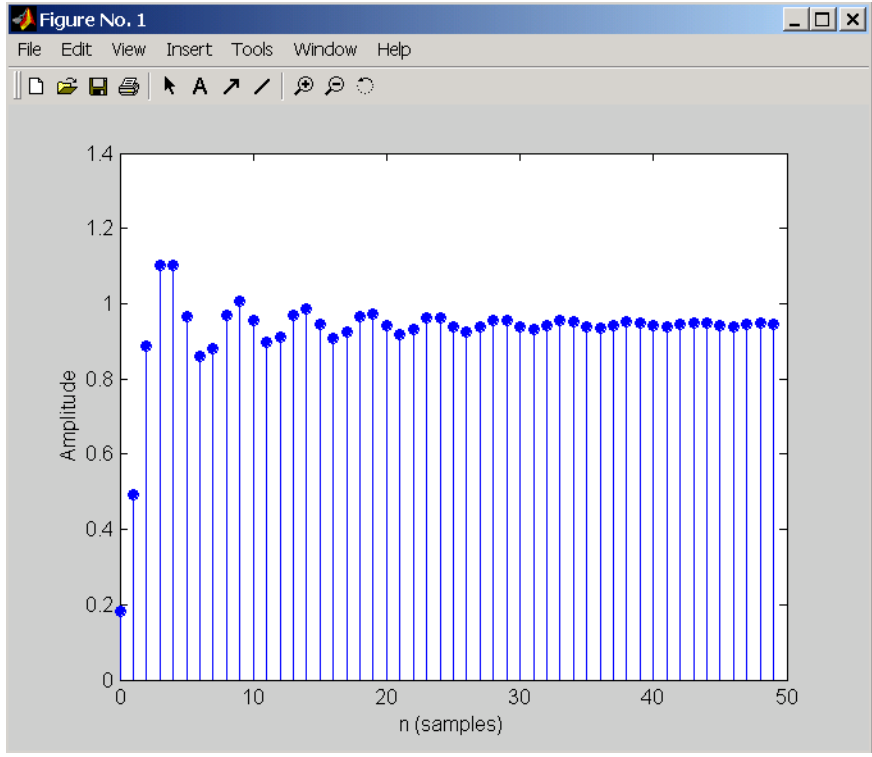
```
[b,a] = butter(3,.4);
Hd=dfilt.df1(b,a);
stepz(Hd)
```



Example 2

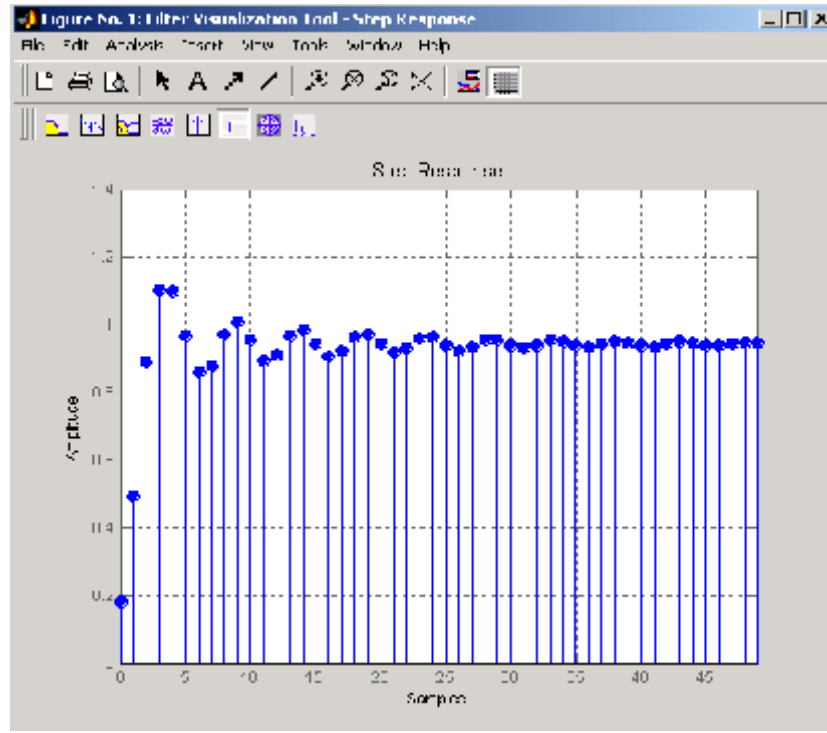
Plot the first 50 samples of the step response of a fourth-order lowpass elliptic filter with cutoff frequency of 0.4 times the Nyquist frequency:

```
[b,a] = ellip(4,0.5,20,0.4);
stepz(b,a,50)
```



The same example using a `dfilt` object and displaying the result in the Filter Visualization Tool (`fvtool`) is

```
[b,a] = ellip(4,0.5,20,0.4);
Hd=dfilt.df1(b,a);
stepz(Hd,50)
```



Algorithm

`stepz` filters a length `n` step sequence using

```
filter(b,a,ones(1,n))
```

and plots the results using `stem`.

To compute `n` in the auto-length case, `stepz` either uses `n = length(b)` for the FIR case or first finds the poles using `p = roots(a)`, if `length(a)` is greater than `1`.

If the filter is unstable, `n` is chosen to be the point at which the term from the largest pole reaches `10^6` times its original value.

If the filter is stable, `n` is chosen to be the point at which the term due to the largest amplitude pole is `5*10^-5` of its original amplitude.

If the filter is oscillatory (poles on the unit circle only), `stepz` computes five periods of the slowest oscillation.

If the filter has both oscillatory and damped terms, `n` is chosen to equal five periods of the slowest oscillation or the point at which the term due to the largest (nonunity) amplitude pole is `5*10^-5` of its original amplitude, whichever is greater.

`stepz` also allows for delays in the numerator polynomial. The number of delays is incorporated into the computation for the number of samples.

See Also

`freqz`, `grpdelay`, `impz`, `phasez`, `zplane`

Purpose

Compute linear model using Steiglitz-McBride iteration

Syntax

```
[b,a] = stmcb(h,nb,na)
[b,a] = stmcb(y,x,nb,na)
[b,a] = stmcb(h,nb,na,niter)
[b,a] = stmcb(y,x,nb,na,niter)
[b,a] = stmcb(h,nb,na,niter,ai)
[b,a] = stmcb(y,x,nb,na,niter,ai)
```

Description

Steiglitz-McBride iteration is an algorithm for finding an IIR filter with a prescribed time domain impulse response. It has applications in both filter design and system identification (parametric modeling).

`[b,a] = stmcb(h,nb,na)` finds the coefficients `b` and `a` of the system $b(z)/a(z)$ with approximate impulse response `h`, exactly `nb` zeros, and exactly `na` poles.

`[b,a] = stmcb(y,x,nb,na)` finds the system coefficients `b` and `a` of the system that, given `x` as input, has `y` as output. `x` and `y` must be the same length.

`[b,a] = stmcb(h,nb,na,niter)` and

`[b,a] = stmcb(y,x,nb,na,niter)` use `niter` iterations. The default for `niter` is 5.

`[b,a] = stmcb(h,nb,na,niter,ai)` and

`[b,a] = stmcb(y,x,nb,na,niter,ai)` use the vector `ai` as the initial estimate of the denominator coefficients. If `ai` is not specified, `stmcb` uses the output argument from `[b,ai] = prony(h,0,na)` as the vector `ai`.

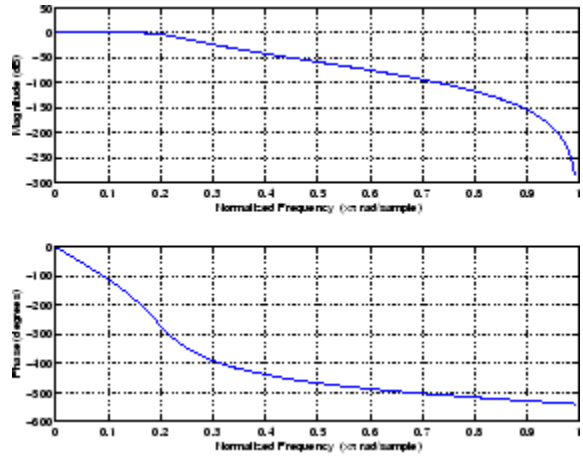
`stmcb` returns the IIR filter coefficients in length `nb+1` and `na+1` row vectors `b` and `a`. The filter coefficients are ordered in descending powers of z.

$$H(z) = \frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \cdots + b(nb+1)z^{-nb}}{a(1) + a(2)z^{-1} + \cdots + a(na+1)z^{-na}}$$

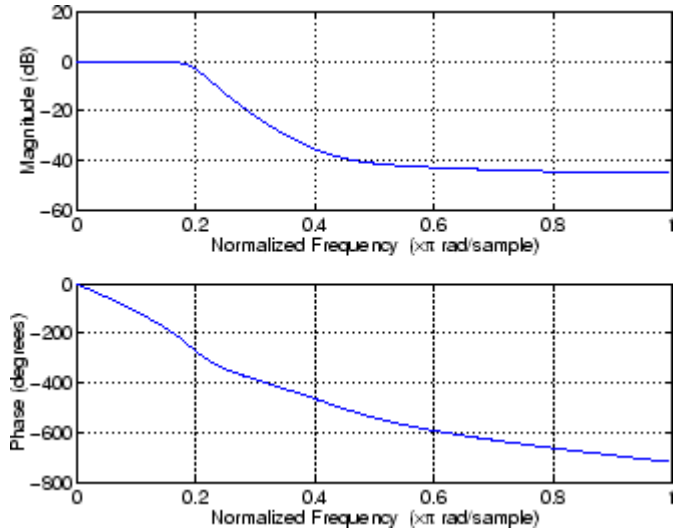
Examples

Approximate the impulse response of a Butterworth filter with a system of lower order:

```
[b,a] = butter(6,0.2);
h = filter(b,a,[1 zeros(1,100)]);
freqz(b,a,128)
```




```
[bb,aa] = stmcb(h,4,4);
freqz(bb,aa,128)
```



Algorithm

`stmcb` attempts to minimize the squared error between the impulse response h of $b(z)/a(z)$ and the input signal x.

$$\min_{a,b} \sum_{i=0}^{\infty} |x(i) - h(i)|^2$$

`stmcb` iterates using two steps:

1. It prefilters `h` and `x` using $1/a(z)$.
2. It solves a system of linear equations for `b` and `a` using $\backslash$.

`stmcb` repeats this process `niter` times. No checking is done to see if the `b` and `a` coefficients have converged in fewer than `niter` iterations.

Diagnostics

If `x` and `y` have different lengths, `stmcb` produces this error message,

```
Input signal X and output signal Y must
have the same length.
```

References

[1] Steiglitz, K., and L.E. McBride, "A Technique for the Identification of Linear Systems," *IEEE Trans. Automatic Control, Vol. AC-10* (1965), pp. 461-464.

[2] Ljung, L., *System Identification: Theory for the User*, Prentice-Hall, Englewood Cliffs, NJ, 1987, p. 297.

See Also

`levinson`, `lpc`, `aryule`, `prony`

Purpose

Strip plot

Syntax

```
strips(x)
strips(x,n)
strips(x,sd,fs)
strips(x,sd,fs,scale)
```

Description

`strips(x)` plots vector `x` in horizontal strips of length 250. If `x` is a matrix, `strips(x)` plots each column of `x`. The left-most column (column 1) is the top horizontal strip.

`strips(x,n)` plots vector `x` in strips that are each `n` samples long.

`strips(x,sd,fs)` plots vector `x` in strips of duration `sd` seconds, given a sampling frequency of `fs` samples per second.

`strips(x,sd,fs,scale)` scales the vertical axes.

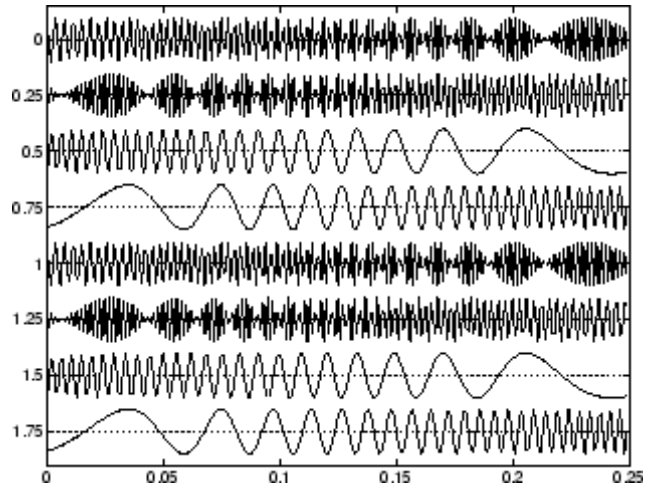
If `x` is a matrix, `strips(x,n)`, `strips(x,sd,fs)`, and `strips(x,sd,fs,scale)` plot the different columns of `x` on the same strip plot.

`strips` ignores the imaginary part of complex-valued `x`.

Examples

Plot two seconds of a frequency modulated sinusoid in 0.25 second strips:

```
fs = 1000;                          % Sampling frequency
t = 0:1/fs:2;                       % Time vector
x = vco(sin(2*pi*t),[10 490],fs);  % FM waveform
strips(x,0.25,fs)
```



See Also

`plot`, `stem`

Purpose

Taylor window

Syntax

```
w = taylorwin(n)
w = taylorwin(n,nbar)
w = taylorwin(n,nbar,sll)
```

Description

Taylor windows are similar to Chebyshev windows. While a Chebyshev window has the narrowest possible mainlobe for a specified sidelobe level, a Taylor window allows you to make tradeoffs between the mainlobe width and sidelobe level. The Taylor distribution avoids edge discontinuities, so Taylor window sidelobes decrease monotonically. Taylor window coefficients are not normalized. Taylor windows are typically used in radar applications, such as weighting synthetic aperature radar images and antenna design.

`w = taylorwin(n)` returns an `n`-point Taylor window in a column vector `w`. The values in this vector are the window weights or coefficients. `n` must be a positive integer. The default value for the number of approximately equal height sidelobes (`nbar`) is 4 and for the maximum sidelobe level (`sll`) is -30.

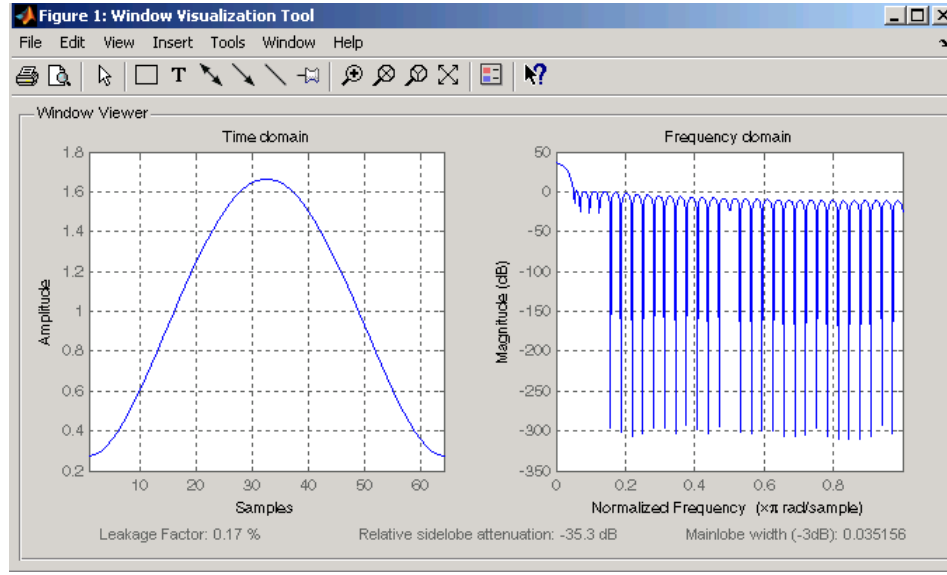
`w = taylorwin(n,nbar)` returns an `n`-point Taylor window with `nbar` nearly constant-level sidelobes adjacent to the mainlobe. These sidelobes are "nearly constant-level" because some decay occurs in the transition region. `nbar` must be a positive integer.

`w = taylorwin(n,nbar,sll)` returns an `n`-point Taylor window with a maximum sidelobe level of `sll` dB relative to the mainlobe peak.`sll` must be a negative value, such as -30, which produces sidelobes with peaks 30 dB down from the mainlobe peak.

Example

Generate a 64-point Taylor window with four nearly constant-level sidelobes and a peak sidelobe level of -35 dB relative to the mainlobe peak.

```
w = taylorwin(64,4,-35);
wvtool(w);
```



References

[1] Carrara, W.G., R.M. Majewski and R.S. Goodman, *Spotlight Synthetic Aperature Radar: Signal Processing Algorithms*, Artech House Publishers, Boston, 1995, Appendix D.2.

[2] Brookner, Eli, *Practical Phased Array Antenna Systems*, Lex Book, Lexington, MA, 1991.

Purpose

Convert transfer function filter parameters to lattice filter form

Syntax

```
[k,v] = tf2latc(b,a)
k = tf2latc(1,a)
[k,v] = tf2latc(1,a)
k = tf2latc(b)
k = tf2latc(b, 'phase')
```

Description

`[k,v] = tf2latc(b,a)` finds the lattice parameters `k` and the ladder parameters `v` for an IIR (ARMA) lattice-ladder filter, normalized by `a(1)`. Note that an error is generated if one or more of the lattice parameters are exactly equal to 1.

`k = tf2latc(1,a)` finds the lattice parameters `k` for an IIR all-pole (AR) lattice filter.

`[k,v] = tf2latc(1,a)` returns the scalar ladder coefficient at the correct position in vector `v`. All other elements of `v` are zero.

`k = tf2latc(b)` finds the lattice parameters `k` for an FIR (MA) lattice filter, normalized by `b(1)`.

`k = tf2latc(b, 'phase')` specifies the type of FIR (MA) lattice filter, where *`'phase'`* is

- `'max'`, for a maximum phase filter.
- `'min'`, for a minimum phase filter.

See Also

`latc2tf`, `latcfilt`, `tf2sos`, `tf2ss`, `tf2zp`, `tf2zpk`

Purpose

Convert digital filter transfer function data to second-order sections form

Syntax

```
[sos,g] = tf2sos(b,a)
[sos,g] = tf2sos(b,a,'order')
[sos,g] = tf2sos(b,a,'order','scale')
sos = tf2sos(...)
```

Description

`tf2sos` converts a transfer function representation of a given digital filter to an equivalent second-order section representation.

`[sos,g] = tf2sos(b,a)` finds a matrix `sos` in second-order section form with gain `g` that is equivalent to the digital filter represented by transfer function coefficient vectors `a` and `b`.

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_1 + b_2 z^{-1} + \cdots + b_{n+1} z^{-n}}{a_1 + a_2 z^{-1} + \cdots + a_{m+1} z^{-m}}$$

`sos` is an L-by-6 matrix

$$sos = \begin{bmatrix} b_{01} & b_{11} & b_{21} & 1 & a_{11} & a_{21} \\ b_{02} & b_{12} & b_{22} & 1 & a_{12} & a_{22} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{0L} & b_{1L} & b_{2L} & 1 & a_{1L} & a_{2L} \end{bmatrix}$$

whose rows contain the numerator and denominator coefficients b_{ik} and a_{ik} of the second-order sections of $H(z)$.

$$H(z) = g \prod_{k=1}^{L} H_k(z) = g \prod_{k=1}^{L} \frac{b_{0k} + b_{1k} z^{-1} + b_{2k} z^{-2}}{1 + a_{1k} z^{-1} + a_{2k} z^{-2}}$$

`[sos,g] = tf2sos(b,a,'order')` specifies the order of the rows in `sos`, where `'order'` is

- `'down'`, to order the sections so the first row of `sos` contains the poles closest to the unit circle
- `'up'`, to order the sections so the first row of `sos` contains the poles farthest from the unit circle (default)

`[sos,g] = tf2sos(b,a,'order','scale')` specifies the desired scaling of the gain and numerator coefficients of all second-order sections, where *`'scale'`* is:

- `'none'`, to apply no scaling (default)
- `'inf'`, to apply infinity-norm scaling
- `'two'`, to apply 2-norm scaling

Using infinity-norm scaling in conjunction with `up`-ordering minimizes the probability of overflow in the realization. Using 2-norm scaling in conjunction with `down`-ordering minimizes the peak round-off noise.

Note Infinity-norm and 2-norm scaling are appropriate only for direct-form II implementations.

`sos = tf2sos(...)` embeds the overall system gain, `g`, in the first section, $H_1(z)$, so that

$$H(z) = \prod_{k=1}^{L} H_k(z)$$

Note Embedding the gain in the first section when scaling a direct-form II structure is not recommended and may result in erratic scaling. To avoid embedding the gain, use `ss2sos` with two outputs.

Algorithm

`tf2sos` uses a four-step algorithm to determine the second-order section representation for an input transfer function system:

1. It finds the poles and zeros of the system given by `b` and `a`.
2. It uses the function `zp2sos`, which first groups the zeros and poles into complex conjugate pairs using the `cplxpair` function. `zp2sos` then forms the second-order sections by matching the pole and zero pairs according to the following rules:
   a. Match the poles closest to the unit circle with the zeros closest to those poles.
   b. Match the poles next closest to the unit circle with the zeros closest to those poles.
   c. Continue until all of the poles and zeros are matched.

   `tf2sos` groups real poles into sections with the real poles closest to them in absolute value. The same rule holds for real zeros.
3. It orders the sections according to the proximity of the pole pairs to the unit circle. `tf2sos` normally orders the sections with poles closest to the unit circle last in the cascade. You can tell `tf2sos` to order the sections in the reverse order by specifying the `'down'` flag.
4. `tf2sos` scales the sections by the norm specified in the *`'scale'`* argument. For arbitrary $H(\omega)$, the scaling is defined by

$$\|H\|_p = \left[\frac{1}{2\pi}\int_0^{2\pi} |H(\omega)|^p d\omega\right]^{\frac{1}{p}}$$

   where p can be either ∞ or 2. See the references for details on the scaling. This scaling is an attempt to minimize overflow or peak round-off noise in fixed point filter implementations.

References

[1] Jackson, L.B., *Digital Filters and Signal Processing, 3rd ed.*, Kluwer Academic Publishers, Boston, 1996, Chapter 11.

[2] Mitra, S.K., *Digital Signal Processing: A Computer-Based Approach*, McGraw-Hill, New York, 1998, Chapter 9.

[3] Vaidyanathan, P.P., "Robust Digital Filter Structures," *Handbook for Digital Signal Processing*, S.K. Mitra and J.F. Kaiser, ed., John Wiley & Sons, New York, 1993, Chapter 7.

See Also

`cplxpair`, `sos2tf`, `ss2sos`, `tf2ss`, `tf2zp`, `tf2zpk`, `zp2sos`

Purpose

Convert transfer function filter parameters to state-space form

Syntax

```
[A,B,C,D] = tf2ss(b,a)
```

Description

`tf2ss` converts the parameters of a transfer function representation of a given system to those of an equivalent state-space representation.

`[A,B,C,D] = tf2ss(b,a)` returns the `A`, `B`, `C`, and `D` matrices of a state space representation for the single-input transfer function

$$H(s) = \frac{B(s)}{A(s)} = \frac{b_1 s^{n-1} + \cdots + b_{n-1} s + b_n}{a_1 s^{m-1} + \cdots + a_{m-1} s + a_m} = C(sI - A)^{-1}B + D$$

in controller canonical form

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

The input vector `a` contains the denominator coefficients in descending powers of *s*. The rows of the matrix `b` contain the vectors of numerator coefficients (each row corresponds to an output). In the discrete-time case, you must supply `b` and `a` to correspond to the numerator and denominator polynomials with coefficients in descending powers of *z*.

For discrete-time systems you must make `b` have the same number of columns as the length of `a`. You can do this by padding each numerator represented in `b` (and possibly the denominator represented in the vector `a`) with trailing zeros. You can use the function `eqtflength` to accomplish this if `b` and `a` are vectors of unequal lengths.

The `tf2ss` function is part of the standard MATLAB language.

Examples

Consider the system:

$$H(s) = \frac{\begin{bmatrix} 2s+3 \\ s^2+2s+1 \end{bmatrix}}{s^2+0.4s+1}$$

To convert this system to state-space, type

```
b = [0 2 3; 1 2 1];
a = [1 0.4 1];
[A,B,C,D] = tf2ss(b,a)
A =
   -0.4000   -1.0000
    1.0000         0
B =
     1
     0
C =
    2.0000    3.0000
    1.6000         0
D =
     0
     1
```

Note There is disagreement in the literature on naming conventions for the canonical forms. It is easy, however, to generate similarity transformations that convert these results to other forms.

See Also

`sos2ss`, `ss2tf`, `tf2sos`, `tf2zp`, `tf2zpk`, `zp2ss`

Purpose

Convert transfer function filter parameters to zero-pole-gain form

Syntax

```
[z,p,k] = tf2zp(b,a)
```

Description

`tf2zp` finds the zeros, poles, and gains of a continuous-time transfer function.

Note You should use `tf2zp` when working with positive powers ($s^2 + s + 1$), such as in continuous-time transfer functions. A similar function, `tf2zpk`, is more useful when working with transfer functions expressed in inverse powers ($1 + z^{-1} + z^{-2}$), which is how transfer functions are usually expressed in DSP.

`[z,p,k] = tf2zp(b,a)` finds the matrix of zeros `z`, the vector of poles `p`, and the associated vector of gains `k` from the transfer function parameters `b` and `a`:

- The numerator polynomials are represented as columns of the matrix `b`.
- The denominator polynomial is represented in the vector `a`.

Given a SIMO continuous-time system in polynomial transfer function form

$$H(s) = \frac{B(s)}{A(s)} = \frac{b_1 s^{n-1} + \cdots + b_{n-1}s + b_n}{a_1 s^{m-1} + \cdots + a_{m-1}s + a_m}$$

you can use the output of `tf2zp` to produce the single-input, multioutput (SIMO) factored transfer function form

$$H(s) = \frac{Z(s)}{P(s)} = k\frac{(s-z_1)(s-z_2)\cdots(s-z_m)}{(s-p_1)(s-p_2)\cdots(s-p_n)}$$

The following describes the input and output arguments for `tf2zp`:

- The vector `a` specifies the coefficients of the denominator polynomial $A(s)$ (or $A(z)$) in descending powers of s (z^{-1}).
- The ith row of the matrix `b` represents the coefficients of the ith numerator polynomial (the ith row of $B(s)$ or $B(z)$). Specify as many rows of `b` as there are outputs.
- For continuous-time systems, choose the number nb of columns of `b` to be less than or equal to the length na of the vector `a`.
- For discrete-time systems, choose the number nb of columns of `b` to be equal to the length na of the vector `a`. You can use the function `eqtflength` to provide equal length vectors in the case that `b` and `a` are vectors of unequal lengths. Otherwise, pad the numerators in the matrix `b` (and, possibly, the denominator vector `a`) with zeros.
- The zero locations are returned in the columns of the matrix `z`, with as many columns as there are rows in `b`.
- The pole locations are returned in the column vector `p` and the gains for each numerator transfer function in the vector `k`.

The `tf2zp` function is part of the standard MATLAB language.

Examples

Find the zeros, poles, and gains of this continuous-time system:

$$H(s) = \frac{2s^2 + 3s}{s^2 + 0.4s + 1}$$

```
b = [2 3];
a = [1 0.4 1];
[b,a] = eqtflength(b,a);      % Make lengths equal
[z,p,k] = tf2zp(b,a)          % Obtain zero-pole-gain form
z =
         0
   -1.5000
p =
   -0.2000 + 0.9798i
   -0.2000 - 0.9798i
```

```
k =
     2
```

See Also

sos2zp, ss2zp, tf2sos, tf2ss, tf2zpk, zp2tf

Purpose

Convert transfer function filter parameters to zero-pole-gain form

Syntax

```
[z,p,k] = tf2zpk(b,a)
```

Description

`tf2zpk` finds the zeros, poles, and gains of a discrete-time transfer function.

Note You should use `tf2zpk` when working with transfer functions expressed in inverse powers ($1 + z^{-1} + z^{-2}$), which is how transfer functions are usually expressed in DSP. A similar function, `tf2zp`, is more useful for working with positive powers ($s^2 + s + 1$), such as in continuous-time transfer functions.

`[z,p,k] = tf2zpk(b,a)` finds the matrix of zeros `z`, the vector of poles `p`, and the associated vector of gains `k` from the transfer function parameters `b` and `a`:

- The numerator polynomials are represented as columns of the matrix `b`.
- The denominator polynomial is represented in the vector `a`.

Given a single-input, multiple output (SIMO) discrete-time system in polynomial transfer function form

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_1 + b_2 z^{-1} \cdots + b_{n-1} z^{-n} + b_n z^{-n-1}}{a_1 + a_2 z^{-1} \cdots + a_{m-1} z^{-m} + a_m z^{-m-1}}$$

you can use the output of `tf2zpk` to produce the single-input, multioutput (SIMO) factored transfer function form

$$H(z) = \frac{Z(z)}{P(z)} = k \frac{(z - z_1)(z - z_2) \cdots (z - z_m)}{(z - p_1)(z - p_2) \cdots (z - p_n)}$$

The following describes the input and output arguments for `tf2zpk`:

- The vector `a` specifies the coefficients of the denominator polynomial $A(z)$ in descending powers of z.
- The ith row of the matrix `b` represents the coefficients of the `i`th numerator polynomial (the ith row of $B(s)$ or $B(z)$). Specify as many rows of `b` as there are outputs.
- The zero locations are returned in the columns of the matrix `z`, with as many columns as there are rows in `b`.
- The pole locations are returned in the column vector `p` and the gains for each numerator transfer function in the vector `k`.

Examples

Find the poles, zeros, and gain of a Butterworth filter:

```
[b,a] = butter(3,.4);
[z,p,k] = tf2zpk(b,a)
z =
  -1.0000
  -1.0000 + 0.0000i
  -1.0000 - 0.0000i

p =
   0.2094 + 0.5582i
   0.2094 - 0.5582i
   0.1584

k =
    0.0985
```

See Also

`sos2zp`, `ss2zp`, `tf2sos`, `tf2ss`, `tf2zp`, `zp2tf`

Purpose

Transfer function estimate

Syntax

```
Txy = tfestimate(x,y)
Txy = tfestimate(x,y,window)
Txy = tfestimate(x,y,window,noverlap)
[Txy,W] = tfestimate(x,y,window,noverlap,nfft)
[Txy,F] = tfestimate(x,y,window,noverlap,nfft,fs)
[...] = tfestimate(x,y,...,'whole')
tfestimate(...)
```

Description

`Txy = tfestimate(x,y)` finds a transfer function estimate `Txy` given input signal vector `x` and output signal vector `y`. Vectors `x` and `y` must be the same length. The relationship between the input `x` and output `y` is modeled by the linear, time-invariant transfer function `Txy`. The *transfer function* is the quotient of the cross power spectral density (*Pyx*) of `x` and `y` and the power spectral density (*Pxx*) of `x`.

$$T_{xy}(f) = \frac{P_{yx}(f)}{P_{xx}(f)}$$

If `x` is real, `tfestimate` estimates the transfer function at positive frequencies only; in this case, the output `Txy` is a column vector of length `nfft/2+1` for `nfft` even and `(nfft+1)/2` for `nfft` odd. If `x` or `y` is complex, `tfestimate` estimates the transfer function for both positive and negative frequencies and `Txy` has length `nfft`.

`tfestimate` uses the following default values:

Default Values

Parameter	Description	Default Value
`nfft`	FFT length which determines the frequencies at which the PSD is estimated For real `x` and `y`, the length of `Txy` is (`nfft`/2+1) if `nfft` is even or (`nfft`+1)/2 if `nfft` is odd. For complex `x` or `y`, the length of `Txy` is `nfft`.	Maximum of 256 or the next power of 2 greater than the length of each section of `x` or `y`
`fs`	Sampling frequency	1
`window`	Windowing function and number of samples to use to section x and y	Periodic Hamming window with length equal to the signal segment length that results from dividing the signal `x` into eight sections and then applying the default or specified overlap.
`noverlap`	Number of samples by which the sections overlap	Value to obtain 50% overlap

Note You can use the empty matrix `[]` to specify the default value for any input argument except `x` or `y`. For example, `Txy = tfestimate(x,y,[],[],128)` uses a Hamming window with default length, as described above, default `noverlap` to obtain 50% overlap, and the specified 128 `nfft`.

`Txy = tfestimate(x,y,window)` specifies a windowing function, divides `x` and `y` into overlapping sections of the specified window length, and windows each section using the specified window function. If you supply a scalar for `window`, `Txy` uses a Hamming window of that length. The length of the window must be less than or equal to `nfft`. If the length of the window exceeds `nfft`, `tfestimate` zero pads the sections. To replicate the output of the obsoleted `tfe` function, specify `'hanning(nfft)'` as the window.

`Txy = tfestimate(x,y,window,noverlap)` overlaps the sections of `x` by `noverlap` samples. `noverlap` must be an integer smaller than the length of `window`.

`[Txy,W] = tfestimate(x,y,window,noverlap,nfft)` uses the specified FFT length `nfft` in estimating the PSD and CPSD estimates for the transfer function. It also returns `W`, which is the vector of normalized frequencies (inrad/sample) at which the `tfestimate` is estimated. For real signals, the range of `W` is [0, π] when `nfft` is even and [0, π) when `nfft` is odd. For complex signals, the range of `W` is [0, 2π).

`[Txy,F] = tfestimate(x,y,window,noverlap,nfft,fs)` returns `Txy` as a function of frequency and a vector `F` of frequencies at which `tfestimate` estimates the transfer function. `fs` is the sampling frequency in Hz. `F` is the same size as `Txy`, so `plot(f,Txy)` plots the transfer function estimate versus properly scaled frequency. For real signals, the range of `F` is [0, `fs`/2] when `nfft` is even and [0, `fs`/2) when `nfft` is odd. For complex signals, the range of `F` is [0, `fs`).

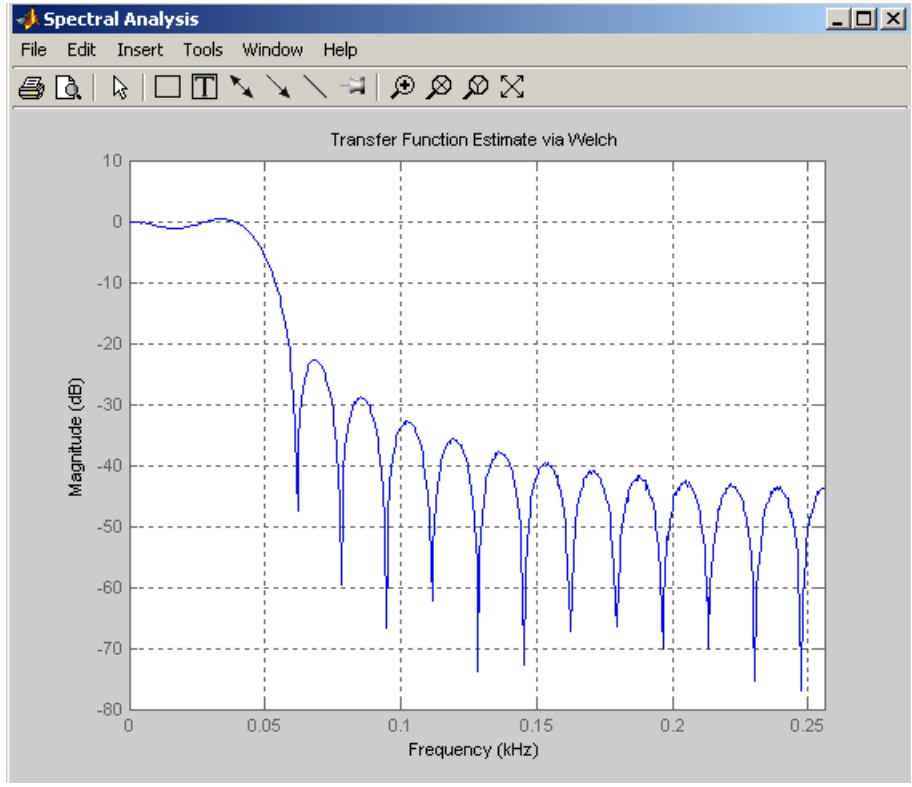
`[...] = tfestimate(x,y,...,'whole')` returns a transfer function estimate with frequencies that range over the whole Nyquist interval. Specifying `'half'` uses half the Nyquist interval.

`tfestimate(...)` with no output arguments plots the transfer function estimate in the current figure window.

Examples

Compute and plot the transfer function estimate between two colored noise sequences `x` and `y`:

```
h = fir1(30,0.2,rectwin(31));
x = randn(16384,1);
y = filter(h,1,x);
tfestimate(x,y,1024,[],[],512)
```



Algorithm

`tfestimate` uses Welch's averaged periodogram method. See `pwelch` for details.

See Also

`cpsd`, `mscohere`, `periodogram`, `pwelch`, `spectrum`

Purpose

Triangular window

Syntax

```
triang(L)
```

Description

`triang(L)` returns an `L`-point triangular window in the column vector `w`. The coefficients of a triangular window are:

For `L` odd:

$$w(n) = \begin{cases} \dfrac{2n}{L+1}, & 1 \le n \le \dfrac{L+1}{2} \\ \dfrac{2(L-n+1)}{L+1}, & \dfrac{L+1}{2} < n \le L \end{cases}$$

For `L` even:

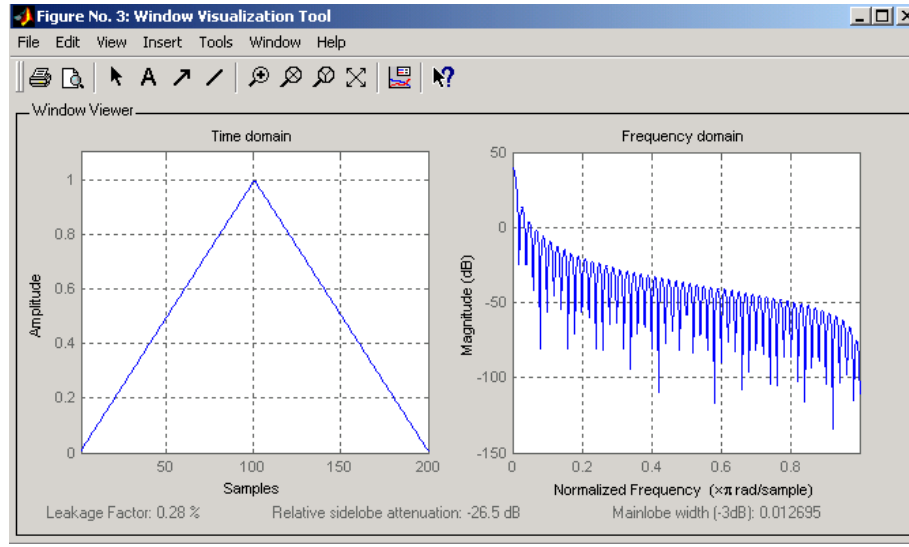
$$w(n) = \begin{cases} \dfrac{2n}{L}, & 1 \le n \le \dfrac{L+1}{2} \\ \dfrac{2(L-n+1)}{L} & \dfrac{L}{2}+1 \le n \le L \end{cases}$$

The triangular window is very similar to a Bartlett window. The Bartlett window always ends with zeros at samples 1 and `L`, while the triangular window is nonzero at those points. For `L` odd, the center `L-2` points of `triang(L-2)` are equivalent to `bartlett(L)`.

Examples

Create a 200-point triangular window and plot the result using WVTool.

```
L=200;
wvtool(triang(L))
```



References

[1] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1989, pp. 447-448.

See Also

`barthannwin`, `bartlett`, `blackmanharris`, `bohmanwin`, `nuttallwin`, `parzenwin`, `rectwin`, `window`, `wintool`, `wvtool`

Purpose

Sampled aperiodic triangle

Syntax

```
y = tripuls(T)
y = tripuls(T,w)
y = tripuls(T,w,s)
```

Description

`y = tripuls(T)` returns a continuous, aperiodic, symmetric, unity-height triangular pulse at the times indicated in array `T`, centered about `T=0` and with a default width of 1.

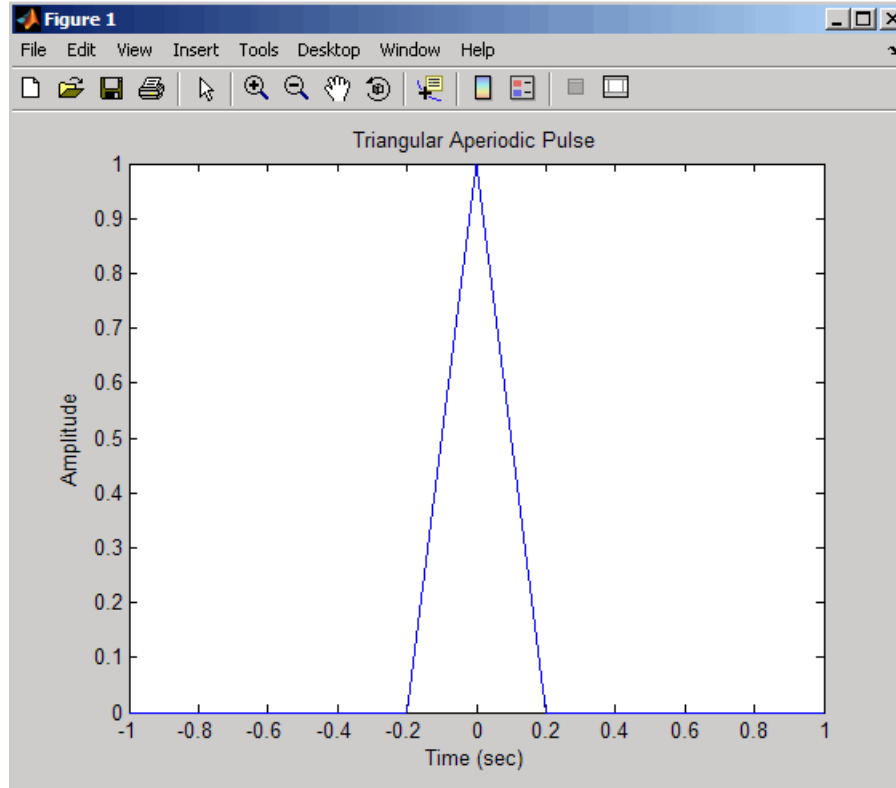
`y = tripuls(T,w)` generates a triangular pulse of width `w`.

`y = tripuls(T,w,s)` generates a triangular pulse with skew `s`, where `-1 < s < 1`. When `s` is 0, a symmetric triangular pulse is generated.

Examples

Create a triangular pulse with width 0.4.

```
fs = 10000;
t = -1:1/fs:1;
w = .4;
x = tripuls(t,w);
figure,plot(t,x)
xlabel('Time (sec)');ylabel('Amplitude');
title('Triangular Aperiodic Pulse')
```



See Also

`chirp`, `cos`, `diric`, `gauspuls`, `pulstran`, `rectpuls`, `sawtooth`, `sin`, `square`, `tripuls`

Purpose

Tukey (tapered cosine) window

Syntax

```
w = tukeywin(L,r)
```

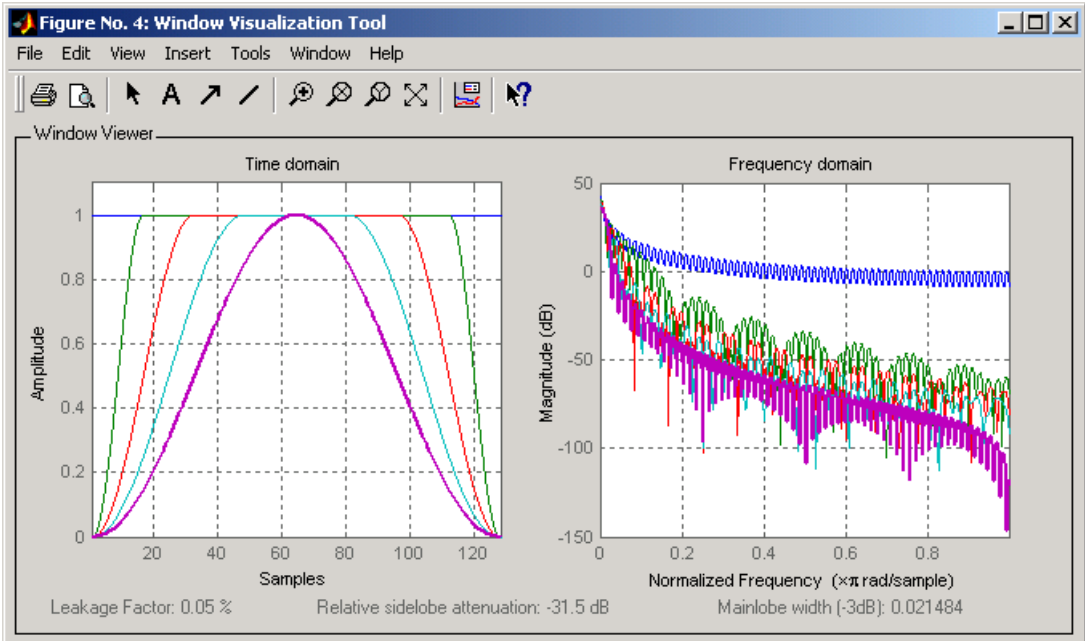
Description

`w = tukeywin(L,r)` returns an `L`-point, Tukey window in column vector `w`. Tukey windows are cosine-tapered windows. `r` is the ratio of taper to constant sections and is between 0 and 1. $r \leq 0$ is a `rectwin` window and $r \geq 1$ is a `hann` window. The default value for `r` is 0.5.

Examples

Compute 128-point Tukey windows with five different tapers and display the results using WVTool:

```
L=128;
t0=tukeywin(L,0);        % Equivalent to rectangular window
t25=tukeywin(L,0.25);
t5=tukeywin(L);          % r=0.5
t75=tukeywin(L,0.75);
t1=tukeywin(L,1);        % Equivalent to Hann window
wvtool(t0,t25,t5,t75,t1)
```



Algorithm

The equation for computing the coefficients of a Tukey window is

$$w(n) = \begin{cases} 1.0, & 0 \le |n| \le \alpha\frac{N}{2} \\ \frac{1}{2}\left(1+\cos\left(\pi\frac{n-\alpha\frac{N}{2}}{2(1-\alpha)\frac{N}{2}}\right)\right), & \alpha\frac{N}{2} \le |n| \le \frac{N}{2} \end{cases}$$

The window length is $L = N + 1$.

References

[1] Harris, F. J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform." *Proceedings of the IEEE*. Vol. 66 (January 1978). pp. 66-67.

See Also

chebwin, gausswin, kaiser, window, wintool, wvtool

udecode

Purpose

Decode 2^n-level quantized integer inputs to floating-point outputs

Syntax

```
y = udecode(u,n)
y = udecode(u,n,v)
y = udecode(u,n,v,'SaturateMode')
```

Description

`y = udecode(u,n)` inverts the operation of `uencode` and reconstructs quantized floating-point values from an encoded multidimensional array of integers `u`. The input argument `n` must be an integer between 2 and 32. The integer `n` specifies that there are 2^n quantization levels for the inputs, so that entries in `u` must be either:

- Signed integers in the range $[-2^n/2, (2^n/2) - 1]$
- Unsigned integers in the range $[0, 2^n-1]$

Inputs can be real or complex values of any integer data type (`uint8`, `uint16`, `uint32`, `int8`, `int16`, `int32`). Overflows (entries in `u` outside of the ranges specified above) are saturated to the endpoints of the range interval. The output `y` has the same dimensions as `u`. Its entries have values in the range `[-1,1]`.

`y = udecode(u,n,v)` decodes `u` such that the output `y` has values in the range `[-v,v]`, where the default value for `v` is 1.

`y = udecode(u,n,v,'SaturateMode')` decodes `u` and treats input overflows (entries in `u` outside of `[-v,v]`) according to the string *`'saturatemode'`*, which can be one of the following:

- `'saturate'`: Saturate overflows. This is the default method for treating overflows.
  - Entries in signed inputs `u` whose values are outside of the range $[-2^n/2, (2^n/2) - 1]$ are assigned the value determined by the closest endpoint of this interval.
  - Entries in unsigned inputs `u` whose values are outside of the range $[0, 2^n-1]$ are assigned the value determined by the closest endpoint of this interval.

- `'wrap'`: Wrap all overflows according to the following:
  - Entries in signed inputs `u` whose values are outside of the range $[-2^n/2, (2^n/2) - 1]$ are wrapped back into that range using modulo 2^n arithmetic (calculated using `u = mod(u+2^n/2,2^n)-(2^n/2)`).
  - Entries in unsigned inputs `u` whose values are outside of the range $[0, 2^n-1]$ are wrapped back into the required range before decoding using modulo 2^n arithmetic (calculated using `u = mod(u,2^n)`).

Examples

```
% Create signed 8-bit integer string
u = int8([-1 1 2 -5]);
% Decode with 3 bits
ysat = udecode(u,3)
ysat =
   -0.2500    0.2500    0.5000   -1.0000
```

Notice the last entry in `u` saturates to 1, the default peak input magnitude. Change the peak input magnitude:

```
ysatv = udecode(u,3,6)   % Set peak input magnitude to 6
ysatv =
   -1.5000    1.5000    3.0000   -6.0000
```

The last input entry still saturates. Try wrapping the overflows:

```
ywrap = udecode(u,3,6,'wrap')
ywrap =
   -1.5000    1.5000    3.0000    4.5000
```

Try adding more quantization levels:

```
yprec = udecode(u,5)
yprec =
   -0.0625    0.0625    0.1250   -0.3125
```

Algorithm

The algorithm adheres to the definition for uniform decoding specified in ITU-T Recommendation G.701. Integer input values are uniquely

mapped (decoded) from one of 2^n uniformly spaced integer values to quantized floating-point values in the range `[-v,v]`. The smallest integer input value allowed is mapped to `-v` and the largest integer input value allowed is mapped to `v`. Values outside of the allowable input range are either saturated or wrapped, according to specification.

The real and imaginary components of complex inputs are decoded independently.

References

[1] *General Aspects of Digital Transmission Systems: Vocabulary of Digital Transmission and Multiplexing, and Pulse Code Modulation (PCM) Terms*, International Telecommunication Union, ITU-T Recommendation G.701, March, 1993.

See Also

`uencode`

Purpose

Quantize and encode floating-point inputs to integer outputs

Syntax

```
y = uencode(u,n)
y = uencode(u,n,v)
y = uencode(u,n,v,'SignFlag')
```

Description

`y = uencode(u,n)` quantizes the entries in a multidimensional array of floating-point numbers `u` and encodes them as integers using 2^n-level quantization. `n` must be an integer between 2 and 32 (inclusive). Inputs can be real or complex, double- or single-precision. The output `y` and the input `u` are arrays of the same size. The elements of the output `y` are unsigned integers with magnitudes in the range [0, 2^n-1]. Elements of the input `u` outside of the range `[-1,1]` are treated as overflows and are saturated.

- For entries in the input `u` that are less than -1, the value of the output of `uencode` is 0.
- For entries in the input `u` that are greater than 1, the value of the output of `uencode` is 2^n-1.

`y = uencode(u,n,v)` allows the input `u` to have entries with floating-point values in the range `[-v,v]` before saturating them (the default value for `v` is 1). Elements of the input `u` outside of the range `[-v,v]` are treated as overflows and are saturated:

- For input entries less than `-v`, the value of the output of `uencode` is 0.
- For input entries greater than `v`, the value of the output of `uencode` is 2^n-1.

`y = uencode(u,n,v,'SignFlag')` maps entries in a multidimensional array of floating-point numbers `u` whose entries have values in the range [`-v,v`] to an integer output `y`. Input entries outside this range are saturated. The integer type of the output depends on the string `'SignFlag'` and the number of quantization levels 2^n. The string `'SignFlag'` can be one of the following:

- `'signed'`: Outputs are signed integers with magnitudes in the range [$-2^n/2$, ($2^n/2$) - 1].
- `'unsigned'` (default): Outputs are unsigned integers with magnitudes in the range [0, 2^n-1].

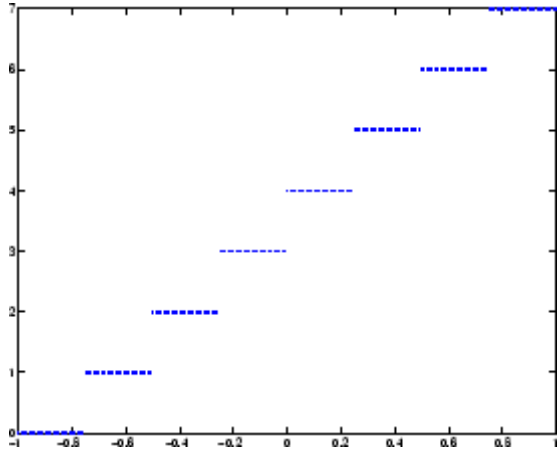
The output data types are optimized for the number of bits as shown in the table below.

n	Unsigned Integer	Signed Integer
2 to 8	`uint8`	`int8`
9 to 16	`uint16`	`int16`
17 to 32	`uint32`	`int32`

Examples

Map floating-point scalars in [-1, 1] to `uint8` (unsigned) integers, and produce a staircase plot. Note that the horizontal axis plots from -1 to 1 and the vertical axis plots from 0 to 7 (`2^3-1`):

```
u = [-1:0.01:1];
y = uencode(u,3);
plot(u,y,'.')
```



Now look at saturation effects when you under specify the peak value for the input:

```
u = [-2:0.5:2];
y = uencode(u,5,1)
y =
     0     0     0     8    16    24    31    31    31
```

Now look at the output for

```
u = [-2:0.5:2];
y = uencode(u,5,2,'signed')
y =
    -16   -12    -8    -4     0     4     8    12    15
```

Algorithm

`uencode` maps the floating-point input value to an integer value determined by the requirement for 2^n levels of quantization. This encoding adheres to the definition for uniform encoding specified in ITU-T Recommendation G.701. The input range `[-v,v]` is divided into 2^n evenly spaced intervals. Input entries in the range `[-v,v]` are first quantized according to this subdivision of the input range, and then mapped to one of 2^n integers. The range of the output depends on whether or not you specify that you want signed integers.

References

[1] *General Aspects of Digital Transmission Systems: Vocabulary of Digital Transmission and Multiplexing, and Pulse Code Modulation (PCM) Term*s, International Telecommunication Union, ITU-T Recommendation G.701, March, 1993.

See Also

`udecode`

upfirdn

Purpose

Upsample, apply FIR filter, and downsample

Syntax

```
yout = upfirdn(xin,h)
yout = upfirdn(xin,h,p)
yout = upfirdn(xin,h,p,q)
```

Description

`upfirdn` performs a cascade of three operations:

1. Upsampling the input data in the matrix `xin` by a factor of the integer `p` (inserting zeros)
2. FIR filtering the upsampled signal data with the impulse response sequence given in the vector or matrix `h`
3. Downsampling the result by a factor of the integer `q` (throwing away samples)

`upfirdn` has been implemented as a MEX-file for maximum speed, so only the outputs actually needed are computed. The FIR filter is usually a lowpass filter, which you must design using another function such as `firpm` or `fir1`.

Note The function `resample` performs an FIR design using `firls`, followed by rate changing implemented with `upfirdn`.

`yout = upfirdn(xin,h)` filters the input signal `xin` with the FIR filter having impulse response `h`. If `xin` is a row or column vector, then it represents a single signal. If `xin` is a matrix, then each column is filtered independently. If `h` is a row or column vector, then it represents one FIR filter. If `h` is a matrix, then each column is a separate FIR impulse response sequence. If `yout` is a row or column vector, then it represents one signal. If `yout` is a matrix, then each column is a separate output. No upsampling or downsampling is implemented with this syntax.

`yout = upfirdn(xin,h,p)` specifies the integer upsampling factor `p`, where `p` has a default value of 1.

`yout = upfirdn(xin,h,p,q)` specifies the integer downsampling factor `q`, where `q` has a default value of 1.

Note Since `upfirdn` performs convolution and rate changing, the `yout` signals have a different length than `xin`. The number of rows of `yout` is approximately `p/q` times the number of rows of `xin`.

Remarks

Usually the inputs `xin` and the filter `h` are vectors, in which case only one output signal is produced. However, when these arguments are arrays, each column is treated as a separate signal or filter. Valid combinations are:

1. `xin` is a vector and `h` is a vector.

   There is one filter and one signal, so the function convolves `xin` with `h`. The output signal `yout` is a row vector if `xin` is a row; otherwise, `yout` is a column vector.

2. `xin` is a matrix and `h` is a vector.

   There is one filter and many signals, so the function convolves `h` with each column of `xin`. The resulting `yout` will be an matrix with the same number of columns as `xin`.

3. `xin` is a vector and `h` is a matrix.

   There are many filters and one signal, so the function convolves each column of `h` with `xin`. The resulting `yout` will be an matrix with the same number of columns as `h`.

4. `xin` is a matrix and `h` is a matrix, both with the same number of columns.

There are many filters and many signals, so the function convolves corresponding columns of `xin` and `h`. The resulting `yout` is an matrix with the same number of columns as `xin` and `h`.

Examples

Change the sampling rate by a factor of 147/160. This factor is used to convert from 48kHz (DAT rate) to 44.1kHz (CD sampling rate).

```
L = 147; M = 160;     % Interpolation/decimation factors.
N = 24*L;
h = fir1(N-1,1/M,kaiser(N,7.8562));
h = L*h; % Passband gain = L
Fs = 48e3;            % Original sampling frequency-48kHz
n = 0:10239;          % 10240 samples, 0.213 seconds long
x  = sin(2*pi*1e3/Fs*n); % Original signal, sinusoid @ 1kHz
y = upfirdn(x,h,L,M);    % 9408 samples, still .213 seconds

% Overlay original (48kHz) with resampled
% signal (44.1kHz) in red.

stem(n(1:49)/Fs,x(1:49)); hold on
stem(n(1:45)/(Fs*L/M),y(13:57),'r','filled');
xlabel('Time (sec)');ylabel('Signal value');
```

Algorithm

`upfirdn` uses a polyphase interpolation structure. The number of multiply-add operations in the polyphase structure is approximately $(L_h L_x\text{-}pL_x)/q$ where L_h and L_x are the lengths of $h[n]$ and $x[n]$, respectively.

A more accurate flops count is computed in the program, but the actual count is still approximate. For long signals $x[n]$, the formula is often exact.

Diagnostics

If `p` and `q` are large and do not have many common factors, you may see this message:

```
Filter length is too large - reduce problem complexity.
```

Instead, you should use an interpolation function, such as `interp1`, to perform the resampling and then filter the input.

References

[1] Crochiere, R.E., and L.R. Rabiner, *Multi-Rate Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1983, pp. 88-91.

[2] Crochiere, R.E., "A General Program to Perform Sampling Rate Conversion of Data by Rational Ratios," *Programs for Digital Signal Processing*, IEEE Press, New York, 1979, pp. 8.2-1 to 8.2-7.

See Also

`conv`, `decimate`, `downsample`, `filter`, `interp`, `intfilt`, `resample`, `upsample`

upsample

Purpose

Increase sampling rate by integer factor

Syntax

```
y = upsample(x,n)
y = upsample(x,n,phase)
```

Description

`y = upsample(x,n)` increases the sampling rate of `x` by inserting `n-1` zeros between samples. `x` can be a vector or a matrix. If `x` is a matrix, each column is considered a separate sequence. The upsampled `y` has `x*n` samples.

`y = upsample(x,n,phase)` specifies the number of samples by which to offset the upsampled sequence. `phase` must be an integer from 0 to `n-1`.

Examples

Increase the sampling rate of a sequence by 3:

```
x = [1 2 3 4];
y = upsample(x,3);
x,y
x =
    1  2  3  4
y =
    1  0  0  2  0  0  3  0  0  4  0  0
```

Increase the sampling rate of the sequence by 3 and add a phase offset of 2:

```
x = [1 2 3 4];
y = upsample(x,3,2);
x,y
x =
    1  2  3  4
y =
    0  0  1  0  0  2  0  0  3  0  0  4
```

Increase the sampling rate of a matrix by 3:

```
x = [1 2; 3 4; 5 6;];
y = upsample(x,3);
```

```
x,y
x =
     1     2
     3     4
     5     6
y =
     1     2
     0     0
     0     0
     3     4
     0     0
     0     0
     5     6
     0     0
     0     0
```

See Also

decimate, downsample, interp, interp1, resample, spline, upfirdn

validstructures

Purpose

Structures for specification object with design method

Syntax

```
validstructures(d)
validstructures(d,'designmethod')
c = validstructures(d,'designmethod')
```

Description

`validstructures(d)` returns the list of structures for all design methods that are available for `d`.

`validstructures(d,'designmethod')` returns a list of the filter structures available for the specification object `d` and the design method in `designmethod`. Knowing which structures apply to your combination of design method and specification makes deciding on a filter structure to implement easier.

To determine the available structures, `validstructures` considers the filter response, such as lowpass or bandstop. It also considers the specifications you use to define the response, such as filter order or stopband attenuation, because changing the filter specifications often changes the available structures.

`c = validstructures(d,'designmethod')` returns the output cell array `c` that contains the filter structures as character strings.

Examples

Design a default lowpass fillter specification object and query all valid structures by design method.

```
d=fdesign.lowpass;
validstructures(d)

ans =

        butter: {'df1sos'  'df2sos'  'df1tsos'  'df2tsos'}
        cheby1: {'df1sos'  'df2sos'  'df1tsos'  'df2tsos'}
        cheby2: {'df1sos'  'df2sos'  'df1tsos'  'df2tsos'}
         ellip: {'df1sos'  'df2sos'  'df1tsos'  'df2tsos'}
    equiripple: {'dffir'  'dffirt'  'dfsymfir'  'fftfir'}
     kaiserwin: {'dffir'  'dffirt'  'dfsymfir'  'fftfir'}
```

Create a highpass filter of order 50 with a 3–dB frequency of 0.2.

```
d=fdesign.highpass('N,F3dB',50,0.2);
validmethod=designmethods(d) %check valid design methods
validmethod

validmethod =

    'butter'
%Butterworth is the only valid design
validstructures(d,'butter') %check valid structures
ans =

    'df1sos'    'df2sos'    'df1tsos'    'df2tsos'
```

See Also

`design`, `designopts`.

Purpose

Voltage controlled oscillator

Syntax

```
y = vco(x,fc,fs)
y = vco(x,[Fmin Fmax],fs)
```

Description

`y = vco(x,fc,fs)` creates a signal that oscillates at a frequency determined by the real input vector or array `x` with sampling frequency `fs`. `fc` is the carrier or reference frequency; when `x` is 0, `y` is an `fc` Hz cosine with amplitude 1 sampled at `fs` Hz. `x` ranges from `-1` to `1`, where `x = -1` corresponds to 0 frequency output, `x = 0` corresponds to `fc`, and `x = 1` corresponds to `2*fc`. Output `y` is the same size as `x`.

`y = vco(x,[Fmin Fmax],fs)` scales the frequency modulation range so that ±1 values of `x` yield oscillations of `Fmin` Hz and `Fmax` Hz respectively. For best results, `Fmin` and `Fmax` should be in the range 0 to `fs/2`.

By default, `fs` is 1 and `fc` is `fs/4`.

If `x` is a matrix, `vco` produces a matrix whose columns oscillate according to the columns of `x`.

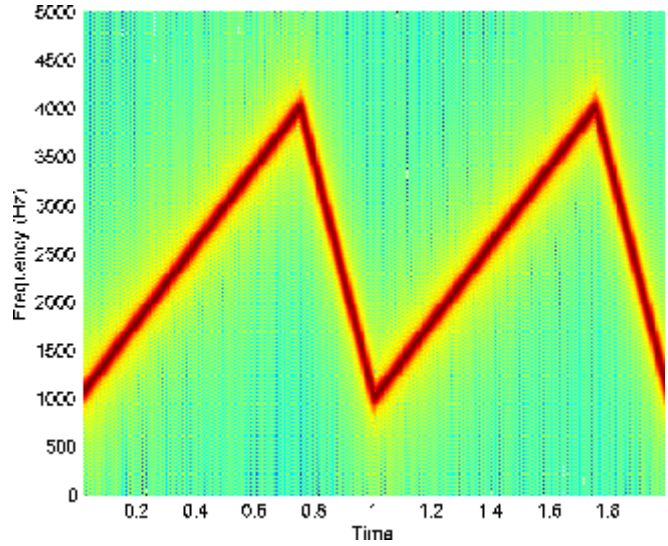
Examples

Generate two seconds of a signal sampled at 10,000 samples/second whose instantaneous frequency is a triangle function of time:

```
fs = 10000;
t = 0:1/fs:2;
x = vco(sawtooth(2*pi*t,0.75),[0.1 0.4]*fs,fs);
```

Plot the spectrogram of the generated signal:

```
spectrogram(x,kaiser(256,5),220,512,fs,'yaxis')
```



Algorithm

`vco` performs FM modulation using the `modulate` function.

Diagnostics

If any values of `x` lie outside [-1, 1], `vco` gives the following error message.

```
X outside of range [-1,1].
```

See Also

`demod`, `modulate`

window

Purpose

Window function gateway

Syntax

```
window
w = window(fhandle,n)
w = window(fhandle,n,winopt)
```

Description

`window` opens the Window Design and Analysis Tool (`wintool`).

`w = window(fhandle,n)` returns the `n`-point window, specified by its function handle, `fhandle`, in column vector `w`. Function handles are window function names preceded by an @.

```
@barthannwin
@bartlett
@blackman
@blackmanharris
@bohmanwin
@chebwin
@flattopwin
@gausswin
@hamming
@hann
@kaiser
@nuttallwin
@parzenwin
@rectwin
@taylorwin
@triang
@tukeywin
```

Note For `chebwin`, `kaiser`, and `tukeywin`, you must use include a window parameter using the syntax below.

For more information on each window function and its option(s), refer to its reference page.

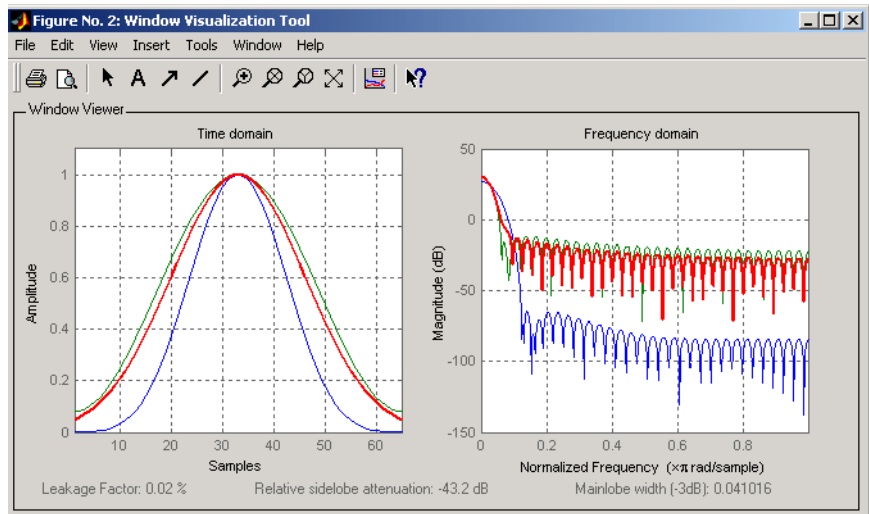
`w = window(fhandle,n,winopt)` returns the window specified by its function handle, `fhandle`, and its *`winopt`* value or sampling flag string. For `chebwin`, `kaiser`, and `tukeywin`, you must enter a *`winopt`* value. For the other windows listed below, *`winopt`* values are optional.

Window	winopt Description	winopt Value
`blackman`	sampling flag string	`'periodic'` or `'symmetric'`
`chebwin`	sidelobe attenuation relative to mainlobe	numeric
`flattopwin`	sampling flag string	`'periodic'` or `'symmetric'`
`gausswin`	alpha value (reciprocal of standard deviation)	numeric
`hamming`	sampling flag string	`'periodic'` or `'symmetric'`
`hann`	sampling flag string	`'periodic'` or `'symmetric'`
`kaiser`	beta value	numeric
`taylorwin`	1. number of sidelobes 2. maximum sidelobe level in dB relative to mainlobe peak	1. integer greater than or equal to 1 2. negative value
`tukeywin`	ratio of taper to constant sections	numeric

Examples

Create Blackman Harris, Hamming, and Gaussian windows and plot them in the same WVTool.

```
N = 65;
w = window(@blackmanharris,N);
w1 = window(@hamming,N);
w2 = window(@gausswin,N,2.5);
wvtool(w,w1,w2)
```



See Also

`barthannwin`, `bartlett`, `blackman`, `blackmanharris`, `bohmanwin`, `chebwin`, `flattopwin`, `gausswin`, `hamming`, `hann`, `kaiser`, `nuttallwin`, `parzenwin`, `rectwin`, `triang`, `taylorwin`, `tukeywin`

Purpose

FIR filter using windowed impulse response

Syntax

```
h = window(d,'window',fcnhndl)
h = window(d,win)
```

description

Note This is a description of the overloaded method used in conjunction with `fdesign` to design a filter from a filter specification object. To access the window function gateway see `window`.

`h = window(d,'window',fcnhndl)` designs an FIR filter using the specifications in filter specification object `d`. Depending on the specification type of `d`, the returned filter is either a single-rate digital filter — a `dfilt`, or a multirate digital filter — an `mfilt`.

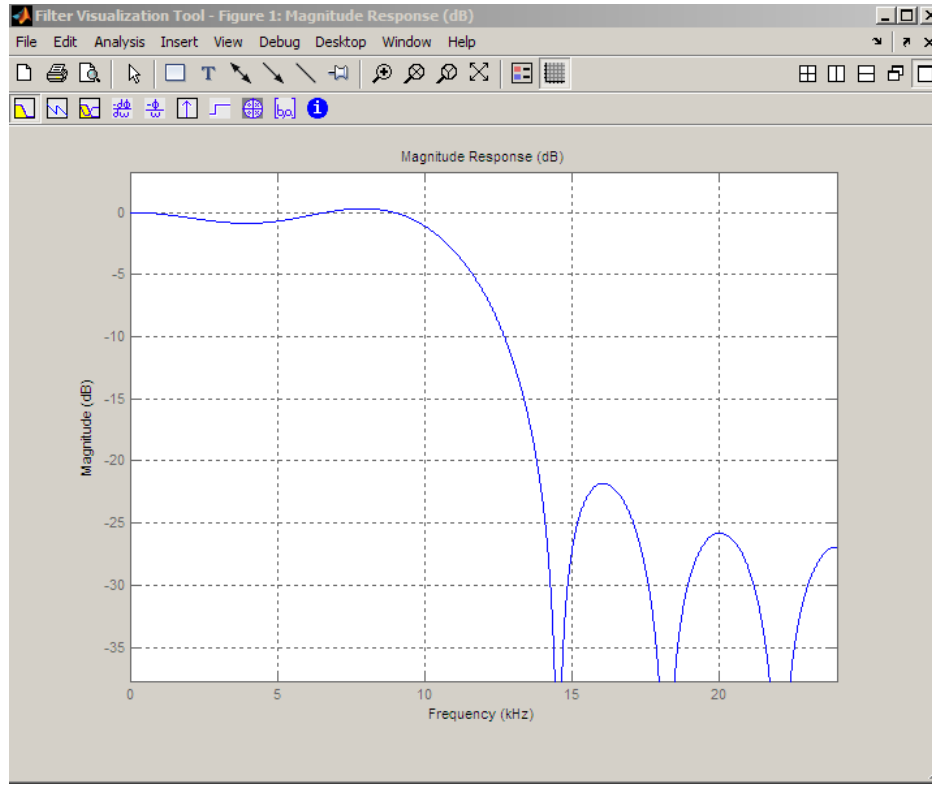
`fcnhndl` is a handle to a filter design function that returns a window vector, such as the `hamming` or `blackman` functions. `fcnarg` is an optional argument that returns a window. You pass the function to `window`. Refer to example 1 in the following section to see the function argument used to design the filter.

`h = window(d,win)` designs a filter using the vector you supply in `win`. The length of vector `win` must be the same as the impulse response of the filter, which is equal to the filter order plus one.

Example

Construct a lowpass filter specification object of order 10 with a cutoff frequency of 12 kilohertz. We use a sampling frequency of 48 kilohertz. Next we use a function handle to the function Kaiser to provide the window.

```
d=fdesign.lowpass('n,fc',10,12000,48000);
Hd=window(d,'window',@kaiser);
fvtool(Hd);
```



See Also

`design`, `designmethods`, `fdesign`

Purpose

Open Window Design and Analysis Tool

Syntax

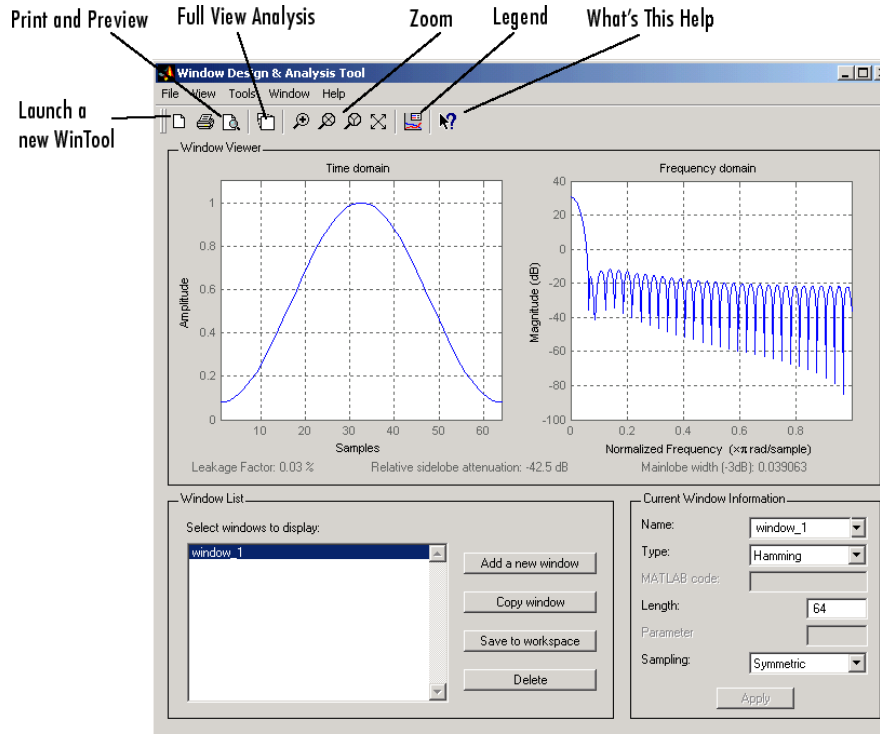
```
wintool
wintool(obj1,obj2,...)
```

Description

`wintool` opens the Window Design and Analysis Tool (WinTool), a graphical user interface (GUI) for designing and analyzing spectral windows. It opens with a default 64-point Hamming window.

`wintool(obj1,obj2,...)` opens WinTool with the `sigwin` window object(s) specified in `obj1`, `obj2`, etc.

Note A related tool, `wvtool`, is available for displaying, annotating, or printing windows.



`wintool` has three panels:

- Window Viewer displays the time domain and frequency domain representations of the selected window(s). The currently active window is shown in bold. Three window measurements are shown below the plots.
  - Leakage factor — ratio of power in the sidelobes to the total window power
  - Relative sidelobe attenuation — difference in height from the mainlobe peak to the highest sidelobe peak
  - Mainlobe width (-3dB) — width of the mainlobe at 3 dB below the mainlobe peak

- Window List lists the windows available for display in the Window Viewer. Highlight one or more windows to display them. The Window List buttons are:
  - **Add a new window** — Adds a default Hamming window with length 64 and symmetric sampling. You can change the information for this window by applying changes made in the **Current Window Information** panel.
  - **Copy window** — Copies the selected window(s).
  - **Save to workspace** — Saves the selected window(s) as vector(s) to the MATLAB workspace. The name of the window in `wintool` is used as the vector name.
  - **Delete** — Removes the selected window(s) from the window list.
- *Current Window Information* displays information about the currently active window. The active window name is shown in the `Name` field. To make another window active, select its name from the **Name** menu.

Window Parameters

Each window is defined by the parameters in the Current Window Information panel. You can change the current window's characteristics by changing its parameters and clicking **Apply**. The parameters of the current window are

- **Name** — Name of the window. The name is used for the legend in the Window Viewer, in the Window List, and for the vector saved to the workspace. You can either select a name from the menu or type the desired name in the edit box.
- **Type** — Algorithm for the window. Select the type from the menu. All Signal Processing Toolbox windows are available.
- **MATLAB code** — Any valid MATLAB expression that returns a vector defining the window if `Type = User Defined`.
- **Length** — Number of samples.

- **Parameter** — Additional parameter for windows that require it, such as Chebyshev, which requires you to specify the sidelobe attenuation. Note that the title "Parameter" changes to the appropriate parameter name.
- **Sampling** — Type of sampling to use for generalized cosine windows (Hamming, Hann, and Blackman) — `Periodic` or `Symmetric`. `Periodic` computes a length `n+1` window and returns the first `n` points, and `Symmetric` computes and returns the `n` points specified in `Length`.

WinTool Menus

In addition to the usual menus items, `wintool` contains these `wintool`-specific menu commands:

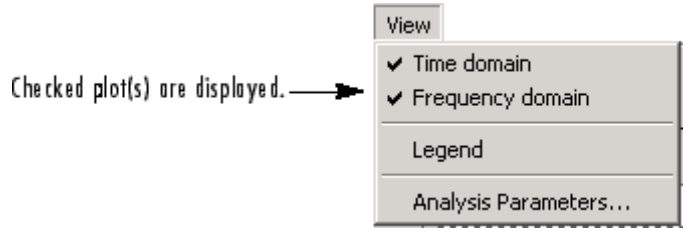
File menu:

- **Export** — Exports window coefficient vectors or `sigwin` window objects to the MATLAB workspace, a text file, or a MAT-file.

  In the **Window List** in WinTool, highlight the window(s) you want to export and then select **File > Export**. For exporting to the workspace or a MAT-file, specify the variable name for each window coefficient or object. To overwrite variables in the workspace, select the Overwrite variables check box.
- **Full View Analysis** — Copies the windows shown in both plots to a separate `wvtool` figure window. This is useful for printing and annotating. This option is also available with the Full View Analysis toolbar button.

View menu:

- **Time domain** — Select to show the time domain plot in the Window Viewer panel.
- **Frequency domain** — Select to show the frequency domain plot in the Window Viewer panel.



- **Legend** — Toggles the window name legend on and off. This option is also available with the Legend toolbar button.
- **Analysis Parameters** — Controls the response plot parameters, including number of points, range, *x*- and *y*-axis units, sampling frequency, and normalized magnitude.

  You can also access the Analysis Parameters by right-clicking the *x*-axis label of a plot in the Window Viewer panel. The *x*-axis units for the time domain plot depend on the selected `Sampling Frequency` units.

Frequency Domain	Time Domain
Hz	sec
kHz	ms
MHz	µs
GHz	picosec

Tools menu:

- **Zoom In** — Zooms in along both *x*- and *y*-axes.
- **Zoom X** — Zooms in along the *x*-axis only. Drag the mouse in the *x* direction to select the zoom area.
- **Zoom Y** — Zooms in along the y-axis only. Drag the mouse in the *y* direction to select the zoom area.
- **Full View** — Returns to full view.

See Also

window, wvtool

Purpose

Open Window Visualization Tool

Syntax

```
wvtool(winname(n))
wvtool(winname1(n),winname2(n),...winnamem(n))
wvtool(w)
h = wvtool(...)
```

Description

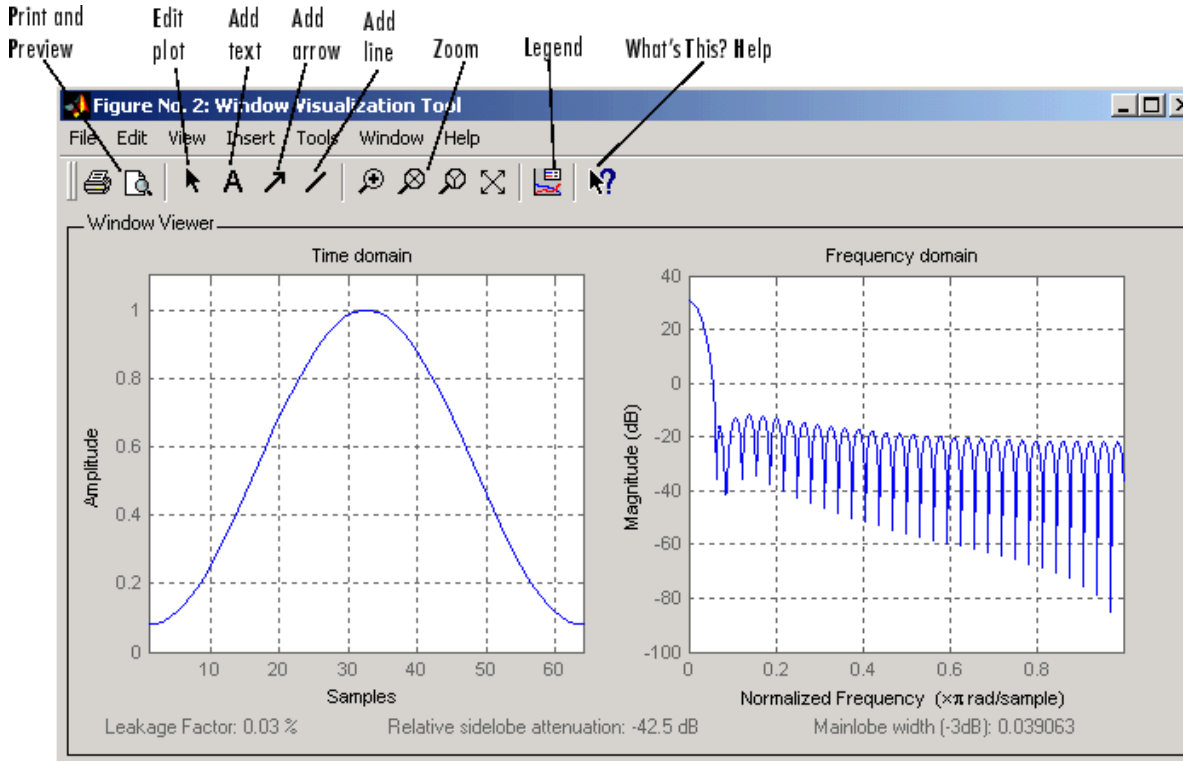
`wvtool(winname(n))` opens the Window Visualization Tool (WVTool) with the time and frequency domain plots of the `n`-length window specified in `winname`, which can be any Signal Processing Toolbox window. For a list of valid window names, see the `window` function. In the `wvtool` command, do not precede the window name with @.

`wvtool(winname1(n),winname2(n),...winnamem(n))` opens WVTool with a time-domain plot and a frequency-domain plot that contain all the windows specified in $\texttt{winname}_1$,...$\texttt{winname}_m$. The plots are shown on the same axes so that window characteristics can be compared and contrasted easily. WVTool is useful for displaying, annotating, and printing window responses.

`wvtool(w)` launches the Window Visualization Tool with `sigwin` object `w`.

`h = wvtool(...)` returns the Handle Graphics figure handle `h`.

Note A related tool, `wintool`, is available for designing and analyzing windows.



Note If you launch WVTool from FDATool, an **Add/Replace** icon, which controls how new windows are added from FDATool, appears on the toolbar.

WVTool Menus

In addition to the usual menus items, `wvtool` contains these `wvtool`-specific menu commands:

File menu:

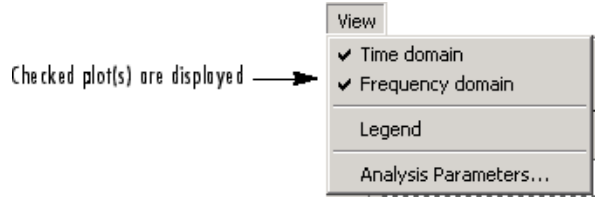
- **Export** — Exports the displayed plot(s) to a graphic file.

Edit menu:

- **Copy figure** — Copies the displayed plot(s) to the clipboard (available only on Windows platforms).
- **Copy options** — Displays the Preferences dialog box (available only on Windows platforms).
- **Figure**, **Axes**, and **Current Object Properties** — Displays the Property Editor.

View menu:

- **Time domain** — Check to show the time domain plot.
- **Frequency domain** — Check to show the frequency domain plot.



- **Legend** — Toggles the window name legend on and off. This option is also available with the **Legend** toolbar button.
- *Analysis Parameters* — Controls the response plot parameters, including number of points, range, *x*- and *y*-axis units, sampling frequency, and normalized magnitude.

  You can also access the Analysis Parameters by right-clicking the *x*-axis label of a plot in the Window Viewer panel.
- **Insert** menu:

You use the **Insert** menu to add labels, titles, arrows, lines, text, and axes to your plots.

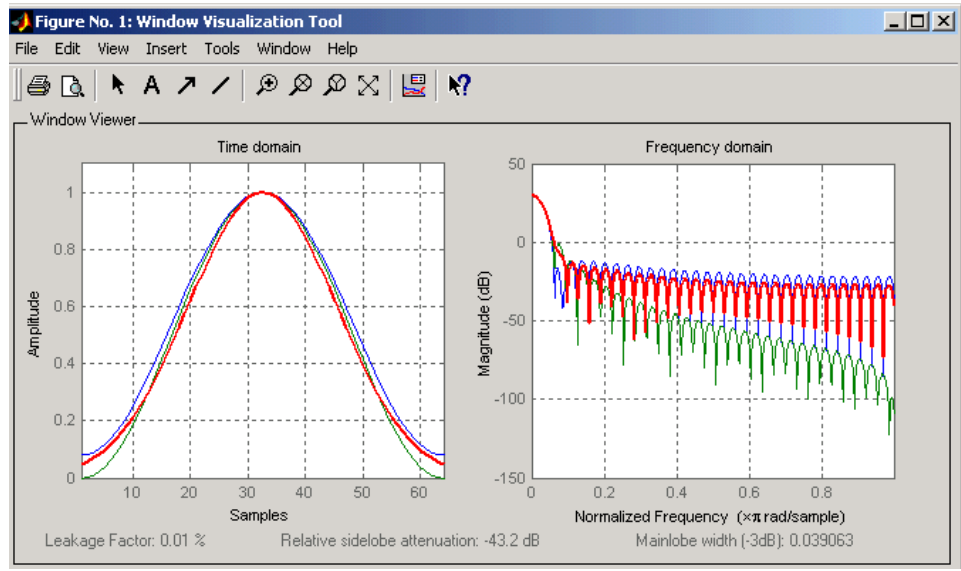
Tools menu:

- **Edit Plot** — Turns on plot editing mode
- **Zoom In** — Zooms in along both *x*- and *y*-axes.
- **Zoom X** — Zooms in along the *x*-axis only. Drag the mouse in the *x* direction to select the zoom area.
- **Zoom Y** — Zooms in along the y-axis only. Drag the mouse in the *y* direction to select the zoom area.
- **Full View** — Returns to full view.

Examples

Compare Hamming, Hann, and Gaussian windows:

```
wvtool(hamming(64),hann(64),gausswin(64))
```



See Also

`fdatool`, `window`, `wintool`

Purpose

Cross-correlation

Syntax

```
c = xcorr(x,y)
c = xcorr(x)
c = xcorr(x,y,'option')
c = xcorr(x,'option')
c = xcorr(x,y,maxlags)
c = xcorr(x,maxlags)
c = xcorr(x,y,maxlags,'option')
c = xcorr(x,maxlags,'option')
[c,lags] = xcorr(...)
```

Description

`xcorr` estimates the cross-correlation sequence of a random process. Autocorrelation is handled as a special case.

The true cross-correlation sequence is

$$R_{xy}(m) = E\{x_{n+m}y^*_n\} = E\{x_n y^*_{n-m}\}$$

where x_n and y_n are jointly stationary random processes, $-\infty < n < \infty$, and $E\{\}$ is the expected value operator. `xcorr` must estimate the sequence because, in practice, only a finite segment of one realization of the infinite-length random process is available.

`c = xcorr(x,y)` returns the cross-correlation sequence in a length $2*N-1$ vector, where `x` and `y` are length N vectors ($N>1$). If `x` and `y` are not the same length, the shorter vector is zero-padded to the length of the longer vector.

Note The maximum allowable vector length for inputs to `xcorr` is 2^20. If you need to process longer sequences, see `dfilt.fftfir`.

By default, `xcorr` computes raw correlations with no normalization.

$$\hat{R}_{xy}(m) = \begin{cases} \sum_{n=0}^{N-m-1} x_{n+m} y_n^* & m \geq 0 \\ \hat{R}_{yx}^*(-m) & m < 0 \end{cases}$$

The output vector `c` has elements given by $c(m) = R_{xy}(m\text{-}N)$, m=1, ..., 2N-1.

In general, the correlation function requires normalization to produce an accurate estimate (see below).

`c = xcorr(x)` is the autocorrelation sequence for the vector `x`. If `x` is an N-by-P matrix, c is a matrix with 2N-1 rows whose P^2 columns contain the cross-correlation sequences for all combinations of the columns of `x`. For more information on matrix processing with `xcorr`, see "Multiple Channels" on page 6-4.

`xcorr` produces correlations identically equal to 1.0 at zero lag only when you perform an autocorrelation and only when you set the `'coeff'` option. For example,

```
x=0:0.01:10;
X = sin(x);
[r,lags]=xcorr(X,'coeff');
max(r)

ans =
     1
```

`c = xcorr(x,y,'option')` specifies a normalization option for the cross-correlation, where *`'option'`* is

- `'biased'`: Biased estimate of the cross-correlation function

$$R_{xy,\,biased}(m) = \frac{1}{N} R_{xy}(m)$$

- `'unbiased'`: Unbiased estimate of the cross-correlation function

$$R_{xy,\,unbiased}(m) = \frac{1}{N - |m|} R_{xy}(m)$$

- `'coeff'`: Normalizes the sequence so the autocorrelations at zero lag are identically 1.0.
- `'none'`, to use the raw, unscaled cross-correlations (default)

See [1] for more information on the properties of biased and unbiased correlation estimates.

`c = xcorr(x,'option')` specifies one of the above normalization options for the autocorrelation.

`c = xcorr(x,y,maxlags)` returns the cross-correlation sequence over the lag range `[-maxlags:maxlags]`. Output `c` has length `2*maxlags+1`.

`c = xcorr(x,maxlags)` returns the autocorrelation sequence over the lag range `[-maxlags:maxlags]`. Output `c` has length `2*maxlags+1`. If `x` is an *N*-by-*P* matrix, `c` is a matrix with `2*maxlags+1` rows whose P^2 columns contain the autocorrelation sequences for all combinations of the columns of `x`.

`c = xcorr(x,y,maxlags,'option')` specifies both a maximum number of lags and a scaling option for the cross-correlation.

`c = xcorr(x,maxlags,'option')` specifies both a maximum number of lags and a scaling option for the autocorrelation.

`[c,lags] = xcorr(...)` returns a vector of the lag indices at which `c` was estimated, with the range `[-maxlags:maxlags]`. When `maxlags` is not specified, the range of `lags` is `[-N+1:N-1]`.

In all cases, the cross-correlation or autocorrelation computed by `xcorr` has the zeroth lag in the middle of the sequence, at element or row `maxlags+1` (element or row `N` if `maxlags` is not specified).

Examples

The second output, `lags`, is useful for plotting the cross-correlation or autocorrelation. For example, the estimated autocorrelation of zero-mean Gaussian white noise $c_{ww}(m)$ can be displayed for $-10 \le m \le 10$ using:

```
ww = randn(1000,1);
[c_ww,lags] = xcorr(ww,10,'coeff');
stem(lags,c_ww)
```

Swapping the `x` and `y` input arguments reverses (and conjugates) the output correlation sequence. For row vectors, the resulting sequences are reversed left to right; for column vectors, up and down. The following example illustrates this property (`mat2str` is used for a compact display of complex numbers):

```
x = [1,2i,3]; y = [4,5,6];
[c1,lags] = xcorr(x,y);
c1 = mat2str(c1,2), lags
c1 =
    [6-i*8.9e-016 5+i*12 22+i*10 15+i*8 12+i*8.9e-016]
lags =
    -2    -1     0     1     2
c2 = conj(fliplr(xcorr(y,x)));
c2 = mat2str(c2,2)
c2 =
    [6-i*8.9e-016 5+i*12 22+i*10 15+i*8 12+i*8.9e-016]
```

For the case where input argument `x` is a matrix, the output columns are arranged so that extracting a row and rearranging it into a square array produces the cross-correlation matrix corresponding to the lag of the chosen row. For example, the cross-correlation at zero lag can be retrieved by:

```
randn('state',0)
X = randn(2,2);
[M,P] = size(X);
c = xcorr(X);
c0 = zeros(P); c0(:) = c(M,:)    % Extract zero-lag row
c0 =
    2.9613   -0.5334
   -0.5334    0.0985
```

You can calculate the matrix of correlation coefficients that the MATLAB function `corrcoef` generates by substituting:

```
c = xcov(X,'coef')
```

in the last example. The function `xcov` subtracts the mean and then calls `xcorr`.

Use `fftshift` to move the second half of the sequence starting at the zeroth lag to the front of the sequence. `fftshift` swaps the first and second halves of a sequence.

Algorithm

For more information on estimating covariance and correlation functions, see [1].

References

[1] Orfanidis, S.J., *Optimum Signal Processing. An Introduction. 2nd Edition*, Prentice-Hall, Englewood Cliffs, NJ, 1996.

See Also

`conv`, `corrcoef`, `cov`, `xcorr2`, `xcov`

Purpose

2–D cross-correlation

Syntax

```
C = xcorr2(A,B)
xcorr2(A)
```

Description

`C = xcorr2(A,B)` returns the cross-correlation of matrices `A` and `B` with no scaling. `xcorr2` is the two-dimensional version of `xcorr`. It has its maximum value when the two matrices are aligned so that they are shaped as similarly as possible.

If matrix *A* has dimensions (*Ma*, *Na*) and matrix *B* has dimensions (*Mb*, *Nb*), The equation for the two-dimensional discrete cross-correlation is

$$C(i,j) = \sum_{m=0}^{(Ma-1)} \sum_{n=0}^{(Na-1)} A(m,n) \cdot \mathrm{conj}(B(m+i,n+j))$$

where $0 \le i < Ma + Mb - 1$ and $0 \le j < Na + Nb - 1$.

`xcorr2(A)` is the autocorrelation matrix of input matrix `A`. It is identical to `xcorr2(A,A)`.

Examples

Output Matrix Size

If matrix I1 has dimensions (4,3) and matrix I2 has dimensions (2,2), the following equations determine the number of rows and columns of the output matrix:

$$C_{\text{full}_{\text{rows}}} = I1_{\text{rows}} + I2_{\text{rows}} - 1 = 4 + 2 - 1 = 5$$

$$C_{\text{full}_{\text{columns}}} = I1_{\text{columns}} + I2_{\text{columns}} - 1 = 3 + 2 - 1 = 4$$

The resulting matrix is

$$C_{\text{full}} = \begin{bmatrix} c_{00} & c_{01} & c_{02} & c_{03} \\ c_{10} & c_{11} & c_{12} & c_{13} \\ c_{20} & c_{21} & c_{22} & c_{23} \\ c_{30} & c_{31} & c_{32} & c_{33} \\ c_{40} & c_{41} & c_{42} & c_{43} \end{bmatrix}$$

Computing a Specific Element

$$C_{\text{valid}_{\text{columns}}} = I1_{\text{columns}} - I2_{\text{columns}} + 1 = 2$$

In cross-correlation, the value of an output element is computed as a weighted sum of neighboring elements. For example, suppose the first input matrix represents an image and is defined as

```
I1 = [17  24   1   8  15
      23   5   7  14  16
       4   6  13  20  22
      10  12  19  21   3
      11  18  25   2   9]
```

The second input matrix also represents an image and is defined as

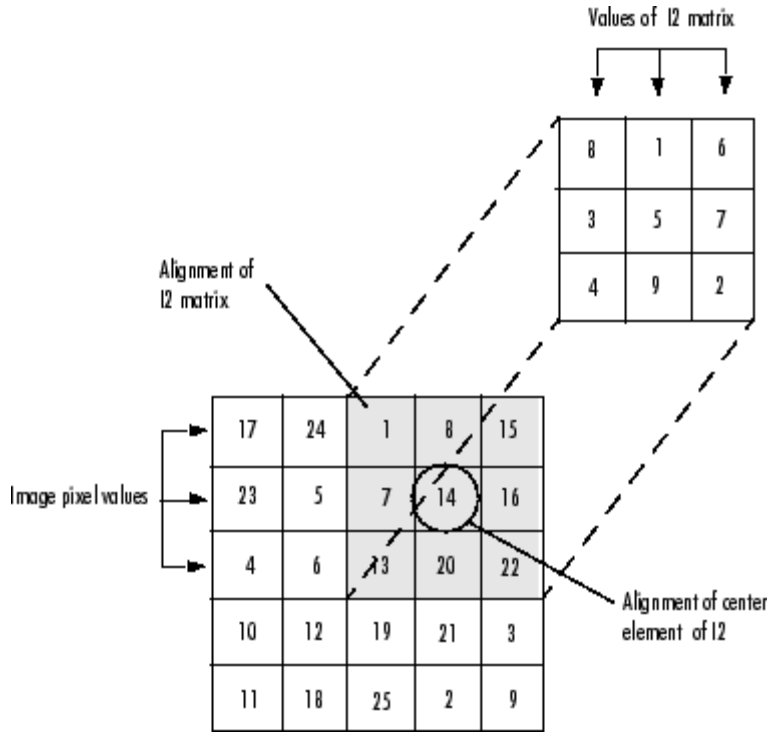
```
I2 = [8   1   6
      3   5   7
      4   9   2]
```

The following figure shows how to compute the (2,4) output element (zero-based indexing) using these steps:

1. Slide the center element of I2 so that lies on top of the (1,3) element of I1.
2. Multiply each weight in I2 by the element of I1 underneath.
3. Sum the individual products from step 2.

The (2,4) output element from the cross-correlation is

$$1 \cdot 8 + 8 \cdot 1 + 15 \cdot 6 + 7 \cdot 3 + 14 \cdot 5 + 16 \cdot 7 + 13 \cdot 4 + 20 \cdot 9 + 22 \cdot 2 = 585$$



The normalized cross-correlation of the (2,4) output element is

```
585/sqrt(sum(dot(I1p,I1p))*sum(dot(I2,I2))) = 0.8070
```

where I1p = [1 8 15; 7 14 16; 13 20 22].

See Also

`conv2`, `filter2`, `xcorr`

Purpose

Cross-covariance

Syntax

```
v = xcov(x,y)
v = xcov(x)
v = xcov(x,'option')
[c,lags] = xcov(x,y,maxlags)
[c,lags] = xcov(x,maxlags)
[c,lags] = xcov(x,maxlags)
[c,lags] = xcov(x,y,maxlags,'option')
```

Description

`xcov` estimates the cross-covariance sequence of random processes. Autocovariance is handled as a special case.

The true cross-covariance sequence is the cross-correlation of mean-removed sequences

$$\phi_{xy}(\mu) = E\{(x_{n+m} - \mu_x)(y_n - \mu_y)^*\}$$

where μ_x and μ_y are the mean values of the two stationary random processes, and $E\{\cdot\}$ is the expected value operator. `xcov` estimates the sequence because, in practice, access is available to only a finite segment of the infinite-length random process.

`v = xcov(x,y)` returns the cross-covariance sequence in a length $2N$-1 vector, where `x` and `y` are length N vectors. For information on how arrays are processed with xcov, see "Multiple Channels" on page 6-4.

`v = xcov(x)` is the autocovariance sequence for the vector `x`. Where `x` is an N-by-P array, v = xcov(x) returns an array with $2N$-1 rows whose P^2 columns contain the cross-covariance sequences for all combinations of the columns of `x`.

By default, `xcov` computes raw covariances with no normalization. For a length N vector

$$c_{xy}(m) = \begin{cases} \sum_{n=0}^{N-|m|-1} \left(x(n+m) - \frac{1}{N}\sum_{i=0}^{N-1} x_i \right)\left(y_n^* - \frac{1}{N}\sum_{i=0}^{N-1} y_i^* \right) & m \geq 0 \\ c_{yx}^*(-m) & m < 0 \end{cases}$$

The output vector `c` has elements given by `c(m)` = $c_{xy}(m\text{-}N)$, m = 1, ..., $2N$-1.

The covariance function requires normalization to estimate the function properly.

`v = xcov(x,'option')` specifies a scaling option, where *`'option'`* is

- `'biased'`, for biased estimates of the cross-covariance function
- `'unbiased'`, for unbiased estimates of the cross-covariance function
- `'coeff'`, to normalize the sequence so the auto-covariances at zero lag are identically 1.0
- `'none'`, to use the raw, unscaled cross-covariances (default)

See [1] for more information on the properties of biased and unbiased correlation and covariance estimates.

`[c,lags] = xcov(x,y,maxlags)` where `x` and `y` are length `m` vectors, returns the cross-covariance sequence in a length `2*maxlags+1` vector `c`. `lags` is a vector of the lag indices where `c` was estimated, that is, `[-maxlags:maxlags]`.

`[c,lags] = xcov(x,maxlags)` is the autocovariance sequence over the range of lags `[-maxlags:maxlags]`.

`[c,lags] = xcov(x,maxlags)` where `x` is an `m`-by-`p` array, returns array `c` with `2*maxlags+1` rows whose P^2 columns contain the cross-covariance sequences for all combinations of the columns of `x`.

`[c,lags] = xcov(x,y,maxlags,'option')` specifies a scaling option, where *`'option'`* is the last input argument.

In all cases, `xcov` gives an output such that the zeroth lag of the covariance vector is in the middle of the sequence, at element or row `maxlag+1` or at `m`.

Examples

The second output `lags` is useful when plotting. For example, the estimated autocovariance of uniform white noise $c_{ww}(m)$ can be displayed for $-10 \le m \le 10$ using:

```
ww = randn(1000,1);     % White Gaussian noise
[cov_ww,lags] = xcov(ww,10,'coeff');
stem(lags,cov_ww)
```

Algorithm

`xcov` computes the mean of its inputs, subtracts the mean, and then calls `xcorr`. For more information on estimating covariance and correlation functions, see [1].

Diagnostics

`xcov` does not check for any errors other than the correct number of input arguments. Instead, it relies on the error checking in `xcorr`.

References

[1] Orfanidis, S.J., *Optimum Signal Processing. An Introduction. 2nd Edition*, Prentice-Hall, Englewood Cliffs, NJ, 1996.

See Also

`conv`, `corrcoef`, `cov`, `xcorr`, `xcorr2`

Purpose

Recursive digital filter design

Syntax

```
[b,a] = yulewalk(n,f,m)
```

Description

`yulewalk` designs recursive IIR digital filters using a least-squares fit to a specified frequency response.

`[b,a] = yulewalk(n,f,m)` returns row vectors `b` and `a` containing the `n+1` coefficients of the order `n` IIR filter whose frequency-magnitude characteristics approximately match those given in vectors `f` and `m`:

- `f` is a vector of frequency points, specified in the range between 0 and 1, where 1 corresponds to half the sample frequency (the Nyquist frequency). The first point of `f` must be 0 and the last point 1, with all intermediate points in increasing order. Duplicate frequency points are allowed, corresponding to steps in the frequency response.
- `m` is a vector containing the desired magnitude response at the points specified in `f`.
- `f` and `m` must be the same length.
- `plot(f,m)` displays the filter shape.

The output filter coefficients are ordered in descending powers of z.

$$\frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \cdots + b(n+1)z^{-n}}{a(1) + a(2)z^{-1} + \cdots + a(n+1)z^{-n}}$$

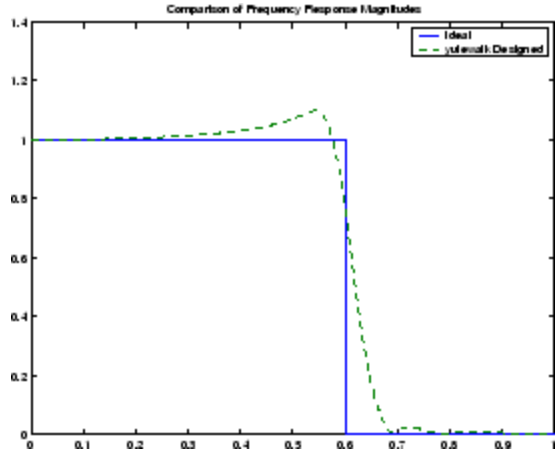
When specifying the frequency response, avoid excessively sharp transitions from passband to stopband. You may need to experiment with the slope of the transition region to get the best filter design.

Examples

Design an 8th-order lowpass filter and overplot the desired frequency response with the actual frequency response:

```
f = [0 0.6 0.6 1];
m = [1 1 0 0];
[b,a] = yulewalk(8,f,m);
```

```
[h,w] = freqz(b,a,128);
plot(f,m,w/pi,abs(h),'--')
legend('Ideal','yulewalk Designed')
title('Comparison of Frequency Response Magnitudes')
```



Algorithm

`yulewalk` performs a least-squares fit in the time domain. It computes the denominator coefficients using modified Yule-Walker equations, with correlation coefficients computed by inverse Fourier transformation of the specified frequency response. To compute the numerator, `yulewalk` takes the following steps:

1. Computes a numerator polynomial corresponding to an additive decomposition of the power frequency response.
2. Evaluates the complete frequency response corresponding to the numerator and denominator polynomials.
3. Uses a spectral factorization technique to obtain the impulse response of the filter.
4. Obtains the numerator polynomial by a least-squares fit to this impulse response.

References

[1] Friedlander, B., and B. Porat, "The Modified Yule-Walker Method of ARMA Spectral Estimation," *IEEE Transactions on Aerospace Electronic Systems, AES-20, No. 2* (March 1984), pp. 158-173.

See Also

`butter`, `cheby1`, `cheby2`, `ellip`, `fir2`, `firls`, `maxflat`, `firpm`

Purpose

Zero-phase response of digital filter

Syntax

```
[Hr,w] = zerophase(b,a)
[Hr,w] = zerophase(b,a,nfft)
[Hr,w] = zerophase(b,a,nfft,'whole')
[Hr,w] = zerophase(b,a,w)
[Hr,f] = zerophase(...,fs)
[Hr,w,phi] = zerophase(...)
zerophase(...)
```

Description

`[Hr,w] = zerophase(b,a)` returns the zero-phase response `Hr`, and the frequency vector `w` (in radians/sample) at which `Hr` is computed, given a filter defined by numerator `b` and denominator `a`. For FIR filters where `a=1`, you can omit the value `a` from the command. The zero-phase response is evaluated at `512` equally spaced points on the upper half of the unit circle.

The zero-phase response, `Hr(w)`, is related to the frequency response, `H(w)` by

$$H(e^{j\omega}) = Hr(\omega)\, e^{j\varphi(\omega)}$$

where $H(e^{j\omega})$ is the frequency response, $Hr(\omega)$ is the zero-phase response and [[PHI1]]*(w)* is the continuous phase.

Note The zero-phase response is always real, but it is not the equivalent of the magnitude response. The former can be negative while the latter cannot be negative.

`[Hr,w] = zerophase(b,a,nfft)` returns the zero-phase response `Hr` and frequency vector `w` (radians/sample), using `nfft` frequency points on the upper half of the unit circle.

`[Hr,w] = zerophase(b,a,nfft,'whole')` returns the zero-phase response `Hr` and frequency vector `w` (radians/sample), using `nfft` frequency points around the whole unit circle.

`[Hr,w] = zerophase(b,a,w)` returns the zero-phase response `Hr` and frequency vector `w` (radians/sample) at frequencies in vector `w`.

`[Hr,f] = zerophase(...,fs)` returns the zero-phase response `Hr` and frequency vector `f` (Hz), using the sampling frequency `fs` (in Hz), to determine the frequency vector `f` (in Hz) at which `Hr` is computed.

`[Hr,w,phi] = zerophase(...)` returns the zero-phase response `Hr`, frequency vector `w` (rad/sample), and the continuous phase component, `phi`. (Note that this quantity is not equivalent to the phase response of the filter when the zero-phase response is negative.)

`zerophase(...)` with no output arguments, plots the zero-phase response versus frequency.

Examples

Example 1

Plot the zero-phase response of a constrained least squares FIR filter:

```
b=fircls1(54,.3,.02,.008);
zerophase(b);
```

Example 2

Plot the zero-phase response of an elliptic filter:

```
[b,a]=ellip(10,.5,20,.4);
zerophase(b,a,512,'whole');
```

See Also

`freqs`, `freqz`, `fvtool`, `grpdelay`, `invfreqz`, `phasedelay`, `phasez`

Purpose

Convert zero-pole-gain filter parameters to second-order sections form

Syntax

```
[sos,g] = zp2sos(z,p,k,''order'')
[sos,g] = zp2sos(z,p,k,'order', 'scale')
[sos,g] = zp2sos(z,p,k,'order','scale',zeroflag)
sos = zp2sos(...)
```

Description

`zp2sos` converts a discrete-time zero-pole-gain representation of a given digital filter to an equivalent second-order section representation.

Use `[sos,g] = zp2sos(z,p,k)` to obtain a matrix `sos` in second-order section form with gain `g` equivalent to the discrete-time zero-pole-gain filter represented by input arguments `z`, `p`, and `k`. Vectors `z` and `p` contain the zeros and poles of the filter's transfer function $H(z)$, not necessarily in any particular order.

$$H(z) = k\frac{(z-z_1)(z-z_2)\cdots(z-z_n)}{(z-p_1)(z-p_2)\cdots(z-p_m)}$$

where n and m are the lengths of `z` and `p`, respectively, and `k` is a scalar gain. The zeros and poles must be real or complex conjugate pairs. `sos` is an L-by-6 matrix

$$sos = \begin{bmatrix} b_{01} & b_{11} & b_{21} & 1 & a_{11} & a_{21} \\ b_{02} & b_{12} & b_{22} & 1 & a_{12} & a_{22} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{0L} & b_{1L} & b_{2L} & 1 & a_{1L} & a_{2L} \end{bmatrix}$$

whose rows contain the numerator and denominator coefficients b_{ik} and a_{ik} of the second-order sections of $H(z)$.

$$H(z) = g\prod_{k=1}^{L} H_k(z) = g\prod_{k=1}^{L}\frac{b_{0k}+b_{1k}z^{-1}+b_{2k}z^{-2}}{1+a_{1k}z^{-1}+a_{2k}z^{-2}}$$

The number L of rows of the matrix `sos` is the closest integer greater than or equal to the maximum of $n/2$ and $m/2$.

`[sos,g] = zp2sos(z,p,k,'order')` specifies the order of the rows in `sos`, where *`'order'`* is

- `'down'`, to order the sections so the first row of `sos` contains the poles closest to the unit circle
- `'up'`, to order the sections so the first row of `sos` contains the poles farthest from the unit circle (default)

`[sos,g] = zp2sos(z,p,k,'order', 'scale')` specifies the desired scaling of the gain and the numerator coefficients of all second-order sections, where *`'scale'`* is

- `'none'`, to apply no scaling (default)
- `'inf'`, to apply infinity-norm scaling
- `'two'`, to apply 2-norm scaling

Using infinity-norm scaling in conjunction with `up`-ordering minimizes the probability of overflow in the realization. Using 2-norm scaling in conjunction with `down`-ordering minimizes the peak round-off noise.

Note Infinity-norm and 2-norm scaling are appropriate only for direct-form II implementations.

`[sos,g] = zp2sos(z,p,k,'order','scale',zeroflag)` specifies whether to keep together real zeros that are the negatives of each other instead of ordering them according to proximity to poles. Setting `zeroflag` to `true` keeps the zeros together and results in a numerator with a middle coefficient equal to zero. The default for `zeroflag` is `false`.

`sos = zp2sos(...)` embeds the overall system gain, `g`, in the first section, $H_1(z)$, so that

$$H(z) = \prod_{k=1}^{L} H_k(z)$$

Note Embedding the gain in the first section when scaling a direct-form II structure is not recommended and may result in erratic scaling. To avoid embedding the gain, use `ss2sos` with two outputs.

Examples

Find a second-order section form of a Butterworth lowpass filter:

```
[z,p,k] = butter(5,0.2);
sos = zp2sos(z,p,k);
```

Algorithm

`zp2sos` uses a four-step algorithm to determine the second-order section representation for an input zero-pole-gain system:

1. It groups the zeros and poles into complex conjugate pairs using the `cplxpair` function.
2. It forms the second-order section by matching the pole and zero pairs according to the following rules:
   a. Match the poles closest to the unit circle with the zeros closest to those poles.
   b. Match the poles next closest to the unit circle with the zeros closest to those poles.
   c. Continue until all of the poles and zeros are matched.

   `zp2sos` groups real poles into sections with the real poles closest to them in absolute value. The same rule holds for real zeros.
3. It orders the sections according to the proximity of the pole pairs to the unit circle. `zp2sos` normally orders the sections with poles closest to the unit circle last in the cascade. You can tell `zp2sos` to order the sections in the reverse order by specifying the `down` flag.

4 `zp2sos` scales the sections by the norm specified in the *`'scale'`* argument. For arbitrary $H(\omega)$, the scaling is defined by

$$\|H\|_p = \left[\frac{1}{2\pi}\int_0^{2\pi} |H(\omega)|^p d\omega\right]^{\frac{1}{p}}$$

where p can be either ∞ or 2. See the references for details on the scaling. This scaling is an attempt to minimize overflow or peak round-off noise in fixed point filter implementations.

References

[1] Jackson, L.B., *Digital Filters and Signal Processing*, 3rd ed., Kluwer Academic Publishers, Boston, 1996, Chapter 11.

[2] Mitra, S.K., *Digital Signal Processing: A Computer-Based Approach*, McGraw-Hill, New York, 1998, Chapter 9.

[3] Vaidyanathan, P.P., "Robust Digital Filter Structures," *Handbook for Digital Signal Processing*, S.K. Mitra and J.F. Kaiser, ed., John Wiley & Sons, New York, 1993, Chapter 7.

See Also

`cplxpair`, `filternorm`, `sos2zp`, `ss2sos`, `tf2sos`, `zp2ss`, `zp2tf`

Purpose

Convert zero-pole-gain filter parameters to state-space form

Syntax

```
[A,B,C,D] = zp2ss(z,p,k)
```

Description

`zp2ss` converts a zero-pole-gain representation of a given system to an equivalent state-space representation.

`[A,B,C,D] = zp2ss(z,p,k)` finds a single input, multiple output, state-space representation

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

given a system in factored transfer function form.

$$H(s) = \frac{Z(s)}{P(s)} = k\frac{(s-z_1)(s-z_2)\cdots(s-z_n)}{(s-p_1)(s-p_2)\cdots(s-p_n)}$$

Column vector `p` specifies the pole locations, and matrix `z` the zero locations with as many columns as there are outputs. The gains for each numerator transfer function are in vector `k`. The `A`, `B`, `C`, and `D` matrices are returned in controller canonical form.

`Inf` values may be used as place holders in `z` if some columns have fewer zeros than others.

Algorithm

`zp2ss`, for single-input systems, groups complex pairs together into two-by-two blocks down the diagonal of the `A` matrix. This requires the zeros and poles to be real or complex conjugate pairs.

See Also

`sos2ss`, `ss2zp`, `tf2ss`, `zp2sos`, `zp2tf`

Purpose

Convert zero-pole-gain filter parameters to transfer function form

Syntax

```
[b,a] = zp2tf(z,p,k)
```

Description

`zp2tf` forms transfer function polynomials from the zeros, poles, and gains of a system in factored form.

`[b,a] = zp2tf(z,p,k)` finds a rational transfer function

$$\frac{B(s)}{A(s)} = \frac{b_1 s^{(n-1)} + \cdots + b_{(n-1)} s + b_n}{a_1 s^{(m-1)} + \cdots + a_{(m-1)} s + a_m}$$

given a system in factored transfer function form

$$H(s) = \frac{Z(s)}{P(s)} = k\frac{(s-z_1)(s-z_2)\cdots(s-z_m)}{(s-p_1)(s-p_2)\cdots(s-p_n)}$$

Column vector `p` specifies the pole locations, and matrix `z` specifies the zero locations, with as many columns as there are outputs. The gains for each numerator transfer function are in vector `k`. The zeros and poles must be real or come in complex conjugate pairs. The polynomial denominator coefficients are returned in row vector `a` and the polynomial numerator coefficients are returned in matrix `b`, which has as many rows as there are columns of `z`.

`Inf` values can be used as place holders in `z` if some columns have fewer zeros than others.

Algorithm

The system is converted to transfer function form using `poly` with `p` and the columns of `z`.

See Also

`sos2tf`, `ss2tf`, `tf2zp`, `tf2zpk`, `zp2sos`, `zp2ss`

Purpose

Zero-pole plot

Syntax

```
zplane(z,p)
zplane(b,a)
zplane(Hd)
[hz,hp,ht] = zplane(z,p)
```

Description

This function displays the poles and zeros of discrete-time systems.

`zplane(z,p)` plots the zeros specified in column vector `z` and the poles specified in column vector `p` in the current figure window. The symbol `'o'` represents a zero and the symbol `'x'` represents a pole. The plot includes the unit circle for reference. If `z` and `p` are arrays, `zplane` plots the poles and zeros in the columns of `z` and `p` in different colors.

You can override the automatic scaling of `zplane` using

```
axis([xmin xmax ymin ymax])
```

or

```
set(gca,'ylim',[ymin ymax])
```

or

```
set(gca,'xlim',[xmin xmax])
```

after calling `zplane`. This is useful in the case where one or a few of the zeros or poles have such a large magnitude that the others are grouped tightly around the origin and are hard to distinguish.

`zplane(b,a)` where `b` and `a` are row vectors, first uses `roots` to find the zeros and poles of the transfer function represented by numerator coefficients `b` and denominator coefficients `a`. The transfer function is defined in terms of z^{-1}:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \cdots + b(n+1)z^{-n}}{a(1) + a(2)z^{-1} + \cdots + a(m+1)z^{-m}}$$

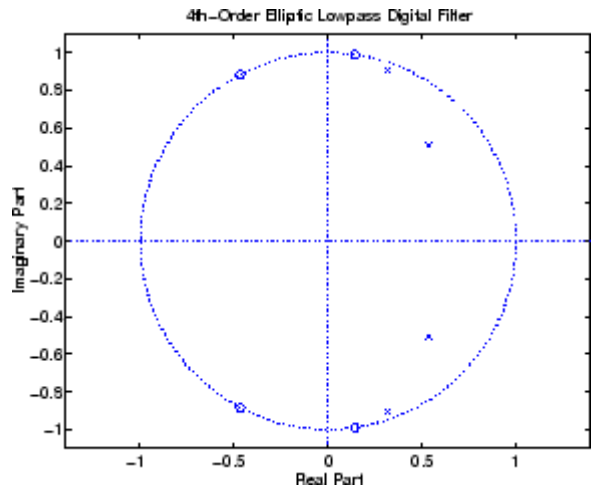
`zplane(Hd)` finds the zeros and poles of the transfer function represented by the `dfilt` filter object `Hd`. The pole-zero plot is displayed in `fvtool`.

`[hz,hp,ht] = zplane(z,p)` returns vectors of handles to the zero lines, `hz`, and the pole lines, `hp`. `ht` is a vector of handles to the axes/unit circle line and to text objects, which are present when there are multiple zeros or poles. If there are no zeros or no poles, `hz` or `hp` is the empty matrix `[]`.

Examples

For data sampled at 1000 Hz, plot the poles and zeros of a 4th-order elliptic lowpass digital filter with cutoff frequency of 200 Hz, 3 dB of ripple in the passband, and 30 dB of attenuation in the stopband:

```
[z,p,k] = ellip(4,3,30,200/500);
zplane(z,p);
title('4th-Order Elliptic Lowpass Digital Filter');
```



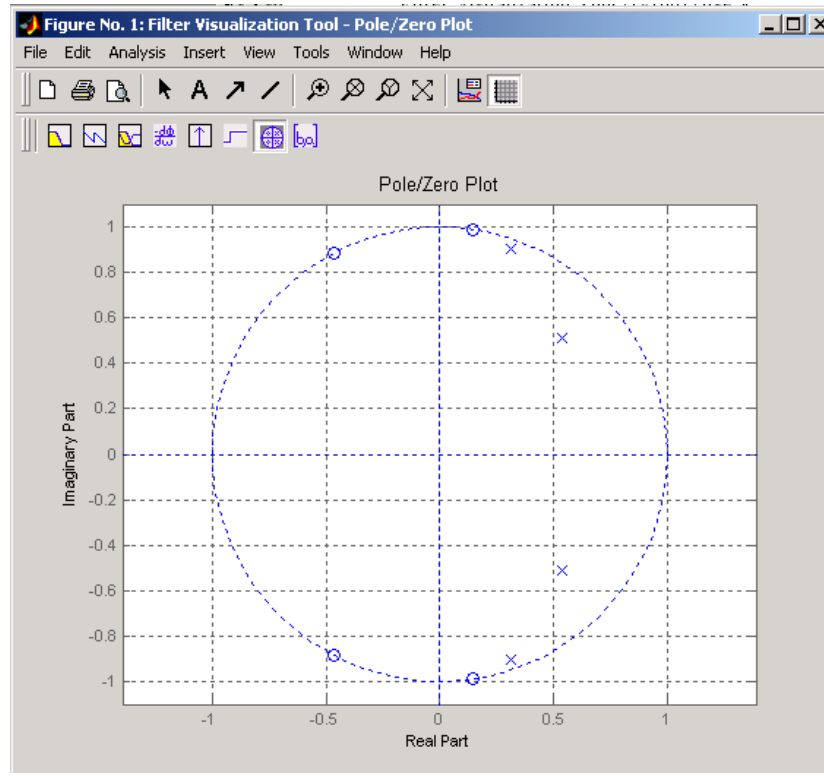
To generate the same plot with a transfer function representation of the filter, use:

```
[b,a] = ellip(4,3,30,200/500);    % Transfer function
```

```
zplane(b,a)
```

To generate the same plot using a `dfilt` object and displaying the result in the Filter Visualization Tool (`fvtool`) use:

```
[b,a] = ellip(4,3,30,200/500);
Hd=dfilt.df1(b,a);
zplane(Hd)
```



See Also

`freqz`

Technical Conventions

This manual and all Signal Processing Toolbox functions use the following technical notations.

Term or Symbol	Description
Nyquist frequency	One-half the sampling frequency. Some toolbox functions normalize this value to 1.
`x(1)`	The first element of a data sequence or filter, corresponding to zero lag.
Ω or w	Analog frequency in radians per second.
ω or w	Digital frequency in radians per sample.
`f`	Digital frequency in hertz.
[*x*, *y*)	The interval from *x* to *y*, including *x* but not including *y*.
`...`	Ellipses in the argument list for a given syntax on a function reference page indicate all possible argument lists for that function appearing prior to the given syntax are valid.

B

Examples

Use this list to find examples in the documentation.

Filtering

IIR Filter Design

FIR Filter Design

Spectral Analysis

Parametric Modeling

Windows

Cepstrum and Transforms

Index

Symbols and Numerics

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z