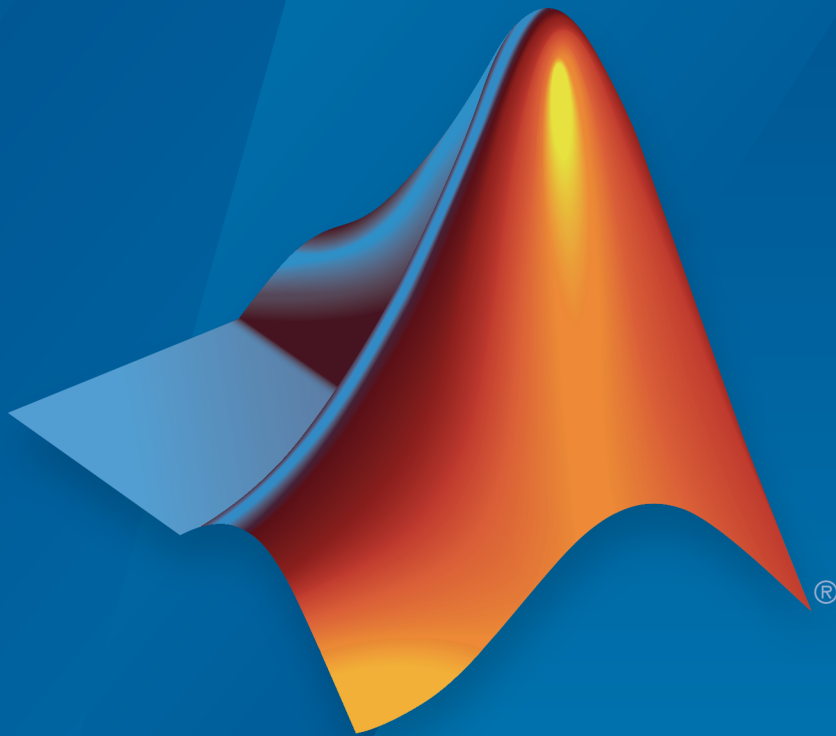


# Signal Processing Toolbox™

## Reference



# MATLAB®

R2015a

 MathWorks®

## How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://www.mathworks.com/support/contact_us)



Phone: 508-647-7000



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

*Signal Processing Toolbox™ Reference*

© COPYRIGHT 1988–2015 by The MathWorks, Inc.

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

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

### Trademarks

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

### Patents

MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://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)  |
| September 2009 | Online only     | Revised for Version 6.12 (Release 2009b)  |
| March 2010     | Online only     | Revised for Version 6.13 (Release 2010a)  |
| September 2010 | Online only     | Revised for Version 6.14 (Release 2010b)  |
| April 2011     | Online only     | Revised for Version 6.15 (Release 2011a)  |
| September 2011 | Online only     | Revised for Version 6.16 (Release 2011b)  |
| March 2012     | Online only     | Revised for Version 6.17 (Release 2012a)  |
| September 2012 | Online only     | Revised for Version 6.18 (Release 2012b)  |
| March 2013     | Online only     | Revised for Version 6.19 (Release 2013a)  |
| September 2013 | Online only     | Revised for Version 6.20 (Release 2013b)  |
| March 2014     | Online only     | Revised for Version 6.21 (Release 2014a)  |
| October 2014   | Online only     | Revised for Version 6.22 (Release 2014b)  |
| March 2015     | Online only     | Revised for Version 7.0 (Release 2015a)   |



|          |                                      |
|----------|--------------------------------------|
| <b>1</b> | <b>Functions — Alphabetical List</b> |
|----------|--------------------------------------|



# Functions — Alphabetical List

---

# abs

Absolute value (magnitude)

## Description

abs is a MATLAB® function.

## Examples

### Magnitude of the DFT of a Sequence

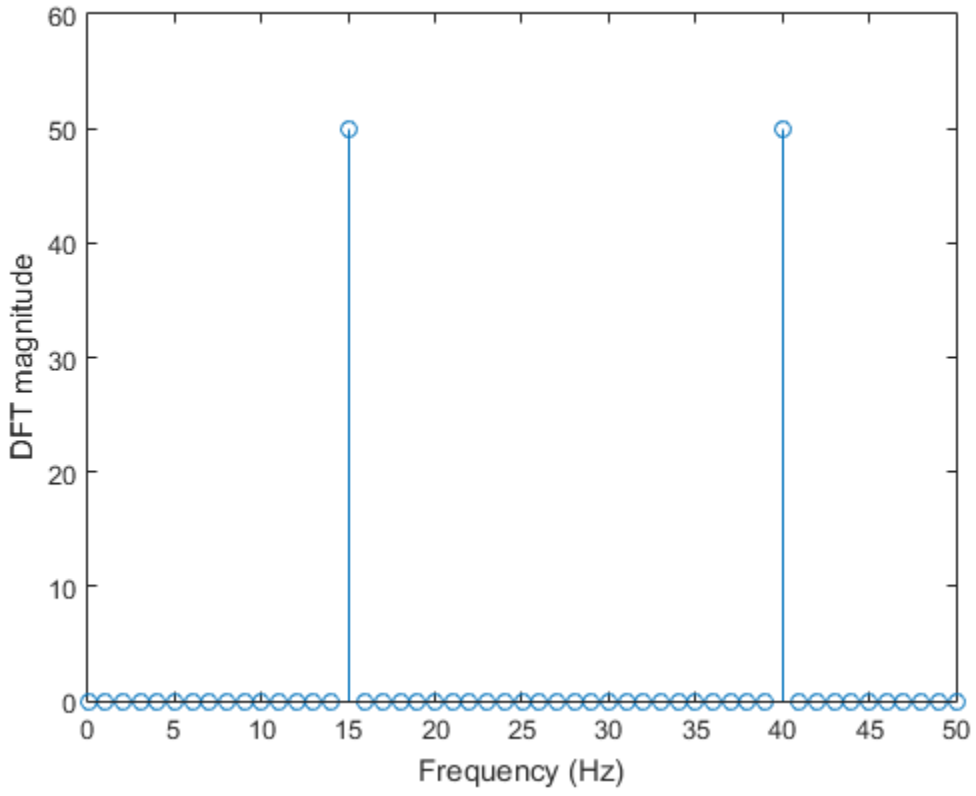
Generate a signal composed of two sinusoids sampled at 100 Hz. Specify the sinusoid frequencies as 15 and 40 Hz. Compute the DFT of the sequence.

```
t = (0:99)/100;           % Time vector
x = sin(2*pi*15*t) + sin(2*pi*40*t); % Signal
y = fft(x);              % DFT of x
m = abs(y);              % Magnitude
```

Plot the magnitude of the DFT.

```
f = 0:50;                % Frequency vector
m = m(1:51);            % Unique magnitudes
stem(f,m)
ylabel 'DFT magnitude'
xlabel 'Frequency (Hz)'
```





## ac2poly

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 polynomial represents the coefficients of a prediction filter that outputs a signal with autocorrelation sequence approximately equal to `r`.

`[a,efinal] = ac2poly(r)` returns the final prediction error, `efinal`, determined by running the filter for `length(r)` steps.

### Examples

#### Prediction Polynomial from Autocorrelation Sequence

Given an autocorrelation sequence, `r`, determine the equivalent linear prediction filter polynomial and the final prediction error.

```
r = [5.0000 -1.5450 -3.9547 3.9331 1.4681 -4.7500];
```

```
[a,efinal] = ac2poly(r)
```

```
a =
```

```
    1.0000    0.6147    0.9898    0.0004    0.0034   -0.0077
```

```
efinal =
```

0.1791

## More About

### Tips

You can apply this function to real or complex data.

## References

- [1] Kay, Steven M. *Modern Spectral Estimation*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

### See Also

ac2rc | poly2ac | rc2poly

## ac2rc

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`.

### More About

#### Tips

You can apply this function to real or complex data.

### References

[1] Kay, Steven M. *Modern Spectral Estimation*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

### See Also

`ac2poly` | `rc2ac` | `poly2rc`

# angle

Phase angle

## Description

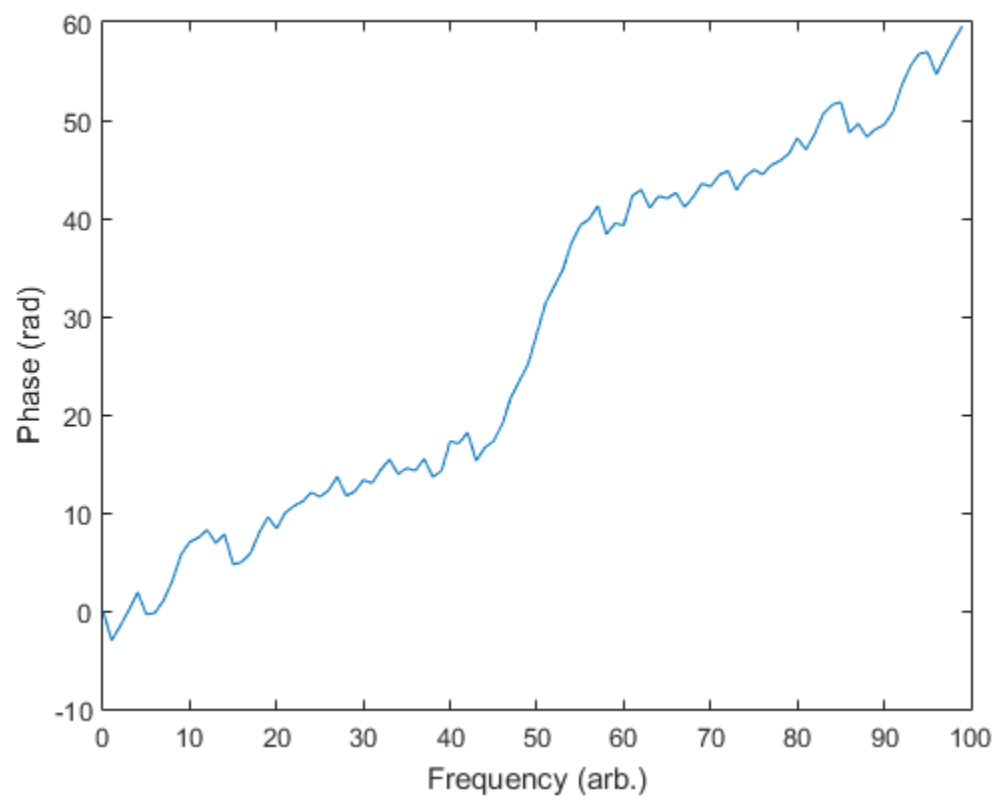
`angle` is a MATLAB function.

## Examples

### FFT Phase

Generate a signal and compute its DFT. Find the phase of the transform and plot it as a function of frequency.

```
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
f = (0:length(y)-1)/length(y)*100; % Frequency vector
plot(f,p)
xlabel 'Frequency (arb.)'
ylabel 'Phase (rad)'
```



# arburg

Autoregressive all-pole model parameters — Burg's method

## Syntax

```
a = arburg(x,p)
[a,e] = arburg(x,p)
[a,e,rc] = arburg(x,p)
```

## Description

`a = arburg(x,p)` returns the normalized autoregressive (AR) parameters corresponding to a model of order `p` for the input array, `x`. If `x` is a vector, then the output array, `a`, is a row vector. If `x` is a matrix, then the parameters along the  $n$ th row of `a` model the  $n$ th column of `x`. `a` has `p + 1` columns. `p` must be less than the number of elements (or rows) of `x`.

`[a,e] = arburg(x,p)` returns the estimated variance, `e`, of the white noise input.

`[a,e,rc] = arburg(x,p)` returns the reflection coefficients in `rc`.

## Examples

### Parameter Estimation Using Burg's Method

Use a vector of polynomial coefficients to generate an AR(4) process by filtering 1024 samples of white noise. Reset the random number generator for reproducible results. Use Burg's method to estimate the coefficients.

```
rng default
A = [1 -2.7607 3.8106 -2.6535 0.9238];
y = filter(1,A,0.2*randn(1024,1));
```

```
arcoeffs = arburg(y,4)
```

```
arcoeffs =
```

```
    1.0000   -2.7743    3.8408   -2.6843    0.9360
```

Generate 50 realizations of the process, changing each time the variance of the input noise. Compare the Burg-estimated variances to the actual values.

```
nrealiz = 50;
```

```
noisestdz = rand(1,nrealiz)+0.5;
```

```
randnoise = randn(1024,nrealiz);
```

```
for k = 1:nrealiz
```

```
    y = filter(1,A,noisestdz(k) * randnoise(:,k));
```

```
    [arcoeffs,noisevar(k)] = arburg(y,4);
```

```
end
```

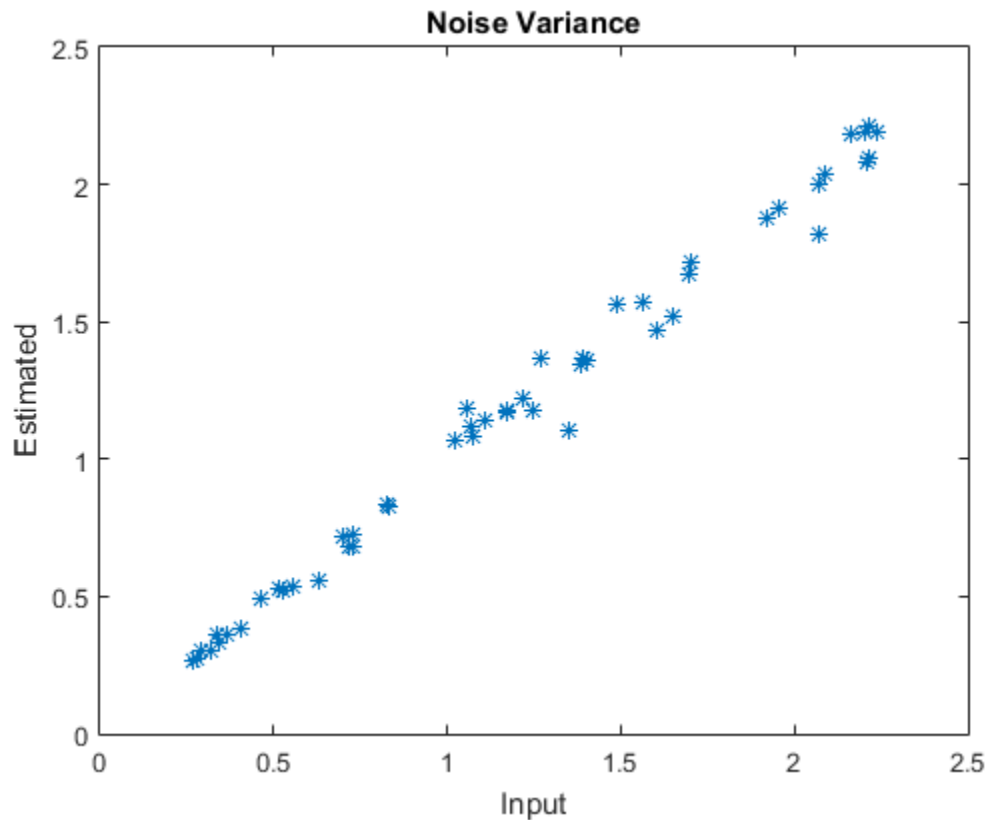
```
plot(noisestdz.^2,noisevar,'*')
```

```
title('Noise Variance')
```

```
xlabel('Input')
```

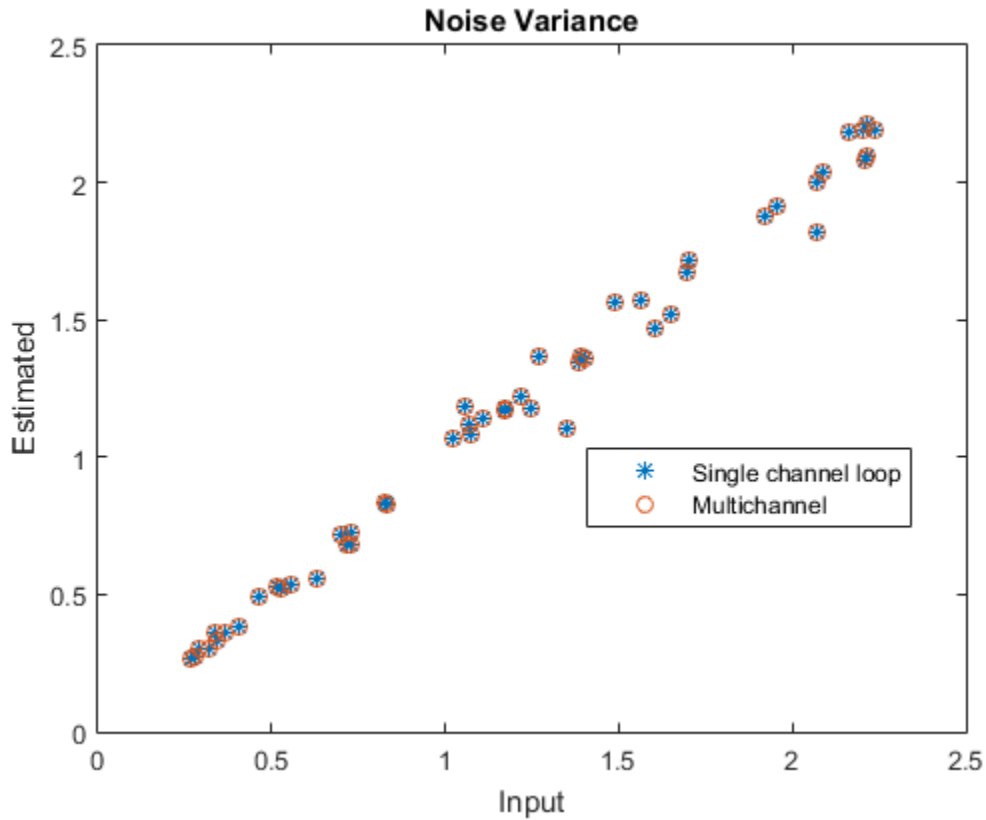
```
ylabel('Estimated')
```





Repeat the procedure using arburg's multichannel syntax.

```
realiz = bsxfun(@times,noisestdz,randnoise);
Y = filter(1,A,realiz);
[coeffs,variances] = arburg(Y,4);
hold on
plot(noisestdz.^2,variances,'o')
q = legend('Single channel loop','Multichannel');
q.Location = 'best';
```



## More About

### AR(p) Model

In an AR model of order  $p$ , the current output is a linear combination of the past  $p$  outputs plus a white noise input. The weights on the  $p$  past outputs minimize the mean-square prediction error of the autoregression. If  $y(n)$  is the current value of the output and  $x(n)$  is a zero mean white noise input, the AR( $p$ ) model is:

$$y(n) + \sum_{k=1}^p a(k)y(n-k) = x(n).$$

## Reflection Coefficients

The *reflection coefficients* are the partial autocorrelation coefficients scaled by  $-1$ . The reflection coefficients indicate the time dependence between  $y(n)$  and  $y(n - k)$  after subtracting the prediction based on the intervening  $k - 1$  time steps.

## Algorithms

The Burg method estimates the reflection coefficients and uses the reflection coefficients to estimate the AR parameters recursively. You can find the recursion and lattice filter relations describing the update of the forward and backward prediction errors in [1].

- “Parametric Modeling”

## References

[1] Kay, Steven M. *Modern Spectral Estimation: Theory and Application*. Englewood Cliffs, NJ: Prentice Hall, 1988.

## See Also

arcov | armcov | aryule | levinson | lpc

## **arconv**

Autoregressive all-pole model parameters — covariance method

### **Syntax**

```
a = arconv(x,p)
[a,e] = arconv(x,p)
```

### **Description**

`a = arconv(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. The output array,  $\mathbf{a}$ , contains normalized estimates of the AR system parameters,  $A(z)$ , in descending powers of  $z$ .  $\mathbf{a}$  has  $p + 1$  columns. If  $x$  is a vector, then  $\mathbf{a}$  is a row vector. If  $\mathbf{a}$  is a matrix, then the coefficients along the  $n$ th row of  $\mathbf{a}$  model the  $n$ th column of  $x$ .

`[a,e] = arconv(x,p)` returns the variance estimate,  $e$ , of the white noise input to the AR model.

### **Examples**

#### **Parameter Estimation Using the Covariance Method**

Use a vector of polynomial coefficients to generate an AR(4) process by filtering 1024 samples of white noise. Reset the random number generator for reproducible results. Use the covariance method to estimate the coefficients.

```
rng default
A = [1 -2.7607 3.8106 -2.6535 0.9238];
y = filter(1,A,0.2*randn(1024,1));
```

```
arcoeffs = arccov(y,4)
```

```
arcoeffs =
```

```
    1.0000   -2.7746    3.8419   -2.6857    0.9367
```

Generate 50 realizations of the process, changing each time the variance of the input noise. Compare the covariance-estimated variances to the actual values.

```
nrealiz = 50;
```

```
noisestdz = rand(1,nrealiz)+0.5;
```

```
randnoise = randn(1024,nrealiz);
```

```
for k = 1:nrealiz
```

```
    y = filter(1,A,noisestdz(k) * randnoise(:,k));
```

```
    [arcoeffs,noisevar(k)] = arccov(y,4);
```

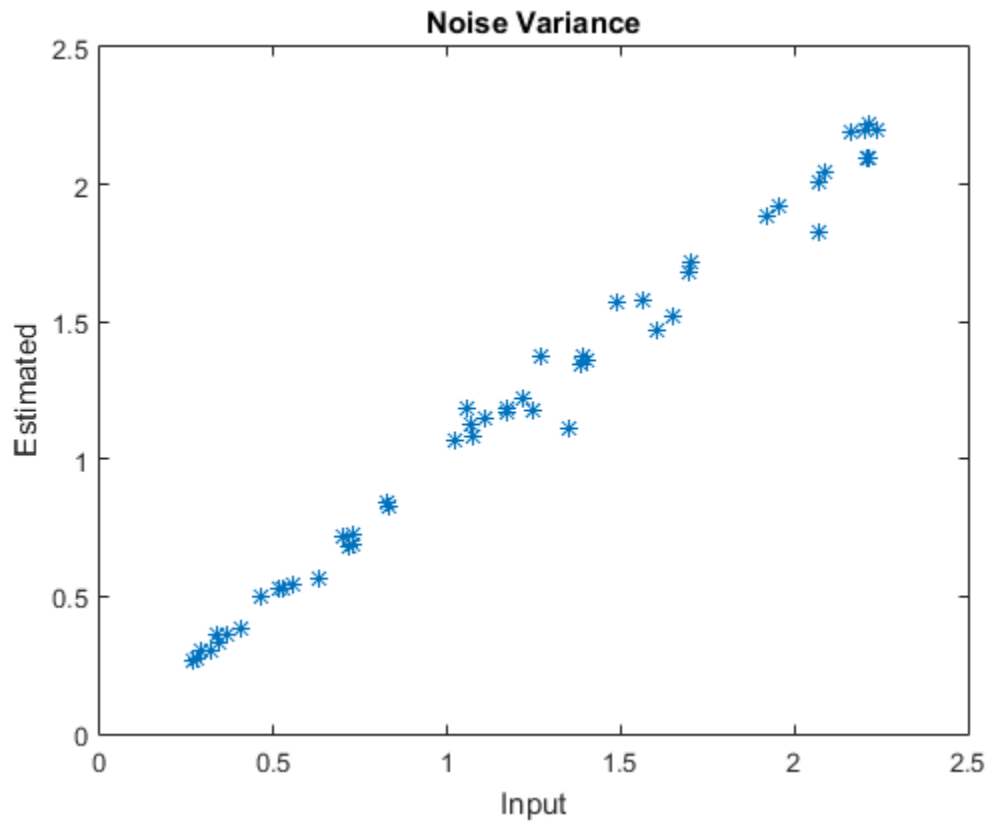
```
end
```

```
plot(noisestdz.^2,noisevar,'*')
```

```
title('Noise Variance')
```

```
xlabel('Input')
```

```
ylabel('Estimated')
```



Repeat the procedure using `arccov`'s multichannel syntax.

```

realiz = bsxfun(@times,noisestdz,randnoise);

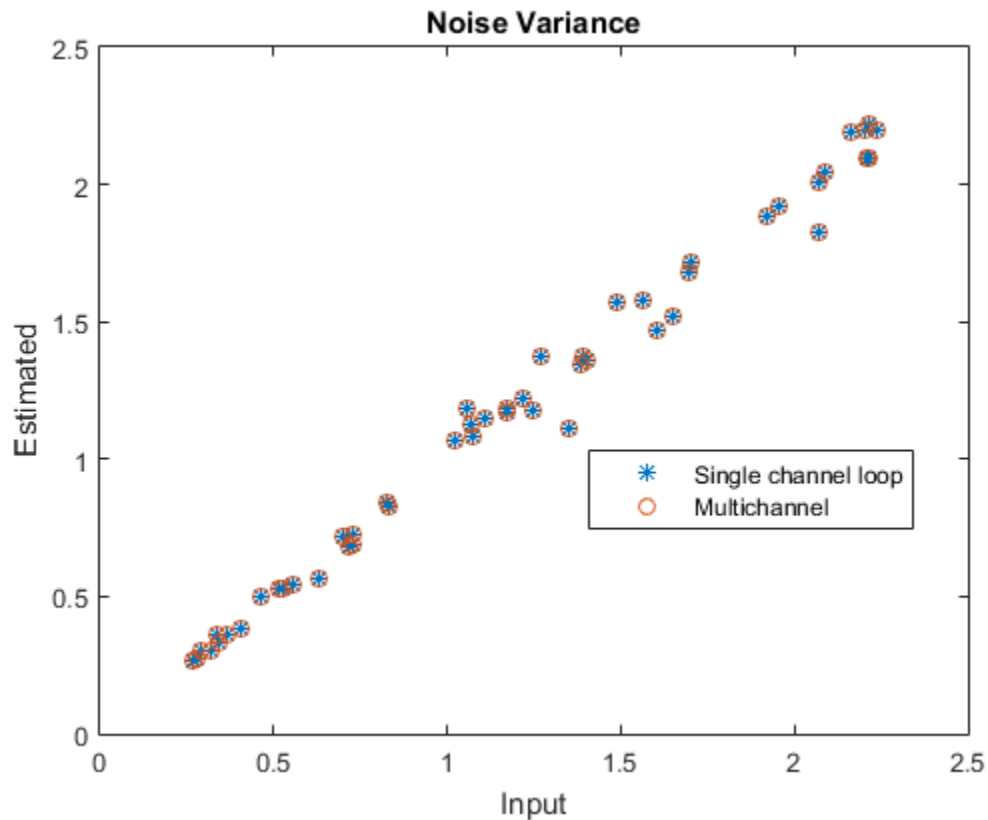
Y = filter(1,A,realiz);

[coeffs,variances] = arccov(Y,4);

hold on
plot(noisestdz.^2,variances,'o')

q = legend('Single channel loop','Multichannel');
q.Location = 'best';

```



## More About

### AR(p) Model

Let  $y(n)$  be a wide-sense stationary random process obtained by filtering white noise of variance  $e$  with the system function  $A(z)$ . If  $P_y(e^{j\omega})$  is the power spectral density of  $y(n)$ , then

$$P_y(e^{j\omega}) = \frac{e}{|A(e^{j\omega})|^2} = \frac{e}{\left|1 + \sum_{k=1}^p \alpha(k)e^{-j\omega k}\right|^2}.$$

Because the method characterizes the input data using an all-pole model, the correct choice of the model order,  $p$ , is important.

**See Also**

arburg | armcov | aryule | lpc | pcov | prony



## armcov

Autoregressive all-pole model parameters — 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. The output array,  $\mathbf{a}$ , contains the normalized estimates of the AR system parameters,  $A(z)$ , in descending powers of  $z$ .  $\mathbf{a}$  has  $p + 1$  columns. If  $x$  is a vector, then  $\mathbf{a}$  is a row vector. If  $\mathbf{a}$  is a matrix, then the coefficients along the  $n$ th row of  $\mathbf{a}$  model the  $n$ th column of  $x$ .

`[a,e] = armcov(x,p)` returns the variance estimate,  $\mathbf{e}$ , of the white noise input to the AR model.

### Examples

#### Parameter Estimation Using the Modified Covariance Method

Use a vector of polynomial coefficients to generate an AR(4) process by filtering 1024 samples of white noise. Reset the random number generator for reproducible results. Use the modified covariance method to estimate the coefficients.

```
rng default
A = [1 -2.7607 3.8106 -2.6535 0.9238];
y = filter(1,A,0.2*randn(1024,1));
```

```
arcoeffs = armcov(y,4)
```

```
arcoeffs =
```

```
    1.0000   -2.7741    3.8404   -2.6841    0.9360
```

Generate 50 realizations of the process, changing each time the variance of the input noise. Compare the modified-covariance-estimated variances to the actual values.

```
nrealiz = 50;
```

```
noisestdz = rand(1,nrealiz)+0.5;
```

```
randnoise = randn(1024,nrealiz);
```

```
for k = 1:nrealiz
```

```
    y = filter(1,A,noisestdz(k) * randnoise(:,k));
```

```
    [arcoeffs,noisevar(k)] = armcov(y,4);
```

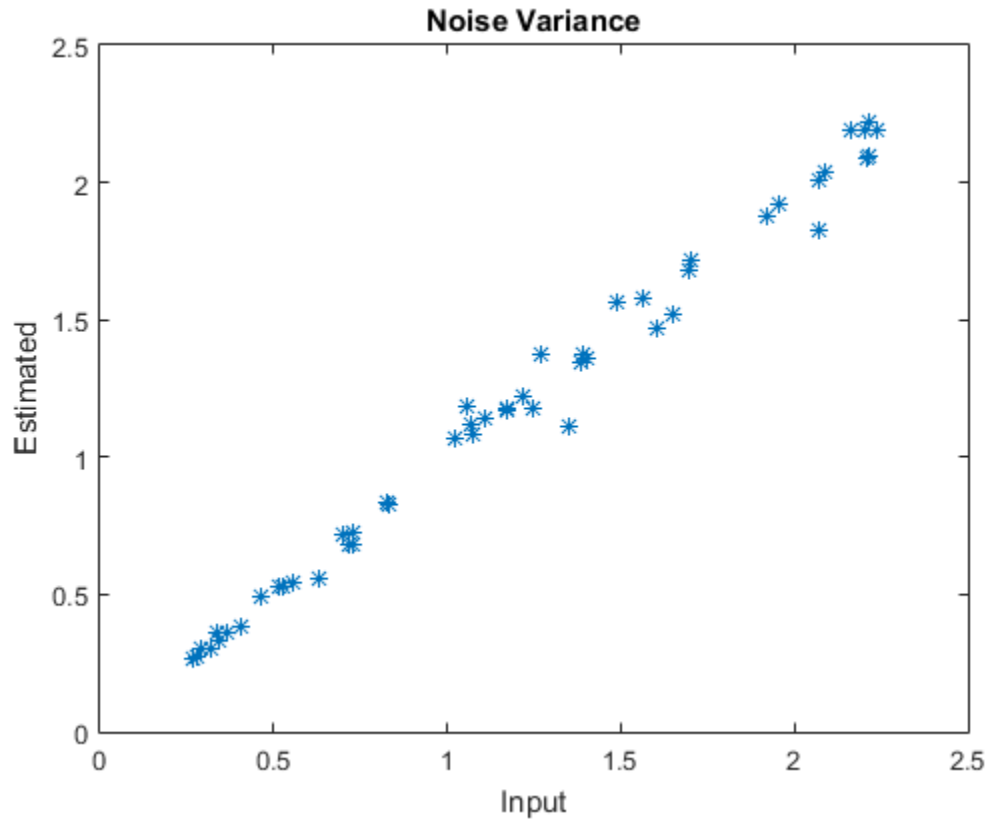
```
end
```

```
plot(noisestdz.^2,noisevar,'*')
```

```
title('Noise Variance')
```

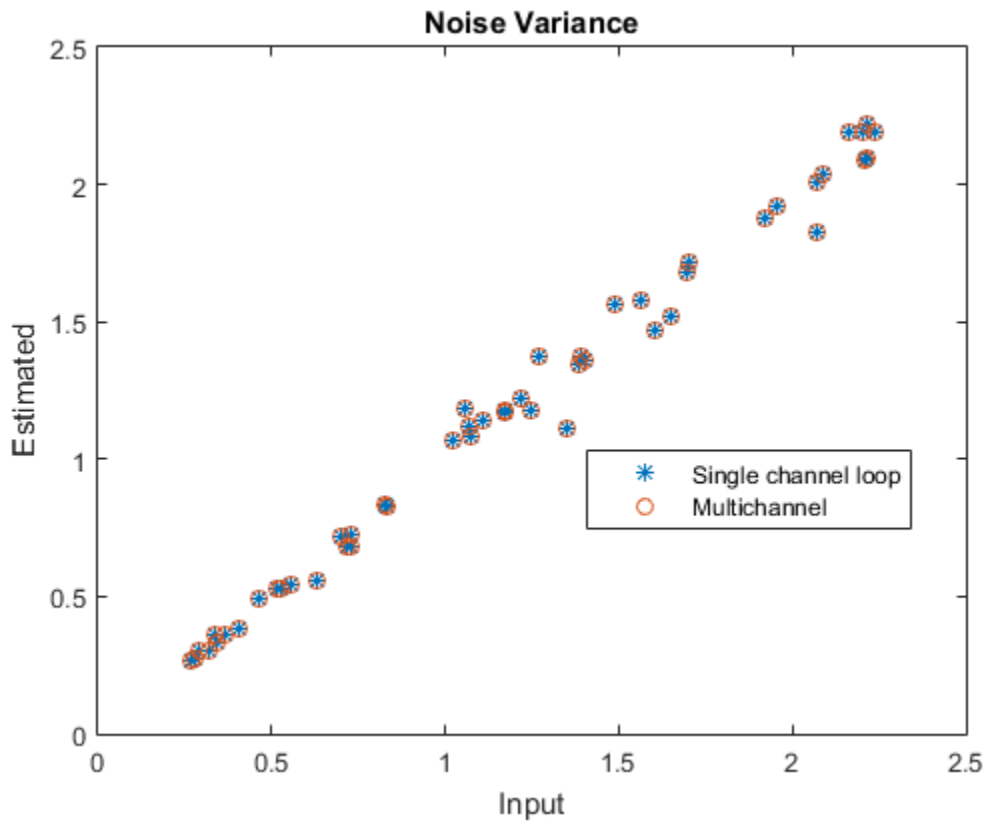
```
xlabel('Input')
```

```
ylabel('Estimated')
```



Repeat the procedure using `armcov`'s multichannel syntax.

```
realiz = bsxfun(@times,noisestdz,randnoise);
Y = filter(1,A,realiz);
[coeffs,variances] = armcov(Y,4);
hold on
plot(noisestdz.^2,variances,'o')
q = legend('Single channel loop','Multichannel');
q.Location = 'best';
```



## More About

### AR(p) Model

Let  $y(n)$  be a wide-sense stationary random process obtained by filtering a white noise input with variance  $e$  with the system function  $A(z)$ . If  $P_y(e^{j\omega})$  is the power spectral density of  $y(n)$ , then

$$P_y(e^{j\omega}) = \frac{e}{|A(e^{j\omega})|^2} = \frac{e}{\left|1 + \sum_{k=1}^p \alpha(k)e^{-j\omega k}\right|^2}.$$

Because the method characterizes the input data using an all-pole model, the correct choice of the model order,  $p$ , is important.

### See Also

arburg | arcov | aryule | lpc | pmcov | prony

# aryule

Autoregressive all-pole model parameters — Yule-Walker method

## Syntax

```
a = aryule(x,p)
[a,e] = aryule(x,p)
[a,e,rc] = aryule(x,p)
```

## Description

`a = aryule(x,p)` returns the normalized autoregressive (AR) parameters corresponding to a model of order `p` for the input array, `x`. If `x` is a vector, then the output array, `a`, is a row vector. If `x` is a matrix, then the parameters along the  $n$ th row of `a` model the  $n$ th column of `x`. `a` has `p + 1` columns. `p` must be less than the number of elements (or rows) of `x`.

`[a,e] = aryule(x,p)` returns the estimated variance, `e`, of the white noise input.

`[a,e,rc] = aryule(x,p)` returns the reflection coefficients in `rc`.

## Examples

### Parameter Estimation Using the Yule-Walker Method

Use a vector of polynomial coefficients to generate an AR(4) process by filtering 1024 samples of white noise. Reset the random number generator for reproducible results. Use the Yule-Walker method to estimate the coefficients.

```
rng default
A = [1 -2.7607 3.8106 -2.6535 0.9238];
y = filter(1,A,0.2*randn(1024,1));
```

```
arcoeffs = aryule(y,4)
```

```
arcoeffs =
```

```
    1.0000   -2.7262    3.7296   -2.5753    0.8927
```

Generate 50 realizations of the process, changing each time the variance of the input noise. Compare the Yule-Walker-estimated variances to the actual values.

```
nrealiz = 50;
```

```
noisestdz = rand(1,nrealiz)+0.5;
```

```
randnoise = randn(1024,nrealiz);
```

```
for k = 1:nrealiz
```

```
    y = filter(1,A,noisestdz(k) * randnoise(:,k));
```

```
    [arcoeffs,noisevar(k)] = aryule(y,4);
```

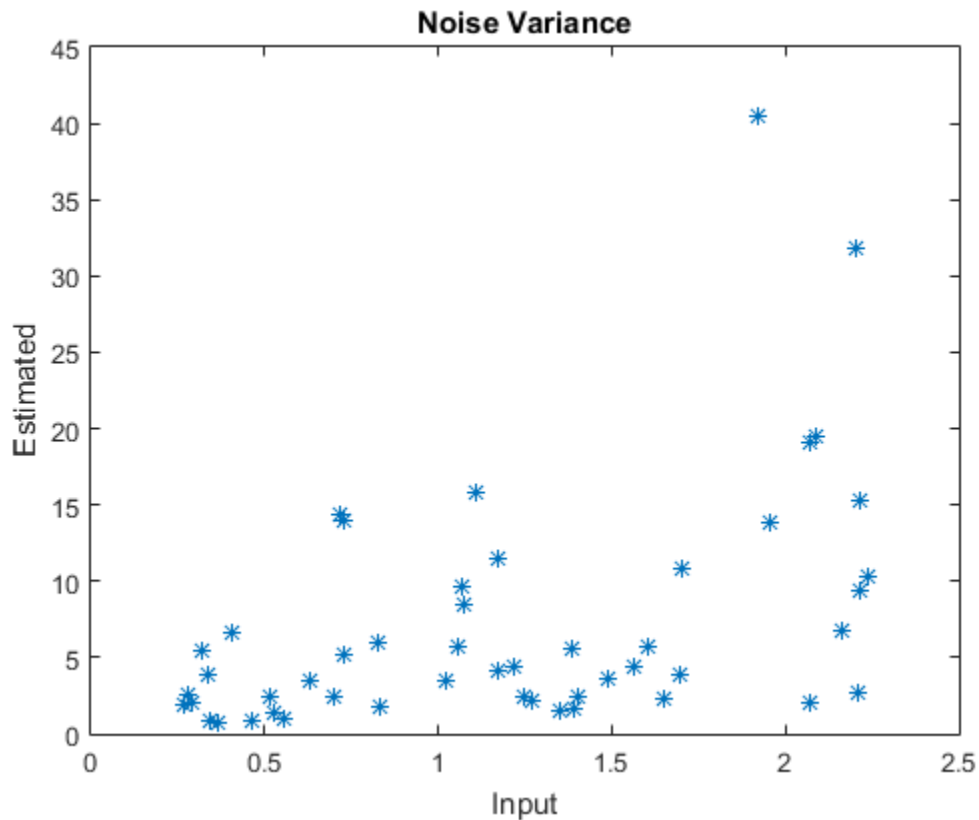
```
end
```

```
plot(noisestdz.^2,noisevar,'*')
```

```
title('Noise Variance')
```

```
xlabel('Input')
```

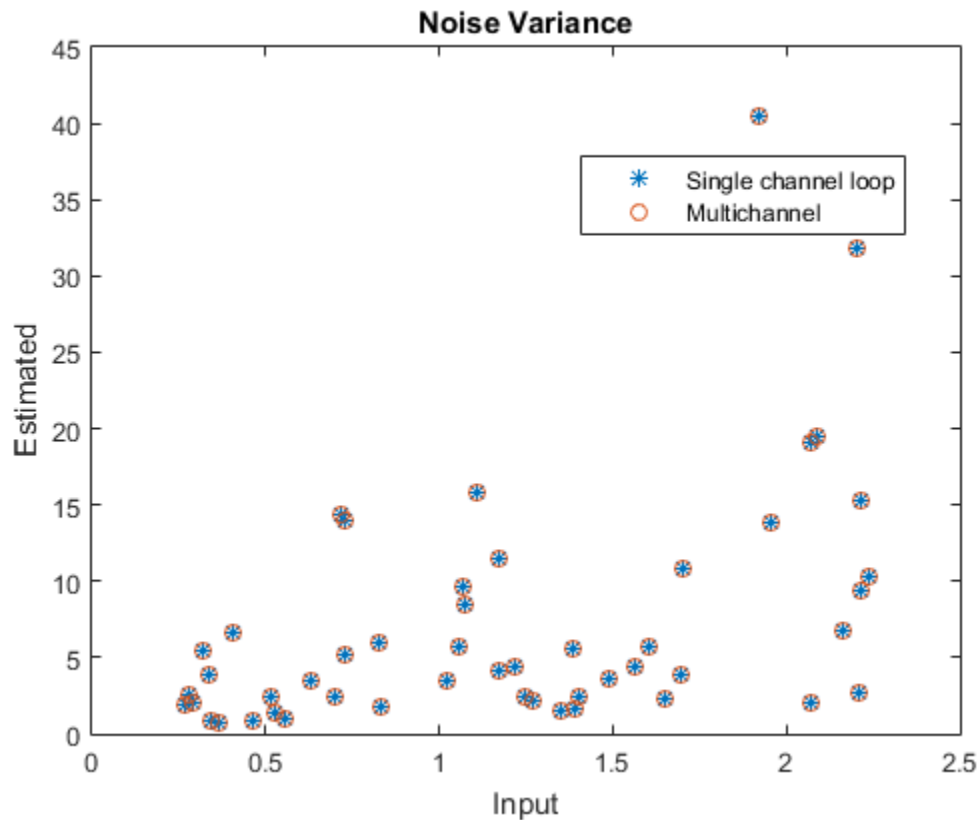
```
ylabel('Estimated')
```



Repeat the procedure using `aryule`'s multichannel syntax.

```
realiz = bsxfun(@times,noisestdz,randnoise);
Y = filter(1,A,realiz);
[coeffs,variances] = aryule(Y,4);
hold on
plot(noisestdz.^2,variances,'o')
q = legend('Single channel loop','Multichannel');
q.Location = 'best';
```





## More About

### AR(p) Model

In an AR model of order  $p$ , the current output is a linear combination of the past  $p$  outputs plus a white noise input. The weights on the  $p$  past outputs minimize the mean-square prediction error of the autoregression. If  $y(n)$  is the current value of the output and  $x(n)$  is a zero-mean white noise input, the AR( $p$ ) model is:

$$\sum_{k=0}^p a(k)y(n-k) = x(n).$$

### **Reflection Coefficients**

The reflection coefficients are the partial autocorrelation coefficients scaled by  $-1$ . The reflection coefficients indicate the time dependence between  $y(n)$  and  $y(n - k)$  after subtracting the prediction based on the intervening  $k - 1$  time steps.

### **Algorithms**

`aryule` uses the Levinson-Durbin recursion on the biased estimate of the sample autocorrelation sequence to compute the parameters.

- “Parametric Modeling”

### **References**

[1] Hayes, Monson H. *Statistical Digital Signal Processing and Modeling*. New York: John Wiley & Sons, 1996.

### **See Also**

`arburg` | `arcov` | `armcov` | `levinson` | `lpc`

# bandpower

Band power

## Syntax

```
p = bandpower(x)
p = bandpower(x,fs,freqrange)

p = bandpower(pxx,f,'psd')
p = bandpower(pxx,f,freqrange,'psd')
```

## Description

`p = bandpower(x)` returns the average power in the input signal, `x`. If `x` is a matrix, then `bandpower` computes the average power in each column independently.

`p = bandpower(x,fs,freqrange)` returns the average power in the frequency range, `freqrange`, specified as a two-element vector. You must input the sampling frequency, `fs`, to return the power in a specified frequency range. `bandpower` uses a modified periodogram to determine the average power in `freqrange`.

`p = bandpower(pxx,f,'psd')` returns the average power computed by integrating the power spectral density (PSD) estimate, `pxx`. The integral is approximated by the rectangle method. The input, `f`, is a vector of frequencies corresponding to the PSD estimates in `pxx`. The string `'psd'` indicates the input is a PSD estimate and not time series data.

`p = bandpower(pxx,f,freqrange,'psd')` returns the average power contained in the frequency interval, `freqrange`. If the frequencies in `freqrange` do not match values in `f`, the closest values are used. The average power is computed by integrating the power spectral density (PSD) estimate, `pxx`. The integral is approximated by the rectangle method. The string `'psd'` indicates the input is a PSD estimate and not time series data.

## Examples

### Comparison with Euclidean Norm

Create a signal consisting of a 100 Hz sine wave in additive  $N(0,1)$  white Gaussian noise. The sampling frequency is 1 kHz. Determine the average power and compare it against the  $l_2$  norm.

```
t = 0:0.001:1-0.001;  
x = cos(2*pi*100*t)+randn(size(t));
```

```
p = bandpower(x)  
l2norm = norm(x,2)^2/numel(x)
```

```
p =  
  
    1.5264
```

```
l2norm =  
  
    1.5264
```

### Percentage of Total Power in Frequency Interval

Determine the percentage of the total power in a specified frequency interval.

Create a signal consisting of a 100 Hz sine wave in additive  $N(0,1)$  white Gaussian noise. The sampling frequency is 1 kHz. Determine the percentage of the total power in the frequency interval between 50 Hz and 150 Hz.

```
t = 0:0.001:1-0.001;  
x = cos(2*pi*100*t)+randn(size(t));
```

```
pband = bandpower(x,1000,[50 100]);  
ptot = bandpower(x,1000,[0 500]);  
per_power = 100*(pband/ptot)
```

```
per_power =
```

40.0243

### Periodogram Input

Determine the average power by first computing a PSD estimate using the periodogram. Input the PSD estimate to `bandpower`.

Create a signal consisting of a 100 Hz sine wave in additive  $N(0,1)$  white Gaussian noise. The sampling frequency is 1 kHz. Obtain the periodogram and use the 'psd' flag to compute the average power using the PSD estimate. Compare the result against the average power computed in the time domain.

```
t = 0:0.001:1-0.001;
Fs = 1000;
x = cos(2*pi*100*t)+randn(size(t));

[Pxx,F] = periodogram(x,rectwin(length(x)),length(x),Fs);
p = bandpower(Pxx,F,'psd')
avpow = norm(x,2)^2/numel(x)
```

```
p =
    1.5264
```

```
avpow =
    1.5264
```

### Percentage of Power in Frequency Band (Periodogram)

Determine the percentage of the total power in a specified frequency interval using the periodogram as the input.

Create a signal consisting of a 100 Hz sine wave in additive  $N(0,1)$  white Gaussian noise. The sampling frequency is 1 kHz. Obtain the periodogram and corresponding frequency vector. Using the PSD estimate, determine the percentage of the total power in the frequency interval between 50 Hz and 150 Hz.

```
Fs = 1000;
```

```
t = 0:1/Fs:1-0.001;
x = cos(2*pi*100*t)+randn(size(t));

[Pxx,F] = periodogram(x,rectwin(length(x)),length(x),Fs);
pband = bandpower(Pxx,F,[50 100],'psd');
ptot = bandpower(Pxx,F,'psd');
per_power = 100*(pband/ptot)
```

```
per_power =
    42.0767
```

### Average Power of a Multichannel Signal

Create a multichannel signal consisting of three sinusoids in additive  $N(0,1)$  white Gaussian noise. The sinusoids' frequencies are 100 Hz, 200 Hz, and 300 Hz. The sampling frequency is 1 kHz, and the signal has a duration of 1 s.

```
Fs = 1000;
t = 0:1/Fs:1-1/Fs;
f = [100;200;300];
x = cos(2*pi*f*t)'+randn(length(t),3);
```

Determine the average power of the signal and compare it to the  $\ell_2$  norm.

```
p = bandpower(x)
    1.5264    1.5382    1.4717

l2norm = dot(x,x)/length(x)

p =
    1.5264    1.5382    1.4717

l2norm =
    1.5264    1.5382    1.4717
```

## Input Arguments

### **x** — Time series input

vector | matrix

Input time series data, specified as a row or column vector or as a matrix. If **x** is a matrix, then its columns are treated as independent channels.

Example: `cos(pi/4*(0:159))'+randn(160,1)` is a single-channel column-vector signal.

Example: `cos(pi./[4;2]*(0:159))'+randn(160,2)` is a two-channel noisy sinusoid.

Data Types: `double` | `single`

Complex Number Support: Yes

### **fs** — Sampling frequency

1 (default) | positive scalar

Sampling frequency for the input time series data, specified as a positive scalar.

Data Types: `double` | `single`

### **freqrange** — Frequency range for band power computation

two-element real-valued row or column vector

Frequency range for the band power computation, specified as a two-element real-valued row or column vector. If the input signal, **x**, contains **N** samples, **freqrange** must be within the following intervals.

- $[0, fs/2]$  if **x** is real-valued and **N** is even
- $[0, (N-1)fs/(2N)]$  if **x** is real-valued and **N** is odd
- $[-(N-2)fs/(2N), fs/2]$  if **x** is complex-valued and **N** is even
- $[-(N-1)fs/(2N), (N-1)fs/(2N)]$  if **x** is complex-valued and **N** is odd

Data Types: `double` | `single`

### **pxx** — PSD estimates

real-valued column vector with nonnegative elements

One- or two-sided PSD estimate, specified as a column vector with nonnegative elements.

Data Types: `double` | `single`

**f** — **Frequency vector for PSD estimates**

column vector with real-valued elements

Frequency vector, specified as a column vector. The frequency vector, `f`, contains the frequencies corresponding to the PSD estimates in `pxx`.

Data Types: `double` | `single`

## Output Arguments

**p** — **Average band power**

nonnegative scalar

Average band power, returned as a nonnegative scalar.

Data Types: `double` | `single`

## See Also

`periodogram` | `sfd`



# barthannwin

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.

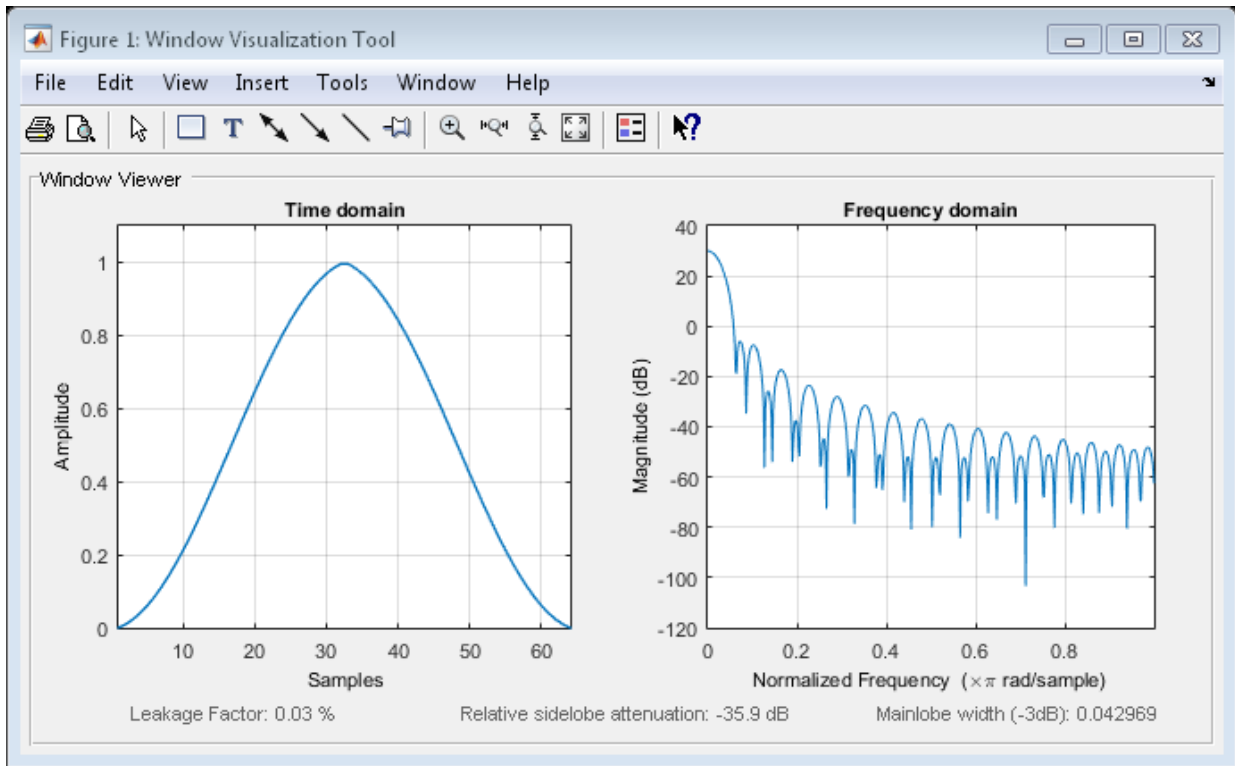
---

## Examples

### Bartlett-Hann Window

Create a 64-point Bartlett-Hann window. Display the result using `wvtool`.

```
L = 64;  
wvtool(barthannwin(L))
```



## More About

### Algorithms

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  $0 \leq n \leq N$  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<sup>®</sup> Transactions on Acoustics, Speech, and Signal Processing*. Vol. 37, Number 2, 1999, pp. 298–301.
- [2] Oppenheim, Alan V., Ronald W. Schafer, and John R. Buck. *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

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:

$$w(n) = \begin{cases} \frac{2n}{N}, & 0 \leq n \leq \frac{N}{2} \\ 2 - \frac{2n}{N}, & \frac{N}{2} \leq n \leq N \end{cases}$$

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 has zeros at the first and last samples, 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.

---

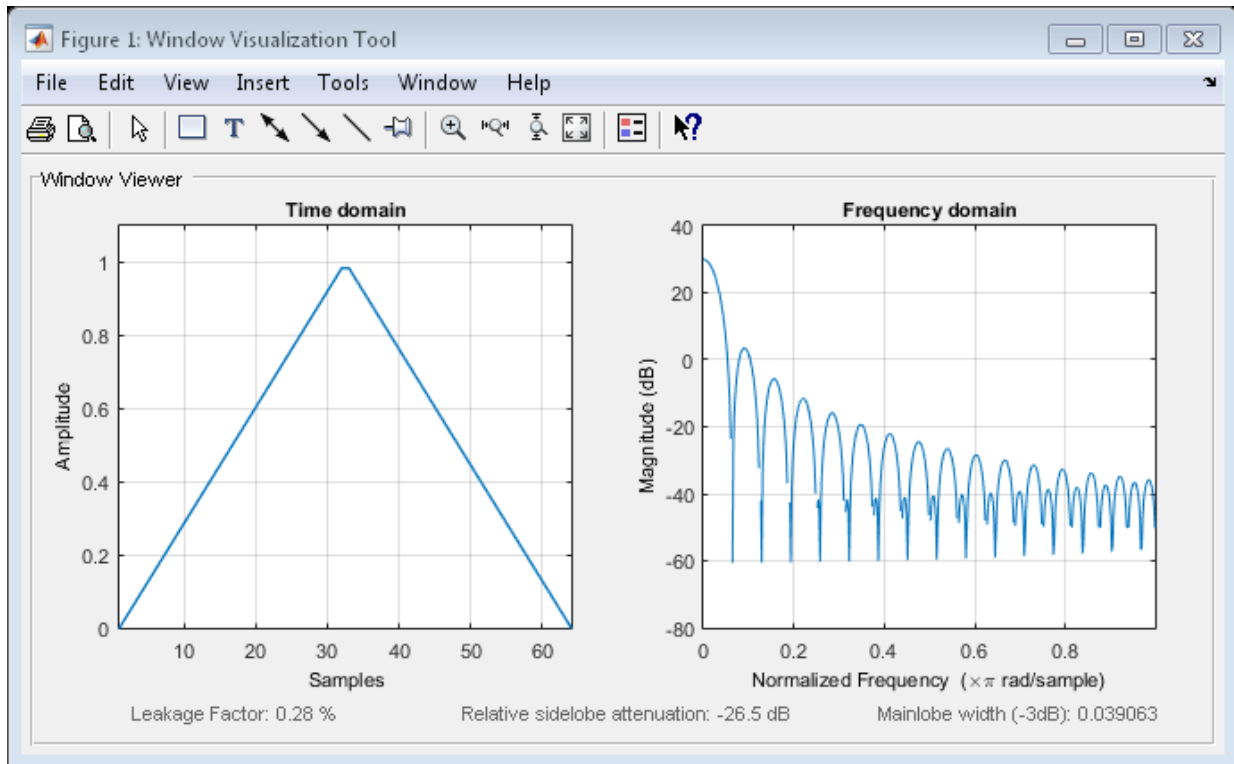
## Examples

### Bartlett Window

Create a 64-point Bartlett window. Display the result using `wvtool`.

```
L = 64;
```

```
bw = bartlett(L);
wvtool(bw)
```



## References

- [1] Oppenheim, Alan V., Ronald W. Schaffer, and John R. Buck. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1999.

## See Also

barthannwin | blackmanharris | bohmanwin | nuttallwin | parzenwin |  
rectwin | triang | window | wintool | wvtool

## besselap

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  $1/\sqrt{2}$  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

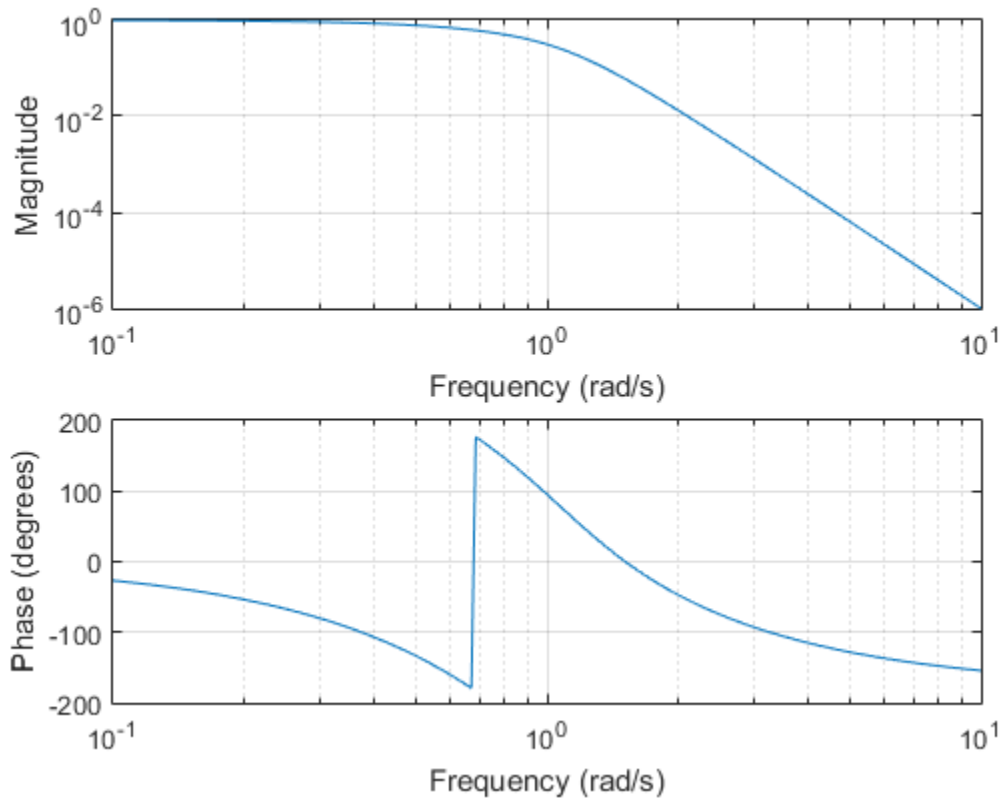
$$\left( \frac{(2n)!}{2^n n!} \right)^{1/n}$$

### Examples

#### Frequency Response of an Analog Bessel Filter

Design a 6th-order Bessel analog lowpass filter. Display its magnitude and phase responses.

```
[z,p,k] = besselap(6);      % Lowpass filter prototype
[num,den] = zp2tf(z,p,k);  % Convert to transfer function form
freqs(num,den)             % Frequency response of analog filter
```



## More About

### Algorithms

`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, pp.228–230.

## See Also

besself | buttap | cheb1ap | cheb2ap | ellipap



# besself

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:

$$H(s) = \frac{B(s)}{A(s)} = \frac{b(1)s^n + b(2)s^{n-1} + \dots + b(n+1)}{a(1)s^n + a(2)s^{n-1} + \dots + a(n+1)}$$

`[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

$$\begin{aligned}\dot{x} &= Ax + Bu \\ y &= Cx + Du.\end{aligned}$$

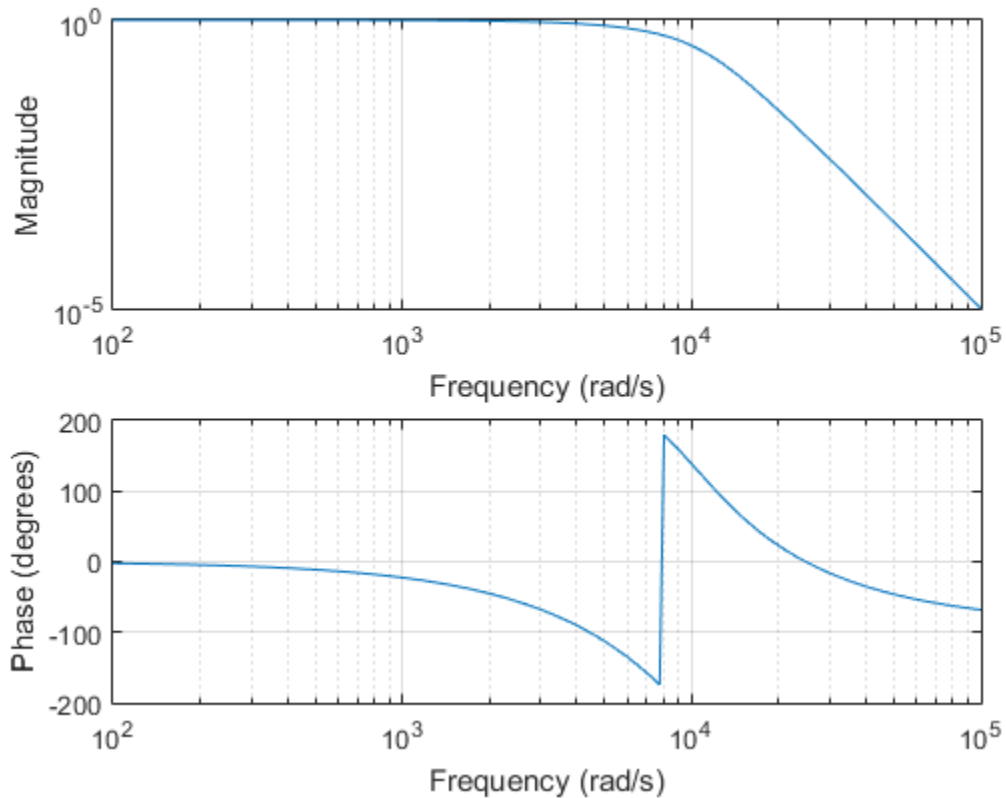
and  $u$  is the input,  $x$  is the state vector, and  $y$  is the output.

## Examples

### Frequency Response of an Analog Bessel Filter

Design a 5th-order analog lowpass Bessel filter with approximately constant group delay up to  $10^4$  rad/s. Plot the magnitude and phase responses of the filter using `freqs`.

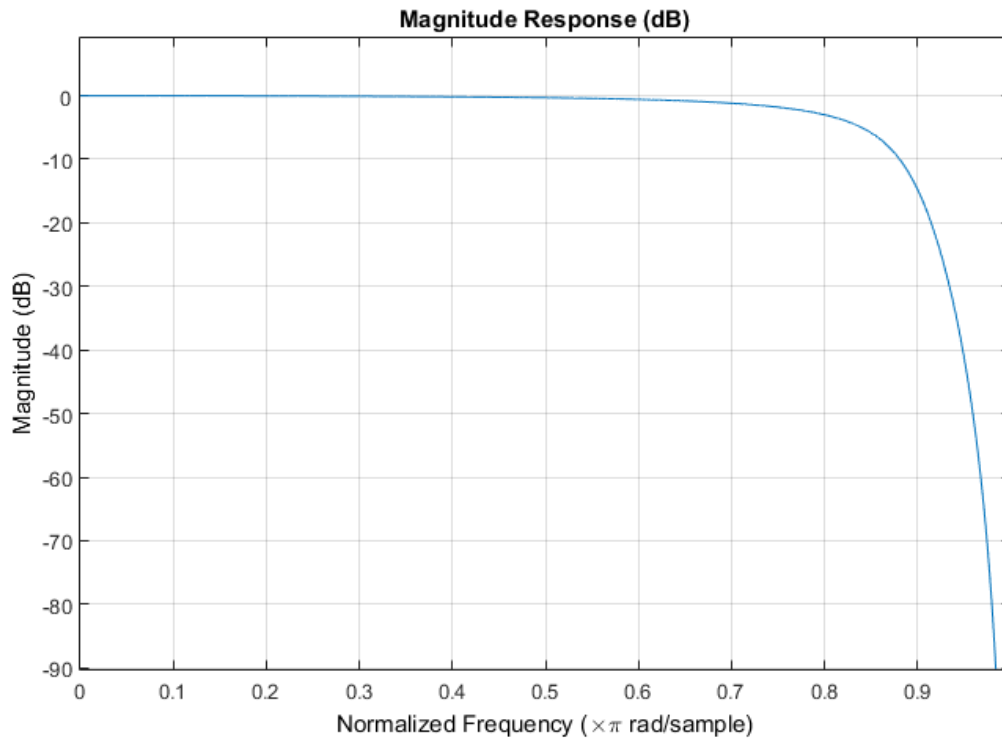
```
[b,a] = besself(5,10000);  
freqs(b,a)
```



### Frequency Response of a Digital Bessel Filter

Design an analog Bessel filter of order 5. Convert it to a digital IIR filter using `bilinear`. Display its frequency response.

```
Fs = 100; % Sampling Frequency
[z,p,k] = besself(5,1000); % Bessel analog filter design
[zd,pd,kd] = bilinear(z,p,k,Fs); % Analog to digital mapping
sos = zp2sos(zd,pd,kd); % Convert to SOS form
fvtool(sos) % Visualize the digital filter
```



## 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.

## More About

### Algorithms

`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.

## References

- [1] Parks, T. W., and C. S. Burrus. *Digital Filter Design*. New York: John Wiley & Sons, 1987, chap.7.

### See Also

`besselap` | `butter` | `cheby1` | `cheby2` | `ellip`

# bilinear

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  $\pi$ ) 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

$$H(z) = H(s) \Big|_{s = \frac{2\pi f_p}{\tan\left(\pi \frac{f_p}{f_z}\right)} \frac{(z-1)}{(z+1)}}$$

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  $\pi$ ) 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  $s$ . Let  $B(s)$  be the numerator polynomial and  $A(s)$  be the denominator polynomial. The transfer function is:

$$\frac{B(s)}{A(s)} = \frac{B(1)s^n + \dots + B(n)s + B(n+1)}{A(1)s^m + \dots + A(m)s + A(m+1)}$$

`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`

$$\begin{aligned}\dot{x} &= Ax + Bu \\ y &= Cx + Du\end{aligned}$$

to the discrete-time system:

$$\begin{aligned}x[n+1] &= A_d x[n] + B_d u[n] \\ y[n] &= C_d x[n] + D_d u[n]\end{aligned}$$

`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.

## Examples

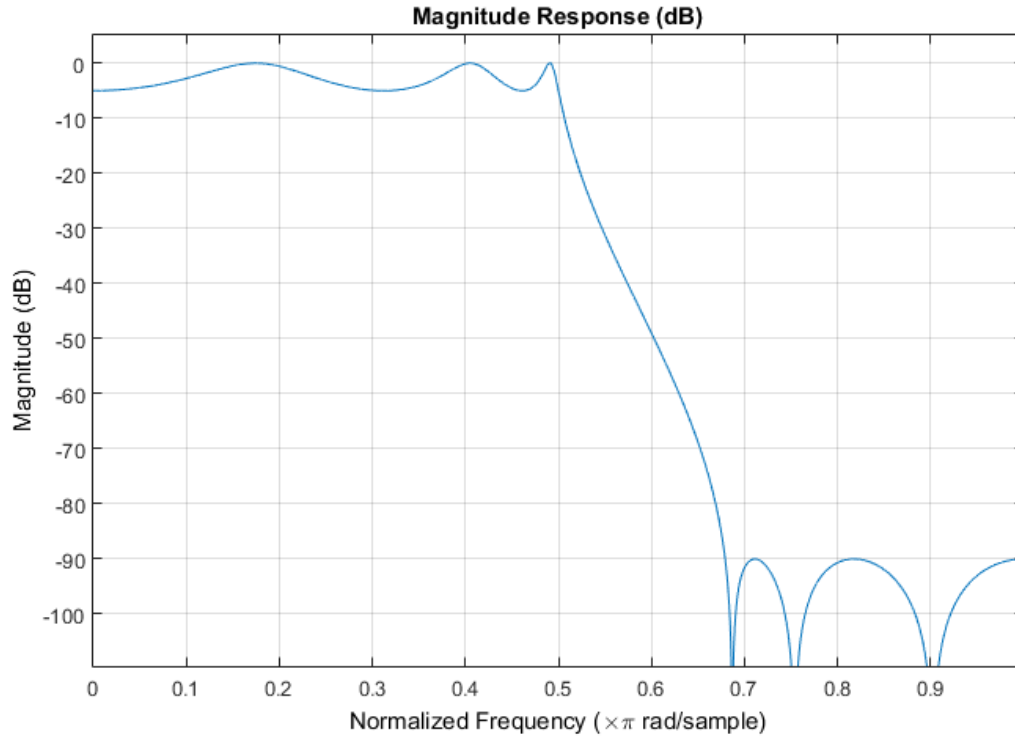
### Discrete-Time Representation of an Elliptic Filter

Design a 6th-order elliptic analog lowpass filter with 5 dB of ripple in the passband and a stopband 90 dB down. Use `bilinear` to transform it to a discrete-time IIR filter.

```
Fs = 0.5; % Sampling frequency
[z,p,k] = ellipap(6,5,90); % Lowpass filter prototype
[num,den] = zp2tf(z,p,k); % Convert to transfer function form
[numd,dend] = bilinear(num,den,Fs); % Analog to digital conversion
```

```
fvtool(numd,dend)
```

```
% Visualize the filter
```



## 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.
```



## More About

### Algorithms

`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, let

$$\lambda = \frac{\pi f_p}{\tan(\pi f_p / f_s)}$$

If  $f_p$  is not present, let  $\lambda = fs$ .

- 2** Compute  $A_d$ ,  $B_d$ ,  $C_d$ , and  $D_d$  in terms of  $A$ ,  $B$ ,  $C$ , and  $D$  using

$$A_d = (I - A \frac{1}{2\lambda})^{-1} (I + A \frac{1}{2\lambda})$$

$$B_d = \frac{1}{\sqrt{\lambda}} (I - A \frac{1}{2\lambda})^{-1} B$$

$$C_d = \frac{1}{\sqrt{\lambda}} C (I - A \frac{1}{2\lambda})^{-1}$$

$$D_d = \frac{1}{2\lambda} C (I - A \frac{1}{2\lambda})^{-1} B + D$$

## References

- [1] Parks, Thomas W., and C. Sidney Burrus. *Digital Filter Design*. New York: John Wiley & Sons, 1987, pp.209–213.
- [2] Oppenheim, Alan V., Ronald W. Schaffer, and John R. Buck. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1999, pp.450–454.

## See Also

[impinvar](#) | [lp2bp](#) | [lp2bs](#) | [lp2hp](#) | [lp2lp](#)

# bitrevorder

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                  |

| Linear Index | Bits | Bit- Reversed | Bit-Reversed Index |
|--------------|------|---------------|--------------------|
| 4            | 100  | 001           | 1                  |
| 5            | 101  | 101           | 5                  |
| 6            | 110  | 011           | 3                  |
| 7            | 111  | 111           | 7                  |

## Examples

### Vector in Bit-Reversed Order

Create a column vector and obtain its bit-reversed version. Verify by displaying the elements as binary strings.

```
x = (0:15)';
v = bitrevorder(x);

x_bin = dec2bin(x);
v_bin = dec2bin(v);

T = table(x,x_bin,v,v_bin)
```

T =

| x  | x_bin | v  | v_bin |
|----|-------|----|-------|
| 0  | 0000  | 0  | 0000  |
| 1  | 0001  | 8  | 1000  |
| 2  | 0010  | 4  | 0100  |
| 3  | 0011  | 12 | 1100  |
| 4  | 0100  | 2  | 0010  |
| 5  | 0101  | 10 | 1010  |
| 6  | 0110  | 6  | 0110  |
| 7  | 0111  | 14 | 1110  |
| 8  | 1000  | 1  | 0001  |
| 9  | 1001  | 9  | 1001  |
| 10 | 1010  | 5  | 0101  |
| 11 | 1011  | 13 | 1101  |
| 12 | 1100  | 3  | 0011  |

|    |      |    |      |
|----|------|----|------|
| 13 | 1101 | 11 | 1011 |
| 14 | 1110 | 7  | 0111 |
| 15 | 1111 | 15 | 1111 |

**See Also**

[fft](#) | [digitrevorder](#) | [ifft](#)

# blackman

Blackman window

## Syntax

```
w = blackman(N)
w = blackman(N,SFLAG)
```

## Description

`w = blackman(N)` returns the N-point symmetric Blackman window in the column vector `w`, where `N` is a positive integer.

`w = blackman(N,SFLAG)` returns an N-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 `N+1` window and returns the first `N` points. When using windows for filter design, the 'symmetric' flag should be used.

See “Definitions” on page 1-57 for a description of the difference between the symmetric and periodic windows.

---

**Note** If you specify a one-point window (set `N = 1`), the value 1 is returned.

---

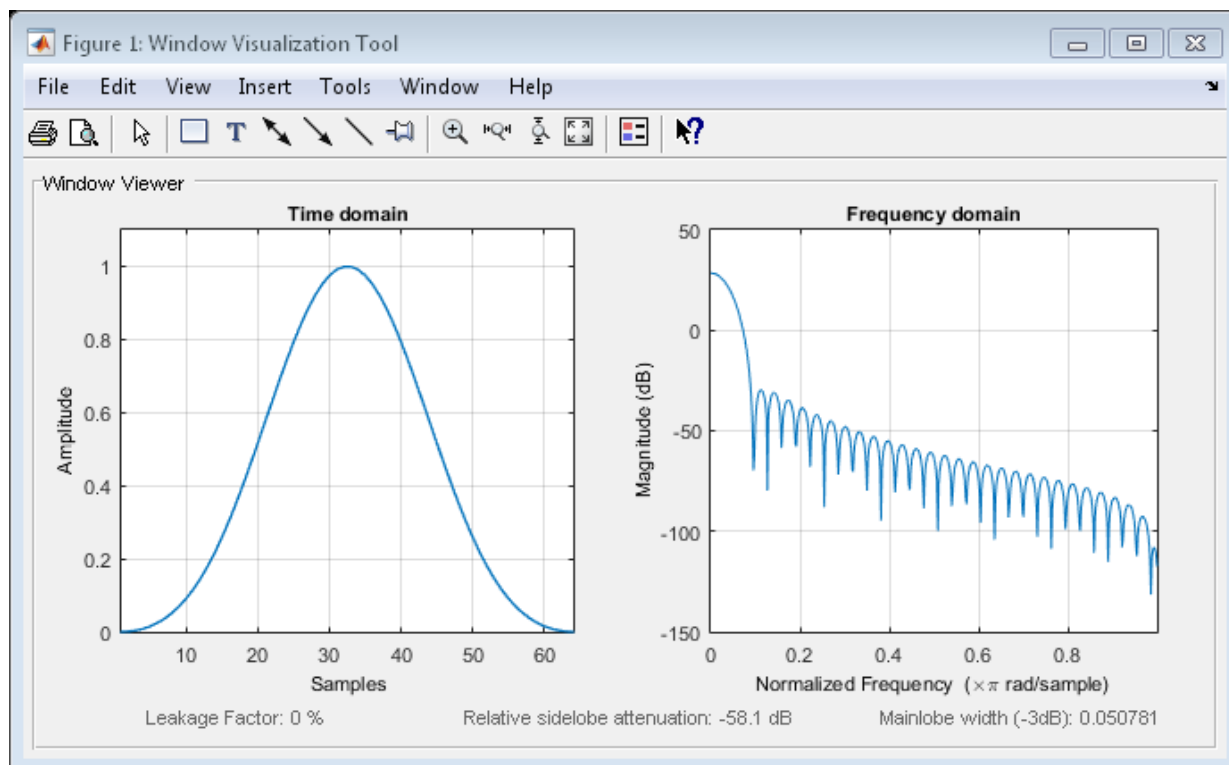
## Examples

### Blackman Window

Create a 64-point Blackman window. Display the result using `wvtool`.

```
L = 64;
```

```
wvtool(blackman(L))
```



## Definitions

The following equation defines the Blackman window of length  $N$ :

$$w(n) = 0.42 - 0.5 \cos \frac{2\pi n}{N-1} + 0.08 \cos \frac{4\pi n}{N-1}, \quad 0 \leq n \leq M-1$$

where  $M$  is  $N/2$  for  $N$  even and  $(N+1)/2$  for  $N$  odd.

In the *symmetric* case, the second half of the Blackman window  $M \leq n \leq N-1$  is obtained by flipping the first half around the midpoint. The symmetric option is the preferred method when using a Blackman window in FIR filter design.

The *periodic* Blackman window is constructed by extending the desired window length by one sample to  $N + 1$ , constructing a symmetric window, and removing the last sample. The periodic version is the preferred method when using a Blackman window in spectral analysis because the discrete Fourier transform assumes periodic extension of the input vector.

## References

- [1] Oppenheim, Alan V., Ronald W. Schaffer, and John R. Buck. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1999, pp. 468–471.

## See Also

flattopwin | hann | window | hamming | wintool | wvtool



# blackmanharris

Minimum 4-term Blackman-Harris window

## Syntax

```
w = blackmanharris(N)
w = blackmanharris(N,SFLAG)
```

## Description

`w = blackmanharris(N)` returns an  $N$ -point symmetric 4-term Blackman-Harris window in the column vector  $w$ . The window is minimum in the sense that its maximum sidelobes are minimized.

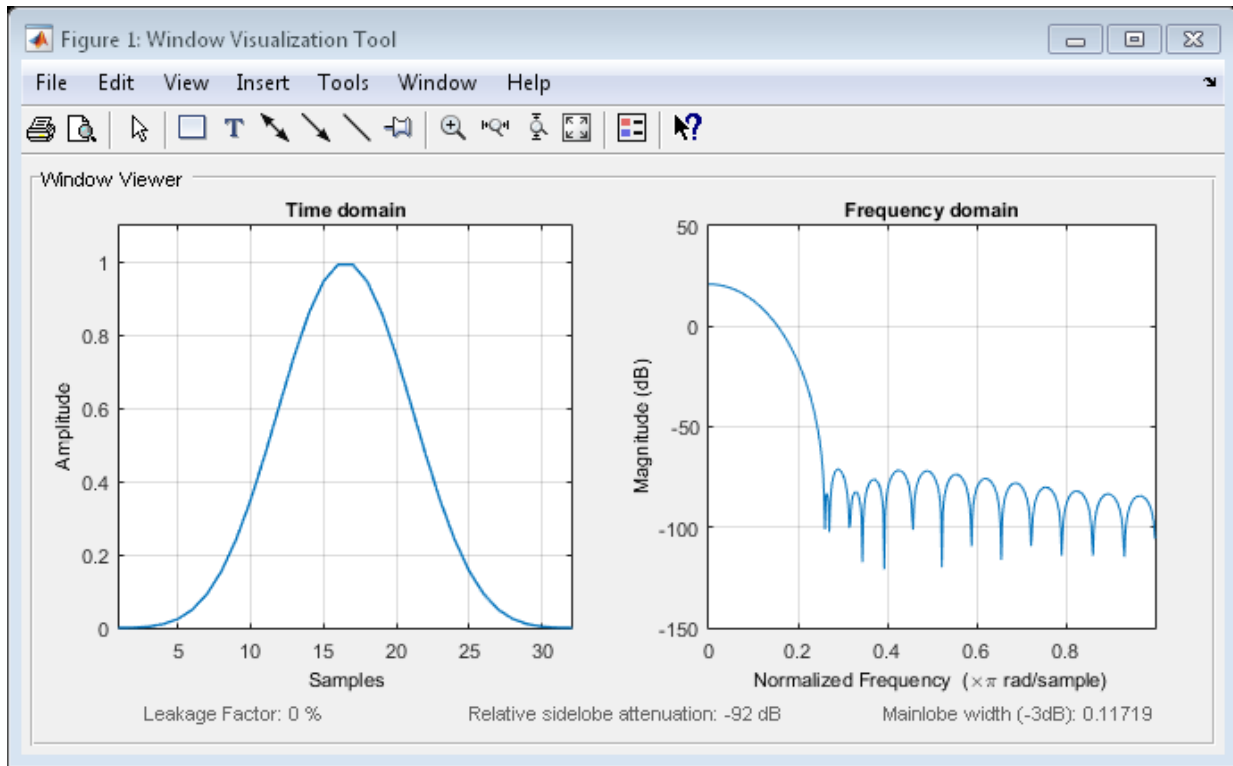
`w = blackmanharris(N,SFLAG)` uses `SFLAG` window sampling. `SFLAG` can be 'symmetric' or 'periodic'. The default is 'symmetric'. You can find the equations defining the symmetric and periodic windows in “Definitions” on page 1-60.

## Examples

### Blackman-Harris Window

Create a 32-point symmetric Blackman-Harris window. Display the result using `wvtool`.

```
N = 32;
wvtool(blackmanharris(N))
```



## Definitions

The equation for the **symmetric** 4-term Blackman-harris window of length  $N$  is

$$w(n) = a_0 - a_1 \cos\left(\frac{2\pi n}{N-1}\right) + a_2 \cos\left(\frac{4\pi n}{N-1}\right) - a_3 \cos\left(\frac{6\pi n}{N-1}\right), \quad 0 \leq n \leq N-1$$

The equation for the **periodic** 4-term Blackman-harris window of length  $N$  is

$$w(n) = a_0 - a_1 \cos\frac{2\pi n}{N} + a_2 \cos\frac{4\pi n}{N} - a_3 \cos\frac{6\pi n}{N}, \quad 0 \leq n \leq N-1$$

The periodic window is  $N$ -periodic.

The following table lists the coefficients:

| Coefficient | Value   |
|-------------|---------|
| $a_0$       | 0.35875 |
| $a_1$       | 0.48829 |
| $a_2$       | 0.14128 |
| $a_3$       | 0.01168 |

## References

- [1] Harris, Fredric J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform." *Proceedings of the IEEE*. Vol. 66, January 1978, pp. 51–83.

## See Also

barthannwin | bartlett | bohmanwin | nuttallwin | parzenwin | rectwin |  
triang | window | wintool | wvtool

# bohmanwin

Bohman window

## Syntax

```
w = bohmanwin(L)
```

## Description

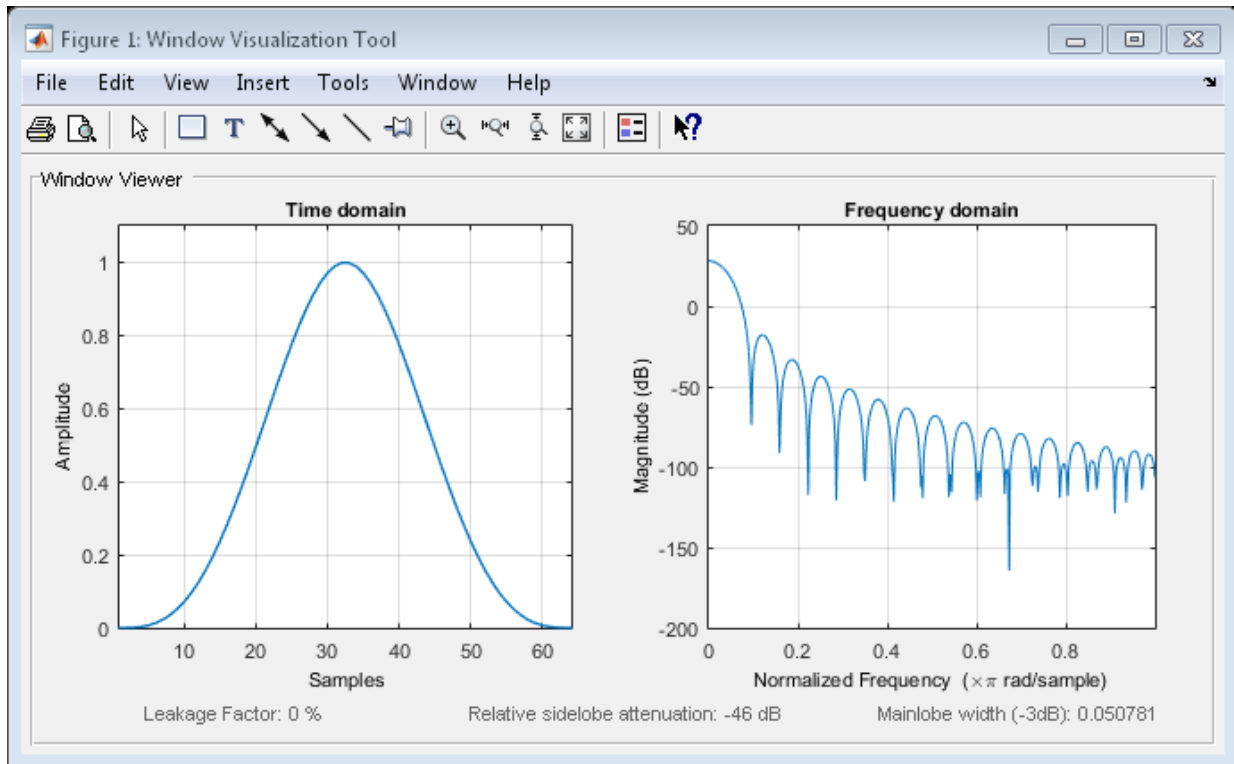
`w = bohmanwin(L)` returns an  $L$ -point Bohman window in column vector `w`. A Bohman window is the convolution of two half-duration cosine lobes. In the time domain, it is the product of a triangular window and a single cycle of a cosine with a term added to set the first derivative to zero at the boundary. Bohman windows fall off as  $1/w^4$ .

## Examples

### Bohman Window

Compute a 64-point Bohman window. Display the result using `wvtool`.

```
L = 64;  
bw = bohmanwin(L);  
wvtool(bw)
```



## More About

### Algorithms

The equation for computing the coefficients of a Bohman window is

$$w(x) = (1 - |x|) \cos(\pi |x|) + \frac{1}{\pi} \sin(\pi |x|), \quad -1 \leq x \leq 1$$

where  $x$  is a length- $L$  vector of linearly spaced values generated using `linspace`. The first and last elements of the Bohman window are forced to be identically zero.

## References

- [1] Harris, Fredric J. “On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform.” *Proceedings of the IEEE*. Vol. 66, January 1978, pp. 51–83.

## See Also

barthannwin | bartlett | blackmanharris | nuttallwin | parzenwin | rectwin  
| triang | window | wintool | wvtool

# buffer

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  $\text{ceil}(L/n)$  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.

`y =`

|   |   |    |    |    |    |    |    |
|---|---|----|----|----|----|----|----|
| 0 | 2 | 6  | 10 | 14 | 18 | 22 | 26 |
| 0 | 3 | 7  | 11 | 15 | 19 | 23 | 27 |
| 0 | 4 | 8  | 12 | 16 | 20 | 24 | 28 |
| 1 | 5 | 9  | 13 | 17 | 21 | 25 | 29 |
| 2 | 6 | 10 | 14 | 18 | 22 | 26 | 30 |
| 3 | 7 | 11 | 15 | 19 | 23 | 27 | 0  |
| 4 | 8 | 12 | 16 | 20 | 24 | 28 | 0  |

The first frame starts with  $p$  zeros (the default initial condition), and the number of columns in  $y$  is  $\text{ceil}(L/(n-p))$ .

- 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

```

skipped { 8 18 28 }  
           { 9 19 29 }  
           { 10 20 30 }

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 1-68 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 1-68 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 1-68 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 is the input argument to `buffer`, and `opt` on the left 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 1-68 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.

| Iteration | Input frame [z;x]' | opt (input) | opt (output) | Output buffer (y)  | Overflow (z) |
|-----------|--------------------|-------------|--------------|--|--------------|
| i=1       | [1:11]             | -5          | 9            | $\begin{bmatrix} -5 & 3 & 6 \\ 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$                                | [10 11]      |
| i=2       | [10 11 12:22]      | 9           | 21           | $\begin{bmatrix} 9 & 12 & 15 & 18 \\ 10 & 13 & 16 & 19 \\ 11 & 14 & 17 & 20 \\ 12 & 15 & 18 & 21 \end{bmatrix}$  | [22]         |
| i=3       | [22 23:33]         | 21          | 33           | $\begin{bmatrix} 21 & 24 & 27 & 30 \\ 22 & 25 & 28 & 31 \\ 23 & 26 & 29 & 32 \\ 24 & 27 & 30 & 33 \end{bmatrix}$ | []           |
| i=4       | [34:44]            | 33          | 42           | $\begin{bmatrix} 33 & 36 & 39 \\ 34 & 37 & 40 \\ 35 & 38 & 41 \\ 36 & 39 & 42 \end{bmatrix}$                     | [43 44]      |

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.

| Iteration | Input frame [z;x]' | opt (input) | opt (output) | Output buffer (y) | Overflow (z) |
|-----------|--------------------|-------------|--------------|-------------------|--------------|
| i=1       | [1:11]             | 1           | 2            |                   | []           |
| i=2       | [12:22]            | 2           | 0            |                   | [20 21 22]   |
| i=3       | [20 21 22 23:33]   | 0           | 0            |                   | [32 33]      |

## Diagnostics

Error messages are displayed when  $p \geq n$  or  $\text{length}(\text{opt}) \neq \text{length}(p)$  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



# buttap

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  $2n-1$  derivatives of the squared magnitude response are zero at  $\omega = 0$ . The squared magnitude response function is

$$|H(\omega)|^2 = \frac{1}{1 + (\omega / \omega_0)^{2n}}$$

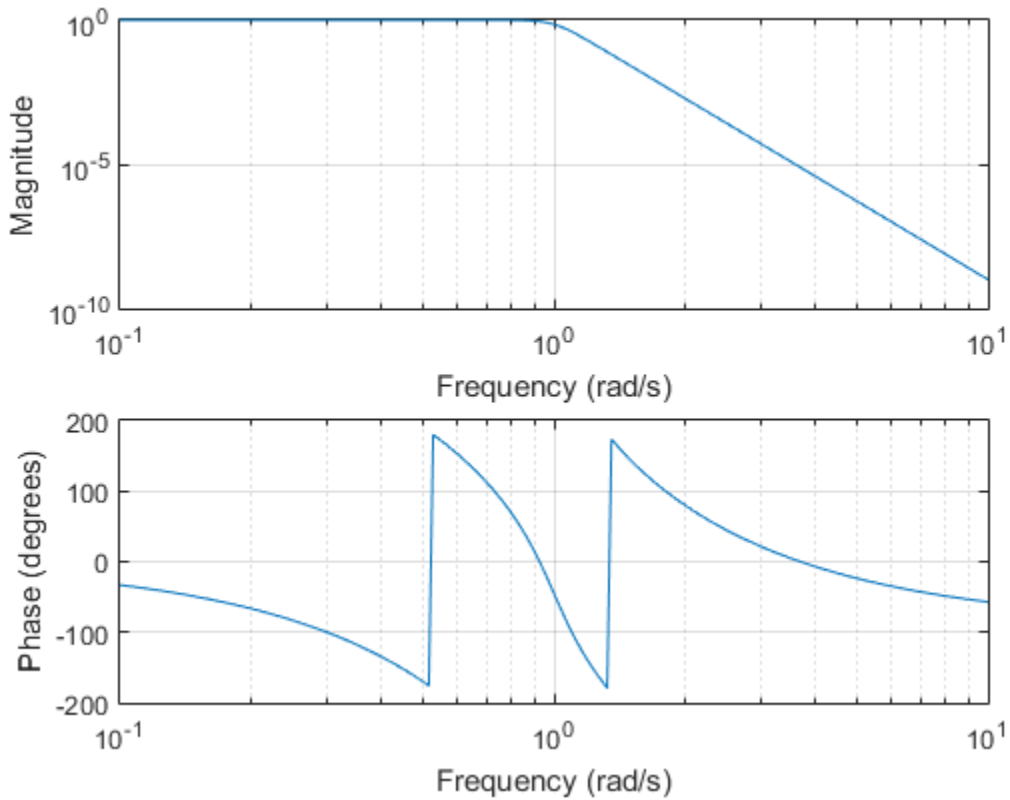
corresponding to a transfer function with poles equally spaced around a circle in the left half plane. The magnitude response at the cutoff angular frequency  $\omega_0$  is always  $1/\sqrt{2}$  regardless of the filter order. `buttap` sets  $\omega_0$  to 1 for a normalized result.

## Examples

### Frequency Response of a Butterworth Analog Filter

Design a 9th-order Butterworth analog lowpass filter. Display its magnitude and phase responses.

```
[z,p,k] = buttap(9);           % Butterworth filter prototype
[num,den] = zp2tf(z,p,k);     % Convert to transfer function form
freqs(num,den)                % Frequency response of analog filter
```



## More About

### Algorithms

```
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, chap.7.

## See Also

butter | besselpap | cheb1ap | cheb2ap | ellipap

## butter

Butterworth filter design

### Syntax

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

[z,p,k] = butter(____)
[A,B,C,D] = butter(____)

[____] = butter(____,'s')
```

### Description

`[b,a] = butter(n,Wn)` returns the transfer function coefficients of an  $n$ th-order lowpass digital Butterworth filter with normalized cutoff frequency  $Wn$ .

`[b,a] = butter(n,Wn,ftype)` designs a lowpass, highpass, bandpass, or bandstop Butterworth filter, depending on the value of `ftype` and the number of elements of  $Wn$ . The resulting bandpass and bandstop designs are of order  $2n$ .

---

**Note:** See “Limitations” on page 1-88 for information about numerical issues that affect forming the transfer function.

---

`[z,p,k] = butter(____)` designs a lowpass, highpass, bandpass, or bandstop digital Butterworth filter and returns its zeros, poles, and gain. This syntax can include any of the input arguments in previous syntaxes.

`[A,B,C,D] = butter(____)` designs a lowpass, highpass, bandpass, or bandstop digital Butterworth filter and returns the matrices that specify its state-space representation.

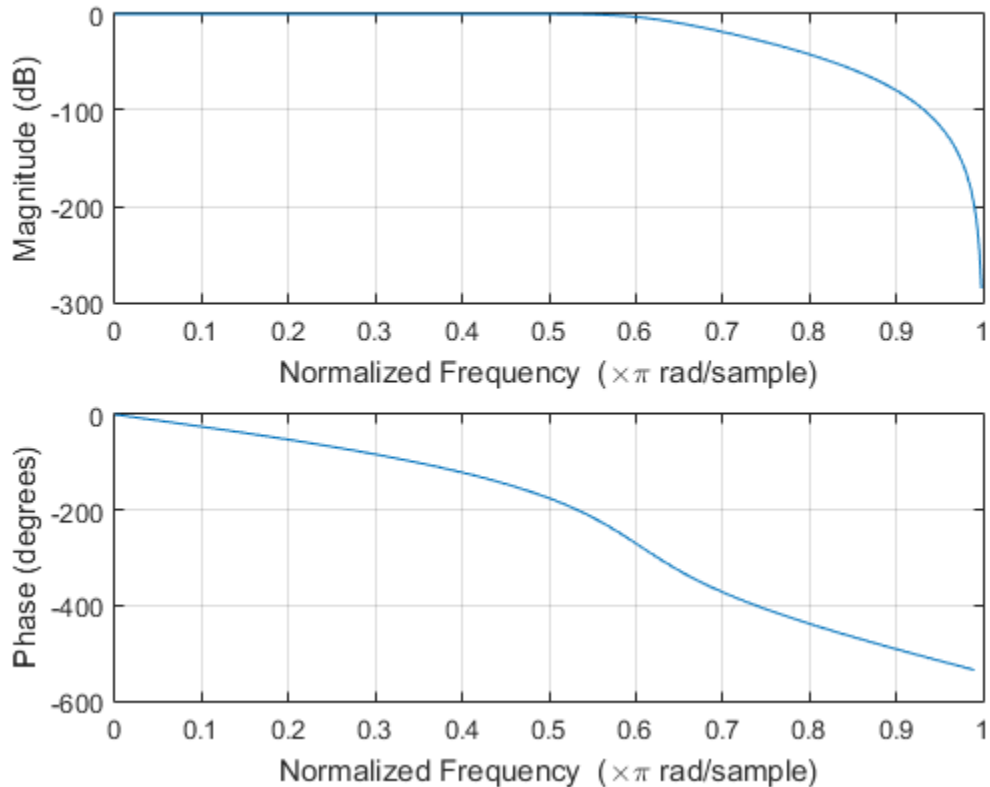
`[____] = butter(____,'s')` designs a lowpass, highpass, bandpass, or bandstop analog Butterworth filter with cutoff angular frequency  $Wn$ .

## Examples

### Lowpass Butterworth Transfer Function

Design a 6th-order lowpass Butterworth filter with a cutoff frequency of 300 Hz, which, for data sampled at 1000 Hz, corresponds to  $0.6\pi$  rad/sample. Plot its magnitude and phase responses. Use it to filter a 1000-sample random signal.

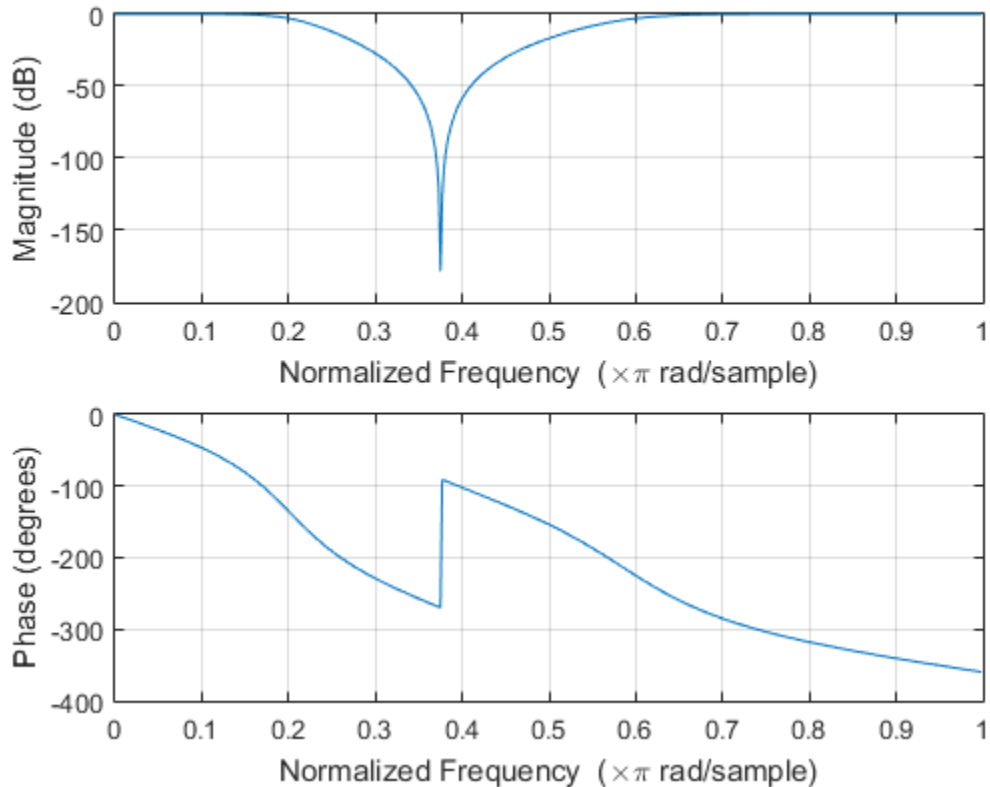
```
[b,a] = butter(6,0.6);  
freqz(b,a)  
dataIn = randn(1000,1);  
dataOut = filter(b,a,dataIn);
```



### Bandstop Butterworth Filter

Design a 6th-order Butterworth bandstop filter with normalized edge frequencies of  $0.2\pi$  and  $0.6\pi$  rad/sample. Plot its magnitude and phase responses. Use it to filter random data.

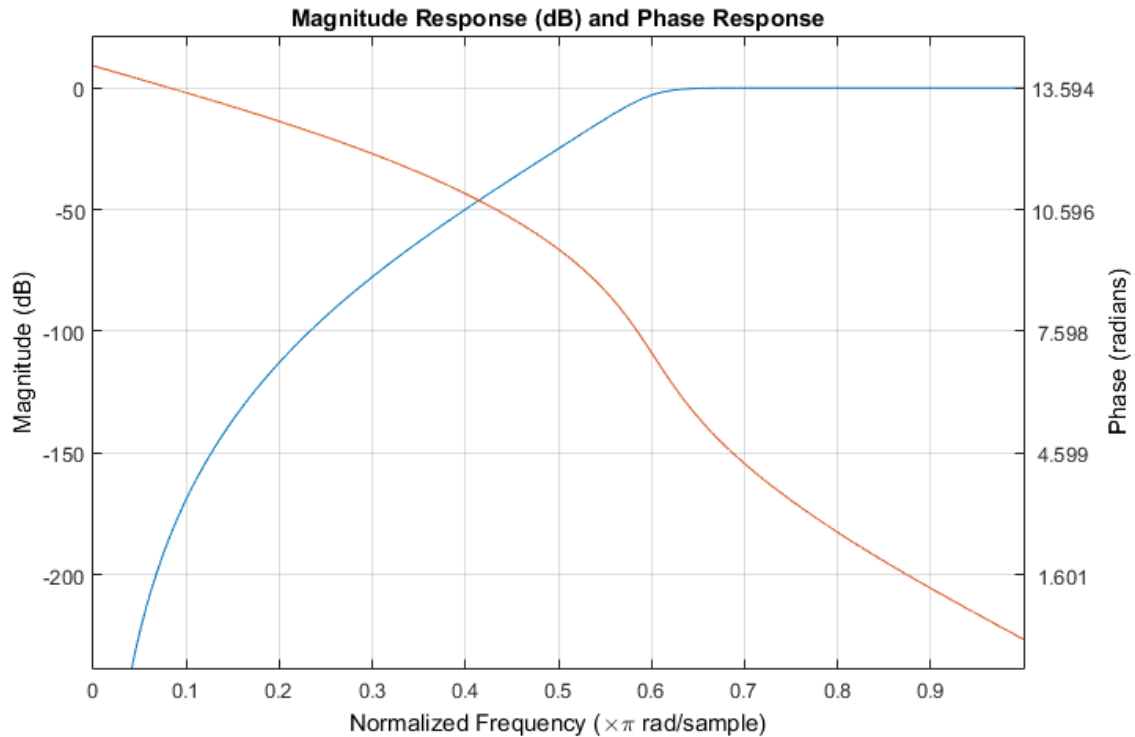
```
[b,a] = butter(3,[0.2 0.6], 'stop');  
freqz(b,a)  
dataIn = randn(1000,1);  
dataOut = filter(b,a,dataIn);
```



### Highpass Butterworth Filter

Design a 9th-order highpass Butterworth filter. Specify a cutoff frequency of 300 Hz, which, for data sampled at 1000 Hz, corresponds to  $0.6\pi$  rad/sample. Plot the magnitude and phase responses. Convert the zeros, poles, and gain to second-order sections for use by `fvtool`.

```
[z,p,k] = butter(9,300/500,'high');  
sos = zp2sos(z,p,k);  
fvtool(sos,'Analysis','freq')
```



### Bandpass Butterworth Filter

Design a 20th-order Butterworth bandpass filter with a lower cutoff frequency of 500 Hz and a higher cutoff frequency of 560 Hz. Specify a sample rate of 1500 Hz. Use the state-space representation. Design an identical filter using `designfilt`.

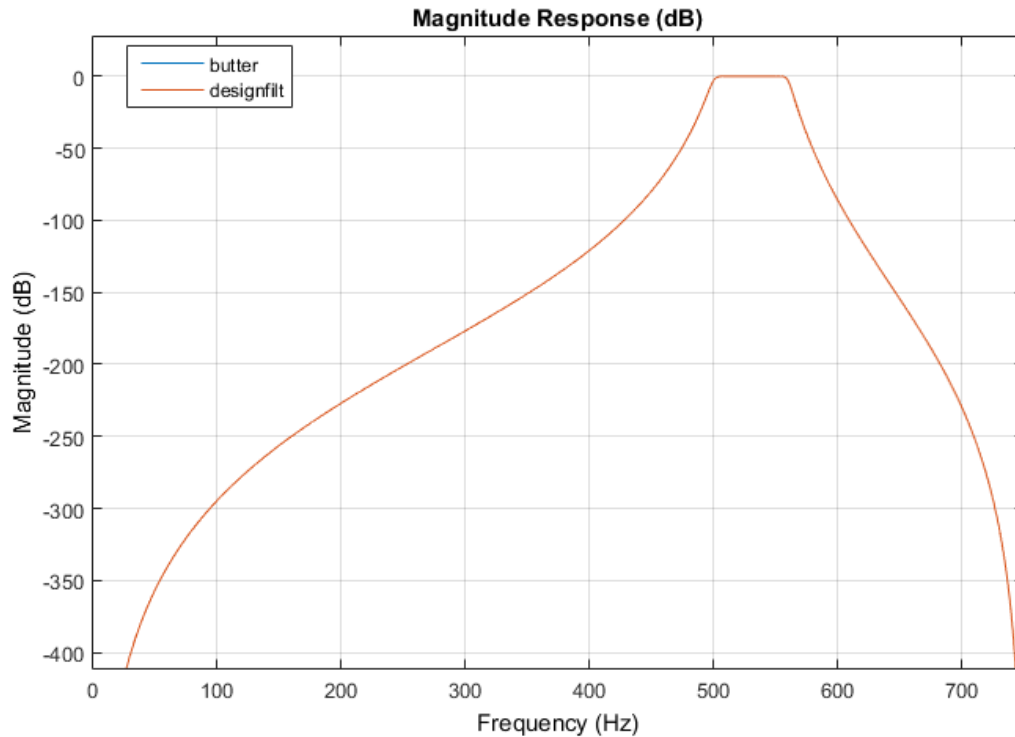
```
[A,B,C,D] = butter(10,[500 560]/750);
d = designfilt('bandpassiir','FilterOrder',20, ...
    'HalfPowerFrequency1',500,'HalfPowerFrequency2',560, ...
    'SampleRate',1500);
```

Convert the state-space representation to second-order sections. Visualize the frequency responses using `fvtool`.

```
sos = ss2sos(A,B,C,D);
fvtool(sos,d,'Fs',1500);
```



```
legend(fvt, 'butter', 'designfilt')
```



### Comparison of Analog IIR Lowpass Filters

Design a 5th-order analog Butterworth lowpass filter with a cutoff frequency of 2 GHz. Multiply by  $2\pi$  to convert the frequency to radians per second. Compute the frequency response of the filter at 4096 points.

```
n = 5;
f = 2e9;

[zb,pb,kb] = butter(n,2*pi*f,'s');
[bb,ab] = zp2tf(zb,pb,kb);
[hb,wb] = freqs(bb,ab,4096);
```

Design a 5th-order Chebyshev Type I filter with the same edge frequency and 3 dB of passband ripple. Compute its frequency response.

```
[z1,p1,k1] = cheby1(n,3,2*pi*f,'s');
[b1,a1] = zp2tf(z1,p1,k1);
[h1,w1] = freqs(b1,a1,4096);
```

Design a 5th-order Chebyshev Type II filter with the same edge frequency and 30 dB of stopband attenuation. Compute its frequency response.

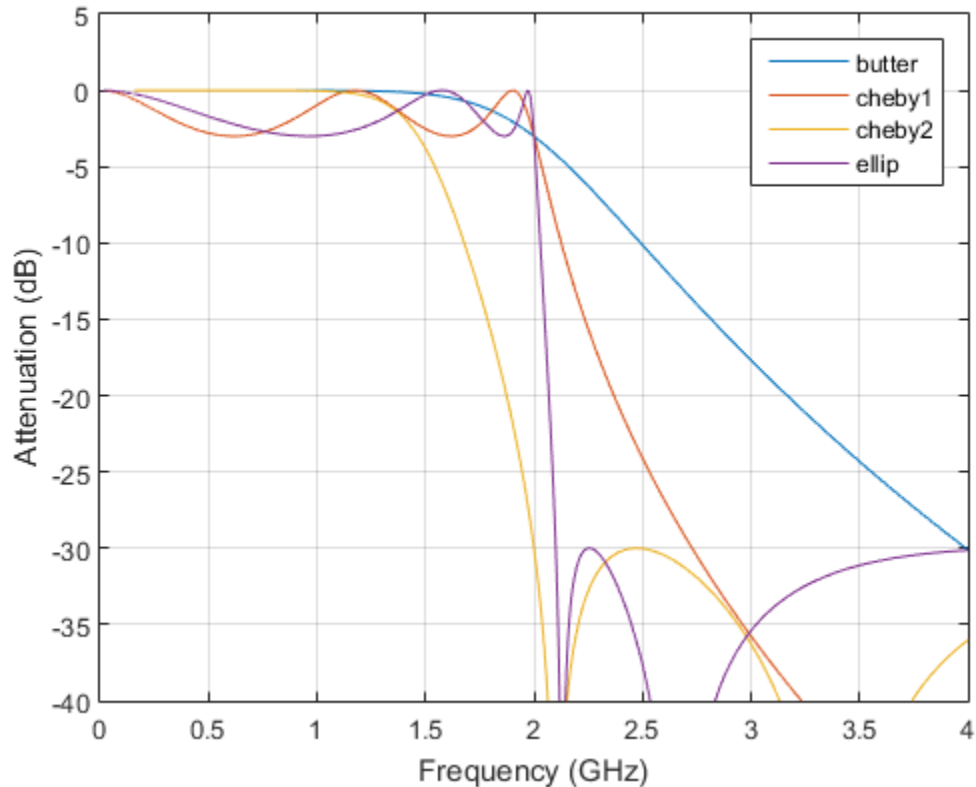
```
[z2,p2,k2] = cheby2(n,30,2*pi*f,'s');
[b2,a2] = zp2tf(z2,p2,k2);
[h2,w2] = freqs(b2,a2,4096);
```

Design a 5th-order elliptic filter with the same edge frequency, 3 dB of passband ripple, and 30 dB of stopband attenuation. Compute its frequency response.

```
[ze,pe,ke] = ellip(n,3,30,2*pi*f,'s');
[be,ae] = zp2tf(ze,pe,ke);
[he,we] = freqs(be,ae,4096);
```

Plot the attenuation in decibels. Express the frequency in gigahertz. Compare the filters.

```
plot(wb/(2e9*pi),mag2db(abs(hb)))
hold on
plot(w1/(2e9*pi),mag2db(abs(h1)))
plot(w2/(2e9*pi),mag2db(abs(h2)))
plot(we/(2e9*pi),mag2db(abs(he)))
axis([0 4 -40 5])
grid
xlabel('Frequency (GHz)')
ylabel('Attenuation (dB)')
legend('butter','cheby1','cheby2','ellip')
```



The Butterworth and Chebyshev Type II filters have flat passbands and wide transition bands. The Chebyshev Type I and elliptic filters roll off faster but have passband ripple. The frequency input to the Chebyshev Type II design function sets the beginning of the stopband rather than the end of the passband.

## Input Arguments

### **n** — Filter order

integer scalar

Filter order, specified as an integer scalar.

Data Types: double

**Wn — Cutoff frequency**

scalar | two-element vector

Cutoff frequency, specified as a scalar or a two-element vector. The cutoff frequency is the frequency at which the magnitude response of the filter is  $1/\sqrt{2}$ .

- If **Wn** is scalar, then **butter** designs a lowpass or highpass filter with cutoff frequency **Wn**.

If **Wn** is the two-element vector [**w1 w2**], where  $w1 < w2$ , then **butter** designs a bandpass or bandstop filter with lower cutoff frequency **w1** and higher cutoff frequency **w2**.

- For digital filters, the cutoff frequencies must lie between 0 and 1, where 1 corresponds to the Nyquist rate—half the sample rate or  $\pi$  rad/sample.

For analog filters, the cutoff frequencies must be expressed in radians per second and can take on any positive value.

Data Types: double

**ftype — Filter type**

'low' | 'bandpass' | 'high' | 'stop'

Filter type, specified as a string.

- 'low' specifies a lowpass filter with cutoff frequency **Wn**. 'low' is the default for scalar **Wn**.
- 'high' specifies a highpass filter with cutoff frequency **Wn**.
- 'bandpass' specifies a bandpass filter of order  $2n$  if **Wn** is a two-element vector. 'bandpass' is the default when **Wn** has two elements.
- 'stop' specifies a bandstop filter of order  $2n$  if **Wn** is a two-element vector.

Data Types: char

## Output Arguments

**b, a — Transfer function coefficients**

row vectors

Transfer function coefficients of the filter, returned as row vectors of length  $n + 1$  for lowpass and highpass filters and  $2n + 1$  for bandpass and bandstop filters.

- For digital filters, the transfer function is expressed in terms of **b** and **a** as

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

- For analog filters, the transfer function is expressed in terms of **b** and **a** as

$$H(s) = \frac{B(s)}{A(s)} = \frac{b(1)s^n + b(2)s^{n-1} + \dots + b(n+1)}{a(1)s^n + a(2)s^{n-1} + \dots + a(n+1)}.$$

Data Types: double

### **z, p, k — Zeros, poles, and gain**

column vectors, scalar

Zeros, poles, and gain of the filter, returned as two column vectors of length  $n$  ( $2n$  for bandpass and bandstop designs) and a scalar.

- For digital filters, the transfer function is expressed in terms of **z**, **p**, and **k** as

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

- For analog filters, the transfer function is expressed in terms of **z**, **p**, and **k** as

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

Data Types: double

### **A, B, C, D — State-space matrices**

matrices

State-space representation of the filter, returned as matrices. If  $m = n$  for lowpass and highpass designs and  $m = 2n$  for bandpass and bandstop filters, then **A** is  $m \times m$ , **B** is  $m \times 1$ , **C** is  $1 \times m$ , and **D** is  $1 \times 1$ .

- For digital filters, the state-space matrices relate the state vector  $x$ , the input  $u$ , and the output  $y$  through

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

- For analog filters, the state-space matrices relate the state vector  $x$ , the input  $u$ , and the output  $y$  through

$$\begin{aligned}\dot{x} &= Ax + Bu \\ y &= Cx + Du.\end{aligned}$$

Data Types: double

## More About

### Limitations

#### Numerical Instability of Transfer Function Syntax

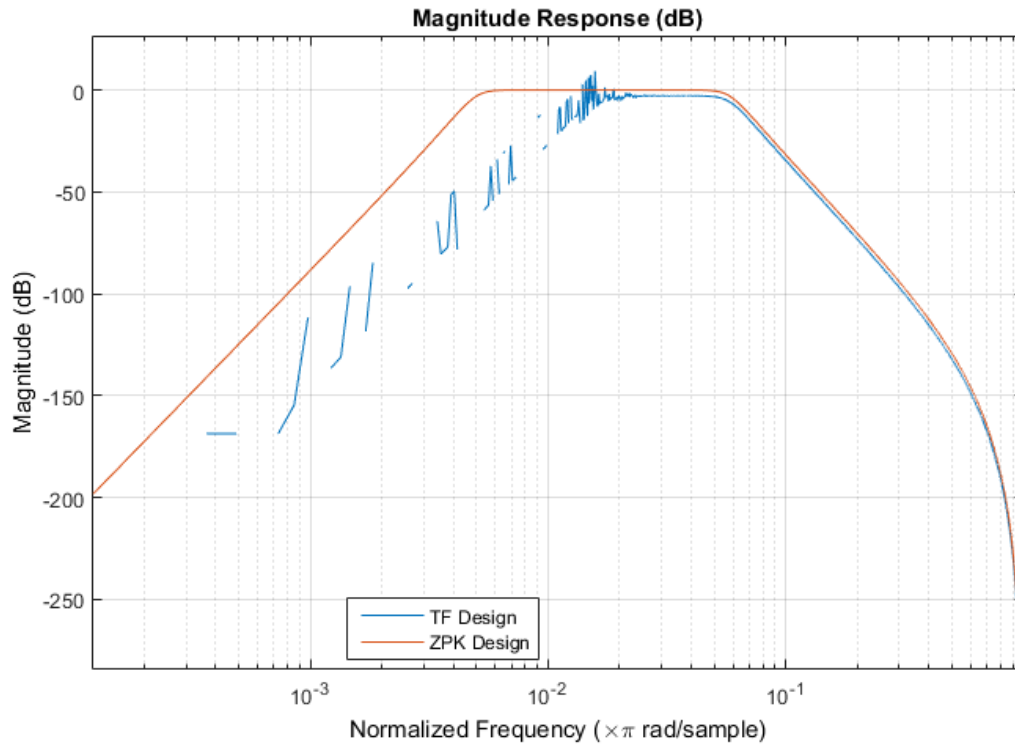
In general, use the  $[z, p, k]$  syntax to design IIR filters. To analyze or implement your filter, you can then use the  $[z, p, k]$  output with `zp2sos`. If you design the filter using the  $[b, a]$  syntax, you might encounter numerical problems. These problems are due to round-off errors and can occur for  $n$  as low as 4. The following example illustrates this limitation.

```
n = 6;
Wn = [2.5e6 29e6]/500e6;
ftype = 'bandpass';

% Transfer Function design
[b,a] = butter(n,Wn,ftype);      % This is an unstable filter

% Zero-Pole-Gain design
[z,p,k] = butter(n,Wn,ftype);
sos = zp2sos(z,p,k);

% Display and compare results
hfvt = fvtool(b,a,sos,'FrequencyScale','log');
legend(hfvt,'TF Design','ZPK Design')
```



## Algorithms

Butterworth filters have a magnitude response that is maximally flat in the passband and monotonic overall. This smoothness comes at the price of decreased rolloff steepness. Elliptic and Chebyshev filters generally provide steeper rolloff for a given filter order.

`butter` uses a five-step algorithm:

- 1 It finds the lowpass analog prototype poles, zeros, and gain using the function `buttap`.
- 2 It converts the poles, zeros, and gain into state-space form.
- 3 If required, it uses a state-space transformation to convert the lowpass filter into a bandpass, highpass, or bandstop filter with the desired frequency constraints.
- 4 For digital filter design, it uses `bilinear` to convert the analog filter into a digital filter through a bilinear transformation with frequency prewarping. Careful

frequency adjustment enables the analog filters and the digital filters to have the same frequency response magnitude at  $\omega_n$  or at  $\omega_1$  and  $\omega_2$ .

- 5 It converts the state-space filter back to its transfer function or zero-pole-gain form, as required.

## See Also

`besself` | `buttap` | `buttord` | `cheby1` | `cheby2` | `designfilt` | `ellip` | `filter` | `maxflat` | `sosfilt`



# butterd

Butterworth filter order and cutoff frequency

## Syntax

```
[n,Wn] = butterd(Wp,Ws,Rp,Rs)
[n,Wn] = butterd(Wp,Ws,Rp,Rs, 's')
```

## Description

`butterd` calculates the minimum order of a digital or analog Butterworth filter required to meet a set of filter design specifications.

### Digital Domain

`[n,Wn] = butterd(Wp,Ws,Rp,Rs)` returns the lowest order, `n`, of the digital Butterworth filter with no more than `Rp` dB of passband ripple and 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  |
|-----------------|--|
| <code>Wp</code> | Passband corner frequency <code>Wp</code> , 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, $\pi$ radians per sample. |
| <code>Ws</code> | Stopband corner frequency <code>Ws</code> , is a scalar or a two-element vector with values between 0 and 1, with 1 corresponding to the normalized Nyquist frequency.   |
| <code>Rp</code> | Passband ripple in decibels.   |
| <code>Rs</code> | 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     | $W_p < W_s$ , both scalars   | $(W_s, 1)$                      | $(0, W_p)$                      |
| Highpass    | $W_p > W_s$ , both scalars   | $(0, W_s)$                      | $(W_p, 1)$                      |
| Bandpass    | The interval specified by $W_s$ contains the one specified by $W_p$ ( $W_s(1) < W_p(1) < W_p(2) < W_s(2)$ ). | $(0, W_s(1))$ and $(W_s(2), 1)$ | $(W_p(1), W_p(2))$              |
| Bandstop    | The interval specified by $W_p$ contains the one specified by $W_s$ ( $W_p(1) < W_s(1) < W_s(2) < W_p(2)$ ). | $(W_s(1), W_s(2))$              | $(0, W_p(1))$ and $(W_p(2), 1)$ |

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  $W_n$  for an analog Butterworth filter. You specify the frequencies  $W_p$  and  $W_s$  similar those described in the Description of Stopband and Passband Filter Parameters 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 table above.

## Examples

### Lowpass Butterworth Filter

For data sampled at 1000 Hz, design a lowpass filter with no more than 3 dB of ripple in a passband from 0 to 40 Hz, and at least 60 dB of attenuation in the stopband. Find the filter order and cutoff frequency.

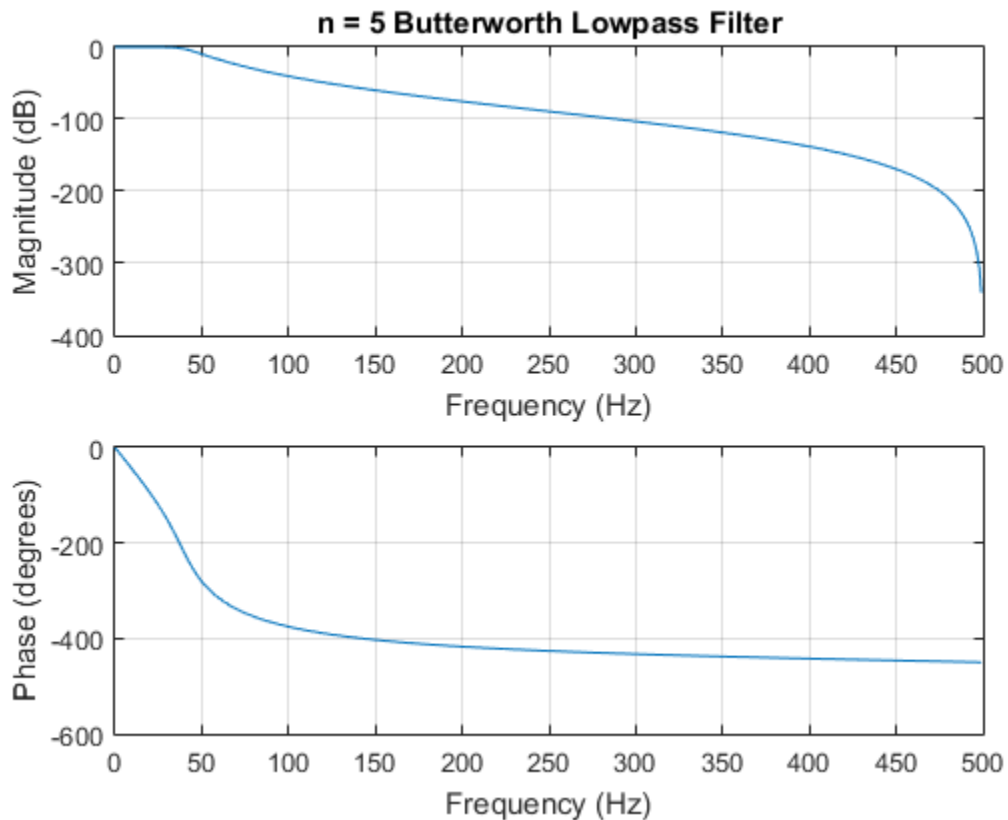
```
Wp = 40/500;  
Ws = 150/500;  
  
[n,Wn] = butterd(Wp,Ws,3,60)
```

```
n =  
  
    5
```

```
Wn =  
  
    0.0810
```

Specify the filter in terms of second-order sections and plot the frequency response.

```
[z,p,k] = butter(n,Wn);  
sos = zp2sos(z,p,k);  
  
freqz(sos,512,1000)  
title(sprintf('n = %d Butterworth Lowpass Filter',n))
```



### Bandpass Butterworth Filter

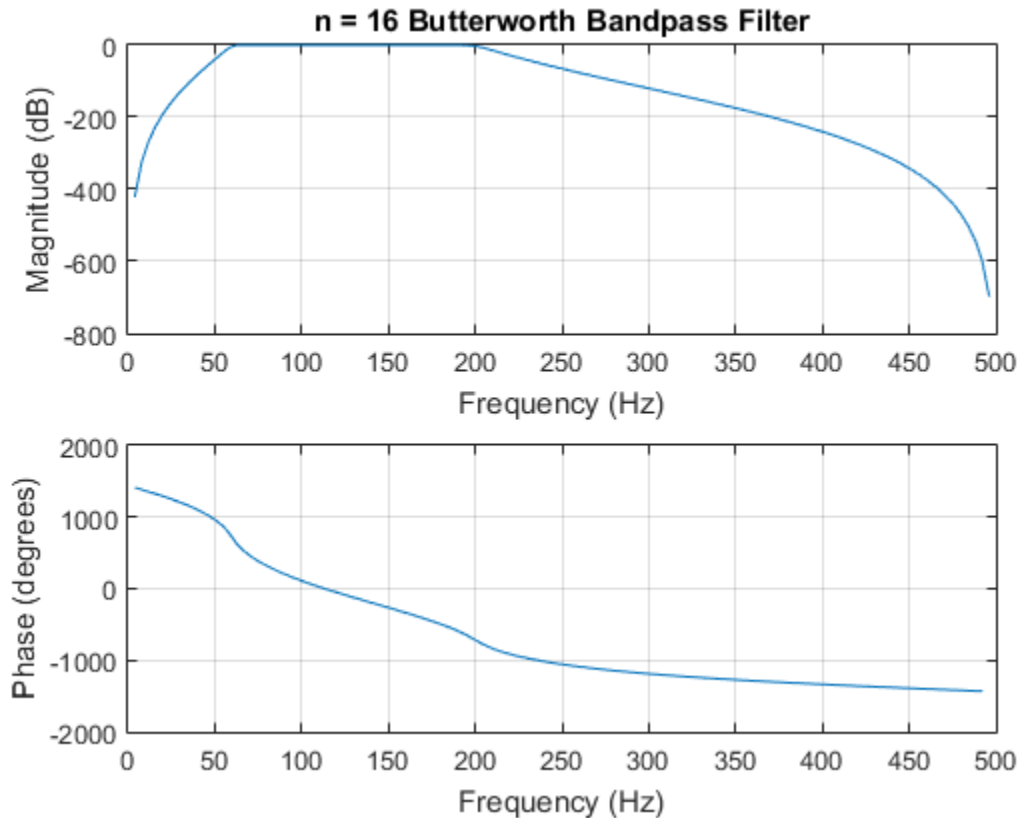
Design a bandpass filter with a passband from 60 to 200 Hz with at most 3 dB of passband ripple and at least 40 dB attenuation in the stopbands. Specify a sampling rate of 1 kHz. Have the stopbands be 50 Hz wide on both sides of the passband. Find the filter order and cutoff frequencies.

```
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
```

Specify the filter in terms of second-order sections and plot the frequency response.

```
[z,p,k] = butter(n,Wn);  
sos = zp2sos(z,p,k);  
  
freqz(sos,128,1000)  
title(sprintf('n = %d Butterworth Bandpass Filter',n))
```



## More About

### Algorithms

`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, Lawrence R., and Bernard Gold. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975.

## See Also

butter | cheb1ord | cheb2ord | ellipord | kaiserord

## **cceps**

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  $\pi$  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`.

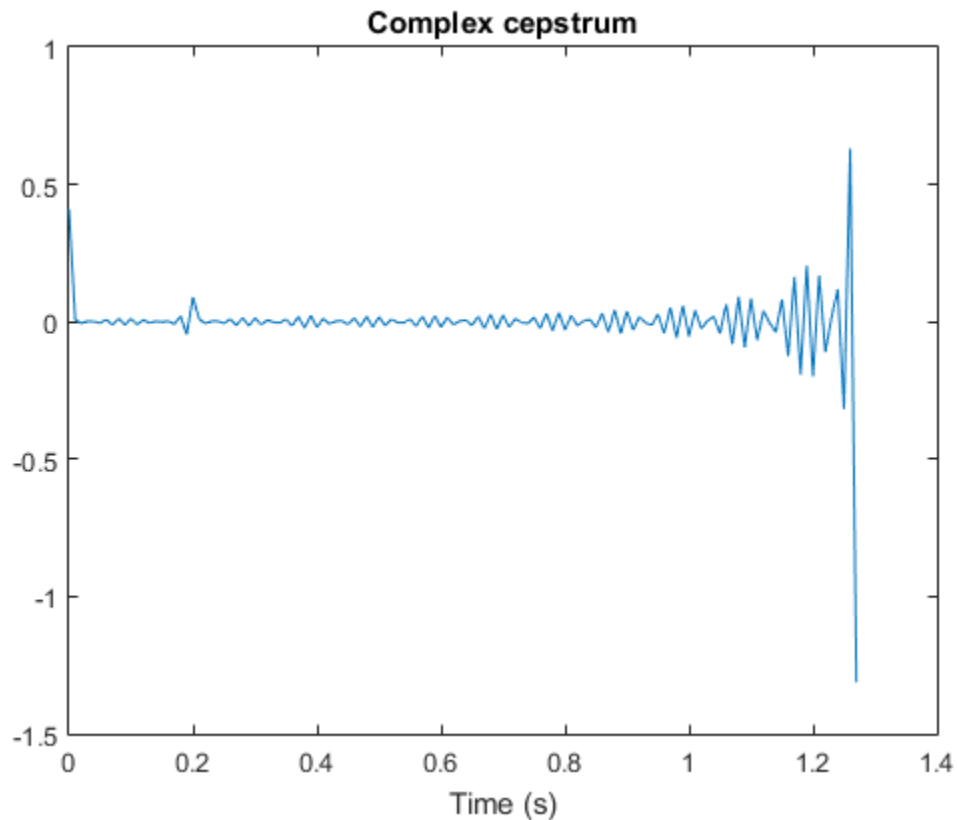


## Examples

### Using cceps to show an echo

This example uses `cceps` to show an echo. Generate a sine of frequency 45 Hz, sampled at 100 Hz. Add an echo with half the amplitude and 0.2 s later. Compute the complex cepstrum of the signal. Notice the echo at 0.2 s.

```
Fs = 100;  
t = 0:1/Fs:1.27;  
  
s1 = sin(2*pi*45*t);  
s2 = s1 + 0.5*[zeros(1,20) s1(1:108)];  
  
c = cceps(s2);  
  
plot(t,c)  
xlabel('Time (s)')  
title('Complex cepstrum')
```



## More About

### Algorithms

`cceps` is an 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).

## References

- [1] Oppenheim, Alan V., Ronald W. Schaffer, and John R. Buck. *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.” *Proceedings of the 1977 IEEE International Conference on Acoustics, Speech and Signal Processing*, pp.723–726.
- [3] Digital Signal Processing Committee of the IEEE Acoustics, Speech, and Signal Processing Society, eds. *Programs for Digital Signal Processing*. New York: IEEE Press, 1979.

## See Also

`icceps` | `hilbert` | `rceps` | `unwrap`

## **cconv**

Modulo-N circular convolution

### **Syntax**

```
c = cconv(a,b,n)
c = cconv(gpuArrayA,gpuArrayB,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).

`c = cconv(gpuArrayA,gpuArrayB,n)` returns the circular convolution of the input vectors of class `gpuArray`. See “Establish Arrays on a GPU” for details on `gpuArray` objects. Using `cconv` with `gpuArray` objects requires Parallel Computing Toolbox™ software and a CUDA-enabled NVIDIA GPU with compute capability 1.3 or above. See <http://www.mathworks.com/products/parallel-computing/requirements.html> for details. The output vector, `c`, is a `gpuArray` object. See “Circular Convolution using the GPU” on page 1-103 for an example of using the GPU to compute the circular convolution.

### **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
dif = norm(c-cref)

dif =
    9.7422e-16
```

The following example uses `C CONV` to compute the circular cross-correlation of two sequences. The result is compared to the cross-correlation computed using `X CORR`.

```
a = [1 2 2 1]+1i;
b = [1 3 4 1]-2*1i;
c = cconv(a,conj(fliplr(b)),7); % Compute using cconv
cref = xcorr(a,b);             % Compute using xcorr
dif = norm(c-cref)

dif =
    3.3565e-15
```

## Circular Convolution using the GPU

The following example requires Parallel Computing Toolbox software and a CUDA-enabled NVIDIA GPU with compute capability 1.3 or above. See <http://www.mathworks.com/products/parallel-computing/requirements.html> for details.

Create two signals consisting of a 1 kHz sine wave in additive white Gaussian noise. The sampling rate is 10 kHz

```
Fs = 1e4;
t = 0:1/Fs:10-(1/Fs);
x = cos(2*pi*1e3*t)+randn(size(t));
y = sin(2*pi*1e3*t)+randn(size(t));
```

Put `x` and `y` on the GPU using `gpuArray`. Obtain the circular convolution using the GPU.

```
x = gpuArray(x);  
y = gpuArray(y);  
cirC = cconv(x,y,length(x)+length(y)-1);
```

Compare the result to the linear convolution of `x` and `y`.

```
linC = conv(x,y);  
norm(linC-cirC,2)
```

Return the circular convolution, `cirC`, to the MATLAB workspace using `gather`.

```
cirC = gather(cirC);
```

## References

- [1] Orfanidis, S. J. *Introduction to Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1996, pp. 524–529.

## See Also

`conv` | `xcorr`

## cell2sos

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  $b_i$  and  $a_i$  are row vectors of at most length 3, and  $i = 1, 2, \dots, L$ . The resulting matrix `m` is given by

```
m = [b1 a1;b2 a2; ... ;bL aL]
```

### Examples

#### Second-Order Sections from Cell Array Input

Generate a cell array of 1-by-2 cell arrays of 1-by-3 row vectors. Convert it to a matrix of second-order sections.

```
c11 = {[3 6 7] [1 1 2]}
      {[1 4 5] [1 9 3]}
      {[2 7 1] [1 7 8]};
sos = cell2sos(c11)
```

```
sos =
```

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| 3 | 6 | 7 | 1 | 1 | 2 |
| 1 | 4 | 5 | 1 | 9 | 3 |
| 2 | 7 | 1 | 1 | 7 | 8 |

## See Also

`sos2cell` | `tf2sos`



# cfirpm

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)
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

$\mathbf{b} = \text{cfirpm}(n, \mathbf{f}, \{\text{@invsinc}, \mathbf{a}, \mathbf{d}\}, \dots)$  specify gain  $\mathbf{a}$  for the sinc-function, computed as  $\text{sinc}(\mathbf{a} * \mathbf{g})$ , where  $\mathbf{g}$  contains the optimization grid frequencies normalized to the range  $[-1, 1]$ . By default,  $\mathbf{a} = 1$ . The group-delay offset is  $\mathbf{d}$ , such that the filter response will have a group delay of  $N/2 + \mathbf{d}$  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,  $\mathbf{d} = 0$ .

$\mathbf{b} = \text{cfirpm}(n, \mathbf{f}, \text{@fresp}, \mathbf{w})$  uses the real, nonnegative weights in vector  $\mathbf{w}$  to weight the fit in each frequency band. The length of  $\mathbf{w}$  is half the length of  $\mathbf{f}$ , so there is exactly one weight per band.

$\mathbf{b} = \text{cfirpm}(n, \mathbf{f}, \mathbf{a})$  is a synonym for  $\mathbf{b} = \text{cfirpm}(n, \mathbf{f}, \{\text{@multiband}, \mathbf{a}\})$ .

$\mathbf{b} = \text{cfirpm}(n, \mathbf{f}, \mathbf{a}, \mathbf{w})$  applies an optional set of positive weights, one per band, for use during optimization. If  $\mathbf{w}$  is not specified, the weights are set to unity.

$\mathbf{b} = \text{cfirpm}(\dots, \text{'sym'})$  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  $N$ .

$\mathbf{b} = \text{cfirpm}(\dots, \text{'skip\_stage2'})$  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  $2^{\text{nextpow2}(\text{lgrid} \cdot n)}$  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  |
|------------------------|--|
| <code>opt.fgrid</code> | Frequency grid vector used for the filter design optimization          |
| <code>opt.des</code>   | Desired frequency response for each point in <code>opt.fgrid</code>    |
| <code>opt.wt</code>    | Weighting for each point in <code>opt.fgrid</code>                     |
| <code>opt.H</code>     | Actual frequency response for each point in <code>opt.fgrid</code>     |
| <code>opt.error</code> | Error at each point in <code>opt.fgrid</code>                          |
| <code>opt.iextr</code> | Vector of indices into <code>opt.fgrid</code> for extremal frequencies |
| <code>opt.fextr</code> | 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,...})
```

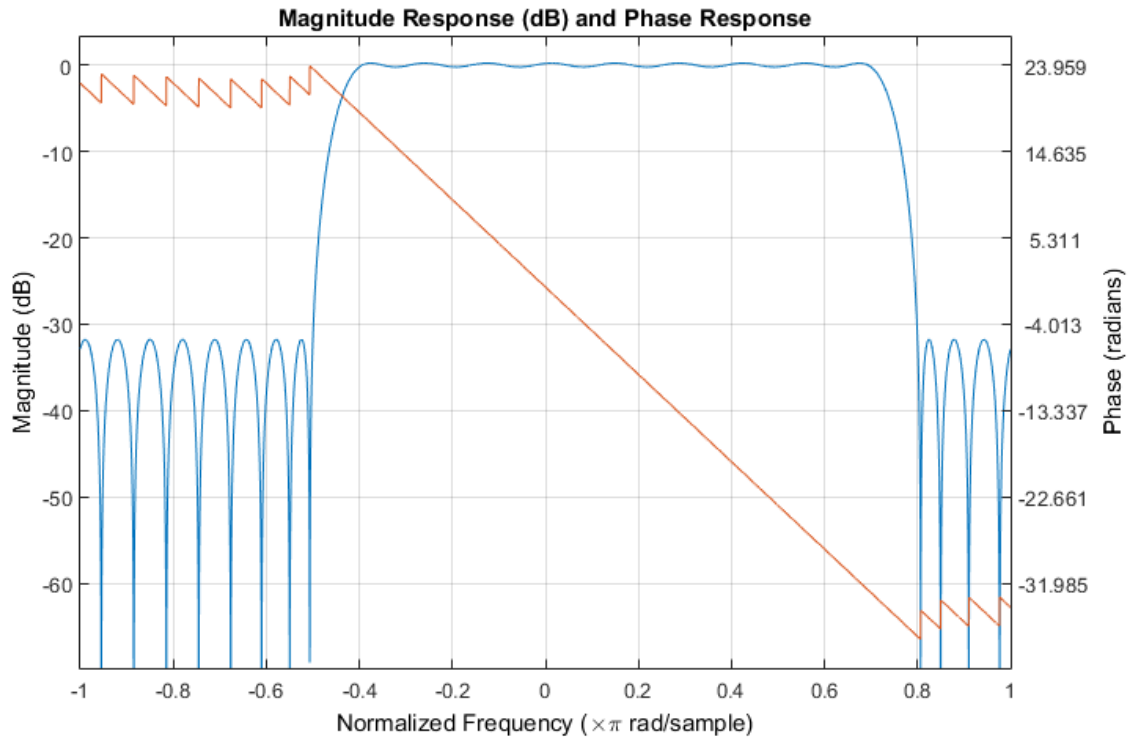
The arguments may be used in determining an appropriate symmetry default as necessary. The function `private/lowpass.m` may be useful as a template for generating new frequency response functions.

## Examples

### Equiripple Lowpass Filter

Design a 31-tap linear-phase lowpass filter. Display its magnitude and phase responses.

```
b = cfirpm(30,[-1 -0.5 -0.4 0.7 0.8 1],@lowpass);
fvtool(b,1,'OverlaidAnalysis','phase')
```



### FIR Approximation to Allpass Response

Design a nonlinear-phase allpass FIR filter of order 22 with frequency response given approximately by  $\exp(-j\pi f N/2 + j4\pi f|f|)$ , where  $f \in [-1, 1]$ .

```
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
```

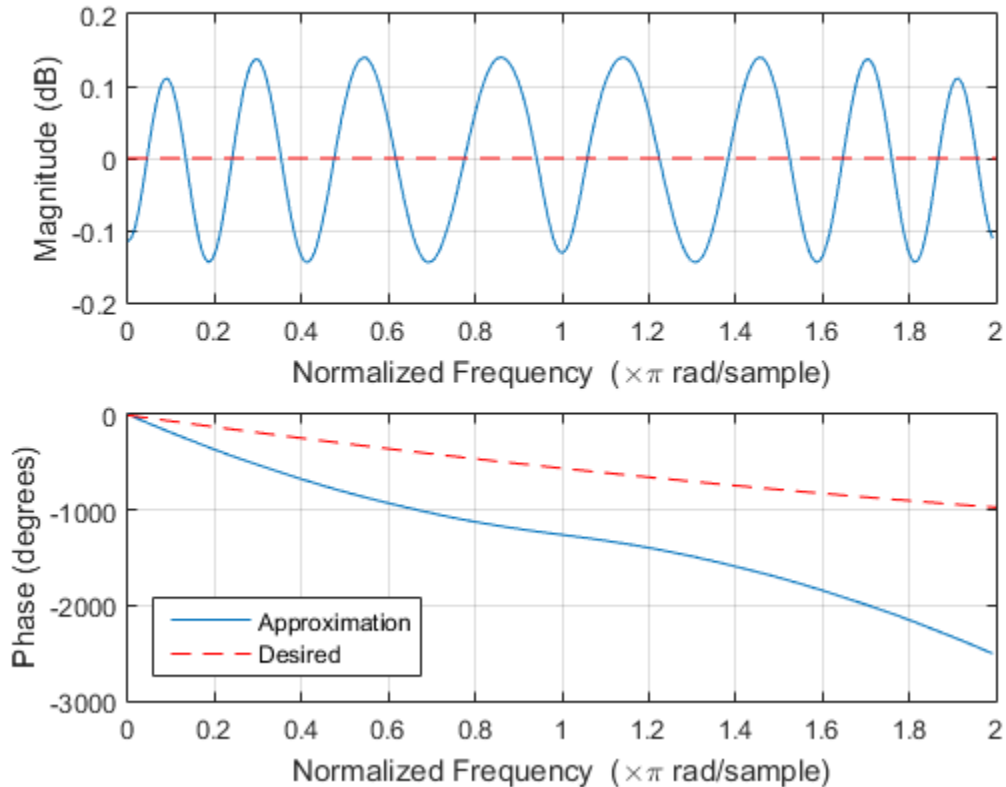
Use `cfirpm` to compute the FIR filter. Plot the actual and approximate magnitude responses in dB and the phase responses in degrees.

```
b = cfirpm(n,f,'allpass',w,'real'); % Approximation
```

```
freqz(b,1,256,'whole')

subplot(2,1,1)                                % Overlay response
hold on
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','Location','SouthWest')
```



## More About

### Algorithms

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

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  $R_p$  dB of ripple in the passband. The function returns the poles in the length  $n$  column vector  $\mathbf{p}$  and the gain in scalar  $k$ .  $\mathbf{z}$  is an empty matrix, because there are no zeros. The transfer function is

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

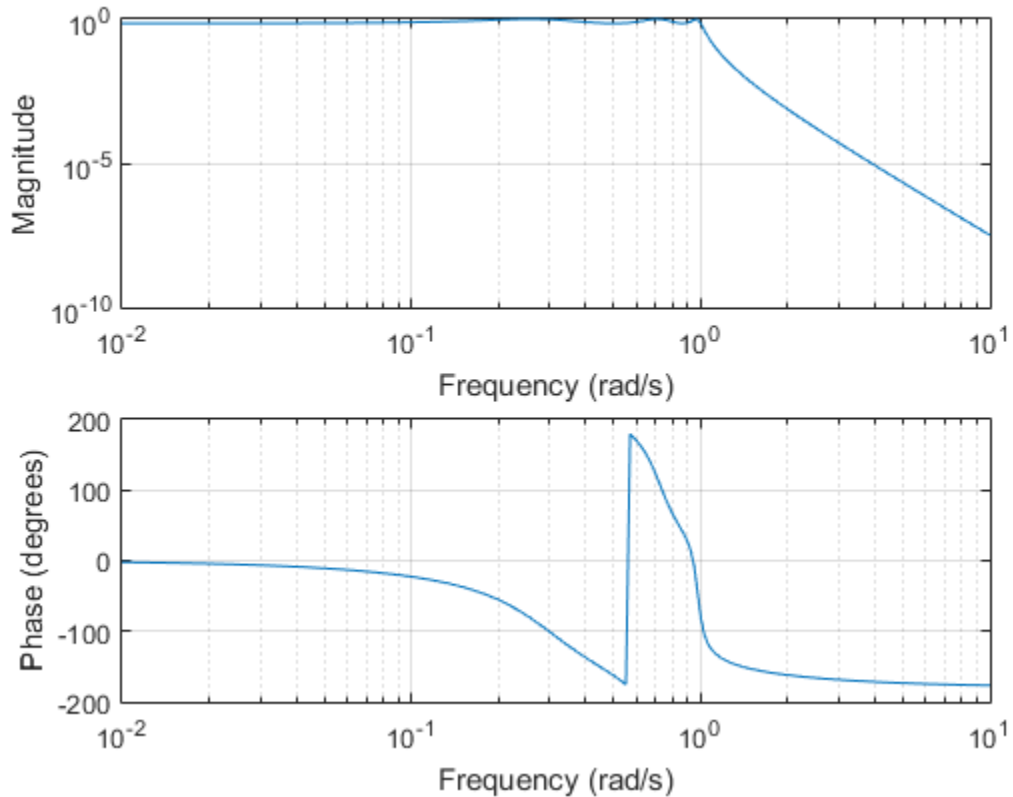
Chebyshev Type I filters are equiripple in the passband and monotonic in the stopband. The poles are evenly spaced about an ellipse in the left half plane. The Chebyshev Type I passband edge angular frequency  $\omega_0$  is set to 1.0 for a normalized result. This is the frequency at which the passband ends and the filter has magnitude response of  $10^{-R_p/20}$ .

### Examples

#### Frequency Response of an Analog Chebyshev Type I Filter

Design a 6th-order Chebyshev Type I analog lowpass filter with 3 dB of ripple in the passband. Display its magnitude and phase responses.

```
[z,p,k] = cheb1ap(6,3);           % Lowpass filter prototype
[num,den] = zp2tf(z,p,k);        % Convert to transfer function form
freqs(num,den)                   % Frequency response of analog filter
```



## References

- [1] Parks, Thomas W., and C. Sidney Burrus. *Digital Filter Design*. New York: John Wiley & Sons, 1987, chap.7.

## See Also

besselap | buttap | cheby1 | cheb2ap | ellipap

# cheb1ord

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  |
|-----------------|--|
| <code>Wp</code> | Passband corner frequency <code>Wp</code> , 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, $\pi$ radians per sample. |
| <code>Ws</code> | Stopband corner frequency <code>Ws</code> , is a scalar or a two-element vector with values between 0 and 1, with 1 corresponding to the normalized Nyquist frequency.   |
| <code>Rp</code> | Passband ripple, in decibels. This value is the maximum permissible passband loss in decibels.   |

| Parameter | Description   |
|-----------|---|
| 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     | $W_p < W_s$ , both scalars   | $(W_s, 1)$                      | $(0, W_p)$         |
| Highpass    | $W_p > W_s$ , both scalars   | $(0, W_s)$                      | $(W_p, 1)$         |
| Bandpass    | The interval specified by $W_s$ contains the one specified by $W_p$ ( $W_s(1) < W_p(1) < W_p(2) < W_s(2)$ ). | $(0, W_s(1))$ and $(W_s(2), 1)$ | $(W_p(1), W_p(2))$ |
| Bandstop    | The interval specified by $W_p$ contains the one specified by $W_s$ ( $W_p(1) < W_s(1) < W_s(2) < W_p(2)$ ). | $(0, W_p(1))$ and $(W_p(2), 1)$ | $(W_s(1), W_s(2))$ |

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

$[n, W_p] = \text{cheb1ord}(W_p, W_s, R_p, R_s, 's')$  finds the minimum order  $n$  and cutoff frequencies  $W_p$  for an analog Chebyshev Type I filter. You specify the frequencies  $W_p$  and  $W_s$  similar to those described in the Description of Stopband and Passband Filter Parameters 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 table above.

## Examples

### Chebyshev Type I Filter Design

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.

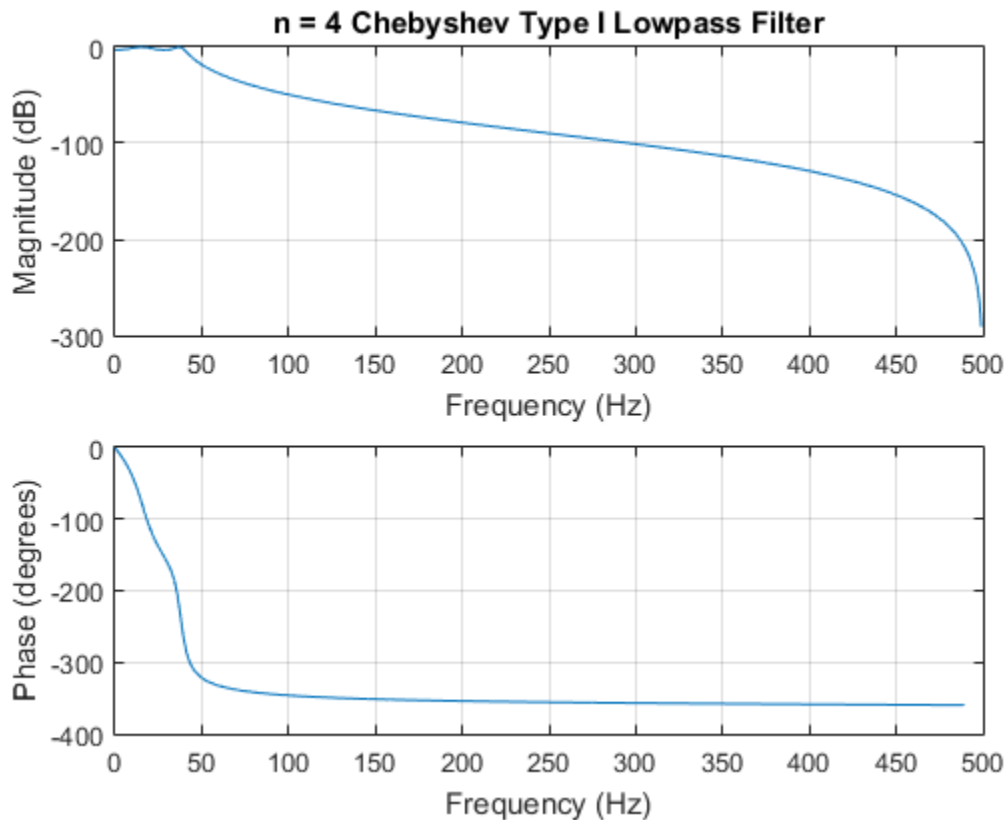
```
Wp = 40/500;  
Ws = 150/500;  
Rp = 3;  
Rs = 60;  
[n,Wp] = cheb1ord(Wp,Ws,Rp,Rs)  
[b,a] = cheby1(n,Rp,Wp);  
freqz(b,a,512,1000)  
title('n = 4 Chebyshev Type I Lowpass Filter')
```

```
n =
```

```
    4
```

```
Wp =
```

```
    0.0800
```



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)
[b,a] = cheby1(n,Rp,Wp);
freqz(b,a,512,1000)
title('n = 7 Chebyshev Type I Bandpass Filter')

```

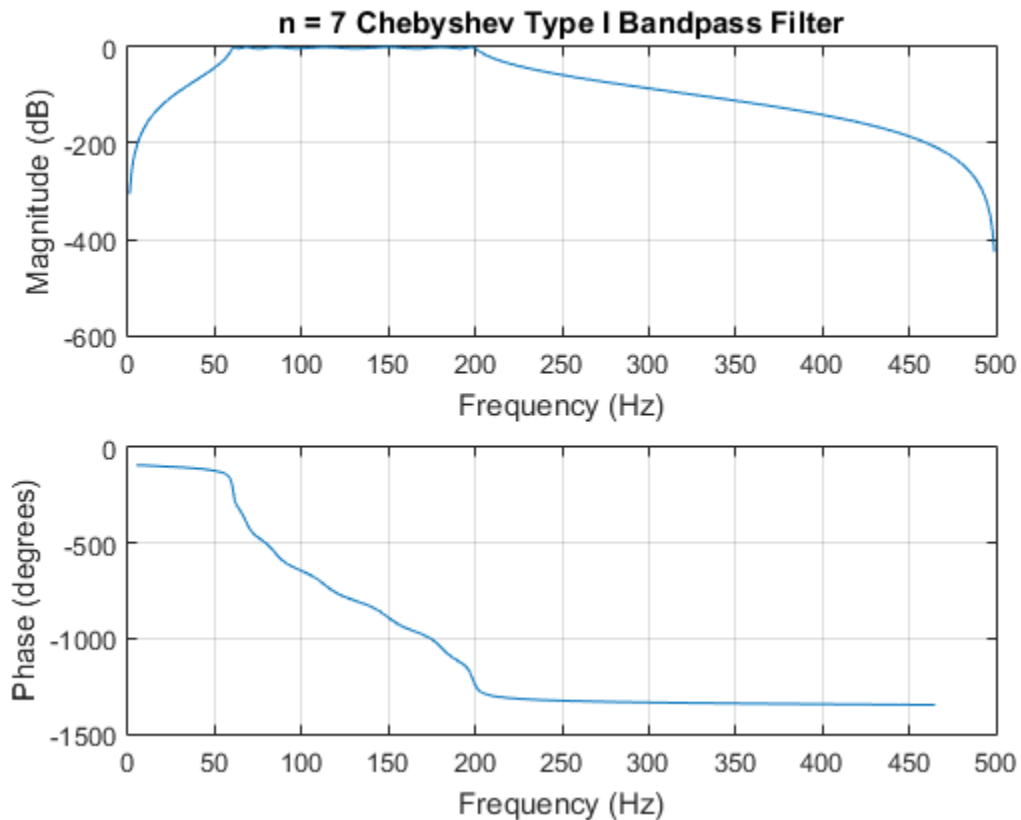
n =

7

Wp =

0.1200    0.4000





## More About

### Algorithms

`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, Lawrence R., and Bernard Gold. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975.

## See Also

`buttord` | `cheby1` | `cheb2ord` | `ellipord` | `kaiserord`

# cheb2ap

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

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

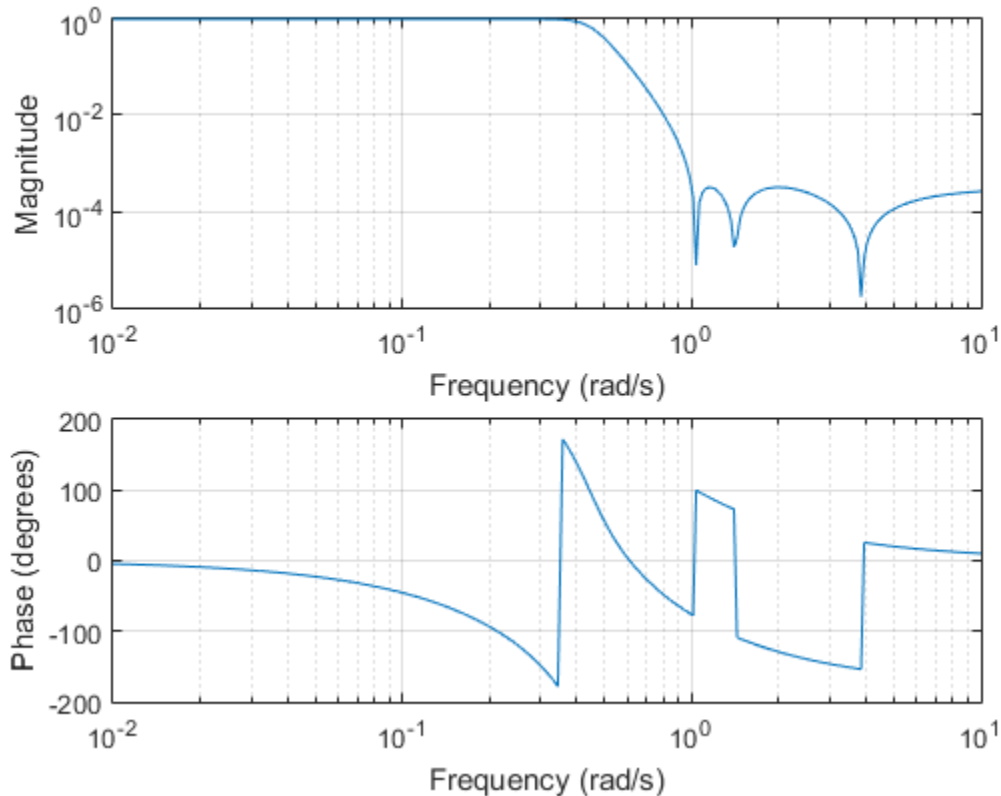
Chebyshev Type II filters are monotonic in the passband and equiripple in the stopband. The pole locations are the inverse of the pole locations of `cheb1ap`, whose poles are evenly spaced about an ellipse in the left half plane. The Chebyshev Type II stopband edge angular frequency  $\omega_0$  is set to 1 for a normalized result. This is the frequency at which the stopband begins and the filter has magnitude response of  $10^{-Rs/20}$ .

## Examples

### Frequency Response of an Analog Chebyshev Type II Filter

Design a 6th-order Chebyshev Type II analog lowpass filter with 70 dB of ripple in the stopband. Display its magnitude and phase responses.

```
[z,p,k] = cheb2ap(6,70);      % Lowpass filter prototype
[num,den] = zp2tf(z,p,k);    % Convert to transfer function form
freqs(num,den)              % Frequency response of analog filter
```



## More About

### Algorithms

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  $\omega$  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, Thomas W., and C. Sidney Burrus. *Digital Filter Design*. New York: John Wiley & Sons, 1987, chap.7.

## See Also

besselap | cheby2 | buttap | cheb1ap | ellipap

## cheb2ord

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  |
|-----------------|--|
| <code>Wp</code> | Passband corner frequency <code>Wp</code> , 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, $\pi$ radians per sample. |
| <code>Ws</code> | Stopband corner frequency <code>Ws</code> , is a scalar or a two-element vector with values between 0 and 1, with 1 corresponding to the normalized Nyquist frequency.   |
| <code>Rp</code> | Passband ripple, in decibels. This value is the maximum permissible passband loss in decibels.   |

| Parameter | Description   |
|-----------|---|
| 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     | $W_p < W_s$ , both scalars   | $(W_s, 1)$                            | $(0, W_p)$         |
| Highpass    | $W_p > W_s$ , both scalars   | $(0, W_s)$                            | $(W_p, 1)$         |
| Bandpass    | The interval specified by $W_s$ contains the one specified by $W_p$ ( $W_s(1) < W_p(1) < W_p(2) < W_s(2)$ ). | $(0, W_s(1))$<br>and<br>$(W_s(2), 1)$ | $(W_p(1), W_p(2))$ |
| Bandstop    | The interval specified by $W_p$ contains the one specified by $W_s$ ( $W_p(1) < W_s(1) < W_s(2) < W_p(2)$ ). | $(0, W_p(1))$<br>and<br>$(W_p(2), 1)$ | $(W_s(1), W_s(2))$ |

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

$[n, W_s] = \text{cheb2ord}(W_p, W_s, R_p, R_s, 's')$  finds the minimum order  $n$  and cutoff frequencies  $W_s$  for an analog Chebyshev Type II filter. You specify the frequencies  $W_p$  and  $W_s$  similar to those described in the Description of Stopband and Passband Filter Parameters 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 table above.

## Examples

### Chebyshev Type II Filter Design

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.

```
Wp = 40/500;  
Ws = 150/500;  
Rp = 3;  
Rs = 60;
```

```
[n,Ws] = cheb2ord(Wp,Ws,Rp,Rs)  
[b,a] = cheby2(n,Rs,Ws);
```

```
freqz(b,a,512,1000)  
title('n = 4 Chebyshev Type II Lowpass Filter')
```

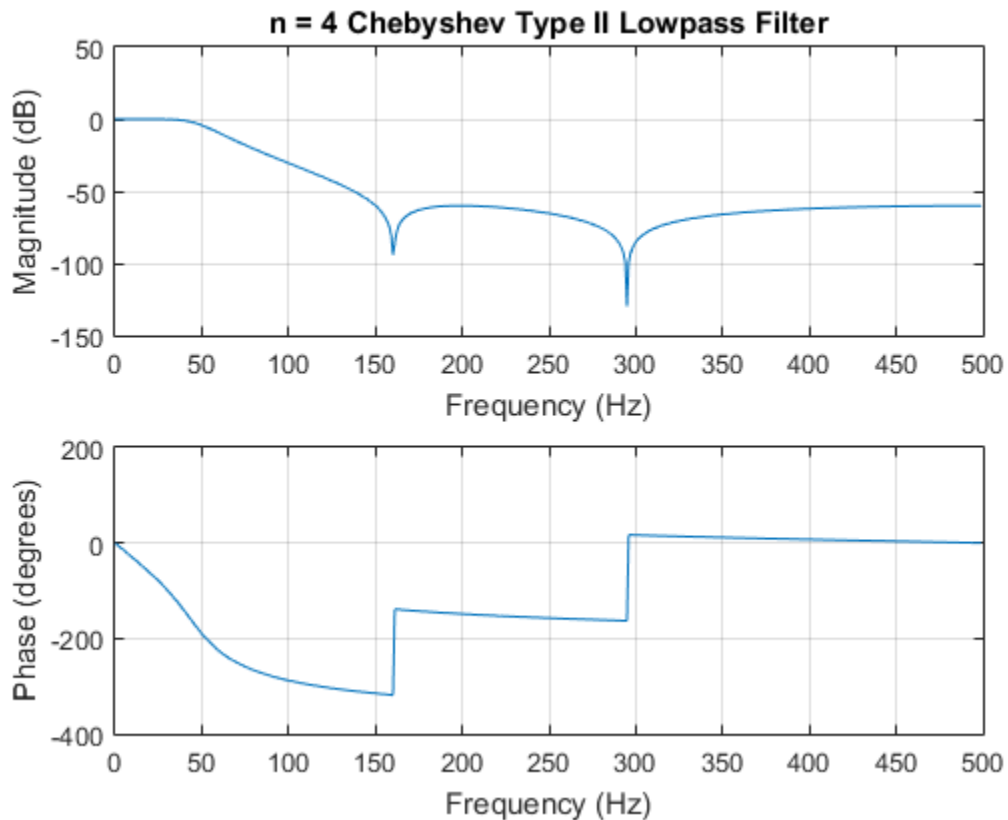
```
n =
```

```
    4
```

```
Ws =
```

```
    0.3000
```





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)
[b,a] = cheby2(n,Rs,Ws);
```

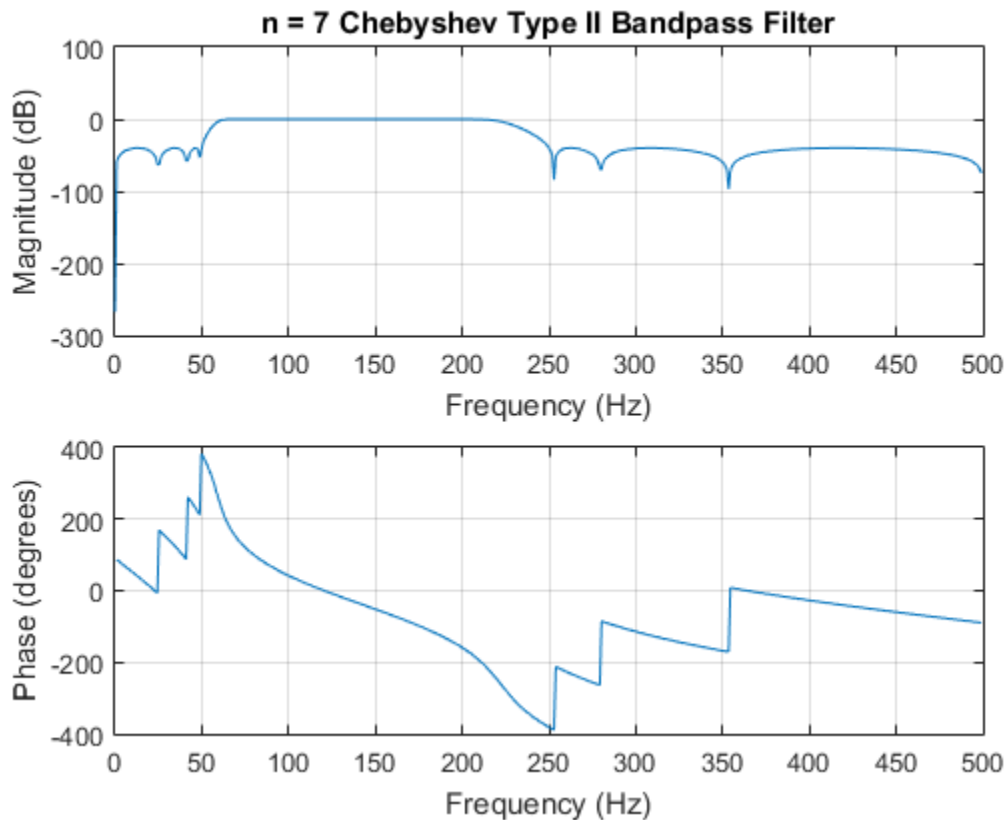
```
freqz(b,a,512,1000)
title('n = 7 Chebyshev Type II Bandpass Filter')
```

n =

7

Ws =

0.1000    0.5000



## More About

### Algorithms

`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, Lawrence R., and Bernard Gold. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975.

## See Also

`buttord` | `cheby2` | `cheb1ord` | `ellipord` | `kaiserord`

# chebwin

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.

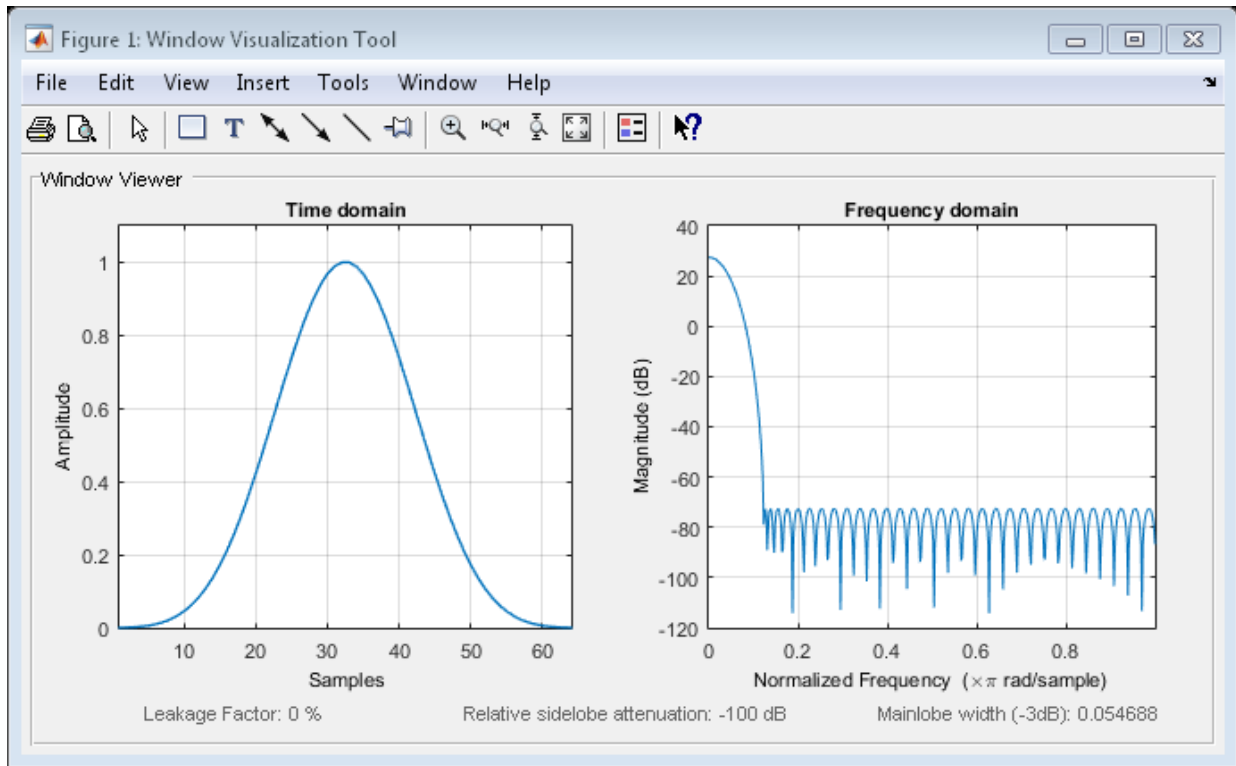
---

## Examples

### Chebyshev Window

Create a 64-point Chebyshev window with 100 dB of sidelobe attenuation. Display the result using `wvtool`.

```
L = 64;  
bw = chebwin(L);  
wvtool(bw)
```



## More About

### Algorithms

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].

The equivalent noise bandwidth of a Chebyshev window does not grow monotonically with increasing sidelobe attenuation when the attenuation is smaller than about 45 dB. For spectral analysis, use larger sidelobe attenuation values, or, if you need to work with small attenuations, use a Kaiser window.

## References

- [1] Digital Signal Processing Committee of the IEEE Acoustics, Speech, and Signal Processing Society, eds. *Programs for Digital Signal Processing*. New York: IEEE Press, 1979, program 5.2.
- [2] Harris, Fredric J. *Multirate Signal Processing for Communication Systems*. Upper Saddle River, NJ: Prentice Hall PTR, 2004, pp. 60–64.

## See Also

gausswin | window | kaiser | tukeywin | wintool | wvtool

# cheby1

Chebyshev Type I filter design

## Syntax

```
[b,a] = cheby1(n,Rp,Wp)
[b,a] = cheby1(n,Rp,Wp,ftype)

[z,p,k] = cheby1(____)
[A,B,C,D] = cheby1(____)

[____] = cheby1(____,'s')
```

## Description

`[b,a] = cheby1(n,Rp,Wp)` returns the transfer function coefficients of an  $n$ th-order lowpass digital Chebyshev Type I filter with normalized passband edge frequency  $W_p$  and  $R_p$  decibels of peak-to-peak passband ripple.

`[b,a] = cheby1(n,Rp,Wp,ftype)` designs a lowpass, highpass, bandpass, or bandstop Chebyshev Type I filter, depending on the value of `ftype` and the number of elements of  $W_p$ . The resulting bandpass and bandstop designs are of order  $2n$ .

---

**Note:** See “Limitations” on page 1-148 for information about numerical issues that affect forming the transfer function.

---

`[z,p,k] = cheby1(____)` designs a lowpass, highpass, bandpass, or bandstop digital Chebyshev Type I filter and returns its zeros, poles, and gain. This syntax can include any of the input arguments in previous syntaxes.

`[A,B,C,D] = cheby1(____)` designs a lowpass, highpass, bandpass, or bandstop digital Chebyshev Type I filter and returns the matrices that specify its state-space representation.



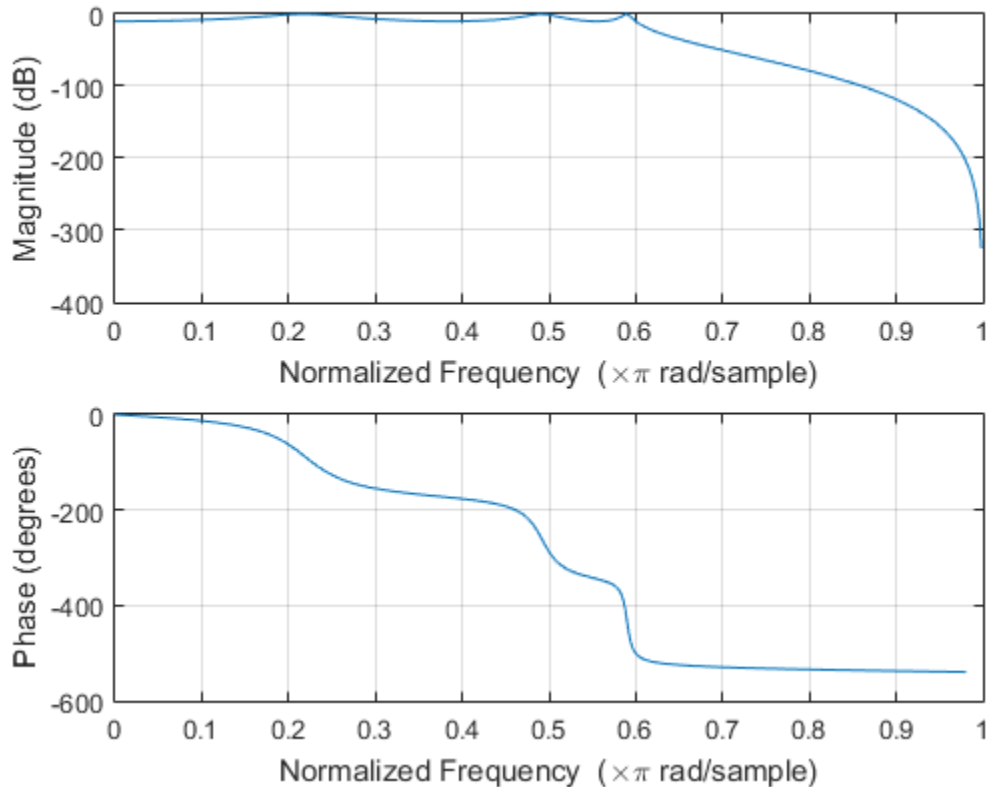
[    ] = cheby1(    , 's') designs a lowpass, highpass, bandpass, or bandstop analog Chebyshev Type I filter with passband edge angular frequency  $W_p$  and  $R_p$  decibels of passband ripple.

## Examples

### Lowpass Chebyshev Type I Transfer Function

Design a 6th-order lowpass Chebyshev Type I filter with 10 dB of passband ripple and a passband edge frequency of 300 Hz, which, for data sampled at 1000 Hz, corresponds to  $0.6\pi$  rad/sample. Plot its magnitude and phase responses. Use it to filter a 1000-sample random signal.

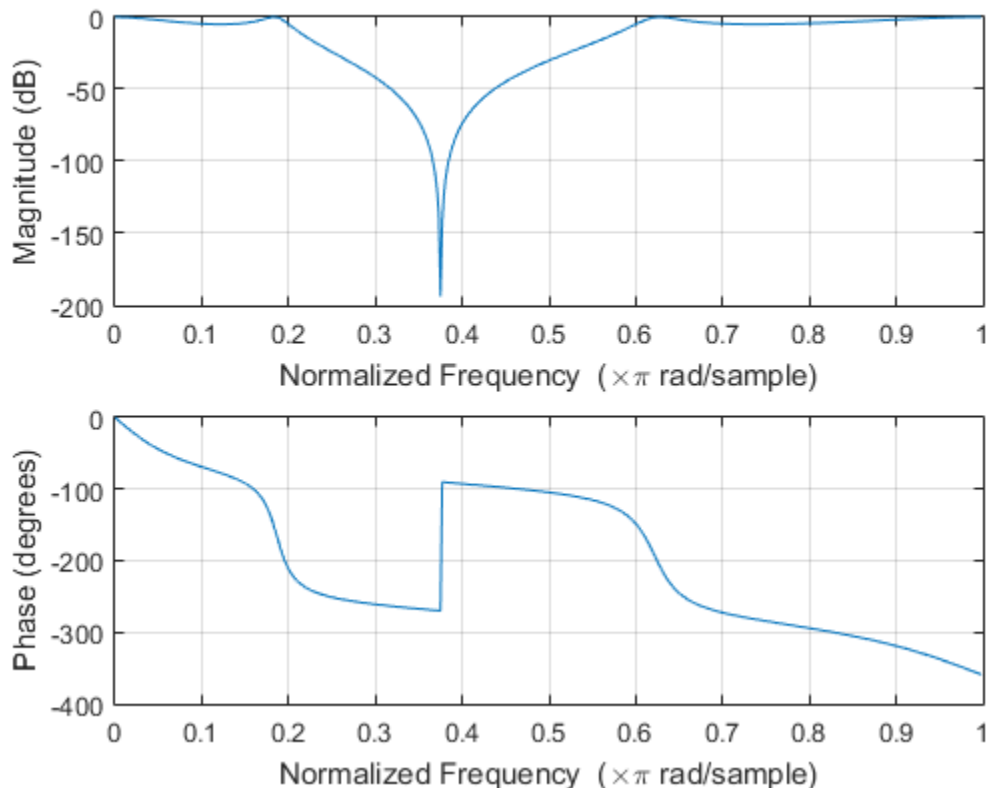
```
[b,a] = cheby1(6,10,0.6);  
freqz(b,a)  
dataIn = randn(1000,1);  
dataOut = filter(b,a,dataIn);
```



### Bandstop Chebyshev Type I Filter

Design a 6th-order Chebyshev Type I bandstop filter with normalized edge frequencies of  $0.2\pi$  and  $0.6\pi$  rad/sample and 5 dB of passband ripple. Plot its magnitude and phase responses. Use it to filter random data.

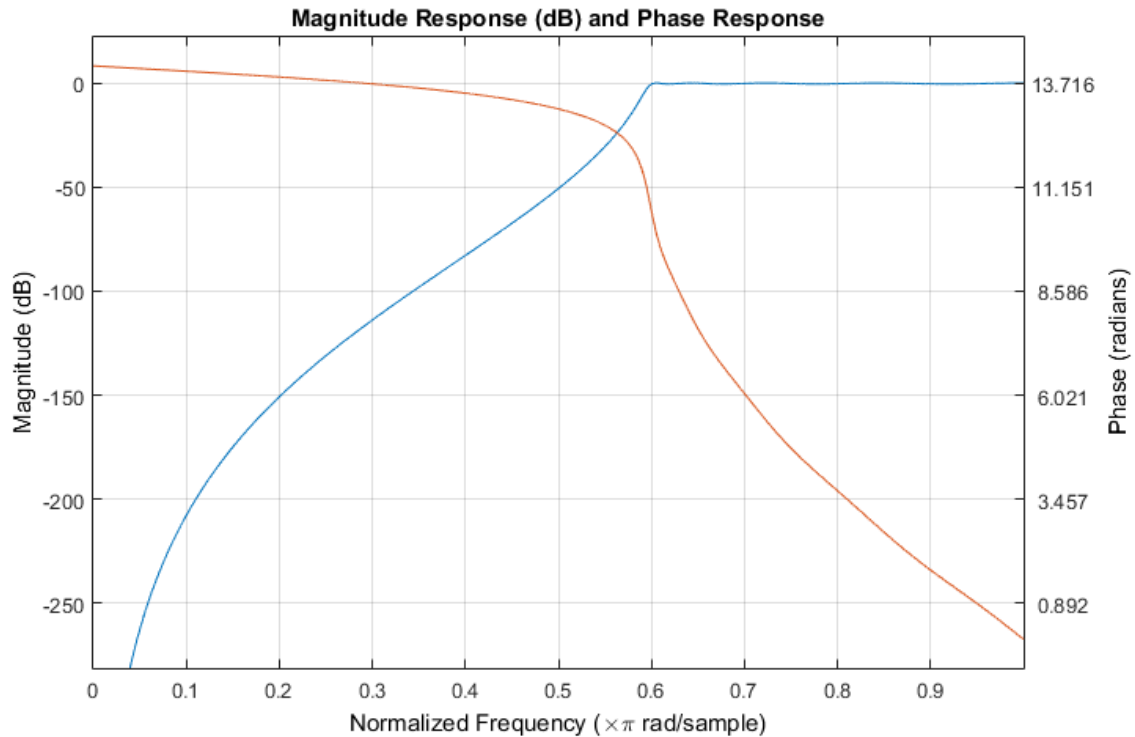
```
[b,a] = cheby1(3,5,[0.2 0.6], 'stop');  
freqz(b,a)  
dataIn = randn(1000,1);  
dataOut = filter(b,a,dataIn);
```



### Highpass Chebyshev Type I Filter

Design a 9th-order highpass Chebyshev Type I filter with 0.5 dB of passband ripple and a passband edge frequency of 300 Hz, which, for data sampled at 1000 Hz, corresponds to  $0.6\pi$  rad/sample. Plot the magnitude and phase responses. Convert the zeros, poles, and gain to second-order sections for use by `fvtool`.

```
[z,p,k] = cheby1(9,0.5,300/500,'high');
sos = zp2sos(z,p,k);
fvtool(sos,'Analysis','freq')
```



### Bandpass Chebyshev Type I Filter

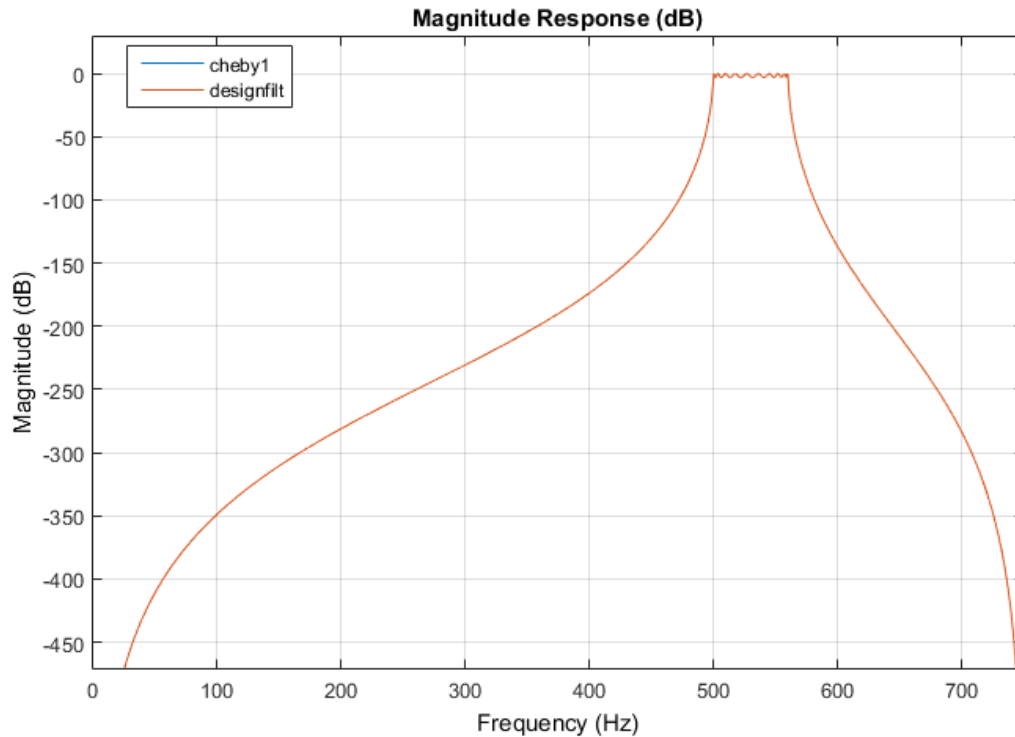
Design a 20th-order Chebyshev Type I bandpass filter with a lower passband frequency of 500 Hz and a higher passband frequency of 560 Hz. Specify a passband ripple of 3 dB and a sample rate of 1500 Hz. Use the state-space representation. Design an identical filter using `designfilt`.

```
[A,B,C,D] = cheby1(10,3,[500 560]/750);
d = designfilt('bandpassiir','FilterOrder',20, ...
    'PassbandFrequency1',500,'PassbandFrequency2',560, ...
    'PassbandRipple',3,'SampleRate',1500);
```

Convert the state-space representation to second-order sections. Visualize the frequency responses using `fvtool`.

```
sos = ss2sos(A,B,C,D);
```

```
fvt = fvtool(sos,d,'Fs',1500);
legend(fvt,'cheby1','designfilt')
```



### Comparison of Analog IIR Lowpass Filters

Design a 5th-order analog Butterworth lowpass filter with a cutoff frequency of 2 GHz. Multiply by  $2\pi$  to convert the frequency to radians per second. Compute the frequency response of the filter at 4096 points.

```
n = 5;
f = 2e9;

[zb,pb,kb] = butter(n,2*pi*f,'s');
[bb,ab] = zp2tf(zb,pb,kb);
[hb,wb] = freqs(bb,ab,4096);
```

Design a 5th-order Chebyshev Type I filter with the same edge frequency and 3 dB of passband ripple. Compute its frequency response.

```
[z1,p1,k1] = cheby1(n,3,2*pi*f,'s');
[b1,a1] = zp2tf(z1,p1,k1);
[h1,w1] = freqs(b1,a1,4096);
```

Design a 5th-order Chebyshev Type II filter with the same edge frequency and 30 dB of stopband attenuation. Compute its frequency response.

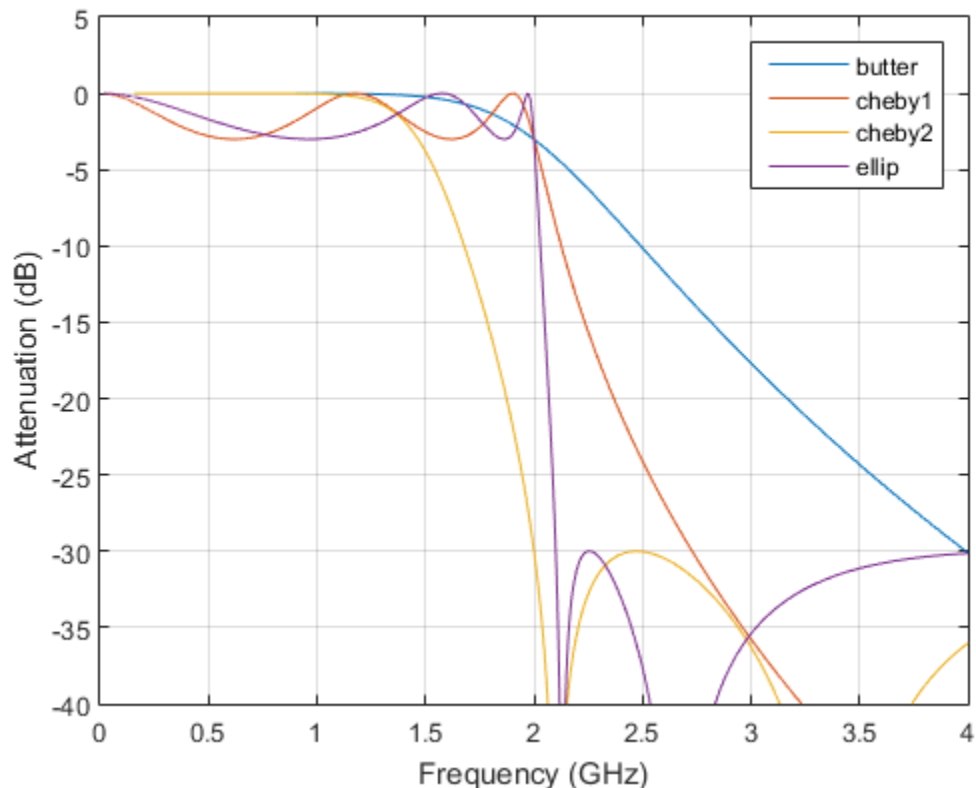
```
[z2,p2,k2] = cheby2(n,30,2*pi*f,'s');
[b2,a2] = zp2tf(z2,p2,k2);
[h2,w2] = freqs(b2,a2,4096);
```

Design a 5th-order elliptic filter with the same edge frequency, 3 dB of passband ripple, and 30 dB of stopband attenuation. Compute its frequency response.

```
[ze,pe,ke] = ellip(n,3,30,2*pi*f,'s');
[be,ae] = zp2tf(ze,pe,ke);
[he,we] = freqs(be,ae,4096);
```

Plot the attenuation in decibels. Express the frequency in gigahertz. Compare the filters.

```
plot(wb/(2e9*pi),mag2db(abs(hb)))
hold on
plot(w1/(2e9*pi),mag2db(abs(h1)))
plot(w2/(2e9*pi),mag2db(abs(h2)))
plot(we/(2e9*pi),mag2db(abs(he)))
axis([0 4 -40 5])
grid
xlabel('Frequency (GHz)')
ylabel('Attenuation (dB)')
legend('butter','cheby1','cheby2','ellip')
```



The Butterworth and Chebyshev Type II filters have flat passbands and wide transition bands. The Chebyshev Type I and elliptic filters roll off faster but have passband ripple. The frequency input to the Chebyshev Type II design function sets the beginning of the stopband rather than the end of the passband.

## Input Arguments

### **n** — Filter order

integer scalar

Filter order, specified as an integer scalar.

Data Types: `double`

**Rp — Peak-to-peak passband ripple**

positive scalar

Peak-to-peak passband ripple, specified as a positive scalar expressed in decibels.

If your specification,  $\ell$ , is in linear units, you can convert it to decibels using

$$R_p = 40 \log_{10}((1+\ell)/(1-\ell)).$$

Data Types: `double`

**Wp — Passband edge frequency**

scalar | two-element vector

Passband edge frequency, specified as a scalar or a two-element vector. The passband edge frequency is the frequency at which the magnitude response of the filter is  $-R_p$  decibels. Smaller values of passband ripple,  $R_p$ , result in wider transition bands.

- If  $W_p$  is a scalar, then `cheby1` designs a lowpass or highpass filter with edge frequency  $W_p$ .

If  $W_p$  is the two-element vector  $[w_1 \ w_2]$ , where  $w_1 < w_2$ , then `cheby1` designs a bandpass or bandstop filter with lower edge frequency  $w_1$  and higher edge frequency  $w_2$ .

- For digital filters, the passband edge frequencies must lie between 0 and 1, where 1 corresponds to the Nyquist rate—half the sample rate or  $\pi$  rad/sample.

For analog filters, the passband edge frequencies must be expressed in radians per second and can take on any positive value.

Data Types: `double`

**f type — Filter type**

'low' | 'bandpass' | 'high' | 'stop'

Filter type, specified as a string.

- 'low' specifies a lowpass filter with passband edge frequency  $W_p$ . 'low' is the default for scalar  $W_p$ .
- 'high' specifies a highpass filter with passband edge frequency  $W_p$ .



- 'bandpass' specifies a bandpass filter of order  $2n$  if  $W_p$  is a two-element vector. 'bandpass' is the default when  $W_p$  has two elements.
- 'stop' specifies a bandstop filter of order  $2n$  if  $W_p$  is a two-element vector.

Data Types: char

## Output Arguments

### **b, a** — Transfer function coefficients

row vectors

Transfer function coefficients of the filter, returned as row vectors of length  $n + 1$  for lowpass and highpass filters and  $2n + 1$  for bandpass and bandstop filters.

- For digital filters, the transfer function is expressed in terms of **b** and **a** as

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

- For analog filters, the transfer function is expressed in terms of **b** and **a** as

$$H(s) = \frac{B(s)}{A(s)} = \frac{b(1)s^n + b(2)s^{n-1} + \dots + b(n+1)}{a(1)s^n + a(2)s^{n-1} + \dots + a(n+1)}.$$

Data Types: double

### **z, p, k** — Zeros, poles, and gain

column vectors, scalar

Zeros, poles, and gain of the filter, returned as two column vectors of length  $n$  ( $2n$  for bandpass and bandstop designs) and a scalar.

- For digital filters, the transfer function is expressed in terms of **z**, **p**, and **k** as

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

- For analog filters, the transfer function is expressed in terms of **z**, **p**, and **k** as

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

Data Types: double

### **A, B, C, D** — State-space matrices matrices

State-space representation of the filter, returned as matrices. If  $m = n$  for lowpass and highpass designs and  $m = 2n$  for bandpass and bandstop filters, then **A** is  $m \times m$ , **B** is  $m \times 1$ , **C** is  $1 \times m$ , and **D** is  $1 \times 1$ .

- For digital filters, the state-space matrices relate the state vector  $x$ , the input  $u$ , and the output  $y$  through

$$\begin{aligned}x(k+1) &= A x(k) + B u(k) \\ y(k) &= C x(k) + D u(k).\end{aligned}$$

- For analog filters, the state-space matrices relate the state vector  $x$ , the input  $u$ , and the output  $y$  through

$$\begin{aligned}\dot{x} &= A x + B u \\ y &= C x + D u.\end{aligned}$$

Data Types: double

## More About

### Limitations

#### Numerical Instability of Transfer Function Syntax

In general, 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`. If you design the filter using the `[b, a]` syntax, you might encounter numerical problems. These problems are due to round-off errors and can occur for  $n$  as low as 4. The following example illustrates this limitation.

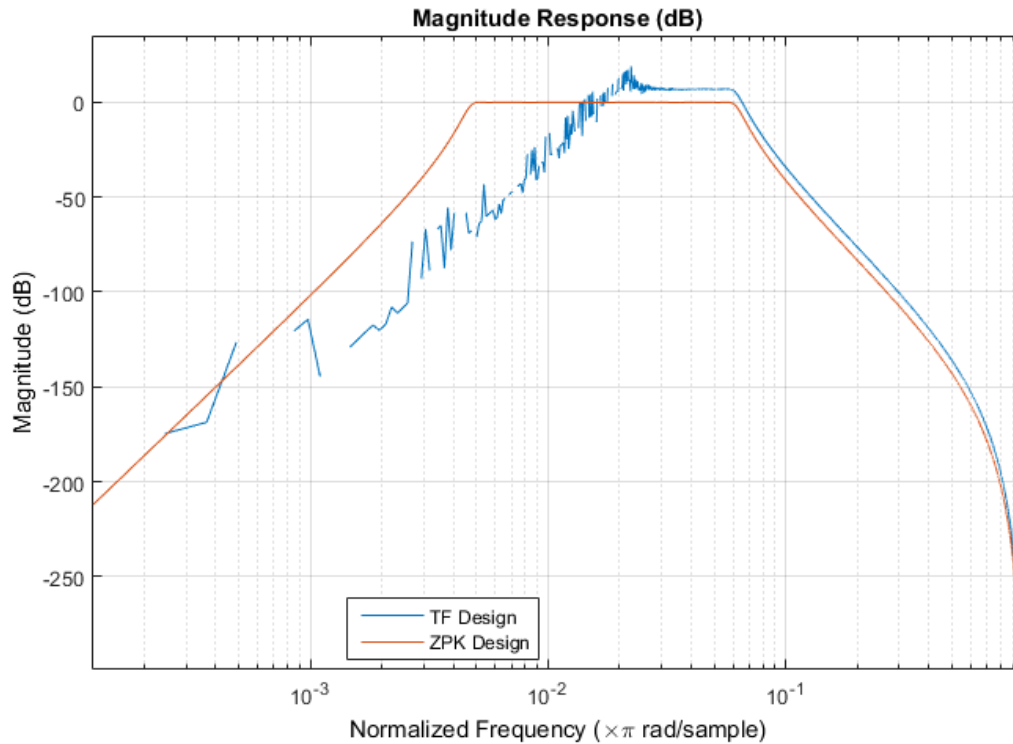
```
n = 6;
```

```
Rp = 0.1;
Wn = [2.5e6 29e6]/500e6;
ftype = 'bandpass';

% Transfer function design
[b,a] = cheby1(n,Rp,Wn,ftype);      % This filter is unstable

% Zero-pole-gain design
[z,p,k] = cheby1(n,Rp,Wn,ftype);
sos = zp2sos(z,p,k);

% Plot and compare the results
hfvt = fvtool(b,a,sos,'FrequencyScale','log');
legend(hfvt,'TF Design','ZPK Design')
```



## Algorithms

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.

`cheby1` uses a five-step algorithm:

- 1 It finds the lowpass analog prototype poles, zeros, and gain using the function `cheb1ap`.
- 2 It converts the poles, zeros, and gain into state-space form.
- 3 If required, it uses a state-space transformation to convert the lowpass filter to a highpass, bandpass, or bandstop filter with the desired frequency constraints.
- 4 For digital filter design, it uses `bilinear` to convert the analog filter into a digital filter through a bilinear transformation with frequency prewarping. Careful

frequency adjustment enables the analog filters and the digital filters to have the same frequency response magnitude at  $\omega_p$  or  $\omega_1$  and  $\omega_2$ .

- 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 | designfilt | ellip | filter  
| sosfilt

## cheby2

Chebyshev Type II filter design

### Syntax

```
[b,a] = cheby2(n,Rs,Ws)
[b,a] = cheby2(n,Rs,Ws,ftype)
```

```
[z,p,k] = cheby2(____)
[A,B,C,D] = cheby2(____)
```

```
[____] = cheby2(____,'s')
```

### Description

`[b,a] = cheby2(n,Rs,Ws)` returns the transfer function coefficients of an  $n$ th-order lowpass digital Chebyshev Type II filter with normalized stopband edge frequency  $W_s$  and  $R_s$  decibels of stopband attenuation down from the peak passband value.

`[b,a] = cheby2(n,Rs,Ws,ftype)` designs a lowpass, highpass, bandpass, or bandstop Chebyshev Type II filter, depending on the value of `ftype` and the number of elements of  $W_s$ . The resulting bandpass and bandstop designs are of order  $2n$ .

---

**Note:** See “Limitations” on page 1-162 for information about numerical issues that affect forming the transfer function.

---

`[z,p,k] = cheby2(____)` designs a lowpass, highpass, bandpass, or bandstop digital Chebyshev Type II filter and returns its zeros, poles, and gain. This syntax can include any of the input arguments in previous syntaxes.

`[A,B,C,D] = cheby2(____)` designs a lowpass, highpass, bandpass, or bandstop digital Chebyshev Type II filter and returns the matrices that specify its state-space representation.

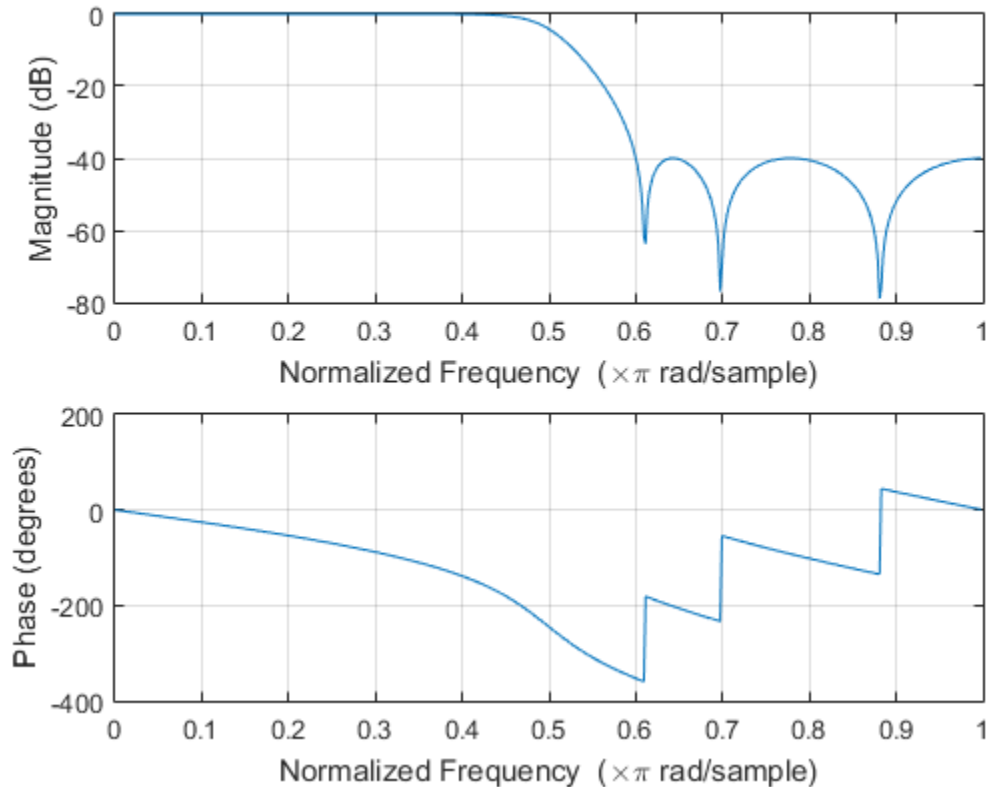
[ \_\_\_\_ ] = cheby2( \_\_\_\_, 's' ) designs a lowpass, highpass, bandpass, or bandstop analog Chebyshev Type II filter with stopband edge angular frequency  $W_s$  and  $R_s$  decibels of stopband attenuation.

## Examples

### Lowpass Chebyshev Type II Transfer Function

Design a 6th-order lowpass Chebyshev Type II filter with 40 dB of stopband attenuation and a stopband edge frequency of 300 Hz, which, for data sampled at 1000 Hz, corresponds to  $0.6\pi$  rad/sample. Plot its magnitude and phase responses. Use it to filter a 1000-sample random signal.

```
[b,a] = cheby2(6,40,0.6);  
freqz(b,a)  
dataIn = randn(1000,1);  
dataOut = filter(b,a,dataIn);
```

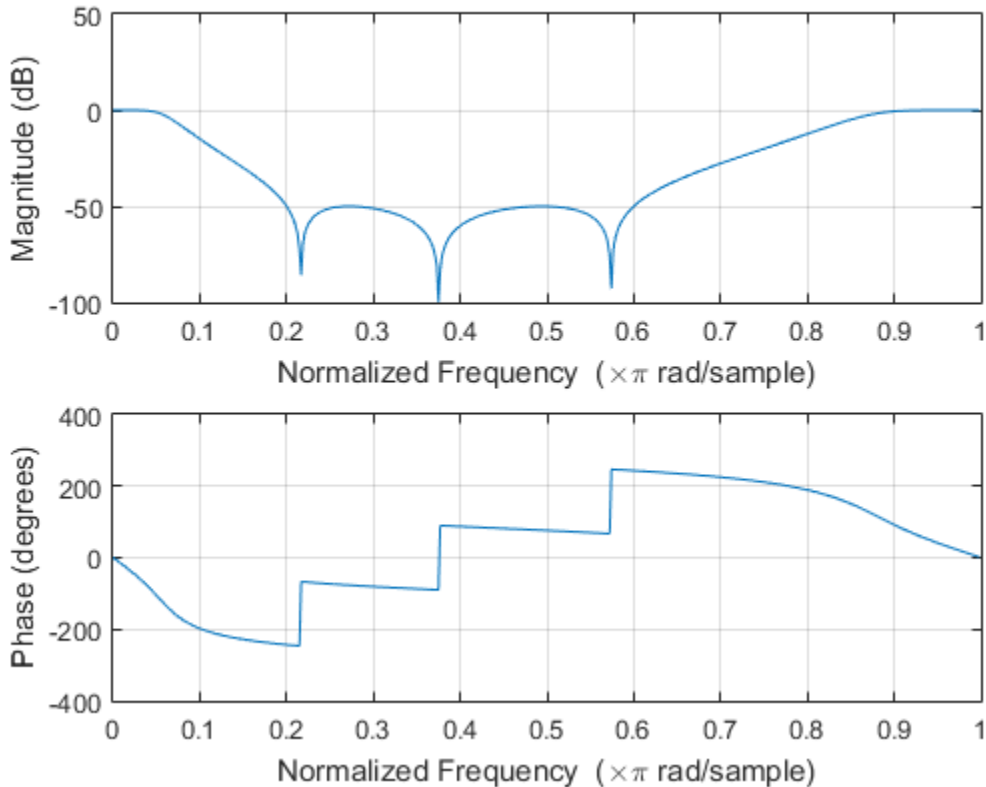


### Bandstop Chebyshev Type II Filter

Design a 6th-order Chebyshev Type II bandstop filter with normalized edge frequencies of  $0.2\pi$  and  $0.6\pi$  rad/sample and 50 dB of stopband attenuation. Plot its magnitude and phase responses. Use it to filter random data.

```
[b,a] = cheby2(3,50,[0.2 0.6], 'stop');  
freqz(b,a)  
dataIn = randn(1000,1);  
dataOut = filter(b,a,dataIn);
```

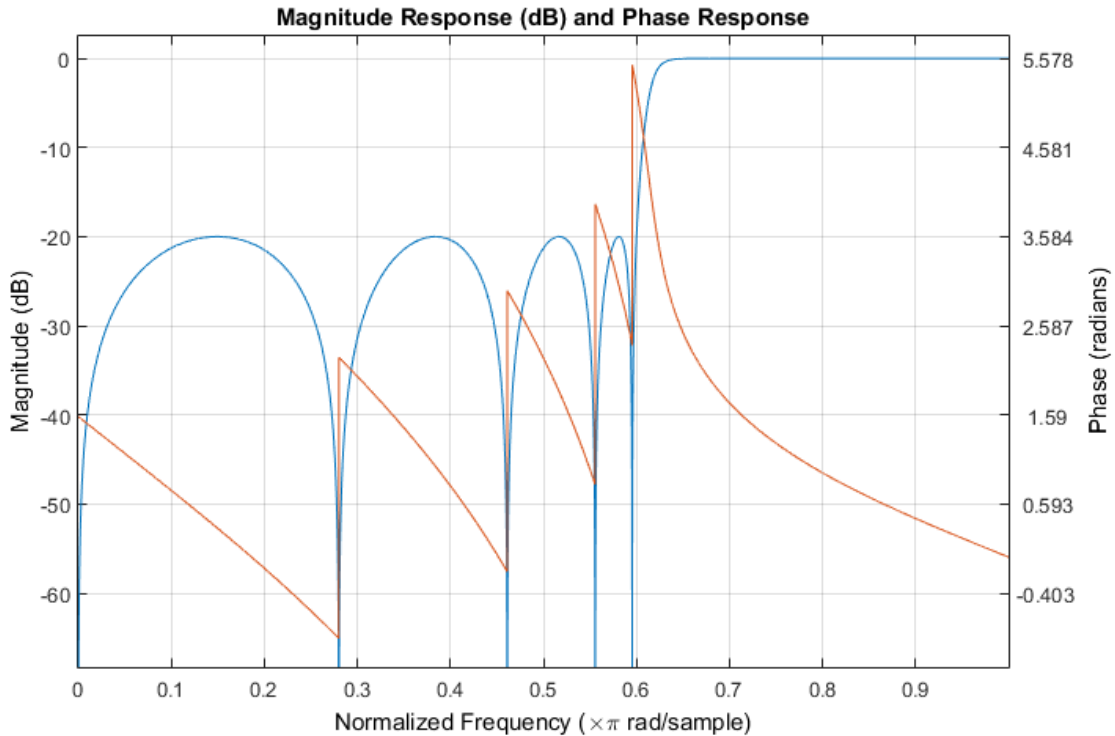




### Highpass Chebyshev Type II Filter

Design a 9th-order highpass Chebyshev Type II filter with 20 dB of stopband attenuation and a stopband edge frequency of 300 Hz, which, for data sampled at 1000 Hz, corresponds to  $0.6\pi$  rad/sample. Plot the magnitude and phase responses. Convert the zeros, poles, and gain to second-order sections for use by `fvtool`.

```
[z,p,k] = cheby2(9,20,300/500, 'high');
sos = zp2sos(z,p,k);
fvtool(sos, 'Analysis', 'freq')
```



### Bandpass Chebyshev Type II Filter

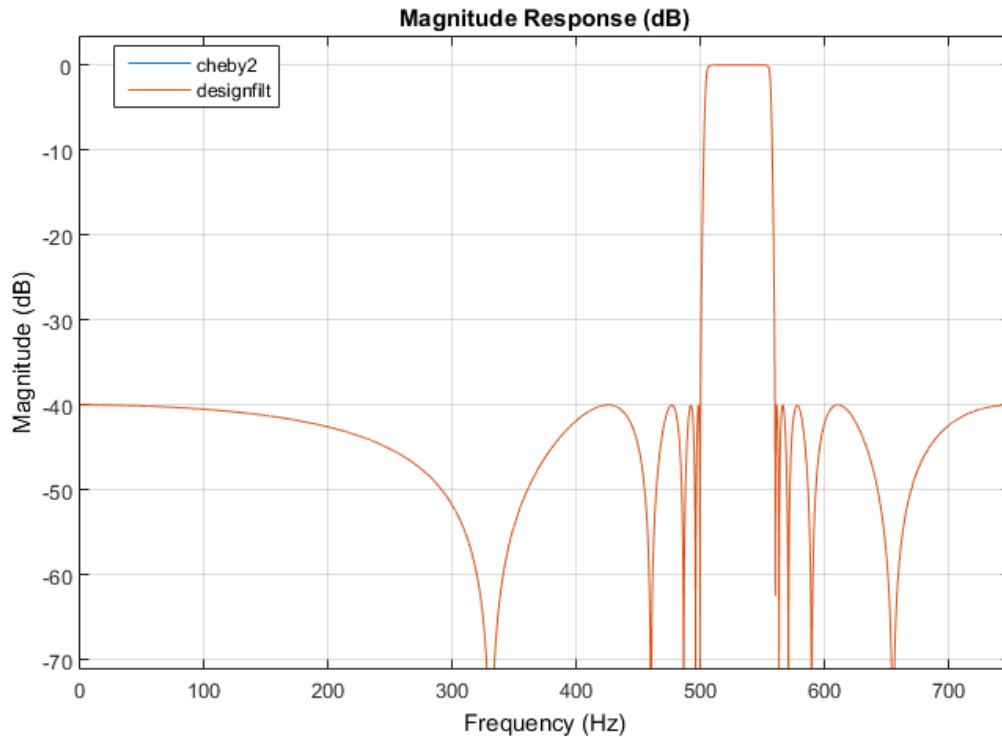
Design a 20th-order Chebyshev Type II bandpass filter with a lower stopband frequency of 500 Hz and a higher stopband frequency of 560 Hz. Specify a stopband attenuation of 40 dB and a sample rate of 1500 Hz. Use the state-space representation. Design an identical filter using `designfilt`.

```
[A,B,C,D] = cheby2(10,40,[500 560]/750);
d = designfilt('bandpassiir','FilterOrder',20, ...
    'StopbandFrequency1',500,'StopbandFrequency2',560, ...
    'StopbandAttenuation',40,'SampleRate',1500);
```

Convert the state-space representation to second-order sections. Visualize the frequency responses using `fvttool`.

```
sos = ss2sos(A,B,C,D);
```

```
fvt = fvtool(sos,d,'Fs',1500);
legend(fvt,'cheby2','designfilt')
```



### Comparison of Analog IIR Lowpass Filters

Design a 5th-order analog Butterworth lowpass filter with a cutoff frequency of 2 GHz. Multiply by  $2\pi$  to convert the frequency to radians per second. Compute the frequency response of the filter at 4096 points.

```
n = 5;
f = 2e9;

[zb,pb,kb] = butter(n,2*pi*f,'s');
[bb,ab] = zp2tf(zb,pb,kb);
[hb,wb] = freqs(bb,ab,4096);
```

Design a 5th-order Chebyshev Type I filter with the same edge frequency and 3 dB of passband ripple. Compute its frequency response.

```
[z1,p1,k1] = cheby1(n,3,2*pi*f,'s');
[b1,a1] = zp2tf(z1,p1,k1);
[h1,w1] = freqs(b1,a1,4096);
```

Design a 5th-order Chebyshev Type II filter with the same edge frequency and 30 dB of stopband attenuation. Compute its frequency response.

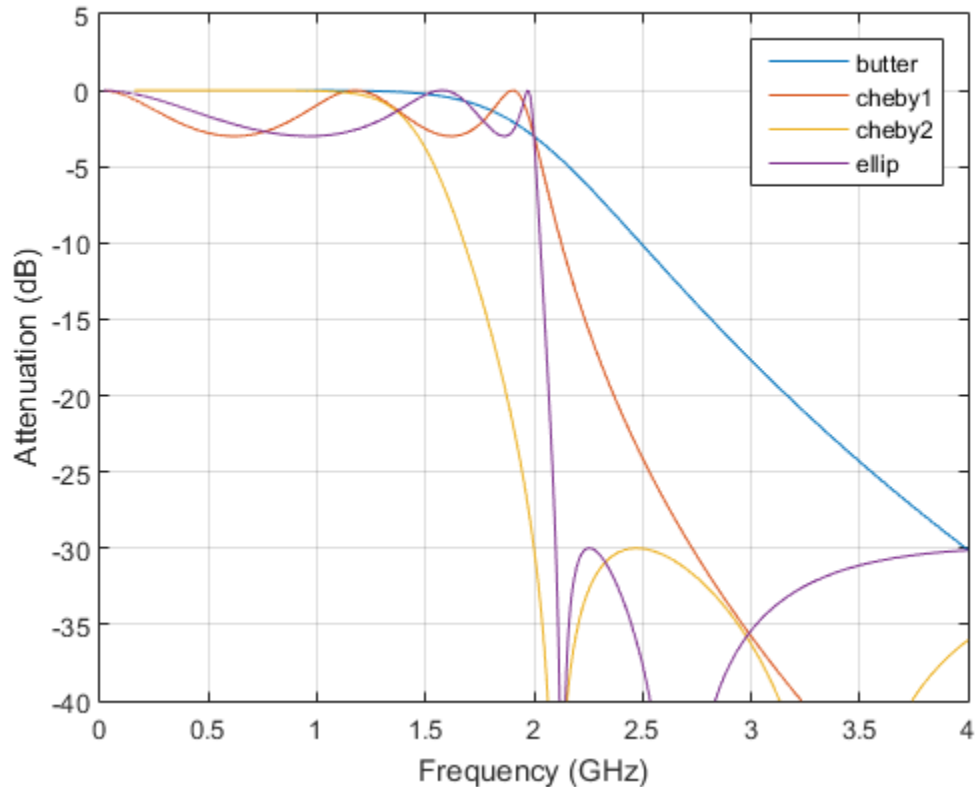
```
[z2,p2,k2] = cheby2(n,30,2*pi*f,'s');
[b2,a2] = zp2tf(z2,p2,k2);
[h2,w2] = freqs(b2,a2,4096);
```

Design a 5th-order elliptic filter with the same edge frequency, 3 dB of passband ripple, and 30 dB of stopband attenuation. Compute its frequency response.

```
[ze,pe,ke] = ellip(n,3,30,2*pi*f,'s');
[be,ae] = zp2tf(ze,pe,ke);
[he,we] = freqs(be,ae,4096);
```

Plot the attenuation in decibels. Express the frequency in gigahertz. Compare the filters.

```
plot(wb/(2e9*pi),mag2db(abs(hb)))
hold on
plot(w1/(2e9*pi),mag2db(abs(h1)))
plot(w2/(2e9*pi),mag2db(abs(h2)))
plot(we/(2e9*pi),mag2db(abs(he)))
axis([0 4 -40 5])
grid
xlabel('Frequency (GHz)')
ylabel('Attenuation (dB)')
legend('butter','cheby1','cheby2','ellip')
```



The Butterworth and Chebyshev Type II filters have flat passbands and wide transition bands. The Chebyshev Type I and elliptic filters roll off faster but have passband ripple. The frequency input to the Chebyshev Type II design function sets the beginning of the stopband rather than the end of the passband.

## Input Arguments

### **n** — Filter order

integer scalar

Filter order, specified as an integer scalar.

Data Types: double

**Rs — Stopband attenuation**

positive scalar

Stopband attenuation down from the peak passband value, specified as a positive scalar expressed in decibels.

If your specification,  $\ell$ , is in linear units, you can convert it to decibels using  $Rs = -20 \log_{10}\ell$ .

Data Types: double

**Ws — Stopband edge frequency**

scalar | two-element vector

Stopband edge frequency, specified as a scalar or a two-element vector. The stopband edge frequency is the frequency at which the magnitude response of the filter is  $-Rs$  decibels. Larger values of stopband attenuation,  $Rs$ , result in wider transition bands.

- If  $Ws$  is a scalar, then `cheby2` designs a lowpass or highpass filter with edge frequency  $Ws$ .

If  $Ws$  is the two-element vector  $[w1 \ w2]$ , where  $w1 < w2$ , then `cheby2` designs a bandpass or bandstop filter with lower edge frequency  $w1$  and higher edge frequency  $w2$ .

- For digital filters, the stopband edge frequencies must lie between 0 and 1, where 1 corresponds to the Nyquist rate—half the sample rate or  $\pi$  rad/sample.

For analog filters, the stopband edge frequencies must be expressed in radians per second and can take on any positive value.

Data Types: double

**f type — Filter type**

'low' | 'bandpass' | 'high' | 'stop'

Filter type, specified as a string.

- 'low' specifies a lowpass filter with stopband edge frequency  $Ws$ . 'low' is the default for scalar  $Ws$ .
- 'high' specifies a highpass filter with stopband edge frequency  $Ws$ .

- 'bandpass' specifies a bandpass filter of order 2n if Ws is a two-element vector. 'bandpass' is the default when Ws has two elements.
- 'stop' specifies a bandstop filter of order 2n if Ws is a two-element vector.

Data Types: char

## Output Arguments

### **b, a** — Transfer function coefficients

row vectors

Transfer function coefficients of the filter, returned as row vectors of length n + 1 for lowpass and highpass filters and 2n + 1 for bandpass and bandstop filters.

- For digital filters, the transfer function is expressed in terms of **b** and **a** as

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

- For analog filters, the transfer function is expressed in terms of **b** and **a** as

$$H(s) = \frac{B(s)}{A(s)} = \frac{b(1)s^n + b(2)s^{n-1} + \dots + b(n+1)}{a(1)s^n + a(2)s^{n-1} + \dots + a(n+1)}.$$

Data Types: double

### **z, p, k** — Zeros, poles, and gain

column vectors, scalar

Zeros, poles, and gain of the filter, returned as two column vectors of length n (2n for bandpass and bandstop designs) and a scalar.

- For digital filters, the transfer function is expressed in terms of **z**, **p**, and **k** as

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

- For analog filters, the transfer function is expressed in terms of **z**, **p**, and **k** as

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

Data Types: double

### **A, B, C, D — State-space matrices** matrices

State-space representation of the filter, returned as matrices. If  $m = n$  for lowpass and highpass designs and  $m = 2n$  for bandpass and bandstop filters, then **A** is  $m \times m$ , **B** is  $m \times 1$ , **C** is  $1 \times m$ , and **D** is  $1 \times 1$ .

- For digital filters, the state-space matrices relate the state vector  $x$ , the input  $u$ , and the output  $y$  through

$$\begin{aligned}x(k+1) &= A x(k) + B u(k) \\ y(k) &= C x(k) + D u(k).\end{aligned}$$

- For analog filters, the state-space matrices relate the state vector  $x$ , the input  $u$ , and the output  $y$  through

$$\begin{aligned}\dot{x} &= A x + B u \\ y &= C x + D u.\end{aligned}$$

Data Types: double

## More About

### Limitations

#### Numerical Instability of Transfer Function Syntax

In general, 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`. If you design the filter using the `[b, a]` syntax, you might encounter numerical problems. These problems are due to round-off errors and can occur for  $n$  as low as 4. The following example illustrates this limitation.

```
n = 6;
```

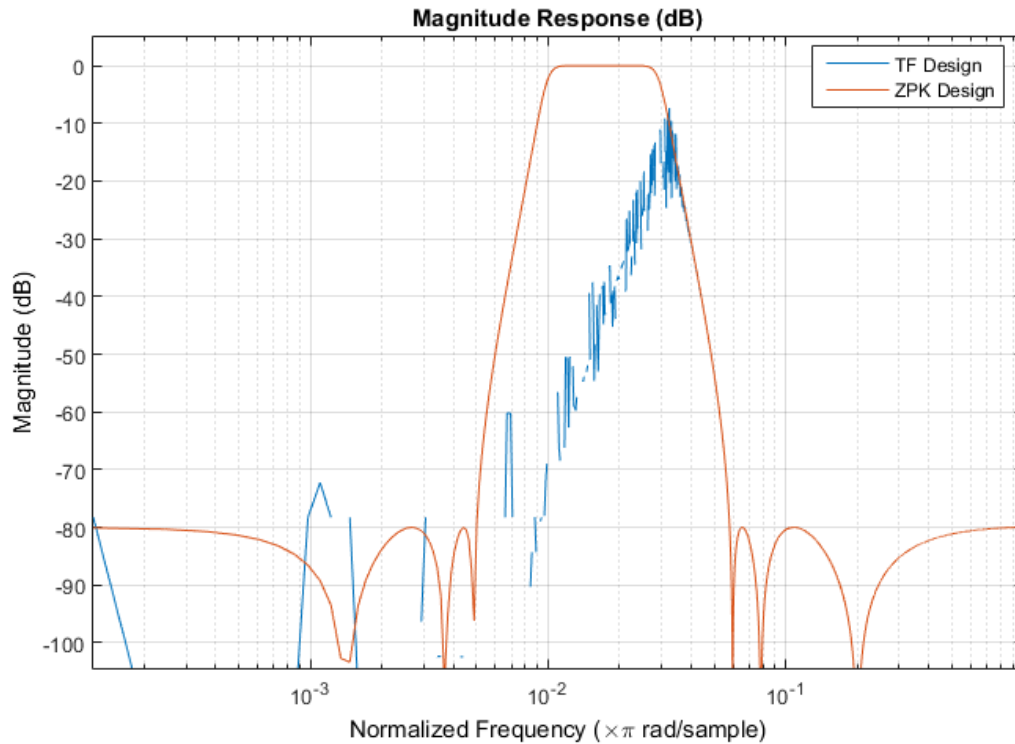


```
Rs = 80;
Wn = [2.5e6 29e6]/500e6;
ftype = 'bandpass';

% Transfer function design
[b,a] = cheby2(n,Rs,Wn,ftype);      % This filter is unstable

% Zero-pole-gain design
[z,p,k] = cheby2(n,Rs,Wn,ftype);
sos = zp2sos(z,p,k);

% Plot and compare the results
hfvt = fvtool(b,a,sos,'FrequencyScale','log');
legend(hfvt,'TF Design','ZPK Design')
```



## Algorithms

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.

`cheby2` uses a five-step algorithm:

- 1 It finds the lowpass analog prototype poles, zeros, and gain using the function `cheb2ap`.
- 2 It converts poles, zeros, and gain into state-space form.
- 3 If required, it uses a state-space transformation to convert the lowpass filter into a bandpass, highpass, or bandstop filter with the desired frequency constraints.
- 4 For digital filter design, it uses `bilinear` to convert the analog filter into a digital filter through a bilinear transformation with frequency prewarping. Careful

frequency adjustment the analog filters and the digital filters to have the same frequency response magnitude at  $W_s$  or  $w_1$  and  $w_2$ .

- 5 It converts the state-space filter back to transfer function or zero-pole-gain form, as required.

### **See Also**

besself | butter | cheb1ord | cheb2ap | cheby1 | designfilt | ellip | filter  
| sosfilt

# chirp

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.  $\beta$  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

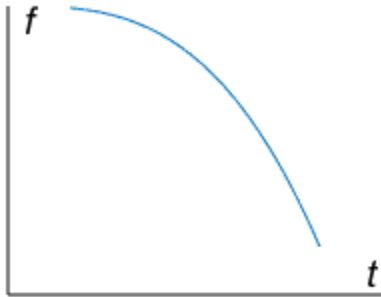
$$\beta = \left( \frac{f_1}{f_0} \right)^{\frac{1}{t_1}}$$

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.

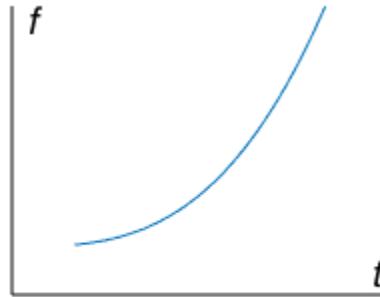
Each of the above methods can be entered as 'li', 'q', and 'lo', respectively.

`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

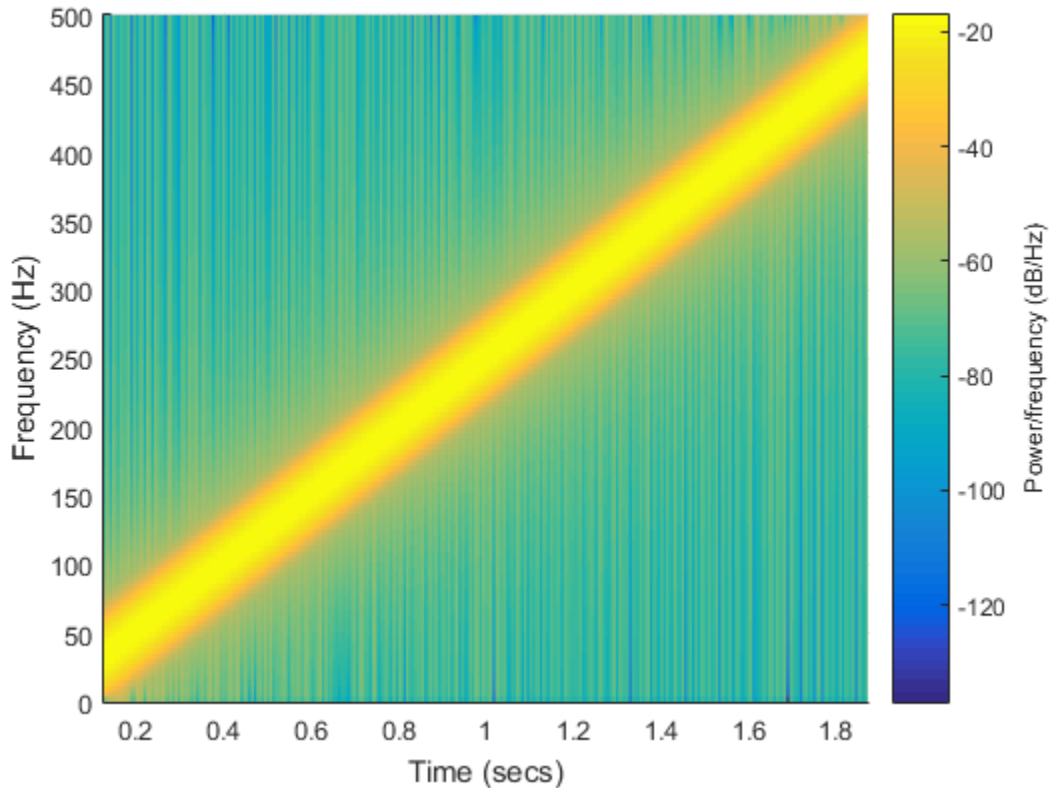
### Linear Chirp

Generate a chirp with linear instantaneous frequency deviation. The chirp is sampled at 1 kHz for 2 seconds. The instantaneous frequency is 0 at  $t = 0$  and crosses 250 Hz at  $t = 1$  second.

```
t = 0:1/1e3:2;  
y = chirp(t,0,1,250);
```

Compute and plot the spectrogram of the chirp. Specify 256 DFT points, a Hamming window of the same length, and 250 samples of overlap.

```
spectrogram(y,256,250,256,1e3,'yaxis')
```



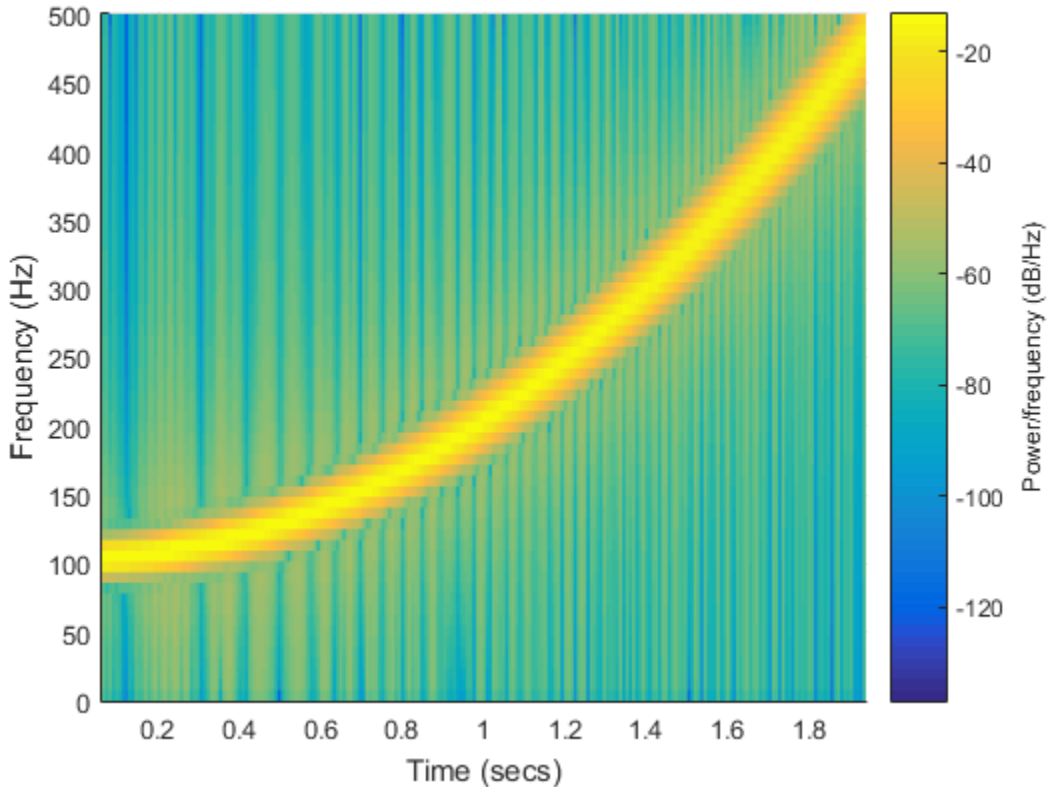
### Quadratic Chirp

Generate a chirp with quadratic instantaneous frequency deviation. The chirp is sampled at 1 kHz for 2 seconds. The instantaneous frequency is 100 Hz at  $t = 0$  and crosses 200 Hz at  $t = 1$  second.

```
t = 0:1/1e3:2;  
y = chirp(t,100,1,200,'quadratic');
```

Compute and plot the spectrogram of the chirp. Specify 128 DFT points, a Hamming window of the same length, and 120 samples of overlap.

```
spectrogram(y,128,120,128,1e3,'yaxis')
```



### Convex Quadratic Chirp

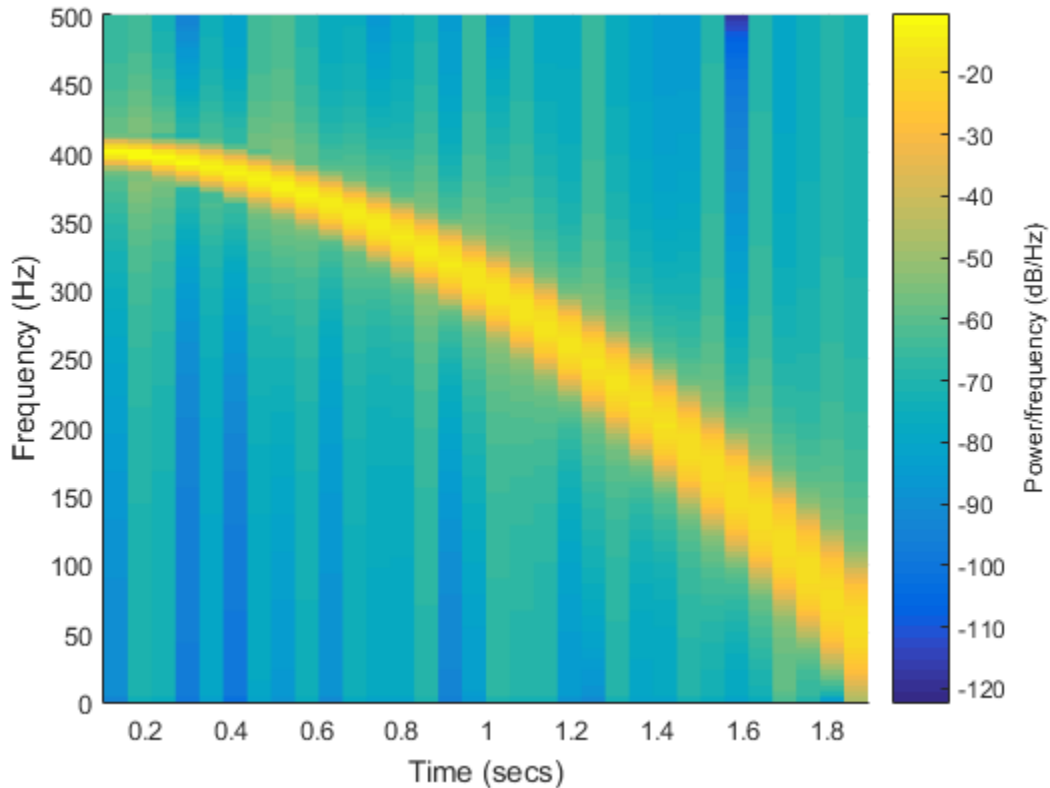
Generate a convex quadratic chirp sampled at 1 kHz for 2 seconds. The instantaneous frequency is 400 Hz at  $t = 0$  and crosses 300 Hz at  $t = 1$  second.

```
t = 0:1/1e3:2;  
fo = 400;  
f1 = 300;  
y = chirp(t,fo,1,f1,'quadratic',[],'convex');
```

Compute and plot the spectrogram of the chirp. Specify 256 DFT points, a Hamming window of the same length, and 200 samples of overlap.

```
spectrogram(y,256,200,256,1e3,'yaxis')
```





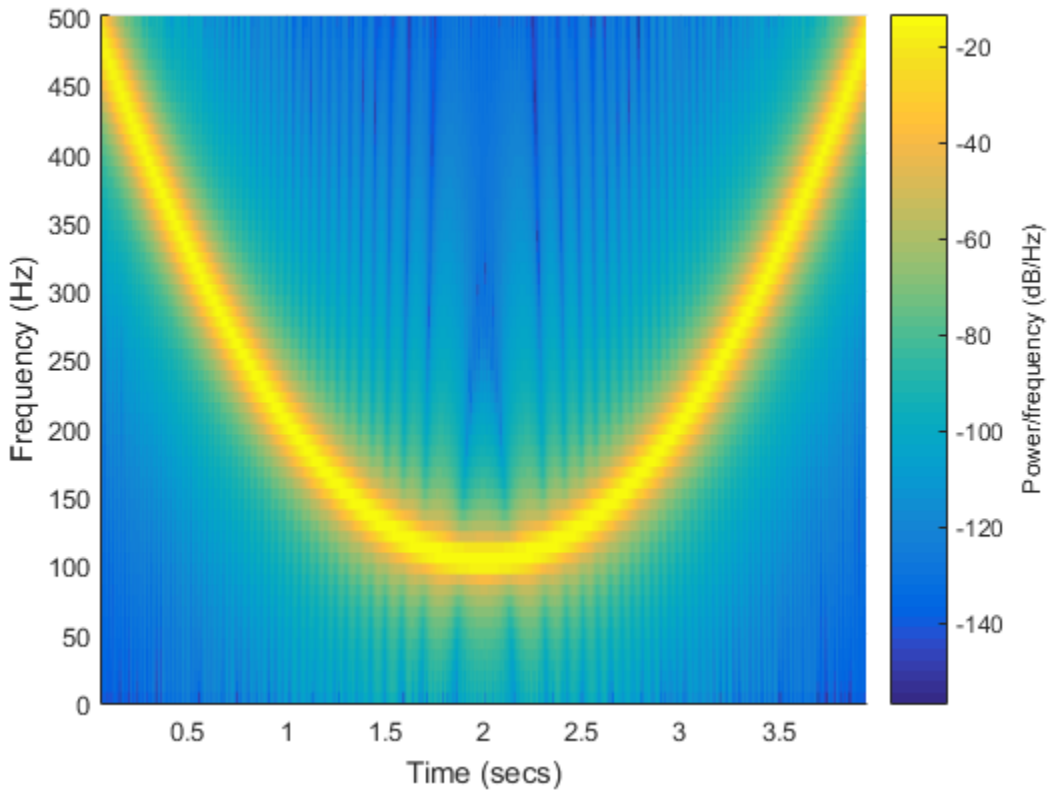
### Symmetric Concave Quadratic Chirp

Generate a concave quadratic chirp sampled at 1 kHz for 4 seconds. Specify the time vector so that the instantaneous frequency is symmetric about the halfway point of the sampling interval, with a minimum frequency of 100 Hz and a maximum frequency of 500 Hz.

```
t = -2:1/1e3:2;  
fo = 100;  
f1 = 200;  
y = chirp(t,fo,1,f1,'quadratic',[],'concave');
```

Compute and plot the spectrogram of the chirp. Specify 128 DFT points, a Hann window of the same length, and 120 samples of overlap. Note that the spectrogram function measures time starting at  $t = 0$ ;

```
spectrogram(y,hann(128),120,128,1e3,'yaxis')
```



### Logarithmic Chirp

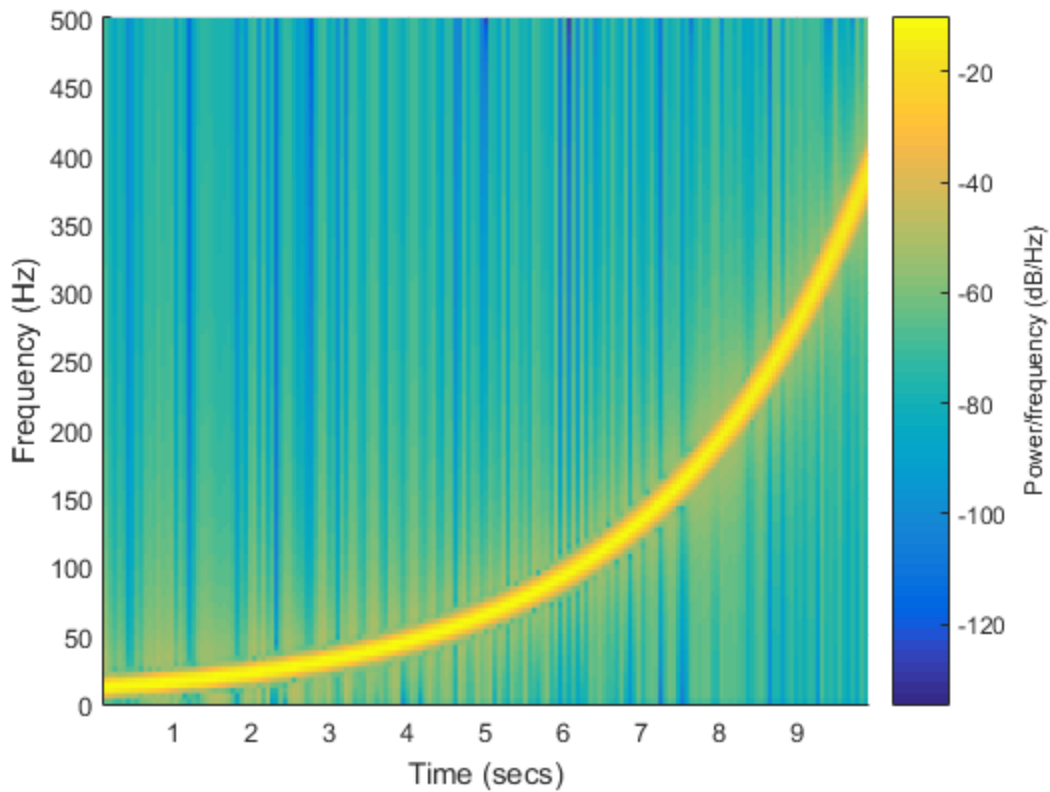
Generate a logarithmic chirp sampled at 1 kHz for 10 seconds. The instantaneous frequency is 10 Hz initially and 400 Hz at the end.

```
t = 0:1/1e3:10;  
fo = 10;  
f1 = 400;
```

```
y = chirp(t,fo,10,f1,'logarithmic');
```

Compute and plot the spectrogram of the chirp. Specify 256 DFT points, a Hamming window of the same length, and 200 samples of overlap.

```
spectrogram(y,256,200,256,1e3,'yaxis')
```



### See Also

cos | diric | gauspuls | pulstran | rectpuls | sawtooth | sin | sinc | square  
| tripuls

## convmtx

Convolution matrix

### Syntax

```
A = convmtx(h,n)
```

### Description

`A = convmtx(h,n)` returns the convolution matrix, **A**, such that the product of **A** and a vector, **x**, is the convolution of **h** and **x**.

- If **h** is a column vector of length **m**, **A** is  $(m+n-1)$ -by-**n** and the product of **A** and a column vector, **x**, of length **n** is the convolution of **h** and **x**.
- If **h** is a row vector of length **m**, **A** is **n**-by- $(m+n-1)$  and the product of a row vector, **x**, of length **n** with **A** is the convolution of **h** and **x**.

`convmtx` handles edge conditions by zero padding.

## Examples

### Efficient Computation of Convolution

It is generally more efficient to compute a convolution using `conv` when the signals are vectors. For multichannel signals, `convmtx` might be more efficient.

Compute the convolution of two random vectors, **a** and **b**, using both `conv` and `convmtx`. The signals have 1000 samples each. Compare the time spent by the two functions. Eliminate random fluctuations by repeating the calculation 30 times and averaging.

```
Nt = 30;  
Na = 1000;  
Nb = 1000;
```

```
tcnv = 0;  
tmtx = 0;
```

```

for kj = 1:Nt
    a = randn(Na,1);
    b = randn(Nb,1);

    tic
    n = conv(a,b);
    tcnv = tcnv+toc;

    tic
    c = convmtx(b,Na);
    d = c*a;
    tmtx = tmtx+toc;
end

t1col = [tcnv tmtx]/Nt
t1rat = tcnv\tmtx

```

```
t1col =
```

```
    0.0008    0.0603
```

```
t1rat =
```

```
    72.8542
```

`conv` is about two orders of magnitude more efficient.

Repeat the exercise for the case where `a` is a multichannel signal with 1000 channels. Optimize `conv`'s performance by preallocating.

```
Nchan = 1000;
```

```
tcnv = 0;
```

```
tmtx = 0;
```

```
n = zeros(Na+Nb-1,Nchan);
```

```

for kj = 1:Nt
    a = randn(Na,Nchan);
    b = randn(Nb,1);

```

```
tic
for k = 1:Nchan
    n(:,k) = conv(a(:,k),b);
end
tcnv = tcnv+toc;

tic
c = convmtx(b,Na);
d = c*a;
tmtx = tmtx+toc;
end

tmcol = [tcnv tmtx]/Nt
tmrat = tcnv/tmtx
```

```
tmcol =

    0.4676    0.0993

tmrat =

    4.7110
```

convmtx is about 3 times as efficient as conv.

## More About

### Algorithms

convmtx uses the function `toeplitz` to generate the convolution matrix.

### See Also

`conv` | `convn` | `conv2` | `corrmtx` | `dftmtx`

## corrmtx

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  $x$ .  $m$  must be a positive integer strictly smaller than the length of the input  $x$ .

`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

### Modified Data and Autocorrelation Matrices

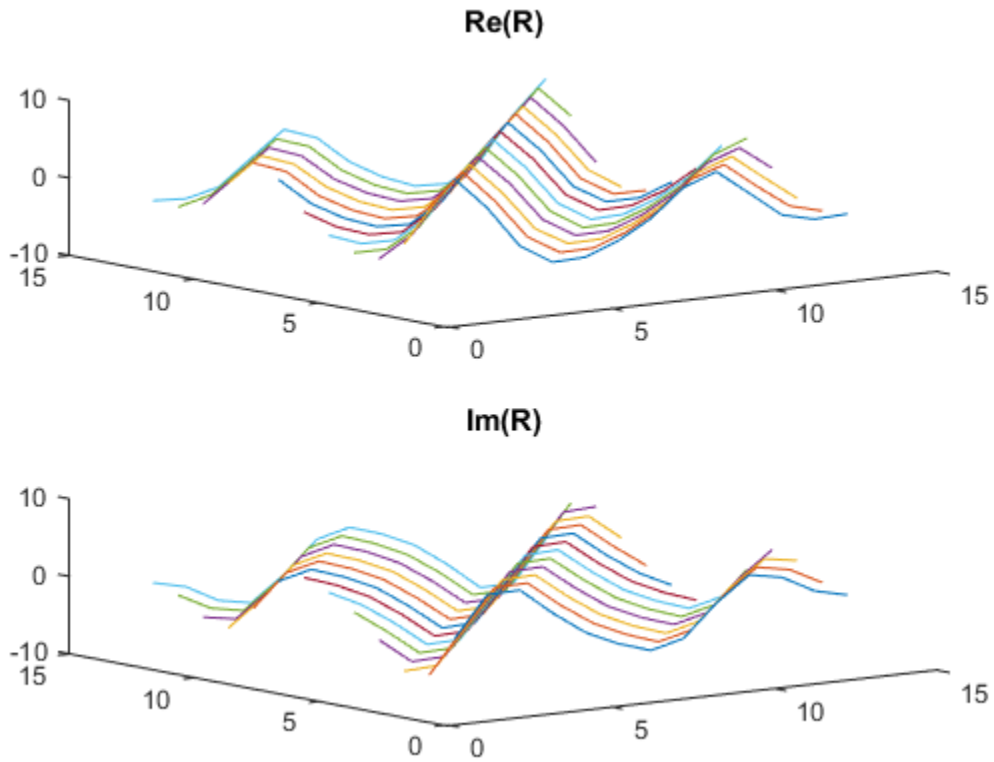
Generate a signal composed of three complex exponentials embedded in white Gaussian noise. Compute the data and autocorrelation matrices using the 'modified' method.

```
n = 0:99;  
s = exp(i*pi/2*n)+2*exp(i*pi/4*n)+exp(i*pi/3*n)+randn(1,100);  
m = 12;  
[X,R] = corrmtx(s,m,'modified');
```

Plot the real and imaginary parts of the autocorrelation matrix.

```
[A,B] = ndgrid(1:m+1);  
subplot(2,1,1)  
plot3(A,B,real(R))  
title('Re(R)')  
subplot(2,1,2)  
plot3(A,B,imag(R))  
title('Im(R)')
```





## More About

### Algorithms

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.

$$X = \begin{bmatrix} x(1) & \cdots & 0 \\ \vdots & \ddots & \vdots \\ \hline x(m+1) & \cdots & x(1) \\ \vdots & \ddots & \vdots \\ x(n-m) & \cdots & x(m+1) \\ \vdots & \ddots & \vdots \\ \hline x(n) & \cdots & x(n-m) \\ \vdots & \ddots & \vdots \\ \hline 0 & \cdots & x(n) \end{bmatrix}$$

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  $X_{\text{mod}}$  shown below.

$$X_{\text{mod}} = \begin{bmatrix} x(m+1) & \cdots & x(1) \\ \vdots & \ddots & \vdots \\ x(n-m) & \cdots & x(m+1) \\ \vdots & \ddots & \vdots \\ x(n) & \cdots & x(n-m) \\ x^*(1) & \cdots & x^*(m+1) \\ \vdots & \ddots & \vdots \\ x^*(m+1) & \cdots & x^*(n-m) \\ \vdots & \ddots & \vdots \\ x^*(n-m) & \cdots & x^*(n) \end{bmatrix}$$

## References

- [1] Marple, S. Lawrence. *Digital Spectral Analysis*. Englewood Cliffs, NJ: Prentice-Hall, 1987.

## See Also

peig | pmusic | rooteig | xcorr | rootmusic

## cpsd

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 two discrete-time signals, `x` and `y`, using Welch's averaged, modified periodogram method of spectral estimation.

The input signals may be either vectors or two-dimensional matrices. If both are vectors, they must have the same length. If both are matrices, they must have the same size, and `cpsd` operates columnwise: `Pxy(:,n) = cpsd(x(:,n),y(:,n))`. If one is a matrix and the other is a vector, then the vector is converted to a column vector and internally expanded so both inputs have the same number of columns.

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   |
|-------------------|--|---|
| <code>nfft</code> | FFT length which determines the frequencies at which the PSD is estimated<br><br>For real <code>x</code> and <code>y</code> , the length of <code>Pxy</code> is $(nfft/2+1)$ if <code>nfft</code> is even or $(nfft+1)/2$ if <code>nfft</code> is odd. For | Maximum of 256 or the next power of 2 greater than the length of each section of <code>x</code> or <code>y</code> |

| Parameter             | Description  | Default Value   |
|-----------------------|--|---|
|                       | complex $x$ or $y$ , the length of $P_{xy}$ is $nfft$ .<br><br>If $nfft$ is greater than the signal length, the data is zero-padded.<br>If $nfft$ is less than the signal length, the segment is wrapped so that the length is equal to $nfft$ . |   |
| <code>fs</code>       | Sampling frequency   | 1   |
| <code>window</code>   | Windowing function and number of samples to use for each section   | Periodic Hamming window of length to obtain eight equal sections of $x$ and $y$ |
| <code>noverlap</code> | 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.  $x$  and  $y$  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  $P_{xy}$  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.

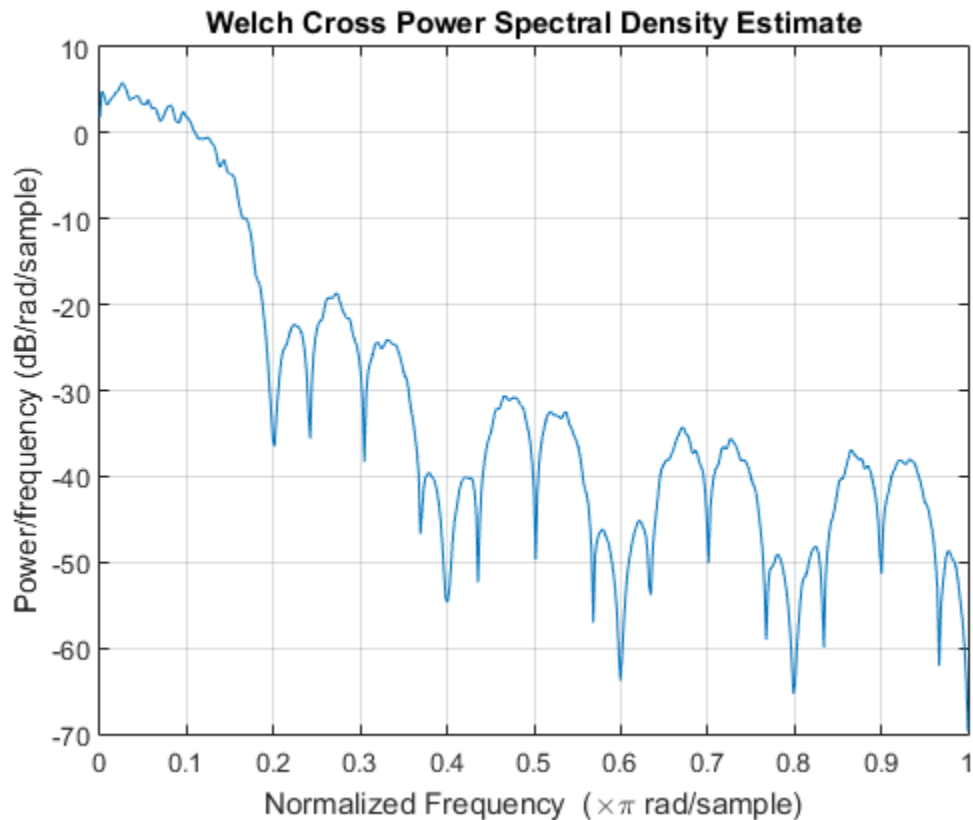
`cpsd(...)` plots the CPSD versus frequency in the current figure window.

## Examples

### CPSD of Colored Noise Signals

Generate two colored noise signals and plot their CPSD. Specify a length-1024 FFT and a 500-point triangular window with no overlap.

```
rng default
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)
```



## More About

### Cross Power Spectral Density

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.

### Algorithms

cpsd uses Welch's averaged periodogram method. See the references listed below.

## References

- [1] Rabiner, Lawrence R., and B. Gold. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975, pp.414–419.
- [2] Welch, Peter D. "The Use of the Fast Fourier Transform for the Estimation of Power Spectra: A Method Based on Time Averaging Over Short, Modified Periodograms." *IEEE Transactions on Audio and Electroacoustics*, Vol. AU-15, June 1967, pp.70–73.
- [3] Oppenheim, Alan V., Ronald W. Schaffer, and John R. Buck. *Discrete-Time Signal Processing*. 2nd Ed. Upper Saddle River, NJ: Prentice Hall, 1999.

### See Also

dspdata | pburg | pcov | peig | mscohere | periodogram | pmcov | pmtm | pmusic | pwelch | pyulear | spectrum | tfestimate



## czt

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

$$z = a \cdot (w.^{(0:m-1)})$$

`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

### CZT of a Random Vector

Create a random vector, `x`, of length 1013. Compute its DFT using `czt`.

```
rng default
x = randn(1013,1);
```

```
y = czt(x);
```

### Narrow-Band Section of a Frequency Response

Use `czt` to zoom in on a narrow-band section of a filter's frequency response.

Design a 30th-order lowpass FIR filter using the window method. Specify a sample rate of 1 kHz and a cutoff frequency of 125 Hz. Use a rectangular window. Find the transfer function of the filter.

```
fs = 1000;
d = designfilt('lowpassfir','FilterOrder',30,'CutoffFrequency',125, ...
    'DesignMethod','window','Window',@rectwin,'SampleRate',fs);
h = tf(d);
```

Compute the DFT and the CZT of the filter. Restrict the frequency range of the CZT to the band between 100 and 150 Hz. Generate 1024 samples in each case.

```
m = 1024;
y = fft(h,m);

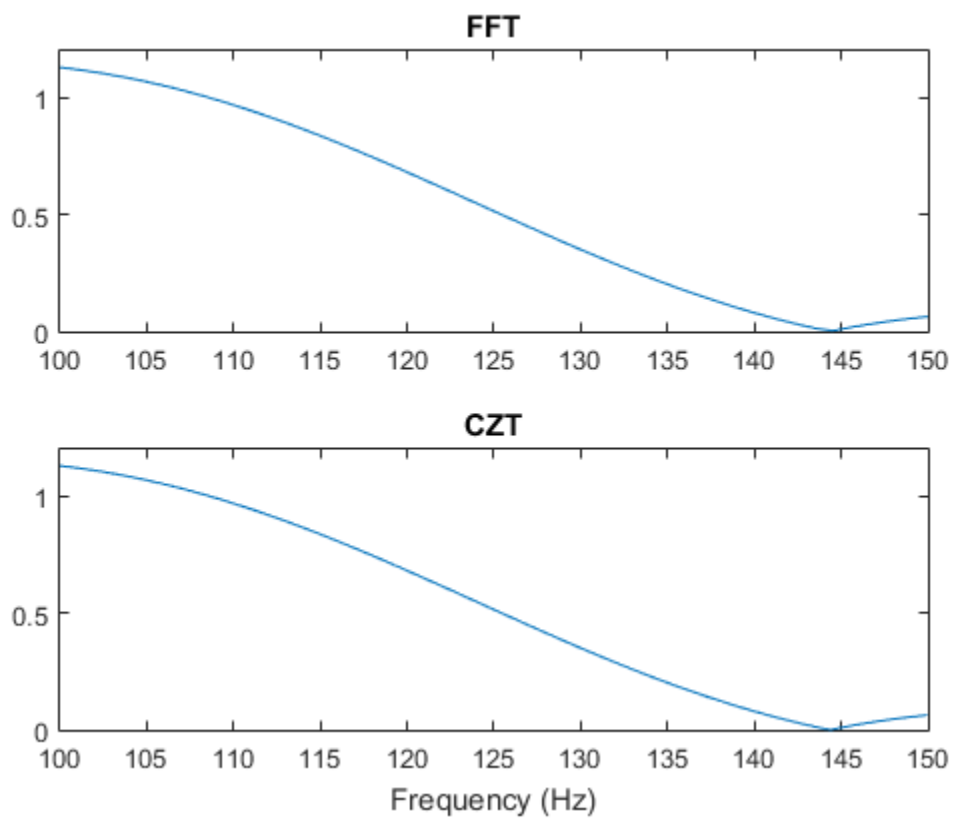
f1 = 100;
f2 = 150;
w = exp(-j*2*pi*(f2-f1)/(m*fs));
a = exp(j*2*pi*f1/fs);
z = czt(h,m,w,a);
```

Plot the transforms. Zoom in on the area of interest.

```
fn = (0:m-1)/m;
fy = fs*fn;
fz = (f2-f1)*fn + f1;

subplot(2,1,1)
plot(fy,abs(y))
axis([f1 f2 0 1.2])
title('FFT')

subplot(2,1,2)
plot(fz,abs(z))
axis([f1 f2 0 1.2])
title('CZT')
xlabel('Frequency (Hz)')
```



## Diagnostics

If  $m$ ,  $w$ , or  $a$  is not a scalar, `czt` gives the following error message:

```
Inputs M, W, and A must be scalars.
```

## More About

### Algorithms

`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].

## References

- [1] Rabiner, Lawrence R., and Bernard Gold. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975, pp.393–399.

### See Also

`fft` | `freqz`

# db

Convert energy or power measurements to decibels

## Syntax

```
dboutput = db(X)
dboutput = db(X,SignalType)
dboutput = db(X,R)
dboutput = db(X,'voltage',R)
```

## Description

`dboutput = db(X)` converts the elements of the vector or matrix `X` to decibels (dB). The elements of `X` are voltage measurements across a resistance of 1 ohm.

`dboutput = db(X,SignalType)` specifies the signal type represented by the elements of `X` as 'voltage' or 'power'. The entries are not case sensitive. The default value is 'voltage'. For voltage measurements, the resistance defaults to 1 ohm. If you specify `SignalType` as 'power', the elements of `X` must be nonnegative.

`dboutput = db(X,R)` specifies the resistance `R` for voltage measurements. You can specify a resistance only when the signal measurements are voltages.

`dboutput = db(X,'voltage',R)` specifies the resistance `R` for voltage measurements. This syntax is equivalent to `db(X,R)`.

## Input Arguments

### **x**

Signal measurements. `X` must be a vector or matrix. If the elements of `X` are power measurements, all elements must be nonnegative.

**SignalType**

Type of signal measurements. Valid entries for SignalType are 'voltage' or 'power'. The entries are not case sensitive. If you specify SignalType as 'power', the elements of X must be nonnegative.

**Default:** 'voltage'

**R**

Resistive load in ohms. You can specify resistance only when the SignalType is 'voltage'.

**Default:** 1

**Output Arguments****dboutput**

The energy or power measurements in the input X in decibels. dboutput has the same dimensions as the input X.

If the input X contains voltage (energy) measurements, dboutput is:

$$dB = 10 \log_{10}(|X|^2 / R)$$

If the input X contains power measurements, dboutput is:

$$dB = 10 \log_{10}(X)$$

**Examples****Decibels from Voltage and Power**

Convert voltage to decibels. Assume that the resistance is 2 ohms. Compare the answer to the definition,  $10 \log_{10} \frac{1}{2}$ .

V = 1;

```
R = 2;
dboutput = db(V,2);
compvoltage = [dboutput 10*log10(1/2)]
```

```
compvoltage =
    -3.0103    -3.0103
```

Convert a vector of power measurements to decibels. Compare the answer to the result of using the definition.

```
rng default
X = abs(rand(10,1));
dboutput = db(X, 'power');
comppower = [dboutput 10*log10(X)]
```

```
comppower =
    -0.8899    -0.8899
    -0.4297    -0.4297
    -8.9624    -8.9624
    -0.3935    -0.3935
    -1.9904    -1.9904
   -10.1082   -10.1082
    -5.5518    -5.5518
    -2.6211    -2.6211
    -0.1886    -0.1886
    -0.1552    -0.1552
```

## Alternatives

- `mag2db` — Converts magnitude measurements to decibels.
- `pow2db` — Converts power measurements to decibels.

## See Also

`db2mag` | `db2pow` | `mag2db` | `pow2db`

## db2mag

Convert decibels to magnitude

### Syntax

```
y = db2mag(ydb)
```

### Description

`y = db2mag(ydb)` returns the magnitude measurements, `y`, that correspond to the decibel (dB) values specified in `ydb`. The relationship between magnitude and decibels is  $ydb = 20 \log_{10}(y)$ .

### Examples

#### Magnitudes of Random Numbers

Generate a 2-by-4-by-2 array of Gaussian random numbers. Assume the numbers are expressed in decibels and compute the corresponding magnitudes.

```
r = randn(2,4,2);
```

```
mags = db2mag(r)
```

```
mags(:,:,1) =
```

```
    1.0639    0.7710    1.0374    0.9513  
    1.2351    1.1044    0.8602    1.0402
```

```
mags(:,:,2) =
```

```
    1.5098    0.8561    1.0871    1.0858  
    1.3755    1.4182    0.9928    0.9767
```



Use the definition to check the calculation.

```
chck = 10.^(r/20)
```

```
chck(:, :, 1) =
```

```
    1.0639    0.7710    1.0374    0.9513  
    1.2351    1.1044    0.8602    1.0402
```

```
chck(:, :, 2) =
```

```
    1.5098    0.8561    1.0871    1.0858  
    1.3755    1.4182    0.9928    0.9767
```

## Input Arguments

### **ydb** — Input array in decibels

scalar | vector | matrix | N-D array

Input array in decibels, specified as a scalar, vector, matrix, or N-D array. When ydb is nonscalar, **db2mag** is an element-wise operation.

Data Types: `single` | `double`

## Output Arguments

### **y** — Magnitude measurements

scalar | vector | matrix | N-D array

Magnitude measurements, returned as a scalar, vector, matrix, or N-D array of the same size as ydb.

## See Also

`db` | `db2pow` | `mag2db` | `pow2db`

## db2pow

Convert decibels to power

### Syntax

```
y = db2pow(ydb)
```

### Description

`y = db2pow(ydb)` returns the power measurements, `y`, that correspond to the decibel (dB) values specified in `ydb`. The relationship between power and decibels is  $ydb = 10 \log_{10}(y)$ .

### Examples

#### Power Values of Random Numbers

Generate a 2-by-4-by-2 array of Gaussian random numbers. Assume the numbers are expressed in decibels and compute the corresponding power measurements.

```
r = randn(2,4,2);
```

```
pows = db2pow(r)
```

```
pows(:, :, 1) =
```

|        |        |        |        |
|--------|--------|--------|--------|
| 1.1318 | 0.5944 | 1.0762 | 0.9050 |
| 1.5254 | 1.2196 | 0.7400 | 1.0821 |

```
pows(:, :, 2) =
```

|        |        |        |        |
|--------|--------|--------|--------|
| 2.2795 | 0.7328 | 1.1818 | 1.1789 |
| 1.8921 | 2.0114 | 0.9856 | 0.9539 |

Use the definition to check the calculation.

```
chck = 10.^(r/10)
```

```
chck(:, :, 1) =
```

```
    1.1318    0.5944    1.0762    0.9050  
    1.5254    1.2196    0.7400    1.0821
```

```
chck(:, :, 2) =
```

```
    2.2795    0.7328    1.1818    1.1789  
    1.8921    2.0114    0.9856    0.9539
```

## Input Arguments

### **ydb** — Input array in decibels

scalar | vector | matrix | N-D array

Input array in decibels, specified as a scalar, vector, matrix, or N-D array. When ydb is nonscalar, **db2pow** is an element-wise operation.

Data Types: `single` | `double`

## Output Arguments

### **y** — Power measurements

scalar | vector | matrix | N-D array

Power measurements, returned as a scalar, vector, matrix, or N-D array of the same size as ydb.

## See Also

`db` | `db2mag` | `mag2db` | `pow2db`

## dct

Discrete cosine transform (DCT)

### Syntax

$y = \text{dct}(x)$   
 $y = \text{dct}(x, n)$

### Description

$y = \text{dct}(x)$  returns the unitary discrete cosine transform of  $x$ ,

$$y(k) = w(k) \sum_{n=1}^N x(n) \cos\left(\frac{\pi}{2N} (2n-1)(k-1)\right), \quad k = 1, 2, \dots, N,$$

where

$$w(k) = \begin{cases} \frac{1}{\sqrt{N}}, & k = 1, \\ \sqrt{\frac{2}{N}}, & 2 \leq k \leq N, \end{cases}$$

$N$  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  $N - 1$ .

$y = \text{dct}(x, n)$  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

### Energy Stored in DCT Coefficients

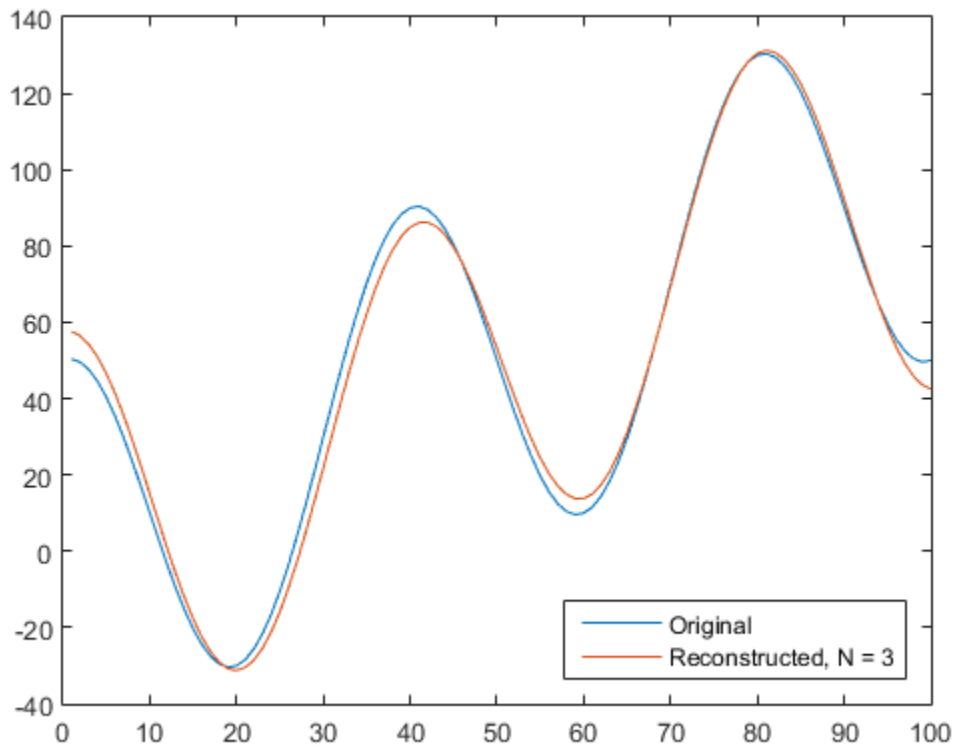
Find how many DCT coefficients represent 99% of the energy in a sequence.

```
x = (1:100) + 50*cos((1:100)*2*pi/40);
X = dct(x);
[XX,ind] = sort(abs(X),'descend');
i = 1;
while norm(X(ind(1:i)))/norm(X)<0.99
    i = i + 1;
end
Needed = i;
```

Reconstruct the signal and compare to the original.

```
X(ind(Needed+1:end)) = 0;
xx = idct(X);

plot([x;xx]')
legend('Original',['Reconstructed, N = ' int2str(Needed)], ...
       'Location','SouthEast')
```



## 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: Van Nostrand-Reinhold, 1993, chap.4.

## See Also

`fft` | `idct` | `dct2` | `idct2`

# decimate

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 of a sequence to a lower rate. It is the opposite of interpolation. `decimate` lowpass filters the input to guard against aliasing and downsamples the result.

`y = decimate(x,r)` reduces the sampling rate of `x`, the input signal, by a factor of `r`. The decimated vector, `y`, is shortened by a factor of `r` so that `length(y) = ceil(length(x)/r)`. By default, `decimate` uses a lowpass Chebyshev Type I IIR filter of order 8.

`y = decimate(x,r,n)` uses a Chebyshev filter of order `n`. Orders above 13 are not recommended because of numerical instability. The function displays a warning in those cases.

`y = decimate(x,r,'fir')` uses an FIR filter designed using the window method with a Hamming window. The filter has order 30.

`y = decimate(x,r,n,'fir')` uses an FIR filter of order `n`.

---

**Note:** For better results when `r` is greater than 13, divide `r` into smaller factors and call `decimate` several times.

---

## Input Arguments

### **x** — Input signal

vector

Input signal, specified as a vector.

Data Types: `double`

### **r** — Decimation factor

positive integer scalar

Decimation factor, specified as a positive integer scalar.

Data Types: `double`

### **n** — Filter order

positive integer scalar

Filter order, specified as a positive integer scalar. When using the IIR filter, avoid values above 13 because sometimes the results are unreliable.

Data Types: `double`

## Output Arguments

### **y** — Decimated signal

vector

Decimated signal, returned as a vector.

Data Types: `double`

## Examples

### Decimate a Signal

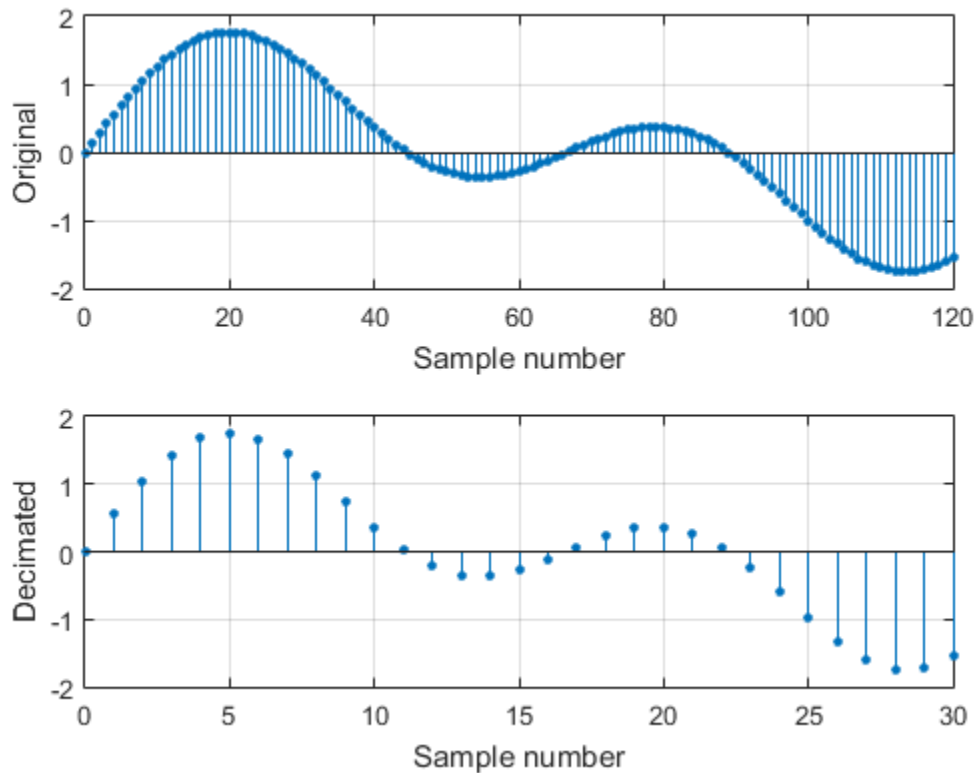
Create a sinusoidal signal sampled at 4 kHz. Decimate it by a factor of four.

```
t = 0:.00025:1;  
x = sin(2*pi*30*t) + sin(2*pi*60*t);  
y = decimate(x,4);
```



Plot the original and decimated signals.

```
subplot 211
stem(0:120,x(1:121),'filled','markersize',3)
grid on
xlabel 'Sample number',ylabel 'Original'
subplot 212
stem(0:30,y(1:31),'filled','markersize',3)
grid on
xlabel 'Sample number',ylabel 'Decimated'
```



### Decimate a Signal Using the Chebyshev Filter

Create a signal with two sinusoids. Decimate it by a factor of 13 using a Chebyshev IIR filter of order 5. Plot the original and decimated signals.

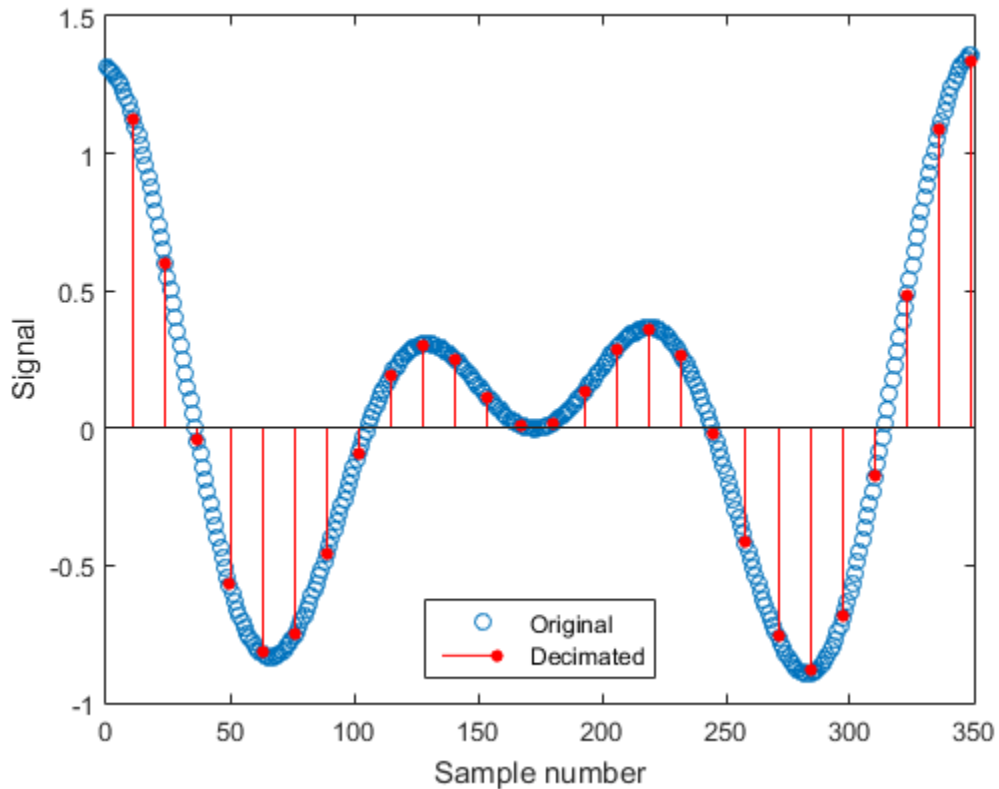
```

r = 13;
n = 16:365;
lx = length(n);
x = sin(2*pi*n/153) + cos(2*pi*n/127);

plot(0:lx-1,x,'o')
hold on
y = decimate(x,r,5);
stem(lx-1:-r:0,flip1r(y),'ro','filled','markersize',4)

legend('Original','Decimated','Location','south')
xlabel('Sample number')
ylabel('Signal')

```



The original and decimated signals have matching *last* elements.

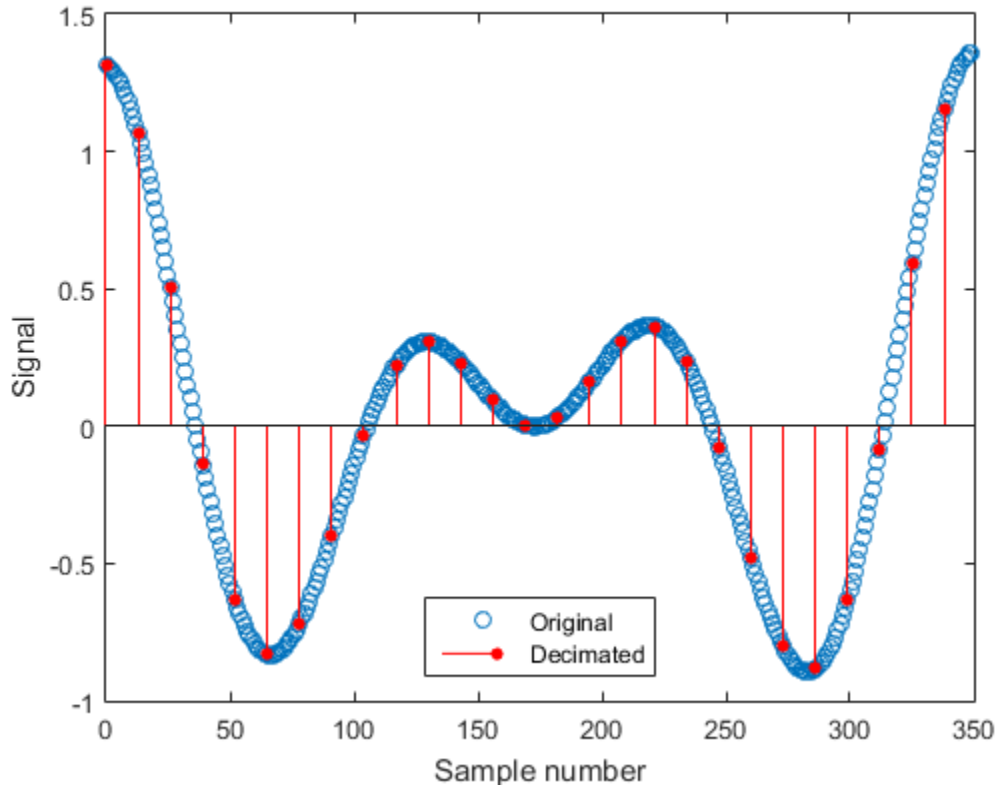
### Decimate a Signal Using the FIR Filter

Create a signal with two sinusoids. Decimate it by a factor of 13 using an FIR filter of order 82. Plot the original and decimated signals.

```
r = 13;
n = 16:365;
lx = length(n);
x = sin(2*pi*n/153) + cos(2*pi*n/127);

plot(0:lx-1,x,'o')
hold on
y = decimate(x,r,82,'fir');
stem(0:r:lx-1,y,'ro','filled','markersize',4)

legend('Original','Decimated','Location','south')
xlabel('Sample number')
ylabel('Signal')
```



The original and decimated signals have matching *first* elements.

## More About

### Algorithms

`decimate` uses decimation algorithms 8.2 and 8.3 from [1].

- 1 `decimate` creates a lowpass filter. The default is a Chebyshev Type I filter designed using `cheby1`. This filter has normalized cutoff frequency  $0.8/r$  and passband ripple 0.05 dB. Sometimes, the specified filter order produces passband distortion due to roundoff errors accumulated from the convolutions needed to create the

transfer function. The filter order is automatically reduced when distortion causes the magnitude response at the cutoff frequency to differ from the ripple by more than  $10^{-6}$ .

When the 'fir' option is chosen, `decimate` uses `fir1` to design a lowpass FIR filter with cutoff frequency  $1/r$ .

- 2 When using the FIR filter, `decimate` filters the input sequence in only one direction. This conserves memory and is useful for working with long sequences. In the IIR case, `decimate` applies the filter in forward and reverse directions using `filtfilt` to remove phase distortion. This in effect doubles the filter order. In both cases, the function minimizes transient effects at both ends of the signal by matching endpoint conditions.
- 3 Finally, `decimate` resamples the data by selecting every  $r$ th point from the interior of the filtered signal. The resampled sequence is such that `y(end)` matches `x(end)` when the IIR filter is used and `y(1)` matches `x(1)` in the FIR case.

## References

- [1] Digital Signal Processing Committee of the IEEE Acoustics, Speech, and Signal Processing Society, eds. *Programs for Digital Signal Processing*. New York: IEEE Press, 1979, chap. 8.

## See Also

`cheby1` | `downsample` | `filtfilt` | `fir1` | `interp` | `resample`

## demod

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`.)

---

**Note:** Use `demod` and `modulate` in the Signal Processing Toolbox™ with real-valued signals to obtain real-valued outputs. `demod` and `modulate` are not intended to accept complex-valued inputs or produce complex-valued outputs.

---

| Method   | Description   |
|--|---|
| <code>amdsb-sc</code><br>or<br><code>am</code> | Amplitude demodulation, double sideband, suppressed carrier. Multiplies <code>y</code> by a sinusoid of frequency <code>fc</code> and applies a fifth-order Butterworth lowpass filter using <code>filtfilt</code> .<br><br><pre>x = y.*cos(2*pi*fc*t);<br/><br/>[b,a] = butter(5,fc*2/fs);<br/><br/>x = filtfilt(b,a,x);</pre> |

| Method   | Description  |
|----------|--|
| amdsb-tc | <p>Amplitude demodulation, double sideband, transmitted carrier. Multiplies <math>y</math> by a sinusoid of frequency <math>fc</math> and applies a fifth-order Butterworth lowpass filter using <code>filtfilt</code>.</p> <pre>x = y.*cos(2*pi*fc*t); [b,a] = butter(5,fc*2/fs); x = filtfilt(b,a,x);</pre> <p>If you specify <code>opt</code>, <code>demod</code> subtracts scalar <code>opt</code> from <math>x</math>. The default value for <code>opt</code> is 0.</p> |
| amssb    | <p>Amplitude demodulation, single sideband. Multiplies <math>y</math> by a sinusoid of frequency <math>fc</math> and applies a fifth-order Butterworth lowpass filter using <code>filtfilt</code>.</p> <pre>x = y.*cos(2*pi*fc*t); [b,a] = butter(5,fc*2/fs); x = filtfilt(b,a,x);</pre>   |
| fm       | <p>Frequency demodulation. Demodulates the FM waveform by modulating the Hilbert transform of <math>y</math> by a complex exponential of frequency <math>-fc</math> Hz and obtains the instantaneous frequency of the result.</p>  |
| pm       | <p>Phase demodulation. Demodulates the PM waveform by modulating the Hilbert transform of <math>y</math> by a complex exponential of frequency <math>-fc</math> Hz and obtains the instantaneous phase of the result.</p>  |
| ppm      | <p>Pulse-position demodulation. Finds the pulse positions of a pulse-position modulated signal <math>y</math>. For correct demodulation, the pulses cannot overlap. <math>x</math> is length <code>length(t)*fc/fs</code>.</p>   |
| pwm      | <p>Pulse-width demodulation. Finds the pulse widths of a pulse-width modulated signal <math>y</math>. <code>demod</code> returns in <math>x</math> a vector whose elements specify the width of each pulse in fractions of a period. The pulses in <math>y</math> should start at the beginning of each carrier period, that is, they should be left justified.</p>  |

| Method | Description  |
|--------|--|
| qam    | <p>Quadrature amplitude demodulation.</p> <p><code>[x1,x2] = demod(y,fc,fs,'qam')</code> multiplies <code>y</code> by a cosine and a sine of frequency <code>fc</code> and applies a fifth-order Butterworth lowpass filter using <code>filtfilt</code>.</p> <pre>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);</pre> |

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` | `mksdemod` | `pamdmod` | `pmdemod` | `qamdmod` | `vco` | `fskdemod` | `genqamdmod`



# design

Apply design method to filter specification object

## Syntax

```
H = design(D)
H = design(D,METHOD)
H = design(D,METHOD,PARAM1,VALUE1,PARAM2,VALUE2,...)
H = design(D,METHOD,OPTS)
Hs = design(D,...,'SystemObject',sysobjflag)
```

## Description

`H = design(D)` uses the filter specifications object `D` to generate a filter `H`. When you do not provide a design method as an input argument, `design` uses a default design method. Use `designmethods(D, 'default')` to see the default design method for your filter specifications object.

`H = design(D, METHOD)` forces the design method specified by the string `METHOD`. `METHOD` must be one of the strings returned by `designmethods`. Use `designmethods(D, 'default')` to determine which algorithm is used by default.

The design method you provide as the `designmethod` input argument must be one of the methods returned by

```
designmethods(d)
```

To help you design filters more quickly, the input argument `METHOD` accepts a variety of special keywords that force `design` to behave in different ways. The following table presents the keywords you can use for `METHOD` and how `design` responds to the keyword.

| Designmethod Keyword | Description of the design Response  |
|----------------------|---|
| 'FIR'                | Forces <code>design</code> to produce an FIR filter. When no FIR design method exists for object <code>D</code> , <code>design</code> returns an error. |
| 'IIR'                | Forces <code>design</code> to produce an IIR filter. When no IIR design method exists for object <code>D</code> , <code>design</code> 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, <b>design</b> 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, <b>design</b> returns multiple filters in the output object.  |
| 'ALL'                | Designs filters using all applicable design methods for the specifications object D. As a result, <b>design</b> returns multiple filters, one for each design method. <b>design</b> uses the design methods in the order that <b>designmethods(D)</b> returns them. |

Keywords are not case sensitive

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)

H = **design**(D,METHOD,PARAM1,VALUE1,PARAM2,VALUE2,...) specifies design-method options. Use **help**(D,METHOD) for complete information on which design-method-specific options are available. You can also use **designopts**(D,METHOD) for a less-detailed listing of the design-method-specific options.

H = **design**(D,METHOD,OPTS) specifies design-method options using the structure OPTS. OPTS is usually obtained from **designopts** and then specified as an input to **design**. Use **help**(D,METHOD) for more information on optional inputs.

Hs = **design**(D,...,'SystemObject',*sysobjflag*) uses the filter specifications object D to generate a filter System object Hs when *sysobjflag* is true. To generate System objects, you must have the DSP System Toolbox™ product installed. When *sysobjflag* is false, the function generates a **dfilt** or **mfilt** object H, as described previously. Design methods and design options for filter System objects are not necessarily the same as those for **dfilt** and **mfilt** objects. To check design methods for System objects, use **designmethods** with the 'SystemObject',*sysobjflag* syntax.

If you are specifying design-method-specific options using OPTS, you can also set OPTS.SystemObject to true instead of calling **design** with the 'SystemObject',*sysobjflag* syntax.

## Examples

Design an FIR equiripple lowpass filter. The passband edge frequency is  $0.2\pi$  radians/sample, and the stopband edge frequency is  $0.25\pi$  radians/sample. The passband ripple is 0.5 dB, and the stopband attenuation is 40 dB.

```
D = fdesign.lowpass('Fp,Fst,Ap,Ast',0.2,0.25,0.5,40);
H = design(D); % Uses the default equiripple method.
```

If you have the DSP System Toolbox software installed, you can design a minimum-phase FIR equiripple filter. Design a minimum-phase filter and compare the pole-zero plots of the original and minimum-phase designs.

```
Hmin = design(D,'equiripple','MinPhase',true);
hfvt = fvtool([H Hmin],'analysis','polezero');
legend(hfvt,'Original Design','Minimum Phase Design');
```

Design a Butterworth lowpass filter. The passband edge frequency is  $0.2\pi$  radians/sample, and the stopband edge frequency is  $0.25\pi$  radians/sample. The passband ripple is 0.5 dB, and the stopband attenuation is 40 dB. Obtain help on the design options specific to the Butterworth design method. Design the filter with the 'MatchExactly' option set to 'Passband'.

```
D = fdesign.lowpass('Fp,Fst,Ap,Ast',0.2,0.25,0.5,40);
% Query design-method-specific options
help(D,'butter')
% Match passband exactly
H = design(D,'butter','MatchExactly','passband');
```

If you have the DSP System Toolbox software, you can specify the  $P$ -th norm scaling on the second-order sections. Use L-infinity norm scaling in the time domain.

```
H = design(D,'butter','MatchExactly','passband','SOSScaleNorm','linf');
```

If you have the DSP System Toolbox software, you can create a filter System object.

```
Hs = design(D,'SystemObject',true);
```

## See Also

designmethods | designopts

# designfilt

Design digital filters

## Syntax

```
d = designfilt(resp,Name,Value)
```

```
designfilt(d)
```

## Description

`d = designfilt(resp,Name,Value)` designs a `digitalFilter` object, `d`, with response type `resp`. Specify the filter further using a set of `Name,Value` pairs. The allowed specification sets depend on the response type, `resp`, and consist of combinations of the following:

- *Frequency constraints* correspond to the frequencies at which a filter exhibits a desired behavior. Examples include `'PassbandFrequency'` and `'CutoffFrequency'`. (See the complete list under “Name-Value Pair Arguments” on page 1-239.) You must always specify the frequency constraints.
- *Magnitude constraints* describe the filter behavior at particular frequency ranges. Examples include `'PassbandRipple'` and `'StopbandAttenuation'`. (See the complete list under “Name-Value Pair Arguments” on page 1-239.) `designfilt` provides default values for magnitude constraints left unspecified. In arbitrary-magnitude designs you must always specify the vectors of desired amplitudes.
- `'FilterOrder'`. Some design methods let you specify the order. Others produce minimum-order designs. That is, they generate the smallest filters that satisfy the specified constraints.
- `'DesignMethod'` is the algorithm used to design the filter. Examples include constrained least squares (`'cls'`) and windowing (`'window'`). For some specification sets, there are multiple design methods available to choose from. In other cases, you can use only one method to meet the desired specifications.
- *Design options* are parameters specific to a given design method. Examples include `'Window'` for the windowing method and optimization `'Weights'` for arbitrary-

magnitude equiripple designs. (See the complete list under “Name-Value Pair Arguments” on page 1-239.) `designfilt` provides default values for design options left unspecified.

- 'SampleRate' is the frequency at which the filter operates. `designfilt` has a default sample rate of 2 Hz. Using this value is equivalent to working with normalized frequencies.

---

**Note:** If you specify an incomplete or inconsistent set of name-value pairs at the command line, `designfilt` offers to open a “Filter Design Assistant” on page 1-248. The assistant helps you design the filter and pastes the corrected MATLAB code on the command line.

If you call `designfilt` from a script or function with an incorrect set of specifications, `designfilt` issues an error message with a link to open a “Filter Design Assistant” on page 1-248. The assistant helps you design the filter, comments out the faulty code in the function or script, and pastes the corrected MATLAB code on the next line.

---

- Use `filter` in the form `dataOut = filter(d,dataIn)` to filter a signal with a `digitalFilter`, `d`.
- Use `fvtool` to visualize a `digitalFilter`, `d`.
- See `digitalFilter` for a list of the filtering and analysis functions available for use with `digitalFilter` objects.

`designfilt(d)` lets you edit an existing digital filter, `d`. It opens a “Filter Design Assistant” on page 1-248 populated with the filter’s specifications, which you can then modify. This is the only way you can edit a `digitalFilter` object.

## Examples

### Lowpass FIR Filter

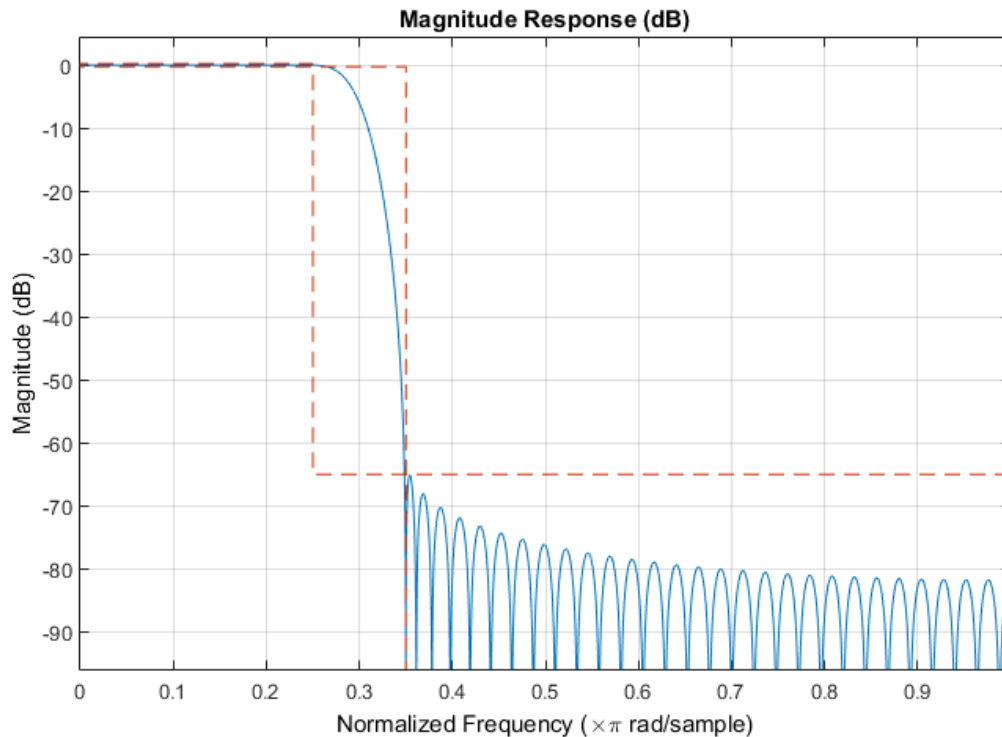
Design a minimum-order lowpass FIR filter with normalized passband frequency  $0.25\pi$  rad/s, stopband frequency  $0.35\pi$  rad/s, passband ripple 0.5 dB, and stopband attenuation 65 dB. Use a Kaiser window to design the filter. Visualize its magnitude response. Use it to filter a vector of random data.

```
lpFilt = designfilt('lowpassfir', 'PassbandFrequency', 0.25, ...
```

```

        'StopbandFrequency',0.35,'PassbandRipple',0.5, ...
        'StopbandAttenuation',65,'DesignMethod','kaiserwin');
fvtool(lpFilt)
dataIn = rand(1000,1);
dataOut = filter(lpFilt,dataIn);

```



### Lowpass IIR Filter

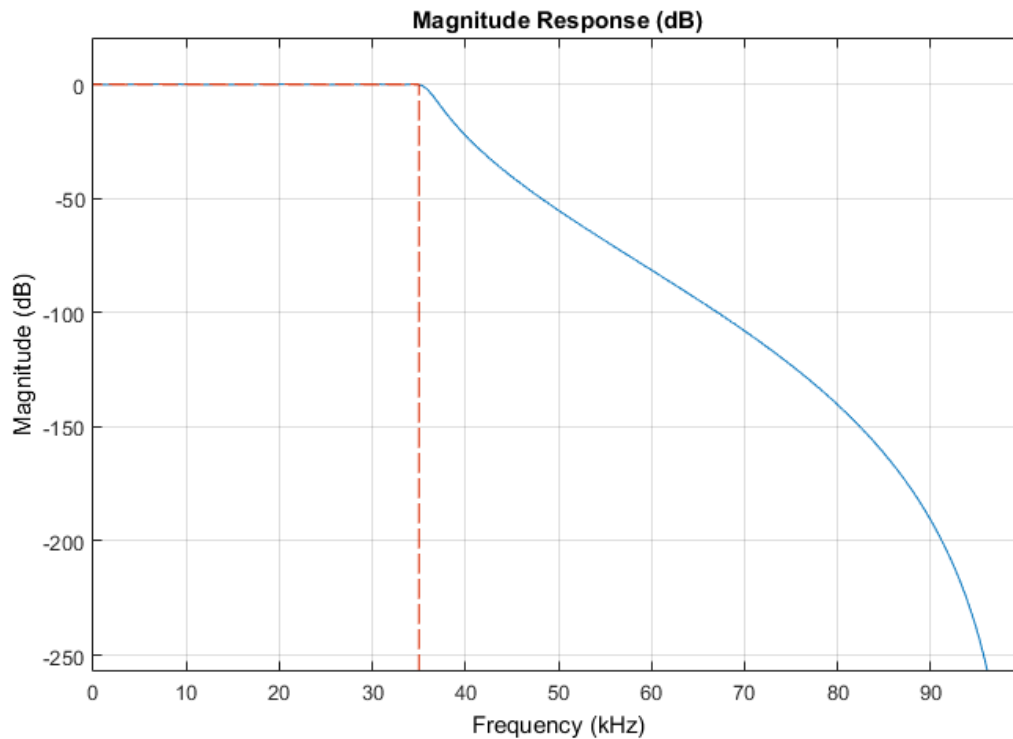
Design a lowpass IIR filter with order 8, passband frequency 35 kHz, and passband ripple 0.2 dB. Specify a sample rate of 200 kHz. Visualize the magnitude response of the filter. Use it to filter a 1000-sample random signal.

```

lpFilt = designfilt('lowpassiir','FilterOrder',8, ...
    'PassbandFrequency',35e3,'PassbandRipple',0.2, ...
    'SampleRate',200e3);

```

```
fvtool(lpFilt)
dataIn = randn(1000,1);
dataOut = filter(lpFilt,dataIn);
```

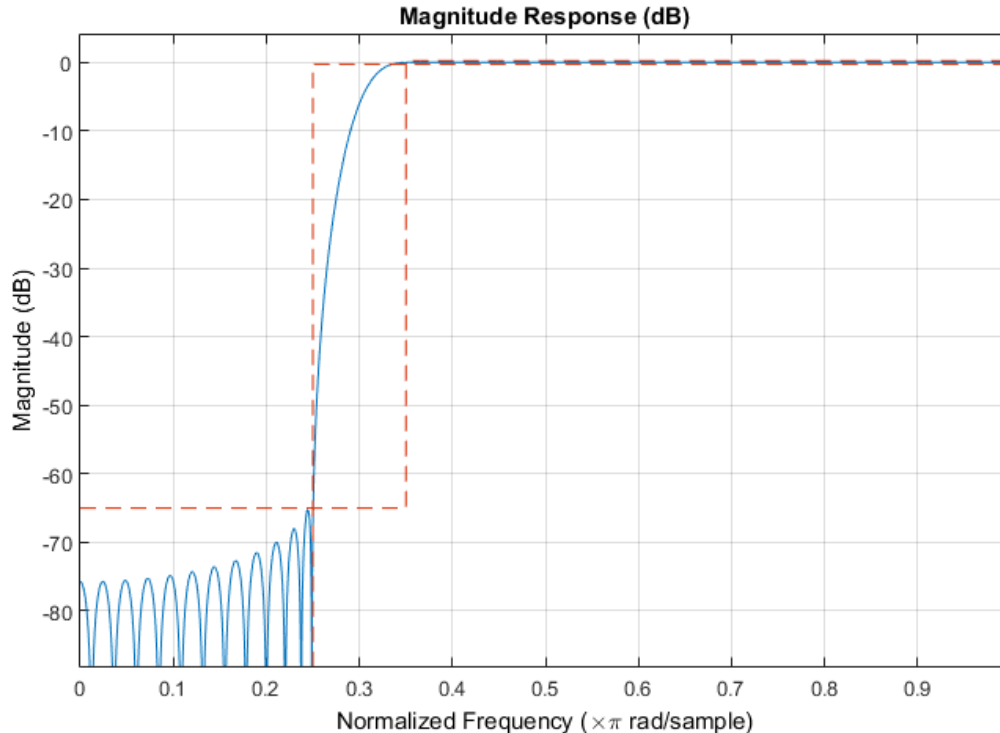


### Highpass FIR Filter

Design a minimum-order highpass FIR filter with normalized stopband frequency  $0.25\pi$  rad/s, passband frequency  $0.35\pi$  rad/s, passband ripple 0.5 dB, and stopband attenuation 65 dB. Use a Kaiser window to design the filter. Visualize its magnitude response. Use it to filter 1000 samples of random data.

```
hpFilt = designfilt('highpassfir','StopbandFrequency',0.25, ...
    'PassbandFrequency',0.35,'PassbandRipple',0.5, ...
    'StopbandAttenuation',65,'DesignMethod','kaiserwin');
fvtool(hpFilt)
```

```
dataIn = randn(1000,1);
dataOut = filter(hpFilt,dataIn);
```

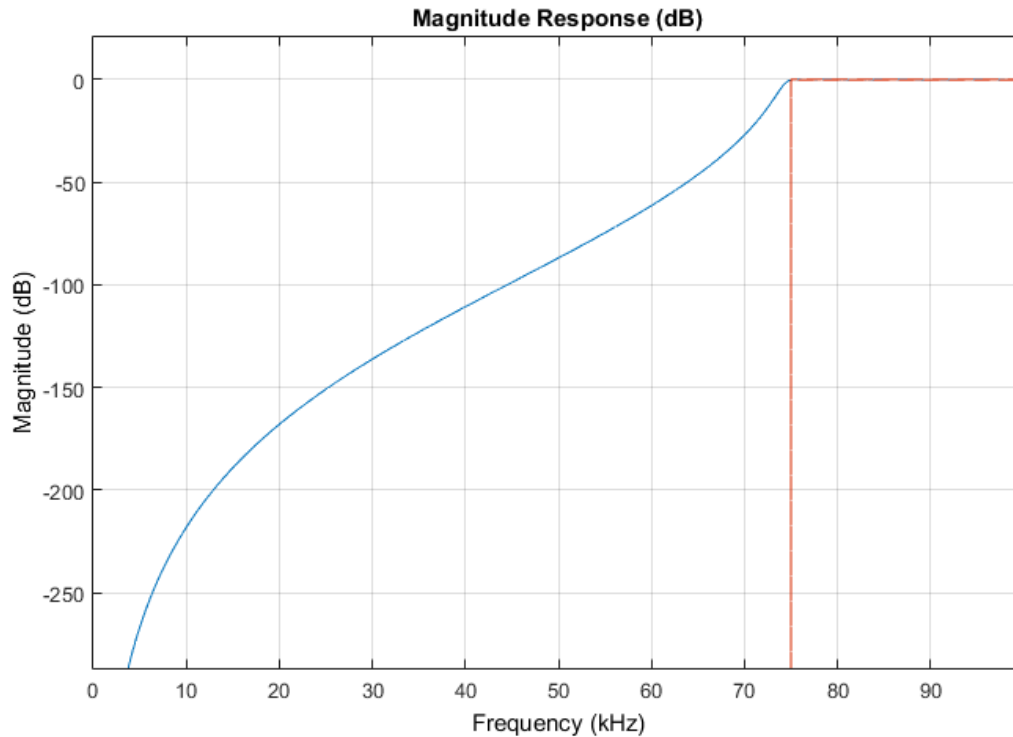


### Highpass IIR Filter

Design a highpass IIR filter with order 8, passband frequency 75 kHz, and passband ripple 0.2 dB. Specify a sample rate of 200 kHz. Visualize the filter's magnitude response. Apply the filter to a 1000-sample vector of random data.

```
hpFilt = designfilt('highpassiir','FilterOrder',8, ...
    'PassbandFrequency',75e3,'PassbandRipple',0.2, ...
    'SampleRate',200e3);
fvtool(hpFilt)
dataIn = randn(1000,1);
dataOut = filter(hpFilt,dataIn);
```

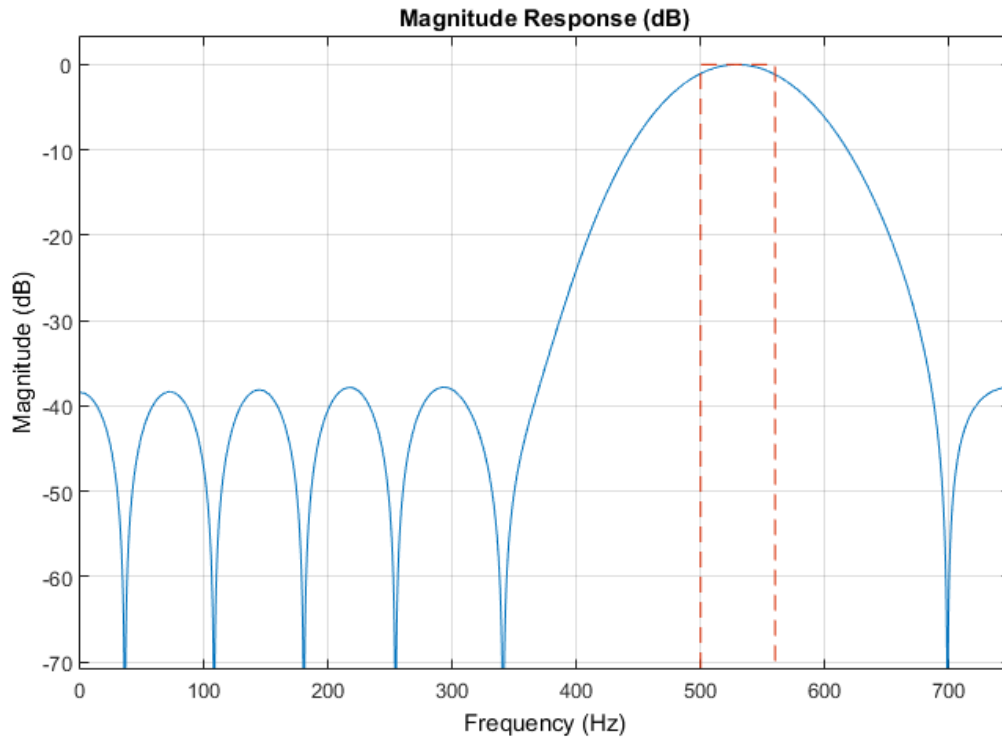




### Bandpass FIR Filter

Design a 20th-order bandpass FIR filter with lower cutoff frequency 500 Hz and higher cutoff frequency 560 Hz. The sample rate is 1500 Hz. Visualize the magnitude response of the filter. Use it to filter a random signal containing 1000 samples.

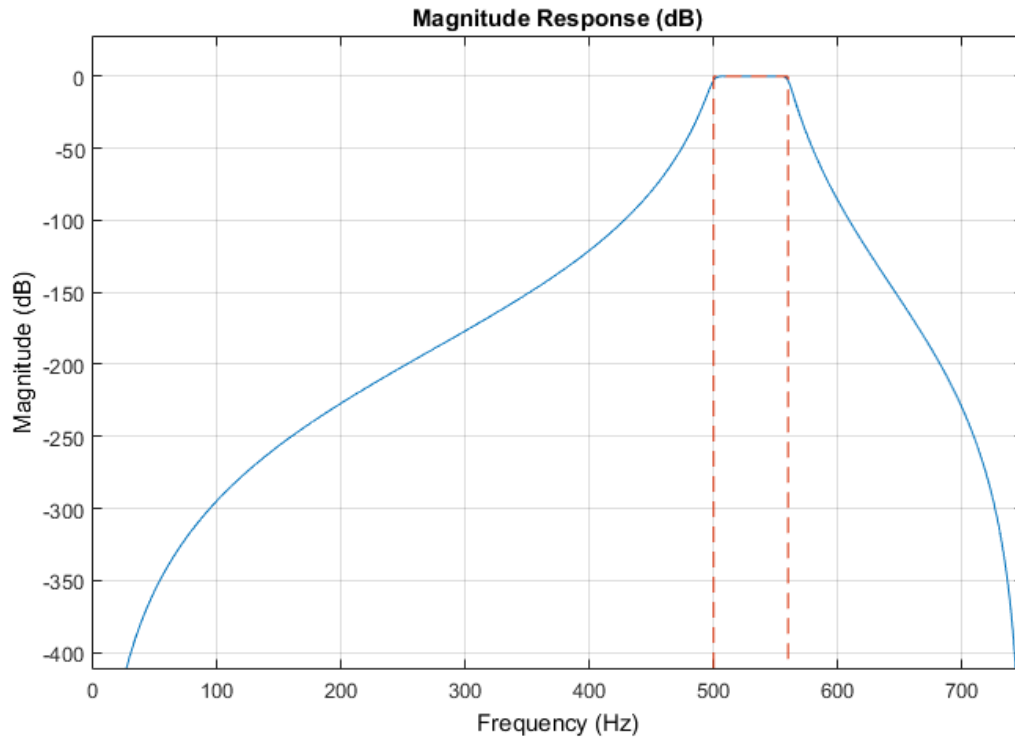
```
bpFilt = designfilt('bandpassfir','FilterOrder',20, ...  
                  'CutoffFrequency1',500,'CutoffFrequency2',560, ...  
                  'SampleRate',1500);  
fvtool(bpFilt)  
dataIn = randn(1000,1);  
dataOut = filter(bpFilt,dataIn);
```



### Bandpass IIR Filter

Design a 20th-order bandpass IIR filter with lower 3-dB frequency 500 Hz and higher 3-dB frequency 560 Hz. The sample rate is 1500 Hz. Visualize the frequency response of the filter. Use it to filter a 1000-sample random signal.

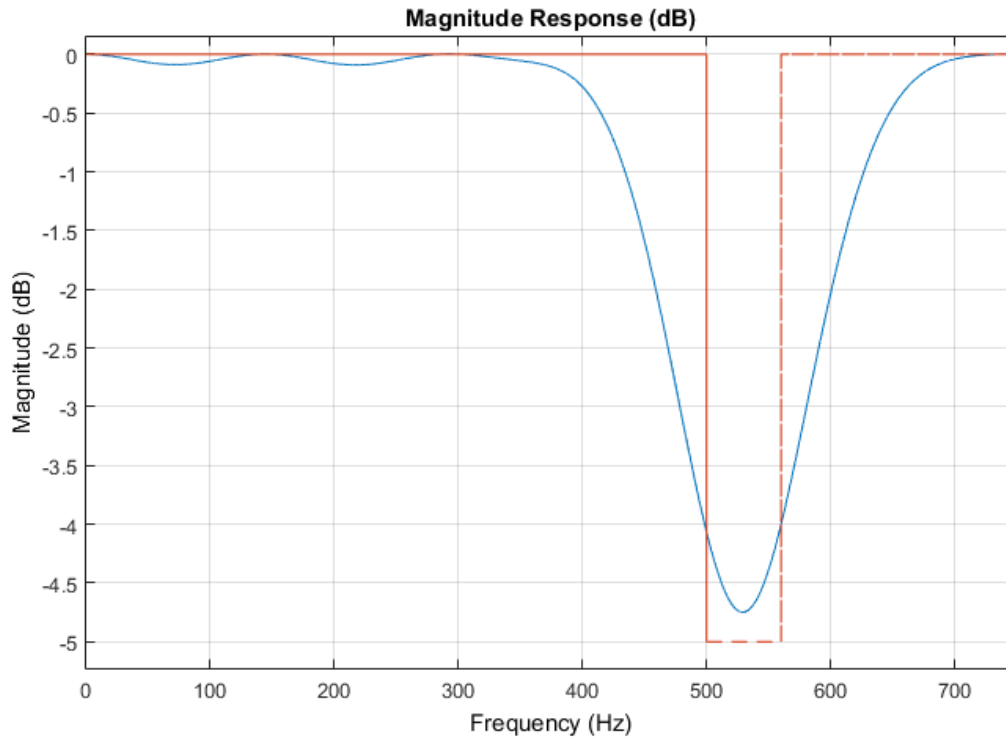
```
bpFilt = designfilt('bandpassiir','FilterOrder',20, ...  
                  'HalfPowerFrequency1',500,'HalfPowerFrequency2',560, ...  
                  'SampleRate',1500);  
fvtool(bpFilt)  
dataIn = randn(1000,1);  
dataOut = filter(bpFilt,dataIn);
```



### Bandstop FIR Filter

Design a 20th-order bandstop FIR filter with lower cutoff frequency 500 Hz and higher cutoff frequency 560 Hz. The sample rate is 1500 Hz. Visualize the magnitude response of the filter. Use it to filter 1000 samples of random data.

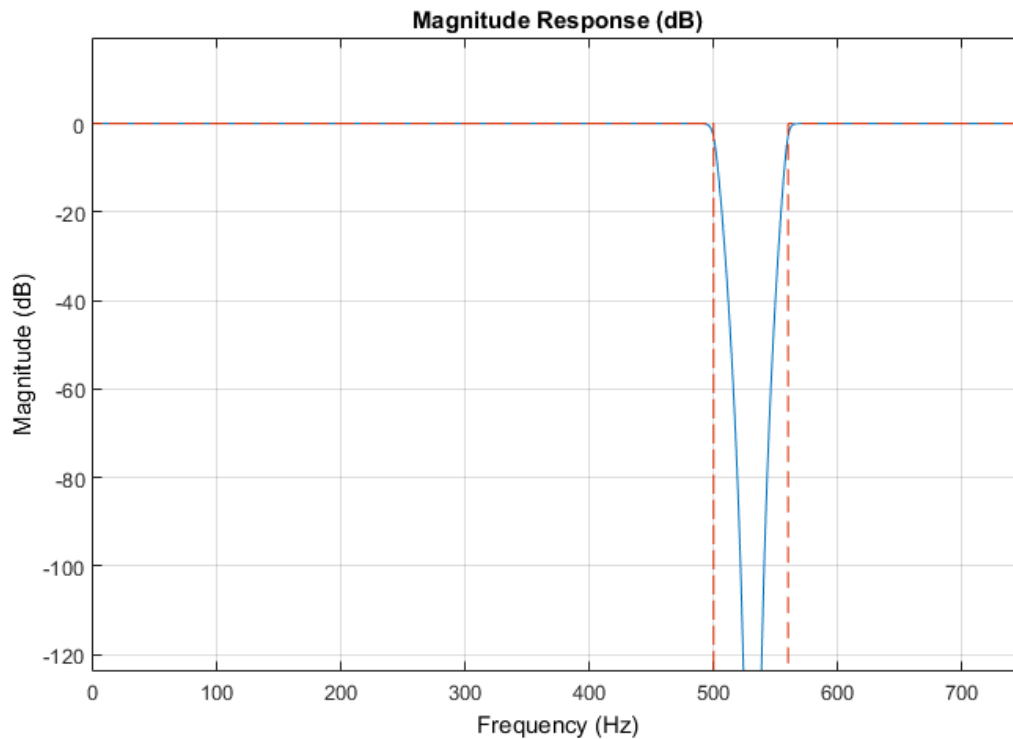
```
bsFilt = designfilt('bandstopfir','FilterOrder',20, ...  
                  'CutoffFrequency1',500,'CutoffFrequency2',560, ...  
                  'SampleRate',1500);  
fvtool(bsFilt)  
dataIn = randn(1000,1);  
dataOut = filter(bsFilt,dataIn);
```



### Bandstop IIR Filter

Design a 20th-order bandstop IIR filter with lower 3-dB frequency 500 Hz and higher 3-dB frequency 560 Hz. The sample rate is 1500 Hz. Visualize the magnitude response of the filter. Use it to filter 1000 samples of random data.

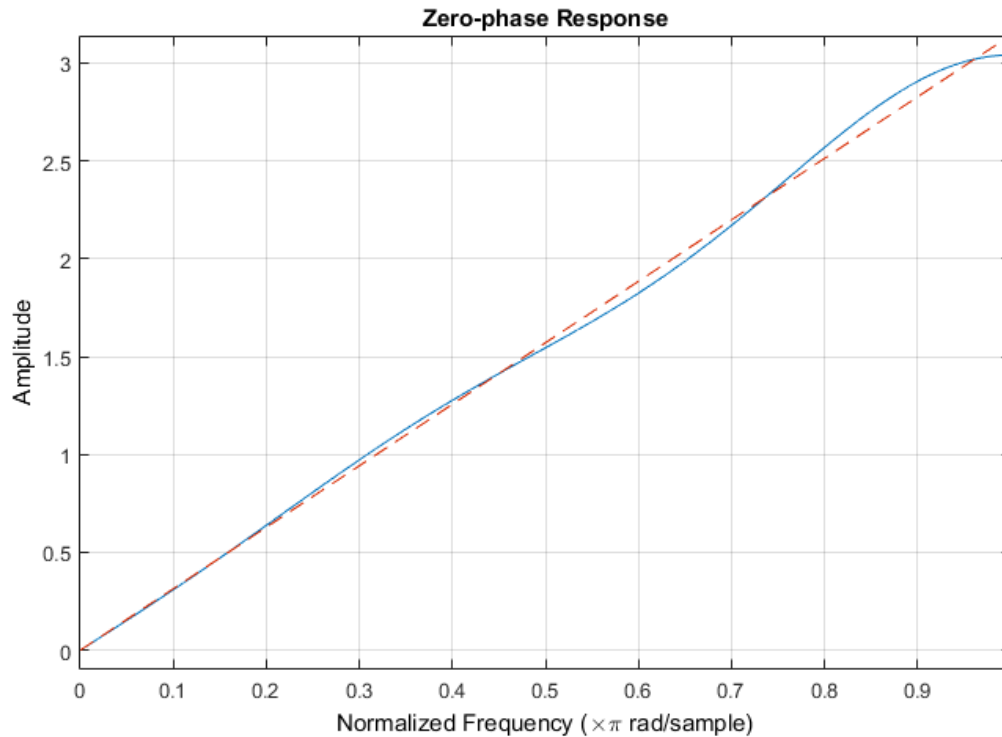
```
bsFilt = designfilt('bandstopiir','FilterOrder',20, ...  
                  'HalfPowerFrequency1',500,'HalfPowerFrequency2',560, ...  
                  'SampleRate',1500);  
fvtool(bsFilt)  
dataIn = randn(1000,1);  
dataOut = filter(bsFilt,dataIn);
```



### FIR Differentiator

Design a full-band differentiator filter of order 7. Display its zero-phase response. Use it to filter a 1000-sample vector of random data.

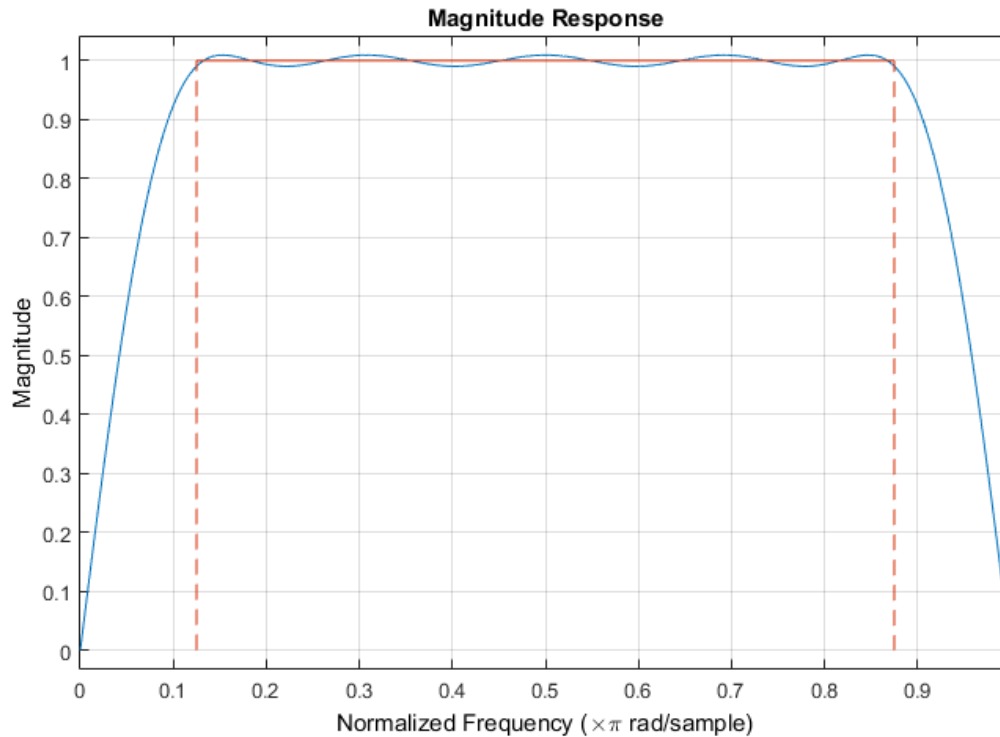
```
dFilt = designfilt('differentiatorfir','FilterOrder',7);  
fvtool(dFilt,'MagnitudeDisplay','Zero-phase')  
dataIn = randn(1000,1);  
dataOut = filter(dFilt,dataIn);
```



### FIR Hilbert Transformer

Design a Hilbert transformer of order 18. Specify a normalized transition width of  $0.25\pi$  rad/s. Display in linear units the magnitude response of the filter. Use it to filter a 1000-sample vector of random data.

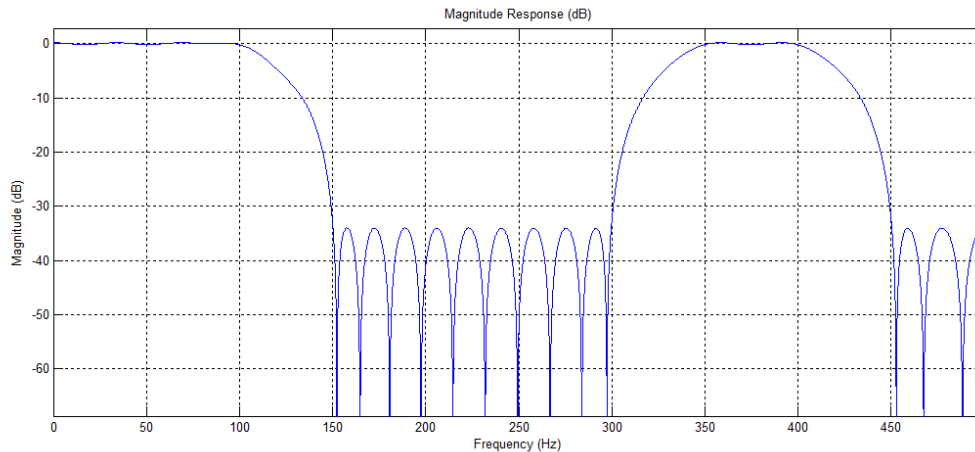
```
hFilt = designfilt('hilbertfir','FilterOrder',18,'TransitionWidth',0.25);  
fvtool(hFilt,'MagnitudeDisplay','magnitude')  
dataIn = randn(1000,1);  
dataOut = filter(hFilt,dataIn);
```



### Arbitrary-Magnitude FIR Filter

You are given a signal sampled at 1 kHz. Design an equiripple filter that stops frequencies from 100 Hz to 350 Hz and frequencies greater than 400 Hz. Specify a filter order of 60. Visualize the frequency response of the filter. Use it to filter a 1000-sample random signal.

```
mbFilt = designfilt('arbmagfir','FilterOrder',60, ...
    'Frequencies',0:50:500, ...
    'Amplitudes',[1 1 1 0 0 0 0 1 1 0 0], ...
    'DesignMethod','equiripple', ...
    'SampleRate',1000);
fvtool(mbFilt)
dataIn = randn([1000 1]); dataOut = filter(mbFilt,dataIn);
```



- “Practical Introduction to Digital Filter Design”
- “Filter Design Gallery”
- “Practical Introduction to Digital Filtering”

## Input Arguments

### **resp** — Filter response and type

'lowpassfir' | 'lowpassiir' | 'highpassfir' | 'highpassiir' |  
 'bandpassfir' | 'bandpassiir' | 'bandstopfir' | 'bandstopiir' |  
 'differentiatorfir' | 'hilbertfir' | 'arbmagfir'

Filter response and type, specified as a string. Click one of the possible values of resp to expand a table of allowed specification sets.

### 'lowpassfir'

Choose this option to design a finite impulse response (FIR) lowpass filter. This example uses the fifth specification set from the following table.

```
d = designfilt('lowpassfir', ...           % Response type
               'FilterOrder',25, ...       % Filter order
               'PassbandFrequency',400, ... % Frequency constraints
               'StopbandFrequency',550, ...
               'DesignMethod','ls', ...    % Design method
```



```

'PassbandWeight',1, ...           % Design method options
'StopbandWeight',2, ...
'SampleRate',2000)                % Sample rate

```

- If you omit 'FilterOrder' (when required), or any of the frequency constraints, `designfilt` throws an error.
- If you omit the magnitude constraints, `designfilt` uses default values.
- If you omit 'DesignMethod', `designfilt` uses the default design method for the specification set.
- If you omit the design method options, `designfilt` uses the defaults for the design method of choice.
- If you omit 'SampleRate', `designfilt` sets it to 2 Hz.

| Filter Order Argument Names | Frequency Constraint Argument Names        | Magnitude Constraint Argument Names       | 'DesignMethod' Argument Values | Design Option Argument Names         |
|-----------------------------|--|---|--------------------------------|--------------------------------------|
| N/A (Minimum-order design)  | 'PassbandFrequency'                        | 'PassbandRipple'                          | 'equiripple' (default)         | N/A                                  |
|                             | 'StopbandFrequency'                        | 'StopbandAttenuation'                     | 'kaiserwin'                    | 'ScalePassband'                      |
| 'FilterOrder'               | 'HalfPowerFrequency'                       | N/A                                       | 'maxflat'                      | N/A                                  |
| 'FilterOrder'               | 'CutoffFrequency'                          | N/A                                       | 'window'                       | 'Window'<br>'ScalePassband'          |
| 'FilterOrder'               | 'CutoffFrequency'                          | 'PassbandRipple'<br>'StopbandAttenuation' | 'cls'                          | 'PassbandOffset'<br>'ZeroPhase'      |
| 'FilterOrder'               | 'PassbandFrequency'<br>'StopbandFrequency' | N/A                                       | 'equiripple' (default)         | 'PassbandWeight'<br>'StopbandWeight' |
|                             |  |   | 'ls'                           | 'PassbandWeight'<br>'StopbandWeight' |

### 'lowpassiir'

Choose this option to design an infinite impulse response (IIR) lowpass filter. This example uses the first specification set from the following table.

```
d = designfilt('lowpassiir', ...           % Response type
              'PassbandFrequency',400, ... % Frequency constraints
              'StopbandFrequency',550, ...
              'PassbandRipple',4, ...     % Magnitude constraints
              'StopbandAttenuation',55, ...
              'DesignMethod','ellip', ... % Design method
              'MatchExactly','passband', ... % Design method options
              'SampleRate',2000)          % Sample rate
```

- If you omit 'FilterOrder' (when required), or any of the frequency constraints, `designfilt` throws an error.
- If you omit the magnitude constraints, `designfilt` uses default values.
- If you omit 'DesignMethod', `designfilt` uses the default design method for the specification set.
- If you omit the design method options, `designfilt` uses the defaults for the design method of choice.
- If you omit 'SampleRate', `designfilt` sets it to 2 Hz.

| Filter Order Argument Names | Frequency Constraint Argument Names | Magnitude Constraint Argument Names | 'DesignMethod' Argument Values | Design Option Argument Names |
|-----------------------------|-------------------------------------|-------------------------------------|--------------------------------|------------------------------|
| N/A (Minimum-order design)  | 'PassbandFrequency'                 | 'PassbandRipple'                    | 'butter' (default)             | 'MatchExactly'               |
|                             | 'StopbandFrequency'                 | 'StopbandAttenuation'               | 'cheby1'                       | 'MatchExactly'               |
|                             |                                     |                                     | 'cheby2'                       | 'MatchExactly'               |
|                             |                                     |                                     | 'ellip'                        | 'MatchExactly'               |
| 'FilterOrder'               | 'HalfPowerFrequency'                | N/A                                 | 'butter'                       | N/A                          |
| 'FilterOrder'               | 'PassbandFrequency'                 | 'PassbandRipple'                    | 'cheby1'                       | N/A                          |
| 'FilterOrder'               | 'PassbandFrequency'                 | 'PassbandRipple'                    | 'ellip'                        | N/A                          |
|                             |                                     | 'StopbandAttenuation'               |                                |                              |
| 'FilterOrder'               | 'StopbandFrequency'                 | 'StopbandAttenuation'               | 'cheby2'                       | N/A                          |
| 'NumeratorOrder'            | 'HalfPowerFrequency'                | N/A                                 | 'butter'                       | N/A                          |
| 'DenominatorOrder'          |                                     |                                     |                                |                              |

**'highpassfir'**

Choose this option to design a finite impulse response (FIR) highpass filter. This example uses the first specification set from the following table.

```
d = designfilt('highpassfir', ...           % Response type
              'StopbandFrequency',400, ... % Frequency constraints
              'PassbandFrequency',550, ...
              'StopbandAttenuation',55, ... % Magnitude constraints
              'PassbandRipple',4, ...
              'DesignMethod','kaiserwin', ... % Design method
              'ScalePassband',false, ...    % Design method options
              'SampleRate',2000)           % Sample rate
```

- If you omit 'FilterOrder' (when required), or any of the frequency constraints, `designfilt` throws an error.
- If you omit the magnitude constraints, `designfilt` uses default values.
- If you omit 'DesignMethod', `designfilt` uses the default design method for the specification set.
- If you omit the design method options, `designfilt` uses the defaults for the design method of choice.
- If you omit 'SampleRate', `designfilt` sets it to 2 Hz.

| Filter Order Argument Names | Frequency Constraint Argument Names        | Magnitude Constraint Argument Names       | 'DesignMethod' Argument Values        | Design Option Argument Names         |
|-----------------------------|--|---|---------------------------------------|--------------------------------------|
| N/A (Minimum-order design)  | 'StopbandFrequency'<br>'PassbandFrequency' | 'StopbandAttenuation'<br>'PassbandRipple' | 'equiripple' (default)<br>'kaiserwin' | N/A<br>'ScalePassband'               |
| 'FilterOrder'               | 'CutoffFrequency'                          | N/A                                       | 'window'                              | 'Window'<br>'ScalePassband'          |
| 'FilterOrder'               | 'CutoffFrequency'                          | 'StopbandAttenuation'<br>'PassbandRipple' | 'cls'                                 | 'PassbandOffset'<br>'ZeroPhase'      |
| 'FilterOrder'               | 'StopbandFrequency'<br>'PassbandFrequency' | N/A                                       | 'equiripple' (default)                | 'PassbandWeight'<br>'StopbandWeight' |

| Filter Order Argument Names | Frequency Constraint Argument Names | Magnitude Constraint Argument Names | 'DesignMethod' Argument Values | Design Option Argument Names         |
|-----------------------------|-------------------------------------|-------------------------------------|--------------------------------|--------------------------------------|
|                             |                                     |                                     | 'ls'                           | 'PassbandWeight'<br>'StopbandWeight' |

**'highpassiir'**

Choose this option to design an infinite impulse response (IIR) highpass filter. This example uses the first specification set from the following table.

```
d = designfilt('highpassiir', ... % Response type
    'StopbandFrequency',400, ... % Frequency constraints
    'PassbandFrequency',550, ...
    'StopbandAttenuation',55, ... % Magnitude constraints
    'PassbandRipple',4, ...
    'DesignMethod','cheby1', ... % Design method
    'MatchExactly','stopband', ... % Design method options
    'SampleRate',2000) % Sample rate
```

- If you omit 'FilterOrder' (when required), or any of the frequency constraints, `designfilt` throws an error.
- If you omit the magnitude constraints, `designfilt` uses default values.
- If you omit 'DesignMethod', `designfilt` uses the default design method for the specification set.
- If you omit the design method options, `designfilt` uses the defaults for the design method of choice.
- If you omit 'SampleRate', `designfilt` sets it to 2 Hz.

| Filter Order Argument Names | Frequency Constraint Argument Names | Magnitude Constraint Argument Names | 'DesignMethod' Argument Values | Design Option Argument Names |
|-----------------------------|-------------------------------------|-------------------------------------|--------------------------------|------------------------------|
| N/A (Minimum-order design)  | 'StopbandFrequency'                 | 'StopbandAttenuation'               | 'butter'<br>(default)          | 'MatchExactly'               |
|                             | 'PassbandFrequency'                 | 'PassbandRipple'                    | 'cheby1'                       | 'MatchExactly'               |
|                             |                                     |                                     | 'cheby2'                       | 'MatchExactly'               |

| Filter Order Argument Names        | Frequency Constraint Argument Names | Magnitude Constraint Argument Names | 'DesignMethod' Argument Values | Design Option Argument Names |
|------------------------------------|-------------------------------------|-------------------------------------|--------------------------------|------------------------------|
|                                    |                                     |                                     | 'ellip'                        | 'MatchExactly'               |
| 'FilterOrder'                      | 'HalfPowerFrequ                     | N/A                                 | 'butter'                       | N/A                          |
| 'FilterOrder'                      | 'PassbandFrequ                      | 'PassbandRipple                     | 'cheby1'                       | N/A                          |
| 'FilterOrder'                      | 'PassbandFrequ                      | 'StopbandAttenu<br>'PassbandRipple  | 'ellip'                        | N/A                          |
| 'FilterOrder'                      | 'StopbandFrequ                      | 'StopbandAttenu                     | 'cheby2'                       | N/A                          |
| 'NumeratorOrder<br>'DenominatorOrd | 'HalfPowerFrequ                     | N/A                                 | 'butter'                       | N/A                          |

### 'bandpassfir'

Choose this option to design a finite impulse response (FIR) bandpass filter. This example uses the fourth specification set from the following table.

```
d = designfilt('bandpassfir', ...           % Response type
    'FilterOrder',86, ...                   % Filter order
    'StopbandFrequency1',400, ...          % Frequency constraints
    'PassbandFrequency1',450, ...
    'PassbandFrequency2',600, ...
    'StopbandFrequency2',650, ...
    'DesignMethod','ls', ...              % Design method
    'StopbandWeight1',1, ...               % Design method options
    'PassbandWeight', 2, ...
    'StopbandWeight2',3, ...
    'SampleRate',2000)                    % Sample rate
```

- If you omit 'FilterOrder' (when required), or any of the frequency constraints, `designfilt` throws an error.
- If you omit the magnitude constraints, `designfilt` uses default values.
- If you omit 'DesignMethod', `designfilt` uses the default design method for the specification set.
- If you omit the design method options, `designfilt` uses the defaults for the design method of choice.

- If you omit 'SampleRate', designfilt sets it to 2 Hz.

| Filter Order Argument Names | Frequency Constraint Argument Names | Magnitude Constraint Argument Names | 'DesignMethod' Argument Values | Design Option Argument Names |
|-----------------------------|-------------------------------------|-------------------------------------|--------------------------------|------------------------------|
| N/A (Minimum-order design)  | 'StopbandFrequency1'                | 'StopbandAttenuation'               | 'equiripple' (default)         | N/A                          |
|                             | 'PassbandFrequency1'                | 'PassbandRipple'                    | 'kaiserwin'                    | 'ScalePassband'              |
|                             | 'PassbandFrequency2'                | 'StopbandAttenuation'               |                                |                              |
|                             | 'StopbandFrequency2'                |                                     |                                |                              |
| 'FilterOrder'               | 'CutoffFrequency'                   | N/A                                 | 'window'                       | 'Window'                     |
|                             | 'CutoffFrequency'                   |                                     |                                | 'ScalePassband'              |
| 'FilterOrder'               | 'CutoffFrequency'                   | 'StopbandAttenuation'               | 'cls'                          | 'PassbandOffset'             |
|                             | 'CutoffFrequency'                   | 'PassbandRipple'                    |                                | 'ZeroPhase'                  |
|                             | 'CutoffFrequency'                   | 'StopbandAttenuation'               |                                |                              |
| 'FilterOrder'               | 'StopbandFrequency1'                | N/A                                 | 'equiripple' (default)         | 'StopbandWeight1'            |
|                             | 'PassbandFrequency1'                |                                     |                                | 'PassbandWeight1'            |
|                             | 'PassbandFrequency2'                |                                     |                                | 'StopbandWeight2'            |
|                             | 'StopbandFrequency2'                |                                     | 'ls'                           | 'StopbandWeight1'            |
|                             |                                     |                                     |                                | 'PassbandWeight1'            |
|                             |                                     |                                     |                                | 'StopbandWeight2'            |

**'bandpassiir'**

Choose this option to design an infinite impulse response (IIR) bandpass filter. This example uses the first specification set from the following table.

```
d = designfilt('bandpassiir', ...           % Response type
    'StopbandFrequency1',400, ...         % Frequency constraints
    'PassbandFrequency1',450, ...
    'PassbandFrequency2',600, ...
```

```

'StopbandFrequency2',650, ...
'StopbandAttenuation1',40, ... % Magnitude constraints
'PassbandRipple',1, ...
'StopbandAttenuation2',50, ...
'DesignMethod','ellip', ... % Design method
'MatchExactly','passband', ... % Design method options
'SampleRate',2000) % Sample rate

```

- If you omit 'FilterOrder' (when required), or any of the frequency constraints, `designfilt` throws an error.
- If you omit the magnitude constraints, `designfilt` uses default values.
- If you omit 'DesignMethod', `designfilt` uses the default design method for the specification set.
- If you omit the design method options, `designfilt` uses the defaults for the design method of choice.
- If you omit 'SampleRate', `designfilt` sets it to 2 Hz.

| Filter Order Argument Names | Frequency Constraint Argument Names | Magnitude Constraint Argument Names | 'DesignMethod' Argument Values | Design Option Argument Names |
|-----------------------------|-------------------------------------|-------------------------------------|--------------------------------|------------------------------|
| N/A (Minimum-order design)  | 'StopbandFrequency'                 | 'StopbandAttenuation1'              | 'butter' (default)             | 'MatchExactly'               |
|                             | 'PassbandFrequency'                 | 'PassbandRipple'                    | 'cheby1'                       | 'MatchExactly'               |
|                             | 'PassbandFrequency'                 | 'StopbandAttenuation2'              | 'cheby2'                       | 'MatchExactly'               |
|                             | 'StopbandFrequency'                 |                                     | 'ellip'                        | 'MatchExactly'               |
| 'FilterOrder'               | 'HalfPowerFrequency'                | N/A                                 | 'butter'                       | N/A                          |
|                             | 'HalfPowerFrequency'                |                                     |                                |                              |
| 'FilterOrder'               | 'PassbandFrequency'                 | 'PassbandRipple'                    | 'cheby1'                       | N/A                          |
|                             | 'PassbandFrequency'                 |                                     |                                |                              |
| 'FilterOrder'               | 'PassbandFrequency'                 | 'StopbandAttenuation1'              | 'ellip'                        | N/A                          |
|                             | 'PassbandFrequency'                 | 'PassbandRipple'                    |                                |                              |
|                             |                                     | 'StopbandAttenuation2'              |                                |                              |

| Filter Order Argument Names | Frequency Constraint Argument Names | Magnitude Constraint Argument Names | 'DesignMethod' Argument Values | Design Option Argument Names |
|-----------------------------|-------------------------------------|-------------------------------------|--------------------------------|------------------------------|
| 'FilterOrder'               | 'StopbandFreque<br>'StopbandFreque  | 'StopbandAttenu                     | 'cheby2'                       | N/A                          |

**'bandstopfir'**

Choose this option to design a finite impulse response (FIR) bandstop filter. This example uses the fourth specification set from the following table.

```
d = designfilt('bandstopfir', ...           % Response type
    'FilterOrder',32, ...                   % Filter order
    'PassbandFrequency1',400, ...          % Frequency constraints
    'StopbandFrequency1',500, ...
    'StopbandFrequency2',700, ...
    'PassbandFrequency2',850, ...
    'DesignMethod','ls', ...              % Design method
    'PassbandWeight1',1, ...               % Design method options
    'StopbandWeight',3, ...
    'PassbandWeight2',5, ...
    'SampleRate',2000)                    % Sample rate
```

- If you omit 'FilterOrder' (when required), or any of the frequency constraints, `designfilt` throws an error.
- If you omit the magnitude constraints, `designfilt` uses default values.
- If you omit 'DesignMethod', `designfilt` uses the default design method for the specification set.
- If you omit the design method options, `designfilt` uses the defaults for the design method of choice.
- If you omit 'SampleRate', `designfilt` sets it to 2 Hz.

| Filter Order Argument Names | Frequency Constraint Argument Names | Magnitude Constraint Argument Names | 'DesignMethod' Argument Values           | Design Option Argument Names |
|-----------------------------|-------------------------------------|-------------------------------------|--|------------------------------|
| N/A (Minimum-order design)  | 'PassbandFreque<br>'StopbandFreque  | 'PassbandRipple<br>'StopbandAttenu  | 'equiripple'<br>(default)<br>'kaiserwin' | N/A<br>'ScalePassband'       |



| Filter Order Argument Names | Frequency Constraint Argument Names  | Magnitude Constraint Argument Names                           | 'DesignMethod' Argument Values     | Design Option Argument Names   |
|-----------------------------|--|---|------------------------------------|--|
|                             | 'StopbandFrequency1'<br>'PassbandFrequency1'   | 'PassbandRipple'  |                                    |  |
| 'FilterOrder'               | 'CutoffFrequency1'<br>'CutoffFrequency2'   | N/A   | 'window'                           | 'Window'<br>'ScalePassband'  |
| 'FilterOrder'               | 'CutoffFrequency1'<br>'CutoffFrequency2'   | 'PassbandRipple'<br>'StopbandAttenuation'<br>'PassbandRipple' | 'cls'                              | 'PassbandOffset'<br>'ZeroPhase'  |
| 'FilterOrder'               | 'PassbandFrequency1'<br>'StopbandFrequency1'<br>'StopbandFrequency2'<br>'PassbandFrequency2' | N/A   | 'equiripple' (default)<br><br>'ls' | 'PassbandWeight1'<br>'StopbandWeight1'<br>'PassbandWeight2'<br>'PassbandWeight1'<br>'StopbandWeight1'<br>'PassbandWeight2' |

### 'bandstopiir'

Choose this option to design an infinite impulse response (IIR) bandstop filter. This example uses the first specification set from the following table.

```
d = designfilt('bandstopiir', ...           % Response type
    'PassbandFrequency1',400, ...         % Frequency constraints
    'StopbandFrequency1',500, ...
    'StopbandFrequency2',700, ...
    'PassbandFrequency2',850, ...
    'PassbandRipple1',1, ...             % Magnitude constraints
    'StopbandAttenuation',55, ...
    'PassbandRipple2',1, ...
    'DesignMethod','ellip', ...         % Design method
    'MatchExactly','both', ...         % Design method options
    'SampleRate',2000)                 % Sample rate
```

- If you omit 'FilterOrder' (when required), or any of the frequency constraints, `designfilt` throws an error.
- If you omit the magnitude constraints, `designfilt` uses default values.
- If you omit 'DesignMethod', `designfilt` uses the default design method for the specification set.
- If you omit the design method options, `designfilt` uses the defaults for the design method of choice.
- If you omit 'SampleRate', `designfilt` sets it to 2 Hz.

| Filter Order Argument Names | Frequency Constraint Argument Names | Magnitude Constraint Argument Names | 'DesignMethod' Argument Values | Design Option Argument Names |
|-----------------------------|-------------------------------------|-------------------------------------|--------------------------------|------------------------------|
| N/A (Minimum-order design)  | 'PassbandFreque                     | 'PassbandRipple                     | 'butter' (default)             | 'MatchExactly'               |
|                             | 'StopbandFreque                     | 'StopbandAttenuation                | 'cheby1'                       | 'MatchExactly'               |
|                             | 'StopbandFreque                     | 'PassbandRipple2                    | 'cheby2'                       | 'MatchExactly'               |
|                             | 'PassbandFreque                     |                                     | 'ellip'                        | 'MatchExactly'               |
| 'FilterOrder'               | 'HalfPowerFreque                    | N/A                                 | 'butter'                       | N/A                          |
|                             | 'HalfPowerFreque                    |                                     |                                |                              |
| 'FilterOrder'               | 'PassbandFreque                     | 'PassbandRipple                     | 'cheby1'                       | N/A                          |
|                             | 'PassbandFreque                     |                                     |                                |                              |
| 'FilterOrder'               | 'PassbandFreque                     | 'PassbandRipple                     | 'ellip'                        | N/A                          |
|                             | 'PassbandFreque                     | 'StopbandAttenu                     |                                |                              |
| 'FilterOrder'               | 'StopbandFreque                     | 'StopbandAttenu                     | 'cheby2'                       | N/A                          |
|                             | 'StopbandFreque                     |                                     |                                |                              |

**'differentiatorfir'**

Choose this option to design a finite impulse response (FIR) differentiator filter. This example uses the second specification set from the following table.

```
d = designfilt('differentiatorfir', ... % Response type
```

```

'FilterOrder',42, ...           % Filter order
'PassbandFrequency',400, ...   % Frequency constraints
'StopbandFrequency',500, ...
'DesignMethod','equiripple', ... % Design method
'PassbandWeight',1, ...       % Design method options
'StopbandWeight',4, ...
'SampleRate',2000)           % Sample rate

```

- If you omit 'FilterOrder', or any of the frequency constraints when designing a partial-band differentiator, `designfilt` throws an error.
- If you omit 'DesignMethod', `designfilt` uses the default design method for the specification set.
- If you omit the design method options, `designfilt` uses the defaults for the design method of choice.
- If you omit 'SampleRate', `designfilt` sets it to 2 Hz.

| Filter Order Argument Names | Frequency Constraint Argument Names  | Magnitude Constraint Argument Names | 'DesignMethod' Argument Values | Design Option Argument Names         |
|-----------------------------|--------------------------------------|-------------------------------------|--------------------------------|--------------------------------------|
| 'FilterOrder'               | N/A                                  | N/A                                 | 'equiripple'<br>(default)      | N/A                                  |
|                             |                                      |                                     | 'ls'                           | N/A                                  |
| 'FilterOrder'               | 'PassbandFreque'<br>'StopbandFreque' | N/A                                 | 'equiripple'<br>(default)      | 'PassbandWeight'<br>'StopbandWeight' |
|                             |                                      |                                     | 'ls'                           | N/A                                  |

### 'hilbertfir'

Choose this option to design a finite impulse response (FIR) Hilbert transformer filter. This example uses the specification set from the following table.

```

d = designfilt('hilbertfir', ...   % Response type
'FilterOrder',12, ...           % Filter order
'TransitionWidth',400, ...      % Frequency constraints
'DesignMethod','ls', ...       % Design method
'SampleRate',2000)             % Sample rate

```

- If you omit 'FilterOrder' or 'TransitionWidth', `designfilt` throws an error.

- If you omit 'DesignMethod', designfilt uses the default design method for Hilbert transformers.
- If you omit 'SampleRate', designfilt sets it to 2 Hz.

| Filter Order Argument Names | Frequency Constraint Argument Names | Magnitude Constraint Argument Names | 'DesignMethod' Argument Values | Design Option Argument Names |
|-----------------------------|-------------------------------------|-------------------------------------|--------------------------------|------------------------------|
| 'FilterOrder'               | 'TransitionWidth'                   | N/A                                 | 'equiripple' (default)         | N/A                          |
|                             |                                     |                                     | 'ls'                           | N/A                          |

**'arbmagfir'**

Choose this option to design a finite impulse response (FIR) filter of arbitrary magnitude response. This example uses the second specification set from the following table.

```
d = designfilt('arbmagfir', ...           % Response type
    'FilterOrder',88, ...                 % Filter order
    'NumBands',4, ...                     % Frequency constraints
    'BandFrequencies1',[0 20], ...
    'BandFrequencies2',[25 40], ...
    'BandFrequencies3',[45 65], ...
    'BandFrequencies4',[70 100], ...
    'BandAmplitudes1',[2 2], ...         % Magnitude constraints
    'BandAmplitudes2',[0 0], ...
    'BandAmplitudes3',[1 1], ...
    'BandAmplitudes4',[0 0], ...
    'DesignMethod','ls', ...             % Design method
    'BandWeights1',[1 1]/10, ...         % Design method options
    'BandWeights2',[3 1], ...
    'BandWeights3',[2 4], ...
    'BandWeights4',[5 1], ...
    'SampleRate',200)                   % Sample rate
```

- If you omit 'FilterOrder', or any of the frequency or magnitude constraints, designfilt throws an error.
- If you omit 'DesignMethod', designfilt uses the default design method for the specification set.
- If you omit the design method options, designfilt uses the defaults for the design method of choice.
- If you omit 'SampleRate', designfilt sets it to 2 Hz.

| Filter Order Argument Names | Frequency Constraint Argument Names | Magnitude Constraint Argument Names | 'DesignMethod' Argument Values | Design Option Argument Names |
|-----------------------------|-------------------------------------|-------------------------------------|--------------------------------|------------------------------|
| 'FilterOrder'               | 'Frequencies'                       | 'Amplitudes'                        | 'freqsamp'<br>(default)        | 'Window'                     |
|                             |                                     |                                     | 'equiripple'                   | 'Weights'                    |
|                             |                                     |                                     | 'ls'                           | 'Weights'                    |
| 'FilterOrder'<br>'NumBands' | 'BandFrequencies'                   | 'BandAmplitudes'                    | 'equiripple'<br>(default)      | 'BandWeights1'               |
|                             | ...                                 | ...                                 |                                | ...                          |
|                             | 'BandFrequenciesN'                  | 'BandAmplitudesN'                   |                                | 'BandWeightsN'               |
|                             |                                     |                                     | 'ls'                           | 'BandWeights1'               |
|                             |                                     |                                     |                                | ...                          |
|                             |                                     |                                     |                                | 'BandWeightsN'               |

Data Types: char

### **d** – Digital filter

`digitalFilter` object

Digital filter, specified as a `digitalFilter` object generated by `designfilt`. Use this input to change the specifications of an existing `digitalFilter`.

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Not all combinations of `Name,Value` pairs are valid. The valid combinations depend on the filter response that you need and on the frequency and magnitude constraints of your design.

Example: `'FilterOrder',20,'CutoffFrequency',0.4` suffices to specify a lowpass FIR filter.

## Filter Order

### 'FilterOrder' — Filter order

positive integer scalar

Filter order, specified as the comma-separated pair consisting of 'FilterOrder' and a positive integer scalar.

Data Types: double

### 'NumeratorOrder' — Numerator order

positive integer scalar

Numerator order of an IIR design, specified as the comma-separated pair consisting of 'NumeratorOrder' and a positive integer scalar.

Data Types: double

### 'DenominatorOrder' — Denominator order

positive integer scalar

Denominator order of an IIR design, specified as the comma-separated pair consisting of 'DenominatorOrder' and a positive integer scalar.

Data Types: double

## Frequency Constraints

### 'PassbandFrequency', 'PassbandFrequency1', 'PassbandFrequency2' — Passband frequency

positive scalar

Passband frequency, specified as the comma-separated pair consisting of 'PassbandFrequency' and a positive scalar. The frequency value must be within the Nyquist range.

'PassbandFrequency1' is the lower passband frequency for a bandpass or bandstop design.

'PassbandFrequency2' is the higher passband frequency for a bandpass or bandstop design.

Data Types: double

**'StopbandFrequency', 'StopbandFrequency1', 'StopbandFrequency2' – Stopband frequency**

positive scalar

Stopband frequency, specified as the comma-separated pair consisting of 'StopbandFrequency' and a positive scalar. The frequency value must be within the Nyquist range.

'StopbandFrequency1' is the lower stopband frequency for a bandpass or bandstop design

'StopbandFrequency2' is the higher stopband frequency for a bandpass or bandstop design.

Data Types: double

**'CutoffFrequency', 'CutoffFrequency1', 'CutoffFrequency2' – 6-dB frequency**

positive scalar

6-dB frequency, specified as the comma-separated pair consisting of 'CutoffFrequency' and a positive scalar. The frequency value must be within the Nyquist range.

'CutoffFrequency1' is the lower 6-dB frequency for a bandpass or bandstop design.

'CutoffFrequency2' is the higher 6-dB frequency for a bandpass or bandstop design.

Data Types: double

**'HalfPowerFrequency', 'HalfPowerFrequency1', 'HalfPowerFrequency2' – 3-dB frequency**

positive scalar

3-dB frequency, specified as the comma-separated pair consisting of 'HalfPowerFrequency' and a positive scalar. The frequency value must be within the Nyquist range.

'HalfPowerFrequency1' is the lower 3-dB frequency for a bandpass or bandstop design.

'HalfPowerFrequency2' is the higher 3-dB frequency for a bandpass or bandstop design.

Data Types: double

**'TransitionWidth' — Width of transition region**

positive scalar

Width of the transition region between passband and stopband for a Hilbert transformer, specified as the comma-separated pair consisting of 'TransitionWidth' and a positive scalar.

Data Types: double

**'Frequencies' — Response frequencies**

vector

Response frequencies, specified as the comma-separated pair consisting of 'Frequencies' and a vector. Use this variable to list the frequencies at which a filter of arbitrary magnitude response has desired amplitudes. The frequencies must be monotonically increasing and lie within the Nyquist range. The first element of the vector must be either 0 or  $-f_s/2$ , where  $f_s$  is the sample rate, and its last element must be  $f_s/2$ . If you do not specify a sample rate, `designfilt` uses the default value of 2 Hz.

Data Types: double

**'NumBands' — Number of bands**

positive integer scalar

Number of bands in a multiband design, specified as the comma-separated pair consisting of 'NumBands' and a positive integer scalar not greater than 10.

Data Types: double

**'BandFrequencies1', '...', 'BandFrequenciesN' — Multiband response frequencies**

vectors

Multiband response frequencies, specified as comma-separated pairs consisting of 'BandFrequencies $i$ ' and a numeric vector. 'BandFrequencies $i$ ', where  $i$  runs from 1 through NumBands, is a vector containing the frequencies at which the  $i$ th band of a multiband design has the desired values, 'BandAmplitudes $i$ '. NumBands can be at most 10. The frequencies must lie within the Nyquist range and must be specified in monotonically increasing order. Adjacent frequency bands must have the same amplitude at their junction.

Data Types: double



## Magnitude Constraints

**'PassbandRipple', 'PassbandRipple1', 'PassbandRipple2' — Passband ripple**  
1 (default) | positive scalar

Passband ripple, specified as the comma-separated pair consisting of 'PassbandRipple' and a positive scalar expressed in decibels.

'PassbandRipple1' is the lower-band passband ripple for a bandstop design.

'PassbandRipple2' is the higher-band passband ripple for a bandstop design.

Data Types: double

**'StopbandAttenuation', 'StopbandAttenuation1', 'StopbandAttenuation2' — Stopband attenuation**  
60 (default) | positive scalar

Stopband attenuation, specified as the comma-separated pair consisting of 'StopbandAttenuation' and a positive scalar expressed in decibels.

'StopbandAttenuation1' is the lower-band stopband attenuation for a bandpass design.

'StopbandAttenuation2' is the higher-band stopband attenuation for a bandpass design.

Data Types: double

**'Amplitudes' — Desired response amplitudes**  
vector

Desired response amplitudes of an arbitrary magnitude response filter, specified as the comma-separated pair consisting of 'Amplitudes' and a vector. Express the amplitudes in linear units. The vector must have the same length as 'Frequencies'.

Data Types: double

**'BandAmplitudes1', '...', 'BandAmplitudesN' — Multiband response amplitudes**  
vectors

Multiband response amplitudes, specified as comma-separated pairs consisting of 'BandAmplitudes $i$ ' and a numeric vector. 'BandAmplitudes $i$ ', where  $i$  runs from

1 through NumBands, is a vector containing the desired amplitudes in the *i*th band of a multiband design. NumBands can be at most 10. Express the amplitudes in linear units. 'BandAmplitudes*i*' must have the same length as 'BandFrequencies*i*'. Adjacent frequency bands must have the same amplitude at their junction.

Data Types: double

## Design Method

### 'DesignMethod' — Design method

'butter' | 'cheby1' | 'cheby2' | 'cls' | 'ellip' | 'equiripple' |  
'freqsamp' | 'kaiserwin' | 'ls' | 'maxflat' | 'window'

Design method, specified as the comma-separated pair consisting of 'DesignMethod' and a string. The choice of design method depends on the set of frequency and magnitude constraints that you specify.

- 'butter' designs a Butterworth IIR filter. Butterworth filters have a smooth monotonic frequency response that is maximally flat in the passband. They sacrifice rolloff steepness for flatness.
- 'cheby1' designs a Chebyshev type I IIR filter. Chebyshev type I filters have a frequency response that is equiripple in the passband and maximally flat in the stopband. Their passband ripple increases with increasing rolloff steepness.
- 'cheby2' designs a Chebyshev type II IIR filter. Chebyshev type II filters have a frequency response that is maximally flat in the passband and equiripple in the stopband.
- 'cls' designs an FIR filter using constrained least squares. The method minimizes the discrepancy between a specified arbitrary piecewise-linear function and the filter's magnitude response. At the same time, it lets you set constraints on the passband ripple and stopband attenuation.
- 'ellip' designs an elliptic IIR filter. Elliptic filters have a frequency response that is equiripple in both passband and stopband.
- 'equiripple' designs an equiripple FIR filter using the Parks-McClellan algorithm. Equiripple filters have a frequency response that minimizes the maximum ripple magnitude over all bands.
- 'freqsamp' designs an FIR filter of arbitrary magnitude response by sampling the frequency response uniformly and taking the inverse Fourier transform.

- `'kaiserwin'` designs an FIR filter using the Kaiser window method. The method truncates the impulse response of an ideal filter and uses a Kaiser window to attenuate the resulting truncation oscillations.
- `'ls'` designs an FIR filter using least squares. The method minimizes the discrepancy between a specified arbitrary piecewise-linear function and the filter's magnitude response.
- `'maxflat'` designs a maximally flat FIR filter. These filters have a smooth monotonic frequency response that is maximally flat in the passband.
- `'window'` designs an FIR filter using the windowing method. Window-based designs truncate the impulse response of an ideal filter and use a window function to attenuate the resulting truncation oscillations.

Data Types: char

## Design Method Options

### 'Window' — Window

vector | string | function handle | cell array

Window, specified as the comma-separated pair consisting of `'Window'` and a vector of length  $N + 1$ , where  $N$  is the filter order. `'Window'` can also be paired with a string or function handle that specifies the function used to generate the window. Any such function must take  $N + 1$  as first input. Additional inputs can be passed by specifying a cell array. By default, `'Window'` is an empty vector for the `'freqsamp'` design method and `@hamming` for the `'window'` design method.

For a list of available windows, see “Windows”.

Example: `'Window', hann(N+1)` and `'Window', (1 - cos(2*pi*(0:N)'/N))/2` both specify a Hann window to use with a filter of order  $N$ .

Example: `'Window', 'hamming'` specifies a Hamming window of the required order.

Example: `'Window', @mywindow` lets you define your own window function.

Example: `'Window', {@kaiser, 0.5}` specifies a Kaiser window of the required order with shape parameter 0.5.

Data Types: double | char | function\_handle | cell

### 'MatchExactly' — Band to match exactly

'stopband' | 'passband' | 'both'

Band to match exactly, specified as the comma-separated pair consisting of `'MatchExactly'` and either `'stopband'`, `'passband'`, or `'both'`. `'both'` is available only for the elliptic design method, where it is the default. `'stopband'` is the default for the `'butter'` and `'cheby2'` methods. `'passband'` is the default for `'cheby1'`.

Data Types: char

#### **'PassbandOffset' — Passband offset**

0 (default) | positive scalar

Passband offset, specified as the comma-separated pair consisting of `'PassbandOffset'` and a positive scalar expressed in decibels. `'PassbandOffset'` specifies the filter gain in the passband.

Example: `'PassbandOffset', 0` results in a filter with unit gain in the passband.

Example: `'PassbandOffset', 2` results in a filter with a passband gain of 2 dB or 1.259.

Data Types: double

#### **'ScalePassband' — Scale passband**

true (default) | false

Scale passband, specified as the comma-separated pair consisting of `'ScalePassband'` and a logical scalar. When you set `'ScalePassband'` to `true`, the passband is scaled, after windowing, so that the filter has unit gain at zero frequency.

Example: `'Window', {@kaiser, 0.1}, 'ScalePassband', true` help specify a filter whose magnitude response at zero frequency is exactly 0 dB. This is not the case when you specify `'ScalePassband', false`. To verify, visualize the filter with `fvtool` and zoom in.

Data Types: logical

#### **'ZeroPhase' — Zero phase**

false (default) | true

Zero phase, specified as the comma-separated pair consisting of `'ZeroPhase'` and a logical scalar. When you set `'ZeroPhase'` to `true`, the zero-phase response of the resulting filter is always positive. This lets you perform spectral factorization on the result and obtain a minimum-phase filter from it.

Data Types: logical

**'PassbandWeight', 'PassbandWeight1', 'PassbandWeight2' — Passband optimization weight**

1 (default) | positive scalar

Passband optimization weight, specified as the comma-separated pair consisting of 'PassbandWeight' and a positive scalar.

'PassbandWeight1' is the lower-band passband optimization weight for a bandstop FIR design.

'PassbandWeight2' is the higher-band passband optimization weight for a bandstop FIR design.

Data Types: double

**'StopbandWeight', 'StopbandWeight1', 'StopbandWeight2' — Stopband optimization weight**

1 (default) | positive scalar

Stopband optimization weight, specified as the comma-separated pair consisting of 'StopbandWeight' and a positive scalar.

'StopbandWeight1' is the lower-band stopband optimization weight for a bandpass FIR design.

'StopbandWeight2' is the higher-band stopband optimization weight for a bandpass FIR design.

Data Types: double

**'Weights' — Optimization weights**

1 (default) | positive scalar | vector

Optimization weights, specified as the comma-separated pair consisting of 'Weights' and a positive scalar or a vector of the same length as 'Amplitudes'.

Data Types: double

**'BandWeights1', '...', 'BandWeightsN' — Multiband weights**

1 (default) | positive scalar | vectors

Multiband weights, specified as comma-separated pairs consisting of 'BandWeights $i$ ' and a set of positive scalars or of vectors. 'BandWeights $i$ ', where  $i$  runs from 1 through NumBands, is a scalar or vector containing the optimization weights of the  $i$ th band of a

multiband design. If specified as a vector, 'BandWeights $i$ ' must have the same length as 'BandAmplitudes $i$ '.

Data Types: `double`

## Sample Rate

'**SampleRate**' — Sample rate

2 (default) | positive scalar

Sample rate, specified as the comma-separated pair consisting of 'SampleRate' and a positive scalar expressed in hertz. To work with normalized frequencies, set 'SampleRate' to 2, or simply omit it.

Data Types: `double`

## Output Arguments

**d** — Digital filter

`digitalFilter` object

Digital filter, returned as a `digitalFilter` object.

## More About

### Filter Design Assistant

If you specify an incomplete or inconsistent set of design parameters, `designfilt` offers to open a Filter Design Assistant.

(In the argument description for `resp` there is a complete list of valid specification sets for all available response types.)

The assistant behaves differently if you call `designfilt` at the command line or within a script or function.

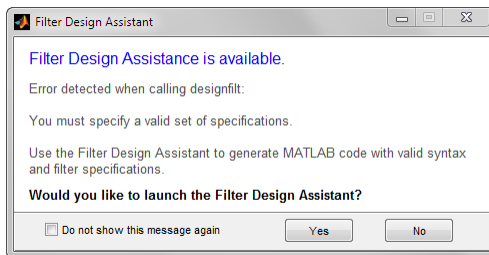
### Filter Design Assistant at the Command Line

You are given a signal sampled at 2 kHz. You are asked to design a lowpass FIR filter that suppresses frequency components higher than 650 Hz. The “cutoff frequency” sounds

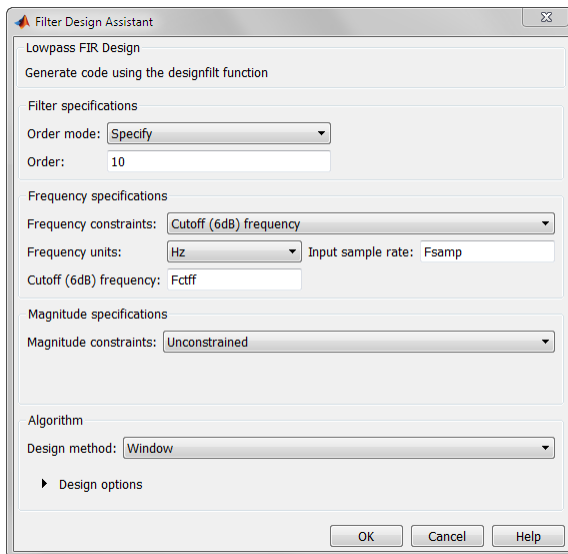
like a good candidate for a specification parameter. At the MATLAB command line, you type the following.

```
Fsamp = 2e3;
Fctff = 650;
d = designfilt('lowpassfir','CutoffFrequency',Fctff, ...
              'SampleRate',Fsamp);
```

Something seems to be amiss because this dialog box appears on your screen.



You click **Yes** and get a new dialog box that offers to generate code. You see that the variables you defined before have been inserted where expected.



After exploring some of the options offered, you decide to test the corrected filter. You click **OK** and get the following code on the command line.

```
dee = designfilt('lowpassfir', 'FilterOrder', 10, ...  
                'CutoffFrequency', Fctff, 'SampleRate', Fsamp);
```

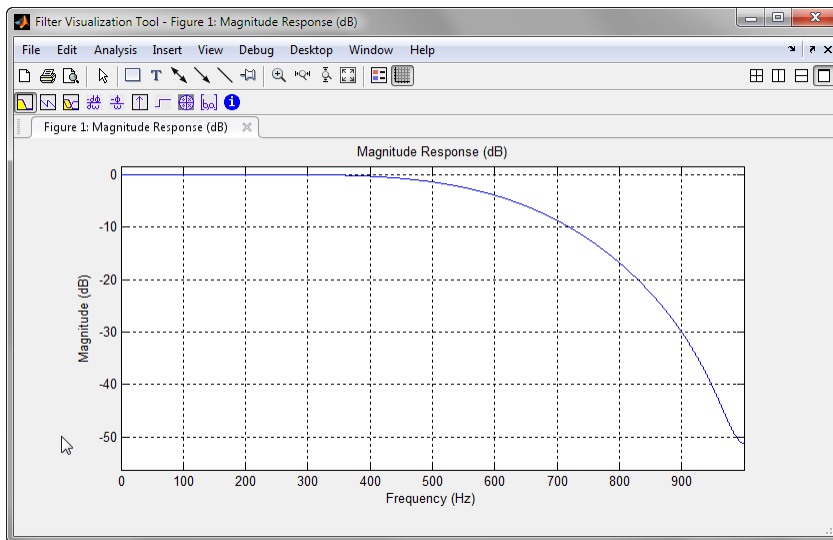
Typing the name of the filter reiterates the information from the dialog box.

dee

```
dee =  
digitalFilter with properties:  
  
    Coefficients: [1x11 double]  
Specifications:  
    FrequencyResponse: 'lowpass'  
    ImpulseResponse: 'fir'  
        SampleRate: 2000  
        FilterOrder: 10  
    CutoffFrequency: 650  
    DesignMethod: 'window'  
Use fvtool to visualize filter  
Use filter function to filter data
```

You invoke `fvtool` and get a plot of `dee`'s frequency response.

```
fvtool(dee)
```



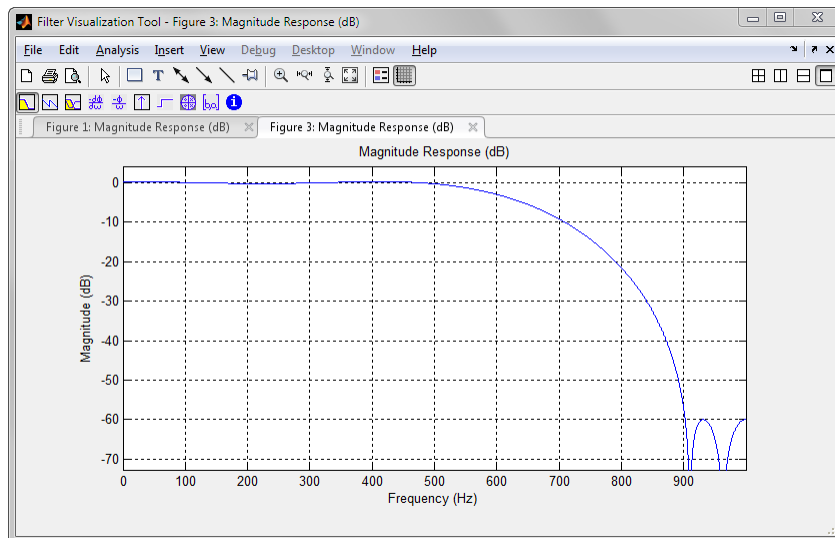


The cutoff does not look particularly sharp. The response is above 40 dB for most frequencies. You remember that the assistant had an option to set up a “magnitude constraint” called the “stopband attenuation”. Open the assistant by calling `designfilt` with the filter name as input.

```
designfilt(dee)
```

Click the **Magnitude constraints** drop-down menu and select **Passband ripple** and **stopband attenuation**. You see that the design method has changed from **Window** to **FIR constrained least-squares**. The default value for the attenuation is 60 dB, which is higher than 40. Click **OK** and visualize the resulting filter.

```
dee = designfilt('lowpassfir', 'FilterOrder', 10, ...
                'CutoffFrequency', Fctff, ...
                'PassbandRipple', 1, 'StopbandAttenuation', 60, ...
                'SampleRate', Fsamp);
fvtool(dee)
```

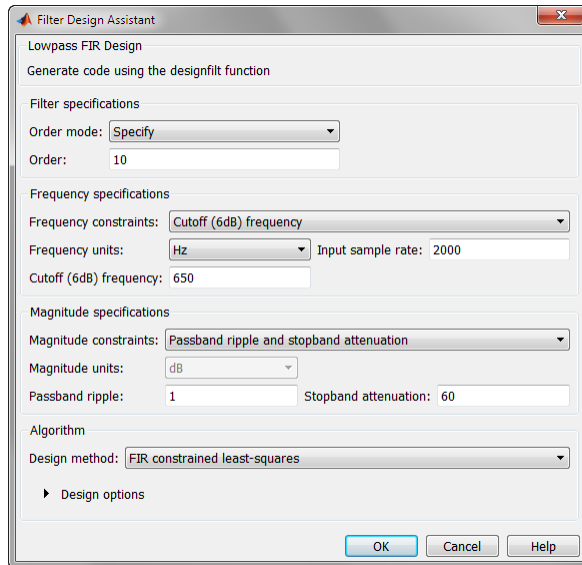


The cutoff still does not look sharp. The attenuation is indeed 60 dB, but for frequencies above 900 Hz.

Again invoke `designfilt` with your filter as input.

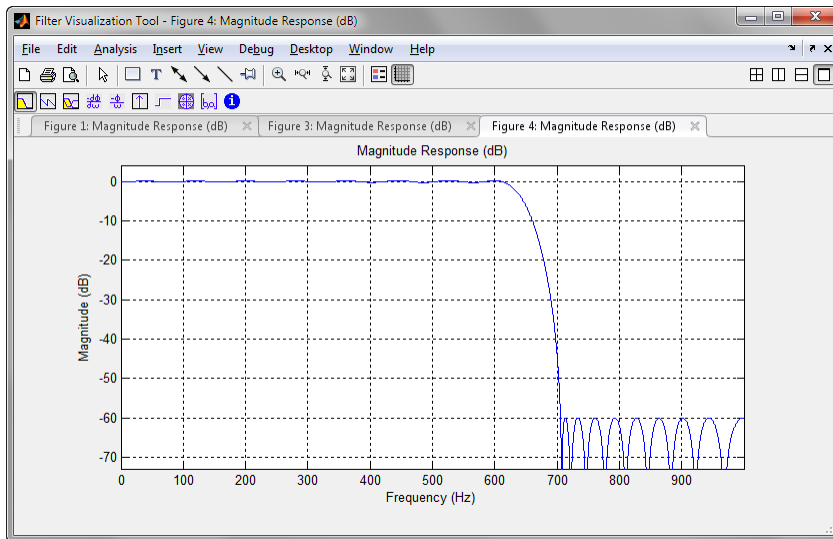
```
designfilt(dee)
```

The assistant reappears.



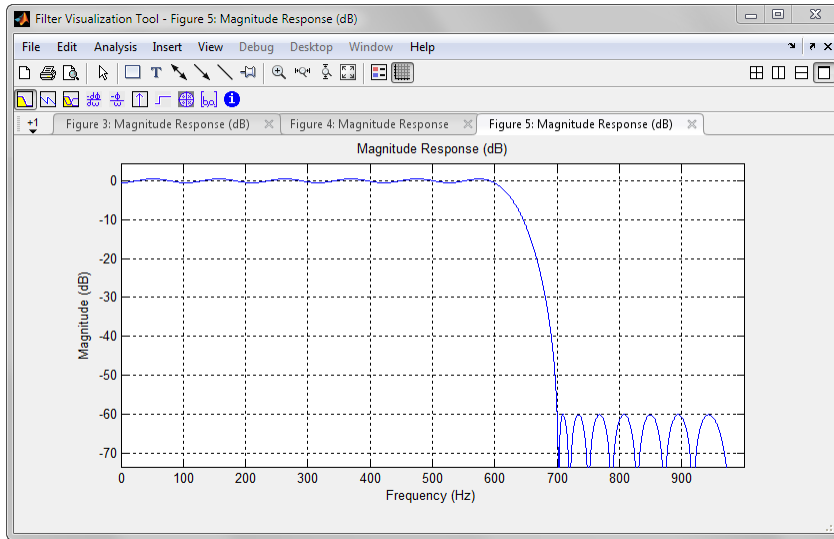
To narrow the distinction between accepted and rejected frequencies, increase the order of the filter or change **Frequency constraints** from **Cutoff (6dB) frequency** to **Passband and stopband frequencies**. If you change the filter order from 10 to 50, you get a sharper filter.

```
dee = designfilt('lowpassfir', 'FilterOrder', 50, ...
                'CutoffFrequency', 650, ...
                'PassbandRipple', 1, 'StopbandAttenuation', 60, ...
                'SampleRate', 2000);
fvtool(dee)
```



A little experimentation shows that you can obtain a similar filter by setting the passband and stopband frequencies respectively to 600 Hz and 700 Hz.

```
dee = designfilt('lowpassfir', 'PassbandFrequency', 600, ...  
                'StopbandFrequency', 700, ...  
                'PassbandRipple', 1, 'StopbandAttenuation', 60, ...  
                'SampleRate', 2000);  
fvtool(dee)
```



## Filter Design Assistant in a Script or Function

You are given a signal sampled at 2 kHz. You are asked to design a highpass filter that stops frequencies below 700 Hz. You don't care about the phase of the signal, and you need to work with a low-order filter. Thus an IIR filter seems adequate. You are not sure what filter order is best, so you write a function that accepts the order as input. Open the MATLAB Editor and create the file.

```
function dataOut = hipassfilt(N,dataIn)
hpFilter = designfilt('highpassiir','FilterOrder',N);
dataOut = filter(hpFilter,dataIn);
end
```

To test your function, create a signal composed of two sinusoids with frequencies 500 and 800 Hz and generate samples for 0.1 s. A 5th-order filter seems reasonable as an initial guess. Create a script called `driveHPfilt.m`.

```
% script driveHPfilt.m
Fsamp = 2e3;
Fsm = 500;
Fbg = 800;
t = 0:1/Fsamp:0.1;
sgin = sin(2*pi*Fsm*t)+sin(2*pi*Fbg*t);
Order = 5;
```

```
sgout = hipassfilt(Order,sgin);
```

When you run the script at the command line, you get an error message.

```
>> driveHPfilt
Error using designfilt (line 457)
```

[Click here](#) to launch an assistant that can correct your code.

You have specified too few parameters for 'highpassiir'.

The following are valid parameter sets that are close to your inputs:

- FilterOrder, HalfPowerFrequency
- FilterOrder, PassbandFrequency, PassbandRipple
- FilterOrder, PassbandFrequency, StopbandAttenuation, PassbandRipple
- FilterOrder, StopbandFrequency, StopbandAttenuation

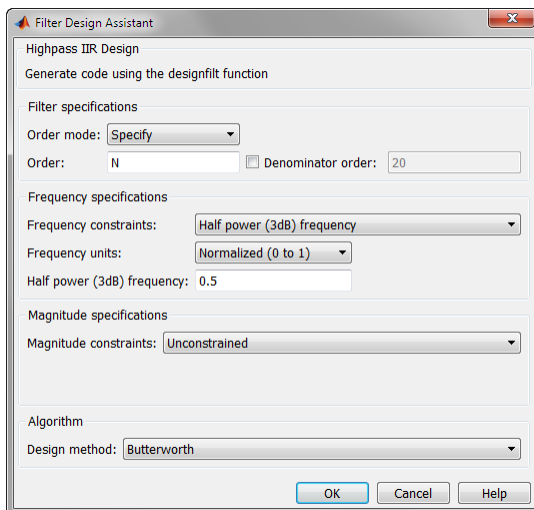
```
Error in hipassfilt (line 2)
```

```
hpFilter = designfilt('highpassiir','FilterOrder',N);
```

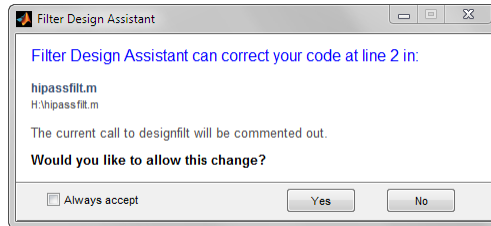
```
Error in driveHPfilt (line 7)
```

```
sgout = hipassfilt(Order,sgin);
```

The error message gives you the choice of opening an assistant to correct the MATLAB code. Click [Click here](#) to get the Filter Design Assistant on your screen.



You see the problem: You did not specify the frequency constraint. You also forgot to set a sample rate. After experimenting, you find that you can specify **Frequency units** as Hz, **Passband frequency** equal to 700 Hz, and **Input Fs** equal to 2000 Hz. The **Design method** changes from Butterworth to Chebyshev type I. You click **OK** and get the following.



The assistant has correctly identified the file where you call `designfilt`. Click **Yes** to accept the change. The function has the corrected MATLAB code.

```
function dataOut = hipassfilt(N,dataIn)
% hpFilter = designfilt('highpassiir','FilterOrder',N);
hpFilter = designfilt('highpassiir', 'FilterOrder', N, ...
    'PassbandFrequency', 700, 'PassbandRipple', 1, ...
    'SampleRate', 2000);
dataOut = filter(hpFilter,dataIn);
end
```

You can now run the script with different values of the filter order. Depending on your design constraints, you can change your specification set.

### Filter Design Assistant Preferences

You can set `designfilt` to never offer the Filter Design Assistant. This action sets a MATLAB preference that can be unset with `setpref`:

- Use `setpref('dontshowmeagain','filterDesignAssistant',false)` to be offered the assistant every time. With this command, you can get the assistant again after having disabled it.
- Use `setpref('dontshowmeagain','filterDesignAssistant',true)` to disable the assistant permanently. You can also click **Do not show this message again** in the initial dialog box.

You can set `designfilt` to always correct faulty specifications without asking. This action sets a MATLAB preference that can be unset by using `setpref`:

- Use `setpref('dontshowmeagain','filterDesignAssistantCodeCorrection',false)` to have `designfilt` correct your MATLAB code without asking for confirmation. You can also click **Always accept** in the confirmation dialog box.
- Use `setpref('dontshowmeagain','filterDesignAssistantCodeCorrection',true)` to ensure that `designfilt` corrects your MATLAB code only when you confirm you want the changes. With this command, you can undo the effect of having clicked **Always accept** in the confirmation dialog box.

## Troubleshooting

There are some instances in which, given an invalid set of specifications, `designfilt` does not offer a Filter Design Assistant, either through a dialog box or through a link in an error message.

- You are not offered an assistant if you use code-section evaluation, either from the MATLAB Toolstrip or by pressing **Ctrl+Enter**. (See “Divide Your File into Code Sections” for more information.)
- You are not offered an assistant if your code has multiple calls to `designfilt`, at least one of those calls is incorrect, and
  - You paste the code on the command line and execute it by pressing **Enter**.
  - You select the code in the Editor and execute it by pressing **F9**.
- You are not offered an assistant if you run `designfilt` using an anonymous function. (See “Anonymous Functions” for more information.) For example, this input offers an assistant.

```
d = designfilt('lowpassfir','CutoffFrequency',0.6)
```

This input does not.

```
myFilterDesigner = @designfilt;
```

```
d = myFilterDesigner('lowpassfir','CutoffFrequency',0.6)
```

- You are not offered an assistant if you run `designfilt` using `eval`. For example, this input offers an assistant.

```
d = designfilt('lowpassfir','CutoffFrequency',0.6)
```

This input does not.

```
myFilterDesigner = ...
```

```
    sprintf('designfilt('%s','CutoffFrequency',%f)', ...
```

```
                                'lowpassfir',0.6);  
d = eval(myFilterDesigner)
```

The Filter Design Assistant requires Java<sup>®</sup> software and the MATLAB desktop to run. It is not supported if you run MATLAB with the `-nojvm`, `-nodisplay`, or `-nodesktop` options.

### See Also

`digitalFilter` | `double` | `fftfilt` | `filt2block` | `filter` | `filtfilt` | `filtord` | `firtype` | `freqz` | `fvtool` | `grpdelay` | `impz` | `impzlength` | `info` | `isallpass` | `isdouble` | `isfir` | `islinphase` | `ismaxphase` | `isminphase` | `issingle` | `isstable` | `phasedelay` | `phasez` | `single` | `ss` | `stepz` | `tf` | `zerophase` | `zpk` | `zplane`



# designmethods

Methods available for designing filter from specification object

## Syntax

```
M = designmethods(D)
M = designmethods(D, 'default')
M = designmethods(D, TYPE)
M = designmethods(D, 'full')
Ms = designmethods(D, ..., 'SystemObject', sysobjflag)
```

## Description

`M = designmethods(D)` returns the available design methods for the filter specification object, `D`, and the current value of the `Specification` property.

`M = designmethods(D, 'default')` returns the default design method for the filter specification object `D` and the current value of the `Specification` property.

`M = designmethods(D, TYPE)` returns either the `TYPE` design methods that apply to `D`. `TYPE` can be either `'FIR'` or `'IIR'`.

`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.

`Ms = designmethods(D, ..., 'SystemObject', sysobjflag)` returns the available design methods for designing filter System objects when *sysobjflag* is `true`. To use System objects, you must have the DSP System Toolbox product installed. When *sysobjflag* is `false`, the function checks methods for creating `dfilt` and `mfilt` objects, as described previously. Design methods and design options for filter System objects are not necessarily the same as those for `dfilt` and `mfilt` objects.

## Examples

Construct a lowpass filter specification object and determine the valid design methods. Obtain detailed command line help on the Chebyshev type I design method.

```
D = fdesign.lowpass('Fp,Fst,Ap,Ast',500,600,0.5,60,1e4);  
M = designmethods(D)  
help(D,M{2})
```

The last line of the example is equivalent to `help(D, 'cheby1')`.

If you have DSP System Toolbox software installed, use the `'SystemObject'`, *sysobjflag* syntax to return design methods for a filter System object:

```
Ms = designmethods(D, 'SystemObject', true);
```

## See Also

`design` | `designopts` | `fdesign`

# designopts

Valid input arguments and values for specification object and method

## Syntax

```
OPTS = designopts(D,METHOD)
```

## Description

`OPTS = designopts(D,METHOD)` returns a structure array with the default design parameters used by the design method `METHOD`. `METHOD` must be one of the strings returned by `designmethods`.

Use `help(D,METHOD)` to get a description of the design parameters.

If you have DSP System Toolbox software installed, `OPTS` has the `SystemObject` property if at least one of the structures available for that design method is supported by System objects. However, not all structures for that design method are supported by System objects.

## Examples

Create a lowpass filter with a numerator and denominator order of 10 and a 3-dB frequency of  $0.2\pi$  radians/sample. Obtain the default design parameters for a Butterworth design, and test whether the filter structure is a direct-form II biquad.

```
D = fdesign.lowpass('Nb,Na,F3dB',10,10,0.2);
OPTS = designopts(D,'butter');
if (OPTS.FilterStructure == 'df2sos')
    fprintf('The default filter structure is Direct-Form II\n');
    fprintf('with second-order sections.\n');
end
```

If you have DSP System Toolbox software installed, `OPTS` has the `SystemObject` property.

**See Also**

`design` | `designmethods` | `fdesign` | `validstructures`

# dfilt

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

Available structures for the `dfilt` object are shown below. The target block for the `block` method depends on the filter structure. Depending on the target block, the DSP System Toolbox software may be required.

| <code>dfilt.structure</code> | Description   | Coefficient Mapping Support in <code>realizemdl</code> | Target Filter Block for <code>block</code> Method |
|------------------------------|---------------|--|---|
| <code>dfilt.delay</code>     | Delay         | Not supported  | Delay<br><br>Requires DSP System Toolbox          |
| <code>dfilt.df1</code>       | Direct-form I | Supported  | Discrete Filter                                   |

| <b>dfilt.structure</b> | <b>Description</b>                               | <b>Coefficient Mapping Support in realizemdl</b> | <b>Target Filter Block for block Method</b>           |
|------------------------|--|--|---|
| dfilt.df1sos           | Direct-form I, second-order sections             | Supported  | Discrete Filter<br>Requires DSP System Toolbox        |
| dfilt.df1t             | Direct-form I transposed                         | Supported  | Discrete Filter                                       |
| dfilt.df1tsos          | Direct-form I transposed, second-order sections  | Supported  | Biquad Filter<br>Requires DSP System Toolbox          |
| dfilt.df2              | Direct-form II                                   | Supported  | Discrete Filter                                       |
| dfilt.df2sos           | Direct-form II, second-order sections            | Supported  | Discrete Filter                                       |
| dfilt.df2t             | Direct-form II transposed                        | Supported  | Discrete Filter                                       |
| dfilt.df2tsos          | Direct-form II transposed, second-order sections | Supported  | Biquad Filter<br>Requires DSP System Toolbox          |
| dfilt.dffir            | Direct-form FIR                                  | Supported  | Discrete FIR Filter                                   |
| dfilt.dffirt           | Direct-form FIR transposed                       | Supported  | Discrete FIR Filter                                   |
| dfilt.dfsymfir         | Direct-form symmetric FIR                        | Supported  | Discrete FIR Filter                                   |
| dfilt.dfasymfir        | Direct-form antisymmetric FIR                    | Supported  | Discrete FIR Filter                                   |
| dfilt.fftir            | Overlap-add FIR                                  | Not supported                                    | Overlap-Add FFT Filter<br>Requires DSP System Toolbox |
| dfilt.latticeall       | Lattice allpass                                  | Supported  | Not supported   |

| <b>dfilt.structure</b>       | <b>Description</b>                            | <b>Coefficient Mapping Support in <code>realizemdl</code></b> | <b>Target Filter Block for <code>block</code> Method</b>         |
|------------------------------|---|---|--|
| <code>dfilt.latticear</code> | Lattice autoregressive (AR)                   | Supported   | Allpole Filter<br>Requires DSP System Toolbox                    |
| <code>dfilt.latticear</code> | Lattice autoregressive moving- average (ARMA) | Supported   | Not supported  |
| <code>dfilt.latticema</code> | Lattice moving-average (MA) for maximum phase | Supported   | Not supported  |
| <code>dfilt.latticema</code> | Lattice moving-average (MA) for minimum phase | Supported   | Discrete FIR Filter  |
| <code>dfilt.statespac</code> | State-space                                   | Supported.  | Not supported  |
| <code>dfilt.scalar</code>    | Scalar gain object                            | Supported   | Gain<br>Requires DSP System Toolbox                              |
| <code>dfilt.cascade</code>   | Filters arranged in series                    | Supported   | Target blocks depend on filter structures in the series          |
| <code>dfilt.parallel</code>  | Filters arranged in parallel                  | Supported   | Target blocks depend on filter structures in the parallel system |

For more information on each structure, use the syntax `help dfilt.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  |
|-----------------------|--|
| <code>addstage</code> | Adds a stage to a <code>cascade</code> or <code>parallel</code> object, where a stage is a separate, modular filter. See <code>dfilt.cascade</code> and <code>dfilt.parallel</code> .  |
| <code>block</code>    | <p><code>block(Hd)</code> creates a Simulink filter block of the <code>dfilt</code> object. The target filter block depends on the filter structure. You must have Simulink to use this method. Additionally, the DSP System Toolbox may be required depending on the filter structure. See “Structures” on page 1-263 for a mapping between the target blocks and filter structures.</p> <p>The <code>block</code> method can specify these properties/values:</p> <p>'<code>MapCoeffstoPorts</code>' indicates whether to map the filter coefficients to constant blocks connected to the generated block. Default value is '<code>off</code>'. Setting '<code>MapCoeffstoPorts</code>' to '<code>on</code>' turns on the mapping and enables the '<code>CoeffNames</code>' property, which defines the constant block parameter names. '<code>CoeffNames</code>' is a cell array of strings. Default values are { '<code>Num</code>' } for Direct form FIR filters, { '<code>K</code>' } for lattice filters, { '<code>Num</code>' , '<code>Den</code>' } for IIR filters, and { '<code>Num</code>' , '<code>Den</code>' , '<code>g</code>' } for biquad filters. Variables, defined by '<code>CoeffNames</code>', are created in the MATLAB workspace and have the same data type as the filter's '<code>Arithmetic</code>' property. Any existing variable with the same name is overwritten. Note that you can use either '<code>Link2Obj</code>' or '<code>MapCoeffstoPorts</code>', but not both simultaneously.</p> |



| Method    | Description   |
|-----------|---|
|           | <p>'InputProcessing' specifies sample-based, 'elementsaschannels', frame-based, 'columnsaschannels', processing, or 'inherited'. The default is frame-based processing. If you do not have the DSP System Toolbox software, explicitly set the 'InputProcessing' property to 'elementsaschannels' to avoid a runtime error. Setting 'InputProcessing' to 'inherited' targets the Digital Filter block regardless of structure.</p>  |
| cascade   | Returns the series combination of two <code>dfilt</code> objects. See <code>dfilt.cascade</code> .  |
| coeffs    | Returns the filter coefficients in a structure containing fields that use the same property names as those in the original <code>dfilt</code> .   |
| convert   | Converts a <code>dfilt</code> object from one filter structure to another filter structure.   |
| fcfwrite  | <p>Writes a filter coefficient ASCII file. The file can contain a single filter or a vector of objects. If the DSP System Toolbox product is installed, the file can contain multirate filters (<code>mfilt</code>) or adaptive filters (<code>adaptfilt</code>). Default filename is <code>untitled.fcf</code>.</p> <p><code>fcfwrite(Hd,filename)</code> writes to a disk file named <code>filename</code> in the current working directory. The <code>.fcf</code> extension is added automatically.</p> <p><code>fcfwrite(...,fmt)</code> writes the coefficients in the format <code>fmt</code>, where valid <code>fmt</code> strings are:</p> <ul style="list-style-type: none"> <li>'hex' for hexadecimal</li> <li>'dec' for decimal</li> <li>'bin' for binary representation.</li> </ul> |
| fftcoeffs | Returns the frequency-domain coefficients used when filtering with a <code>dfilt.fftfir</code> .  |

| Method                  | Description  |
|-------------------------|--|
| <code>filter</code>     | <p>Performs filtering using the <code>dfilt</code> object.</p> <p><code>y = filter(Hd,x)</code> filters <code>x</code> using the <code>Hd</code> filter and returns the filtered data in <code>y</code>. See “Using Filter States” on page 1-273 for information on using initial conditions. If <code>x</code> is a matrix, each column is filtered as an independent channel. If <code>x</code> is a multidimensional array, <code>filter</code> operates on the first nonsingleton dimension.</p> <p><code>y = filter(Hd,x,dim)</code> operates along the dimension <code>dim</code>. If <code>x</code> is a vector or matrix and <code>dim</code> is 1, every column of <code>x</code> is a channel. If <code>dim</code> is 2, every row is a channel.</p> |
| <code>firtype</code>    | Returns the type (1-4) of a linear phase FIR filter.   |
| <code>freqz</code>      | Plots the frequency response in <code>fvtool</code> . Note that unlike the <code>freqz</code> function, this <code>dfilt</code> <code>freqz</code> method has a default length of 8192.  |
| <code>grpdelay</code>   | Plots the group delay in <code>fvtool</code> .   |
| <code>impz</code>       | Plots the impulse response in <code>fvtool</code> .  |
| <code>impzlength</code> | Returns the length of the impulse response.  |
| <code>info</code>       | Displays brief <code>dfilt</code> 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 <code>info(Hd, 'long')</code> . The default display is <code>'short'</code> . For multistage filters ( <code>cascade</code> and <code>parallel</code> ), use <code>info(Hd.Stage(x))</code> , where <code>x</code> is the stage number, to see information about that stage.  |
| <code>isallpass</code>  | Returns a logical 1 (i.e., true) if the <code>dfilt</code> object in an allpass filter or a logical 0 (i.e., false) if it is not.  |
| <code>iscascade</code>  | Returns a logical 1 if the <code>dfilt</code> object is cascaded or a logical 0 if it is not.  |
| <code>isfir</code>      | Returns a logical 1 if the <code>dfilt</code> object has finite impulse response (FIR) or a logical 0 if it does not.  |
| <code>islinphase</code> | Returns a logical 1 if the <code>dfilt</code> object is linear phase or a logical 0 if it is not.  |

| Method                  | Description  |
|-------------------------|--|
| <code>ismaxphase</code> | Returns a logical 1 if the <code>dfilt</code> object is maximum-phase or a logical 0 if it is not.   |
| <code>isminphase</code> | Returns a logical 1 if the <code>dfilt</code> object is minimum-phase or a logical 0 if it is not.   |
| <code>isparallel</code> | Returns a logical 1 if the <code>dfilt</code> object has parallel stages or a logical 0 if it does not.  |
| <code>isreal</code>     | Returns a logical 1 if the <code>dfilt</code> object has real-valued coefficients or a logical 0 if it does not.   |
| <code>isscalar</code>   | Returns a logical 1 if the <code>dfilt</code> object is a scalar or a logical 0 if it is not scalar.   |
| <code>issos</code>      | Returns a logical 1 if the <code>dfilt</code> object has second-order sections or a logical 0 if it does not.  |
| <code>isstable</code>   | Returns a logical 1 if the <code>dfilt</code> object is stable or a logical 0 if it are not.   |
| <code>nsections</code>  | Returns the number of sections in a second-order sections filter. If a multistage filter contains stages with multiple sections, using <code>nsections</code> returns the total number of sections in all the stages (a stage with a single section returns 1).  |
| <code>nstages</code>    | Returns the number of stages of the filter, where a stage is a separate, modular filter.   |
| <code>nstates</code>    | Returns the number of states for an object.  |
| <code>order</code>      | Returns the filter order. If <code>Hd</code> is a single-stage filter, the order is given by the number of delays needed for a minimum realization of the filter. If <code>Hd</code> has multiple stages, the order is given by the number of delays needed for a minimum realization of the overall filter. |
| <code>parallel</code>   | Returns the parallel combination of two <code>dfilt</code> filters. See <code>dfilt.parallel</code> .  |
| <code>phasez</code>     | Plots the phase response in <code>fvtool</code> .  |

| Method     | Description  |
|------------|--|
| realizemdl | <p>(Available only with Simulink software.)</p> <p><code>realizemdl(Hd)</code> creates a Simulink model containing a subsystem block realization of your <code>dfilt</code>.</p> <p><code>realizemdl(Hd,p1,v1,p2,v2,...)</code> creates the block using the properties <code>p1</code>, <code>p2</code>,... and values <code>v1</code>, <code>v2</code>,... specified.</p> <p>The following properties are available:</p> <p>'<code>Blockname</code>' specifies the name of the block. The default value is '<code>Filter</code>'.</p> <p>'<code>Destination</code>' 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 '<code>current</code>', '<code>new</code>', or the name of an existing subsystem in your model. Default value is '<code>current</code>'.</p> <p>'<code>OverwriteBlock</code>' specifies whether to overwrite an existing block that was created by <code>realizemdl</code> or create a new block. Valid values are '<code>on</code>' and '<code>off</code>' and the default is '<code>off</code>'. Note that only blocks created by <code>realizemdl</code> are overwritten.</p> <p>The following properties optimize the block structure. Specifying '<code>on</code>' turns the optimization on and '<code>off</code>' creates the block without optimization. The default for each of the following is '<code>on</code>'.</p> <p>'<code>OptimizeZeros</code>' removes zero-gain blocks.</p> <p>'<code>OptimizeOnes</code>' replaces unity-gain blocks with a direct connection.</p> <p>'<code>OptimizeNegOnes</code>' replaces negative unity-gain blocks with a sign change at the nearest summation block.</p> |

| Method      | Description   |
|-------------|---|
|             | '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 <code>dfilt</code> . See <code>dfilt.cascade</code> and <code>dfilt.parallel</code> .  |
| setstage    | Overwrites a stage of a cascade or parallel <code>dfilt</code> . See <code>dfilt.cascade</code> and <code>dfilt.parallel</code> .   |
| sos         | <p>Converts the <code>dfilt</code> to a second-order sections <code>dfilt</code>. If <code>Hd</code> has a single section, the returned filter has the same class.</p> <p><code>sos(Hd, flag)</code> specifies the ordering of the second-order sections. If <code>flag='UP'</code>, the first row contains the poles closest to the origin, and the last row contains the poles closest to the unit circle. If <code>flag='down'</code>, the sections are ordered in the opposite direction. The zeros are always paired with the poles closest to them.</p> <p><code>sos(Hd, flag, scale)</code> specifies the scaling of the gain and the numerator coefficients of all second-order sections. <code>scale</code> can be <code>'none'</code>, <code>'inf'</code> (infinity-norm) or <code>'two'</code> (2-norm). Using infinity-norm scaling with <code>up</code> ordering minimizes the probability of overflow in the realization. Using 2-norm scaling with <code>down</code> ordering minimizes the peak roundoff noise.</p> |
| ss          | Converts the <code>dfilt</code> to state-space. To see the separate <code>A, B, C, D</code> matrices for the state-space model, use <code>[A, B, C, D]=ss(Hd)</code> .  |
| stepz       | <p>Plots the step response in <code>fvtool</code>.</p> <p><code>stepz(Hd, n)</code> computes the first <code>n</code> samples of the step response.</p> <p><code>stepz(Hd, n, Fs)</code> separates the time samples by <math>T = 1/Fs</math>, where <code>Fs</code> is assumed to be in Hz.</p>   |

| Method                 | Description   |
|------------------------|---|
| <code>sysobj</code>    | Converts the <code>dfilt</code> to a filter System object. See the reference page for a list of supported objects. To use this method, you must have DSP System Toolbox software installed. |
| <code>tf</code>        | Converts the <code>dfilt</code> to a transfer function.   |
| <code>zerophase</code> | Plots the zero-phase response in <code>fvtool</code> .  |
| <code>zpk</code>       | Converts the <code>dfilt</code> to zeros-pole-gain form.  |
| <code>zplane</code>    | Plots a pole-zero plot in <code>fvtool</code> .   |

For more information on each method, use the syntax `help dfilt/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)
```

## 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.

Alternatively, you can get or set a property value with `Object.property`:

```
b = [0.05 0.9 0.05];  
Hd = dfilt.dffir(b);  
% Lowpass direct-form I FIR filter  
Hd.arithmetic % get arithmetic property  
% returns double
```

```
Hd.arithmetic = 'single';  
% Set arithmetic property to single precision
```

## 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)
```

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')
```

or alternatively

```
Hd.numerator
```

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
```



## dfilt.cascade

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 (`Hd1`) to the end of an existing cascade (`Hd`), use

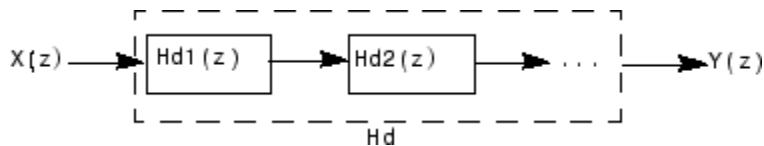
```
addstage(Hd,Hd1)
```

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
```

To view details of the first stage, use

```
info(Hcas.Stage(1))
```

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

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

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.

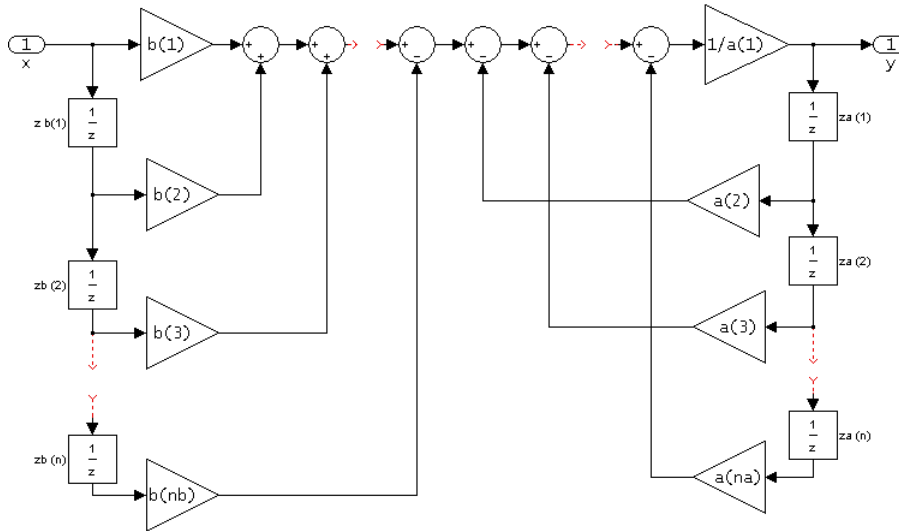
`Hd = dfilt.df1` returns a default, discrete-time, direct-form I filter, `Hd`, with `b=1` and `a=1`. This filter passes the input through to the output unchanged.

---

**Note** The leading coefficient of the denominator `a(1)` cannot be 0.

---

**df1**  
**(Direct-form I)**



**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 dfilter.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)
```

## See Also

[dfilt](#) | [dfilt.df1t](#) | [dfilt.df2](#) | [dfilt.df2t](#)

## **dfilt.df1sos**

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.

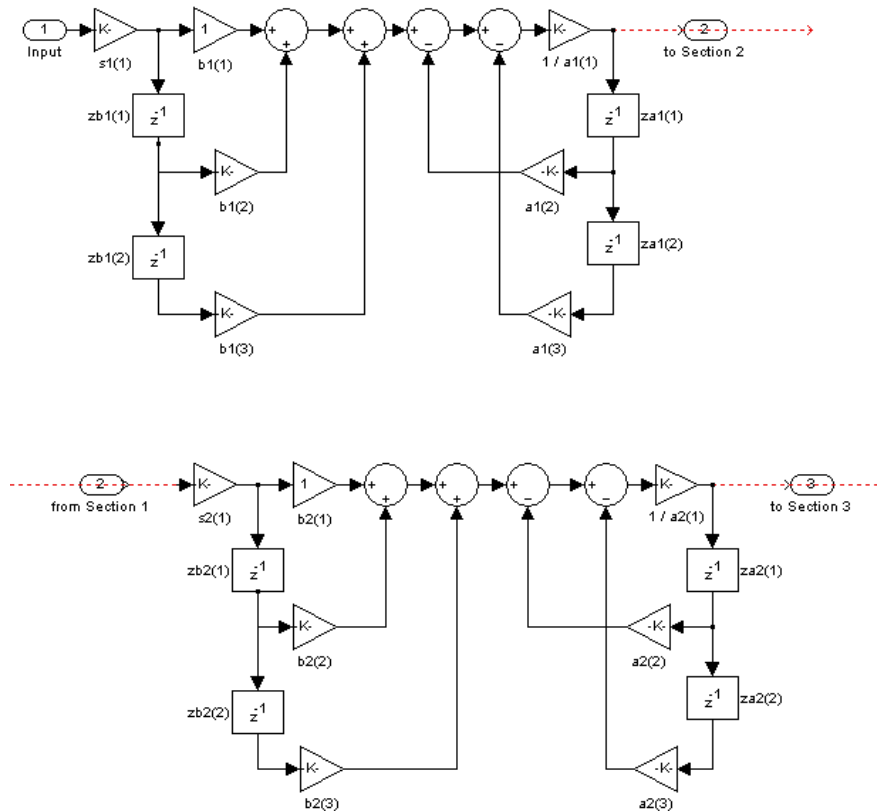
---

**Note** The leading coefficient of the denominator `a(1)` cannot be 0.

---



### df1sos (Direct-form I, second-order sections)



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)
```

## See Also

`dfilt` | `dfilt.df1tsos` | `dfilt.df2sos` | `dfilt.df2tsos`

# dfilt.df1t

Discrete-time, direct-form I transposed filter

## Syntax

```
Hd = dfilt.df1t(b,a)  
Hd = dfilt.df1t
```

## Description

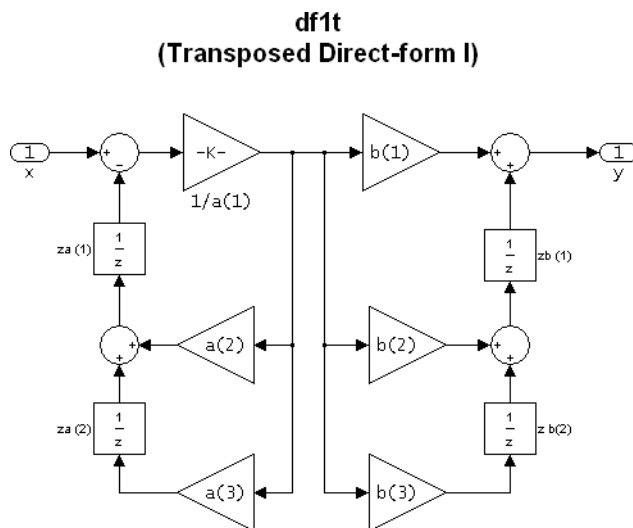
`Hd = dfilt.df1t(b,a)` returns a discrete-time, direct-form I transposed filter, `Hd`, with numerator coefficients `b` and denominator coefficients `a`. The filter states for this object are stored in a `filtstates` object.

`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 of states is:

$$\begin{pmatrix} zb(1) \\ zb(2) \\ \vdots \\ zb(M) \\ za(1) \\ za(2) \\ \vdots \\ za(N) \end{pmatrix}$$

Alternatively, you can access the states in the `filtstates` object:

```
b = [0.05 0.9 0.05];
Hd = dfilt.df1t(b,1);
Hd.States
% Returns
% Numerator: [2x1 double]
% Denominator: [0x1 double]
Hd.States.Numerator(1)=1; %Set zb(1) equal to 1.
```

## 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)
```

## See Also

`dfilt` | `dfilt.df1` | `dfilt.df2` | `dfilt.df2t`

## dfilt.df1tsos

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, transposed 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.

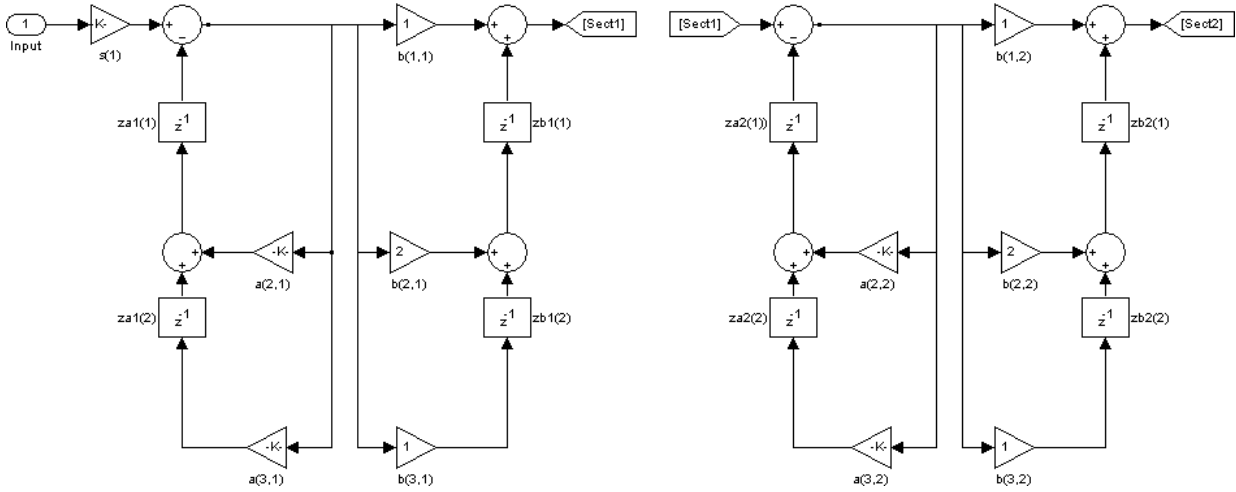
`Hd = dfilt.df1tsos` returns a default, discrete-time, second-order section, direct-form I, transposed filter, `Hd`. This filter passes the input through to the output unchanged.

---

**Note** The leading coefficient of the denominator `a(1)` cannot be 0.

---

**df1tsos**  
**(Transposed Direct-form I, second-order sections)**



To display the filter states, use this code to access the `filtstates` object.

```
Hs = Hd.states % Where Hd is the dfilter.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)
```

**See Also**

```
dfilt | dfilt.df1sos | dfilt.df2sos | dfilt.df2tsos
```

## dfilt.df2

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`.

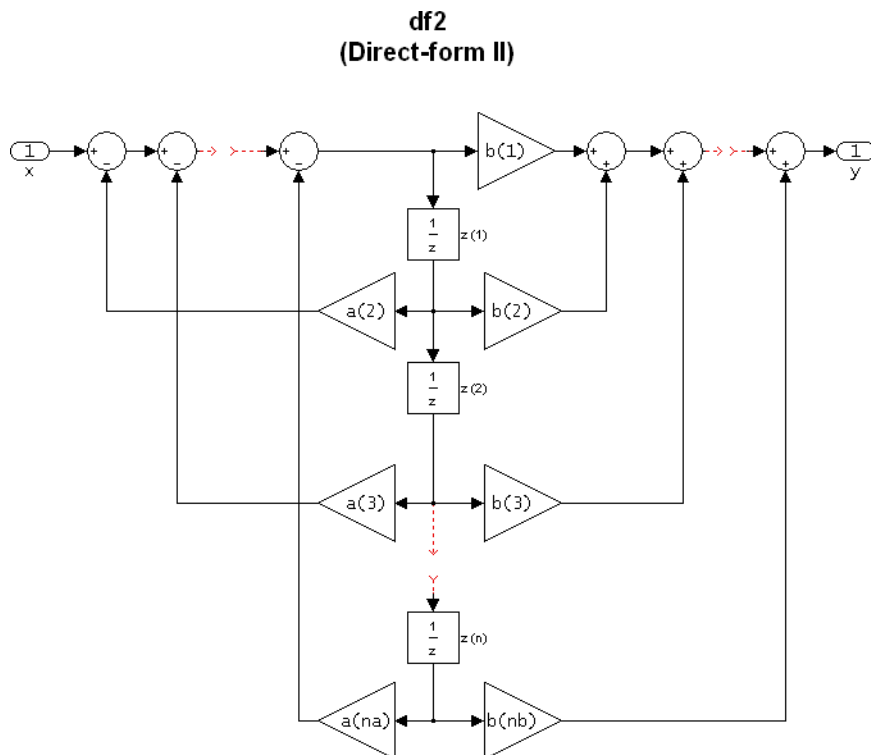
`Hd = dfilt.df2` returns a default, discrete-time, direct-form II filter, `Hd`, with `b=1` and `a=1`. This filter passes the input through to the output unchanged.

---

**Note** The leading coefficient of the denominator `a(1)` cannot be 0.

---





The resulting filter states column vector is

$$\begin{bmatrix} z(1) \\ z(2) \\ \vdots \\ z(n) \end{bmatrix}$$

## 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)
```

**See Also**

`dfilt` | `dfilt.df1` | `dfilt.df1t` | `dfilt.df2t`

## dfilt.df2sos

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.

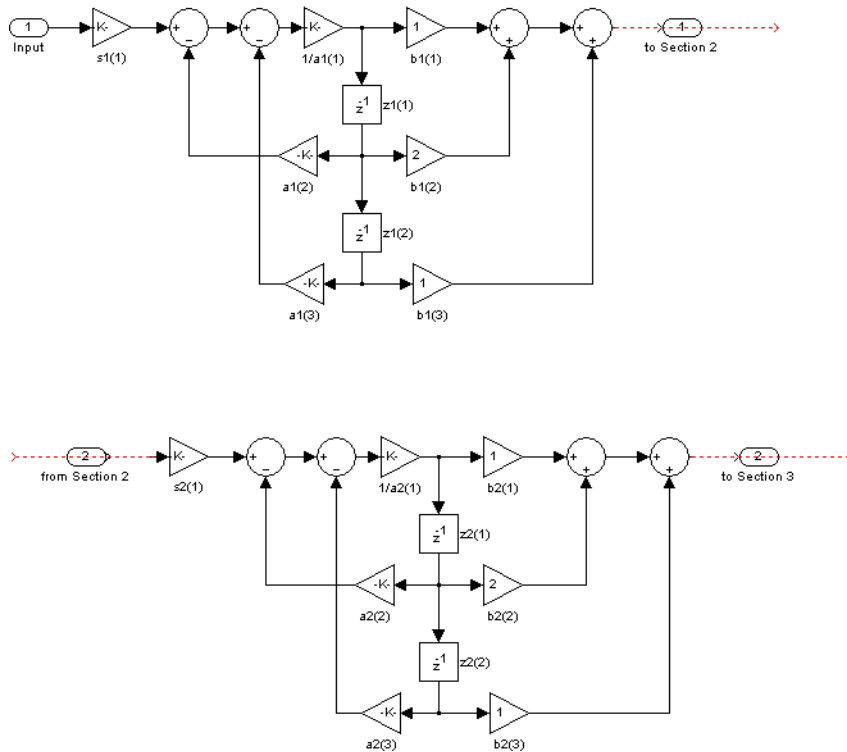
`Hd = dfilt.df2sos` returns a default, discrete-time, second-order section, direct-form II filter, `Hd`. This filter passes the input through to the output unchanged.

---

**Note** The leading coefficient of the denominator `a(1)` cannot be 0.

---

**df2sos**  
(Direct-form II, second-order sections)



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)
```

### **See Also**

dfilt | dfilt.df1sos | dfilt.df1tsos | dfilt.df2tsos

## dfilt.df2t

Discrete-time, direct-form II transposed filter

### Syntax

```
Hd = dfilt.df2t(b,a)  
Hd = dfilt.df2t
```

### Description

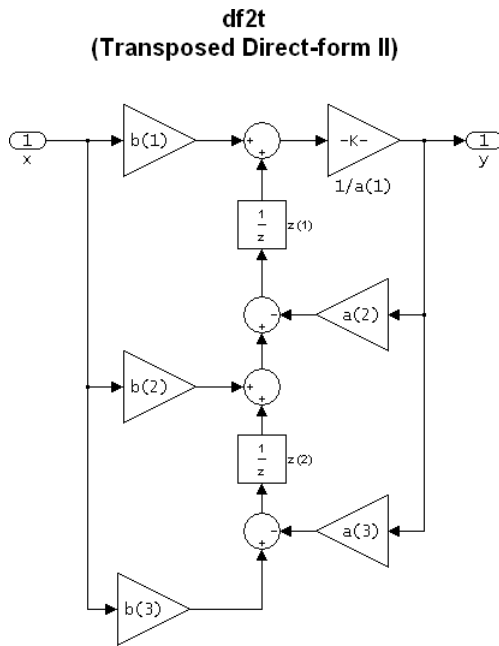
`Hd = dfilt.df2t(b,a)` returns a discrete-time, direct-form II transposed filter, `Hd`, with numerator coefficients `b` and denominator coefficients `a`.

`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 filter states of `dfilt.df2t` object can be extracted as a column vector with:

```
b = [1 2];
a = [1 -0.9];
Hd = dfilt.df2t(b,a);
FiltStates = double(Hd.States);
```

The resulting filter states column vector is

$$\begin{pmatrix} z(1) \\ z(2) \end{pmatrix}$$

## Examples

Create a direct-form II transposed discrete-time filter with coefficients from a 4–th order lowpass Butterworth design:

```
[b,a] = butter(4,.5);
```

```
Hd = dfilt.df2t(b,a);
```

**See Also**

`dfilt` | `dfilt.df1` | `dfilt.df1t` | `dfilt.df2`



## dfilt.df2tsos

Discrete-time, second-order section, direct-form II transposed filter

### Syntax

```
Hd = dfilt.df2tsos(s)
Hd = dfilt.df2tsos(b1,a1,b2,a2,...)
Hd = dfilt.df2tsos(...,g)
Hd = dfilt.df2tsos
```

### Description

`Hd = dfilt.df2tsos(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, transposed 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.

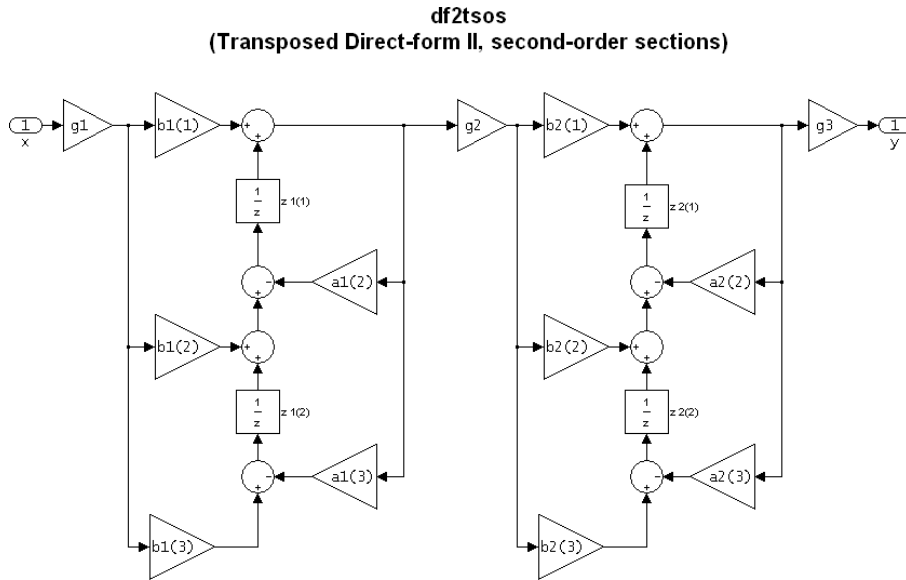
`Hd = dfilt.df2tsos(...,g)` includes a gain vector `g`. The elements of `g` are the gains for each section. The maximum length of `g` is the number of sections plus one. If `g` is not specified, all gains default to one.

`Hd = dfilt.df2tsos` 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

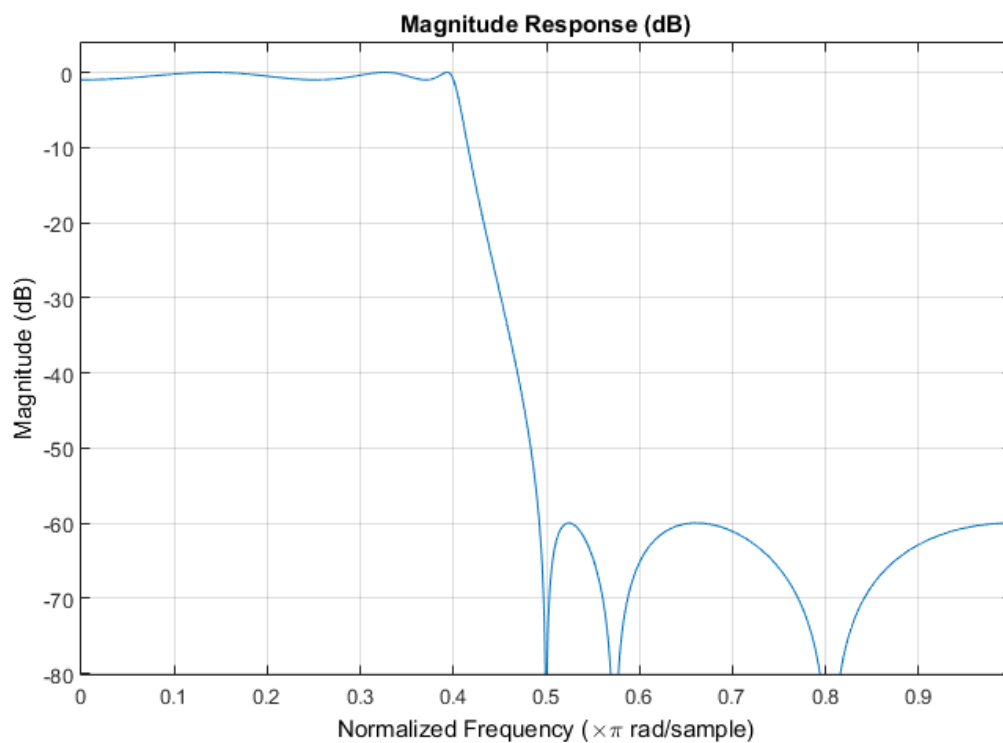
### Elliptic Filter as Second-Order Sections

Design a second-order sections, direct-form II, transposed discrete-time filter starting from a 6th-order lowpass elliptic filter. Specify a passband edge frequency of  $0.4\pi$  rad/sample, a passband ripple of 1 dB, and a stopband attenuation of 60 dB. Visualize the filter response.

```
[z,p,k] = ellip(6,1,60,0.4);    % Obtain filter coefficients
[s,g] = zp2sos(z,p,k);         % Convert to SOS

Hd = dfilt.df2tsos(s,g);

fvtool(Hd)
```



### See Also

dfilt | dfilt.df1sos | dfilt.df1tsos | dfilt.df2sos

## dfilt.dfasymfir

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`.

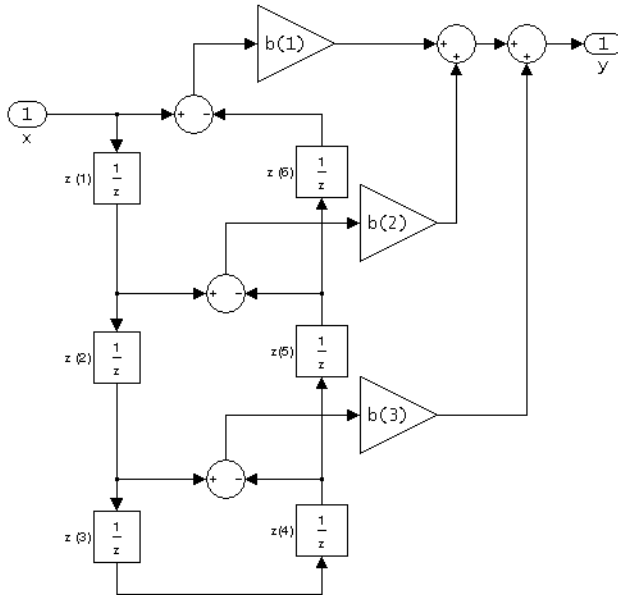
`Hd = dfilt.dfasymfir` returns a default, discrete-time, direct-form, antisymmetric FIR filter, `Hd`, with `b=1`. This filter passes the input through to the output unchanged.

---

**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  $b(6) = -b(1)$ .

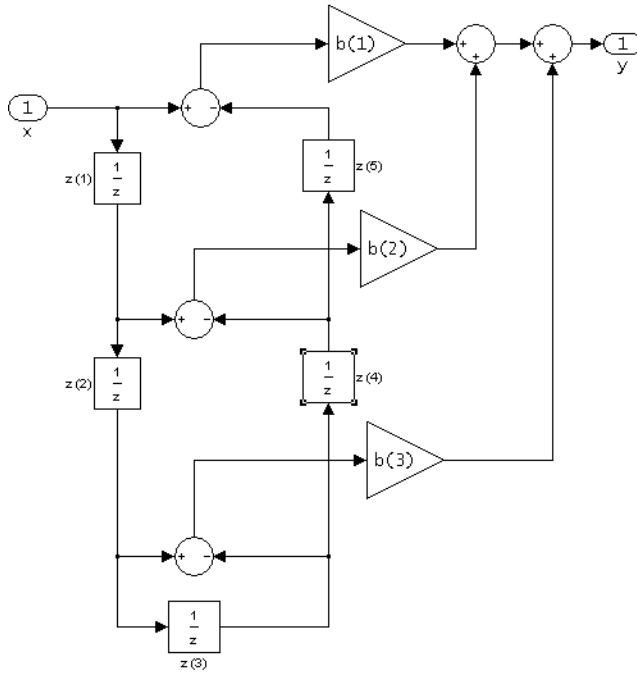
---

**dfasymfir**  
**(Antisymmetric FIR)**  
**Even order**  
**Odd number of coefficients, length(b) = 7**



Note that antisymmetry is defined as  
 $b(i) == -b(\text{end} - i + 1)$   
 so that the middle coefficient is zero for odd length  
 $b((\text{end}+1)/2) = 0$

**dfasymfir**  
**(Antisymmetric FIR)**  
**Even number of coefficients, length(b) = 6**



$$\mathbf{b}(i) == -\mathbf{b}(\text{end} - i + 1)$$

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 25<sup>th</sup> order highpass direct-form antisymmetric FIR filter structure for a `dfilt` object, `Hd`, with the following code:

```
Num_coeffs = firpm(25,[0 .4 .5 1],[0 0 1 1],'h');  
Hd = dfilt.dfasymfir(Num_coeffs);
```

### Even Order

Create a 44<sup>th</sup> order lowpass direct-form antisymmetric FIR differentiator filter structure for a `dfilt` object, `Hd`, with the following code:

```
Num_coeffs = firpm(44,[0 .3 .4 1],[0 .2 0 0],'differentiator');  
Hd = dfilt.dfasymfir(Num_coeffs);
```

### See Also

`dfilt` | `dfilt.dffir` | `dfilt.dffirt` | `dfilt.dfsymfir`

## dfilt.dffir

Discrete-time, direct-form, FIR filter

### Syntax

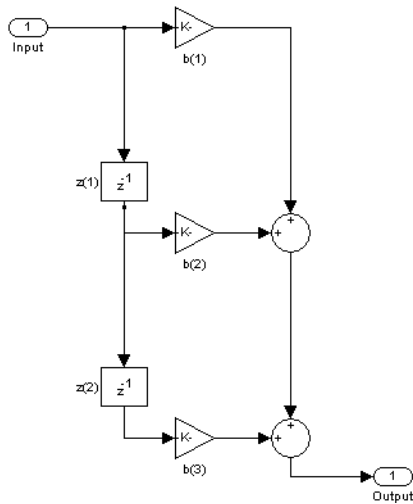
```
Hd = dfilt.dffir(b)
Hd = dfilt.dffir
```

### 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



$$\begin{pmatrix} z(1) \\ z(2) \end{pmatrix}$$

## Examples

Create a direct-form FIR discrete-time filter with coefficients from a 30<sup>th</sup> order lowpass equiripple design:

```
b = firpm(30,[0 .1 .2 .5]*2,[1 1 0 0]);  
Hd = dfilt.dffir(b)
```

## See Also

[dfilt](#) | [dfilt.dfasymfir](#) | [dfilt.dffirt](#) | [dfilt.dfsymfir](#)

## dfilt.dffirt

Discrete-time, direct-form FIR transposed filter

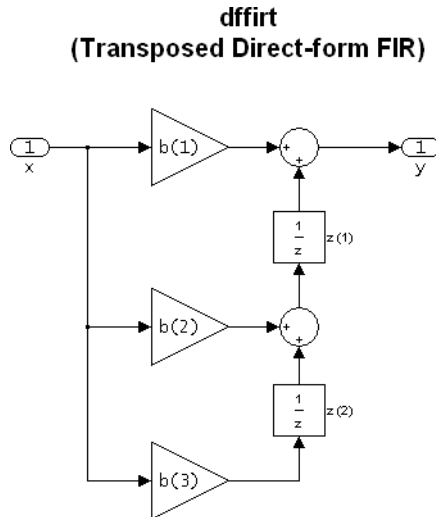
### Syntax

```
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.



The resulting filter states column vector is

$$\begin{pmatrix} z(1) \\ z(2) \end{pmatrix}$$

## Examples

Create a direct-form FIR transposed discrete-time filter with coefficients from a 30<sup>th</sup> order lowpass equiripple design:

```
b = firpm(30,[0 .1 .2 .5]*2,[1 1 0 0]);  
Hd = dfilt.dffirt(b)
```

## See Also

[dfilt](#) | [dfilt.dffir](#) | [dfilt.dfasymfir](#) | [dfilt.dfsymfir](#)

## dfilt.dfsymfir

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`.

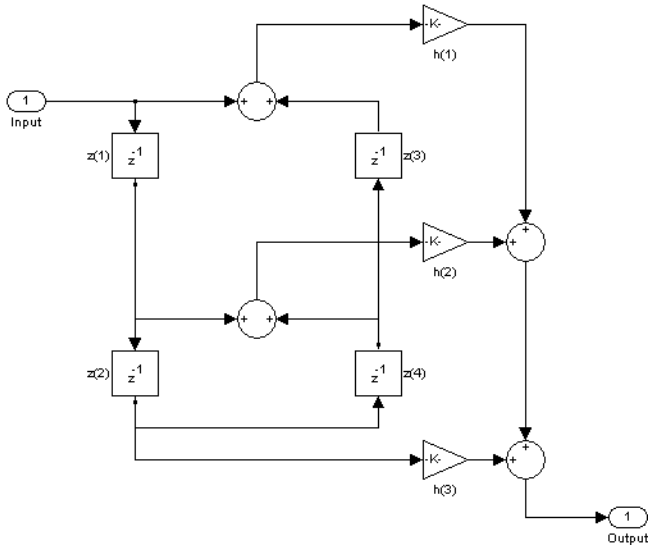
`Hd = dfilt.dfsymfir` returns a default, discrete-time, direct-form symmetric FIR filter, `Hd`, with `b=1`. This filter passes the input through to the output unchanged.

---

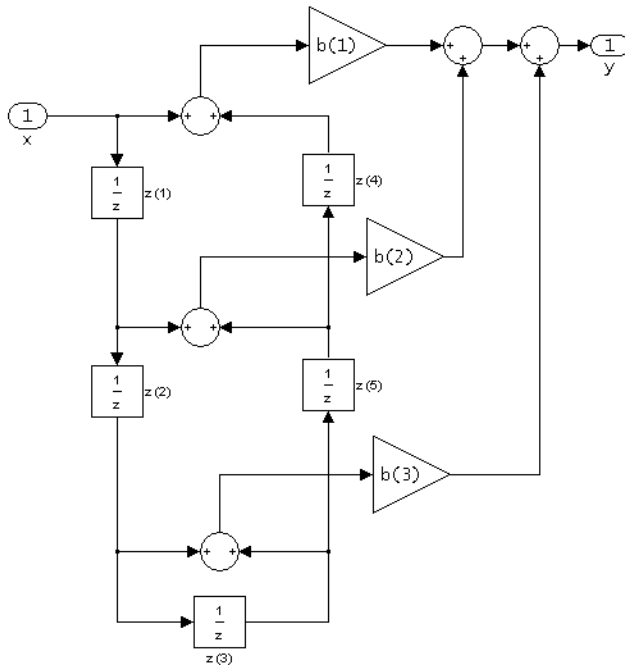
**Note** Only the first half of vector `b` is used because the second half is assumed to be symmetric. In the figure below for an odd number of coefficients,  $b(3) = 0$ ,  $b(4) = b(2)$  and  $b(5) = b(1)$ , and in the next figure for an even number of coefficients,  $b(4) = b(3)$ ,  $b(5) = b(2)$ , and  $b(6) = b(1)$ .

---

**dfsymfir**  
**(Symmetric FIR)**  
**Even order**  
**Odd number of coefficients, length(b) = 5**  
 **$b(i) == b(\text{end} - i + 1)$**



**dfsymfir**  
**(Symmetric FIR)**  
**Odd order**  
**Even number of coefficients, length(b) = 6**  
 **$b(i) == b(\text{end} - i + 1)$**



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)
```

### 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)
```

### See Also

`dfilt` | `dfilt.dfasymfir` | `dfilt.dffir` | `dfilt.dffirt`

## dfilt.fftfir

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.

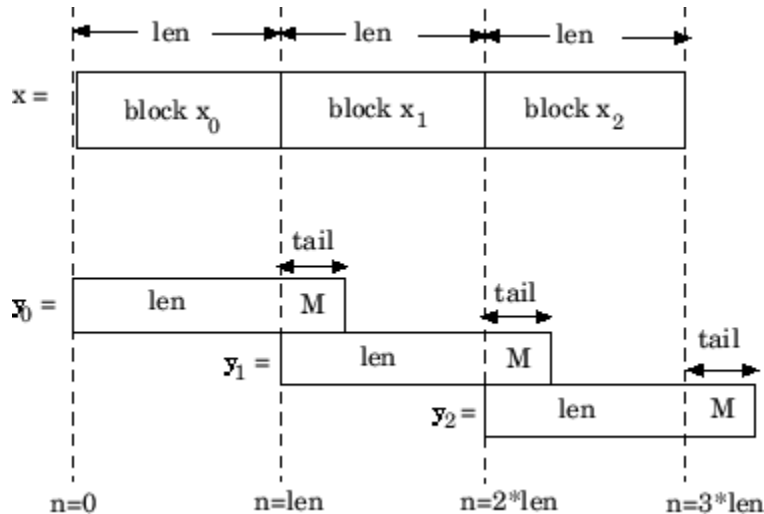
---

**Note** When you use a `dfilt.fftfir` object to filter data, the filter always operates on a segment of the signal equal in length to an integer multiple of the object's block length, `len`. If the input signal length is not equal to an integer multiple of the block length, the signal length is truncated to the nearest integer satisfying this requirement. If the `PersistentMemory` property is set to `true`, the next time you use the filter object the remaining signal samples are prepended to the subsequent input. 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.fffir` are the tails of the final convolution.

## Examples

Create an FFT FIR discrete-time filter with coefficients from a 30<sup>th</sup> order lowpass equiripple design:

```
b = firpm(30,[0 .1 .2 .5]*2,[1 1 0 0]);
Hd = dfilt.fffir(b)
```

To view the frequency domain coefficients used in the filtering, use the following command.

```
freq_coeffs = fftcoeffs(Hd);
```

## See Also

`dfilt` | `dfilt.dffir` | `dfilt.dfasymfir` | `dfilt.dffirt` | `dfilt.dfsymfir`

## dfilt.latticeallpass

Discrete-time, lattice allpass filter

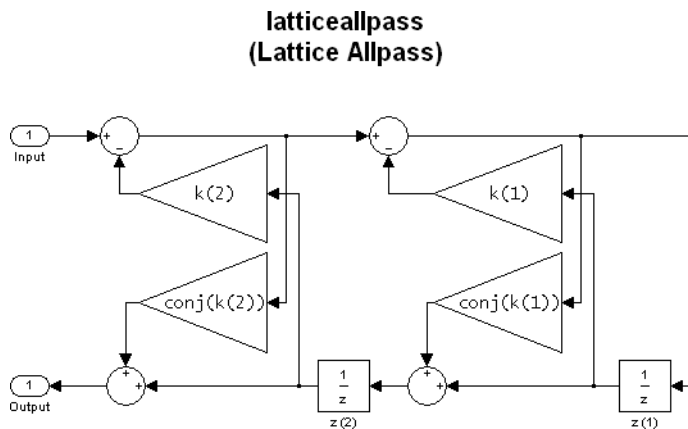
### Syntax

```
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 `Hd.States` is

$$\begin{pmatrix} z(1) \\ z(2) \end{pmatrix}$$

## 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)
```

## See Also

`dfilt` | `dfilt.latticear` | `dfilt.latticearma` | `dfilt.latticemamax` |  
`dfilt.latticemamin`

## dfilt.latticear

Discrete-time, lattice, autoregressive filter

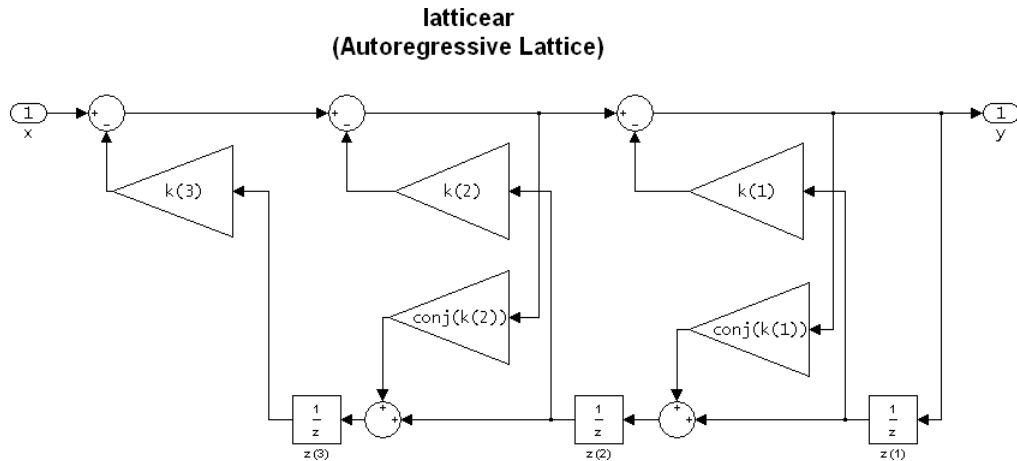
### Syntax

```
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)
```

## See Also

`dfilt` | `dfilt.latticeallpass` | `dfilt.latticearma` | `dfilt.latticemamax` |  
`dfilt.latticemamin`

## dfilt.latticearma

Discrete-time, lattice, autoregressive, moving-average filter

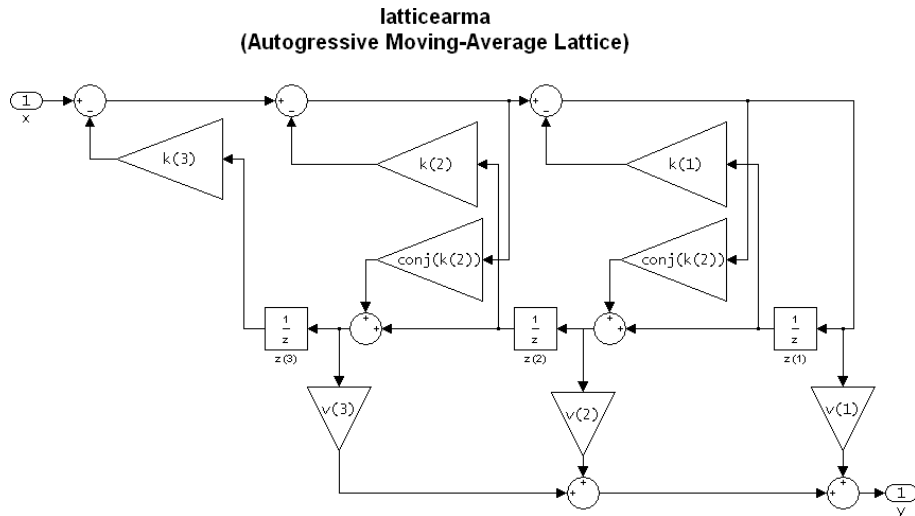
### Syntax

```
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)
```

## See Also

`dfilt` | `dfilt.latticeallpass` | `dfilt.latticear` | `dfilt.latticemamax` |  
`dfilt.latticemamin`

## **dfilt.latticemamax**

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`.

---

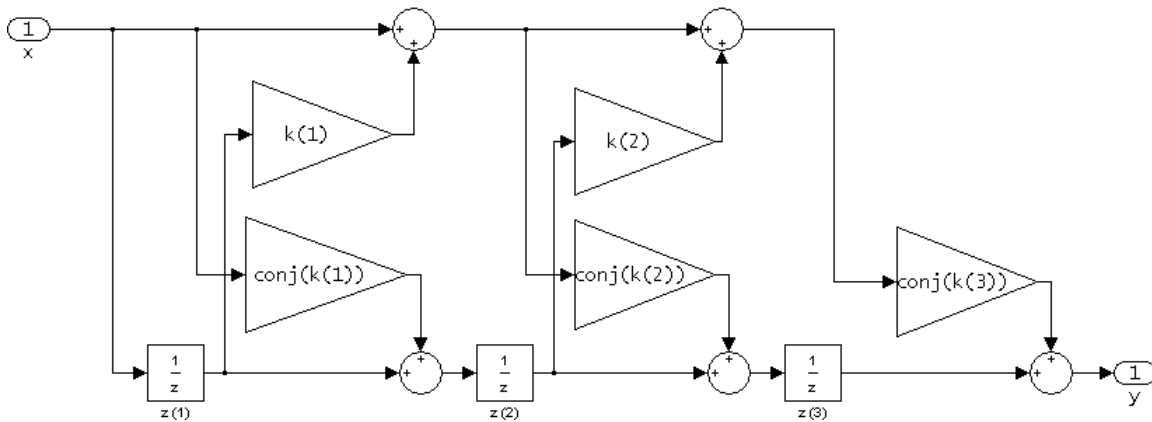
**Note** If the `k` coefficients define a maximum phase filter, the resulting filter in this structure is maximum phase. If your coefficients do not define a maximum phase filter, placing them in this structure does not produce a maximum phase filter.

---

`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)
```

## See Also

`dfilt` | `dfilt.latticeallpass` | `dfilt.latticear` | `dfilt.latticearma` | `dfilt.latticemamin`

## **dfilt.latticemamin**

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`.

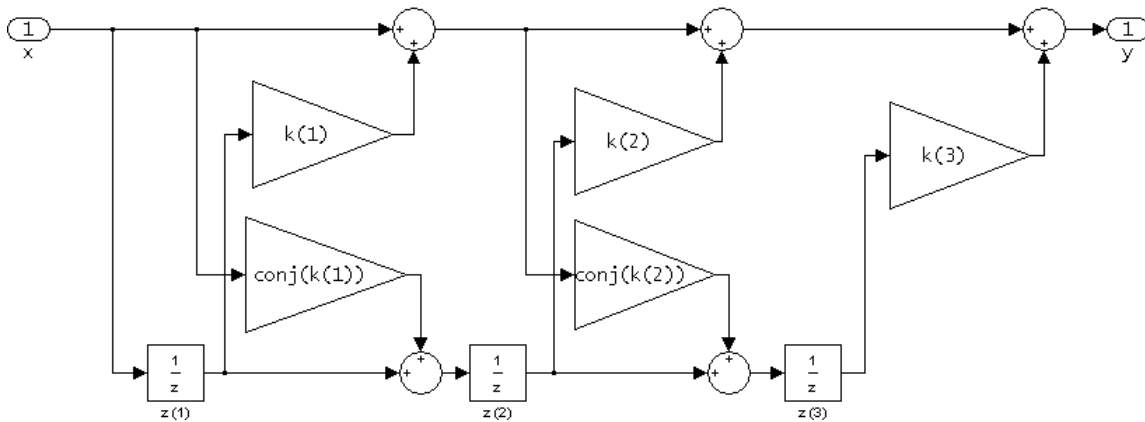
---

**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, minimum phase, filter structure for a `dfilt` object, `Hd`, using the following lattice coefficients.

```
k = [.66 .7 .44];
Hd = dfilt.latticemamin(k)
```

## See Also

`dfilt` | `dfilt.latticeallpass` | `dfilt.latticear` | `dfilt.latticearma` | `dfilt.latticemamax`

## dfilt.parallel

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

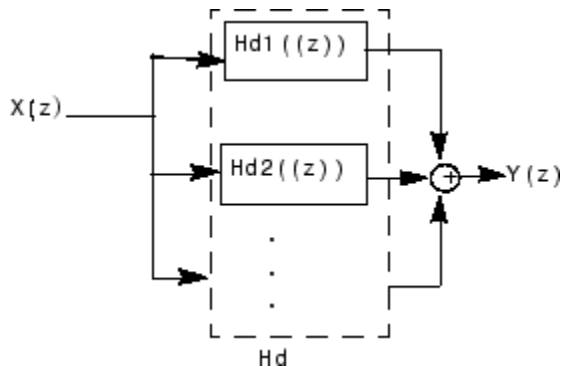
```
Hd.stage(1).states
```

To append a filter (`Hd1`) onto an existing parallel filter (`Hd`), use

```
addstage(Hd,Hd1)
```

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

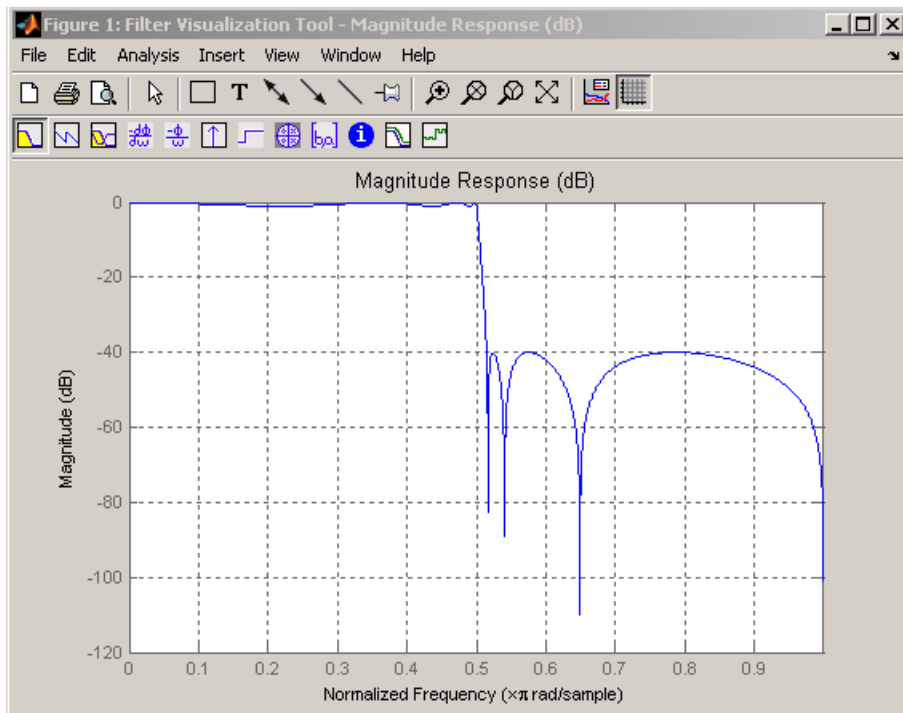
```
info(Hcas.Stage(1))
```

and

```
info(Hcas.Stage(2))
```

To view this filter, use

```
fvtool(Hcas)
```



**See Also**

`dfilt` | `dfilt.cascade`

# dfilt.scalar

Discrete-time, scalar filter

## Syntax

```
Hd = dfilt.scalar(g)  
Hd = dfilt.scalar
```

## Description

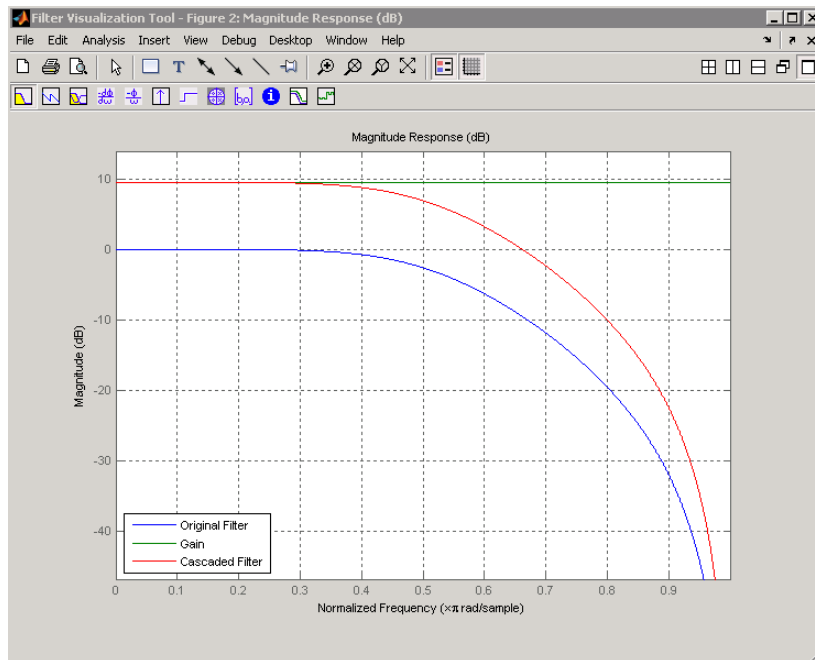
`Hd = dfilt.scalar(g)` returns a discrete-time, scalar filter, `Hd`, with gain `g`, where `g` is a scalar.

`Hd = dfilt.scalar` returns a default, discrete-time scalar gain filter, `Hd`, with gain 1.

## Examples

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 = cascade(Hd_gain,Hd_filt);  
hfvtool(Hd_filt,Hd_gain,Hd_cascade);  
legend(hfvtool,'Original Filter','Gain','Cascaded Filter',...  
'location','southwest');
```



To view the stages of the cascaded filter, use

```
Hd.stage(1)
```

and

```
Hd.stage(2)
```

## See Also

```
dfilt | dfilt.cascade
```



## dfilt.statespace

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`.

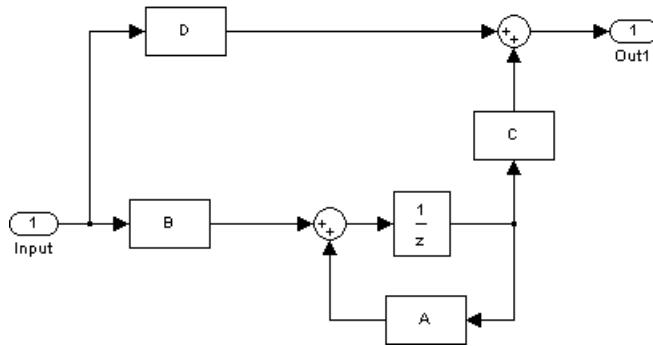
`A`, `B`, `C`, and `D` are from the matrix or state-space form of a filter's difference equations

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

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-output 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.

## Statespace



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)
```

## See Also

dfilt

# dftmtx

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,  $x$ , computes the discrete Fourier transform of  $x$ . In other words,  $y = A*x$  is the same as  $y = \text{fft}(x)$ .

The inverse discrete Fourier transform matrix is

```
Ai = conj(dftmtx(n))/n
```

## Examples

### The FFT and the DFT Matrix

In practice, it is more efficient to compute the discrete Fourier transform with the FFT than with the DFT matrix. The FFT also uses less memory. The two procedures give the same result.

```
x = 1:256;
```

```
y1 = fft(x);
```

```
n = length(x);  
y2 = x*dftmtx(n);
```

```
norm(y1-y2)
```

```
ans =
```

```
7.7528e-12
```

## More About

### Algorithms

dftmtx takes the FFT of the identity matrix to generate the transform matrix.

### See Also

convmtx | fft

# digitalFilter class

Digital filter

## Description

- Use `designfilt` in the form `d = designfilt(resp,Name,Value)` to design a digital filter, `d`, with response type `resp`. Specify the filter further using a set of `Name,Value` pairs.
- Use `designfilt` in the form `designfilt(d)` to edit an existing filter, `d`.
- Use `filter` in the form `dataOut = filter(d,dataIn)` to filter a signal with a `digitalFilter`, `d`. The input can be a double- or single-precision vector. It can also be a matrix with as many columns as there are input channels.
- Use `fvtool` to visualize a `digitalFilter`, `d`.
- The following functions take `digitalFilter` objects as input.

## Filtering and Analysis Functions

### Filtering

| Function              | Description  |
|-----------------------|--|
| <code>fftfilt</code>  | Filters a signal with a <code>digitalFilter</code> using an FFT-based overlap-add method |
| <code>filter</code>   | Filters a signal using a <code>digitalFilter</code>                                      |
| <code>filtfilt</code> | Performs zero-phase filtering of a signal with a <code>digitalFilter</code>              |

### Filter Analysis

| Function                | Description   |
|-------------------------|---|
| <code>double</code>     | Casts the coefficients of a <code>digitalFilter</code> to double precision      |
| <code>filt2block</code> | Generates a Simulink filter block corresponding to a <code>digitalFilter</code> |
| <code>filtord</code>    | Returns the filter order of a <code>digitalFilter</code>                        |

| <b>Function</b>         | <b>Description</b>   |
|-------------------------|--|
| <code>firtype</code>    | Returns the type (1, 2, 3, or 4) of an FIR <code>digitalFilter</code>  |
| <code>freqz</code>      | Returns or plots the frequency response of a <code>digitalFilter</code>  |
| <code>fvtool</code>     | Opens the Filter Visualization Tool and displays the magnitude response of a <code>digitalFilter</code>                                      |
| <code>grpdelay</code>   | Returns or plots the group delay response of a <code>digitalFilter</code>  |
| <code>impz</code>       | Returns or plots the impulse response of a <code>digitalFilter</code>  |
| <code>impzlength</code> | Returns the length of the impulse response of a <code>digitalFilter</code> , whether actual (for FIR filters) or effective (for IIR filters) |
| <code>info</code>       | Returns a string matrix with information about a <code>digitalFilter</code>  |
| <code>isallpass</code>  | Returns <code>true</code> if a <code>digitalFilter</code> is allpass   |
| <code>isdouble</code>   | Returns <code>true</code> if the coefficients of a <code>digitalFilter</code> are double precision   |
| <code>isfir</code>      | Returns <code>true</code> if a <code>digitalFilter</code> has a finite impulse response  |
| <code>islinphase</code> | Returns <code>true</code> if a <code>digitalFilter</code> has linear phase   |
| <code>ismaxphase</code> | Returns <code>true</code> if a <code>digitalFilter</code> is maximum phase   |
| <code>isminphase</code> | Returns <code>true</code> if a <code>digitalFilter</code> is minimum phase   |
| <code>issingle</code>   | Returns <code>true</code> if the coefficients of a <code>digitalFilter</code> are single precision   |
| <code>isstable</code>   | Returns <code>true</code> if a <code>digitalFilter</code> is stable  |
| <code>phasedelay</code> | Returns or plots the phase delay response of a <code>digitalFilter</code>  |
| <code>phasez</code>     | Returns or plots the (unwrapped) phase response of a <code>digitalFilter</code>  |
| <code>single</code>     | Casts the coefficients of a <code>digitalFilter</code> to single precision   |
| <code>ss</code>         | Returns the state-space representation of a <code>digitalFilter</code>   |
| <code>stepz</code>      | Returns or plots the step response of a <code>digitalFilter</code>   |

| Function               | Description   |
|------------------------|---|
| <code>tf</code>        | Returns the transfer function representation of a <code>digitalFilter</code>                      |
| <code>zerophase</code> | Returns or plots the zero-phase response of a <code>digitalFilter</code>                          |
| <code>zpk</code>       | Returns the zero-pole-gain representation of a <code>digitalFilter</code>                         |
| <code>zplane</code>    | Displays the poles and zeros of the transfer function represented by a <code>digitalFilter</code> |

## See Also

`designfilt` | `double` | `fftfilt` | `filt2block` | `filter` | `filtfilt` | `filtord` | `firtype` | `freqz` | `fvtool` | `grpdelay` | `impz` | `impzlength` | `info` | `isallpass` | `isdouble` | `isfir` | `islinphase` | `ismaxphase` | `isminphase` | `issingle` | `isstable` | `phasedelay` | `phasez` | `single` | `ss` | `stepz` | `tf` | `zerophase` | `zpk` | `zplane`

# digitrevorder

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                 |



| Linear Index | Base-4 Digits | Digit-Reversed | Digit-Reversed Index | Base-2 Bits | Base-2 Reversed (bitrevorder) | Bit-Reversed Index |
|--------------|---------------|----------------|----------------------|-------------|-------------------------------|--------------------|
| 4            | 10            | 01             | 1                    | 0100        | 0010                          | 2                  |
| 5            | 11            | 11             | 5                    | 0101        | 1010                          | 10                 |
| 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

### Base-3 Digit-Reversed Order

Obtain the digit-reversed, radix base-3 ordered output of a vector containing 9 values. Obtain the same result by converting to base 3 and reversing the digits.

```
x = (0:8)';
y = digitrevorder(x,3);

c1 = dec2base(x,3);
c2 = fliplr(c1);
c3 = base2dec(c2,3);

T = table(x,y,c1,c2,c3)
```

T =

| x | y | c1 | c2 | c3 |
|---|---|----|----|----|
| — | — | —  | —  | —  |
| 0 | 0 | 00 | 00 | 0  |
| 1 | 3 | 01 | 10 | 3  |
| 2 | 6 | 02 | 20 | 6  |
| 3 | 1 | 10 | 01 | 1  |
| 4 | 4 | 11 | 11 | 4  |
| 5 | 7 | 12 | 21 | 7  |
| 6 | 2 | 20 | 02 | 2  |
| 7 | 5 | 21 | 12 | 5  |
| 8 | 8 | 22 | 22 | 8  |

## See Also

`bitrevorder` | `fft` | `ifft`

# diric

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

$$D(x) = \begin{cases} \frac{\sin(Nx/2)}{N \sin(x/2)} & x \neq 2\pi k, \quad k = 0, \pm 1, \pm 2, \pm 3, \dots \\ (-1)^{k(N-1)} & x = 2\pi k, \quad k = 0, \pm 1, \pm 2, \pm 3, \dots \end{cases}$$

for any nonzero integer `n`. This function has period  $2\pi$  for `n` odd and period  $4\pi$  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`

## double

Cast coefficients of digital filter to double precision

### Syntax

```
f2 = double(f1)
```

### Description

`f2 = double(f1)` casts coefficients in a digital filter, `f1`, to double precision and returns a new digital filter, `f2`, that contains these coefficients.

### Examples

#### Lowpass FIR Filter in Single and Double Precision

Use `designfilt` to design a 5th-order FIR lowpass filter. Specify a normalized passband frequency of  $0.2\pi$  rad/sample and a normalized stopband frequency of  $0.55\pi$  rad/sample.

Cast the filter to single precision and cast it back to double precision. Display the first coefficient of each filter.

```
format long
d = designfilt('lowpassfir','FilterOrder',5, ...
              'PassbandFrequency',0.2,'StopbandFrequency', 0.55);
e = single(d);
f = double(e);

coed = d.Coefficients(1)
coee = e.Coefficients(1)
coef = f.Coefficients(1)
```

```
coed =
    0.003947882145754
```

```
coee =  
    0.0039479  
  
coef =  
    0.003947881981730
```

Use `double` to analyze, in double precision, the effects of single-precision quantization of filter coefficients.

## Input Arguments

### **f1** — Single-precision digital filter

`digitalFilter` object

Single-precision digital filter, specified as a `digitalFilter` object. Use `designfilt` to generate a digital filter based on frequency-response specifications and `single` to cast it to single precision.

Example: `f1=`

```
single(designfilt('lowpassfir','FilterOrder',3,'HalfPowerFrequency',0.5))  
specifies a third-order Butterworth filter with normalized 3-dB frequency  $0.5\pi$  rad/sample cast in single precision.
```

## Output Arguments

### **f2** — Double-precision digital filter

`digitalFilter` object

Double-precision digital filter, returned as a `digitalFilter` object.

## See Also

`designfilt` | `digitalFilter` | `isdouble` | `issingle` | `single`

## downsample

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 Sampling Rates

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)
```

y =

1      2      3  
10     11     12

## See Also

[decimate](#) | [interp](#) | [interp1](#) | [resample](#) | [spline](#) | [upfirdn](#) | [upsample](#)

## dpss

Discrete prolate spheroidal (Slepian) sequences

### Syntax

```
dps_seq = dpss(seq_length,time_halfbandwidth)
[dps_seq,lambda] = dpss(seq_length,time_halfbandwidth)
[...] = dpss(seq_length,time_halfbandwidth,num_seq)
[...] = dpss(seq_length,time_halfbandwidth,'interp_method')
[...] = dpss(...,Ni)
[...] = dpss(...,'trace')
```

### Description

`dps_seq = dpss(seq_length,time_halfbandwidth)` returns the first  $\text{round}(2*\text{time\_halfbandwidth})$  discrete prolate spheroidal (DPSS), or Slepian sequences of length `seq_length`. `dps_seq` is a matrix with `seq_length` rows and  $\text{round}(2*\text{time\_halfbandwidth})$  columns. `time_halfbandwidth` must be strictly less than `seq_length/2`.

`[dps_seq,lambda] = dpss(seq_length,time_halfbandwidth)` returns the frequency-domain energy concentration ratios of the column vectors in `dps_seq`. The ratios represent the amount of energy in the passband  $[-W,W]$  to the total energy from  $[-F_s/2,F_s/2]$ , where  $F_s$  is the sampling frequency. `lambda` is a column vector equal in length to the number of Slepian sequences.

`[...] = dpss(seq_length,time_halfbandwidth,num_seq)` returns the first `num_seq` Slepian sequences with time half bandwidth product `time_halfbandwidth` ordered by their energy concentration ratios. If `num_seq` is a two-element vector, the returned Slepian sequences range from `num_seq(1)` to `num_seq(2)`.

`[...] = dpss(seq_length,time_halfbandwidth,'interp_method')` uses interpolation to compute the DPSSs from a user-created database of DPSSs. Create the database of DPSSs with `dpsssave` and ensure that the resulting file, `dpss.mat`, is in the MATLAB search path. Valid options for `'interp_method'` are `'spline'` and



'linear'. The interpolation method uses the Slepian sequences in the database with time half bandwidth product `time_halfbandwidth` and length closest to `seq_length`.

[...] = `dpss(...,Ni)` interpolates from DPSSs of length `Ni` in the database `dpss.mat`.

[...] = `dpss(...,'trace')` prints the method used to compute the DPSSs in the command window. Possible methods include: direct, spline interpolation, and linear interpolation.

## Examples

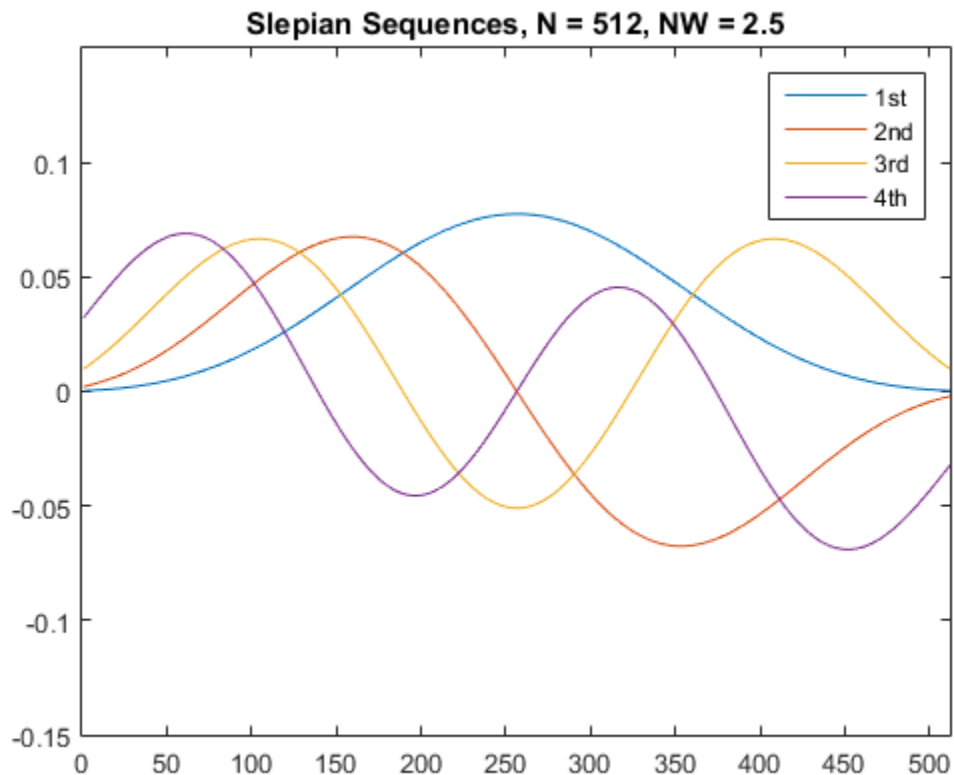
### Generate a Set of Slepian Sequences

Construct the first four discrete prolate spheroidal sequences of length 512. Specify a time half bandwidth product of 2.5. Plot the sequences and find the concentration ratios.

```
seq_length = 512;
time_halfbandwidth = 2.5;
num_seq = 2*(2.5)-1;
[dps_seq,lambda] = dpss(seq_length,time_halfbandwidth,num_seq);
```

```
plot(dps_seq)
title('Slepian Sequences, N = 512, NW = 2.5')
axis([0 512 -0.15 0.15])
legend('1st','2nd','3rd','4th')
concentration_ratios = lambda'
```

```
concentration_ratios =
    1.0000    0.9998    0.9962    0.9521
```



## More About

### Discrete Prolate Spheroidal Sequences

The discrete prolate spheroidal or Slepian sequences derive from the following time-frequency concentration problem. For all finite-energy sequences  $x[n]$  index limited to some set  $[N_1, N_1 + N_2]$ , which sequence maximizes the following ratio:

$$\lambda = \frac{\int_{-W}^W |X(f)|^2 df}{\int_{-F_s/2}^{F_s/2} |X(f)|^2 df}$$

where  $F_s$  is the sampling frequency and  $|W| < F_s / 2$ . Accordingly, this ratio determines which index-limited sequence has the largest proportion of its energy in the band  $[-W, W]$ . For index-limited sequences, the ratio must satisfy the inequality  $0 < \lambda < 1$ . The sequence maximizing the ratio is the first discrete prolate spheroidal or Slepian sequence. The second Slepian sequence maximizes the ratio and is orthogonal to the first Slepian sequence. The third Slepian sequence maximizes the ratio of integrals and is orthogonal to both the first and second Slepian sequences. Continuing in this way, the Slepian sequences form an orthogonal set of bandlimited sequences.

### Time Half Bandwidth Product

The time half bandwidth product is  $NW$  where  $N$  is the length of the sequence and  $[-W, W]$  is the effective bandwidth of the sequence. In constructing Slepian sequences, you choose the desired sequence length and bandwidth  $2W$ . Both the sequence length and bandwidth affect how many Slepian sequences have concentration ratios near one. As a rule, there are  $2NW - 1$  Slepian sequences with energy concentration ratios approximately equal to one. Beyond  $2NW - 1$  Slepian sequences, the concentration ratios begin to approach zero. Common choices for the time half bandwidth product are: 2.5, 3, 3.5, and 4.

You can specify the bandwidth of the Slepian sequences in Hz by defining the time half bandwidth product as  $NWF_s$ , where  $F_s$  is the sampling frequency.

- “Nonparametric Methods”

## References

Percival, D. B., and A. T. Walden. *Spectral Analysis for Physical Applications*. Cambridge, UK: Cambridge University Press, 1993.

### See Also

dpsscLEAR | dpssload | dpsssave | pmtm

## **dpssc**clear

Remove discrete prolate spheroidal sequences from database

### **Syntax**

```
dpssc
```

clear(*n*,*nw*)

### **Description**

`dpssc`clear(*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`

# dpssdir

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`. Create the DPSS MAT-file database with `dpsssave`.

`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`.

## See Also

`dpss` | `dpssc` | `dpssclear` | `dpssload` | `dpsssave`

## **dpssload**

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`. Create the `dpss.mat` file using `dpssave`.

### **See Also**

`dpss` | `dpssc` | `dpssclear` | `dpssdir` | `dpsssave`

# dpsssave

Discrete prolate spheroidal or Slepian sequence database

## Syntax

```
dpsssave(time_halfbandwidth,dps_seq,lambda)
status = dpsssave(time_halfbandwidth,dps_seq,lambda)
```

## Description

`dpsssave(time_halfbandwidth,dps_seq,lambda)` creates a database of discrete prolate spheroidal (DPSS) or Slepian sequences and saves the results in `dpss.mat`. The time half bandwidth product `time_halfbandwidth` is a real-valued scalar determining the frequency concentration of the Slepian sequences in `dps_seq`. `dps_seq` is a  $N \times K$  matrix of Slepian sequences where  $N$  is the length of the sequences. `lambda` is a  $1 \times K$  vector containing the frequency concentration ratios of the Slepian sequences in `dps_seq`.

If the database `dpss.mat` exists, subsequent calls to `dpsssave` append the Slepian sequences to the existing file. If the sequences are already in the existing file, `dpsssave` overwrites the old values and issues a warning.

`status = dpsssave(time_halfbandwidth,dps_seq,lambda)` returns a 0 if the database operation was successful or a 1 if unsuccessful.

## Examples

### Create a Database of Slepian Sequences

Construct the first four discrete prolate spheroidal sequences of length 512. Specify a time half bandwidth product of 2.5. Use them to create a database of Slepian sequences, `dpss.mat`, in the current working directory. The output variable, `status`, is 0 if there is success.

```
seq_length = 512;
```

```
time_halfbandwidth = 2.5;
num_seq = 4;
[dps_seq,lambda] = dps(seq_length,time_halfbandwidth);
status = dpssave(time_halfbandwidth,dps_seq,lambda)

status =

    0
```

## More About

### Discrete Prolate Spheroidal Sequences

The discrete prolate spheroidal or Slepian sequences derive from the following time-frequency concentration problem. For all finite-energy sequences  $x[n]$  index limited to some set  $[N_1, N_1 + N_2]$ , which sequence maximizes the following ratio:

$$\lambda = \frac{\int_{-W}^W |X(f)|^2 df}{\int_{-Fs/2}^{Fs/2} |X(f)|^2 df}$$

where  $F_s$  is the sampling frequency and  $|W| < F_s / 2$ . Accordingly, this ratio determines which index-limited sequence has the largest proportion of its energy in the band  $[-W, W]$ . For index-limited sequences, the ratio must satisfy the inequality  $0 < \lambda < 1$ . The sequence maximizing the ratio is the first discrete prolate spheroidal or Slepian sequence. The second Slepian sequence maximizes the ratio and is orthogonal to the first Slepian sequence. The third Slepian sequence maximizes the ratio of integrals and is orthogonal to both the first and second Slepian sequences. Continuing in this way, the Slepian sequences form an orthogonal set of bandlimited sequences.

### Time Half Bandwidth Product

The time half bandwidth product is  $NW$  where  $N$  is the length of the sequence and  $[-W, W]$  is the effective bandwidth of the sequence. In constructing Slepian sequences,



you choose the desired sequence length and bandwidth  $2W$ . Both the sequence length and bandwidth affect how many Slepian sequences have concentration ratios near one. As a rule, there are  $2NW - 1$  Slepian sequences with energy concentration ratios approximately equal to one. Beyond  $2NW - 1$  Slepian sequences, the concentration ratios begin to approach zero. Common choices for the time half bandwidth product are: 2.5, 3, 3.5, and 4.

You can specify the bandwidth of the Slepian sequences in Hz by defining the time half bandwidth product as  $NW/F_s$ , where  $F_s$  is the sampling frequency.

## References

Percival, D. B., and A. T. Walden. *Spectral Analysis for Physical Applications*. Cambridge, UK: Cambridge University Press, 1993.

## See Also

`dpss` | `dpssc` | `dpssc` | `dpssdir` | `dpssload`

## dspdata

DSP data parameter information

### Syntax

```
Hs = dspdata.dataobj(input1,...)
```

### Description

---

**Note:** The use of `dspdata.dataobj` is not recommended. Use the appropriate function interface instead.

---

`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.

| <code>dspdata.dataobj</code>  | Description                                   | Corresponding Functions                         |
|-------------------------------|---|---|
| <code>dspdata.msspectr</code> | Mean-square spectrum data (power)             | <code>periodogram</code><br><code>pwelch</code> |
| <code>dspdata.psd</code>      | Power spectral density data (power/frequency) | <code>pburg</code><br><code>pcov</code>         |

| <code>dspdata.dataobj</code>  | Description                 | Corresponding Functions  |
|-------------------------------|-----------------------------|--|
|                               |                             | <p>periodogram</p> <p>pmcov</p> <p>pmtm</p> <p>pwelch</p> <p>pyulear</p> |
| <code>dspdata.pseudosp</code> | Pseudospectrum data (power) | <p>peig</p> <p>pmusic</p>  |

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  |
|-----------------------|--|
| <code>avgpower</code> | <p>This method applies only to <code>dspdata.psd</code> objects.</p> <p><code>avgpower(Hs)</code> computes the average power of a signal, <code>Hs</code>, in a given frequency band. The technique uses a rectangle approximation of the integral of the 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. The <code>SpectrumType</code> property determines whether the total average power is contained in the one-sided or the two-sided spectrum. For a one-sided spectrum, the range is <code>[0,pi]</code> if the number of frequency points is even and <code>[0,pi]</code> if it is odd. For a two-sided spectrum, the range is <code>[0,2pi]</code>.</p> <p><code>avgpower(Hs, freqrangle)</code> specifies the frequency range over which to calculate the average power. <code>freqrange</code> is a two-element vector containing the lower and upper bounds</p> |

| Method                       | Description  |
|------------------------------|--|
|                              | <p>of the frequency range. If a frequency value does not match exactly the frequency in <code>HS</code>, the next closest value is used. The first frequency value in <code>freqrange</code> is included in the calculation and the second value is excluded.</p>  |
| <p><code>centerdc</code></p> | <p><code>centerdc(Hs)</code> or <code>centerdc(Hs, true)</code> shifts the data and frequency values so that the DC component is at the center of the spectrum. If the <code>SpectrumType</code> property is <code>'onesided'</code>, it is changed to <code>'twosided'</code> and then the DC component is centered.</p> <p><code>centerdc(Hs, 'false')</code> shifts the data and frequency values so that the DC component is at the left edge of the spectrum.</p> |

| Method    | Description   |
|-----------|---|
| findpeaks | <p><code>findpeaks(Hs)</code> finds local maxima or peaks. If no peaks are found, <code>findpeaks</code> returns an empty vector.</p> <p><code>[pks,frqs] = findpeaks(x)</code> returns the peaks' values, <code>pks</code>, and the frequencies, <code>frqs</code>, at which they occur.</p> <p><code>findpeaks(x, 'minpeakheight', mph)</code> returns only peaks greater than the minimum peak height <code>mph</code>, where <code>mph</code> is a real scalar. The default is <code>-Inf</code>.</p> <p><code>findpeaks(x, 'minpeakdistance', mpd)</code> returns only peaks separated by the minimum frequency units distance <code>mpd</code>, which is a positive integer. Setting the minimum peak distance ignores smaller peaks that may occur close to larger local peaks. The default is 1.</p> <p><code>findpeaks(x, 'threshold', th)</code> returns only peaks greater than their neighbors by at least the threshold, <code>th</code>, which is a real, scalar value greater than or equal to 0. The default is 0.</p> <p><code>findpeaks(x, 'npeaks', np)</code> returns a maximum of <code>np</code> number of peaks. When <code>np</code> peaks are found, the search stops. The default is to return all peaks.</p> <p><code>findpeaks(x, 'sortstr', str)</code> specifies the sorting order, where <code>str</code> is <code>'ascend'</code>, <code>'descend'</code>, or <code>'none'</code>. When <code>str</code> is set to <code>'ascend'</code>, the peaks are sorted from smallest to largest. When <code>str</code> is set to <code>'descend'</code> the peaks are sorted in descending order. When <code>str</code> is set to <code>'none'</code>, the peaks are returned in the order in which they occur.</p> |

| Method                            | Description  |
|-----------------------------------|--|
| <p><code>halfrange</code></p>     | <p><code>halfrange(Hs)</code> converts the spectrum of <code>Hs</code> 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 <code>dspdata.pseudospectrum</code> objects.</p> <p>The spectrum is assumed to be from a real signal. That is, <code>halfrange</code> uses half the data points regardless of whether the data is symmetric.</p>   |
| <p><code>normalizefreq</code></p> | <p><code>normalizefreq(Hs)</code> or <code>normalizefreq(Hs,true)</code> normalizes the frequency specifications in the <code>Hs</code> object to <code>Fs</code> so the frequencies are between 0 and 1. It also sets the <code>NormalizedFrequency</code> property to <code>true</code>.</p> <p><code>normalizefreq(Hs,false)</code> converts the frequencies to linear frequencies.</p> <p><code>normalizefreq(Hs,false,Fs)</code> sets a new sampling frequency, <code>Fs</code>. This can be used only with <code>false</code>.</p> |
| <p><code>onesided</code></p>      | <p><code>onesided(Hs)</code> converts the spectrum of <code>Hs</code> 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 <code>dspdata.psd</code> and <code>dspdata.msspectrum</code> objects.</p> <p>The spectrum is assumed to be from a real signal. That is, <code>onesided</code> uses half the data points regardless of whether the data is symmetric.</p>          |

| Method   | Description   |
|----------|---|
| plot     | <p>Displays the data graphically in the current figure window.</p> <p>For a <code>dspdata.psd</code> object, it displays the power spectral density in dB/Hz.</p> <p>For a <code>dspdata.msspectrum</code> object, it displays the mean-square in dB.</p> <p>For a <code>dspdata.pseudospectrum</code> object, it displays the pseudospectrum in dB.</p>  |
| sfdr     | <p>This method applies only to <code>dspdata.msspectrum</code> objects.</p> <p><code>sfdr(Hs)</code> computes the spurious-free dynamic range (SFDR) in dB of a mean square spectrum object <code>Hs</code>. SFDR is the usable range before spurious noise interferes with the signal.</p> <p><code>[sfd, spur, frq] = sfdr(Hs)</code> returns the magnitude of the highest spur and the frequency <code>frq</code> at which it occurs.</p> <p><code>sfdr(Hs, 'minspurlevel', msl)</code> ignores spurs below the minimum spur level <code>msl</code>, which is a real scalar in dB.</p> <p><code>sfdr(Hs, 'minspurdistance', msd)</code> includes spurs only if they are separated by at least the minimum spur distance <code>msd</code>, which is a real, positive scalar in frequency units.</p> |
| twosided | <p><code>twosided(Hs)</code> converts the <code>Hs</code> 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 <code>dspdata.psd</code> and <code>dspdata.msspectrum</code> objects.</p> <p>If your data is nonuniformly sampled, converting from <code>onesided</code> to <code>twosided</code> may produce incorrect results.</p>   |

| Method     | Description  |
|------------|--|
| wholerange | <p><code>wholerange(Hs)</code> converts the <code>Hs</code> 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 <code>dspdata.pseudospectrum</code> objects.</p> <p>If your data is nonuniformly sampled, converting from <code>half</code> to <code>wholerange</code> may produce incorrect results.</p> |

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.

## 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 an object, `Hs`, from two-sided to one-sided, use `onesided(Hs)`.

## Examples

See the `dspdata.msspectrum`, `dspdata.psd`, and `dspdata.pseudospectrum` reference pages for specific examples.

## See Also

`pburg` | `pcov` | `peig` | `periodogram` | `pmcov` | `pmtm` | `pmusic` | `pwelch` | `pyulear`



# dspdata.msspectrum

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

---

**Note:** The use of `dspdata.msspectrum` is not recommended. Use `periodogram` or `pwelch` instead.

---

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 | [ ]<br>type double     | <p>Vector of frequencies at which the spectrum is evaluated. The range of this vector depends on the <b>SpectrumType</b> 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 <b>Fs</b> is specified. For a two-sided spectrum, it is [0, 2pi) or [0, Fs).</p> <p>The length of the <b>Frequencies</b> vector must match the length of the columns of <b>Data</b>.</p> <p>If you do not specify <b>Frequencies</b>, a default vector is created. If one-sided is selected, then the whole number of FFT points (<b>nFFT</b>) for this vector is assumed to be even.</p> <p>If <b>onesided</b> is selected and you specify <b>Frequencies</b>, the last frequency point is compared to the next-to-last point and to pi (or <b>Fs/2</b>, if <b>Fs</b> is specified). If the last point is closer to pi (or <b>Fs/2</b>) than it is to the previous point, <b>nFFT</b> is assumed to be even. If it is closer to the previous point, <b>nFFT</b> is assumed to be odd.</p> |
| Fs          | 'Normalized'           | Sampling frequency, which is 'Normalized' if <b>NormalizedFrequency</b> is true. If <b>NormalizedFrequency</b> is false <b>Fs</b> 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 <code>onesided</code> and <code>twosided</code> methods in <code>dspdata</code> for information on changing this property.<br><br>The interval for <code>Onesided</code> is $[0 \pi]$ or $[0 \pi]$ depending on the number of FFT points, and for <code>Twosided</code> the interval is $[0 2\pi]$ . |
| NormalizedFrequency | true          | Whether the frequency is normalized ( <code>true</code> ) or not ( <code>false</code> ). This property is set automatically at construction time based on <code>Fs</code> . If <code>Fs</code> is specified, <code>NormalizedFrequency</code> is set to <code>false</code> . See the <code>normalizefreq</code> method in <code>dspdata</code> 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  $[0 2\pi]$ , 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`
- `sfd`
- `twosided`

For example, to normalize the frequency and set the `NormalizedFrequency` parameter to `true`, use

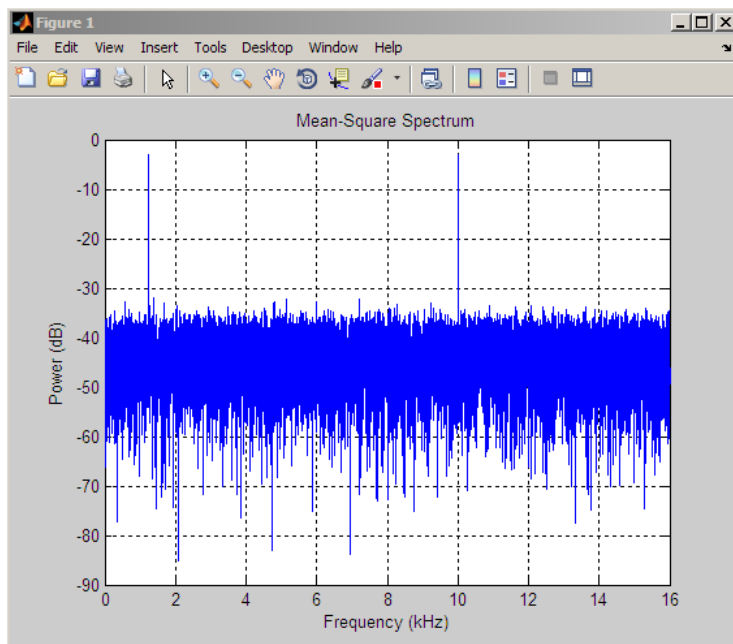
```
Hmss = normalizefreq(Hs)
```

For detailed information on using the methods and plotting the spectrum, see the `dspdata` reference page.

## Examples

In this example, we construct a mean-square spectrum data object from the one-sided PSD estimate of a signal. The signal consists of two sinusoids in additive noise.

```
Fs = 32e3;
t = 0:1/Fs:1-(1/Fs);
x = cos(2*pi*t*1.24e3)+cos(2*pi*t*10e3)+randn(size(t));
X = fft(x);
X = X(1:length(X)/2+1); % One-sided DFT
P = (abs(X)/length(x)).^2; % Compute the mean-square power
P(2:end-1) = 2*P(2:end-1); % Factor of two for one-sided estimate
% at all frequencies except zero and the Nyquist
Hmss = dspdata.msspectrum(P,'Fs',Fs,'spectrumtype','onesided');
plot(Hmss) % Plot the mean-square spectrum.
```



## See Also

periodogram | pwelch

## dspdata.psd

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

---

**Note:** The use of `dspdata.psd` is not recommended. Use `pburg`, `pcov`, `periodogram`, `pmtm`, `pwelch`, or `pyulear` instead.

---

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 | [ ]<br>type double | <p>Vector of frequencies at which the power spectral density is evaluated. The range of this vector depends on the <b>SpectrumType</b> 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 <b>Fs</b> is specified. For two-sided, it is [0, 2pi) or [0, Fs).</p> <p>If you do not specify <b>Frequencies</b>, a default vector is created. If one-sided is selected, then the whole number of FFT points (<b>nFFT</b>) for this vector is assumed to be even.</p> <p>If <b>onesided</b> is selected and you specify <b>Frequencies</b>, the last frequency point is compared to the next-to-last point and to pi (or Fs/2, if <b>Fs</b> is specified). If the last point is closer to pi (or Fs/2) than it is to the previous point, <b>nFFT</b> is assumed to be even. If it is closer to the previous point, <b>nFFT</b> is assumed to be odd.</p> <p>The length of the <b>Frequencies</b> vector must match the length of the columns of <b>Data</b>.</p> |
| Fs          | 'Normalized '      | <p>Sampling frequency, which is 'Normalized' if <b>NormalizedFrequency</b> is true. If <b>NormalizedFrequency</b> is false <b>Fs</b> defaults to 1.</p>   |

| Property            | Default Value | Description   |
|---------------------|---------------|---|
| SpectrumType        | 'Onesided'    | <p>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 <code>onesided</code> and <code>twosided</code> methods in <code>dspdata</code> for information on changing this property.</p> <p>The range for half the Nyquist interval is <math>[0 \pi)</math> or <math>[0 \pi]</math> depending on the number of FFT points. For the whole Nyquist interval, the range is <math>[0 2\pi)</math>.</p> |
| NormalizedFrequency | true          | <p>Whether the frequency is normalized (<code>true</code>) or not (<code>false</code>). This property is set automatically at construction time based on <code>Fs</code>. If <code>Fs</code> is specified, <code>NormalizedFrequency</code> is set to <code>false</code>. See the <code>normalizefreq</code> method in <code>dspdata</code> for information on changing this property.</p>  |

`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  $[0 2\pi)$ , 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)
```

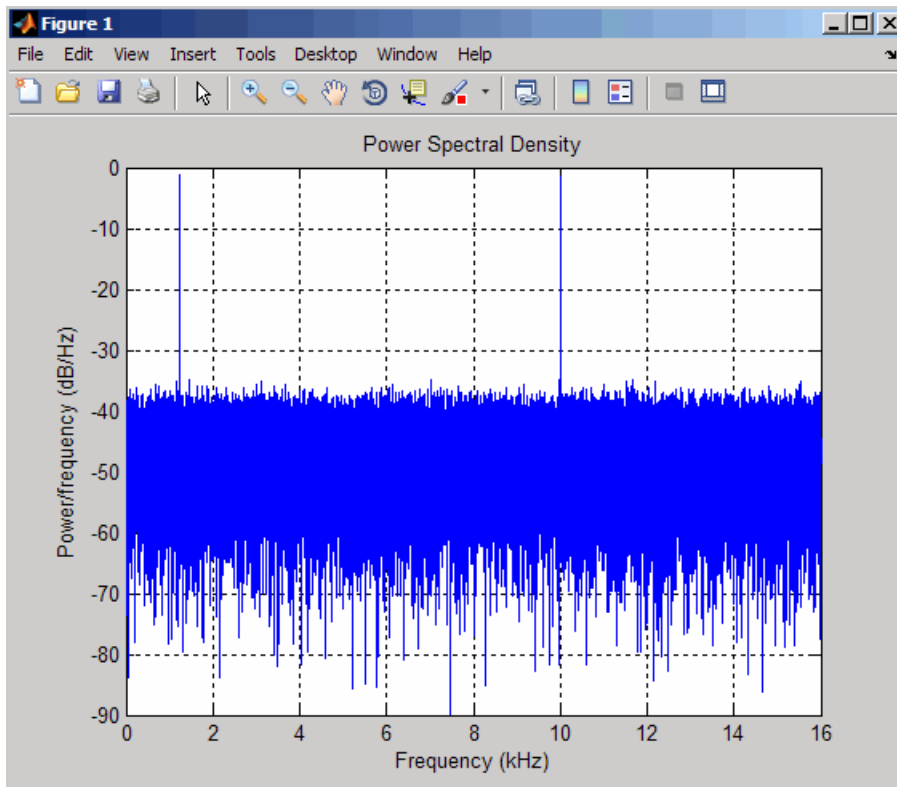
For detailed information on using the methods and plotting the spectrum, see the `dspdata` reference page.

## Examples

### Resolving Signal Components

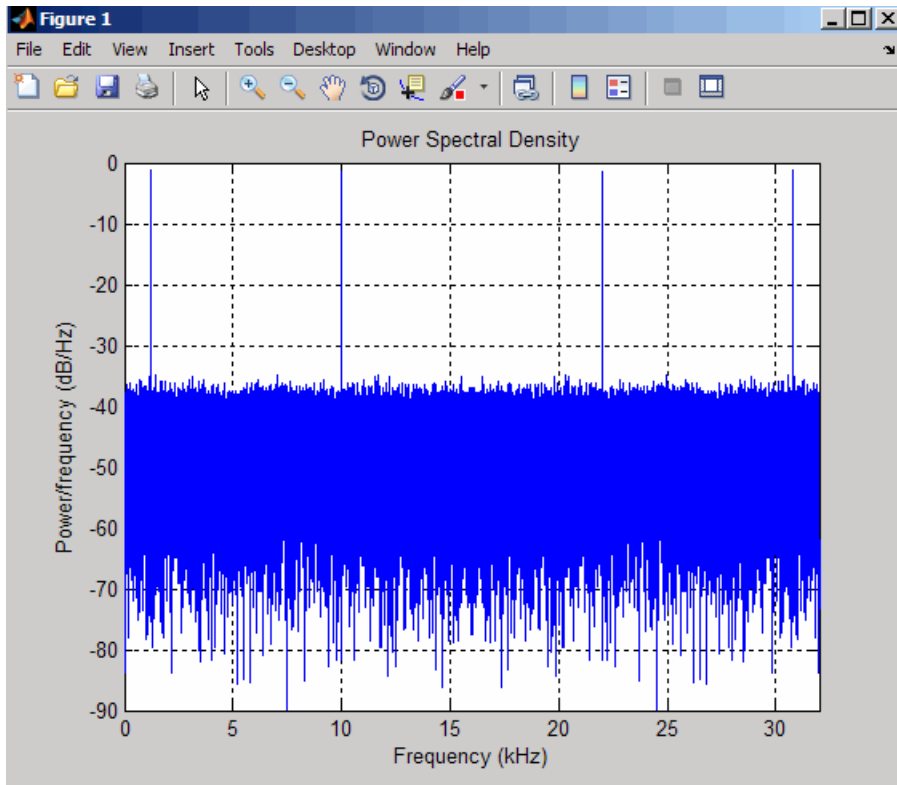
Estimate the power spectral density of a noisy sinusoidal signal with two frequency components and then store the results in a PSD data object and plot it.

```
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

[pburg](#) | [pcov](#) | [periodogram](#) | [pmcov](#) | [pmtm](#) | [pwelch](#) | [pyulear](#)

## dspdata.pseudospectrum

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

---

**Note:** The use of `dspdata.pseudospectrum` is not recommended. Use `peig` or `pmusic` instead.

---

A pseudospectrum is an indicator of the presence of sinusoidal components in a signal.

`Hps = dspdata.pseudospectrum(Data)` uses the pseudospectrum 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 | [ ]<br>type double | <p>Vector of frequencies at which the power spectral density is evaluated. The range of this vector depends on the <b>SpectrumRange</b> 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 <b>Fs</b> is specified. For whole, it is [0, 2pi) or [0, Fs).</p> <p>If you do not specify <b>Frequencies</b>, a default vector is created. If half the Nyquist range is selected, then the whole number of FFT points (<b>nFFT</b>) for this vector is assumed to be even.</p> <p>If half the Nyquist range is selected and you specify <b>Frequencies</b>, the last frequency point is compared to the next-to-last point and to pi (or <b>Fs/2</b>, if <b>Fs</b> is specified). If the last point is closer to pi (or <b>Fs/2</b>) than it is to the previous point, <b>nFFT</b> is assumed to be even. If it is closer to the previous point, <b>nFFT</b> is assumed to be odd.</p> <p>The length of the <b>Frequencies</b> vector must match the length of the columns of <b>Data</b>.</p> |
| Fs          | 'Normalized'       | Sampling frequency, which is 'Normalized' if <b>NormalizedFrequency</b> is true. If <b>NormalizedFrequency</b> is false <b>Fs</b> defaults to 1.  |

| Property            | Default Value | Description   |
|---------------------|---------------|---|
| SpectrumRange       | 'Half'        | <p>Nyquist interval over which the pseudospectrum is calculated. Valid values are 'Half' and 'Whole'. See the <code>half</code> and <code>whole</code> methods in <code>dspdata</code> for information on changing this property.</p> <p>The interval for <code>Half</code> is <math>[0 \text{ pi}]</math> or <math>[0 \text{ pi}]</math> depending on the number of FFT points, and for <code>Whole</code> the interval is <math>[0 \text{ 2pi}]</math>.</p> |
| NormalizedFrequency | true          | <p>Whether the frequency is normalized (<code>true</code>) or not (<code>false</code>). This property is set automatically at construction time based on <code>Fs</code>. If <code>Fs</code> is specified, <code>NormalizedFrequency</code> is set to <code>false</code>. See the <code>normalizefreq</code> method in <code>dspdata</code> for information on changing this property.</p>  |

`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  $[0 \text{ pi}]$  or  $[0 \text{ pi}]$ , set the `SpectrumRange` to `half`; for data that ranges from  $[0 \text{ 2pi}]$ , 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`
- `wholerrange`

For example, to normalize the frequency and set the `NormalizedFrequency` parameter to true, use

```
Hps = normalizefreq(Hps)
```

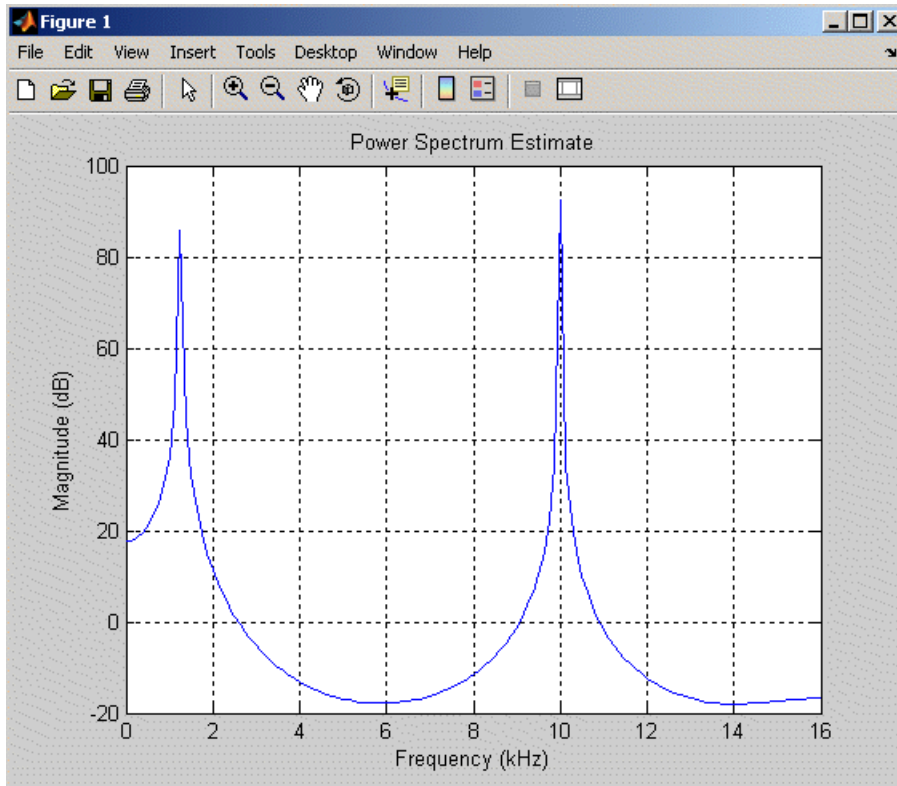
For detailed information on using the methods and plotting the pseudospectrum, see the `dspdata` reference page.

## 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**  
peig | pmusic



# dspfwiz

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.

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 Delay blocks, in a Simulink model.

If you also have the DSP System Toolbox software installed, you can create a Biquad Filter block or a Discrete FIR Filter block instead of a subsystem block, by deselecting the **Build model using basic elements** check box.

## See Also

fdatool | dfilt

# dutycycle

Duty cycle of pulse waveform

## Syntax

```
D = dutycycle(X)
D = dutycycle(X,FS)
D = dutycycle(X,T)
D = dutycycle(TAU,PRF)
[D,INITCROSS] = dutycycle(X,...)
[D,INITCROSS,FINALCROSS] = dutycycle(X,...)
[D,INITCROSS,FINALCROSS,NEXTCROSS] = dutycycle(X,...)
[D,INITCROSS,FINALCROSS,NEXTCROSS,MIDLEV] = dutycycle(X,...)
[D,INITCROSS,FINALCROSS,NEXTCROSS] = dutycycle(X,...,Name,Value)
dutycycle(X,...)
```

## Description

`D = dutycycle(X)` returns the ratio of pulse width to pulse period for each positive-polarity pulse. `D` has length equal to the number of pulse periods in `X`. The sample instants of `X` correspond to the indices of `X`. To determine the transitions that define each pulse, `dutycycle` estimates the state levels of the input waveform by a histogram method. `dutycycle` identifies all regions, which cross the upper-state boundary of the low state and the lower-state boundary of the high state. The low-state and high-state boundaries are expressed as the state level plus or minus a multiple of the difference between the state levels. See “State-Level Tolerances” on page 1-386.

`D = dutycycle(X,FS)` specifies the sampling frequency, `FS`, in hertz as a positive scalar. The first sample instant of `X` corresponds to `t=0`.

`D = dutycycle(X,T)` specifies the sample instants, `T`, as a vector with the same number of elements as `X`.

`D = dutycycle(TAU,PRF)` returns the ratio of pulse width to pulse period for a pulse width of `TAU` seconds and a pulse repetition frequency of `PRF`. The product of `TAU` and `PRF` must be less than or equal to 1.

`[D, INITCROSS] = dutyCycle(X, ...)` returns a vector, `INITCROSS`, whose elements correspond to the mid-crossings (mid-reference level instants) of the initial transition of each pulse with a corresponding `NEXTCROSS`.

`[D, INITCROSS, FINALCROSS] = dutyCycle(X, ...)` returns a vector, `FINALCROSS`, whose elements correspond to the mid-crossings (mid-reference level instants) of the final transition of each pulse with a corresponding `NEXTCROSS`.

`[D, INITCROSS, FINALCROSS, NEXTCROSS] = dutyCycle(X, ...)` returns a vector, `NEXTCROSS`, whose elements correspond to the mid-crossings (mid-reference level instants) of the next detected transition for each pulse.

`[D, INITCROSS, FINALCROSS, NEXTCROSS, MIDLEV] = dutyCycle(X, ...)` returns the mid-reference level, `MIDLEV`. Because in a bilevel pulse waveform the state levels are constant, `MIDLEV` is a scalar.

`[D, INITCROSS, FINALCROSS, NEXTCROSS] = dutyCycle(X, ..., Name, Value)` returns the ratio of pulse width to pulse period with additional options specified by one or more `Name, Value` pair arguments.

`dutyCycle(X, ...)` plots the waveform, `X`, and marks the location of the mid-reference level instants and the associated reference levels. The state levels and associated lower and upper state boundaries are also plotted.

## Input Arguments

### **X**

Bilevel waveform. `X` is a real-valued row or column vector.

### **FS**

Sample rate in hertz.

### **T**

Vector of sample instants. The length of `T` must equal the length of the bilevel waveform, `X`.

### **TAU**

Pulse width in seconds. The product of `TAU` and `PRF` must be less than or equal to 1.

**PRF**

Pulse repetition frequency in pulses/second. The product of TAU and PRF must be less than or equal to 1.

**Name-Value Pair Arguments****'MidPercentReferenceLevel'**

Mid-reference level as a percentage of the waveform amplitude.

**Default:** 50

**'Polarity'**

Pulse polarity. Specify the polarity as 'positive' or 'negative'. If you specify 'positive', `dutycycle` looks for pulses with positive-going (positive polarity) initial transitions. If you specify 'negative', `dutycycle` looks for pulses with negative-going (negative polarity) initial transitions. See “Pulse Polarity” on page 1-385 for examples of positive and negative-polarity pulses.

**Default:** 'positive'

**'StateLevels'**

Low- and high-state levels. `StateLevels` is a 1-by-2 real-valued vector. The first element is the low-state level. The second element is the high-state level. If you do not specify low- and high-state levels, `dutycycle` estimates the state levels from the input waveform using the histogram method.

**'Tolerance'**

Tolerance levels (lower- and upper-state boundaries) expressed as a percentage. See “State-Level Tolerances” on page 1-386.

**Default:** 2

**Output Arguments****D**

Duty cycle. Duty cycle is the ratio of the pulse width to the pulse period. Because the pulse width cannot exceed the pulse period,  $0 \leq D \leq 1$ .

**INITCROSS**

Mid-reference level instant of initial transition. Because the duty cycle is defined as the ratio of pulse width to pulse period, initial transitions are only reported when `dutycycle` finds a corresponding `NEXTCROSS`.

**FINALCROSS**

Mid-reference level instant of final transition. The duty cycle is defined as the ratio of pulse width to pulse period. Thus, final transitions are only reported when `dutycycle` finds a corresponding `NEXTCROSS`.

**NEXTCROSS**

Mid-reference level instant of the first initial transition after the final transition of the preceding pulse.

**MIDLEV**

Mid-reference level. The waveform value that corresponds to the mid-reference level.

## Examples

**Duty Cycle of Bilevel Waveform**

Determine the duty cycle of a bilevel waveform. Use the vector indices as the sample instants.

```
load('pulseex.mat', 'x');  
d = dutycycle(x);
```

**Duty Cycle of Bilevel Waveform with Sampling Frequency**

Determine the duty cycle of a bilevel waveform. The sampling frequency is 4 MHz.

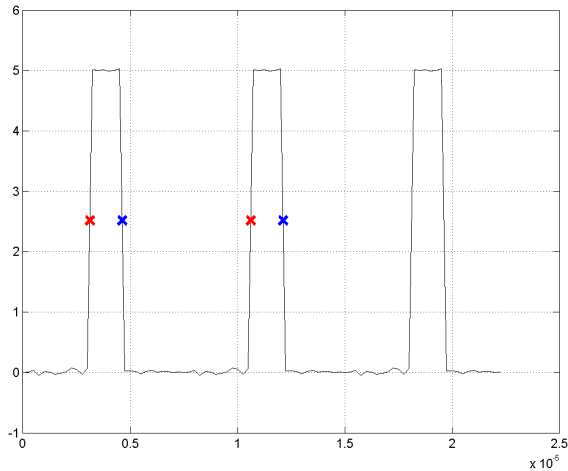
```
load('pulseex.mat', 'x', 't');  
fs = 1/(t(2)-t(1));  
d = dutycycle(x,fs);
```

**Duty Cycle of Bilevel Waveform with Three Pulses**

Create a pulse waveform with three pulses. The sampling frequency is 4 MHz. Determine the initial and final mid-reference level instants. Plot the result.

Even though there are three pulses, only two pulses have corresponding subsequent transitions.

```
load('pulseex.mat','x');
dt = 1/4e6;
ts = reshape repmat(x(1:30),1,3),90,1);
t = 0:dt:(length(ts)*dt)-dt;
[d,initcross,finalcross,~,midlev] = dutycycle(ts,t);
plot(t,ts,'k'); hold on; grid on;
h0 = plot(initcross, midlev*ones(length(initcross)),'rx');
set(h0,'markersize',10,'linewidth',2.5);
h1 = plot(finalcross,midlev*ones(length(finalcross)),'bx');
set(h1,'markersize',10,'linewidth',2.5);
```



## More About

### Duty Cycle

The energy in a bilevel, or rectangular, pulse is equal to the product of the peak power,  $Pt$ , and the pulse width,  $\tau$ . Devices to measure energy in a waveform operate on time scales longer than the duration of a single pulse. Therefore, it is common to measure the average power

$$P_{\text{av}} = \frac{P_t \tau}{T},$$

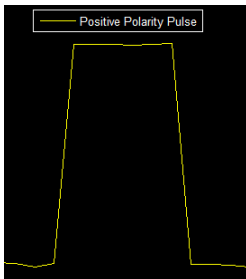
where  $T$  is the pulse period.

The ratio of average power to peak power is the duty cycle:

$$D = \frac{P_t \tau / T}{P_t}$$

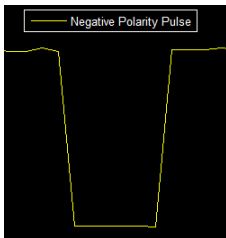
### Pulse Polarity

If the pulse has a positive-going initial transition, the pulse has positive polarity. The following figure shows a positive polarity pulse.



Equivalently, a positive-polarity (positive-going) pulse has a terminating state more positive than the originating state.

If the pulse has a negative-going initial transition, the pulse has negative polarity. The following figure shows a negative-polarity pulse.



Equivalently, a negative-polarity (negative-going) pulse has a originating state more positive than the terminating state.

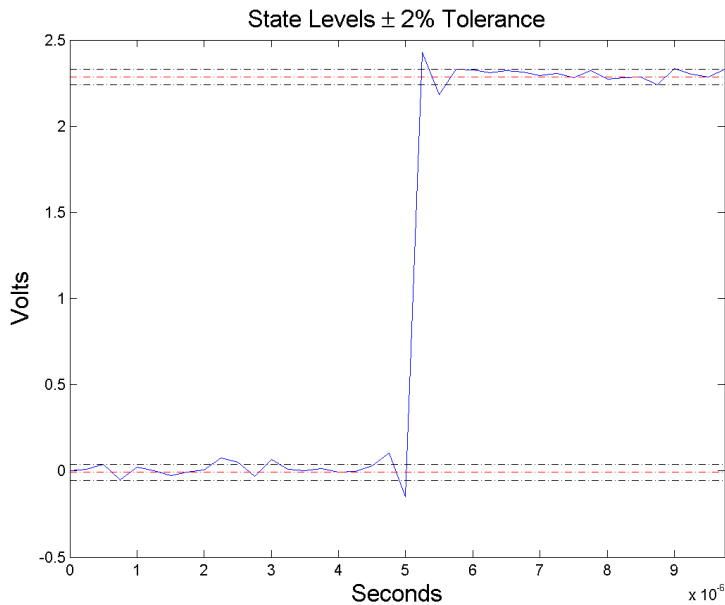
### State-Level Tolerances

Each state level can have an associated lower- and upper-state boundary. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference between the high state and low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the  $\alpha\%$  tolerance region for the low state is defined as

$$S_1 \pm \frac{\alpha}{100}(S_2 - S_1)$$

where  $S_1$  is the low-state level and  $S_2$  is the high-state level. Replace the first term in the equation with  $S_2$  to obtain the  $\alpha\%$  tolerance region for the high state.

The following figure illustrates lower and upper 2% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The estimated state levels are indicated by a dashed red line.





## References

[1] Skolnik, M.I. *Introduction to Radar Systems*. New York, NY: McGraw-Hill, 1980.

[2] *IEEE Standard on Transitions, Pulses, and Related Waveforms*. IEEE Standard 181, 2003.

## See Also

midcross | pulseperiod | pulsesep | pulsewidth

# ellip

Elliptic filter design

## Syntax

```
[b,a] = ellip(n,Rp,Rs,Wp)
[b,a] = ellip(n,Rp,Rs,Wp,ftype)

[z,p,k] = ellip(____)
[A,B,C,D] = ellip(____)

[____] = ellip(____,'s')
```

## Description

`[b,a] = ellip(n,Rp,Rs,Wp)` returns the transfer function coefficients of an  $n$ th-order lowpass digital elliptic filter with normalized passband edge frequency  $W_p$ . The resulting filter has  $R_p$  decibels of peak-to-peak passband ripple and  $R_s$  decibels of stopband attenuation down from the peak passband value.

`[b,a] = ellip(n,Rp,Rs,Wp,ftype)` designs a lowpass, highpass, bandpass, or bandstop elliptic filter, depending on the value of `ftype` and the number of elements of  $W_p$ . The resulting bandpass and bandstop designs are of order  $2n$ .

---

**Note:** See “Limitations” on page 1-398 for information about numerical issues that affect forming the transfer function.

---

`[z,p,k] = ellip(____)` designs a lowpass, highpass, bandpass, or bandstop digital elliptic filter and returns its zeros, poles, and gain. This syntax can include any of the input arguments in previous syntaxes.

`[A,B,C,D] = ellip(____)` designs a lowpass, highpass, bandpass, or bandstop digital elliptic filter and returns the matrices that specify its state-space representation.

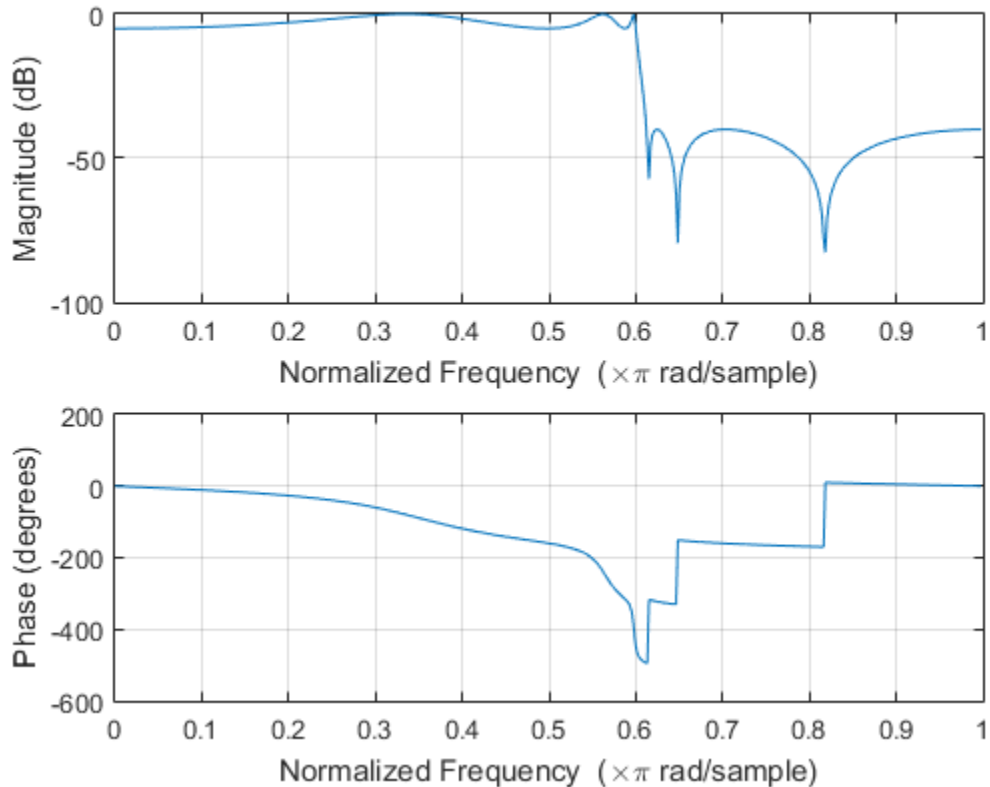
[ \_\_\_ ] = `ellip( ___, 's' )` designs a lowpass, highpass, bandpass, or bandstop analog elliptic filter with passband edge angular frequency  $W_p$ ,  $R_p$  decibels of passband ripple, and  $R_s$  decibels of stopband attenuation.

## Examples

### Lowpass Elliptic Transfer Function

Design a 6th-order lowpass elliptic filter with 5 dB of passband ripple, 40 dB of stopband attenuation, and a passband edge frequency of 300 Hz, which, for data sampled at 1000 Hz, corresponds to  $0.6\pi$  rad/sample. Plot its magnitude and phase responses. Use it to filter a 1000-sample random signal.

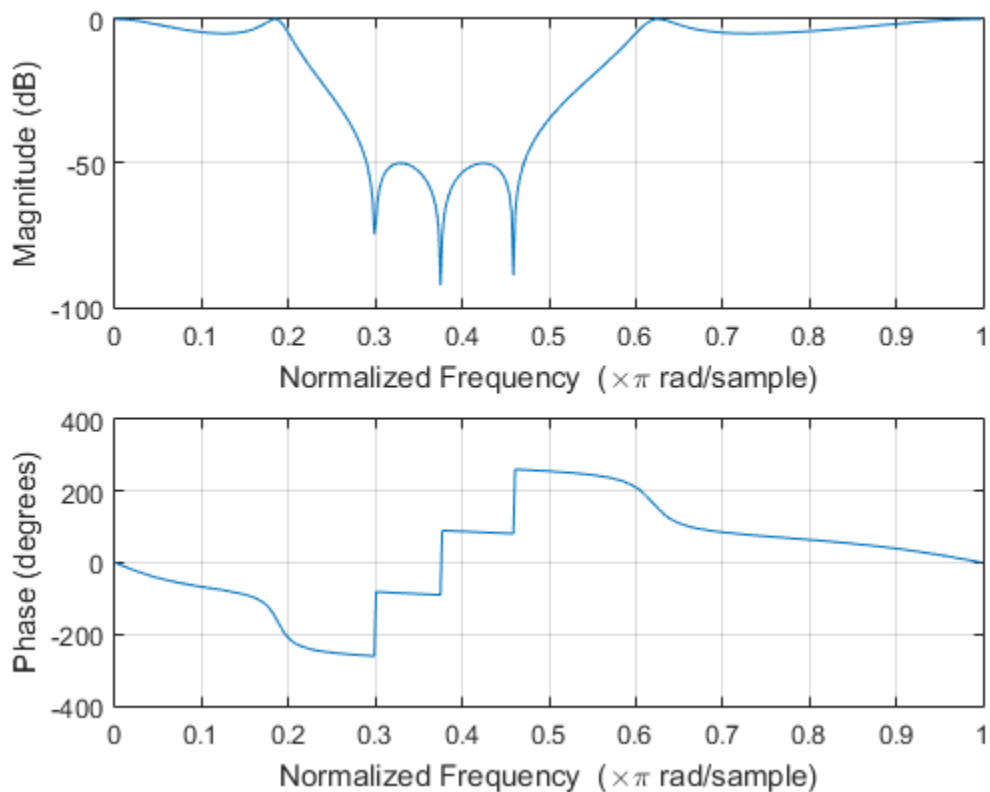
```
[b,a] = ellip(6,5,40,0.6);  
freqz(b,a)  
dataIn = randn(1000,1);  
dataOut = filter(b,a,dataIn);
```



### Bandstop Elliptic Filter

Design a 6th-order elliptic bandstop filter with normalized edge frequencies of  $0.2\pi$  and  $0.6\pi$  rad/sample, 5 dB of passband ripple, and 50 dB of stopband attenuation. Plot its magnitude and phase responses. Use it to filter random data.

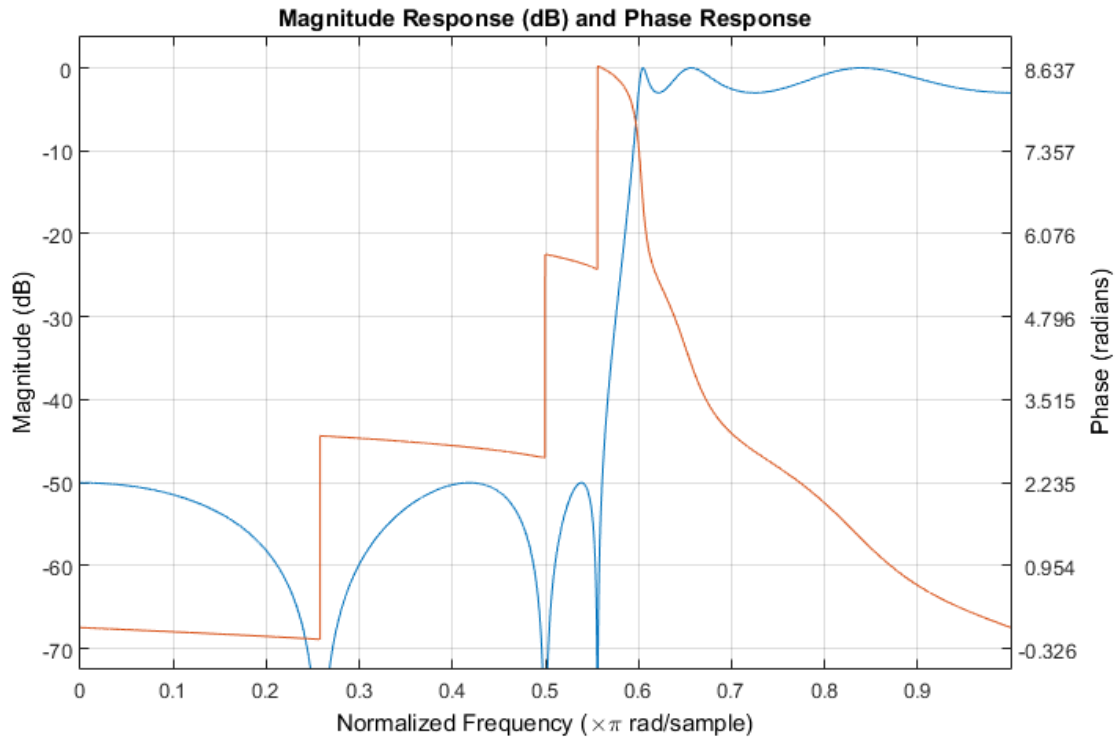
```
[b,a] = ellip(3,5,50,[0.2 0.6], 'stop');  
freqz(b,a)  
dataIn = randn(1000,1);  
dataOut = filter(b,a,dataIn);
```



### Highpass Elliptic Filter

Design a 6th-order highpass elliptic filter with a passband edge frequency of 300 Hz, which, for data sampled at 1000 Hz, corresponds to  $0.6\pi$  rad/sample. Specify 3 dB of passband ripple and 50 dB of stopband attenuation. Plot the magnitude and phase responses. Convert the zeros, poles, and gain to second-order sections for use by `fvtool`.

```
[z,p,k] = ellip(6,3,50,300/500,'high');
sos = zp2sos(z,p,k);
fvtool(sos,'Analysis','freq')
```



### Bandpass Elliptic Filter

Design a 20th-order elliptic bandpass filter with a lower passband frequency of 500 Hz and a higher passband frequency of 560 Hz. Specify a passband ripple of 3 dB, a stopband attenuation of 40 dB, and a sample rate of 1500 Hz. Use the state-space representation. Design an identical filter using `designfilt`.

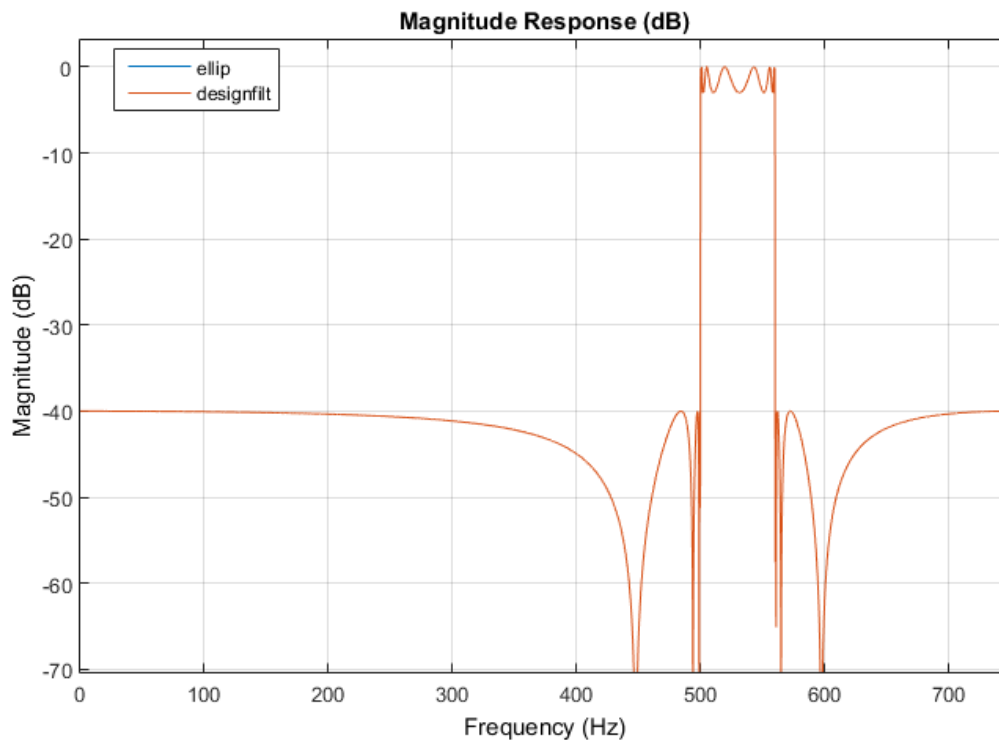
```
[A,B,C,D] = ellip(10,3,40,[500 560]/750);
d = designfilt('bandpassiir','FilterOrder',20, ...
    'PassbandFrequency1',500,'PassbandFrequency2',560, ...
    'PassbandRipple',3, ...
    'StopbandAttenuation1',40,'StopbandAttenuation2',40, ...
    'SampleRate',1500);
```

Convert the state-space representation to second-order sections. Visualize the frequency responses using `fvtool`.

```

sos = ss2sos(A,B,C,D);
fvt = fvtool(sos,d,'Fs',1500);
legend(fvt,'ellip','designfilt')

```



### Comparison of Analog IIR Lowpass Filters

Design a 5th-order analog Butterworth lowpass filter with a cutoff frequency of 2 GHz. Multiply by  $2\pi$  to convert the frequency to radians per second. Compute the frequency response of the filter at 4096 points.

```

n = 5;
f = 2e9;

```

```

[zb,pb,kb] = butter(n,2*pi*f,'s');
[bb,ab] = zp2tf(zb,pb,kb);
[hb,wb] = freqs(bb,ab,4096);

```

Design a 5th-order Chebyshev Type I filter with the same edge frequency and 3 dB of passband ripple. Compute its frequency response.

```
[z1,p1,k1] = cheby1(n,3,2*pi*f,'s');
[b1,a1] = zp2tf(z1,p1,k1);
[h1,w1] = freqs(b1,a1,4096);
```

Design a 5th-order Chebyshev Type II filter with the same edge frequency and 30 dB of stopband attenuation. Compute its frequency response.

```
[z2,p2,k2] = cheby2(n,30,2*pi*f,'s');
[b2,a2] = zp2tf(z2,p2,k2);
[h2,w2] = freqs(b2,a2,4096);
```

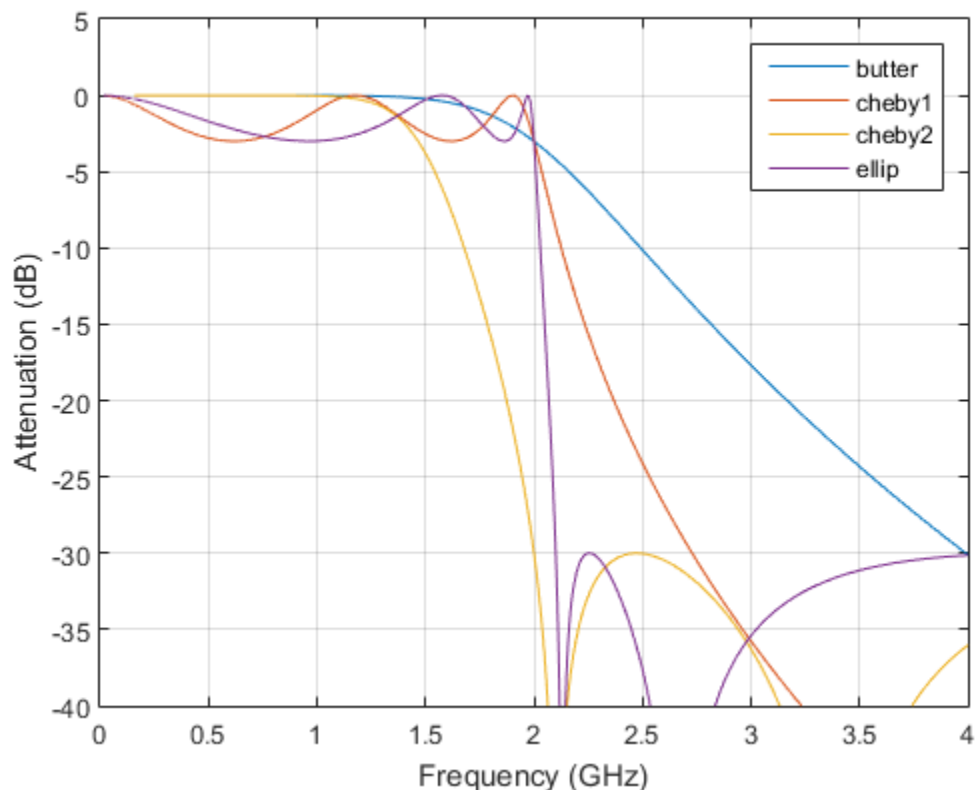
Design a 5th-order elliptic filter with the same edge frequency, 3 dB of passband ripple, and 30 dB of stopband attenuation. Compute its frequency response.

```
[ze,pe,ke] = ellip(n,3,30,2*pi*f,'s');
[be,ae] = zp2tf(ze,pe,ke);
[he,we] = freqs(be,ae,4096);
```

Plot the attenuation in decibels. Express the frequency in gigahertz. Compare the filters.

```
plot(wb/(2e9*pi),mag2db(abs(hb)))
hold on
plot(w1/(2e9*pi),mag2db(abs(h1)))
plot(w2/(2e9*pi),mag2db(abs(h2)))
plot(we/(2e9*pi),mag2db(abs(he)))
axis([0 4 -40 5])
grid
xlabel('Frequency (GHz)')
ylabel('Attenuation (dB)')
legend('butter','cheby1','cheby2','ellip')
```





The Butterworth and Chebyshev Type II filters have flat passbands and wide transition bands. The Chebyshev Type I and elliptic filters roll off faster but have passband ripple. The frequency input to the Chebyshev Type II design function sets the beginning of the stopband rather than the end of the passband.

## Input Arguments

### **n** — Filter order

integer scalar

Filter order, specified as an integer scalar.

Data Types: `double`

**Rp — Peak-to-peak passband ripple**

positive scalar

Peak-to-peak passband ripple, specified as a positive scalar expressed in decibels.

If your specification,  $\ell$ , is in linear units, you can convert it to decibels using  $R_p = 40 \log_{10}((1+\ell)/(1-\ell))$ .

Data Types: `double`

**Rs — Stopband attenuation**

positive scalar

Stopband attenuation down from the peak passband value, specified as a positive scalar expressed in decibels.

If your specification,  $\ell$ , is in linear units, you can convert it to decibels using  $R_s = -20 \log_{10}\ell$ .

Data Types: `double`

**Wp — Passband edge frequency**

scalar | two-element vector

Passband edge frequency, specified as a scalar or a two-element vector. The passband edge frequency is the frequency at which the magnitude response of the filter is  $-R_p$  decibels. Smaller values of passband ripple,  $R_p$ , and larger values of stopband attenuation,  $R_s$ , both result in wider transition bands.

- If  $W_p$  is a scalar, then `ellip` designs a lowpass or highpass filter with edge frequency  $W_p$ .

If  $W_p$  is the two-element vector  $[w_1 \ w_2]$ , where  $w_1 < w_2$ , then `ellip` designs a bandpass or bandstop filter with lower edge frequency  $w_1$  and higher edge frequency  $w_2$ .

- For digital filters, the passband edge frequencies must lie between 0 and 1, where 1 corresponds to the Nyquist rate—half the sample rate or  $\pi$  rad/sample.

For analog filters, the passband edge frequencies must be expressed in radians per second and can take on any positive value.

Data Types: `double`

**f<sub>type</sub>** — Filter type

'low' | 'bandpass' | 'high' | 'stop'

Filter type, specified as a string.

- 'low' specifies a lowpass filter with passband edge frequency  $W_p$ . 'low' is the default for scalar  $W_p$ .
- 'high' specifies a highpass filter with passband edge frequency  $W_p$ .
- 'bandpass' specifies a bandpass filter of order  $2n$  if  $W_p$  is a two-element vector. 'bandpass' is the default when  $W_p$  has two elements.
- 'stop' specifies a bandstop filter of order  $2n$  if  $W_p$  is a two-element vector.

Data Types: char

## Output Arguments

**b, a** — Transfer function coefficients

row vectors

Transfer function coefficients of the filter, returned as row vectors of length  $n + 1$  for lowpass and highpass filters and  $2n + 1$  for bandpass and bandstop filters.

- For digital filters, the transfer function is expressed in terms of **b** and **a** as

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

- For analog filters, the transfer function is expressed in terms of **b** and **a** as

$$H(s) = \frac{B(s)}{A(s)} = \frac{b(1)s^n + b(2)s^{n-1} + \dots + b(n+1)}{a(1)s^n + a(2)s^{n-1} + \dots + a(n+1)}.$$

Data Types: double

**z, p, k** — Zeros, poles, and gain

column vectors, scalar

Zeros, poles, and gain of the filter, returned as two column vectors of length  $n$  ( $2n$  for bandpass and bandstop designs) and a scalar.

- For digital filters, the transfer function is expressed in terms of  $z$ ,  $p$ , and  $k$  as

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

- For analog filters, the transfer function is expressed in terms of  $z$ ,  $p$ , and  $k$  as

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

Data Types: double

### **A, B, C, D — State-space matrices** matrices

State-space representation of the filter, returned as matrices. If  $m = n$  for lowpass and highpass designs and  $m = 2n$  for bandpass and bandstop filters, then **A** is  $m \times m$ , **B** is  $m \times 1$ , **C** is  $1 \times m$ , and **D** is  $1 \times 1$ .

- For digital filters, the state-space matrices relate the state vector  $x$ , the input  $u$ , and the output  $y$  through

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

- For analog filters, the state-space matrices relate the state vector  $x$ , the input  $u$ , and the output  $y$  through

$$\begin{aligned} \dot{x} &= Ax + Bu \\ y &= Cx + Du. \end{aligned}$$

Data Types: double

## More About

### Limitations

### Numerical Instability of Transfer Function Syntax

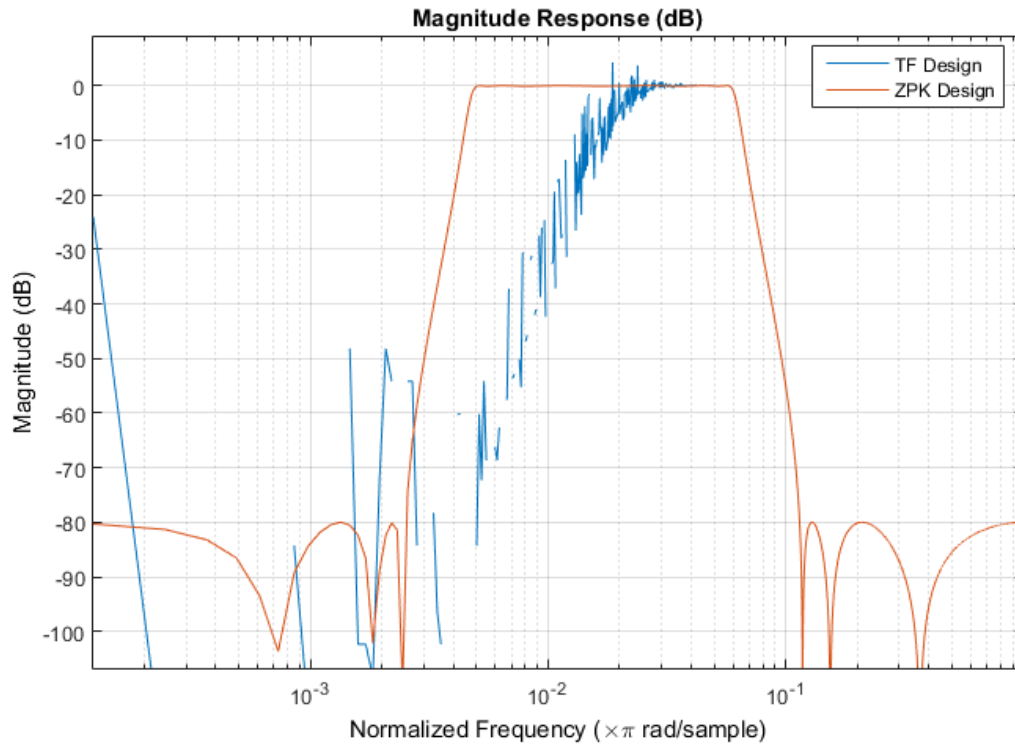
In general, use the `[z,p,k]` syntax to design IIR filters. To analyze or implement your filter, you can then use the `[z,p,k]` output with `zp2sos`. If you design the filter using the `[b,a]` syntax, you might encounter numerical problems. These problems are due to round-off errors and can occur for `n` as low as 4. The following example illustrates this limitation.

```
n = 6;
Rp = 0.1;
Rs = 80;
Wn = [2.5e6 29e6]/500e6;
ftype = 'bandpass';

% Transfer Function design
[b,a] = ellip(n,Rp,Rs,Wn,ftype);           % This filter is unstable

% Zero-Pole-Gain design
[z,p,k] = ellip(n,Rp,Rs,Wn,ftype);
sos = zp2sos(z,p,k);

% Plot and compare the results
hfvt = fvtool(b,a,sos,'FrequencyScale','log');
legend(hfvt,'TF Design','ZPK Design')
```



## Algorithms

Elliptic filters offer steeper rolloff characteristics than Butterworth or Chebyshev filters, but are equiripple in both the passband and the stopband. In general, elliptic filters meet given performance specifications with the lowest order of any filter type.

`ellip` uses a five-step algorithm:

- 1 It finds the lowpass analog prototype poles, zeros, and gain using the function `ellipap`.
- 2 It converts the poles, zeros, and gain into state-space form.
- 3 If required, it uses a state-space transformation to convert the lowpass filter to a bandpass, highpass, or bandstop filter with the desired frequency constraints.
- 4 For digital filter design, it uses `bilinear` to convert the analog filter into a digital filter through a bilinear transformation with frequency prewarping. Careful

frequency adjustment enables the analog filters and the digital filters to have the same frequency response magnitude at  $\omega_p$  or  $\omega_1$  and  $\omega_2$ .

- 5 It converts the state-space filter back to transfer function or zero-pole-gain form, as required.

### **See Also**

besself | butter | cheby1 | cheby2 | designfilt | ellipap | ellipord | filter | sosfilt

## ellipap

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  $R_p$  dB of ripple in the passband, and a stopband  $R_s$  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 in factored zero-pole form is

$$H(s) = \frac{z(s)}{p(s)} = k \frac{(s - z_1)(s - z_2)\dots(s - z_N)}{(s - p_1)(s - p_2)\dots(s - p_M)}$$

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.

`ellipap` sets the passband edge angular frequency  $\omega_0$  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  $10^{-R_p/20}$ .

## More About

### Algorithms

`ellipap` uses the algorithm outlined in [1]. It employs `ellipke` to calculate the complete elliptic integral of the first kind and `ellipj` to calculate Jacobi elliptic functions.



## References

- [1] Parks, T. W., and C. S. Burrus. *Digital Filter Design*. New York: John Wiley & Sons, 1987, chap. 7.

## See Also

besselap | buttap | cheb1ap | cheb2ap | ellip

# ellipord

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  |
|-----------------|--|
| <code>Wp</code> | Passband corner frequency <code>Wp</code> , 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, $\pi$ radians per sample. |
| <code>Ws</code> | Stopband corner frequency <code>Ws</code> , is a scalar or a two-element vector with values between 0 and 1, with 1 corresponding to the normalized Nyquist frequency.   |
| <code>Rp</code> | Passband ripple, in decibels. This value is the maximum permissible passband loss in decibels.   |
| <code>Rs</code> | 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     | $W_p < W_s$ , both scalars   | $(W_s, 1)$                            | $(0, W_p)$         |
| Highpass    | $W_p > W_s$ , both scalars   | $(0, W_s)$                            | $(W_p, 1)$         |
| Bandpass    | The interval specified by $W_s$ contains the one specified by $W_p$ ( $W_s(1) < W_p(1) < W_p(2) < W_s(2)$ ). | $(0, W_s(1))$<br>and<br>$(W_s(2), 1)$ | $(W_p(1), W_p(2))$ |
| Bandstop    | The interval specified by $W_p$ contains the one specified by $W_s$ ( $W_p(1) < W_s(1) < W_s(2) < W_p(2)$ ). | $(0, W_p(1))$<br>and<br>$(W_p(2), 1)$ | $(W_s(1), W_s(2))$ |

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  $W_p$  for an analog filter. You specify the frequencies  $W_p$  and  $W_s$  similar to those described in the Description of Stopband and Passband Filter Parameters 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 table above.

## Examples

### Lowpass Elliptic Filter Order

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. Find the filter order and cutoff frequency.

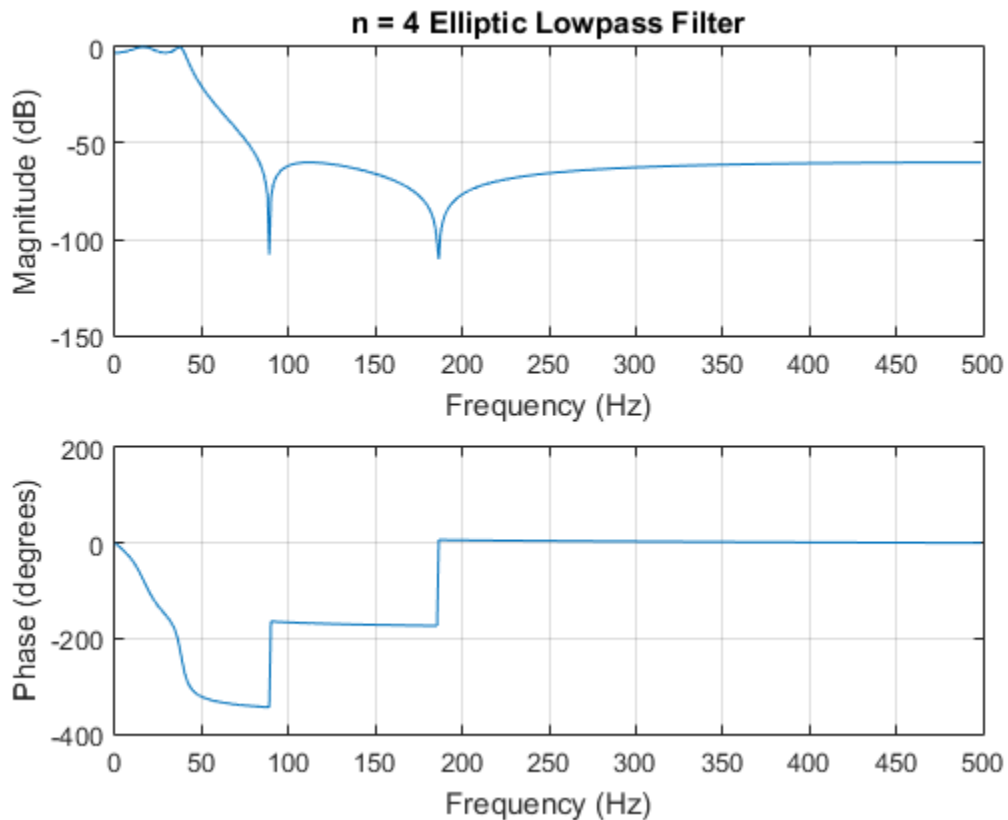
```
Wp = 40/500;  
Ws = 150/500;  
Rp = 3;  
Rs = 60;  
[n,Wp] = ellipord(Wp,Ws,Rp,Rs)
```

```
n =  
  
    4
```

```
Wp =  
  
    0.0800
```

Specify the filter in terms of second-order sections and plot the frequency response.

```
[z,p,k] = ellip(n,Rp,Rs,Wp);  
sos = zp2sos(z,p,k);  
freqz(sos,512,1000)  
title(sprintf('n = %d Elliptic Lowpass Filter',n))
```



### Bandpass Elliptic Filter Order

Design a bandpass filter with a passband from 60 Hz to 200 Hz with at most 3 dB of ripple and at least 40 dB attenuation in the stopbands. Specify a sampling rate of 1 kHz. Have the stopbands be 50 Hz wide on both sides of the passband. Find the filter order and cutoff frequencies.

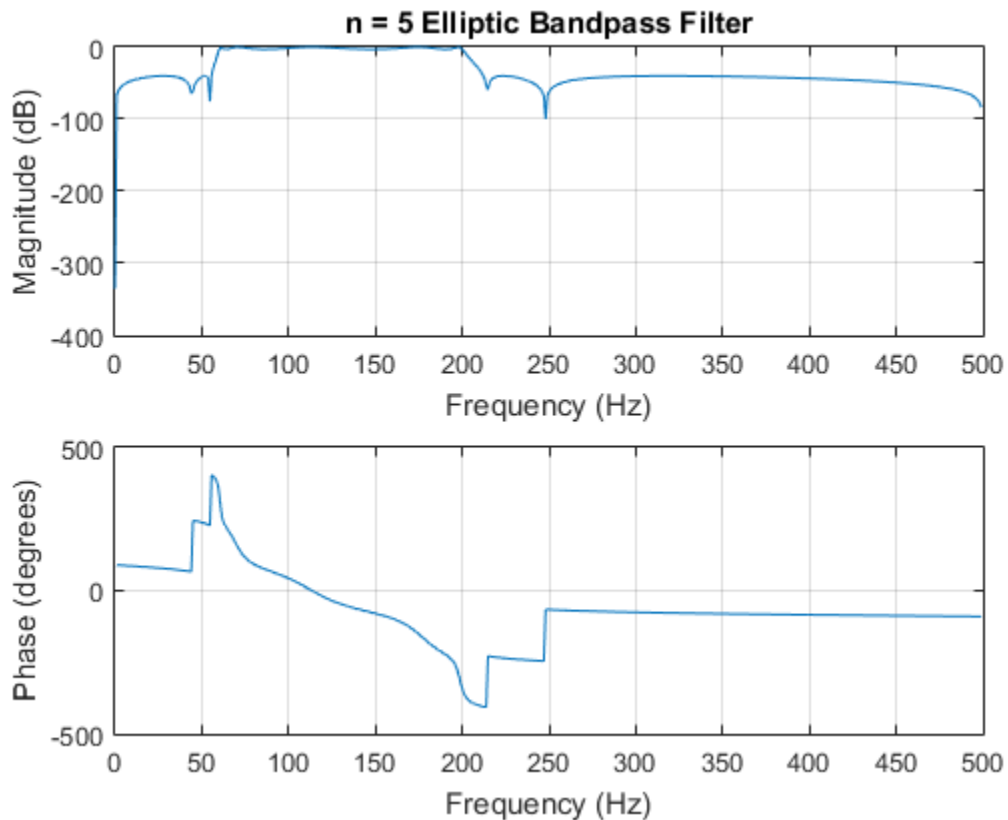
```
Wp = [60 200]/500;
Ws = [50 250]/500;
Rp = 3;
Rs = 40;
```

```
[n,Wp] = ellipord(Wp,Ws,Rp,Rs)
```

```
n =  
    5  
  
Wp =  
    0.1200    0.4000
```

Specify the filter in terms of second-order sections and plot the frequency response.

```
[z,p,k] = ellip(n,Rp,Rs,Wp);  
sos = zp2sos(z,p,k);  
  
freqz(sos,512,1000)  
title(sprintf('n = %d Elliptic Bandpass Filter',n))
```



## More About

### Algorithms

`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, Lawrence R., and B. Gold. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975.

## See Also

`buttord` | `cheb1ord` | `cheb2ord` | `ellip`



## enbw

Equivalent noise bandwidth

### Syntax

```
bw = enbw(window)
bw = enbw(window, fs)
```

### Description

`bw = enbw(window)` returns the two-sided equivalent noise bandwidth, `bw`, for a uniformly sampled window, `window`. The equivalent noise bandwidth is normalized by the noise power per frequency bin.

`bw = enbw(window, fs)` returns the two-sided equivalent noise bandwidth, `bw`, in Hz.

### Examples

#### Equivalent Noise Bandwidth of Hamming Window

Determine the equivalent noise bandwidth of a Hamming window 1,000 samples in length.

```
bw = enbw(hamming(1000));
```

#### Equivalent Noise Bandwidth of Flat Top Window

Determine the equivalent noise bandwidth in Hz of a flat top window 10,000 samples in length. The sampling frequency is 44.1 kHz.

```
bw = enbw(flattopwin(10000), 44.1e3);
```

#### Equivalent Rectangular Noise Bandwidth

Obtain the equivalent rectangular noise bandwidth of a Von Hann window and overlay the equivalent rectangular bandwidth on the window's magnitude spectrum. The window is 1000 samples in length and the sampling frequency is 10 kHz.

Set the sampling frequency, create the window, and obtain the discrete Fourier transform of the window with 0 frequency in the center of the spectrum.

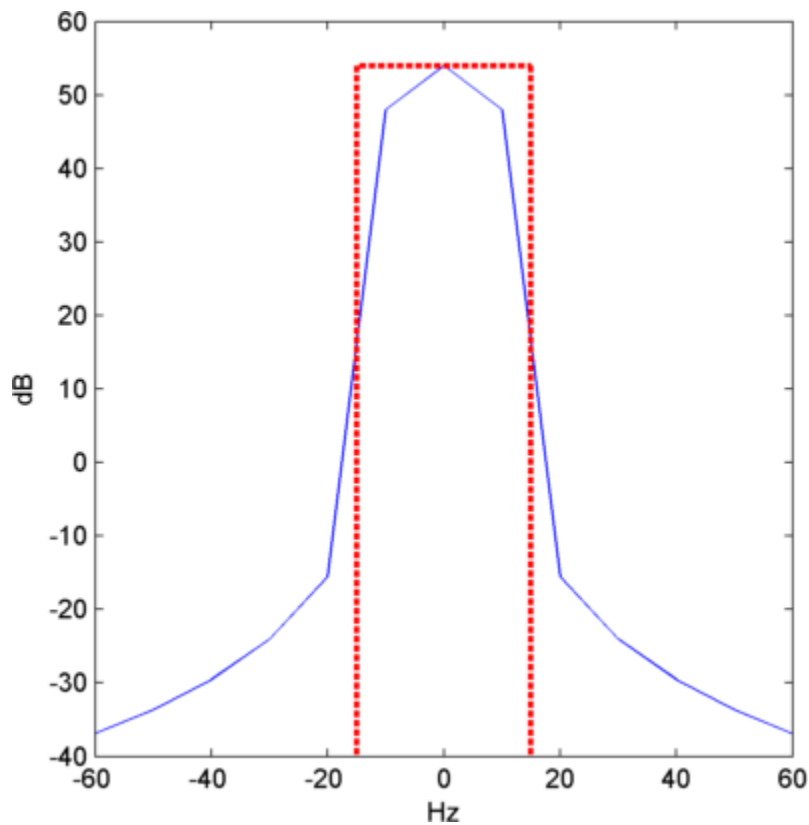
```
Fs = 10000;  
win = hann(1000);  
windft = fftshift(fft(win));
```

Obtain the equivalent (rectangular) noise bandwidth of the Von Hann window.

```
bw = enbw(hann(1000),Fs);
```

Plot the squared-magnitude DFT of the window and use the equivalent noise bandwidth to overlay the equivalent rectangle.

```
freq = -(Fs/2):Fs/length(win):Fs/2-(Fs/length(win));  
plot(freq,20*log10(abs(windft))); xlabel('Hz'); ylabel('dB');  
axis([-60 60 -40 60])  
maxgain = 20*log10(abs(windft(length(win)/2+1)));  
hold on;  
plot([-bw -bw],[-40 maxgain],'r--',...  
      [bw bw],[-40 maxgain],'r--','linewidth',2);  
plot([-bw bw],[maxgain maxgain],'r--','linewidth',2);
```



## Input Arguments

### **window** — Window vector

real-valued row or column vector

Uniformly sampled window vector, specified as a row or column vector with real-valued elements.

Example: `hamming(1000)`

Data Types: `double` | `single`

### **fs** — Sampling frequency

positive scalar

Sampling frequency, specified as a positive scalar.

## Output Arguments

### **bw** — Equivalent noise bandwidth

positive scalar

Equivalent noise bandwidth, specified as a positive scalar.

Data Types: `double` | `single`

## More About

### Equivalent Noise Bandwidth

The equivalent noise bandwidth of a window is the width of a rectangle whose area contains the same total power as the window. The height of the rectangle is the peak squared magnitude of the window's Fourier transform.

Assuming a sampling interval of 1, the total energy for the window,  $w(n)$ , can be expressed in the frequency or time-domain as

$$\int_{-1/2}^{1/2} |W(f)|^2 df = \sum_n |w(n)|^2$$

The peak magnitude of the window's spectrum occurs at  $f=0$ . This is given by

$$|W(0)|^2 = \left| \sum_n w(n) \right|^2$$

To find the width of the equivalent rectangular bandwidth, divide the area by the height.

$$\frac{\int_{-1/2}^{1/2} |W(f)|^2 df}{|W(0)|^2} = \frac{\sum_n |w(n)|^2}{\left| \sum_n w(n) \right|^2}$$

See “Equivalent Rectangular Noise Bandwidth” on page 1-411 for an example that plots the equivalent rectangular bandwidth over the magnitude spectrum of a Von Hann window.

**See Also**

bandpower | sfd

## equiripple

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

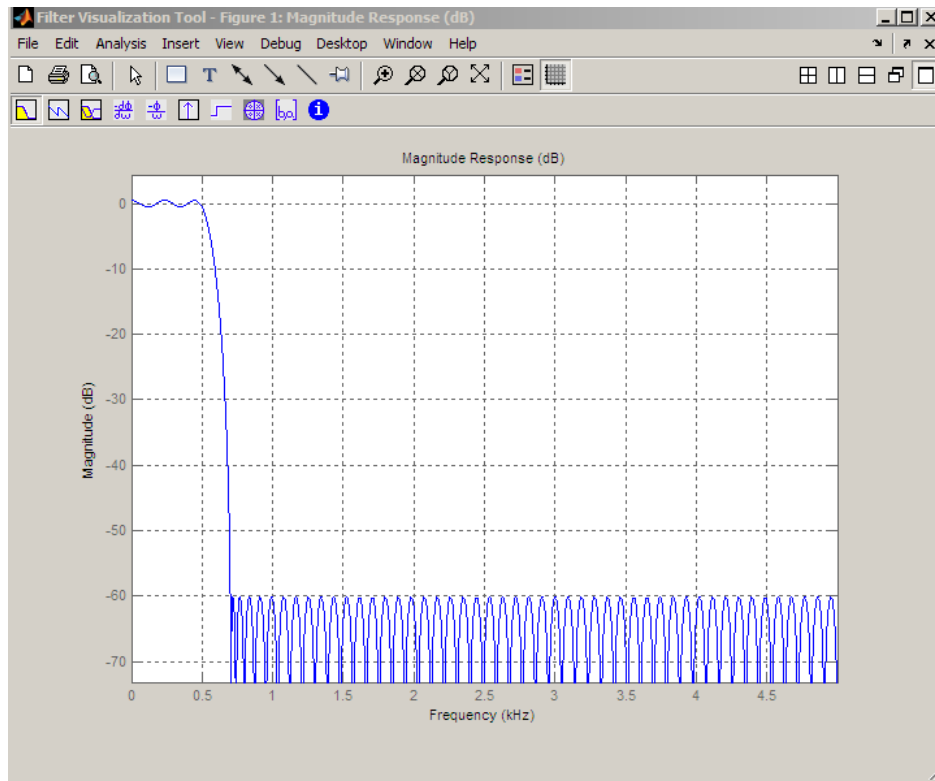
First create a lowpass equiripple filter. Assume the data is sampled at 10,000 Hertz. The passband frequency is 500 Hertz with a stopband frequency of 700 Hz. The desired passband ripple is 1 dB with 60 dB of stopband attenuation.

```

Fs=10000;
Hd=fdesign.lowpass('Fp,Fst,Ap,Ast',500,700,1,60,10000);
d=design(Hd,'equiripple');
fvtool(d);

```

Displaying the filter in FVTool shows the equiripple nature of the filter.



The next example designs a lowpass equiripple filter with a direct-form transposed structure and density factor of 20 by specifying the `FilterStructure` and `DensityFactor` properties.

To set the design options for the filter, use the `designopts` method to obtain a structure array containing the current design options.

Change the fields of the structure array to specify your design options and invoke the `design` method with the structure array as an input argument.

```
% Use the same filter design as the previous example
Fs = 10000;
Hd = fdesign.lowpass('Fp,Fst,Ap,Ast',500,700,1,60,10000);
% Return the design options for the filter as a struct array
opts = designopts(Hd,'equiripple');
fieldnames(opts)
% Print out the filter structure- - direct-form FIR
opts.FilterStructure
% Change the filter structure to direct-form FIR transposed
opts.FilterStructure = 'dffirt';
% Change the filter density factor to 20
opts.DensityFactor = 20;
% Design the filter
d = design(Hd,'equiripple',opts)
```

An alternate way to design the preceding filter without using the structure array is:

```
Fs = 10000;
Hd = fdesign.lowpass('Fp,Fst,Ap,Ast',500,700,1,60,10000);
d = design(Hd,'equiripple','FilterStructure','dffirt','DensityFactor',20);
```

## See Also

[design](#) | [designmethods](#)



## eqtflength

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);
```

### More About

#### Algorithms

`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

# falltime

Fall time of negative-going bilevel waveform transitions

## Syntax

```
F = falltime(X)
F = falltime(X,FS)
F = falltime(X,T)
[F,LT,UT] = falltime(...)
[F,LT,UT,LL,UL] = falltime(...)
[...] = falltime(...,Name,Value)
falltime(...)
```

## Description

`F = falltime(X)` returns a vector, `F`, containing the time each transition of the bilevel waveform, `X`, takes to cross from the 90% to 10% reference levels. See “Percent Reference Levels” on page 1-426. To determine the transitions, `falltime` estimates the state levels of the input waveform by a histogram method. `falltime` identifies all regions, which cross the lower-state boundary of the high state and the upper-state boundary of the low state. The low-state and high-state boundaries are expressed as the state level plus or minus a multiple of the difference between the state levels. See “State-Level Tolerances” on page 1-426. Because `falltime` uses interpolation, `F` may contain values that do not correspond to sampling instants of the bilevel waveform, `X`.

`F = falltime(X,FS)` specifies the sampling frequency in hertz. The sampling frequency determines the sample instants corresponding to the elements in `X`. The first sample instant in `X` corresponds to  $t=0$ . Because `falltime` uses interpolation, `F` may contain values that do not correspond to sampling instants of the bilevel waveform, `X`.

`F = falltime(X,T)` specifies the sample instants, `T`, as a vector with the same number of elements as `X`.

`[F,LT,UT] = falltime(...)` returns vectors, `LT` and `UT`, whose elements correspond to the time instants where `X` crosses the lower and upper percent reference levels.

`[F,LT,UT,LL,UL] = falltime(...)` returns the levels, LL and UL, corresponding to the lower- and upper-percent reference levels.

`[...] = falltime(...,Name,Value)` returns the fall times with additional options specified by one or more Name,Value pair arguments.

`falltime(...)` plots the signal and darkens the regions of each transition where fall time is computed. The plot marks the lower and upper crossings and the associated reference levels. The state levels and the associated lower- and upper-state boundaries are also displayed.

## Input Arguments

### **X**

Bilevel waveform. X is a real-valued row or column vector.

### **FS**

Sample rate in hertz.

### **T**

Vector of sample instants. The length of T must equal the length of the bilevel waveform, X.

## Name-Value Pair Arguments

### **'PercentReferenceLevels'**

Reference levels as a percentage of the waveform amplitude. The low-state level is defined to be 0 percent. The high-state level is defined to be 100 percent. See “Percent Reference Levels” on page 1-426. 'PercentReferenceLevels' is a 2-element real row vector whose elements correspond to the lower- and upper-percent reference levels.

**Default:** [10 90]

### **'StateLevels'**

Low and high-state levels. Specifies the levels to use for the low- and high-state levels as a 2-element real-valued row vector whose first and second elements correspond to the low- and high-state levels.

**'Tolerance'**

Tolerance levels (lower- and upper-state boundaries) expressed as a percentage. See “State-Level Tolerances” on page 1-426.

**Default:** 2

## Output Arguments

**F**

Fall times. F is a vector containing the duration of each negative-going transition. If you specify the sampling rate, FS, or the sampling instants, T, fall times are in seconds. If you do not specify a sampling rate, or sampling instants, fall times are in samples.

**LT**

Instants when negative-going transition crosses the lower-reference level. By default, the lower-reference level is the 10% reference level. You can change the default reference levels by specifying the 'PercentReferenceLevels' name-value pair.

**UT**

Instants when negative-going transition crosses the upper-reference level. By default, the upper reference level is the 90% reference level. You can change the default reference levels by specifying the 'PercentReferenceLevels' name-value pair.

**LL**

Lower-reference level in waveform amplitude units. LL is a vector containing the waveform values corresponding to the lower-reference level in each negative-going transition. By default, the lower-reference level is the 10% reference level. You can change the default reference levels by specifying the 'PercentReferenceLevels' name-value pair.

**UL**

Upper-reference level in waveform amplitude units. LL is a vector containing the waveform values corresponding to the upper-reference level in each negative-going

transition. By default, the upper-reference level is the 90% reference level. You can change the default reference levels by specifying the 'PercentReferenceLevels' name-value pair.

## Examples

### Falltime in a Bilevel Waveform

Determine the fall time in samples for a 2.3 V clock waveform.

Load the 2.3 V clock data. Determine the fall time in samples. Use the default [10 90] percent reference levels.

```
load('negtransitionex.mat', 'x');  
F = falltime(x);
```

The fall time is less than 1, indicating that the transition occurred in a fraction of a sample.

### Falltime with 20% and 80% Reference Levels

Determine the fall time in a 2.3 V clock waveform sampled at 4 MHz. Compute the fall time using the 20% and 80% reference levels.

Load the 2.3 V clock data with sampling instants. Plot the waveform.

```
load('negtransitionex.mat', 'x', 't');  
plot(t,x);
```

Determine the fall time using the 20% and 80% reference levels..

```
F = falltime(x, 'PercentReferenceLevels', [20 80]);
```

### Falltime, Reference-Level Instants, and Reference Levels

Determine the fall time, reference-level instants, and reference levels in a 2.3 V clock waveform sampled at 4 MHz.

Load the 2.3 V clock waveform along with the sampling instants.

```
load('negtransitionex.mat', 'x', 't');
```

Determine the falltime, reference-level instants, and reference levels.

```
[F,LT,UT,LL,UL] = falltime(x,t);
```

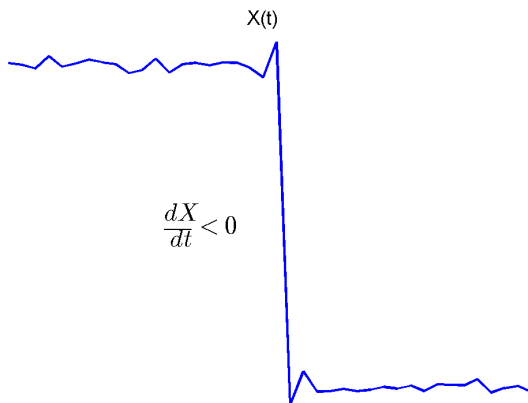
Plot the waveform in microseconds with the upper and lower reference levels and reference level instants. Show that the fall time is the difference between the lower- and upper-reference level instants.

```
plot(t.*1e6,x);
xlabel('microseconds'); ylabel('Volts');
hold on; grid on;
plot(LT.*1e6,LL,'ro','markerfacecolor',[1 0 0]);
plot(UT.*1e6,UL,'ro','markerfacecolor',[1 0 0]);
fprintf('Rise time is %1.4f microseconds.\n',(LT-UT)*1e6)
```

## More About

### Negative-Going Transition

A negative-going transition in a bilevel waveform is a transition from the high-state level to the low-state level. If the waveform is differentiable in the neighborhood of the transition, an equivalent definition is a transition with a negative first derivative. The following figure shows a negative-going transition.



In the preceding figure, the amplitude values of the waveform are not displayed because a negative-going transition does not depend on the actual waveform values. A negative-going transition is defined by the direction of the transition.

### Percent Reference Levels

If  $S_1$  is the low state,  $S_2$  is the high state, and  $U$  is the *upper*-percent reference level. The waveform value corresponding to the upper percent reference level is

$$S_1 + \frac{U}{100}(S_2 - S_1)$$

If  $L$  is the *lower* percent reference level, the waveform value corresponding to the lower percent reference level is

$$S_1 + \frac{L}{100}(S_2 - S_1)$$

### State-Level Tolerances

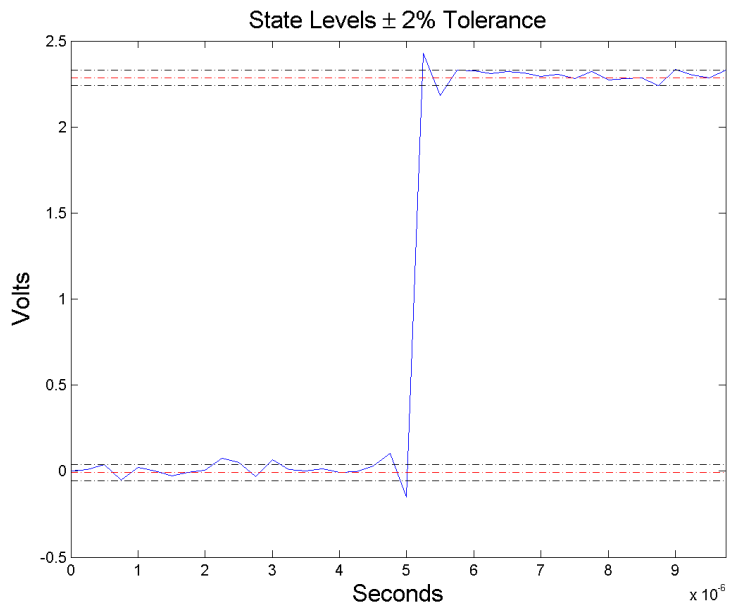
Each state level can have associated lower- and upper-state boundaries. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference between the high state and low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the  $\alpha\%$  tolerance region for the low state is defined as

$$S_1 \pm \frac{\alpha}{100}(S_2 - S_1)$$

where  $S_1$  is the low-state level and  $S_2$  is the high-state level. Replace the first term in the equation with  $S_2$  to obtain the  $\alpha\%$  tolerance region for the high state.

The following figure illustrates lower and upper 2% state boundaries (tolerance regions) for a positive-polarity (positive-going) bilevel waveform. The estimated state levels are indicated by a dashed red line.





## References

- [1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003, pp. 15–17.

## See Also

`risetime` | `slewrates` | `statelevels`

# **fdatool**

Open Filter Design and Analysis Tool

## **Syntax**

`fdatool`

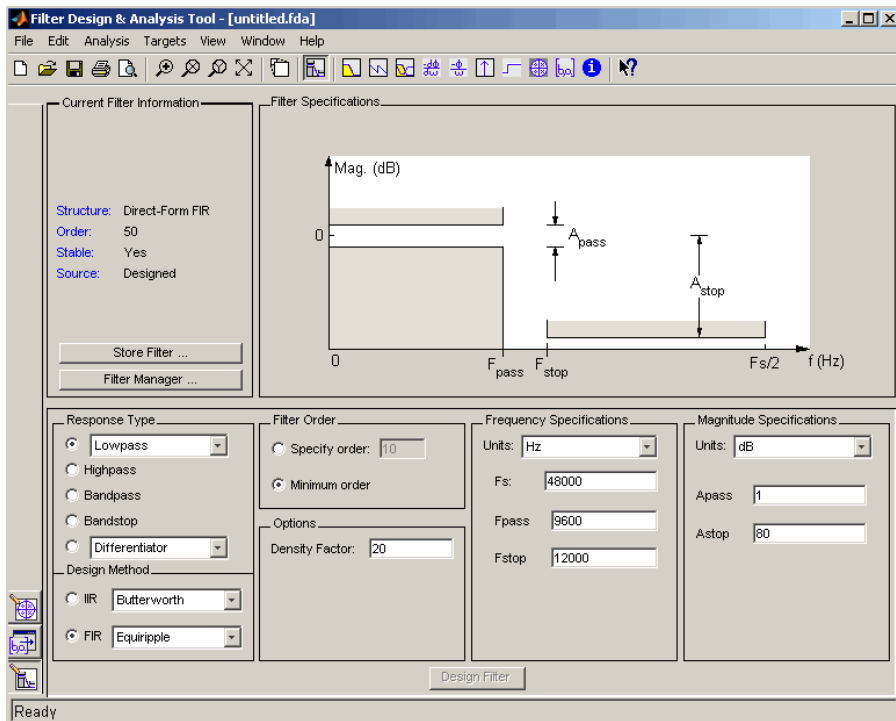
## **Description**

`fdatool` opens the Filter Design and Analysis Tool (FDATool). Use this tool to

- Design filters
- Analyze filters
- Modify existing filter designs

See “Using FDATool” for detailed information about the Filter Design and Analysis Tool.

If the DSP System Toolbox product is installed, FDATool seamlessly integrates advanced filter design methods and the ability to quantize filters. For more information, see the DSP System Toolbox `FDATool` documentation.



## More About

### Tips

The Filter Design and Analysis Tool provides more design methods than the SPTool Filter Designer, which will be removed in a future release. The Filter Design and Analysis Tool also integrates advanced filter design methods from the DSP System Toolbox software.

---

**Note** The Filter Design and Analysis Tool requires a screen resolution greater than 640 x 480.

---

### See Also

fvtool | sptool | wvtool

# fdesign

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 “Using Filter Design Methods with Specification Objects” on page 1-435

---

**Note:** Several of the filter response types described below are only available if your installation includes the DSP System Toolbox. The DSP System 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')`.

*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.

| <b>fdesign Response String</b> | <b>Description</b>  |
|--------------------------------|---|
| <code>arbgrpdelay</code>       | <code>fdesign.arbgrpdelay</code> creates an object to specify allpass arbitrary group delay filters. Requires the DSP System Toolbox  |
| <code>arbmag</code>            | <code>fdesign.arbmag</code> creates an object to specify IIR filters that have arbitrary magnitude responses defined by the input arguments.  |
| <code>arbmagnphase</code>      | <code>fdesign.arbmagnphase</code> creates an object to specify IIR filters that have arbitrary magnitude and phase responses defined by the input arguments. Requires the DSP System Toolbox.   |
| <code>audioweighting</code>    | <code>fdesign.audioweighting</code> creates a filter specification object for audio weighting filters. The supported audio weighting types are: A, C, C-message, ITU-T 0.41, and ITU-R 468-4 weighting. Requires the DSP System Toolbox |
| <code>bandpass</code>          | <code>fdesign.bandpass</code> creates an object to specify bandpass filters.  |
| <code>bandstop</code>          | <code>fdesign.bandstop</code> creates an object to specify bandstop filters.  |
| <code>ciccomp</code>           | <code>fdesign.ciccomp</code> creates an object to specify filters that compensate for the CIC decimator or interpolator response curves. Requires the DSP System Toolbox.   |
| <code>comb</code>              | <code>fdesign.comb</code> creates an object to specify a notching or peaking comb filter. Requires the DSP System Toolbox.  |
| <code>decimator</code>         | <code>fdesign.decimator</code> creates an object to specify decimators. Requires the DSP System Toolbox   |
| <code>differentiator</code>    | <code>fdesign.differentiator</code> creates an object to specify an FIR differentiator filter.  |

| <b>fdesign Response String</b> | <b>Description</b>   |
|--------------------------------|--|
| fracdelay                      | <code>fdesign.fracdelay</code> creates an object to specify fractional delay filters. Requires the DSP System Toolbox.               |
| halfband                       | <code>fdesign.halfband</code> creates an object to specify halfband filters. Requires the DSP System Toolbox.                        |
| highpass                       | <code>fdesign.highpass</code> creates an object to specify highpass filters.   |
| hilbert                        | <code>fdesign.hilbert</code> creates an object to specify an FIR Hilbert transformer.  |
| interpolator                   | <code>fdesign.interpolator</code> creates an object to specify interpolators. Requires the DSP System Toolbox.                       |
| isinchp                        | <code>fdesign.isinchp</code> creates an object to specify an inverse sinc highpass filter. Requires the DSP System Toolbox.          |
| isinclp                        | <code>fdesign.isinclp</code> creates an object to specify an inverse sinc lowpass filters. Requires the DSP System Toolbox.          |
| lowpass                        | <code>fdesign.lowpass</code> creates an object to specify lowpass filters.   |
| notch                          | <code>fdesign.notch</code> creates an object to specify notch filters. Requires the DSP System Toolbox.                              |
| nyquist                        | <code>fdesign.nyquist</code> creates an object to specify nyquist filters. Requires the DSP System Toolbox.                          |
| octave                         | <code>fdesign.octave</code> creates an object to specify octave and fractional octave filters. Requires the DSP System Toolbox.      |
| parameq                        | <code>fdesign.parameq</code> creates an object to specify parametric equalizer filters. Requires the DSP System Toolbox.             |
| peak                           | <code>fdesign.peak</code> creates an object to specify peak filters. Requires the DSP System Toolbox.                                |
| polysrc                        | <code>fdesign.polysrc</code> creates an object to specify polynomial sample-rate converter filters. Requires the DSP System Toolbox. |
| rsrc                           | <code>fdesign.rsrc</code> creates an object to specify rational-factor sample-rate convertors. Requires the DSP System Toolbox.      |

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.   |
| Specification       | Depends on the chosen type            | Defines the filter characteristics used to define the desired filter performance, such as the cutoff frequency <code>Fc</code> or the filter order <code>N</code> .                      |
| 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  |

| Property Name | Default Value | Description  |
|---------------|---------------|--|
|               |               | frequency band from 0 to $F_s/2$ , the sampling frequency. Accepts either <code>true</code> or <code>false</code> without single quotation marks. Audio weighting filters do not support normalized frequency. |

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 <code>mfilt</code> Objects | Description   |
|---|---|
| <code>DecimationFactor</code>                   | Specifies the amount to decrease the sampling rate. Always a positive integer.  |
| <code>InterpolationFactor</code>                | Specifies the amount to increase the sampling rate. Always a positive integer.  |
| <code>PolyphaseLength</code>                    | 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 <code>p1</code> and the rate change factors. <code>p1</code> 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. The specifications are applied to the filter design method you choose to design your filter.

For example, when you create a default lowpass filter specification object, `fdesign.lowpass` sets the passband frequency `Fp`, the stopband frequency `Fst`, the stopband attenuation `Ast`, and the passband ripple `Ap` :

```
H = fdesign.lowpass
% Use without a terminating semicolon
% to display the filter specifications
```

The default specification '`Fp,Fst,Ap,Ast`' is only one of the possible specifications for `fdesign.lowpass`. To see all available specifications:

```
H = fdesign.lowpass;
set(H,'specification')
```



The DSP System Toolbox software supports all available specification strings. The Signal Processing Toolbox supports a subset of the specification strings. See the reference pages for the filter specification object to determine which specification strings your installation supports.

One important note is that the specification string you choose determines which design methods apply to the filter specifications object.

Specifications that do not contain the filter order result in minimum order designs when you invoke the `design` method:

```
d = fdesign.lowpass;  
% Specification is Fp,Fst,Ap,Ast  
Hd = design(d,'equiripple');  
length(Hd.Numerator) % Returns 43  
% Filter order is 42  
fvtool(Hd) % View magnitude
```

`d = fdesign.response(...,Fs)` specifies the sampling frequency in Hz to use in the filter specifications. The sampling frequency is a scalar trailing all other input arguments. If you specify a sampling frequency, all frequency specifications are in Hz.

`d = fdesign.response(...,magunits)` specifies the units for any magnitude specification you provide in the input arguments. `magunits` can be one of the following strings:

- '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. Use `designmethods` to determine valid design methods for your filter specification object.

```
d = fdesign.lowpass('N,Fc,Ap,Ast',10,0.2,0.5,40);
```

```
designmethods(d)
% Design FIR equiripple filter
hd = design(d, 'equiripple');
```

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  |
|----------------------------|--|
| <code>setspecs</code>      | Set all of the specifications simultaneously.  |
| <code>designmethods</code> | Return the design methods.   |
| <code>designopts</code>    | 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.

Filter object constructors take the input arguments in the same order as `setspecs` and the order in the strings for `Specification`. Enter `doc setspecs` at the prompt for more information about using `setspecs`.

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 kHz. The stopband between 2 and 2.4 kHz is attenuated at least 80 dB:

```
H = 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 kHz. 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.

```
H = fdesign.lowpass('Fp,Fst,Ap,Ast',500,750,1,80,10000);
```

## Example 3—Highpass Filter

A default highpass filter specification object.

```
H = fdesign.highpass % Creates specifications object.
H.Description
```

Notice the correspondence between the property values in `Specification` and `Description` — in `Description` you see in words the definitions of the variables shown in `Specification`.

## Example 4—Lowpass Butterworth Filter Specification and Design

Use a filter specification object to construct a lowpass Butterworth filter with the default `Specification`, `'Fp,Fst,Ap,Ast'`. Set the passband edge frequency to  $0.4\pi$  rad/sample, the stopband frequency to  $0.5\pi$  rad/sample, the passband ripple to 1 dB, and the stopband attenuation to 80 dB.

```
d = fdesign.lowpass(0.4,0.5,1,80);
```

Determine which design methods apply to `d`.

```
designmethods(d)
```

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

```
butter
```

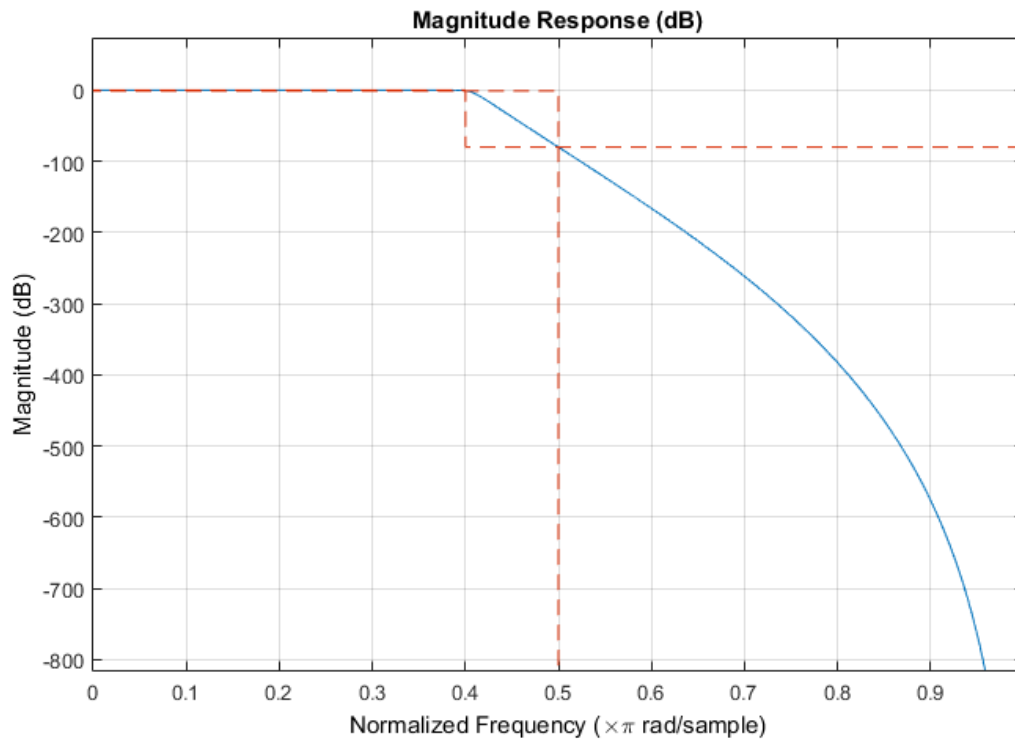
```
cheby1  
cheby2  
ellip  
equiripple  
ifir  
kaiserwin  
multistage
```

You can use `d` and the `butter` design method to design a Butterworth filter.

```
hd = design(d, 'butter', 'matchexactly', 'passband');
```

The resulting filter magnitude response shown by FVTool appears in the following figure.

```
fvtool(hd);
```



If you have the DSP System Toolbox™ software installed, the preceding figure appears with the filter specification mask.

**See Also**

`designmethods` | `designopts` | `fdatool` | `filterbuilder` | `fvtool`

## fdesign.arbmag

Arbitrary response magnitude filter specification object

### Syntax

```
D= fdesign.arbmag
D= fdesign.arbmag(SPEC)
D = fdesign.arbmag(SPEC,specvalue1,specvalue2,...)
D = fdesign.arbmag(specvalue1,specvalue2,specvalue3)
D = fdesign.arbmag(...,Fs)
```

### Description

D= fdesign.arbmag constructs an arbitrary magnitude filter specification object D.

D= fdesign.arbmag(SPEC) initializes the **Specification** property to SPEC. The input argument SPEC must be one of the strings shown in the following table. Specification strings are not case sensitive.

---

**Note:** Specifications strings marked with an asterisk require the DSP System Toolbox software.

---

- 'N,F,A' — Single band design (default)
- 'F,A,R' — Single band minimum order design \*
- 'N,B,F,A' — Multiband design
- 'N,B,F,A,C' — Constrained multiband design \*
- 'B,F,A,R' — Multiband minimum order design \*
- 'Nb,Na,F,A' — Single band design \*
- 'Nb,Na,B,F,A' — Multiband design \*

The string entries are defined as follows:

- **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 value designs, use `fdesign.arbmagnphase`.
- **B** — Number of bands in the multiband filter
- **C** — Constrained band flag. This enables you to constrain the passband ripple in your multiband design. You cannot constrain the passband ripple in all bands simultaneously.
- **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
- **R** — Ripple

By default, this method assumes that all frequency specifications are supplied in normalized frequency.

## Specifying Frequency and Amplitude Vectors

**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 table shows how **F** and **A** are related.

Define the frequency vector **F** as [0 0.25 0.3 0.4 0.5 0.6 0.7 0.75 1.0]

Define the response vector **A** as [1 1 0 0 0 0 0 1 1]

These specifications connect **F** and **A** as shown here:

| <b>F (Normalized Frequency)</b> | <b>A (Response Desired at F)</b> |
|---------------------------------|----------------------------------|
| 0                               | 1                                |
| 0.25                            | 1                                |
| 0.3                             | 0                                |

| F (Normalized Frequency) | A (Response Desired at F) |
|--------------------------|---------------------------|
| 0.4                      | 0                         |
| 0.5                      | 0                         |
| 0.6                      | 0                         |
| 0.7                      | 0                         |
| 0.75                     | 1                         |
| 1.0                      | 1                         |

Different specifications can have different design methods available. Use `designmethods` to get a list of design methods available for a given specification string and filter specification object.

Use `designopts` to get a list of design options available for a filter specification object and a given design method. Enter `help(D,METHOD)` to get detailed help on the available design options for a given design method.

`D = fdesign.arbmag(SPEC,specvalue1,specvalue2,...)` initializes the specifications with `specvalue1`, `specvalue2`. Use `get(D,'Description')` for descriptions of the various specifications `specvalue1`, `specvalue2`, ... `specvalueN`.

`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

### Design of a multiband arbitrary-magnitude filter

Use `fdesign.arbmag` to design a 3–band filter.

Use the given frequency and amplitude vectors in “Specifying Frequency and Amplitude Vectors” on page 1-441.

```
N = 150;
```

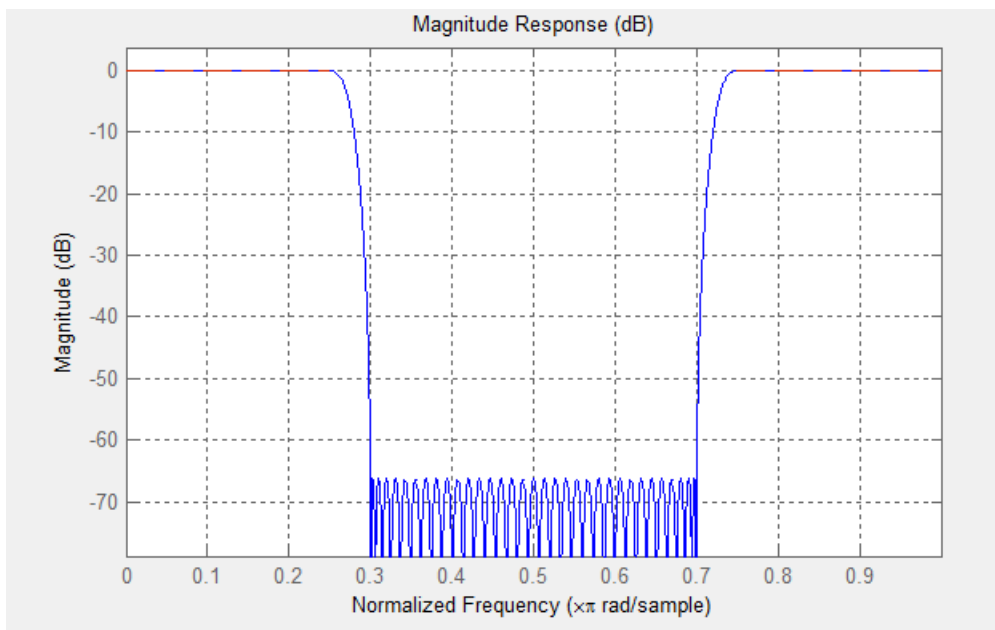


```

B = 3;
F = [0 .25 .3 .4 .5 .6 .7 .75 1];
A = [1 1 0 0 0 0 0 1 1];
A1 = A(1:2);
A2 = A(3:7);
A3 = A(8:end);
F1 = F(1:2);
F2 = F(3:7);
F3 = F(8:end);
d = fdesign.arbmag('N,B,F,A',N,B,F1,A1,F2,A2,F3,A3);
Hd = design(d);
fvtool(Hd)

```

A response with two passbands — one roughly between 0 and 0.25 and the second between 0.75 and 1 — results from the mapping between F and A.



### Design of a single band arbitrary-magnitude filter

Use `fdesign.arbmag` to design a single band equiripple filter.

```
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;
d = fdesign.arbmag('N,F,A',n,f,a);
hd1 = design(d,'equiripple');
```

If you have the DSP System Toolbox, you can design a minimum-phase equiripple filter.

```
hd2 = design(d,'equiripple','MinPhase',true);
hfvt = fvtool([hd1 hd2],'analysis','polezero');
legend(hfvt,'Equiripple Filter','Minimum-phase Equiripple Filter');
```

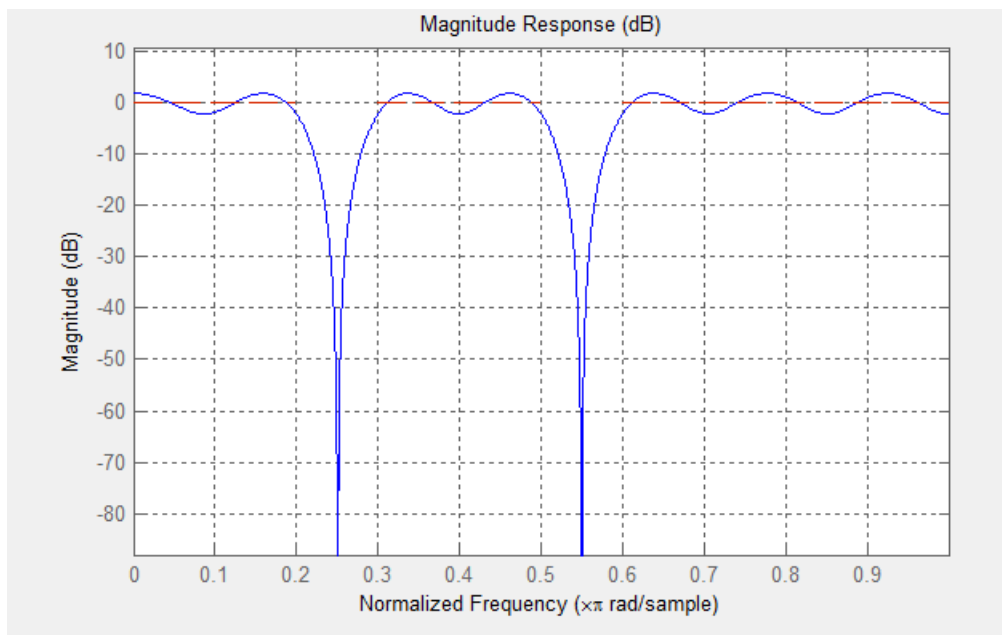
### **Design of a multiband minimum order arbitrary-magnitude filter**

Use `fdesign.arbmag` to design a multiband minimum order filter.

This example requires the DSP System Toolbox.

Place the notches at  $0.25\pi$  and  $0.55\pi$  radians/sample

```
d = fdesign.arbmag('B,F,A,R');
d.NBands = 5;
d.B1Frequencies = [0 0.2];
d.B1Amplitudes = [1 1];
d.B1Ripple = 0.25;
d.B2Frequencies = 0.25;
d.B2Amplitudes = 0;
d.B3Frequencies = [0.3 0.5];
d.B3Amplitudes = [1 1];
d.B3Ripple = 0.25;
d.B4Frequencies = 0.55;
d.B4Amplitudes = 0;
d.B5Frequencies = [0.6 1];
d.B5Amplitudes = [1 1];
d.B5Ripple = 0.25;
Hd = design(d,'equiripple');
fvtool(Hd)
```



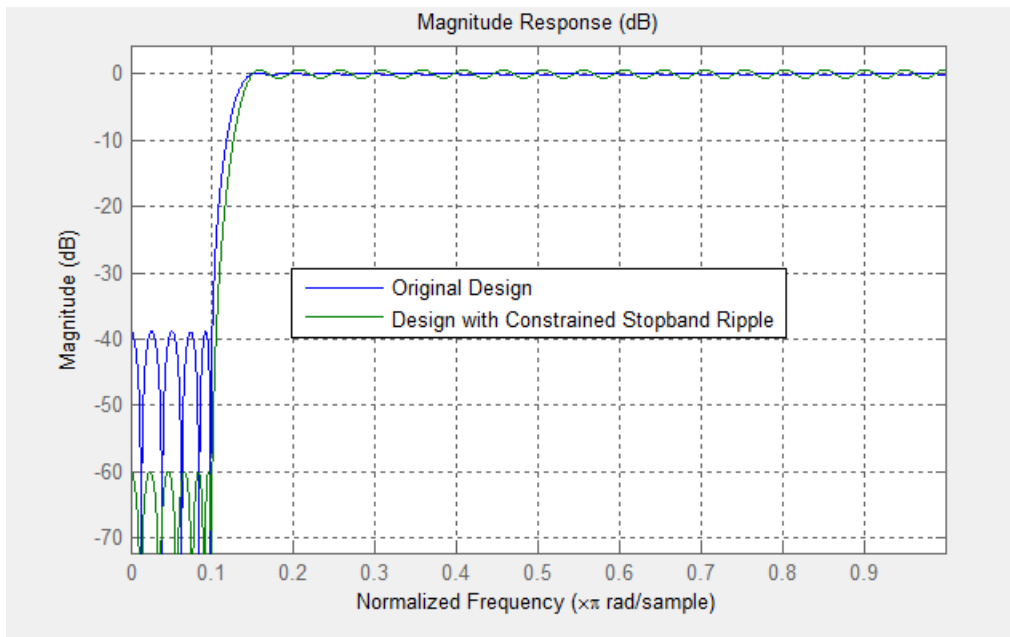
### Design of a multiband constrained arbitrary-magnitude filter

Use `fdesign.arbmag` to design a multiband constrained FIR filter.

This example requires the DSP System Toolbox.

Force the frequency response at  $0.15\pi$  radians/sample to 0 dB.

```
d = fdesign.arbmag('N,B,F,A,C',82,2);
d.B1Frequencies = [0 0.06 .1];
d.B1Amplitudes = [0 0 0];
d.B2Frequencies = [.15 1];
d.B2Amplitudes = [1 1];
% Design a filter with no constraints
Hd1 = design(d,'equiripple','B2ForcedFrequencyPoints',0.15);
% Add a constraint to the first band to increase attenuation
d.B1Constrained = true;
d.B1Ripple = .001;
Hd2 = design(d,'equiripple','B2ForcedFrequencyPoints',0.15);
hfvt = fvtool(Hd1,Hd2);
legend(hfvt,'Original Design','Design with Constrained Stopband Ripple');
```



**See Also**

`design` | `designmethods` | `fdesign`

# fdesign.bandpass

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.

`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.

---

**Note:** Specifications strings marked with an asterisk require the DSP System Toolbox software.

---

- '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,C' \*
- 'N,Fst1,Fp1,Fp2,Fst2,Ap' \*
- 'N,Fst1,Fst2,Ast'
- 'Nb,Na,Fst1,Fp1,Fp2,Fst2' \*

The string entries are defined as follows:

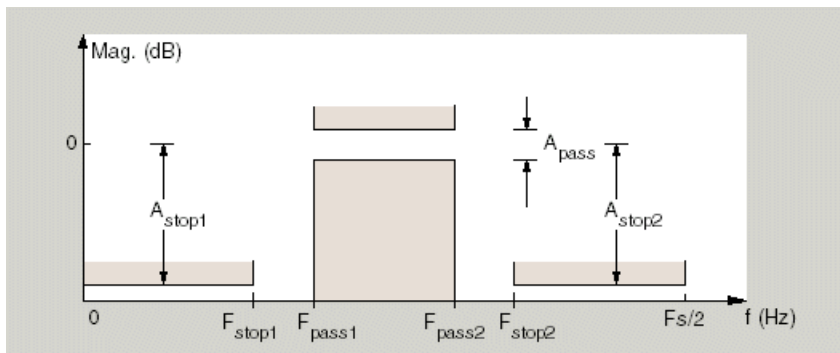
- **Ap** — amount of ripple allowed in the pass band. Also called **Apass**.
- **Ast1** — attenuation in the first stop band in decibels (the default units). Also called **Astop1**.
- **Ast2** — attenuation in the second stop band in decibels (the default units). Also called **Astop2**.
- **BWp** — bandwidth of the filter passband. Specified in normalized frequency units.
- **BWst** — bandwidth of the filter stopband. Specified in normalized frequency units.
- **C** — Constrained band flag. This enables you to specify passband ripple or stopband attenuation for fixed-order designs in one or two of the three bands.

In the specification string 'N,Fst1,Fp1,Fp2,Fst2,C', you cannot specify constraints in both stopbands and the passband simultaneously. You can specify constraints in any one or two bands.

- **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 6 dB point below the passband value for the first cutoff. Specified in normalized frequency units. (FIR filters)
- **Fc2** — cutoff frequency for the point 6 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 methods apply to an object and the **Specification** property value.

Use **designopts** to determine the design options for a given design method. Enter **help(D, METHOD)** at the MATLAB command line to obtain detailed help on the design options for a given design method, **METHOD**.

**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` 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

Filter a discrete-time signal with a bandpass filter. The signal is a sum of three discrete-time sinusoids,  $\pi/8$ ,  $\pi/2$ , and  $3\pi/4$  radians/sample.

```
n = 0:159;
x = cos(pi/8*n)+cos(pi/2*n)+sin(3*pi/4*n);
```

Design an FIR equiripple bandpass filter to remove the lowest and highest discrete-time sinusoids.

```
d = fdesign.bandpass('Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2',1/4,3/8,5/8,6/8,60,1,60);
Hd = design(d,'equiripple');
```

Apply the filter to the discrete-time signal.

```
y = filter(Hd,x);
freq = 0:(2*pi)/length(x):pi;
xdft = fft(x);
```



```

ydft = fft(y);
plot(freq,abs(xdft(1:length(x)/2+1)));
hold on;
plot(freq,abs(ydft(1:length(x)/2+1)),'r','linewidth',2);
legend('Original Signal','Bandpass Signal');

```

Design an IIR Butterworth filter of order 10 with 3-dB frequencies of 1 and 1.2 kHz. The sampling frequency is 10 kHz

```

d = fdesign.bandpass('N,F3dB1,F3dB2',10,1e3,1.2e3,1e4);
Hd = design(d,'butter');
fvtool(Hd)

```

This example requires the DSP System Toolbox software.

Design a constrained-band FIR equiripple filter of order 100 with a passband of [1, 1.4] kHz. Both stopband attenuation values are constrained to 60 dB. The sampling frequency is 10 kHz.

```

d = fdesign.bandpass('N,Fst1,Fp1,Fp2,Fst2,C',100,800,1e3,1.4e3,1.6e3,1e4);
d.Stopband1Constrained = true; d.Astop1 = 60;
d.Stopband2Constrained = true; d.Astop2 = 60;
Hd = design(d,'equiripple');
fvtool(Hd);
measure(Hd)

```

The passband ripple is slightly over 2 dB. Because the design constrains both stopbands, you cannot constrain the passband ripple.

## See Also

fdesign, fdesign.bandstop, fdesign.highpass, fdesign.lowpass

## fdesign.bandstop

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`.

`D = fdesign.bandstop(SPEC)` constructs object `D` and sets the `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. The strings are not case sensitive.

---

**Note:** Specifications strings marked with an asterisk require the DSP System Toolbox software.

---

- 'Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2' (default spec)
- '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,C' \*
- 'N,Fp1,Fst1,Fst2,Fp2,Ap' \*
- 'N,Fst1,Fst2,Ast'
- 'Nb,Na,Fp1,Fst1,Fst2,Fp2' \*

The string entries are defined as follows:

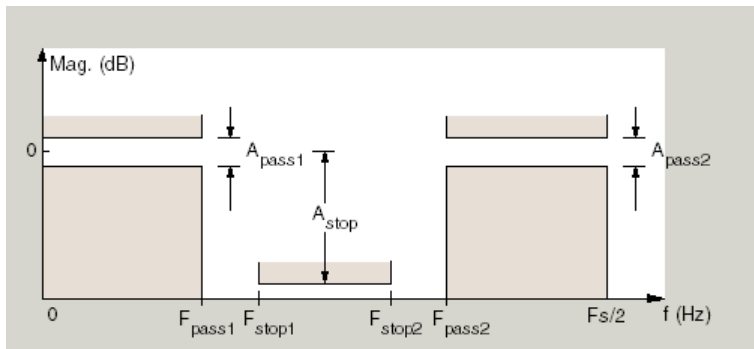
- **Ap** — amount of ripple allowed in the passband in decibels (the default units). Also called **Apass**.
- **Ap1** — amount of ripple allowed in the pass band in decibels (the default units). Also called **Apass1**.
- **Ap2** — amount of ripple allowed in the pass band in decibels (the default units). 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.
- **C** — Constrained band flag. This enables you to specify passband ripple or stopband attenuation for fixed-order designs in one or two of the three bands.

In the specification string 'N,Fp1,Fst1,Fst2,Fp2,C', you cannot specify constraints simultaneously in both passbands and the stopband. You can specify constraints in any one or two bands.

- **F3dB1** — cutoff frequency for the point 3 dB point below the passband value for the first cutoff.
- **F3dB2** — cutoff frequency for the point 3 dB point below the passband value for the second cutoff.
- **Fc1** — cutoff frequency for the point 6 dB point below the passband value for the first cutoff. (FIR filters)

- $F_{c2}$  — cutoff frequency for the point 6 dB point below the passband value for the second cutoff. (FIR filters)
- $F_{p1}$  — frequency at the start of the pass band. Also called  $F_{pass1}$ .
- $F_{p2}$  — frequency at the end of the pass band. Also called  $F_{pass2}$ .
- $F_{st1}$  — frequency at the end of the first stop band. Also called  $F_{stop1}$ .
- $F_{st2}$  — frequency at the start of the second stop band. Also called  $F_{stop2}$ .
- $N$  — filter order.
- $N_a$  — denominator order for IIR filters.
- $N_b$  — numerator order for IIR filters.

Graphically, the filter specifications look similar to those shown in the following figure.



Regions between specification values like  $F_{p1}$  and  $F_{st1}$  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 methods apply to an object and the `Specification` property value.

Use `designopts` to determine the design options for a given design method. Enter `help(D, METHOD)` at the MATLAB command line to obtain detailed help on the design options for a given design method, `METHOD`.

`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, 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. If you specify the sampling frequency as a trailing scalar, 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

Construct a bandstop filter to reject the discrete frequency band between  $3\pi/8$  and  $5\pi/8$  radians/sample. Apply the filter to a discrete-time signal consisting of the superposition of three discrete-time sinusoids.

Design an FIR equiripple filter and view the magnitude response.

```
d = fdesign.bandstop('Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2',2/8,3/8,5/8,6/8,1,60,1);
Hd = design(d,'equiripple');
fvtool(Hd)
```

Construct the discrete-time signal to filter.

```
n = 0:99;
x = cos(pi/5*n)+sin(pi/2*n)+cos(4*pi/5*n);
y = filter(Hd,x);
xdft = fft(x);
ydft = fft(y);
freq = 0:(2*pi)/length(x):pi;
plot(freq,abs(xdft(1:length(x)/2+1)));
```

```
hold on;  
plot(freq,abs(ydft(1:length(y)/2+1)), 'r', 'linewidth',2);  
xlabel('Radians/Sample'); ylabel('Magnitude');  
legend('Original Signal', 'Bandstop Signal');
```

Create a Butterworth bandstop filter for data sampled at 10 kHz. The stopband is [1,1.5] kHz. The order of the filter is 20.

```
d = fdesign.bandstop('N,F3dB1,F3dB2',20,1e3,1.5e3,1e4);  
Hd = design(d,'butter');  
fvtool(Hd);
```

Zoom in on the magnitude response plot to verify that the 3-dB down points are located at 1 and 1.5 kHz.

The following example requires the DSP System Toolbox license.

Design a constrained-band FIR equiripple filter of order 100 for data sampled at 10 kHz. You can specify constraints on at most two of the three bands: two passbands and one stopband. In this example, you choose to constrain the passband ripple to be 0.5 dB in each passband. Design the filter, visualize the magnitude response and measure the filter's design.

```
d = fdesign.bandstop('N,Fp1,Fst1,Fst2,Fp2,C',100,800,1e3,1.5e3,1.7e3,1e4);  
d.Passband1Constrained = true; d.Apass1 = 0.5;  
d.Passband2Constrained = true; d.Apass2 = 0.5;  
Hd = design(d,'equiripple');  
fvtool(Hd);  
measure(Hd)
```

With this order filter and passband ripple constraints, you achieve approximately 50 dB of stopband attenuation.

## See Also

fdesign, fdesign.bandpass, fdesign.highpass, fdesign.lowpass

# fdesign.differentiator

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.

---

**Note:** Specifications strings marked with an asterisk require the DSP System Toolbox software.

---

- 'N' — Full band differentiator (default)
- 'N,Fp,Fst' — Partial band differentiator
- 'N,Fp,Fst,Ap' — Partial band differentiator \*
- 'N,Fp,Fst,Ast' — 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.

Use `designopts` to determine the design options for a given design method. Enter `help(D, METHOD)` at the MATLAB command line to obtain detailed help on the design options for a given design method, `METHOD`.

`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

Use an FIR equiripple differentiator to transform frequency modulation into amplitude modulation, which can be detected using an envelope detector.

Modulate a message signal consisting of a 20-Hz sine wave with a 1 kHz carrier frequency. The sampling frequency is 10 kHz .

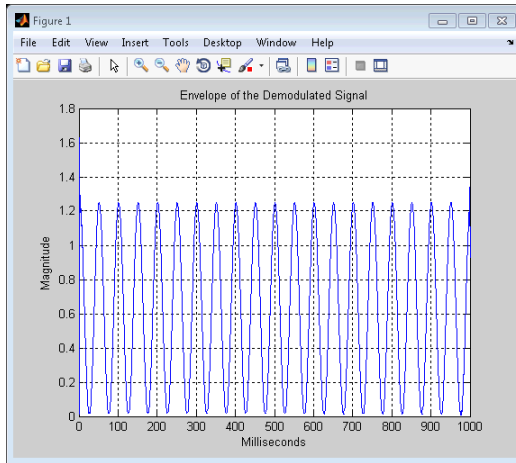
```
t = linspace(0,1,1e4);  
x = cos(2*pi*20*t);  
Fc = 1e3;  
Fs = 1e4;  
y = modulate(x,Fc,Fs,'fm');
```

Design the equiripple FIR differentiator of order 31.

```
d = fdesign.differentiator(31,1e4);  
Hd = design(d,'equiripple');
```

Filter the modulated signal and take the Hilbert transform to obtain the envelope.

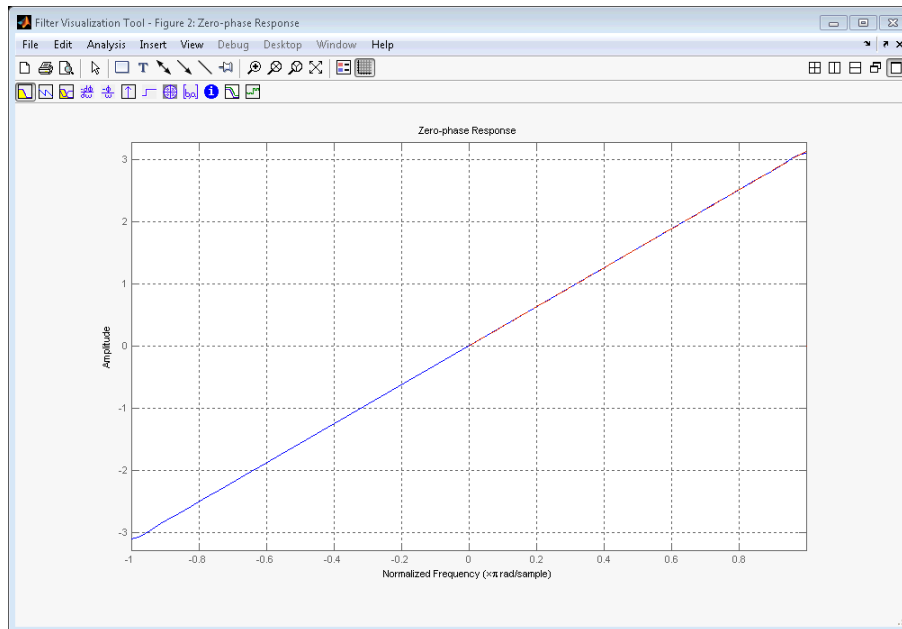
```
y1 = filter(Hd,y);  
y1 = hilbert(y1);  
% Plot the envelope  
plot(t.*1000,abs(y1));  
xlabel('Milliseconds'); ylabel('Magnitude');  
grid on;  
title('Envelope of the Demodulated Signal');
```



From the preceding figure, you see that the envelope completes two cycles every 100 milliseconds. The envelope is oscillating at 20 Hz, which corresponds to the frequency of the message signal.

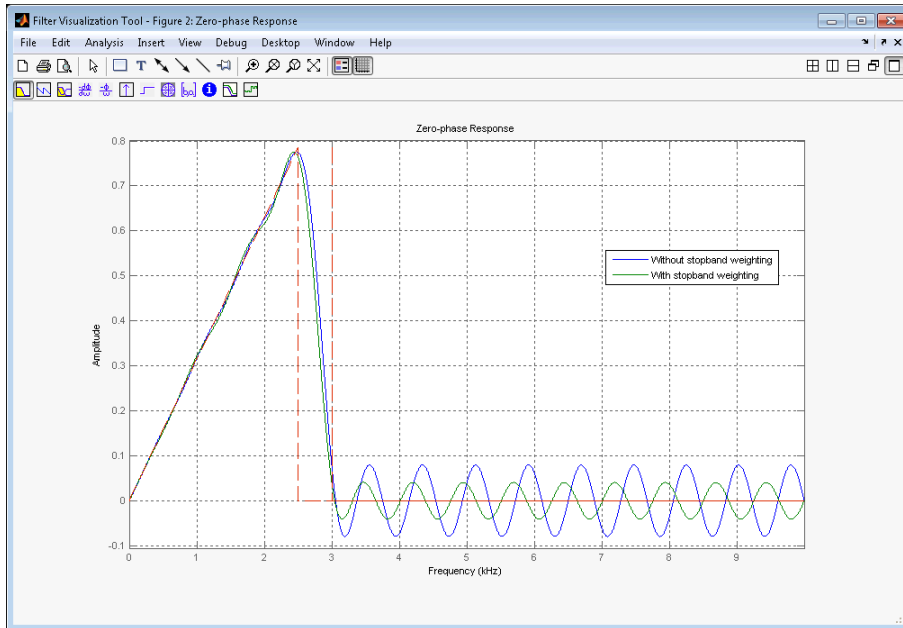
Design an FIR differentiator using least squares and plot the zero phase response.

```
d = fdesign.differentiator(33); % Filter order is 33.
hd = design(d,'firls');
fvtool(hd,'magnitudedisplay','zero-phase',...
'frequencyrange','[-pi, pi]')
```



Design a narrow band differentiator. Differentiate the first 25 percent of the frequencies in the Nyquist range and filter the higher frequencies.

```
Fs=20000; %sampling frequency
d = fdesign.differentiator('N,Fp,Fst',54,2500,3000,Fs);
Hd= design(d,'equiripple');
% Weight the stopband to increase attenuation
Hd1 = design(d,'equiripple','Wstop',4);
hfvt = fvtool(Hd,Hd1,'magnitudedisplay','zero-phase',...
'frequencyrange',[0, Fs/2]);
legend(hfvt,'Without stopband weighting',...
'With stopband weighting');
```



**See Also**  
design | fdesign

# fdesign.highpass

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 specification string, `'Fst,Fp,Ast,Ap'`.

`D = fdesign.highpass(SPEC)` constructs object `D` and sets the `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. The strings are not case sensitive.

---

**Note:** Specifications strings marked with an asterisk require the DSP System Toolbox software.

---

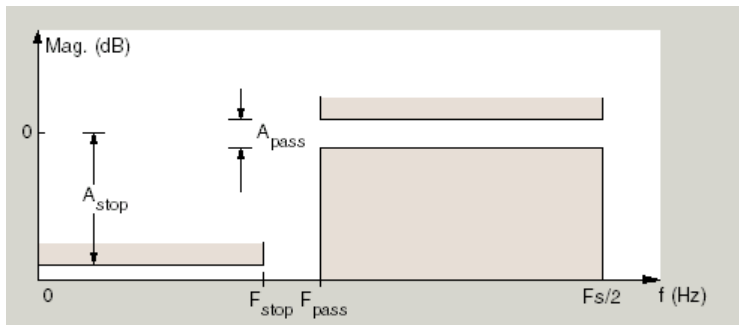
- `'Fst,Fp,Ast,Ap'` (default spec)
- `'N,F3db'`
- `'N,F3db,Ap' *`
- `'N,F3db,Ast' *`
- `'N,F3db,Ast,Ap' *`
- `'N,F3db,Fp' *`

- '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 6 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  $F_{st}$  and  $F_p$  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.

Use `designopts` to determine which design options are valid for a given design method. For detailed information on design options for a given design method, `METHOD`, enter `help(D,METHOD)` at the MATLAB command line.

`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 default `Specification` property and the values you enter for `specvalue1,specvalue2,...`.

`D = fdesign.highpass(...,Fs)` provides the sampling frequency for the filter specification object. `Fs` is in Hz and must be specified as a scalar trailing the other numerical values provided. If you specify a sampling frequency, all other frequency specifications are in Hz.

`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

Highpass filter a discrete-time signal consisting of two sine waves.

Create a highpass filter specification object. Specify the passband frequency to be  $0.25\pi$  radians/sample and the stopband frequency to be  $0.15\pi$  radians/sample. Specify 1 dB of allowable passband ripple and a stopband attenuation of 60 dB.

```
d = fdesign.highpass('Fst,Fp,Ast,Ap',0.15,0.25,60,1);
```

Query the valid design methods for your filter specification object, `d`.

```
designmethods(d)
```

Create an FIR equiripple filter and view the filter magnitude response with `fvtool`.

```
Hd = design(d,'equiripple');  
fvtool(Hd);
```

Create a signal consisting of the sum of two discrete-time sinusoids with frequencies of  $\pi/8$  and  $\pi/4$  radians/sample and amplitudes of 1 and 0.25 respectively. Filter the discrete-time signal with the FIR equiripple filter object, `Hd`

```
n = 0:159;  
x = cos((pi/8)*n)+0.25*sin((pi/4)*n);  
y = filter(Hd,x);  
Domega = (2*pi)/160;  
freq = 0:(2*pi)/160:pi;  
xdft = fft(x);  
ydft = fft(y);  
plot(freq,abs(xdft(1:length(x)/2+1)));  
hold on;  
plot(freq,abs(ydft(1:length(y)/2+1)),'r','linewidth',2);  
legend('Original Signal','Lowpass Signal', ...  
      'Location','NorthEast');  
ylabel('Magnitude'); xlabel('Radians/Sample');
```

Create a filter of order 10 with a 6-dB frequency of 9.6 kHz and a sampling frequency of 48 kHz.



```
d=fdesign.highpass('N,Fc',10,9600,48000);
designmethods(d)
% only valid design method is FIR window method
Hd = design(d);
% Display filter magnitude response
fvtool(Hd);
```

If you have the DSP System Toolbox software, you can specify the shape of the stopband and the rate at which the stopband decays.

Create two FIR equiripple filters with different linear stopband slopes. Specify the passband frequency to be  $0.3\pi$  radians/sample and the stopband frequency to be  $0.35\pi$  radians/sample. Specify 1 dB of allowable passband ripple and a stopband attenuation of 60 dB. Design one filter with a 20 dB/rad/sample stopband slope and another filter with 40 dB/rad/sample.

```
D = fdesign.highpass('Fst,Fp,Ast,Ap',0.3,0.35,60,1);
Hd1 = design(D,'equiripple','StopBandShape','linear','StopBandDecay',20);
Hd2 = design(D,'equiripple','StopBandShape','linear','StopBandDecay',40);
hfvt = fvtool([Hd1 Hd2]);
legend(hfvt,'20 dB/rad/sample','40 dB/rad/sample');
```

## See Also

design | designmethods | fdesign

## fdesign.hilbert

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 30 and `TW`, the transition width set to  $0.1\pi$  radians/sample.

`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 specification strings are not case sensitive.

---

**Note:** Specifications strings marked with an asterisk require the DSP System Toolbox software.

---

- `'N,TW'` default spec string.
- `'TW,Ap'` \*

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. 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

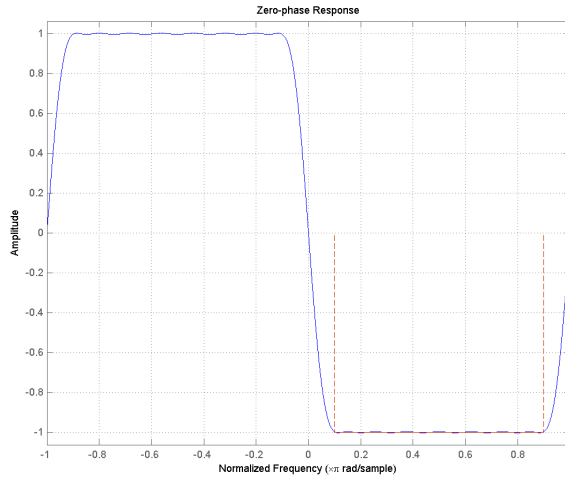
When you omit the `MAGUNITS` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

## Examples

Design a Hilbert transformer of order 30 with a transition width of  $0.2\pi$  radians/sample. Plot the zero phase response from  $[-\pi,\pi]$  radians/sample and the impulse response.

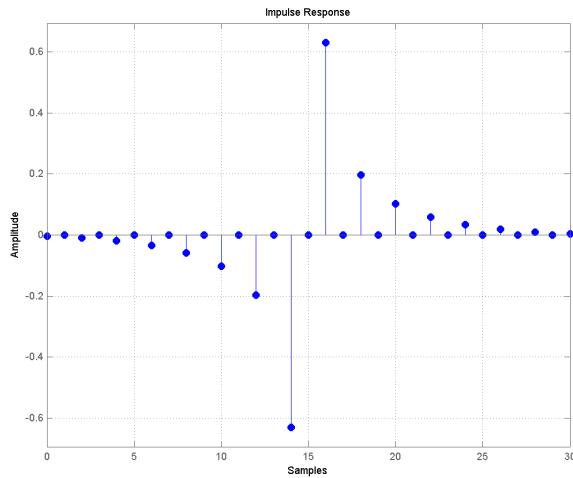
```
d = fdesign.hilbert('N,TW',30,0.2);
% Show available design methods
designmethods(d)
% Use least square minimization to obtain linear-phase FIR filter
Hd = design(d,'equiripple');
```

```
% Display zero phase response from [-pi,pi)
fvtool(Hd,'magnitudedisplay','zero-phase',...
'frequencyrange','[-pi, pi)')
```



The impulse response of this even order filter is antisymmetric (type III).

```
fvtool(Hd,'analysis','impulse')
```

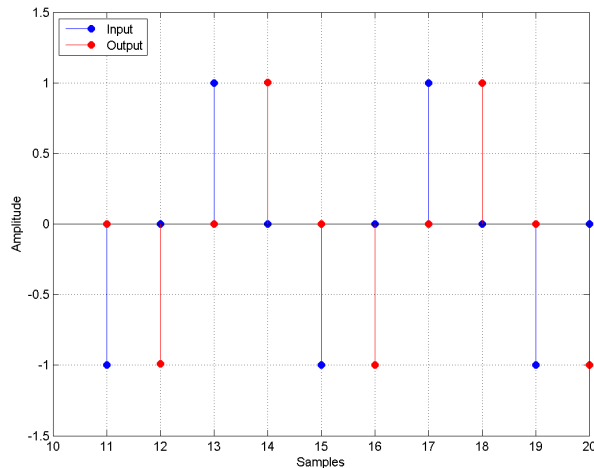


Apply the filter to a discrete-time sinusoid with a frequency of  $\pi/2$  radians/sample.

```
n = 0:99;
x = cos(pi/2*n);
y = filter(Hd,x);
% Correct for the filter delay
Delay = floor(length(Hd.Numerator)/2);
y = y(Delay+1:end);
```

Plot the filter input and output and validate the approximate  $\pi/2$  phase shift obtained with the Hilbert transformer.

```
stem(x(1:end-Delay), 'markerfacecolor', [0 0 1]);
hold on;
stem(y, 'Color', [1 0 0], 'markerfacecolor', [1 0 0]);
axis([10 20 -1.5 1.5]); grid on;
xlabel('Samples'); ylabel('Amplitude');
legend('Input', 'Output', 'Location', 'NorthWest')
```



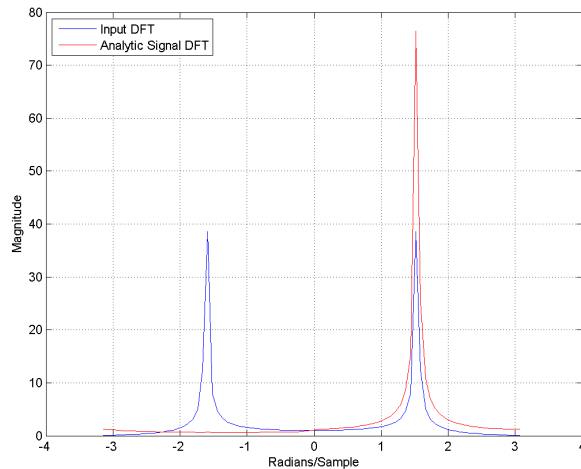
Because the frequency of the discrete-time sinusoid is  $\pi/2$  radians/sample, a one sample shift corresponds to a phase shift of  $\pi/2$ .

Form the analytic signal and demonstrate that the frequency content of the analytic signal is zero for negative frequencies and approximately twice the spectrum of the input for positive frequencies.

```

x1 = x(1:end-Delay);
% Form the analytic signal
xa = x1+1j*y;
freq = -pi:(2*pi)/length(x1):pi-(2*pi)/length(x);
plot(freq,abs(fftshift(fft(x1))));
hold on;
plot(freq,abs(fftshift(fft(xa))), 'r'); grid on;
xlabel('Radians/Sample'); ylabel('Magnitude');
legend('Input DFT', 'Analytic Signal DFT', 'Location', 'NorthWest');

```

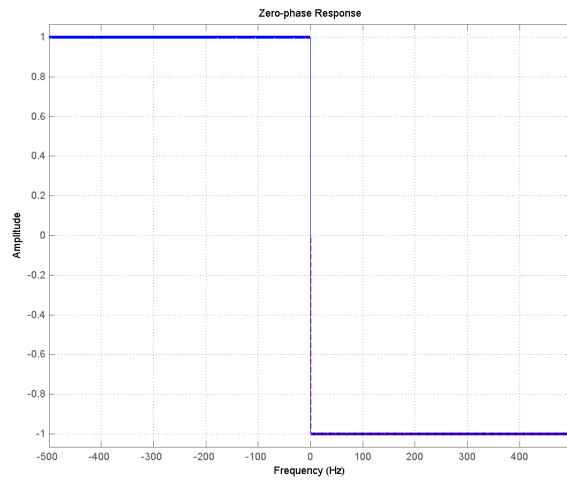


Design a minimum-order Hilbert transformer that has a sampling frequency of 1 kHz. Specify the passband ripple to be 1 dB.

```

d = fdesign.hilbert('TW,Ap',1,0.1,1e3);
hd = design(d,'equiripple');
fvtool(hd,'magnitudedisplay','zero-phase', ...
'frequencyrange','[-Fs/2, Fs/2)');

```



## See Also

`design` | `fdesign` | `setspecs`

## fdesign.lowpass

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 default specification string `'Fp,Fst,Ap,Ast'`.

`D = fdesign.lowpass(SPEC)` constructs object `D` and sets the `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.

---

**Note:** Specifications strings marked with an asterisk require the DSP System Toolbox software.

---

- `'Fp,Fst,Ap,Ast'` (default spec)
- `'N,F3db'`
- `'N,F3db,Ap' *`
- `'N,F3db,Ap,Ast' *`
- `'N,F3db,Ast' *`
- `'N,F3db,Fst' *`
- `'N,Fc'`

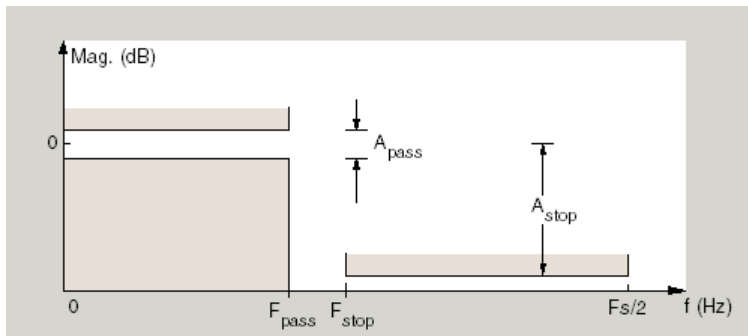


- 'N,Fc,Ap,Ast'
- 'N,Fp,Ap'
- 'N,Fp,Ap,Ast'
- 'N,Fp,Fst,Ap' \*
- 'N,Fp,F3db' \*
- 'N,Fp,Fst'
- 'N,Fp,Fst,Ast' \*
- 'N,Fst,Ap,Ast' \*
- 'N,Fst,Ast'
- 'Nb,Na,Fp,Fst' \*

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 6 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  $F_p$  and  $F_{st}$  are transition regions where the filter response is not explicitly defined.

`D = fdesign.lowpass(SPEC, specvalue1, specvalue2, ...)` constructs an object `D` and sets the 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

### Lowpass Filtering and Filter Visualization

Lowpass filter a discrete-time signal consisting of two sine waves.

Create a lowpass filter specification object. Specify the passband frequency to be  $0.15\pi$  rad/sample and the stopband frequency to be  $0.25\pi$  rad/sample. Specify 1 dB of allowable passband ripple and a stopband attenuation of 60 dB.

```
d = fdesign.lowpass('Fp,Fst,Ap,Ast',0.15,0.25,1,60);
```

Query the valid design methods for your filter specification object, d.

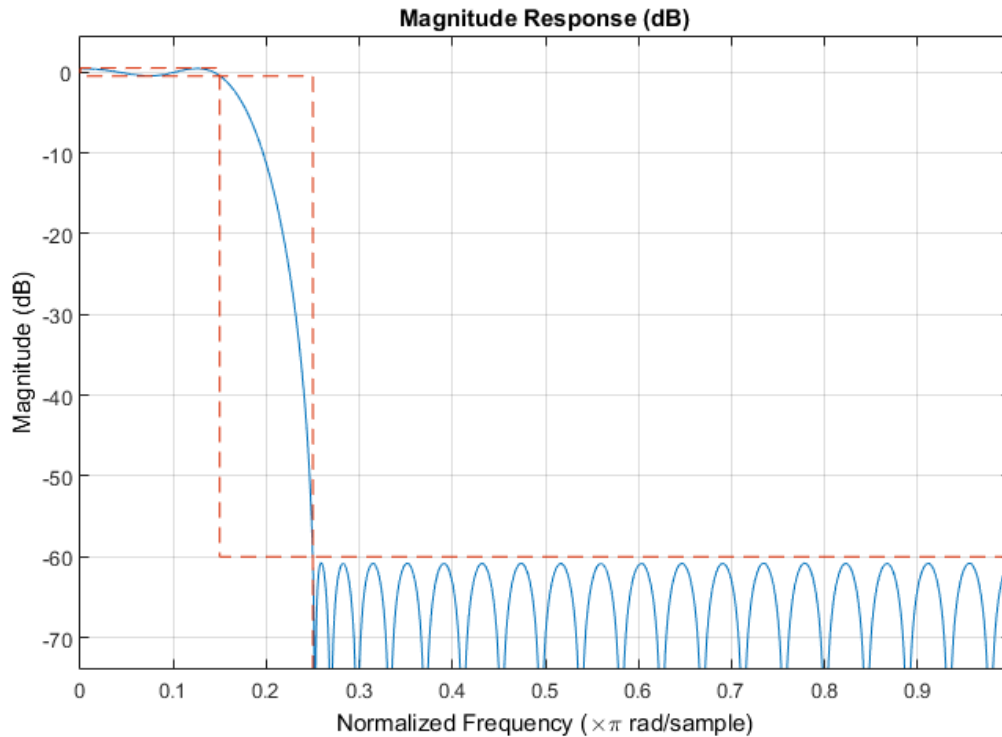
```
designmethods(d)
```

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

```
butter  
cheby1  
cheby2  
ellip  
equiripple  
ifir  
kaiserwin  
multistage
```

Create an FIR equiripple filter and view the filter magnitude response with fvtool.

```
Hd = design(d,'equiripple');  
fvtool(Hd)
```



Create a signal consisting of the sum of two discrete-time sinusoids with frequencies of  $\pi/8$  and  $\pi/4$  rad/sample and amplitudes of 1 and 0.25, respectively. Filter the discrete-time signal with the FIR equiripple filter object, Hd.

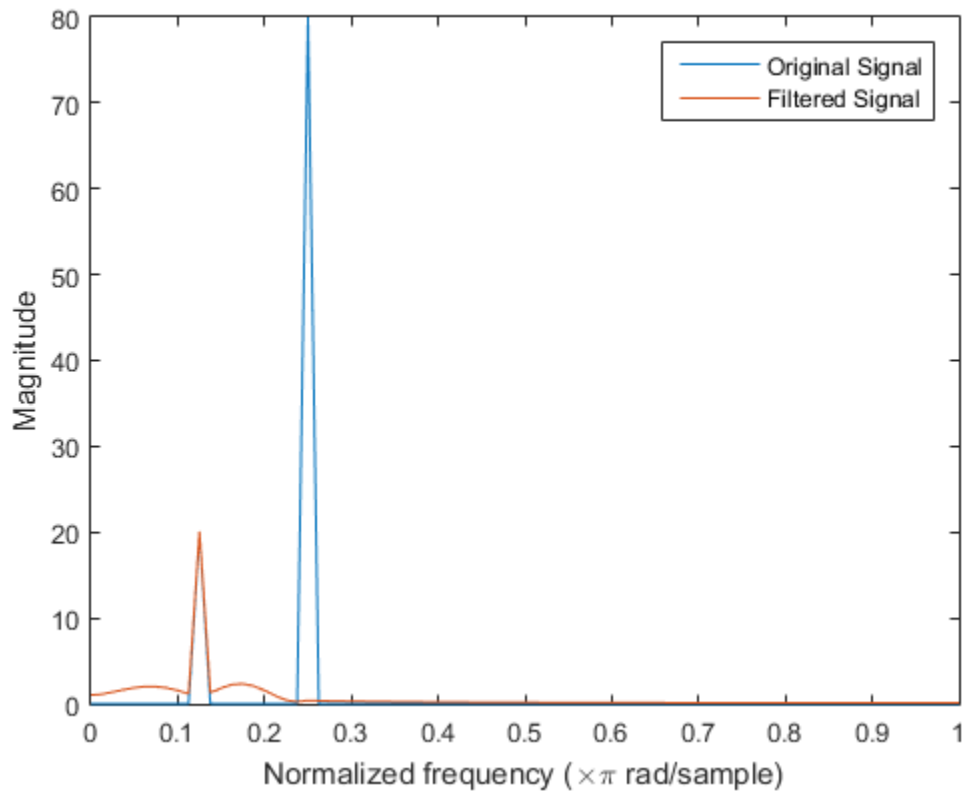
```
n = 0:159;
x = 0.25*cos((pi/8)*n)+sin((pi/4)*n);
y = filter(Hd,x);
```

Compute the Fourier transform of the original signal and the filtered signal. Verify that the high-frequency component has been filtered out.

```
freq = 0:(2*pi)/160:pi;
xdft = fft(x);
ydft = fft(y);
```

```
figure
```

```
plot(freq/pi,abs(xdft(1:length(x)/2+1)))  
hold on  
plot(freq/pi,abs(ydft(1:length(y)/2+1)))  
  
legend('Original Signal','Filtered Signal')  
ylabel('Magnitude')  
xlabel('Normalized frequency (\times\pi rad/sample)')
```



Create a filter of order 10 with a 6-dB frequency of 9.6 kHz and a sampling frequency of 48 kHz. The only valid design method is the FIR window method.

```
d = fdesign.lowpass('N,Fc',10,9600,48000);  
designmethods(d)
```

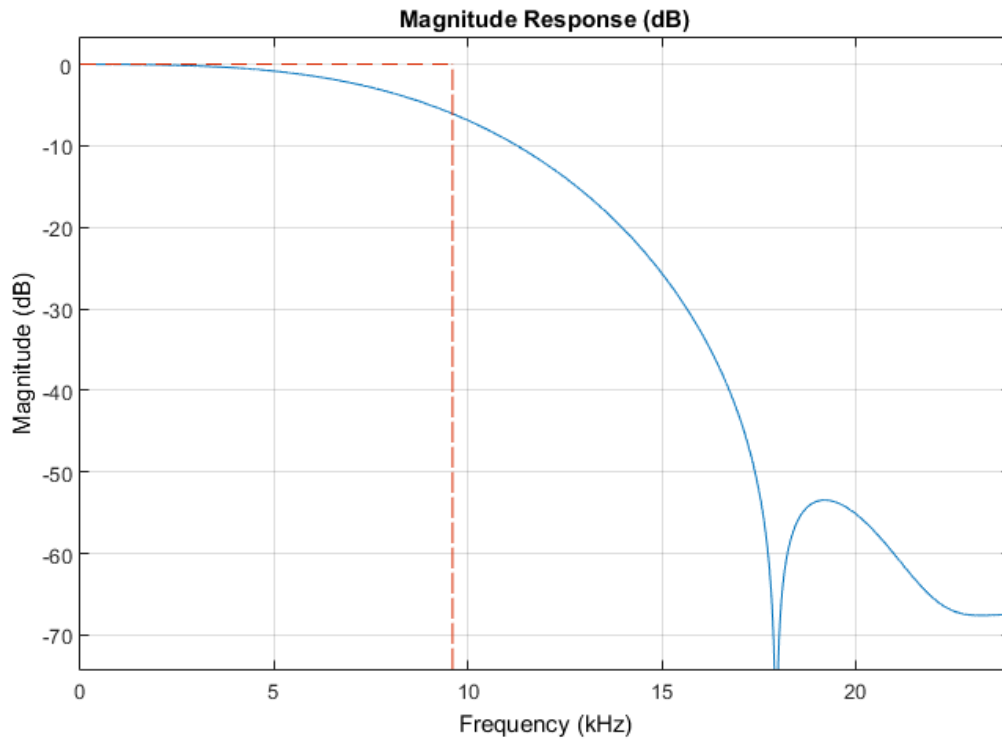
```
Hd = design(d);
```

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

```
window
```

Display the filter magnitude response. The -6 dB point is at 9.6 kHz, as expected.

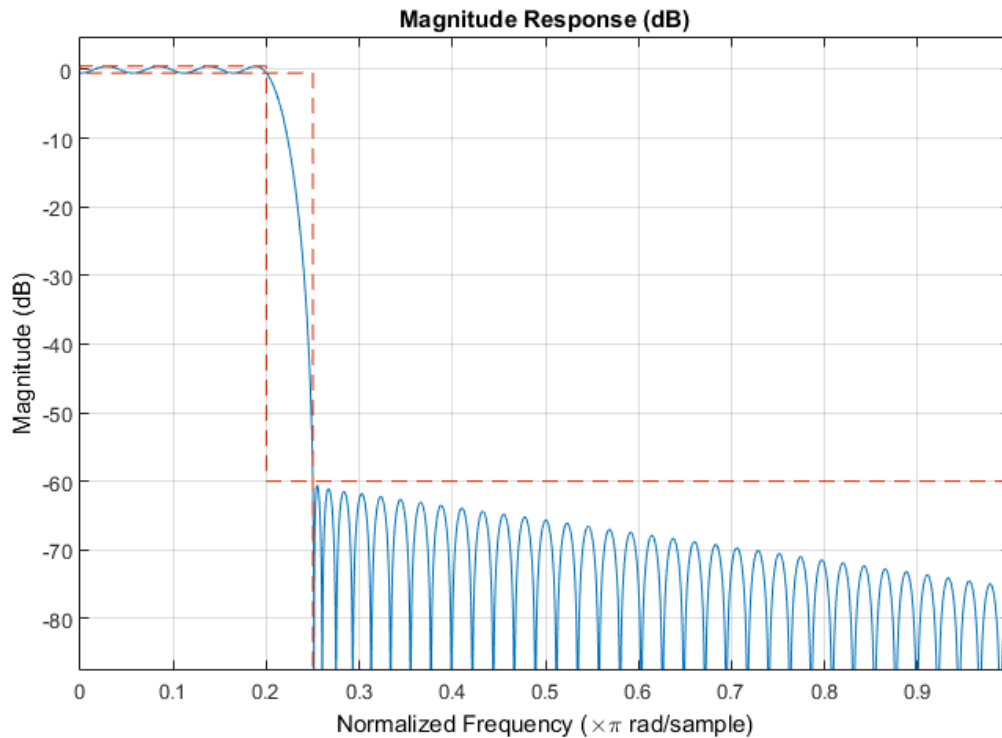
```
fvtool(Hd)
```



If you have the DSP System Toolbox software, you can specify the shape of the stopband and the rate at which the stopband decays. The following example requires the DSP System Toolbox.

Create an FIR equiripple filter with a passband frequency of  $0.2\pi$  rad/sample, a stopband frequency of  $0.25\pi$  rad/sample, a passband ripple of 1 dB, and a stopband attenuation of 60 dB. Design the filter with a 20 dB/rad/sample linear stopband.

```
D = fdesign.lowpass('Fp,Fst,Ap,Ast',0.2,0.25,1,60);  
Hd = design(D,'equiripple','StopbandShape','linear','StopbandDecay',20);  
fvtool(Hd)
```



## See Also

[design](#) | [designmethods](#) | [fdesign](#)

## fftfilt

FFT-based FIR filtering using overlap-add method

### Syntax

```
y = fftfilt(b,x)
y = fftfilt(b,x,n)
y = fftfilt(d,x)
y = fftfilt(d,x,n)
y = fftfilt(gpuArrayb,gpuArrayX,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:

$$y(n) = b(1)x(n) + b(2)x(n-1) + \dots + b(nb+1)x(n-nb)$$

An equivalent representation is the Z-transform or *frequency domain* description:

$$Y(z) = (b(1) + b(2)z^{-1} + \dots + b(nb+1)z^{-nb}) X(z)$$

By default, `fftfilt` chooses an FFT length and a 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 “Algorithms” on page 1-485 for information.

`y = fftfilt(d,x)` filters the data in vector `x` with a `digitalFilter` object, `d`. Use `designfilt` to generate `d` based on frequency-response specifications.



`y = fftfilt(d,x,n)` uses `n` to determine the length of the FFT.

`y = fftfilt(gpuArrayb,gpuArrayX,n)` filters the data in the `gpuArray` object, `gpuArrayX`, with the FIR filter coefficients in the `gpuArray`, `gpuArrayb`. See “Establish Arrays on a GPU” for details on `gpuArray` objects. Using `fftfilt` with `gpuArray` objects requires Parallel Computing Toolbox software and a CUDA-enabled NVIDIA GPU with compute capability 1.3 or above. See <http://www.mathworks.com/products/parallel-computing/requirements.html> for details. The filtered data, `y`, is a `gpuArray` object. See “Overlap-Add Filtering on the GPU” on page 1-484 for example of overlap-add filtering on the GPU.

`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  $\frac{1}{2}L\log_2L$

where  $L$  is the block length. It then performs  $L$  point-wise multiplications for a total cost of

$$L + L\log_2L = L(1 + \log_2L)$$

multiplications. The cost ratio is therefore

$$L(1+\log_2L) / (NL) = (1 + \log_2L)/N$$

which is approximately  $\log_2L / N$ .

Therefore, `fftfilt` becomes advantageous when  $\log_2L$  is less than  $N$ .

## Examples

### fftfilt and filter for Short and Long Filters

Verify that `filter` is more efficient for smaller operands and `fftfilt` is more efficient for large operands. Filter  $10^6$  random numbers with two random filters, a short one, with 20 taps, and a long one, with 2000. Use `tic` and `toc` to measure the execution times.

```
rng default
x = rand(1,1e6);

shrt = 20;
fprintf('Filter of length %d:\n',shrt)
bshrt = rand(1,shrt);

tic
sfs = fftfilt(bshrt,x);
tsfs = toc;

tic
sls = filter(bshrt,1,x);
tsls = toc;

fprintf('fftfilt takes %f s to run, filter takes %f s to run\n',tsfs,tsls)

long = 2000;
fprintf('\nFilter of length %d:\n',long)
blong = rand(1,long);

tic
sfl = fftfilt(blong,x);
tsfl = toc;

tic
sll = filter(blong,1,x);
tsll = toc;

fprintf('fftfilt takes %f s to run, filter takes %f s to run\n',tsfl,tsll)

Filter of length 20:
fftfilt takes 0.887197 s to run, filter takes 0.007273 s to run

Filter of length 2000:
fftfilt takes 0.101962 s to run, filter takes 0.097609 s to run
```

### **Overlap-Add Filtering on the GPU**

The following example requires Parallel Computing Toolbox software and a CUDA-enabled NVIDIA GPU with compute capability 1.3 or above. See <http://www.mathworks.com/products/parallel-computing/requirements.html> for details.

Create a signal consisting of a sum of sine waves in white Gaussian additive noise. The sine wave frequencies are 2.5, 5, 10, and 15 kHz. The sampling frequency is 50 kHz.

```

Fs = 50e3;
t = 0:1/Fs:10-(1/Fs);
x = cos(2*pi*2500*t)+0.5*sin(2*pi*5000*t)+0.25*cos(2*pi*10000*t)+ ...
    0.125*sin(2*pi*15000*t)+randn(size(t));

```

Design a lowpass FIR equiripple filter using `designfilt`.

```

d = designfilt('lowpassfir','SampleRate',Fs, ...
              'PassbandFrequency',5500,'StopbandFrequency',6000, ...
              'PassbandRipple',0.5,'StopbandAttenuation',50);
B = d.Coefficients;

```

Filter the data on the GPU using the overlap-add method. Put the data on the GPU using `gpuArray`. Return the output to the MATLAB workspace using `gather` and plot the power spectral density estimate of the filtered data.

```

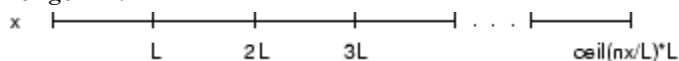
y = fftfilt(gpuArray(B),gpuArray(x));
periodogram(gather(y),rectwin(length(y)),length(y),50e3);

```

## More About

### Algorithms

`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  $N$ .



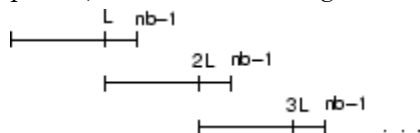
and convolves each block with the filter  $b$  by

```

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

$$2^{\text{nextpow2}(\text{length}(b) + \text{length}(x) - 1)}$$

This essentially computes

$$y = \text{ifft}(\text{fft}(B, \text{nfft}) .* \text{fft}(X, \text{nfft}))$$

If you supply a value for `n`, `fftfilt` chooses an FFT length, `nfft`, of  $2^{\text{nextpow2}(n)}$  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, Alan V., Ronald W. Schaffer, and John R. Buck. *Discrete-Time Signal Processing*. 2nd Ed. Upper Saddle River, NJ: Prentice Hall, 1999.

## See Also

`conv` | `designfilt` | `digitalFilter` | `filter` | `filtfilt`

# filter

Filter data with recursive (IIR) or nonrecursive (FIR) filter

## Description

`filter` is a MATLAB function.

## Signal-Specific Information

### Filter Using `digitalFilter` Objects

Use `filter` in the form `y = filter(d,x)` to filter an input signal, `x`, with a `digitalFilter`, `d`, and obtain output data, `y`.

Use `designfilt` to generate `d` based on frequency-response specifications.

`x` can be a double- or single-precision vector. It can also be a matrix with as many columns as there are input channels.

### 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)`.

### See Also

`designfilt` | `digitalFilter` | `fftfilt` | `filtfilt` | `sosfilt`

# filterbuilder

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 DSP System 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 “Filterbuilder Design Process”.

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. After you select a filter response type, `filterbuilder` launches the appropriate 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`, `mfilt`, or filter System object created using `fdesign`.)

---

**Note:** You must have the DSP System Toolbox software to create and import filter System objects.

---

- 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.

---

**Note:** You must have the DSP System Toolbox software to implement a number of the filter designs listed in the following table. If you only have the Signal Processing Toolbox software, you can design a limited set of the following filter-response types.

---

| Response String | Description of Resulting Filter Design          | Filter Object   |
|-----------------|---|---|
| arbgrpdelay     | Arbitrary group delay filter design             | <code>fdesign.arbgrpdelay</code>  |
| arbmag          | Arbitrary magnitude filter design               | <code>fdesign.arbmag</code>   |
| arbmagnphase    | Arbitrary response filter (magnitude and phase) | <code>fdesign.arbmagnphase</code>   |
| audioweighting  | Audio weighting filter                          | <code>fdesign.audioweighting</code>   |
| bandpass or bp  | Bandpass filter                                 | <code>fdesign.bandpass</code>   |
| bandstop or bs  | Bandstop filter                                 | <code>fdesign.bandstop</code>   |
| cic             | CIC filter                                      | <code>fdesign.decimator(M,'cic',...)</code><br>or<br><code>fdesign.interpolator(L,'cic',...)</code><br>See <code>fdesign.decimator</code><br>and<br><code>fdesign.interpolator</code> |
| ciccomp         | CIC compensator                                 | <code>fdesign.ciccomp</code>  |
| comb            | Comb filter                                     | <code>fdesign.comb</code>   |
| diff            | Differentiator filter                           | <code>fdesign.differentiator</code>   |
| fracdelay       | Fractional delay filter                         | <code>fdesign.fracdelay</code>  |

| Response String                  | Description of Resulting Filter Design     | Filter Object                          |
|----------------------------------|--|--|
| halfband or hb                   | Halfband filter                            | fdesign.halfband                       |
| highpass or hp                   | Highpass filter                            | fdesign.highpass                       |
| hilb                             | Hilbert filter                             | fdesign.hilbert                        |
| isinc,<br>isinclp, or<br>isinchp | Inverse sinc lowpass or<br>highpass filter | fdesign.isinclp and<br>fdesign.isinchp |
| lowpass or lp                    | Lowpass filter (default)                   | fdesign.lowpass                        |
| notch                            | Notch filter                               | fdesign.notch                          |
| nyquist                          | Nyquist filter                             | fdesign.nyquist                        |
| octave                           | Octave filter                              | fdesign.octave                         |
| parameq                          | Parametric equalizer filter                | fdesign.parameq                        |
| peak                             | Peak filter                                | fdesign.peak                           |

---

**Note:** Because they do not change the filter structure, the magnitude specifications and design method are tunable when using `filterbuilder`.

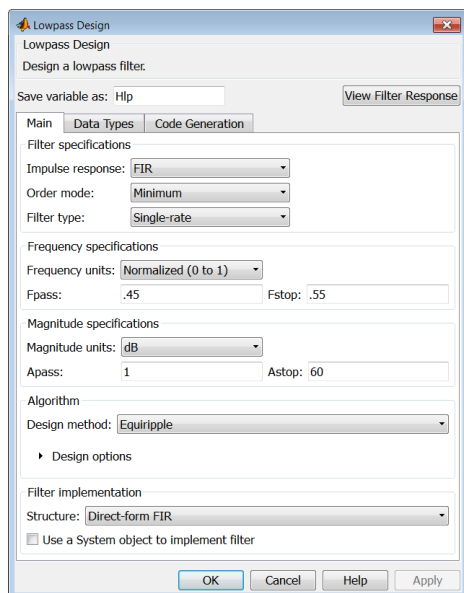
---

## Filterbuilder Design Panes

### Main Design Pane

The main pane of `filterbuilder` varies depending on the filter response type, but the basic structure is the same. The following figure shows the basic layout of the dialog box.





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**).

---

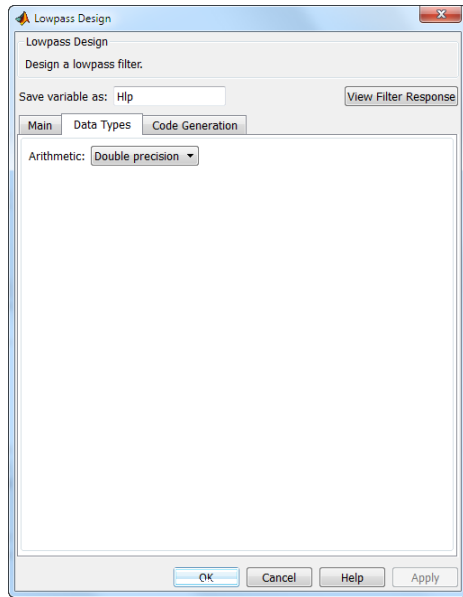
**Note:** The **filterbuilder** dialog box includes an **Apply** option. Each time you click **Apply**, **filterbuilder** writes the modified filter to your MATLAB workspace. This modified filter has the variable name you assign in **Save variable as**. To apply changes

without overwriting the variable in your 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 |

| Arithmetic List Entry | Effect on the Filter   |
|-----------------------|--|
|                       | filterbuilder to create a filter, double precision is the default value for the Arithmetic property.   |
| 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 Designer™ software along with DSP System Toolbox software. |

The following figure shows the **Data Types** pane after you select **Fixed point** for **Arithmetic** and set **Filter internals** to **Specify precision**. This figure shows the **Data Types** pane for the case where the **Use a System object to implement filter** check box is not selected in the **Main** pane.

Bandpass Design

Design a bandpass filter.

Save variable as:  View Filter Response

Main | Data Types | Code Generation

Arithmetic:

Fixed-point data types

|                  | Mode   | Signed                              | Word length                     | Fraction length                 |
|------------------|--|-------------------------------------|---------------------------------|---------------------------------|
| Input signal     | Binary point scaling                             | yes                                 | <input type="text" value="16"/> | <input type="text" value="15"/> |
| Coefficients     | <input type="text" value="Specify word length"/> | <input checked="" type="checkbox"/> | <input type="text" value="16"/> |                                 |
| Filter internals | <input type="text" value="Specify precision"/>   |                                     |                                 |                                 |
| Product          | Binary point scaling                             | yes                                 | <input type="text" value="32"/> | <input type="text" value="29"/> |
| Accum            | Binary point scaling                             | yes                                 | <input type="text" value="40"/> | <input type="text" value="29"/> |
| Output           | Binary point scaling                             | yes                                 | <input type="text" value="16"/> | <input type="text" value="15"/> |

Fixed-point operational parameters

Rounding mode:  Overflow mode:

OK Cancel Help Apply

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:

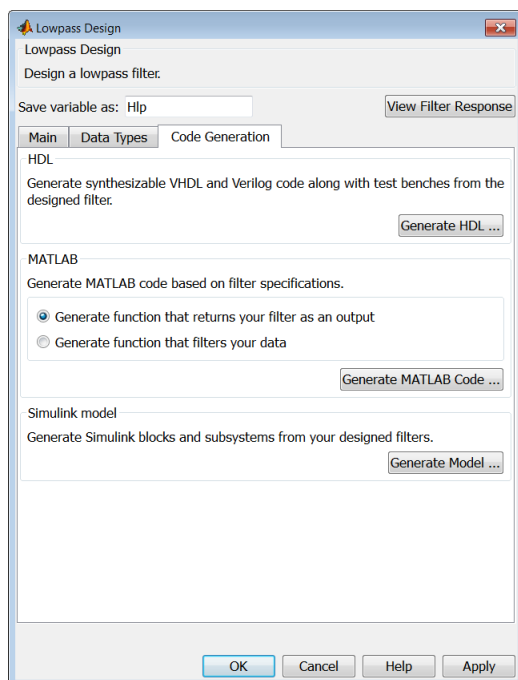
- **Cleared** — Configures filter summation operations to retain the addends in the format carried from the previous operation.



- 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. Depending on your installation, you can generate MATLAB, VHDL, and Verilog code from the designed filter. 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 Filter Design HDL Coder GUI From the filterbuilder GUI”.

### MATLAB

### Generate MATLAB code based on filter specifications

- **Generate function that returns your filter as an output**

Selecting this option generates a function that designs either a DFILT/MFILT object or a system object (depending on whether you have selected the **Use a System object to implement the filter** check box) using `fdesign`. The function call returns a filter object.

- **Generate function that filters your data**

Selecting this option generates a function that takes data as input, and outputs data filtered with the designed filter.

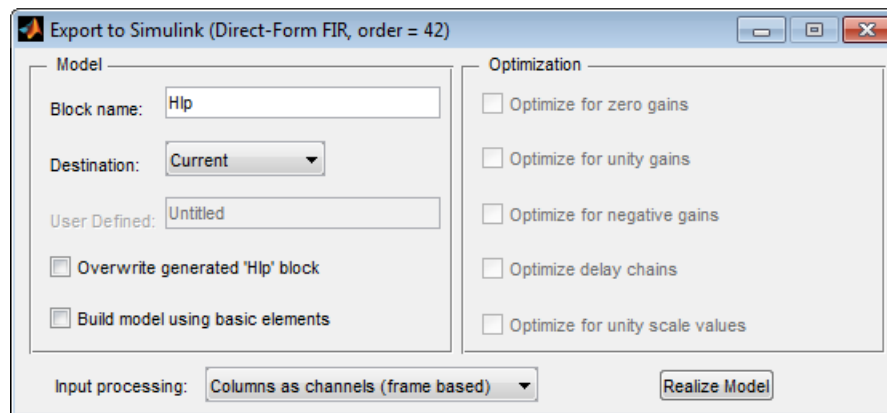
Clicking on the **Generate MATLAB code** button, brings up a Save File dialog. Specify the file name and location, and save. The filter is now contained in an editable file.

### Simulink Model

#### Generate Simulink blocks and subsystems from your designed filters

When the **Use a System object to implement filter** check box is selected in the **Main** pane, you are able to generate Simulink models as long as the **Arithmetic** is not set to **Fixed point** in the **Data Types** pane. If the **Arithmetic** is set to **Fixed point**, the **Generate Model** button in the **Simulink model** panel will be disabled.

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, DSP System 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, DSP System Toolbox software builds the model using only basic blocks.
- **Optimize for zero gains** — When this check box is selected, DSP System Toolbox software removes all zero gain blocks from the model.
- **Optimize for unity gains** — When this check box is selected, DSP System Toolbox software replaces all unity gains with direct connections.
- **Optimize for negative gains** — When this check box is selected, DSP System 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, DSP System Toolbox software replaces delay chains made up of  $n$  unit delays with a single delay by  $n$ .
- **Optimize for unity scale values** — When this check box is selected, DSP System Toolbox software removes all scale value multiplications by 1 from the filter structure.
- **Input processing** — Specify how the generated filter block or subsystem block processes the input. Depending on the type of filter you are designing, one or both of the following options may be available:
  - **Columns as channels (frame based)** — When you select this option, the block treats each column of the input as a separate channel.
  - **Elements as channels (sample based)** — When you select this option, the block treats each element of the input as a separate channel.

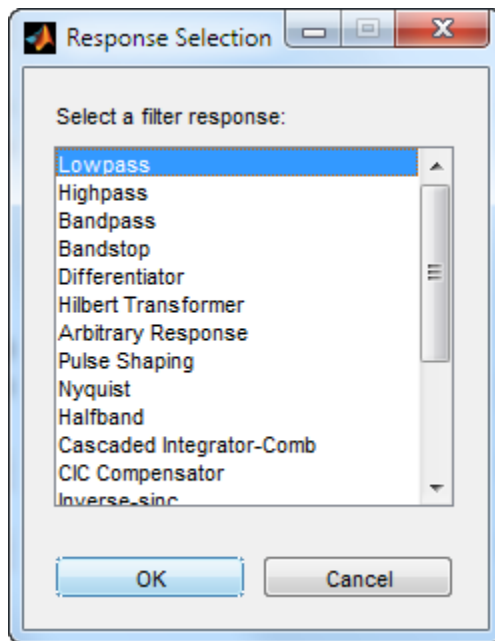
For more information about sample- and frame-based processing, see “Sample- and Frame-Based Concepts”.

- **Realize Model** — DSP System Toolbox software builds the model with the set parameters.

## Filter Responses

Select your filter response from the `filterbuilder` **Response Selection** main menu.

If you have the DSP System Toolbox software, the following **Response Selection** menu appears.



Select your desired filter response from the menu and design your filter.

The following sections describe the options available for each response type.

Arbitrary Response Design

Design an arbitrary response filter. The constraint can be on the magnitude only, or on the magnitude and the phase.

Save variable as: Ham View Filter Response

Main **Data Types** Code Generation

Filter specifications

Impulse response: FIR

Order mode: Specify

Order: 20

Filter type: Single-rate

Response specifications

Number of bands: 1

Specify response as: Amplitudes

Frequency units: Normalized (0 to 1)

Band properties

|   | Frequencies        | Amplitudes                          |
|---|--------------------|-------------------------------------|
| 1 | linspace(0, 1, 30) | [ones(1, 7) zeros(1,8) ones(1,8) ze |

Algorithm

Design method: Frequency Sampling

▸ Design options

Filter implementation

Structure: Direct-form FIR

Use a System object to implement filter

OK Cancel Help Apply

ne

### Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

#### Impulse response

This dialog only applies if you have the DSP System Toolbox software. 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. Arbitrary group delay designs are only available if **Impulse response** is **IIR**. Without the DSP System Toolbox, the only available arbitrary response filter design is **FIR**.

#### Order mode

This dialog only applies if you have the DSP System Toolbox software. Choose **Minimum** or **Specify**. Choosing **Specify** enables the **Order** dialog.

#### Order

This dialog only applies when **Order mode** is **Specify**. For an **FIR** design, specify the filter order. For an **IIR** design, you can specify an equal order for the numerator and denominator, or you can specify different numerator and denominator orders. The default is equal orders. To specify a different denominator order, check the **Denominator order** box. Because the Signal Processing Toolbox only supports **FIR** arbitrary-magnitude filters, you do not have the option to specify a denominator order.

#### 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 dialog only applies if you have the DSP System Toolbox software and is only available for **FIR** filters. 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 for **Decimator** and 3 for **Sample-rate converter**.

### **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**, **Frequency response**, or **Group delay**. **Amplitudes** is the only option if you do not have the DSP System Toolbox software. **Group delay** is only available for IIR designs.

#### **Frequency units**

Specify frequency units as either **Normalized**, **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 **Frequency units** is set to an option in hertz.

### **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 you select **Amplitudes** in the **Specify response as** drop-down box.

- **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

The options for each design are specific for each design method. In the arbitrary response design, the available options also depend on the **Response specifications**. This section does not present all of the available options for all designs and design methods.

## Design Method

Select the design method for the filter. Different methods are enabled depending on the defining parameters entered in the previous sections.

## Design Options

- **Window** — Valid when the **Design method** is **Frequency Sampling**. 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}`.
- **Density factor** — Valid when the **Design method** is **equiripple**. 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 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

The default changes to 20 for an IIR arbitrary group delay design.

- **Phase constraint** — Valid when the **Design method** is **equiripple**, you have the DSP System Toolbox installed, and **Specify response as** is set to **Amplitudes**. Choose one of **Linear**, **Minimum**, or **Maximum**.
- **Weights** — Uses the weights in **Weights** to weight the error for a single-band design. If you have multiple frequency bands, the **Weights** design option changes



to **B1 Weights**, **B2 Weights** to designate the separate bands. Use **Bi Weights** to specify weights for the *i*-th band. The **Bi Weights** design option is only available when you specify the *i*-th band as an unconstrained.

- **Bi forced frequency point** — This option is only available in a multi-band constrained equiripple design when **Specify response as** is **Amplitudes**. **Bi forced frequency point** is the frequency point in the *i*-th band at which the response is forced to be zero. The index *i* corresponds to the frequency bands in **Band properties**. For example, if you specify two bands in **Band properties**, you have **B1 forced frequency point** and **B2 forced frequency point**.
- **Norm** — Valid only for IIR arbitrary group delay designs. **Norm** is the norm used in the optimization. The default value is 128, which essentially equals the L-infinity norm. The norm must be even.
- **Max pole radius** — Valid only for IIR arbitrary group delay designs. Constrains the maximum pole radius. The default is 0.999999. Reducing the **Max pole radius** can produce a transfer function more resistant to quantization.
- **Init norm** — Valid only for IIR arbitrary group delay designs. The initial norm used in the optimization. The default initial norm is 2.
- **Init numerator** — Specifies an initial estimate of the filter numerator coefficients.
- **Init denominator** — Specifies an initial estimate of the filter denominator coefficients. This may be useful in difficult optimization problems. In allpass filters, you only have to specify either the denominator or numerator coefficients. If you specify the denominator coefficients, you can obtain the numerator coefficients.

## Filter implementation

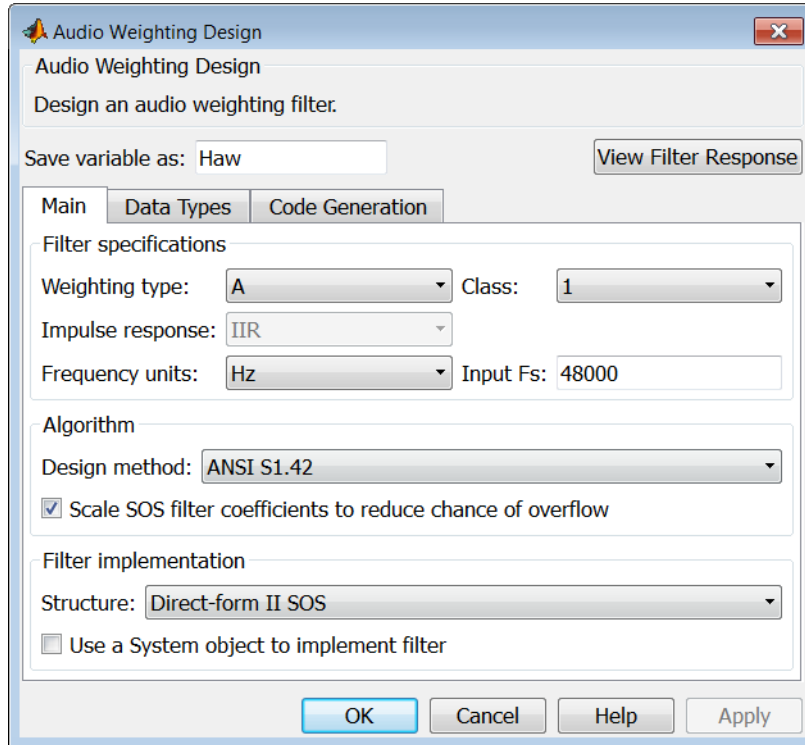
### Structure

Select the structure for the filter. The available filter structures depend on the parameters you select for your filter.

### Use a System object to implement filter

Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

## Audio Weighting Filter Design Dialog Box — Main Pane



### Filter specifications

- **Weighting type** — The weighting type defines the frequency response of the filter. The valid weighting types are: A, C, C-message, ITU-T 0.41, and ITU-R 468–4 weighting. See `fdesign.audioweighting` for definitions of the weighting types.
- **Class** — Filter class is only applicable for A weighting and C weighting filters. The filter class describes the frequency-dependent tolerances specified in the relevant standards. There are two possible class values: 1 and 2. Class 1 weighting filters have stricter tolerances than class 2 filters. The filter class value does not affect the design. The class value is only used to provide a specification mask in `fvtool` for the analysis of the filter design.

- **Impulse response** — Impulse response type as one of IIR or FIR. For A, C, C-message, and ITU-R 468–4 filter, IIR is the only option. For a ITU-T 0.41 weighting filter, FIR is the only option.
- **Frequency units** — Choose Hz, kHz, MHz, or GHz. Normalized frequency designs are not supported for audio weighting filters.
- **Input Fs** — The sampling frequency in **Frequency units**. For example, if **Frequency units** is set to kHz, setting **Input Fs** to 40 is equivalent to a 40 kHz sampling frequency.

### Algorithm

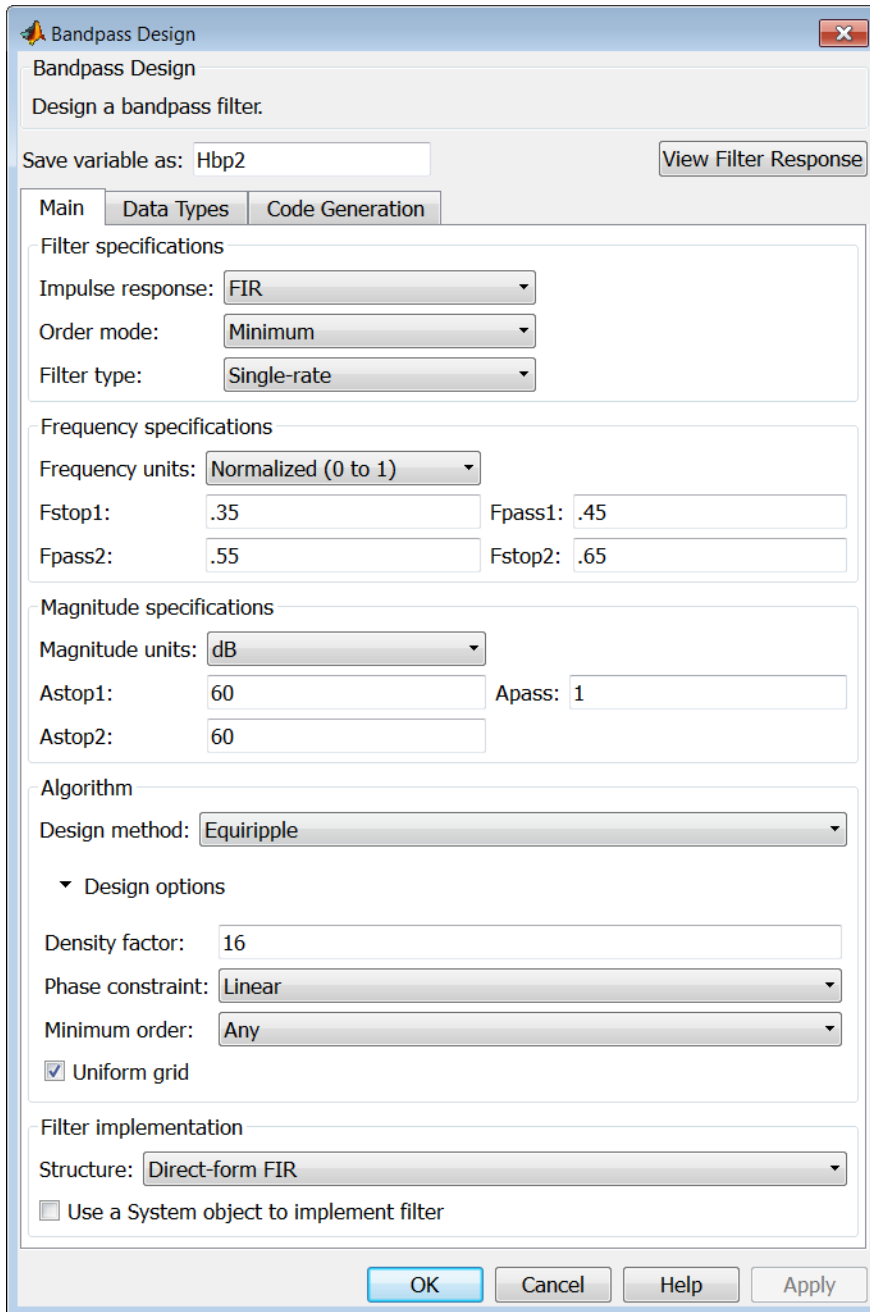
- **Design method** — Valid design methods depend on the weighting type. For type A and C weighting filters, the only valid design type is ANSI S1.42. This is an IIR design method that follows ANSI standard S1.42–2001. For a C message filter, the only valid design method is Bell 41009, which is an IIR design method following the Bell System Technical Reference PUB 41009. For a ITU-R 468–4 weighting filter, you can design an IIR or FIR filter. If you choose an IIR design, the design method is IIR least p-norm. If you choose an FIR design, the design method choices are: Equiripple or Frequency Sampling. For an ITU-T 0.41 weighting filter, the available FIR design methods are equiripple or Frequency Sampling
- **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.

### Filter implementation

- **Structure** — For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. For audio weighting IIR filter designs, you can choose direct form I or II biquad (SOS). You can also choose to implement these structures in transposed form.

For FIR designs, you can choose direct form, direct-form transposed, direct-form symmetric, direct-form asymmetric structures, or an overlap and add structure.

- **Use a System object to implement filter** — Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.



## 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 **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.

---

### Order mode

Select **Minimum** (the default) or **Specify** from the drop-down box. Selecting **Specify** enables the **Order** option so you can enter the filter order.

If you have the DSP System Toolbox software installed, you can specify IIR filters with different numerator and denominator orders. The default is equal orders. To specify a different denominator order, check the **Denominator order** box.

**Filter type** — This dialog only applies if you have the DSP System Toolbox software.

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.

### Order

Enter the filter order. This option is enabled only if you select **Specify** for **Order mode**.

### Decimation Factor

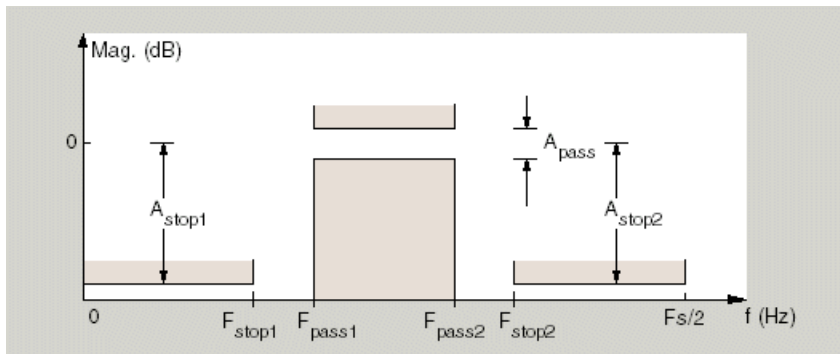
Enter the decimation factor. This option is enabled only if the **Filter type** is set to **Decimator** or **Sample-rate converter**. The default factor value is 2.

### 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  $F_{stop1}$  and  $F_{pass1}$  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. This dialog applies only when **Order mode** is **Specify**.

- **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.
- **3dB points** — Define the filter response by specifying the locations of the 3 dB points (IIR filters). The 3-dB point is the frequency for the point 3 dB below the passband value.
- **3dB 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. (IIR filters)

- **3dB 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. (IIR filters)
- **6dB points** — Define the filter response by specifying the locations of the 6-dB points. The 6-dB point is the frequency for the point 6dB below the passband value. (FIR filters)

### 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 hertz, 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 constraints**

Specify as **Unconstrained** or **Constrained** bands. You must have the DSP System Toolbox software to select **Constrained** bands. Selecting **Constrained** bands enables dialogs for both stopbands and the passband: **Astop1**, **Astop2**, and **Apass**. You cannot specify constraints for all three bands simultaneously.

Setting **Magnitude constraints** to **Constrained** bands enables the **Wstop** and **Wpass** options under **Design options**.

**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.

**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 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

### Phase constraint

Valid when the **Design method** is equiripple and you have the DSP System Toolbox installed. Choose one of **Linear**, **Minimum**, or **Maximum**.

### Minimum order

This option only applies when you have the DSP System Toolbox software and **Order mode** is **Minimum**.

Select **Any** (default), **Even**, or **Odd**. Selecting **Even** or **Odd** forces the minimum-order design to be an even or odd order.

### Wstop1

Weight for the first stopband.

### Wpass

Passband weight.

### Wstop2

Weight for the second stopband.

**Max pole radius**

Valid only for IIR designs. Constrains the maximum pole radius. The default is 1. Reducing the max pole radius can produce a transfer function more resistant to quantization.

**Init norm**

Valid only for IIR designs. The initial norm used in the optimization. The default initial norm is 2.

**Init numerator**

Specifies an initial estimate of the filter numerator coefficients. This may be useful in difficult optimization problems.

**Init denominator**

Specifies an initial estimate of the filter denominator coefficients. This may be useful in difficult optimization problems.

**Filter implementation****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.

**Use a System object to implement filter**

Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

Bandstop Design

Bandstop Design

Design a bandstop filter.

Save variable as: Hbs View Filter Response

Main **Data Types** Code Generation

Filter specifications

Impulse response: FIR

Order mode: Minimum

Filter type: Single-rate

Frequency specifications

Frequency units: Normalized (0 to 1)

Fpass1: .35 Fstop1: .45

Fstop2: .55 Fpass2: .65

Magnitude specifications

Magnitude units: dB

Apass1: 1 Astop: 60

Apass2: 1

Algorithm

Design method: Equiripple

Design options

Density factor: 16

Phase constraint: Linear

Uniform grid

Filter implementation

Structure: Direct-form FIR

Use a System object to implement filter

OK Cancel Help Apply

### 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 **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.

---

#### Order mode

Select **Minimum** (the default) or **Specify** from the drop-down list. Selecting **Specify** enables the **Order** option so you can enter the filter order.

If you have the DSP System Toolbox software installed, you can specify IIR filters with different numerator and denominator orders. The default is equal orders. To specify a different denominator order, check the **Denominator order** box.

#### 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 **Order mode**.

#### Decimation Factor

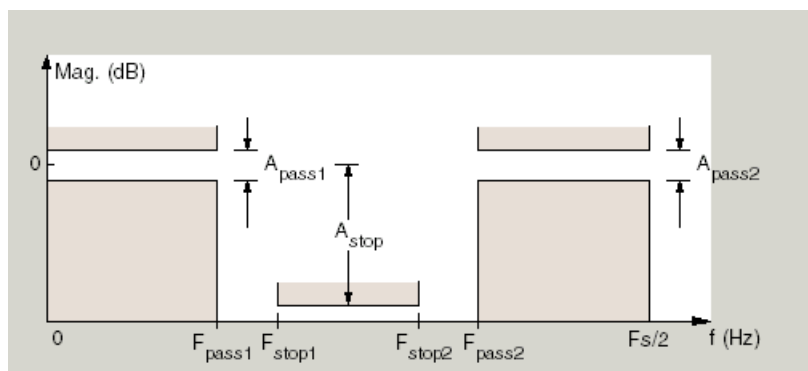
Enter the decimation factor. This option is enabled only if the **Filter type** is set to Decimator or Sample-rate converter. The default factor value is 2.

### 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. This dialog applies only when **Order mode** is **Specify**.

- **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.
- **3dB points** — Define the filter response by specifying the locations of the 3 dB points (IIR filters). The 3 dB point is the frequency for the point 3 dB below the passband value.

- **3dB 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 (IIR filters).
- **3dB 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 (IIR filters).
- **6dB points** — Define the filter response by specifying the locations of the 6-dB points (FIR filters). The 6-dB point is the frequency for the point 6 dB point below the passband value.

## 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 constraints**

Specify as **Unconstrained** or **Constrained** bands. You must have the DSP System Toolbox software to select **Constrained** bands. Selecting **Constrained** bands enables dialogs for both passbands and the stopband: **Apass1**, **Apass2**, and **Astop**. You cannot specify constraints for all three bands simultaneously.

Setting **Magnitude constraints** to **Constrained** bands enables the **Wstop** and **Wpass** options under **Design options**.

**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.

### **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 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

### **Phase constraint**

Valid when the **Design method** is equiripple and you have the DSP System Toolbox installed. Choose one of **Linear**, **Minimum**, or **Maximum**.

### **Minimum order**

This option only applies when you have the DSP System Toolbox software and **Order mode** is **Minimum**.

Select **Any** (default), **Even**, or **Odd**. Selecting **Even** or **Odd** forces the minimum-order design to be an even or odd order.

### **Wpass1**



Weight for the first passband.

**Wstop**

Stopband weight.

**Wpass2**

Weight for the second passband.

**Match exactly**

Specifies that the resulting filter design matches either the passband or stopband or both bands when you select `passband` or `stopband` .

**Max pole radius**

Valid only for IIR designs. Constrains the maximum pole radius. The default is 1. Reducing the max pole radius can produce a transfer function more resistant to quantization.

**Init norm**

Valid only for IIR designs. The initial norm used in the optimization. The default initial norm is 2.

**Init numerator**

Specifies an initial estimate of the filter numerator coefficients. This may be useful in difficult optimization problems.

**Init denominator**

Specifies an initial estimate of the filter denominator coefficients. This may be useful in difficult optimization problems.

**Filter implementation****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.

**Use a System object to implement filter**

Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

## CIC Filter Design Dialog Box — Main Pane

CIC Design

CIC Design

Design a Cascaded Integrator-Comb filter.

Save variable as: Hcic View Filter Response

Main Data Types Code Generation

Filter specifications

Filter type: Decimator Factor: 2

Differential delay: 1

Frequency specifications

Frequency units: Normalized (0 to 1)

Fpass: .01

Magnitude specifications

Magnitude units: dB

Astop: 60

Filter implementation

Use a System object to implement filter

OK Cancel Help Apply

### 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 as an integer value greater than or equal to 1. The default value is 1. The differential delay changes the shape, number, and location of nulls in the filter response. Increasing the differential delay increases the sharpness of the nulls and the response between the nulls. In practice, differential delay values of 1 or 2 are the most common.

### **Factor**

Specify the decimation or interpolation factor for your filter as an integer value greater than or equal to 1. The default value is 2.

### **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**.

### **Magnitude specifications**

#### **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.

### **Filter implementation**

#### **Use a System object to implement filter**

Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

**CIC Compensator Design**

CIC Compensator Design  
Design a CIC compensating filter.

Save variable as:

Main | Data Types | Code Generation

Filter specifications

Order mode:

Filter type:

Number of CIC sections:  Differential delay:

CIC rate change factor:

Frequency specifications

Frequency units:

Fpass:  Fstop:

Magnitude specifications

Magnitude units:

Apass:  Astop:

Algorithm

Design method:

Design options

Density factor:

Phase constraint:

Minimum order:

Stopband shape:

Stopband decay:

Filter implementation

Structure:

Use a System object to implement filter

## Filter specifications

Parameters in this group enable you to specify your filter format, such as the filter order mode and the filter type.

### Order mode

Select **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 **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.

### **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 16 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.

## **Filter implementation**

### **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.

### **Use a System object to implement filter**

Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

## Comb Filter Design Dialog Box—Main Pane

Comb Design

Design a comb filter.

Save variable as:

Main | Data Types | Code Generation

Filter specifications

Comb Type:

Order mode:  Order:

Frequency specifications

Frequency constraints:

Quality factor:

Frequency units:

Notch Frequencies:

Magnitude specifications

No magnitude constraints can be specified when specifying a filter order.

Algorithm

Design method:

Filter implementation

Structure:

Use a System object to implement filter

### Filter specifications

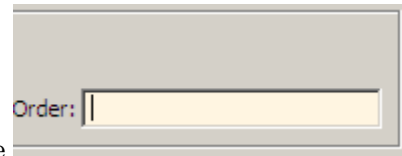
Parameters in this group enable you to specify the type of comb filter and the number of peaks or notches.

#### Comb Type

Select **Notch** or **Peak** from the drop-down list. **Notch** creates a comb filter that attenuates a set of harmonically related frequencies. **Peak** creates a comb filter that amplifies a set of harmonically related frequencies.

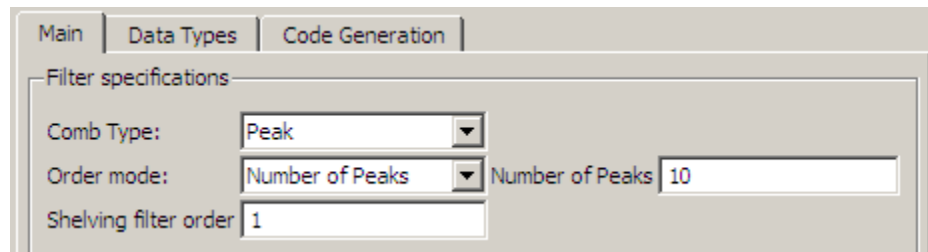
#### Order mode

Select **Order** or **Number of Peaks/Notches** from the drop-down menu.



Select **Order** to enter the desired filter order in the dialog box. The comb filter has notches or peaks at increments of  $2/\text{Order}$  in normalized frequency units.

Select **Number of Peaks** or **Number of Notches** to specify the number of peaks or notches and the **Shelving filter order**



#### Shelving filter order

The **Shelving filter order** is a positive integer that determines the sharpness of the peaks or notches. Larger values result in sharper peaks or notches.

#### Frequency specifications

Parameters in this group enable you to specify the frequency constraints and frequency units.

### Frequency specifications

Select **Quality factor** or **Bandwidth**.

**Quality factor** is the ratio of the center frequency of the peak or notch to the bandwidth calculated at the  $-3$  dB point.

**Bandwidth** specifies the bandwidth of the peak or notch. By default the bandwidth is measured at the  $-3$  dB point. For example, setting the bandwidth equal to 0.1 results in 3 dB frequencies at normalized frequencies 0.05 above and below the center frequency of the peak or notch.

### Frequency Units

Specify the frequency units. The default is normalized frequency. Choosing an option in Hz enables the **Input Fs** dialog box.

### Magnitude specifications

Specify the units for the magnitude specification and the gain at which the bandwidth is measured. This menu is disabled if you specify a filter order. Select one of the following magnitude units from the drop down list:

- **dB** — Specify the magnitude in decibels (default).
- **Squared** — Specify the magnitude in squared units.

**Bandwidth gain** — Specify the gain at which the bandwidth is measured. The default is  $-3$  dB.

### Algorithm

The parameters in this group allow you to specify the design method and structure that `filterbuilder` uses to implement your filter.

#### Design Method

The IIR Butterworth design is the only option for peaking or notching comb filters.

#### Filter implementation

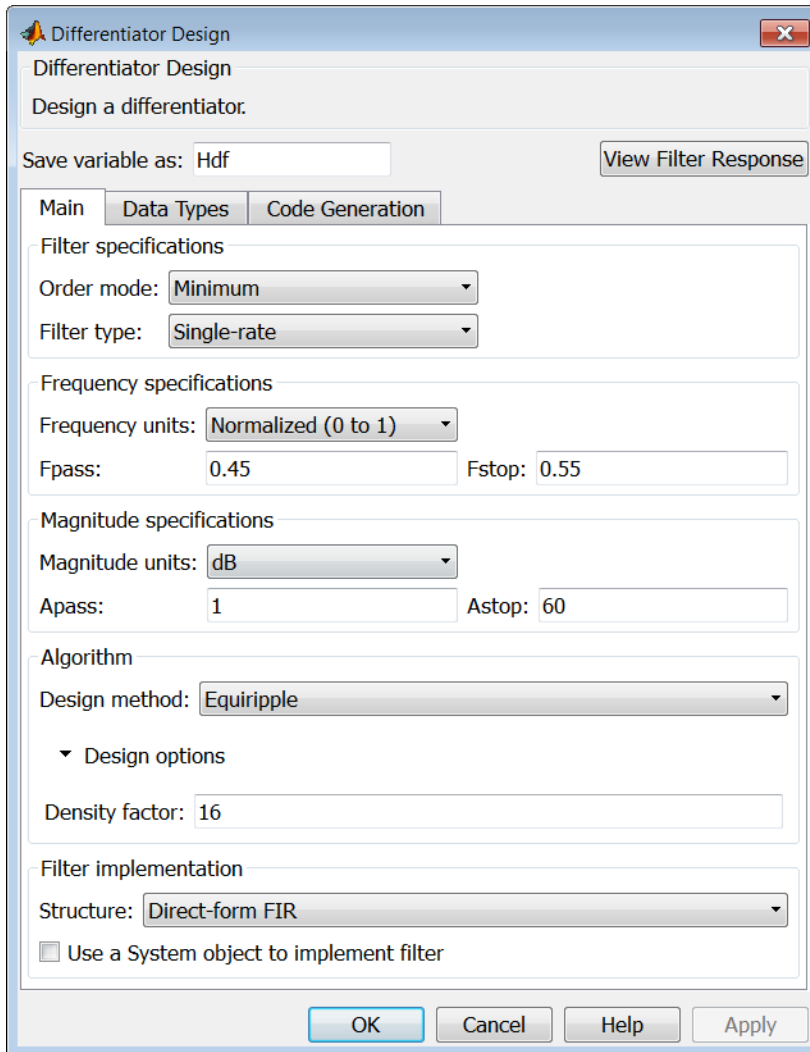
##### Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter.

### Use a System object to implement filter

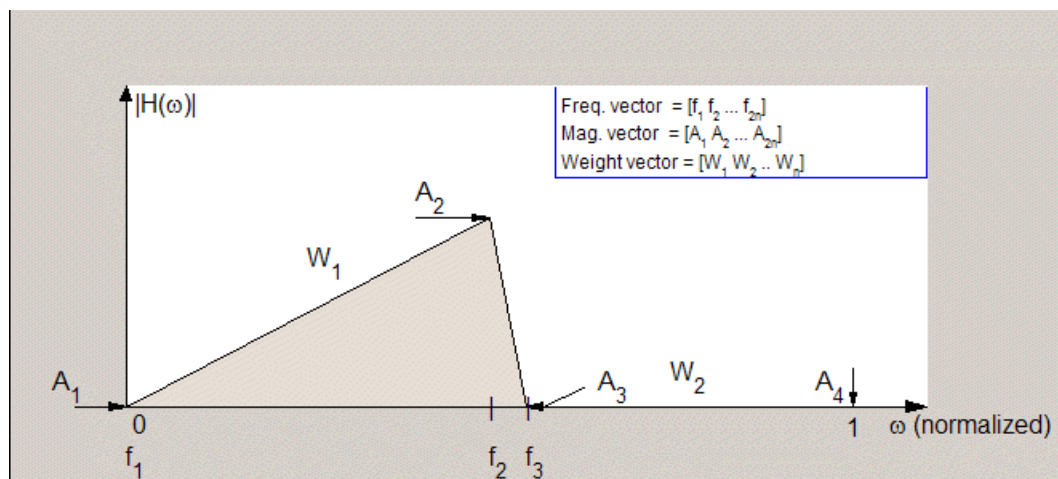
Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off.

## 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.

## Order mode

Select **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 **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 constraints**

This option is only available when you specify the order of the filter design. Supported options are **Unconstrained** and **Passband edge and stopband edge**.

#### **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 constraints

This option is only available when you specify the order of your filter design. The options for **Magnitude constraints** depend on the value of the **Frequency constraints**. If the value of **Frequency constraints** is **Unconstrained**, **Magnitude constraints** must be **Unconstrained**. If the value of **Frequency constraints** is **Passband edge and stopband edge**, **Magnitude constraints** can be **Unconstrained**, **Passband ripple**, or **Stopband attenuation**.

### 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.

### A<sub>pass</sub>

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

### A<sub>stop2</sub>

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.

### 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.

### 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 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

### Wpass

Passband weight. This option is only available for a specified-order design when **Frequency constraints** is equal to Passband edge and stopband edge and the **Design method** is Equiripple.

### Wstop

Stopband weight. This option is only available for a specified-order design when **Frequency constraints** is equal to Passband edge and stopband edge and the **Design method** is Equiripple.

### Filter implementation

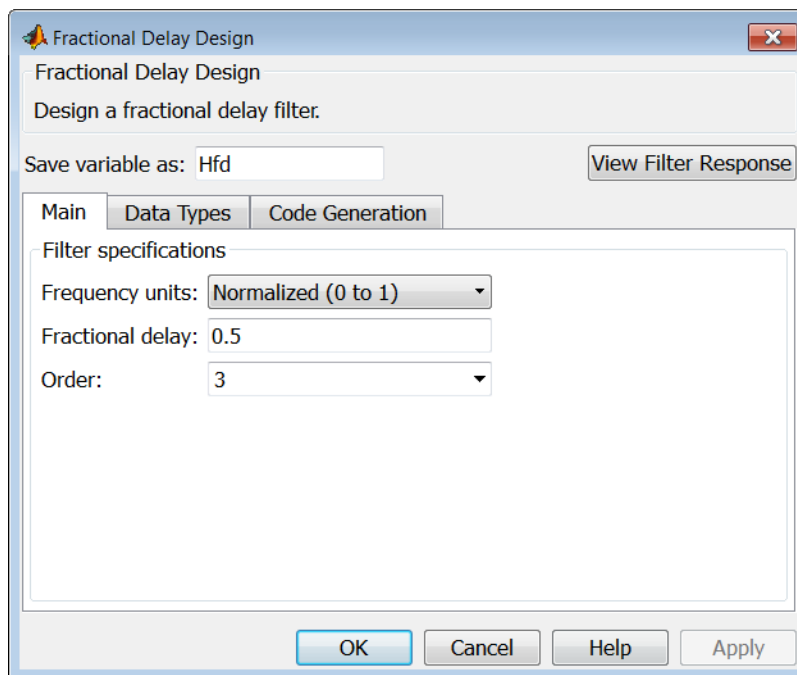
#### Structure

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.

#### Use a System object to implement filter

Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

## 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 **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

Halfband Design

Halfband Design  
Design a Halfband filter.

Save variable as:

Main | Data Types | Code Generation

Frequency specifications

Impulse response:

Order mode:

Response type:

Filter type:

Frequency specifications

Frequency units:

Transition width:

Magnitude specifications

Magnitude units:

Astop:

Algorithm

Design method:

▼ Design options

Minimum phase

Stopband shape:

Stopband decay:

Filter implementation

Structure:

Use a System object to implement filter

### Filter specifications

Parameters in this group enable you to specify your filter type and order.

### Impulse response

Select **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.

---

### Order mode

Select **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**, or **Interpolator**. By default, `filterbuilder` specifies single-rate filters.

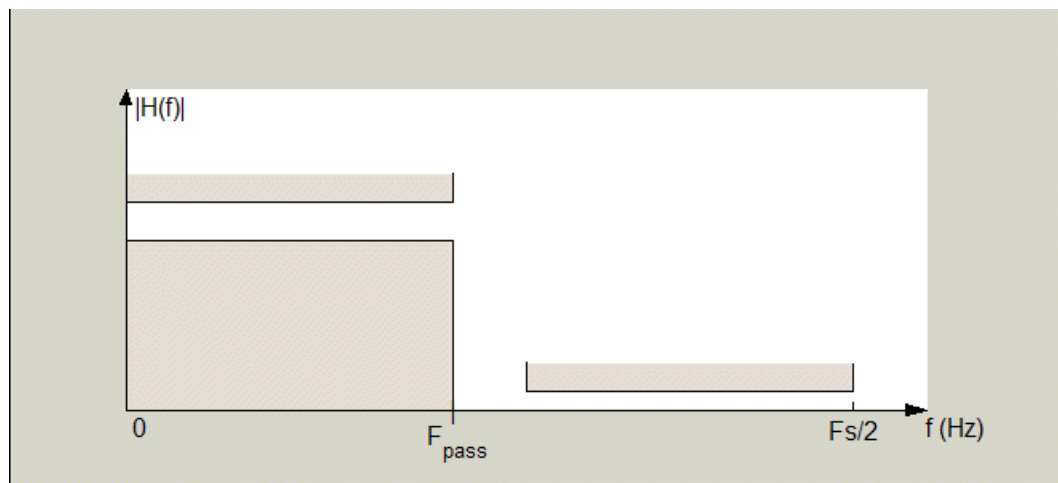
When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that decimates or interpolates your input by a factor of two.

### Order

Enter the filter order. This option is enabled only if **Specify** was selected for **Order mode**.

### 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).

### 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. For FIR halfband filters, the available design options are **Equiripple** and **Kaiser window**. For IIR halfband filters, the available design options are **Butterworth**, **Elliptic**, and **IIR quasi-linear phase**.

### Design Options

The following design options are available for FIR halfband filters when the user specifies an equiripple design:

#### 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.

#### 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.

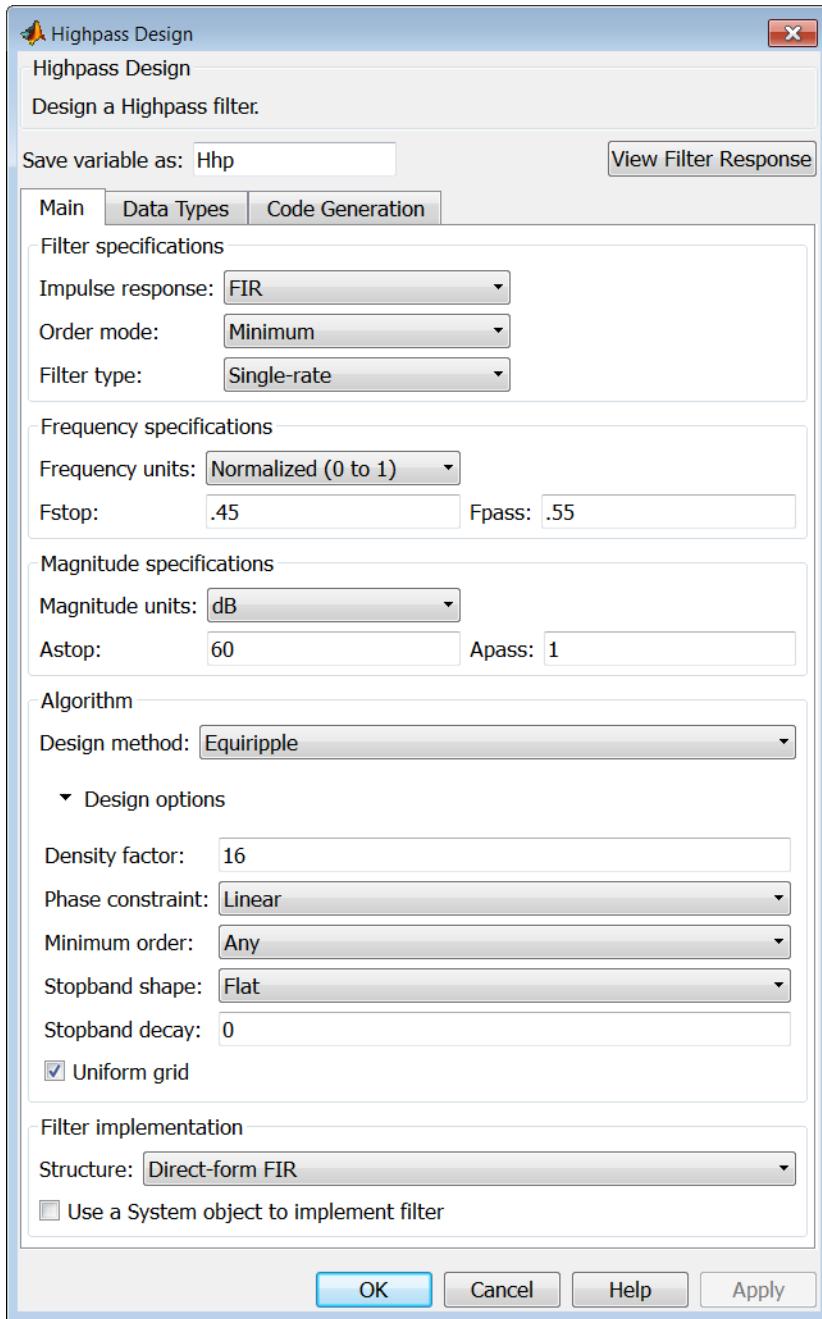
### Filter implementation

#### Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter.

#### Use a System object to implement filter

Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.



### 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 **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.

---

#### Order mode

Select **Minimum** (the default) or **Specify** from the drop-down list. Selecting **Specify** enables the **Order** option so you can enter the filter order.

If your **Impulse response** is **IIR**, you can specify an equal order for the numerator and denominator, or different numerator and denominator orders. The default is equal orders. To specify a different denominator order, check the **Denominator order** box.

#### Filter type

This option is only available if you have the DSP System Toolbox software. 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 highpass filter that either decimates or interpolates your input signal.

#### Order

Enter the filter order. This option is enabled only if **Specify** was selected for **Order mode**.

#### Decimation Factor

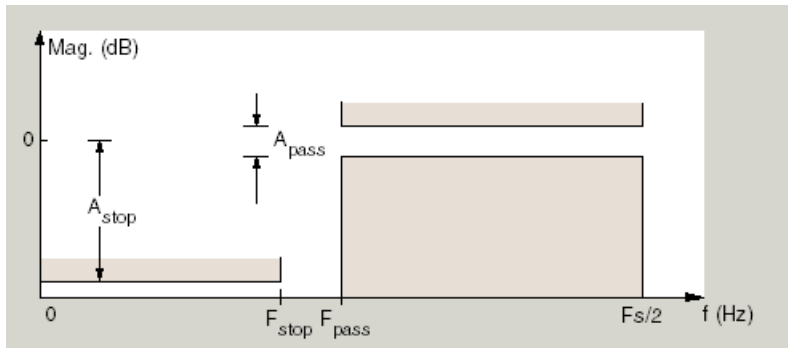
Enter the decimation factor. This option is enabled only if the **Filter type** is set to **Decimator** or **Sample-rate converter**. The default factor value is 2.

### Interpolation Factor

Enter the interpolation 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  $F_{stop}$  and  $F_{pass}$  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.

- **Stopband edge and passband edge** — Define the filter by specifying the frequencies for the edges for the stopband and passband.
- **Passband edge** — Define the filter by specifying the frequency for the edge of the passband.
- **Stopband edge** — Define the filter by specifying the frequency for the edges of the stopband.
- **Stopband edge and 3dB point** — Define the filter by specifying the stopband edge frequency and the 3-dB down point (IIR designs).
- **3dB point and passband edge** — Define the filter by specifying the 3-dB down point and passband edge frequency (IIR designs).

- **3dB point** — Define the filter by specifying the frequency for the 3-dB point (IIR designs or maxflat FIR).
- **6dB point** — Define the filter by specifying the frequency for the 6-dB point in the filter response (FIR designs).

### 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.

### 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.

### **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 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

### **Phase constraint**

This option only applies when you have the DSP System Toolbox software and when the **Design method** is `equiripple`. Select one of `Linear`, `Minimum`, or `Maximum`.

**Minimum order** — This option only applies when you have the DSP System Toolbox software and the **Order mode** is `Minimum`.

Select `Any` (default), `Even`, or `Odd`. Selecting `Even` or `Odd` forces the minimum-order design to be an even or odd order.

### Match Exactly

Specifies that the resulting filter design matches either the passband or stopband when you select `Passband` or `Stopband`.

### 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 effect 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.

### Wpass

Passband weight. This option only applies when **Impulse response** is FIR and **Order mode** is Specify.

**Wstop**

Stopband weight. This option only applies when **Impulse response** is FIR and **Order mode** is Specify.

**Filter implementation**

**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.

**Use a System object to implement filter**

Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.



## Hilbert Filter Design Dialog Box — Main Pane

Hilbert Design

Design a Hilbert filter.

Save variable as:

Main

Filter specifications

Impulse response:

Order mode:

Filter type:

Frequency specifications

Frequency units:

Transition width

Magnitude specifications

Magnitude units:

Apass:

Algorithm

Design method:

▼ Design options

Density factor:

FIR Type:

Filter implementation

Structure:

Use a System object to implement filter

### 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 **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.

---

#### Order mode

This option is only available if you have the DSP System Toolbox software. 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

This option is only available if you have the DSP System Toolbox software. 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 **Order mode**.

#### Decimation Factor

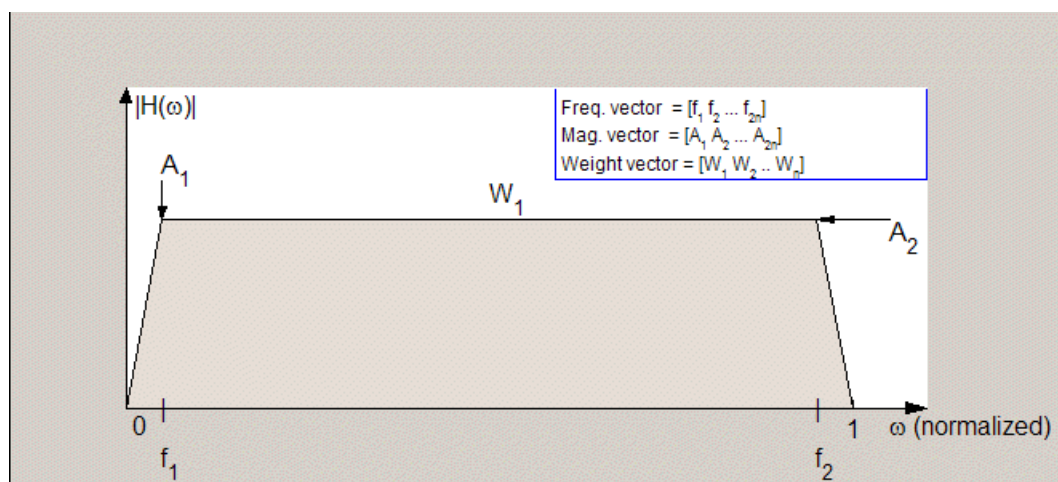
Enter the decimation factor. This option is enabled only if the **Filter type** is set to **Decimator** or **Sample-rate converter**. The default factor value is 2.

## 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.

### **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 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

### FIR Type

This option is only available in a minimum-order design. 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 3 or 4 from the drop-down list.

## Filter implementation

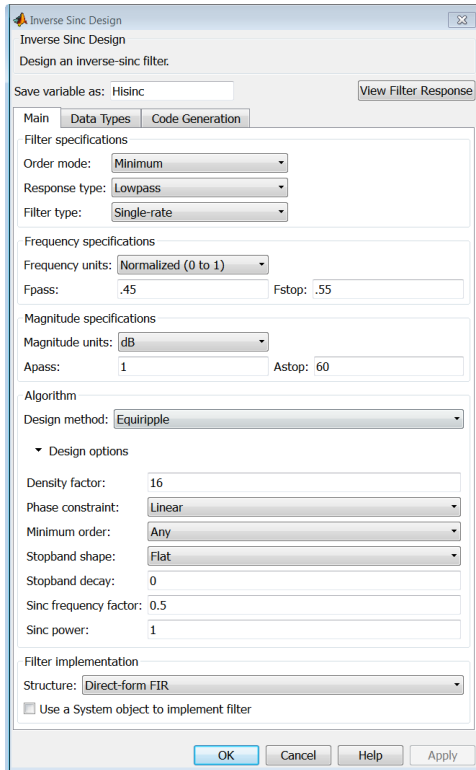
### Structure

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.

### Use a System object to implement filter

Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

## 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.

### Order mode

Select **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.

### Response type

Select **Lowpass** or **Highpass** to design an inverse sinc lowpass or highpass filter.

### 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 **Order mode**.

### **Decimation Factor**

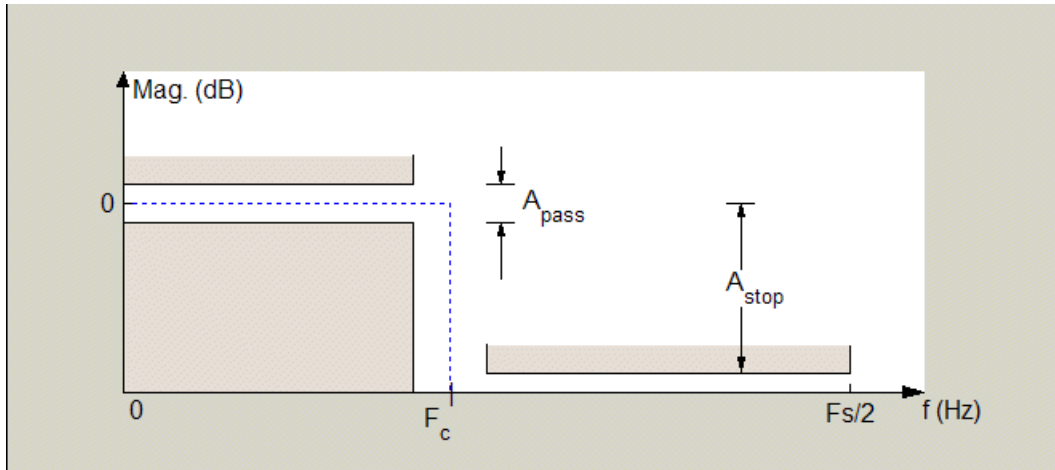
Enter the decimation factor. This option is enabled only if the **Filter type** is set to **Decimator** or **Sample-rate converter**. The default factor value is 2.

### **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  $F_{\text{pass}}$  and  $F_{\text{stop}}$  represent transition regions where the filter response is not explicitly defined.

### Frequency constraints

This option is only available when you specify the filter order. The following options are available:

- **Passband and stopband edges** — Define the filter by specifying the frequencies for the edges for the stop- and passbands.
- **Passband edge** — Define the filter by specifying frequencies for the edges of the passband.
- **Stopband edge** — Define the filter by specifying frequencies for the edges of the stopbands.
- **6dB point** — The 6-dB point is the frequency for the point 6 dB below the passband value.

### 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



$F_s$ , 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.

## 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 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

### Phase constraint

Available options are **Linear**, **Minimum**, and **Maximum**.

### 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 effect 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.

### Sinc frequency factor

A frequency dilation factor. The sinc frequency factor,  $C$ , parameterizes the passband magnitude response for a lowpass design through  $H(\omega) = \text{sinc}(C\omega)^{-P}$  and for a highpass design through  $H(\omega) = \text{sinc}(C(1-\omega))^{-P}$ .

### Sinc power

Negative power of passband magnitude response. The sinc power,  $P$ , parameterizes the passband magnitude response for a lowpass design through  $H(\omega) = \text{sinc}(C\omega)^{-P}$  and for a highpass design through  $H(\omega) = \text{sinc}(C(1-\omega))^{-P}$ .

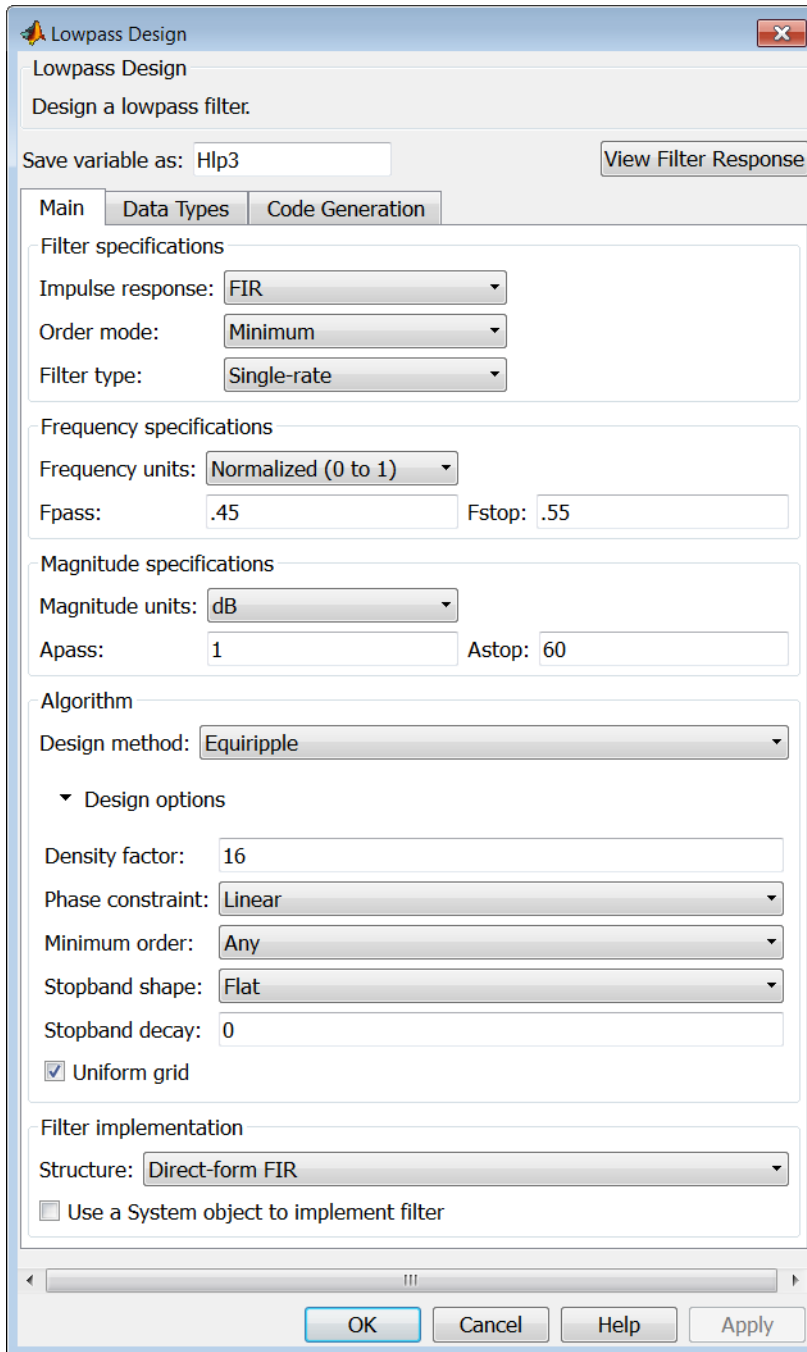
### Filter implementation

#### 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.

#### Use a System object to implement filter

Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.



## 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 **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.

---

### Order mode

Select **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.

If your **Impulse response** is **IIR**, you can specify an equal order for the numerator and denominator, or different numerator and denominator orders. The default is equal orders. To specify a different denominator order, check the **Denominator order** box.

### Filter type

This option is only available if you have the DSP System Toolbox. 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 **Order mode**.

### Decimation Factor

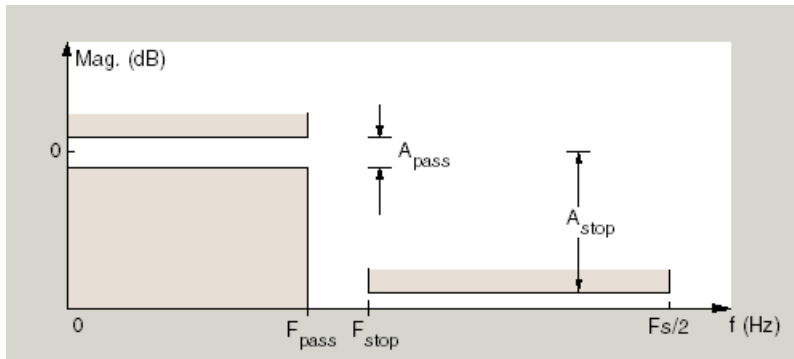
Enter the decimation factor. This option is enabled only if the **Filter type** is set to Decimator or Sample-rate converter. The default factor value is 2.

### 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  $F_{\text{pass}}$  and  $F_{\text{stop}}$  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 edge** — Define the filter by specifying the frequencies for the edge of the stopband and passband.
- **Passband edge** — Define the filter by specifying the frequency for the edge of the passband.
- **Stopband edge** — Define the filter by specifying the frequency for the edges of the stopband.

- **Passband edge and 3dB point** — Define the filter by specifying the passband edge frequency and the 3-dB down point (IIR designs).
- **3dB point and stopband edge** — Define the filter by specifying the 3-dB down point and stopband edge frequency (IIR designs).
- **3dB point** — Define the filter by specifying the frequency for the 3-dB point (IIR designs or maxflat FIR).
- **6dB point** — Define the filter by specifying the frequency for the 6-dB point in the filter response (FIR designs).

### 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.

## **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 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

### Phase constraint

This option only applies when you have the DSP System Toolbox software and when the **Design method** is `equiripple`. Select one of `Linear`, `Minimum`, or `Maximum`.

**Minimum order** — This option only applies when you have the DSP System Toolbox software and the **Order mode** is `Minimum`.

Select `Any` (default), `Even`, or `Odd`. Selecting `Even` or `Odd` forces the minimum-order design to be an even or odd order.

### Match Exactly

Specifies that the resulting filter design matches either the passband or stopband when you select `Passband` or `Stopband`.

### 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 effect 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.

### **Wpass**

Passband weight. This option only applies when **Impulse response** is **FIR** and **Order mode** is **Specify**.

### **Wstop**

Stopband weight. This option only applies when **Impulse response** is **FIR** and **Order mode** is **Specify**.

## **Filter implementation**

### **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.

### **Use a System object to implement filter**

Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

## **Notch**

See “Peak/Notch Filter Design Dialog Box — Main Pane” on page 1-587.

## Nyquist Filter Design Dialog Box — Main Pane

Nyquist Design

Nyquist Design

Design a Nyquist filter.

Save variable as: Hnyq View Filter Response

Main **Data Types** Code Generation

Filter specifications

Band: 2

Impulse response: FIR

Filter order mode: Minimum

Filter type: Single-rate

Frequency specifications

Frequency units: Normalized (0 to 1)

Transition width: .1

Magnitude specifications

Magnitude units: dB

Astop: 80

Algorithm

Design method: Kaiser window

Filter implementation

Structure: Direct-form FIR

Use a System object to implement filter

OK Cancel Help Apply

### 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 `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.

---

**Order mode**

Select `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 **Order mode**.

**Decimation Factor**

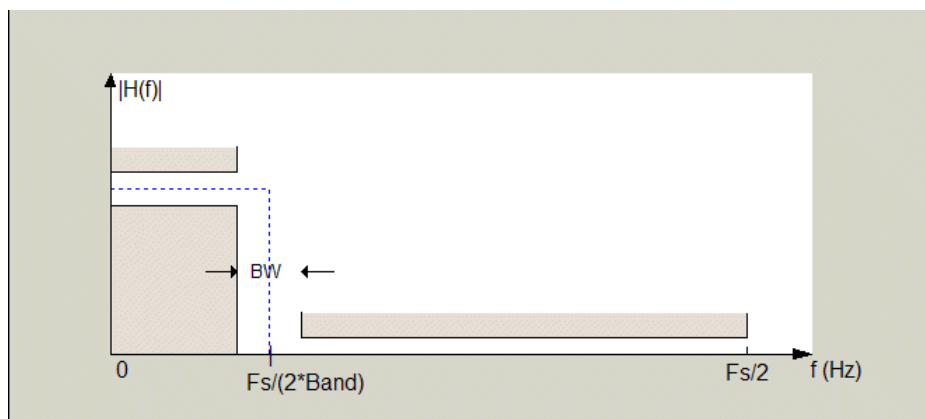
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

### 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 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.

### 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 16 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 designs 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.

### **Filter implementation**

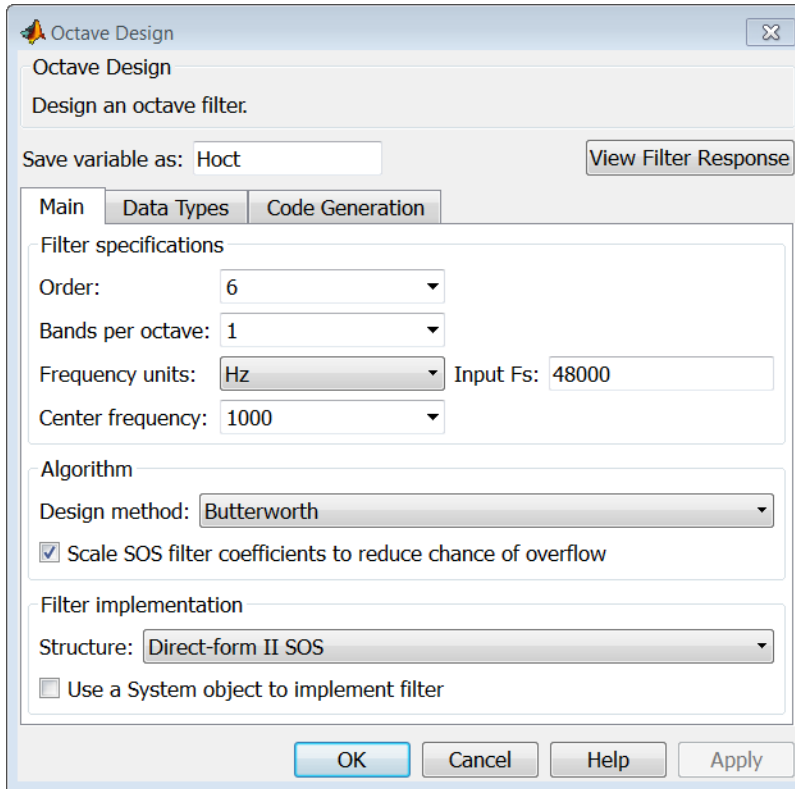
#### **Structure**

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.

#### **Use a System object to implement filter**

Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

## 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.

**Scale SOS filter coefficients to reduce chance of overflow**

Select the check box to scale the filter coefficients.

**Filter implementation****Structure**

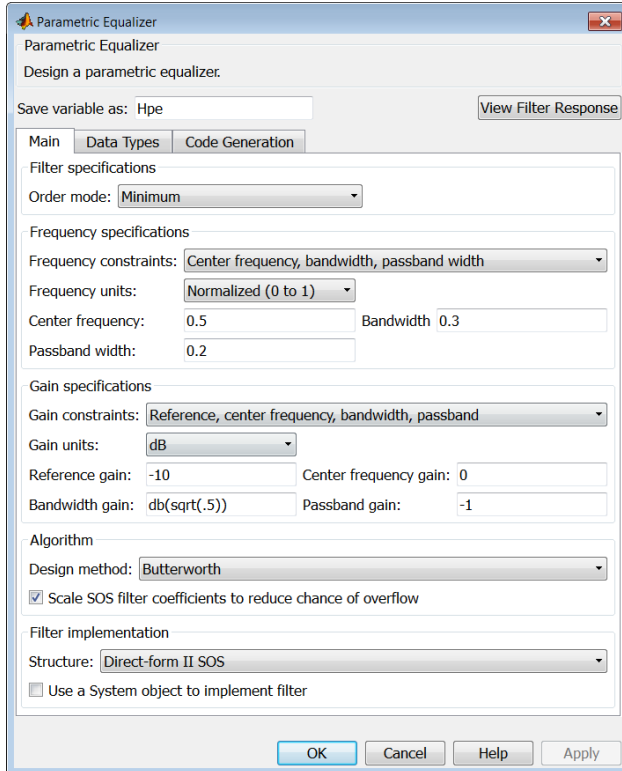
Specify filter structure. Choose from:

- Direct-form I SOS
- Direct-form II SOS
- Direct-form I transposed SOS
- Direct-form II transposed SOS

**Use a System object to implement filter**

Selecting this check box gives you the choice of using a system object to implement the filter. By default, the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

## Parametric Equalizer Filter Design Dialog Box — Main Pane



### Filter specifications

#### Order mode

Select **Minimum** to design a minimum order filter that meets the design specifications, 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, the available frequency constraints are:

- Center frequency, bandwidth, passband width
- Center frequency, bandwidth, stopband width

If you select **Specify**, the available frequency constraints are:

- Center frequency, bandwidth
- Center frequency, quality factor
- Shelf type, cutoff frequency, quality factor
- Shelf type, cutoff frequency, shelf slope parameter
- 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. Once you choose the frequency constraints, the input boxes in this area change to reflect the selection.

### **Frequency constraints**

Select the specification to represent the 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 specified order only)
- Center frequency, quality factor (available for a specified order only)
- Shelf type, cutoff frequency, quality factor (available for a specified order only)
- Shelf type, cutoff frequency, shelf slope parameter (available for a specified order only)
- Low frequency, high frequency (available for a 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 in the units specified by the value in **Frequency units**.

### **Bandwidth**

The bandwidth determines the frequency points at which the filter magnitude is attenuated by the value specified as the **Bandwidth gain** in the **Gain specifications** section. By default, the **Bandwidth gain** defaults to  $\text{db}(\text{sqrt}(.5))$ , or  $-3$  dB relative to the center frequency. The **Bandwidth** property only applies when the **Frequency constraints** are: **Center frequency, bandwidth, passband width, Center frequency, bandwidth, stopband width, or Center frequency, bandwidth.**

### **Passband width**

The passband width determines the frequency points at which the filter magnitude is attenuated by the value specified as the **Passband gain** in the **Gain specifications** section. 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**

The stopband width determines the frequency points at which the filter magnitude is attenuated by the value specified as the **Stopband gain** in the **Gain specifications** section. 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 cutoff. This option is enabled only if the filter order is user specified and the frequency constraint selected is **Low frequency, high frequency.** The filter magnitude is attenuated by the amount specified in **Bandwidth gain.**

### **High frequency**

Enter the high frequency cutoff. This option is enabled only if the filter order is user specified and the frequency constraint selected is **Low frequency, high frequency.** The filter magnitude is attenuated by the amount specified in **Bandwidth gain.**

## 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 gain

The reference gain determines the level to which the filter magnitude attenuates in **Gain units**. The reference gain is a *floor* gain for the filter magnitude response. For example, you may use the reference gain together with the **Center frequency gain** to leave certain frequencies unattenuated (reference gain of 0 dB) while boosting other frequencies.

## Bandwidth gain

Specifies the gain in **Gain units** at which the bandwidth is defined. This property applies only when the **Frequency constraints** specification contains a **bandwidth** parameter, or is **Low frequency**, **high frequency**.

## Center frequency gain

Specify the center frequency in **Gain units**

## Passband gain

The passband gain determines the level in **Gain units** at which the passband is defined. The passband is determined either by the **Passband width** value, or the **Low frequency** and **High frequency** values in the **Frequency specifications** section.

### Stopband gain

The stopband gain is the level in **Gain units** at which the stopband is defined. This property applies only when the **Order mode** is minimum and the **Frequency constraints** are Center frequency, bandwidth, stopband width.

### Boost/cut gain

The boost/cut gain applies only when the designing a shelving filter. Shelving filters include the **Shelf type** parameter in the **Frequency constraints** specification. The gain in the passband of the shelving filter is increased by **Boost/cut gain** dB from a *floor* gain of 0 dB.

### Algorithm

#### Design method

Select the design method from the drop-down list. Different IIR design methods are available depending on the filter constraints you specify.

#### Scale SOS filter coefficients to reduce chance of overflow

Select the check box to scale the filter coefficients.

### Filter implementation

#### 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

#### Use a System object to implement filter

Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.



## Peak/Notch Filter Design Dialog Box — Main Pane

Peak/Notch Design

Peak/Notch Design

Design a peak or notch filter.

Save variable as: Hpn View Filter Response

Main Data Types Code Generation

Filter specifications

Response: Peak Order: 6

Frequency specifications

Frequency constraints: Center frequency and quality factor

Frequency units: Normalized (0 to 1)

Center frequency: 0.5 Quality factor 2.5

Magnitude specifications

Magnitude constraints: Unconstrained

Algorithm

Design method: Butterworth

Scale SOS filter coefficients to reduce chance of overflow

Filter implementation

Structure: Direct-form II SOS

Use a System object to implement filter

OK Cancel Help Apply

### 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.

## **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 you specified in **Frequency units**.

### **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.

### **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.

### **Filter implementation**

#### **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

### **Use a System object to implement filter**

Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

## Pulse-shaping Filter Design Dialog Box—Main Pane

Pulse-shaping Design

Pulse-shaping Design  
Design a pulse-shaping filter.

Save variable as: Hps View Filter Response

Main Data Types Code Generation

Filter specifications

Pulse shape: Raised Cosine

Order mode: Minimum

Samples per symbol: 8

Filter type: Single-rate

Frequency specifications

Rolloff factor: .5

Frequency units: Normalized (0 to 1)

Magnitude specifications

Magnitude units: dB

Astop: 50

Algorithm

Design method: Window

Filter implementation

Structure: Direct-form FIR

Use a System object to implement filter

OK Cancel Help Apply

### 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**.

### **Filter Type**

This option is only available if you have the DSP System Toolbox software. Choose **Single rate**, **Decimator**, **Interpolator**, or **Sample-rate converter**. If you select **Decimator** or **Interpolator**, the decimation and interpolation factors default to the value of the **Samples per symbol**. If you select **Sample-rate converter**, the interpolation factor defaults to **Samples per symbol** and the decimation factor defaults to 3.

## 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.

## Filter implementation

### 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.

**Use a System object to implement filter**

Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.



# filternorm

2-norm or infinity-norm of digital filter

## Syntax

```
L = filternorm(b,a)
L = filternorm(b,a,pnorm)
L = 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.

`L = filternorm(b,a)` computes the 2-norm of the digital filter defined by the numerator coefficients in `b` and denominator coefficients in `a`.

`L = filternorm(b,a,pnorm)` computes the 2- or infinity-norm (inf-norm) of the digital filter, where `pnorm` is either `2` or `inf`.

`L = 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  $10^{-8}$ .

## Examples

### Filter Norms

Compute the 2-norm of a Butterworth IIR filter with tolerance  $10^{-10}$ . Specify a normalized cutoff frequency of  $0.5\pi$  rad/s and a filter order of 5.

```
[b,a] = butter(5,0.5);
L2 = filternorm(b,a,2,1e-10)
```

L2 =

0.7071

Compute the infinity-norm of an FIR Hilbert transformer of order 30 and normalized transition width  $0.2\pi$  rad/s.

```
b = firpm(30,[.1 .9],[1 1],'Hilbert');
Linf = filternorm(b,1,inf)
```

Linf =

1.0028

## More About

### Algorithms

Given a filter with frequency response  $H(e^{j\omega})$ , the  $L_p$ -norm for  $1 \leq p < \infty$  is given by

$$\|H(e^{j\omega})\|_p = \left( \frac{1}{2\pi} \int_{-\pi}^{\pi} |H(e^{j\omega})|^p d\omega \right)^{1/p}.$$

For the case  $p \rightarrow \infty$ , the  $L_\infty$ -norm is

$$\|H(e^{j\omega})\|_\infty = \max_{-\pi \leq \omega \leq \pi} |H(e^{j\omega})|.$$

For the case  $p = 2$ , Parseval's theorem states that

$$\|H(e^{j\omega})\|_2 = \left( \frac{1}{2\pi} \int_{-\pi}^{\pi} |H(e^{j\omega})|^2 d\omega \right)^{1/2} = \left( \sum_n |h(n)|^2 \right)^{1/2},$$

where  $h(n)$  is the impulse response of the filter. The energy of the impulse response is the squared  $L_2$ -norm.

## References

- [1] Jackson, L. B. *Digital Filters and Signal Processing: with MATLAB Exercises*. 3rd Ed. Hingham, MA: Kluwer Academic Publishers, 1996, Chapter 11.

## See Also

zp2sos | norm

## **filtfilt**

Zero-phase digital filtering

### **Syntax**

```
y = filtfilt(b,a,x)
y = filtfilt(SOS,G,x)
y = filtfilt(d,x)
```

### **Description**

`y = filtfilt(b,a,x)` performs zero-phase digital filtering by processing the input data, `x`, in both the forward and reverse directions [1]. `filtfilt` operates along the first nonsingleton dimension of `x`. The vector `b` provides the numerator coefficients of the filter and the vector `a` provides the denominator coefficients. If you use an all-pole filter, enter 1 for `b`. If you use an all-zero filter (FIR), enter 1 for `a`. After filtering the data in the forward direction, `filtfilt` reverses the filtered sequence and runs it back through the filter. The result has the following characteristics:

- Zero-phase distortion
- A filter transfer function, which equals the squared magnitude of the original filter transfer function
- A 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 you can use it for both real and complex inputs. Do not use `filtfilt` with differentiator and Hilbert FIR filters, because 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, defined as  $\max(\text{length}(b) - 1, \text{length}(a) - 1)$ .

---

`y = filtfilt(SOS,G,x)` zero-phase filters the data, `x`, using the second-order section (biquad) filter represented by the matrix `SOS` and scale values `G`. `SOS` is an  $L$ -by-6 matrix containing the  $L$  second-order sections. `SOS` must be of the form

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

where each row are the coefficients of a biquad filter. The vector of filter scale values,  $\mathbf{G}$ , must have a length between 1 and  $L + 1$ .

---

**Note:** When implementing zero-phase filtering using a second-order section filter, the length of the input,  $\mathbf{x}$ , must be more than three times the filter order. You can use `filtord` to obtain the order of the filter.

---

`y = filtfilt(d,x)` zero-phase filters the input data,  $\mathbf{x}$ , using a digital filter,  $\mathbf{d}$ . Use `designfilt` to generate  $\mathbf{d}$  based on frequency-response specifications.

## Examples

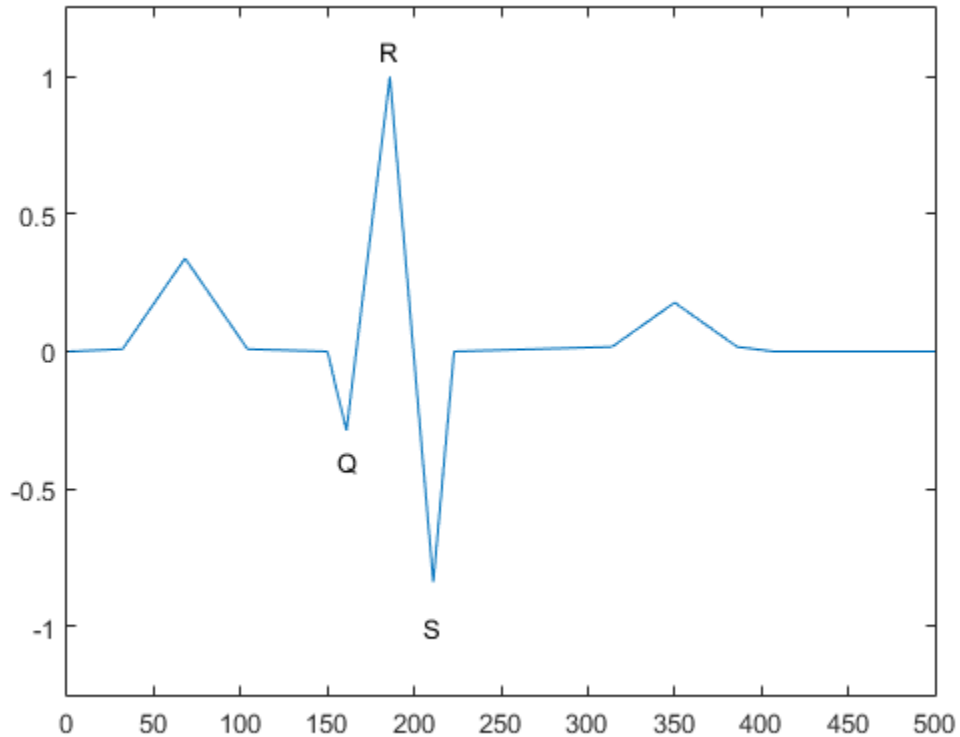
### Zero-Phase Filtering of an Electrocardiogram Waveform

Zero-phase filtering helps preserve features in a filtered time waveform exactly where they occur in the unfiltered signal.

To illustrate the use of `filtfilt` for zero-phase filtering, consider an electrocardiogram waveform.

```
wform = ecg(500);

plot(wform)
axis([0 500 -1.25 1.25])
text(155,-0.4,'Q')
text(180,1.1,'R')
text(205,-1,'S')
```



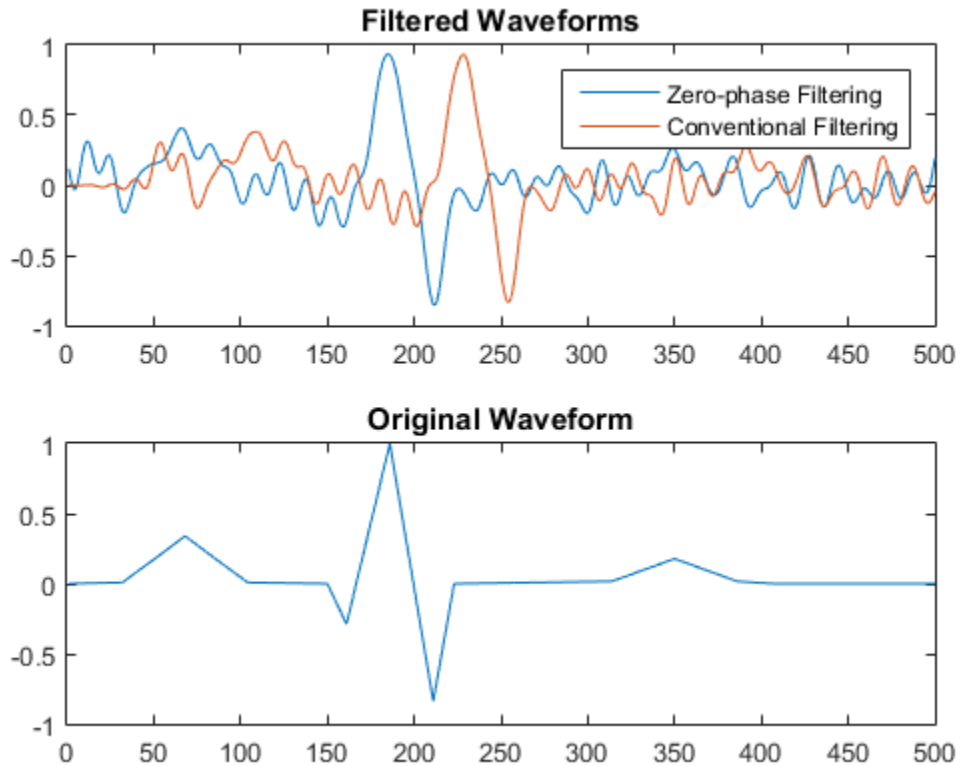
The QRS complex is an important feature in the ECG. Here it begins around time point 160.

Corrupt the ECG with additive noise. Reset the random number generator for reproducible results. Construct a lowpass FIR equiripple filter and filter the noisy waveform using both zero-phase and conventional filtering.

```
rng default

x = wform' + 0.25*randn(500,1);
d = designfilt('lowpassfir', ...
    'PassbandFrequency',0.15,'StopbandFrequency',0.2, ...
    'PassbandRipple',1,'StopbandAttenuation',60, ...
    'DesignMethod','equiripple');
```

```
y = filtfilt(d,x);  
y1 = filter(d,x);  
  
subplot(2,1,1)  
plot([y y1])  
title('Filtered Waveforms')  
legend('Zero-phase Filtering', 'Conventional Filtering')  
  
subplot(2,1,2)  
plot(wform)  
title('Original Waveform')
```



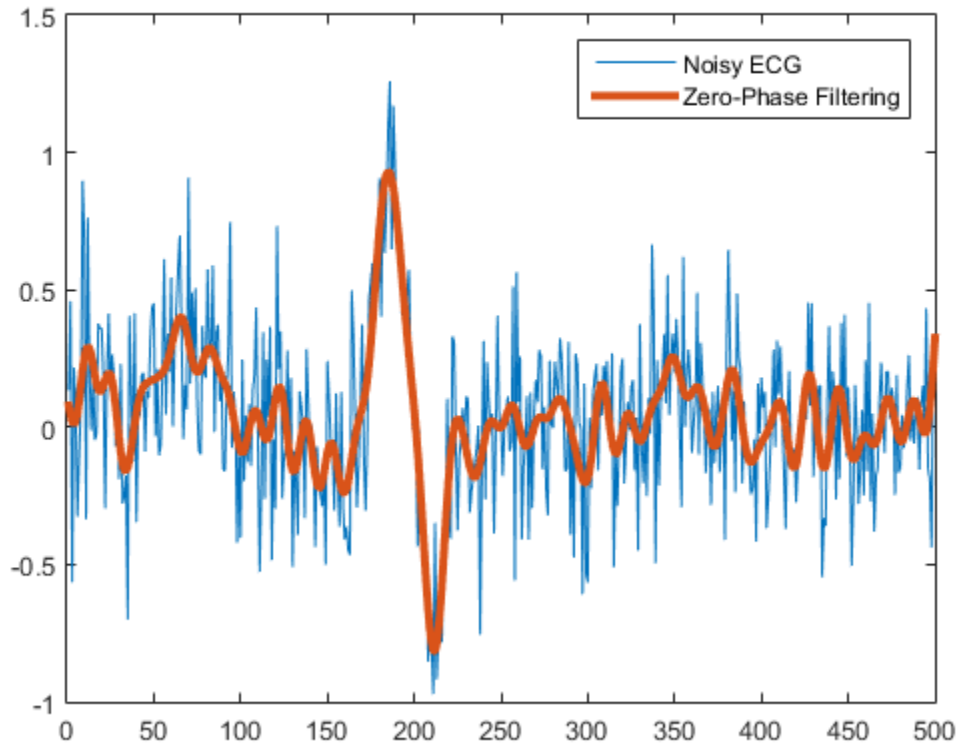
Zero-phase filtering reduces noise in the signal and preserves the QRS complex at the same time it occurs in the original. Conventional filtering reduces noise in the signal, but delays the QRS complex.

Repeat the above using a Butterworth second-order section filter.

```
d1 = designfilt('lowpassiir','FilterOrder',12, ...
    'HalfPowerFrequency',0.15,'DesignMethod','butter');
y = filtfilt(d1,x);

subplot(1,1,1)
plot(x)
hold on
plot(y,'LineWidth',3)
legend('Noisy ECG','Zero-Phase Filtering')
```





## References

- [1] Oppenheim, Alan V., Ronald W. Schaffer, and John R. Buck. *Discrete-Time Signal Processing*. 2nd Ed. Upper Saddle River, NJ: Prentice Hall, 1999.
- [2] Mitra, Sanjit K. *Digital Signal Processing*. 2nd Ed. New York: McGraw-Hill, 2001, secs. 4.4.2 and 8.2.5.
- [3] Gustafsson, F. "Determining the initial states in forward-backward filtering." *IEEE Transactions on Signal Processing*. Vol. 44, April 1996, pp. 988–992.

**See Also**

`designfilt` | `digitalFilter` | `fftfilt` | `filter` | `filter2`

## **filtic**

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.

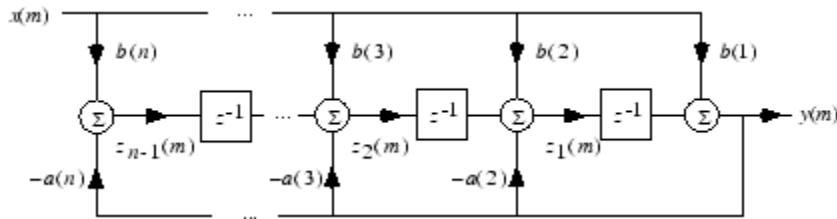
$$x = [x(-1), x(-2), x(-3), \dots, x(-n)]$$
$$y = [y(-1), y(-2), y(-3), \dots, y(-m)]$$

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.

## 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.

## More About

### Algorithms

`filtic` performs a reverse difference equation to obtain the delay states `z`.

## References

- [1] Oppenheim, A.V., and R.W. Schaffer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, pp.296, 301-302.

### See Also

`filter` | `filtfilt`

# filtord

Filter order

## Syntax

```
n = filtord(b,a)
n = filtord(sos)
n = filtord(d)
```

## Description

`n = filtord(b,a)` returns the filter order, `n`, for the causal rational system function specified by the numerator coefficients, `b`, and denominator coefficients, `a`.

`n = filtord(sos)` returns the filter order for the filter specified by the second-order sections matrix, `sos`. `sos` is a  $K$ -by-6 matrix. The number of sections,  $K$ , must be greater than or equal to 2. Each row of `sos` corresponds to the coefficients of a second-order filter. The  $i$ th row of the second-order section matrix corresponds to `[bi(1) bi(2) bi(3) ai(1) ai(2) ai(3)]`.

`n = filtord(d)` returns the filter order, `n`, for the digital filter, `d`. Use the function `designfilt` to generate `d`.

## Examples

### Verify Order of FIR Filter

Design a 20th-order FIR filter with normalized cutoff frequency  $0.5\pi$  rad/sample using the window method. Verify the filter order.

```
b = fir1(20,0.5);
n = filtord(b)
```

```
n =
```

20

Design the same filter using `designfilt` and verify its order.

```
di = designfilt('lowpassfir', 'FilterOrder', 20, 'CutoffFrequency', 0.5);  
ni = filtord(di)
```

ni =

20

### Determine the Order Difference Between FIR and IIR Designs

Design FIR equiripple and IIR Butterworth filters from the same set of specifications. Determine the difference in filter order between the two designs.

```
fir = designfilt('lowpassfir', 'DesignMethod', 'equiripple', 'SampleRate', 1e3, ...  
               'PassbandFrequency', 100, 'StopbandFrequency', 120, ...  
               'PassbandRipple', 0.5, 'StopbandAttenuation', 60);  
iir = designfilt('lowpassiir', 'DesignMethod', 'butter', 'SampleRate', 1e3, ...  
               'PassbandFrequency', 100, 'StopbandFrequency', 120, ...  
               'PassbandRipple', 0.5, 'StopbandAttenuation', 60);  
FIR = filtord(fir)  
IIR = filtord(iir)
```

FIR =

114

IIR =

41

## Input Arguments

**b** — Numerator coefficients

vector | scalar

Numerator coefficients, specified as a scalar or a vector. If the filter is an allpole filter, `b` is a scalar. Otherwise, `b` is a row or column vector.

Example: `b = fir1(20,0.25)`

Data Types: `single` | `double`

Complex Number Support: Yes

### **a** — Denominator coefficients

vector | scalar

Denominator coefficients, specified as a scalar or a vector. If the filter is an FIR filter, `a` is a scalar. Otherwise, `a` is a row or column vector.

Example: `[b,a] = butter(20,0.25)`

Data Types: `single` | `double`

Complex Number Support: Yes

### **sos** — Matrix of second-order sections

matrix

Matrix of second order-sections, specified as a  $K$ -by-6 matrix. The system function of the  $K$ th biquad filter has the rational  $Z$ -transform

$$H_k(z) = \frac{B_k(1) + B_k(2)z^{-1} + B_k(3)z^{-2}}{A_k(1) + A_k(2)z^{-1} + A_k(3)z^{-2}}.$$

The coefficients in the  $K$ th row of the matrix, `sos`, are ordered as follows.

$$[B_k(1) \ B_k(2) \ B_k(3) \ A_k(1) \ A_k(2) \ A_k(3)].$$

The frequency response of the filter is the system function evaluated on the unit circle with

$$z = e^{j2\pi f}.$$

Data Types: `single` | `double`

Complex Number Support: Yes

### **d** — Digital filter

`digitalFilter` object

Digital filter, specified as a `digitalFilter` object. Use `designfilt` to generate a digital filter based on frequency-response specifications.

Example: `d = designfilt('lowpassiir','FilterOrder',3,'HalfPowerFrequency',0.5)` specifies a third-order Butterworth filter with normalized 3-dB frequency  $0.5\pi$  rad/sample.

## Output Arguments

### **n** — Filter order

integer

Filter order, specified as an integer.

### See Also

`digitalFilter` | `designfilt` | `isallpass` | `ismaxphase` | `isminphase` | `isstable`



# filtstates

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 DSP System 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.

| <code>filtstates.structure</code> | Description  |
|-----------------------------------|--|
| <code>filtstates.dfir</code>      | filtstates for IIR direct-form I filters ( <code>dfilt.df1</code> , <code>dfilt.df1t</code> , <code>dfilt.df1sos</code> , and <code>dfilt.df1tsos</code> ) |
| <code>filtstates.cic</code>       | filtstates for cascaded integrator comb filters. (Available only with DSP System Toolbox and Fixed-Point Designer 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

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   |
|---------------------|---|
| <code>double</code> | Converts a <code>filtstates</code> 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. |
| <code>single</code> | Converts a <code>filtstates</code> object to a single-precision vector containing the values of the numerator and denominator states. (This method is used with the DSP System 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`

# filt2block

Generate Simulink filter block

## Syntax

```
filt2block(b)
filt2block(b, 'subsystem')
filt2block( ____, 'FilterStructure', structure)

filt2block(b, a)
filt2block(b, a, 'subsystem')
filt2block( ____, 'FilterStructure', structure)

filt2block(sos)
filt2block(sos, 'subsystem')
filt2block( ____, 'FilterStructure', structure)

filt2block(d)
filt2block(d, 'subsystem')
filt2block( ____, 'FilterStructure', structure)

filt2block( ____, Name, Value)
```

## Description

`filt2block(b)` generates a **Discrete FIR Filter** block with filter coefficients, `b`.

`filt2block(b, 'subsystem')` generates a Simulink subsystem block that implements an FIR filter using sum, gain, and delay blocks.

`filt2block( ____, 'FilterStructure', structure)` specifies the filter structure for the FIR filter.

`filt2block(b, a)` generates a **Discrete Filter** block with numerator coefficients, `b`, and denominator coefficients, `a`.

`filt2block(b, a, 'subsystem')` generates a Simulink subsystem block that implements an IIR filter using sum, gain, and delay blocks.

`filt2block( ____, 'FilterStructure', structure)` specifies the filter structure for the IIR filter.

`filt2block(sos)` generates a **Biquad Filter block** with second order sections matrix, `sos`. `sos` is a  $K$ -by-6 matrix, where the number of sections,  $K$ , must be greater than or equal to 2. You must have the DSP System Toolbox software installed to use this syntax.

`filt2block(sos, 'subsystem')` generates a Simulink subsystem block that implements a biquad filter using sum, gain, and delay blocks.

`filt2block( ____, 'FilterStructure', structure)` specifies the filter structure for the biquad filter.

`filt2block(d)` generates a Simulink block that implements a digital filter, `d`. Use the function `designfilt` to create `d`. The block is a **Discrete FIR Filter block** if `d` is FIR and a **Biquad Filter block** if `d` is IIR.

`filt2block(d, 'subsystem')` generates a Simulink subsystem block that implements `d` using sum, gain, and delay blocks.

`filt2block( ____, 'FilterStructure', structure)` specifies the filter structure to implement `d`.

`filt2block( ____, Name, Value)` uses additional options specified by one or more `Name, Value` pair arguments.

## Examples

### Generate Block from FIR Filter

Design an order 30 FIR filter using the window method. Specify a cutoff frequency of  $\pi/4$  radians/sample. Create a Simulink block.

```
b = fir1(30,0.25);  
filt2block(b)
```

### Generate Block from IIR Filter

Design an order 30 IIR Butterworth filter. Specify a cutoff frequency of  $\pi/4$  radians/sample. Create a Simulink block.

```
[b,a] = butter(30,0.25);
filt2block(b,a)
```

### Generate FIR Filter with Direct Form I Transposed Structure

Design an order 30 FIR filter using the window method. Specify a cutoff frequency of  $\pi/4$  radians/sample. Create a Simulink block with a direct form I transposed structure

```
b = fir1(30,0.25);
filt2block(b, 'FilterStructure', 'directFormTransposed')
```

### Generate IIR Filter with Direct Form I Structure

Design an order 30 IIR Butterworth filter. Specify a cutoff frequency of  $\pi/4$  radians/sample. Create a Simulink block with a direct form I structure.

```
[b,a] = butter(30,0.25);
filt2block(b,a, 'FilterStructure', 'directForm1')
```

### Generate Simulink Subsystem Block from Second Order Section Matrix

Design a 5-th order Butterworth filter with a cutoff frequency of  $0.2\pi$  radians/sample. Obtain the filter in biquad form and generate a Simulink subsystem block from the second order sections.

```
[z,p,k] = butter(5,0.2);
sos = zp2sos(z,p,k);
filt2block(sos, 'subsystem')
```

### Lowpass FIR Filter Block with Sample-Based Processing

Generate a Simulink subsystem block that implements an FIR lowpass filter using sum, gain, and delay blocks. Specify the input processing to be elements as channels by specifying 'FrameBasedProcessing' as false.

```
B = fir1(30, .25);
filt2block(B, 'subsystem', 'BlockName', 'Lowpass FIR', ...
    'FrameBasedProcessing', false)
```

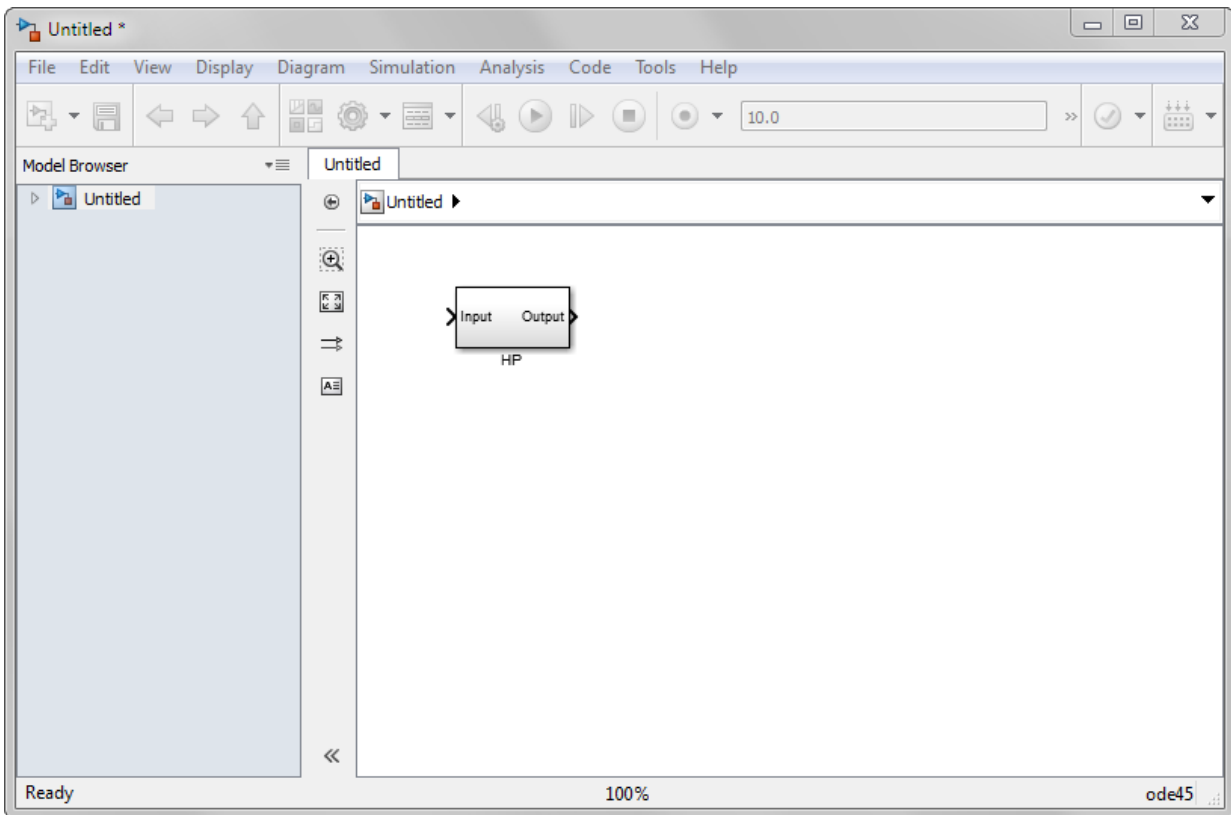
### New Model with Highpass Elliptic Filter Block

Design a highpass elliptic filter with normalized stopband frequency 0.45 and normalized passband frequency 0.55. Specify a stopband attenuation of 40 dB and a passband ripple

of 0.5 dB. Implement the filter as a Direct Form II structure, call it “HP”, and place it in a new Simulink model.

```
d = designfilt('highpassiir','DesignMethod','ellip', ...
              'StopbandFrequency',0.45,'PassbandFrequency',0.55, ...
              'StopbandAttenuation',40,'PassbandRipple',0.5);

filt2block(d,'subsystem','FilterStructure','directForm2', ...
           'Destination','new','BlockName','HP')
```



## Input Arguments

**b** — Numerator filter coefficients

row or column vector



Numerator filter coefficients, specified as a row or column vector. The filter coefficients are ordered in descending powers of  $z^{-1}$  with the first element corresponding to the coefficient for  $z^0$ .

Example: `b = fir1(30,0.25);`

Data Types: `single` | `double`

Complex Number Support: Yes

### **a** — Denominator filter coefficients

row or column vector

Denominator filter coefficients, specified as a row or column vector. The filter coefficients are ordered in descending powers of  $z^{-1}$  with the first element corresponding to the coefficient for  $z^0$ . The first filter coefficient must be 1.

Data Types: `single` | `double`

Complex Number Support: Yes

### **sos** — Second-order section matrix

$K$ -by-2 matrix

Second order section matrix, specified as a  $K$ -by-2 matrix. Each row of the matrix contains the coefficients for a biquadratic rational function in  $z^{-1}$ . The Z-transform of the  $K$ th rational biquadratic system impulse response is

$$H_k(z) = \frac{B_k(1) + B_k(2)z^{-1} + B_k(3)z^{-2}}{A_k(1) + A_k(2)z^{-1} + A_k(3)z^{-2}}$$

The coefficients in the  $K$ th row of the matrix, `sos`, are ordered as follows:

$$[B_k(1) \ B_k(2) \ B_k(3) \ A_k(1) \ A_k(2) \ A_k(3)]$$

The frequency response of the filter is its transfer function evaluated on the unit circle with  $z = e^{j2\pi f}$ .

Data Types: `single` | `double`

Complex Number Support: Yes

**d — Digital filter**

digitalFilter object

Digital filter, specified as a digitalFilter object. Use designfilt to generate a digital filter based on frequency-response specifications.

Example: d = designfilt('lowpassfir','FilterOrder',3,'HalfPowerFrequency',0.5) specifies a third-order Butterworth filter with normalized 3-dB frequency 0.5π rad/sample.

**structure — Filter structure**

string

Filter structure, specified as a string. Valid options for structure depend on the input arguments. The following table lists the valid filter structures by input.

| Input | Filter Structures  |
|-------|--|
| b     | 'directForm' (default), 'directFormTransposed', 'directFormSymmetric', 'directFormAntiSymmetric', 'overlapAdd'. The 'overlapAdd' structure is only available when you omit 'subsystem' and requires a DSP System Toolbox software license. |
| a     | 'directForm2' (default), 'directForm1', 'directForm1Transposed', 'directForm2', 'directForm2Transposed'  |
| sos   | 'directForm2Transposed' (default), 'directForm1', 'directForm1Transposed', 'directForm2'   |
| d     | • For FIR filters: 'directForm' (default), 'directFormTransposed',   |

| Input | Filter Structures   |
|-------|---|
|       | <p>'directFormSymmetric',<br/>'directFormAntiSymmetric',<br/>'overlapAdd'. The 'overlapAdd' structure is only available when you omit 'subsystem' and requires a DSP System Toolbox software license.</p> <ul style="list-style-type: none"> <li>For IIR filters:<br/>'directForm2Transposed' (default), 'directForm1', 'directForm1Transposed', 'directForm2'</li> </ul> |

## Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: `filt2block(...,'subsystem','BlockName','Lowpass FIR','FrameBasedProcessing',false)`

### 'Destination' — Destination for Simulink filter block

'current' (default) | 'new' | user-defined string

Destination for the Simulink filter block, specified as a string. You can add the filter block to your current model with 'current', add the filter block to a new model with 'new', or specify the name of a target subsystem.

Example: `filt2block(b,'subsystem','MyFilterBlock')`

Data Types: char

### 'BlockName' — Block name

string

Block name, specified as a string.

### 'OverwriteBlock' — Overwrite block

false (default) | true

Overwrite block, specified as a logical `false` or `true`. If you use a value for `'BlockName'` that is the same as an existing block, the value of `'OverwriteBlock'` determines whether the block is overwritten. The default value is `false`.

Data Types: `logical`

**'MapCoefficientsToPorts' — Map coefficients to ports**

`false` (default) | `true`

Map coefficients to ports, specified as a logical `false` or `true`.

Data Types: `logical`

**'CoefficientNames' — Coefficient variable names**

cell array of strings

Coefficient variable names, specified as a cell array. This name-value pair is only applicable when `'MapCoefficientsToPorts'` is `true`. The default values are `{'Num'}`, `{'Num', 'Den'}`, and `{'Num', 'Den', 'g'}` for FIR, IIR, and biquad filters.

Data Types: `cell`

**'FrameBasedProcessing' — Frame-based or sample-based processing**

`true` (default) | `false`

Frame-based or sample-based processing, specified as a logical `true` or `false`. The default is `true` and frame-based processing is used.

Data Types: `logical`

**'OptimizeZeros' — Remove zero-gain blocks**

`true` (default) | `false`

Remove zero-gain blocks, specified as a logical `true` or `false`. By default zero-gain blocks are removed.

Data Types: `logical`

**'OptimizeOnes' — Replace unity-gain blocks with direct connection**

`true` (default) | `false`

Replace unity-gain blocks with direct connection, specified as a logical `true` or `false`. The default is `true`.

Data Types: `logical`

**'OptimizeNegativeOnes'** — Replace negative unity-gain blocks with sign change  
true (default) | false

Replace negative unity-gain blocks with a sign change at the nearest block, specified as a logical `true` or `false`. The default is `true`.

Data Types: `logical`

**'OptimizeDelayChains'** — Replace cascaded delays with a single delay  
true (default) | false

Replace cascaded delays with a single delay, specified as a logical `true` or `false`. The default is `true`.

Data Types: `logical`

## See Also

`digitalFilter` | `designfilt` | `realizemdl`

# findpeaks

Find local maxima

## Syntax

```
pks = findpeaks(data)
[pks,locs] = findpeaks(data)
[pks,locs,w,p] = findpeaks(data)

[ ___ ] = findpeaks(data,x)
[ ___ ] = findpeaks(data,Fs)

[ ___ ] = findpeaks( ___,Name,Value)

findpeaks( ___ )
```

## Description

`pks = findpeaks(data)` returns a vector with the local maxima (peaks) of the input signal vector, `data`. A *local peak* is a data sample that is either larger than its two neighboring samples or is equal to `Inf`. Non-`Inf` signal endpoints are excluded. If a peak is flat, the function returns only the point with the lowest index.

`[pks,locs] = findpeaks(data)` additionally returns the indices at which the peaks occur.

`[pks,locs,w,p] = findpeaks(data)` additionally returns the widths of the peaks as the vector `w` and the prominences of the peaks as the vector `p`.

`[ ___ ] = findpeaks(data,x)` specifies `x` as the location vector and returns any of the output arguments from previous syntaxes. `locs` and `w` are expressed in terms of `x`.

`[ ___ ] = findpeaks(data,Fs)` specifies the sample rate, `Fs`, of the data. The first sample of data is assumed to have been taken at time zero. `locs` and `w` are converted to time units.

`[ ___ ] = findpeaks( ___,Name,Value)` specifies options using name-value pair arguments in addition to any of the input arguments in previous syntaxes.

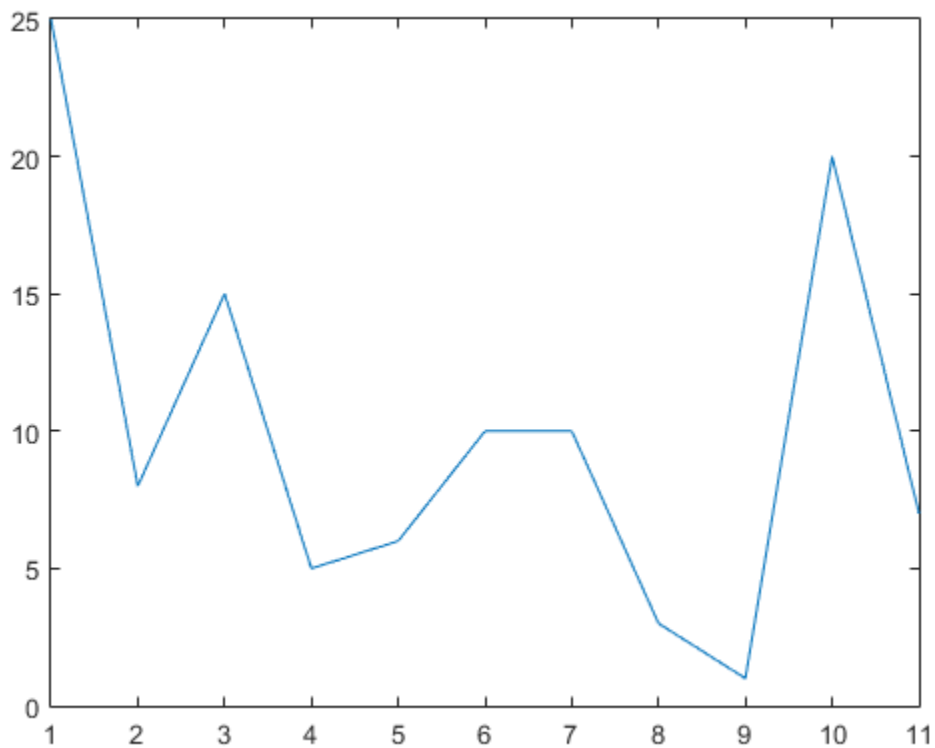
`findpeaks( ___ )` without output arguments plots the signal and overlays the peak values.

## Examples

### Find Peaks in a Vector

Define a vector with three peaks and plot it.

```
data = [25 8 15 5 6 10 10 3 1 20 7];  
plot(data)
```



Find the local maxima. The peaks are output in order of occurrence. The first sample is not included despite being the maximum. For the flat peak, the function returns only the point with lowest index.

```
pks = findpeaks(data)
```

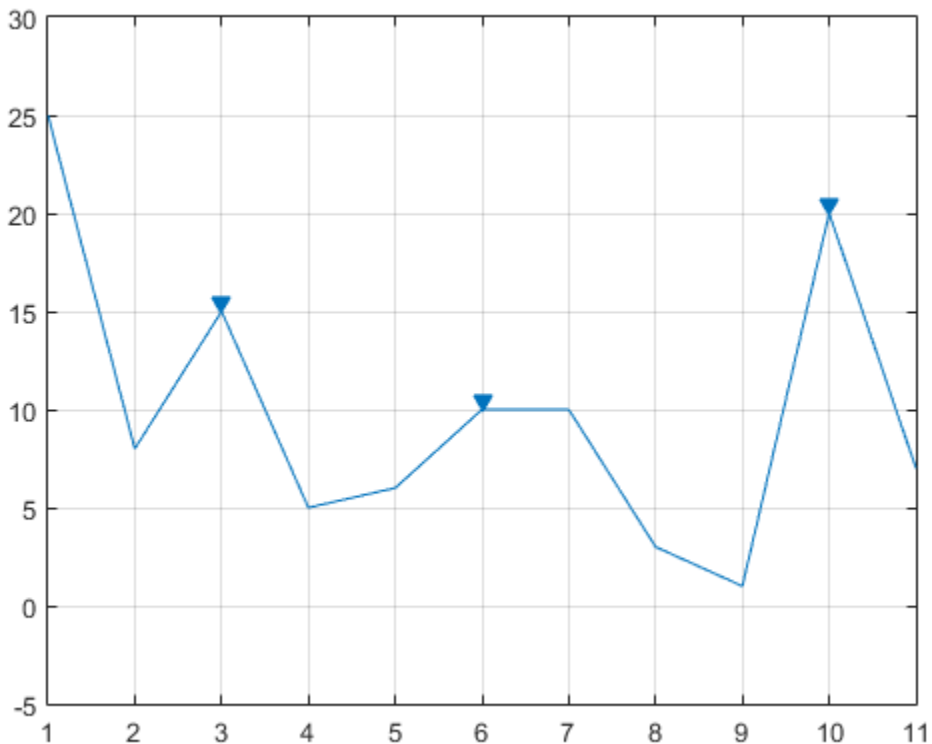
```
pks =
```

```
    15    10    20
```

Use `findpeaks` without output arguments to display the peaks.

```
findpeaks(data)
```





### Find Peaks and Their Locations

Create a signal that consists of a sum of bell curves. Specify the location, height, and width of each curve.

```
x = linspace(0,1,1000);
```

```
Pos = [1 2 3 5 7 8]/10;
```

```
Hgt = [4 4 4 2 2 3];
```

```
Wdt = [2 6 3 3 4 6]/100;
```

```
for n = 1:length(Pos)
```

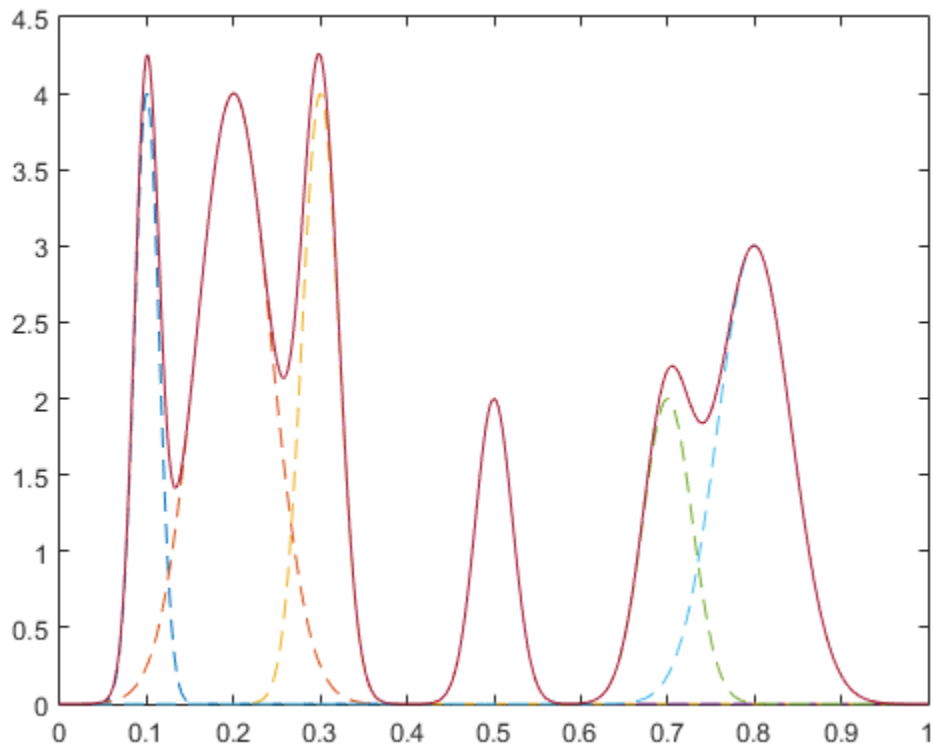
```
    Gauss(n,:) = Hgt(n)*exp(-((x - Pos(n))/Wdt(n)).^2);
```

```
end
```

```
PeakSig = sum(Gauss);
```

Plot the individual curves and their sum.

```
plot(x,Gauss,'- -',x,PeakSig)
```



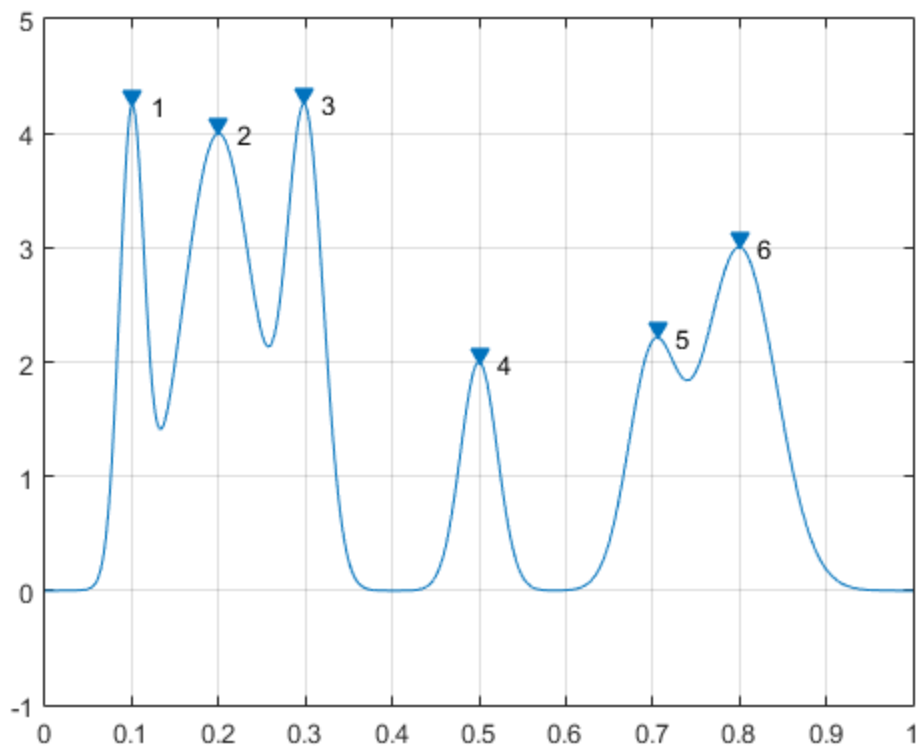
Use `findpeaks` with default settings to find the peaks of the signal and their locations.

```
[pks,locs] = findpeaks(PeakSig,x);
```

Plot the peaks using `findpeaks` and label them.

```
findpeaks(PeakSig,x)
```

```
text(locs+.02,pks,num2str((1:numel(pks))'))
```

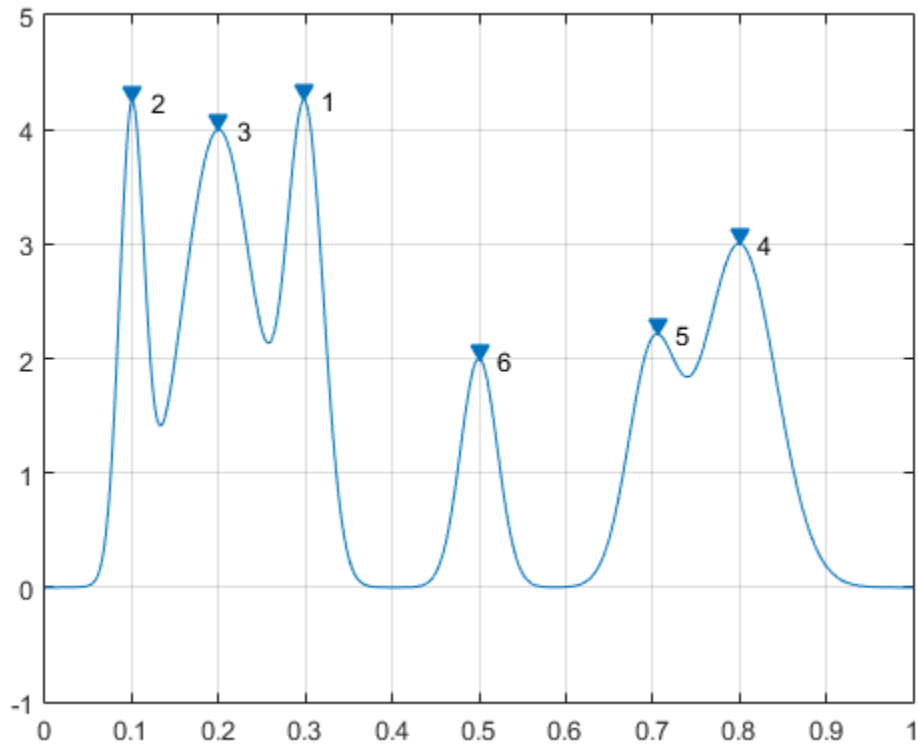


Sort the peaks from tallest to shortest.

```
[psor,lsor] = findpeaks(PeakSig,x,'SortStr','descend');
```

```
findpeaks(PeakSig,x)
```

```
text(lsor+.02,psor,num2str((1:numel(psor))'))
```



### Peak Prominences

Create a signal that consists of a sum of bell curves riding on a full period of a cosine. Specify the location, height, and width of each curve.

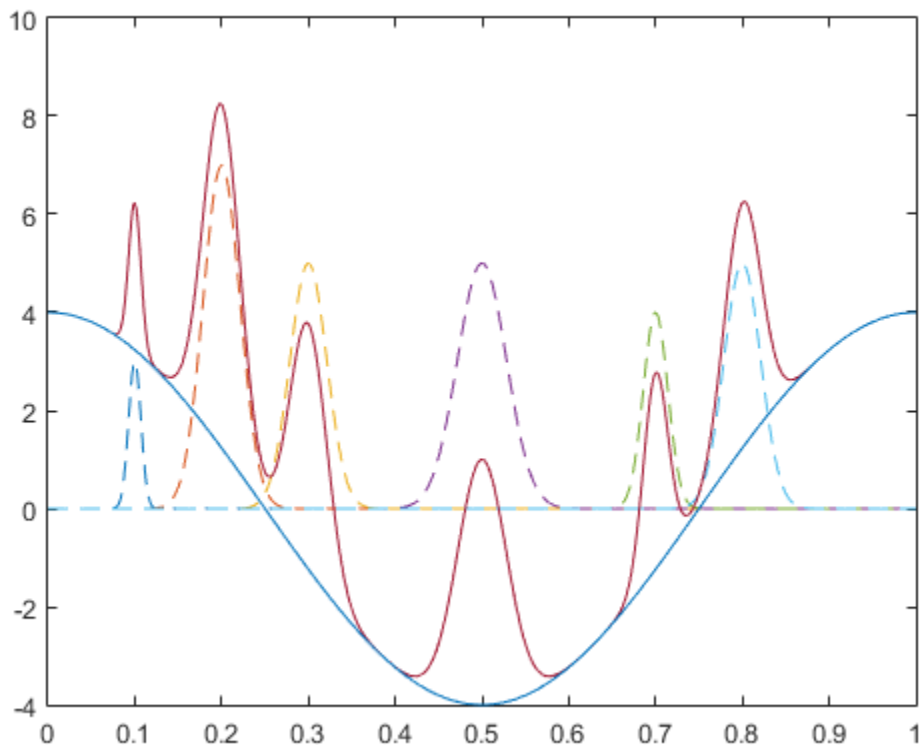
```
x = linspace(0,1,1000);
base = 4*cos(2*pi*x);
Pos = [1 2 3 5 7 8]/10;
Hgt = [3 7 5 5 4 5];
Wdt = [1 3 3 4 2 3]/100;
for n = 1:length(Pos)
```

```
Gauss(n,:) = Hgt(n)*exp(-((x - Pos(n))/Wdt(n)).^2);
end
```

```
PeakSig = sum(Gauss)+base;
```

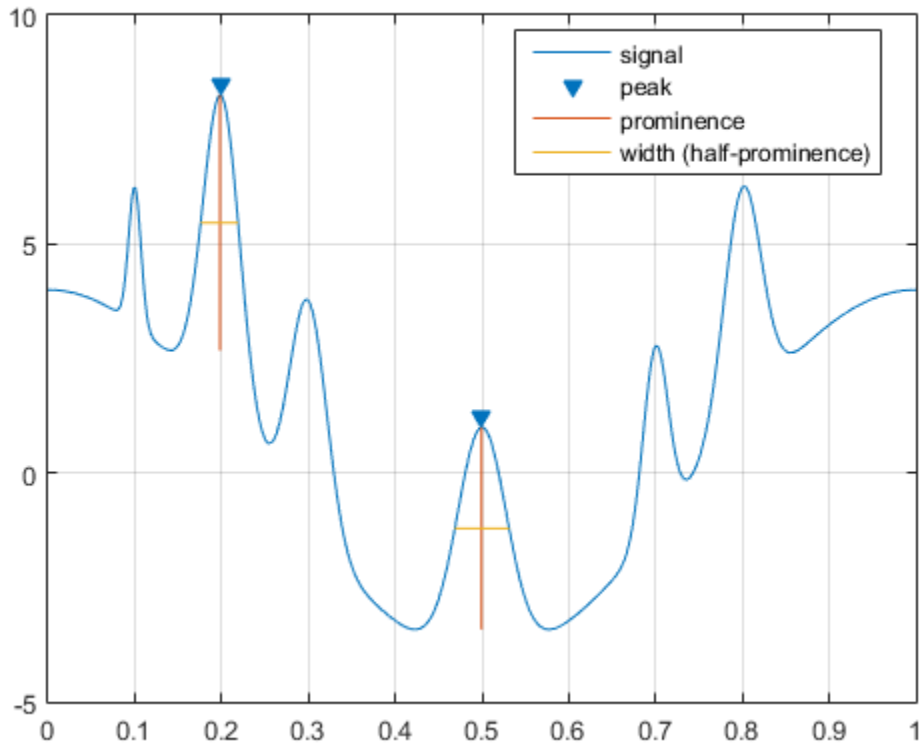
Plot the individual curves and their sum.

```
plot(x,Gauss,'--',x,PeakSig,x,base)
```



Use `findpeaks` to locate and plot the peaks that have a prominence of at least 4.

```
findpeaks(PeakSig,x,'MinPeakProminence',4,'Annotate','extents')
```



The highest and lowest peaks are the only ones that satisfy the condition.

Display the prominences and the widths at half prominence of all the peaks.

```
[pks,locs,widths,proms] = findpeaks(PeakSig,x);
widths
proms
```

```
widths =
```

```
    0.0154    0.0431    0.0377    0.0625    0.0274    0.0409
```

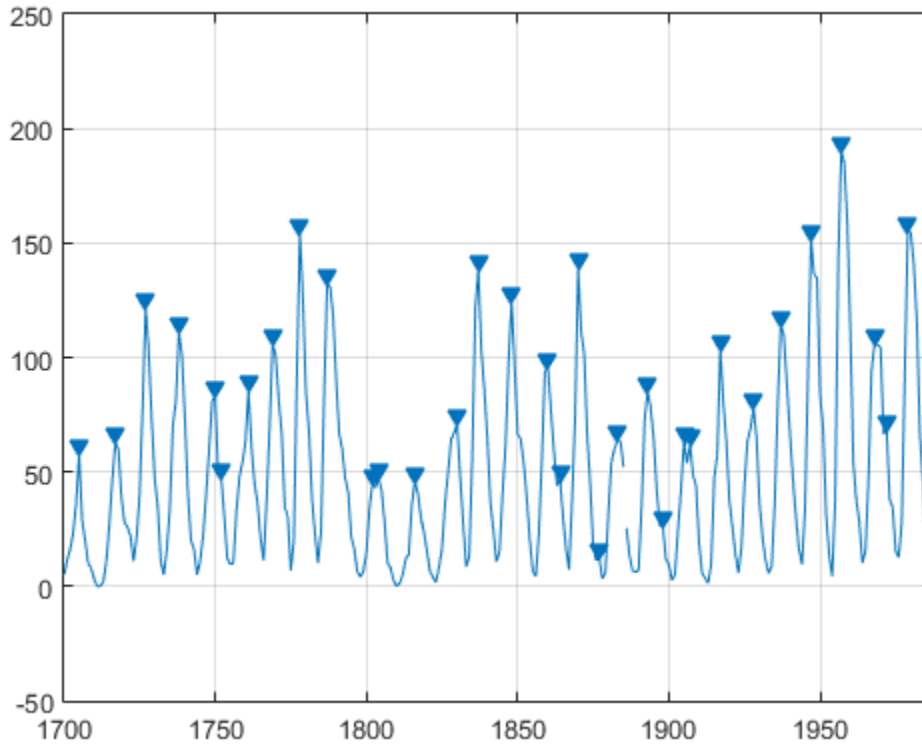
```
proms =  
    2.6816    5.5773    3.1448    4.4171    2.9191    3.6363
```

### **Find Peaks with Minimum Separation**

Sunspots are a cyclic phenomenon. Their number is known to peak roughly every 11 years.

Load the file `sunspot.dat`, which contains the average number of sunspots observed every year from 1700 to 1987. Find and plot the maxima.

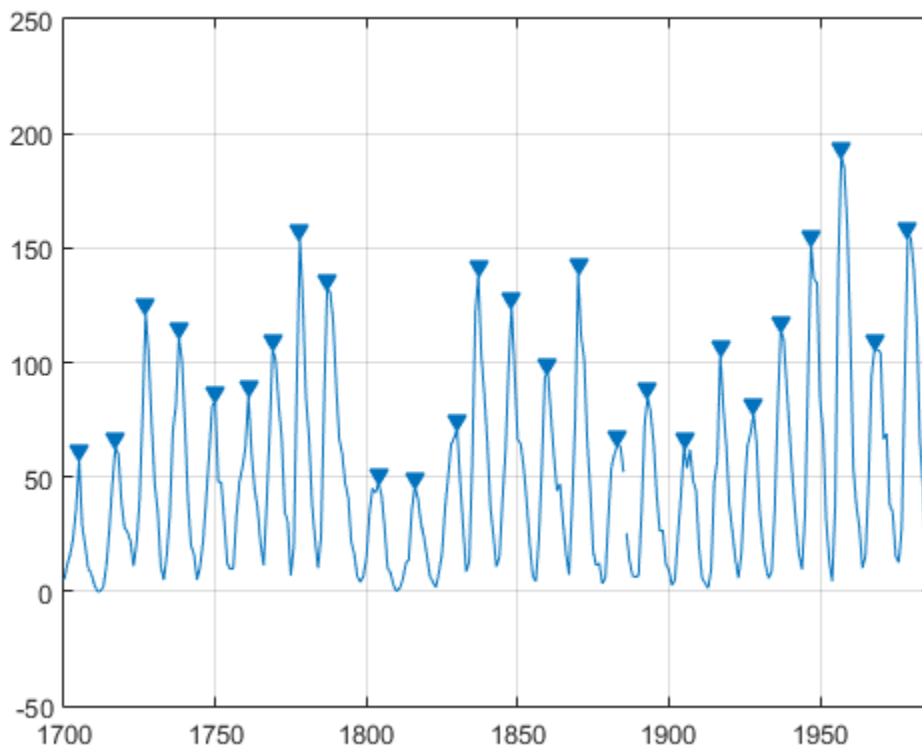
```
load sunspot.dat  
  
year = sunspot(:,1);  
avSpots = sunspot(:,2);  
  
findpeaks(avSpots,year)
```



Improve your estimate of the cycle duration by ignoring peaks that are very close to each other. Find and plot the peaks again, but now restrict the acceptable peak-to-peak separations to values greater than six years.

```
findpeaks(avSpots,year, 'MinPeakDistance',6)
```





Use the peak locations returned by `findpeaks` to compute the mean interval between maxima.

```
[pks,locs] = findpeaks(avSpots,year,'MinPeakDistance',6);
```

```
meanCycle = mean(diff(locs))
```

```
meanCycle =
```

```
10.9600
```

### Constrain Peak Features

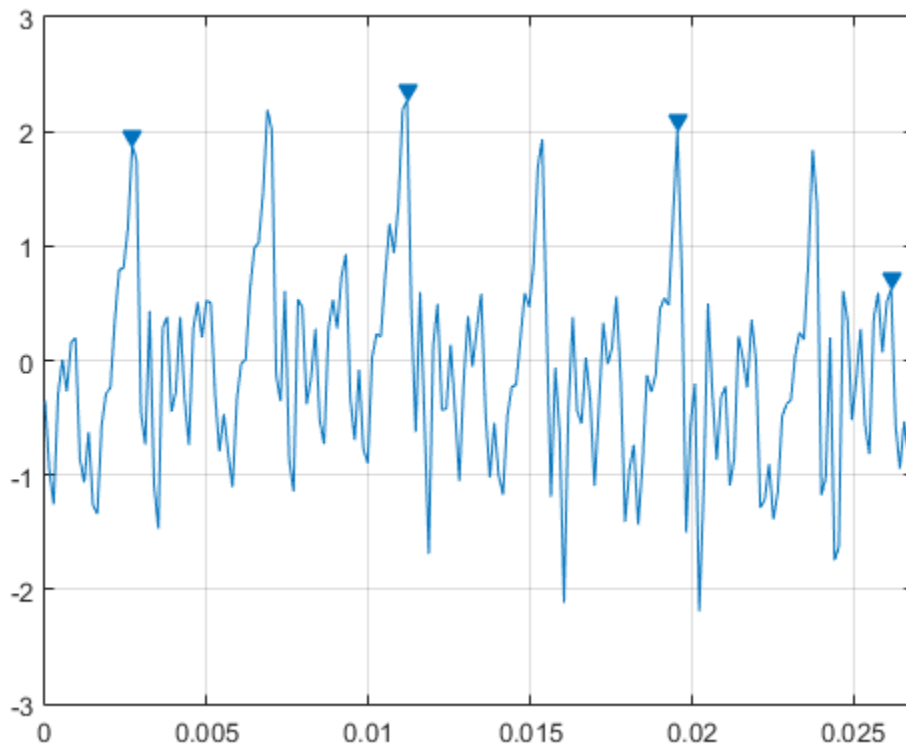
Load an audio signal sampled at 7418 Hz. Select 200 samples.

```
load mtlb  
select = mtlb(1001:1200);
```

Find the peaks that are separated by at least 5 ms.

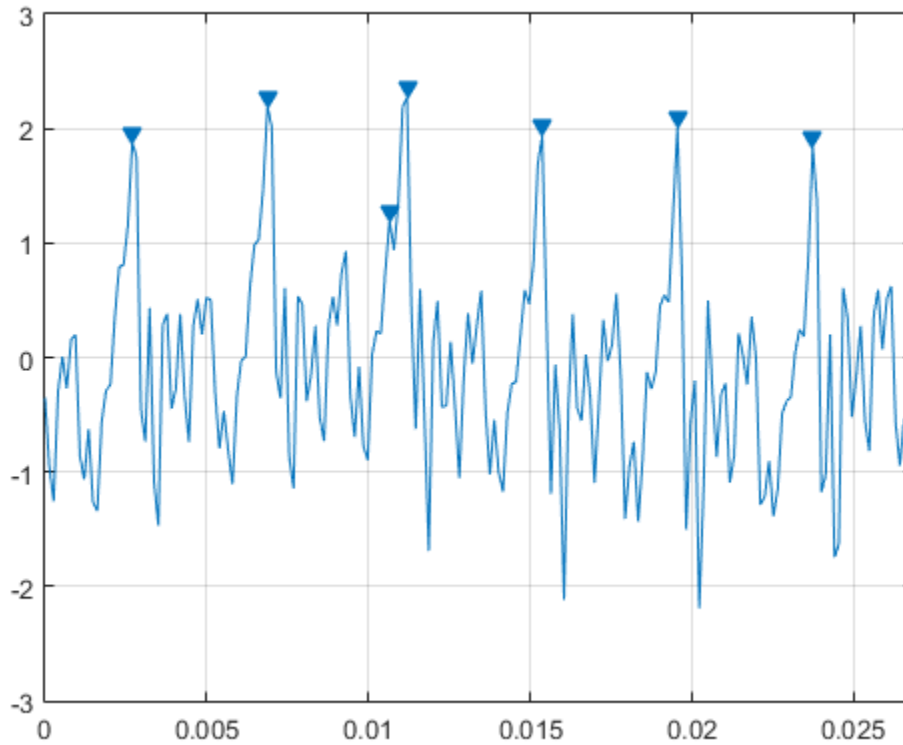
To apply this constraint, `findpeaks` chooses the tallest peak in the signal and eliminates all peaks within 5 ms of it. The function then repeats the procedure for the tallest remaining peak and iterates until it runs out of peaks to consider.

```
findpeaks(select,Fs,'MinPeakDistance',0.005)
```



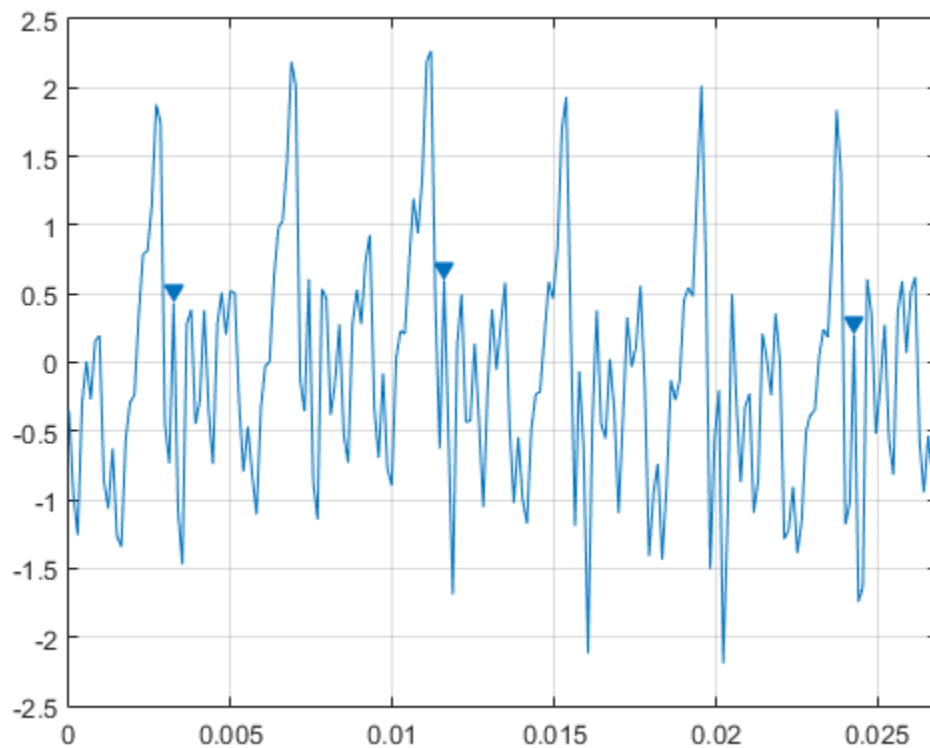
Find the peaks that have an amplitude of at least 1 V.

```
findpeaks(select,Fs, 'MinPeakHeight',1)
```



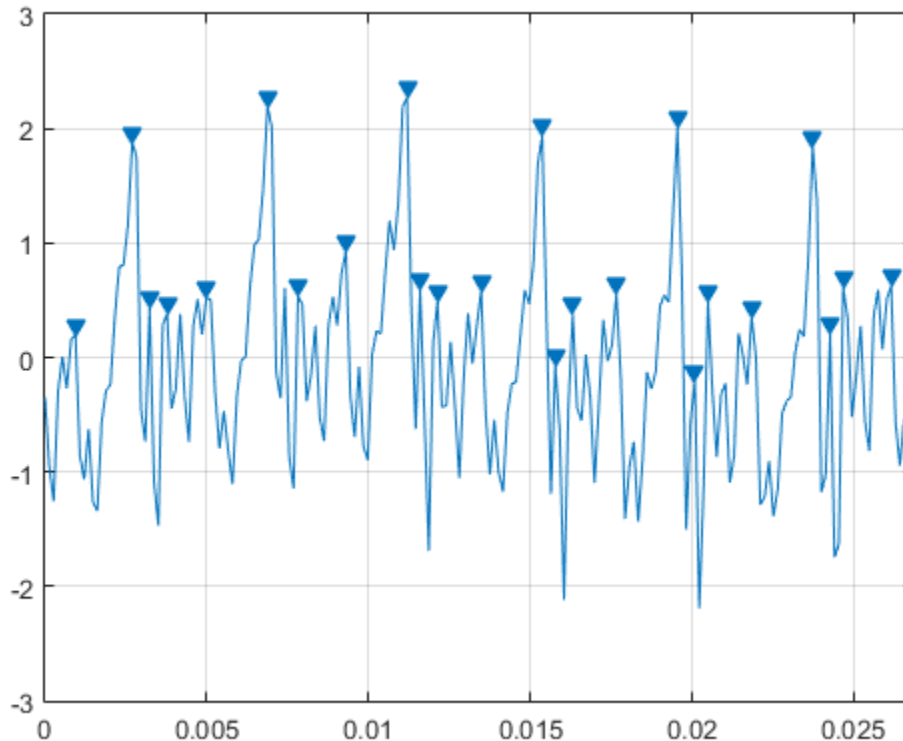
Find the peaks that are at least 1 V higher than their neighboring samples.

```
findpeaks(select,Fs, 'Threshold',1)
```



Find the peaks that drop at least 1 V on either side before the signal attains a higher value.

```
findpeaks(select,Fs, 'MinPeakProminence', 1)
```



### Peaks of Saturated Signal

Sensors can return clipped readings if the data are larger than a given saturation point. You can choose to disregard these peaks as meaningless or incorporate them to your analysis.

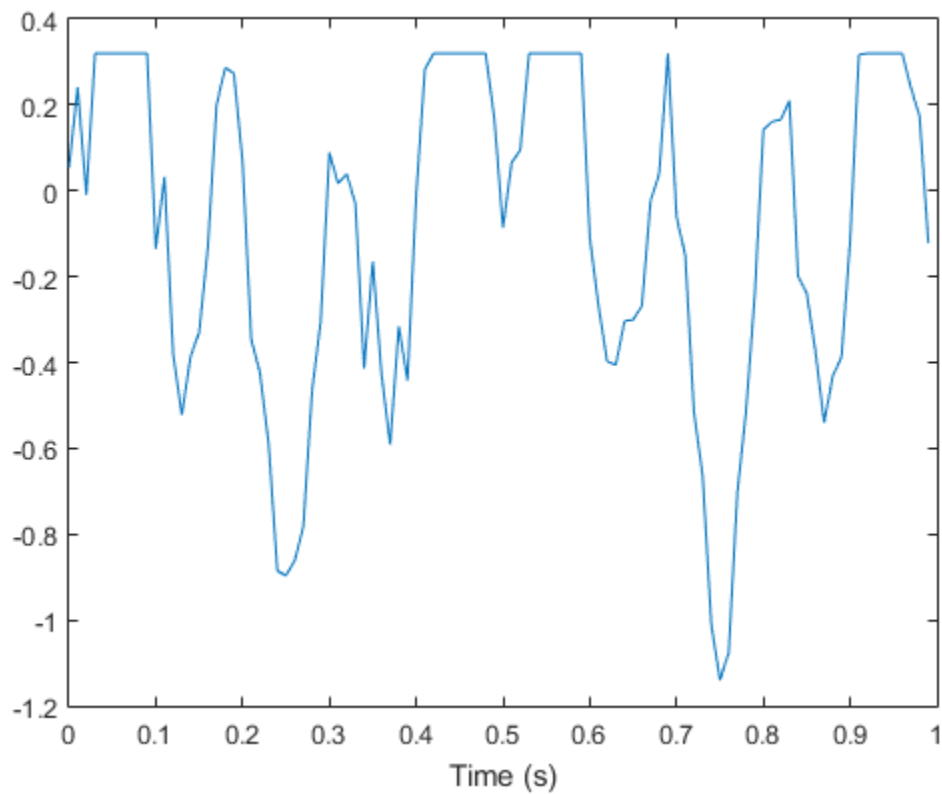
Generate a signal that consists of a product of trigonometric functions of frequencies 5 Hz and 3 Hz embedded in white Gaussian noise of variance  $0.1^2$ . Specify a sample rate of 100 Hz and a sample time of one second. Reset the random number generator for reproducible results.

```
rng default
```

```
fs = 1e2;  
t = 0:1/fs:1-1/fs;  
  
s = sin(2*pi*5*t).*sin(2*pi*3*t)+randn(size(t))/10;
```

Simulate a saturated measurement by truncating every reading that is greater than a specified bound of 0.32. Plot the saturated signal.

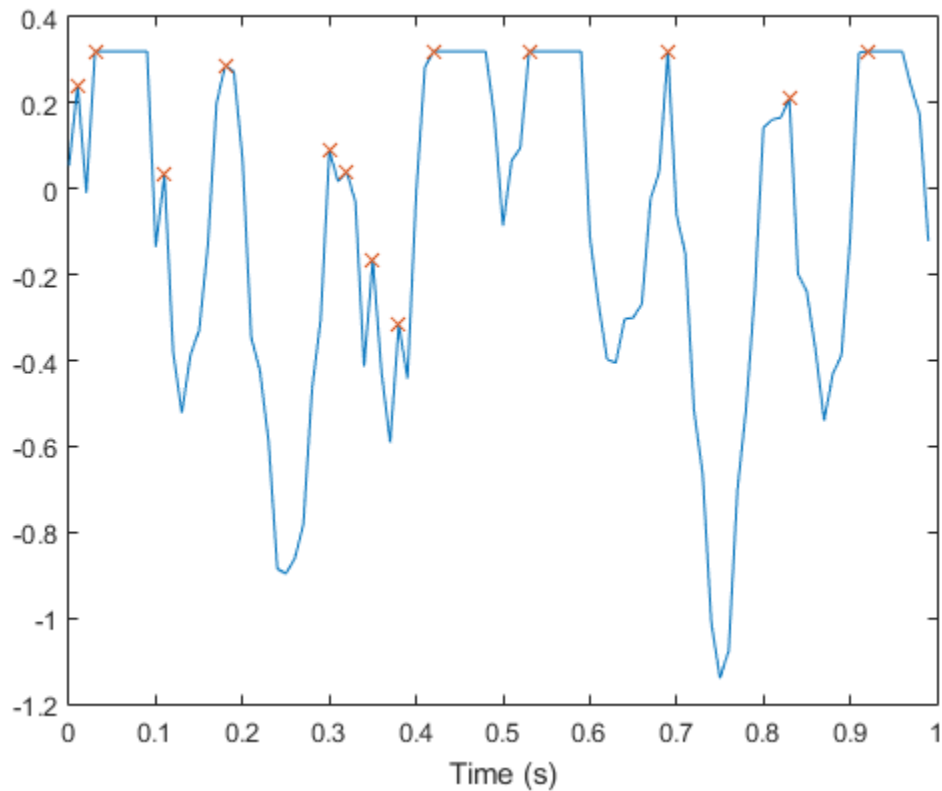
```
bnd = 0.32;  
s(s>bnd) = bnd;  
  
plot(t,s)  
xlabel('Time (s)')
```



Locate the peaks of the signal. `findpeaks` reports only the rising edge of each flat peak.

```
[pk,lc] = findpeaks(s,t);
```

```
hold on  
plot(lc,pk,'x')
```

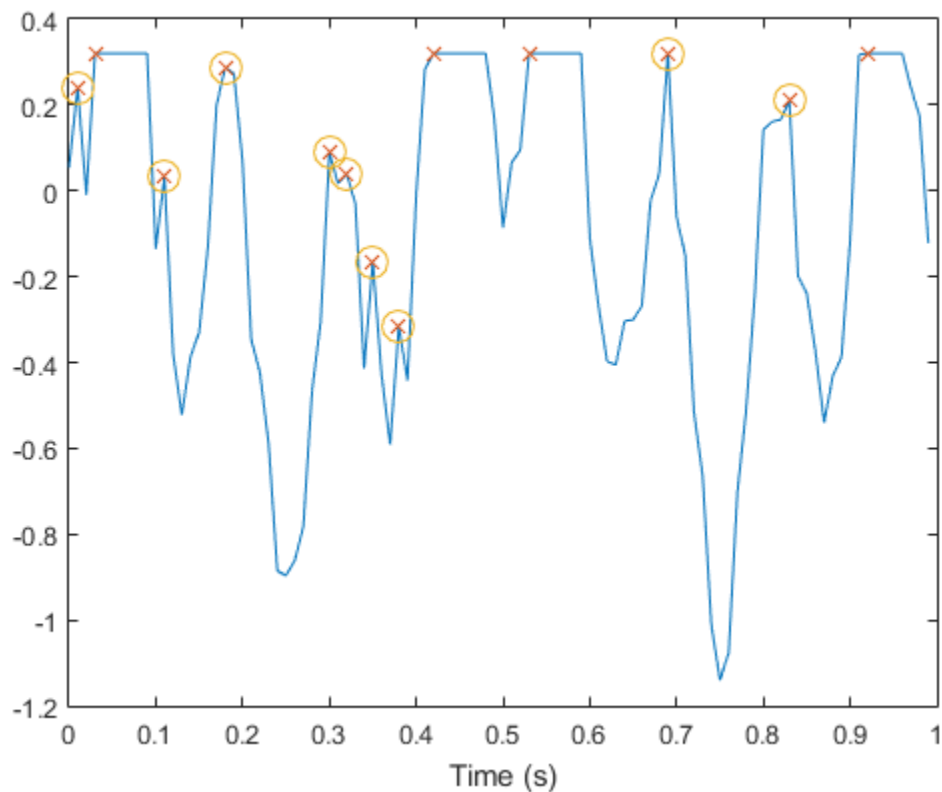


Use the 'Threshold' name-value pair to exclude the flat peaks. Require a minimum amplitude difference of  $10^{-4}$  between a peak and its neighbors.

```
[pkt,lct] = findpeaks(s,t,'Threshold',1e-4);
```

```
plot(lct,pkt,'o','MarkerSize',12)
```





### Determine Peak Widths

Create a signal that consists of a sum of bell curves. Specify the location, height, and width of each curve.

```
x = linspace(0,1,1000);
```

```
Pos = [1 2 3 5 7 8]/10;
```

```
Hgt = [4 4 2 2 2 3];
```

```
Wdt = [3 8 4 3 4 6]/100;
```

```
for n = 1:length(Pos)
```

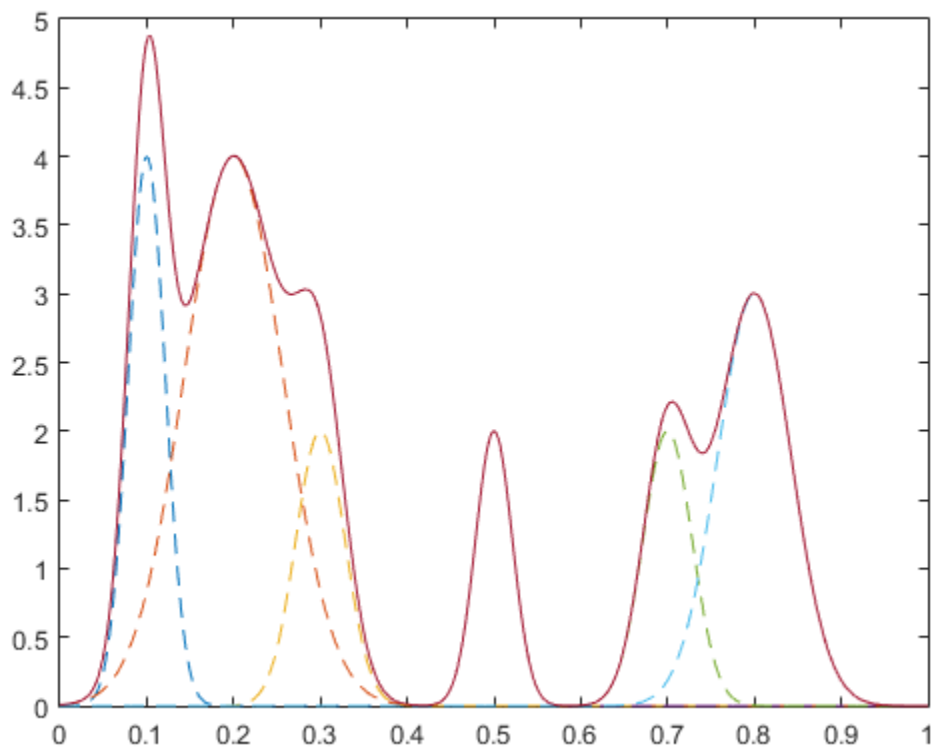
```
    Gauss(n,:) = Hgt(n)*exp(-((x - Pos(n))/Wdt(n)).^2);
```

```
end
```

```
PeakSig = sum(Gauss);
```

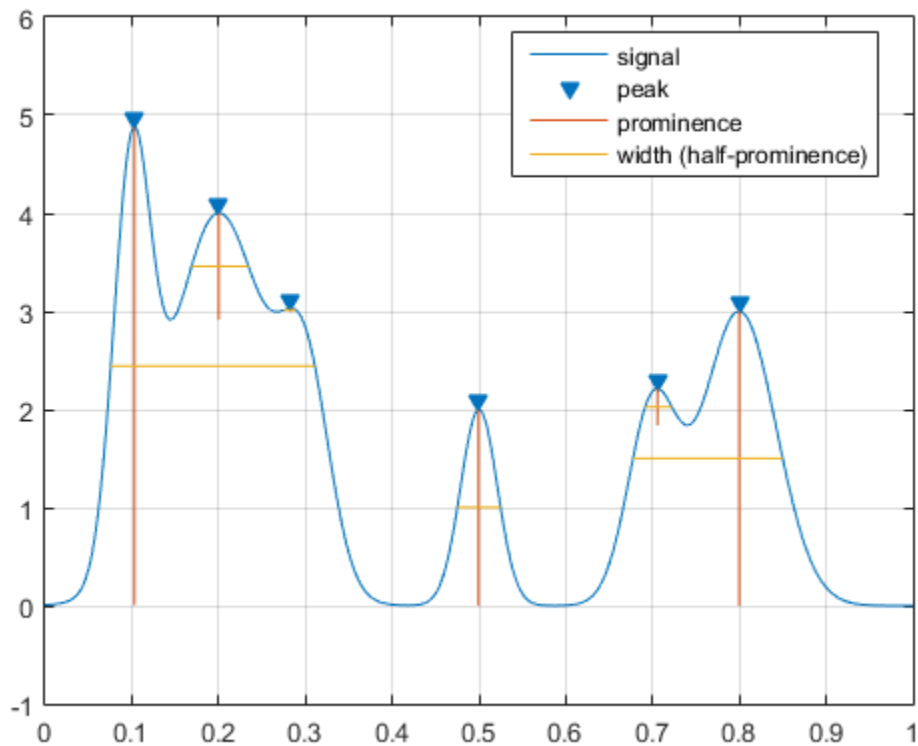
Plot the individual curves and their sum.

```
plot(x,Gauss,'--',x,PeakSig)
```



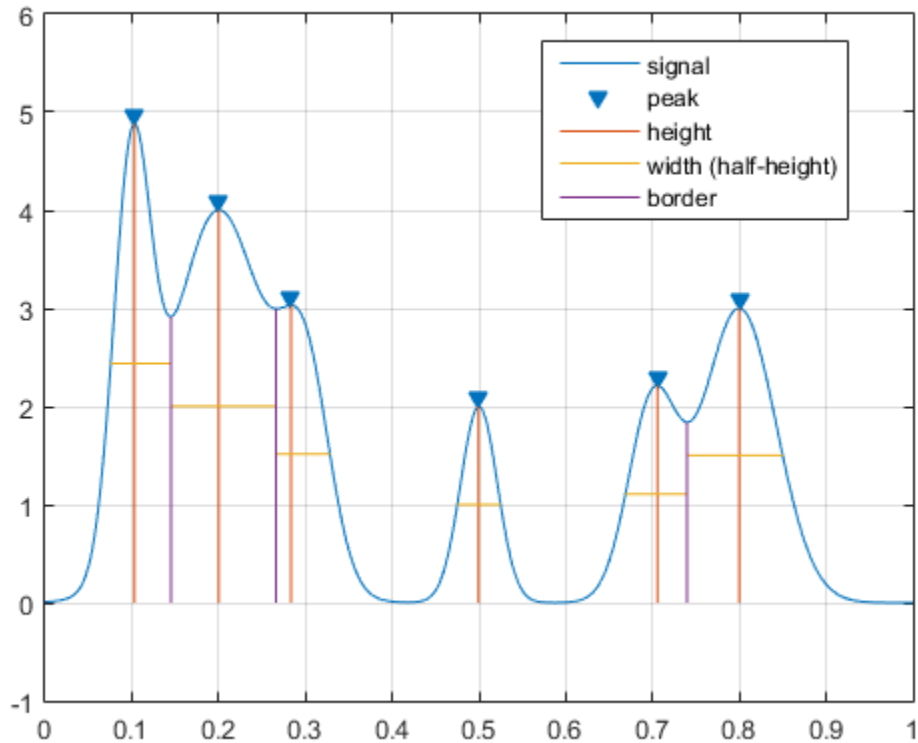
Measure the widths of the peaks using the half prominence as reference.

```
findpeaks(PeakSig,x,'Annotate','extents')
```



Measure the widths again, this time using the half height as reference.

```
findpeaks(PeakSig,x,'Annotate','extents','WidthReference','halfheight')
```



- “Peak Analysis”
- “Find Peaks in Data”

## Input Arguments

### **data** — Input data

vector

Input data, specified as a vector. data must be real and must have at least three elements.

Data Types: `single` | `double`

**x — Locations**

vector

Locations, specified as a vector. `x` must increase monotonically and have the same length as data. If `x` is omitted, then the indices of data are used as locations.

Data Types: `single` | `double`**Fs — Sample rate**

positive scalar

Sample rate, specified as a positive scalar. The sample rate is the number of samples per unit time. If the unit of time is seconds, the sample rate has units of hertz.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( `' '` ). You can specify several name and value pair arguments in any order as `Name1`, `Value1`, ..., `NameN`, `ValueN`.

Example: `'SortStr', 'descend', 'NPeaks', 3` finds the three tallest peaks of the signal.

**'NPeaks' — Maximum number of peaks**

positive integer scalar

Maximum number of peaks to return, specified as the comma-separated pair consisting of `'NPeaks'` and a positive integer scalar. `findpeaks` operates from the first element of the input data and terminates when the number of peaks reaches the value of `'NPeaks'`.

Data Types: `single` | `double`**'SortStr' — Peak sorting**`'none'` (default) | `'ascend'` | `'descend'`

Peak sorting, specified as the comma-separated pair consisting of `'SortStr'` and one of these values:

- `'none'` returns the peaks in the order in which they occur in the input data.
- `'ascend'` returns the peaks in ascending or increasing order, from the smallest to the largest value.

- `'descend'` returns the peaks in descending order, from the largest to the smallest value.

Data Types: `char`

**'MinPeakHeight' — Minimum peak height**

`-Inf` (default) | real scalar

Minimum peak height, specified as the comma-separated pair consisting of `'MinPeakHeight'` and a real scalar. Use this argument to have `findpeaks` return only those peaks higher than `'MinPeakHeight'`. Specifying a minimum peak height can reduce processing time.

Data Types: `single` | `double`

**'MinPeakProminence' — Minimum peak prominence**

`0` (default) | real scalar

Minimum peak prominence, specified as the comma-separated pair consisting of `'MinPeakProminence'` and a real scalar. Use this argument to have `findpeaks` return only those peaks that have a relative importance of at least `'MinPeakProminence'`. See “Prominence” on page 1-652 for more information.

Data Types: `single` | `double`

**'Threshold' — Minimum height difference**

`0` (default) | nonnegative real scalar

Minimum height difference between a peak and its neighbors, specified as the comma-separated pair consisting of `'Threshold'` and a nonnegative real scalar. Use this argument to have `findpeaks` return only those peaks that exceed their immediate neighboring values by at least the value of `'Threshold'`.

Data Types: `single` | `double`

**'MinPeakDistance' — Minimum peak separation**

`0` (default) | positive real scalar

Minimum peak separation, specified as the comma-separated pair consisting of `'MinPeakDistance'` and a positive real scalar. When you specify a value for `'MinPeakDistance'`, the algorithm chooses the tallest peak in the signal and ignores all peaks within `'MinPeakDistance'` of it. The function then repeats the procedure for the tallest remaining peak and iterates until it runs out of peaks to consider.

- If you specify a location vector, `x`, then `'MinPeakDistance'` must be expressed in terms of `x`.
- If you specify a sample rate, `Fs`, then `'MinPeakDistance'` must be expressed in units of time.
- If you specify neither `x` nor `Fs`, then `'MinPeakDistance'` must be expressed in units of samples.

Use this argument to have `findpeaks` ignore small peaks that occur in the neighborhood of a larger peak.

Data Types: `single` | `double`

### **'WidthReference' — Reference height for width measurements**

`'halfprom'` (default) | `'halfheight'`

Reference height for width measurements, specified as the comma-separated pair consisting of `'WidthReference'` and either `'halfprom'` or `'halfheight'`.

`findpeaks` estimates the width of a peak as the distance between the points where the descending signal intercepts a horizontal reference line. The height of the line is selected using the criterion specified in `'WidthReference'`:

- `'halfprom'` positions the reference line beneath the peak at a vertical distance equal to half the peak prominence. See “Prominence” on page 1-652 for more information.
- `'halfheight'` positions the reference line at one-half the peak height. The line is truncated if any of its intercept points lie beyond the borders of the peaks selected by setting `'MinPeakHeight'`, `'MinPeakProminence'`, and `'Threshold'`. The border between peaks is defined by the horizontal position of the lowest valley between them. Peaks with height less than zero are discarded.

The locations of the intercept points are computed by linear interpolation.

Data Types: `char`

### **'MinPeakWidth' — Minimum peak width**

`0` (default) | positive real scalar

Minimum peak width, specified as the comma-separated pair consisting of `'MinPeakWidth'` and a positive real scalar. Use this argument to select only those peaks that have widths of at least `'MinPeakWidth'`.

- If you specify a location vector, `x`, then `'MinPeakWidth'` must be expressed in terms of `x`.

- If you specify a sample rate,  $F_s$ , then `'MinPeakWidth'` must be expressed in units of time.
- If you specify neither  $x$  nor  $F_s$ , then `'MinPeakWidth'` must be expressed in units of samples.

Data Types: `single` | `double`

### **'MaxPeakWidth' — Maximum peak width**

`Inf` (default) | positive real scalar

Maximum peak width, specified as the comma-separated pair consisting of `'MaxPeakWidth'` and a positive real scalar. Use this argument to select only those peaks that have widths of at most `'MaxPeakWidth'`.

- If you specify a location vector,  $x$ , then `'MaxPeakWidth'` must be expressed in terms of  $x$ .
- If you specify a sample rate,  $F_s$ , then `'MaxPeakWidth'` must be expressed in units of time.
- If you specify neither  $x$  nor  $F_s$ , then `'MaxPeakWidth'` must be expressed in units of samples.

Data Types: `single` | `double`

### **'Annotate' — Plot style**

`'peaks'` (default) | `'extents'`

Plot style, specified as the comma-separated pair consisting of `'Annotate'` and one of these values:

- `'peaks'` plots the signal and annotates the location and value of every peak.
- `'extents'` plots the signal and annotates the location, value, width, and prominence of every peak.

This argument is ignored if you call `findpeaks` with output arguments.

Data Types: `char`

## **Output Arguments**

### **pks — Local maxima**

vector



Local maxima, returned as a vector of signal values. If there are no local maxima, then `pks` is empty.

Data Types: `single` | `double`

### **locs** — Peak locations

vector

Peak locations, returned as a vector.

- If you specify a location vector, `x`, then `locs` contains the values of `x` at the peak indices.
- If you specify a sample rate, `Fs`, then `locs` is a vector of time instants.
- If you specify neither `x` nor `Fs`, then `locs` is a vector of integer indices.

Data Types: `single` | `double`

### **w** — Peak widths

vector

Peak widths, returned as a vector of real numbers. The width of each peak is computed as the distance between the points to the left and right of the peak where the signal intercepts a reference line whose height is specified by `WidthReference`. The points themselves are found by linear interpolation.

- If you specify a location vector, `x`, then the widths are expressed in terms of `x`.
- If you specify a sample rate, `Fs`, then the widths are expressed in units of time.
- If you specify neither `x` nor `Fs`, then the widths are expressed in units of samples.

Data Types: `single` | `double`

### **p** — Peak prominences

vector

Peak prominences, returned as a vector of real numbers. The prominence of a peak is the minimum vertical distance that the signal must descend on either side of the peak before either climbing back to a level higher than the peak or reaching an endpoint. See “Prominence” on page 1-652 for more information.

Data Types: `single` | `double`

## More About

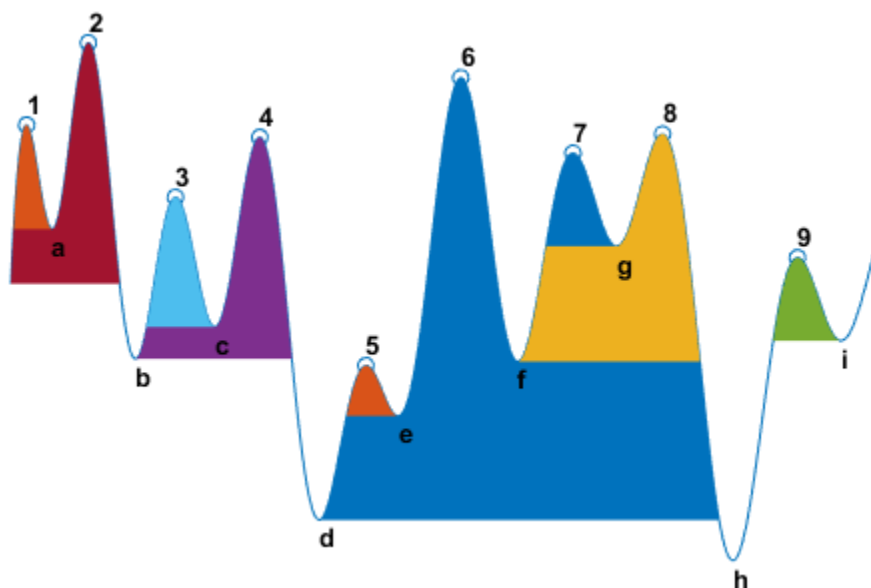
### Prominence

The *prominence* of a peak measures how much the peak stands out due to its intrinsic height and its location relative to other peaks. A low isolated peak can be more prominent than one that is higher but is an otherwise unremarkable member of a tall range.

To measure the prominence of a peak:

- 1 Place a marker on the peak.
- 2 Extend a horizontal line from the peak to the left and right until the line does one of the following:
  - Crosses the signal because there is a higher peak
  - Reaches the left or right end of the signal
- 3 Find the minimum of the signal in each of the two intervals defined in Step 2. This point is either a valley or one of the signal endpoints.
- 4 The higher of the two interval minima specifies the reference level. The height of the peak above this level is its prominence.

`findpeaks` makes no assumption about the behavior of the signal beyond its endpoints, whatever their height. This is reflected in Steps 2 and 4 and often affects the value of the reference level. Consider for example the peaks of this signal:



| Peak Number | Left Interval Lies Between Peak and | Right Interval Lies Between Peak and | Lowest Point on the Left Interval | Lowest Point on the Right Interval | Reference Level (Highest Minimum) |
|-------------|-------------------------------------|--------------------------------------|-----------------------------------|------------------------------------|-----------------------------------|
| 1           | Left end                            | Crossing due to peak 2               | Left endpoint                     | a                                  | a                                 |
| 2           | Left end                            | Right end                            | Left endpoint                     | h                                  | Left endpoint                     |
| 3           | Crossing due to peak 2              | Crossing due to peak 4               | b                                 | c                                  | c                                 |
| 4           | Crossing due to peak 2              | Crossing due to peak 6               | b                                 | d                                  | b                                 |
| 5           | Crossing due to peak 4              | Crossing due to peak 6               | d                                 | e                                  | e                                 |
| 6           | Crossing due to peak 2              | Right end                            | d                                 | h                                  | d                                 |
| 7           | Crossing due to peak 6              | Crossing due to peak 8               | f                                 | g                                  | g                                 |

| <b>Peak Number</b> | <b>Left Interval Lies Between Peak and</b> | <b>Right Interval Lies Between Peak and</b> | <b>Lowest Point on the Left Interval</b> | <b>Lowest Point on the Right Interval</b> | <b>Reference Level (Highest Minimum)</b> |
|--------------------|--|---|--|---|--|
| <b>8</b>           | Crossing due to peak <b>6</b>              | Right end                                   | <b>f</b>                                 | <b>h</b>                                  | <b>f</b>                                 |
| <b>9</b>           | Crossing due to peak <b>8</b>              | Crossing due to right endpoint              | <b>h</b>                                 | <b>i</b>                                  | <b>i</b>                                 |

**See Also**

fminbnd | fminsearch | fzero | max

# fir1

Window-based FIR filter design

## Syntax

```
b = fir1(n,Wn)
b = fir1(n,Wn,ftype)

b = fir1( ____,window)
b = fir1( ____,scaleopt)
```

## Description

`b = fir1(n,Wn)` uses a Hamming window to design an  $n$ th-order lowpass, bandpass, or multiband FIR filter with linear phase. The filter type depends on the number of elements of `Wn`.

`b = fir1(n,Wn,ftype)` designs a lowpass, highpass, bandpass, bandstop, or multiband filter, depending on the value of `ftype` and the number of elements of `Wn`.

`b = fir1( ____,window)` designs the filter using the vector specified in `window` and any of the arguments from previous syntaxes.

`b = fir1( ____,scaleopt)` additionally specifies whether or not the magnitude response of the filter is normalized.

---

**Note** Use `fir2` for windowed filters with arbitrary frequency response.

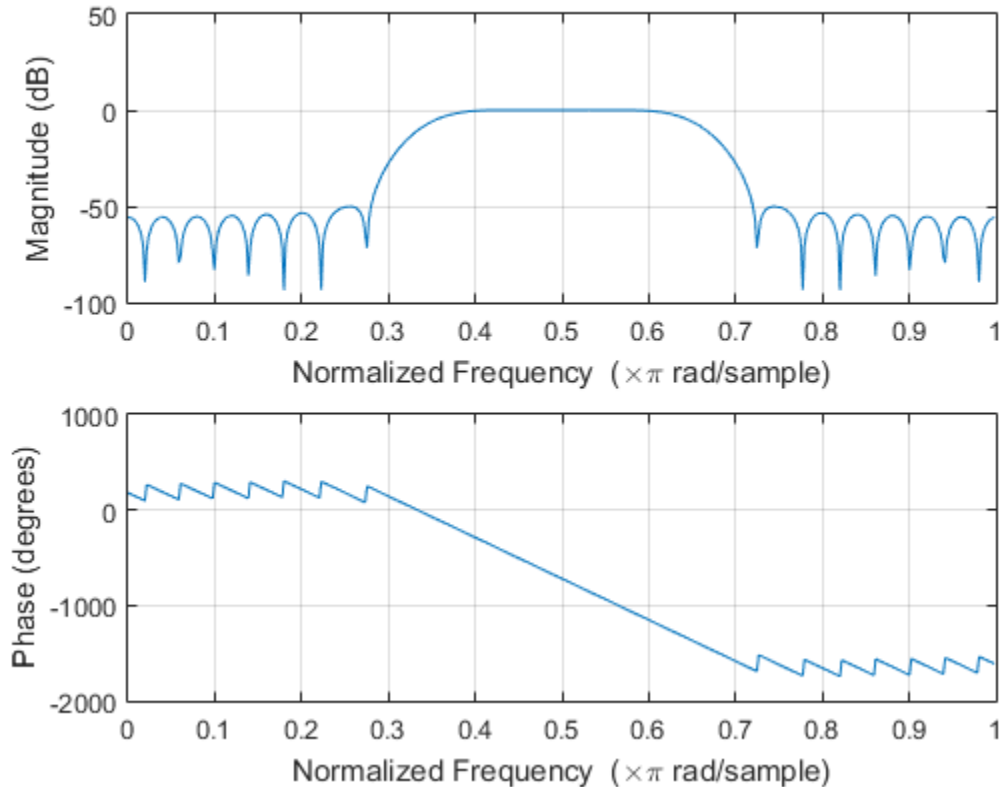
---

## Examples

### FIR Bandpass Filter

Design a 48th-order FIR bandpass filter with passband  $0.35\pi \leq \omega \leq 0.65\pi$  rad/sample. Visualize its magnitude and phase responses.

```
b = fir1(48,[0.35 0.65]);
freqz(b,1,512)
```

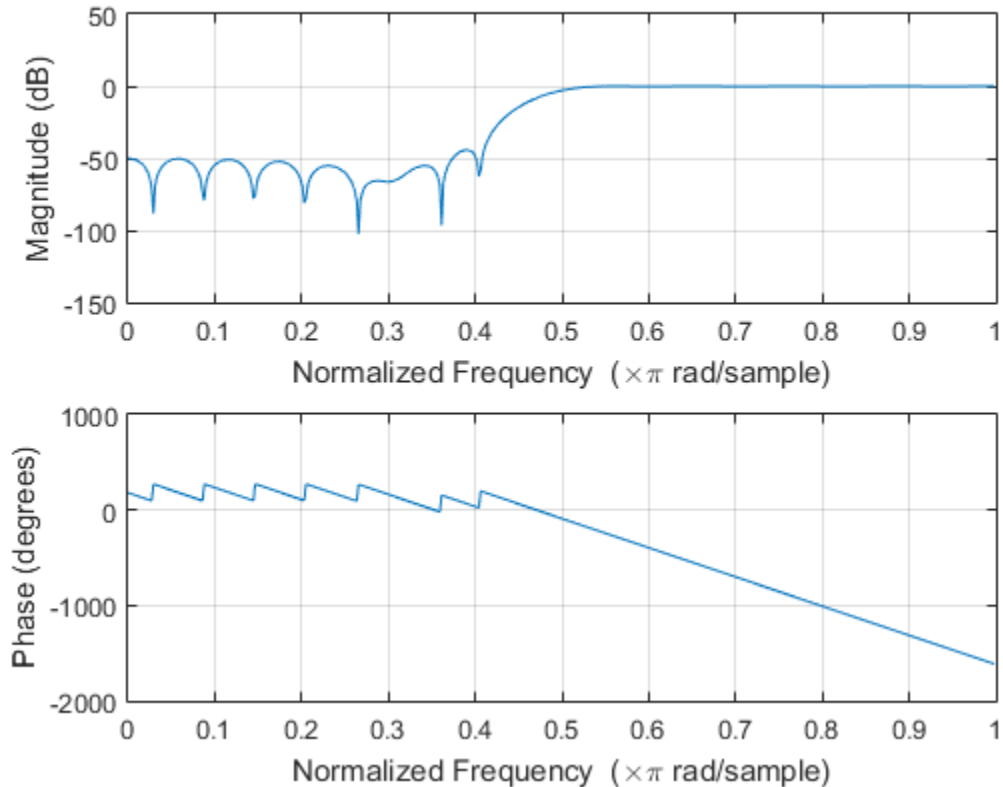


### FIR Highpass Filter

Load `chirp.mat`. The file contains a signal, `y`, that has most of its power above  $F_s/4$ , or half the Nyquist frequency. The sample rate is 8192 Hz.

Design a 34th-order FIR highpass filter to attenuate the components of the signal below  $F_s/4$ . Use a cutoff frequency of 0.48 and a Chebyshev window with 30 dB of ripple.

```
load chirp
t = (0:length(y)-1)/Fs;
bhi = fir1(34,0.48,'high',chebwin(35,30));
freqz(bhi,1)
```



Filter the signal. Display the original and highpass-filtered signals. Use the same y-axis scale for both plots.

```

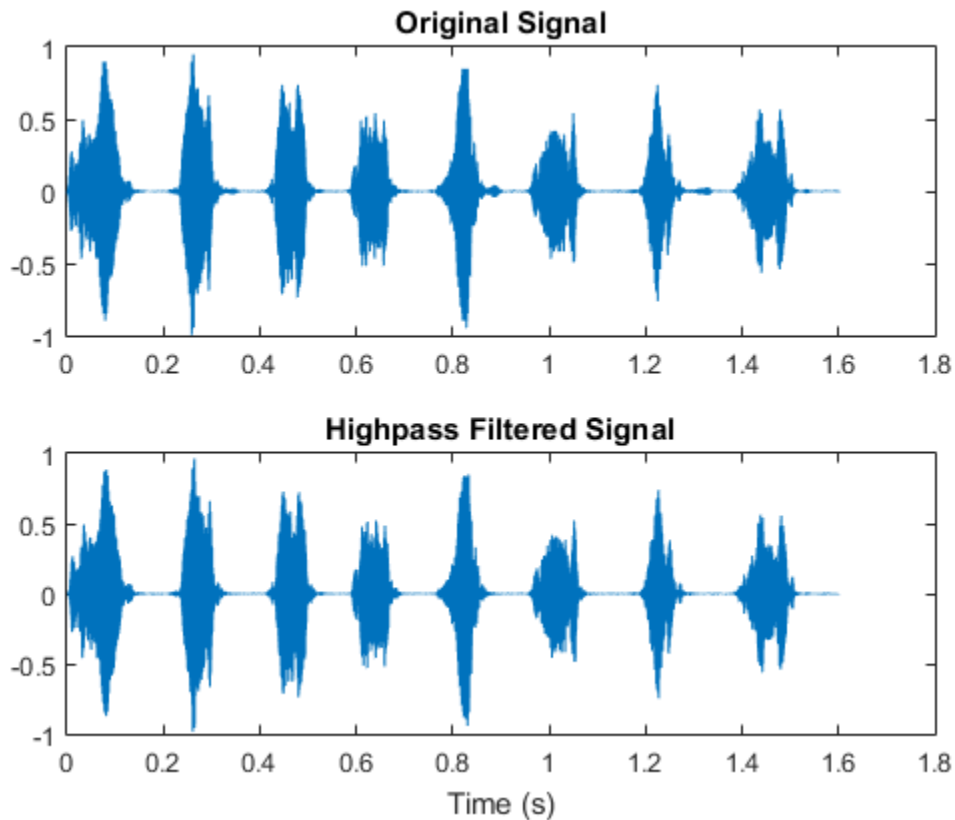
outhi = filter(bhi,1,y);

subplot(2,1,1)
plot(t,y)
title('Original Signal')
ys = ylim;

subplot(2,1,2)
plot(t,outhi)
title('Highpass Filtered Signal')
xlabel('Time (s)')

```

```
ylim(ys)
```

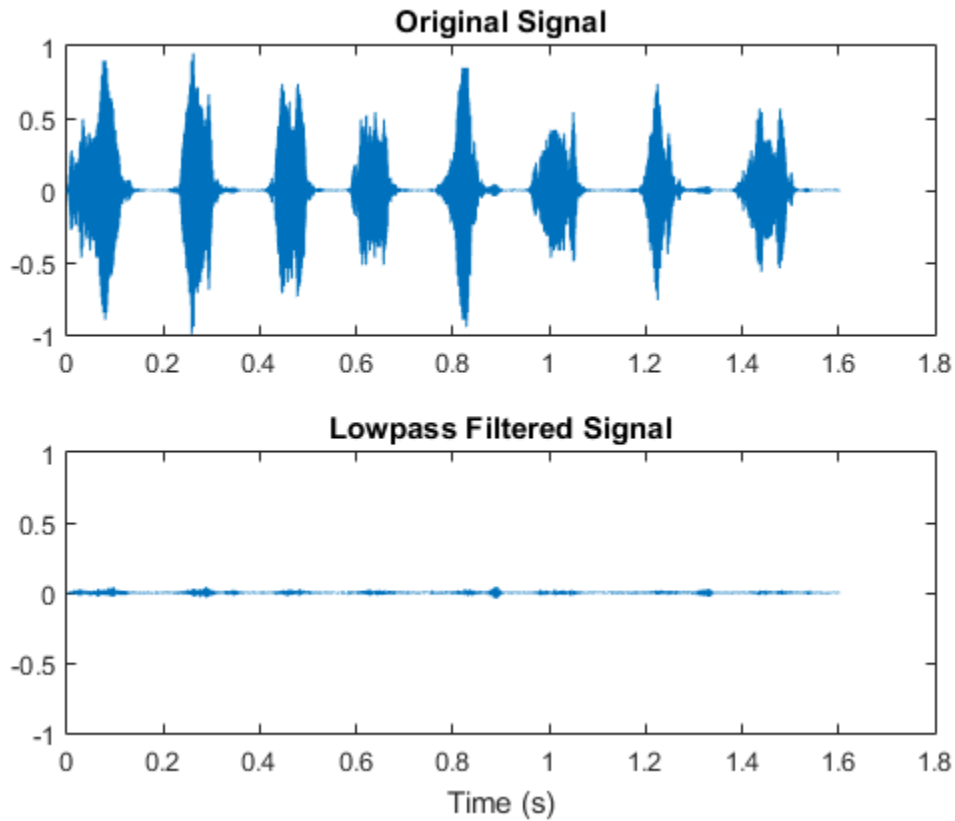


Design a lowpass filter with the same specifications. Filter the signal and compare the result to the original. Use the same y-axis scale for both plots.

```
blo = fir1(34,0.48,chebwin(35,30));  
  
outlo = filter(blo,1,y);  
  
subplot(2,1,1)  
plot(t,y)  
title('Original Signal')  
ys = ylim;
```



```
subplot(2,1,2)
plot(t,outlo)
title('Lowpass Filtered Signal')
xlabel('Time (s)')
ylim(ys)
```



### Multiband FIR Filter

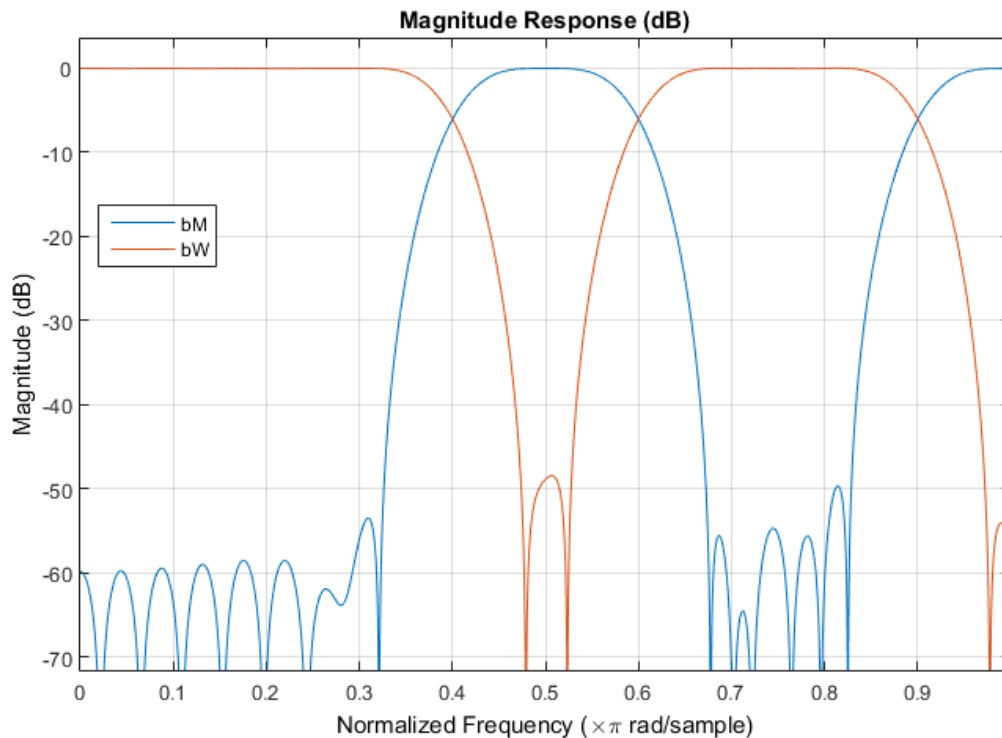
Design a 44th-order FIR filter that attenuates normalized frequencies below  $0.4\pi$  rad/sample and between  $0.6\pi$  and  $0.9\pi$  rad/sample. Call it `bm`.

```
ord = 44;
```

```
low = 0.4;  
bnd = [0.6 0.9];  
  
bM = fir1(ord,[low bnd]);
```

Redesign `bM` so that it passes the bands it was attenuating and stops the other frequencies. Call the new filter `bW`. Use `fvtool` to display the frequency responses of the filters.

```
bW = fir1(ord,[low bnd], 'DC-1');  
  
hfvt = fvtool(bM,1,bW,1);  
legend(hfvt, 'bM', 'bW')
```



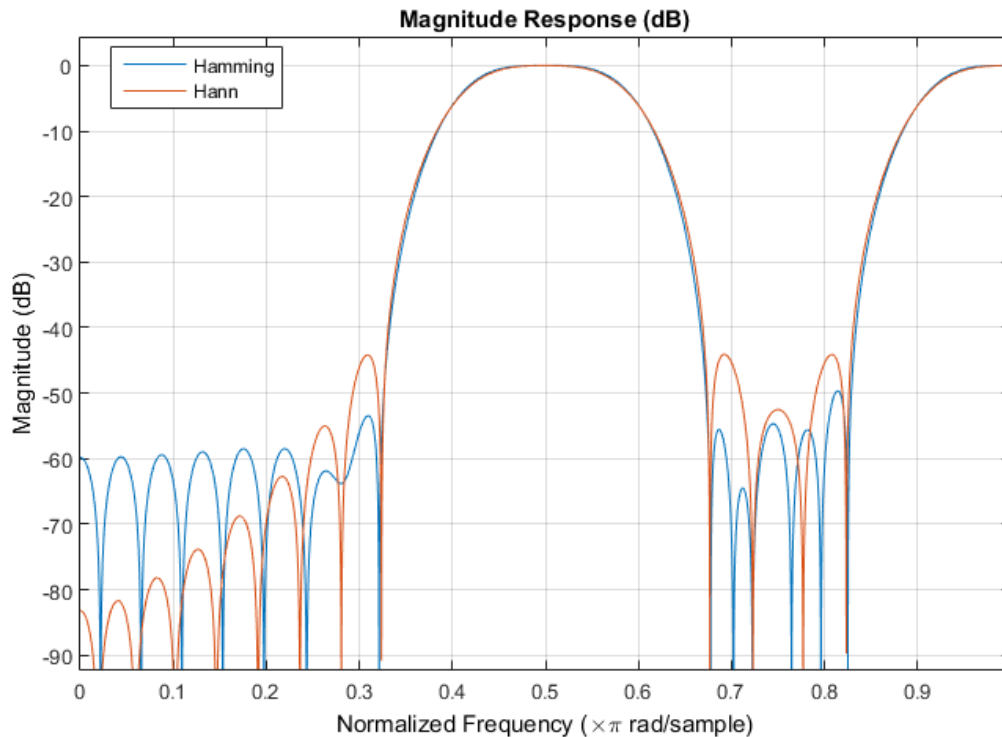
Redesign `bM` using a Hann window. (The string 'DC-0' is optional.) Compare the magnitude responses of the Hamming and Hann designs.

```

hM = fir1(ord,[low bnd],'DC-0',hann(ord+1));

hfvt = fvtool(bM,1,hM,1);
legend(hfvt,'Hamming','Hann')

```



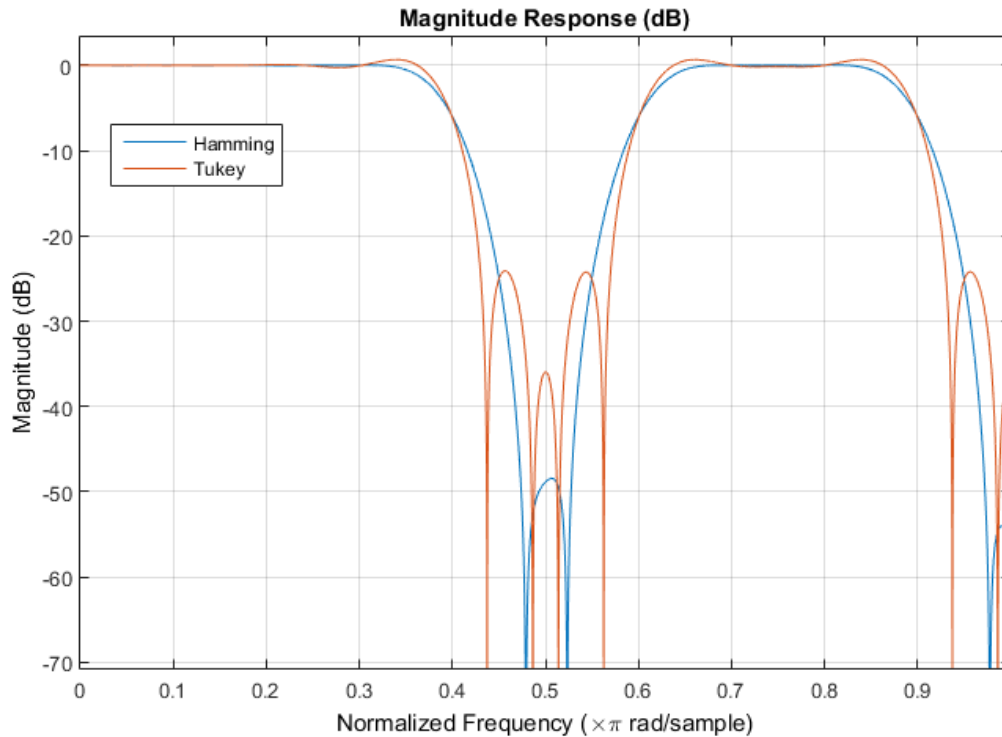
Redesign bW using a Tukey window. Compare the magnitude responses of the Hamming and Tukey designs.

```

tW = fir1(ord,[low bnd],'DC-1',tukeywin(ord+1));

hfvt = fvtool(bW,1,tW,1);
legend(hfvt,'Hamming','Tukey')

```



## Input Arguments

### **n** — Filter order

integer scalar

Filter order, specified as an integer scalar.

For highpass and bandstop configurations, `fir1` always uses an even filter order. The order must be even because odd-order symmetric FIR filters must have zero gain at the Nyquist frequency. If you specify an odd `n` for a highpass or bandstop filter, then `fir1` increments `n` by 1.

Data Types: double

**Wn — Frequency constraints**

scalar | two-element vector | multi-element vector

Frequency constraints, specified as a scalar, a two-element vector, or a multi-element vector. All elements of  $W_n$  must be strictly greater than 0 and strictly smaller than 1, where 1 corresponds to the Nyquist frequency:  $0 < W_n < 1$ . The Nyquist frequency is half the sample rate or  $\pi$  rad/sample.

- If  $W_n$  is a scalar, then `fir1` designs a lowpass or highpass filter with cutoff frequency  $W_n$ . The cutoff frequency is the frequency at which the normalized gain of the filter is  $-6$  dB.
- If  $W_n$  is the two-element vector  $[w_1 \ w_2]$ , where  $w_1 < w_2$ , then `fir1` designs a bandpass or bandstop filter with lower cutoff frequency  $w_1$  and higher cutoff frequency  $w_2$ .
- If  $W_n$  is the multi-element vector  $[w_1 \ w_2 \ \dots \ w_n]$ , where  $w_1 < w_2 < \dots < w_n$ , then `fir1` returns an  $n$ th-order multiband filter with bands  $0 < \omega < w_1$ ,  $w_1 < \omega < w_2$ , ...,  $w_n < \omega < 1$ .

Data Types: double

**ftype — Filter type**

'low' | 'bandpass' | 'high' | 'stop' | 'DC-0' | 'DC-1'

Filter type, specified as a string.

- 'low' specifies a lowpass filter with cutoff frequency  $W_n$ . 'low' is the default for scalar  $W_n$ .
- 'high' specifies a highpass filter with cutoff frequency  $W_n$ .
- 'bandpass' specifies a bandpass filter if  $W_n$  is a two-element vector. 'bandpass' is the default when  $W_n$  has two elements.
- 'stop' specifies a bandstop filter if  $W_n$  is a two-element vector.
- 'DC-0' specifies that the first band of a multiband filter is a stopband. 'DC-0' is the default when  $W_n$  has more than two elements.
- 'DC-1' specifies that the first band of a multiband filter is a passband.

Data Types: char

**window — Window**

vector

Window, specified as a vector. The window vector must have  $n + 1$  elements. If you do not specify window, then `fir1` uses a Hamming window. For a list of available windows, see “Windows”.

`fir1` does not automatically increase the length of window if you attempt to design a highpass or bandstop filter of odd order.

Example: `kaiser(n+1,0.5)` specifies a Kaiser window with shape parameter 0.5 to use with a filter of order  $n$ .

Example: `hamming(n+1)` is equivalent to leaving the window unspecified.

Data Types: double

### **scaleopt — Normalization option**

'scale' (default) | 'noscale'

Normalization option, specified as a string.

- 'scale' normalizes the coefficients so that the magnitude response of the filter at the center of the passband is 1 (0 dB).
- 'noscale' does not normalize the coefficients.

Data Types: char

## **Output Arguments**

### **b — Filter coefficients**

row vector

Filter coefficients, returned as a row vector of length  $n + 1$ . The coefficients are sorted in descending powers of the Z-transform variable  $z$ :

$$B(z) = b(1) + b(2)z + \dots + b(n+1)z^{-n}.$$

## **More About**

### **Algorithms**

`fir1` uses the window method of FIR filter design. If the impulse response of an ideal filter is  $h(n)$ , and  $w(n)$  denotes a window function, then the filter coefficients are given

by  $b(n) = w(n)h(n)$ , where  $1 \leq n \leq N$ . The window truncates the impulse response and attenuates the resulting truncation oscillations.

## References

- [1] Digital Signal Processing Committee of the IEEE Acoustics, Speech, and Signal Processing Society, eds. *Programs for Digital Signal Processing*. New York: IEEE Press, 1979, Algorithm 5.2.

## See Also

cfirpm | designfilt | filter | fir2 | fircls | fircls1 | fir1s | firpm |  
freqz | hamming | kaiserord | window

## **fir2**

Frequency sampling-based FIR filter design

### **Syntax**

```
b = fir2(n,f,m)
b = fir2(n,f,m,npt,lap)
b = fir2( ____,window)
```

### **Description**

`b = fir2(n,f,m)` returns an *n*th-order FIR filter with frequency-magnitude characteristics specified in the vectors `f` and `m`. The function linearly interpolates the desired frequency response onto a dense grid and then uses the inverse Fourier transform and a Hamming window to obtain the filter coefficients.

`b = fir2(n,f,m,npt,lap)` specifies `npt`, the number of points in the interpolation grid, and `lap`, the length of the region that `fir2` inserts around duplicate frequency points which specify steps in the frequency response.

`b = fir2( ____,window)` specifies a window vector to use in the design in addition to any input arguments from previous syntaxes.

---

**Note** Use `fir1` for window-based standard lowpass, bandpass, highpass, bandstop, and multiband configurations.

---

### **Examples**

#### **Attenuation of Low Frequencies**

Load the MAT-file `chirp`. The file contains a signal, `y`, sampled at a frequency  **$F_s = 8192$  Hz**. `y` has most of its power above  $F_s/4$ , or half the Nyquist frequency. Add random noise to the signal.

```
load chirp
```

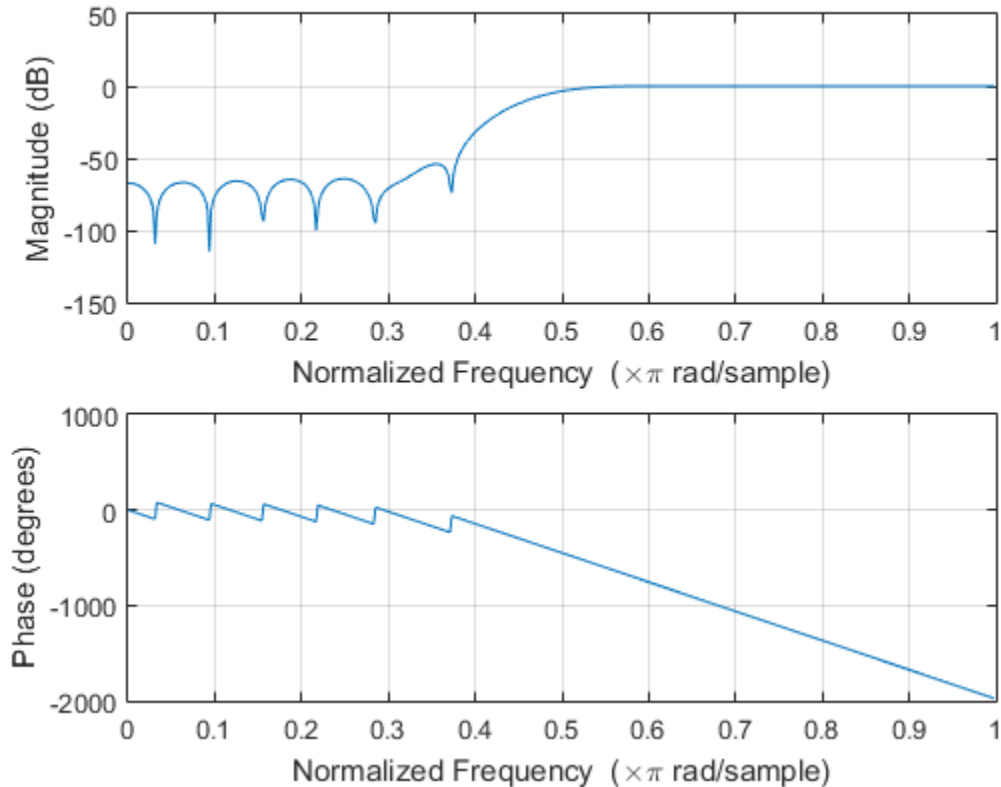


```
y = y + 0.25*(rand(size(y))-0.5);
```

Design a 34th-order FIR highpass filter to attenuate the components of the signal below  $F_s/4$ . Specify a cutoff frequency of 0.48. Visualize the frequency response of the filter.

```
f = [0 0.48 0.48 1];
mhi = [0 0 1 1];
bhi = fir2(34,f,mhi);
```

```
freqz(bhi,1)
```

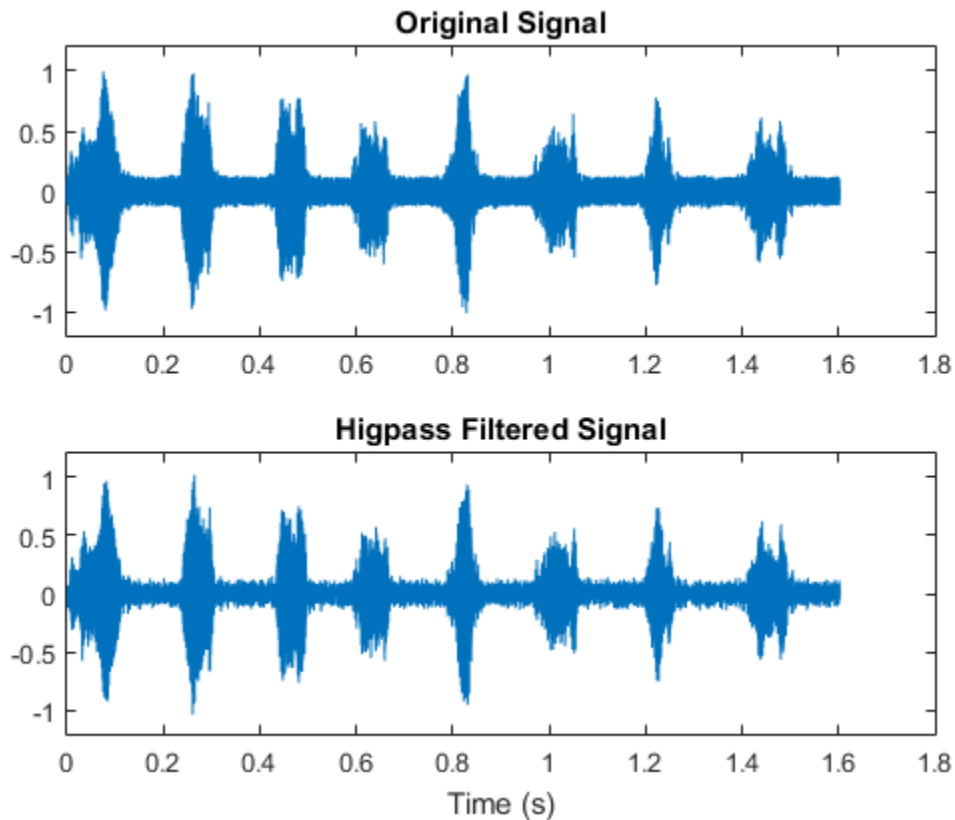


Filter the chirp signal. Plot the signal before and after filtering.

```
outhi = filter(bhi,1,y);
t = (0:length(y)-1)/Fs;
```

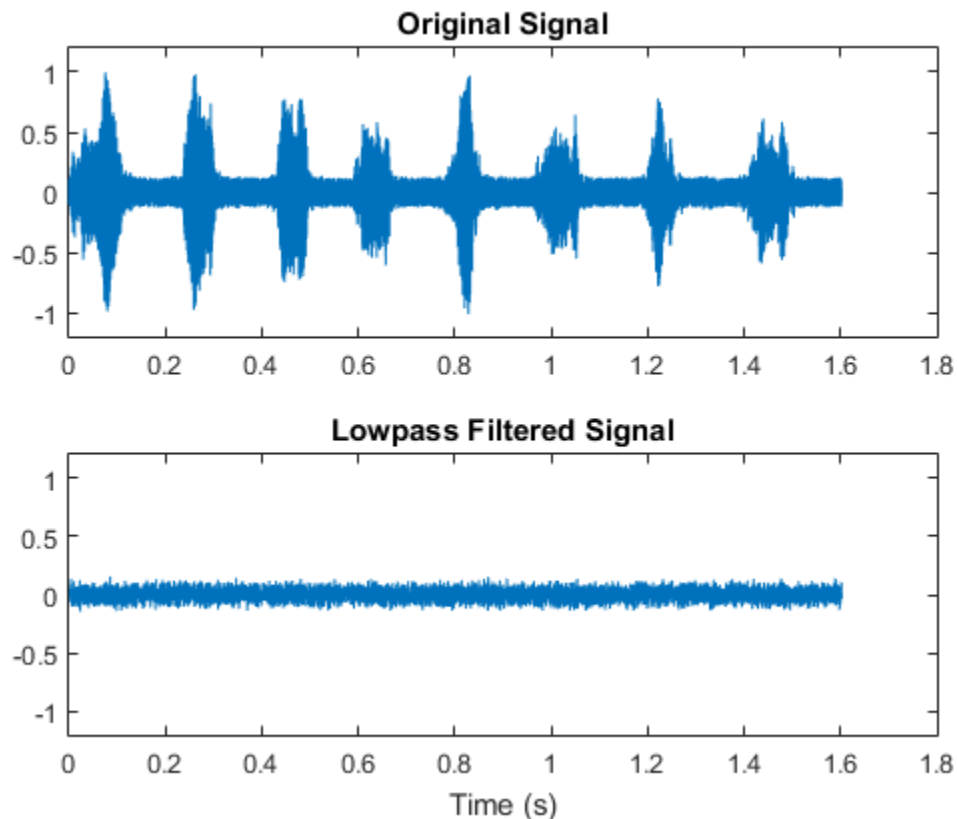
```
subplot(2,1,1)
plot(t,y)
title('Original Signal')
ylim([-1.2 1.2])

subplot(2,1,2)
plot(t,outhi)
title('Higpass Filtered Signal')
xlabel('Time (s)')
ylim([-1.2 1.2])
```



Change the filter from highpass to lowpass. Use the same order and cutoff. Filter the signal again. The result is mostly noise.

```
mlo = [1 1 0 0];  
blo = fir2(34,f,mlo);  
outlo = filter(blo,1,y);  
  
subplot(2,1,1)  
plot(t,y)  
title('Original Signal')  
ylim([-1.2 1.2])  
  
subplot(2,1,2)  
plot(t,outlo)  
title('Lowpass Filtered Signal')  
xlabel('Time (s)')  
ylim([-1.2 1.2])
```

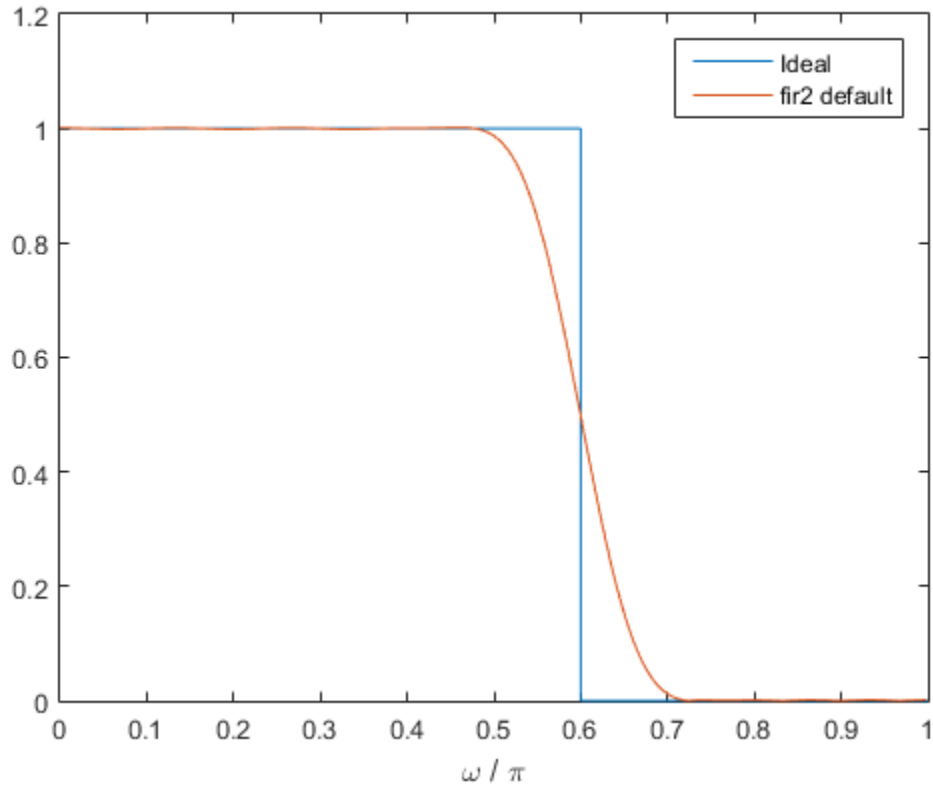


### FIR Lowpass Filter

Design a 30th-order lowpass filter with a normalized cutoff frequency of  $0.6\pi$  rad/sample. Plot the ideal frequency response overlaid with the actual frequency response.

```
f = [0 0.6 0.6 1];  
m = [1 1 0 0];  
  
b1 = fir2(30,f,m);  
[h1,w] = freqz(b1,1);  
  
plot(f,m,w/pi,abs(h1))  
xlabel('\omega / \pi');  
lgs = {'Ideal', 'fir2 default'};
```

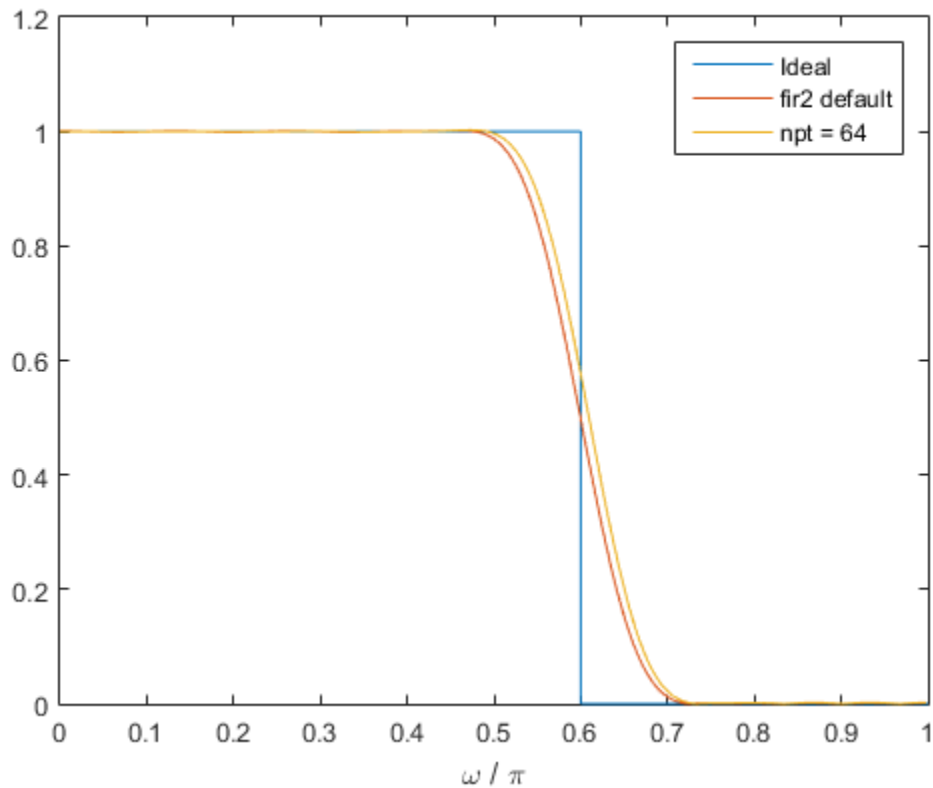
```
legend(lgs)
```



Redesign the filter using a 64-point interpolation grid.

```
b2 = fir2(30,f,m,64);  
h2 = freqz(b2,1);
```

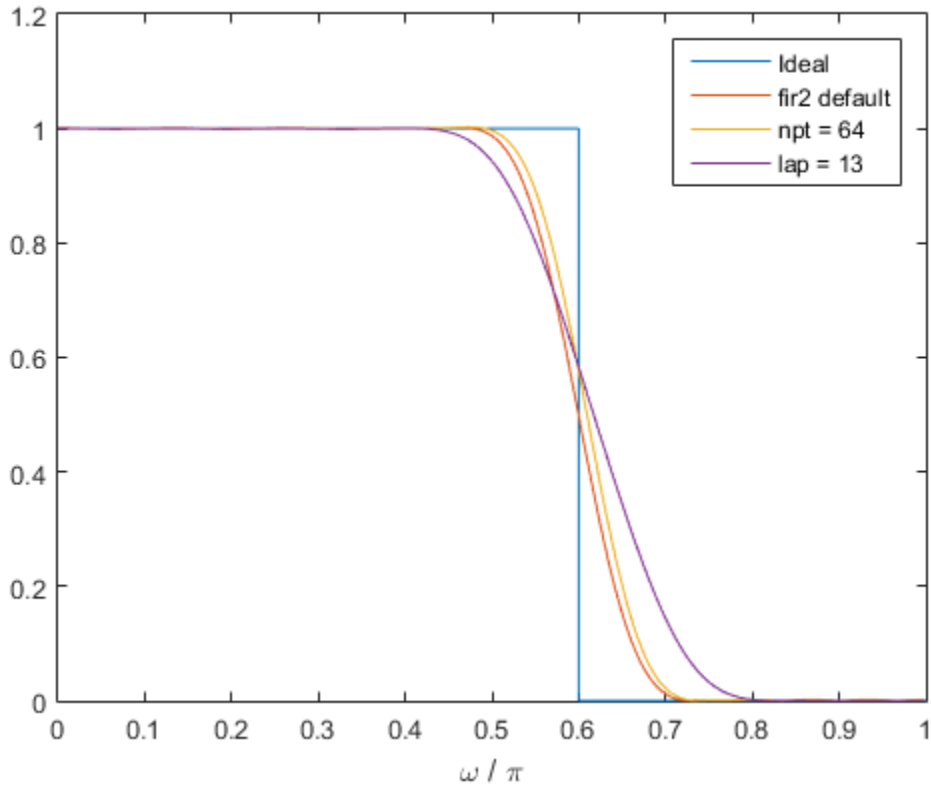
```
hold on  
plot(w/pi,abs(h2))  
lgs{3} = 'npt = 64';  
legend(lgs)
```



Redesign the filter using the 64-point interpolation grid and a 13-point interval around the cutoff frequency.

```
b3 = fir2(30,f,m,64,13);  
h3 = freqz(b3,1);
```

```
plot(w/pi,abs(h3))  
lgs{4} = 'lap = 13';  
legend(lgs)
```



### Arbitrary Magnitude Filter

Design an FIR filter with the following frequency response:

- A sinusoid between 0 and  $0.18\pi$  rad/sample.

```
F1 = 0:0.01:0.18;
A1 = 0.5+sin(2*pi*7.5*F1)/4;
```

- A piecewise linear section between  $0.2\pi$  rad/sample and  $0.78\pi$  rad/sample.

```
F2 = [0.2 0.38 0.4 0.55 0.562 0.585 0.6 0.78];
A2 = [0.5 2.3 1 1 -0.2 -0.2 1 1];
```

- A quadratic section between  $0.79\pi$  rad/sample and the Nyquist frequency.

```
F3 = 0.79:0.01:1;  
A3 = 0.2+18*(1-F3).^2;
```

Design the filter using a Hamming window. Specify a filter order of 50.

```
N = 50;  
  
FreqVect = [F1 F2 F3];  
AmplVect = [A1 A2 A3];  
  
ham = fir2(N,FreqVect,AmplVect);
```

Repeat the calculation using a Kaiser window that has a shape parameter of 3.

```
kai = fir2(N,FreqVect,AmplVect,kaiser(N+1,3));
```

Redesign the filter using the `designfilt` function. `designfilt` uses a rectangular window by default. Compute the zero-phase response of the filter over 1024 points.

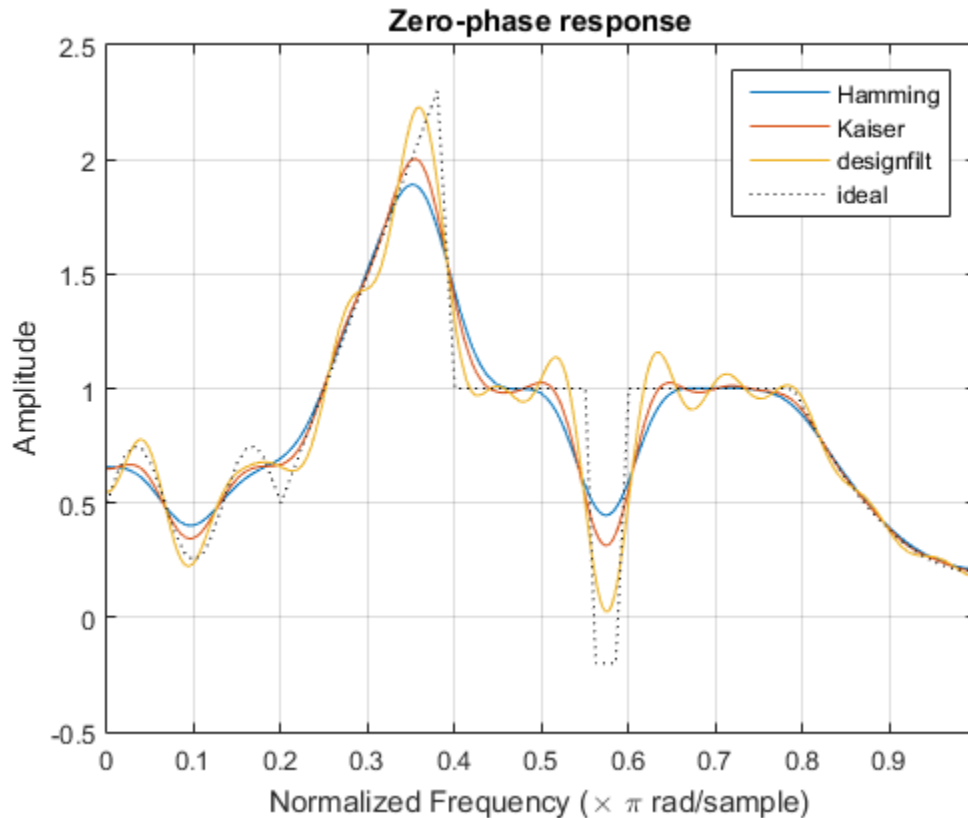
```
d = designfilt('arbmagfir','FilterOrder',N, ...  
             'Frequencies',FreqVect,'Amplitudes',AmplVect);
```

```
[zd,wd] = zerophase(d,1024);
```

Display the zero-phase responses of the three filters. Overlay the ideal response.

```
zerophase(ham,1)  
hold on  
zerophase(kai,1)  
plot(wd/pi,zd)  
plot(FreqVect,AmplVect,'k:')  
legend('Hamming','Kaiser','designfilt','ideal')
```





## Input Arguments

### **n** — Filter order

integer scalar

Filter order, specified as an integer scalar.

For configurations with a passband at the Nyquist frequency, `fir2` always uses an even order. If you specify an odd-valued `n` for one of those configurations, then `fir2` increments `n` by 1.

Data Types: double

**f, m — Frequency-magnitude characteristics**

vectors

Frequency-magnitude characteristics, specified as vectors of the same length.

- **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 must be 1. **f** must be sorted in increasing order. Duplicate frequency points are allowed and are treated as steps in the frequency response.
- **m** is a vector containing the desired magnitude response at each of the points specified in **f**.

Data Types: double

**npt — Number of grid points**

512 (default) | positive integer scalar

Number of grid points, specified as a positive integer scalar. **npt** must be larger than one-half the filter order:  $npt > n/2$ .

Data Types: double

**lap — Length of region around duplicate frequency points**

25 (default) | positive integer scalar

Length of region around duplicate frequency points, specified as a positive integer scalar.

Data Types: double

**window — Window**

column vector

Window, specified as a column vector. The window vector must have  $n + 1$  elements. If you do not specify **window**, then `fir2` uses a Hamming window. For a list of available windows, see “Windows”.

`fir2` does not automatically increase the length of window if you attempt to design a filter of odd order with a passband at the Nyquist frequency.

Example: `kaiser(n+1,0.5)` specifies a Kaiser window with shape parameter 0.5 to use with a filter of order  $n$ .

Example: `hamming(n+1)` is equivalent to leaving the window unspecified.

Data Types: double

## Output Arguments

### **b** — Filter coefficients

row vector

Filter coefficients, returned as a row vector of length  $n + 1$ . The coefficients are sorted in descending powers of the Z-transform variable  $z$ :

$$B(z) = b(1) + b(2)z + \dots + b(n+1)z^{-n}.$$

## More About

### Algorithms

`fir2` uses frequency sampling to design filters. The function interpolates the desired frequency response linearly onto a dense, evenly spaced grid of length `npt`. `fir2` also sets up regions of lap points around repeated values of `f` to provide steep but smooth transitions. To obtain the filter coefficients, the function applies an inverse fast Fourier transform to the grid and multiplies by window.

## References

- [1] Mitra, Sanjit K. *Digital Signal Processing: A Computer Based Approach*. New York: McGraw-Hill, 1998.
- [2] Jackson, L. B. *Digital Filters and Signal Processing*. 3rd Ed. Boston: Kluwer Academic Publishers, 1996.

## See Also

`butter` | `cheby1` | `cheby2` | `designfilt` | `ellip` | `fir1` | `firpm` | `hamming` | `maxflat` | `yulewalk`

## **fircls**

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.

---

**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.

---

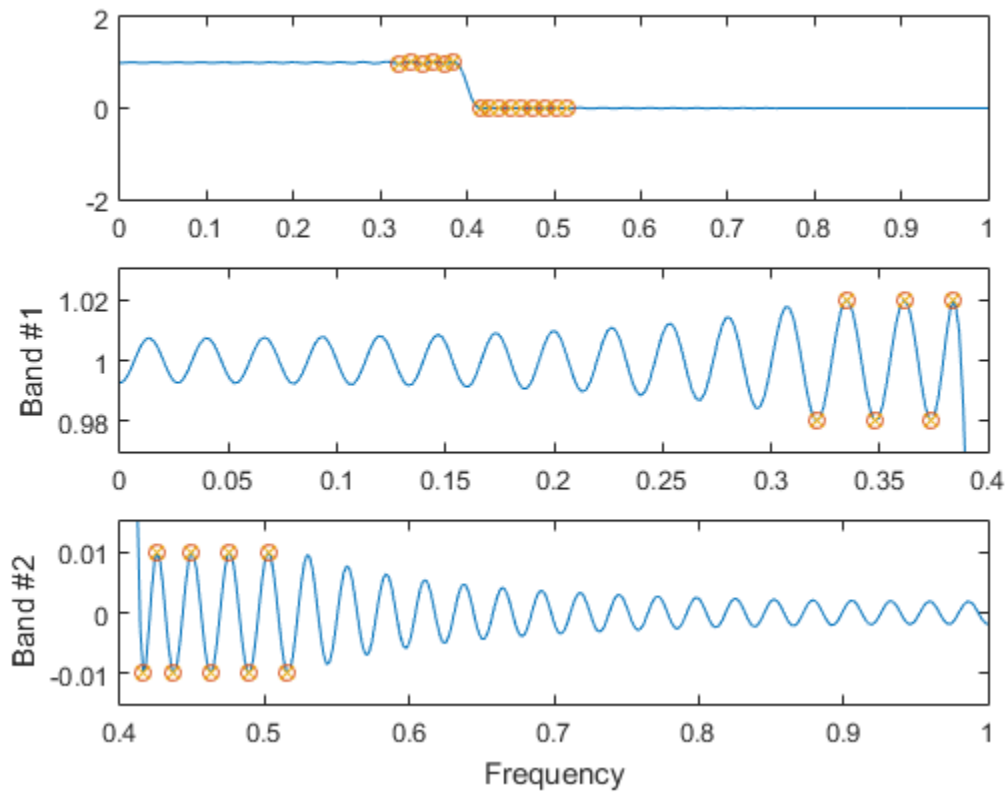
## Examples

### Constrained Least Square Lowpass Filter

Design an order 150 lowpass filter with normalized cutoff frequency 0.4. Specify a maximum absolute error of 0.02 in the passband and 0.01 in the stopband. Display plots of the bands.

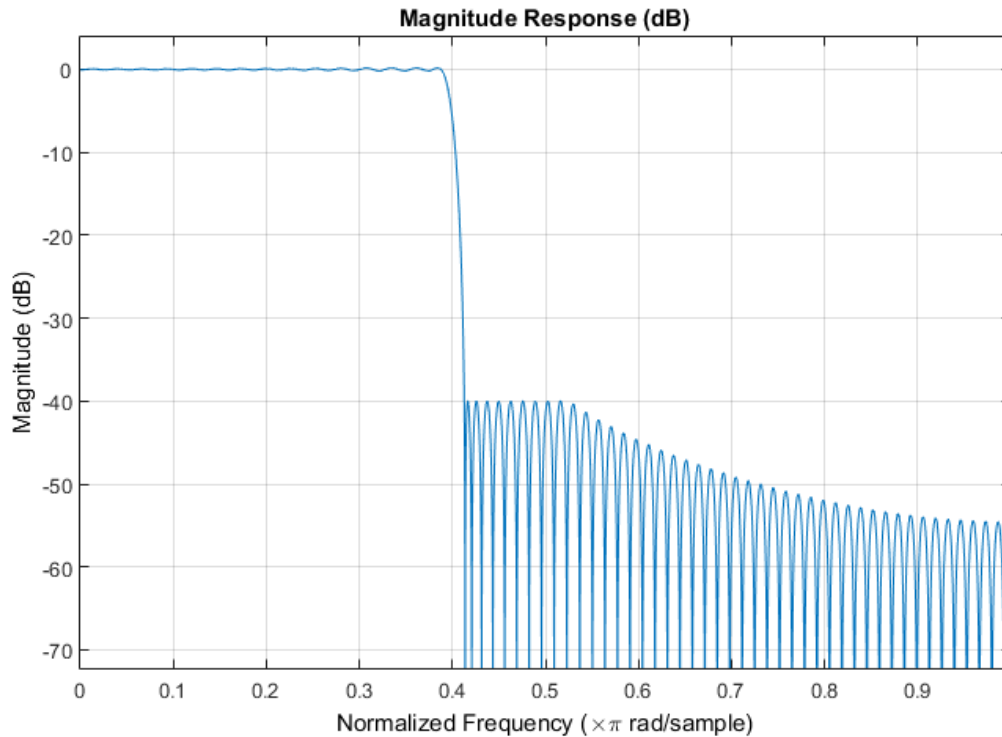
```
n = 150;
f = [0 0.4 1];
a = [1 0];
up = [1.02 0.01];
lo = [0.98 -0.01];
b = fircls(n,f,a,up,lo,'both');
```

```
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 Bound Violations denote the iterations of the procedure as the design converges. Display the magnitude response of the filter.

`fvtool(b)`



## More About

### Algorithms

`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 1995*

*International Conference on Acoustics, Speech, and Signal Processing*. Vol. 2, 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, Number 8, 1996, pp. 1879–1892.

### **See Also**

`fircls1` | `firls` | `firpm`



# fircls1

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  $0 < \omega < wt$ .

- $0 < \omega_0 < \omega_t < 1$ : the amplitude of the filter is within  $\delta_s$  of 0 over the frequency range  $\omega_t < \omega < 1$ .
- Highpass:
  - $0 < \omega_t < \omega_0 < 1$ : the amplitude of the filter is within  $\delta_s$  of 0 over the frequency range  $0 < \omega < \omega_t$ .
  - $0 < \omega_0 < \omega_t < 1$ : the amplitude of the filter is within  $\delta_p$  of 1 over the frequency range  $\omega_t < \omega < 1$ .

$\mathbf{b} = \text{fircls1}(n, \omega_0, \delta_p, \delta_s, \omega_p, \omega_s, k)$  generates a lowpass FIR filter  $\mathbf{b}$  with a weighted function, where  $n+1$  is the filter length,  $\omega_0$  is the normalized cutoff frequency,  $\delta_p$  is the maximum passband deviation from 1 (passband ripple), and  $\delta_s$  is the maximum stopband deviation from 0 (stopband ripple).  $\omega_p$  is the passband edge of the L2 weight function and  $\omega_s$  is the stopband edge of the L2 weight function, where  $\omega_p < \omega_0 < \omega_s$ .  $k$  is the ratio (passband L2 error)/(stopband L2 error)

$$k = \frac{\int_0^{\omega_p} |A(\omega) - D(\omega)|^2 d\omega}{\int_{\omega_s}^{\pi} |A(\omega) - D(\omega)|^2 d\omega}$$

$\mathbf{b} = \text{fircls1}(n, \omega_0, \delta_p, \delta_s, \omega_p, \omega_s, k, \text{'high'})$  generates a highpass FIR filter  $\mathbf{b}$  with a weighted function, where  $\omega_s < \omega_0 < \omega_p$ .

$\mathbf{b} = \text{fircls1}(n, \omega_0, \delta_p, \delta_s, \dots, \text{'design\_flag'})$  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  $\delta_p$  and  $\delta_s$ , there may not exist a filter of the given length that meets the specifications.

---

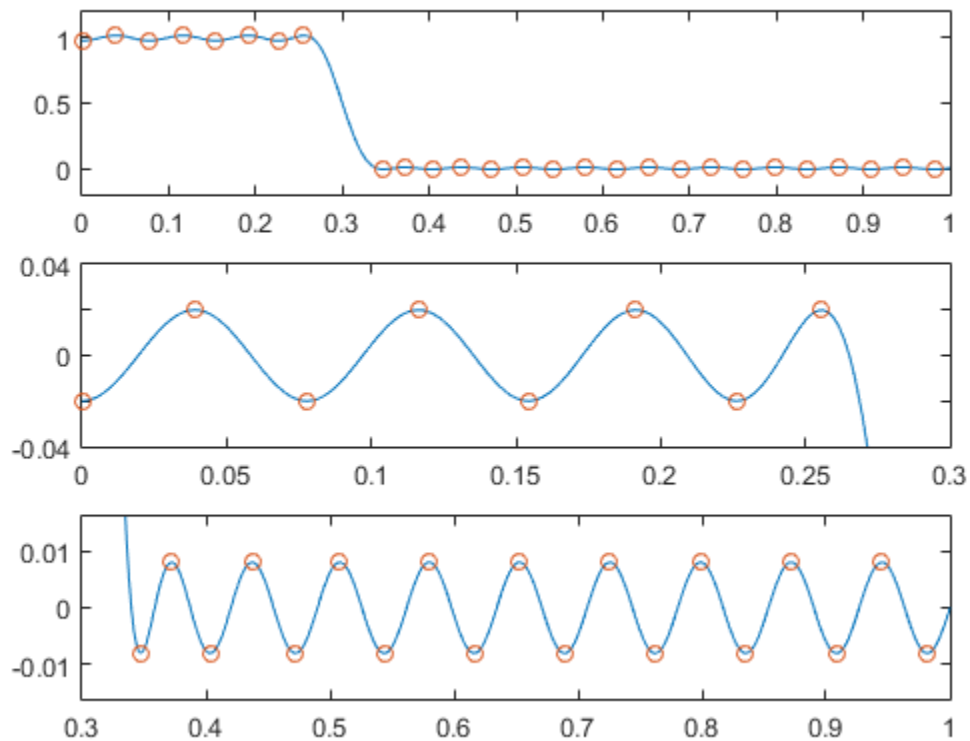
## Examples

### Filter Design with `fircls1`

Design an order 55 lowpass filter with normalized cutoff frequency 0.3. Specify a passband ripple of 0.02 and a stopband ripple of 0.008. Display plots of the bands.

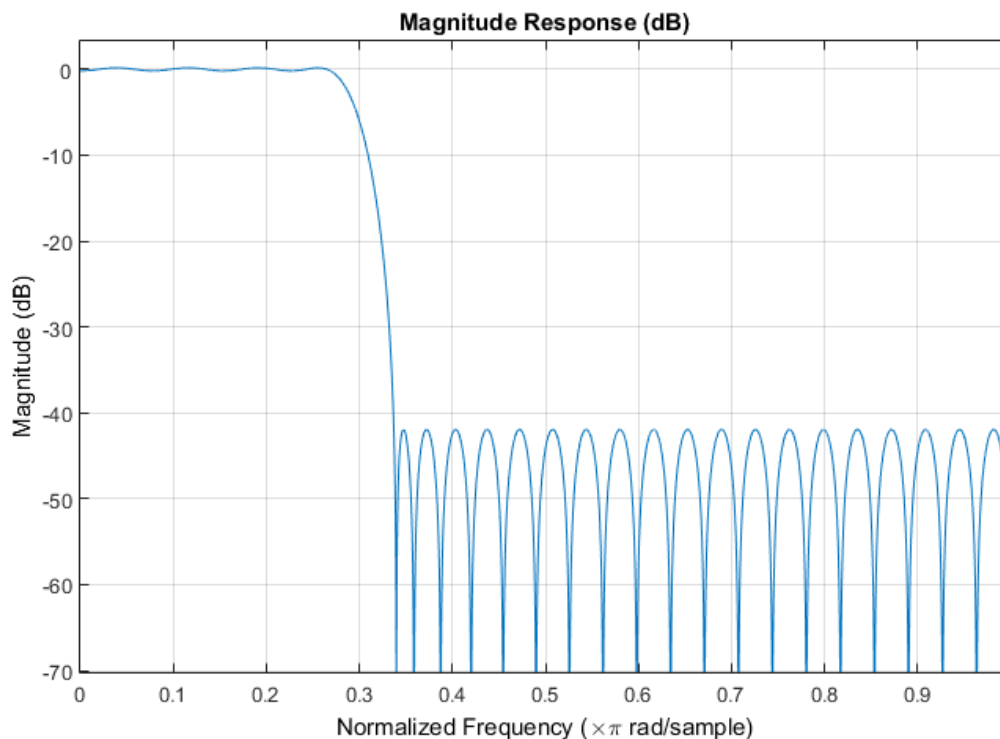
```
n = 55;
wo = 0.3;
dp = 0.02;
ds = 0.008;
b = fircls1(n,wo,dp,ds, 'both');

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 Bound Violations denote the iterations of the procedure as the design converges. Display the magnitude response of the filter.

`fvtool(b)`



## More About

### Algorithms

`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 1995*

*International Conference on Acoustics, Speech, and Signal Processing*. Vol. 2, 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, Number 8, 1996, pp. 1879–1892.

### **See Also**

`fircls` | `firls` | `firpm`

# firls

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.

$$b(k) = b(n+2-k), \quad k = 1, \dots, n+1$$

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`.

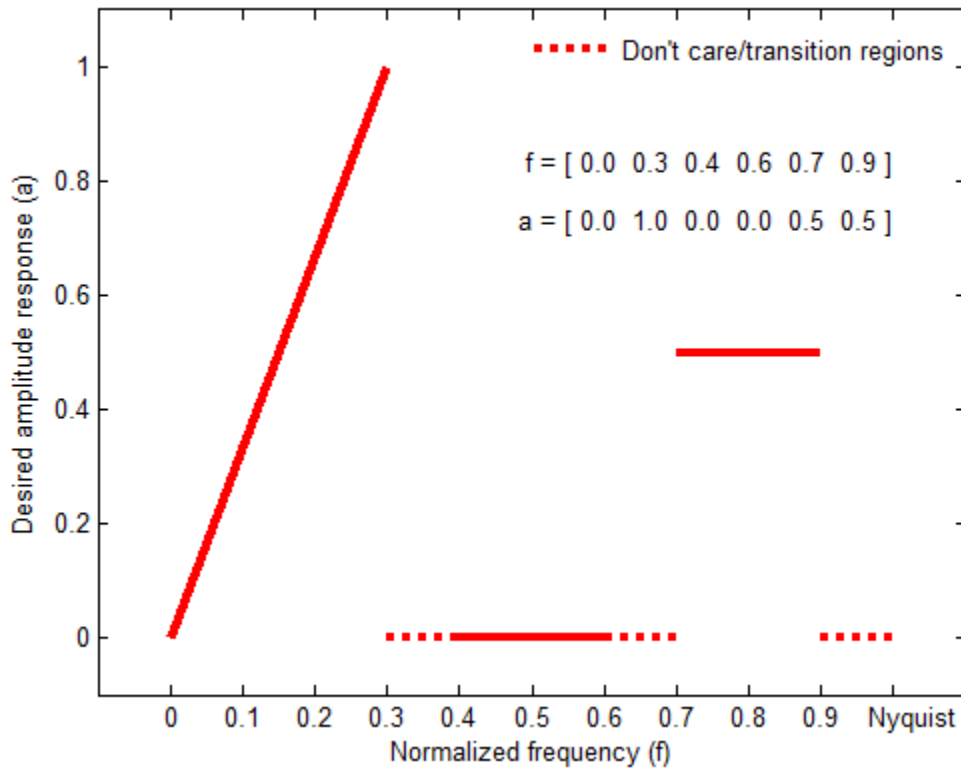
The desired amplitude function at frequencies between pairs of points ( $f(k)$ ,  $f(k+1)$ ) for  $k$  odd is the line segment connecting the points ( $f(k)$ ,  $a(k)$ ) and ( $f(k+1)$ ,  $a(k+1)$ ).

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

$$b(k) = -b(n+2-k), \quad k = 1, \dots, n+1.$$

This class of filters includes the Hilbert transformer, which has a desired amplitude of 1 across the entire band.

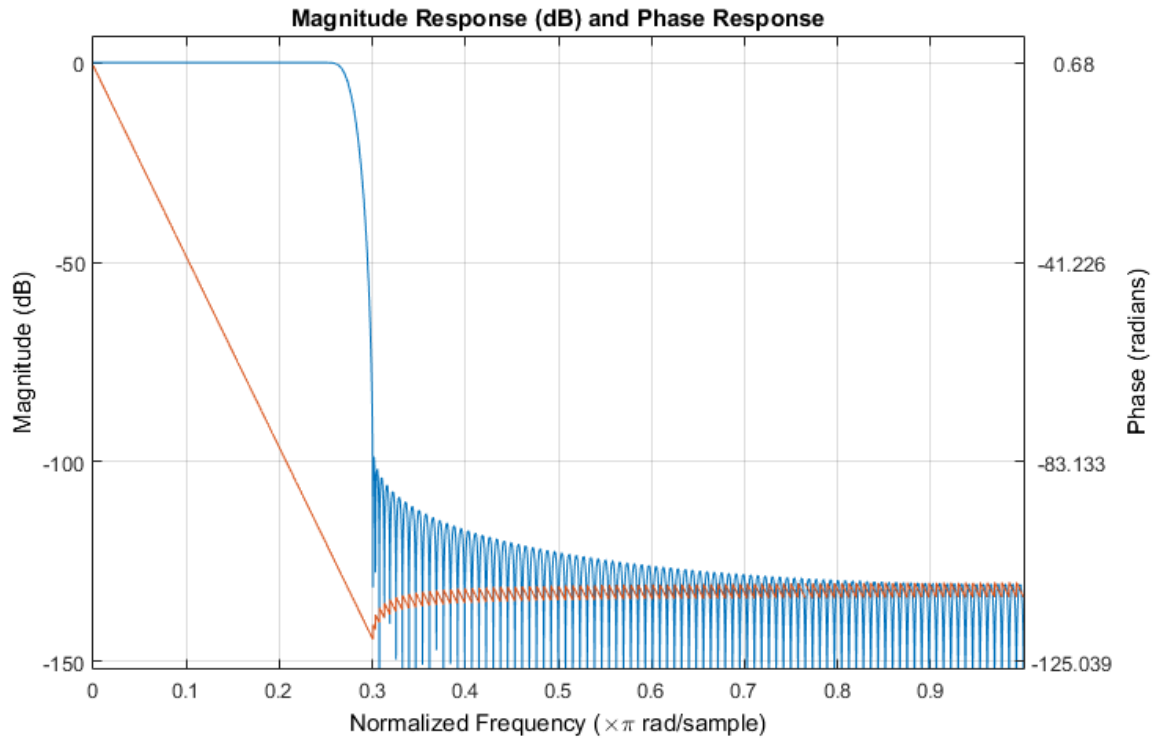
- `'differentiator'` for type III and type IV filters, using a special weighting technique. For nonzero amplitude bands, the integrated squared error has a weight of  $(1/f)^2$  so that the error at low frequencies is much smaller than at high frequencies. For FIR differentiators, which have an amplitude 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

### Filter with Transition Band

Design an FIR lowpass filter of order 255 with transition band between  $0.25\pi$  and  $0.3\pi$ . Use `fvtool` to display the magnitude and phase responses of the filter.

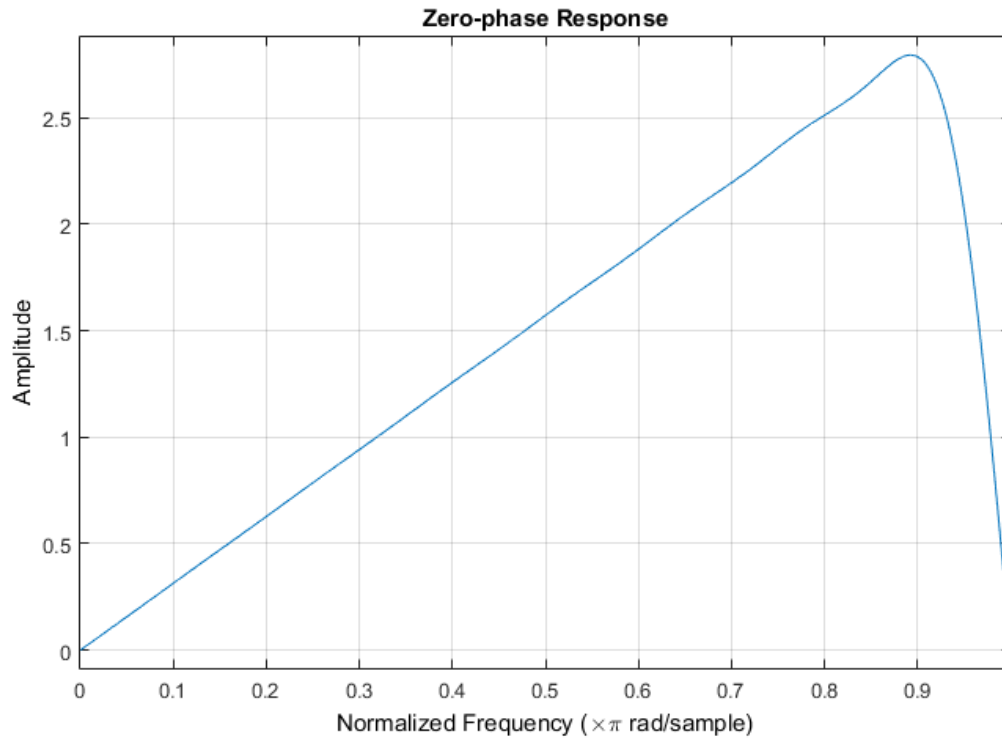
```
b = firls(255,[0 0.25 0.3 1],[1 1 0 0]);
fvtool(b,1,'OverlaidAnalysis','phase')
```



### Design of a Differentiator

An ideal differentiator has a frequency response given by  $D(\omega) = j\omega$ . Design a differentiator of order 30 that attenuates frequencies above  $0.9\pi$ . Include a factor of  $\pi$  in the amplitude because the frequencies are normalized by  $\pi$ . Display the zero-phase response of the filter.

```
b = firls(30,[0 0.9],[0 0.9*pi],'differentiator');
fvtool(b,1,'magnitudedisplay','zero-phase')
```



### Filter with Piecewise Linear Passbands

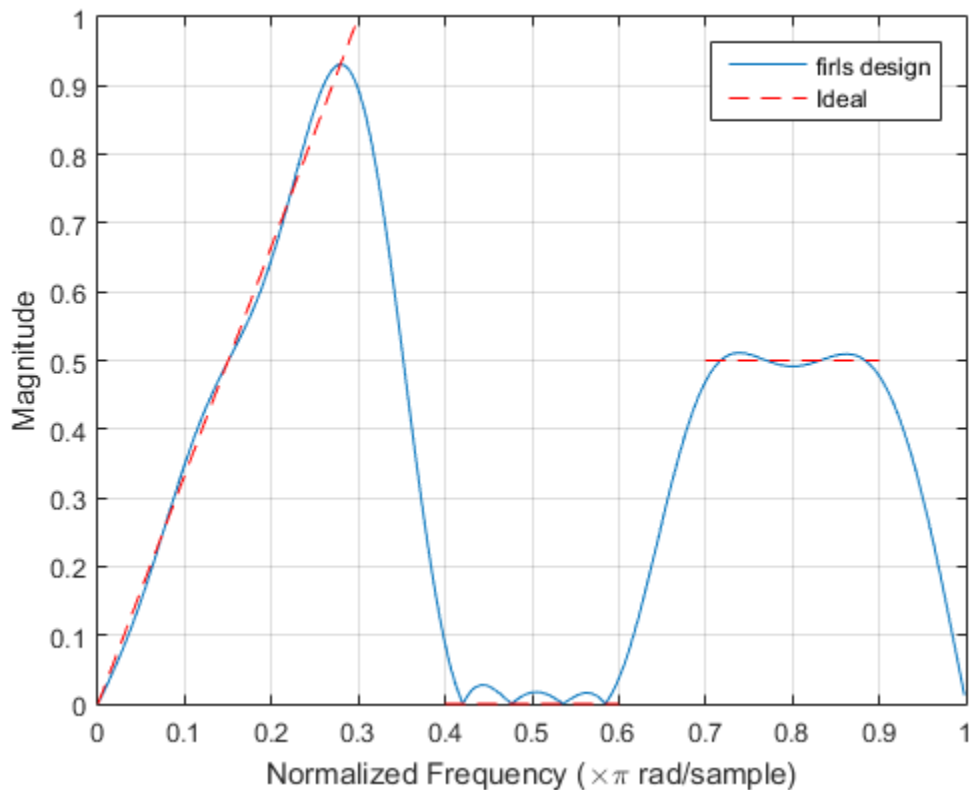
Design a 24th-order antisymmetric filter with piecewise linear passbands.

```
F = [0 0.3 0.4 0.6 0.7 0.9];
A = [0 1.0 0.0 0.0 0.5 0.5];
b = firls(24,F,A,'hilbert');
```

Plot the desired and actual frequency responses.

```
[H,f] = freqz(b,1,512,2);
plot(f,abs(H))
hold on
for i = 1:2:6,
    plot([F(i) F(i+1)],[A(i) A(i+1)],'r--')
end
legend('firls design','Ideal')
```

```
grid on
xlabel('Normalized Frequency (\times\pi rad/sample)')
ylabel('Magnitude')
```



## 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.

## More About

### Algorithms

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         | $b(k) = b(n + 2 - k)$ , $k = 1, \dots, n + 1$  | No restriction            | No restriction                      |
| Type II                  | Even         | $b(k) = b(n + 2 - k)$ , $k = 1, \dots, n + 1$  | No restriction            | $H(1) = 0$                          |
| Type III                 | Odd          | $b(k) = -b(n + 2 - k)$ , $k = 1, \dots, n + 1$ | $H(0) = 0$                | $H(1) = 0$                          |
| Type IV                  | Odd          | $b(k) = -b(n + 2 - k)$ , $k = 1, \dots, n + 1$ | $H(0) = 0$                | No restriction                      |

## References

- [1] Parks, Thomas W., and C. Sidney Burrus. *Digital Filter Design*. New York: John Wiley & Sons, 1987, pp. 54–83.

[2] Oppenheim, Alan V., Ronald W. Schafer, and John R. Buck. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1999.

**See Also**

`fir1` | `fir2` | | `firpm` | `rcosdesign`

# firpm

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:

$$b(k) = b(n+2-k), \quad k = 1, \dots, n+1$$

Vectors `f` and `a` specify the frequency-magnitude characteristics of the filter:

- $\mathbf{f}$  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.
- $\mathbf{a}$  is a vector containing the desired amplitudes at the points specified in  $\mathbf{f}$ .

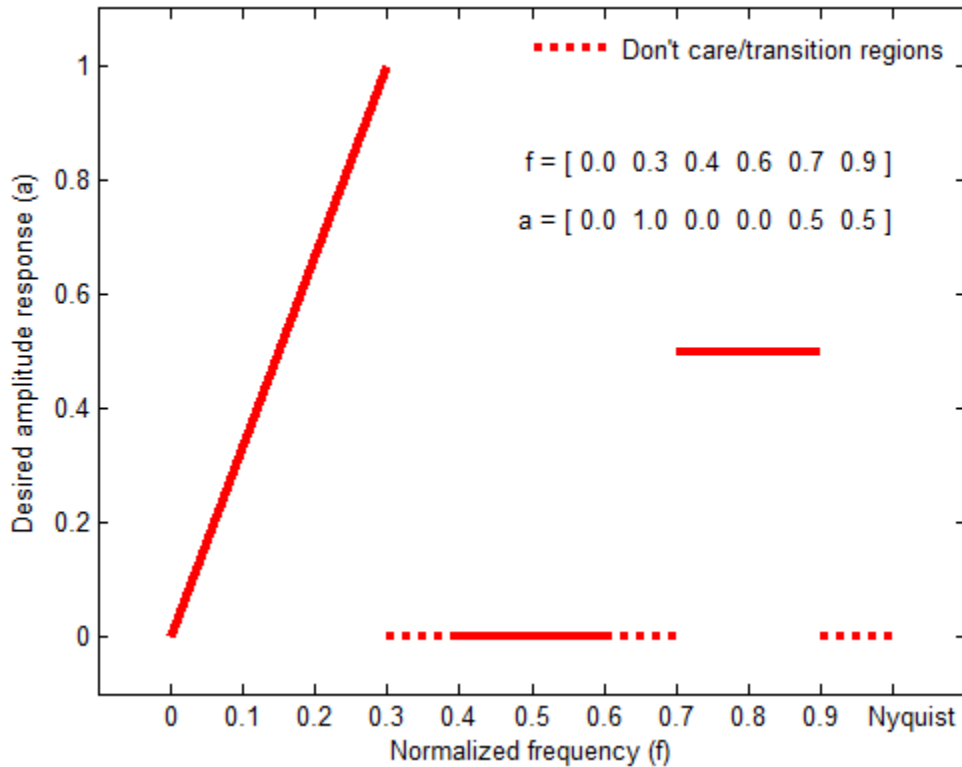
The desired amplitude at frequencies between pairs of points  $(f(k), f(k+1))$  for  $k$  odd is the line segment connecting the points  $(f(k), a(k))$  and  $(f(k+1), a(k+1))$ .

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.

- $\mathbf{f}$  and  $\mathbf{a}$  must be the same length. The length must be an even number.

The relationship between the  $\mathbf{f}$  and  $\mathbf{a}$  vectors in defining a desired frequency response is shown in the illustration below.





`firpm` always uses an even filter order for configurations with even symmetry and a nonzero passband at the Nyquist frequency. This is because for impulse responses exhibiting even symmetry and 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  $b(k) = -b(n+2-k)$ ,  $k = 1, \dots, n+1$ . 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.

|                        |   |
|------------------------|---|
| <code>res.fgrid</code> | Frequency grid vector used for the filter design optimization       |
| <code>res.des</code>   | Desired frequency response for each point in <code>res.fgrid</code> |

|                        |  |
|------------------------|--|
| <code>res.wt</code>    | Weighting for each point in <code>opt.fgrid</code>                           |
| <code>res.H</code>     | Actual frequency response for each point in <code>res.fgrid</code>           |
| <code>res.error</code> | Error at each point in <code>res.fgrid</code> ( <code>res.des-res.H</code> ) |
| <code>res.iextr</code> | Vector of indices into <code>res.fgrid</code> for extremal frequencies       |
| <code>res.fextr</code> | 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  $\text{sym}$ , which `firpm` expects to be either 'even' or 'odd'. If *fresp* does not support this calling syntax, `firpm` defaults to even symmetry.

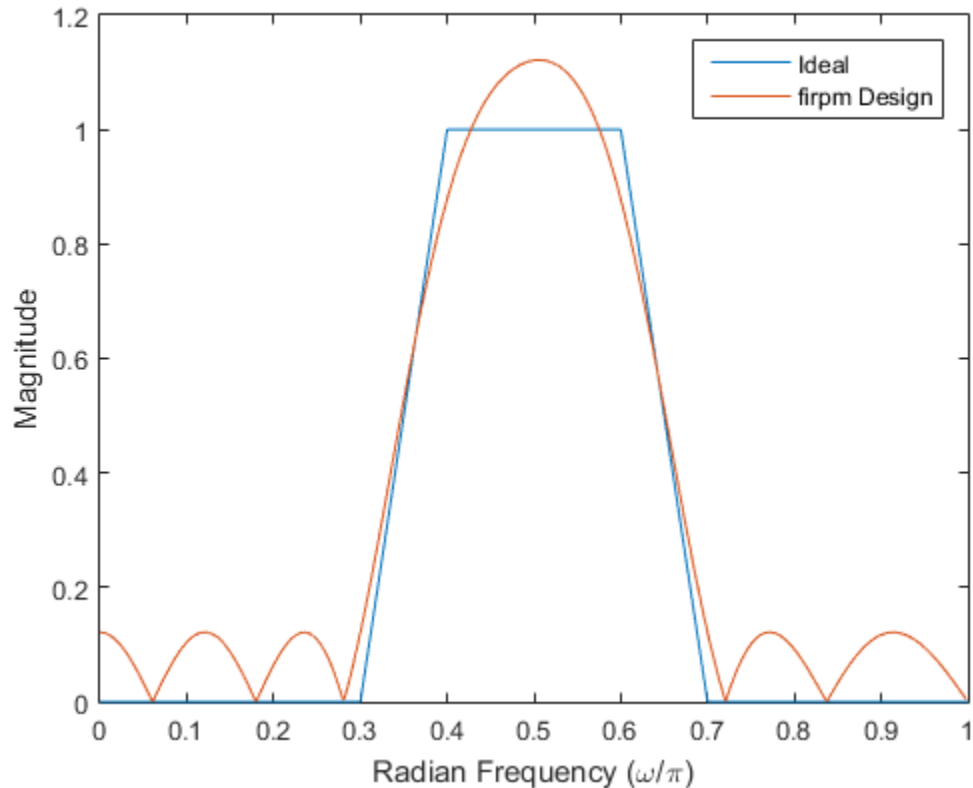
## Examples

### Parks-McClellan Bandpass Filter

Use the Parks-McClellan algorithm to design an FIR bandpass filter of order 17. Specify normalized stopband frequencies of  $0.3\pi$  and  $0.7\pi$  rad/sample and normalized passband frequencies of  $0.4\pi$  and  $0.6\pi$  rad/sample. Plot the ideal and actual magnitude responses.

```
f = [0 0.3 0.4 0.6 0.7 1];
a = [0 0.0 1.0 1.0 0.0 0];
b = firpm(17,f,a);

[h,w] = freqz(b,1,512);
plot(f,a,w/pi,abs(h))
legend('Ideal','firpm Design')
xlabel 'Radian Frequency (\omega/\pi)', ylabel 'Magnitude'
```



## Tips

If your filter design fails to converge, it is possible that the filter design is correct. Verify the design by checking the frequency response.

If your filter design fails to converge and the resulting filter design is not correct, attempt one or more of the following:

- Increase the filter order
- Relax the filter design by reducing the attenuation in the stopbands and/or broadening the transition regions

## More About

### Algorithms

`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:<br>$b(k) = b(n + 2 - k), \quad k = 1, \dots, n + 1$ | No restriction         | No restriction   |
| Type II                  | Odd          | even:<br>$b(k) = b(n + 2 - k), \quad k = 1, \dots, n + 1$ | No restriction         | $H(1) = 0$<br><br><code>firpm</code> increments the filter order by 1 if you attempt to construct a type II filter with a nonzero passband at the Nyquist frequency. |
| Type III                 | Even         | odd:<br>$b(k) = -b(n + 2 - k), \quad k = 1, \dots, n + 1$ | $H(0) = 0$             | $H(1) = 0$   |
| Type IV                  | Odd          | odd:<br>$b(k) = -b(n + 2 - k), \quad k = 1, \dots, n + 1$ | $H(0) = 0$             | No restriction   |

## References

- [1] Digital Signal Processing Committee of the IEEE Acoustics, Speech, and Signal Processing Society, eds. *Programs for Digital Signal Processing*. New York: IEEE Press, 1979, algorithm5.1.
- [2] Digital Signal Processing Committee of the IEEE Acoustics, Speech, and Signal Processing Society, eds. *Selected Papers in Digital Signal Processing*. Vol. II. New York: IEEE Press, 1976.
- [3] Parks, Thomas W., and C. Sidney Burrus. *Digital Filter Design*. New York: John Wiley & Sons, 1987, p.83.
- [4] Rabiner, Lawrence R., James H. McClellan, and Thomas W. Parks. “FIR Digital Filter Design Techniques Using Weighted Chebyshev Approximation.” *Proceedings of the IEEE*. Vol.63, Number4, 1975, pp.595–610.
- [5] Oppenheim, Alan V., Ronald W. Schaffer, and John R. Buck. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1999, p.486.

## See Also

butter | cheby1 | cheby2 | cfirpm | ellip | fir1 | fir2 | fircls | fircls1 |  
firls | | firpmord | function\_handle | rcosdesign | yulewalk

# firpmord

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.

`c = firpmord(f,a,dev,fs,'cell')` generates a cell-array whose elements are the parameters to `firpm`.



**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  $n+2$ .

---

## Examples

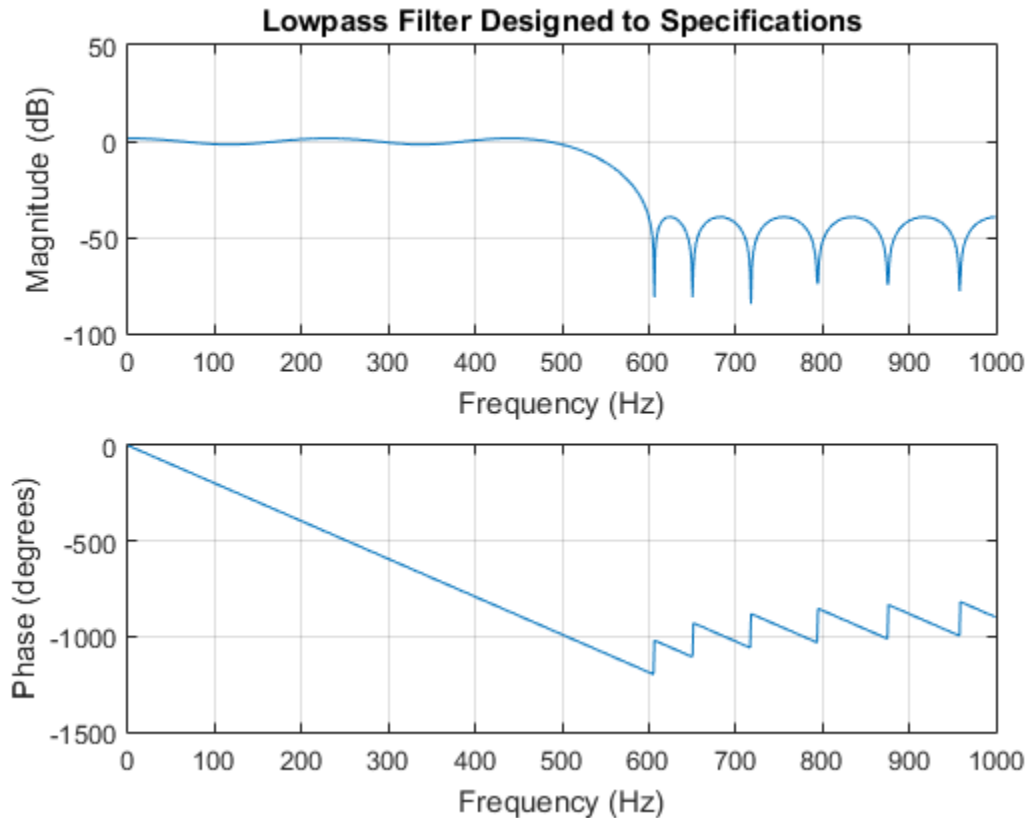
### Minimum-Order Lowpass Filter

Design a minimum-order lowpass filter with a 500 Hz passband cutoff frequency and 600 Hz stopband cutoff frequency. Specify a sampling frequency of 2000 Hz. Require at least 40 dB of 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
```

Convert the deviations to linear units. Design the filter and visualize its magnitude and phase responses.

```
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.

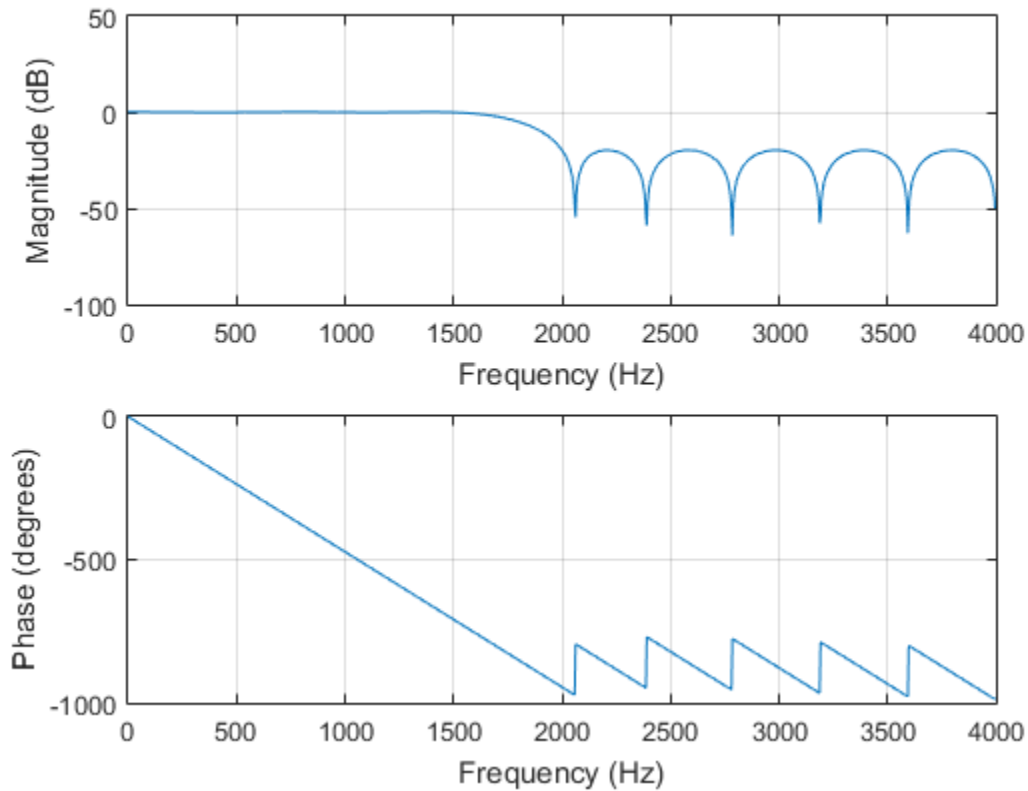
### Parks-McClellan Order of Lowpass Filter

Design a lowpass filter with a 1500 Hz passband cutoff frequency and 2000 Hz stopband cutoff frequency. Specify a sampling frequency of 8000 Hz. Require 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);
```

Obtain an equivalent result by having `firpmord` generate a cell array. Visualize the frequency response of the filter.

```
c = firpmord([1500 2000],[1 0],[0.01 0.1],8000,'cell');  
B = firpm(c{:});  
freqz(B,1,1024,8000)
```



## More About

### Algorithms

`firpmord` uses the algorithm suggested in [1]. This method is inaccurate for band edges close to either 0 or the Nyquist frequency,  $f_s/2$ .

## References

- [1] Rabiner, Lawrence R., and Otto Herrmann. “The Predictability of Certain Optimum Finite-Impulse-Response Digital Filters.” *IEEE Transactions on Circuit Theory*. Vol. 20, Number 4, 1973, pp.401–408.
- [2] Rabiner, Lawrence R., and Bernard 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

# firtype

Type of linear phase FIR filter

## Syntax

```
t = firtype(b)
t = firtype(d)
```

## Description

`t = firtype(b)` determines the type, `t`, of an FIR filter with coefficients `b`. `t` can be 1, 2, 3, or 4. The filter must be real and have linear phase.

`t = firtype(d)` determines the type, `t`, of an FIR filter, `d`. `t` can be 1, 2, 3, or 4. The filter must be real and have linear phase.

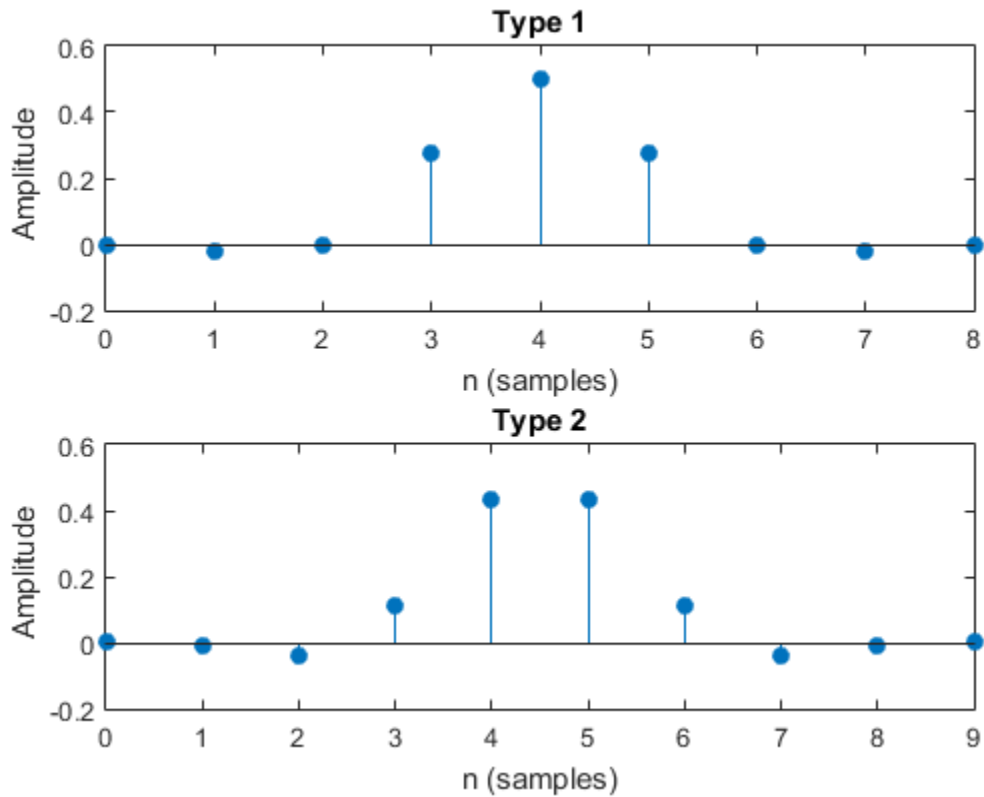
## Examples

### Types of Linear Phase Filters

Design two FIR filters using the window method, one of even order and the other of odd order. Determine their types and plot their impulse responses.

```
subplot(2,1,1)
b = fir1(8,0.5);
impz(b), title(['Type ' int2str(firtype(b))])

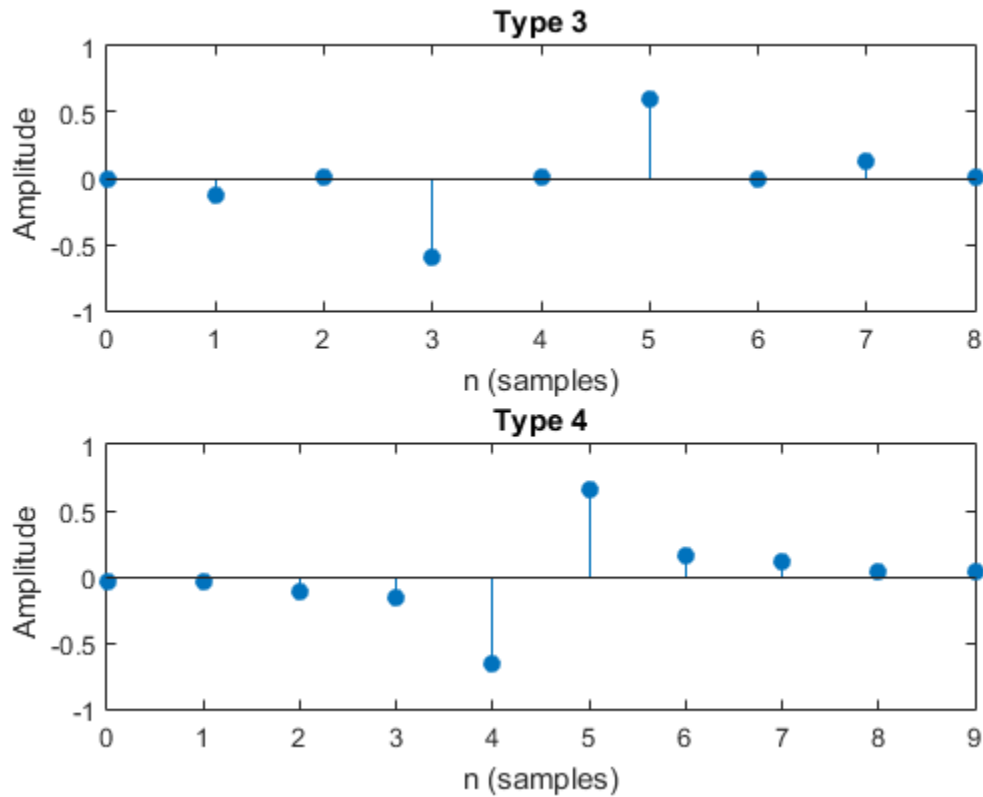
subplot(2,1,2)
b = fir1(9,0.5);
impz(b), title(['Type ' int2str(firtype(b))])
```



Design two equiripple Hilbert transformers, one of even order and the other of odd order. Determine their types and plot their impulse responses.

```
subplot(2,1,1)
b = firpm(8,[0.2 0.8],[1 1], 'hilbert');
impz(b), title(['Type ' int2str(firtype(b))])
```

```
subplot(2,1,2)
b = firpm(9,[0.2 0.8],[1 1], 'hilbert');
impz(b), title(['Type ' int2str(firtype(b))])
```



### Types of FIR digitalFilter Objects

Use `designfilt` to design the filters from the previous example. Display their types.

```
d1 = designfilt('lowpassfir','DesignMethod','window', ...
               'FilterOrder',8,'CutoffFrequency',0.5);
disp(['d1 is of type ' int2str(firtype(d1))])
d2 = designfilt('lowpassfir','DesignMethod','window', ...
               'FilterOrder',9,'CutoffFrequency',0.5);
disp(['d2 is of type ' int2str(firtype(d2))])
d3 = designfilt('hilbertfir','DesignMethod','equiripple', ...
               'FilterOrder',8,'TransitionWidth',0.4);
disp(['d3 is of type ' int2str(firtype(d3))])
d4 = designfilt('hilbertfir','DesignMethod','equiripple', ...
```

```
                'FilterOrder',9,'TransitionWidth',0.4);  
disp(['d4 is of type ' int2str(firtype(d4))])  
  
d1 is of type 1  
d2 is of type 2  
d3 is of type 3  
d4 is of type 4
```

## Input Arguments

### **b** — Filter coefficients

vector

Filter coefficients of the FIR filter, specified as a double- or single-precision real-valued row or column vector.

Data Types: `double` | `single`

### **d** — FIR filter

`digitalFilter` object | `filter System` object | `dfilt` object | `mfilt` object

FIR filter, specified as any of the following:

- A `digitalFilter` object. Use `designfilt` to generate a digital filter based on frequency-response specifications.
- A Filter System object™. You can use this input if you have a license for DSP System Toolbox software. `firtype` supports the following Filter System objects.

|                                       |
|---------------------------------------|
| <code>dsp.AllpassFilter</code>        |
| <code>dsp.AllpoleFilter</code>        |
| <code>dsp.BiquadFilter</code>         |
| <code>dsp.CICDecimator</code>         |
| <code>dsp.CICInterpolator</code>      |
| <code>dsp.CoupledAllpassFilter</code> |
| <code>dsp.FIRDecimator</code>         |
| <code>dsp.FIRFilter</code>            |
| <code>dsp.FIRInterpolator</code>      |



|                                   |
|-----------------------------------|
| <code>dsp.FIRRateConverter</code> |
|-----------------------------------|

|                            |
|----------------------------|
| <code>dsp.IIRFilter</code> |
|----------------------------|

- A `dfilt` filter object. You can use this input if you have a license for DSP System Toolbox software.
- A multirate `mfilt` filter object. You can use this input if you have a license for DSP System Toolbox software.

## Output Arguments

### **t** — Filter type

1 | 2 | 3 | 4

Filter type, returned as either 1, 2, 3, or 4. Filter types are defined as follows:

- Type 1 — Even-order symmetric coefficients
- Type 2 — Odd-order symmetric coefficients
- Type 3 — Even-order antisymmetric coefficients
- Type 4 — Odd-order antisymmetric coefficients

### See Also

`designfilt` | `digitalFilter` | `islinphase`

# flattopwin

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`.

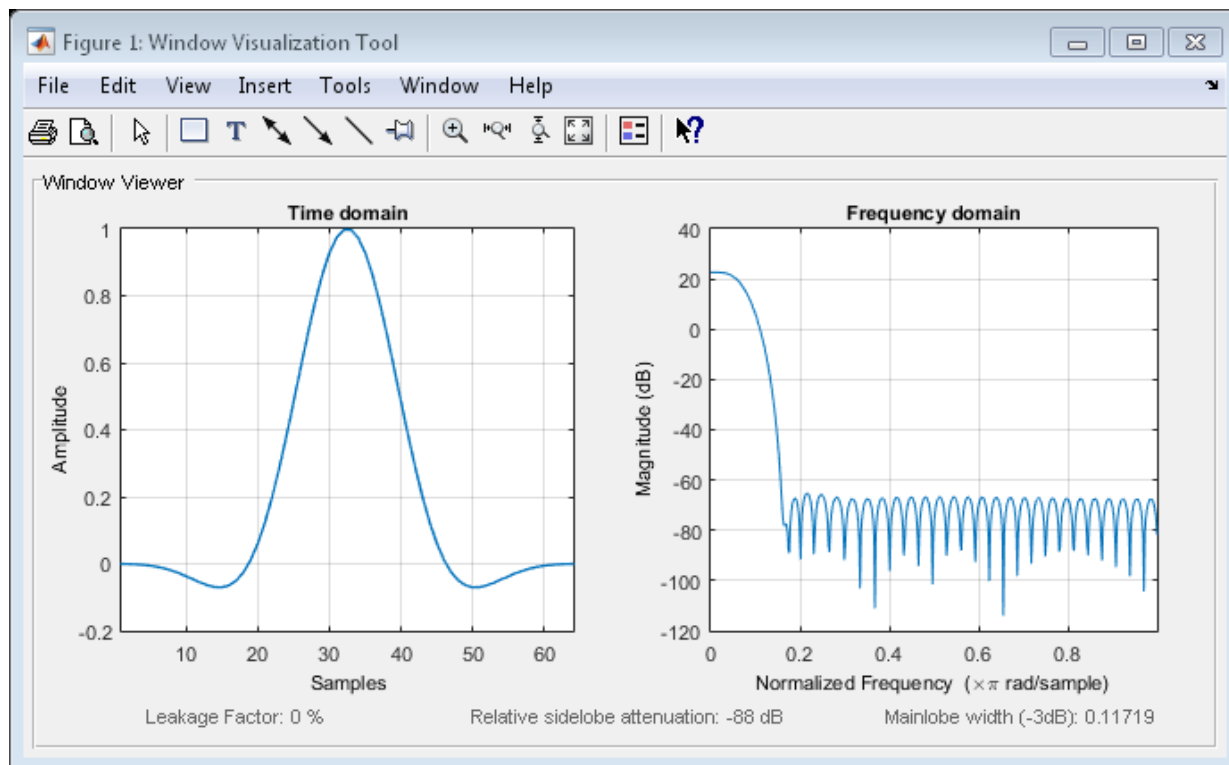
`w = flattopwin(L,sflag)` returns the L-point symmetric flat top window using `sflag` window sampling, where `sflag` is either 'symmetric' or 'periodic'. The 'periodic' flag is useful for DFT/FFT purposes, such as in spectral analysis. The DFT/FFT contains an implicit periodic extension and the periodic 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.

## Examples

### Flat Top Window

Create a 64-point symmetric flat top window. View the result using `wvtool`.

```
N = 64;
w = flattopwin(N);
wvtool(w)
```



## More About

### Algorithms

Flat top windows are summations of cosines. The coefficients of a flat top window are computed from the following equation:

$$w(n) = a_0 - a_1 \cos\left(\frac{2\pi n}{N-1}\right) + a_2 \cos\left(\frac{4\pi n}{N-1}\right) - a_3 \cos\left(\frac{6\pi n}{N-1}\right) + a_4 \cos\left(\frac{8\pi n}{N-1}\right),$$

where  $0 \leq n \leq N-1$ . The coefficient values are

| Coefficient | Value      |
|-------------|------------|
| $a_0$       | 0.21557895 |

| <b>Coefficient</b> | <b>Value</b> |
|--------------------|--------------|
| $a_1$              | 0.41663158   |
| $a_2$              | 0.277263158  |
| $a_3$              | 0.083578947  |
| $a_4$              | 0.006947368  |

## References

- [1] D'Antona, Gabriele, and A. Ferrero. *Digital Signal Processing for Measurement Systems*. New York: Springer Media, 2006, pp. 70–72.
- [2] Gade, Svend, and Henrik Herlufsen. “Use of Weighting Functions in DFT/FFT Analysis (Part I).” *Windows to FFT Analysis (Part I): Brüel & Kjær Technical Review*, No. 3, 1987, pp. 1–28.

## See Also

blackman | hann | hamming

# freqs

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

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

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/s 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.

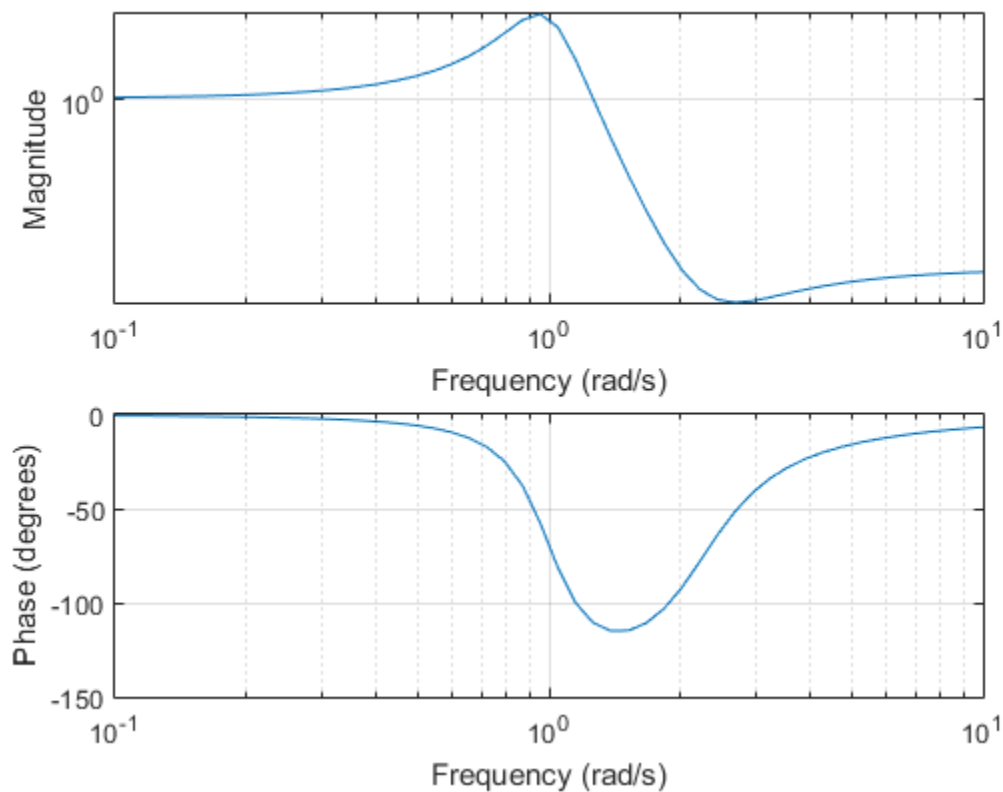
## Examples

### Frequency Response from the Transfer Function

Find and graph the frequency response of the transfer function

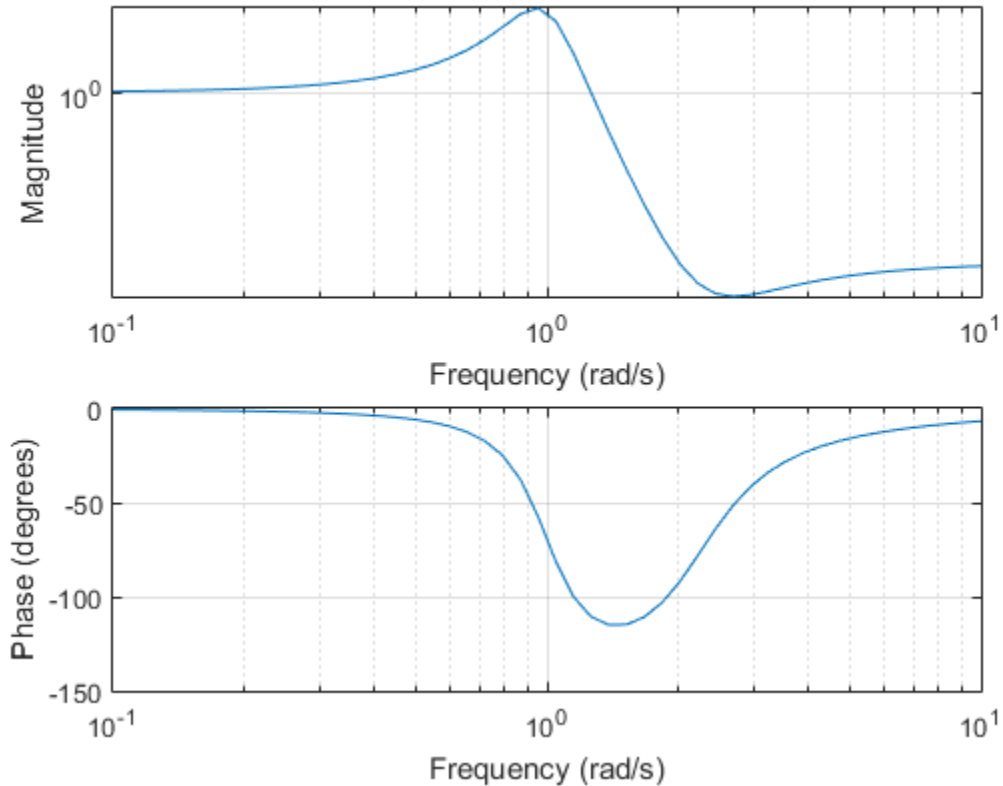
$$H(s) = \frac{0.2s^2 + 0.3s + 1}{s^2 + 0.4s + 1}.$$

```
a = [1 0.4 1];  
b = [0.2 0.3 1];  
w = logspace(-1,1);  
freqs(b,a,w)
```



You can also compute the results and use them to generate the plots.

```
h = freqs(b,a,w);  
mag = abs(h);  
phase = angle(h);  
phasedeg = phase*180/pi;  
  
subplot(2,1,1), loglog(w,mag), grid on  
xlabel 'Frequency (rad/s)', ylabel Magnitude  
subplot(2,1,2), semilogx(w,phasedeg), grid on  
xlabel 'Frequency (rad/s)', ylabel 'Phase (degrees)'
```

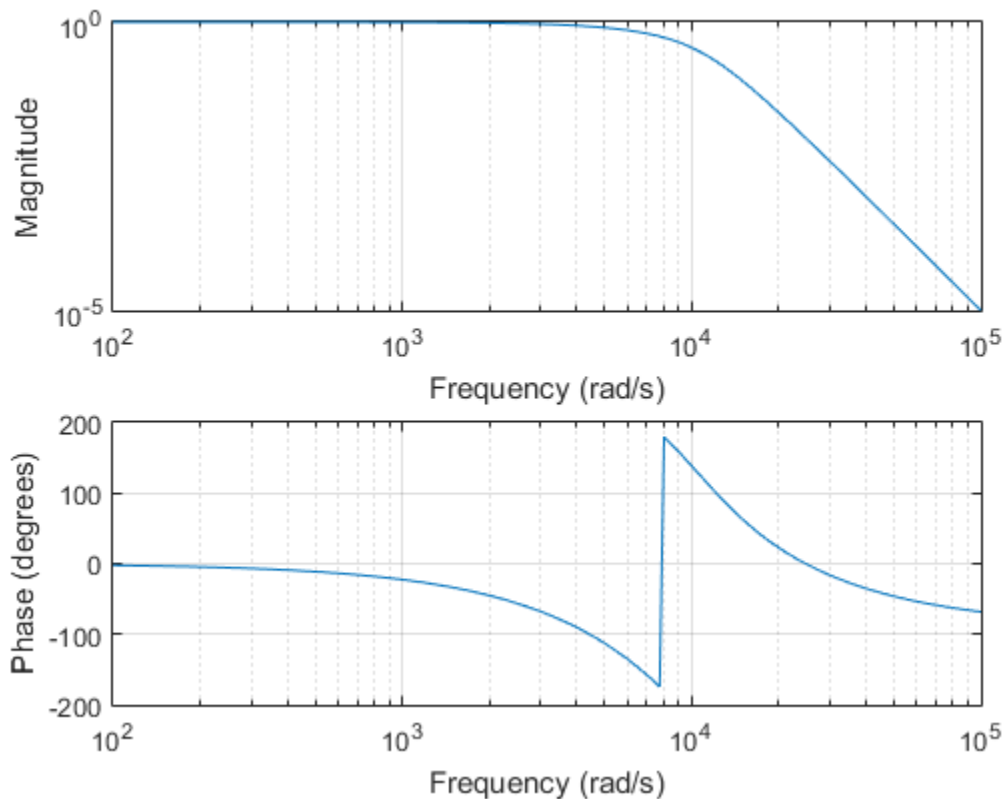


### Frequency Response of a Lowpass Analog Bessel Filter

Design a 5th-order analog lowpass Bessel filter with an approximately constant group delay up to  $10^4$  rad/s. Plot the frequency response of the filter using `freqs`.

```
[b,a] = besself(5,10000); % Bessel analog filter design
freqs(b,a)                % Plot frequency response
```





## More About

### Algorithms

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

## freqsamp

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 filter specifications object `d`.

`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 |
|------------------------|---|
| <code>dffir</code>     | Direct-form FIR filter                    |
| <code>dffirt</code>    | Transposed direct-form FIR filter         |
| <code>dfsymfir</code>  | Symmetrical direct-form FIR filter        |
| <code>dfasymfir</code> | Asymmetrical direct-form FIR filter       |
| <code>fftfir</code>    | 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 `'hamming'`. See `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 first example shows a cell array input argument.

- The window vector itself.

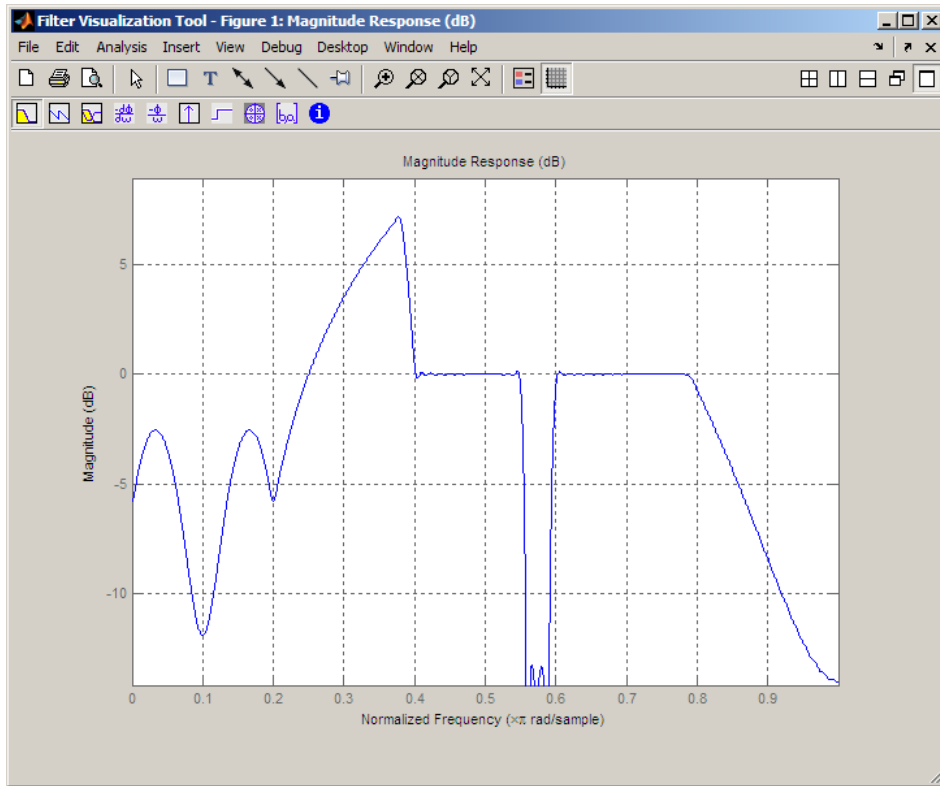
## Examples

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)
```

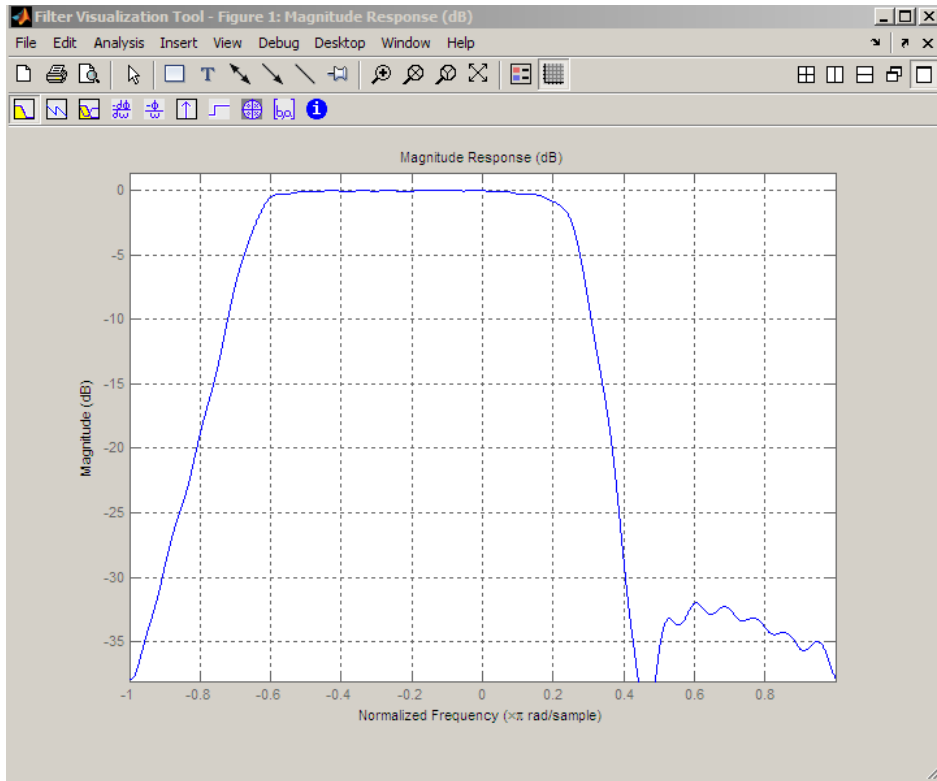
The plot from FVTool shows the response for hd.



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`

## freqz

Frequency response of digital filter

### Syntax

```
[h,w] = freqz(b,a,n)
[h,w] = freqz(sos,n)
[h,w] = freqz(d,n)
[h,w] = freqz( ____,n,'whole' )

[h,f] = freqz( ____,n,fs)
[h,f] = freqz( ____,n,'whole',fs)

h = freqz( ____,w)
h = freqz( ____,f,fs)

freqz( ____)
```

### Description

`[h,w] = freqz(b,a,n)` returns the  $n$ -point frequency response vector,  $h$ , and the corresponding angular frequency vector,  $w$ , for the digital filter with numerator and denominator polynomial coefficients stored in  $\mathbf{b}$  and  $\mathbf{a}$ , respectively.

`[h,w] = freqz(sos,n)` returns the  $n$ -point complex frequency response corresponding to the second-order sections matrix,  $\mathbf{sos}$ .

`[h,w] = freqz(d,n)` returns the  $n$ -point complex frequency response for the digital filter,  $d$ .

`[h,w] = freqz( ____,n,'whole' )` returns the frequency response at  $n$  sample points around the entire unit circle.

`[h,f] = freqz( ____,n,fs)` returns the frequency response vector,  $h$ , and the corresponding physical frequency vector,  $f$ , for the digital filter with numerator and denominator polynomial coefficients stored in  $\mathbf{b}$  and  $\mathbf{a}$ , respectively, given the sampling frequency,  $fs$ .

`[h,f] = freqz( ____, n, 'whole', fs)` returns the frequency at `n` points ranging between 0 and `fs`.

`h = freqz( ____, w)` returns the frequency response vector, `h`, at the normalized frequencies supplied in `w`.

`h = freqz( ____, f, fs)` returns the frequency response vector, `h`, at the physical frequencies supplied in `f`.

`freqz( ____, )` with no output arguments plots the frequency response of the filter.

---

**Note:** If the input to `freqz` is single precision, the frequency response is calculated using single-precision arithmetic. The output, `h`, is single precision.

---

## Examples

### Frequency Response from Transfer Function

Compute and display the magnitude response of the third-order IIR lowpass filter described by the following transfer function:

$$H(z) = \frac{0.05634(1 + z^{-1})(1 - 1.0166z^{-1} + z^{-2})}{(1 - 0.683z^{-1})(1 - 1.4461z^{-1} + 0.7957z^{-2})}$$

Express the numerator and denominator as polynomial convolutions. Find the frequency response at 2001 points spanning the complete unit circle.

```
b0 = 0.05634;
b1 = [1 1];
b2 = [1 -1.0166 1];
a1 = [1 -0.683];
a2 = [1 -1.4461 0.7957];

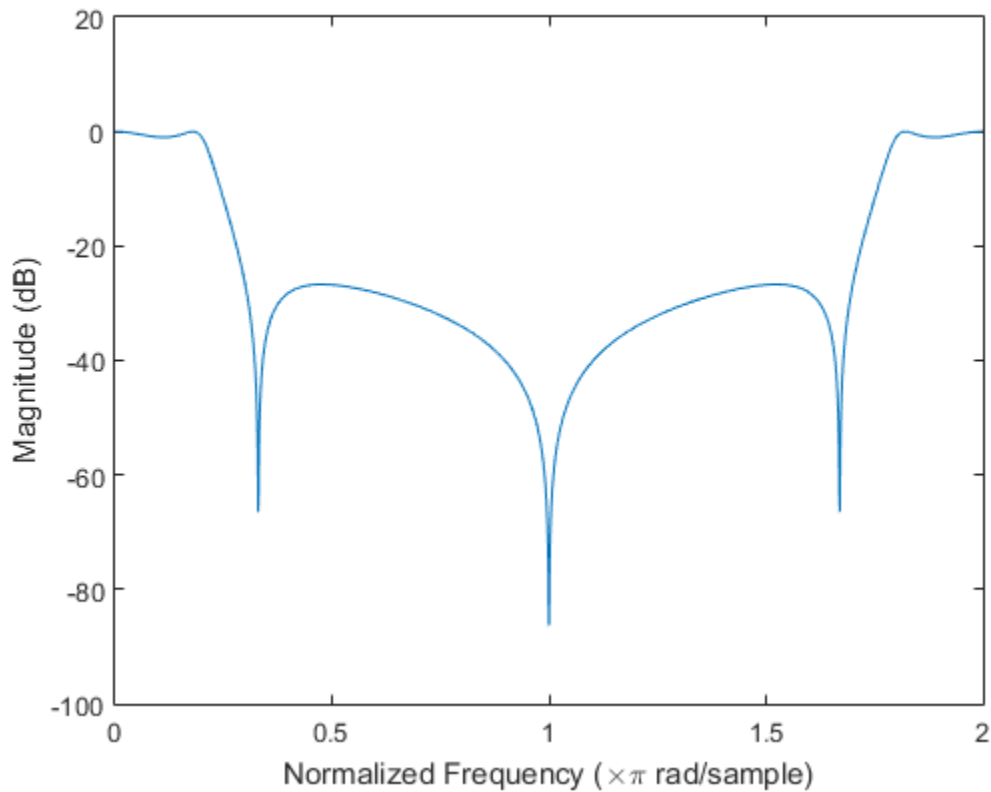
b = b0*conv(b1,b2);
a = conv(a1,a2);

[h,w] = freqz(b,a,'whole',2001);
```

Plot the magnitude response expressed in decibels.

```
plot(w/pi,20*log10(abs(h)))
```

```
ax = gca;  
ax.YLim = [-100 20];  
ax.XTick = 0:.5:2;  
xlabel('Normalized Frequency (\times\pi rad/sample)')  
ylabel('Magnitude (dB)')
```



### Frequency Response from Second-Order Sections

Compute and display the magnitude response of the third-order IIR lowpass filter described by the following transfer function:

$$H(z) = \frac{0.05634(1 + z^{-1})(1 - 1.0166z^{-1} + z^{-2})}{(1 - 0.683z^{-1})(1 - 1.4461z^{-1} + 0.7957z^{-2})}$$



Express the transfer function in terms of second-order sections. Find the frequency response at 2001 points spanning the complete unit circle.

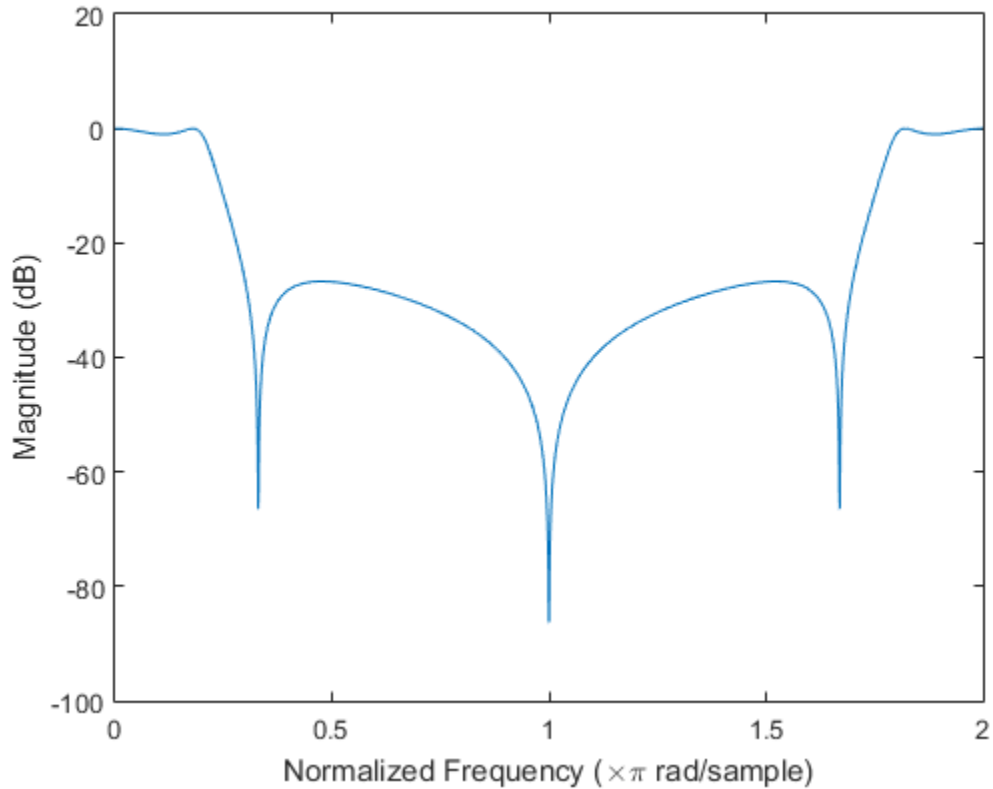
```
b0 = 0.05634;
b1 = [1 1];
b2 = [1 -1.0166 1];
a1 = [1 -0.683];
a2 = [1 -1.4461 0.7957];

sos1 = [b0*[b1 0] [a1 0]];
sos2 = [b2 a2];

[h,w] = freqz([sos1;sos2], 'whole',2001);
```

Plot the magnitude response expressed in decibels.

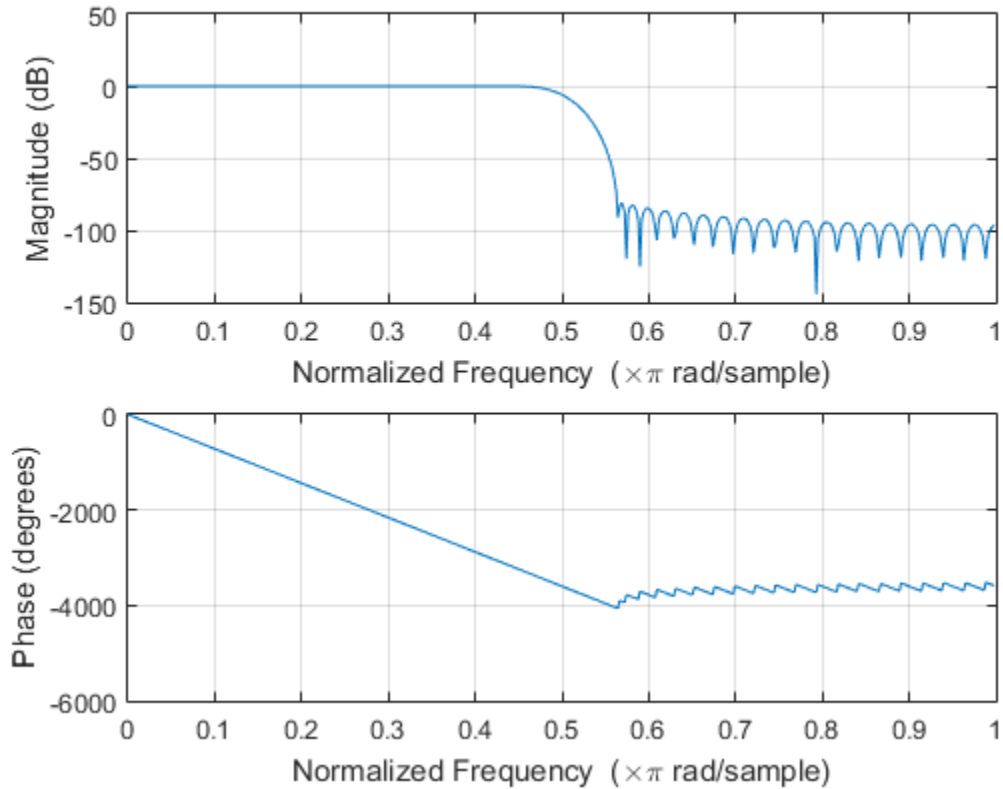
```
plot(w/pi,20*log10(abs(h)))
ax = gca;
ax.YLim = [-100 20];
ax.XTick = 0:.5:2;
xlabel('Normalized Frequency (\times\pi rad/sample)')
ylabel('Magnitude (dB)')
```



### Frequency Response of an FIR filter

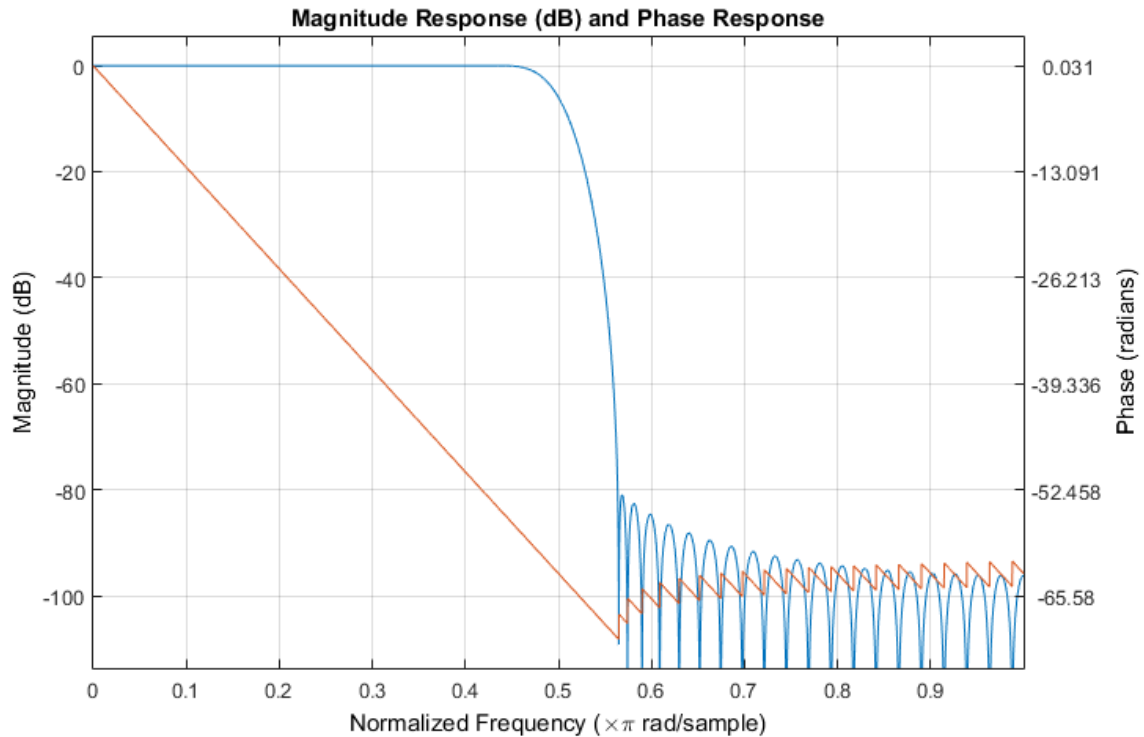
Design an FIR lowpass filter of order 80 using a Kaiser window with  $\beta = 8$ . Specify a normalized cutoff frequency of  $0.5\pi$  rad/sample. Display the magnitude and phase responses of the filter.

```
b = fir1(80,0.5,kaiser(81,8));  
freqz(b,1)
```



Design the same filter using `designfilt`. Display its magnitude and phase responses using `fvtool`.

```
d = designfilt('lowpassfir','FilterOrder',80, ...  
              'CutoffFrequency',0.5,'Window',{'kaiser',8});  
freqz(d)
```



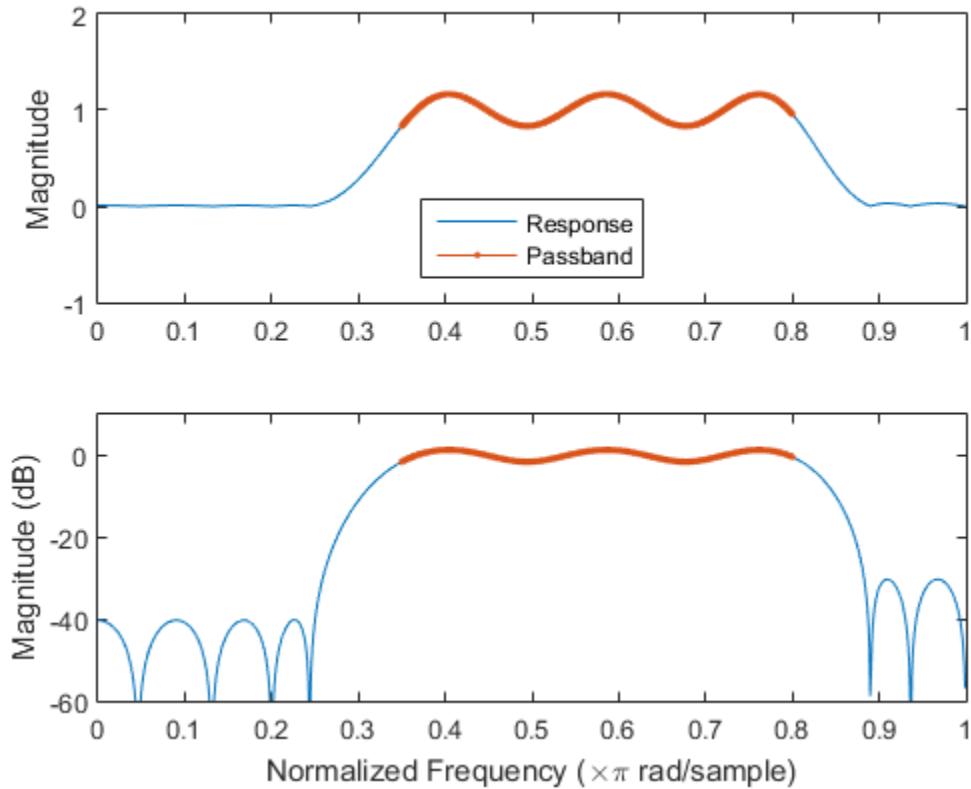
### Frequency Response of an FIR Bandpass Filter

Design an FIR bandpass filter with passband between  $0.35\pi$  and  $0.8\pi$  rad/sample and 3 dB of ripple. The first stopband goes from 0 to  $0.1\pi$  rad/sample and has an attenuation of 40 dB. The second stopband goes from  $0.9\pi$  rad/sample to the Nyquist frequency and has an attenuation of 30 dB. Compute the frequency response. Plot its magnitude in both linear units and decibels. Highlight the passband.

```
sf1 = 0.1;
pf1 = 0.35;
pf2 = 0.8;
sf2 = 0.9;
pb = linspace(pf1,pf2,1e3)*pi;

bp = designfilt('bandpassfir', ...
    'StopbandAttenuation1',40, 'StopbandFrequency1',sf1,...
```

```
'PassbandFrequency1',pf1,'PassbandRipple',3,'PassbandFrequency2',pf2, ...  
'StopbandFrequency2',sf2,'StopbandAttenuation2',30);  
  
[h,w] = freqz(bp,1024);  
hpb = freqz(bp,pb);  
  
subplot(2,1,1)  
plot(w/pi,abs(h),pb/pi,abs(hpb),'.-')  
axis([0 1 -1 2])  
legend('Response','Passband','Location','South')  
ylabel('Magnitude')  
  
subplot(2,1,2)  
plot(w/pi,db(h),pb/pi,db(hpb),'.-')  
axis([0 1 -60 10])  
xlabel('Normalized Frequency (\times\pi rad/sample)')  
ylabel('Magnitude (dB)')
```



## Input Arguments

**b, a** — Transfer function coefficients

vectors

Transfer function coefficients, specified as vectors. Express the transfer function in terms of b and a as

$$H(e^{j\omega}) = \frac{B(e^{j\omega})}{A(e^{j\omega})} = \frac{b(1) + b(2)e^{-j\omega} + b(3)e^{-j2\omega} + \dots + b(M)e^{-j(M-1)\omega}}{a(1) + a(2)e^{-j\omega} + a(3)e^{-j2\omega} + \dots + a(N)e^{-j(N-1)\omega}}$$

Example:  $b = [1 \ 3 \ 3 \ 1]/6$  and  $a = [3 \ 0 \ 1 \ 0]/3$  specify a third-order Butterworth filter with normalized 3-dB frequency  $0.5\pi$  rad/sample.

Data Types: `double` | `single`

Complex Number Support: Yes

### **n** — Number of evaluation points

512 (default) | positive integer scalar

Number of evaluation points, specified as a positive integer scalar no less than 2. When  $n$  is absent, it defaults to 512. For best results, set  $n$  to a value greater than the filter order.

Data Types: `double`

### **sos** — Second-order section coefficients

matrix

Second-order section coefficients, specified as a matrix. `sos` is a  $K$ -by-6 matrix, where the number of sections,  $K$ , must be greater than or equal to 2. If the number of sections is less than 2, `freqz` considers the input to be a numerator vector. Each row of `sos` corresponds to the coefficients of a second-order (biquad) filter. The  $i$ th row of `sos` corresponds to `[bi(1) bi(2) bi(3) ai(1) ai(2) ai(3)]`.

Example:  $s = [2 \ 4 \ 2 \ 6 \ 0 \ 2; 3 \ 3 \ 0 \ 6 \ 0 \ 0]$  specifies a third-order Butterworth filter with normalized 3-dB frequency  $0.5\pi$  rad/sample.

Data Types: `double` | `single`

Complex Number Support: Yes

### **d** — Digital filter

`digitalFilter` object

Digital filter, specified as a `digitalFilter` object. Use `designfilt` to generate a digital filter based on frequency-response specifications.

Example: `d = designfilt('lowpassiir','FilterOrder',3,'HalfPowerFrequency',0.5)` specifies a third-order Butterworth filter with normalized 3-dB frequency  $0.5\pi$  rad/sample.

### **fs** — Sampling rate

positive scalar

Sampling rate, specified as a positive scalar. When the unit of time is seconds,  $fs$  is expressed in hertz.

Data Types: double

**w — Angular frequencies**

vector

Angular frequencies, specified as a vector and expressed in radians/sample.  $w$  must have at least two elements.  $w = \pi$  corresponds to the Nyquist frequency.

**f — Frequencies**

vector

Frequencies, specified as a vector.  $f$  must have at least two elements. When the unit of time is seconds,  $f$  is expressed in hertz.

Data Types: double

## Output Arguments

**h — Frequency response**

vector

Frequency response, returned as a vector. If you specify  $n$ ,  $h$  has length  $n$ . If you do not specify  $n$ , or specify  $n$  as the empty vector,  $h$  has length 512.

**w — Angular frequencies**

vector

Angular frequencies, returned as a vector.  $w$  has values ranging from 0 to  $\pi$ . If you specify 'whole' in your input, the values in  $w$  range from 0 to  $2\pi$ . If you specify  $n$ ,  $w$  has length  $n$ . If you do not specify  $n$ , or specify  $n$  as the empty vector,  $w$  has length 512.

**f — Frequencies**

vector

Frequencies, returned as a vector expressed in hertz.  $f$  has values ranging from 0 to  $f_s/2$  Hz. If you specify 'whole' in your input, the values in  $f$  range from 0 to  $f_s$  Hz. If you specify  $n$ ,  $f$  has length  $n$ . If you do not specify  $n$ , or specify  $n$  as the empty vector,  $f$  has length 512.



## More About

### Algorithms

The frequency response of a digital filter can be interpreted as the transfer function evaluated at  $z = e^{j\omega}$  [1].

`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, `freqz` evaluates the polynomials at each frequency point using Horner's method of nested polynomial evaluation, dividing the numerator response by the denominator response.

### References

[1] Oppenheim, Alan V., Ronald W. Schafer, and John R. Buck. *Discrete-Time Signal Processing*. 2nd Ed. Upper Saddle River, NJ: Prentice Hall, 1999.

### See Also

`digitalFilter` | `abs` | `angle` | `designfilt` | `fft` | `filter` | `freqs` | `impz` | `invfreqs` | `logspace`

# fvtool

Open Filter Visualization Tool

## Syntax

```
fvtool(b,a)
fvtool(sos)
fvtool(d)
fvtool(b1,a1,b2,a2,...,bN,aN)
fvtool(sos1,sos2,...,sosN)
fvtool(Hd)
fvtool(Hd1,Hd2,...,HdN)
h = fvtool(...)
```

## Description

`fvtool(b,a)` opens FVTool and displays 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**.

---

**Note:** If the input to `fvtool` is single precision, the magnitude response is calculated using single-precision arithmetic.

---

`fvtool(sos)` opens FVTool and displays the magnitude response of the digital filter defined by the matrix of second order sections, **sos**. **sos** is a  $K$ -by-6 matrix, where the number of sections,  $K$ , must be greater than or equal to 2. If the number of sections is less than 2, `fvtool` considers the input to be a numerator vector. Each row of **sos** corresponds to the coefficients of a second order (biquad) filter. The  $i$ th row of the **sos** matrix corresponds to  $[bi(1) \ bi(2) \ bi(3) \ ai(1) \ ai(2) \ ai(3)]$ .

`fvtool(d)` opens FVTool and displays the magnitude response of a digital filter, **d**. Use `designfilt` to generate **d** based on frequency-response specifications.

`fvtool(b1,a1,b2,a2,...,bN,aN)` opens FVTool and displays the magnitude responses of multiple filters defined with numerators, `b1, ..., bN`, and denominators, `a1, ..., aN`.

`fvtool(sos1,sos2,...,sosN)` opens FVTool and displays the magnitude responses of multiple filters defined with second order section matrices, `sos1, sos2, ..., sosN`.

`fvtool(Hd)` opens FVTool and displays the magnitude responses for the `dfilt` filter object, `Hd`, or the array of `dfilt` filter objects.

`fvtool(Hd1,Hd2,...,HdN)` opens FVTool and displays the magnitude responses of the filters in the `dfilt` objects `Hd1, Hd2, ...HdN`.

If you have the DSP System Toolbox product installed, you can also use `fvtool(H)` and `fvtool(H1,H2,...)` to analyze:

- Quantized filter objects (`dfilt` with arithmetic set to 'single' or 'fixed')
- Multirate filter (`mfilt`) objects
- Adaptive filter (`adaptfilt`) objects
- Any of the following filter System objects.

The following Filter System objects are supported by this analysis function:

| Filter System objects                        |
|--|
| <code>dsp.AllpassFilter</code>               |
| <code>dsp.AllpoleFilter</code>               |
| <code>dsp.BiquadFilter</code>                |
| <code>dsp.CICCompensationDecimator</code>    |
| <code>dsp.CICCompensationInterpolator</code> |
| <code>dsp.CICDecimator</code>                |
| <code>dsp.CICInterpolator</code>             |
| <code>dsp.CoupledAllpassFilter</code>        |
| <code>dsp.FarrowRateConverter</code>         |
| <code>dsp.FilterCascade</code>               |
| <code>dsp.FIRDecimator</code>                |

| Filter System objects                       |
|---|
| <code>dsp.FIRFilter</code>                  |
| <code>dsp.FIRHalfbandDecimator</code>       |
| <code>dsp.FIRHalfbandInterpolator</code>    |
| <code>dsp.FIRInterpolator</code>            |
| <code>dsp.FIRRateConverter</code>           |
| <code>dsp.HighpassFilter</code>             |
| <code>dsp.IIRFilter</code>                  |
| <code>dsp.LowpassFilter</code>              |
| <code>dsp.NotchPeakFilter</code>            |
| <code>dsp.ParametricEQFilter</code>         |
| <code>dsp.VariableBandwidthFIRFilter</code> |
| <code>dsp.VariableBandwidthIIRFilter</code> |

When the input filter is a `dfilt` or `mfilt` object, `FVTool` performs fixed-point analysis if the arithmetic property of the filter objects is set to 'fixed'. However, for filter System objects, `fvtool(H, 'Arithmetic', ARITH, ...)` analyzes `H`, based on the arithmetic specified in the `ARITH` input.

`ARITH` can be one of 'double', 'single', or 'fixed'. The 'Arithmetic' input is only relevant for the analysis of filter System objects. The arithmetic setting `ARITH`, applies to all the filter System objects that you input to `fvtool`. When you specify 'double' or 'single', the function performs double- or single-precision analysis. When you specify 'fixed', the arithmetic changes depending on the setting of the `CoefficientDataType` property and whether the System object is locked or unlocked.

#### Details for Fixed-Point Arithmetic

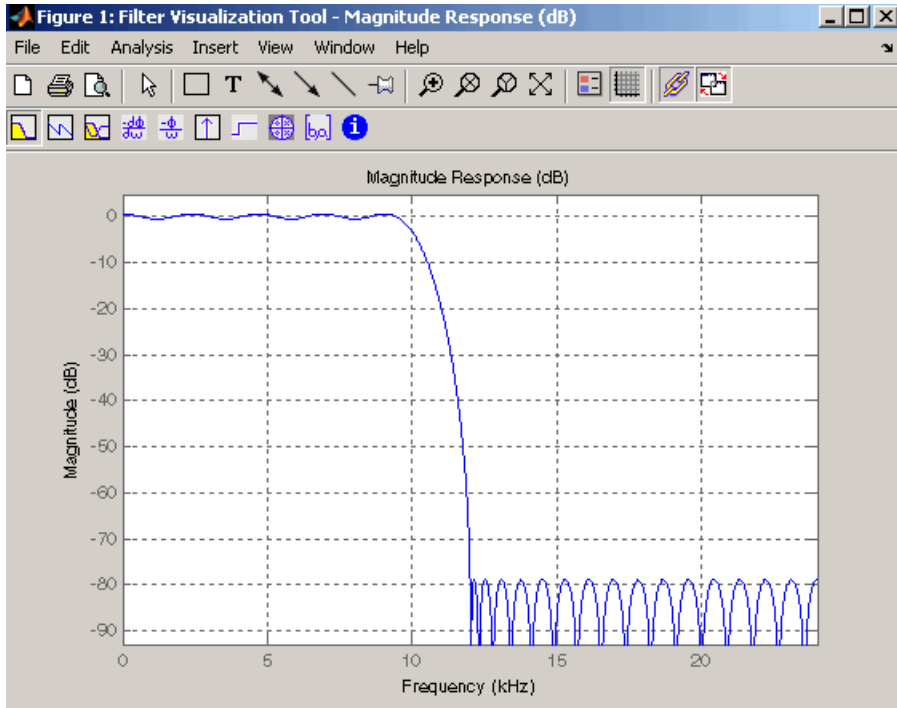
| System Object State | Coefficient Data Type | Rule  |
|---------------------|-----------------------|---|
| Unlocked            | 'Same as input'       | The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption. |

| System Object State | Coefficient Data Type | Rule   |
|---------------------|-----------------------|--|
| Unlocked            | 'Custom'              | The function performs fixed-point analysis based on the setting of the <code>CustomCoefficientsDataType</code> property.   |
| Locked              | 'Same as input'       | When the input data type is 'double' or 'fixed', the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption. |
| Locked              | 'Custom'              | The function performs fixed-point analysis based on the setting of the <code>CustomCoefficientsDataType</code> property.   |

If you do not specify the arithmetic for non-CIC structures, and the System object is in an unlocked state, the function uses double-precision arithmetic. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.





Analysis methods `noisepsd` and `freqrespest` have behavior restrictions in `fvtool`. To see the rules, click the links to these methods.



`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 1-748.













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 <b>View &gt; Full View</b> . |
|  | Toggle legend  |
|  | Toggle grid  |
|  | Link to FDATool (appears only if FVTool was started from FDATool)  |

| Icon  | Description   |
|---|---|
|  | 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 <code>freqz</code> and <code>zerophase</code> for more information.<br><br>To see the zero-phase response, right-click the <i>y</i> -axis label of the Magnitude plot and select <b>Zero-phase</b> from the context menu. |
|    | Phase response of the current filter. See <code>phasez</code> for more information.   |
|    | Superimposes the magnitude response and the phase response of the current filter. See <code>freqz</code> 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 <code>grpdelay</code> 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 <code>phasedelay</code> for more information.  |
|  | Impulse response of the current filter. The impulse response is the response of the filter to a impulse input. See <code>impz</code> for more information.  |
|  | Step response of the current filter. The step response is the response of the filter to a step input. See <code>stepz</code> for more information.  |
|  | Pole-zero plot, which shows the pole and zero locations of the current filter on the <i>z</i> -plane. See <code>zplane</code> 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<br>Linear Frequency | Magnitude<br>Magnitude(dB)<br>Magnitude squared<br>Zero-Phase   |
| Phase               | Normalized Frequency<br>Linear Frequency | Phase<br>Continuous Phase<br>Degrees<br>Radians   |
| Magnitude and Phase | Normalized Frequency<br>Linear Frequency | ( <i>y</i> -axis on left side)<br>Magnitude<br>Magnitude(dB)<br>Magnitude squared<br>Zero-Phase<br><br>( <i>y</i> -axis on right side)<br>Phase |



| Plot             | X-Axis Units                             | Y-Axis Units                           |
|------------------|--|--|
|                  |  | Continuous Phase<br>Degrees<br>Radians |
| Group Delay      | Normalized Frequency<br>Linear Frequency | Samples<br>Time                        |
| Phase Delay      | Normalized Frequency<br>Linear Frequency | Degrees<br>Radians                     |
| Impulse Response | Samples<br>Time                          | Amplitude                              |
| Step Response    | Samples<br>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 “Display Data Values Interactively” in the MATLAB documentation for information on data tips.

If you have the DSP System Toolbox software, FVTool displays a specification mask along with your designed filter on a magnitude plot.

---

**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 1-752 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<sup>®</sup> properties (see Handle Graphics in the MATLAB documentation), FVTool has the following properties:

- **'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 DSP System 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'          |
| Magnitude response estimate<br><br>(available only with the DSP System Toolbox product, see <code>freqrespest</code> for more information) | 'magestimate'   |
| Round-off noise power<br><br>(available only with the DSP System Toolbox product, see <code>noisepsd</code> 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 DSP System 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.

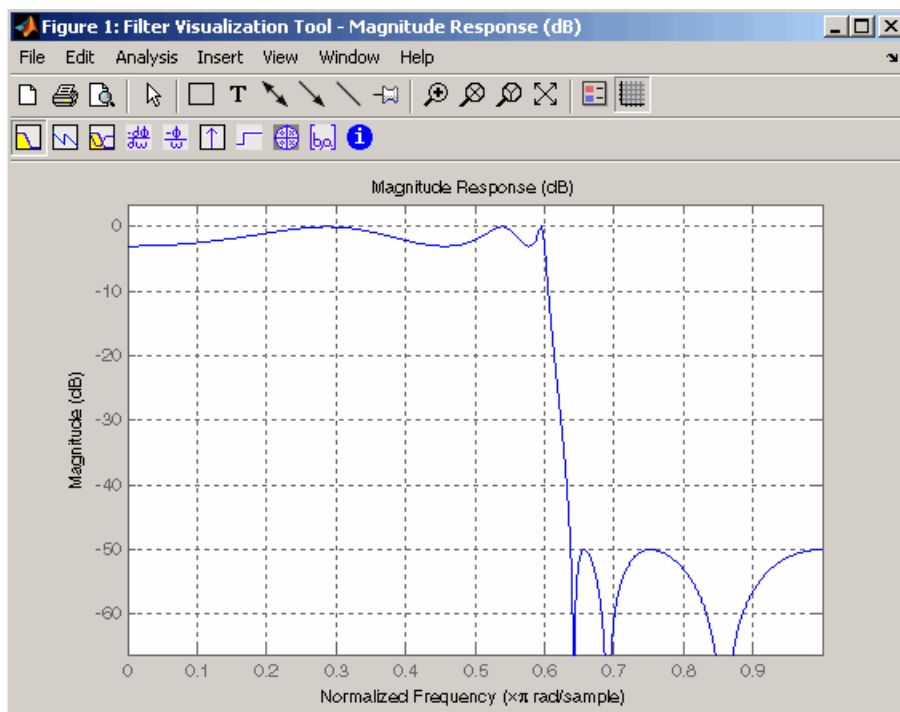
`legend(h, str1, str2, ...)` creates a legend in FVTool by associating `str1` with filter 1, `str2` with filter 2, etc. See `legend` in the MATLAB documentation for information.

## Examples

### Example 1

Display the magnitude response of an elliptic filter, starting FVTool from the command line:

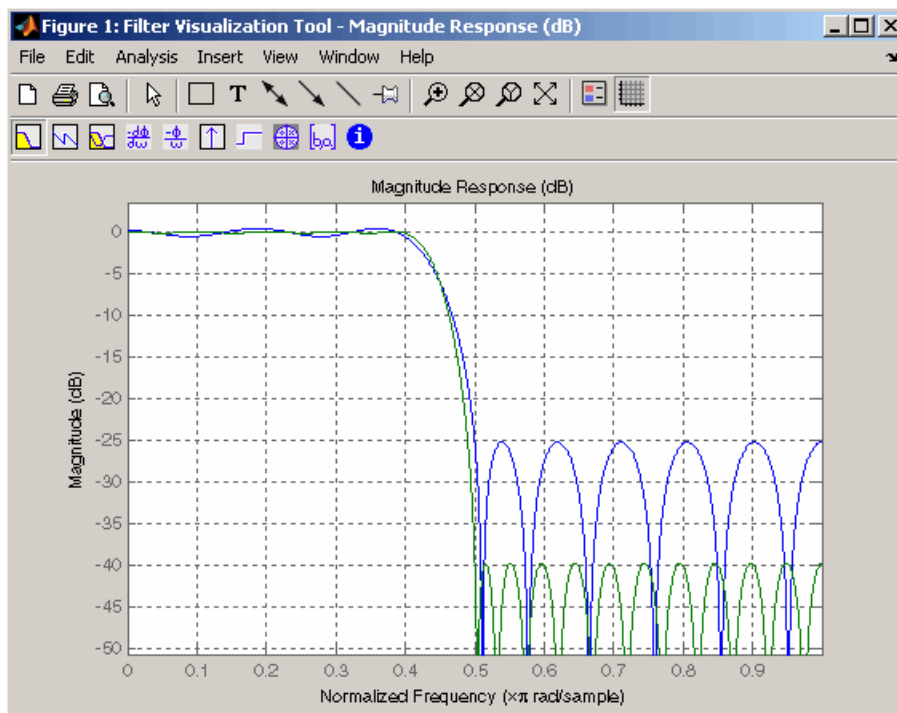
```
[b,a] = ellip(6,3,50,300/500);  
fvtool(b,a);
```

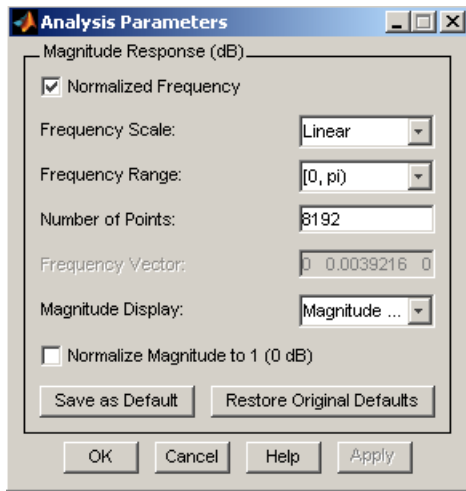


## Example 2

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

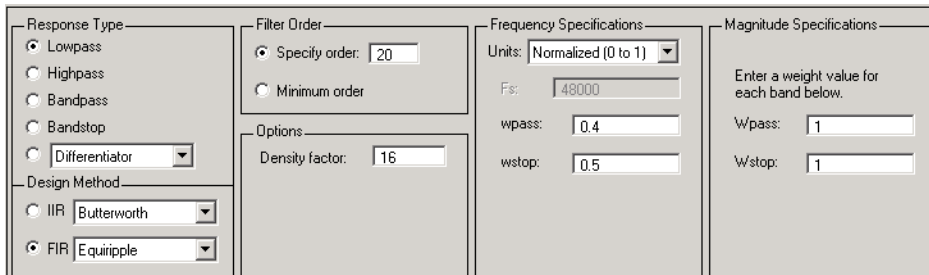
Create a lowpass, equiripple filter of order 20 in FDATool and display it in FVTool.

```
fdatool          % Start FDATool
```

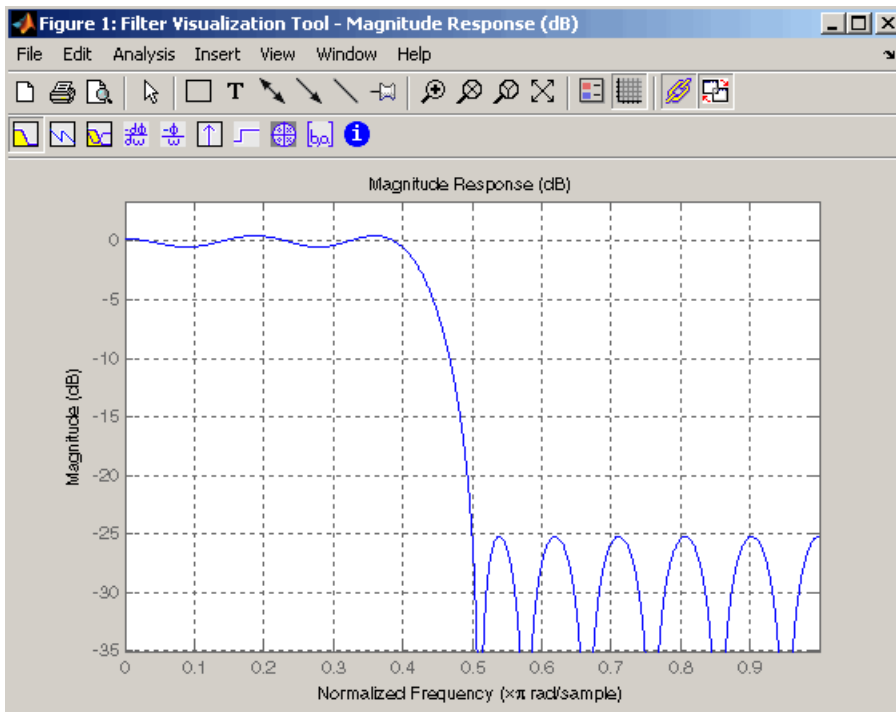
Set these parameters in `fdatool`:

| Parameter  | Setting             |
|--|---------------------|
| <b>Response Type</b>                             | Lowpass             |
| <b>Design Method: FIR</b>                        | Equiripple          |
| <b>Filter Order: Specify order</b>               | 20                  |
| <b>Options: Density Factor</b>                   | 16                  |
| <b>Frequency Specifications: Units</b>           | Normalized (0 to 1) |
| <b>wpass</b>                                     | 0.4                 |
| <b>wstop</b>                                     | 0.5                 |
| <b>Magnitude Specifications: Wpass and Wstop</b> | 1                   |

and then click the **Design Filter** button.



Click the **Full View Analysis** button to start FVTool.



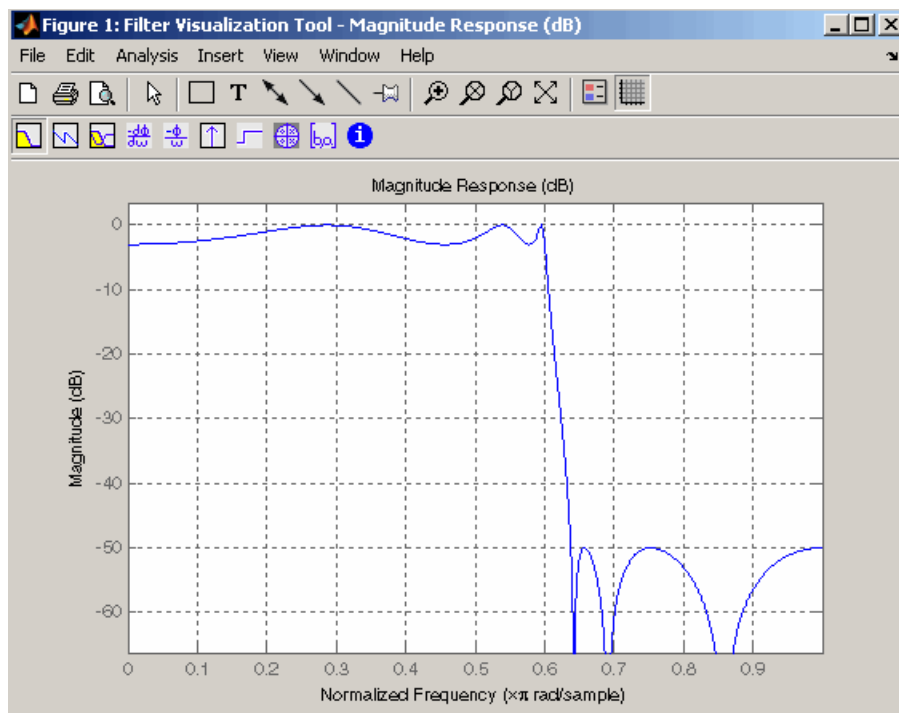
### Example 4

Create an elliptic filter and use some of FVTool's figure handle commands:

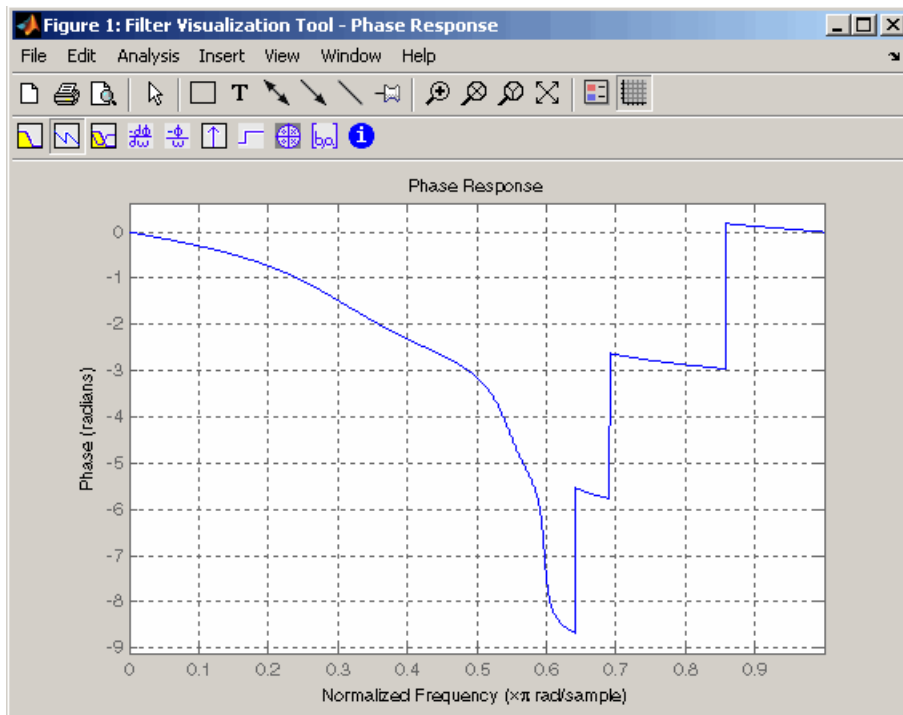
```
[b,a]=ellip(6,3,50,300/500);
```



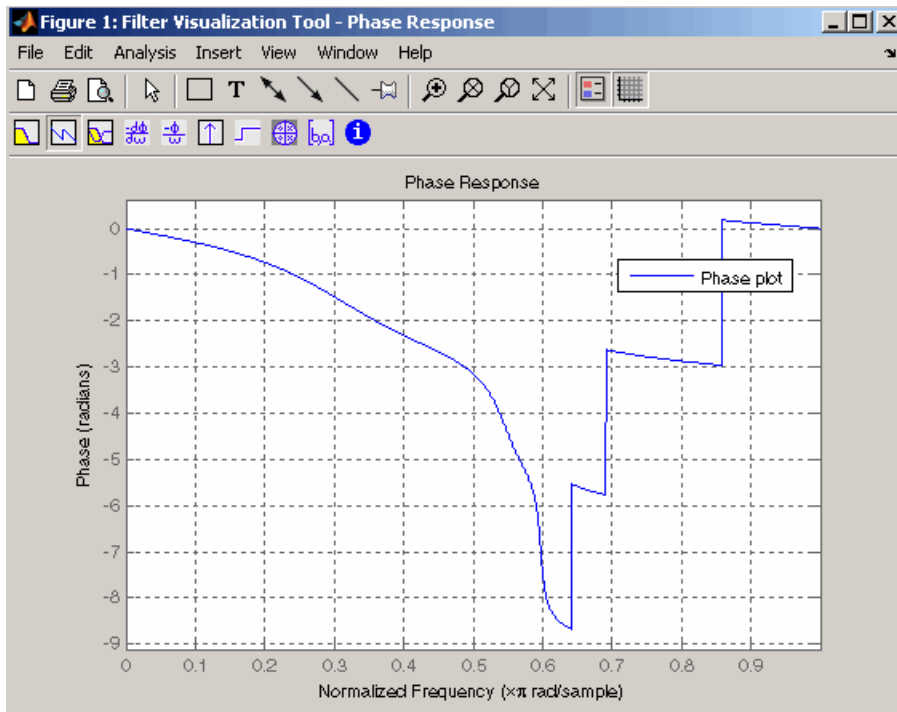
```
h = fvtool(b,a); % Create handle, h and start FVTool  
                % with magnitude plot
```



```
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'
        Grid: 'on'
        Legend: 'on'
        DesignMask: 'off'
        Fs: 1
    SOSViewSettings: [1x1 dspopts.sosview]
        Analysis: 'phase'
    OverlaidAnalysis: ''
        ShowReference: 'on'
        PolyphaseView: 'off'
    NormalizedFrequency: 'on'
        FrequencyScale: 'Linear'
        FrequencyRange: '[0, pi)'
        NumberofPoints: 8192
        FrequencyVector: [1x256 double]

```

```
PhaseUnits: 'Radians'  
PhaseDisplay: 'Phase'
```

**See Also**

`designfilt` | `digitalFilter` | `fdatool` | `sptool`

## fwht

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 the 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 the 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”.

## Examples

### Walsh-Hadamard Transform of a Signal

This example shows a simple input signal and its Walsh-Hadamard transform.

```
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 locations 0, 1, 3, and 6. Form the Walsh functions with the sequency values 0, 1, 3, and 6 to recreate x.

```
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 = y(0+1)*w0 + y(1+1)*w1 + y(3+1)*w3 + y(6+1)*w6
```

w =

```
      19      -1      11      -9      -7      13      -15      5
```

## More About

### Algorithms

The fast Walsh-Hadamard transform 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, Kenneth G. *Applications of Walsh and Related Functions: With an Introduction to Sequency Theory*. London: Academic Press, 1984.
- [2] Beer, Tom. “Walsh Transforms.” *American Journal of Physics*. Vol. 49, 1981, pp.466–472.

## See Also

ifwht | dct | idct | fft | ifft



# gauspuls

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. Note that the fractional bandwidth is specified in terms of power ratios. This corresponds to the -3 dB point expressed in magnitude ratios.

`[yi,yq] = gauspuls(...)` returns both the in-phase and quadrature pulses.

`[yi,yq,ye] = gauspuls(...)` returns the RF signal envelope.

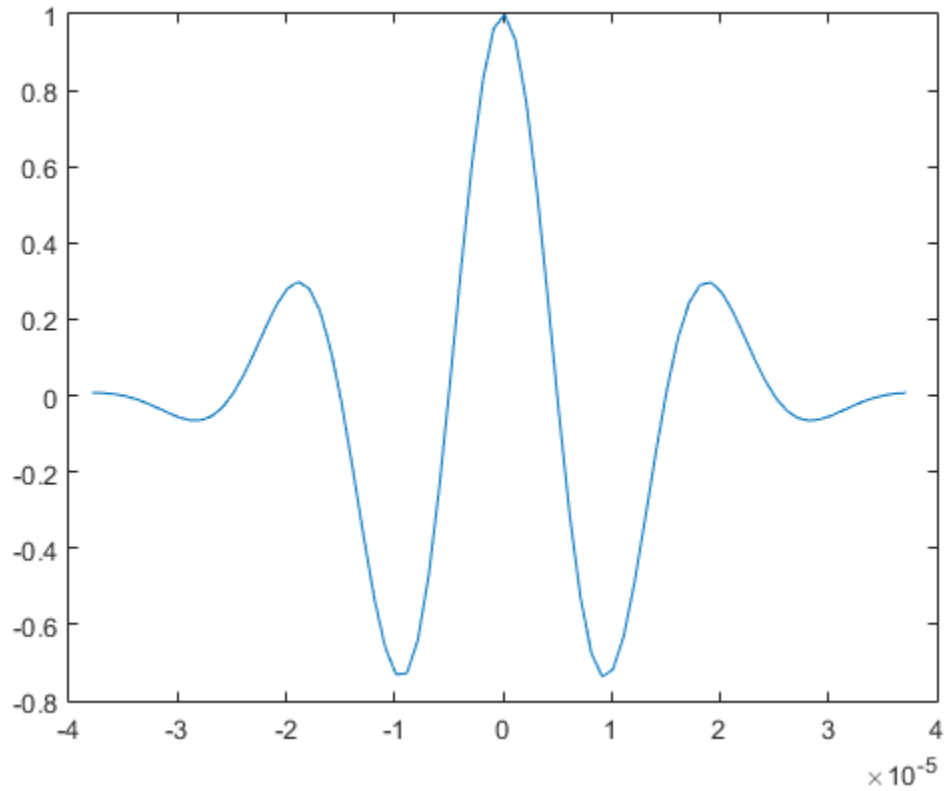
`tc = gauspuls('cutoff',fc,bw,bwr,tpe)` returns the cutoff time `tc` (greater than or equal to 0) at which the trailing pulse envelope falls below `tpe` dB with respect to the peak envelope amplitude. The trailing pulse envelope level `tpe` must be less than 0, because it indicates a reference level less than the peak (unity) envelope amplitude. The default value for `tpe` is -60 dB.

## Examples

### Generate a Gaussian RF Pulse

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)
```



## More About

### Tips

Default values are substituted for empty or omitted trailing input arguments.

### See Also

`chirp` | `cos` | `diric` | `pulstran` | `rectpuls` | `sawtooth` | `sin` | `sinc` | `square` | `tripuls`

## gaussdesign

Gaussian FIR pulse-shaping filter design

### Syntax

```
h = gaussdesign(bt,span,sps)
```

### Description

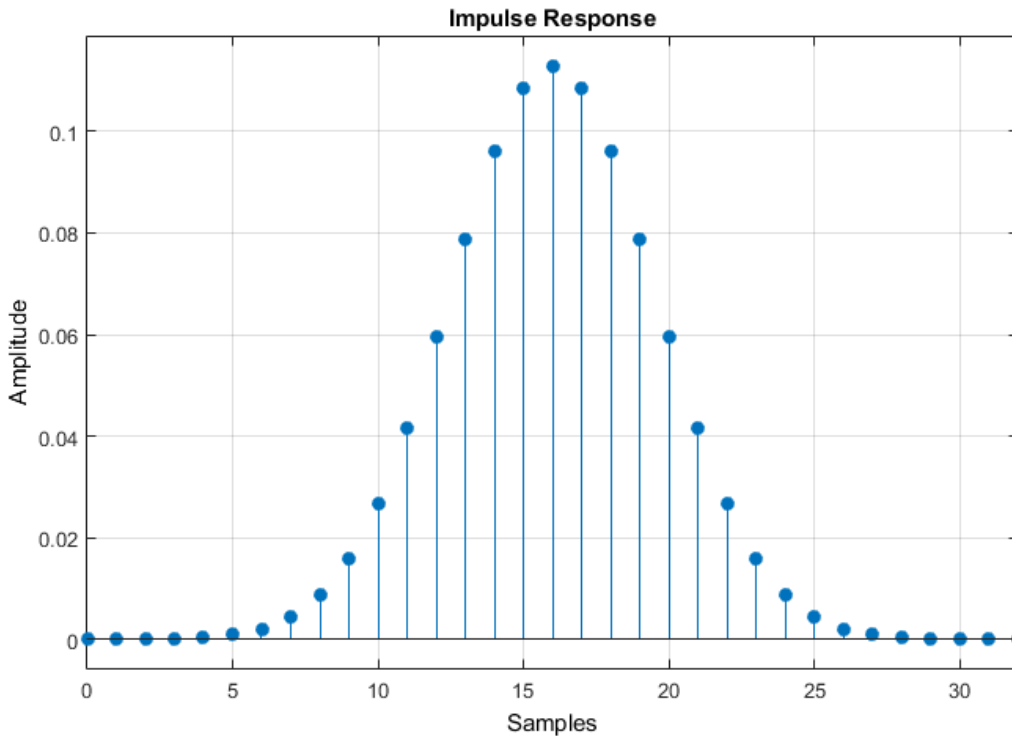
`h = gaussdesign(bt,span,sps)` designs a lowpass FIR Gaussian pulse-shaping filter and returns a vector, `h`, of filter coefficients. The filter is truncated to `span` symbols, and each symbol period contains `sps` samples. The order of the filter, `sps*span`, must be even.

### Examples

#### Gaussian Filter for a GSM GMSK Digital Cellular Communication System

Specify that the modulation used to transmit the bits is a Gaussian minimum-shift keying (GMSK) pulse. This pulse has a 3-dB bandwidth equal to 0.3 of the bit rate. Truncate the filter to 4 symbols and represent each symbol with 8 samples.

```
bt = 0.3;  
span = 4;  
sps = 8;  
h = gaussdesign(bt,span,sps);  
fvtool(h,'impulse')
```



## Input Arguments

### **bt** — 3-dB bandwidth-symbol time product

positive real scalar

Product of the 3-dB one-sided bandwidth, in hertz, and the symbol time, in seconds. Specify this value as a positive real scalar. Smaller values of **bt** produce larger pulse widths.

Data Types: double | single

### **span** — Number of symbols

3 (default) | positive integer scalar

Number of symbols, specified as a positive integer scalar.

Data Types: `double` | `single`

## **sps — Samples per symbol**

2 (default) | positive integer scalar

Number of samples per symbol period (oversampling factor), specified as a positive integer scalar.

Data Types: `double` | `single`

## **Output Arguments**

### **h — FIR filter coefficients**

row vector

FIR coefficients of the Gaussian pulse-shaping filter, returned as a row vector. The coefficients are normalized so that the nominal passband gain is always 1.

Data Types: `double` | `single`

## **References**

- [1] Rappaport, Theodore S. *Wireless Communications: Principles and Practice*. 2nd Ed. Upper Saddle River, NJ: Prentice Hall, 2002.
- [2] Krishnapura, N., S. Pavan, C. Mathiazhagan, and B. Ramamurthi. “A baseband pulse shaping filter for Gaussian minimum shift keying.” *Proceedings of the 1998 IEEE International Symposium on Circuits and Systems*. Vol. 1, 1998, pp. 249–252.

## **See Also**

`rcosdesign`

# gausswin

Gaussian window

## Syntax

```
w = gausswin(N)
w = gausswin(N,Alpha)
```

## Description

`w = gausswin(N)` returns an N-point Gaussian window in a column vector, `w`. `N` is a positive integer.

`w = gausswin(N,Alpha)` returns an N-point Gaussian window with `Alpha` proportional to the reciprocal of the standard deviation. The width of the window is inversely related to the value of  $\alpha$ . A larger value of  $\alpha$  produces a more narrow window. The value of  $\alpha$  defaults to 2.5.

---

**Note** If the window appears to be clipped, increase `N`, the number of points.

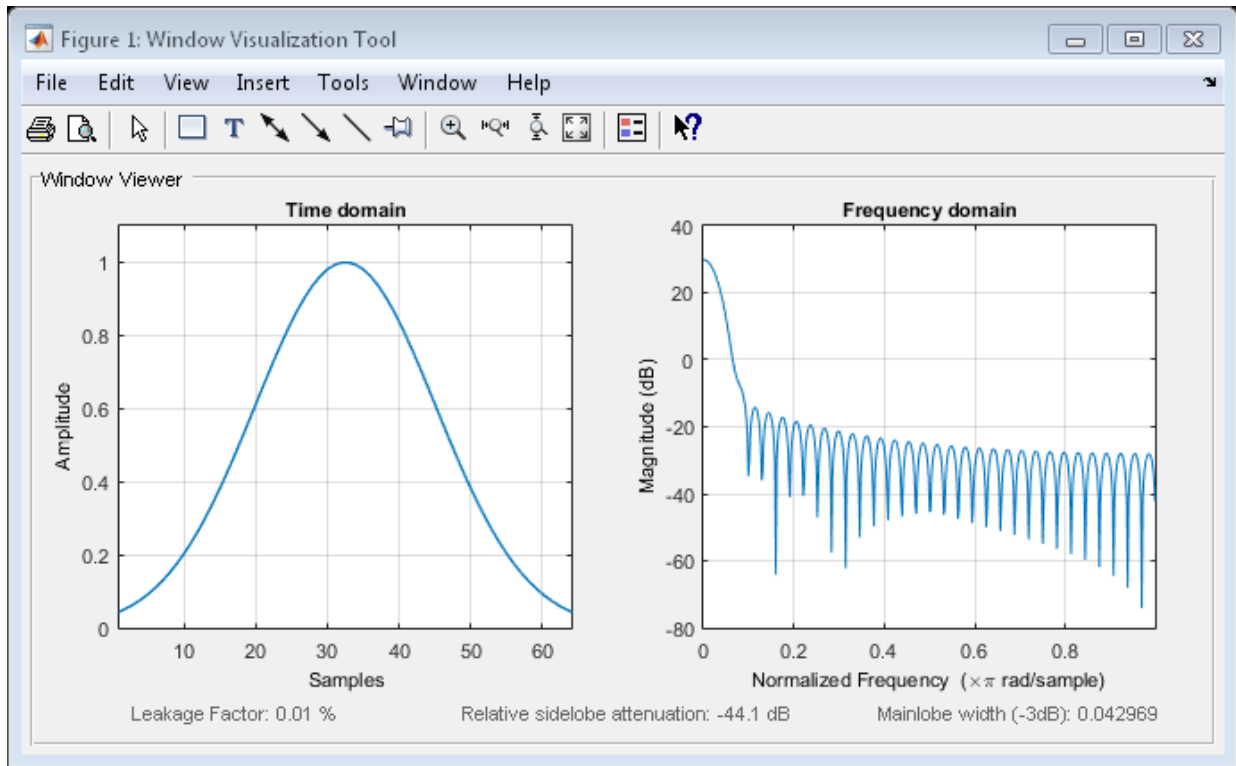
---

## Examples

### Gaussian Window

Create a 64-point Gaussian window. Display the result in `wvtool`.

```
L = 64;
wvtool(gausswin(L))
```



### Gaussian Window and the Fourier Transform

This example shows that the Fourier transform of the Gaussian window is also Gaussian with a reciprocal standard deviation. This is an illustration of the time-frequency uncertainty principle.

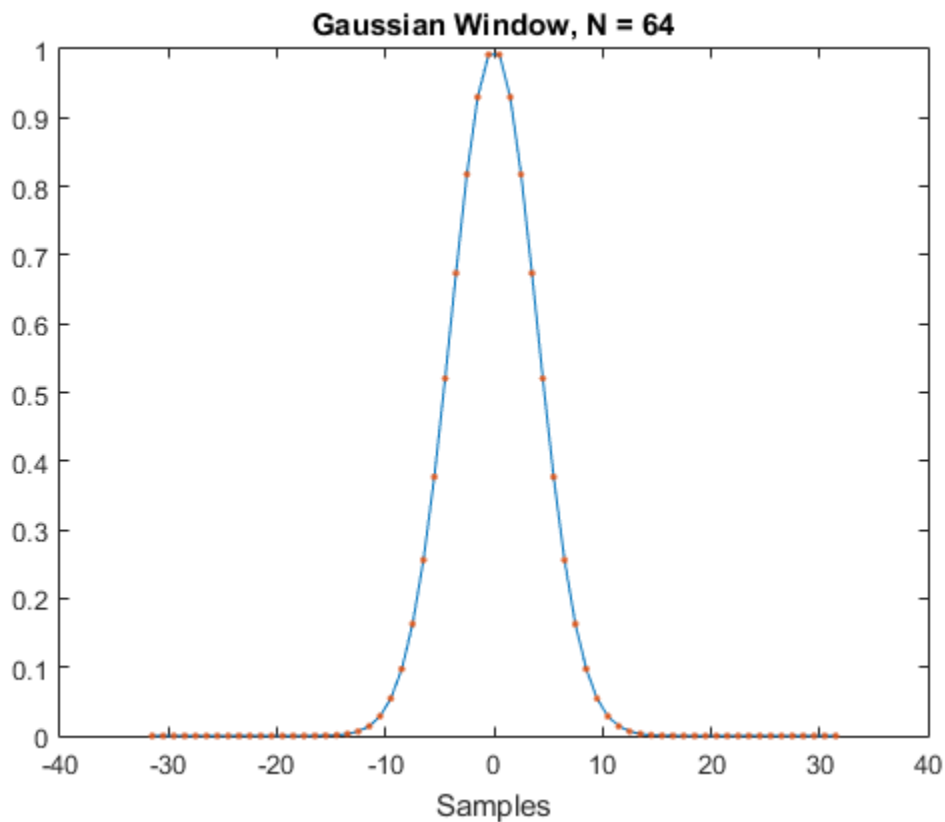
Create a Gaussian window of length 64 by using `gausswin` and the defining equation. Set  $\alpha = 8$ , which results in a standard deviation of  $64/16 = 4$ . Accordingly, you expect that the Gaussian is essentially limited to the mean plus or minus 3 standard deviations, or an approximate support of  $[-12, 12]$ .

```
N = 64;
n = -(N-1)/2:(N-1)/2;
alpha = 8;

w = gausswin(N,alpha);
```



```
stdev = (N-1)/(2*alpha);  
y = exp(-1/2*(n/stdev).^2);  
  
plot(n,w)  
hold on  
plot(n,y, '. ')  
hold off  
  
xlabel('Samples')  
title('Gaussian Window, N = 64')
```

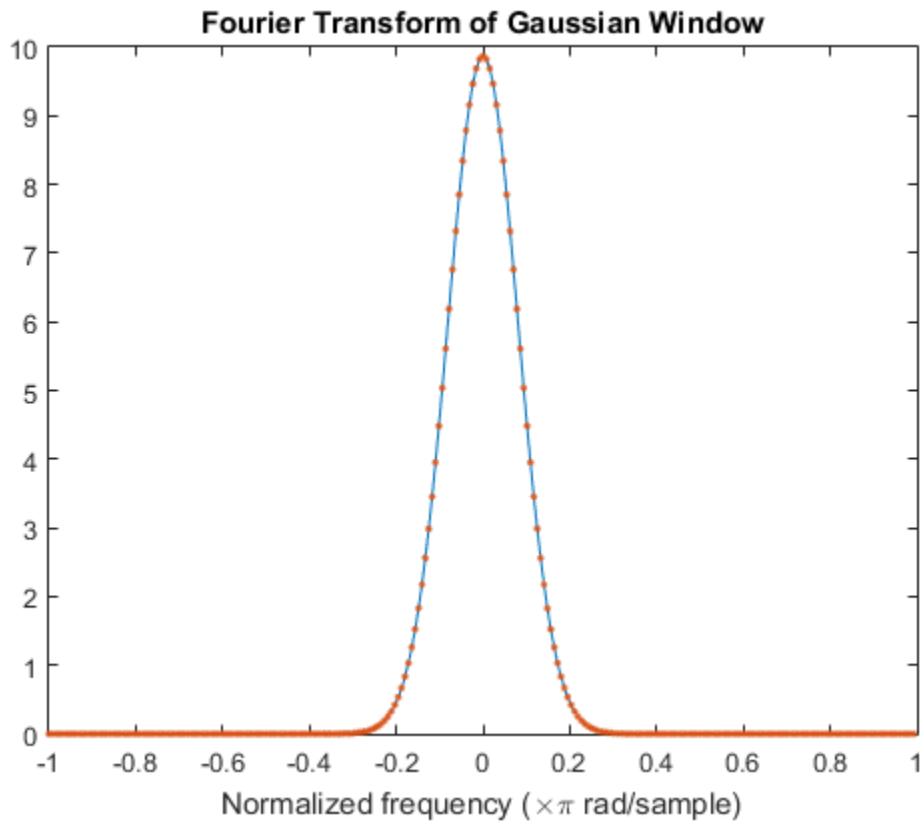


Obtain the Fourier transform of the Gaussian window at 256 points. Use `fftshift` to center the Fourier transform at zero frequency (DC).

```
nfft = 4*N;  
freq = -pi:2*pi/nfft:pi-pi/nfft;  
  
wdft = fftshift(fft(w,nfft));
```

The Fourier transform of the Gaussian window is also Gaussian with a standard deviation that is the reciprocal of the time-domain standard deviation. Include the Gaussian normalization factor in your computation.

```
ydfc = exp(-1/2*(freq/(1/stdev)).^2)*(stdev*sqrt(2*pi));  
  
plot(freq/pi,abs(wdft))  
hold on  
plot(freq/pi,abs(ydfc),'.')  
hold off  
  
xlabel('Normalized frequency (\times\pi rad/sample)')  
title('Fourier Transform of Gaussian Window')
```



## More About

### Algorithms

The coefficients of a Gaussian window are computed from the following equation:

$$w(n) = e^{-\frac{1}{2} \left( \alpha \frac{n}{(N-1)/2} \right)^2},$$

where  $-(N-1)/2 \leq n \leq (N-1)/2$  and  $a$  is inversely proportional to the standard deviation of a Gaussian random variable. The exact correspondence with the standard deviation,  $\sigma$ , of a Gaussian probability density function is  $\sigma = (N-1)/(2a)$ .

## References

- [1] Harris, Fredric J. “On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform.” *Proceedings of the IEEE*. Vol. 66, January 1978, pp.51–83.
- [2] Roberts, Richard A., and C. T. Mullis. *Digital Signal Processing*. Reading, MA: Addison-Wesley, 1987, pp.135–136.

## See Also

`chebwin` | `window` | `kaiser` | `tukeywin` | `wintool` | `wvtool`

# **gmonopuls**

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.

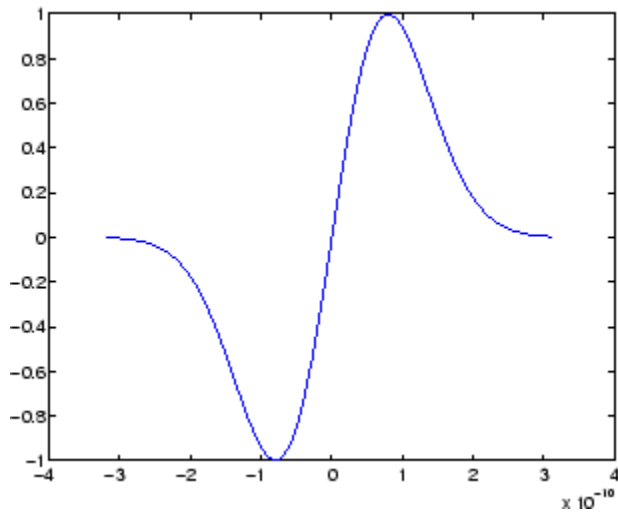
`tc = gmonopuls('cutoff',fc)` returns the time duration between the maximum and minimum amplitudes of the pulse.

## **Examples**

### **Example 1**

Plot a 2 GHz Gaussian monopulse sampled at a rate of 100 GHz:

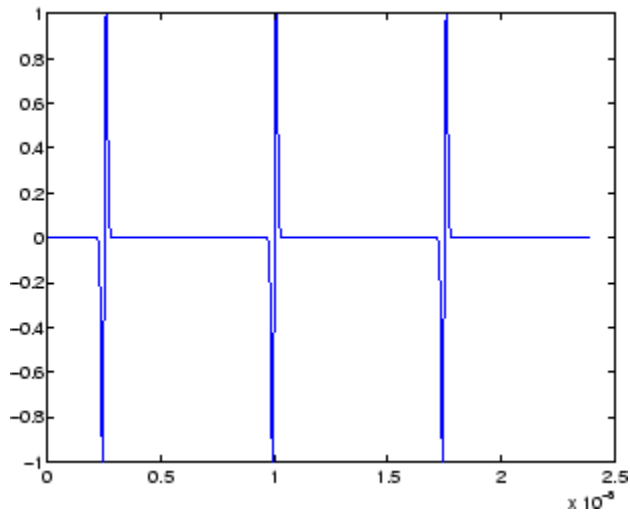
```
fc = 2E9; fs=100E9;
tc = gmonopuls('cutoff',fc);
t = -2*tc : 1/fs : 2*tc;
y = gmonopuls(t,fc); plot(t,y)
```



## 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)
```



## More About

### Tips

Default values are substituted for empty or omitted trailing input arguments.

### See Also

`chirp` | `gauspuls` | `pulstran` | `rectpuls` | `tripuls`

## goertzel

Discrete Fourier transform with second-order Goertzel algorithm

### Syntax

```
dft_data = goertzel(data)
dft_data = goertzel(data,freq_indices)
dft_data = goertzel(data,freq_indices,dim)
```

### Description

`dft_data = goertzel(data)` returns the discrete Fourier transform (DFT) of the input data, `data`, using a second-order Goertzel algorithm. If `data` is a matrix, `goertzel` computes the DFT of each column separately.

`dft_data = goertzel(data,freq_indices)` returns the DFT for the frequency indices `freq_indices`.

`dft_data = goertzel(data,freq_indices,dim)` computes the DFT of the matrix `data` along the dimension `dim`.

### Examples

#### Estimate Telephone Keypad Frequencies

Estimate the frequencies of the tone generated by pressing the 1 button on a telephone keypad.

The number 1 produces a tone with frequencies 697 and 1209 Hz. Generate 205 samples of the tone with a sample rate of 8 kHz.

```
Fs = 8000;
N = 205;
lo = sin(2*pi*697*(0:N-1)/Fs);
```



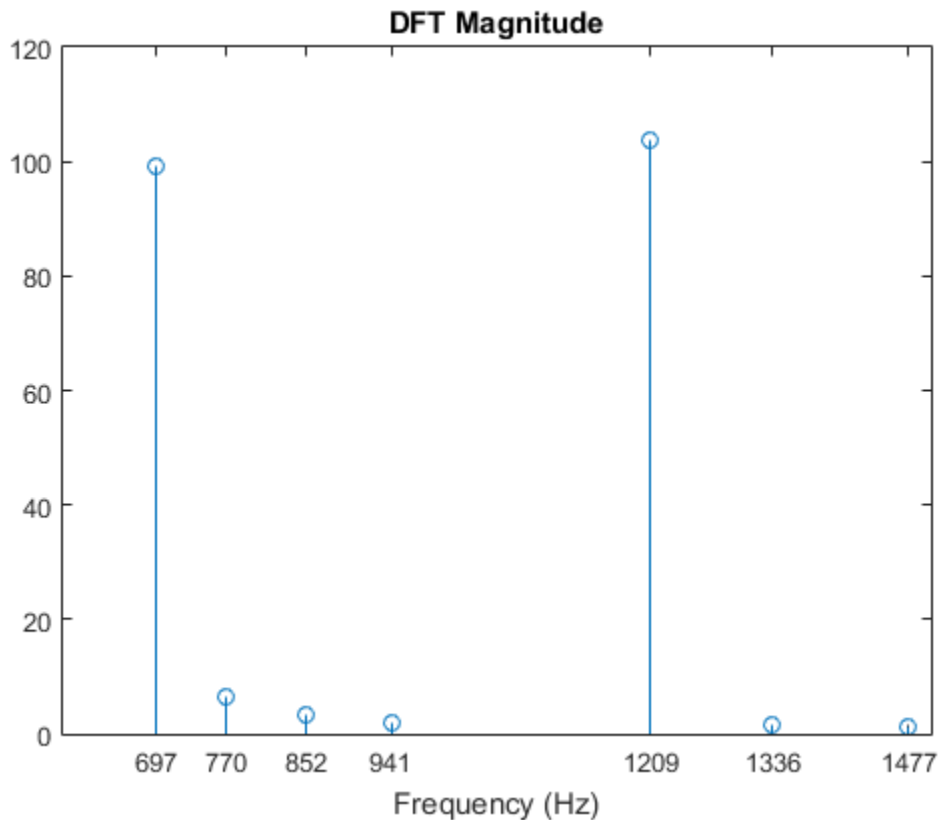
```
hi = sin(2*pi*1209*(0:N-1)/Fs);  
data = lo + hi;
```

Use the Goertzel algorithm to compute the DFT of the tone. Choose the indices corresponding to the frequencies used to generate the numbers 0 through 9.

```
f = [697 770 852 941 1209 1336 1477];  
freq_indices = round(f/Fs*N) + 1;  
dft_data = goertzel(data,freq_indices);
```

Plot the DFT magnitude.

```
stem(f,abs(dft_data))  
  
ax = gca;  
ax.XTick = f;  
xlabel('Frequency (Hz)')  
title('DFT Magnitude')
```



### Resolve Frequency Components of a Noisy Tone

Generate a noisy cosine with frequency components at 1.24 kHz, 1.26 kHz, and 10 kHz. Specify a sample rate of 32 kHz. Reset the random number generator for reproducible results.

```
rng default
```

```
Fs = 32e3;
```

```
t = 0:1/Fs:2.96;
```

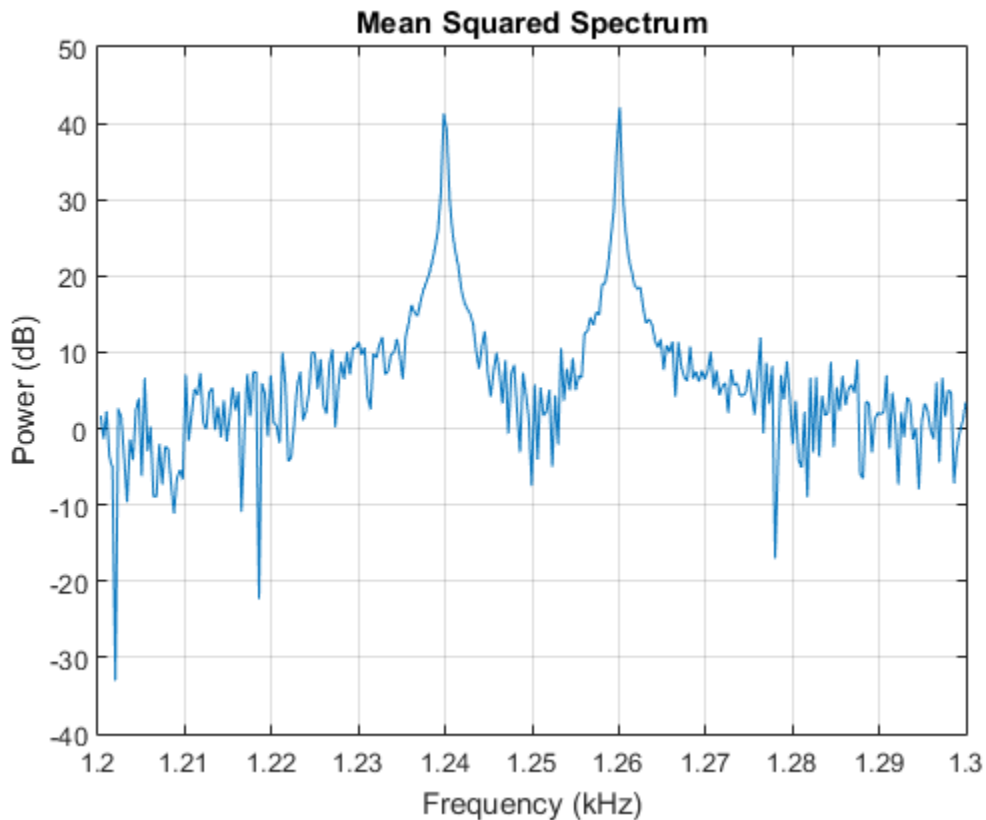
```
x = cos(2*pi*t*10e3) + cos(2*pi*t*1.24e3) + cos(2*pi*t*1.26e3) ...  
    + randn(size(t));
```

Generate the frequency vector. Use the Goertzel algorithm to compute the DFT. Restrict the range of frequencies to between 1.2 and 1.3 kHz.

```
N = (length(x)+1)/2;  
f = (Fs/2)/N*(0:N-1);  
indx = find(f>1.2e3 & f<1.3e3);  
X = goertzel(x,indx);
```

Plot the mean squared spectrum expressed in decibels.

```
plot(f(indx)/1e3,mag2db(abs(X)/length(X)))  
  
title('Mean Squared Spectrum')  
xlabel('Frequency (kHz)')  
ylabel('Power (dB)')  
grid
```



## Alternatives

You can also compute the DFT with:

- `fft`: less efficient than the Goertzel algorithm when you only need the DFT at a few frequencies.
- `czt`, the chirp Z-transform. `czt` calculates the Z-transform of an input on a circular or spiral contour and includes the DFT as a special case.

## More About

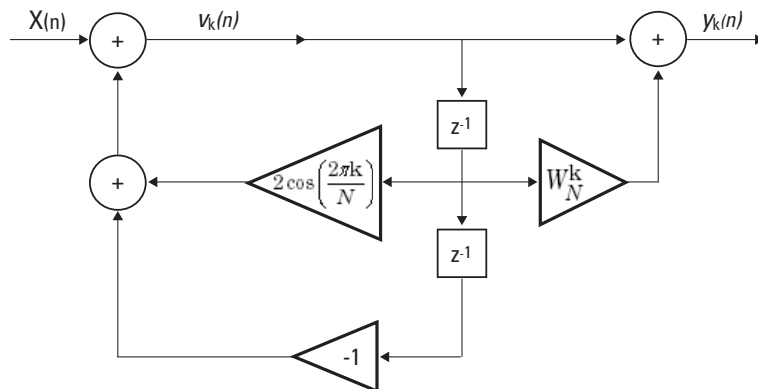
### Algorithms

The Goertzel algorithm implements the DFT as a recursive difference equation. To establish this difference equation, express the DFT as the convolution of an  $N$ -point input,  $x(n)$ , with the impulse response  $h(n) = W_N^{-kn}u(n)$ , where  $W_N^{-kn} = e^{-j2\pi k/N}$  and  $u(n)$  is the unit step sequence.

The Z-transform of the impulse response is

$$H(z) = \frac{1 - W_N^k z^{-1}}{1 - 2 \cos(2\pi k / N) z^{-1} + z^{-2}}$$

The direct form II implementation is:



## References

Proakis, John G., and Dimitris G. Manolakis. *Digital Signal Processing: Principles, Algorithms, and Applications*. 3rd Edition. Upper Saddle River, NJ: Prentice Hall, 1996, pp. 480–481.

Burrus, C. Sidney, and Thomas W. Parks. *DFT/FFT and Convolution Algorithms: Theory and Implementation*. New York: John Wiley & Sons, 1985.

# grpdelay

Average filter delay (group delay)

## Syntax

```
[gd,w] = grpdelay(b,a)
[gd,w] = grpdelay(b,a,n)
[gd,w] = grpdelay(sos,n)
[gd,w] = grpdelay(d,n)
[gd,f] = grpdelay(...,n,fs)
[gd,w] = grpdelay(...,n,'whole')
[gd,f] = grpdelay(...,n,'whole',fs)
gd = grpdelay(...,w)
gd = grpdelay(...,f,fs)
grpdelay(...)
```

## Description

`[gd,w] = grpdelay(b,a)` returns the group delay response, **gd**, of the discrete-time filter specified by the input vectors, **b** and **a**. The input vectors are the coefficients for the numerator, **b**, and denominator, **a**, polynomials in  $z^{-1}$ . The Z-transform of the discrete-time filter is

$$H(z) = \frac{B(z)}{A(z)} = \frac{\sum_{l=0}^{N-1} b(n+1)z^{-l}}{\sum_{l=0}^{M-1} a(l+1)z^{-l}},$$

The filter's group delay response is evaluated at 512 equally spaced points in the interval  $[0,\pi)$  on the unit circle. The evaluation points on the unit circle are returned in **w**.

`[gd,w] = grpdelay(b,a,n)` returns the group delay response of the discrete-time filter evaluated at **n** equally spaced points on the unit circle in the interval  $[0,\pi)$ . **n** is a positive integer. For best results, set **n** to a value greater than the filter order.

`[gd,w] = grpdelay(sos,n)` returns the group delay response for the second-order sections matrix, `sos`. `sos` is a  $K$ -by-6 matrix, where the number of sections,  $K$ , must be greater than or equal to 2. If the number of sections is less than 2, `grpdelay` considers the input to be the numerator vector, `b`. Each row of `sos` corresponds to the coefficients of a second-order (biquad) filter. The  $i$ th row of the `sos` matrix corresponds to `[bi(1) bi(2) bi(3) ai(1) ai(2) ai(3)]`.

`[gd,w] = grpdelay(d,n)` returns the group delay response for the digital filter, `d`. Use `designfilt` to generate `d` based on frequency-response specifications.

`[gd,f] = grpdelay(...,n,fs)` specifies a positive sampling frequency `fs` in hertz. It returns a length-`n` vector, `f`, containing the frequency points in hertz at which the group delay response is evaluated. `f` contains `n` points between 0 and `fs/2`.

`[gd,w] = grpdelay(...,n,'whole')` and `[gd,f] = grpdelay(...,n,'whole',fs)` use `n` points around the whole unit circle (from 0 to  $2\pi$ , or from 0 to `fs`).

`gd = grpdelay(...,w)` and `gd = grpdelay(...,f,fs)` return the group delay response evaluated at the angular frequencies in `w` (in radians/sample) or in `f` (in cycles/unit time), respectively, where `fs` is the sampling frequency. `w` and `f` are vectors with at least two elements.

`grpdelay(...)` with no output arguments plots the group delay response versus frequency.

`grpdelay` works for both real and complex filters.

---

**Note:** If the input to `grpdelay` is single precision, the group delay is calculated using single-precision arithmetic. The output, `gd`, is single precision.

---

## Examples

### Group Delay of a Butterworth Filter

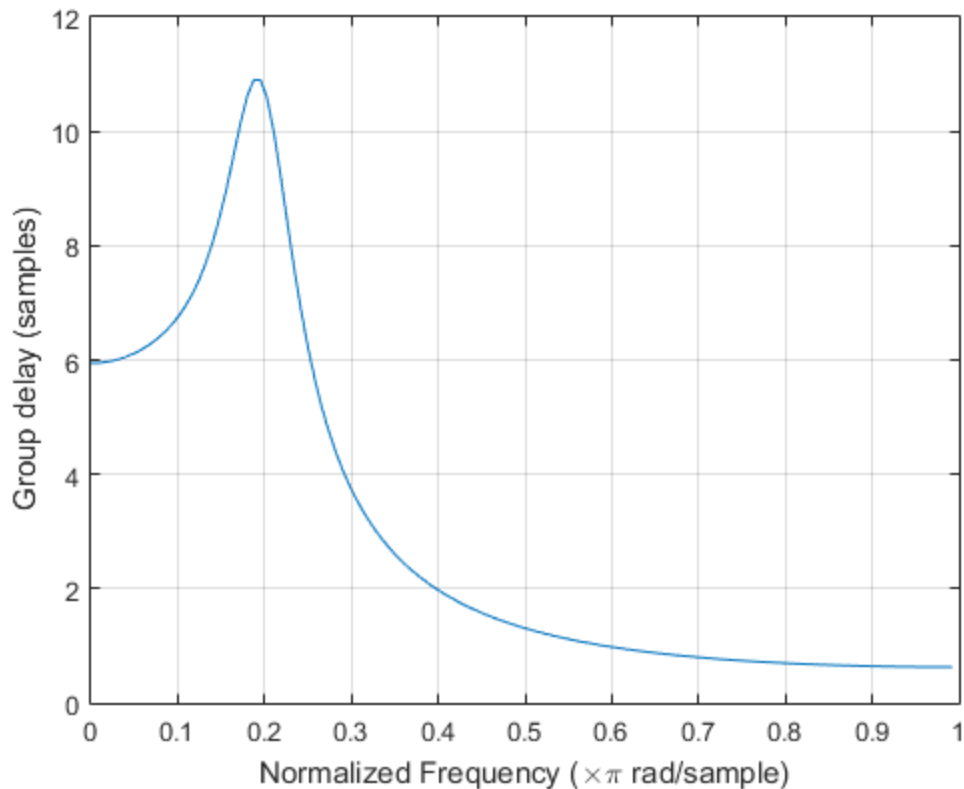
Design a Butterworth filter of order 6 with normalized 3-dB frequency  $0.2\pi$  rad/sample. Use `grpdelay` to display the group delay.

```
[z,p,k] = butter(6,0.2);
```

```

sos = zp2sos(z,p,k);
grpdelay(sos,128)

```



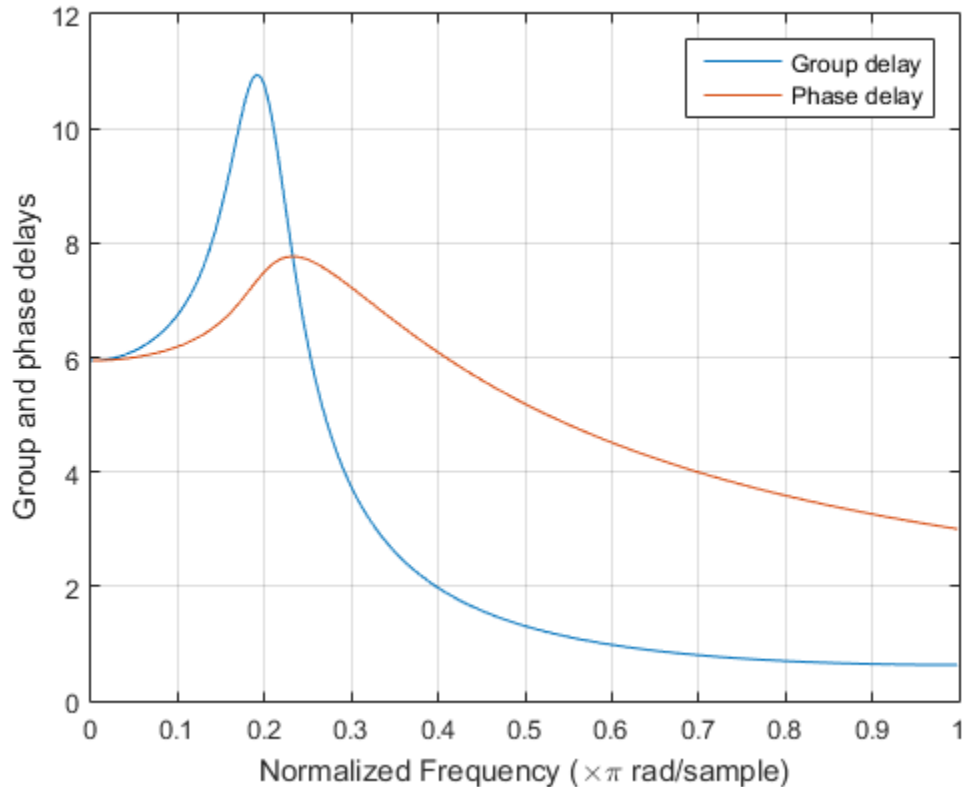
Plot both the group delay and the phase delay of the system on the same figure.

```

gd = grpdelay(sos,512);
[h,w] = freqz(sos,512);
pd = -unwrap(angle(h))./w;
plot(w/pi,gd,w/pi,pd), grid
xlabel 'Normalized Frequency (\times\pi rad/sample)'
ylabel 'Group and phase delays'
legend('Group delay','Phase delay')

```

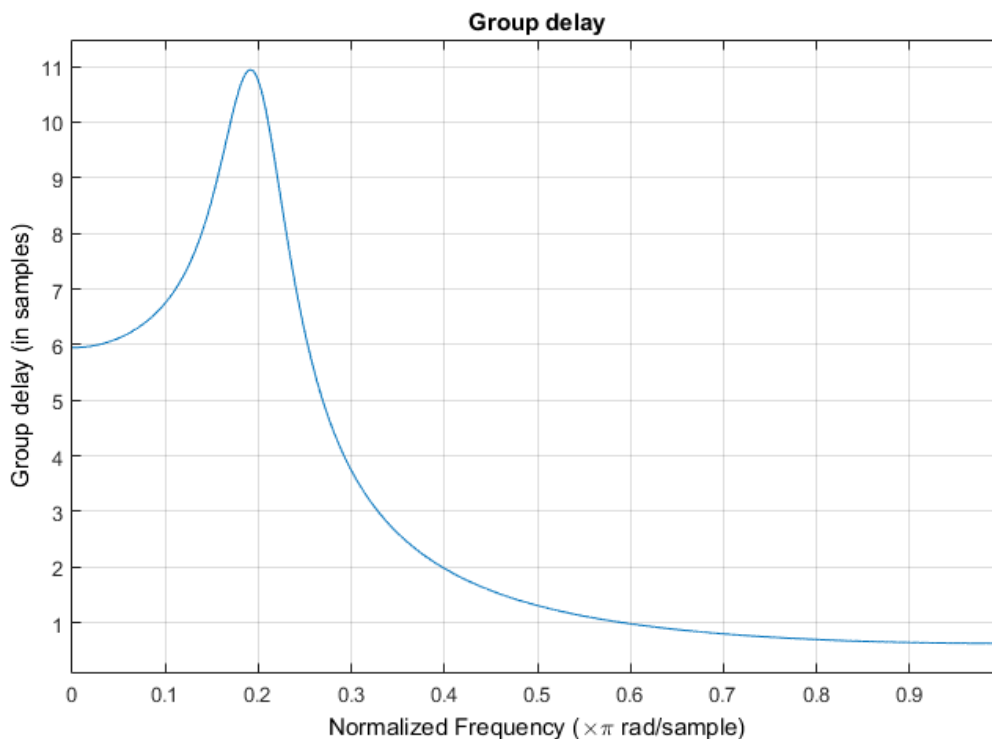




### Group Delay Response of a Butterworth digitalFilter

Use `designfilt` to design a sixth-order Butterworth Filter with normalized 3-dB frequency  $0.2\pi$  rad/sample. Display its group delay response.

```
d = designfilt('lowpassiir', 'FilterOrder', 6, ...  
              'HalfPowerFrequency', 0.2, 'DesignMethod', 'butter');  
grpdelay(d)
```



## More About

### Group Delay

The *group delay response* 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 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  $\theta(\omega)$  is the phase, or argument, of  $H(e^{j\omega})$ .

**Algorithms**

grpdelay multiplies the filter coefficients by a unit ramp. After Fourier transformation, this process corresponds to differentiation.

**See Also**

cceps | designfilt | digitalFilter | fft | freqz | fvtool | hilbert | icceps  
| phasedelay | rceps

# hamming

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.

$$w(n) = 0.54 - 0.46 \cos\left(2\pi \frac{n}{N}\right), \quad 0 \leq n \leq N$$

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.

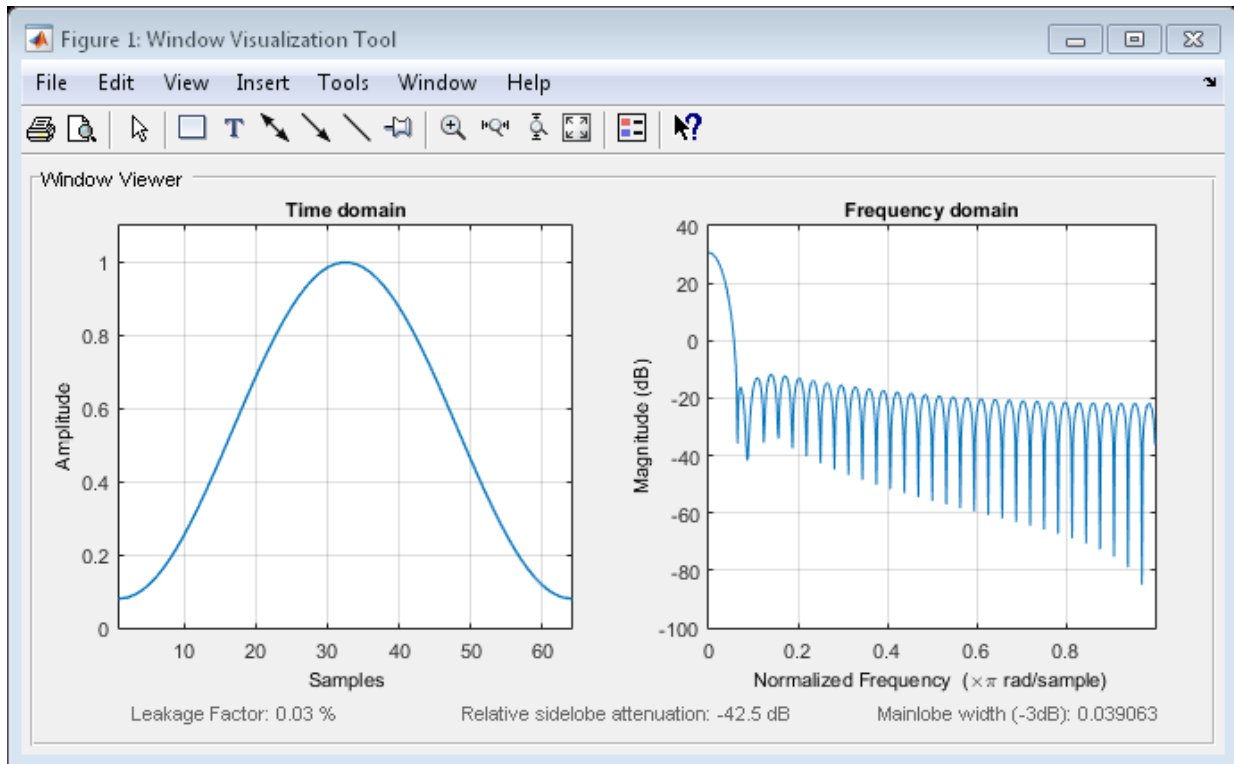
---

## Examples

### Hamming Window

Create a 64-point Hamming window. Display the result using `wvtool`.

```
L = 64;
wvtool(hamming(L))
```



## References

- [1] Oppenheim, Alan V., Ronald W. Schaffer, and John R. Buck. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1999, p. 468.

## See Also

blackman | flattopwin | hann | window | wintool | wvtool

# **hann**

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.

$$w(n) = 0.5 \left( 1 - \cos \left( 2\pi \frac{n}{N} \right) \right), \quad 0 \leq n \leq N$$

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.

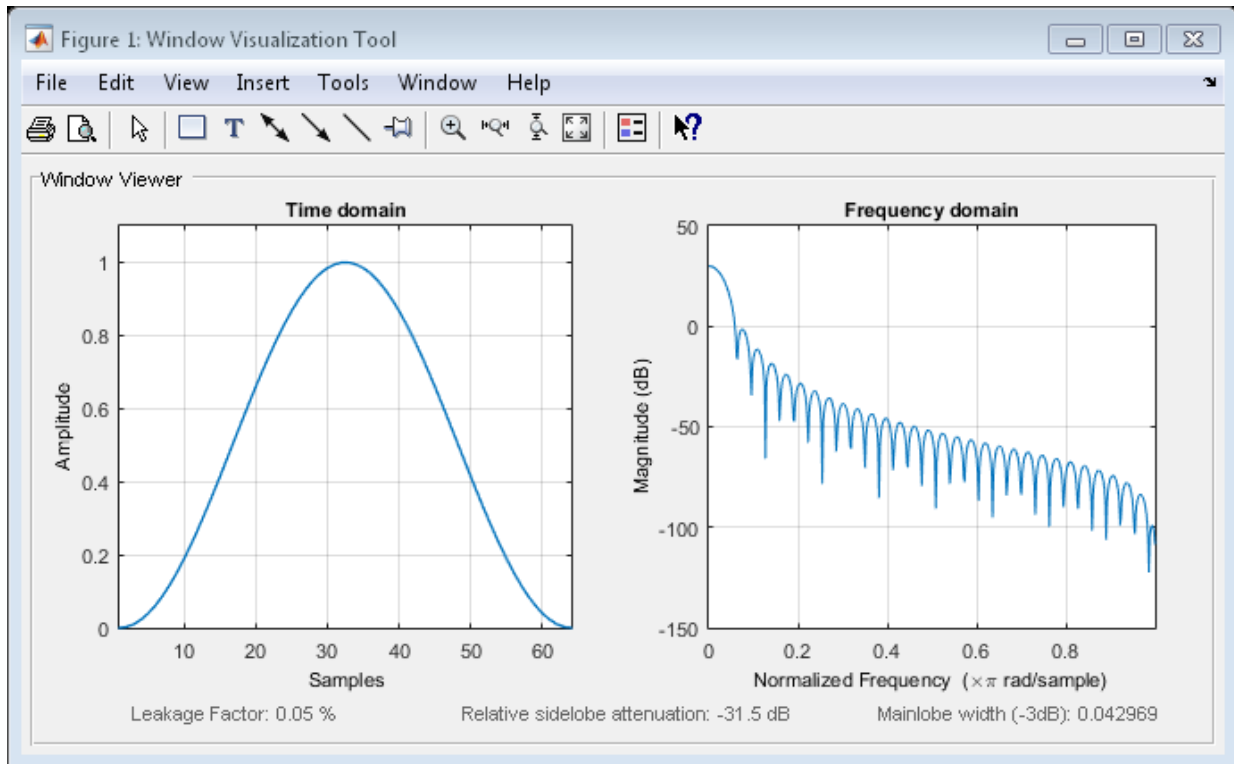
---

## **Examples**

### **Hann Window**

Create a 64-point Hann window. Display the result using `wvtool`.

```
L = 64;
wvtool(hann(L))
```



## References

- [1] Oppenheim, Alan V., Ronald W. Schaffer, and John R. Buck. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1999, p. 468.

## See Also

blackman | flattopwin | hamming | window | wintool | wvtool

# hilbert

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  $x = x_r + i \cdot x_i$  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 conversely. The Hilbert transformed series has the same amplitude and frequency content as the original sequence and includes phase information that depends on the phase of the original.

If `xr` is a matrix, `x = hilbert(xr)` operates columnwise on the matrix, finding the analytic signal corresponding to 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 how 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. See “Hilbert Transform and Instantaneous Frequency” for examples.

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  $\text{fft}(x)$  is zero, and the first (DC) and center (Nyquist) elements of  $\text{fft}(x)$  are purely real.

## Examples

### Analytic Signal of a Sequence

Define a sequence and compute its analytic signal using `hilbert`.

```
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
```

The imaginary part of  $x$  is the Hilbert transform of  $xr$ , and the real part is  $xr$  itself.

```
imx = imag(x)
rex = real(x)
```

```
imx =
```

```
1 -1 -1 1
```

```
rex =
```

```
1 2 3 4
```

The last half of the DFT of  $x$  is zero. (In this example, the last half of the transform is just the last element.) The DC and Nyquist elements of  $\text{fft}(x)$  are purely real.

```
dft = fft(x)
```

```
dft =
```

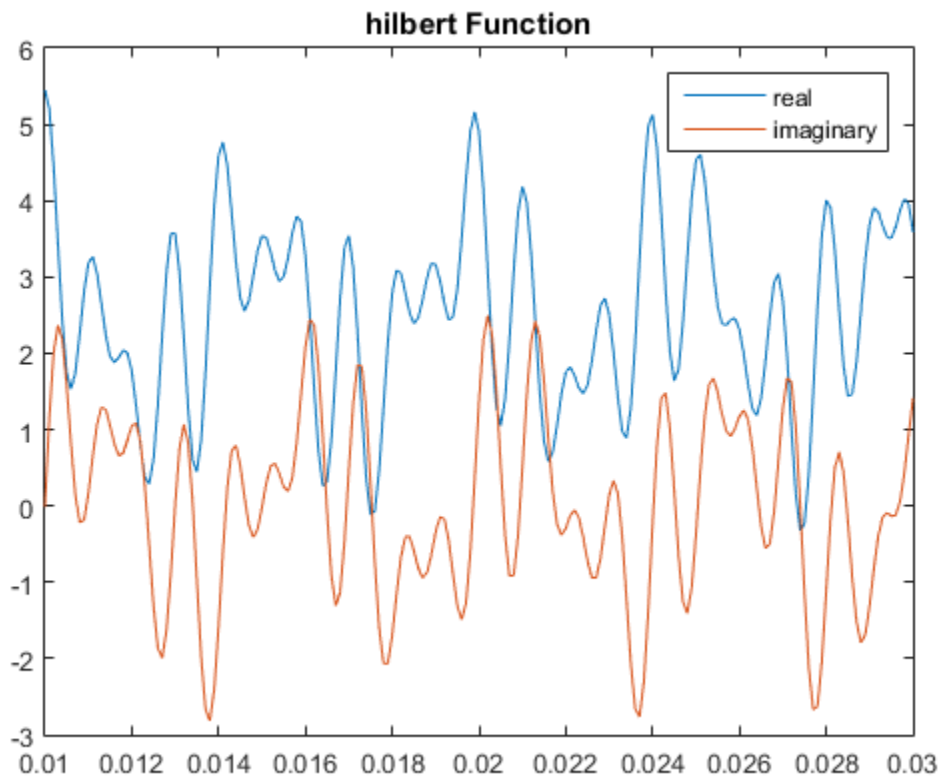
```
10.0000 + 0.0000i -4.0000 + 4.0000i -2.0000 + 0.0000i 0.0000 + 0.0000i
```

### Analytic Signal and Hilbert Transform

The `hilbert` function finds the exact analytic signal for a finite block of data. You can also generate the analytic signal by using an FIR Hilbert transformer filter to compute an approximation to the imaginary part.

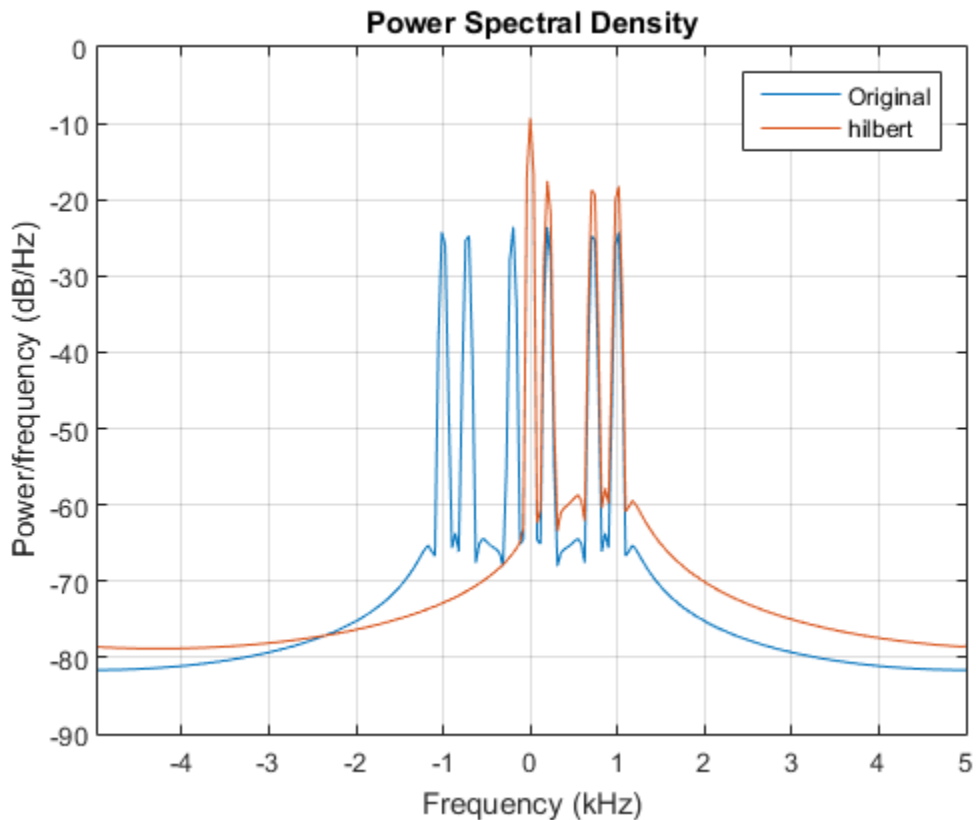
Generate a sequence composed of three sinusoids with frequencies 203, 721, and 1001 Hz. The sequence is sampled at 10 kHz for about 1 second. Use the `hilbert` function to compute the analytic signal. Plot it between 0.01 seconds and 0.03 seconds.

```
fs = 1e4;  
t = 0:1/fs:1;  
  
x = 2.5+cos(2*pi*203*t)+sin(2*pi*721*t)+cos(2*pi*1001*t);  
  
y = hilbert(x);  
  
plot(t,real(y),t,imag(y))  
xlim([0.01 0.03])  
legend('real','imaginary')  
title('hilbert Function')
```



Compute Welch estimates of the power spectral densities of the original sequence and the analytic signal. Divide the sequences into Hamming windowed nonoverlapping sections of length 256. Verify that the analytic signal has no power at negative frequencies.

```
pwelch([x;y].',256,0,[],fs,'centered')  
legend('Original','hilbert')
```



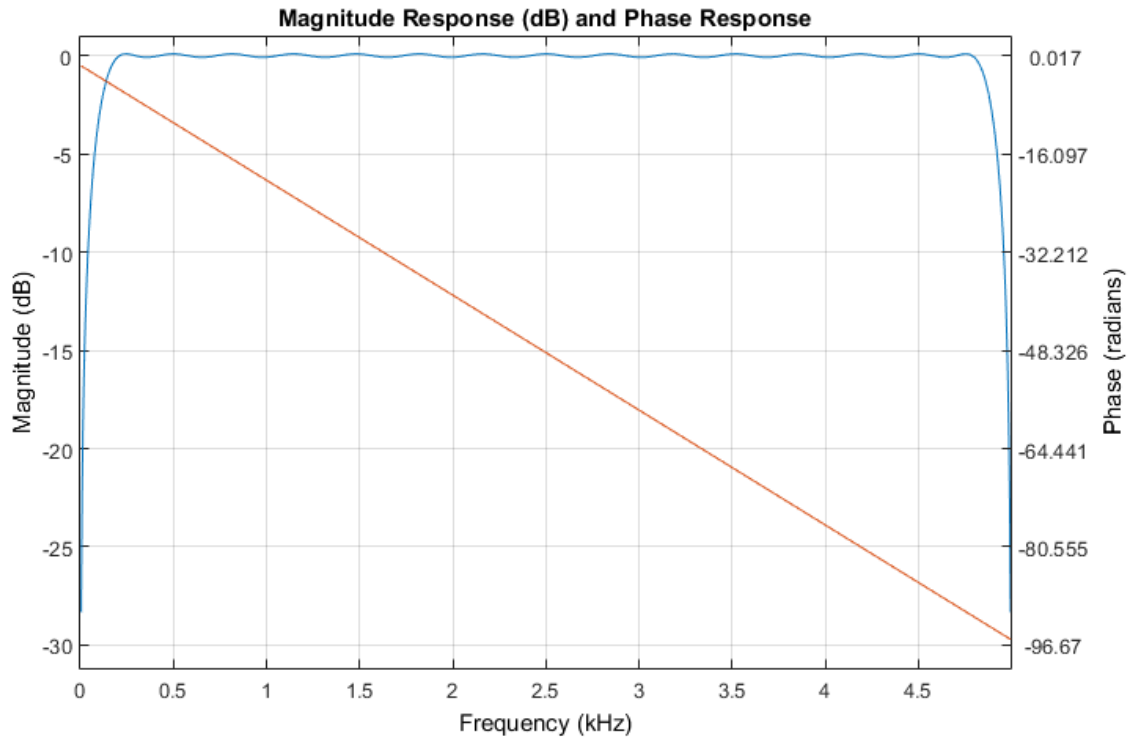
Use the `designfilt` function to design a 60th-order Hilbert transformer FIR filter. Specify a transition width of 400 Hz. Visualize the frequency response of the filter. Filter the sinusoidal sequence to approximate the imaginary part of the analytic signal.

```
fo = 60;

d = designfilt('hilbertfir','FilterOrder',fo, ...
              'TransitionWidth',400,'SampleRate',fs);

freqz(d,1024,fs)

hb = filter(d,x);
```

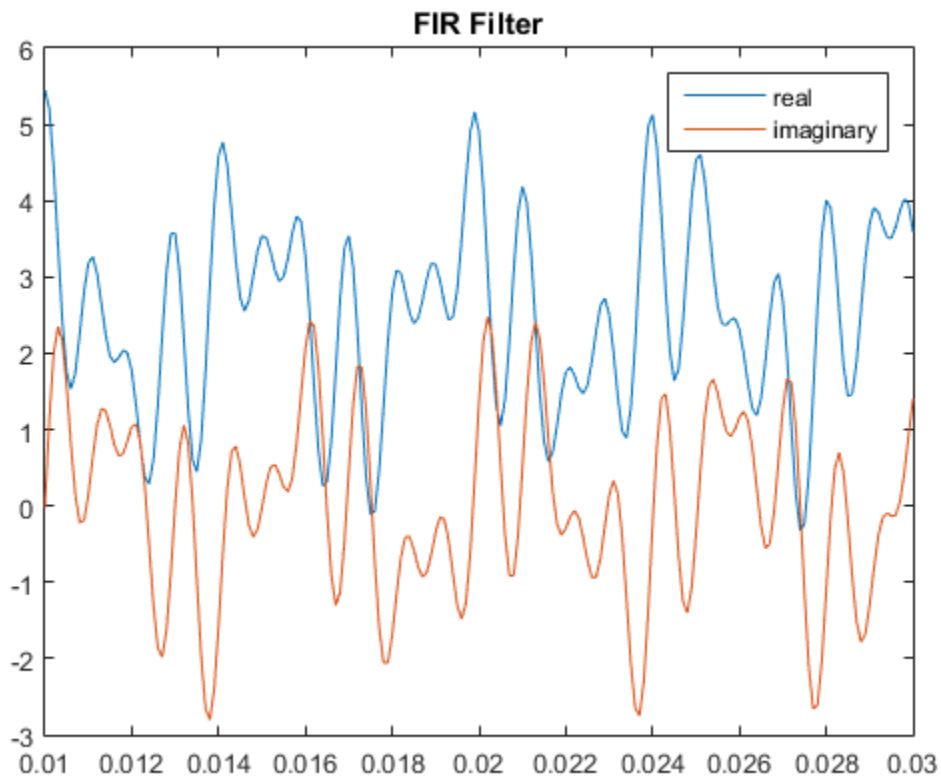


The group delay of the filter, `grd`, is equal to one-half the filter order. Compensate for this delay. Remove the first `grd` samples of the imaginary part and the last `grd` samples of the real part and the time vector. Plot the result between 0.01 seconds and 0.03 seconds.

```
grd = fo/2;

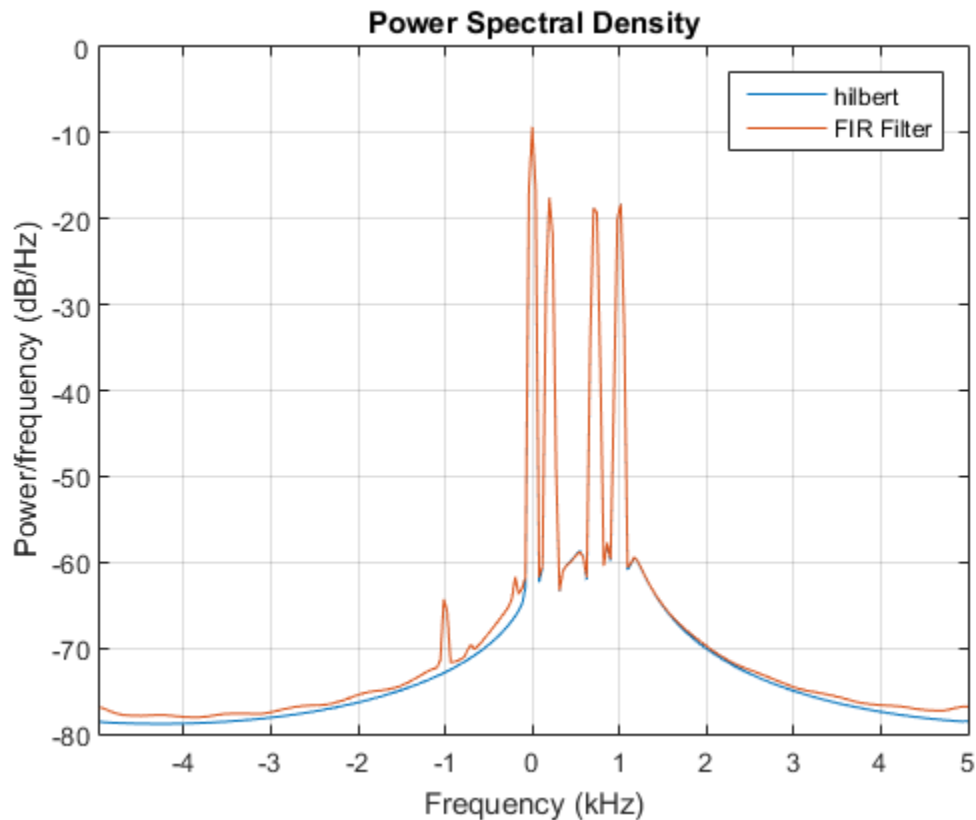
y2 = x(1:end-grd) + 1j*hb(grd+1:end);
t2 = t(1:end-grd);

plot(t2,real(y2),t2,imag(y2))
xlim([0.01 0.03])
legend('real','imaginary')
title('FIR Filter')
```



Estimate the PSD of the approximate analytic signal and compare it to the `hilbert` result.

```
pwelch([y;[y2 zeros(1,grd)]]',256,0,[],fs,'centered')  
legend('hilbert','FIR Filter')
```



## More About

### Algorithms

The analytic signal for a sequence  $x$  has a *one-sided Fourier transform*. That is, the transform vanishes for negative frequencies. 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  $i = 2, 3, \dots, (n/2)$
  - 0 for  $i = (n/2)+2, \dots, n$
- 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.

This algorithm was first introduced in [2]. The technique assumes that the input signal,  $x$ , is a finite block of data. This assumption allows the function to remove the spectral redundancy in  $x$  exactly. Methods based on FIR filtering can only approximate the analytic signal, but they have the advantage that they operate continuously on the data. See “Single-Sideband Amplitude Modulation” for another example of a Hilbert transform computed with an FIR filter.

## References

- [1] Claerbout, Jon F. *Fundamentals of Geophysical Data Processing with Applications to Petroleum Prospecting*. Oxford, UK: Blackwell, 1985, pp. 59–62.
- [2] Marple, S. L. “Computing the Discrete-Time Analytic Signal via FFT.” *IEEE Transactions on Signal Processing*. Vol. 47, 1999, pp. 2600–2603.
- [3] Oppenheim, Alan V., Ronald W. Schaffer, and John R. Buck. *Discrete-Time Signal Processing*. 2nd Ed. Upper Saddle River, NJ: Prentice Hall, 1999.

## See Also

`fft` | `ifft` | `rceps`



# icceps

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  $\pi$  radians.

## References

[1] Oppenheim, A.V., and R.W. Schaffer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989.

## See Also

`cceps` | `hilbert` | `rceps` | `unwrap`

## idct

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`

$$x(n) = \sum_{k=1}^N w(k)y(k) \cos\left(\frac{\pi(2n-1)(k-1)}{2N}\right), \quad n = 1, 2, \dots, N$$

where

$$w(k) = \begin{cases} \frac{1}{\sqrt{N}}, & k = 1 \\ \sqrt{\frac{2}{N}}, & 2 \leq k \leq N \end{cases}$$

and  $N = \text{length}(x)$ , which is the same as  $\text{length}(y)$ . 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

## ifwht

Inverse Fast Walsh-Hadamard transform

### Syntax

```
y = ifwht(x)
y = ifwht(x,n)
y = ifwht(x,n,ordering)
```

### Description

`y = ifwht(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 transform is calculated on each column of `x`. The inverse fast Walsh-Hadamard transform 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 the 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 the 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.                              |

## More About

### Algorithms

The inverse fast Walsh-Hadamard transform 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 for details.

## References

- [1] Beauchamp, Kenneth G. *Applications of Walsh and Related Functions: With an Introduction to Sequency Theory*. London: Academic Press, 1984.
- [2] Beer, Tom. "Walsh Transforms." *American Journal of Physics*. Vol. 49, 1981, pp.466–472.

### See Also

fwht | dct | idct | fft | ifft

## impinvar

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);
```

#### Example 2

Illustrate the relationship between analog and digital impulse responses [2].

---

**Note** This example requires the `impulse` function from Control System Toolbox™ software.

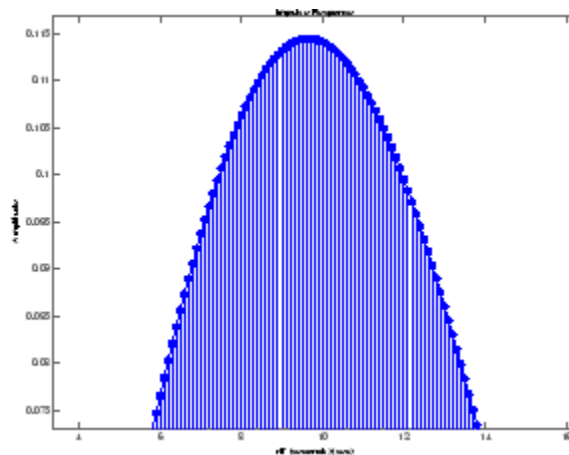
---

The steps used in this example are:

- 1 Create an analog Butterworth filter
- 2 Use `impinvar` with a sampling frequency  $F_s$  of 10 Hz to scale the coefficients by  $1/F_s$ . 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 ( $F_s$ ) to compensate for the  $1/F_s$  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.



## More About

### Algorithms

`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`



# impz

Impulse response of digital filter

## Syntax

```
[h,t] = impz(b,a)
[h,t] = impz(sos)
[h,t] = impz(d)
[h,t] = impz(...,n)
[h,t] = impz(...,n,fs)
impz(...)
```

## Description

`[h,t] = impz(b,a)` returns 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 the sample times in the column vector, `t`. `t = [0:n-1]'` and `n = length(t)` is computed automatically.

`[h,t] = impz(sos)` returns the impulse response for the second-order sections matrix, `sos`. `sos` is a  $K$ -by-6 matrix, where the number of sections,  $K$ , must be greater than or equal to 2. If the number of sections is less than 2, `impz` considers the input to be a numerator vector. Each row of `sos` corresponds to the coefficients of a second order (biquad) filter. The  $i$ th row of the `sos` matrix corresponds to `[bi(1) bi(2) bi(3) ai(1) ai(2) ai(3)]`.

`[h,t] = impz(d)` returns the impulse response of a digital filter, `d`. Use `designfilt` to generate `d` based on frequency-response specifications.

`[h,t] = impz(...,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.

`[h,t] = impz(...,n,fs)` computes `n` samples and produces a vector `t` of length `n` so that the samples are spaced `1/fs` units apart.

`impz(...)` with no output arguments plots the impulse response of the filter.

`impz` works for both real and complex input systems.

---

**Note:** If the input to `impz` is single precision, the impulse response is calculated using single-precision arithmetic. The output, `h`, is single precision.

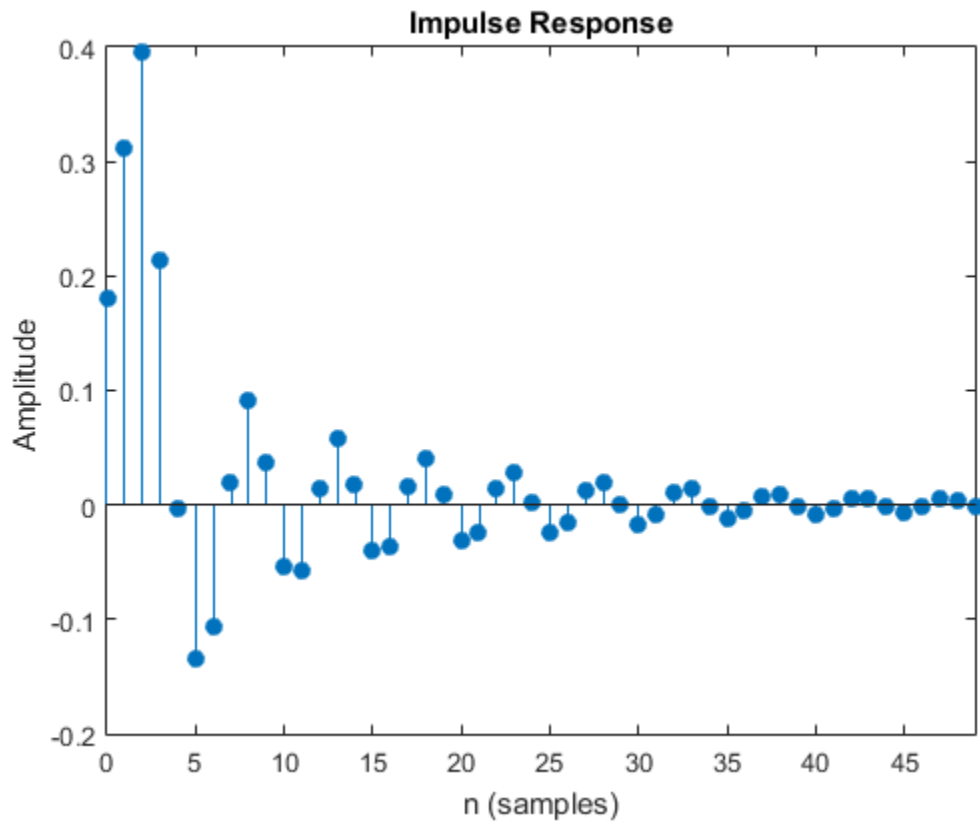
---

## Examples

### Impulse Response of an Elliptic Lowpass Filter

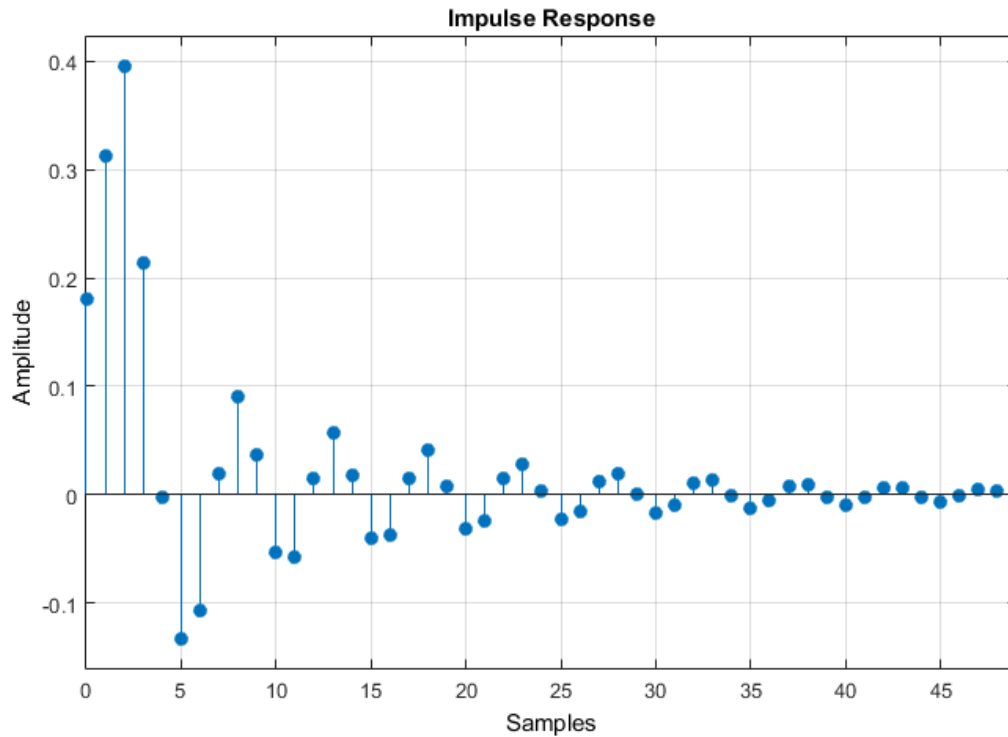
Design a fourth-order lowpass elliptic filter with normalized passband frequency 0.4. Specify a passband ripple of 0.5 dB and a stopband attenuation of 20 dB. Plot the first 50 samples of the impulse response.

```
[b,a] = ellip(4,0.5,20,0.4);  
impz(b,a,50)
```



Design the same filter using `designfilt`. Plot the first 50 samples of its impulse response.

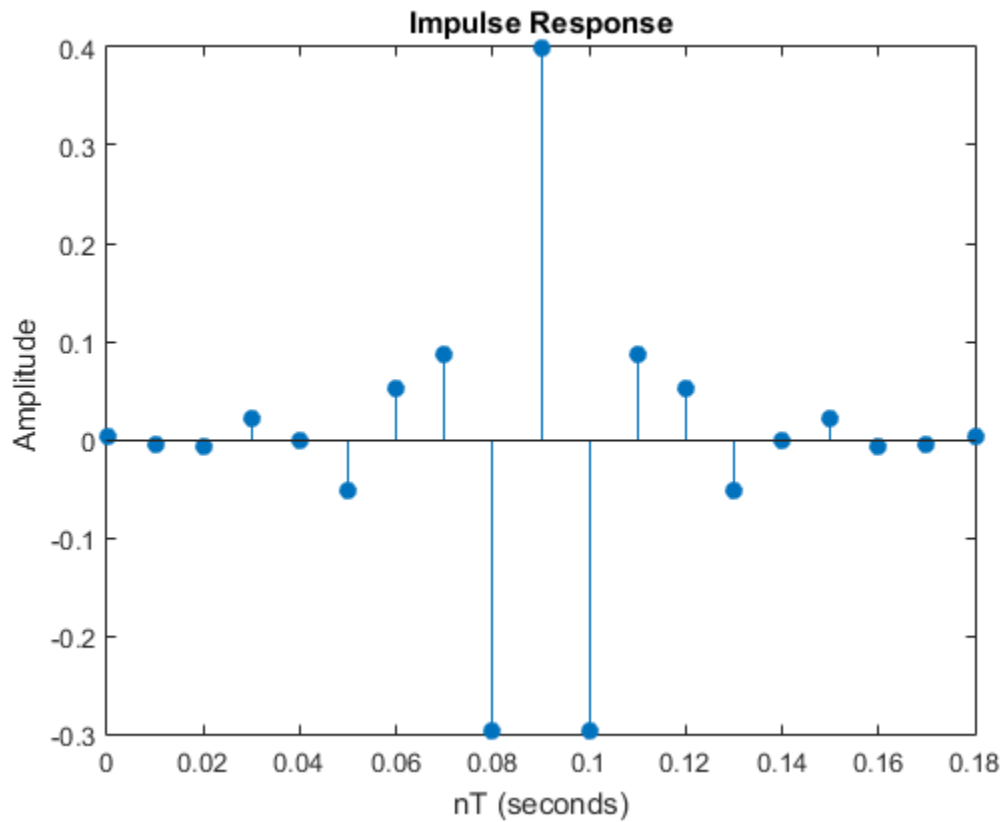
```
d = designfilt('lowpassiir','DesignMethod','ellip','FilterOrder',4, ...  
              'PassbandFrequency',0.4, ...  
              'PassbandRipple',0.5,'StopbandAttenuation',20);  
impz(d,50)
```



### Impulse Response of a Highpass FIR Filter

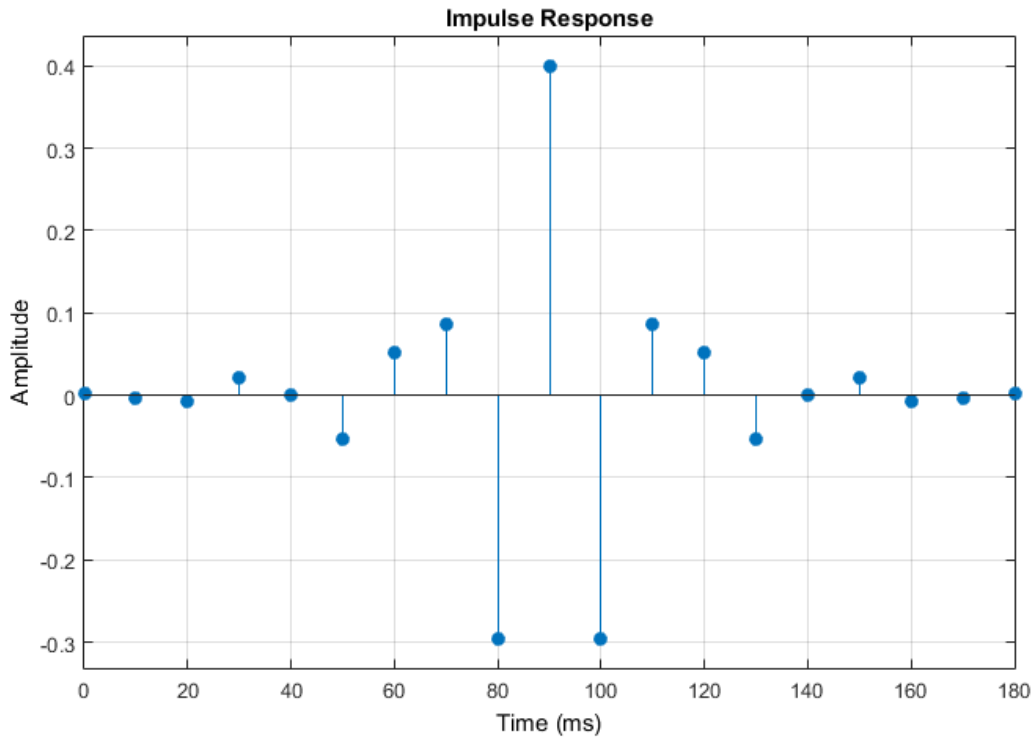
Design an FIR highpass filter of order 18 using a Kaiser window with  $\beta = 4$ . Specify a sampling rate of 100 Hz and a cutoff frequency of 30 Hz. Display the impulse response of the filter.

```
b = fir1(18,30/(100/2),'high',kaiser(19,4));  
impz(b,1,[],100)
```



Design the same filter using `designfilt` and plot its impulse response.

```
d = designfilt('highpassfir', 'FilterOrder', 18, 'SampleRate', 100, ...  
              'CutoffFrequency', 30, 'Window', {'kaiser', 4});  
impz(d, [], 100)
```



## More About

### Algorithms

`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  $5 \times 10^{-5}$  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 pole of largest nonunit amplitude is  $5 \times 10^{-5}$  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**

`designfilt` | `digitalFilter` | `impulse` | `stem`

# impzlength

Impulse response length

## Syntax

```
len = impzlength(b,a)
len = impzlength(sos)
len = impzlength(d)
len = impzlength(hs)
len = impzlength(hd)
len = impzlength( ____,tol)
```

## Description

`len = impzlength(b,a)` returns the impulse response length for the causal discrete-time filter with the rational system function specified by the numerator, `b`, and denominator, `a`, polynomials in  $z^{-1}$ . For stable IIR filters, `len` is the effective impulse response sequence length. Terms in the IIR filter's impulse response after the `len`-th term are essentially zero.

`len = impzlength(sos)` returns the effective impulse response length for the IIR filter specified by the second order sections matrix, `sos`. `sos` is a  $K$ -by-6 matrix, where the number of sections,  $K$ , must be greater than or equal to 2. If the number of sections is less than 2, `impzlength` considers the input to be the numerator vector, `b`. Each row of `sos` corresponds to the coefficients of a second order (biquad) filter. The  $i$ th row of the `sos` matrix corresponds to `[bi(1) bi(2) bi(3) ai(1) ai(2) ai(3)]`.

`len = impzlength(d)` returns the impulse response length for the digital filter, `d`. Use `designfilt` to generate `d` based on frequency-response specifications.

`len = impzlength(hs)` returns the impulse response length for the filter System object, `hs`. You must have the DSP System Toolbox software to use `impzlength` with a filter System object.

`len = impzlength(hd)` returns the impulse response length for the `dfilt` or `mfilt` filter object, `hd`. You must have the DSP System Toolbox software to use `impzlength` with an `mfilt` object. You can also input an array of filter objects. If `hd` is an array of



filter objects, each column of `len` is the impulse response length of the corresponding filter object.

`len = impzlength( ____, tol)` specifies a tolerance for estimating the effective length of an IIR filter's impulse response. By default, `tol` is  $5e-5$ . Increasing the value of `tol` estimates a shorter effective length for an IIR filter's impulse response. Decreasing the value of `tol` produces a longer effective length for an IIR filter's impulse response.

## Examples

### IIR Filter Effective Impulse Response Length — — Coefficients

Create a lowpass allpole IIR filter with a pole at 0.9. Calculate the effective impulse response length, obtain the impulse response, and plot the result.

```
b = 1;
a = [1 -0.9];
len = impzlength(b,a)
[h,t] = impz(b,a);
stem(t,h)
h(len)
```

The value of the impulse response at the estimate length has decayed to approximately  $10^{-6}$ .

### IIR Filter Effective Impulse Response Length — — Second Order Sections

Design a 4th-order lowpass elliptic filter with a cutoff frequency of  $0.4\pi$  rad/sample. Specify 1 dB of passband ripple and 60 dB of stopband attenuation. Design the filter in pole-zero-gain form and obtain the second order section matrix using `zp2sos`. Determine the effective impulse response sequence length from the second order sections matrix.

```
[z,p,k] = ellip(4,1,60,.4);
[sos,g] = zp2sos(z,p,k);
len = impzlength(sos)
```

### IIR Filter Effective Impulse Response Length --- --- Digital Filter

Use `designfilt` to design a 4th-order lowpass elliptic filter with normalized passband frequency  $0.4\pi$  rad/sample. Specify 1 dB of passband ripple and 60 dB of stopband attenuation. Determine the effective impulse response sequence length and visualize it.

```
d = designfilt('lowpassiir','FilterOrder',4,'PassbandFrequency',0.4, ...
```

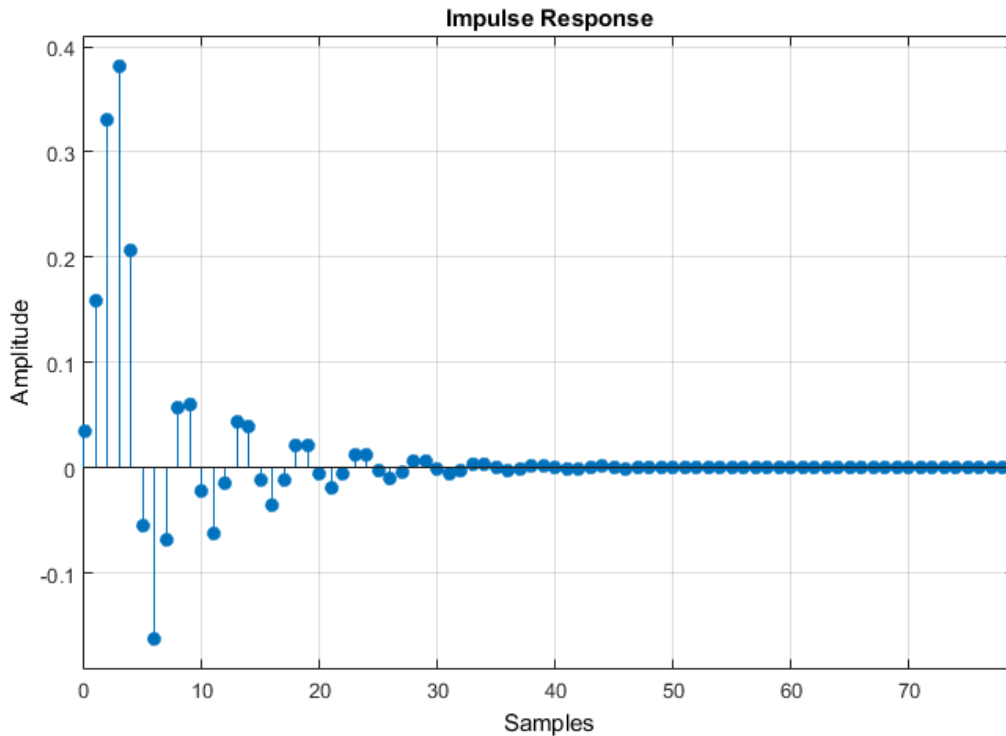
```

        'PassbandRipple',1,'StopbandAttenuation',60, ...
        'DesignMethod','ellip');
len = impzlength(d)
impz(d)

len =

    80

```



### Impulse Response Length of Filter System object

This example requires DSP System Toolbox software.

Design a 4th-order lowpass elliptic filter with a cutoff frequency of  $0.4\pi$  rad/sample. Specify 1 dB of passband ripple and 60 dB of stopband attenuation. Design the filter in

pole-zero-gain form and obtain the second order section matrix using `zp2sos`. Create a biquad filter System object and input the System object to `impzlength`.

```
[z,p,k] = ellip(4,1,60,.4);
[sos,g] = zp2sos(z,p,k);
hBqdFilt = dsp.BiquadFilter('Structure','Direct form I',...
                           'SOSMatrix', sos,...
                           'ScaleValues',g);

len = impzlength(hBqdFilt)
```

### Impulse Response Length — — Filter Objects

Design IIR Butterworth and FIR equiripple filters for data sampled at 1 kHz. The passband frequency is 100 Hz and the stopband frequency is 150 Hz. The passband ripple is 0.5 dB and there is 60 dB of stopband attenuation. Obtain `dfilt` objects for the filters and compare the filter impulse response sequence lengths.

```
d = fdesign.lowpass('Fp,Fst,Ap,Ast',100,150,0.5,60,1000);
Hd1 = design(d,'butter');
Hd2 = design(d,'equiripple');
len = impzlength([Hd1 Hd2])
```

## Input Arguments

### **b** — Numerator coefficients

vector | scalar

Numerator coefficients, specified as a scalar (allpole filter) or a vector.

Example: `b = fir1(20,0.25)`

Data Types: `single` | `double`

Complex Number Support: Yes

### **a** — Denominator coefficients

vector | scalar

Denominator coefficients, specified as a scalar (FIR filter) or vector.

Data Types: `single` | `double`

Complex Number Support: Yes

### **sos** — Matrix of second order sections

matrix

Matrix of second order sections, specified as a  $K$ -by-6 matrix. The system function of the  $K$ -th biquad filter has the rational Z-transform

$$H_k(z) = \frac{B_k(1) + B_k(2)z^{-1} + B_k(3)z^{-2}}{A_k(1) + A_k(2)z^{-1} + A_k(3)z^{-2}}.$$

The coefficients in the  $K$ th row of the matrix, `sos`, are ordered as follows.

$$[B_k(1) \ B_k(2) \ B_k(3) \ A_k(1) \ A_k(2) \ A_k(3)]$$

The frequency response of the filter is the system function evaluated on the unit circle with

$$z = e^{j2\pi f}.$$

#### **d** — Digital filter

`digitalFilter` object

Digital filter, specified as a `digitalFilter` object. Use `designfilt` to generate a digital filter based on frequency-response specifications.

Example: `d = designfilt('lowpassiir', 'FilterOrder', 3, 'HalfPowerFrequency', 0.5)` specifies a third-order Butterworth filter with normalized 3-dB frequency  $0.5\pi$  rad/sample.

#### **hs** — Filter System object

`filter System` object

Filter System object, specified as one of the following:

- `dsp.FIRFilter`
- `dsp.BiquadFilter`
- `dsp.FIRInterpolator`
- `dsp.CICInterpolator`
- `dsp.FIRDecimator`
- `dsp.CICDecimator`
- `dsp.FIRRateConverter`

Using `impzlength` with a filter System object requires the DSP System Toolbox software.

**hd** — Filter object

`dfilt` object | `mfilt` object

Filter object, specified as a `dfilt` or `mfilt` object. You must have the DSP System Toolbox software to input an `mfilt` object.

**tol** — Tolerance for IIR filter effective impulse response length

$5e-5$  (default) | positive scalar

Tolerance for IIR filter effective impulse response length, specified as a positive number. The tolerance determines the term in the absolutely summable sequence after which subsequent terms are considered to be 0. The default tolerance is  $5e-5$ . Increasing the tolerance returns a shorter effective impulse response sequence length. Decreasing the tolerance returns a longer effective impulse response sequence length.

## Output Arguments

**len** — Length of impulse response

positive integer

Length of the impulse response, specified as a positive integer. For stable IIR filters with absolutely summable impulse responses, `impzlength` returns an effective length for the impulse response beyond which the coefficients are essentially zero. You can control this cutoff point by specifying the optional `tol` input argument.

**See Also**

`digitalFilter` | `designfilt` | `impz` | `zp2sos`

## info

Information about digital filter

## Syntax

```
s = info(d)
```

## Description

`s = info(d)` returns a string matrix with information about the digital filter, `d`.

## Examples

### Information on a Lowpass FIR Filter

Design a lowpass FIR filter with normalized passband frequency  $0.4\pi$  rad/sample and normalized stopband frequency  $0.45\pi$  rad/sample. Obtain information about the filter just designed.

```
d = designfilt('lowpassfir','PassbandFrequency',0.4,'StopbandFrequency',0.45);  
s = info(d)
```

```
s =
```

```
FIR Digital Filter (real)
```

```
-----
```

```
Filter Length   : 81  
Stable          : Yes  
Linear Phase    : Yes (Type 1)
```

```
Design Method Information  
Design Algorithm : Equiripple
```

```
Design Specifications  
Sample Rate     : 2 (normalized)  
Response       : Lowpass
```

```
Passband Edge : 0.4
Stopband Edge : 0.45
Passband Ripple : 1 dB
Stopband Atten. : 60 dB
```

## Input Arguments

### **d** — Digital filter

digitalFilter object

Digital filter, specified as a `digitalFilter` object. Use `designfilt` to generate a digital filter based on frequency-response specifications.

Example: `d =`

```
designfilt('lowpassiir','FilterOrder',3,'HalfPowerFrequency',0.5)
```

specifies a third-order Butterworth filter with normalized 3-dB frequency  $0.5\pi$  rad/sample.

## Output Arguments

### **s** — Information table

string matrix

Information table, returned as a string matrix.

## See Also

`designfilt` | `digitalFilter`

## interp

Interpolation — increase sampling rate by integer factor

### Syntax

```
y = interp(x,r)
y = interp(x,r,n,alpha)
[y,b] = interp(x,r,n,alpha)
```

### Description

Interpolation increases the original sampling rate of a sequence to a higher rate. It is the opposite of decimation. `interp` inserts 0s into the original signal and then applies a lowpass interpolating filter to the expanded sequence.

`y = interp(x,r)` increases the sampling rate of `x`, the input signal, by a factor of `r`. The interpolated vector, `y`, is `r` times as long as the original input, `x`.

`y = interp(x,r,n,alpha)` specifies two additional values. `n` is half the number of original sample values used to interpolate the expanded signal. Its default value is 4. It should ideally be less than or equal to 10. `alpha` is the normalized cutoff frequency of the input signal, specified as a fraction of the Nyquist frequency. It defaults to 0.5. The lowpass interpolation filter has length  $2*n*r + 1$ .

`[y,b] = interp(x,r,n,alpha)` also returns a vector, `b`, with the filter coefficients used for the interpolation.

### Input Arguments

#### **x** — Input signal

vector

Input signal, specified as a vector.

Data Types: `double`

#### **r** — Interpolation factor

positive integer scalar



Interpolation factor, specified as a positive integer scalar.

Data Types: `double`

**n** — Half-number of input samples used for interpolation

4 (default) | positive integer scalar

Half-number of input samples used for interpolation, specified as a positive integer scalar. `n` should never be greater than 10.

Data Types: `double`

**a1pha** — Normalized cutoff frequency

0.5 (default) | positive scalar

Normalized cutoff frequency of the input signal, specified as a positive real scalar not greater than 1. A value of 1 means that the signal occupies the full Nyquist interval.

Data Types: `double`

## Output Arguments

**y** — Interpolated signal

vector

Interpolated signal, returned as a vector.

Data Types: `double`

**b** — Lowpass interpolation filter coefficients

column vector

Lowpass interpolation filter coefficients, returned as a column vector.

Data Types: `double`

## Examples

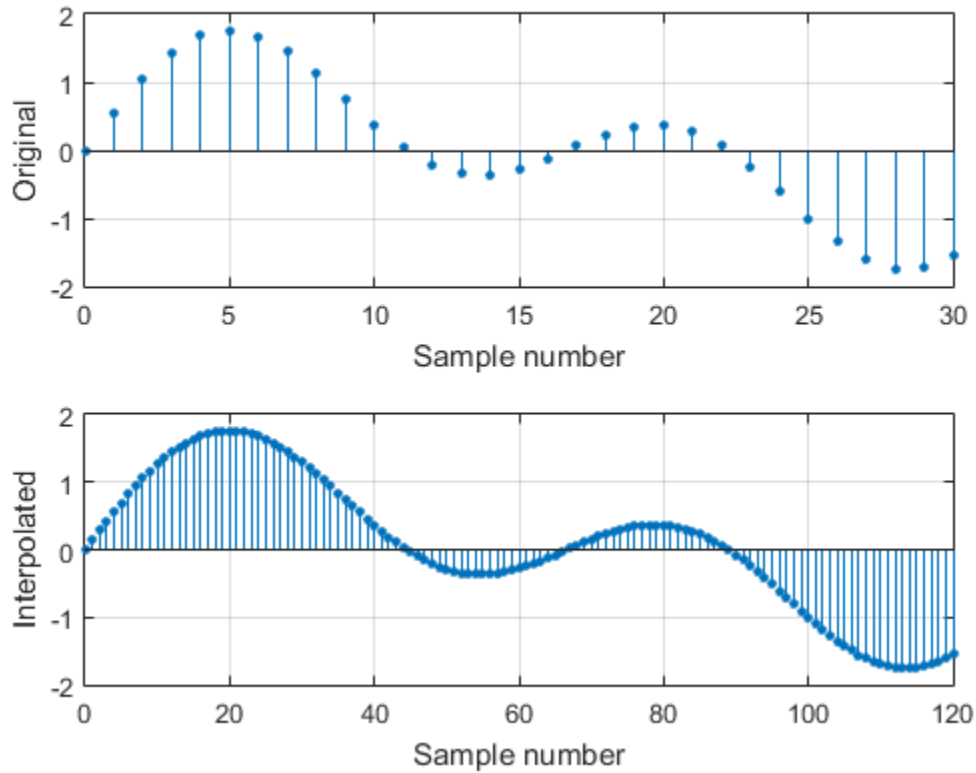
### Interpolate a Signal

Create a sinusoidal signal sampled at 1 kHz. Interpolate it by a factor of four.

```
t = 0:0.001:1;  
x = sin(2*pi*30*t) + sin(2*pi*60*t);  
y = interp(x,4);
```

Plot the original and interpolated signals.

```
subplot 211  
stem(0:30,x(1:31),'filled','markersize',3)  
grid on  
xlabel 'Sample number',ylabel Original  
subplot 212  
stem(0:120,y(1:121),'filled','markersize',3)  
grid on  
xlabel 'Sample number',ylabel Interpolated
```



## More About

### Algorithms

`interp` uses the lowpass interpolation algorithm 8.1 described in [1].

- 1 It expands the input vector to the correct length by inserting 0s between the original data values.
- 2 It designs a special symmetric FIR filter that allows the original data to pass through unchanged and interpolates to minimize the mean-square error between the interpolated points and their ideal values. The filter used by `interp` is the same as the filter returned by `intfilt`.

- 3** It applies the filter to the expanded input vector to produce the output.

## References

- [1] Digital Signal Processing Committee of the IEEE Acoustics, Speech, and Signal Processing Society, eds. *Programs for Digital Signal Processing*. New York: IEEE Press, 1979, chap.8.
- [2] Oetken, G., Thomas W. Parks, and H. W. Schüssler. “New results in the design of digital interpolators.” *IEEE Transactions on Acoustics, Speech, and Signal Processing*. Vol. ASSP-23, 1975, pp.301–309.

## See Also

decimate | downsample | interp1 | intfilt | resample | spline | upfirdn |  
upsample

# intfilt

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  $2 \cdot p$  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  $2 \cdot l \cdot p - 1$ .

$\alpha$  is inversely proportional to the transition bandwidth of the filter and it also affects the bandwidth 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  $(n+1) \cdot l$  for  $n$  even, and length  $(n+1) \cdot l - 1$  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:

```
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
```

## More About

### Algorithms

The bandlimited method uses `firls` to design an interpolation FIR filter. The polynomial method uses Lagrange's polynomial interpolation formula on equally spaced samples to construct the appropriate filter.

**See Also**

decimate | interp | downsample | resample | upsample

## invfreqs

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

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

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  $-\pi$  and  $\pi$ .

## 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
```

`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

You have two vectors, `mag` and `phase`, that contain magnitude and phase data gathered in a laboratory, and a third vector `w` of frequencies. Convert the data into a continuous-time transfer function using `invfreqs`.

```
[b,a] = invfreqs(mag.*exp(j*phase),w,2,3);
```

## More About

### Tips

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.

### Algorithms

By default, `invfreqs` uses an equation error method to identify the best model from the data. This finds `b` and `a` in

$$\min_{b,a} \sum_{k=1}^n wt(k) |h(k)A(w(k)) - B(w(k))|^2$$

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

## invfreqz

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

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

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  $\pi$ , 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  $-\pi$  and  $\pi$ .

## Examples

### Stable Approximate Transfer Function

Convert a simple transfer function to frequency response data and then back to the original filter coefficients. Sketch the zeros and poles of the function.

```
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)
```

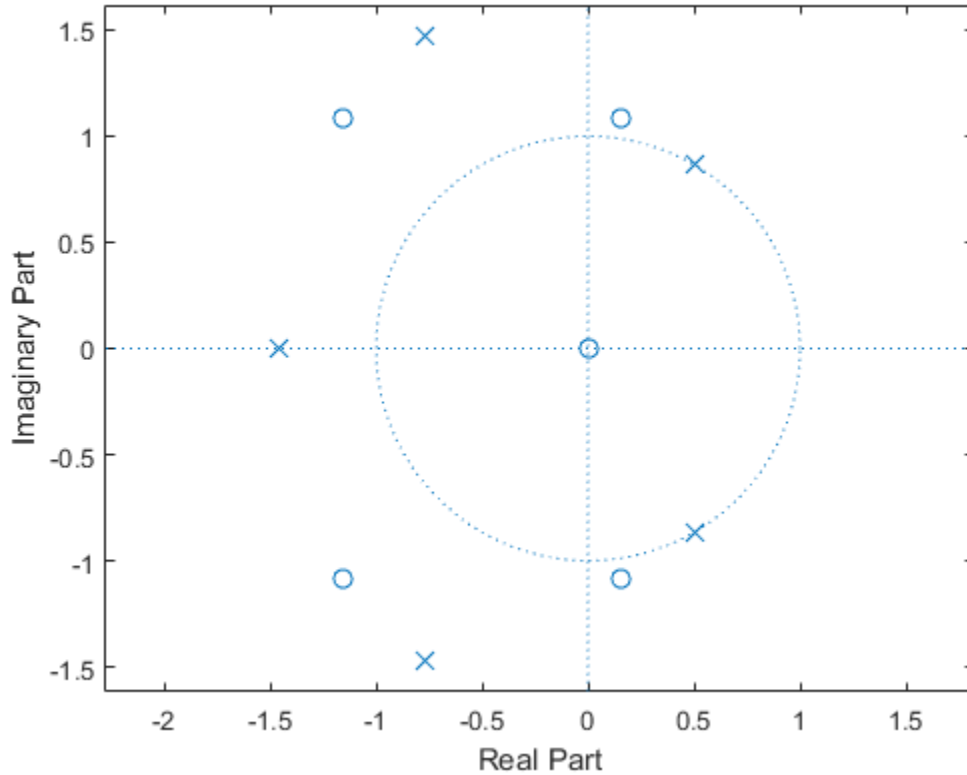
```
zplane(bb,aa)
```

```
bb =
```

```
    1.0000    2.0000    3.0000    2.0000    3.0000
```

```
aa =
```

```
    1.0000    2.0000    3.0000    2.0000    1.0000    4.0000
```



`bb` and `aa` are equivalent to `b` and `a`, respectively. However, the system is unstable because it has poles outside the unit circle. Use `invfreqz`'s iterative algorithm to find a stable approximation to the system. Verify that the poles are within the unit circle.

```
[bbb,aaa] = invfreqz(h,w,4,5,[],30)
```

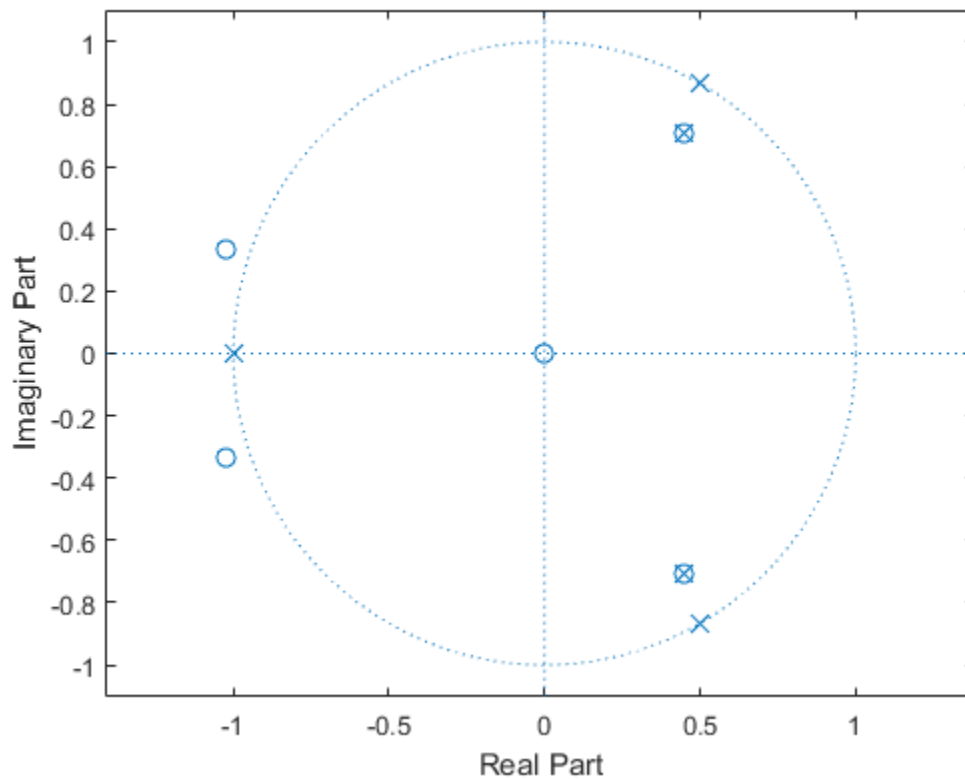
```
zplane(bbb,aaa)
```

```
bbb =
```

```
0.2427    0.2788    0.0069    0.0971    0.1980
```

```
aaa =
```

```
1.0000 -0.8944 0.6954 0.9997 -0.8933 0.6949
```



## More About

### Algorithms

By default, `invfreqz` uses an equation error method to identify the best model from the data. This finds `b` and `a` in

$$\min_{b,a} \sum_{k=1}^n wt(k) |h(k)A(w(k)) - B(w(k))|^2$$

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  $\mathbf{a}$  and  $\mathbf{b}$ , respectively, at the frequency  $\omega(k)$ , and  $n$  is the number of frequency points (the length of  $\mathbf{h}$  and  $\mathbf{w}$ ). This algorithm is 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.

$$\min_{b,a} \sum_{k=1}^n wt(k) \left| h(k) - \frac{B(w(k))}{A(w(k))} \right|^2$$

## References

- [1] Levi, E. C. “Complex-Curve Fitting.” *IRE Transactions 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 | prony



## isallpass

Determine whether filter is allpass

### Syntax

```
flag = isallpass(b,a)
flag = isallpass(sos)
flag = isallpass(d)
flag = isallpass(...,tol)
flag = isallpass(hs,...)
flag = isallpass(hs,'Arithmetic',arithtype)
flag = isallpass(hd)
```

### Description

`flag = isallpass(b,a)` returns a logical output, `flag`, equal to `true` if the filter specified by numerator coefficients, `b`, and denominator coefficients, `a`, is an allpass filter. If the filter is not an allpass filter, `flag` is equal to `false`.

`flag = isallpass(sos)` returns `true` if the filter specified by second order sections matrix, `sos`, is an allpass filter. `sos` is a  $K$ -by-6 matrix, where the number of sections,  $K$ , must be greater than or equal to 2. Each row of `sos` corresponds to the coefficients of a second order (biquad) filter. The  $i$ th row of the `sos` matrix corresponds to `[bi(1) bi(2) bi(3) ai(1) ai(2) ai(3)]`.

`flag = isallpass(d)` returns `true` if the digital filter, `d`, is an allpass filter. Use `designfilt` to generate `d` based on frequency-response specifications.

`flag = isallpass(...,tol)` uses the tolerance, `tol`, to determine when two numbers are close enough to be considered equal. If not specified, `tol`, defaults to `eps^(2/3)`. Specifying a tolerance may be most helpful in fixed-point allpass filters.

`flag = isallpass(hs,...)` returns `true` if the filter System object, `hs`, is an allpass filter. You must have the DSP System Toolbox software to use this syntax.

`flag = isallpass(hs,'Arithmetic',arithtype)` analyzes the filter System object `hs` based on the specified `arithtype`. `arithtype` can be `'double'`, `'single'`, or

'fixed'. When you specify 'double' or 'single', the function performs double- or single-precision analysis. When you specify 'fixed', the arithmetic changes depending on the setting of the `CoefficientDataType` property and whether the System object is locked or unlocked. You must have the DSP System Toolbox software to use this syntax.

**Details for Fixed-Point Arithmetic**

| System Object State | Coefficient Data Type | Rule   |
|---------------------|-----------------------|--|
| Unlocked            | 'Same as input'       | The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.  |
| Unlocked            | 'Custom'              | The function performs fixed-point analysis based on the setting of the <code>CustomCoefficientsDataType</code> property.   |
| Locked              | 'Same as input'       | When the input data type is 'double' or 'fixed', the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption. |
| Locked              | 'Custom'              | The function performs fixed-point analysis based on the setting of the <code>CustomCoefficientsDataType</code> property.   |

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

`flag = isallpass(hd)` returns `true` if the filter object, `hd`, is an allpass filter.

## Examples

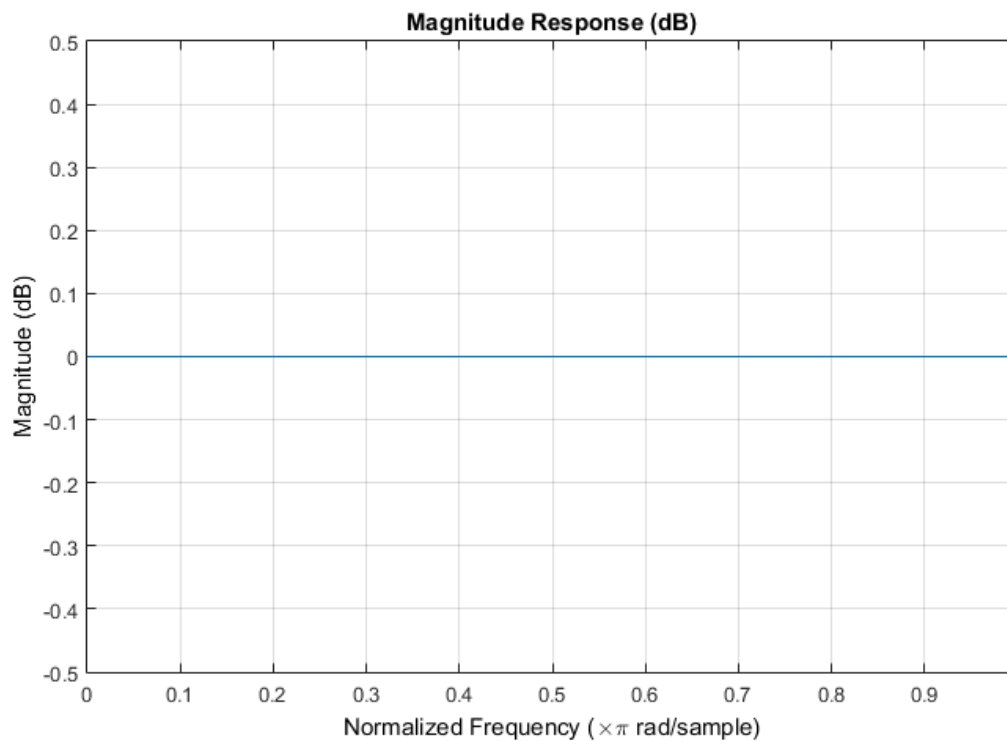
### Allpass Filters

Create an allpass filter and verify that the frequency response is allpass.

```
b = [1/3 1/4 1/5 1];  
a = fliplr(b);  
flag = isallpass(b,a)  
fvtool(b,a)
```

```
flag =
```

```
1
```

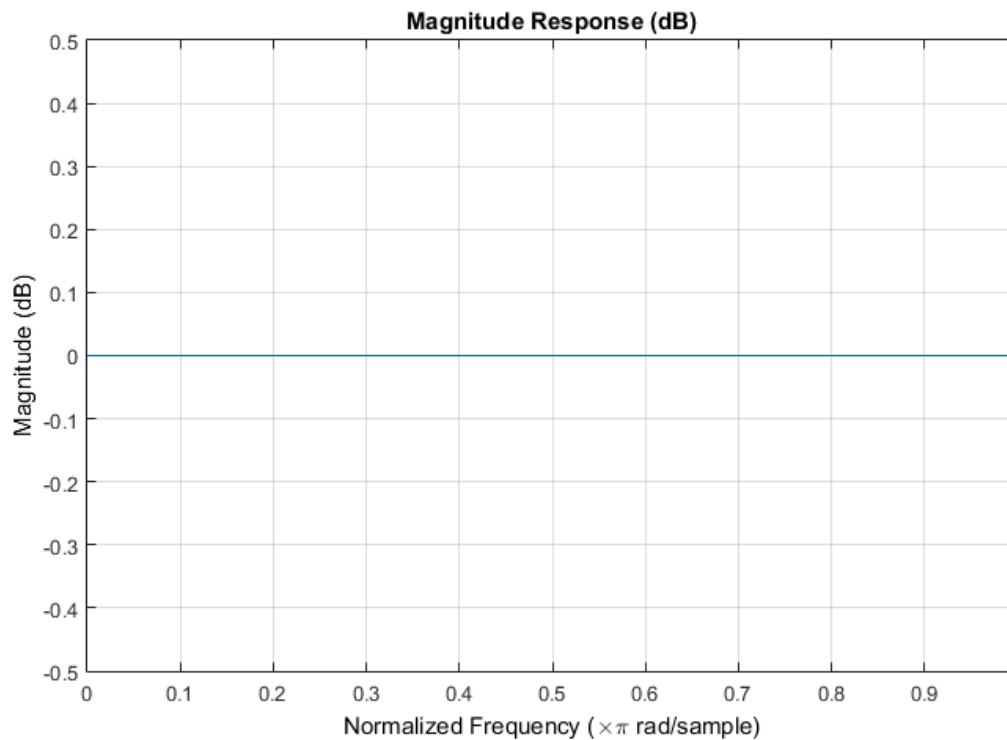


Create a lattice allpass filter and verify that the filter is allpass.

```
k = [1/2 1/3 1/4 1/5];  
[b,a] = latc2tf(k,'allpass');  
flag_isallpass = isallpass(b,a)  
fvtool(b,a)
```

```
flag_isallpass =
```

```
1
```



### See Also

`designfilt` | `digitalFilter` | `islinphase` | `ismaxphase` | `isminphase` | `isstable`

# isdouble

Determine if digital filter coefficients are double precision

## Syntax

```
flag = isdouble(d)
```

## Description

`flag = isdouble(d)` returns `true` if the coefficients of a digital filter, `d`, are double precision.

## Examples

### Double- and Single-Precision Filter

Use `designfilt` to design a sixth-order highpass IIR filter. Specify a normalized passband frequency of  $0.6\pi$  rad/sample. Convert it to a single-precision filter. Identify the precision in each case.

```
fd = designfilt('highpassiir', 'FilterOrder',6, 'PassbandFrequency',0.6);  
isd = isdouble(fd)  
fs = single(fd);  
iss = isdouble(fs)
```

```
isd =
```

```
1
```

```
iss =
```

```
0
```

## Input Arguments

### **d** — Digital filter

`digitalFilter` object

Digital filter, specified as a `digitalFilter` object. Use `designfilt` to generate `d`. If you want a single-precision filter, apply `single` to the output of `designfilt`.

Example: `d =`

```
designfilt('lowpassiir','FilterOrder',3,'HalfPowerFrequency',0.5)
```

specifies a third-order Butterworth filter with normalized 3-dB frequency  $0.5\pi$  rad/sample.

## Output Arguments

### **flag** — Type identification

logical scalar

Type identification, returned as a logical scalar.

### See Also

`designfilt` | `digitalFilter` | `double` | `issingle` | `single`

## isfir

Determine if digital filter has finite impulse response

### Syntax

```
flag = isfir(d)
```

### Description

`flag = isfir(d)` returns `true` if a digital filter, `d`, has a finite impulse response.

## Examples

### FIR and IIR Digital Filters

Use `designfilt` to design FIR and IIR versions of a highpass filter. Specify a normalized stopband frequency of 0.3 and a normalized passband frequency of 0.6. Verify that each filter is of the correct class. Display the frequency responses of the filters.

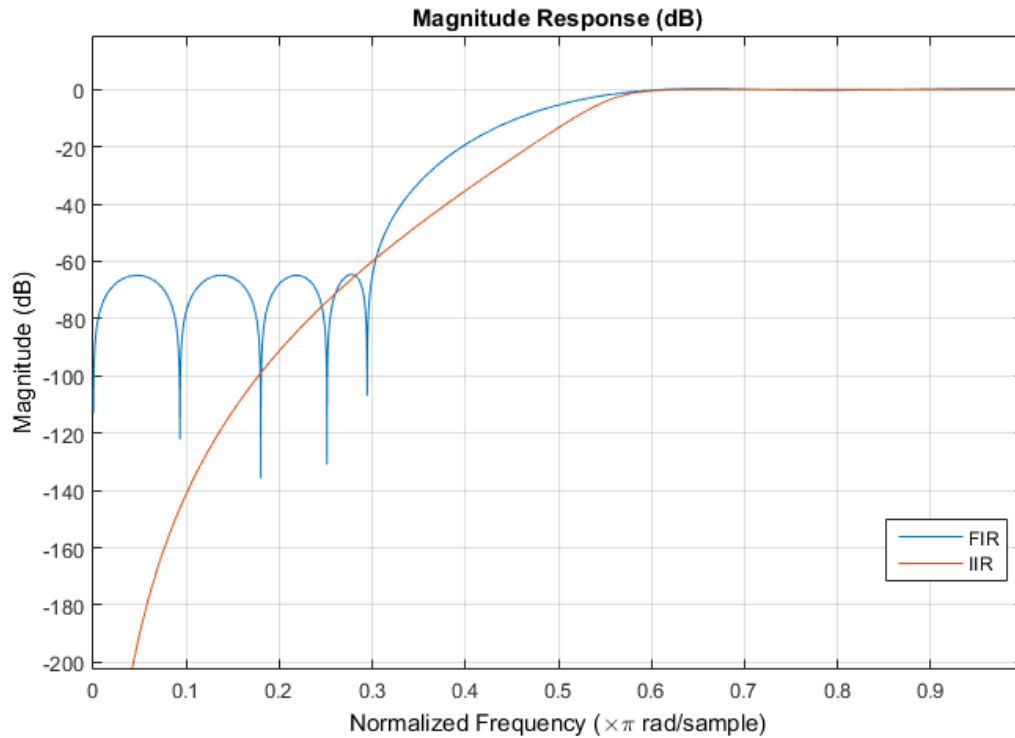
```
fir = designfilt('highpassfir','StopbandFrequency',0.3,'PassbandFrequency',0.6);  
iir = designfilt('highpassiir','StopbandFrequency',0.3,'PassbandFrequency',0.6);  
isfirFIR = isfir(fir)  
isiirFIR = isfir(iir)  
fvt = fvtool(fir,iir);  
legend(fvt,'FIR','IIR')
```

```
isfirFIR =
```

```
1
```

```
isiirFIR =
```

```
0
```



## Input Arguments

### **d** — Digital filter

digitalFilter object

Digital filter, specified as a `digitalFilter` object. Use `designfilt` to generate a digital filter based on frequency-response specifications.

Example: `d = designfilt('lowpassiir','FilterOrder',3,'HalfPowerFrequency',0.5)` specifies a third-order Butterworth filter with normalized 3-dB frequency  $0.5\pi$  rad/sample.



## Output Arguments

### **flag** — Filter class identification

logical scalar

Filter class identification, returned as a logical scalar.

### **See Also**

`designfilt` | `digitalFilter` | `firtype` | `isdouble` | `issingle`

## islinphase

Determine whether filter has linear phase

### Syntax

```
flag = islinphase(b,a)
flag = islinphase(sos)
flag = islinphase(d)
flag = islinphase(...,tol)
flag = islinphase(hs,...)
flag = islinphase(hs,'Arithmetic',arithtype)
flag = islinphase(h)
```

### Description

`flag = islinphase(b,a)` returns a logical output, `flag`, equal to `true` if the filter coefficients in `b` and `a` define a linear phase filter. `flag` is equal to `false` if the filter does not have linear phase.

`flag = islinphase(sos)` returns `true` if the filter specified by second order sections matrix, `sos`, has linear phase. `sos` is a  $K$ -by-6 matrix, where the number of sections,  $K$ , must be greater than or equal to 2. Each row of `sos` corresponds to the coefficients of a second order (biquad) filter. The  $i$ th row of the `sos` matrix corresponds to `[bi(1) bi(2) bi(3) ai(1) ai(2) ai(3)]`.

`flag = islinphase(d)` returns `true` if the digital filter, `d`, has linear phase. Use `designfilt` to generate `d` based on frequency-response specifications.

`flag = islinphase(...,tol)` uses the tolerance, `tol`, to determine when two numbers are close enough to be considered equal. If not specified, `tol`, defaults to `eps^(2/3)`.

`flag = islinphase(hs,...)` determines whether the filter System object, `hs`, has linear phase. You must have the DSP System Toolbox to use `islinphase` with a System object.

`flag = islinphase(hs,'Arithmetic',arithtype)` analyzes the filter System object `hs` based on the specified `arithtype`. `arithtype` can be one of `'double'`,

'single', or 'fixed'. When you specify 'double' or 'single', the function performs double- or single-precision analysis. When you specify 'fixed', the arithmetic changes depending on the setting of the `CoefficientDataType` property and whether the System object is locked or unlocked. You must have the DSP System Toolbox to use `islinphase` with a System object.

### Details for Fixed-Point Arithmetic

| System Object State | Coefficient Data Type | Rule   |
|---------------------|-----------------------|--|
| Unlocked            | 'Same as input'       | The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.  |
| Unlocked            | 'Custom'              | The function performs fixed-point analysis based on the setting of the <code>CustomCoefficientsDataType</code> property.   |
| Locked              | 'Same as input'       | When the input data type is 'double' or 'fixed', the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption. |
| Locked              | 'Custom'              | The function performs fixed-point analysis based on the setting of the <code>CustomCoefficientsDataType</code> property.   |

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

`flag = islinphase(h)` determines if the filter object `h` has linear phase. `islinphase` accepts an `adapfilt`, `dfilt`, or `mfilt` object. To create an `adapfilt` or `mfilt` object, you must have the DSP System Toolbox software.

## Examples

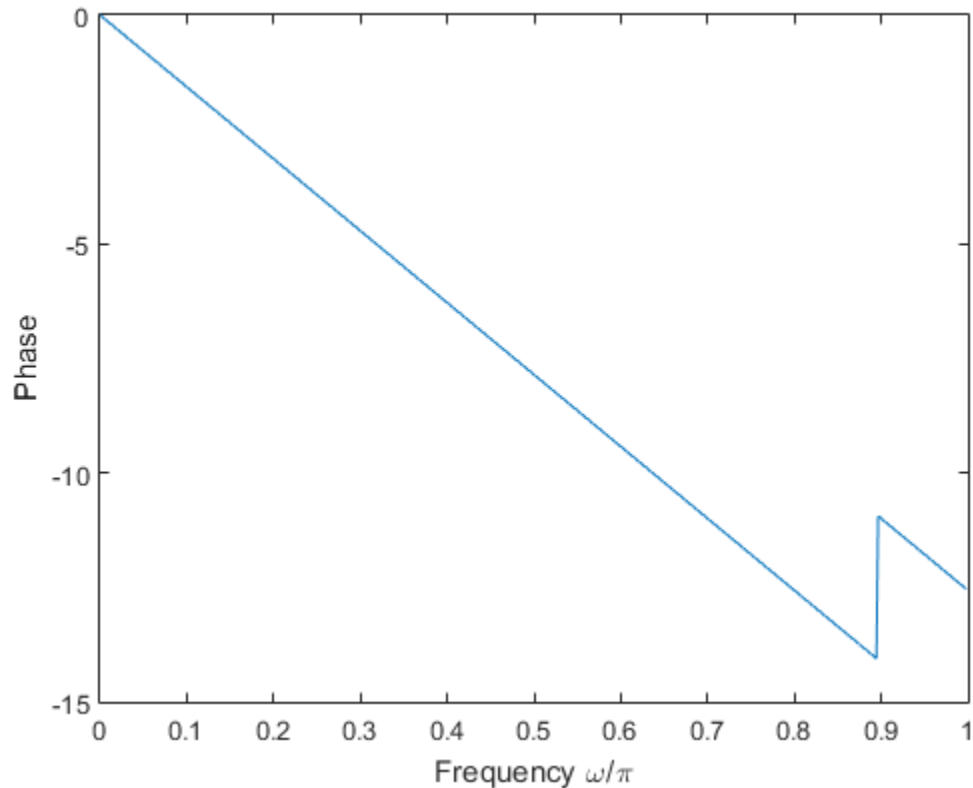
### Linear and Nonlinear Phase

Use the window method to design a tenth-order lowpass FIR filter with normalized cutoff frequency 0.55. Verify that the filter has linear phase.

```
d = designfilt('lowpassfir','DesignMethod','window', ...
              'FilterOrder',10,'CutoffFrequency',0.55);
flag = islinphase(d)
[phs,w] = phasez(d);
plot(w/pi,phs)
xlabel('Frequency \omega/\pi')
ylabel('Phase')
```

```
flag =
```

```
1
```



IIR filters in general do not have linear phase. Verify the statement by constructing eighth-order Butterworth, Chebyshev, and elliptic filters with similar specifications.

```
ord = 8;  
Wcut = 0.35;  
atten = 20;  
ripp1 = 1;
```

```
[zb,pb,kb] = butter(ord,Wcut);  
sosb = zp2sos(zb,pb,kb);
```

```
[zc,pc,kc] = cheby1(ord,ripp1,Wcut);  
sosC = zp2sos(zc,pc,kc);
```

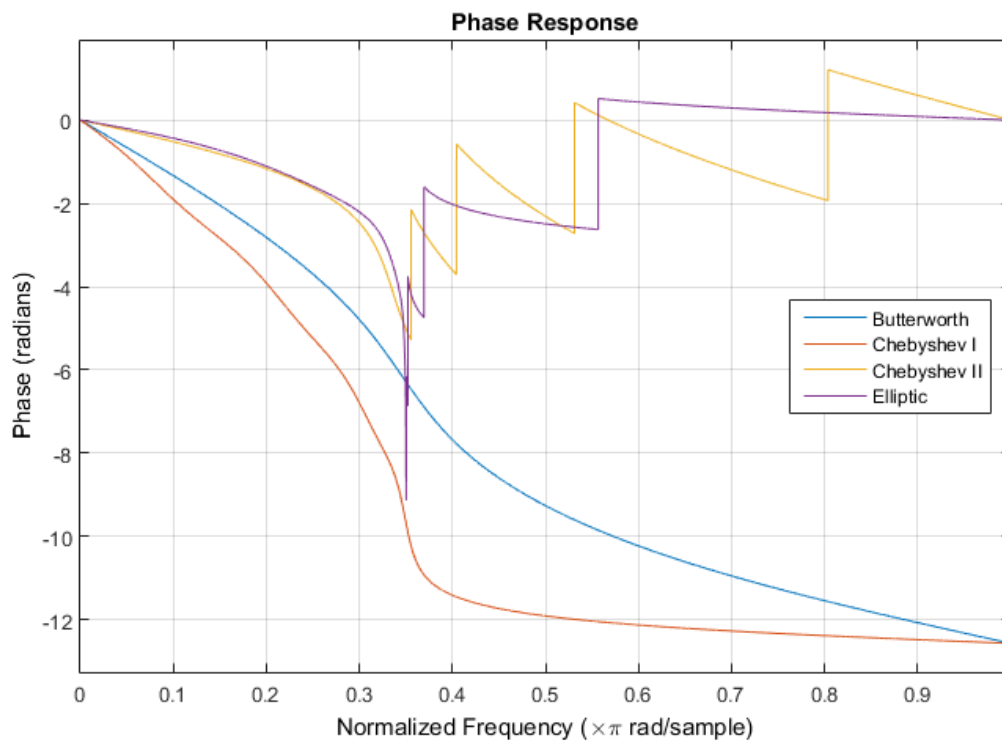
```
[zd,pd,kd] = cheby2(ord,atten,Wcut);
sosd = zp2sos(zd,pd,kd);

[ze,pe,ke] = ellip(ord,ripp1,atten,Wcut);
sose = zp2sos(ze,pe,ke);

fv = fvtool(sosb,sosc,sosd,sose,'Analysis','phase');
legend(fv,'Butterworth','Chebyshev I','Chebyshev II','Elliptic', ...
        'Location','East')
phs = [islinphase(sosb) islinphase(sosc) ...
        islinphase(sosd) islinphase(sose)]

phs =

    0    0    0    0
```



### See Also

`designfilt` | `digitalFilter` | `isallpass` | `ismaxphase` | `isminphase` | `isstable`

## isminphase

Determine whether filter is minimum phase

### Syntax

```
flag = isminphase(b,a)
flag = isminphase(sos)
flag = isminphase(d)
flag = isminphase(...,tol)
flag = isminphase(hs,...)
isminphase(hs,'Arithmetic',arithtype)
flag = isminphase(h)
```

### Description

A filter is *minimum phase* when all the zeros of its transfer function are on or inside the unit circle, or the numerator is a scalar. An equivalent definition for a minimum phase filter is a causal and stable system with a causal and stable inverse.

`flag = isminphase(b,a)` returns a logical output, `flag`, equal to `true` if the filter specified by numerator coefficients, `b`, and denominator coefficients, `a`, is a minimum phase filter.

`flag = isminphase(sos)` returns `true` if the filter specified by second order sections matrix, `sos`, is minimum phase. `sos` is a  $K$ -by-6 matrix, where the number of sections,  $K$ , must be greater than or equal to 2. Each row of `sos` corresponds to the coefficients of a second order (biquad) filter. The  $i$ th row of the `sos` matrix corresponds to `[bi(1) bi(2) bi(3) ai(1) ai(2) ai(3)]`.

`flag = isminphase(d)` returns `true` if the digital filter, `d`, has minimum phase. Use `designfilt` to generate `d` based on frequency-response specifications.

`flag = isminphase(...,tol)` uses the tolerance, `tol`, to determine when two numbers are close enough to be considered equal. If not specified, `tol`, defaults to `eps^(2/3)`.



`flag = isminphase(hs,...)` determines whether the filter System object `hs` is minimum phase, returning 1 if true and 0 if false. You must have the DSP System Toolbox software to use this syntax.

`isminphase(hs,'Arithmetic',arithtype)` analyzes the filter System object `hs` based on the specified *arithtype*. *arithtype* can be 'double', 'single', or 'fixed'. When you specify 'double' or 'single', the function performs double- or single-precision analysis. When you specify 'fixed', the arithmetic changes depending on the setting of the `CoefficientDataType` property and whether the System object is locked or unlocked. You must have the DSP System Toolbox software to use this syntax.

### Details for Fixed-Point Arithmetic

| System Object State | Coefficient Data Type | Rule   |
|---------------------|-----------------------|--|
| Unlocked            | 'Same as input'       | The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.  |
| Unlocked            | 'Custom'              | The function performs fixed-point analysis based on the setting of the <code>CustomCoefficientsDataType</code> property.   |
| Locked              | 'Same as input'       | When the input data type is 'double' or 'fixed', the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption. |
| Locked              | 'Custom'              | The function performs fixed-point analysis based on the setting of the <code>CustomCoefficientsDataType</code> property.   |

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

`flag = isminphase(h)` determines if the `dfilt` filter object `h` is minimum phase. If you have the DSP System Toolbox software, `isminphase` works with `adapfilt` and `mfilt` objects.

## Examples

### Minimum Phase Filters

Design a sixth-order lowpass Butterworth IIR filter using second order sections. Specify a normalized 3-dB frequency of 0.15. Check if the filter has minimum phase.

```
[z,p,k] = butter(6,0.15);  
SOS = zp2sos(z,p,k);  
min_flag = isminphase(SOS)
```

```
min_flag =
```

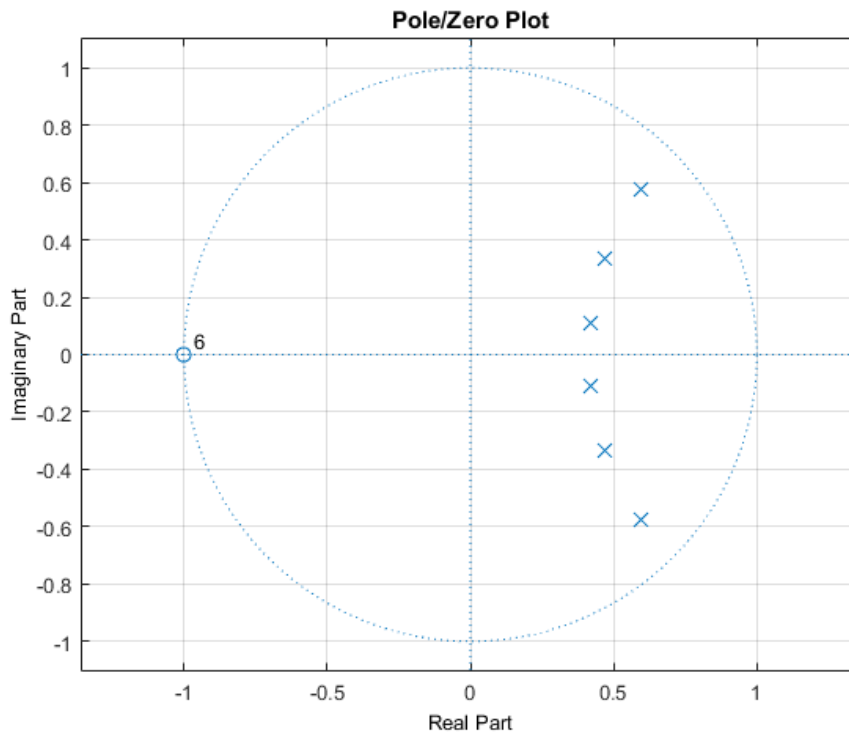
```
1
```

Redesign the filter using `designfilt`. Check that the zeros and poles of the transfer function are on or within the unit circle.

```
d = designfilt('lowpassiir','DesignMethod','butter','FilterOrder',6, ...  
              'HalfPowerFrequency',0.25);  
d_flag = isminphase(d)  
zplane(d)
```

```
d_flag =
```

```
1
```



Given a filter defined with a set of single-precision numerator and denominator coefficients, check if it has minimum phase for different tolerance values.

```
b = single([1 1.00001]);
a = single([1 0.45]);
min_flag1 = isminphase(b,a)
min_flag2 = isminphase(b,a,1e-3)
```

```
min_flag1 =
```

```
0
```

```
min_flag2 =
```

```
1
```

**See Also**

`designfilt` | `digitalFilter` | `isallpass` | `islinphase` | `ismaxphase` | `isstable`

# ismaxphase

Determine whether filter is maximum phase

## Syntax

```
flag = ismaxphase(b,a)
flag = ismaxphase(sos)
flag = ismaxphase(d)
flag = ismaxphase(...,tol)
flag = ismaxphase(hs,...)
flag = ismaxphase(hs,'Arithmetic',arithtype)
flag = ismaxphase(h)
```

## Description

`flag = ismaxphase(b,a)` returns a logical output, `flag`, equal to `true` if the filter specified by numerator coefficients, `b`, and denominator coefficients, `a`, is a maximum phase filter.

`flag = ismaxphase(sos)` returns `true` if the filter specified by second order sections matrix, `sos`, is a maximum phase filter. `sos` is a  $K$ -by-6 matrix, where the number of sections,  $K$ , must be greater than or equal to 2. Each row of `sos` corresponds to the coefficients of a second order (biquad) filter. The  $i$ th row of the `sos` matrix corresponds to `[bi(1) bi(2) bi(3) ai(1) ai(2) ai(3)]`.

`flag = ismaxphase(d)` returns `true` if the digital filter, `d`, has maximum phase. Use `designfilt` to generate `d` based on frequency-response specifications.

`flag = ismaxphase(...,tol)` uses the tolerance, `tol`, to determine when two numbers are close enough to be considered equal. If not specified, `tol`, defaults to `eps^(2/3)`.

`flag = ismaxphase(hs,...)` returns `true` if the filter System object `hs` is a maximum phase filter. You must have the DSP System Toolbox software to use this syntax.

`flag = ismaxphase(hs, 'Arithmetic', arithtype)` analyzes the filter System object `hs` based on the specified *arithtype*. *arithtype* can be 'double', 'single', or 'fixed'. When you specify 'double' or 'single', the function performs double- or single-precision analysis. When you specify 'fixed', the arithmetic changes depending on the setting of the `CoefficientDataType` property and whether the System object is locked or unlocked. You must have the DSP System Toolbox software to use this syntax.

### Details for Fixed-Point Arithmetic

| System Object State | Coefficient Data Type | Rule   |
|---------------------|-----------------------|--|
| Unlocked            | 'Same as input'       | The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.  |
| Unlocked            | 'Custom'              | The function performs fixed-point analysis based on the setting of the <code>CustomCoefficientsDataType</code> property.   |
| Locked              | 'Same as input'       | When the input data type is 'double' or 'fixed', the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption. |
| Locked              | 'Custom'              | The function performs fixed-point analysis based on the setting of the <code>CustomCoefficientsDataType</code> property.   |

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

`flag = ismaxphase(h)` returns `true` if the `dfilt` filter object `h` is a maximum phase filter. If you have the DSP System Toolbox software, `ismaxphase` works with `adapfilt` and `mfilt` objects.

## Examples

### Maximum- and Minimum-Phase Filters

Design maximum-phase and minimum-phase lattice filters and verify their phase type.

```
k = [1/6 1/1.4];
bmax = latc2tf(k, 'max');
bmin = latc2tf(k, 'min');
max_flag = ismaxphase(bmax)
min_flag = isminphase(bmin)
```

```
max_flag =
```

```
1
```

```
min_flag =
```

```
1
```

Given a filter defined with a set of single precision numerator and denominator coefficients, check if it is maximum phase for different values of the tolerance.

```
b = single([1 -0.9999]);
a = single([1 0.45]);
max_flag1 = ismaxphase(b,a)
max_flag2 = ismaxphase(b,a,1e-3)
```

```
max_flag1 =
```

```
0
```

```
max_flag2 =
```

1

**See Also**

`designfilt` | `digitalFilter` | `isallpass` | `islinphase` | `isminphase` |  
`isstable`



# issingle

Determine if digital filter coefficients are single precision

## Syntax

```
flag = issingle(d)
```

## Description

`flag = issingle(d)` returns `true` if the coefficients of a digital filter, `d`, are single precision.

## Examples

### Single- and Double-Precision Filters

Use `designfilt` to design a 6th-order highpass IIR filter. Specify a normalized passband frequency of  $0.6\pi$  rad/sample. Convert it to a single-precision filter. Identify the precision in each case.

```
fd = designfilt('highpassiir', 'FilterOrder',6, 'PassbandFrequency',0.6);  
isd = issingle(fd)  
fs = single(fd);  
iss = issingle(fs)
```

```
isd =
```

```
0
```

```
iss =
```

```
1
```

## Input Arguments

### **d** — Digital filter

`digitalFilter` object

Digital filter, specified as a `digitalFilter` object. Use `designfilt` to generate `d` based on frequency-response specifications. If you want a single-precision filter, apply `single` to the output of `designfilt`.

Example: `d = designfilt('lowpassiir','FilterOrder',3,'HalfPowerFrequency',0.5)` specifies a third-order Butterworth filter with normalized 3-dB frequency  $0.5\pi$  rad/sample.

## Output Arguments

### **flag** — Type identification

logical scalar

Type identification, returned as a logical scalar.

### See Also

`designfilt` | `digitalFilter` | `double` | `isdouble` | `single`

# isstable

Determine whether filter is stable

## Syntax

```
flag = isstable(b,a)
flag = isstable(sos)
flag = isstable(d)
flag = isstable(hs)
flag = isstable(hs,'Arithmetic',arithtype)
flag = isstable(h)
```

## Description

`flag = isstable(b,a)` returns a logical output, `flag`, equal to `true` if the filter specified by numerator coefficients, `b`, and denominator coefficients, `a`, is a stable filter. If the poles lie on or outside the circle, `isstable` returns `false`. If the poles are inside the circle, `isstable` returns `true`.

`flag = isstable(sos)` returns `true` if the filter specified by second order sections matrix, `sos`, is stable. `sos` is a  $K$ -by-6 matrix, where the number of sections,  $K$ , must be greater than or equal to 2. Each row of `sos` corresponds to the coefficients of a second order (biquad) filter. The  $i$ th row of the `sos` matrix corresponds to `[bi(1) bi(2) bi(3) ai(1) ai(2) ai(3)]`.

`flag = isstable(d)` returns `true` if the digital filter, `d`, is stable. Use `designfilt` to generate `d` based on frequency-response specifications.

`flag = isstable(hs)` returns `true` if the filter System object `hs` is stable. You must have the DSP System Toolbox software to use this syntax.

`flag = isstable(hs,'Arithmetic',arithtype)` analyzes the filter System object `hs` based on the specified `arithtype`. `arithtype` can be `'double'`, `'single'`, or `'fixed'`. When you specify `'double'` or `'single'`, the function performs double- or single-precision analysis. When you specify `'fixed'`, the arithmetic changes depending

on the setting of the `CoefficientDataType` property and whether the System object is locked or unlocked. You must have the DSP System Toolbox software to use this syntax.

**Details for Fixed-Point Arithmetic**

| System Object State | Coefficient Data Type | Rule   |
|---------------------|-----------------------|--|
| Unlocked            | 'Same as input'       | The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.  |
| Unlocked            | 'Custom'              | The function performs fixed-point analysis based on the setting of the <code>CustomCoefficientsDataType</code> property.   |
| Locked              | 'Same as input'       | When the input data type is 'double' or 'fixed', the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption. |
| Locked              | 'Custom'              | The function performs fixed-point analysis based on the setting of the <code>CustomCoefficientsDataType</code> property.   |

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

`flag = isstable(h)` returns `true` if the filter object, `h`, is stable. If you have the DSP System Toolbox, you can use `isstable` with `adaptfilt` and `mfilt` objects.

## Examples

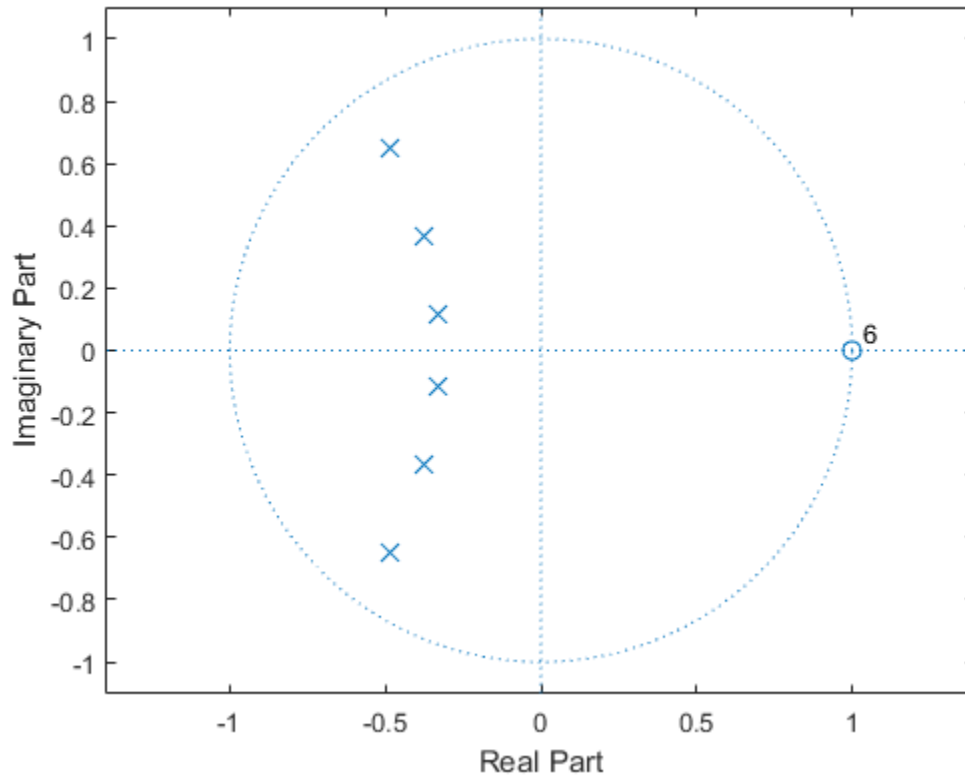
### Filter Stability

Design a sixth-order Butterworth highpass IIR filter using second order sections. Specify a normalized 3-dB frequency of 0.7. Determine if the filter is stable.

```
[z,p,k] = butter(6,0.7,'high');  
SOS = zp2sos(z,p,k);  
flag = isstable(SOS)  
zplane(z,p)
```

```
flag =
```

```
1
```

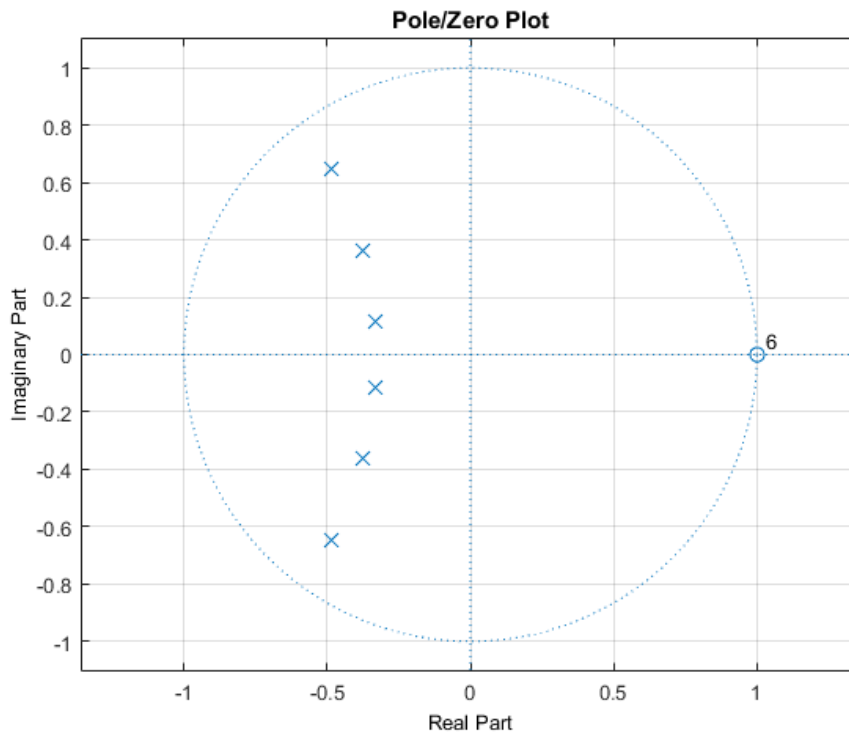


Redesign the filter using `designfilt` and check it for stability.

```
d = designfilt('highpassiir', 'DesignMethod', 'butter', 'FilterOrder', 6, ...
              'HalfPowerFrequency', 0.7);
dflg = isstable(d)
zplane(d)
```

dflg =

1



Create a filter and determine its stability at double and single precision.

```
b = [1 -0.5];  
a = [1 -0.999999999];  
act_flag1 = isstable(b,a)  
act_flag2 = isstable(single(b),single(a))
```

```
act_flag1 =
```

```
1
```

```
act_flag2 =
```

```
0
```

**See Also**

`designfilt` | `digitalFilter` | `isallpass` | `islinphase` | `ismaxphase` |  
`isminphase` | `zplane`



## is2rc

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

#### Compute Reflection Coefficients

Define a vector, `isin`, of inverse sine parameters and determine the corresponding reflection coefficients.

```
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, John R., John G. Proakis, and John H. L. Hansen. *Discrete-Time Processing of Speech Signals*. New York: Macmillan, 1993.

### See Also

`ac2rc` | `lar2rc` | `poly2rc` | `rc2is`

# kaiser

Kaiser window

## Syntax

```
w = kaiser(L,beta)
```

## Description

`w = kaiser(L,beta)` returns an L-point Kaiser window in the column vector `w`. `beta` is the Kaiser window 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  $\alpha$  dB, use the following  $\beta$ .

$$\beta = \begin{cases} 0.1102(\alpha - 8.7), & \alpha > 50 \\ 0.5842(\alpha - 21)^{0.4} + 0.07886(\alpha - 21), & 50 \geq \alpha \geq 21 \\ 0, & \alpha < 21 \end{cases}$$

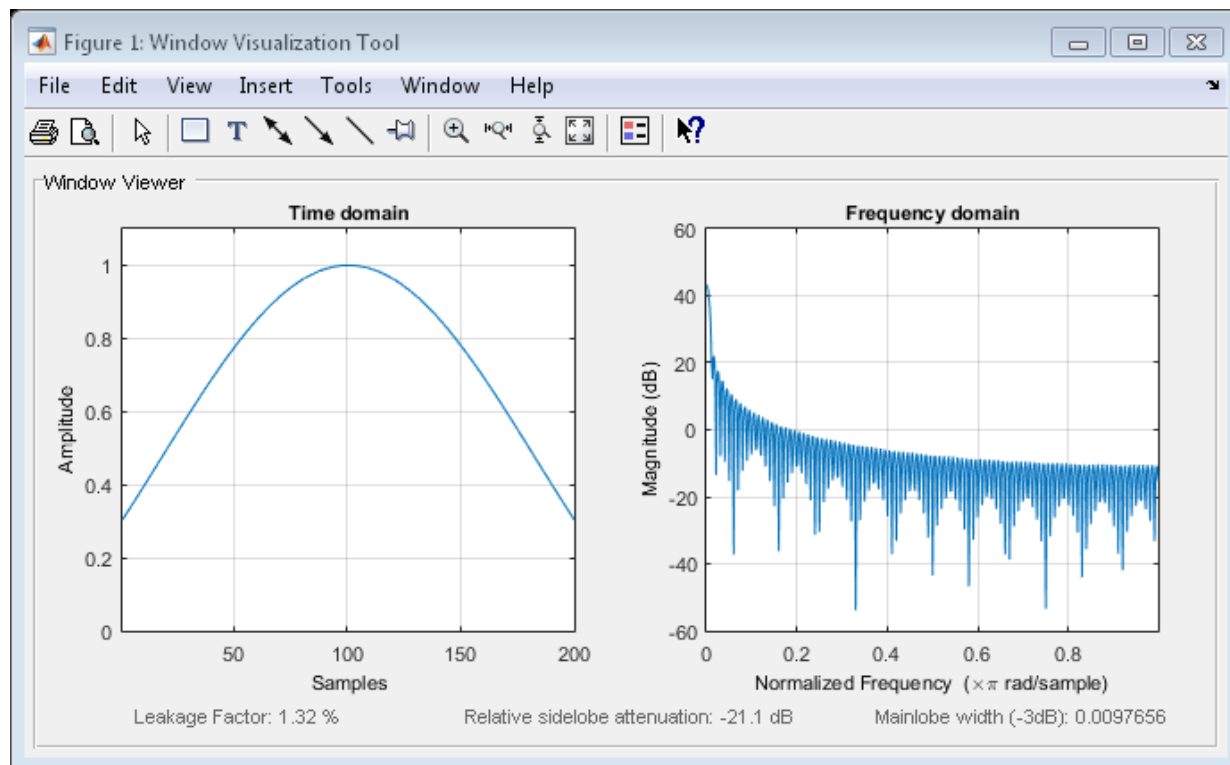
Increasing `beta` widens the mainlobe and decreases the amplitude of the sidelobes (i.e., increases the attenuation).

## Examples

### Kaiser Window

Create a 200-point Kaiser window with a beta of 2.5. Display the result using `wvtool`.

```
w = kaiser(200,2.5);  
wvtool(w)
```



## References

- [1] Kaiser, James F. "Nonrecursive Digital Filter Design Using the  $I_0$ -Sinh Window Function." *Proceedings of the 1974 IEEE International Symposium on Circuits and Systems*. April, 1974, pp.20–23.
- [2] Digital Signal Processing Committee of the IEEE Acoustics, Speech, and Signal Processing Society, eds. *Selected Papers in Digital Signal Processing*. Vol. II. New York: IEEE Press, 1976.
- [3] Oppenheim, Alan V., Ronald W. Schaffer, and John R. Buck. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1999, p. 474.

**See Also**

chebwin | gausswin | kaiserord | tukeywin | window | wintool | wvtool

# kaiserord

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%).

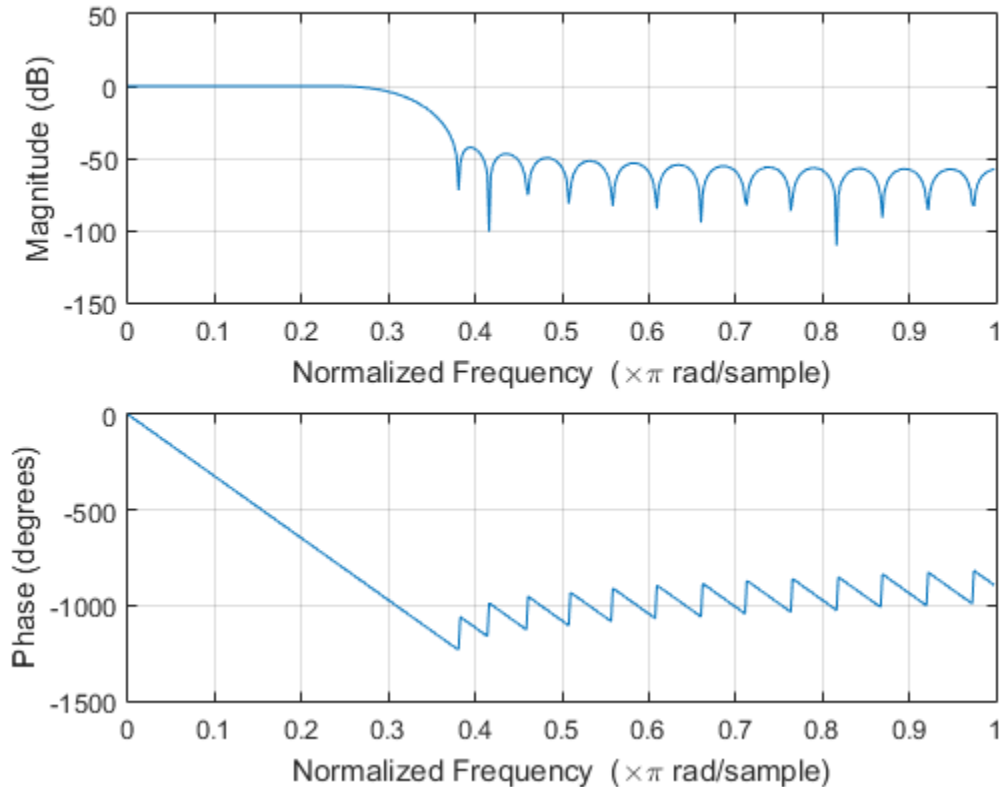
---

## Examples

### Kaiser Window Filter Design

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)
```



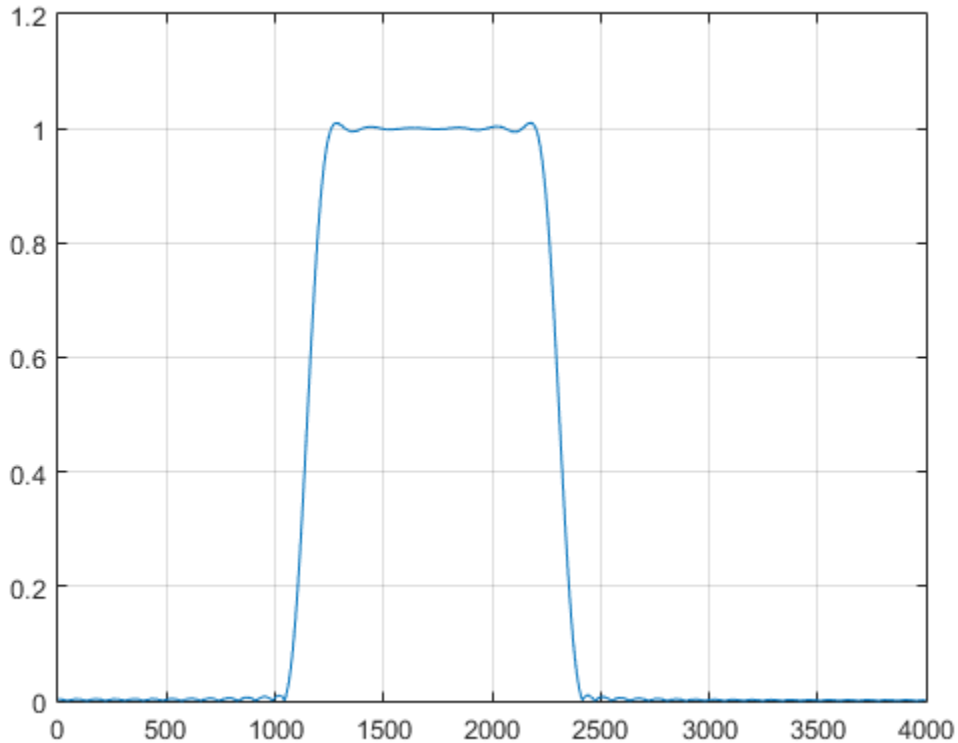
Design an odd-length bandpass filter. Note that odd length means even order, so the input to `fir1` must be an even integer.

```
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

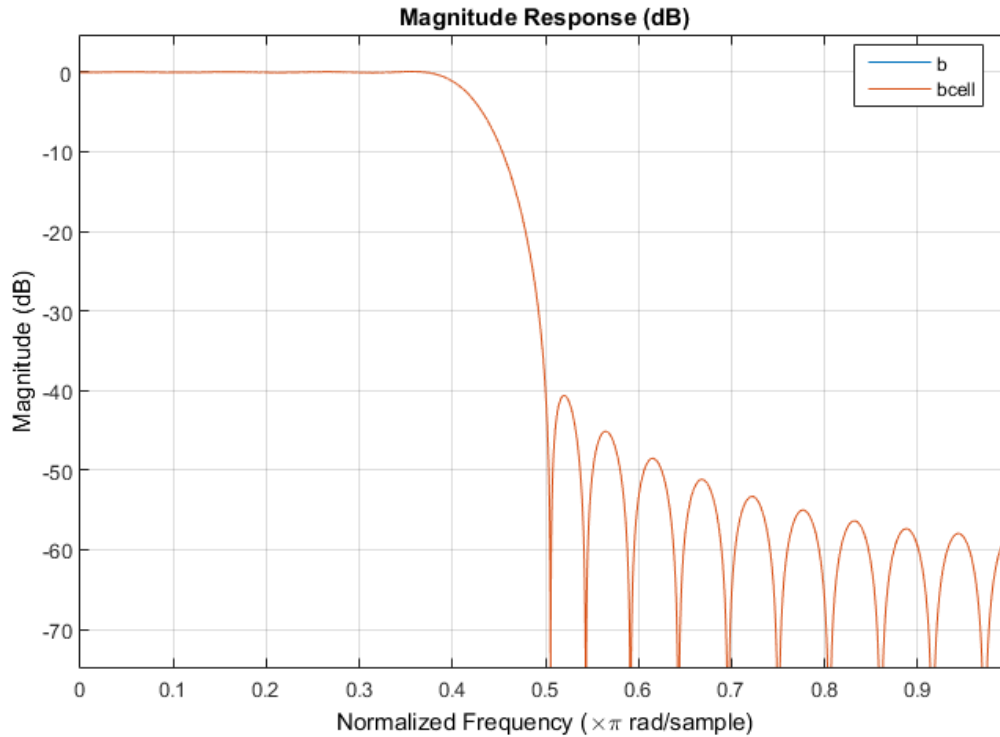


Design a lowpass filter with a passband cutoff of 1500 Hz, a stopband cutoff of 2000 Hz, a passband ripple of 0.01, a stopband ripple of 0.1, and a sample rate of 8000 Hz. Design an equivalent filter using the 'cell' option.

```
[n,Wn,beta,ftype] = kaiserord([1500 2000],[1 0],...  
    [0.01 0.1],8000);  
b = fir1(n,Wn,ftype,kaiser(n+1,beta),'noscale');  
  
c = kaiserord([1500 2000],[1 0],[0.01 0.1],8000,'cell');  
bcell = fir1(c{:});  
  
hfvt = fvtool(b,1,bcell,1);
```



```
legend(hfvt, 'b', 'bcell')
```



## More About

### Tips

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  $n = L-1$  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.

## Algorithms

`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

$$\beta = \begin{cases} 0.1102(\alpha - 8.7), & \alpha > 50 \\ 0.5842(\alpha - 21)^{0.4} + 0.07886(\alpha - 21), & 21 \leq \alpha \leq 50 \\ 0, & \alpha < 21 \end{cases}$$

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, James F. “Nonrecursive Digital Filter Design Using the  $I_0$ -sinh Window Function.” *Proceedings of the 1974 IEEE International Symposium on Circuits and Systems*. 1974, pp.20–23.
- [2] Digital Signal Processing Committee of the IEEE Acoustics, Speech, and Signal Processing Society, eds. *Selected Papers in Digital Signal Processing*. Vol. II. New York: IEEE Press, 1976, pp.123–126.
- [3] Oppenheim, Alan V., Ronald W. Schaffer, and John R. Buck. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1999.

## See Also

`fir1` | `kaiser` | `firpmord`

# kaiserwin

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')
```

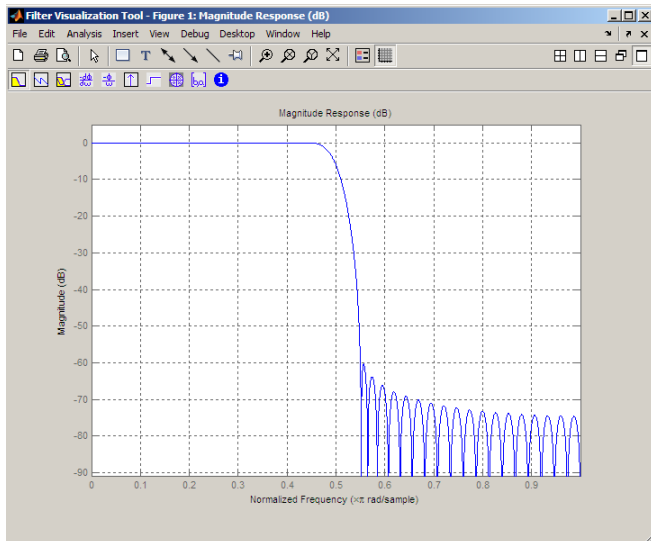
For complete help about using `kaiserwin`, refer to the command line help system. For example, to get specific information about using `kaiserwin` with `d`, the specification object, enter the following at the MATLAB prompt.

```
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](#) | [fdesign](#)

## lar2rc

Convert log area ratio parameters to reflection coefficients

### Syntax

$k = \text{lar2rc}(g)$

### Description

$k = \text{lar2rc}(g)$  returns a vector of reflection coefficients  $k$  from a vector of log area ratio parameters  $g$ .

### Examples

#### Calculate Reflection Coefficients

Given a vector,  $g$ , of log area ratio parameters, determine the corresponding vector of reflection coefficients.

```
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, John R., John G. Proakis, and John H. L. Hansen. *Discrete-Time Processing of Speech Signals*. New York: Macmillan, 1993.

### See Also

ac2rc | is2rc | poly2rc | rc2lar

## latc2tf

Convert lattice filter parameters to transfer function form

### Syntax

```
[num,den] = latc2tf(k,v)
[num,den] = latc2tf(k,'iioption')
num = latc2tf(k,'fioption')
```

### 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,'iioption')` produces an IIR filter transfer function according to the value of the string `'iioption'`:

- `'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,'fioption')` produces an FIR filter according to the value of the string `'fioption'`:

- `'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

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  $|k| \leq 1$ , `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`.

## Examples

Filter data with an FIR lattice filter:

```
%create data
x=randn(512,1);
%reflection coefficients for 3-point MA filter
[f,g]=latcfilt([1/2 1],x);
%compare f vector to dfilt.latticemamin output
Hd=dfilt.latticemamin([1/2 1]);
y=filter(Hd,x);
isequal(y,f) %returns 1
%compare g vector to dfilt.latticemamax output
Hd1=dfilt.latticemamax([1/2 1]);
y1=filter(Hd1,x);
isequal(g,y1) %returns 1
```

## See Also

`dfilt.latticemamax` | `dfilt.latticemamin` | `filter`



# levinson

Levinson-Durbin recursion

## Syntax

```
a = levinson(r)
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)` finds the coefficients of a `length(r) - 1` 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=length(r) - 1` is the default order of the denominator polynomial  $A(z)$ ; that is, `a = [1 a(2) ... a(n+1)]`. The filter coefficients are ordered in descending powers of  $z^{-1}$ .

$$H(z) = \frac{1}{A(z)} = \frac{1}{1 + a(2)z^{-1} + \dots + a(n+1)z^{-n}}$$

`a = levinson(r, n)` returns the coefficients for an autoregressive model of order `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`.

---

## More About

### Algorithms

`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  $\mathbf{r} = [r(1) \dots r(n+1)]$  is the input autocorrelation vector, and  $r(i)^*$  denotes the complex conjugate of  $r(i)$ . The input  $\mathbf{r}$  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` | `stmcb` | `schurrc`

# lp2bp

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`.

$$\frac{B(s)}{A(s)} = \frac{b(1)s^n + \dots + b(n)s + b(n+1)}{a(1)s^m + \dots + a(m)s + a(m+1)}$$

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.

## More About

### Algorithms

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  $\omega_0$  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  $\omega_0$  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` | `lp2bs` | `lp2hp` | `lp2lp` | `impinvar`

## lp2bs

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  $s$ .

$$\frac{B(s)}{A(s)} = \frac{b(1)s^n + \dots + b(n)s + b(n+1)}{a(1)s^m + \dots + a(m)s + a(m+1)}$$

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.

## More About

### Algorithms

lp2bs is a highly accurate state-space formulation of the classic analog filter frequency transformation. If a bandstop filter is to have center frequency  $\omega_0$  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 | lp2bp | lp2hp | lp2lp |impinvar

## lp2hp

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  $s$ .

$$\frac{B(s)}{A(s)} = \frac{b(1)s^n + \dots + b(n)s + b(n+1)}{a(1)s^m + \dots + a(m)s + a(m+1)}$$

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



$$\begin{aligned}\dot{x} &= Ax + Bu \\ y &= Cx + Du\end{aligned}$$

into a highpass filter with cutoff angular frequency  $\omega_0$ . The highpass filter is returned in matrices  $A_t$ ,  $B_t$ ,  $C_t$ ,  $D_t$ .

## More About

### Algorithms

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  $\omega_0$ , the standard  $s$ -domain transformation is

$$s = \frac{\omega_0}{p}$$

The state-space version of this transformation is

$$\begin{aligned}A_t &= \omega_0 \operatorname{inv}(A); \\ B_t &= -\omega_0 (A \setminus B); \\ C_t &= C/A; \\ D_t &= D - C/A * B;\end{aligned}$$

See lp2bp for a derivation of the bandpass version of this transformation.

### See Also

bilinear | lp2bp | lp2bs | lp2lp |impinvar

## lp2lp

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  $s$ .

$$\frac{B(s)}{A(s)} = \frac{b(1)s^n + \dots + b(n)s + b(n+1)}{a(1)s^m + \dots + a(m)s + a(m+1)}$$

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

$$\begin{aligned}\dot{x} &= Ax + Bu \\ y &= Cx + Du\end{aligned}$$

into a lowpass filter with cutoff angular frequency  $\omega_0$ . `lp2lp` returns the lowpass filter in matrices `At`, `Bt`, `Ct`, `Dt`.

## More About

### Algorithms

`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  $\omega_0$ , the standard  $s$ -domain transformation is

$$s = p / \omega_0$$

The state-space version of this transformation is

$$\begin{aligned}At &= \omega_0 * A; \\ Bt &= \omega_0 * B; \\ Ct &= C; \\ Dt &= D;\end{aligned}$$

See `lp2bp` for a derivation of the bandpass version of this transformation.

### See Also

`bilinear` | `lp2bp` | `lp2bs` | `lp2hp` | `impinvar`

## lpc

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.

$$\hat{x}(n) = -a(2)x(n-1) - a(3)x(n-2) - \dots - a(p+1)x(n-p)$$

$p$  is the order of the prediction filter polynomial,  $a = [1 \ a(2) \ \dots \ a(p+1)]$ . If  $p$  is unspecified, `lpc` uses as a default  $p = \text{length}(x) - 1$ . 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 Series Using a Forward Predictor

Estimate a data series using a third-order forward predictor. Compare the estimate to the original signal.

First, create the signal data as the output of an autoregressive process driven by normalized white Gaussian noise. Use the last 4096 samples of the AR process output to avoid startup transients.

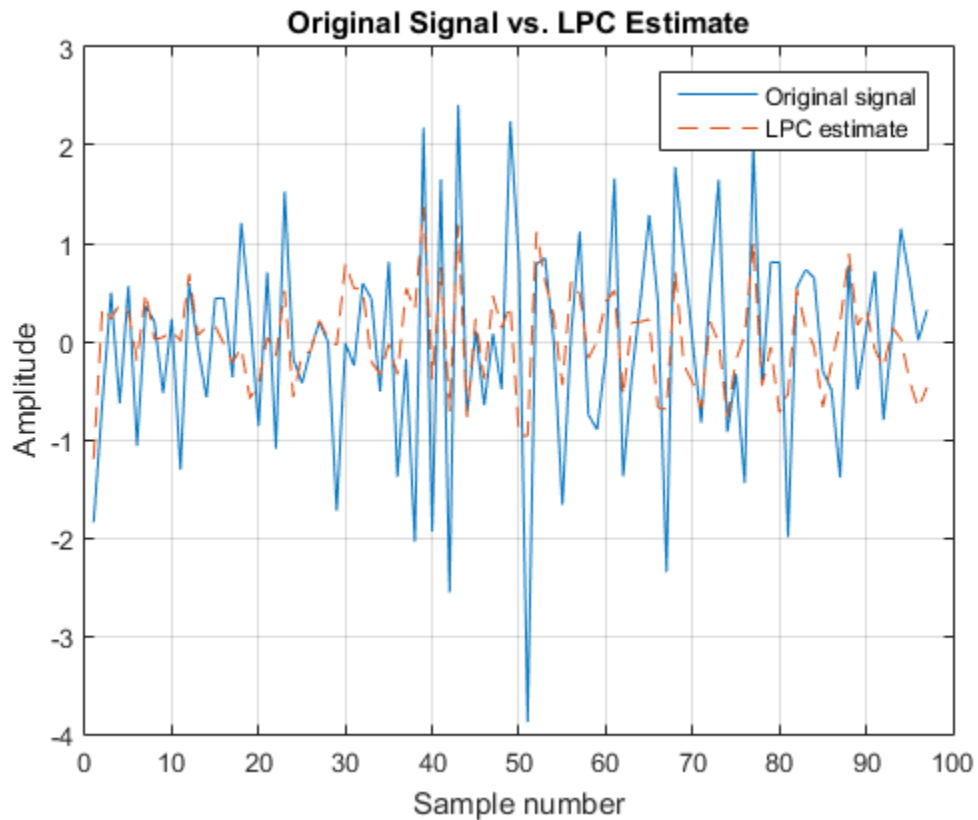
```
noise = randn(50000,1);  
x = filter(1,[1 1/2 1/3 1/4],noise);  
x = x(45904:50000);
```

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);  
e = x-est_x;  
[acs,lags] = xcorr(e,'coeff');
```

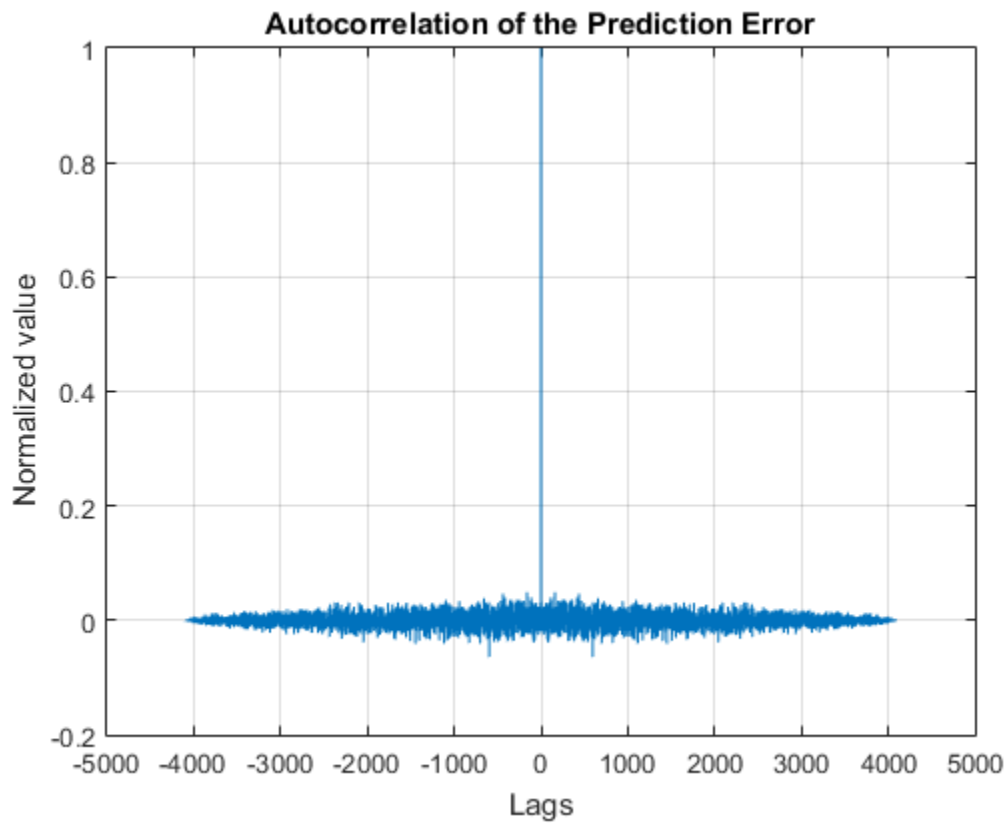
Compare the predicted signal to the original signal.

```
plot(1:97,x(4001:4097),1:97,est_x(4001:4097),'--'), grid  
title 'Original Signal vs. LPC Estimate'  
xlabel 'Sample number', ylabel 'Amplitude'  
legend('Original signal','LPC estimate')
```



Plot the autocorrelation of the prediction error.

```
plot(lags,acs), grid
title 'Autocorrelation of the Prediction Error'
xlabel 'Lags', ylabel 'Normalized value'
```

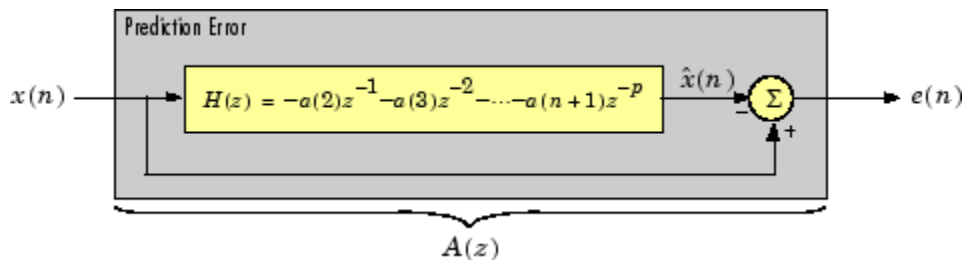


The prediction error is approximately white Gaussian noise, as expected for a third-order AR input process.

## More About

### 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.



### Algorithms

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

$$X\alpha = b$$

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  $r = [r(1) \ r(2) \ \dots \ r(p+1)]$  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*. 2nd Edition. Boston: Kluwer Academic Publishers, 1989, pp.255–257.

## See Also

`aryule` | `levinson` | `stmcb` | `prony` | `pyulear`



# lsf2poly

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  $M \times N$  with separate channels of line spectral frequencies in each column, the returned `a` matrix has the resulting prediction filter coefficients as its rows and is of size  $N \times (M + 1)$ .

## Examples

### Prediction Coefficients from Line Spectral Frequencies

Given a vector, `lsf`, of line spectral frequencies, determine the equivalent prediction filter coefficients.

```
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, John R., John G. Proakis, and John H. L. Hansen. *Discrete-Time Processing of Speech Signals*. New York: Macmillan, 1993.

[2] Rabiner, Lawrence R., and Ronald W. Schafer. *Digital Processing of Speech Signals*. Englewood Cliffs, NJ: Prentice-Hall, 1978.

**See Also**

ac2poly | poly21sf | rc2poly

# mag2db

Convert magnitude to decibels

## Syntax

```
ydb = mag2db(y)
```

## Description

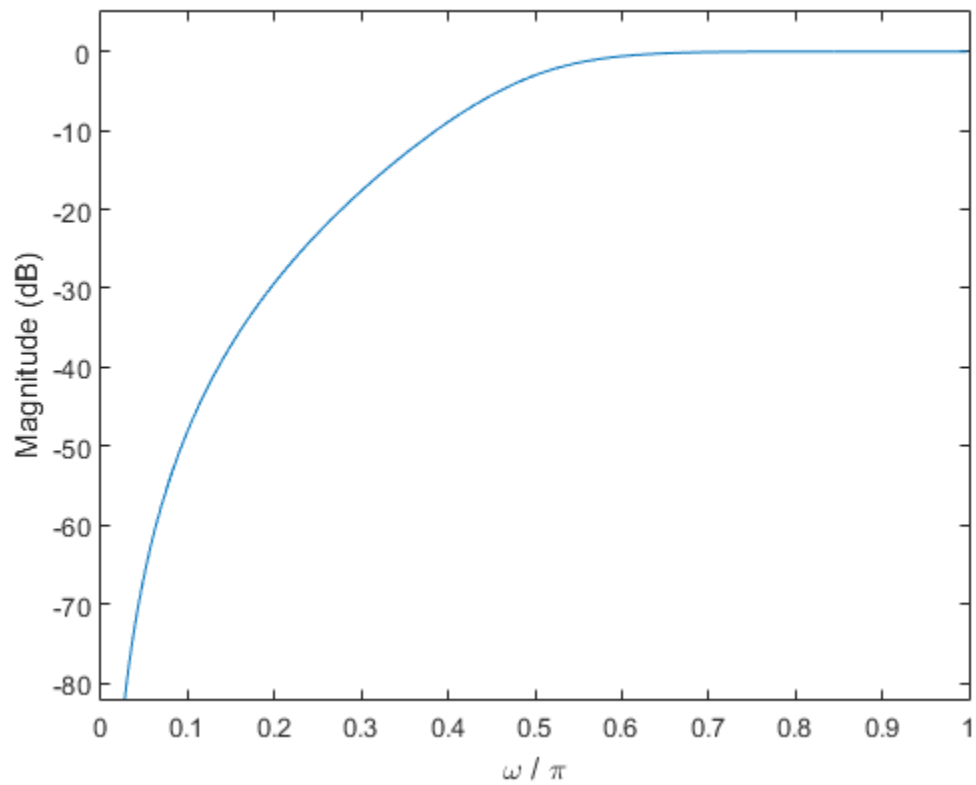
`ydb = mag2db(y)` expresses in decibels (dB) the magnitude measurements specified in `y`. The relationship between magnitude and decibels is  $ydb = 20 \log_{10}(y)$ .

## Examples

### Magnitude Response of a Highpass Filter

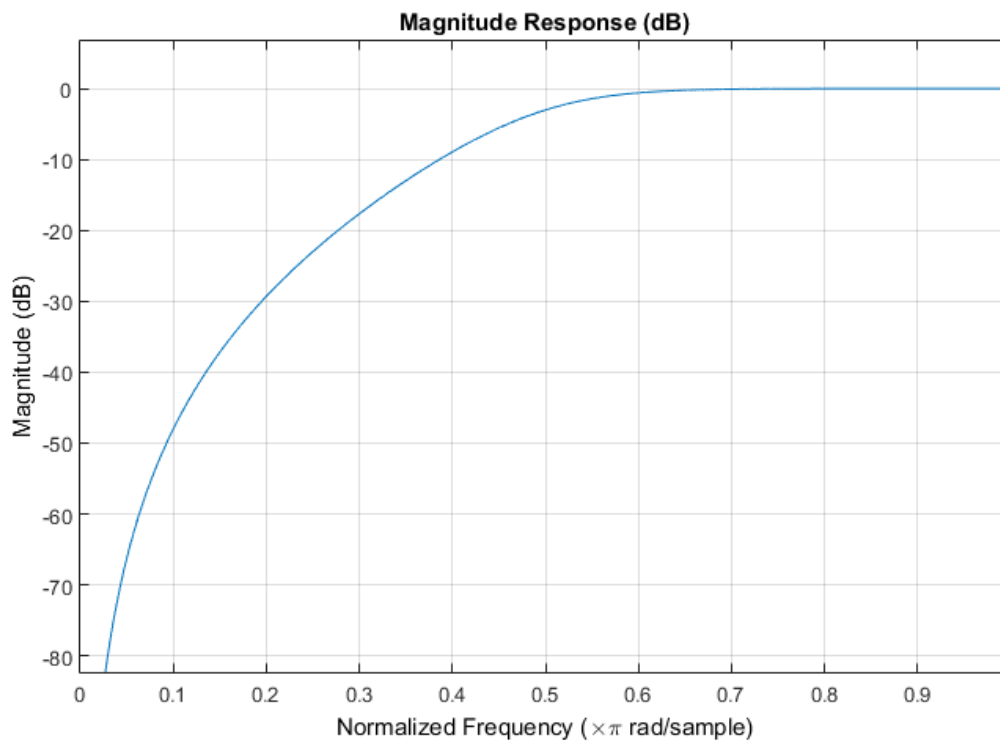
Design a 3rd-order highpass Butterworth filter having a normalized 3-dB frequency of  $0.5\pi$  rad/sample. Compute its frequency response. Express the magnitude response in decibels and plot it.

```
[b,a] = butter(3,0.5,'high');  
[h,w] = freqz(b,a);  
  
dB = mag2db(abs(h));  
  
plot(w/pi,dB)  
xlabel('\omega / \pi')  
ylabel('Magnitude (dB)')  
ylim([-82 5])
```



Repeat the computation using `fvtool`.

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



## Input Arguments

### **y** — Input array

scalar | vector | matrix | N-D array

Input array, specified as a scalar, vector, matrix, or N-D array. When *y* is nonscalar, `mag2db` is an element-wise operation.

Data Types: `single` | `double`

## Output Arguments

### **ydb** — Magnitude measurements in decibels

scalar | vector | matrix | N-D array

Magnitude measurements in decibels, returned as a scalar, vector, matrix, or N-D array of the same size as `y`.

### **See Also**

`db` | `db2mag` | `db2pow` | `pow2db`

## marcumq

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.

## More About

### Algorithms

marcumq uses the algorithm developed in [3]. The paper describes two error criteria: a relative error criterion and an absolute error criterion. marcumq utilizes the absolute error criterion.

## 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.

## See Also

besseli



# maxflat

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  $1/\sqrt{2}$  (approximately  $-3$  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

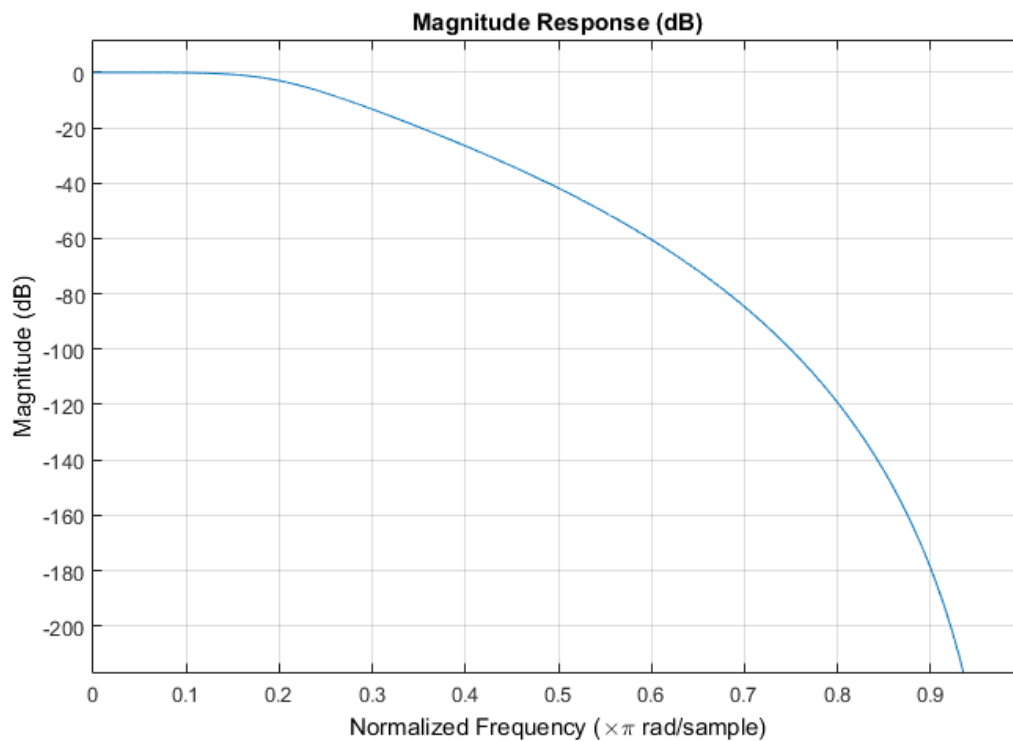
- 'both' for both the textual display and plots

## Examples

### Generalized Butterworth Filter

Design a generalized Butterworth filter with normalized cutoff frequency  $0.2\pi$  rad/s. Specify a numerator order of 10 and a denominator order of 2. Visualize the frequency response of the filter.

```
n = 10;  
m = 2;  
Wn = 0.2;  
  
[b,a] = maxflat(n,m,Wn);  
fvtool(b,a)
```



## More About

### Algorithms

The method consists of the use of formulae, polynomial root finding, and a transformation of polynomial roots.

## References

- [1] Selesnick, Ivan W., and C. Sidney Burrus. "Generalized Digital Butterworth Filter Design." *IEEE Transactions on Signal Processing*. Vol.46, Number6, 1998, pp.1688–1694.

**See Also**

butter | filter | freqz

# meanfreq

Mean frequency

## Syntax

```
freq = meanfreq(x)
freq = meanfreq(x, fs)

freq = meanfreq(pxx, f)
freq = meanfreq(sxx, f, rbw)

freq = meanfreq( ____, freqrange)

[ freq, power ] = meanfreq( ____)

meanfreq( ____)
```

## Description

`freq = meanfreq(x)` estimates the mean normalized frequency, `freq`, of the power spectrum of a time-domain signal, `x`.

`freq = meanfreq(x, fs)` estimates the mean frequency in terms of the sample rate, `fs`.

`freq = meanfreq(pxx, f)` returns the mean frequency of a power spectral density (PSD) estimate, `pxx`. The frequencies, `f`, correspond to the estimates in `pxx`.

`freq = meanfreq(sxx, f, rbw)` returns the mean frequency of a power spectrum estimate, `sxx`, with resolution bandwidth `rbw`.

`freq = meanfreq( ____, freqrange)` specifies the frequency interval over which to compute the mean frequency, using any of the input arguments from previous syntaxes. The default value for `freqrange` is the entire bandwidth of the input signal.

`[ freq, power ] = meanfreq( ____)` also returns the band power, `power`, of the spectrum. If you specify `freqrange`, then `power` contains the band power within `freqrange`.

`meanfreq( ___ )` with no output arguments plots the PSD or power spectrum and annotates the mean frequency.

## Examples

### Mean Frequency of Chirps

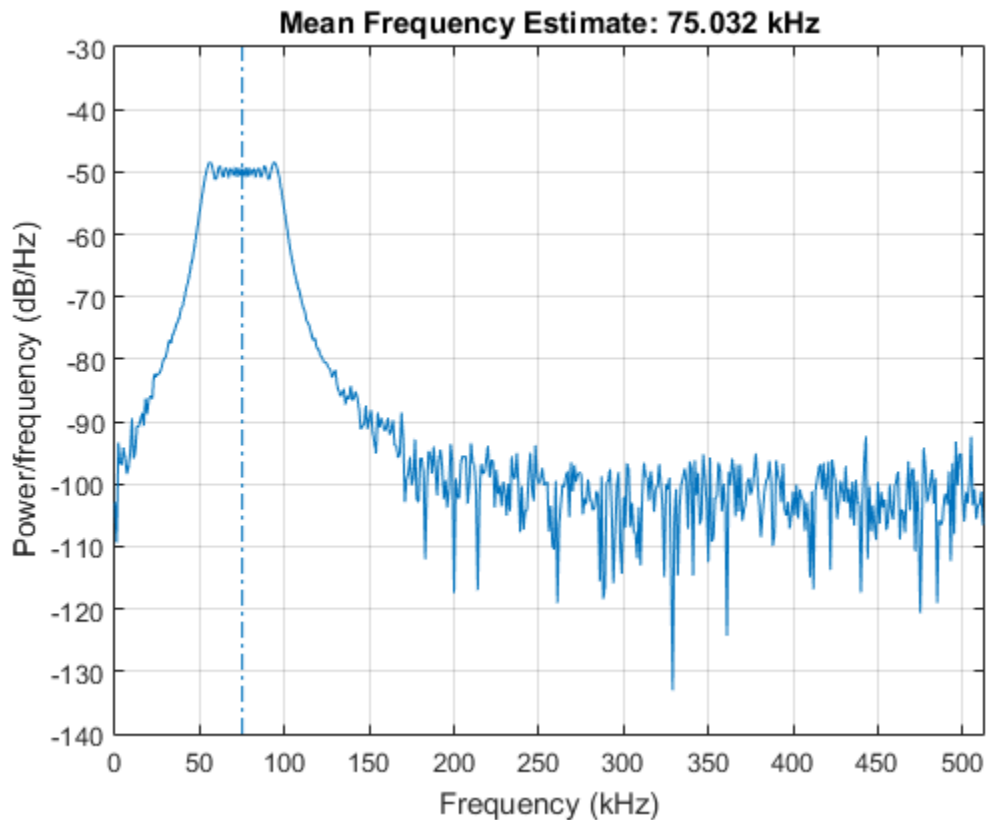
Generate 1024 samples of a chirp sampled at 1024 kHz. The chirp has an initial frequency of 50 kHz and reaches 100 kHz at the end of the sampling. Add white Gaussian noise such that the signal-to-noise ratio is 40 dB. Reset the random number generator for reproducible results.

```
nSamp = 1024;  
Fs = 1024e3;  
SNR = 40;  
rng default  
  
t = (0:nSamp-1)'/Fs;  
  
x = chirp(t,50e3,nSamp/Fs,100e3);  
x = x+randn(size(x))*std(x)/db2mag(SNR);
```

Estimate the mean frequency of the chirp. Plot the power spectral density (PSD) and annotate the mean frequency.

```
meanfreq(x,Fs)
```

```
ans =  
  
7.5032e+04
```



Generate another chirp. Specify an initial frequency of 200 kHz, a final frequency of 300 kHz, and an amplitude that is twice that of the first signal. Add white Gaussian noise.

```
x2 = 2*chirp(t,200e3,nSamp/Fs,300e3);
x2 = x2+randn(size(x2))*std(x2)/db2mag(SNR);
```

Concatenate the chirps to produce a two-channel signal. Estimate the mean frequency of each channel.

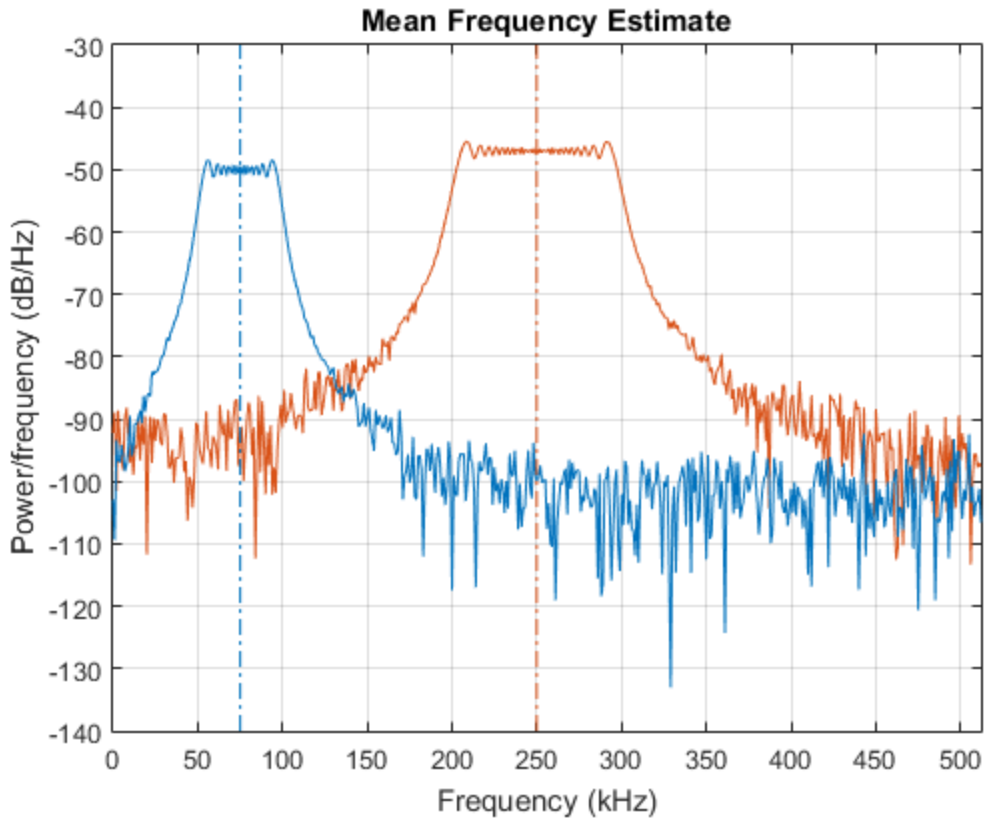
```
y = meanfreq([x x2],Fs)
```

```
y =
```

```
1.0e+05 *
0.7503 2.4999
```

Plot the PSDs of the two channels and annotate their mean frequencies.

```
meanfreq([x x2],Fs);
```



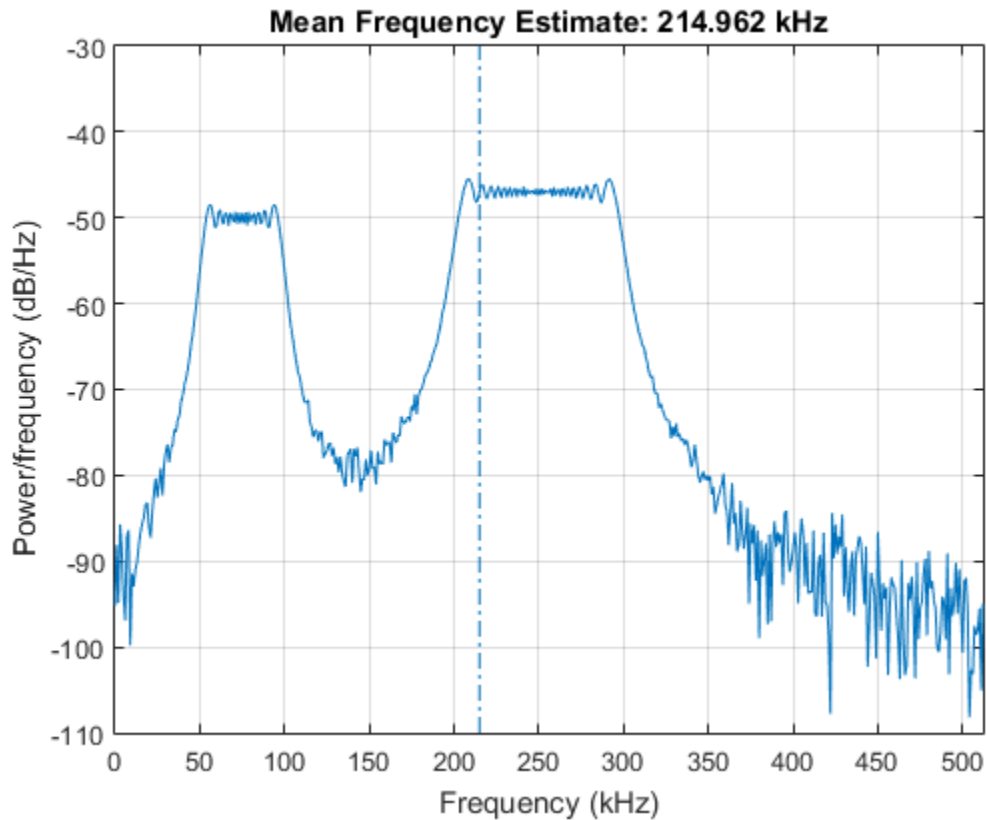
Add the two channels to form a new signal. Plot the PSD and annotate the mean frequency.

```
meanfreq(x+x2,Fs)
```



ans =

2.1496e+05



### Mean Frequency of Sinusoids

Generate 1024 samples of a 100.123 kHz sinusoid sampled at 1024 kHz. Add white Gaussian noise such that the signal-to-noise ratio is 40 dB. Reset the random number generator for reproducible results.

```
nSamp = 1024;  
Fs = 1024e3;  
SNR = 40;
```

```
rng default
```

```
t = (0:nSamp-1)'/Fs;
```

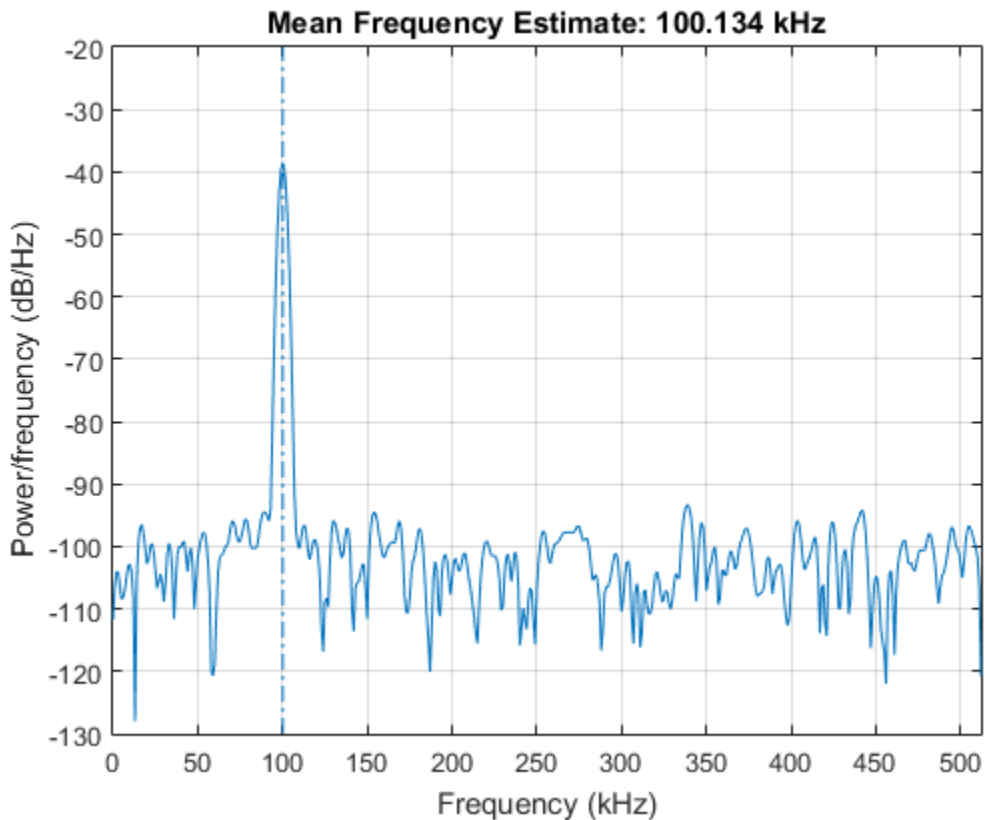
```
x = sin(2*pi*t*100.123e3);
```

```
x = x + randn(size(x))*std(x)/db2mag(SNR);
```

Use the `periodogram` function to compute the power spectral density (PSD) of the signal. Specify a Kaiser window with the same length as the signal and a shape factor of 38. Estimate the mean frequency of the signal and annotate it on a plot of the PSD.

```
[Pxx,f] = periodogram(x,kaiser(nSamp,38),[],Fs);
```

```
meanfreq(Pxx,f);
```



Generate another sinusoid, this one with a frequency of 257.321 kHz and an amplitude that is twice that of the first sinusoid. Add white noise.

```
x2 = 2*sin(2*pi*t*257.321e3);  
x2 = x2 + randn(size(x2))*std(x2)/db2mag(SNR);
```

Concatenate the sinusoids to produce a two-channel signal. Estimate the PSD of each channel and use the result to determine the mean frequency.

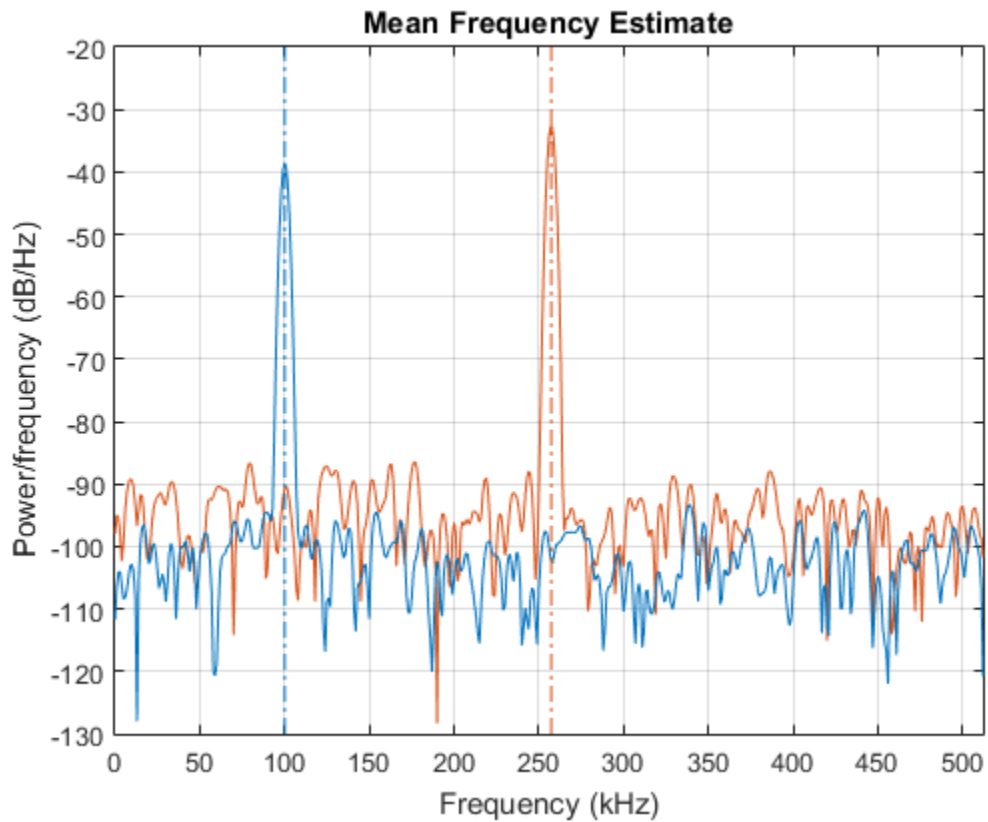
```
[Pyy,f] = periodogram([x x2],kaiser(nSamp,38),[],Fs);  
y = meanfreq(Pyy,f)
```

```
y =
```

```
1.0e+05 *  
1.0013    2.5732
```

Annotate the mean frequencies of the two channels on a plot of the PSDs.

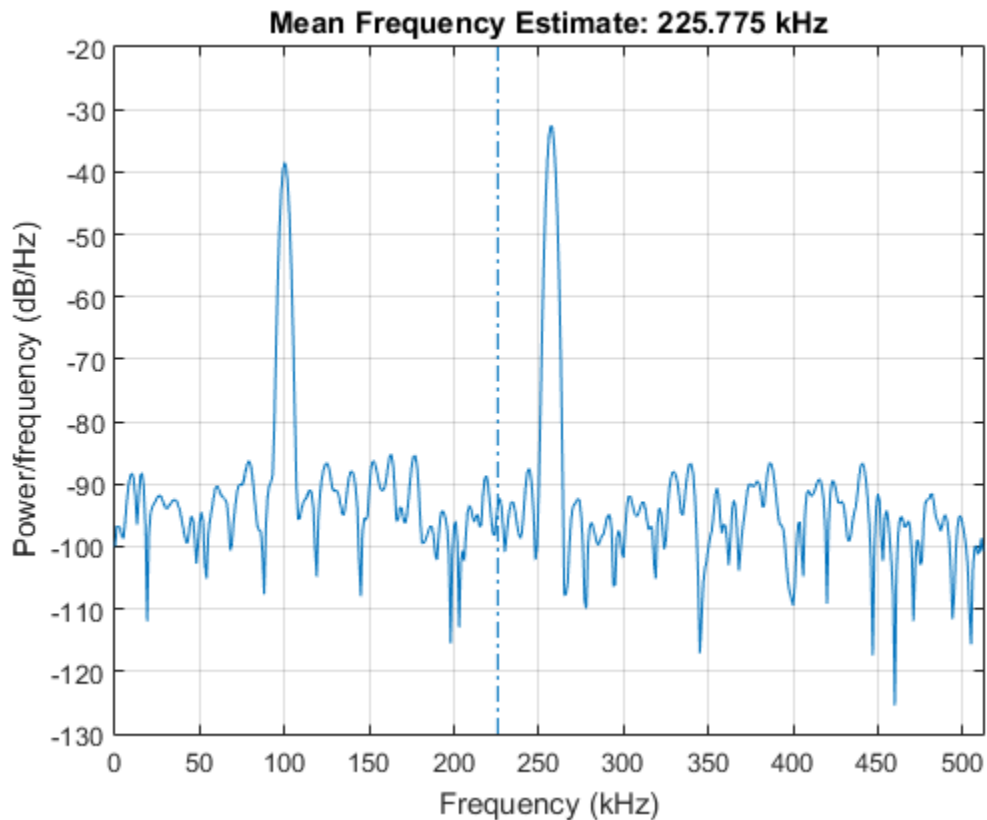
```
meanfreq(Pyy,f);
```



Add the two channels to form a new signal. Estimate the PSD and annotate the mean frequency.

```
[Pzz,f] = periodogram(x+x2,kaiser(nSamp,38),[],Fs);
```

```
meanfreq(Pzz,f);
```



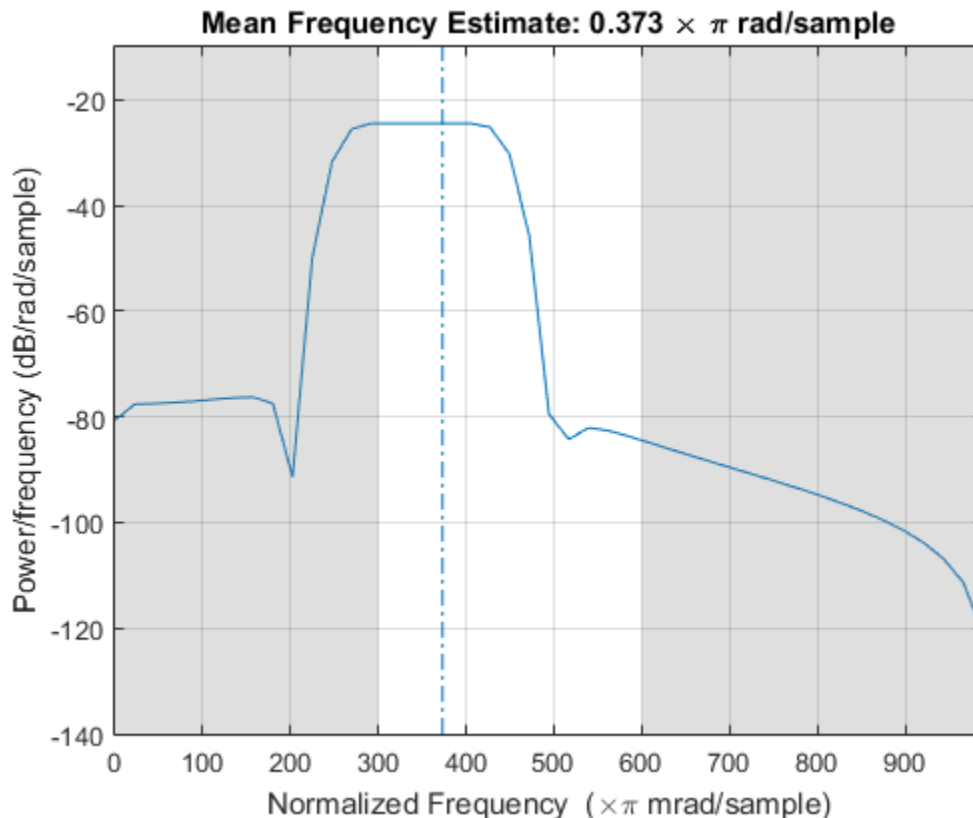
### Mean Frequency of Bandlimited Signals

Generate a signal whose PSD resembles the frequency response of an 88th-order bandpass FIR filter with normalized cutoff frequencies  $0.25\pi$  rad/sample and  $0.45\pi$  rad/sample.

```
d = fir1(88,[0.25 0.45]);
```

Compute the mean frequency of the signal between  $0.3\pi$  rad/sample and  $0.6\pi$  rad/sample. Plot the PSD and annotate the mean frequency and measurement interval.

```
meanfreq(d,[],[0.3 0.6]*pi);
```



Output the mean frequency and the band power of the measurement interval. Specifying a sample rate of  $2\pi$  is equivalent to leaving the rate unset.

```
[mnf,power] = meanfreq(d,2*pi,[0.3 0.6]*pi);
fprintf('Mean = %.3f*pi, power = %.1f%% of total \n', ...
        mnf/pi,power/bandpower(d)*100)
```

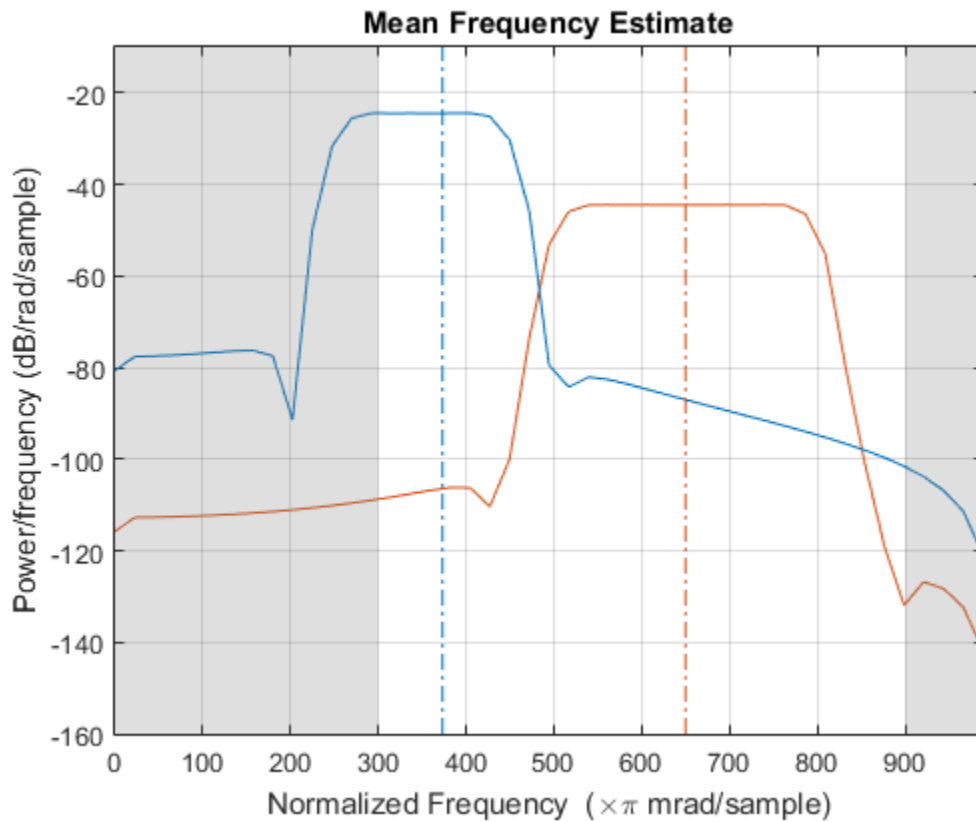
```
Mean = 0.373*pi, power = 75.6% of total
```

Add a second channel with normalized cutoff frequencies  $0.5\pi$  rad/sample and  $0.8\pi$  rad/sample and an amplitude that is one-tenth that of the first channel.

```
d = [d;fir1(88,[0.5 0.8])/10]';
```

Compute the mean frequency of the signal between  $0.3\pi$  rad/sample and  $0.9\pi$  rad/sample. Plot the PSD and annotate the mean frequency of each channel and the measurement interval.

```
meanfreq(d,[],[0.3 0.9]*pi);
```



Output the mean frequency of each channel. Divide by  $\pi$ .

```
mnf = meanfreq(d,[],[0.3 0.9]*pi)/pi
```

```
mnf =
```

0.3730    0.6500

## Input Arguments

### **x** — Input signal

vector | matrix

Input signal, specified as a vector or matrix. If **x** is a vector, it is treated as a single channel. If **x** is a matrix, then `meanfreq` computes the mean frequency of each column of **x** independently. **x** must be finite-valued.

Example: `cos(pi/4*(0:159))+randn(1,160)` is a single-channel row-vector signal.

Example: `cos(pi./[4;2]*(0:159))'+randn(160,2)` is a two-channel signal.

Data Types: `single` | `double`

### **fs** — Sample rate

positive real scalar

Sample rate, specified as a positive real scalar. The sample rate is the number of samples per unit time. If the time is measured in seconds, then the sample rate is in hertz.

Data Types: `single` | `double`

### **pxx** — Power spectral density

vector | matrix

Power spectral density (PSD), specified as a vector or matrix. If **pxx** is a matrix, then `meanfreq` computes the mean frequency of each column of **pxx** independently.

Example: `periodogram(cos(pi./[4;2]*(0:159))'+randn(160,2))` is the periodogram PSD estimate of a two-channel sinusoid embedded in white Gaussian noise.

Data Types: `single` | `double`

### **f** — Frequencies

vector

Frequencies, specified as a vector.

Data Types: `single` | `double`



**sxx — Power spectrum estimate**

vector | matrix

Power spectrum estimate, specified as a vector or matrix. If `sxx` is a matrix, then `meanfreq` computes the mean frequency of each column of `sxx` independently.

Example: `periodogram(cos(pi./[4;2]*(0:159))'+randn(160,2),'power')` is the periodogram power spectrum estimate of a two-channel sinusoid embedded in white Gaussian noise.

Data Types: single | double

**rbw — Resolution bandwidth**

positive scalar

Resolution bandwidth, specified as a positive scalar. The resolution bandwidth is the product of two values: the frequency resolution of the discrete Fourier transform and the equivalent noise bandwidth of the window used to compute the PSD.

Data Types: single | double

**freqrange — Frequency range**

two-element vector

Frequency range, specified as a two-element vector of real values. If you do not specify `freqrange`, then `meanfreq` uses the entire bandwidth of the input signal.

Data Types: single | double

## Output Arguments

**freq — Mean frequency**

scalar | vector

Mean frequency, specified as a scalar or vector.

- If you specify a sample rate, then `freq` has the same units as `fs`.
- If you do not specify a sample rate, then `freq` has units of rad/sample.

**power — Band power**

scalar | vector

Band power, returned as a scalar or vector.

**See Also**

`findpeaks` | `medfreq` | `periodogram` | `plomb` | `pwelch`

**Introduced in R2015a**

# medfilt1

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 the input vector,  $x$ . The default value of  $n$  is 3. The function considers the signal to be 0 beyond the endpoints. The output,  $y$ , has the same length as  $x$ .

For odd  $n$ ,  $y(k)$  is the median of  $x(k - (n - 1) / 2 : k + (n - 1) / 2)$ .

For even  $n$ ,  $y(k)$  is the median of  $x(k - n / 2), x(k - (n / 2) + 1), \dots, x(k + (n / 2) - 1)$ . In this case, `medfilt1` sorts the numbers, then takes the average of the  $n / 2$  and  $(n / 2) + 1$  elements.

`y = medfilt1(x,n,blksz)` computes `blksz` (block size) output samples at a time. By default, `blksz = length(x)`.

`y = medfilt1(x,n,blksz,dim)` specifies the dimension, `dim`, along which the filter operates.

By default, `medfilt1` operates along the first nonsingleton dimension of  $x$ . In particular, if  $x$  is a matrix, the function median filters its columns:  $y(:,i) = \text{medfilt1}(x(:,i),n,\text{blksz})$ .

## Input Arguments

### **x** — Input signal

vector | matrix | N-D array

Input signal, specified as a real-valued vector, matrix, or N-D array.

Data Types: `double`

**n — Filter order**

3 (default) | positive integer scalar

Order of the one-dimensional median filter, specified as a positive integer scalar.

Data Types: `double`

**blksize — Block size**

positive integer scalar

Number of output samples computed at a time, specified as a positive integer scalar. The default is `length(x)` for vectors and `size(x, dim)` for arrays of higher dimension.

Data Types: `double`

**dim — Dimension to filter along**

positive integer scalar

Dimension to filter along, specified as a positive integer scalar. If no value is specified, the default is the first nonsingleton dimension.

Data Types: `double`

## Output Arguments

**y — Filtered signal**

vector | matrix | N-D array

Filtered signal, returned as a real-valued vector, matrix, or N-D array. `y` is the same size as `x`

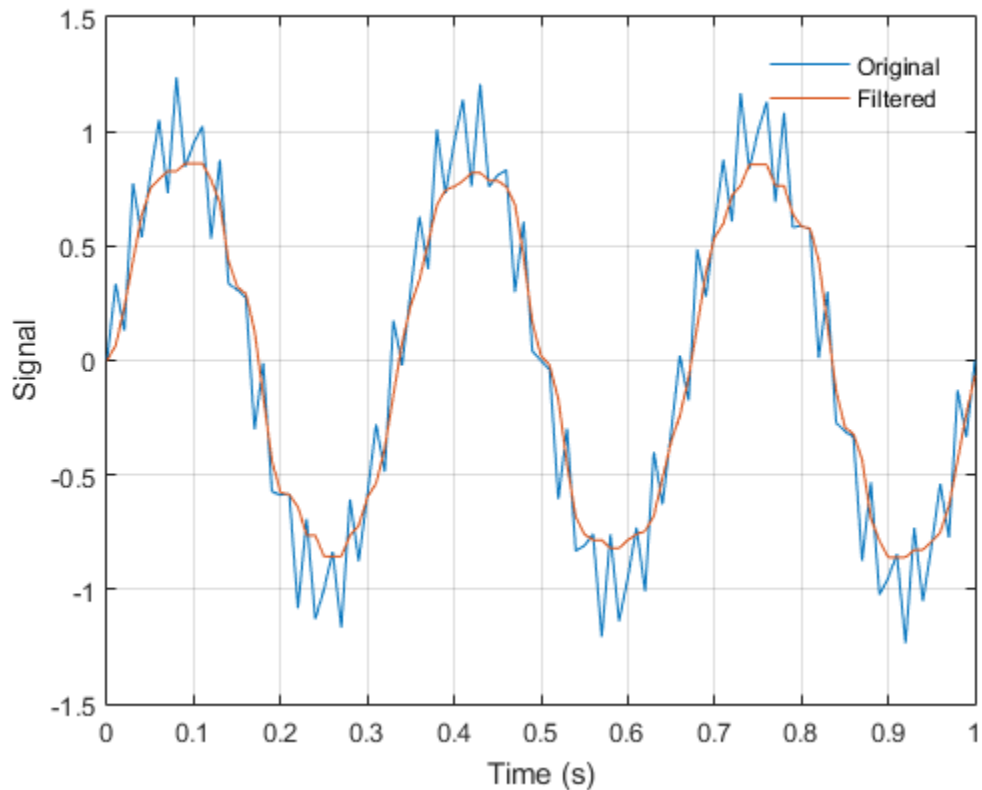
Data Types: `double`

## Examples

**Noise Suppression via Median Filtering**

Generate a sinusoidal signal sampled for 1 second at 100 Hz. Add a higher-frequency sinusoid to simulate noise. Use a 10th-order median filter to smooth the signal. Plot the result.

```
fs = 100;
t = 0:1/fs:1;
x = sin(2*pi*t*3)+0.25*sin(2*pi*t*40);
y = medfilt1(x,10);
figure,clf
plot(t,x,t,y),grid on
xlabel 'Time (s)',ylabel Signal
legend('Original','Filtered')
legend boxoff
```



## More About

### Tips

- If you have a license for Image Processing Toolbox™ software, you can use the function `medfilt2` to perform two-dimensional median filtering.

### References

- [1] Pratt, William K. *Digital Image Processing*. 4th ed. Hoboken, NJ: John Wiley & Sons, 2007, p.277.

## See Also

`filter` | `median` | `sgolayfilt`

## medfreq

Median frequency

### Syntax

```
freq = medfreq(x)
freq = medfreq(x, fs)

freq = medfreq(pxx, f)
freq = medfreq(sxx, f, rbw)

freq = medfreq( ____, freqrange)

[ freq, power ] = medfreq( ____ )

medfreq( ____ )
```

### Description

`freq = medfreq(x)` estimates the median normalized frequency, `freq`, of the power spectrum of a time-domain signal, `x`.

`freq = medfreq(x, fs)` estimates the median frequency in terms of the sample rate, `fs`.

`freq = medfreq(pxx, f)` returns the median frequency of a power spectral density (PSD) estimate, `pxx`. The frequencies, `f`, correspond to the estimates in `pxx`.

`freq = medfreq(sxx, f, rbw)` returns the median frequency of a power spectrum estimate, `sxx`, with resolution bandwidth `rbw`.

`freq = medfreq( ____, freqrange)` specifies the frequency interval over which to compute the median frequency, using any of the input arguments from previous syntaxes. The default value for `freqrange` is the entire bandwidth of the input signal.

`[ freq, power ] = medfreq( ____ )` also returns the band power, `power`, of the spectrum. If you specify `freqrange`, then `power` contains the band power within `freqrange`.



`medfreq( ___ )` with no output arguments plots the PSD or power spectrum and annotates the median frequency.

## Examples

### Median Frequency of Chirps

Generate 1024 samples of a chirp sampled at 1024 kHz. Specify the chirp so that it has an initial frequency of 50 kHz and reaches 100 kHz at the end of the sampling. Add white Gaussian noise such that the signal-to-noise ratio is 40 dB. Reset the random number generator for reproducible results.

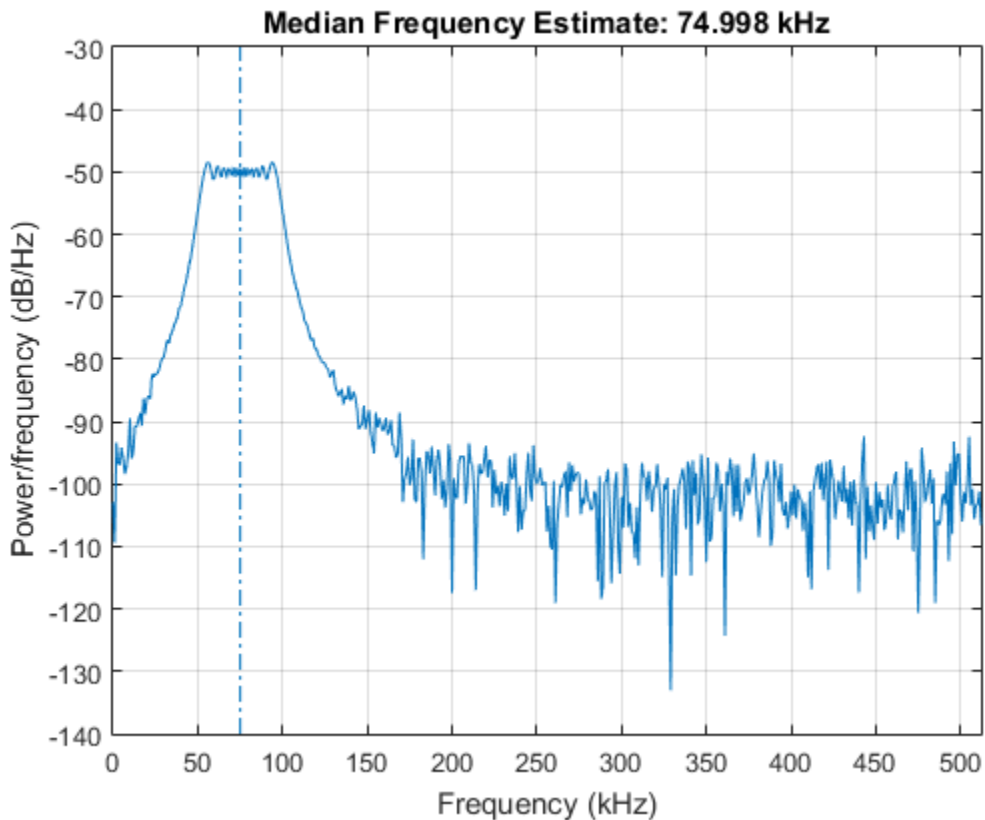
```
nSamp = 1024;  
Fs = 1024e3;  
SNR = 40;  
rng default  
  
t = (0:nSamp-1)'/Fs;  
  
x = chirp(t,50e3,nSamp/Fs,100e3);  
x = x+randn(size(x))*std(x)/db2mag(SNR);
```

Estimate the median frequency of the chirp. Plot the power spectral density (PSD) and annotate the median frequency.

```
medfreq(x,Fs)
```

```
ans =
```

```
7.4998e+04
```



Generate another chirp. Specify an initial frequency of 200 kHz, a final frequency of 300 kHz, and an amplitude that is twice that of the first signal. Add white Gaussian noise.

```
x2 = 2*chirp(t,200e3,nSamp/Fs,300e3);
x2 = x2+randn(size(x2))*std(x2)/db2mag(SNR);
```

Concatenate the chirps to produce a two-channel signal. Estimate the median frequency of each channel.

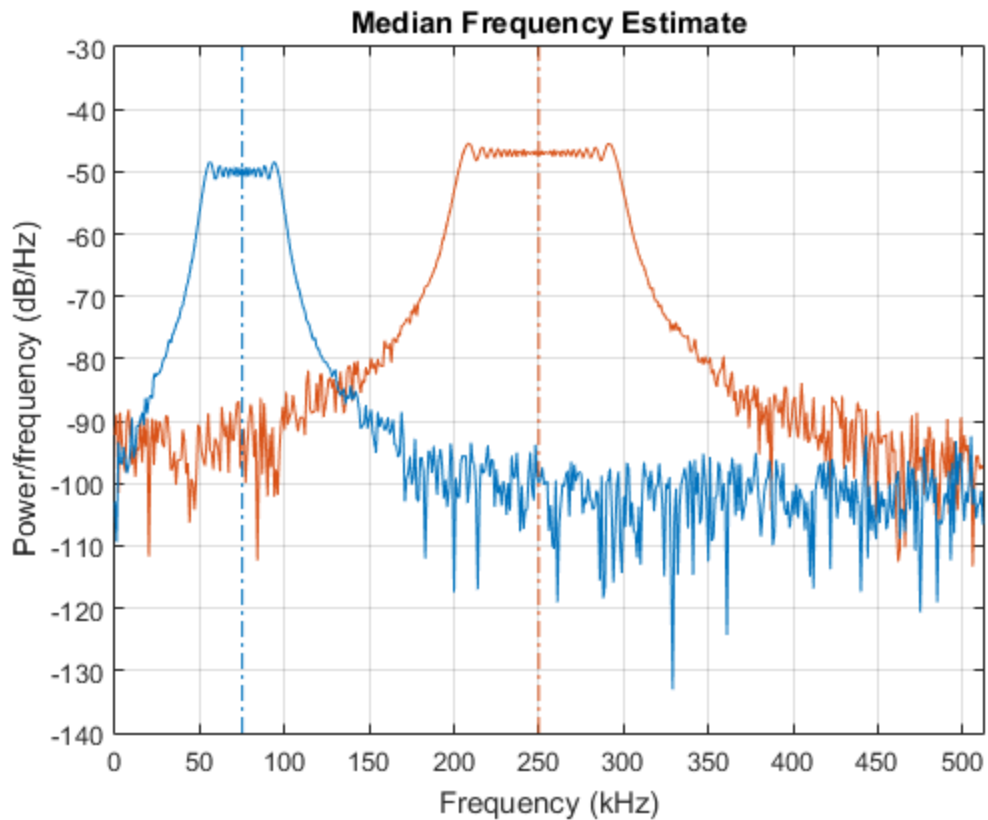
```
y = medfreq([x x2],Fs)
```

```
y =
```

```
1.0e+05 *
0.7500  2.4999
```

Plot the PSDs of the two channels and annotate their median frequencies.

```
medfreq([x x2],Fs);
```

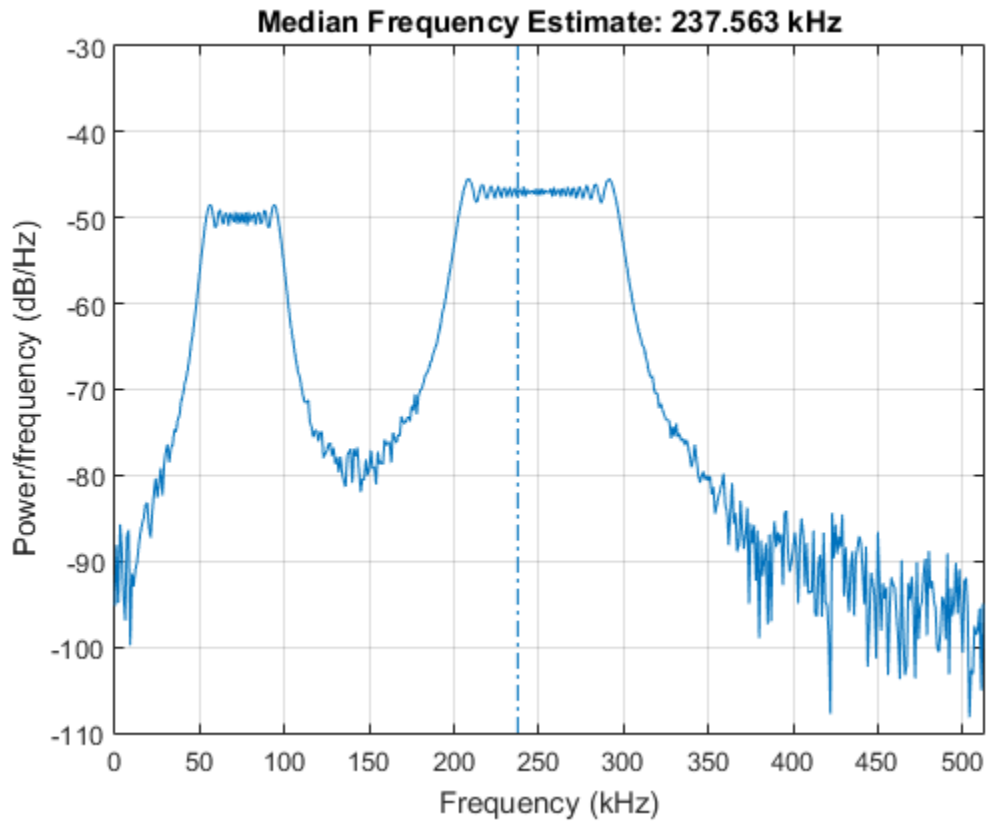


Add the two channels to form a new signal. Plot the PSD and annotate the median frequency.

```
medfreq(x+x2,Fs)
```

ans =

2.3756e+05



### Median Frequency of Sinusoids

Generate 1024 samples of a 100.123 kHz sinusoid sampled at 1024 kHz. Add white Gaussian noise such that the signal-to-noise ratio is 40 dB. Reset the random number generator for reproducible results.

```
nSamp = 1024;
Fs = 1024e3;
SNR = 40;
```

```
rng default
```

```
t = (0:nSamp-1)'/Fs;
```

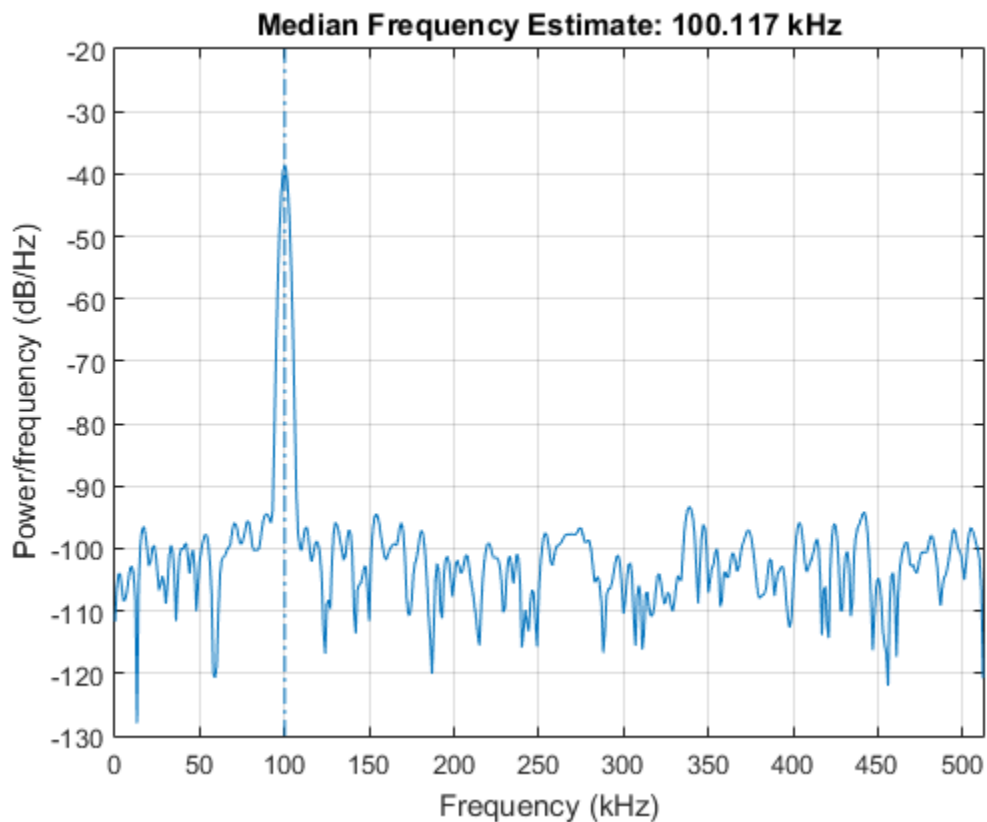
```
x = sin(2*pi*t*100.123e3);
```

```
x = x + randn(size(x))*std(x)/db2mag(SNR);
```

Use the `periodogram` function to compute the power spectral density (PSD) of the signal. Specify a Kaiser window with the same length as the signal and a shape factor of 38. Estimate the median frequency of the signal and annotate it on a plot of the PSD.

```
[Pxx,f] = periodogram(x,kaiser(nSamp,38),[],Fs);
```

```
medfreq(Pxx,f);
```



Generate another sinusoid, this one with a frequency of 257.321 kHz and an amplitude that is twice that of the first sinusoid. Add white noise.

```
x2 = 2*sin(2*pi*t*257.321e3);  
x2 = x2 + randn(size(x2))*std(x2)/db2mag(SNR);
```

Concatenate the sinusoids to produce a two-channel signal. Estimate the PSD of each channel and use the result to determine the median frequency.

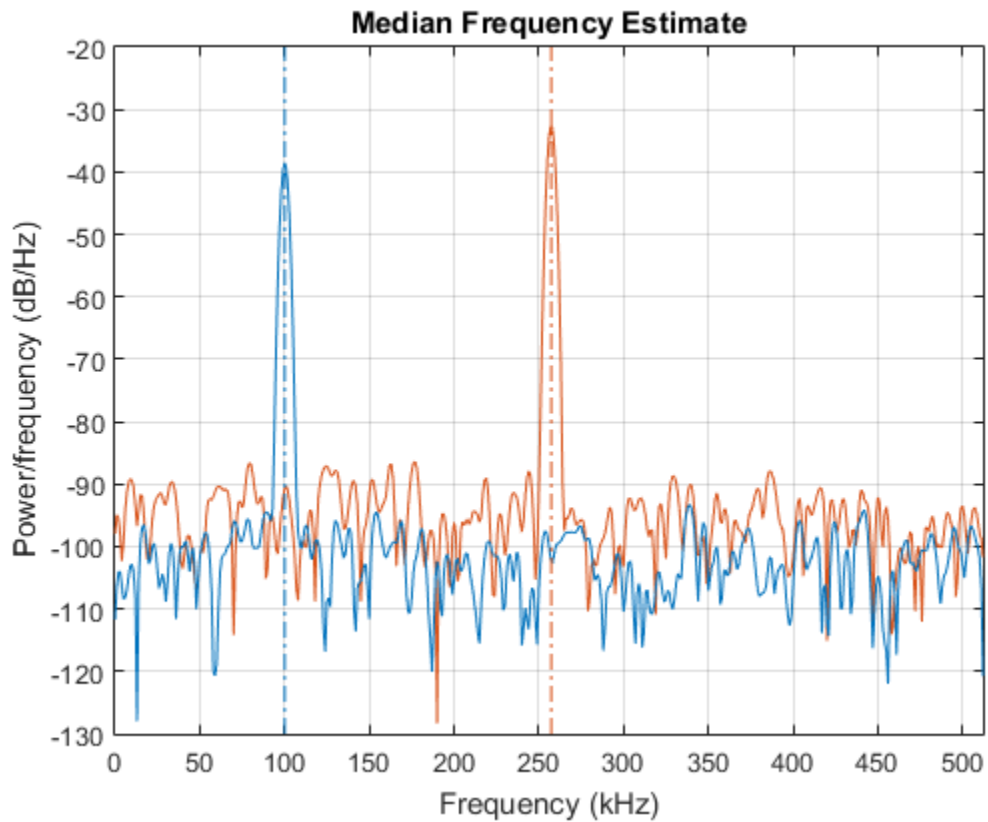
```
[Pyy,f] = periodogram([x x2],kaiser(nSamp,38),[],Fs);  
y = medfreq(Pyy,f)
```

```
y =
```

```
1.0e+05 *  
1.0012    2.5731
```

Annotate the median frequencies of the two channels on a plot of the PSDs.

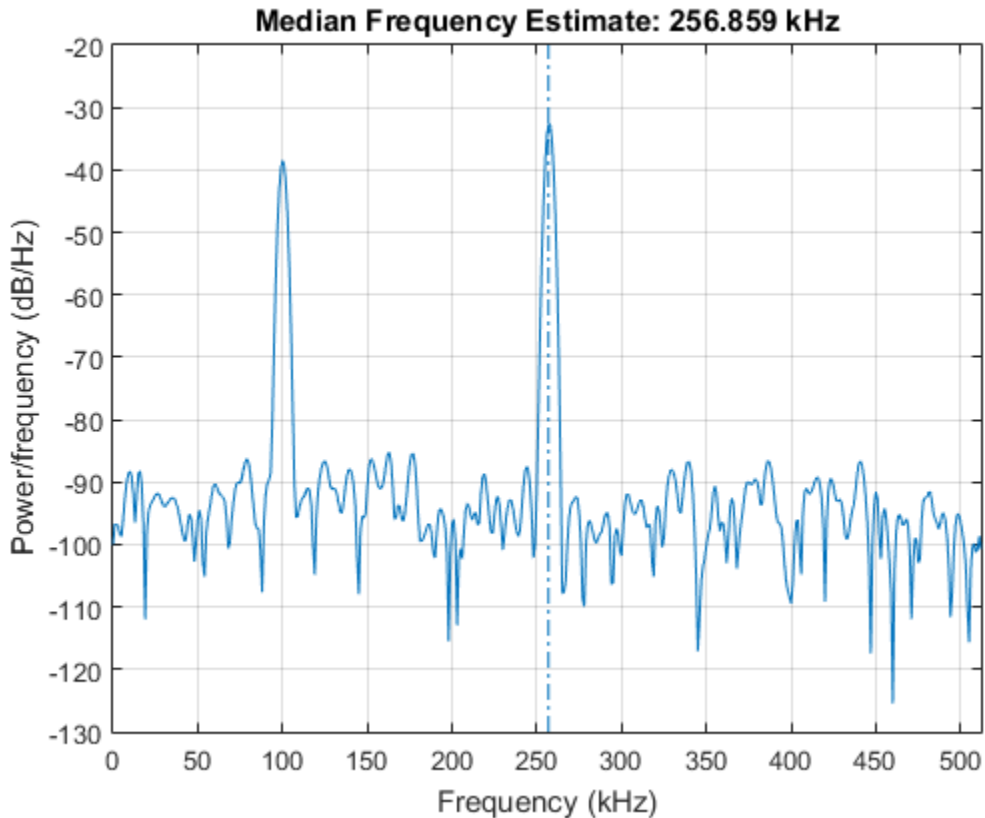
```
medfreq(Pyy,f);
```



Add the two channels to form a new signal. Estimate the PSD and annotate the median frequency.

```
[Pzz,f] = periodogram(x+x2,kaiser(nSamp,38),[],Fs);
```

```
medfreq(Pzz,f);
```



### Median Frequency of Bandlimited Signals

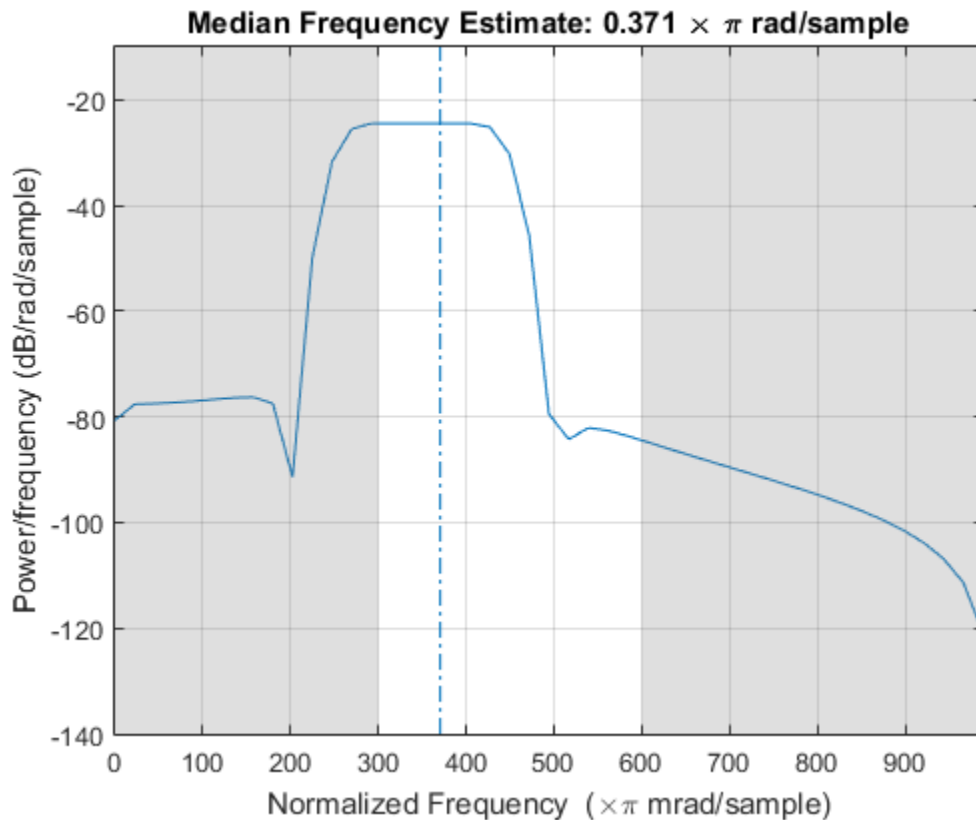
Generate a signal whose PSD resembles the frequency response of an 88th-order bandpass FIR filter with normalized cutoff frequencies  $0.25\pi$  rad/sample and  $0.45\pi$  rad/sample.

```
d = fir1(88,[0.25 0.45]);
```

Compute the median frequency of the signal between  $0.3\pi$  rad/sample and  $0.6\pi$  rad/sample. Plot the PSD and annotate the median frequency and measurement interval.

```
medfreq(d,[],[0.3 0.6]*pi);
```





Output the median frequency and the band power of the measurement interval. Specifying a sample rate of  $2\pi$  is equivalent to leaving the rate unset.

```
[mdf,power] = medfreq(d,2*pi,[0.3 0.6]*pi);
fprintf('Mean = %.3f*pi, power = %.1f%% of total \n', ...
        mdf/pi,power/bandpower(d)*100)
```

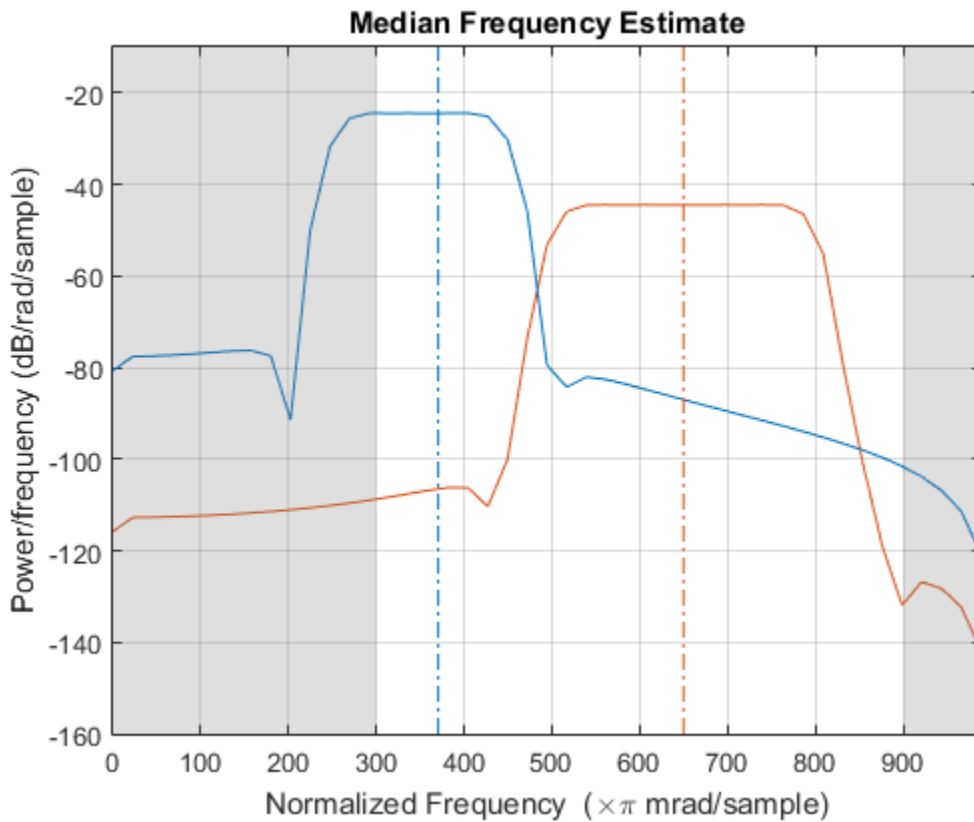
```
Mean = 0.371*pi, power = 77.4% of total
```

Add a second channel with normalized cutoff frequencies  $0.5\pi$  rad/sample and  $0.8\pi$  rad/sample and an amplitude that is one-tenth that of the first channel.

```
d = [d;fir1(88,[0.5 0.8])/10]';
```

Compute the median frequency of the signal between  $0.3\pi$  rad/sample and  $0.9\pi$  rad/sample. Plot the PSD and annotate the median frequency of each channel and the measurement interval.

```
medfreq(d,[],[0.3 0.9]*pi);
```



Output the median frequency of each channel. Divide by  $\pi$ .

```
mdf = medfreq(d,[],[0.3 0.9]*pi)/pi
```

```
mdf =
```

0.3706    0.6500

## Input Arguments

### **x** — Input signal

vector | matrix

Input signal, specified as a vector or matrix. If **x** is a vector, it is treated as a single channel. If **x** is a matrix, then `medfreq` computes the median frequency of each column of **x** independently. **x** must be finite-valued.

Example: `cos(pi/4*(0:159))+randn(1,160)` is a single-channel row-vector signal.

Example: `cos(pi./[4;2]*(0:159))'+randn(160,2)` is a two-channel signal.

Data Types: `single` | `double`

### **fs** — Sample rate

positive real scalar

Sample rate, specified as a positive real scalar. The sample rate is the number of samples per unit time. If the time is measured in seconds, then the sample rate is in hertz.

Data Types: `single` | `double`

### **pxx** — Power spectral density

vector | matrix

Power spectral density (PSD), specified as a vector or matrix. If **pxx** is a matrix, then `medfreq` computes the median frequency of each column of **pxx** independently.

Example: `periodogram(cos(pi./[4;2]*(0:159))'+randn(160,2))` is the periodogram PSD estimate of a two-channel sinusoid embedded in white Gaussian noise.

Data Types: `single` | `double`

### **f** — Frequencies

vector

Frequencies, specified as a vector.

Data Types: `single` | `double`

**sxx — Power spectrum estimate**

vector | matrix

Power spectrum estimate, specified as a vector or matrix. If `sxx` is a matrix, then `medfreq` computes the median frequency of each column of `sxx` independently.

Example: `periodogram(cos(pi./[4;2]*(0:159))'+randn(160,2),'power')` is the periodogram power spectrum estimate of a two-channel sinusoid embedded in white Gaussian noise.

Data Types: `single` | `double`**rbw — Resolution bandwidth**

positive scalar

Resolution bandwidth, specified as a positive scalar. The resolution bandwidth is the product of two values: the frequency resolution of the discrete Fourier transform and the equivalent noise bandwidth of the window used to compute the PSD.

Data Types: `single` | `double`**freorange — Frequency range**

two-element vector

Frequency range, specified as a two-element vector of real values. If you do not specify `freorange`, then `medfreq` uses the entire bandwidth of the input signal.

Data Types: `single` | `double`

## Output Arguments

**freq — Median frequency**

scalar | vector

Median frequency, specified as a scalar or vector.

- If you specify a sample rate, then `freq` has the same units as `fs`.
- If you do not specify a sample rate, then `freq` has units of rad/sample.

**power — Band power**

scalar | vector

Band power, returned as a scalar or vector.

## **See Also**

`findpeaks` | `meanfreq` | `periodogram` | `plomb` | `pwelch`

**Introduced in R2015a**

## midcross

Mid-reference level crossing for bilevel waveform

### Syntax

```
C = midcross(X)
C = midcross(X,FS)
C = midcross(X,T)
[C,MIDLEV] = midcross(...)
C = midcross(X,Name,Value)
midcross(...)
```

### Description

`C = midcross(X)` returns a vector, `C`, of time instants where each transition of the input signal, `X`, crosses the 50% reference level. The sample instants correspond to the indices of the input vector. Because `midcross` uses interpolation to determine the crossing instant, `C` may contain values that do not correspond to sampling instants. To determine the transitions, `midcross` estimates the state levels of `X` by a histogram method. `midcross` identifies all intervals which cross the upper-state boundary of the low state and the lower-state boundary of the high state. The low-state and high-state boundaries are expressed as the state level plus or minus a multiple of the difference between the state levels. See “State-Level Tolerances” on page 1-957.

`C = midcross(X,FS)` specifies the sample rate, `FS`, in hertz as a positive scalar. The first sample instant corresponds to `t=0`. Because `midcross` uses interpolation to determine the crossing instant, `C` may contain values that do not correspond to sampling instants.

`C = midcross(X,T)` specifies the sample instants, `T`, as a vector with the same number of elements as `X`. Because `midcross` uses interpolation to determine the crossing instant, `C` may contain values that do not correspond to sampling instants.

`[C,MIDLEV] = midcross(...)` returns the waveform value corresponding to the mid-reference level.

`C = midcross(X,Name,Value)` returns the time instants corresponding to mid-reference level crossings with additional options specified by one or more Name,Value pair arguments.

`midcross(...)` plots the signal and marks the location of the mid-crossings (mid-reference level instants) and the associated reference levels. `midcross` also plots the state levels with upper and lower state boundaries.

## Input Arguments

### **X**

Bilevel waveform. X is a real-valued row or column vector.

### **FS**

Sample rate in hertz.

### **T**

Vector of sample instants. The length of T must equal the length of the bilevel waveform, X.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

### **'MidPercentReferenceLevel'**

Mid-reference level as a percentage of the waveform amplitude.

**Default:** 50

### **'StateLevels'**

Low and high state levels. StateLevels is a 1-by-2 real-valued vector. The first element is the low state level. The second element is the high state level. If you do not specify

low- and high-state levels, `midcross` estimates the state levels from the input waveform using the histogram method.

### 'Tolerance'

Tolerance levels (lower- and upper-state boundaries) expressed as a percentage. See “State-Level Tolerances” on page 1-957.

**Default:** 2

## Output Arguments

### **C**

Time instants of the mid-reference level crossings.

### **MIDLEV**

Mid-reference level.

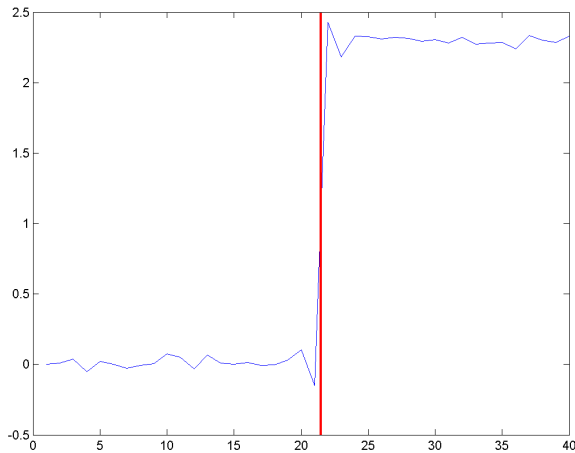
## Examples

### **Mid-Reference Level Instant of Bilevel Waveform**

Assuming a sampling interval of 1, compute the mid-reference level instant of a bilevel waveform and plot the result.

```
load('transitionex.mat', 'x');  
C = midcross(x);  
plot(x); hold on;  
plot([C C],[-0.5 2.5], 'r', 'linewidth', 2);
```





The instant at which the waveform crosses the 50% reference level is 21.5. Note that this is not a sampling instant present in the input vector because `midcross` uses interpolation to identify the mid-reference level crossing.

### Mid-Reference Level Instant with Sampling Frequency

Compute the mid-reference level instant using the sampling rate for a bilevel waveform sampled at 4 MHz.

```
load('transitionex.mat','x','t');
Fs = 1/(t(2)-t(1));
C = midcross(x,Fs);
```

### Mid Reference Level Instant Using Sample Instants

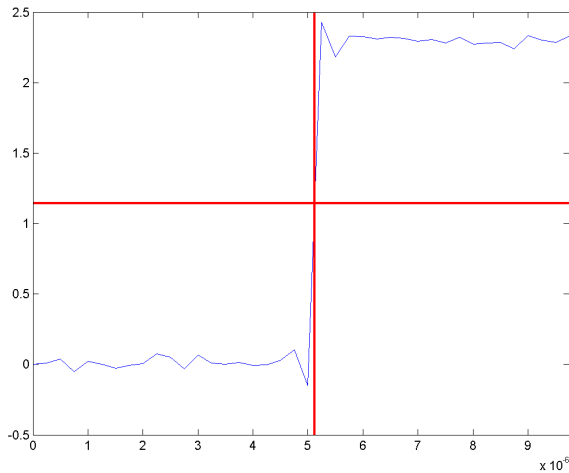
Compute the mid-reference level instants using a vector of sample times equal in length to the bilevel waveform. The sampling rate is 4 MHz.

```
load('transitionex.mat','x','t');
C = midcross(x,t);
```

### Mid-Reference Level Value of Bilevel Waveform

Compute the level corresponding to the mid-reference level instant. Plot the result.

```
load('transitionex.mat','x','t');
[C,MIDLEV] = midcross(x,t);
plot(t,x); hold on;
plot([C C],[-0.5 2.5],'r','linewidth',2);
plot([0 t(end)],[MIDLEV MIDLEV],'r','linewidth',2);
axis tight;
```



### 60% Reference Level Instant and Waveform Value

Obtain the 60% reference level instant and value for a bilevel waveform.

```
load('transitionex.mat','x','t');
[C,Lev60] = midcross(x,t,'MidPercentReferenceLevel',60);
```

## More About

### Mid-Reference Level

The mid-reference level in a bilevel waveform with low-state level,  $S_1$ , and high-state level,  $S_2$ , is

$$S_1 + \frac{1}{2}(S_2 - S_1)$$

### Mid Reference Level Instant

Let  $y_{50\%}$  denote the mid-reference level.

Let  $t_{50\%_-}$  and  $t_{50\%_+}$  denote the two consecutive sampling instants corresponding to the waveform values nearest in value to  $y_{50\%}$ .

Let  $y_{50\%_-}$  and  $y_{50\%_+}$  denote the waveform values at  $t_{50\%_-}$  and  $t_{50\%_+}$ .

The mid-reference level instant is

$$t_{50\%} = t_{50\%_-} + \left( \frac{t_{50\%_+} - t_{50\%_-}}{y_{50\%_+} - y_{50\%_-}} \right) (y_{50\%_+} - y_{50\%_-})$$

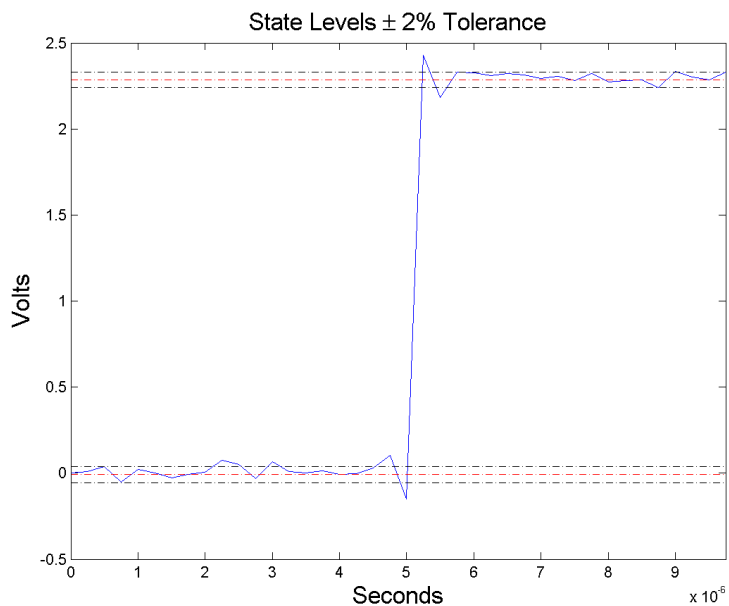
### State-Level Tolerances

Each state level can have associated lower- and upper-state boundaries. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference between the high state and low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the  $\alpha\%$  tolerance region for the low state is defined as

$$S_1 \pm \frac{\alpha}{100} (S_2 - S_1)$$

where  $S_1$  is the low-state level and  $S_2$  is the high-state level. Replace the first term in the equation with  $S_2$  to obtain the  $\alpha\%$  tolerance region for the high state.

The following figure illustrates lower and upper 2% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The red dashed lines indicate the estimated state levels.



## References

- [1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003. p. 20.

## See Also

`falltime` | `pulsewidth` | `risettime` | `settlingtime` | `statelevels`

# modulate

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`.

---

**Note:** Use `modulate` and `demod` in the Signal Processing Toolbox with real-valued signals to obtain real-valued outputs. `modulate` and `demod` are not intended to accept complex-valued inputs or produce complex-valued outputs.

---

| Method               | Description  |
|----------------------|--|
| amdsb-sc<br>or<br>am | Amplitude modulation, double sideband, suppressed carrier. Multiplies <code>x</code> by a sinusoid of frequency <code>fc</code> .<br>$y = x.\cos(2\pi fc t)$   |
| amdsb-tc             | Amplitude modulation, double sideband, transmitted carrier. Subtracts scalar <code>opt</code> from <code>x</code> and multiplies the result by a sinusoid of frequency <code>fc</code> .<br>$y = (x - \text{opt}).\cos(2\pi fc t)$ |

| Method             | Description  |
|--------------------|--|
|                    | <p>If the <code>opt</code> parameter is not present, <code>modulate</code> uses a default of <code>min(min(x))</code> so that the message signal (<code>x-opt</code>) is entirely nonnegative and has a minimum value of 0.</p>  |
| <code>amssb</code> | <p>Amplitude modulation, single sideband. Multiplies <code>x</code> by a sinusoid of frequency <code>fc</code> and adds the result to the Hilbert transform of <code>x</code> multiplied by a phase shifted sinusoid of frequency <code>fc</code>.</p> $y = x.\cos(2\pi*fc*t) + \text{imag}(\text{hilbert}(x)).\sin(2\pi*fc*t)$ <p>This effectively removes the upper sideband.</p>  |
| <code>fm</code>    | <p>Frequency modulation. Creates a sinusoid with instantaneous frequency that varies with the message signal <code>x</code>.</p> $y = \cos(2\pi*fc*t + \text{opt}*\text{cumsum}(x))$ <p><code>cumsum</code> is a rectangular approximation to the integral of <code>x</code>. <code>modulate</code> uses <code>opt</code> as the constant of frequency modulation. If <code>opt</code> is not present, <code>modulate</code> uses a default of</p> $\text{opt} = (fc/fs)*2\pi/(\max(\max(x)))$ <p>so the maximum frequency excursion from <code>fc</code> is <code>fc</code> Hz.</p> |
| <code>pm</code>    | <p>Phase modulation. Creates a sinusoid of frequency <code>fc</code> whose phase varies with the message signal <code>x</code>.</p> $y = \cos(2\pi*fc*t + \text{opt}*x)$ <p><code>modulate</code> uses <code>opt</code> as the constant of phase modulation. If <code>opt</code> is not present, <code>modulate</code> uses a default of</p> $\text{opt} = \pi/(\max(\max(x)))$ <p>so the maximum phase excursion is <math>\pi</math> radians.</p>   |

| Method | Description  |
|--------|--|
| pwm    | <p>Pulse-width modulation. Creates a pulse-width modulated signal from the pulse widths in <math>x</math>. The elements of <math>x</math> 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.</p> <p><code>modulate(x,fc,fs,'pwm','centered')</code></p> <p>yields pulses centered at the beginning of each period. <math>y</math> is length <code>length(x)*fs/fc</code>.</p> |
| ppm    | <p>Pulse-position modulation. Creates a pulse-position modulated signal from the pulse positions in <math>x</math>. The elements of <math>x</math> must be between 0 and 1, specifying the left edge of each pulse in fractions of a period. <code>opt</code> is a scalar between 0 and 1 that specifies the length of each pulse in fractions of a period. The default for <code>opt</code> is 0.1. <math>y</math> is length <code>length(x)*fs/fc</code>.</p>                                      |
| qam    | <p>Quadrature amplitude modulation. Creates a quadrature amplitude modulated signal from signals <math>x</math> and <code>opt</code>.</p> <p><math>y = x.\cos(2\pi\text{fc}t) + \text{opt}.\sin(2\pi\text{fc}t)</math></p> <p><code>opt</code> must be the same size as <math>x</math>.</p>  |

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

# mscohere

Magnitude squared coherence

## Syntax

```
Cxy = mscohere(x,y)
Cxy = mscohere(x,y>window)
Cxy = mscohere(x,y>window,noverlap)
[Cxy,W] = mscohere(x,y>window,noverlap,nfft)
[Cxy,F] = mscohere(x,y>window,noverlap,nfft,fs)
[Cxy,F] = mscohere(x,y>window,noverlap,f,fs)
[...] = mscohere(x,y,...,'twosided')
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 input signals may be either vectors or two-dimensional matrices. If both are vectors, they must have the same length. If both are matrices, they must have the same size, and `mscohere` operates columnwise: `Cxy(:,n) = mscohere(x(:,n),y(:,n))`. If one is a matrix and the other is a vector, then the vector is converted to a column vector and internally expanded so both inputs have the same number of columns.

For real `x` and `y`, `mscohere` returns a one-sided coherence estimate. For complex `x` or `y`, it returns a two-sided estimate.

`mscohere` uses the following default values:

| Parameter         | Description   | Default Value   |
|-------------------|---|---|
| <code>nfft</code> | FFT length which determines the frequencies at which the coherence is estimated | Maximum of 256 or the next power of 2 greater than the length of each section of <code>x</code> or <code>y</code> |



| Parameter             | Description  | Default Value  |
|-----------------------|--|--|
|                       | For real $x$ and $y$ , the length of $C_{xy}$ is $(nfft/2 + 1)$ if $nfft$ is even or $(nfft + 1)/2$ if $nfft$ is odd. For complex $x$ or $y$ , the length of $C_{xy}$ is $nfft$ .<br><br>If $nfft$ is greater than the signal length, the data is zero-padded. If $nfft$ is less than the signal length, the data segment is wrapped so that the length is equal to $nfft$ . |  |
| <code>fs</code>       | Sampling frequency   | 1  |
| <code>window</code>   | Windowing function and number of samples to use for each section   | Periodic Hamming window of sufficient length to obtain eight equal sections of $x$ and $y$ |
| <code>noverlap</code> | 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 = mscohere(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`, then  $C_{xy}$  uses a Hamming window of that length.

`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`.

`[Cxy,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  $y$ ,  $C_{xy}$  length is  $(nfft/2 + 1)$  if  $nfft$  is even; if  $nfft$  is odd, the length is  $(nfft + 1)/2$ . For complex  $x$  or  $y$ , the length of  $C_{xy}$  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  $[0, fs)$ .

`[Cxy, F] = mscohere(x, y, window, noverlap, f, fs)` computes the coherence estimate at the frequencies `f`. `f` is a vector containing two or more elements.

`[...] = mscohere(x, y, ..., 'twosided')` returns a coherence estimate with frequencies that range over the whole Nyquist interval. Specifying `'onesided'` uses half the Nyquist interval.

`mscohere(...)` plots the magnitude squared coherence versus frequency in the current figure window.

---

**Note** If you estimate the magnitude squared coherence with a single window, or section, the value is identically 1 for all frequencies [1]. You must use at least two sections.

---

## Examples

### Coherence Estimate of Two Sequences

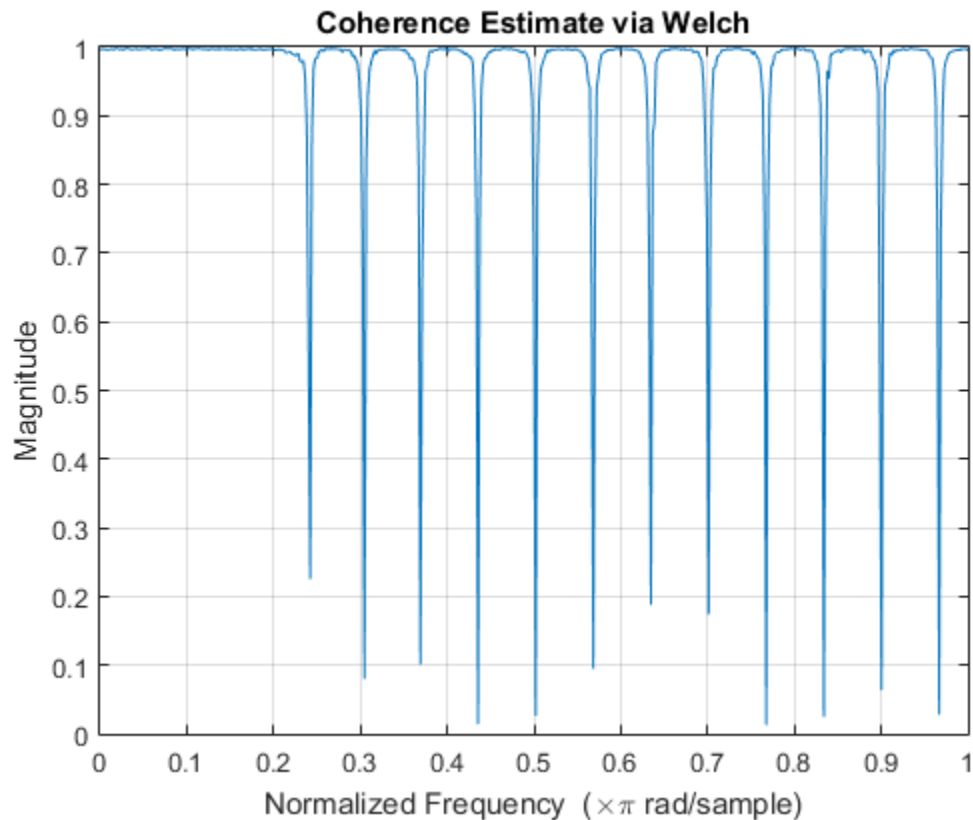
Compute and plot the coherence estimate between two colored noise sequences, `x` and `y`. Reset the random number generator for reproducible results. Use a Hann window to estimate the coherence.

```
rng default
r = randn(16384,1);

h1 = ones(1,10)/sqrt(10);
x = filter(h1,1,r);

h = fir1(30,0.2,rectwin(31));
y = filter(h,1,x);

mscohere(x,y,hanning(1024),512,1024)
```



## More About

### Magnitude Squared Coherence

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 magnitude squared coherence is a function of the power spectral densities,  $P_{xx}(f)$  and  $P_{yy}(f)$ , of  $x$  and  $y$ , and the cross power spectral density,  $P_{xy}(f)$ , of  $x$  and  $y$ :

$$C_{xy}(f) = \frac{|P_{xy}(f)|^2}{P_{xx}(f)P_{yy}(f)}.$$

### Algorithms

`mscohere` estimates the magnitude squared coherence function [2] using Welch's overlapped averaged periodogram method (see references [3] and [4]).

## References

- [1] Stoica, Petre, and Randolph Moses. *Spectral Analysis of Signals*. Upper Saddle River, NJ: Prentice Hall, 2005.
- [2] Kay, Steven M. *Modern Spectral Estimation*. Englewood Cliffs, NJ: Prentice-Hall, 1988.
- [3] Rabiner, Lawrence R., and Bernard Gold. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975.
- [4] Welch, Peter D. "The Use of Fast Fourier Transform for the Estimation of Power Spectra: A Method Based on Time Averaging Over Short, Modified Periodograms." *IEEE Transactions on Audio and Electroacoustics*. Vol. AU-15, 1967, pp. 70–73.

### See Also

`cpsd` | `periodogram` | `pwelch` | `spectrum` | `tffestimate`

# nuttallwin

Nuttall-defined minimum 4-term Blackman-Harris window

## Syntax

```
w = nuttallwin(N)
w = nuttalwin(N,SFLAG)
```

## Description

`w = nuttallwin(N)` returns a Nuttall defined  $N$ -point, 4-term symmetric 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.

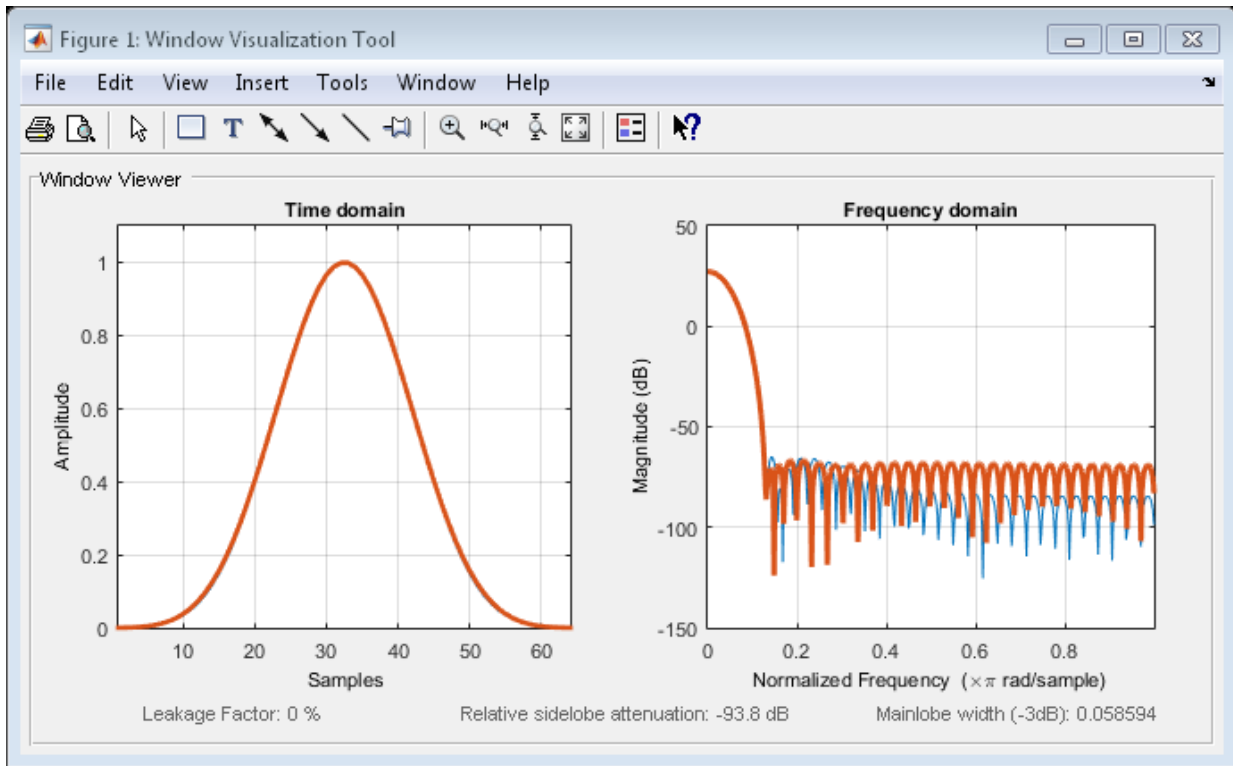
`w = nuttalwin(N,SFLAG)` uses `SFLAG` window sampling. `SFLAG` can be 'symmetric' or 'periodic'. The default is 'symmetric'. You can find the equations defining the symmetric and periodic windows in “Definitions” on page 1-968.

## Examples

### Nuttall and Blackman-Harris Windows

Compare 64-point Nuttall and Blackman-Harris windows. Plot them using `wvtool`.

```
L = 64;
w = blackmanharris(L);
y = nuttallwin(L);
wvtool(w,y)
```



Compute the maximum difference between the two windows.

```
max(abs(y-w))
```

```
ans =
```

```
0.0099
```

## Definitions

The equation for the **symmetric** Nuttall defined 4-term Blackman-Harris window is

$$w(n) = a_0 - a_1 \cos\left(2\pi \frac{n}{N-1}\right) + a_2 \cos\left(4\pi \frac{n}{N-1}\right) - a_3 \cos\left(6\pi \frac{n}{N-1}\right)$$

where  $n = 0, 1, 2, \dots, N-1$ .

The equation for the **periodic** Nuttall defined 4-term Blackman-Harris window 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  $n = 0, 1, 2, \dots, N-1$ . The periodic window is N-periodic.

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

## **obw**

Occupied bandwidth

### **Syntax**

`bw = obw(x)`

`bw = obw(x, fs)`

`bw = obw(pxx, f)`

`bw = obw(sxx, f, rbw)`

`bw = obw( ____, freqrange, p)`

`[bw, flo, fhi, power] = obw( ____, )`

`obw( ____, )`

### **Description**

`bw = obw(x)` returns the 99% occupied bandwidth, `bw`, of the input signal, `x`.

`bw = obw(x, fs)` returns the occupied bandwidth in terms of the sample rate, `fs`.

`bw = obw(pxx, f)` returns the 99% occupied bandwidth of the power spectral density (PSD) estimate, `pxx`. The frequencies, `f`, correspond to the estimates in `pxx`.

`bw = obw(sxx, f, rbw)` computes the occupied bandwidth of the power spectrum estimate, `sxx`. The frequencies, `f`, correspond to the estimates in `sxx`. `rbw` is the resolution bandwidth used to integrate each power estimate.

`bw = obw( ____, freqrange, p)` specifies the frequency interval over which to compute the occupied bandwidth, using any of the input arguments from previous syntaxes.

This syntax also specifies `p`, the percentage of the total signal power contained in the occupied band.



`[bw, flo, fhi, power] = obw( ___ )` also returns the lower and upper bounds of the occupied bandwidth and the occupied band power.

`obw( ___ )` with no output arguments plots the PSD or power spectrum in the current figure window and annotates the bandwidth.

## Examples

### Occupied Bandwidth of Chirps

Generate 1024 samples of a chirp sampled at 1024 kHz. The chirp has an initial frequency of 50 kHz and reaches 100 kHz at the end of the sampling. Add white Gaussian noise such that the signal-to-noise ratio is 40 dB. Reset the random number generator for reproducible results.

```
nSamp = 1024;
Fs = 1024e3;
SNR = 40;
rng default

t = (0:nSamp-1)'/Fs;

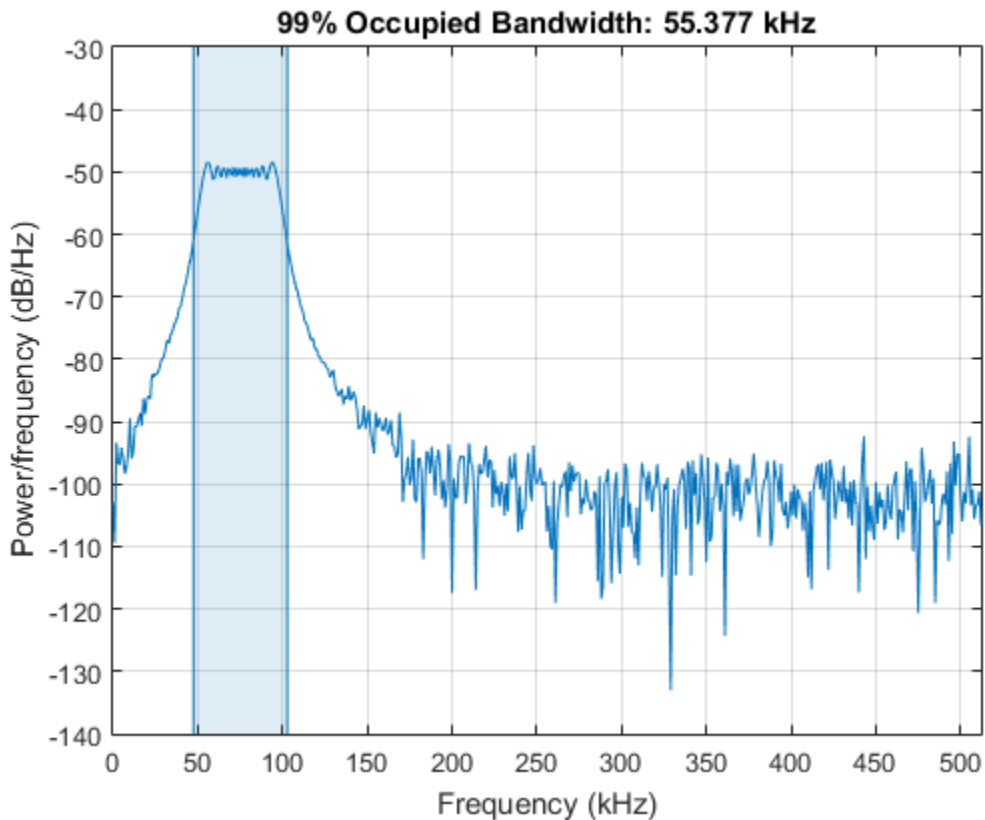
x = chirp(t,50e3,nSamp/Fs,100e3);
x = x+randn(size(x))*std(x)/db2mag(SNR);
```

Estimate the occupied bandwidth of the signal and annotate it on a plot of the power spectral density (PSD).

```
obw(x,Fs)
```

```
ans =
```

```
5.5377e+04
```



Generate another chirp. Specify an initial frequency of 200 kHz, a final frequency of 300 kHz, and an amplitude that is twice that of the first signal. Add white Gaussian noise.

```
x2 = 2*chirp(t,200e3,nSamp/Fs,300e3);
x2 = x2+randn(size(x2))*std(x2)/db2mag(SNR);
```

Concatenate the chirps to produce a two-channel signal. Estimate the occupied bandwidth of each channel.

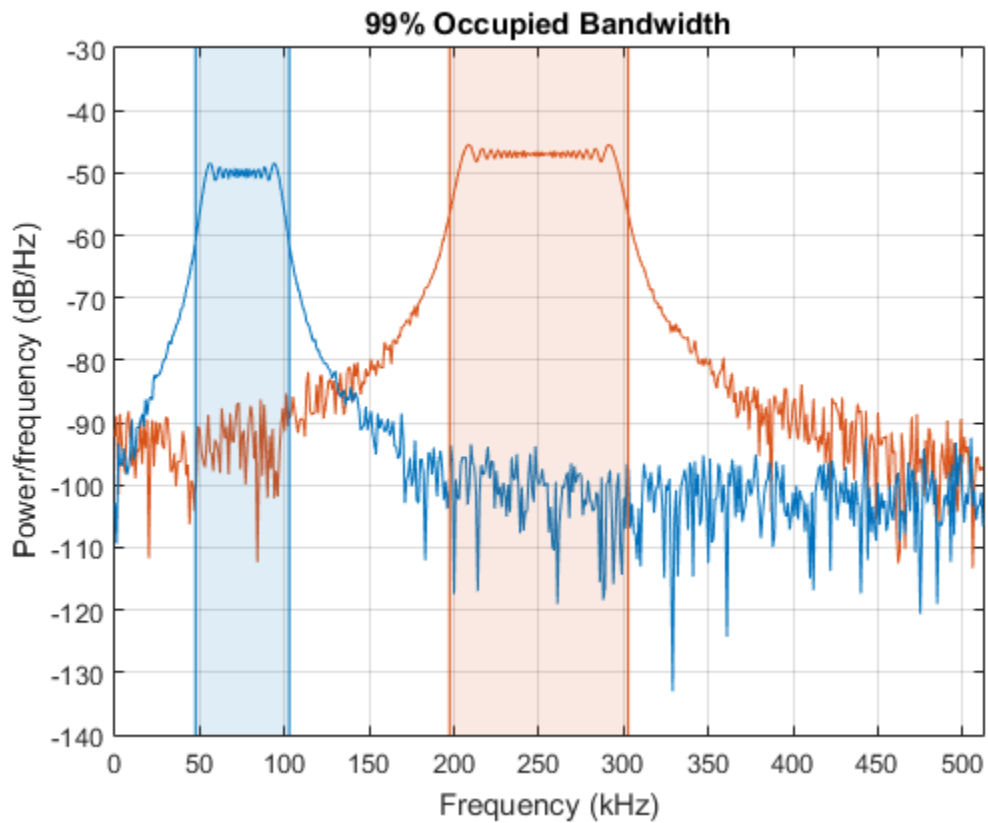
```
y = obw([x x2],Fs)
```

```
y =
```

```
1.0e+05 *
0.5538  1.0546
```

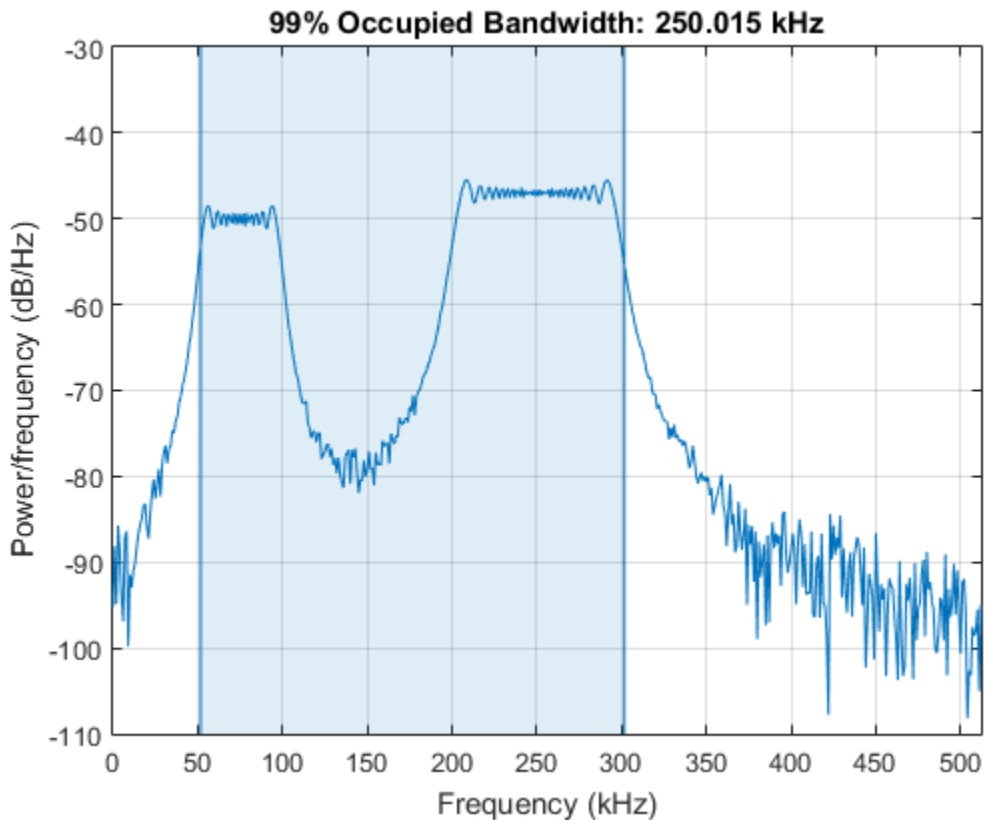
Annotate the occupied bandwidths of the two channels on a plot of the PSDs.

```
obw([x x2],Fs);
```



Add the two channels to form a new signal. Plot the PSD and annotate the occupied bandwidth.

```
obw(x+x2,Fs);
```



### Occupied Bandwidth of Sinusoids

Generate 1024 samples of a 100.123 kHz sinusoid sampled at 1024 kHz. Add white Gaussian noise such that the signal-to-noise ratio is 40 dB. Reset the random number generator for reproducible results.

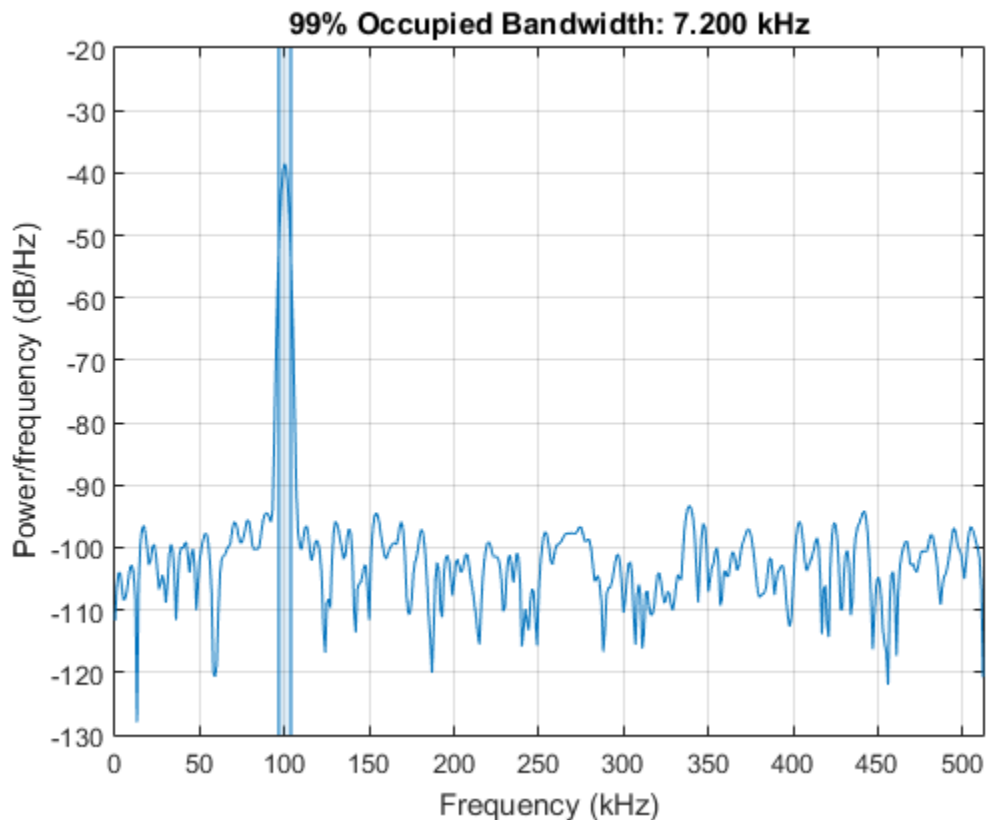
```
nSamp = 1024;  
Fs = 1024e3;  
SNR = 40;  
rng default  
  
t = (0:nSamp-1)'/Fs;  
  
x = sin(2*pi*t*100.123e3);
```

```
x = x + randn(size(x))*std(x)/db2mag(SNR);
```

Use the `periodogram` function to compute the power spectral density (PSD) of the signal. Specify a Kaiser window with the same length as the signal and a shape factor of 38. Estimate the occupied bandwidth of the signal and annotate it on a plot of the PSD.

```
[Pxx,f] = periodogram(x,kaiser(nSamp,38),[],Fs);
```

```
obw(Pxx,f);
```



Generate another sinusoid, this one with a frequency of 257.321 kHz and an amplitude that is twice that of the first sinusoid. Add white Gaussian noise.

```
x2 = 2*sin(2*pi*t*257.321e3);
```

```
x2 = x2 + randn(size(x2))*std(x2)/db2mag(SNR);
```

Concatenate the sinusoids to produce a two-channel signal. Estimate the PSD of each channel and use the result to determine the occupied bandwidth.

```
[Pyy,f] = periodogram([x x2],kaiser(nSamp,38),[],Fs);
```

```
y = obw(Pyy,f)
```

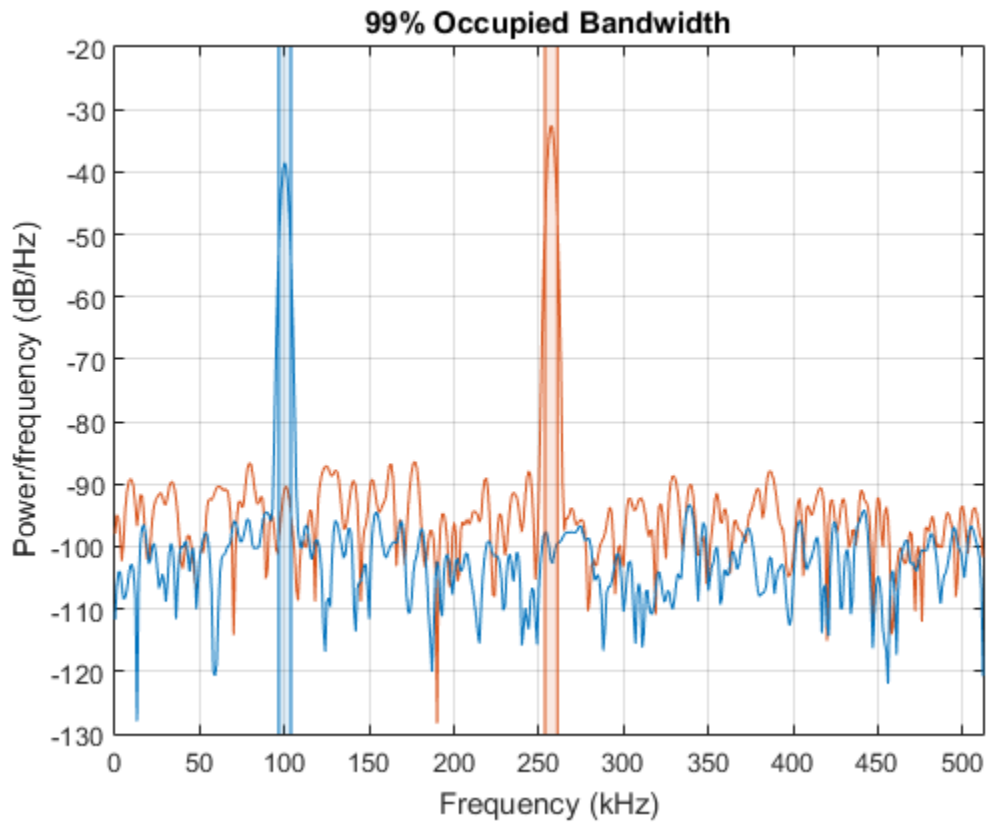
```
y =
```

```
1.0e+03 *
```

```
7.2001    7.3777
```

Annotate the occupied bandwidths of the two channels on a plot of the PSDs.

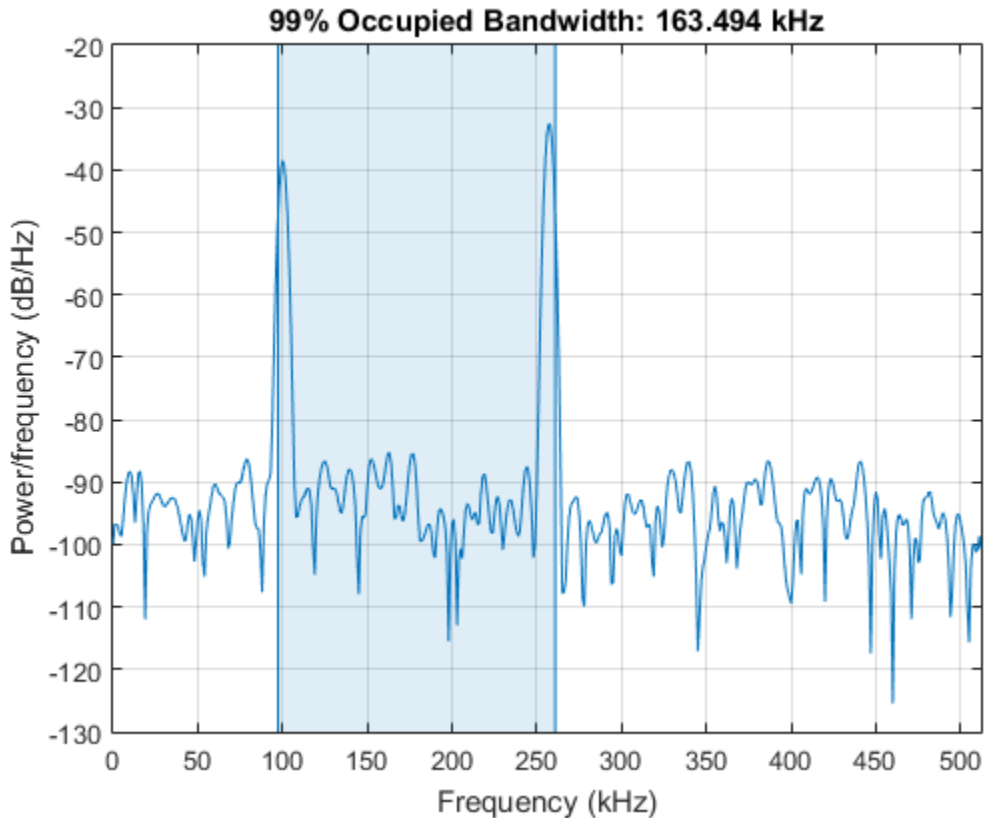
```
obw(Pyy,f);
```



Add the two channels to form a new signal. Estimate the PSD and annotate the occupied bandwidth.

```
[Pzz,f] = periodogram(x+x2,kaiser(nSamp,38),[],Fs);
```

```
obw(Pzz,f);
```



### Occupied Bandwidth of Bandlimited Signals

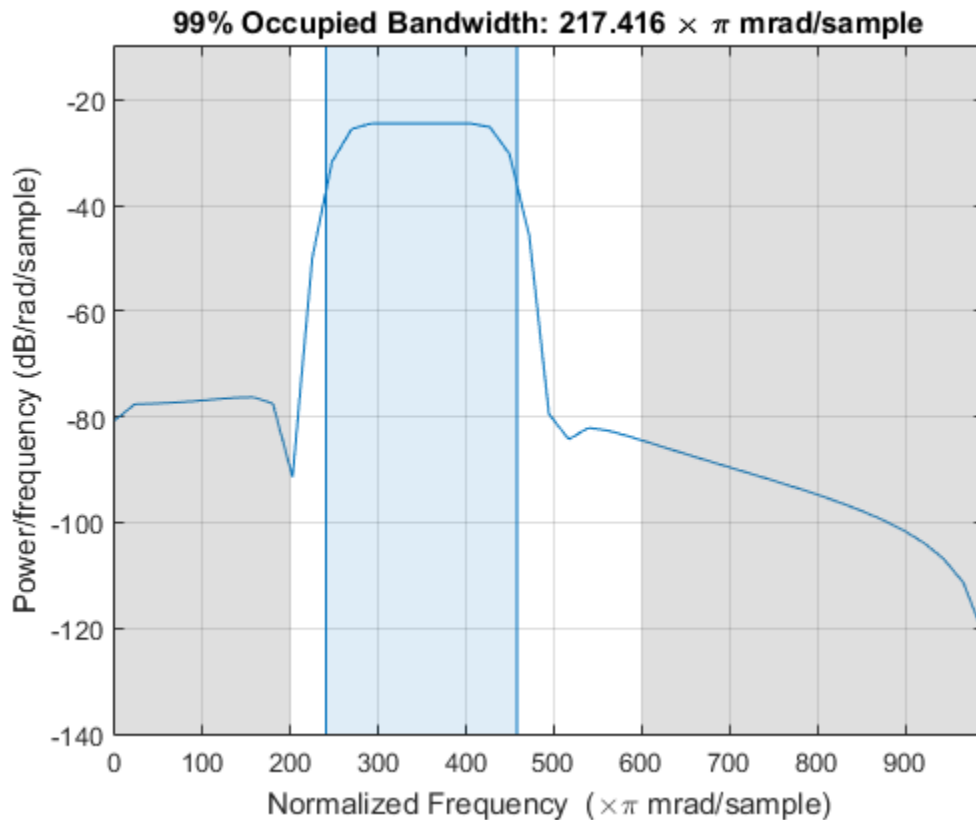
Generate a signal whose PSD resembles the frequency response of an 88th-order bandpass FIR filter with normalized cutoff frequencies  $0.25\pi$  rad/sample and  $0.45\pi$  rad/sample.

```
d = fir1(88,[0.25 0.45]);
```

Compute the 99% occupied bandwidth of the signal between  $0.2\pi$  rad/sample and  $0.6\pi$  rad/sample. Plot the PSD and annotate the occupied bandwidth and measurement interval.

```
obw(d,[],[0.2 0.6]*pi);
```





Output the occupied bandwidth, its lower and upper bounds, and the occupied band power. Specifying a sample rate of  $2\pi$  is equivalent to leaving the rate unset.

```
[bw,flo,fhi,power] = obw(d,2*pi,[0.2 0.6]*pi);
fprintf('bw = %.3f*pi, flo = %.3f*pi, fhi = %.3f*pi \n',[bw flo fhi]/pi)
fprintf('power = %.1f%% of total',power/bandpower(d)*100)

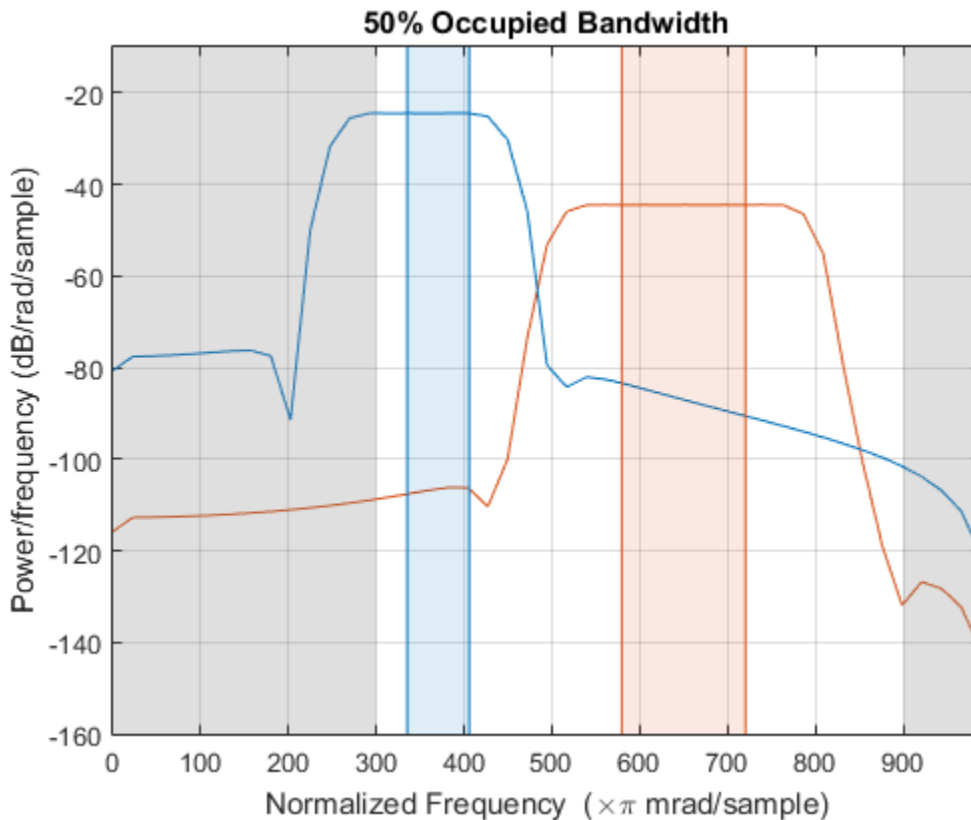
bw = 0.217*pi, flo = 0.240*pi, fhi = 0.458*pi
power = 99.0% of total
```

Add a second channel with normalized cutoff frequencies  $0.5\pi$  rad/sample and  $0.8\pi$  rad/sample and an amplitude that is one-tenth that of the first channel.

```
d = [d;fir1(88,[0.5 0.8])/10]';
```

Compute the 50% occupied bandwidth of the signal between  $0.3\pi$  rad/sample and  $0.9\pi$  rad/sample. Plot the PSD and annotate the occupied bandwidth and measurement interval.

```
obw(d,[],[0.3 0.9]*pi,50);
```



Output the occupied bandwidth of each channel. Divide by  $\pi$ .

```
bw = obw(d,[],[0.3 0.9]*pi,50)/pi
```

```
bw =
```

0.0705    0.1412

## Input Arguments

### **x** — Input signal

vector | matrix

Input signal, specified as a vector or matrix. If **x** is a vector, it is treated as a single channel. If **x** is a matrix, then **obw** computes the occupied bandwidth independently for each column. **x** must be finite-valued.

Example: `cos(pi/4*(0:159))+randn(1,160)` is a single-channel row-vector signal.

Example: `cos(pi./[4;2]*(0:159))'+randn(160,2)` is a two-channel signal.

Data Types: `single` | `double`

### **fs** — Sample rate

positive real scalar

Sample rate, specified as a positive real scalar. The sample rate is the number of samples per unit time. If the time is measured in seconds, then the sample rate is in hertz.

Data Types: `single` | `double`

### **pxx** — Power spectral density

vector | matrix

Power spectral density (PSD), specified as a vector or matrix. If **pxx** is a one-sided estimate, then it must correspond to a real signal. If **pxx** is a matrix, then **obw** computes the occupied bandwidth of each column of **pxx** independently.

Example: `periodogram(cos(pi./[4;2]*(0:159))'+randn(160,2))` is the periodogram PSD estimate of a two-channel sinusoid embedded in white Gaussian noise.

Data Types: `single` | `double`

### **f** — Frequencies

vector

Frequencies, specified as a vector.

Data Types: `single` | `double`

**sxx — Power spectrum estimate**

vector | matrix

Power spectrum estimate, specified as a vector or matrix. If `sxx` is a matrix, then `obw` computes the occupied bandwidth of each column of `sxx` independently.

Example: `periodogram(cos(pi./[4;2]*(0:159))'+randn(160,2),'power')` is the periodogram power spectrum estimate of a two-channel sinusoid embedded in white Gaussian noise.

Data Types: single | double

**rbw — Resolution bandwidth**

positive scalar

Resolution bandwidth, specified as a positive scalar. The resolution bandwidth is the product of two values: the frequency resolution of the discrete Fourier transform and the equivalent noise bandwidth of the window used to compute the PSD.

Data Types: single | double

**freqrange — Frequency range**

two-element vector

Frequency range, specified as a two-element vector of real values. If you do not specify `freqrange`, then `obw` uses the entire bandwidth of the input signal.

Data Types: single | double

**p — Power percentage**

99 (default) | positive scalar

Power percentage, specified as a positive scalar between 0 and 100. `obw` computes the difference in frequency between the points where the integrated power crosses the  $\frac{1}{2}(100 - p)$  and  $\frac{1}{2}(100 + p)$  percentages of the total power in the spectrum.

Data Types: single | double

## Output Arguments

**bw — Occupied bandwidth**

scalar | vector

Occupied bandwidth, returned as a scalar or vector.

- If you specify a sample rate, then `bw` has the same units as `fs`.
- If you do not specify a sample rate, then `bw` has units of rad/sample.

### **f1o, f1i — Bandwidth frequency bounds**

scalars | vectors

Bandwidth frequency bounds, returned as scalars or vectors.

### **power — Power stored in bandwidth**

scalar | vector

Power stored in bandwidth, returned as a scalar or vector.

## **More About**

### **Algorithms**

To determine the occupied bandwidth, `obw` computes a periodogram power spectral density estimate using a rectangular window and integrates the estimate using the midpoint rule. The occupied bandwidth is the difference in frequency between the points where the integrated power crosses 0.5% and 99.5% of the total power in the spectrum.

### **See Also**

`bandpower` | `periodogram` | `plomb` | `powerbw` | `pwelch`

**Introduced in R2015a**

## overshoot

Overshoot metrics of bilevel waveform transitions

### Syntax

```
OS = overshoot(X)
OS = overshoot(X,FS)
OS = overshoot(X,T)
[OS,OSLEV,OSINST] = overshoot(...)
[...] = overshoot(...,Name,Value)
overshoot(...)
```

### Description

`OS = overshoot(X)` returns the greatest absolute deviations larger than the final state levels of each transition in the bilevel waveform, `X`. The overshoots, `OS`, are expressed as a percentage of the difference between the state levels. The length of `OS` corresponds to the number of transitions detected in the input signal. The sample instants in `X` correspond to the vector indices. To determine the transitions, `overshoot` estimates the state levels of the input waveform by a histogram method. `overshoot` identifies all intervals which cross the upper-state boundary of the low state and the lower-state boundary of the high state. The low-state and high-state boundaries are expressed as the state level plus or minus a multiple of the difference between the state levels. See “State-Level Tolerances” on page 1-988.

`OS = overshoot(X,FS)` specifies the sampling frequency in hertz. The sampling frequency determines the sample instants corresponding to the elements in `X`. The first sample instant in `X` corresponds to `t=0`.

`OS = overshoot(X,T)` specifies the sample instants, `T`, as a vector with the same number of elements as `X`.

`[OS,OSLEV,OSINST] = overshoot(...)` returns the levels, `OSLEV`, and sample instants, `OSINST`, of the overshoots for each transition.

`[...] = overshoot(...,Name,Value)` returns the greatest deviations larger than the final state level with additional options specified by one or more `Name,Value` pair arguments.

`overshoot(...)` plots the bilevel waveform and marks the location of the overshoot of each transition as well as the lower and upper reference-level instants and the associated reference levels. The state levels and associated lower and upper-state boundaries are also plotted.

## Input Arguments

### **X**

Bilevel waveform. X is a real-valued row or column vector.

### **FS**

Sample rate in hertz.

### **T**

Vector of sample instants. The length of T must equal the length of the bilevel waveform, X.

## Name-Value Pair Arguments

### **'PercentReferenceLevels'**

Reference levels as a percentage of the waveform amplitude. The lower-state level is defined to be 0 percent. The upper-state level is defined to be 100 percent. The value of `'PercentReferenceLevels'` is a two-element real row vector whose elements correspond to the lower and upper percent reference levels.

**Default:** [10 90]

### **'Region'**

Specifies the region over which to compute the overshoot. Valid values for `'Region'` are `'Preshoot'` or `'Postshoot'`. If you specify `'Preshoot'`, the end of the pretransition aberration region is defined as the last instant where the signal exits the first state. If you specify `'Postshoot'`, the start of the posttransition aberration region is defined as the instant when the signal enters the second state.

**Default:** `'Postshoot'`

**'SeekFactor'**

Aberration region duration. Specifies the duration of the region over which to compute the overshoot for each transition as a multiple of the corresponding transition duration. If the edge of the waveform is reached, or a complete intervening transition is detected before the duration aberration region duration elapses, the duration is truncated to the edge of the waveform or the start of the intervening transition.

**Default:** 3

**'StateLevels'**

Lower and upper state levels. Specifies the levels to use for the lower and upper state levels as a two-element real row vector whose first and second elements correspond to the lower and upper state levels of the input waveform.

**'Tolerance'**

Specifies the tolerance that the initial and final levels of each transition must be within the respective state levels. The **'Tolerance'** value is a scalar expressed as the percentage of the difference between the upper and lower state levels.

**Default:** 2

## Output Arguments

**OS**

Overshoots expressed as a percentage of the state levels. The overshoot percentages are computed based on the greatest deviation from the final state level in each transition. By default overshoots are computed for posttransition aberration regions. See “Overshoot” on page 1-987.

**OSLEV**

Level of the pretransition or posttransition overshoot.

**OSINST**

Sample instants of pretransition or posttransition overshoots. If you specify the sampling frequency or sampling instants, the overshoot instants are in seconds. If you do not specify the sampling frequency or sampling instants, the overshoot instants are the indices of the input vector.



## Definitions

### Overshoot

For a positive-going (positive-polarity) pulse, overshoot expressed as a percentage is

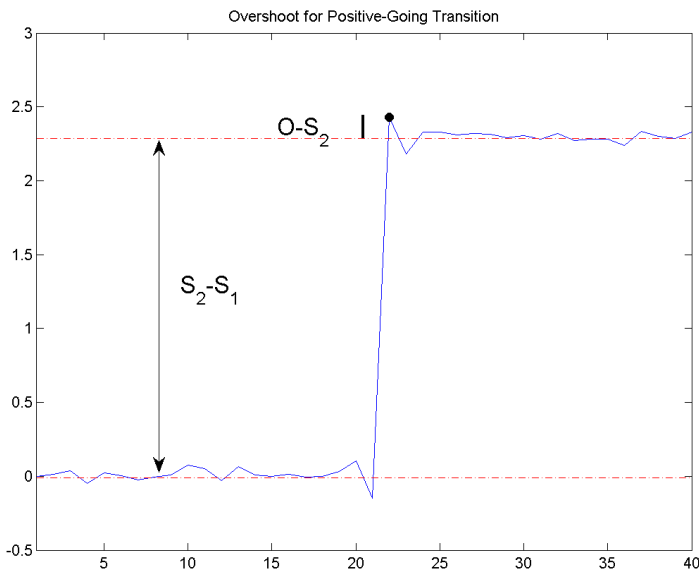
$$100 \frac{(O - S_2)}{(S_2 - S_1)}$$

where  $O$  is the maximum deviation greater the high-state level,  $S_2$  is the high state, and  $S_1$  is the low state.

For a negative-going (negative-polarity) pulse, overshoot expressed as a percentage is

$$100 \frac{(O - S_1)}{(S_2 - S_1)}$$

The following figure illustrates the calculation of overshoot for a positive-going transition.



The red dashed lines indicate the estimated state levels. The double-sided black arrow depicts the difference between the high and low-state levels. The solid black line indicates the difference between the overshoot value and the high-state level.

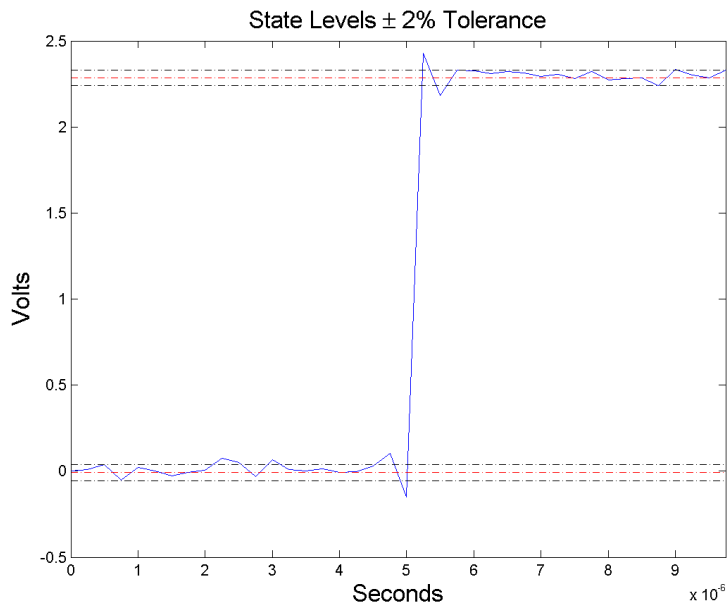
## State-Level Tolerances

Each state level can have associated lower- and upper-state boundaries. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference between the high state and low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the  $\alpha\%$  tolerance region for the low state is defined as

$$S_1 \pm \frac{\alpha}{100}(S_2 - S_1)$$

where  $S_1$  is the low-state level and  $S_2$  is the high-state level. Replace the first term in the equation with  $S_2$  to obtain the  $\alpha\%$  tolerance region for the high state.

The following figure illustrates lower and upper 2% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The estimated state levels are indicated by a dashed red line.



## Examples

### Overshoot Percentage in Posttransition Aberration Region

Determine the maximum percent overshoot relative to the high-state level in a 2.3 V clock waveform.

Load the 2.3 V clock data. Plot the waveform. In this example, you see that the maximum overshoot in the posttransition region occurs near index 22.

```
load('transitionex.mat', 'x');
plot(x);
set(gca, 'xtick', [1 6 12 18 22 28 34 40]);
```

Determine the maximum percent overshoot.

```
os = overshoot(x);
```

### **Overshoot Percentage, Levels, and Sample Instant in Posttransition Aberration Region**

Determine the maximum percent overshoot relative to the high-state level, the level of the overshoot, and the sample instant in a 2.3 V clock waveform.

Load the 2.3 V clock data with sampling instants. Plot the waveform. The clock data is sampled at 4 MHz.

```
load('transitionex.mat', 'x','t');  
plot(t,x);
```

Determine the maximum percent overshoot, the level of the overshoot in volts, and the sampling instant where the maximum overshoot occurs. Plot the result.

```
[os,oslev,osinst] = overshoot(x,t);  
plot(t.*1e6,x); xlabel('Microseconds');  
hold on; grid on;  
plot(osinst*1e6,oslev, 'ro', 'markerfacecolor',[1 0 0]);
```

### **Overshoot Percentage, Levels, and Sample Instant in Pretransition Aberration Region**

Determine the maximum percent overshoot relative to the low-state level, the level of the overshoot, and the sample instant in a 2.3 V clock waveform. Specify the 'Region' as 'Preshoot' to output pretransition metrics.

Load the 2.3 V clock data with sampling instants. Plot the waveform. The clock data is sampled at 4 MHz.

```
load('transitionex.mat', 'x','t');  
plot(t,x);
```

Determine the maximum percent overshoot, the level of the overshoot in volts, and the sampling instant where the maximum overshoot occurs. Plot the result.

```
load('transitionex.mat', 'x','t');  
[os,oslev,osinst] = overshoot(x,t, 'Region', 'Preshoot');  
plot(t.*1e6,x); xlabel('Microseconds');  
hold on; grid on;
```

```
plot(osinst*1e6,oslev,'ro','markerfacecolor',[1 0 0]);
```

## References

- [1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003, pp. 15–17.

## See Also

overshoot | settlingtime | statelevels

## parzenwin

Parzen (de la Vallée Poussin) window

### Syntax

```
w = parzenwin(L)
```

### Description

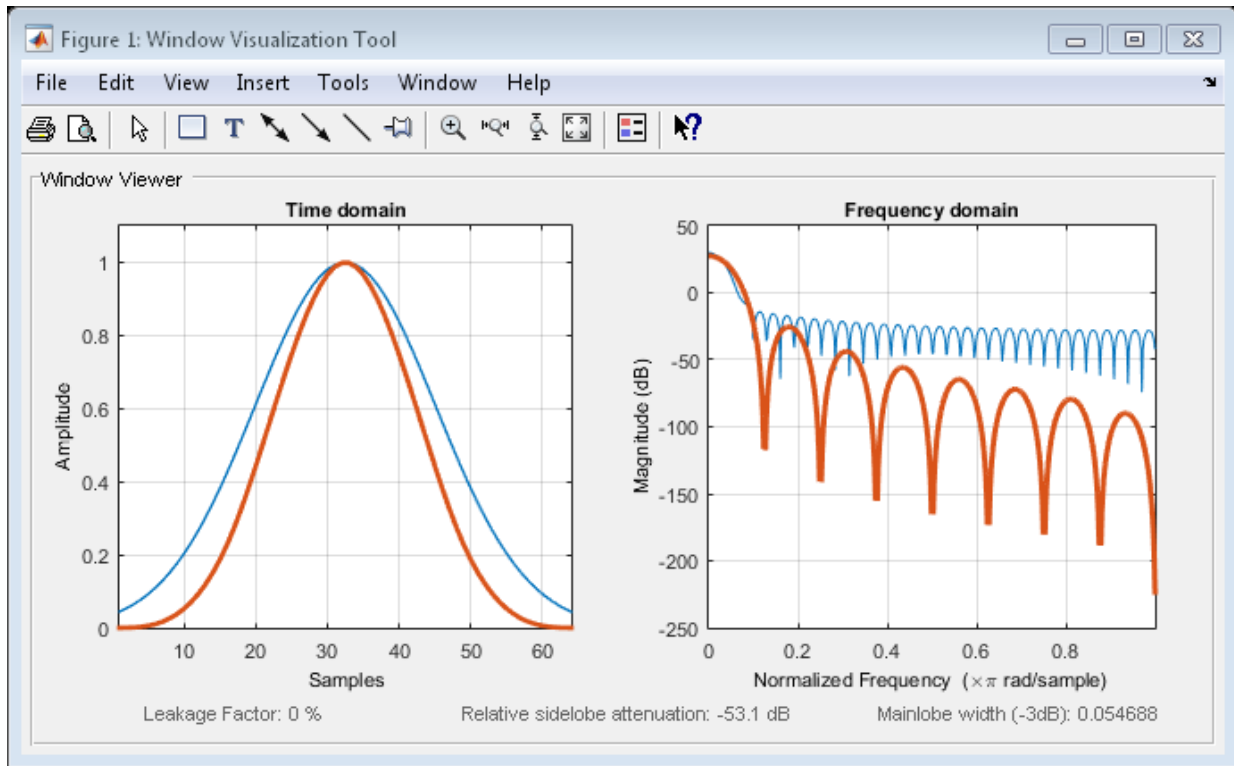
`w = parzenwin(L)` returns the L-point Parzen (de la Vallée Poussin) window in a column vector, `w`. Parzen windows are piecewise-cubic approximations of Gaussian windows. Parzen window sidelobes fall off as  $1/\omega^4$ . See “Definitions” on page 1-993 for the equation that defines the Parzen window.

### Examples

#### Parzen and Gaussian Windows

Compare 64-point Parzen and Gaussian windows. Display the result using `wvtool`.

```
gw = gausswin(64);  
pw = parzenwin(64);  
wvtool(gw,pw)
```



## Definitions

The following equation defines the  $N$ -point Parzen window over the interval

$$-\frac{(N-1)}{2} \leq n \leq \frac{(N-1)}{2} :$$

$$w(n) = \begin{cases} 1 - 6\left(\frac{|n|}{N/2}\right)^2 + 6\left(\frac{|n|}{N/2}\right)^3 & 0 \leq |n| \leq (N-1)/4 \\ 2\left(1 - \frac{|n|}{N/2}\right)^3 & (N-1)/4 < |n| \leq (N-1)/2 \end{cases}$$

## References

- [1] Harris, Fredric J. “On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform.” *Proceedings of the IEEE*. Vol. 66, January 1978, pp. 51–83.

## See Also

barthannwin | bartlett | blackmanharris | bohmanwin | nuttallwin | rectwin  
| triang | window | wintool | wvtool



# pburg

Autoregressive power spectral density estimate — Burg's method

## Syntax

```

pxx = pburg(x,order)
pxx = pburg(x,order,nfft)

[pxx,w] = pburg( ___ )
[pxx,f] = pburg( ___ ,fs)

[pxx,w] = pburg(x,order,w)
[pxx,f] = pburg(x,order,f,fs)

[ ___ ] = pburg(x,order, ___ ,freqrange)

[ ___ ,pxxc] = pburg( ___ , 'ConfidenceLevel',probability)

pburg( ___ )

```

## Description

`pxx = pburg(x,order)` returns the power spectral density (PSD) estimate, `pxx`, of a discrete-time signal, `x`, found using Burg's method. When `x` is a vector, it is treated as a single channel. When `x` is a matrix, the PSD is computed independently for each column and stored in the corresponding column of `pxx`. `pxx` is the distribution of power per unit frequency. The frequency is expressed in units of rad/sample. `order` is the order of the autoregressive (AR) model used to produce the PSD estimate.

`pxx = pburg(x,order,nfft)` uses `nfft` points in the discrete Fourier transform (DFT). For real `x`, `pxx` has length  $(nfft/2+1)$  if `nfft` is even, and  $(nfft+1)/2$  if `nfft` is odd. For complex-valued `x`, `pxx` always has length `nfft`. `pburg` uses a default DFT length of 256.

`[pxx,w] = pburg( ___ )` returns the vector of normalized angular frequencies, `w`, at which the PSD is estimated. `w` has units of rad/sample. For real-valued signals, `w` spans the interval  $[0,\pi]$  when `nfft` is even and  $[0,\pi)$  when `nfft` is odd. For complex-valued signals, `w` always spans the interval  $[0,2\pi)$ .

`[ pxx, f ] = pburg( ____, fs )` returns a frequency vector, `f`, in cycles per unit time. The sampling frequency, `fs`, is the number of samples per unit time. If the unit of time is seconds, then `f` is in cycles/second (Hz). For real-valued signals, `f` spans the interval `[0,fs/2]` when `nfft` is even and `[0,fs/2)` when `nfft` is odd. For complex-valued signals, `f` spans the interval `[0,fs)`.

`[ pxx, w ] = pburg(x, order, w)` returns the two-sided AR PSD estimates at the normalized frequencies specified in the vector, `w`. The vector, `w`, must contain at least two elements.

`[ pxx, f ] = pburg(x, order, f, fs)` returns the two-sided AR PSD estimates at the frequencies specified in the vector, `f`. The vector, `f`, must contain at least two elements. The frequencies in `f` are in cycles per unit time. The sampling frequency, `fs`, is the number of samples per unit time. If the unit of time is seconds, then `f` is in cycles/second (Hz).

`[ ____ ] = pburg(x, order, ____, freqrange)` returns the AR PSD estimate over the frequency range specified by `freqrange`. Valid options for `freqrange` are: `'onesided'`, `'twosided'`, or `'centered'`.

`[ ____, pxxc ] = pburg( ____, 'ConfidenceLevel', probability)` returns the probability  $\times$  100% confidence intervals for the PSD estimate in `pxxc`.

`pburg( ____ )` with no output arguments plots the AR PSD estimate in dB per unit frequency in the current figure window.

## Examples

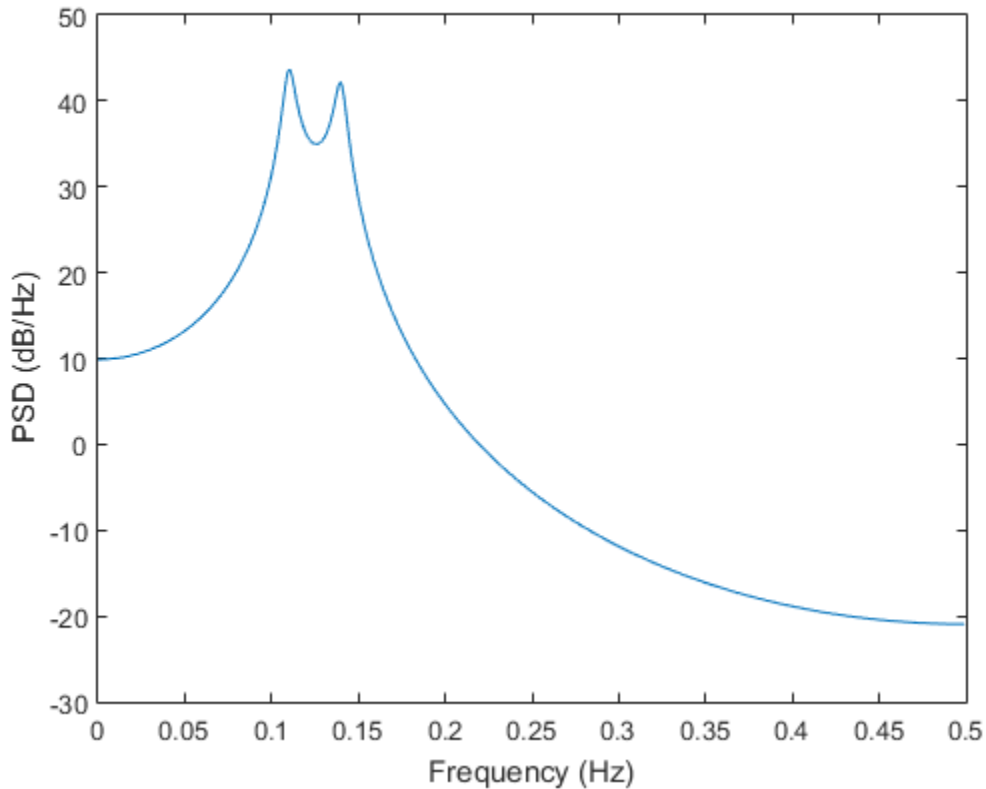
### Burg PSD Estimate of an AR(4) Process

Create a realization of an AR(4) wide-sense stationary random process. Estimate the PSD using Burg's method. Compare the PSD estimate based on a single realization to the true PSD of the random process.

Create an AR(4) system function. Obtain the frequency response and plot the PSD of the system.

```
A = [1 -2.7607 3.8106 -2.6535 0.9238];  
[H,F] = freqz(1,A,[],1);  
plot(F,20*log10(abs(H)))
```

```
xlabel('Frequency (Hz)')  
ylabel('PSD (dB/Hz)')
```

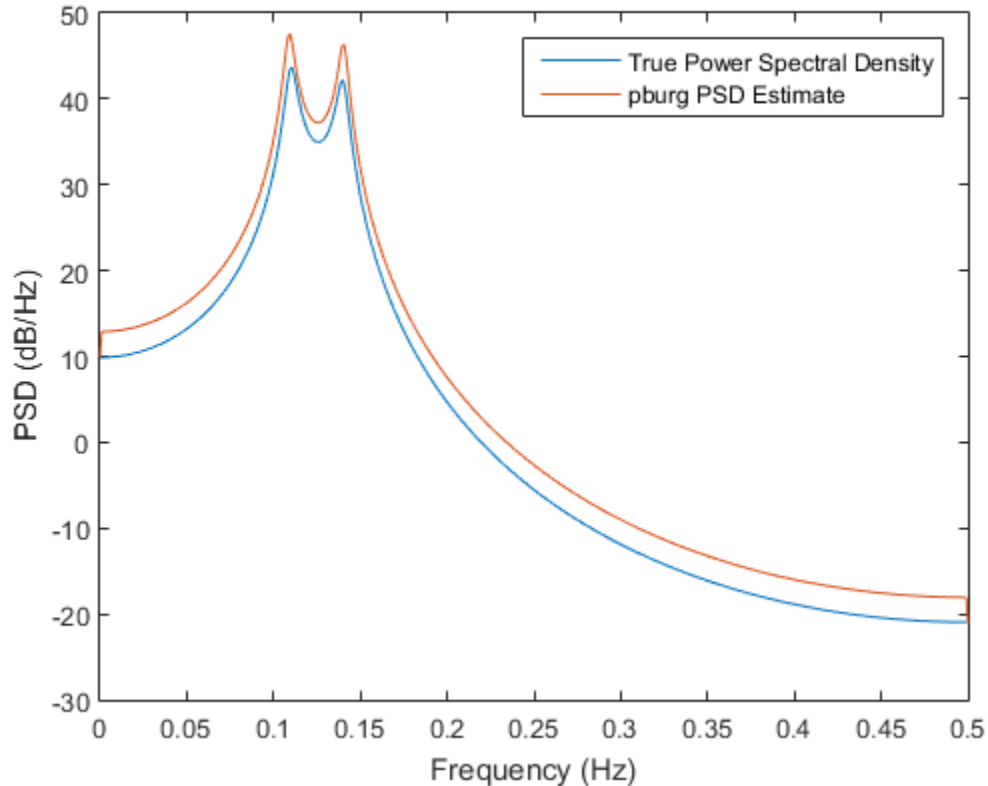


Create a realization of the AR(4) random process. Set the random number generator to the default settings for reproducible results. The realization is 1000 samples in length. Assume a sampling frequency of 1 Hz. Use `pburg` to estimate the PSD for a 4th-order process. Compare the PSD estimate with the true PSD.

```
rng default
```

```
x = randn(1000,1);  
y = filter(1,A,x);  
[Pxx,F] = pburg(y,4,1024,1);
```

```
hold on
plot(F, 10*log10(Pxx))
legend('True Power Spectral Density', 'pburg PSD Estimate')
```

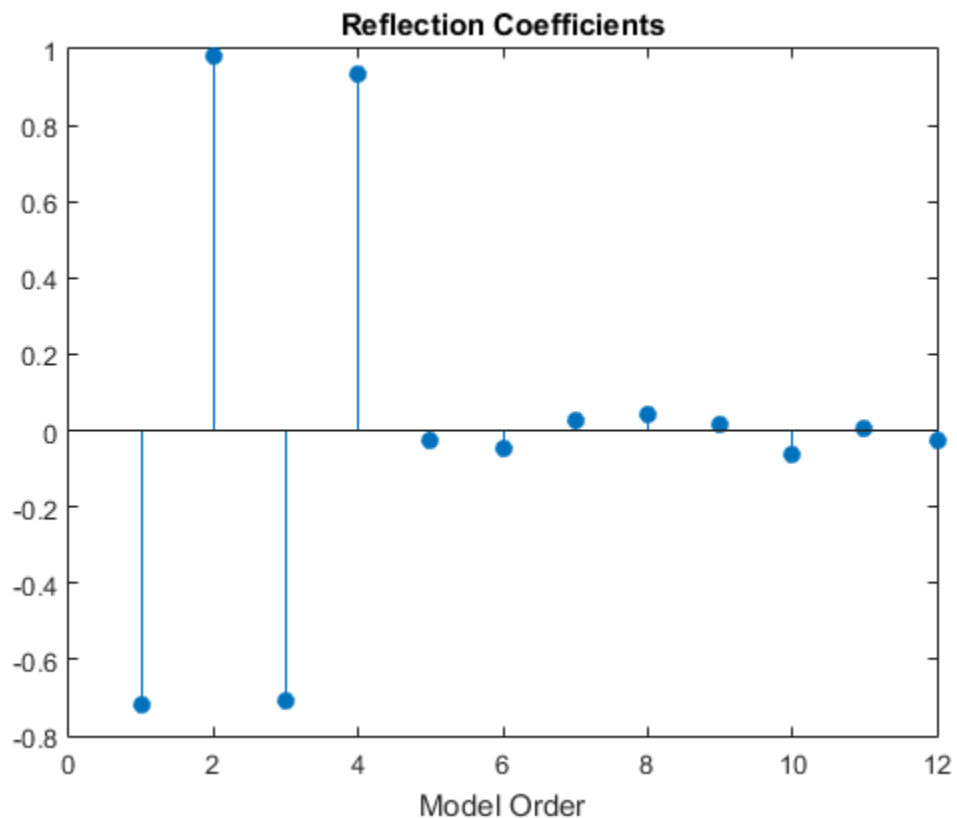


### Reflection Coefficients for Model Order Determination

Create a realization of an AR(4) process. Use `arburg` to determine the reflection coefficients. Use the reflection coefficients to determine an appropriate AR model order for the process. Obtain an estimate of the process PSD.

Create a realization of an AR(4) process 1000 samples in length. Use `arburg` with the order set to 12 to return the reflection coefficients. Plot the reflection coefficients to determine an appropriate model order.

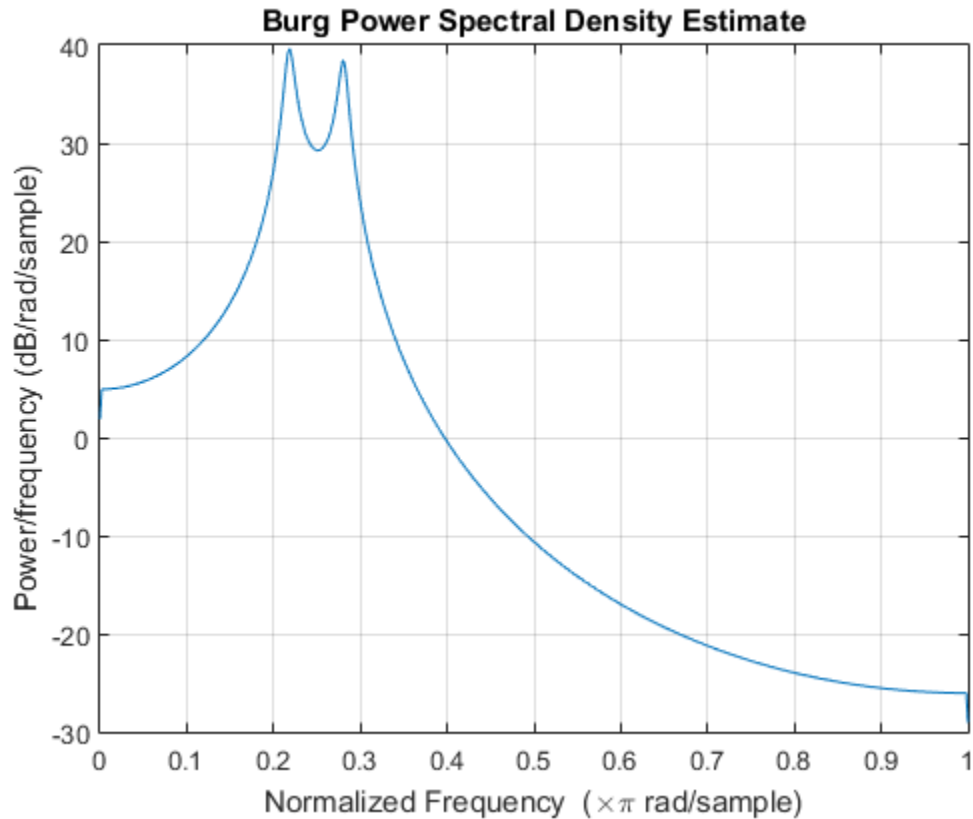
```
A = [1 -2.7607 3.8106 -2.6535 0.9238];  
  
rng default  
  
x = filter(1,A,randn(1000,1));  
  
[a,e,k] = arburg(x,12);  
  
stem(k,'filled')  
title('Reflection Coefficients')  
xlabel('Model Order')
```



The reflection coefficients decay to zero after order 4. This indicates an AR(4) model is most appropriate.

Obtain a PSD estimate of the random process using Burg's method. Use 1000 points in the DFT. Plot the PSD estimate.

```
pburg(x,4,length(x))
```



### Burg PSD Estimate of a Multichannel Signal

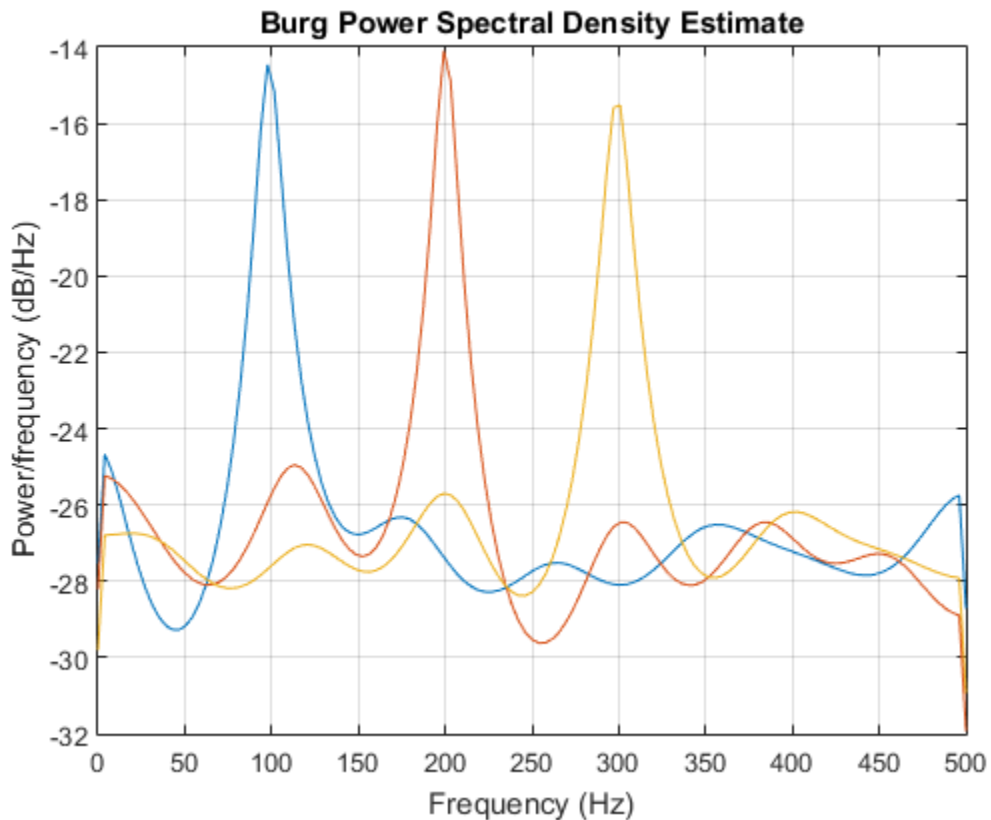
Create a multichannel signal consisting of three sinusoids in additive  $N(0, 1)$  white Gaussian noise. The sinusoids' frequencies are 100 Hz, 200 Hz, and 300 Hz. The sampling frequency is 1 kHz, and the signal has a duration of 1 s.

```
Fs = 1000;
```

```
t = 0:1/Fs:1-1/Fs;  
f = [100;200;300];  
x = cos(2*pi*f*t)' + randn(length(t),3);
```

Estimate the PSD of the signal using Burg's method with a 12th-order autoregressive model. Use the default DFT length. Plot the estimate.

```
morder = 12;  
pburg(x,morder,[],Fs)
```



## Input Arguments

### **x** — Input signal

vector | matrix

Input signal, specified as a row or column vector, or as a matrix. If  $x$  is a matrix, then its columns are treated as independent channels.

Example: `cos(pi/4*(0:159))+randn(1,160)` is a single-channel row-vector signal.

Example: `cos(pi./[4;2]*(0:159))'+randn(160,2)` is a two-channel signal.

Data Types: `single` | `double`



Complex Number Support: Yes

### **order** — Order of autoregressive model

positive integer

Order of the autoregressive model, specified as a positive integer.

Data Types: double

### **nfft** — Number of DFT points

256 (default) | integer | []

Number of DFT points, specified as a positive integer. For a real-valued input signal,  $x$ , the PSD estimate,  $pxx$  has length  $(nfft/2+1)$  if  $nfft$  is even, and  $(nfft+1)/2$  if  $nfft$  is odd. For a complex-valued input signal,  $x$ , the PSD estimate always has length  $nfft$ . If  $nfft$  is specified as empty, the default  $nfft$  is used.

Data Types: single | double

### **fs** — Sampling frequency

positive scalar

Sampling frequency, specified as a positive scalar. The sampling frequency is the number of samples per unit time. If the unit of time is seconds, the sampling frequency has units of hertz.

### **w** — Normalized frequencies

vector

Normalized frequencies, specified as a row or column vector with at least 2 elements. Normalized frequencies are in rad/sample.

Example:  $w = [\pi/4 \ \pi/2]$

Data Types: double

### **f** — Cyclical frequencies

vector

Cyclical frequencies, specified as a row or column vector with at least 2 elements. The frequencies are in cycles per unit time. The unit time is specified by the sampling frequency,  $fs$ . If  $fs$  has units of samples/second, then  $f$  has units of Hz.

Example:  $fs = 1000$ ;  $f = [100 \ 200]$

Data Types: double

**freqrange** — Frequency range for PSD estimate

'onesided' | 'twosided' | 'centered'

Frequency range for the PSD estimate, specified as a one of 'onesided', 'twosided', or 'centered'. The default is 'onesided' for real-valued signals and 'twosided' for complex-valued signals. The frequency ranges corresponding to each option are

- 'onesided' — returns the one-sided PSD estimate of a real-valued input signal,  $x$ . If  $nfft$  is even,  $pxx$  has length  $nfft/2 + 1$  and is computed over the interval  $[0, \pi]$  rad/sample. If  $nfft$  is odd, the length of  $pxx$  is  $(nfft + 1)/2$  and the interval is  $[0, \pi]$  rad/sample. When  $fs$  is optionally specified, the corresponding intervals are  $[0, fs/2]$  cycles/unit time and  $[0, fs/2)$  cycles/unit time for even and odd length  $nfft$  respectively.
- 'twosided' — returns the two-sided PSD estimate for either the real-valued or complex-valued input,  $x$ . In this case,  $pxx$  has length  $nfft$  and is computed over the interval  $[0, 2\pi]$  rad/sample. When  $fs$  is optionally specified, the interval is  $[0, fs)$  cycles/unit time.
- 'centered' — returns the centered two-sided PSD estimate for either the real-valued or complex-valued input,  $x$ . In this case,  $pxx$  has length  $nfft$  and is computed over the interval  $(-\pi, \pi]$  rad/sample for even length  $nfft$  and  $(-\pi, \pi)$  rad/sample for odd length  $nfft$ . When  $fs$  is optionally specified, the corresponding intervals are  $(-fs/2, fs/2]$  cycles/unit time and  $(-fs/2, fs/2)$  cycles/unit time for even and odd length  $nfft$  respectively.

Data Types: char

**probability** — Confidence interval for PSD estimate

0.95 (default) | scalar in the range (0,1)

Coverage probability for the true PSD, specified as a scalar in the range (0,1). The output,  $pxxc$ , contains the lower and upper bounds of the probability  $\times 100\%$  interval estimate for the true PSD.

## Output Arguments

**pxx** — PSD estimate

vector | matrix

PSD estimate, returned as a real-valued, nonnegative column vector or matrix. Each column of `pxx` is the PSD estimate of the corresponding column of `x`. The units of the PSD estimate are in squared magnitude units of the time series data per unit frequency. For example, if the input data is in volts, the PSD estimate is in units of squared volts per unit frequency. For a time series in volts, if you assume a resistance of  $1 \Omega$  and specify the sampling frequency in hertz, the PSD estimate is in watts per hertz.

Data Types: `single` | `double`

### **w** — Normalized frequencies

vector

Normalized frequencies, returned as a real-valued column vector. If `pxx` is a one-sided PSD estimate, `w` spans the interval  $[0, \pi]$  if `nfft` is even and  $[0, \pi)$  if `nfft` is odd. If `pxx` is a two-sided PSD estimate, `w` spans the interval  $[0, 2\pi)$ . For a DC-centered PSD estimate, `f` spans the interval  $(-\pi, \pi]$  rad/sample for even length `nfft` and  $(-\pi, \pi)$  radians/sample for odd length `nfft`.

Data Types: `double`

### **f** — Cyclical frequencies

vector

Cyclical frequencies, returned as a real-valued column vector. For a one-sided PSD estimate, `f` spans the interval  $[0, fs/2]$  when `nfft` is even and  $[0, fs/2)$  when `nfft` is odd. For a two-sided PSD estimate, `f` spans the interval  $[0, fs)$ . For a DC-centered PSD estimate, `f` spans the interval  $(-fs/2, fs/2]$  cycles/unit time for even length `nfft` and  $(-fs/2, fs/2)$  cycles/unit time for odd length `nfft`.

Data Types: `double`

### **pxxc** — Confidence bounds

matrix

Confidence bounds, returned as a matrix with real-valued elements. The row size of the matrix is equal to the length of the PSD estimate, `pxx`. `pxxc` has twice as many columns as `pxx`. Odd-numbered columns contain the lower bounds of the confidence intervals, and even-numbered columns contain the upper bounds. Thus, `pxxc(m, 2*n - 1)` is the lower confidence bound and `pxxc(m, 2*n)` is the upper confidence bound corresponding to the estimate `pxx(m, n)`. The coverage probability of the confidence intervals is determined by the value of the probability input.

Data Types: `single` | `double`

**See Also**

pcov | pmcov | pyulearn

## pcov

Autoregressive power spectral density estimate — covariance method

### Syntax

```
pxx = pcov(x,order)
pxx = pcov(x,order,nfft)

[pxx,w] = pcov( ___ )
[pxx,f] = pcov( ___ ,fs)

[pxx,w] = pcov(x,order,w)
[pxx,f] = pcov(x,order,f,fs)

[ ___ ] = pcov(x,order, ___ ,freqrange)

[ ___ ,pxxc] = pcov( ___ ,'ConfidenceLevel',probability)

pcov( ___ )
```

### Description

`pxx = pcov(x,order)` returns the power spectral density (PSD) estimate, `pxx`, of a discrete-time signal, `x`, found using the covariance method. When `x` is a vector, it is treated as a single channel. When `x` is a matrix, the PSD is computed independently for each column and stored in the corresponding column of `pxx`. `pxx` is the distribution of power per unit frequency. The frequency is expressed in units of rad/sample. `order` is the order of the autoregressive (AR) model used to produce the PSD estimate.

`pxx = pcov(x,order,nfft)` uses `nfft` points in the discrete Fourier transform (DFT). For real `x`, `pxx` has length  $(nfft/2+1)$  if `nfft` is even, and  $(nfft+1)/2$  if `nfft` is odd. For complex-valued `x`, `pxx` always has length `nfft`. `pcov` uses a default DFT length of 256.

`[pxx,w] = pcov( ___ )` returns the vector of normalized angular frequencies, `w`, at which the PSD is estimated. `w` has units of radians/sample. For real-valued signals, `w`

spans the interval  $[0, \pi]$  when `nfft` is even and  $[0, \pi)$  when `nfft` is odd. For complex-valued signals, `w` always spans the interval  $[0, 2\pi]$ .

`[ pxx, f ] = pcov( ____, fs )` returns a frequency vector, `f`, in cycles per unit time. The sampling frequency, `fs`, is the number of samples per unit time. If the unit of time is seconds, then `f` is in cycles/second (Hz). For real-valued signals, `f` spans the interval  $[0, fs/2]$  when `nfft` is even and  $[0, fs/2)$  when `nfft` is odd. For complex-valued signals, `f` spans the interval  $[0, fs)$ .

`[ pxx, w ] = pcov( x, order, w )` returns the two-sided AR PSD estimates at the normalized frequencies specified in the vector, `w`. The vector, `w`, must contain at least two elements.

`[ pxx, f ] = pcov( x, order, f, fs )` returns the two-sided AR PSD estimates at the frequencies specified in the vector, `f`. The vector, `f`, must contain at least two elements. The frequencies in `f` are in cycles per unit time. The sampling frequency, `fs`, is the number of samples per unit time. If the unit of time is seconds, then `f` is in cycles/second (Hz).

`[ ____ ] = pcov( x, order, ____, freqrange )` returns the AR PSD estimate over the frequency range specified by `freqrange`. Valid options for `freqrange` are: `'onesided'`, `'twosided'`, or `'centered'`.

`[ ____, pxxc ] = pcov( ____, 'ConfidenceLevel', probability )` returns the `probability` × 100% confidence intervals for the PSD estimate in `pxxc`.

`pcov( ____ )` with no output arguments plots the AR PSD estimate in dB per unit frequency in the current figure window.

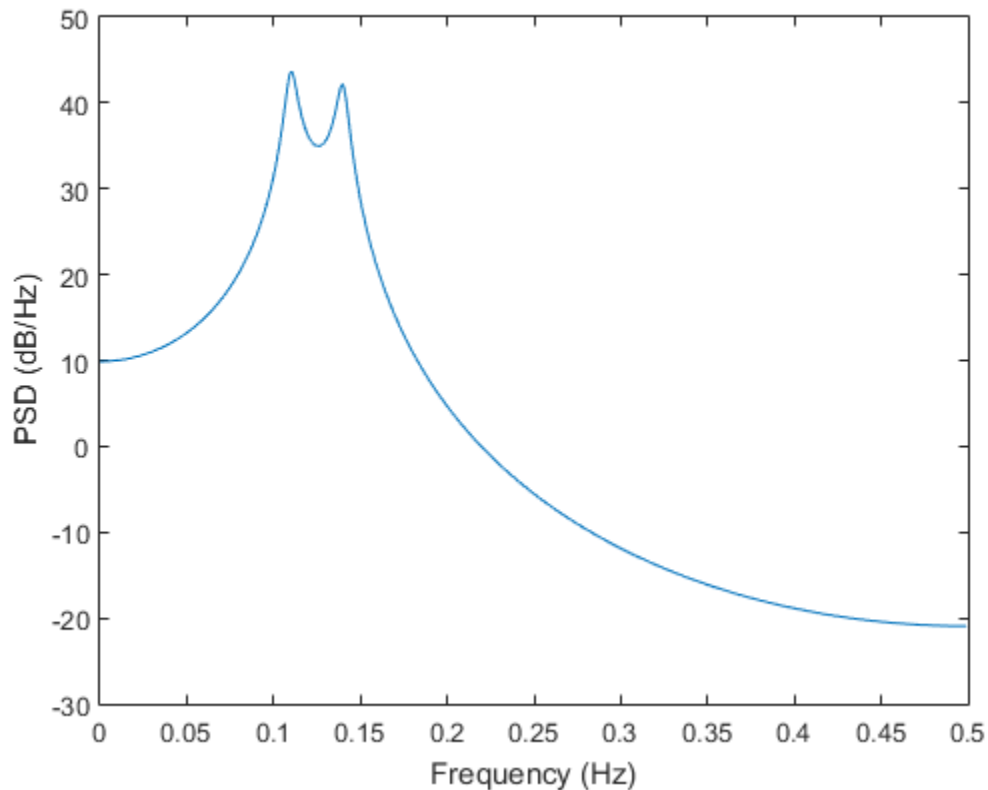
## Examples

### Covariance-Method PSD Estimate of an AR(4) Process

Create a realization of an AR(4) wide-sense stationary random process. Estimate the PSD using the covariance method. Compare the PSD estimate based on a single realization to the true PSD of the random process.

Create an AR(4) system function. Obtain the frequency response and plot the PSD of the system.

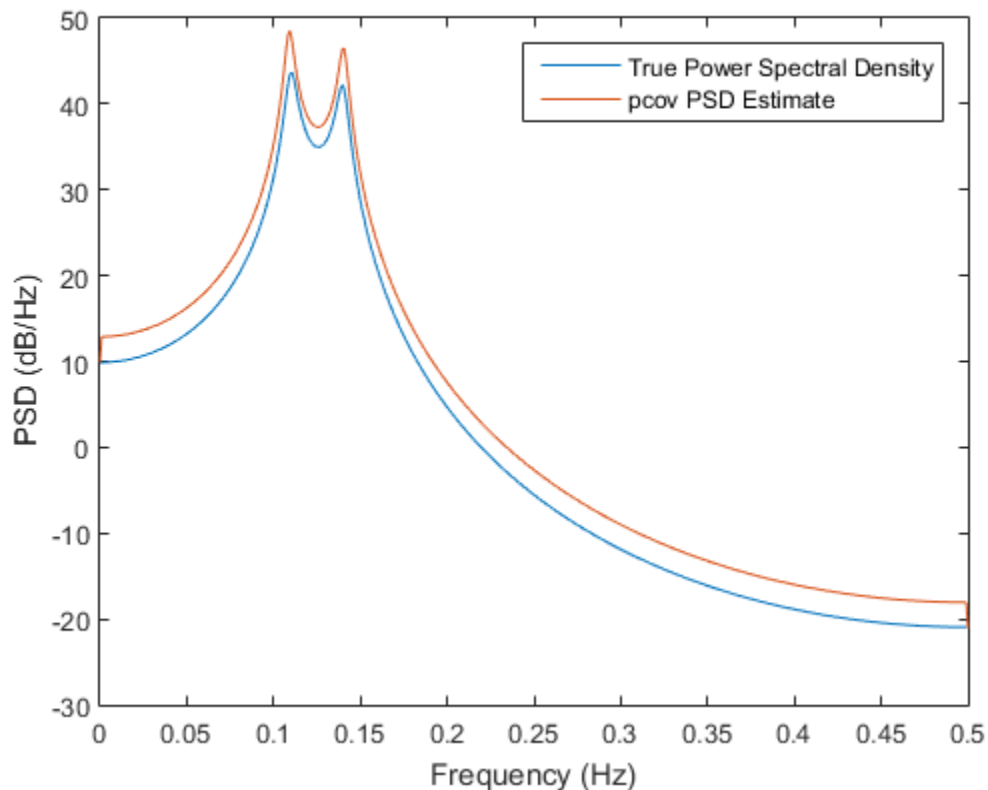
```
A = [1 -2.7607 3.8106 -2.6535 0.9238];  
[H,F] = freqz(1,A,[],1);  
plot(F,20*log10(abs(H))  
  
xlabel('Frequency (Hz)')  
ylabel('PSD (dB/Hz)')
```



Create a realization of the AR(4) random process. Set the random number generator to the default settings for reproducible results. The realization is 1000 samples in length. Assume a sampling frequency of 1 Hz. Use `pcov` to estimate the PSD for a 4th-order process. Compare the PSD estimate with the true PSD.

```
rng default
```

```
x = randn(1000,1);  
y = filter(1,A,x);  
[Pxx,F] = pcov(y,4,1024,1);  
  
hold on  
plot(F,10*log10(Pxx))  
legend('True Power Spectral Density','pcov PSD Estimate')
```



### Covariance-Method PSD Estimate of a Multichannel Signal

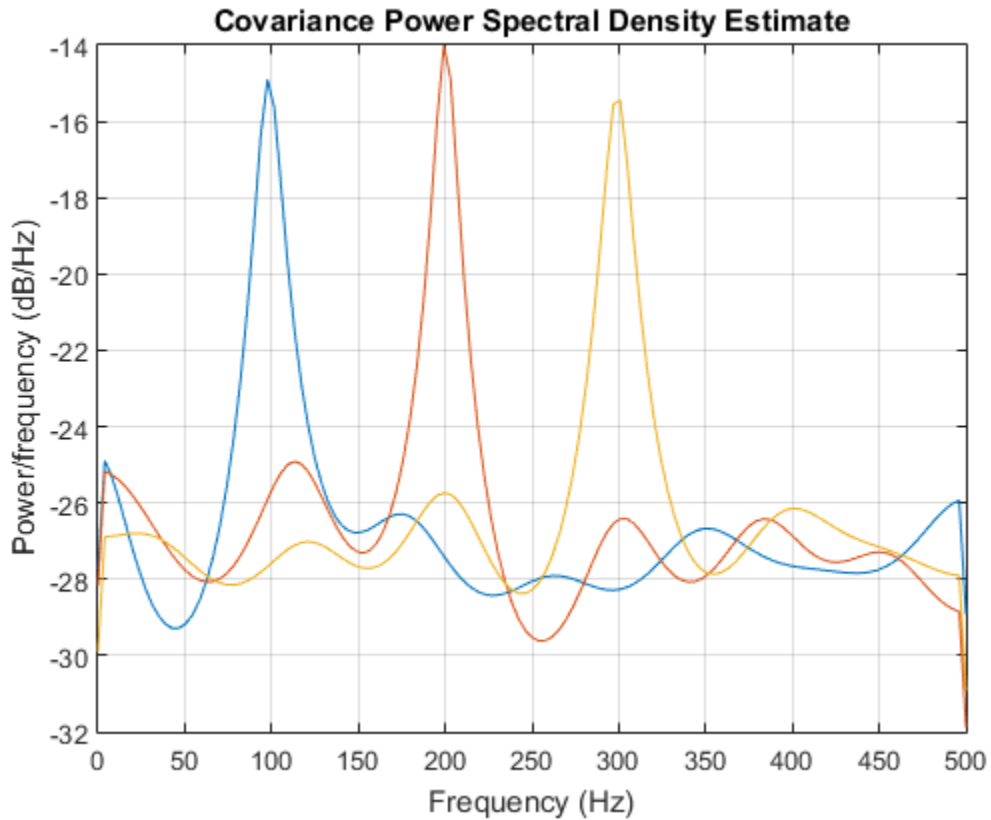
Create a multichannel signal consisting of three sinusoids in additive  $N(0,1)$  white Gaussian noise. The sinusoids' frequencies are 100 Hz, 200 Hz, and 300 Hz. The sampling frequency is 1 kHz, and the signal has a duration of 1 s.



```
Fs = 1000;  
t = 0:1/Fs:1-1/Fs;  
f = [100;200;300];  
x = cos(2*pi*f*t)' + randn(length(t),3);
```

Estimate the PSD of the signal using the covariance method with a 12th-order autoregressive model. Use the default DFT length. Plot the estimate.

```
morder = 12;  
pcov(x,morder,[],Fs)
```



## Input Arguments

### **x** — Input signal

vector | matrix

Input signal, specified as a row or column vector, or as a matrix. If  $x$  is a matrix, then its columns are treated as independent channels.

Example: `cos(pi/4*(0:159))+randn(1,160)` is a single-channel row-vector signal.

Example: `cos(pi./[4;2]*(0:159))'+randn(160,2)` is a two-channel signal.

Data Types: `single` | `double`

Complex Number Support: Yes

### **order** — Order of autoregressive model

positive integer

Order of the autoregressive model, specified as a positive integer.

Data Types: double

### **nfft** — Number of DFT points

256 (default) | integer | []

Number of DFT points, specified as a positive integer. For a real-valued input signal,  $x$ , the PSD estimate,  $pxx$  has length  $(nfft/2+1)$  if  $nfft$  is even, and  $(nfft+1)/2$  if  $nfft$  is odd. For a complex-valued input signal,  $x$ , the PSD estimate always has length  $nfft$ . If  $nfft$  is specified as empty, the default  $nfft$  is used.

Data Types: single | double

### **fs** — Sampling frequency

positive scalar

Sampling frequency, specified as a positive scalar. The sampling frequency is the number of samples per unit time. If the unit of time is seconds, the sampling frequency has units of hertz.

### **w** — Normalized frequencies

vector

Normalized frequencies, specified as a row or column vector with at least 2 elements. Normalized frequencies are in rad/sample.

Example:  $w = [\pi/4 \ \pi/2]$

Data Types: double

### **f** — Cyclical frequencies

vector

Cyclical frequencies, specified as a row or column vector with at least 2 elements. The frequencies are in cycles per unit time. The unit time is specified by the sampling frequency,  $fs$ . If  $fs$  has units of samples/second, then  $f$  has units of Hz.

Example:  $fs = 1000$ ;  $f = [100 \ 200]$

Data Types: double

**freqrange — Frequency range for PSD estimate**

'onesided' | 'twosided' | 'centered'

Frequency range for the PSD estimate, specified as a one of 'onesided', 'twosided', or 'centered'. The default is 'onesided' for real-valued signals and 'twosided' for complex-valued signals. The frequency ranges corresponding to each option are

- 'onesided' — returns the one-sided PSD estimate of a real-valued input signal,  $x$ . If  $nfft$  is even,  $pxx$  has length  $nfft/2 + 1$  and is computed over the interval  $[0, \pi]$  rad/sample. If  $nfft$  is odd, the length of  $pxx$  is  $(nfft + 1)/2$  and the interval is  $[0, \pi]$  rad/sample. When  $fs$  is optionally specified, the corresponding intervals are  $[0, fs/2]$  cycles/unit time and  $[0, fs/2)$  cycles/unit time for even and odd length  $nfft$  respectively.
- 'twosided' — returns the two-sided PSD estimate for either the real-valued or complex-valued input,  $x$ . In this case,  $pxx$  has length  $nfft$  and is computed over the interval  $[0, 2\pi)$  rad/sample. When  $fs$  is optionally specified, the interval is  $[0, fs)$  cycles/unit time.
- 'centered' — returns the centered two-sided PSD estimate for either the real-valued or complex-valued input,  $x$ . In this case,  $pxx$  has length  $nfft$  and is computed over the interval  $(-\pi, \pi]$  rad/sample for even length  $nfft$  and  $(-\pi, \pi)$  rad/sample for odd length  $nfft$ . When  $fs$  is optionally specified, the corresponding intervals are  $(-fs/2, fs/2]$  cycles/unit time and  $(-fs/2, fs/2)$  cycles/unit time for even and odd length  $nfft$  respectively.

Data Types: char

**probability — Confidence interval for PSD estimate**

0.95 (default) | scalar in the range (0,1)

Coverage probability for the true PSD, specified as a scalar in the range (0,1). The output,  $pxxc$ , contains the lower and upper bounds of the probability  $\times 100\%$  interval estimate for the true PSD.

## Output Arguments

**pxx — PSD estimate**

vector | matrix

PSD estimate, returned as a real-valued, nonnegative column vector or matrix. Each column of `pxx` is the PSD estimate of the corresponding column of `x`. The units of the PSD estimate are in squared magnitude units of the time series data per unit frequency. For example, if the input data is in volts, the PSD estimate is in units of squared volts per unit frequency. For a time series in volts, if you assume a resistance of  $1 \Omega$  and specify the sampling frequency in hertz, the PSD estimate is in watts per hertz.

Data Types: `single` | `double`

### **w** — Normalized frequencies

vector

Normalized frequencies, returned as a real-valued column vector. If `pxx` is a one-sided PSD estimate, `w` spans the interval  $[0, \pi]$  if `nfft` is even and  $[0, \pi)$  if `nfft` is odd. If `pxx` is a two-sided PSD estimate, `w` spans the interval  $[0, 2\pi)$ . For a DC-centered PSD estimate, `f` spans the interval  $(-\pi, \pi]$  rad/sample for even length `nfft` and  $(-\pi, \pi)$  radians/sample for odd length `nfft`.

Data Types: `double`

### **f** — Cyclical frequencies

vector

Cyclical frequencies, returned as a real-valued column vector. For a one-sided PSD estimate, `f` spans the interval  $[0, fs/2]$  when `nfft` is even and  $[0, fs/2)$  when `nfft` is odd. For a two-sided PSD estimate, `f` spans the interval  $[0, fs)$ . For a DC-centered PSD estimate, `f` spans the interval  $(-fs/2, fs/2]$  cycles/unit time for even length `nfft` and  $(-fs/2, fs/2)$  cycles/unit time for odd length `nfft`.

Data Types: `double`

### **pxxc** — Confidence bounds

matrix

Confidence bounds, returned as a matrix with real-valued elements. The row size of the matrix is equal to the length of the PSD estimate, `pxx`. `pxxc` has twice as many columns as `pxx`. Odd-numbered columns contain the lower bounds of the confidence intervals, and even-numbered columns contain the upper bounds. Thus, `pxxc(m, 2*n - 1)` is the lower confidence bound and `pxxc(m, 2*n)` is the upper confidence bound corresponding to the estimate `pxx(m, n)`. The coverage probability of the confidence intervals is determined by the value of the probability input.

Data Types: `single` | `double`

**See Also**

`pburg` | `pmcov` | `pyulearn`

# peak2peak

Maximum-to-minimum difference

## Syntax

$Y = \text{peak2peak}(X)$

$Y = \text{peak2peak}(X, \text{DIM})$

## Description

$Y = \text{peak2peak}(X)$  returns the difference between the maximum and minimum values in  $X$ . `peak2peak` operates along the first nonsingleton dimension of  $X$  by default. For example, if  $X$  is a row or column vector,  $Y$  is a real-valued scalar. If  $Y$  is an  $N$ -by- $M$  matrix with  $N > 1$ ,  $Y$  is a 1-by- $M$  row vector containing the maximum-to-minimum differences of the columns of  $X$ .

$Y = \text{peak2peak}(X, \text{DIM})$  computes the maximum-to-minimum differences of  $X$  along the dimension,  $\text{DIM}$ .

## Input Arguments

### **$X$**

Real- or complex-valued input vector or matrix. By default, `peak2peak` acts along the first nonsingleton dimension of  $X$ . For complex-valued inputs, `peak2peak` identifies the maximum and minimum in absolute value. `peak2peak` subtracts the complex number with the minimum modulus from the complex number with the maximum modulus.

### **$\text{DIM}$**

Dimension for maximum-to-minimum difference. The optional  $\text{DIM}$  input argument specifies the dimension along which to compute the maximum-to-minimum differences.

**Default:** First nonsingleton dimension

## Output Arguments

**Y**

Maximum-to-minimum difference. For vectors, Y is a real-valued scalar. For matrices, Y contains the maximum-to-minimum differences computed along the specified dimension, DIM. By default, DIM is the first nonsingleton dimension.

## Examples

### Peak-to-Peak Difference of Sinusoid

Compute the maximum-to-minimum difference of a 100-Hz sinusoid sampled at 1 kHz.

```
t = 0:0.001:1-0.001;  
X = cos(2*pi*100*t);  
Y = peak2peak(X);
```

### Peak-to-Peak Difference of Complex Exponential

Compute the maximum-to-minimum difference of a complex exponential with a frequency of  $\pi/4$  radians/sample.

Create a complex exponential with a frequency of  $\pi/4$  radians/sample. Find the peak-to-peak difference.

```
n = 0:99;  
x = exp(1j*pi/4*n);  
maxmin = peak2peak(x);
```

### Peak-to-Peak Differences of 2-D Matrix

Create a matrix where each column is a 100-Hz sinusoid sampled at 1 kHz with a different amplitude. The amplitude is equal to the column index.

Compute the maximum-to-minimum differences of the columns.

```
t = 0:0.001:1-0.001;  
x = cos(2*pi*100*t)';  
X = repmat(x,1,4);  
amp = 1:4;  
amp = repmat(amp,1e3,1);
```



```
X = X.*amp;  
Y = peak2peak(X);
```

### Peak-to-Peak Differences of 2-D Matrix Along Specified Dimension

Create a matrix where each row is a 100-Hz sinusoid sampled at 1 kHz with a different amplitude. The amplitude is equal to the row index.

Compute the maximum-to-minimum differences of the rows specifying the dimension equal to 2 with the DIM argument.

```
t = 0:0.001:1-0.001;  
x = cos(2*pi*100*t);  
X = repmat(x,4,1);  
amp = (1:4)';  
amp = repmat(amp,1,1e3);  
X = X.*amp;  
Y = peak2peak(X,2);
```

## References

- [1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003.

## See Also

max | min | peak2rms

## peak2rms

Peak-magnitude-to-RMS ratio

### Syntax

$Y = \text{peak2rms}(X)$   
 $Y = \text{peak2rms}(X, \text{DIM})$

### Description

$Y = \text{peak2rms}(X)$  returns the ratio of the largest absolute value in  $X$  to the root-mean-square (RMS) value of  $X$ . `peak2rms` operates along the first nonsingleton dimension of  $X$ . For example, if  $X$  is a row or column vector,  $Y$  is a real-valued scalar. If  $Y$  is an  $N$ -by- $M$  matrix with  $N > 1$ ,  $Y$  is a 1-by- $M$  row vector containing the peak-magnitude-to-RMS levels of the columns of  $Y$ .

$Y = \text{peak2rms}(X, \text{DIM})$  computes the peak-magnitude-to-RMS level of  $X$  along the dimension,  $\text{DIM}$ .

### Input Arguments

#### $X$

Real- or complex-valued input vector or matrix. By default, `peak2rms` acts along the first nonsingleton dimension of  $X$ .

#### $\text{DIM}$

Dimension for peak-magnitude-to-RMS ratio. The optional  $\text{DIM}$  input argument specifies the dimension along which to compute the peak-magnitude-to-RMS level.

**Default:** First nonsingleton dimension

## Output Arguments

Y

Peak-magnitude-to-RMS ratio. For vectors, Y is a real-valued scalar. For matrices, Y contains the peak-magnitude-to-RMS levels computed along the specified dimension, DIM. By default, DIM is the first nonsingleton dimension.

## Examples

### Peak-magnitude-to-RMS Ratio of Sinusoid

Compute the peak-magnitude-to-RMS ratio of a 100-Hz sinusoid sampled at 1 kHz.

```
t = 0:0.001:1-0.001;
X = cos(2*pi*100*t);
Y = peak2rms(X);
```

### Peak-magnitude-to-RMS Ratio of Complex Exponential

Compute the peak-magnitude-to-RMS ratio of a complex exponential with a frequency of  $\pi/4$  radians/sample.

Create a complex exponential with a frequency of  $\pi/4$  radians/sample. Find the peak-magnitude-to-RMS ratio.

```
n = 0:99;
X = exp(1j*pi/4*n);
Y = peak2rms(X);
```

### Peak-magnitude-to-RMS ratio of 2-D Matrix

Create a matrix where each column is a 100-Hz sinusoid sampled at 1 kHz with a different amplitude. The amplitude is equal to the column index.

Compute the peak-magnitude-to-RMS ratio of the columns.

```
t = 0:0.001:1-0.001;
x = cos(2*pi*100*t)';
X = repmat(x,1,4);
amp = 1:4;
amp = repmat(amp,1e3,1);
```

```
X = X.*amp;  
Y = peak2rms(X);
```

### Peak-magnitude-to-RMS ratio of 2-D Matrix Along Specified Dimension

Create a matrix where each row is a 100-Hz sinusoid sampled at 1 kHz with a different amplitude. The amplitude is equal to the row index.

Compute the peak-magnitude-to-RMS ratio of the rows specifying the dimension equal to 2 with the DIM argument.

```
t = 0:0.001:1-0.001;  
x = cos(2*pi*100*t);  
X = repmat(x,4,1);  
amp = (1:4)';  
amp = repmat(amp,1,1e3);  
X = X.*amp;  
Y = peak2rms(X,2);
```

## More About

### Peak-magnitude-to-RMS Level

The peak-magnitude-to-RMS ratio is

$$\frac{\|X\|_{\infty}}{\sqrt{\frac{1}{N} \sum_{n=1}^N |X_n|^2}}$$

where the  $l$ -infinity norm and RMS values are computed along the specified dimension.

## References

- [1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003.

# peig

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(...,freqrange)
[... ,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  $\lambda_{\min}$ , the smallest estimated eigenvalue of the signal's

correlation matrix. Eigenvalues below the threshold  $\lambda_{\min} \times p(2)$  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                   | $[0, \pi]$  |
| Complex-valued          | 256                   | $[0, 2\pi]$                                       |

$[S, w] = \text{peig}(x, p, w)$  returns the pseudospectrum in the vector  $S$  computed at the normalized frequencies specified in vector  $w$ , which has two or more elements

$[S, w] = \text{peig}(\dots, nfft)$  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          | $(nfft/2 + 1)$        | $[0, \pi]$   |
| Real-valued             | Odd           | $(nfft + 1)/2$        | $[0, \pi)$   |
| Complex-valued          | Even or odd   | <code>nfft</code>     | $[0, 2\pi)$  |

$[S, f] = \text{peig}(x, p, nfft, fs)$  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 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   | $[0, fs)$   |

$[S, f] = \text{peig}(x, p, f, fs)$  returns the pseudospectrum in the vector  $S$  computed at the frequencies specified in vector  $f$ , which has two or more elements

$[S, f] = \text{peig}(\dots, 'corr')$  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.

$[S, f] = \text{peig}(x, p, nfft, fs, nwin, noverlap)$  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  $2 \times p(1)$  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 $nwin$ |

| Input data $x$ | Form of $nwin$         | Windowed Data   |
|----------------|------------------------|---|
| Data vector    | Vector of coefficients | Length is <code>length(nwin)</code>   |
| Data matrix    | Scalar                 | Data is not windowed.   |
| Data matrix    | Vector of coefficients | <code>length(nwin)</code> must be the same as the column length of $x$ , and <code>noverlap</code> is not used. |

See the table, Eigenvector Length Depending on Input Data and Syntax, for related information on this syntax.

---

**Note** The arguments `nwin` and `noverlap` are ignored when you include the string 'corr' in the syntax.

---

`[...] = peig(...,freqrange)` specifies the range of frequency values to include in  $f$  or  $w$ . This syntax is useful when  $x$  is real. `freqrange` can be either:

- 'onesided' — returns the one-sided PSD of a real input signal,  $x$ . If `nfft` is even,  $P_{xx}$  has length `nfft/2+1` and is computed over the interval  $[0,\pi]$ . If `nfft` is odd, the length of  $P_{xx}$  is `(nfft + 1)/2` and the frequency interval is  $[0,\pi)$ . When you specify `fs`, the intervals are  $[0,fs/2)$  and  $[0,fs/2]$  for even and odd `length(nfft)` respectively.
- 'twosided' — returns the two-sided PSD for either real or complex input,  $x$ . In this case,  $P_{xx}$  has length `nfft` and is computed over the interval  $[0,2\pi)$ . When you specify `fs`, the frequency interval is  $[0,fs)$ .
- 'centered' — returns the centered two-sided PSD for either real or complex input,  $x$ . In this case,  $P_{xx}$  has length `nfft` and is computed over the interval  $(-\pi, \pi]$  for even `length(nfft)` and  $(-\pi,\pi)$  for odd `length(nfft)`. When you specify `fs`, the frequency intervals are  $(-fs/2, fs/2]$  and  $(-fs/2,fs/2)$  for even and odd `length(nfft)` respectively.

---

**Note** You can put the string arguments `freqrange` 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.



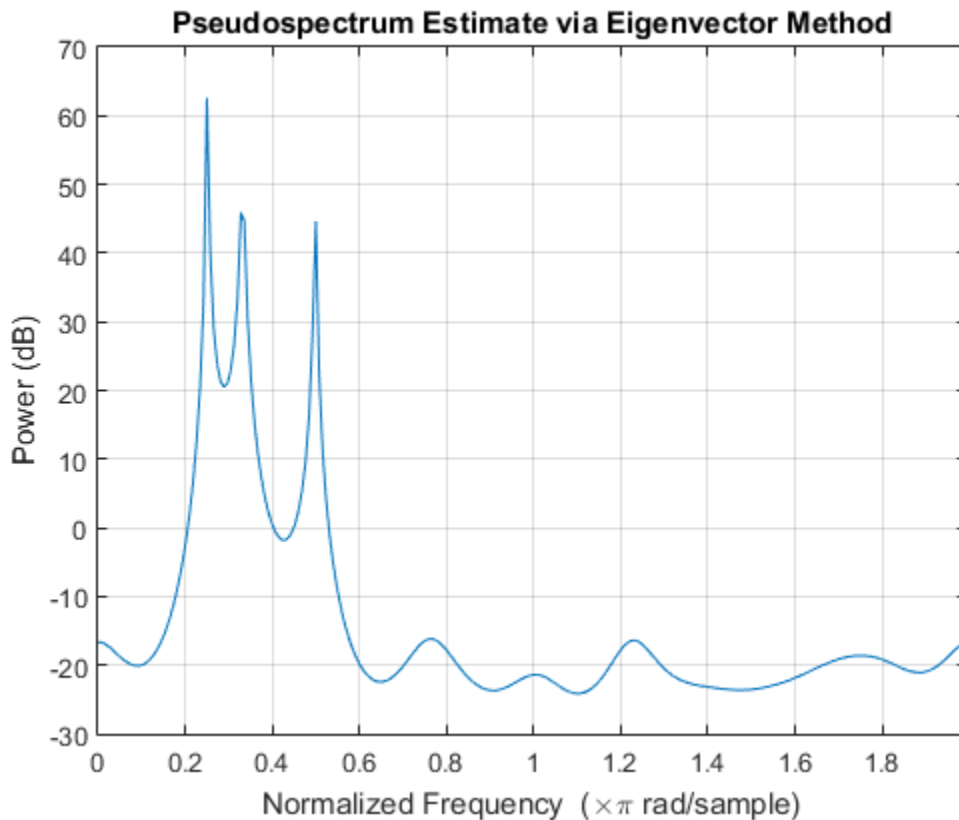
`peig(...)` with no output arguments plots the pseudospectrum in the current figure window.

## Examples

### Pseudospectrum of Sum of Sinusoids

Implement the eigenvector method to find the pseudospectrum of the sum of three sinusoids in noise. Use 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.

```
n = 0:99;  
s = exp(1i*pi/2*n)+2*exp(1i*pi/4*n)+exp(1i*pi/3*n)+randn(1,100);  
X = corrmx(s,12,'mod');  
peig(X,3,'whole')
```



## More About

### Tips

In the process of estimating the pseudospectrum, `peig` computes the noise and signal subspaces from the estimated eigenvectors  $v_j$  and eigenvalues  $\lambda_j$  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.  | $2 \times p(1)$            |
| $l$ -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.

### Algorithms

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

$$P_{ev}(f) = \frac{1}{\sum_{k=p+1}^N |v_k^H e(f)|^2 / \lambda_k}$$

where  $N$  is the dimension of the eigenvectors and  $v_k$  is the  $k$ th 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  $\lambda_k$  of the correlation matrix. The eigenvectors used span the noise subspace. The vector  $e(f)$  consists of complex exponentials, so the inner product

$$v_k^H e(f)$$

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. Lawrence. *Digital Spectral Analysis*. Englewood Cliffs, NJ: Prentice-Hall, 1987, pp.373–378.
- [2] Schmidt, R. O. “Multiple Emitter Location and Signal Parameter Estimation.” *IEEE Transactions on Antennas and Propagation*. Vol.AP-34, March, 1986, pp.276–280.
- [3] Stoica, Petre, and Randolph L. Moses. *Spectral Analysis of Signals*. Upper Saddle River, NJ: Prentice Hall, 2005.

## See Also

corrmtx | pmtm | pmusic | prony | pwelch | pburg | periodogram | rooteig | rootmusic

# periodogram

Periodogram power spectral density estimate

## Syntax

```
pxx = periodogram(x)
pxx = periodogram(x,window)
pxx = periodogram(x,window,nfft)

[pxx,w] = periodogram( ___ )
[pxx,f] = periodogram( ___ ,fs)

[pxx,w] = periodogram(x,window,w)
[pxx,f] = periodogram(x,window,f,fs)

[ ___ ] = periodogram(x,window, ___ ,freqrange)
[ ___ ] = periodogram(x,window, ___ ,spectrumtype)

[ ___ ,pxxc] = periodogram( ___ , 'ConfidenceLevel',probability)
periodogram( ___ )
```

## Description

`pxx = periodogram(x)` returns the periodogram power spectral density (PSD) estimate, `pxx`, of the input signal, `x`, found using a rectangular window. When `x` is a vector, it is treated as a single channel. When `x` is a matrix, the PSD is computed independently for each column and stored in the corresponding column of `pxx`. If `x` is real-valued, `pxx` is a one-sided PSD estimate. If `x` is complex-valued, `pxx` is a two-sided PSD estimate. The number of points, `nfft`, in the discrete Fourier transform (DFT) is the maximum of 256 or the next power of two greater than the signal length.

`pxx = periodogram(x,window)` returns the modified periodogram PSD estimate using the window, `window`. `window` is a vector the same length as `x`.

`pxx = periodogram(x,window,nfft)` uses `nfft` points in the discrete Fourier transform (DFT). If `nfft` is greater than the signal length, `x` is zero-padded to length `nfft`.

If `nfft` is less than the signal length, the signal is wrapped modulo `nfft` and summed using `datawrap`. For example, the input signal `[1 2 3 4 5 6 7 8]` with `nfft` equal to 4 results in the periodogram of `sum([1 5; 2 6; 3 7; 4 8],2)`.

`[pxx,w] = periodogram( ___ )` returns the normalized frequency vector, `w`. If `pxx` is a one-sided periodogram, `w` spans the interval  $[0,\pi]$  if `nfft` is even and  $[0,\pi)$  if `nfft` is odd. If `pxx` is a two-sided periodogram, `w` spans the interval  $[0,2\pi)$ .

`[pxx,f] = periodogram( ___, fs )` returns a frequency vector, `f`, in cycles per unit time. The sampling frequency, `fs`, is the number of samples per unit time. If the unit of time is seconds, then `f` is in cycles/second (Hz). For real-valued signals, `f` spans the interval  $[0,fs/2]$  when `nfft` is even and  $[0,fs/2)$  when `nfft` is odd. For complex-valued signals, `f` spans the interval  $[0,fs)$ .

`[pxx,w] = periodogram(x>window,w)` returns the two-sided periodogram estimates at the normalized frequencies specified in the vector, `w`. `w` must contain at least two elements.

`[pxx,f] = periodogram(x>window,f,fs)` returns the two-sided periodogram estimates at the frequencies specified in the vector, `f`. `f` must contain at least two elements. The frequencies in `f` are in cycles per unit time. The sampling frequency, `fs`, is the number of samples per unit time. If the unit of time is seconds, then `f` is in cycles/second (Hz).

`[ ___ ] = periodogram(x>window, ___, freqrange)` returns the periodogram over the frequency range specified by `freqrange`. Valid options for `freqrange` are: `'onesided'`, `'twosided'`, or `'centered'`.

`[ ___ ] = periodogram(x>window, ___, spectrumtype)` returns the PSD estimate if `spectrumtype` is specified as `'psd'` and returns the power spectrum if `spectrumtype` is specified as `'power'`.

`[ ___, pxxc ] = periodogram( ___, 'ConfidenceLevel', probability)` returns the probability  $\times$  100% confidence intervals for the PSD estimate in `pxxc`.

`periodogram( ___ )` with no output arguments plots the periodogram PSD estimate in dB per unit frequency in the current figure window.

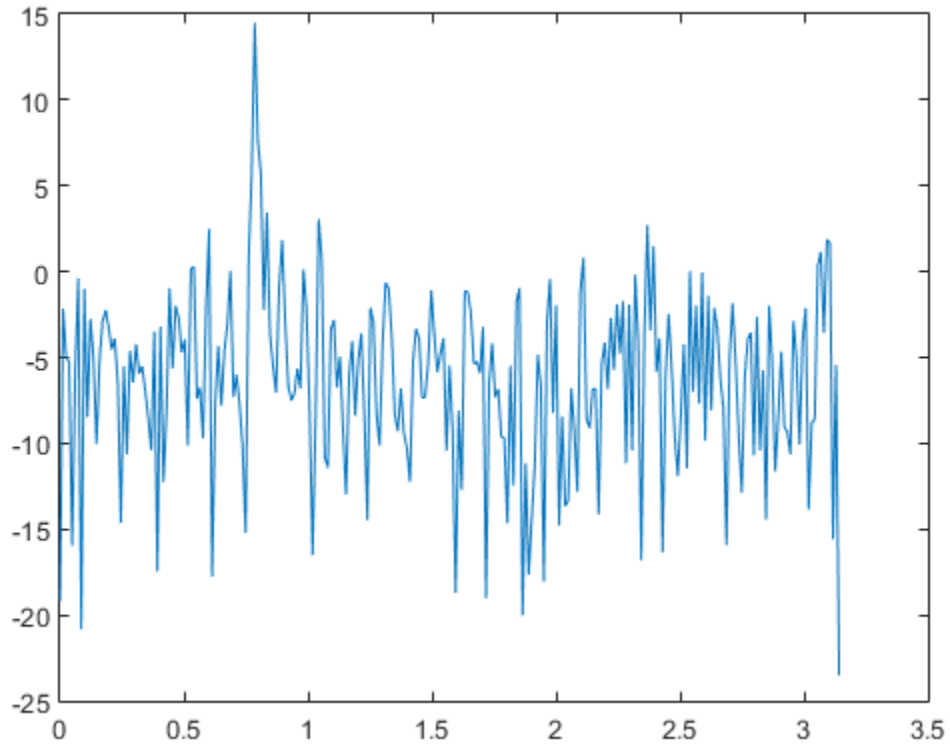
## Examples

### Periodogram Using Default Inputs

Obtain the periodogram of an input signal consisting of a discrete-time sinusoid with an angular frequency of  $\pi/4$  rad/sample with additive  $N(0, 1)$  white noise.

Create a sine wave with an angular frequency of  $\pi/4$  rad/sample with additive  $N(0, 1)$  white noise. The signal is 320 samples in length. Obtain the periodogram using the default rectangular window and DFT length. The DFT length is the next power of two greater than the signal length, or 512 points. Because the signal is real-valued and has even length, the periodogram is one-sided and there are  $512/2+1$  points.

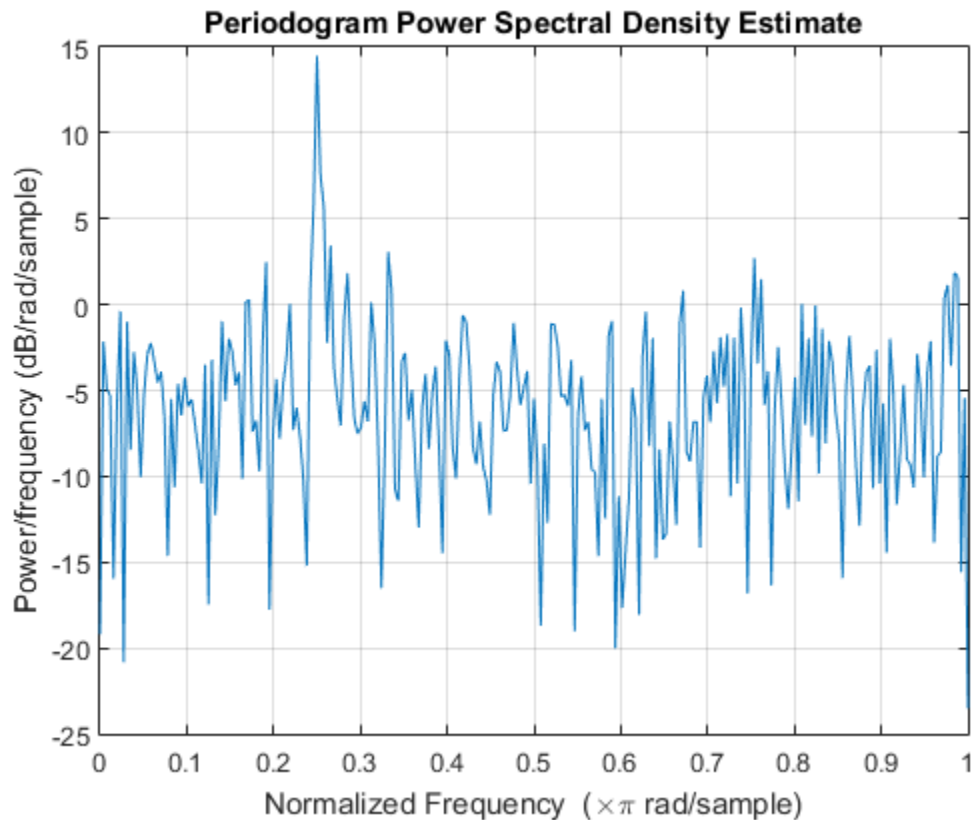
```
n = 0:319;  
x = cos(pi/4*n)+randn(size(n));  
[pxx,w] = periodogram(x);  
plot(w,10*log10(pxx))
```



Repeat the plot using `periodogram` with no outputs.

```
periodogram(x)
```





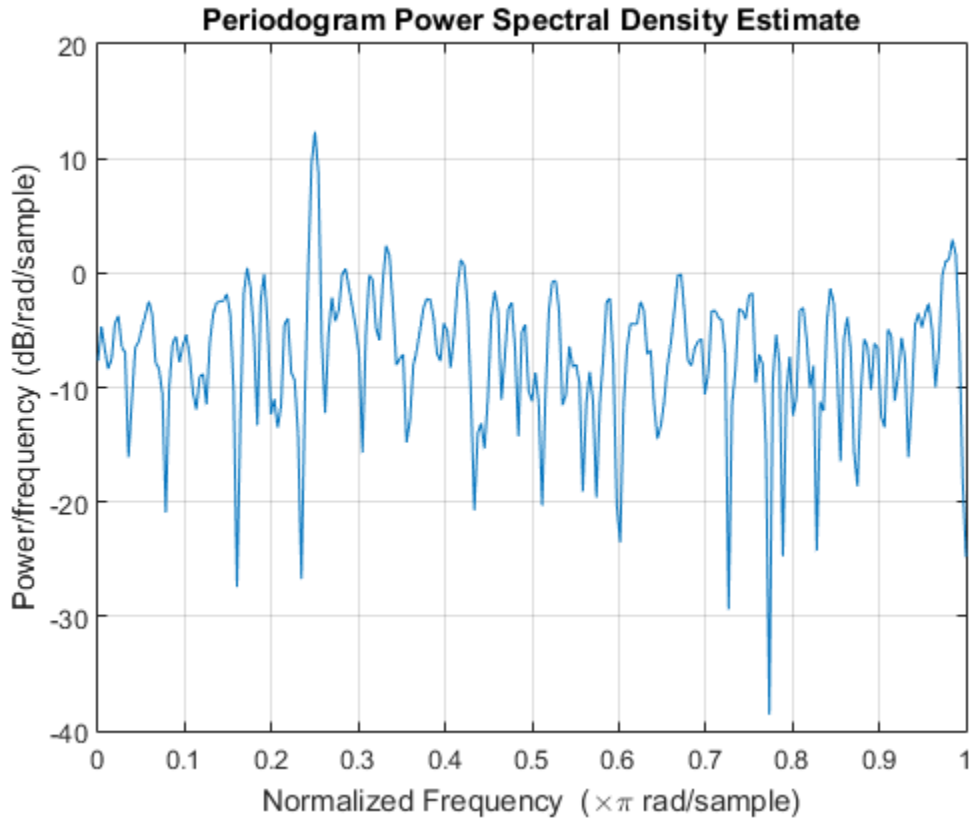
### Modified Periodogram with Hamming Window

Obtain the modified periodogram of an input signal consisting of a discrete-time sinusoid with an angular frequency of  $\pi/4$  radians/sample with additive  $N(0, 1)$  white noise.

Create a sine wave with an angular frequency of  $\pi/4$  radians/sample with additive  $N(0, 1)$  white noise. The signal is 320 samples in length. Obtain the modified periodogram using a Hamming window and default DFT length. The DFT length is the next power of two greater than the signal length, or 512 points. Because the signal is real-valued and has even length, the periodogram is one-sided and there are  $512/2+1$  points.

`n = 0:319;`

```
x = cos(pi/4*n)+randn(size(n));  
periodogram(x,hamming(length(x)))
```



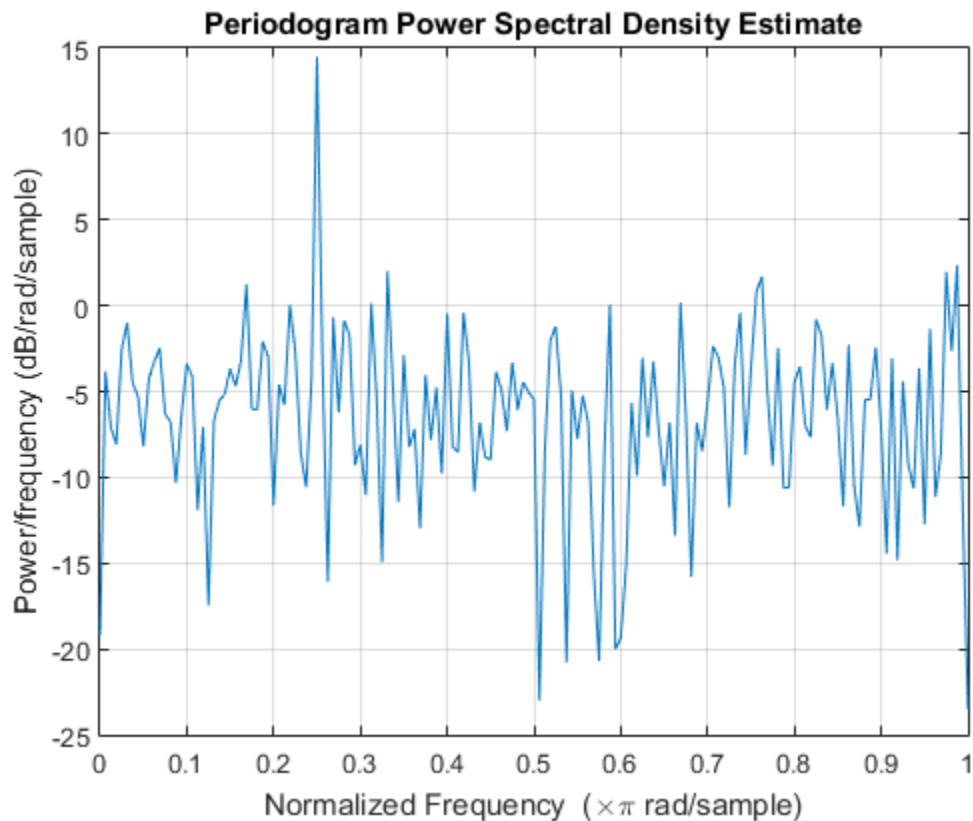
### DFT Length Equal to Signal Length

Obtain the periodogram of an input signal consisting of a discrete-time sinusoid with an angular frequency of  $\pi/4$  radians/sample with additive  $N(0, 1)$  white noise. Use a DFT length equal to the signal length.

Create a sine wave with an angular frequency of  $\pi/4$  radians/sample with additive  $N(0, 1)$  white noise. The signal is 320 samples in length. Obtain the periodogram using the default rectangular window and DFT length equal to the signal length. Because the

signal is real-valued, the one-sided periodogram is returned by default with a length equal to  $320/2+1$ .

```
n = 0:319;  
x = cos(pi/4*n)+randn(size(n));  
nfft = length(x);  
periodogram(x,[],nfft)
```



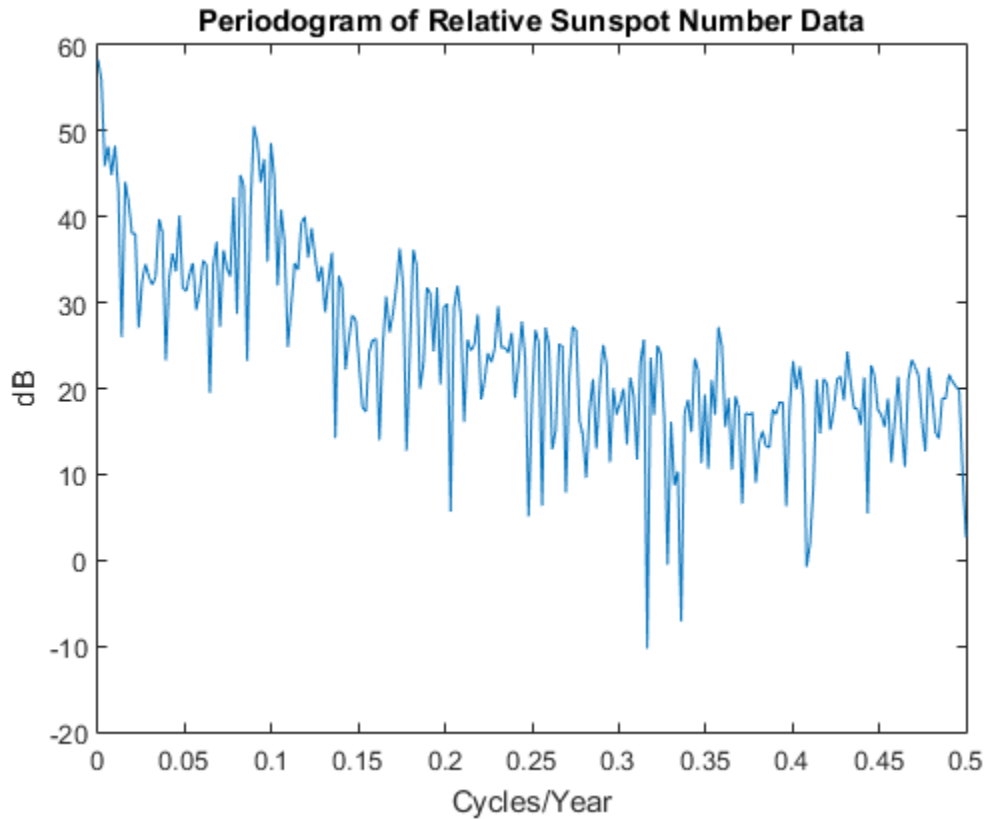
### Periodogram of Relative Sunspot Numbers

Obtain the periodogram of the Wolf (relative sunspot) number data sampled yearly between 1700 and 1987.

Load the relative sunspot number data. Obtain the periodogram using the default rectangular window and number of DFT points (512 in this example). The sample rate for these data is 1 sample/year. Plot the periodogram.

```
load sunspot.dat
relNums=sunspot(:,2);

[pxx,f] = periodogram(relNums,[],[],1);
plot(f,10*log10(pxx))
xlabel('Cycles/Year')
ylabel('dB')
title('Periodogram of Relative Sunspot Number Data')
```



You see in the preceding figure that there is a peak in the periodogram at approximately 0.1 cycles/year, which indicates a period of approximately 10 years.

### Periodogram at a Given Set of Normalized Frequencies

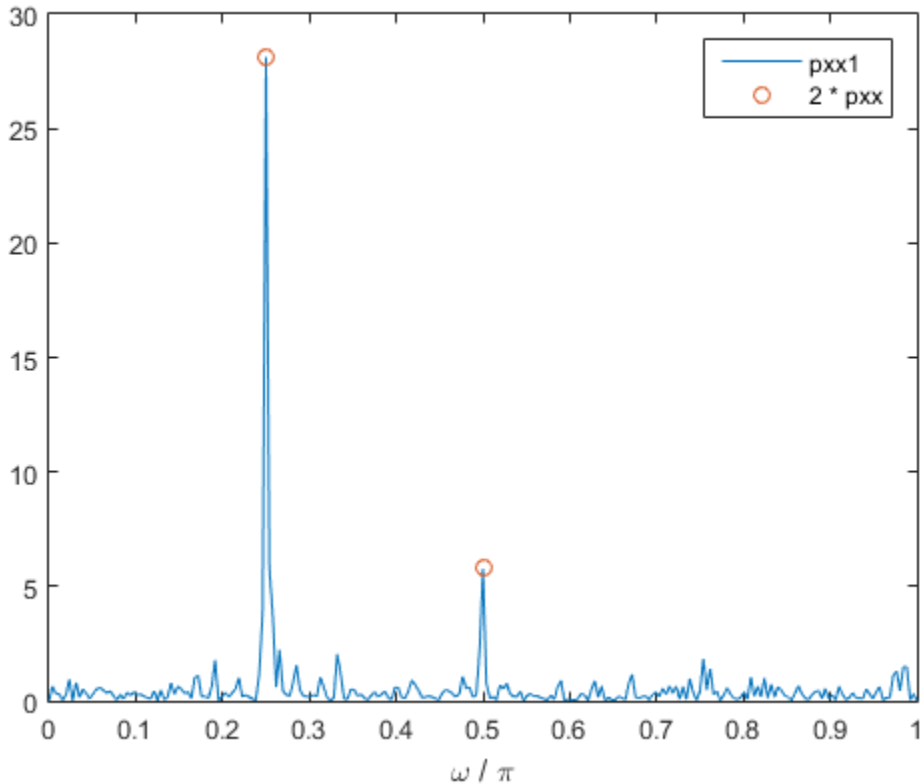
Obtain the periodogram of an input signal consisting of two discrete-time sinusoids with an angular frequencies of  $\pi/4$  and  $\pi/2$  rad/sample in additive  $N(0, 1)$  white noise. Obtain the two-sided periodogram estimates at  $\pi/4$  and  $\pi/2$  rad/sample. Compare the result to the one-sided periodogram.

```
n = 0:319;
x = cos(pi/4*n)+0.5*sin(pi/2*n)+randn(size(n));

[pxx,w] = periodogram(x,[],[pi/4 pi/2]);
pxx
[pxx1,w1] = periodogram(x);
plot(w1/pi,pxx1,w/pi,2*pxx,'o')
legend('pxx1','2 * pxx')
xlabel('\omega / \pi')
```

pxx =

```
14.0589    2.8872
```



The periodogram values obtained are 1/2 the values in the one-sided periodogram. When you evaluate the periodogram at a specific set of frequencies, the output is a two-sided estimate.

### Periodogram at a Given Set of Cyclical Frequencies

Create a signal consisting of two sine waves with frequencies of 100 and 200 Hz in  $N(0,1)$  white additive noise. The sampling frequency is 1 kHz. Obtain the two-sided periodogram at 100 and 200 Hz.

```
fs = 1000;
t = 0:0.001:1-0.001;
x = cos(2*pi*100*t)+sin(2*pi*200*t)+randn(size(t));
```

```
freq = [100 200];
pxx = periodogram(x,[],freq,fs)
```

```
pxx =
    0.2647    0.2313
```

### Upper and Lower 95%-Confidence Bounds

The following example illustrates the use of confidence bounds with the periodogram. While not a necessary condition for statistical significance, frequencies in the periodogram where the lower confidence bound exceeds the upper confidence bound for surrounding PSD estimates clearly indicate significant oscillations in the time series.

Create a signal consisting of the superposition of 100 Hz and 150 Hz sine waves in additive white  $N(0,1)$  noise. The amplitude of the two sine waves is 1. The sampling frequency is 1 kHz.

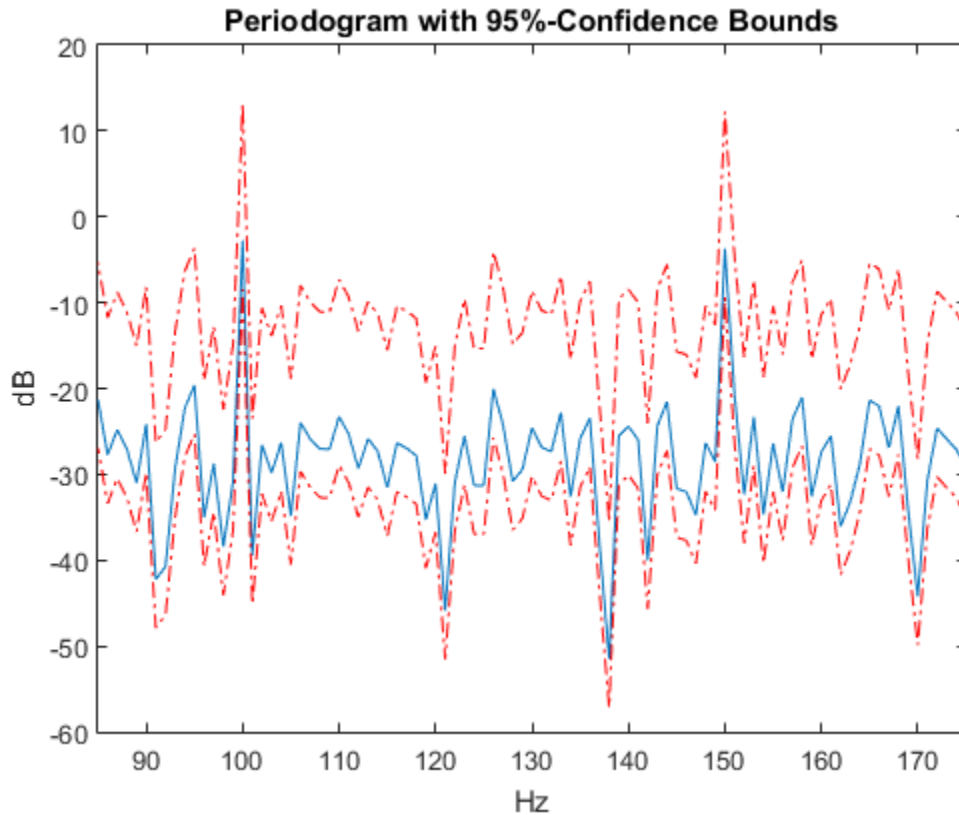
```
fs = 1000;
t = 0:0.001:1-0.001;
x = cos(2*pi*100*t)+sin(2*pi*150*t)+randn(size(t));
```

Obtain the periodogram with 95%-confidence bounds. Plot the periodogram along with the confidence interval and zoom in on the frequency region of interest near 100 and 150 Hz.

```
[pxx,f,pxxc] = periodogram(x,rectwin(length(x)),length(x),fs,...
    'ConfidenceLevel', 0.95);

plot(f,10*log10(pxx))
hold on
plot(f,10*log10(pxxc),'r-.')

xlim([85 175])
xlabel('Hz')
ylabel('dB')
title('Periodogram with 95%-Confidence Bounds')
```



At 100 and 150 Hz, the lower confidence bound exceeds the upper confidence bounds for surrounding PSD estimates.

### Power Estimate of Sinusoid

Estimate the power of sinusoid at a specific frequency using the 'power' option.

Create a 100 Hz sinusoid one second in duration sampled at 1 kHz. The amplitude of the sine wave is 1.8, which equates to a power of  $1.8^2/2 = 1.62$ . Estimate the power using the 'power' option.

```
fs = 1000;
t = 0:1/fs:1-1/fs;
```



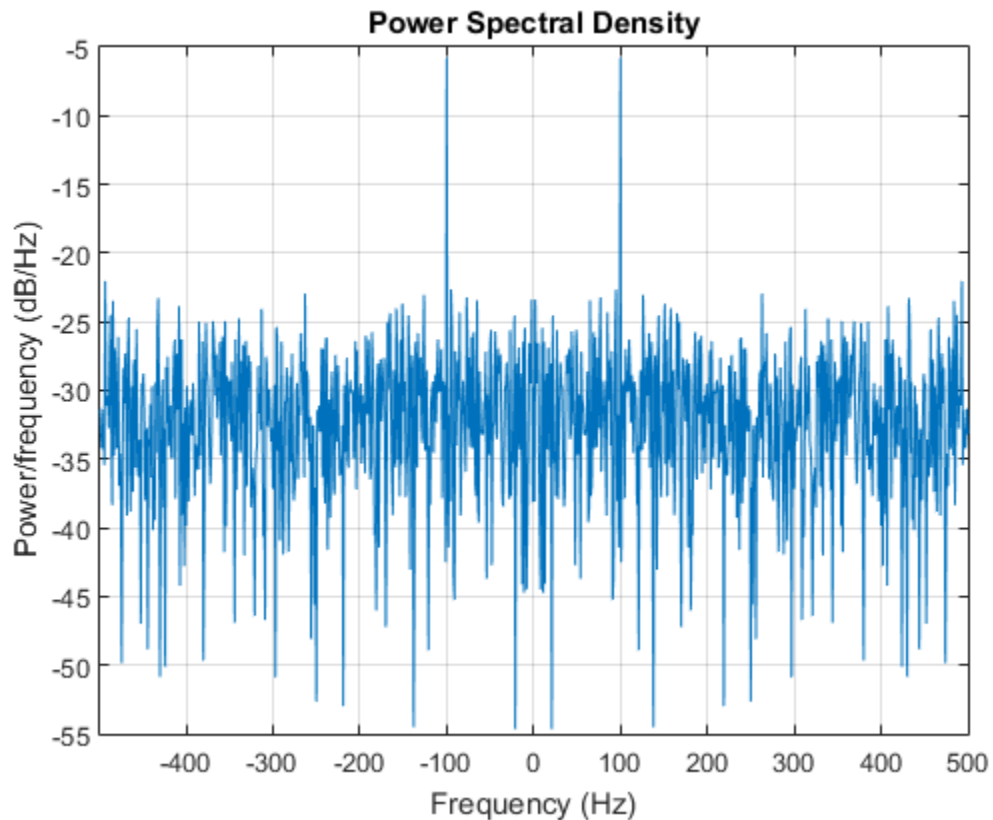
```
x = 1.8*cos(2*pi*100*t);
[pxx,f] = periodogram(x,hamming(length(x)),length(x),fs,'power');
[pwrest,idx] = max(pxx);
fprintf('The maximum power occurs at %3.1f Hz\n',f(idx));
fprintf('The power estimate is %2.2f\n',pwrest);
```

```
The maximum power occurs at 100.0 Hz
The power estimate is 1.62
```

### DC-Centered Periodogram

Obtain the periodogram of a 100 Hz sine wave in additive  $N(0,1)$  noise. The data are sampled at 1 kHz. Use the 'centered' option to obtain the DC-centered periodogram and plot the result.

```
fs = 1000;
t = 0:0.001:1-0.001;
x = cos(2*pi*100*t)+randn(size(t));
periodogram(x,[],length(x),fs,'centered')
```



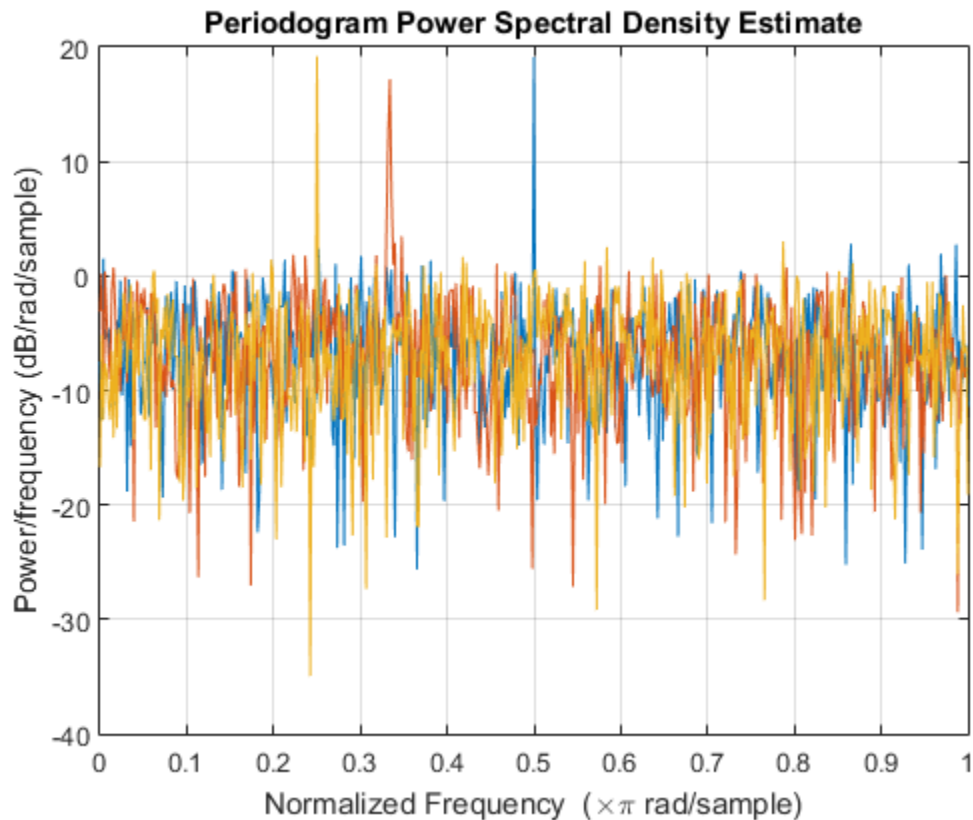
### Periodogram PSD Estimate of a Multichannel Signal

Generate 1024 samples of a multichannel signal consisting of three sinusoids in additive  $N(0, 1)$  white Gaussian noise. The sinusoids' frequencies are  $\pi/2$ ,  $\pi/3$ , and  $\pi/4$  rad/sample. Estimate the PSD of the signal using the periodogram and plot it.

```
N = 1024;
n = 0:N-1;

w = pi./[2;3;4];
x = cos(w*n)' + randn(length(n),3);

periodogram(x)
```



- “Bias and Variability in the Periodogram”
- “Power Spectral Density Estimates Using FFT”

## Input Arguments

### **x** — Input signal

vector | matrix

Input signal, specified as a row or column vector, or as a matrix. If **x** is a matrix, then its columns are treated as independent channels.

Example: `cos(pi/4*(0:159))+randn(1,160)` is a single-channel row-vector signal.

Example: `cos(pi./[4;2]*(0:159))'+randn(160,2)` is a two-channel signal.

Data Types: `single` | `double`

Complex Number Support: Yes

### **window** — Window

`rectwin(length(x))` (default) | [] | vector

Window, specified as a row or column vector the same length as the input signal. If you specify window as empty, the default rectangular window is used.

Data Types: `single` | `double`

### **nfft** — Number of DFT points

`max(256,2^nextpow2(length(x)))` (default) | integer | []

Number of DFT points, specified as a positive integer. For a real-valued input signal, `x`, the PSD estimate, `pxx` has length  $(nfft/2 + 1)$  if `nfft` is even, and  $(nfft + 1)/2$  if `nfft` is odd. For a complex-valued input signal, `x`, the PSD estimate always has length `nfft`. If `nfft` is specified as empty, the default `nfft` is used.

Data Types: `single` | `double`

### **fs** — Sampling frequency

positive scalar

Sampling frequency, specified as a positive scalar. The sampling frequency is the number of samples per unit time. If the unit of time is seconds, the sampling frequency has units of hertz.

### **w** — Normalized frequencies

vector

Normalized frequencies, specified as a row or column vector with at least 2 elements. Normalized frequencies are in rad/sample.

Example: `w = [pi/4 pi/2]`

Data Types: `double`

### **f** — Cyclical frequencies

vector

Cyclical frequencies, specified as a row or column vector with at least 2 elements. The frequencies are in cycles per unit time. The unit time is specified by the sampling frequency, `fs`. If `fs` has units of samples/second, then `f` has units of Hz.

Example: `fs = 1000; f = [100 200]`

Data Types: `double`

### **freqrange** — Frequency range for PSD estimate

`'onesided'` | `'twosided'` | `'centered'`

Frequency range for the PSD estimate, specified as a one of `'onesided'`, `'twosided'`, or `'centered'`. The default is `'onesided'` for real-valued signals and `'twosided'` for complex-valued signals. The frequency ranges corresponding to each option are

- `'onesided'` — returns the one-sided PSD estimate of a real-valued input signal, `x`. If `nfft` is even, `pxx` has length  $nfft/2 + 1$  and is computed over the interval  $[0, \pi]$  rad/sample. If `nfft` is odd, the length of `pxx` is  $(nfft + 1)/2$  and the interval is  $[0, \pi]$  rad/sample. When `fs` is optionally specified, the corresponding intervals are  $[0, fs/2]$  cycles/unit time and  $[0, fs/2)$  cycles/unit time for even and odd length `nfft` respectively.
- `'twosided'` — returns the two-sided PSD estimate for either the real-valued or complex-valued input, `x`. In this case, `pxx` has length `nfft` and is computed over the interval  $[0, 2\pi]$  rad/sample. When `fs` is optionally specified, the interval is  $[0, fs)$  cycles/unit time.
- `'centered'` — returns the centered two-sided PSD estimate for either the real-valued or complex-valued input, `x`. In this case, `pxx` has length `nfft` and is computed over the interval  $(-\pi, \pi]$  rad/sample for even length `nfft` and  $(-\pi, \pi)$  rad/sample for odd length `nfft`. When `fs` is optionally specified, the corresponding intervals are  $(-fs/2, fs/2]$  cycles/unit time and  $(-fs/2, fs/2)$  cycles/unit time for even and odd length `nfft` respectively.

Data Types: `char`

### **spectrumtype** — Power spectrum scaling

`'psd'` (default) | `'power'`

Power spectrum scaling, specified as one of `'psd'` or `'power'`. Omitting the `spectrumtype`, or specifying `'psd'`, returns the power spectral density. Specifying `'power'` scales each estimate of the PSD by the equivalent noise bandwidth of the window. Use the `'power'` option to obtain an estimate of the power at each frequency.

Data Types: `char`

**probability — Confidence interval for PSD estimate**

0.95 (default) | scalar in the range (0,1)

Coverage probability for the true PSD, specified as a scalar in the range (0,1). The output, `pxxc`, contains the lower and upper bounds of the  $\text{probability} \times 100\%$  interval estimate for the true PSD.

## Output Arguments

**pxx — PSD estimate**

vector | matrix

PSD estimate, returned as a real-valued, nonnegative column vector or matrix. Each column of `pxx` is the PSD estimate of the corresponding column of `x`. The units of the PSD estimate are in squared magnitude units of the time series data per unit frequency. For example, if the input data is in volts, the PSD estimate is in units of squared volts per unit frequency. For a time series in volts, if you assume a resistance of  $1 \Omega$  and specify the sampling frequency in hertz, the PSD estimate is in watts per hertz.

Data Types: `single` | `double`**w — Normalized frequencies**

vector

Normalized frequencies, returned as a real-valued column vector. If `pxx` is a one-sided PSD estimate, `w` spans the interval  $[0, \pi]$  if `nfft` is even and  $[0, \pi)$  if `nfft` is odd. If `pxx` is a two-sided PSD estimate, `w` spans the interval  $[0, 2\pi)$ . For a DC-centered PSD estimate, `f` spans the interval  $(-\pi, \pi]$  rad/sample for even length `nfft` and  $(-\pi, \pi)$  radians/sample for odd length `nfft`.

Data Types: `double`**f — Cyclical frequencies**

vector

Cyclical frequencies, returned as a real-valued column vector. For a one-sided PSD estimate, `f` spans the interval  $[0, \text{fs}/2]$  when `nfft` is even and  $[0, \text{fs}/2)$  when `nfft` is odd. For a two-sided PSD estimate, `f` spans the interval  $[0, \text{fs})$ . For a DC-centered PSD estimate, `f` spans the interval  $(-\text{fs}/2, \text{fs}/2]$  cycles/unit time for even length `nfft` and  $(-\text{fs}/2, \text{fs}/2)$  cycles/unit time for odd length `nfft`.

Data Types: double

### **pxxc — Confidence bounds**

matrix

Confidence bounds, returned as a matrix with real-valued elements. The row size of the matrix is equal to the length of the PSD estimate, pxx. pxxc has twice as many columns as pxx. Odd-numbered columns contain the lower bounds of the confidence intervals, and even-numbered columns contain the upper bounds. Thus, pxxc(m, 2\*n-1) is the lower confidence bound and pxxc(m, 2\*n) is the upper confidence bound corresponding to the estimate pxx(m, n). The coverage probability of the confidence intervals is determined by the value of the probability input.

Data Types: single | double

## More About

### Periodogram

The periodogram is a nonparametric estimate of the power spectral density (PSD) of a wide-sense stationary random process. The periodogram is the Fourier transform of the biased estimate of the autocorrelation sequence. For a signal,  $x_n$ , sampled at fs samples per unit time, the periodogram is defined as

$$\hat{P}(f) = \frac{\Delta t}{N} \left| \sum_{n=0}^{N-1} x_n e^{-i2\pi fn} \right|^2 \quad -1/2\Delta t < f \leq 1/2\Delta t$$

where  $\Delta t$  is the sampling interval. For a one-sided periodogram, the values at all frequencies except 0 and the Nyquist,  $1/2\Delta t$ , are multiplied by 2 so that the total power is conserved.

If the frequencies are in radians/sample, the periodogram is defined as

$$\hat{P}(f) = \frac{1}{2\pi N} \left| \sum_{n=0}^{N-1} x_n e^{-i\omega n} \right|^2 \quad -\pi < \omega \leq \pi$$

The frequency range in the preceding equations has variations depending on the value of the freqrange argument. See the description of freqrange in “Input Arguments” on page 1-1045.

The integral of the true PSD,  $P(f)$ , over one period,  $1/\Delta t$  for cyclical frequency and  $2\pi$  for normalized frequency, is equal to the variance of the wide-sense stationary random process.

$$\sigma^2 = \int_{-1/2\Delta t}^{1/2\Delta t} P(f) df$$

For normalized frequencies, replace the limits of integration appropriately.

### Modified Periodogram

The modified periodogram multiplies the input time series by a window function. A suitable window function is nonnegative and decays to zero at the beginning and end points. Multiplying the time series by the window function *tapers* the data gradually on and off and helps to alleviate the leakage in the periodogram. See “Bias and Variability in the Periodogram” for an example.

If  $h_n$  is a window function, the modified periodogram is defined by

$$\hat{P}(f) = \frac{\Delta t}{N} \left| \sum_{n=0}^{N-1} h_n x_n e^{-i2\pi f n} \right|^2 \quad -1/2\Delta t < f \leq 1/2\Delta t$$

where  $\Delta t$  is the sampling interval.

If the frequencies are in radians/sample, the modified periodogram is defined as

$$\hat{P}(f) = \frac{1}{2\pi N} \left| \sum_{n=0}^{N-1} h_n x_n e^{-i\omega n} \right|^2 \quad -\pi < \omega \leq \pi$$

The frequency range in the preceding equations has variations depending on the value of the frequency argument. See the description of frequency in “Input Arguments” on page 1-1045.

- “Nonparametric Methods”

### See Also

bandpower | pburg | pcov | pmcov | pmtm | pwelch | sfdr



# phasedelay

Phase delay of digital filter

## Syntax

```
[phi,w] = phasedelay(b,a,n)
[phi,w] = phasedelay(sos,n)
[phi,w] = phasedelay(d,n)
[phi,w] = phasedelay(...,n,'whole')
phi = phasedelay(...,w)
[phi,f] = phasedelay(...,n,fs)
[phi,f] = phasedelay(...,n,'whole',fs)
phi = phasedelay(...,f,fs)
[phi,w,s] = phasedelay(...)
[phi,f,s] = phasedelay(...)
phasedelay(...)
```

## Description

`[phi,w] = phasedelay(b,a,n)` returns the  $n$ -point phase delay response vector, `phi`, and the  $n$ -point frequency vector in radians/sample, `w`, 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. For best results, set  $n$  to a value greater than the filter order.

`[phi,w] = phasedelay(sos,n)` returns the  $n$ -point phase delay response for the second-order sections matrix, `sos`. `sos` is a  $K$ -by-6 matrix, where the number of sections,  $K$ , must be greater than or equal to 2. If the number of sections is less than 2, `phasedelay` considers the input to be a numerator vector, `b`. Each row of `sos` corresponds to the coefficients of a second order (biquad) filter. The  $i$ th row of the `sos` matrix corresponds to `[bi(1) bi(2) bi(3) ai(1) ai(2) ai(3)]`.

`[phi,w] = phasedelay(d,n)` returns the  $n$ -point phase delay response of the digital filter, `d`. Use `designfilt` to generate `d` based on frequency-response specifications.

`[phi,w] = phasedelay(...,n,'whole')` uses  $n$  equally spaced points around the whole unit circle.

`phi = phasedelay(...,w)` returns the phase delay response at frequencies specified, in radians/sample, in vector `w`. The frequencies are normally between 0 and  $\pi$ . `w` must contain at least two elements.

`[phi,f] = phasedelay(...,n,fs)` and `[phi,f] = phasedelay(...,n,'whole',fs)` return the phase delay vector `f` (in Hz), using the sampling frequency `fs` (in Hz). `f` must contain at least two elements.

`phi = phasedelay(...,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(...)` with no output arguments plots the phase delay response versus frequency.

---

**Note:** If the input to `phasedelay` is single precision, the phase delay response is calculated using single-precision arithmetic. The output, `phi`, is single precision.

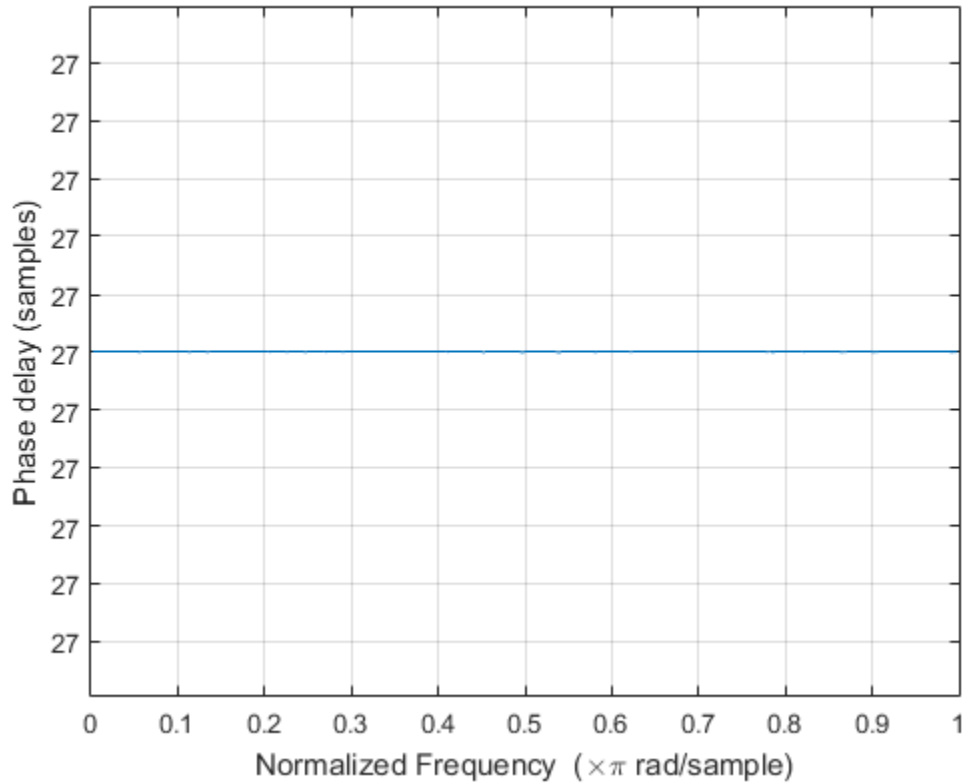
---

## Examples

### Phase Delay Response of an FIR Filter

Use constrained least squares to design a lowpass FIR filter of order 54 and normalized cutoff frequency 0.3. Specify the passband ripple and stopband attenuation as 0.02 and 0.08, respectively, expressed in linear units. Compute and plot the phase delay response of the filter.

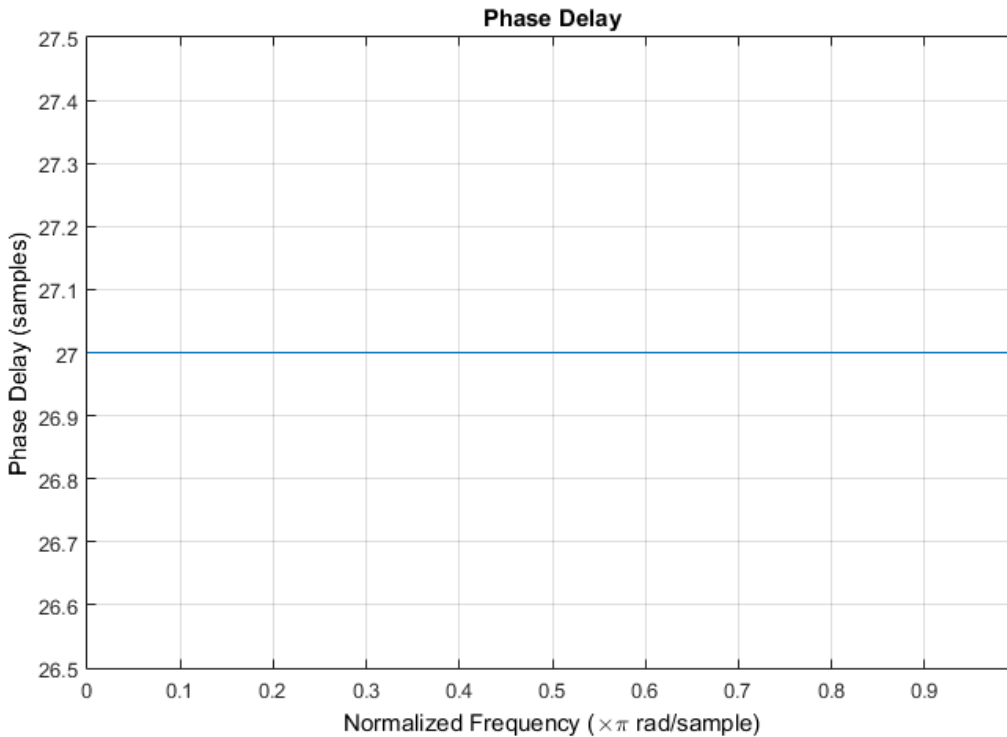
```
Ap = 0.02;  
As = 0.008;  
  
b = fircls1(54,0.3,Ap,As);  
phasedelay(b)
```



Repeat the example using `designfilt`. Keep in mind that this function expresses the ripples in decibels.

```
Apd = 40*log10((1+Ap)/(1-Ap));
Asd = -20*log10(As);
```

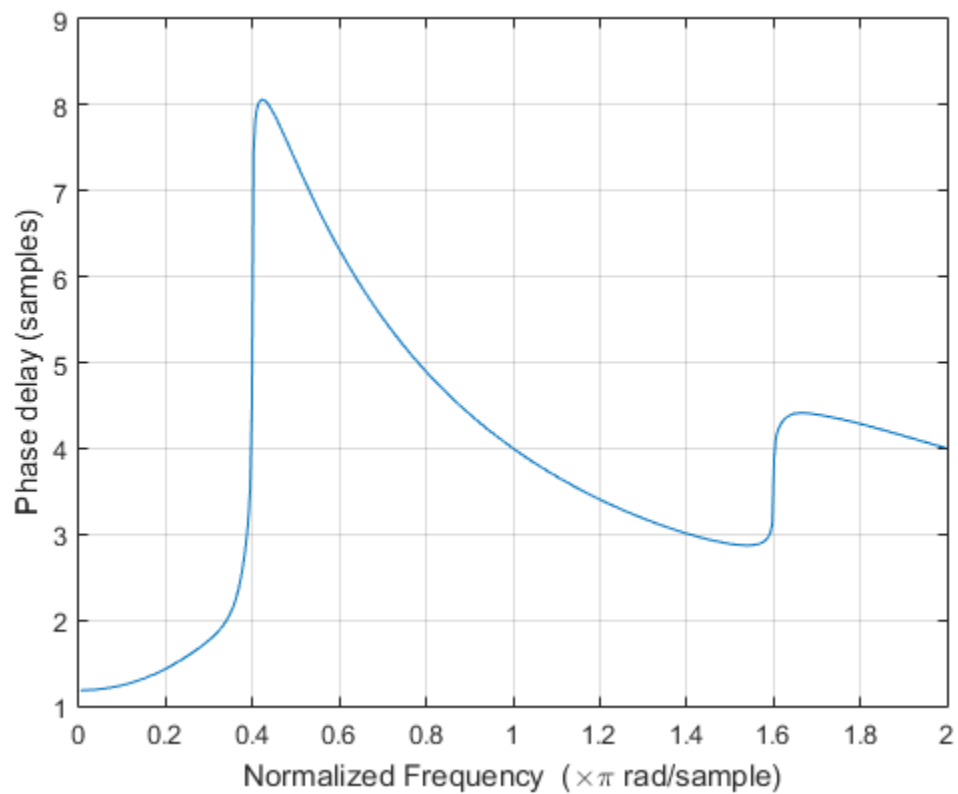
```
d = designfilt('lowpassfir','FilterOrder',54,'CutoffFrequency',0.3, ...
               'PassbandRipple',Apd,'StopbandAttenuation',Asd);
phasedelay(d)
```



### Phase Delay Response of an Elliptic Filter

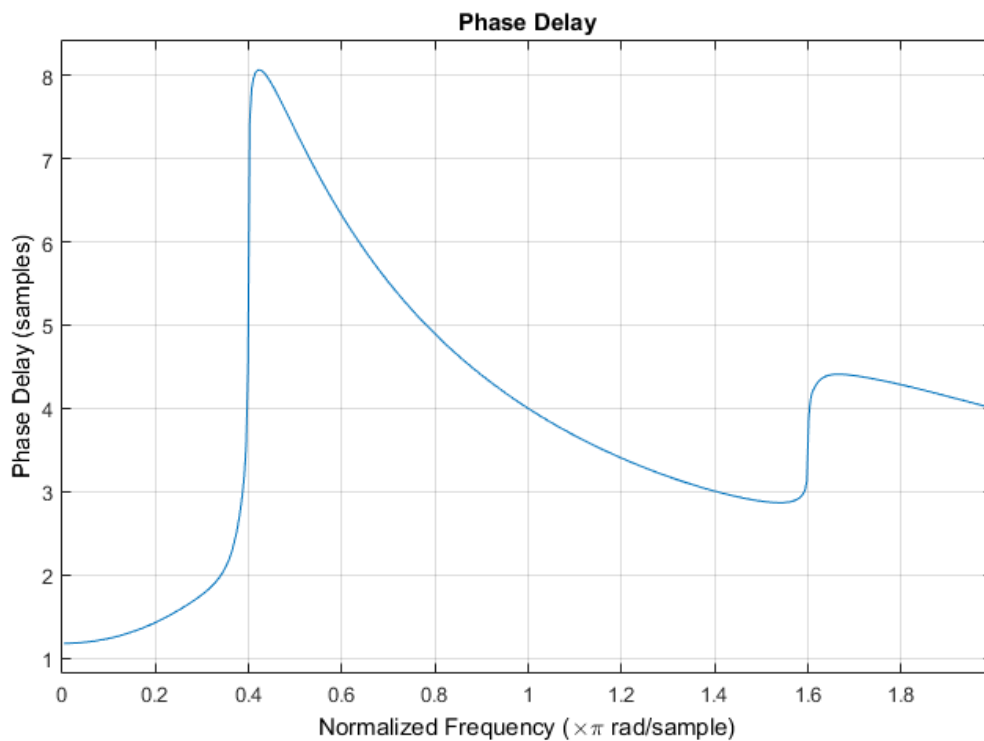
Design an elliptic filter of order 10 and normalized passband frequency 0.4. Specify a passband ripple of 0.5 dB and a stopband attenuation of 20 dB. Display the phase delay response of the filter over the complete unit circle.

```
[b,a] = ellip(10,0.5,20,0.4);  
phasedelay(b,a,512,'whole')
```



Repeat the example using `designfilt`.

```
d = designfilt('lowpassiir', 'DesignMethod', 'ellip', 'FilterOrder', 10, ...  
              'PassbandFrequency', 0.4, ...  
              'PassbandRipple', 0.5, 'StopbandAttenuation', 20);  
phasedelay(d, 512, 'whole')
```



**See Also**

`designfilt` | `digitalFilter` | `freqz` | `fvtool` | `phasez` | `grpdelay`

# phasez

Phase response of digital filter

## Syntax

```
[phi,w] = phasez(b,a,n)
[phi,w] = phasez(sos,n)
[phi,w] = phasez(d,n)
[phi,w] = phasez(...,n,'whole')
phi = phasez(...,w)
[phi,f] = phasez(...,n,fs)
phi = phasez(...f,fs)
[phi,w,s] = phasez(...)
phasez(...)
```

## Description

`[phi,w] = phasez(b,a,n)` returns the  $n$ -point unwrapped phase response vector, `phi`, in radians and the frequency vector, `w`, in radians/sample for the filter coefficients specified in `b` and `a`. The values of the frequency vector, `w`, range from 0 to  $\pi$ . If `n` is omitted, the length of the phase response vector defaults to 512. For best results, set `n` to a value greater than the filter order.

`[phi,w] = phasez(sos,n)` returns the unwrapped phase response for the second order sections matrix, `sos`. `sos` is a  $K$ -by-6 matrix, where the number of sections,  $K$ , must be greater than or equal to 2. If the number of sections is less than 2, `phasez` considers the input to be the numerator vector, `b`. Each row of `sos` corresponds to the coefficients of a second-order (biquad) filter. The  $i$ th row of the `sos` matrix corresponds to `[bi(1) bi(2) bi(3) ai(1) ai(2) ai(3)]`.

`[phi,w] = phasez(d,n)` returns the unwrapped phase response for the digital filter, `d`. Use `designfilt` to generate `d` based on frequency-response specifications.

`[phi,w] = phasez(...,n,'whole')` returns frequency and unwrapped phase response vectors evaluated at  $n$  equally-spaced points around the unit circle from 0 to  $2\pi$  radians/sample.

`phi = phasez(...,w)` returns the unwrapped phase response in radians at frequencies specified in `w` (radians/sample). The frequencies are normally between 0 and  $\pi$ . The vector `w` must have at least two elements.

`[phi,f] = phasez(...,n,fs)` return the unwrapped phase vector `phi` in radians and the frequency vector in hertz. The frequency vector ranges from 0 to the Nyquist frequency, `fs/2`. If the 'whole' option is used, the frequency vector ranges from 0 to the sampling frequency.

`phi = phasez(...,f,fs)` return the phase response in radians at the frequencies specified in the vector `f` (in hertz) using the sampling frequency `fs` (in hertz). The vector `f` must have at least two elements.

`[phi,w,s] = phasez(...)` return plotting information, where `s` is a structure array with fields you can change to display different frequency response plots.

`phasez(...)` with no output arguments plots the phase response of the filter. If you input the filter coefficients or second order sections matrix, the current figure window is used. If you input a `digitalFilter`, the step response is displayed in `fvtool`.

---

**Note:** If the input to `phasez` is single precision, the phase response is calculated using single-precision arithmetic. The output, `phi`, is single precision.

---

## Examples

### Phase Response of an FIR Filter

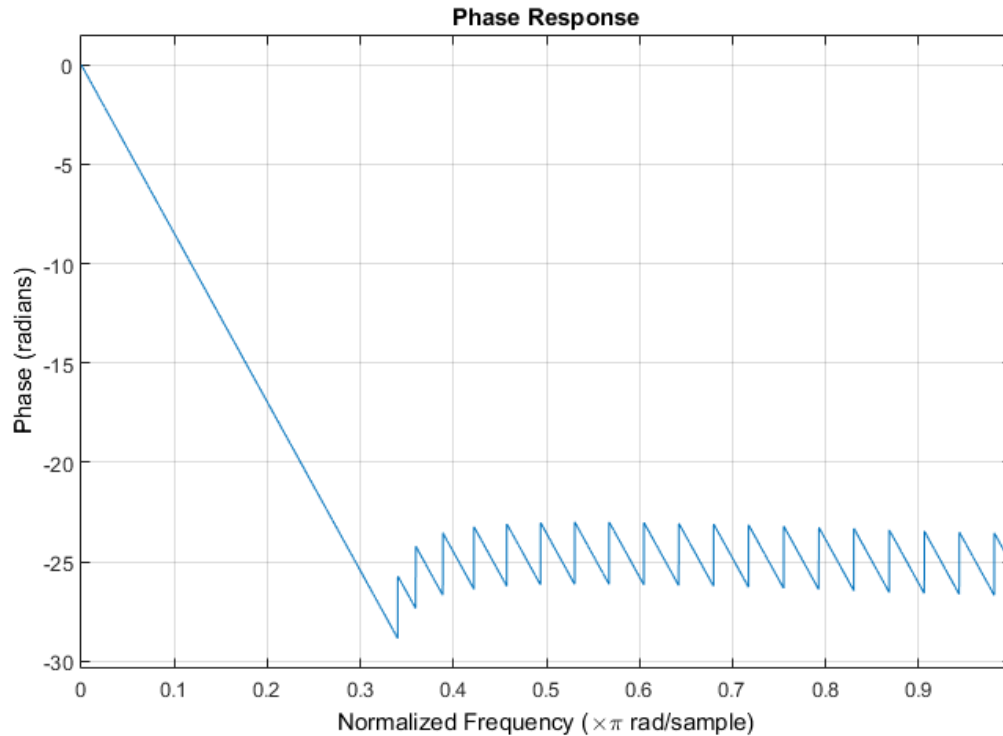
Use `designfilt` to design an FIR filter of order 54, normalized cutoff frequency  $0.3\pi$  rad/s, passband ripple 0.7 dB, and stopband attenuation 42 dB. Use the method of constrained least squares. Display the phase response of the filter.

```
Nf = 54;  
Fc = 0.3;  
Ap = 0.7;  
As = 42;
```

```
d = designfilt('lowpassfir','CutoffFrequency',Fc,'FilterOrder',Nf, ...  
              'PassbandRipple',Ap,'StopbandAttenuation',As, ...  
              'DesignMethod','cls');
```



phasez(d)

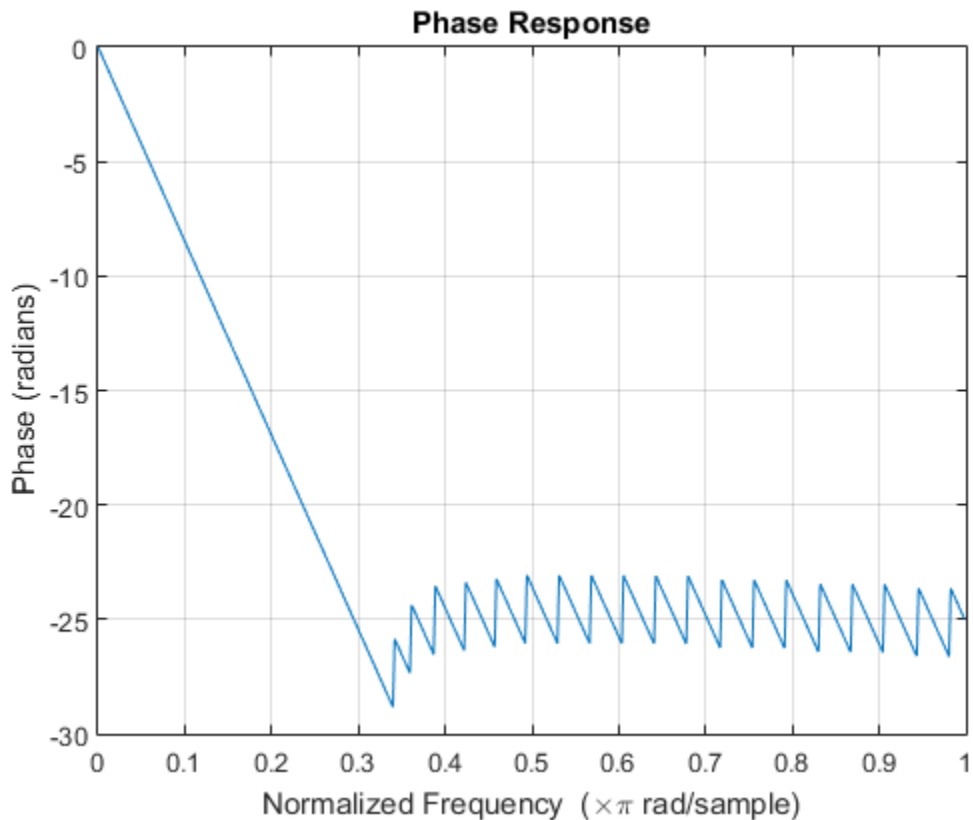


Design the same filter using `fircls1`. Keep in mind that `fircls1` uses linear units to measure the ripple and attenuation.

```
pAp = 10^(Ap/40);
Apl = (pAp - 1)/(pAp + 1);
```

```
pAs = 10^(As/20);
Asl = 1/pAs;
```

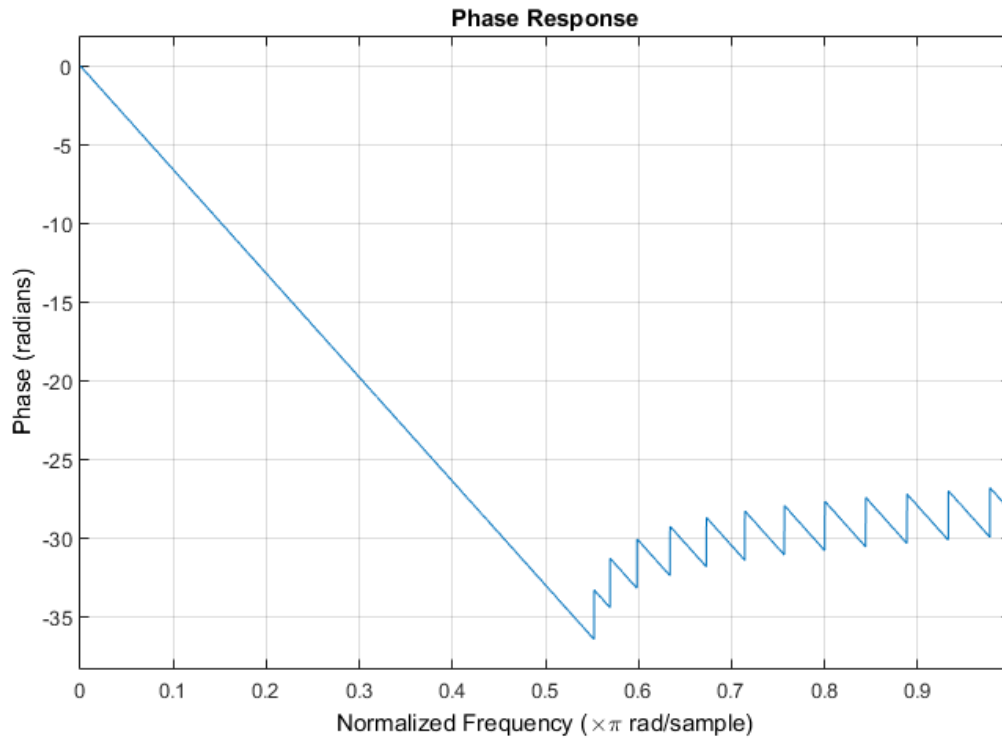
```
b = fircls1(Nf, Fc, Apl, Asl);
phasez(b)
```



### Phase Response of an Equiripple Filter

Design a lowpass equiripple filter with normalized passband frequency  $0.45\pi$  rad/s, normalized stopband frequency  $0.55\pi$  rad/s, passband ripple 1 dB, and stopband attenuation 60 dB. Display the phase response of the filter.

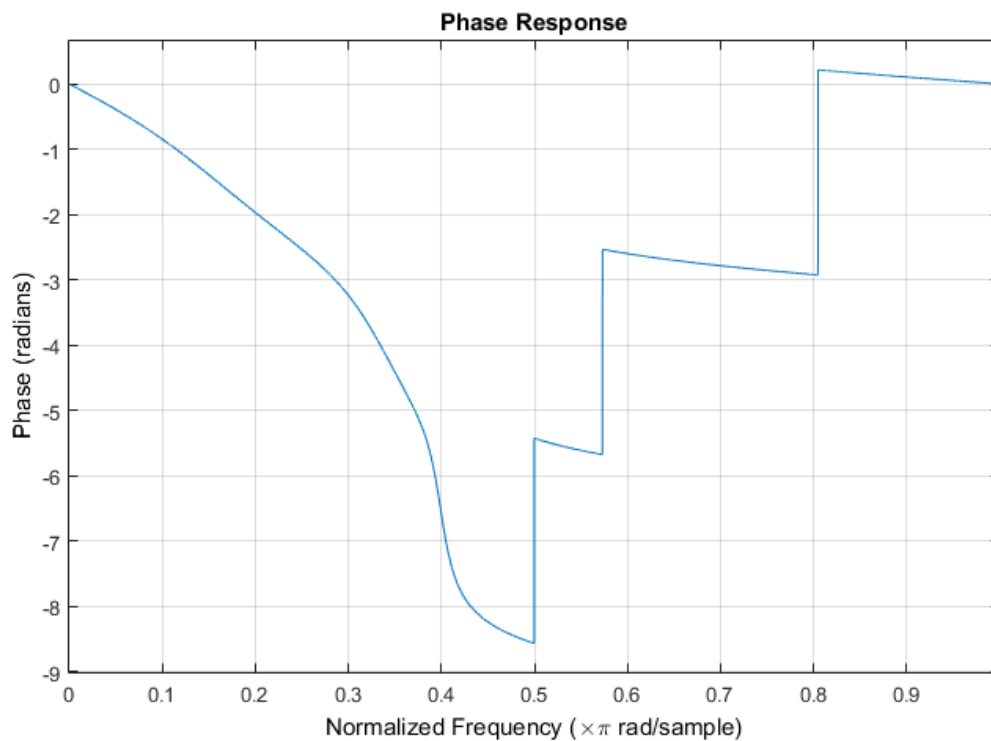
```
d = designfilt('lowpassfir', ...
              'PassbandFrequency',0.45,'StopbandFrequency',0.55, ...
              'PassbandRipple',1,'StopbandAttenuation',60, ...
              'DesignMethod','equiripple');
phasez(d)
```



### Phase Response of an Elliptic Filter

Design an elliptic lowpass IIR filter with normalized passband frequency  $0.4\pi$  rad/s, normalized stopband frequency  $0.5\pi$  rad/s, passband ripple 1 dB, and stopband attenuation 60 dB. Display the phase response of the filter.

```
d = designfilt('lowpassiir', ...
              'PassbandFrequency',0.4,'StopbandFrequency',0.5, ...
              'PassbandRipple',1,'StopbandAttenuation',60, ...
              'DesignMethod','ellip');
phasez(d)
```



**See Also**

`designfilt` | `digitalFilter` | `freqz` | `fvtool` | `phasedelay` | `grpdelay`

# plomb

Lomb-Scargle periodogram

## Syntax

```
[pxx,f] = plomb(x,t)
[pxx,f] = plomb(x,fs)

[pxx,f] = plomb( ____,fmax)
[pxx,f] = plomb( ____,fmax,ofac)

[pxx,fvec] = plomb( ____,fvec)

[ ____ ] = plomb( ____,spectrumtype)

[ ____,pth] = plomb( ____, 'Pd',pdvec)

[pxx,w] = plomb(x)

plomb( ____ )
```

## Description

`[pxx,f] = plomb(x,t)` returns the Lomb-Scargle power spectral density (PSD) estimate, `pxx`, of a signal, `x`, that is sampled at the instants specified in `t`. `t` must increase monotonically but need not be uniformly spaced. All elements of `t` must be nonnegative. `pxx` is evaluated at the frequencies returned in `f`.

- If `x` is a vector, it is treated as a single channel.
- If `x` is a matrix, then `plomb` computes the PSD independently for each column and returns it in the corresponding column of `pxx`.

`x` or `t` can contain NaNs. NaNs are treated as missing data and are excluded from the spectrum computation.

`[pxx,f] = plomb(x,fs)` treats the case where the signal is sampled uniformly, at a rate `fs`, but has missing samples. Specify the missing data using NaNs.

`[pxx, f] = plomb( ____, fmax)` estimates the PSD up to a maximum frequency, `fmax`, using any of the input arguments from previous syntaxes. If the signal is sampled at  $N$  non-NaN instants, and  $\Delta t$  is the time difference between the first and the last of them, then `pxx` is returned at `round(fmax / fmin)` points, where  $f_{\min} = 1/(4 \times N \times t_s)$  is the smallest frequency at which `pxx` is computed and the average sample time is  $t_s = \Delta t / (N - 1)$ . `fmax` defaults to  $1/(2 \times t_s)$ , which for uniformly sampled signals corresponds to the Nyquist frequency.

`[pxx, f] = plomb( ____, fmax, ofac)` specifies an integer oversampling factor, `ofac`. The use of `ofac` to interpolate or smooth a spectrum resembles the zero-padding technique for FFT-based methods. `pxx` is again returned at `round(fmax/fmin)` frequency points, but the minimum frequency considered in this case is  $1/(ofac \times N \times t_s)$ . `ofac` defaults to 4.

`[pxx, fvec] = plomb( ____, fvec)` estimates the PSD of `x` at the frequencies specified in `fvec`. `fvec` must have at least two elements. The second output argument is the same as the input `fvec`.

You cannot specify a maximum frequency or an oversampling factor if you use this syntax.

`[ ____ ] = plomb( ____, spectrumtype)` specifies the normalization of the periodogram.

- Set `spectrumtype` to `'psd'`, or leave it unspecified, to obtain `pxx` as a power spectral density.
- Set `spectrumtype` to `'power'` to get the power spectrum of the input signal.
- Set `spectrumtype` to `'normalized'` to get the standard Lomb-Scargle periodogram, which is scaled by two times the variance of `x`.

`[ ____, pth] = plomb( ____, 'Pd', pdvec)` returns the power-level threshold, `pth`, such that a peak with a value larger than `pth` has a probability `pdvec` of being a true signal peak and not the result of random fluctuations. `pdvec` can be a vector. Every element of `pdvec` must be greater than 0 and smaller than 1. Each row of `pth` corresponds to an element of `pdvec`. `pth` has the same number of channels as `x`. This option is not available if you specify the output frequencies in `fvec`.

`[pxx, w] = plomb(x)` returns the PSD estimate of `x` evaluated at a set of evenly spaced normalized frequencies, `w`, spanning the Nyquist interval. Use NaNs to specify missing

samples. All of the above options are available for normalized frequencies. To access them, specify an empty array as the second input.

`plomb( ___ )` with no output arguments plots the Lomb-Scargle periodogram PSD estimate in the current figure window.

## Examples

### Irregularly Sampled Signal and Signal with Missing Samples

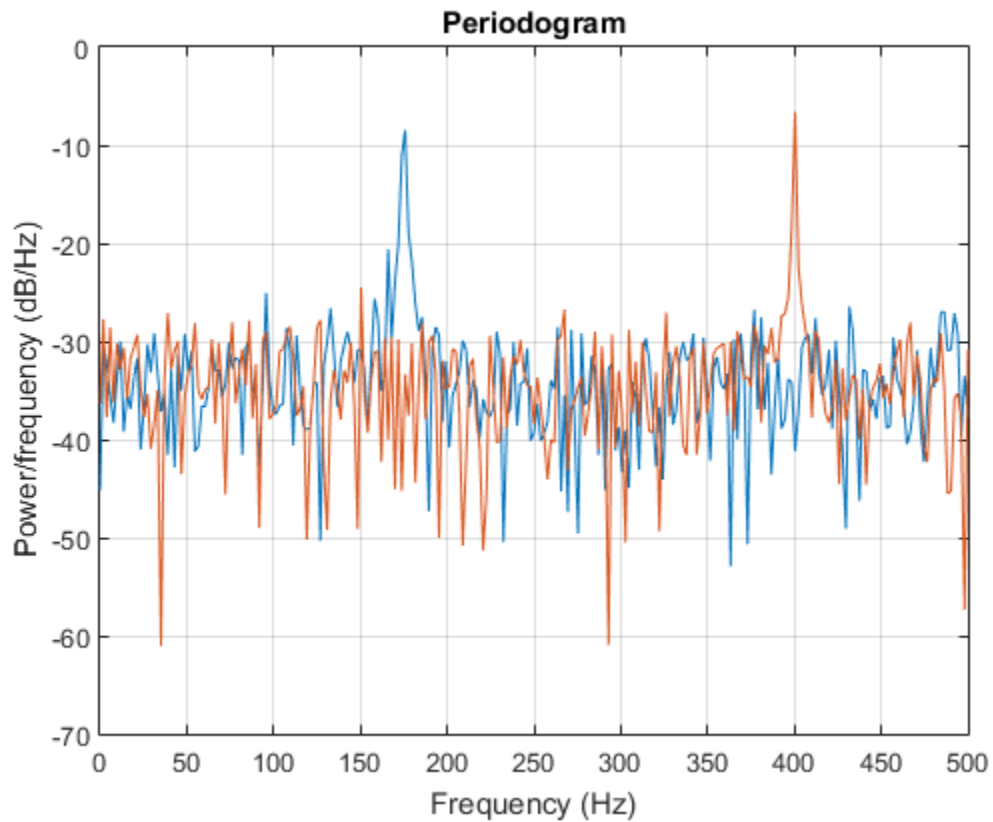
The Lomb-Scargle method can handle signals that have been sampled unevenly or have missing samples.

Generate a two-channel sinusoidal signal sampled at 1 kHz for about 0.5 s. The sinusoid frequencies are 175 Hz and 400 Hz. Embed the signal in white noise with variance  $\sigma^2 = 1/4$ .

```
Fs = 1000;  
f0 = 175;  
f1 = 400;  
  
t = 0:1/Fs:0.5;  
  
wgn = randn(length(t),2)/2;  
  
sigOrig = sin(2*pi*[f0;f1]*t)' + wgn;
```

Compute and plot the periodogram of the signal. Use `periodogram` with the default settings.

```
periodogram(sigOrig,[],[],Fs)  
  
axisLim = axis;  
title('Periodogram')
```

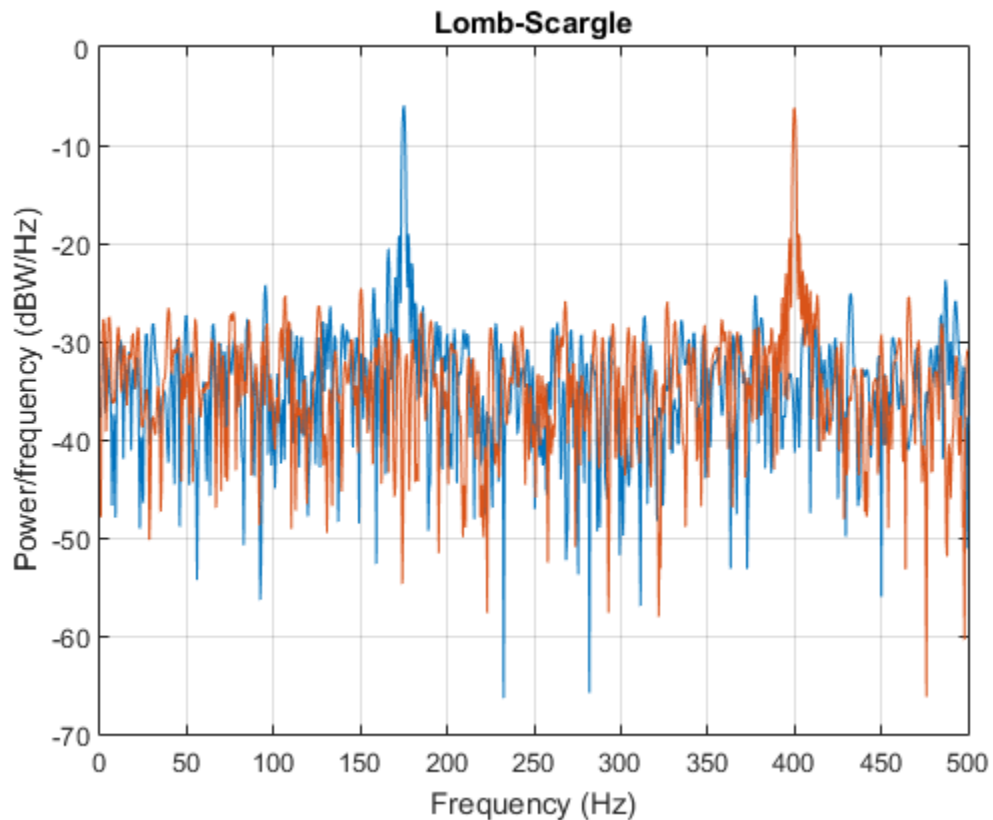


Use `plomb` with the default settings to estimate and plot the PSD of the signal. Use the axis limits from the previous plot.

```
plomb(sigOrig,t)

axis(axisLim)
title('Lomb-Scargle')
```

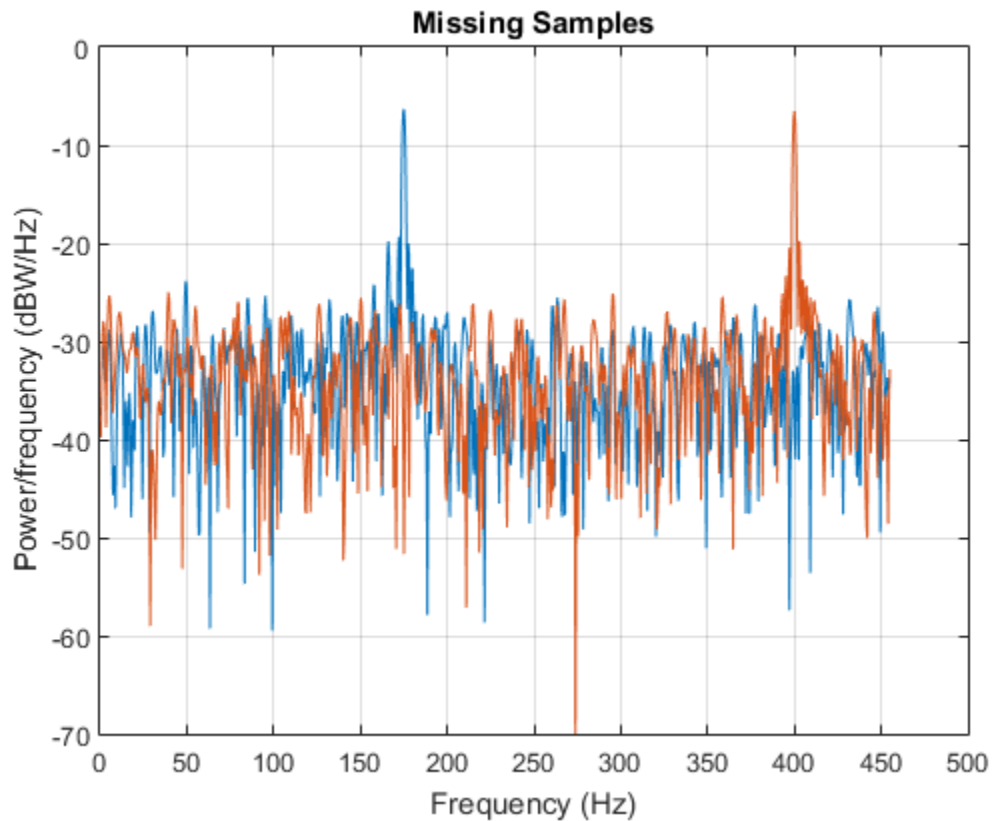




Suppose the signal is missing 10% of the original samples. Place NaN's in random locations to simulate the missing data points. Use `plomb` to estimate and plot the PSD of the signal with missing samples.

```
sinMiss = sigOrig;  
  
misfrac = 0.1;  
nTime = length(t)*2;  
  
sinMiss(randperm(nTime,round(nTime*misfrac))) = NaN;  
  
plomb(sinMiss,t)  
  
axis(axisLim)
```

```
title('Missing Samples')
```



Sample the original signal, but make the sampling nonuniform by adding jitter (uncertainty) to the time measurements. The first instant continues to be at zero. Use `plomb` to estimate and plot the PSD of the nonuniformly sampled signal.

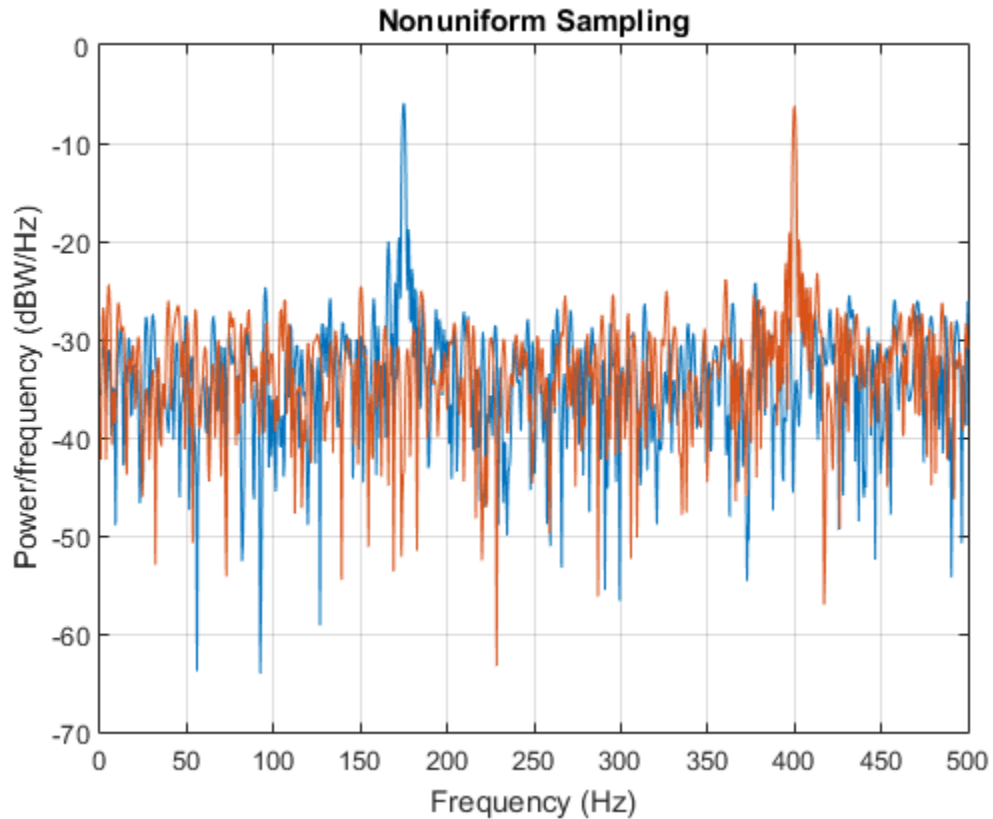
```
tirr = t + (1/2-rand(size(t)))/Fs/2;
tirr(1) = 0;

sinIrreg = sin(2*pi*[f0;f1]*tirr)' + wgn;

plomb(sinIrreg,tirr)

axis(axisLim)
```

```
title('Nonuniform Sampling')
```



### Periodogram of Data Set with Missing Samples

Galileo Galilei observed the motion of Jupiter's four largest satellites during the winter of 1610. When the weather allowed, Galileo recorded the satellites' locations. Use his observations to estimate the orbital period of one of the satellites, Callisto.

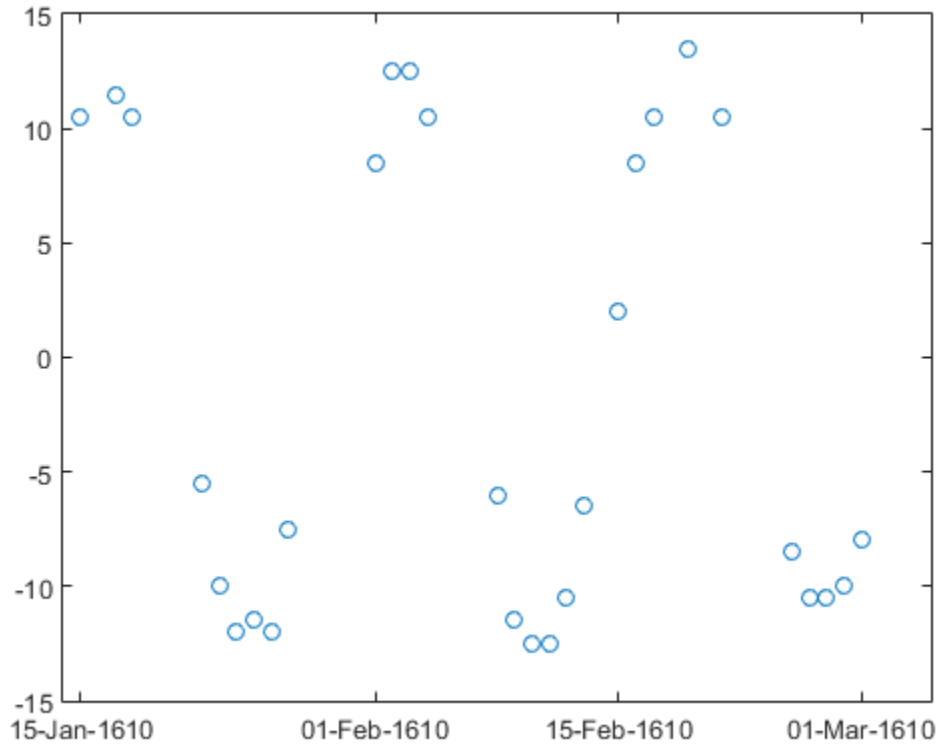
Callisto's angular position is measured in minutes of arc. Missing data due to cloudy conditions are specified using NaN's. The first observation is dated January 15.

```
yg = [10.5 NaN 11.5 10.5 NaN NaN NaN -5.5 -10.0 -12.0 -11.5 -12.0 -7.5 ...
      NaN NaN NaN NaN 8.5 12.5 12.5 10.5 NaN NaN NaN -6.0 -11.5 -12.5 ...
```

```
-12.5 -10.5 -6.5 NaN 2.0 8.5 10.5 NaN 13.5 NaN 10.5 NaN NaN NaN ...
-8.5 -10.5 -10.5 -10.0 -8.0]';
```

```
plot(yg, 'o')
```

```
ax = gca;
nights = [1 18 32 46];
ax.XTick = nights;
ax.XTickLabel = char(datetime(1610,1,nights+14));
```



Estimate the power spectrum of the data using `plomb`. Investigate frequencies up to half the sample rate of one observation per day. Specify an oversampling factor of 10.

```
Fs = 1;
```

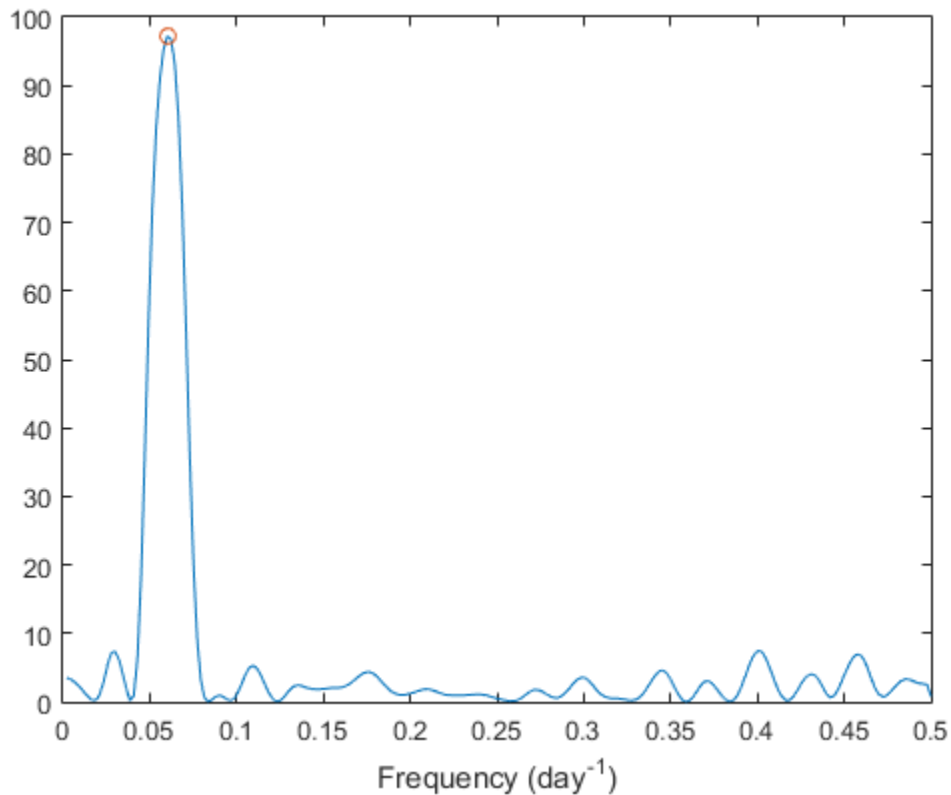
```
Fmax = Fs/2;
```

```
[pxx,f] = plomb(yg,Fs,Fmax,10, 'power');
```

Use `findpeaks` to determine the location of the only prominent peak of the spectrum. Plot the power spectrum and show the peak.

```
[pk,f0] = findpeaks(pxx,f, 'MinPeakHeight',10);
```

```
plot(f,pxx,f0,pk, 'o')  
xlabel('Frequency (day-1)')
```



Determine Callisto's orbital period (in days) as the inverse of the frequency of maximum energy. The result differs by less than 1% from the value published by NASA.

```
Period = 1/f0  
NASA = 16.6890184;  
PercentDiscrep = (Period-NASA)/NASA*100
```

```
Period =  
    16.6454
```

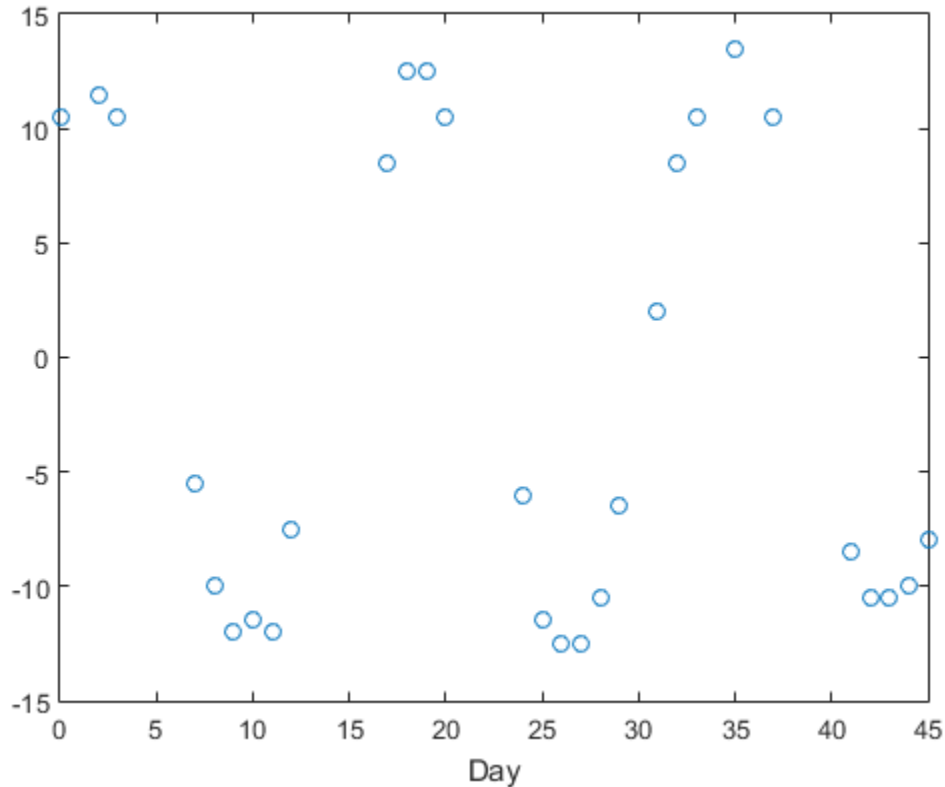
```
PercentDiscrep =  
    -0.2613
```

### **Periodogram of Data Set with Irregular Sampling**

Galileo Galilei discovered Jupiter's four largest satellites in January of 1610 and recorded their locations every clear night until March of that year. Use Galileo's data to find the orbital period of Callisto, the outermost of the four satellites.

Galileo's observations of Callisto's angular position are in minutes of arc. There are several gaps due to cloudy conditions.

```
t = [0 2 3 7 8 9 10 11 12 17 18 19 20 24 25 26 27 28 29 31 32 33 35 37 ...  
    41 42 43 44 45]';  
  
yg = [10.5 11.5 10.5 -5.5 -10.0 -12.0 -11.5 -12.0 -7.5 8.5 12.5 12.5 ...  
    10.5 -6.0 -11.5 -12.5 -12.5 -10.5 -6.5 2.0 8.5 10.5 13.5 10.5 -8.5 ...  
    -10.5 -10.5 -10.0 -8.0]';  
  
plot(t,yg,'o')  
xlabel('Day')
```



Use `plomb` to compute the periodogram of the data. Estimate the power spectrum up to a frequency of  $0.5 \text{ day}^{-1}$ . Specify an oversampling factor of 10. Choose the standard Lomb-Scargle normalization.

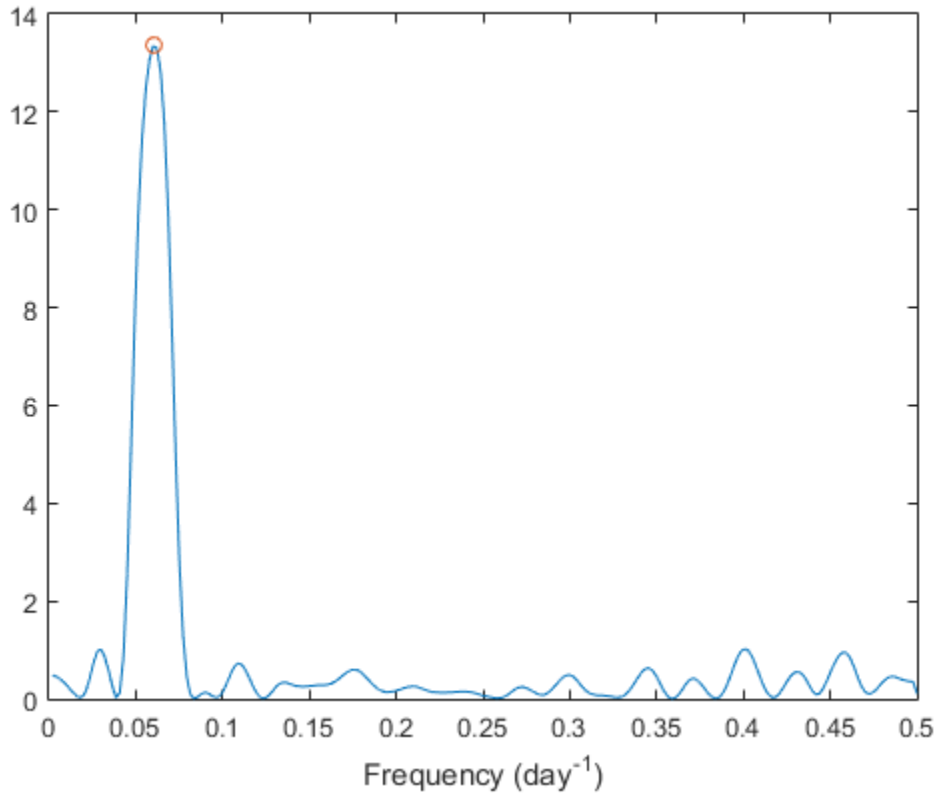
```
[pxx, f] = plomb(yg, t, 0.5, 10, 'normalized');
```

The periodogram has one clear maximum. Name the peak frequency  $f_0$ . Plot the periodogram and annotate the peak.

```
[pmax, lmax] = max(pxx);
f0 = f(lmax);
```

```
plot(f, pxx, f0, pmax, 'o')
```

```
xlabel('Frequency (day-1)')
```



Use linear least squares to fit to the data a function of the form

$$y(t) = A + B \cos 2\pi f_0 t + C \sin 2\pi f_0 t.$$

The fitting parameters are the amplitudes  $A$ ,  $B$ , and  $C$ .

```
ft = 2*pi*f0*t;
```

```
ABC = [ones(size(ft)) cos(ft) sin(ft)] \ yg
```



ABC =

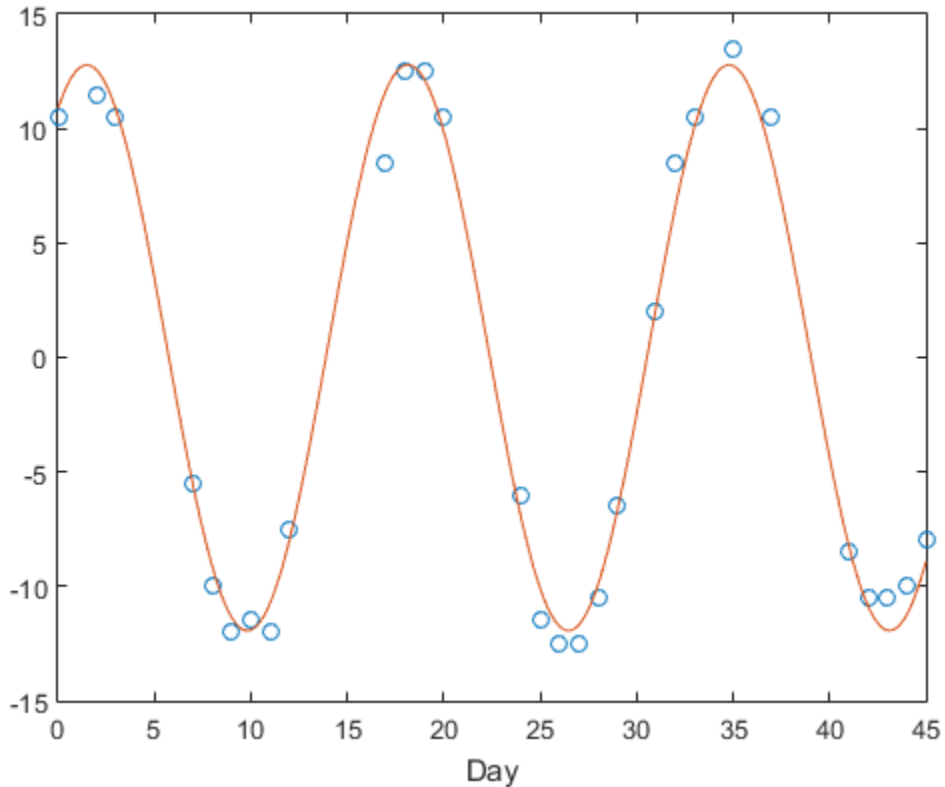
```
0.4243
10.4444
6.6137
```

Use the fitting parameters to construct the fitting function on a 200-point interval. Plot the data and overlay the fit.

```
x = linspace(t(1),t(end),200)';
fx = 2*pi*f0*x;

y = [ones(size(fx)) cos(fx) sin(fx)] * ABC;

plot(t,yg,'o',x,y)
xlabel('Day')
```



### Irregular Sampling and Aliasing

Sample a 0.8 Hz sinusoid at 1 Hz for 100 s. Embed the sinusoid in white noise with a variance of 1/100. Reset the random number generator for repeatable results.

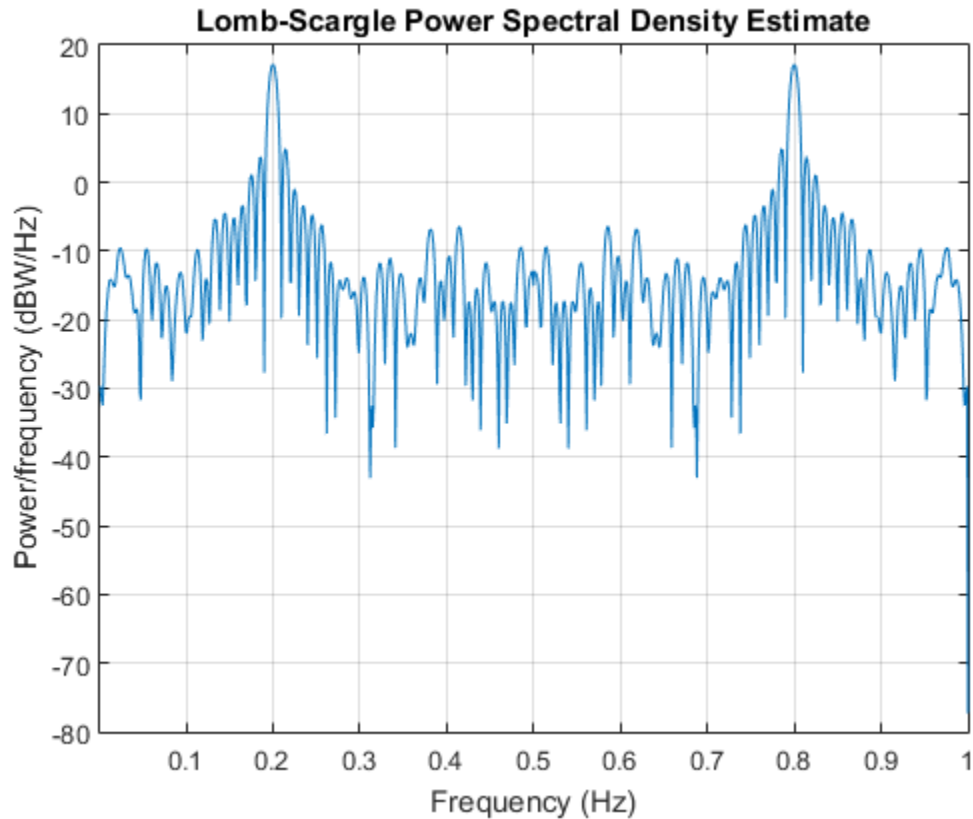
```
f0 = 0.8;
```

```
rng default  
wgn = randn(1,100)/10;
```

```
ts = 1:100;  
s = sin(2*pi*f0*ts) + wgn;
```

Compute and plot the power spectral density estimate up to the sample rate. Specify an oversampling factor of 10.

```
plomb(s,ts,1,10)
```

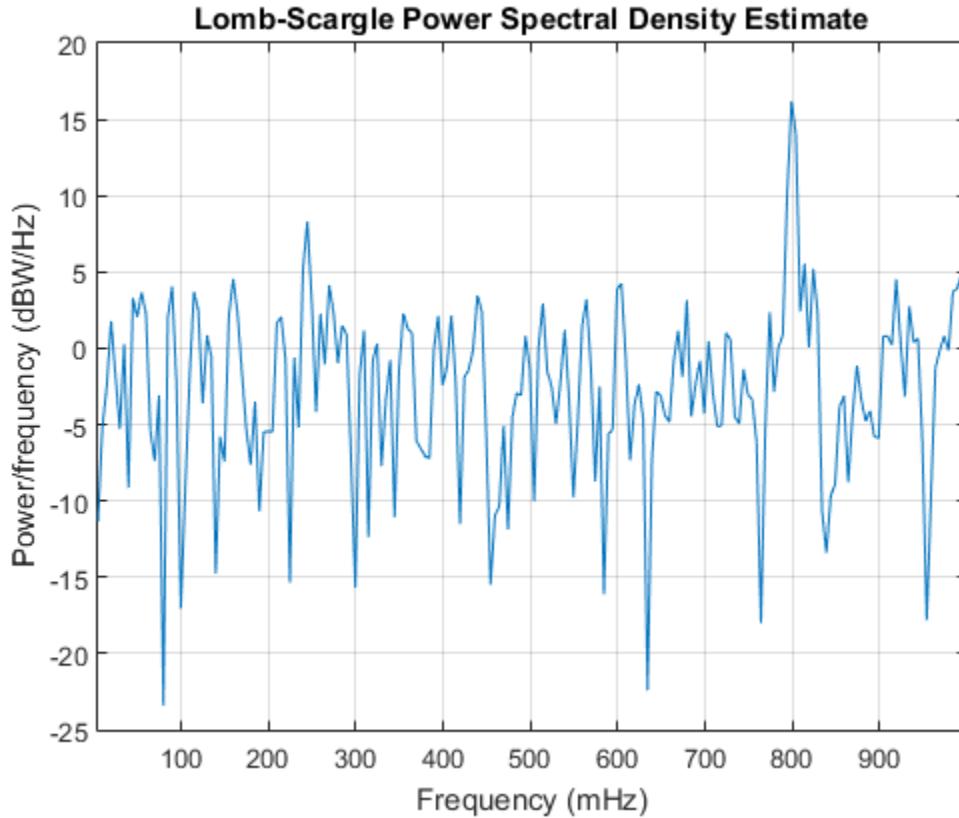


The aliasing results from the fact that the frequency of the sinusoid is greater than the Nyquist frequency.

Repeat the calculation, but now sample the sinusoid at random times. Include frequencies up to 1 Hz. Specify an oversampling factor of 2. Plot the PSD.

```
tn = sort(100*rand(1,100));  
n = sin(2*pi*f0*tn) + wgn;
```

```
ofac = 2;  
plomb(n,tn,1,ofac)
```

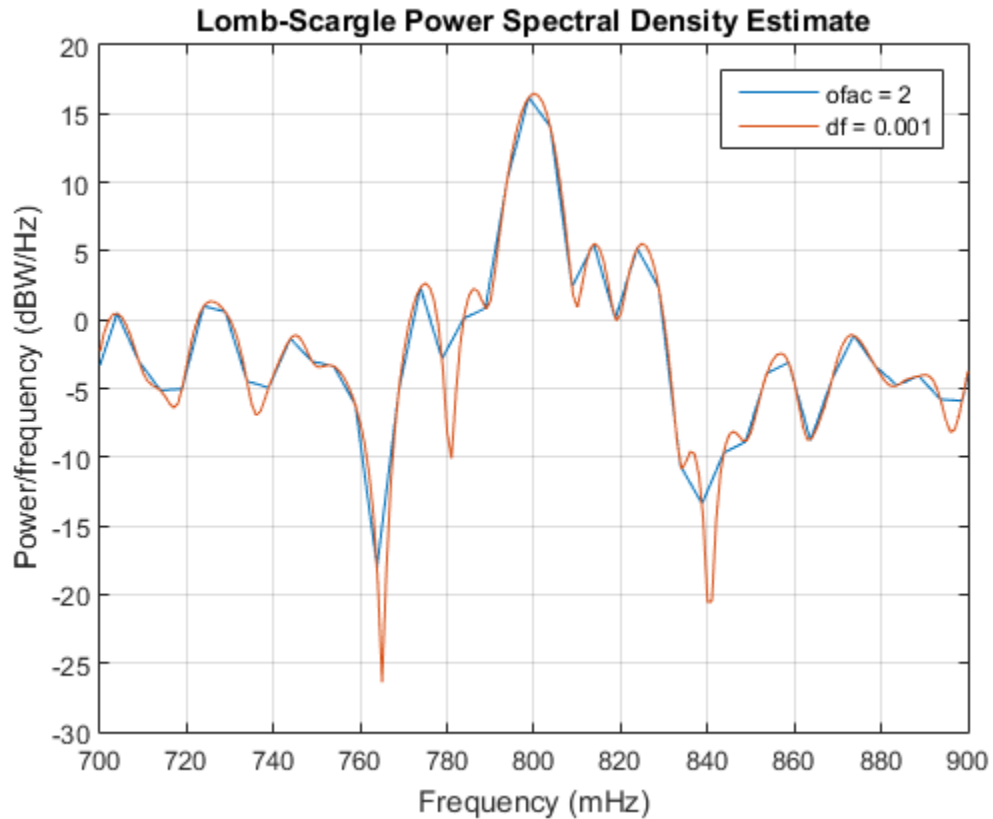


The aliasing disappears. The irregular sampling increases the effective sample rate by shrinking some time intervals.

Zoom in on the frequencies around 0.8 Hz. Use a fine grid with a spacing of 0.001 Hz. You cannot specify an oversampling factor or a maximum frequency in this case.

```
df = 0.001;  
fvec = 0.7:df:0.9;
```

```
hold on
plomb(n,tn,fvec)
legend('ofac = 2','df = 0.001')
```



### Exponential Distribution

Generate  $N = 1024$  samples of white noise with variance  $\sigma = 1$ , given a sample rate of 1 Hz. Compute the power spectrum of the white noise. Choose the Lomb-Scargle normalization and specify an oversampling factor **ofac = 15**. Reset the random number generator for repeatable results.

```
rng default
```

```
N = 1024;
t = (1:N)';
wgn = randn(N,1);

ofac = 15;
[pwgn,f] = plomb(wgn,t,[],ofac,'normalized');
```

Verify that the Lomb-Scargle power spectrum estimate of white noise has an exponential distribution with unit mean. Plot a histogram of the values of `pwgn` and a histogram of a set of exponentially distributed random numbers generated using the distribution function  $f(z|1) = e^{-z}$ . To normalize the histograms, recall that the total number of periodogram samples is  $N \times \text{ofac}/2$ . Specify a bin width of 0.25. Overlay a plot of the theoretical distribution function.

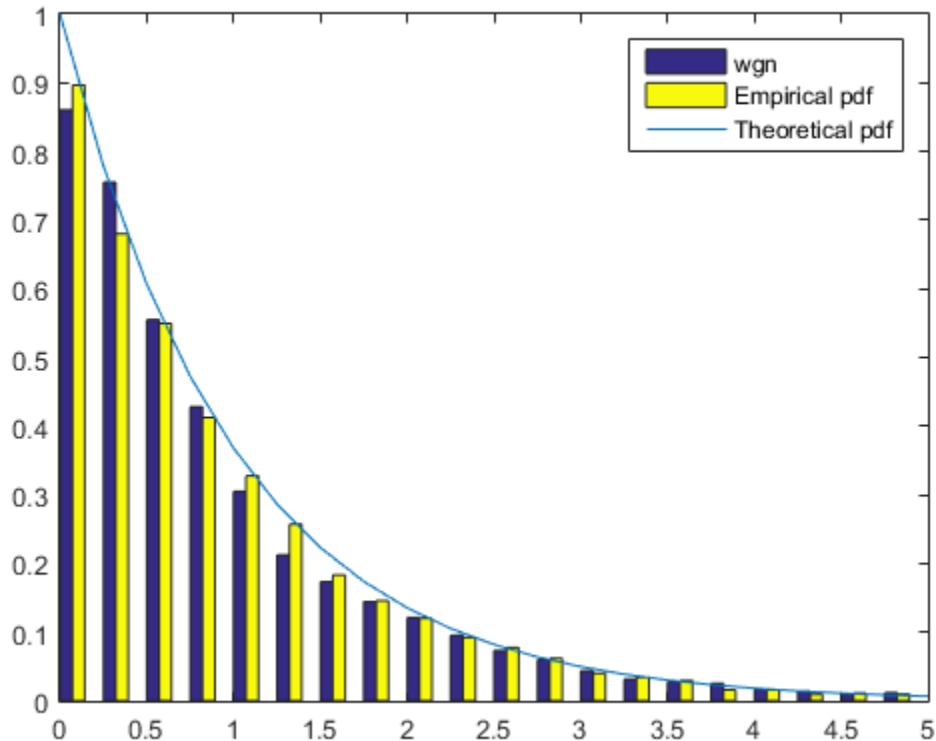
```
dx = 0.25;
br = 0:dx:5;

Nf = N*ofac/2;

hpwgn = histcounts(pwgn,br)';
hRand = histcounts(-log(rand(Nf,1)),br)';

bend = br(1:end-1);

bar(bend,[hpwgn hRand]/Nf/dx,'histc')
hold on
plot(br,exp(-br))
legend('wgn','Empirical pdf','Theoretical pdf')
hold off
```



Embed in the noise a sinusoidal signal of frequency 0.1 Hz. Use a signal-to-noise ratio of  $\xi = 0.01$ . Specify the sinusoid amplitude,  $x_0$ , using the relation  $x_0 = \sigma\sqrt{2\xi}$ . Compute the power spectrum of the signal and plot its histogram alongside the empirical and theoretical distribution functions.

```
SNR = 0.01;
x0 = sqrt(2*SNR);
sigsmall = wgn + x0*sin(2*pi/10*t);

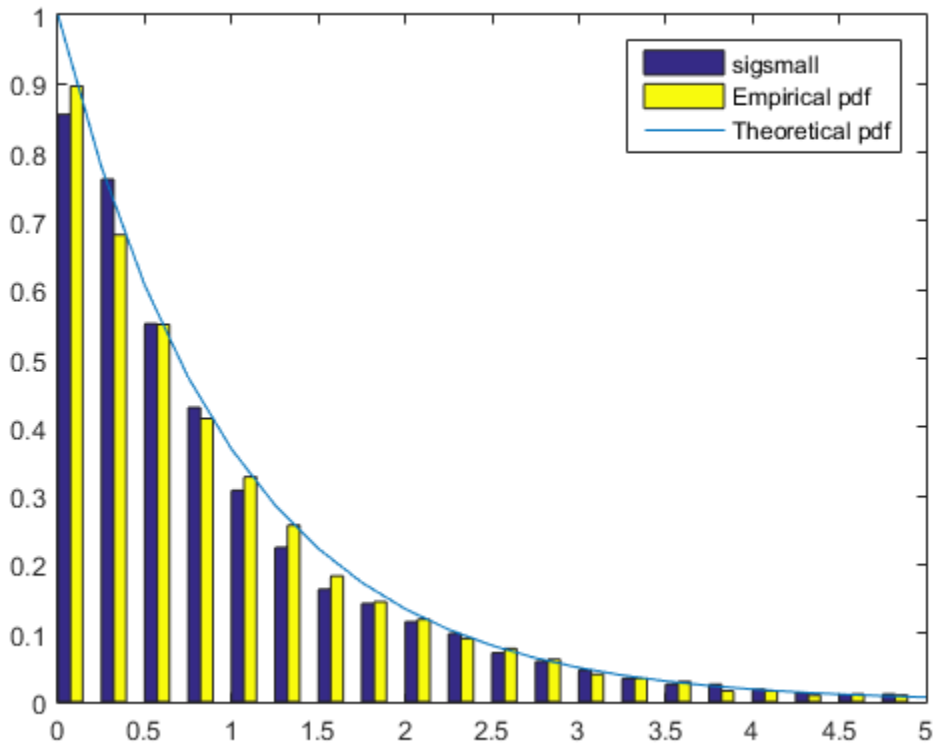
[psigsmall,f] = plomb(sigsmall,t,[],ofac,'normalized');

hpsigsmall = histcounts(psigsmall,br)';
```

```

bar(bend,[hpsigsmall hRand]/Nf/dx,'histc')
hold on
plot(br,exp(-br))
legend('sigsmall','Empirical pdf','Theoretical pdf')
hold off

```



Repeat the calculation using  $\xi = 1$ . The distribution now differs noticeably.

```

SNR = 1;
x0 = sqrt(2*SNR);
siglarge = wgn + x0*sin(2*pi/10*t);

[psiglarge,f] = plomb(siglarge,t,[],ofac,'normalized');

```

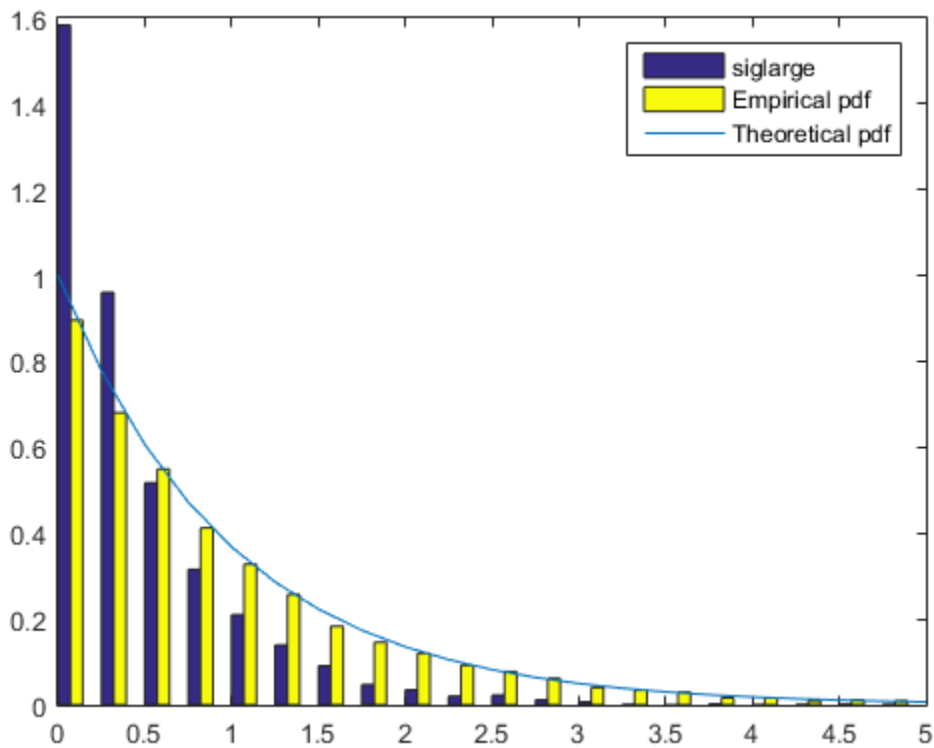


```

hpsiglarge = histcounts(psiglarge,br)';

bar(bend,[hpsiglarge hRand]/Nf/dx,'histc')
hold on
plot(br,exp(-br))
legend('siglarge','Empirical pdf','Theoretical pdf')
hold off

```



### False-Alarm Probabilities

Generate 100 samples of a sinusoidal signal at a sample rate of 1 Hz. Specify an amplitude of 0.75 and a frequency of  $0.6/2\pi \approx 0.096$  Hz. Embed the signal in white noise of variance 0.902. Reset the random number generator for repeatable results.

```
rng default
```

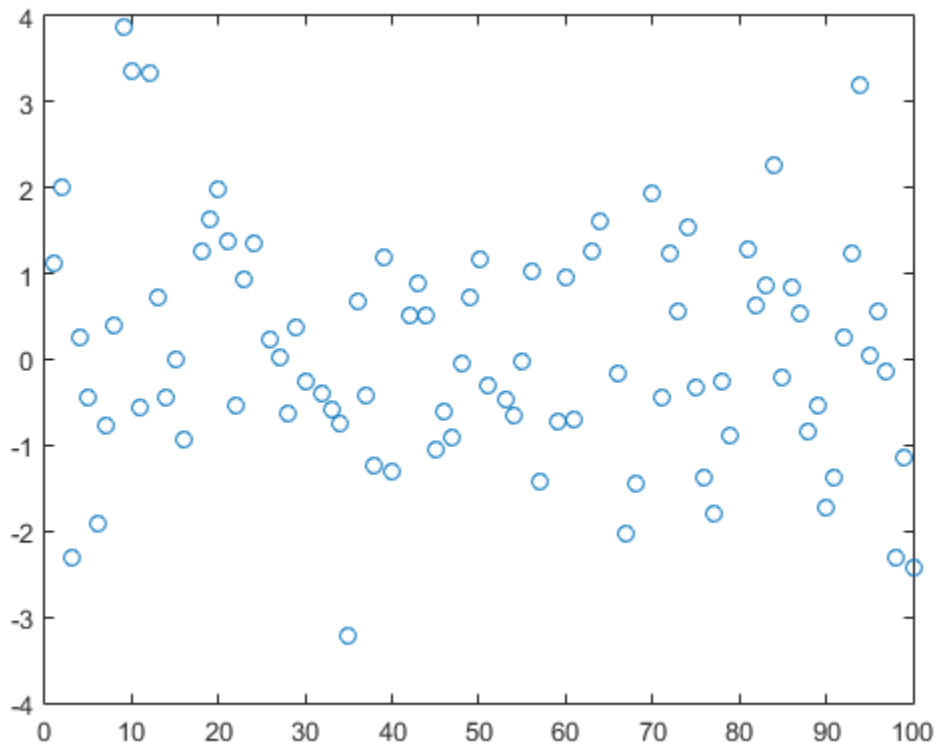
```
X0 = 0.75;  
f0 = 0.6;  
vr = 0.902;
```

```
Nsamp = 100;  
t = 1:Nsamp;  
X = X0*cos(f0*(1:Nsamp))+randn(1,Nsamp)*sqrt(vr);
```

Discard 10% of the samples at random. Plot the signal.

```
X(randperm(Nsamp,Nsamp/10)) = NaN;
```

```
plot(t,X,'o')
```

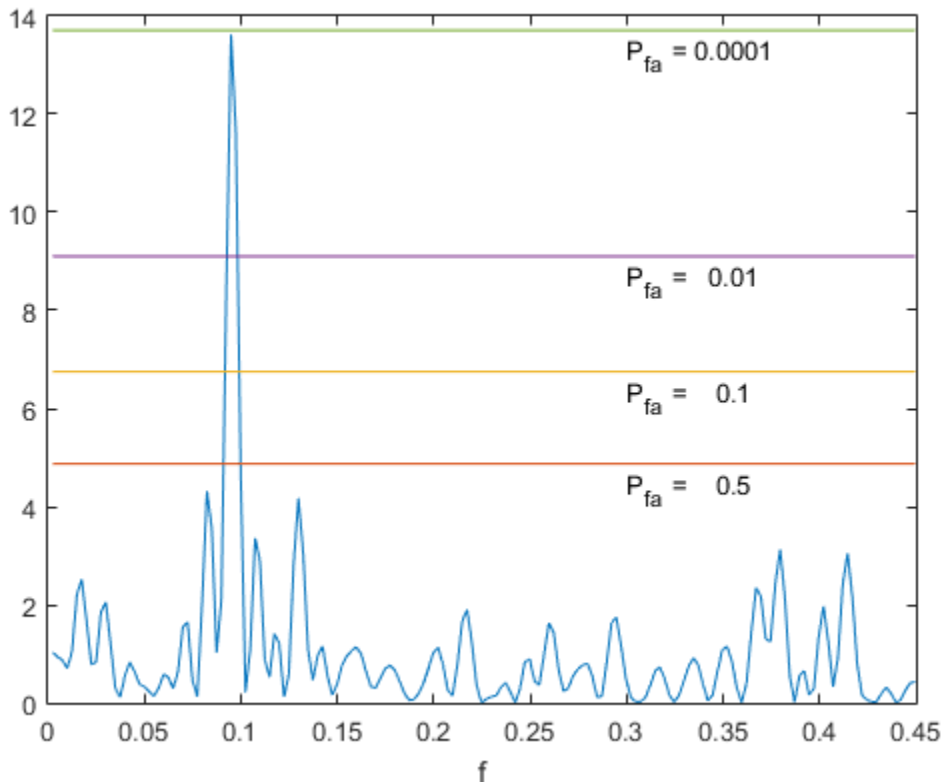


Compute and plot the normalized power spectrum. Annotate the levels that correspond to false-alarm probabilities of 50%, 10%, 1%, and 0.01%. If you generate many 90-sample white noise signals with variance 0.902, then half of them have one or more peaks higher than the 50% line, 10% have one or more peaks higher than the 10% line, and so on.

```
Pfa = [50 10 1 0.01]/100;
Pd = 1-Pfa;

[pxx,f,pth] = plomb(X,1:Nsamp, 'normalized', 'Pd',Pd);

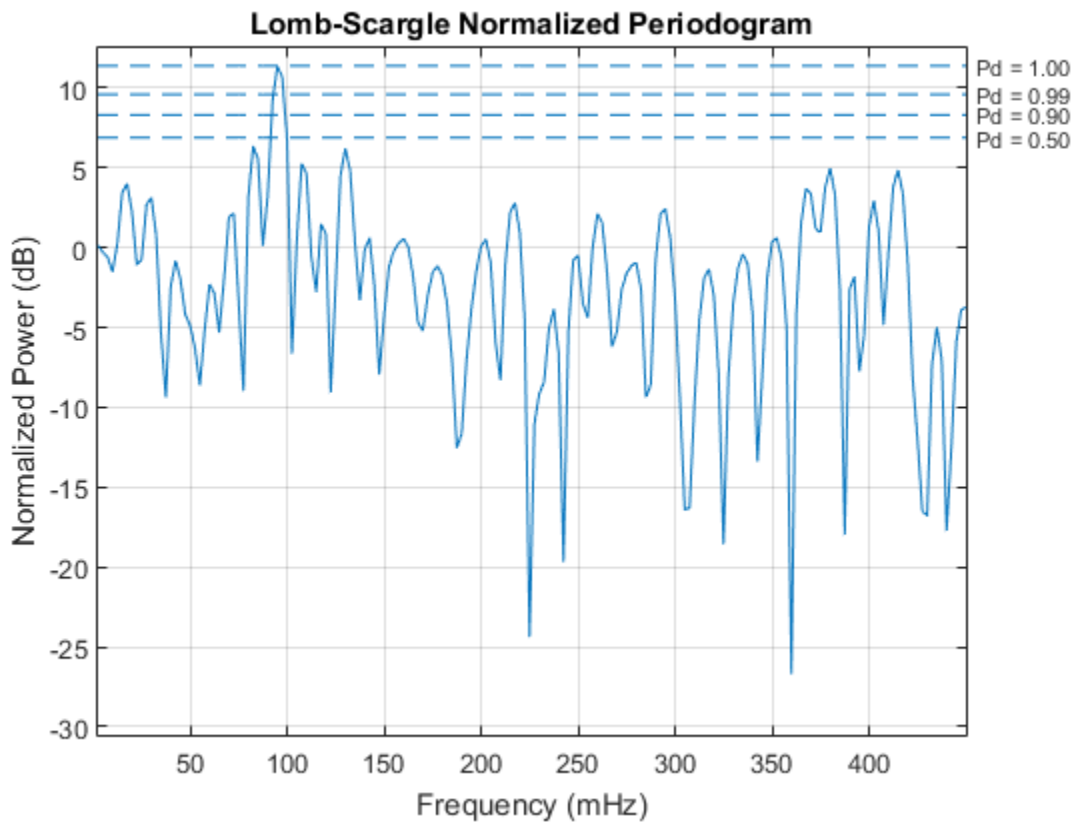
plot(f,pxx,f,pth*ones(size(f')))
xlabel('f')
text(0.3*[1 1 1 1],pth-.5,[repmat('P_{fa} = ',[4 1]) num2str(Pfa')])
```



In this case, the peak is high enough that only about 0.01% of the possible signals can attain it.

Use `plomb` with no output arguments to repeat the calculation. The plot is now logarithmic, and the levels are drawn in terms of detection probabilities.

```
plomb(X,1:Nsamp, 'normalized', 'Pd', Pd)
```



### Lomb-Scargle Periodogram of Noisy Sinusoids

When given a data vector as the only input, `plomb` estimates the power spectral density using normalized frequencies.

Generate 128 samples of a sinusoid of normalized frequency  $\pi/2$  rad/sample embedded in white Gaussian noise of variance 1/100.

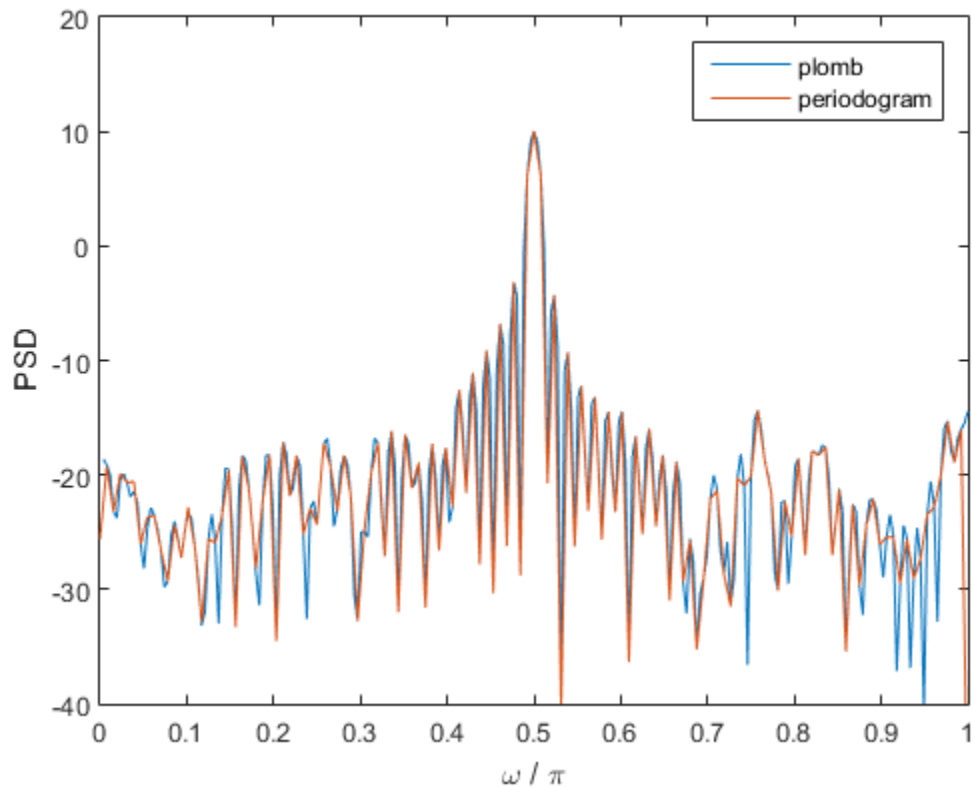
```
t = (0:127)';  
x = sin(2*pi*t/4);  
x = x + randn(size(x))/10;
```

Estimate the PSD using the Lomb-Scargle procedure. Repeat the calculation with periodogram.

```
[p,f] = plomb(x);  
[pper,fper] = periodogram(x);
```

Plot the PSD estimates in decibels. Verify that the results are equivalent.

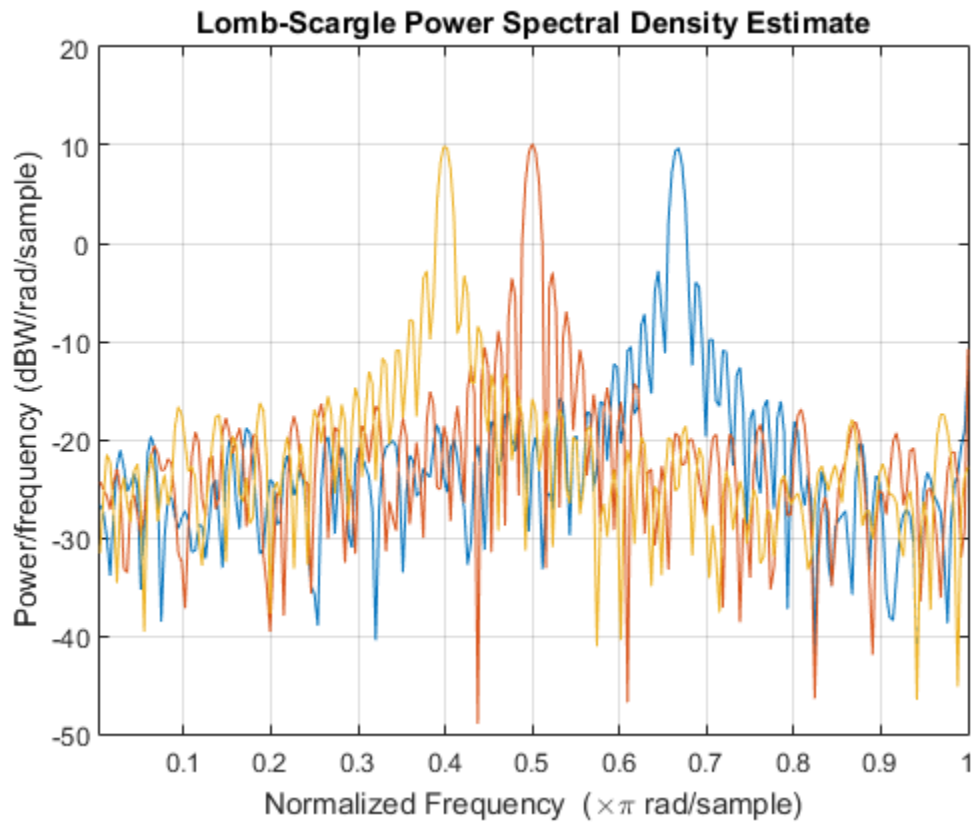
```
plot(f/pi,pow2db(p))  
hold on  
plot(fper/pi,pow2db(pper))  
  
axis([0 1 -40 20])  
xlabel('\omega / \pi')  
ylabel('PSD')  
legend('plomb','periodogram')
```



Estimate the Lomb-Scargle PSD of a three-channel signal composed of sinusoids. Specify the frequencies as  $2\pi/3$  rad/sample,  $\pi/2$  rad/sample, and  $2\pi/5$  rad/sample. Add white Gaussian noise of variance  $1/100$ . Use `plomb` with no output arguments to compute and plot the PSD estimate in decibels.

```
x3 = [sin(2*pi*t/3) sin(2*pi*t/4) sin(2*pi*t/5)];  
x3 = x3 + randn(size(x3))/10;
```

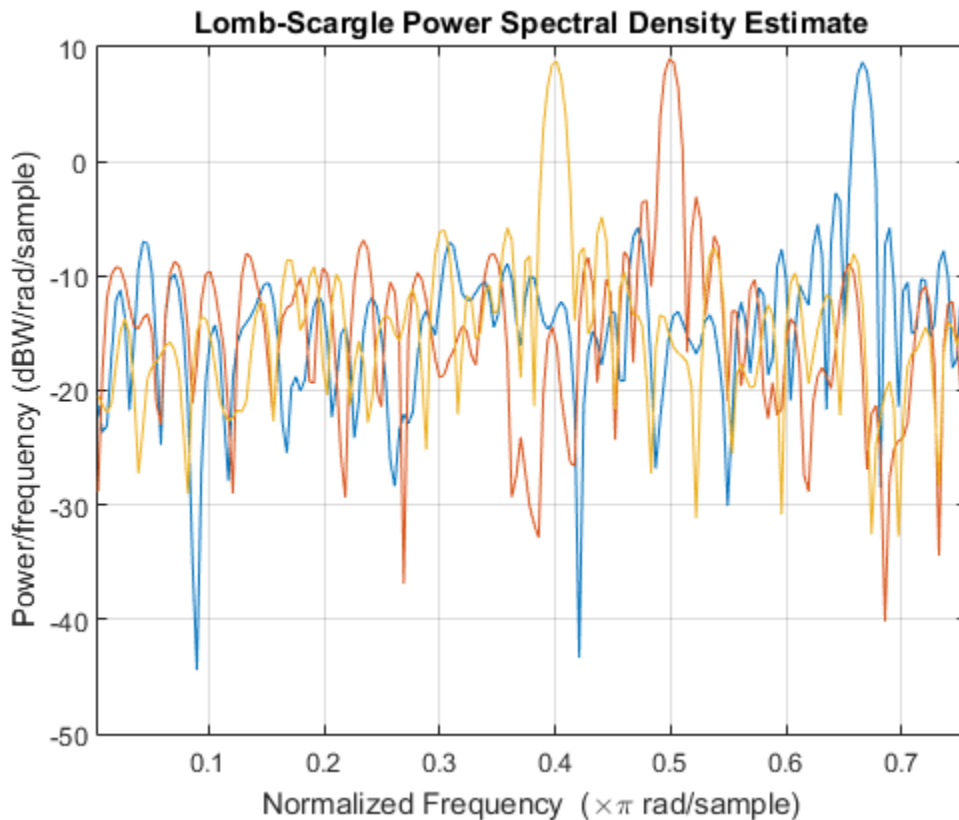
```
figure  
plomb(x3)
```



Compute the PSD estimate again, but now remove 25% of the data at random.

```
x3(randperm(numel(x3),0.25*numel(x3))) = NaN;
```

```
plomb(x3)
```



### Power Spectral Density of Signal with Missing Samples

If you do not have a time vector, use NaN's to specify missing samples in a signal.

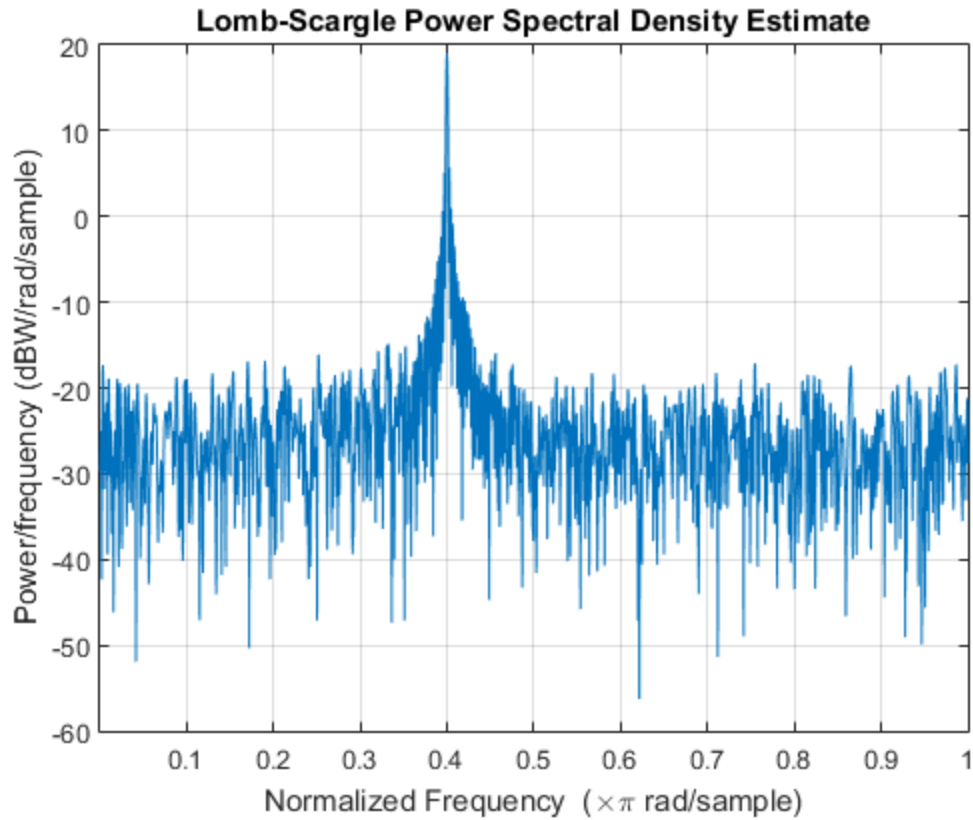
Generate 1024 samples of a sinusoid of normalized frequency  $2\pi/5$  rad/sample embedded in white noise of variance 1/100. Estimate the power spectral density using the Lomb-Scargle procedure. Use `plomb` with no output arguments to plot the estimate.

```
t = (0:1023)';
x = sin(2*pi*t/5);
x = x + randn(size(x))/10;

[pxx,f] = plomb(x);
```



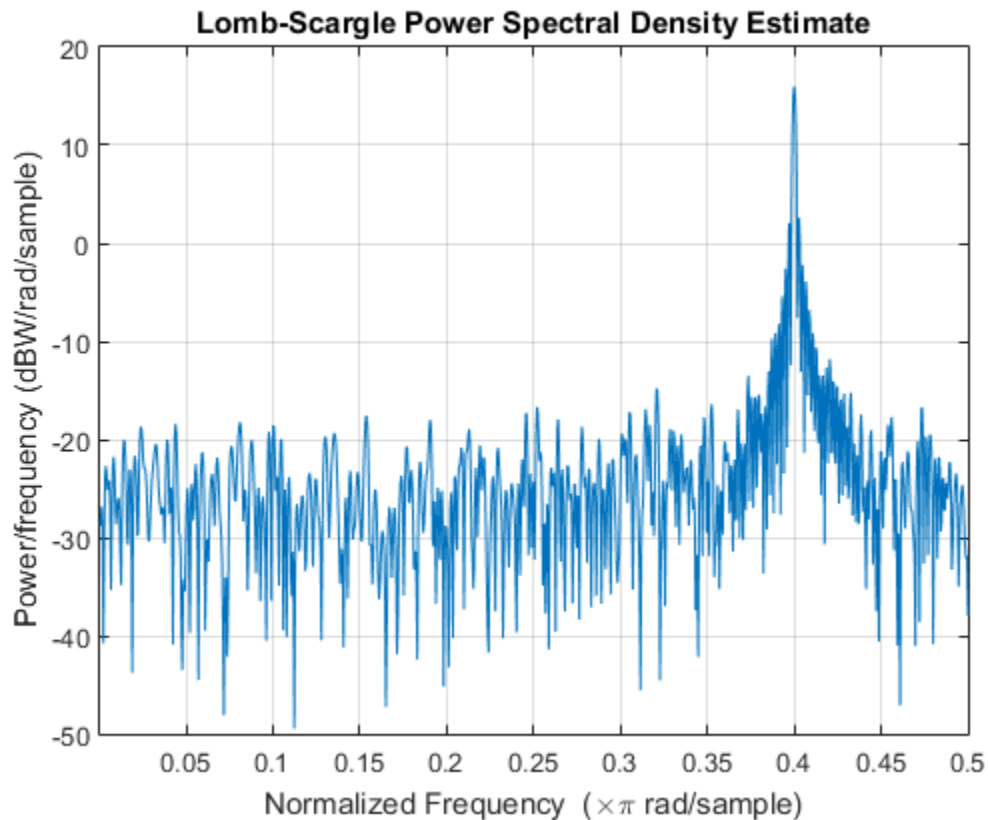
```
plomb(x)
```



Remove every other sample by assigning NaN's. Use `plomb` to compute and plot the PSD. The periodogram peaks at the same frequency because the time axis is unchanged.

```
xnew = x;  
xnew(2:2:end) = NaN;
```

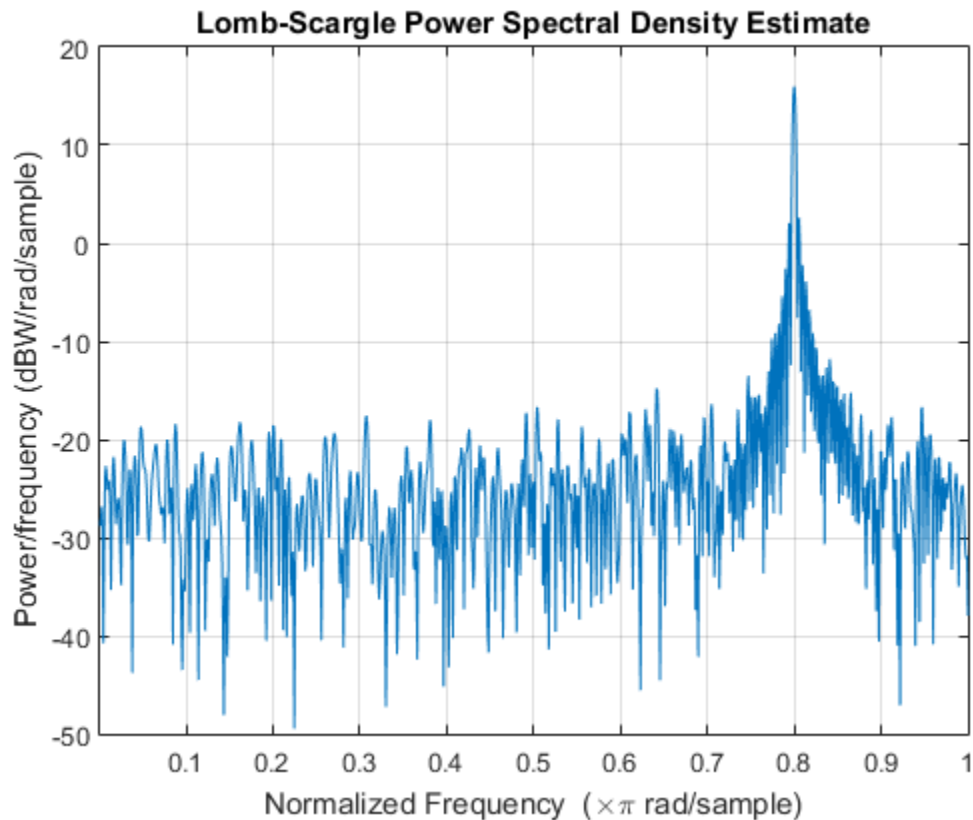
```
plomb(xnew)
```



Remove every other sample by downsampling. The function now estimates the periodicity at twice the original frequency. This is probably not the result you want.

```
xdec = x(1:2:end);
```

```
plomb(xdec)
```



- “Detect Periodicity in a Signal with Missing Samples”
- “Spectral Analysis of Nonuniformly Sampled Signals”

## Input Arguments

**x** — Input signal

vector | matrix

Input signal, specified as a vector or matrix. If **x** is a vector, it is treated as a single channel. If **x** is a matrix, then `plomb` computes the PSD estimate independently for each column and returns it in the corresponding column of `pxx`. **x** can contain NaNs. NaNs are treated as missing data and are excluded from the spectrum computation.

Data Types: `single` | `double`

**t — Time instants**

nonnegative real vector

Time instants, specified as a nonnegative real vector. `t` must increase monotonically but need not be uniformly spaced. `t` can contain NaNs. NaNs are treated as missing data and are excluded from the spectrum computation.

Data Types: `single` | `double`

**fs — Sample rate**

positive scalar

Sample rate, specified as a positive scalar. The sample rate is the number of samples per unit time. If the unit of time is seconds, the sample rate has units of hertz.

Data Types: `single` | `double`

**fmax — Maximum frequency**

positive scalar

Maximum frequency, specified as a positive scalar. `fmax` can be higher than the Nyquist frequency.

Data Types: `single` | `double`

**ofac — Oversampling factor**

4 (default) | positive integer scalar

Oversampling factor, specified as a positive integer scalar.

Data Types: `single` | `double`

**fvec — Input frequencies**

vector

Input frequencies, specified as a vector. `fvec` must have at least two elements.

Data Types: `single` | `double`

**spectrumtype — Power spectrum scaling**

'psd' (default) | 'power' | 'normalized'

Power spectrum scaling, specified as one of 'psd', 'power', or 'normalized'. Omitting `spectrumtype`, or specifying 'psd', returns the power spectral density

estimate. Specifying 'power' scales each estimate of the PSD by the equivalent noise bandwidth of the window. Specify 'power' to obtain an estimate of the power at each frequency. Specifying 'normalized' scales pxx by two times the variance of x.

Data Types: char

### **pddvec — Probabilities of detection**

scalar | vector

Probabilities of detection, specified as the comma-separated pair consisting of 'Pd' and a scalar or a vector of real values between 0 and 1, exclusive. The probability of detection is the probability that a peak in the spectrum is not due to random fluctuations.

Data Types: single | double

## Output Arguments

### **pxx — Lomb-Scargle periodogram**

vector | matrix

Lomb-Scargle periodogram, returned as a vector or matrix. When the input signal, x, is a vector, then pxx is a vector. When x is a matrix, the function treats each column of x as an independent channel and computes the periodogram of each channel.

### **f — Frequencies**

vector

Frequencies, returned as a vector.

Data Types: single | double

### **w — Normalized frequencies**

vector

Normalized frequencies, returned as a vector.

Data Types: single | double

### **pth — Power-level thresholds**

vector | matrix

Power-level thresholds, returned as a vector or matrix. The power-level threshold is the amplitude that a peak in the spectrum must exceed so it can be ruled out (with

probability pdvec) that the peak is due to random fluctuations. Each row of pth corresponds to an element of pdvec. pth has the same number of channels as x.

Data Types: single | double

## More About

### Lomb-Scargle Periodogram

The Lomb-Scargle periodogram lets you find and test weak periodic signals in otherwise random, unevenly sampled data.

Consider  $N$  observations,  $x_k$ , taken at times  $t_k$ , where  $k = 1, \dots, N$ . The Lomb-Scargle periodogram is defined by [1]

$$P_{\text{LS}}(f) = \frac{1}{2\sigma^2} \left\{ \frac{\left[ \sum_{k=1}^N (x_k - \bar{x}) \cos(2\pi f(t_k - \tau)) \right]^2}{\sum_{k=1}^N \cos^2(2\pi f(t_k - \tau))} + \frac{\left[ \sum_{k=1}^N (x_k - \bar{x}) \sin(2\pi f(t_k - \tau)) \right]^2}{\sum_{k=1}^N \sin^2(2\pi f(t_k - \tau))} \right\},$$

where

$$\bar{x} = \frac{1}{N} \sum_{k=1}^N x_k$$

and

$$\sigma^2 = \frac{1}{N-1} \sum_{k=1}^N (x_k - \bar{x})^2$$

are respectively the mean and the variance of the data.

The time offset,  $\tau$ , is chosen as

$$\tan(2(2\pi f)\tau) = \frac{\sum_{k=1}^N \sin(2(2\pi f)t_k)}{\sum_{k=1}^N \cos(2(2\pi f)t_k)}$$

to guarantee the time invariance of the computed spectrum. Any shift  $t_k \rightarrow t_k + T$  in the time measurements results in an identical shift in the offset:  $\tau \rightarrow \tau + T$ . Moreover, the choice ensures that “a maximum in the periodogram occurs at the same frequency which minimizes the sum of squares of the residuals of the fit of a sine wave to the data.” [2] The offset depends only on the measurement times and vanishes when the times are equally spaced.

If the input signal consists of white Gaussian noise, then  $P_{LS}(f)$  follows an exponential probability distribution with unit mean [3].

## References

- [1] Lomb, Nicholas R. “Least-Squares Frequency Analysis of Unequally Spaced Data.” *Astrophysics and Space Science*. Vol. 39, 1976, pp.447–462.
- [2] Scargle, Jeffrey D. “Studies in Astronomical Time Series Analysis. II. Statistical Aspects of Spectral Analysis of Unevenly Spaced Data.” *Astrophysical Journal*. Vol. 263, 1982, pp.835–853.
- [3] Press, William H., and George B. Rybicki. “Fast Algorithm for Spectral Analysis of Unevenly Sampled Data.” *Astrophysical Journal*. Vol. 338, 1989, pp.277–280.
- [4] Horne, James H., and Sallie L. Baliunas. “A Prescription for Period Analysis of Unevenly Sampled Time Series.” *Astrophysical Journal*. Vol. 302, 1986, pp.757–763.

## See Also

bandpower | pburg | pcov | peig | periodogram | pmcov | pmtm | pmusic | pwelch | pyulear | spectrogram

## pmcov

Autoregressive power spectral density estimate — modified covariance method

### Syntax

```
pxx = pmcov(x,order)
pxx = pmcov(x,order,nfft)

[pxx,w] = pmcov( ___ )
[pxx,f] = pmcov( ___ ,fs)

[pxx,w] = pmcov(x,order,w)
[pxx,f] = pmcov(x,order,f,fs)

[ ___ ] = pmcov(x,order, ___ ,freqrange)

[ ___ ,pxxc] = pmcov( ___ ,'ConfidenceLevel',probability)

pmcov( ___ )
```

### Description

`pxx = pmcov(x,order)` returns the power spectral density estimate, `pxx`, of a discrete-time signal, `x`, found using the modified covariance method. When `x` is a vector, it is treated as a single channel. When `x` is a matrix, the PSD is computed independently for each column and stored in the corresponding column of `pxx`. `pxx` is the distribution of power per unit frequency. The frequency is expressed in units of rad/sample. `order` is the order of the autoregressive (AR) model used to produce the PSD estimate.

`pxx = pmcov(x,order,nfft)` uses `nfft` points in the discrete Fourier transform (DFT). For real `x`, `pxx` has length  $(nfft/2+1)$  if `nfft` is even, and  $(nfft+1)/2$  if `nfft` is odd. For complex-valued `x`, `pxx` always has length `nfft`. `pmcov` uses a default DFT length of 256.

`[pxx,w] = pmcov( ___ )` returns the vector of normalized angular frequencies, `w`, at which the PSD is estimated. `w` has units of rad/sample. For real-valued signals, `w`



spans the interval  $[0,\pi]$  when `nfft` is even and  $[0,\pi)$  when `nfft` is odd. For complex-valued signals, `w` always spans the interval  $[0,2\pi)$ .

`[pxx, f] = pmcov( ____, fs)` returns a frequency vector, `f`, in cycles per unit time. The sampling frequency, `fs`, is the number of samples per unit time. If the unit of time is seconds, then `f` is in cycles/second (Hz). For real-valued signals, `f` spans the interval  $[0,fs/2]$  when `nfft` is even and  $[0,fs/2)$  when `nfft` is odd. For complex-valued signals, `f` spans the interval  $[0,fs)$ .

`[pxx, w] = pmcov(x, order, w)` returns the two-sided AR PSD estimates at the normalized frequencies specified in the vector, `w`. The vector, `w`, must contain at least two elements.

`[pxx, f] = pmcov(x, order, f, fs)` returns the two-sided AR PSD estimates at the frequencies specified in the vector, `f`. The vector, `f`, must contain at least two elements. The frequencies in `f` are in cycles per unit time. The sampling frequency, `fs`, is the number of samples per unit time. If the unit of time is seconds, then `f` is in cycles/second (Hz).

`[ ____ ] = pmcov(x, order, ____, freqrange)` returns the AR PSD estimate over the frequency range specified by `freqrange`. Valid options for `freqrange` are: `'onesided'`, `'twosided'`, or `'centered'`.

`[ ____, pxxc] = pmcov( ____, 'ConfidenceLevel', probability)` returns the `probability` × 100% confidence intervals for the PSD estimate in `pxxc`.

`pmcov( ____ )` with no output arguments plots the AR PSD estimate in dB per unit frequency in the current figure window.

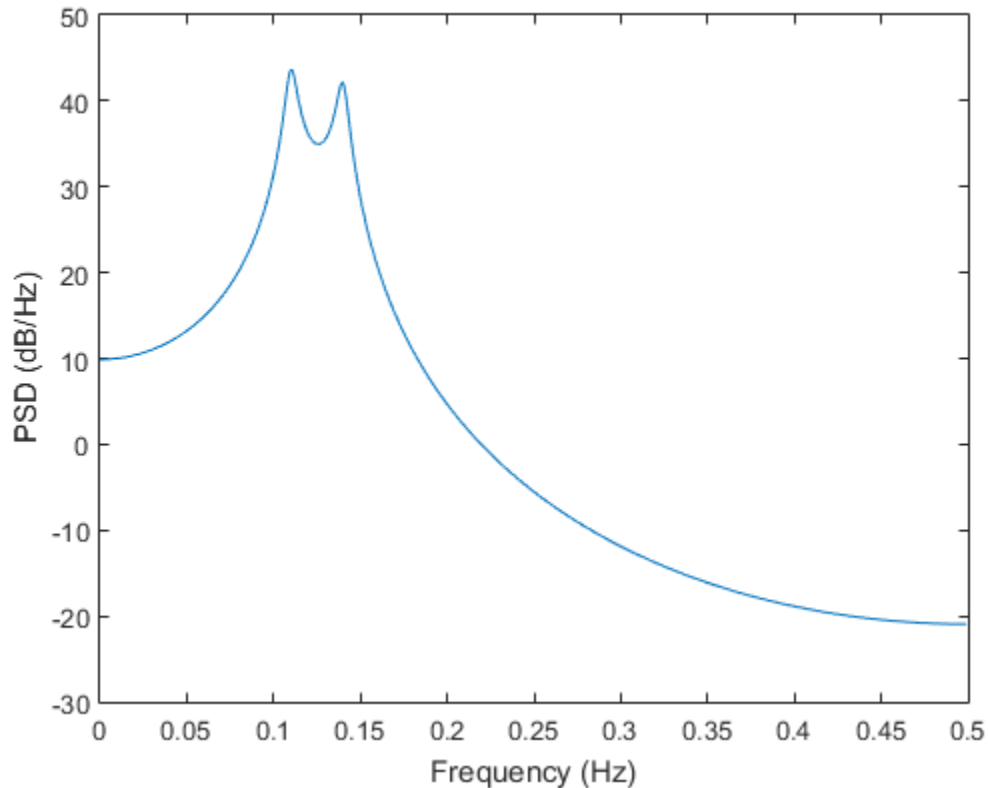
## Examples

### Modified-Covariance PSD Estimate of AR(4) Process

Create a realization of an AR(4) wide-sense stationary random process. Estimate the PSD using the modified covariance method. Compare the PSD estimate based on a single realization to the true PSD of the random process.

Create an AR(4) system function. Obtain the frequency response and plot the PSD of the system.

```
A = [1 -2.7607 3.8106 -2.6535 0.9238];  
[H,F] = freqz(1,A,[],1);  
plot(F,20*log10(abs(H))  
  
xlabel('Frequency (Hz)')  
ylabel('PSD (dB/Hz)')
```



Create a realization of the AR(4) random process. Set the random number generator to the default settings for reproducible results. The realization is 1000 samples in length. Assume a sampling frequency of 1 Hz. Use `pmcov` to estimate the PSD for a 4th-order process. Compare the PSD estimate with the true PSD.

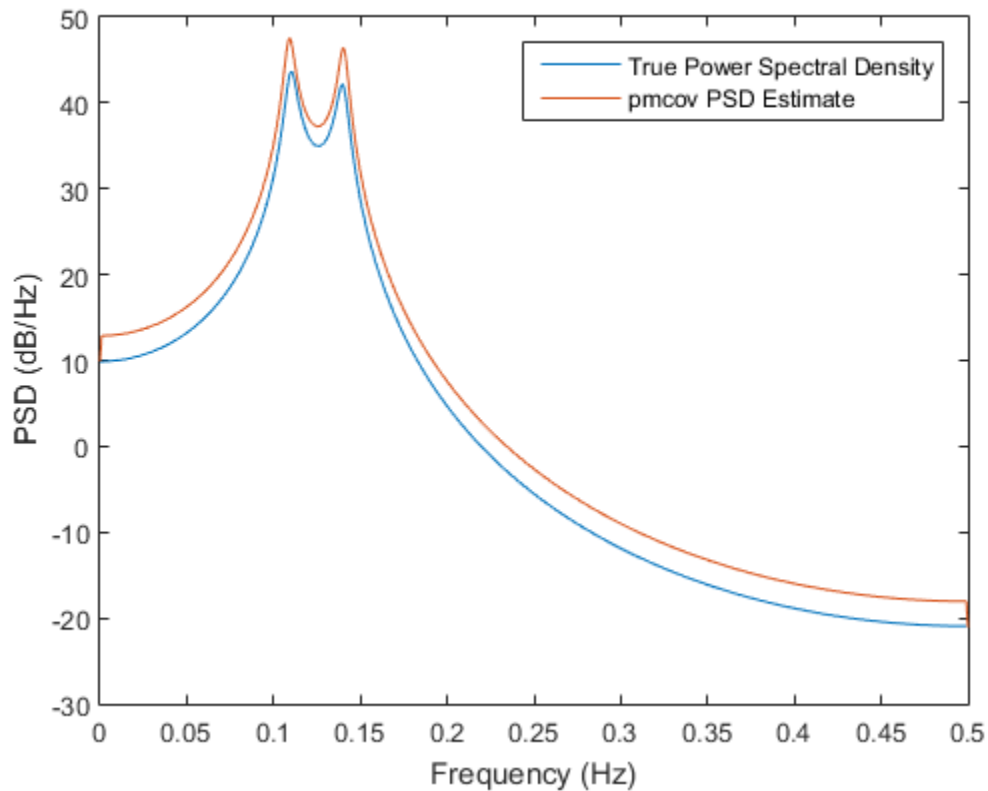
```
rng default
```

```

x = randn(1000,1);
y = filter(1,A,x);
[Pxx,F] = pmcov(y,4,1024,1);

hold on
plot(F,10*log10(Pxx))
legend('True Power Spectral Density','pmcov PSD Estimate')

```



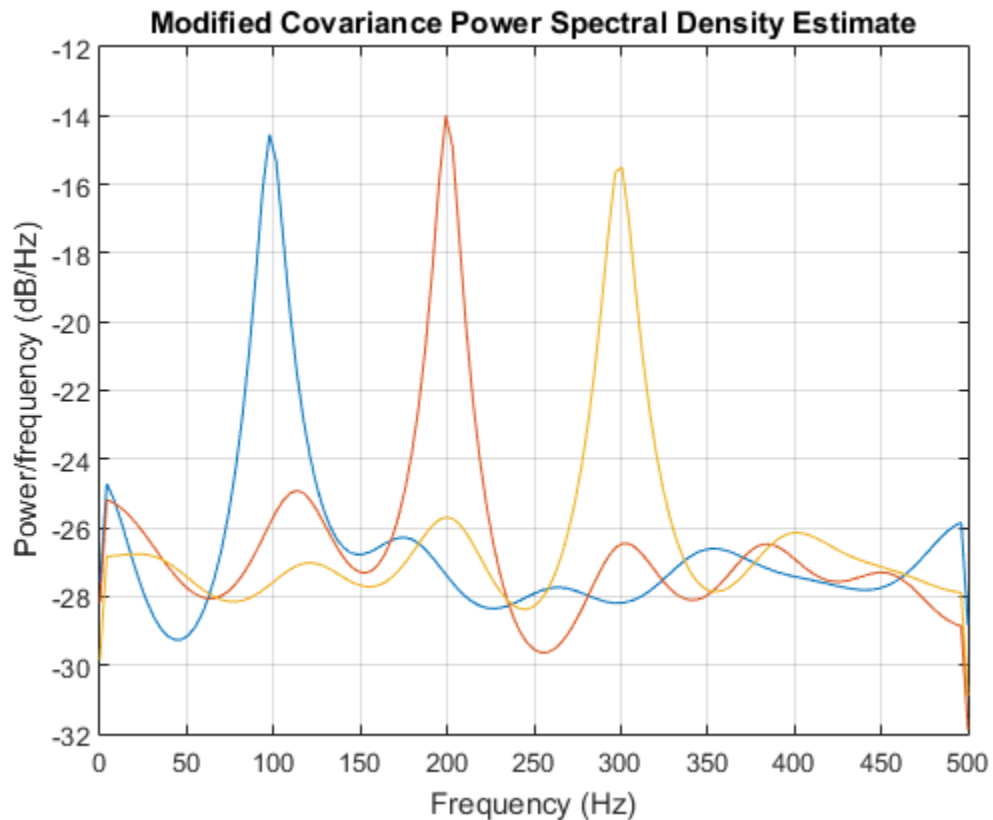
### Modified-Covariance PSD Estimate of a Multichannel Signal

Create a multichannel signal consisting of three sinusoids in additive  $N(0,1)$  white Gaussian noise. The sinusoids' frequencies are 100 Hz, 200 Hz, and 300 Hz. The sampling frequency is 1 kHz, and the signal has a duration of 1 s.

```
Fs = 1000;  
t = 0:1/Fs:1-1/Fs;  
f = [100;200;300];  
x = cos(2*pi*f*t)' + randn(length(t),3);
```

Estimate the PSD of the signal using the modified covariance method with a 12th-order autoregressive model. Use the default DFT length. Plot the estimate.

```
morder = 12;  
pmscov(x,morder,[],Fs)
```



## Input Arguments

### **x** — Input signal

vector | matrix

Input signal, specified as a row or column vector, or as a matrix. If  $x$  is a matrix, then its columns are treated as independent channels.

Example: `cos(pi/4*(0:159))+randn(1,160)` is a single-channel row-vector signal.

Example: `cos(pi./[4;2]*(0:159))'+randn(160,2)` is a two-channel signal.

Data Types: `single` | `double`

Complex Number Support: Yes

**order** — Order of autoregressive model

positive integer

Order of the autoregressive model, specified as a positive integer.

Data Types: double

**nfft** — Number of DFT points

256 (default) | integer | []

Number of DFT points, specified as a positive integer. For a real-valued input signal,  $x$ , the PSD estimate,  $pxx$  has length  $(nfft/2+1)$  if  $nfft$  is even, and  $(nfft+1)/2$  if  $nfft$  is odd. For a complex-valued input signal,  $x$ , the PSD estimate always has length  $nfft$ . If  $nfft$  is specified as empty, the default  $nfft$  is used.

Data Types: single | double

**fs** — Sampling frequency

positive scalar

Sampling frequency, specified as a positive scalar. The sampling frequency is the number of samples per unit time. If the unit of time is seconds, the sampling frequency has units of hertz.

**w** — Normalized frequencies

vector

Normalized frequencies, specified as a row or column vector with at least 2 elements. Normalized frequencies are in rad/sample.

Example:  $w = [\pi/4 \ \pi/2]$

Data Types: double

**f** — Cyclical frequencies

vector

Cyclical frequencies, specified as a row or column vector with at least 2 elements. The frequencies are in cycles per unit time. The unit time is specified by the sampling frequency,  $fs$ . If  $fs$  has units of samples/second, then  $f$  has units of Hz.

Example:  $fs = 1000$ ;  $f = [100 \ 200]$

Data Types: double

### **freqrange** — Frequency range for PSD estimate

'onesided' | 'twosided' | 'centered'

Frequency range for the PSD estimate, specified as a one of 'onesided', 'twosided', or 'centered'. The default is 'onesided' for real-valued signals and 'twosided' for complex-valued signals. The frequency ranges corresponding to each option are

- 'onesided' — returns the one-sided PSD estimate of a real-valued input signal,  $x$ . If  $nfft$  is even,  $pxx$  has length  $nfft/2 + 1$  and is computed over the interval  $[0, \pi]$  rad/sample. If  $nfft$  is odd, the length of  $pxx$  is  $(nfft + 1)/2$  and the interval is  $[0, \pi]$  rad/sample. When  $fs$  is optionally specified, the corresponding intervals are  $[0, fs/2]$  cycles/unit time and  $[0, fs/2)$  cycles/unit time for even and odd length  $nfft$  respectively.
- 'twosided' — returns the two-sided PSD estimate for either the real-valued or complex-valued input,  $x$ . In this case,  $pxx$  has length  $nfft$  and is computed over the interval  $[0, 2\pi)$  rad/sample. When  $fs$  is optionally specified, the interval is  $[0, fs)$  cycles/unit time.
- 'centered' — returns the centered two-sided PSD estimate for either the real-valued or complex-valued input,  $x$ . In this case,  $pxx$  has length  $nfft$  and is computed over the interval  $(-\pi, \pi]$  rad/sample for even length  $nfft$  and  $(-\pi, \pi)$  rad/sample for odd length  $nfft$ . When  $fs$  is optionally specified, the corresponding intervals are  $(-fs/2, fs/2]$  cycles/unit time and  $(-fs/2, fs/2)$  cycles/unit time for even and odd length  $nfft$  respectively.

Data Types: char

### **probability** — Confidence interval for PSD estimate

0.95 (default) | scalar in the range (0,1)

Coverage probability for the true PSD, specified as a scalar in the range (0,1). The output,  $pxxc$ , contains the lower and upper bounds of the probability  $\times 100\%$  interval estimate for the true PSD.

## Output Arguments

### **pxx** — PSD estimate

vector | matrix

PSD estimate, returned as a real-valued, nonnegative column vector or matrix. Each column of `pxx` is the PSD estimate of the corresponding column of `x`. The units of the PSD estimate are in squared magnitude units of the time series data per unit frequency. For example, if the input data is in volts, the PSD estimate is in units of squared volts per unit frequency. For a time series in volts, if you assume a resistance of  $1 \Omega$  and specify the sampling frequency in hertz, the PSD estimate is in watts per hertz.

Data Types: `single` | `double`

### **w** — Normalized frequencies

vector

Normalized frequencies, returned as a real-valued column vector. If `pxx` is a one-sided PSD estimate, `w` spans the interval  $[0, \pi]$  if `nfft` is even and  $[0, \pi)$  if `nfft` is odd. If `pxx` is a two-sided PSD estimate, `w` spans the interval  $[0, 2\pi)$ . For a DC-centered PSD estimate, `f` spans the interval  $(-\pi, \pi]$  rad/sample for even length `nfft` and  $(-\pi, \pi)$  radians/sample for odd length `nfft`.

Data Types: `double`

### **f** — Cyclical frequencies

vector

Cyclical frequencies, returned as a real-valued column vector. For a one-sided PSD estimate, `f` spans the interval  $[0, fs/2]$  when `nfft` is even and  $[0, fs/2)$  when `nfft` is odd. For a two-sided PSD estimate, `f` spans the interval  $[0, fs)$ . For a DC-centered PSD estimate, `f` spans the interval  $(-fs/2, fs/2]$  cycles/unit time for even length `nfft` and  $(-fs/2, fs/2)$  cycles/unit time for odd length `nfft`.

Data Types: `double`

### **pxxc** — Confidence bounds

matrix

Confidence bounds, returned as a matrix with real-valued elements. The row size of the matrix is equal to the length of the PSD estimate, `pxx`. `pxxc` has twice as many columns as `pxx`. Odd-numbered columns contain the lower bounds of the confidence intervals, and even-numbered columns contain the upper bounds. Thus, `pxxc(m, 2*n - 1)` is the lower confidence bound and `pxxc(m, 2*n)` is the upper confidence bound corresponding to the estimate `pxx(m, n)`. The coverage probability of the confidence intervals is determined by the value of the probability input.

Data Types: `single` | `double`



## **See Also**

pburg | pcov | pyulearn

## pmtm

Multitaper power spectral density estimate

### Syntax

```
pxx = pmtm(x)
pxx = pmtm(x, nw)
pxx = pmtm(x, nw, nfft)

[pxx, w] = pmtm( ___ )
[pxx, f] = pmtm( ___ , fs)

[pxx, w] = pmtm(x, nw, w)
[pxx, f] = pmtm(x, nw, f, fs)

[ ___ ] = pmtm( ___ , method)

[ ___ ] = pmtm(x, e, v)
[ ___ ] = pmtm(x, dpss_params)

[ ___ ] = pmtm( ___ , 'DropLastTaper', dropflag)
[ ___ ] = pmtm( ___ , freqrange)
[ ___ , pxxc] = pmtm( ___ , 'ConfidenceLevel', probability)

pmtm( ___ )
```

### Description

`pxx = pmtm(x)` returns Thomson's multitaper power spectral density (PSD) estimate, `pxx`, of the input signal, `x`. When `x` is a vector, it is treated as a single channel. When `x` is a matrix, the PSD is computed independently for each column and stored in the corresponding column of `pxx`. The tapers are the discrete prolate spheroidal (DPSS), or Slepian, sequences. The time-halfbandwidth, `nw`, product is 4. By default, `pmtm` uses the first  $2nw - 1$  DPSS sequences. If `x` is real-valued, `pxx` is a one-sided PSD estimate. If `x` is complex-valued, `pxx` is a two-sided PSD estimate. The number of points, `nfft`, in the discrete Fourier transform (DFT) is the maximum of 256 or the next power of two greater than the signal length.

`pxx = pmtm(x, nw)` use the time-halfbandwidth product, `nw`, to obtain the multitaper PSD estimate. The time-halfbandwidth product controls the frequency resolution of the multitaper estimate. `pmtm` uses  $2nw - 1$  Slepian tapers in the PSD estimate.

`pxx = pmtm(x, nw, nfft)` uses `nfft` points in the DFT. If `nfft` is greater than the signal length, `x` is zero-padded to length `nfft`. If `nfft` is less than the signal length, the signal is wrapped modulo `nfft`.

`[ pxx, w ] = pmtm( ___ )` returns the normalized frequency vector, `w`. If `pxx` is a one-sided PSD estimate, `w` spans the interval  $[0, \pi]$  if `nfft` is even and  $[0, \pi)$  if `nfft` is odd. If `pxx` is a two-sided PSD estimate, `w` spans the interval  $[0, 2\pi)$ .

`[ pxx, f ] = pmtm( ___, fs )` returns a frequency vector, `f`, in cycles per unit time. The sampling frequency, `fs`, is the number of samples per unit time. If the unit of time is seconds, then `f` is in cycles/sec (Hz). For real-valued signals, `f` spans the interval  $[0, fs/2]$  when `nfft` is even and  $[0, fs/2)$  when `nfft` is odd. For complex-valued signals, `f` spans the interval  $[0, fs)$ .

`[ pxx, w ] = pmtm(x, nw, w)` returns the two-sided multitaper PSD estimates at the normalized frequencies specified in the vector, `w`. `w` must contain at least two elements.

`[ pxx, f ] = pmtm(x, nw, f, fs)` returns the two-sided multitaper PSD estimates at the frequencies specified in the vector, `f`. `f` must contain at least two elements. The frequencies in `f` are in cycles per unit time. The sampling frequency, `fs`, is the number of samples per unit time. If the unit of time is seconds, then `f` is in cycles/second (Hz).

`[ ___ ] = pmtm( ___, method)` combines the individual tapered PSD estimates using the method, `method`. `method` can be one of: 'adapt' (default), 'eigen', or 'unity'.

`[ ___ ] = pmtm(x, e, v)` uses the tapers in the  $N$ -by- $K$  matrix `e` with concentrations `v` in the frequency band  $[-w, w]$ .  $N$  is the length of the input signal, `x`. Use `dpss` to obtain the Slepian tapers and corresponding concentrations.

`[ ___ ] = pmtm(x, dpss_params)` uses the cell array, `dpss_params`, to pass input arguments to `dpss` except the number of elements in the sequences. The number of elements in the sequences is the first input argument to `dpss` and is not included in `dpss_params`. An example of this usage is `pxx = pmtm(randn(1000, 1), {2.5, 3})`.

`[ ___ ] = pmtm( ___, 'DropLastTaper', dropflag)` specifies whether `pmtm` drops the last taper in the computation of the multitaper PSD estimate. `dropflag` is a logical. The default value of `dropflag` is `true` and the last taper is not used in the PSD estimate.

[ \_\_\_ ] = `pmtm( ___, freqrange)` returns the multitaper PSD estimate over the frequency range specified by `freqrange`. Valid options for `freqrange` are 'onesided', 'twosided', and 'centered'.

[ \_\_\_, pxxc ] = `pmtm( ___, 'ConfidenceLevel', probability)` returns the probability  $\times$  100% confidence intervals for the PSD estimate in `pxxc`.

`pmtm( ___)` with no output arguments plots the multitaper PSD estimate in the current figure window.

## Examples

### Multitaper Estimate Using Default Inputs

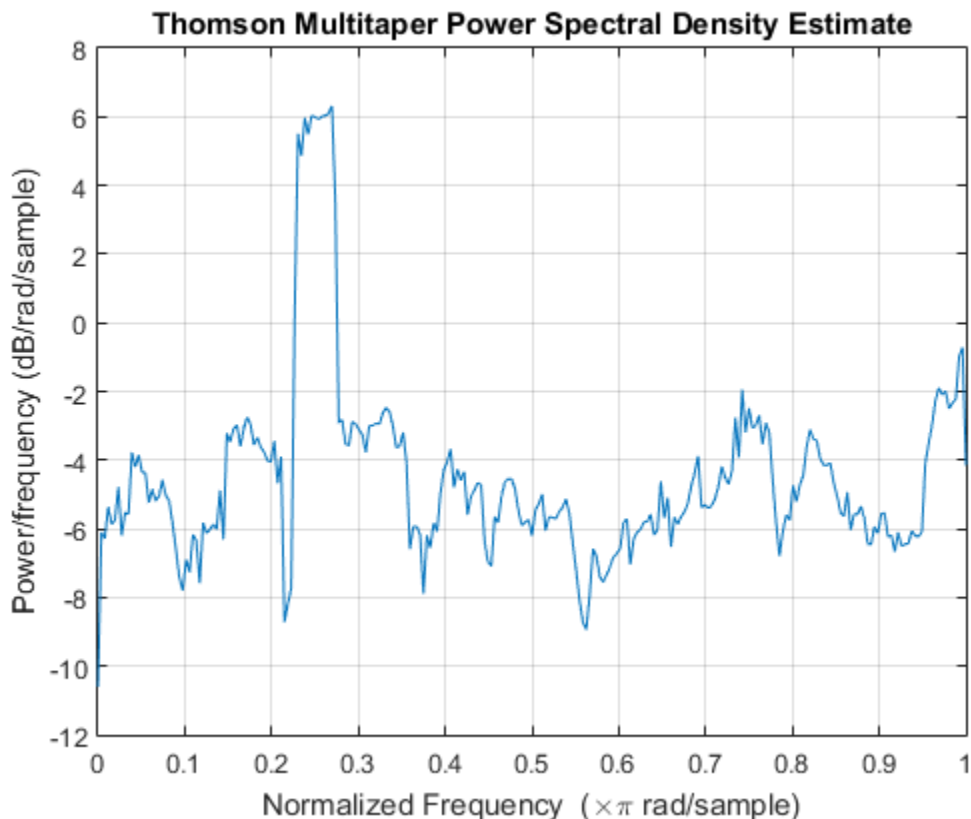
Obtain the multitaper PSD estimate of an input signal consisting of a discrete-time sinusoid with an angular frequency of  $\pi/4$  rad/sample with additive  $N(0,1)$  white noise.

Create a sine wave with an angular frequency of  $\pi/4$  rad/sample with additive  $N(0,1)$  white noise. The signal is 320 samples in length. Obtain the multitaper PSD estimate using the default time-halfbandwidth product of 4 and DFT length. The default number of DFT points is 512. Because the signal is real-valued, the PSD estimate is one-sided and there are  $512/2+1$  points in the PSD estimate.

```
n = 0:319;  
x = cos(pi/4*n)+randn(size(n));  
pxx = pmtm(x);
```

Plot the multitaper PSD estimate.

```
pmtm(x)
```



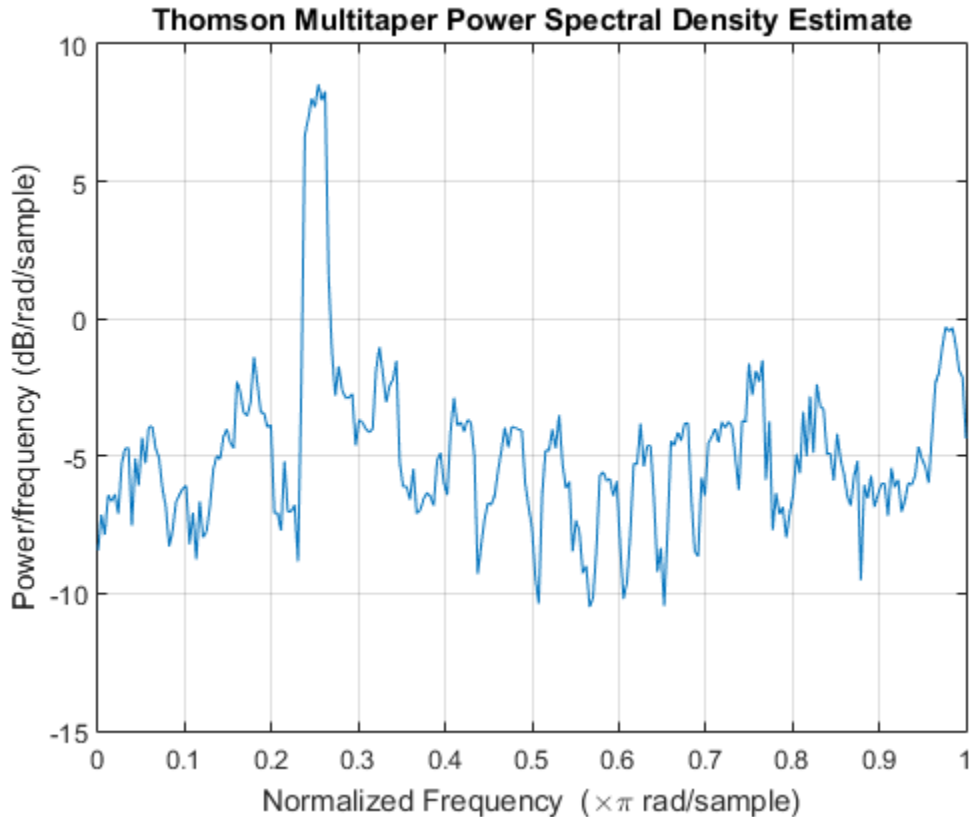
### Specify Time-Halfbandwidth Product

Obtain the multitaper PSD estimate with a specified time-halfbandwidth product.

Create a sine wave with an angular frequency of  $\pi/4$  rad/sample with additive  $N(0,1)$  white noise. The signal is 320 samples in length. Obtain the multitaper PSD estimate with a time-halfbandwidth product of 2.5. The resolution bandwidth is  $[-2.5\pi/320, 2.5\pi/320]$  rad/sample. The default number of DFT points is 512. Because the signal is real-valued, the PSD estimate is one-sided and there are  $512/2+1$  points in the PSD estimate.

```
n = 0:319;
x = cos(pi/4*n)+randn(size(n));
```

```
pmtm(x,2.5)
```

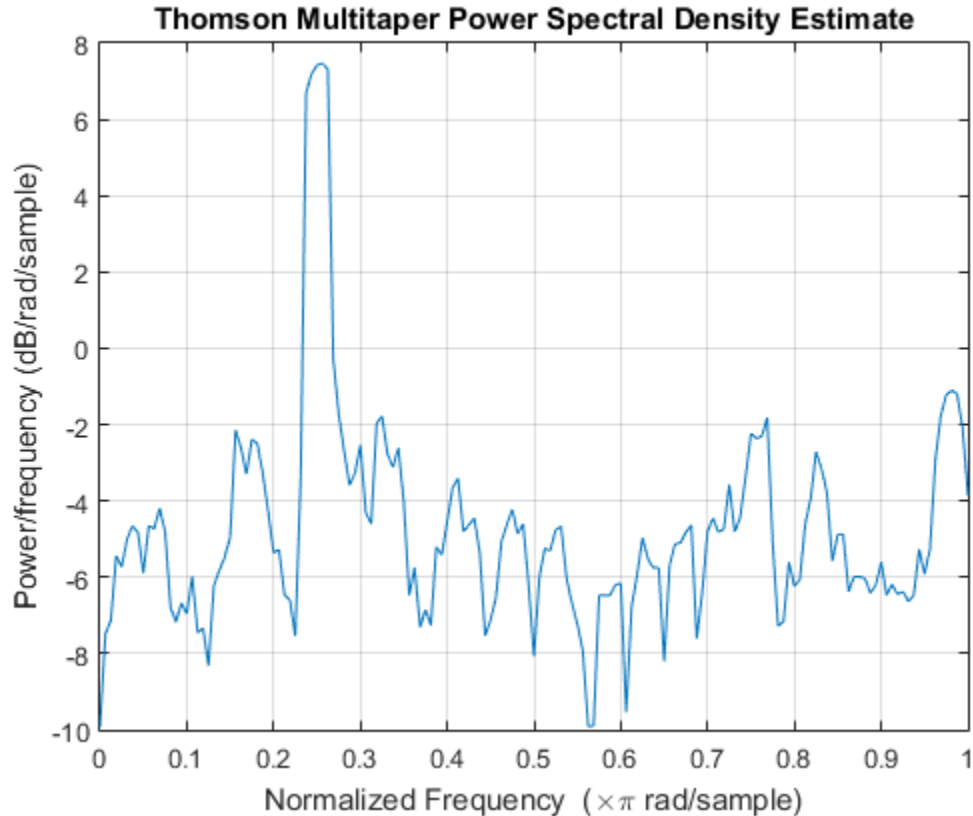


### DFT Length Equal to Signal Length

Obtain the multitaper PSD estimate of an input signal consisting of a discrete-time sinusoid with an angular frequency of  $\pi/4$  rad/sample with additive  $N(0,1)$  white noise. Use a DFT length equal to the signal length.

Create a sine wave with an angular frequency of  $\pi/4$  rad/sample with additive  $N(0,1)$  white noise. The signal is 320 samples in length. Obtain the multitaper PSD estimate with a time-halfbandwidth product of 3 and a DFT length equal to the signal length. Because the signal is real-valued, the one-sided PSD estimate is returned by default with a length equal to  $320/2+1$ .

```
n = 0:319;
x = cos(pi/4*n)+randn(size(n));
pmtm(x,3,length(x))
```



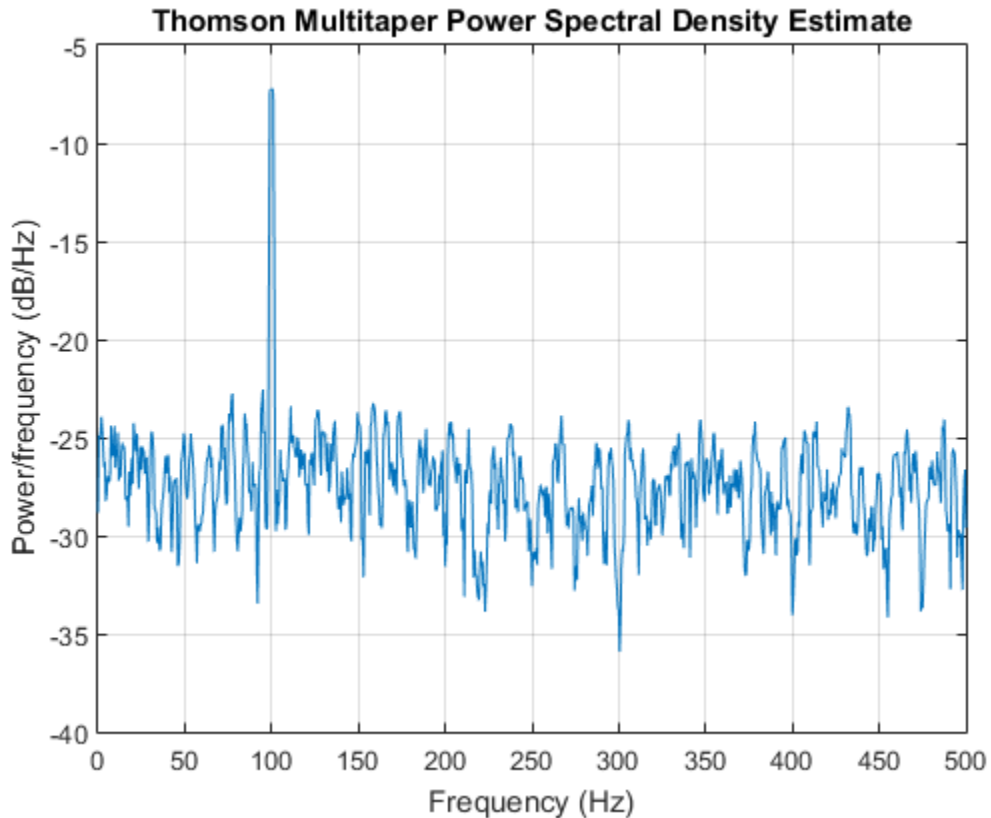
### Multitaper Estimate with Sample Rate

Obtain the multitaper PSD estimate of a signal sampled at 1 kHz. The signal is a 100 Hz sine wave in additive  $N(0,1)$  white noise. The signal duration is 2 s. Use a time-halfbandwidth product of 3 and DFT length equal to the signal length.

```
fs = 1000;
t = 0:1/fs:2-1/fs;
x = cos(2*pi*100*t)+randn(size(t));
[pxx,f] = pmtm(x,3,length(x),fs);
```

Plot the multitaper PSD estimate.

```
pmtm(x,3,length(x),fs)
```



### Average Single-Taper Estimates with Unity Weights

Obtain a multitaper PSD estimate where the individual tapered direct spectral estimates are given equal weight in the average.

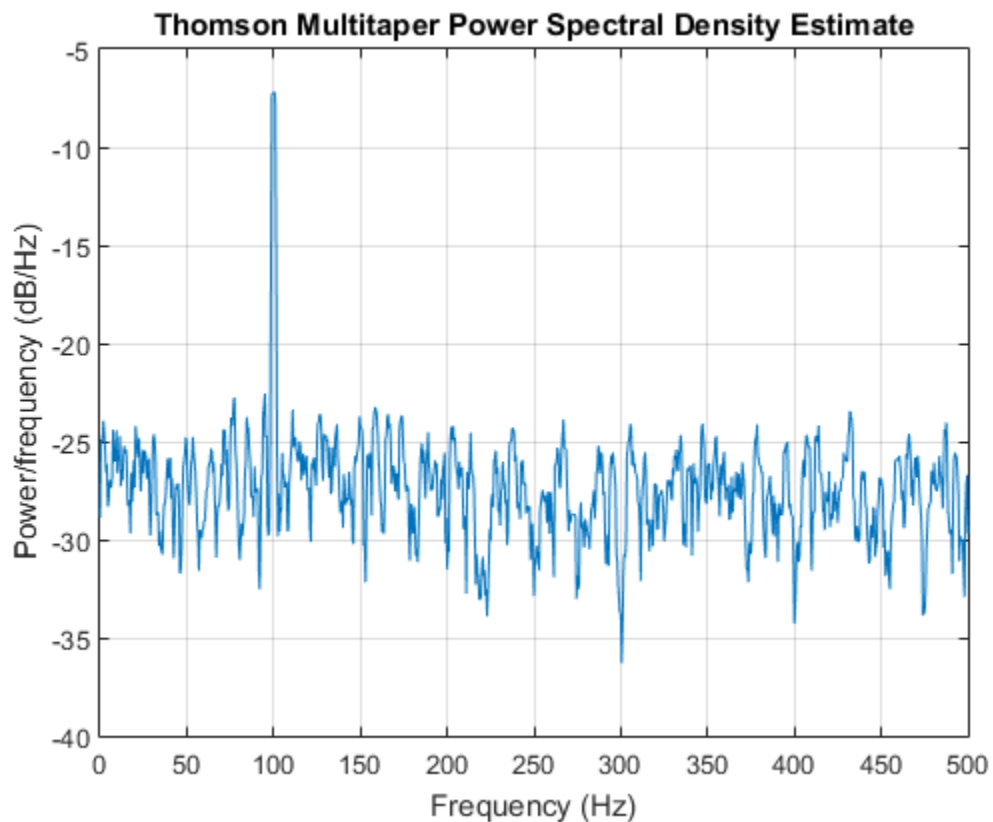
Obtain the multitaper PSD estimate of a signal sampled at 1 kHz. The signal is a 100 Hz sine wave in additive  $N(0,1)$  white noise. The signal duration is 2 s. Use a time-halfbandwidth product of 3 and a DFT length equal to the signal length. Use the 'unity' option to give equal weight in the average to each of the individual tapered direct spectral estimates.



```
fs = 1000;  
t = 0:1/fs:2-1/fs;  
x = cos(2*pi*100*t)+randn(size(t));  
[pxx,f] = pmtm(x,3,length(x),fs,'unity');
```

Plot the multitaper PSD estimate.

```
pmtm(x,3,length(x),fs,'unity')
```



### DPSS Sequences and Their Frequency-Domain Concentrations

This example examines the frequency-domain concentrations of the DPSS sequences. The example produces a multitaper PSD estimate of an input signal by precomputing

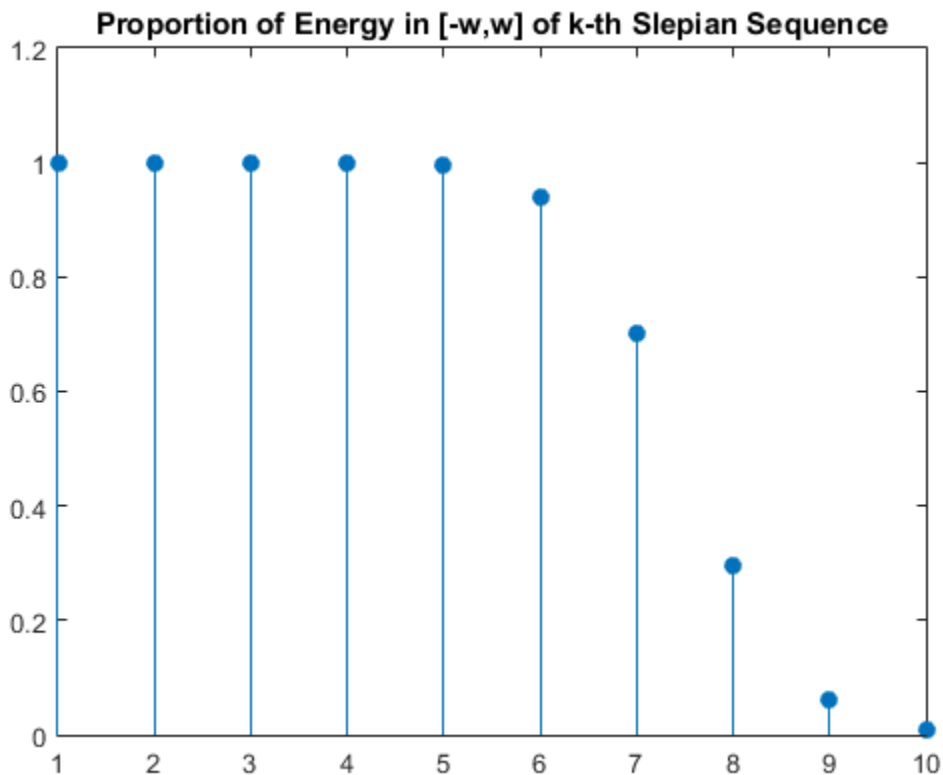
the Slepian sequences and selecting only those with more than 99% of their energy concentrated in the resolution bandwidth.

The signal is a 100 Hz sine wave in additive  $N(0,1)$  white noise. The signal duration is 2 s.

```
fs = 1000;  
t = 0:1/fs:2-1/fs;  
x = cos(2*pi*100*t)+randn(size(t));
```

Set the time-halfbandwidth product to 3.5. For the signal length of 2000 samples and a sampling interval of 0.001 seconds, this results in a resolution bandwidth of  $[-1.75, 1.75]$  Hz. Calculate the first 10 Slepian sequences and examine their frequency concentrations in the specified resolution bandwidth.

```
[e,v] = dpss(length(x),3.5,10);  
stem(1:length(v),v,'filled')  
ylim([0 1.2])  
title('Proportion of Energy in [-w,w] of k-th Slepian Sequence')
```



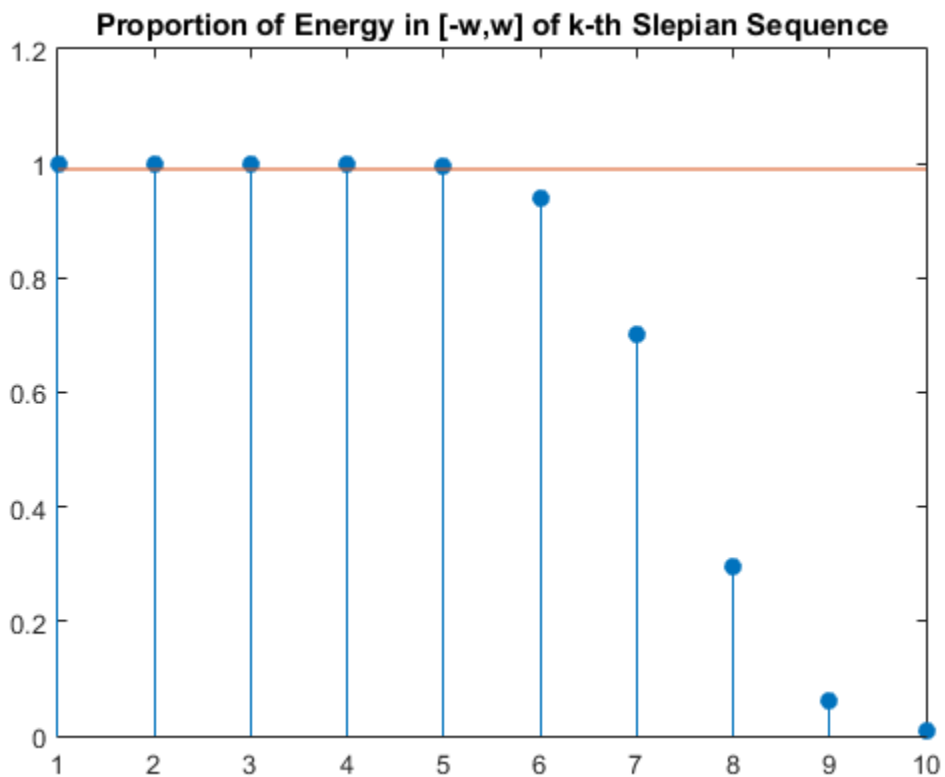
Determine the number of Slepian sequences with energy concentrations greater than 99%. Using the selected DPSS sequences, obtain the multitaper PSD estimate. Set 'DropLastTaper' to false to use all the selected tapers.

```
hold on
plot(1:length(v),0.99*ones(length(v),1))
idx = find(v>0.99,1,'last')

[pxx,f] = pmtm(x,e(:,1:idx),v(1:idx),length(x),fs,'DropLastTaper',false);

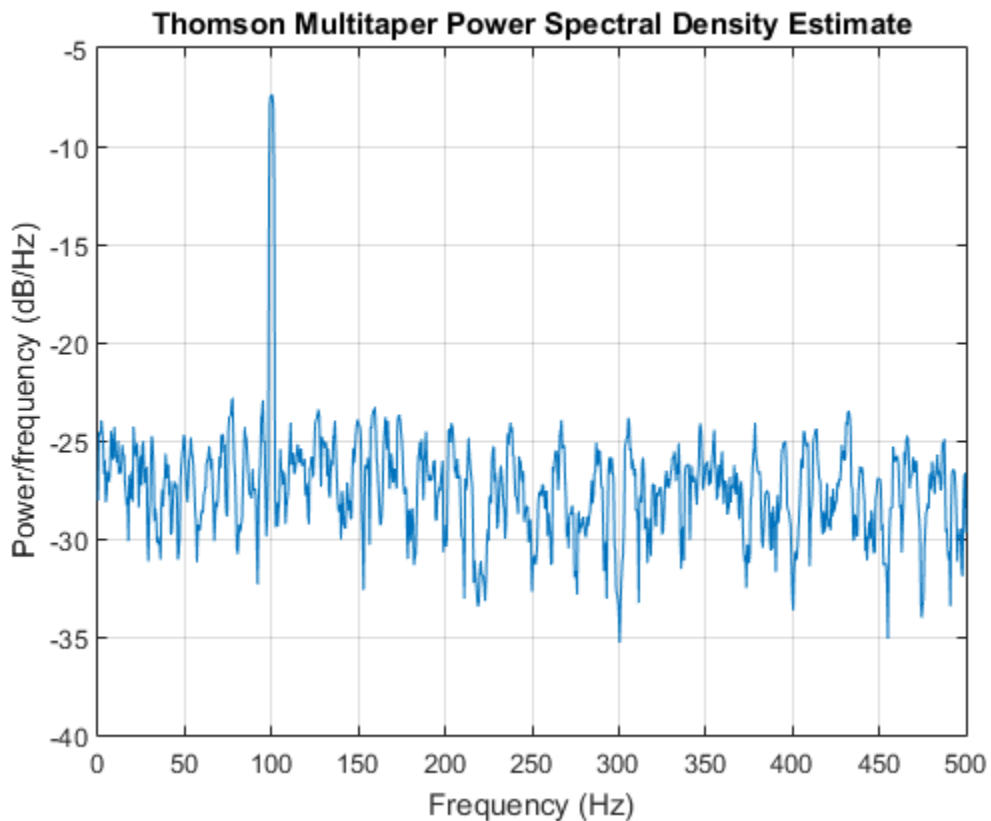
idx =
```

5



Plot the multitaper PSD estimate.

```
figure  
pmtm(x,e(:,1:idx),v(1:idx),length(x),fs,'DropLastTaper',false)
```



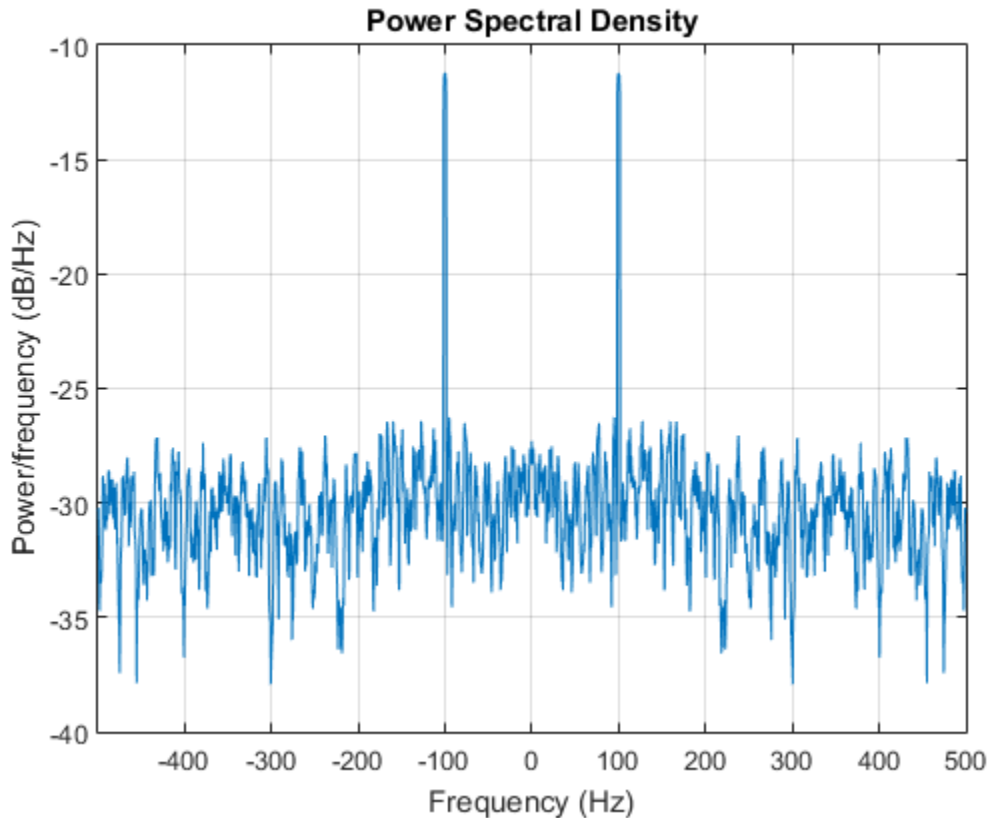
### DC-Centered Multitaper PSD Estimate

Obtain the multitaper PSD estimate of a 100 Hz sine wave in additive  $N(0,1)$  noise. The data are sampled at 1 kHz. Use the 'centered' option to obtain the DC-centered PSD.

```
fs = 1000;
t = 0:1/fs:2-1/fs;
x = cos(2*pi*100*t)+randn(size(t));
[pxx,f] = pmtm(x,3.5,length(x),fs,'centered');
```

Plot the DC-centered PSD estimate.

```
pmtm(x,3.5,length(x),fs,'centered')
```



### Upper and Lower 95%-Confidence Bounds

The following example illustrates the use of confidence bounds with the multitaper PSD estimate. While not a necessary condition for statistical significance, frequencies in the multitaper PSD estimate where the lower confidence bound exceeds the upper confidence bound for surrounding PSD estimates clearly indicate significant oscillations in the time series.

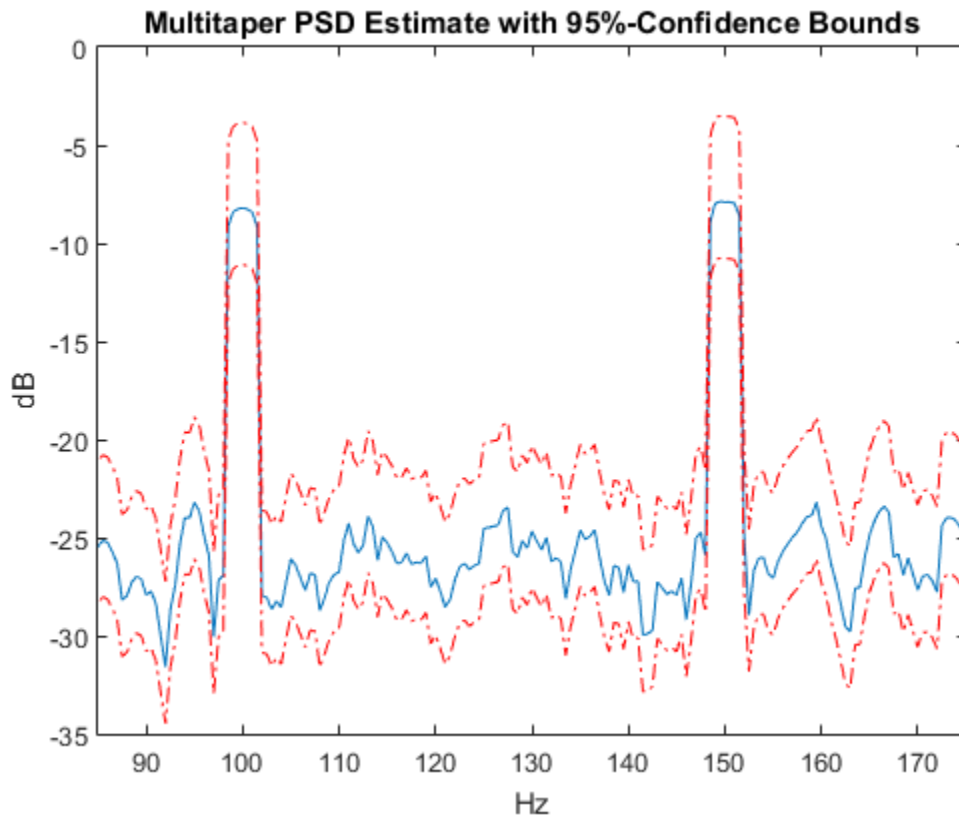
Create a signal consisting of the superposition of 100-Hz and 150-Hz sine waves in additive white  $N(0,1)$  noise. The amplitude of the two sine waves is 1. The sampling frequency is 1 kHz. The signal is 2 s in duration.

```
fs = 1000;
```

```
t = 0:1/fs:2-1/fs;  
x = cos(2*pi*100*t)+cos(2*pi*150*t)+randn(size(t));
```

Obtain the multitaper PSD estimate with 95%-confidence bounds. Plot the PSD estimate along with the confidence interval and zoom in on the frequency region of interest near 100 and 150 Hz.

```
[pxx,f,pxxc] = pmtm(x,3.5,length(x),fs,'ConfidenceLevel',0.95);  
  
plot(f,10*log10(pxx))  
hold on  
plot(f,10*log10(pxxc),'r-.')  
xlim([85 175])  
xlabel('Hz')  
ylabel('dB')  
title('Multitaper PSD Estimate with 95%-Confidence Bounds')
```



At 100 and 150 Hz, the lower confidence bound exceeds the upper confidence bounds for surrounding PSD estimates.

### Multitaper PSD Estimate of a Multichannel Signal

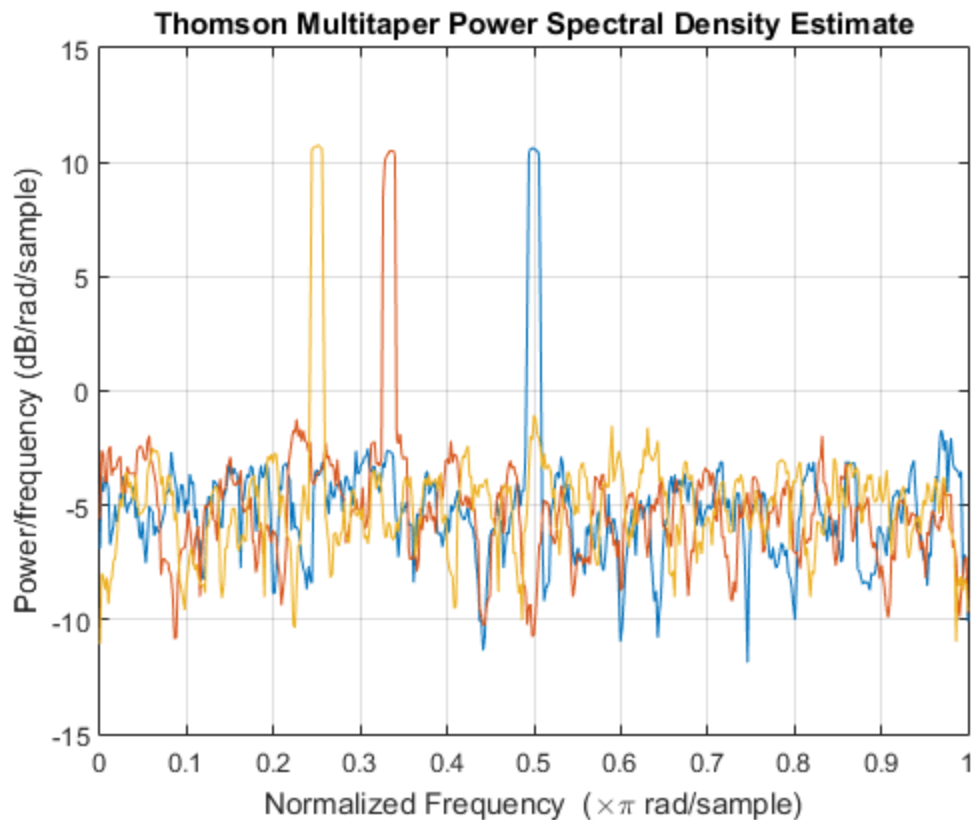
Generate 1024 samples of a multichannel signal consisting of three sinusoids in additive  $N(0, 1)$  white Gaussian noise. The sinusoids' frequencies are  $\pi/2$ ,  $\pi/3$ , and  $\pi/4$  rad/sample. Estimate the PSD of the signal using Thomson's multitaper method and plot it.

```
N = 1024;
n = 0:N-1;
```

```
w = pi./[2;3;4];
```



```
x = cos(w*n)' + randn(length(n),3);  
pmtm(x)
```



## Input Arguments

### **x** — Input signal

vector | matrix

Input signal, specified as a row or column vector, or as a matrix. If  $x$  is a matrix, then its columns are treated as independent channels.

Example: `cos(pi/4*(0:159))+randn(1,160)` is a single-channel row-vector signal.

Example: `cos(pi./[4;2]*(0:159))'+randn(160,2)` is a two-channel signal.

Data Types: `single` | `double`

Complex Number Support: Yes

### **nw — Time-halfbandwidth product**

4 (default) | positive scalar

Time-halfbandwidth product, specified as a positive scalar. In multitaper spectral estimation, the user specifies the resolution bandwidth of the multitaper estimate  $[-W,W]$  where  $W = k/N\Delta t$  for some small  $k > 1$ . Equivalently,  $W$  is some small multiple of the frequency resolution of the DFT. The time-halfbandwidth product is the product of the resolution halfbandwidth and the number of samples in the input signal,  $N$ . The number of Slepian tapers whose Fourier transforms are well-concentrated in  $[-W,W]$  (eigenvalues close to unity) is  $2NW - 1$ .

### **nfft — Number of DFT points**

`max(256,2^nextpow2(length(x)))` (default) | integer | []

Number of DFT points, specified as a positive integer. For a real-valued input signal,  $x$ , the PSD estimate, `pxx` has length  $(nfft/2 + 1)$  if `nfft` is even, and  $(nfft + 1)/2$  if `nfft` is odd. For a complex-valued input signal,  $x$ , the PSD estimate always has length `nfft`. If `nfft` is specified as empty, the default `nfft` is used.

Data Types: `single` | `double`

### **fs — Sampling frequency**

positive scalar

Sampling frequency, specified as a positive scalar. The sampling frequency is the number of samples per unit time. If the unit of time is seconds, the sampling frequency has units of hertz.

### **w — Normalized frequencies**

vector

Normalized frequencies, specified as a row or column vector with at least 2 elements. Normalized frequencies are in rad/sample.

Example: `w = [pi/4 pi/2]`

Data Types: `double`

**f — Cyclical frequencies**

vector

Cyclical frequencies, specified as a row or column vector with at least 2 elements. The frequencies are in cycles per unit time. The unit time is specified by the sampling frequency, `fs`. If `fs` has units of samples/second, then `f` has units of Hz.

Example: `fs = 1000; f = [100 200]`

Data Types: double

**method — Weights on individual tapered PSD estimates**

'adapt' (default) | 'eigen' | 'unity'

Weights on individual tapered PSD estimates, specified as one of 'adapt', 'eigen', or 'unity'. The default is Thomson's adaptive frequency-dependent weights, 'adapt'. The calculation of these weights is detailed on pp. 368–370 in [1]. The 'eigen' method weights each tapered PSD estimate by the eigenvalue (frequency concentration) of the corresponding Slepian taper. The 'unity' method weights each tapered PSD estimate equally.

**e — DPSS (Slepian) sequences**

matrix

DPSS (Slepian) sequences, specified as a N-by-K matrix where N is the length of the input signal, `x`. The matrix `e` is the output of `dpss`.

**v — Eigenvalues for DPSS (Slepian) sequences**

vector

Eigenvalues for DPSS (Slepian) sequences, specified as a column vector. The eigenvalues for the DPSS sequences indicate the proportion of the sequence energy concentrated in the resolution bandwidth,  $[-W, W]$ . The eigenvalues range lie in the interval (0,1) and generally the first  $2NW-1$  eigenvalues are close to 1 and then decrease toward 0.

**dpss\_params — Input arguments for dpss**

cell array

Input arguments for `dpss`, specified as a cell array. The first input argument to `dpss` is the length of the DPSS sequences and is omitted from `dpss_params`. The length of the DPSS sequences is obtained from the length of the input signal, `x`.

Example: `{3.5, 5}`

**dropflag** — Flag indicating whether to drop or keep the last DPSS sequence

true (default) | false

Flag indicating whether to drop or keep the last DPSS sequence, specified as a logical. The default is `true` and `pmtm` drops the last taper. In a multitaper estimate, the first  $2NW - 1$  DPSS sequences have eigenvalues close to unity. If you use less than  $2NW - 1$  sequences, it is likely that all the tapers have eigenvalues close to 1 and you can specify `dropflag` as `false` to keep the last taper.

**freqrange** — Frequency range for PSD estimate

'onesided' | 'twosided' | 'centered'

Frequency range for the PSD estimate, specified as a one of 'onesided', 'twosided', or 'centered'. The default is 'onesided' for real-valued signals and 'twosided' for complex-valued signals. The frequency ranges corresponding to each option are

- 'onesided' — returns the one-sided PSD estimate of a real-valued input signal, `x`. If `nfft` is even, `pxx` has length  $nfft/2 + 1$  and is computed over the interval  $[0, \pi]$  rad/sample. If `nfft` is odd, the length of `pxx` is  $(nfft + 1)/2$  and the interval is  $[0, \pi]$  rad/sample. When `fs` is optionally specified, the corresponding intervals are  $[0, fs/2]$  cycles/unit time and  $[0, fs/2)$  cycles/unit time for even and odd length `nfft` respectively.
- 'twosided' — returns the two-sided PSD estimate for either the real-valued or complex-valued input, `x`. In this case, `pxx` has length `nfft` and is computed over the interval  $[0, 2\pi)$  rad/sample. When `fs` is optionally specified, the interval is  $[0, fs)$  cycles/unit time.
- 'centered' — returns the centered two-sided PSD estimate for either the real-valued or complex-valued input, `x`. In this case, `pxx` has length `nfft` and is computed over the interval  $(-\pi, \pi]$  rad/sample for even length `nfft` and  $(-\pi, \pi)$  rad/sample for odd length `nfft`. When `fs` is optionally specified, the corresponding intervals are  $(-fs/2, fs/2]$  cycles/unit time and  $(-fs/2, fs/2)$  cycles/unit time for even and odd length `nfft` respectively.

Data Types: char

**probability** — Confidence interval for PSD estimate

0.95 (default) | scalar in the range (0,1)

Coverage probability for the true PSD, specified as a scalar in the range (0,1). The output, `pxxc`, contains the lower and upper bounds of the probability  $\times 100\%$  interval estimate for the true PSD.

## Output Arguments

### **pxx** — PSD estimate

vector | matrix

PSD estimate, returned as a real-valued, nonnegative column vector or matrix. Each column of `pxx` is the PSD estimate of the corresponding column of `x`. The units of the PSD estimate are in squared magnitude units of the time series data per unit frequency. For example, if the input data is in volts, the PSD estimate is in units of squared volts per unit frequency. For a time series in volts, if you assume a resistance of  $1 \Omega$  and specify the sampling frequency in hertz, the PSD estimate is in watts per hertz.

Data Types: `single` | `double`

### **w** — Normalized frequencies

vector

Normalized frequencies, returned as a real-valued column vector. If `pxx` is a one-sided PSD estimate, `w` spans the interval  $[0, \pi]$  if `nfft` is even and  $[0, \pi)$  if `nfft` is odd. If `pxx` is a two-sided PSD estimate, `w` spans the interval  $[0, 2\pi)$ . For a DC-centered PSD estimate, `f` spans the interval  $(-\pi, \pi]$  rad/sample for even length `nfft` and  $(-\pi, \pi)$  radians/sample for odd length `nfft`.

Data Types: `double`

### **f** — Cyclical frequencies

vector

Cyclical frequencies, returned as a real-valued column vector. For a one-sided PSD estimate, `f` spans the interval  $[0, fs/2]$  when `nfft` is even and  $[0, fs/2)$  when `nfft` is odd. For a two-sided PSD estimate, `f` spans the interval  $[0, fs)$ . For a DC-centered PSD estimate, `f` spans the interval  $(-fs/2, fs/2]$  cycles/unit time for even length `nfft` and  $(-fs/2, fs/2)$  cycles/unit time for odd length `nfft`.

Data Types: `double`

### **pxxc** — Confidence bounds

matrix

Confidence bounds, returned as a matrix with real-valued elements. The row size of the matrix is equal to the length of the PSD estimate, `pxx`. `pxxc` has twice as many columns as `pxx`. Odd-numbered columns contain the lower bounds of the confidence intervals, and even-numbered columns contain the upper bounds. Thus, `pxxc(m, 2*n - 1)` is the lower

confidence bound and  $\text{pxxc}(m, 2*n)$  is the upper confidence bound corresponding to the estimate  $\text{pxx}(m, n)$ . The coverage probability of the confidence intervals is determined by the value of the probability input.

Data Types: `single` | `double`

## More About

### Discrete Prolate Spheroidal (Slepian) Sequences

The derivation of the Slepian sequences proceeds from the discrete-time — continuous frequency concentration problem. For all  $\ell^2$  sequences index-limited to  $0, 1, \dots, N - 1$ , the problem seeks the sequence having the maximal concentration of its energy in a frequency band  $[-W, W]$  with  $|W| < 1/2\Delta t$ .

This amounts to finding the eigenvalues and corresponding eigenvectors of an  $N$ -by- $N$  self-adjoint positive semi-definite operator. Therefore, the eigenvalues are real and nonnegative and eigenvectors corresponding to distinct eigenvalues are mutually orthogonal. In this particular problem, the eigenvalues are bounded by 1 and the eigenvalue is the measure of the sequence's energy concentration in the frequency interval  $[-W, W]$ .

The eigenvalue problem is given by

$$\sum_{n=0}^{N-1} \frac{\sin(2\pi W(n-m))}{\pi(n-m)} g_n = \lambda_k(N, W) g_m \quad m = 0, 1, 2, \dots, N - 1$$

The 0th-order DPSS sequence,  $g_0$  is the eigenvector corresponding to the largest eigenvalue. The 1-st order DPSS sequence,  $g_1$  is the eigenvector corresponding to the next largest eigenvalue and is orthogonal to the 0-th order sequence. The 2nd-order DPSS sequence,  $g_2$ , is the eigenvector corresponding to the third largest eigenvalue and is orthogonal to the 0-th order and 1-st order DPSS sequences. Because the operator is  $N$ -by- $N$ , there are  $N$  eigenvectors. However, it can be shown that for a given sequence length  $N$  and a specified bandwidth  $[-W, W]$ , there are approximately  $2NW - 1$  DPSS sequences with eigenvalues very close to unity.

### Multitaper Spectral Estimation

The periodogram is not a consistent estimator of the true power spectral density of a wide-sense stationary process. To produce a consistent estimate of the PSD, the

multitaper method averages modified periodograms obtained using a family of mutually orthogonal tapers (windows). In addition to mutual orthogonality, the tapers also have optimal time-frequency concentration properties. Both the orthogonality and time-frequency concentration of the tapers is critical to the success of the multitaper technique. See “Discrete Prolate Spheroidal (Slepian) Sequences” on page 1-1128 for a brief description of the Slepian sequences used in Thomson’s multitaper method.

The multitaper method uses  $K$  modified periodograms with each one obtained using a different Slepian sequence as the window. Let

$$S_k(f) = \Delta t \left| \sum_{n=0}^{N-1} g_{k,n} x_n e^{-i2\pi f n \Delta t} \right|^2$$

denote the modified periodogram obtained with the  $k$ -th Slepian sequence,  $g_{k,n}$ .

In the simplest form, the multitaper method simply averages the  $K$  modified periodograms to produce the multitaper PSD estimate.

$$S^{(\text{MT})}(f) = \frac{1}{K} \sum_{k=0}^{K-1} S_k(f)$$

Note the difference between the multitaper PSD estimate and Welch’s method. Both methods reduce the variability in the periodogram by averaging over approximately uncorrelated estimates of the PSD. However, the two approaches differ in how they produce these uncorrelated PSD estimates. The multitaper method uses the entire signal in each modified periodogram. The orthogonality of the Slepian tapers decorrelates the different modified periodograms. Welch’s overlapped segment averaging approach uses segments of the signal in each modified periodogram and the segmenting decorrelates the different modified periodograms.

The preceding equation corresponds to the 'unity' option in pmtm. However, as explained in “Discrete Prolate Spheroidal (Slepian) Sequences” on page 1-1128, the Slepian sequences do not possess equal energy concentration in the frequency band of interest. The higher the order of the Slepian sequence, the less concentrated the sequence energy is in the band  $[-W, W]$  with the concentration given by the eigenvalue. Consequently, it can be beneficial to use the eigenvalues to weight the  $K$  modified periodograms prior to averaging. This corresponds to the 'eigen' option in pmtm.

Using the sequence eigenvalues to produce a weighted average of modified periodograms accounts for the frequency concentration properties of the Slepian sequences. However, it does not account for the interaction between the power spectral density of the random process and the frequency concentration of the Slepian sequences. Specifically, frequency regions where the random process has little power are less reliably estimated in the modified periodograms using higher order Slepian sequences. This argues for an frequency-dependent adaptive process, which accounts not only for the frequency concentration of the Slepian sequence, but also for the power distribution in the time series. This adaptive weighting corresponds to the 'adapt' option in `pmtm` and is the default for computing the multitaper estimate.

## References

- [1] Percival, D. B., and A. T. Walden, *Spectral Analysis for Physical Applications: Multitaper and Conventional Univariate Techniques*. Cambridge, UK: 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` | `periodogram` | `pwelch`



# pmusic

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(...,freqrange)
[...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  $\mathbf{x}' * \mathbf{x}$  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  $\lambda_{\min}$ , the smallest estimated eigenvalue of the signal's correlation matrix. Eigenvalues below the threshold  $\lambda_{\min} * p(2)$  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                   | $[0, \pi]$  |
| Complex-valued          | 256                   | $[0, 2\pi]$                                       |

$[S, w] = \text{pmusic}(x, p, w)$  returns the pseudospectrum in the vector  $S$  computed at the normalized frequencies specified in vector  $w$ , which has two or more elements

$[S, w] = \text{pmusic}(\dots, \text{nfft})$  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          | $(\text{nfft}/2 + 1)$ | $[0, \pi]$   |

| Real/Complex Input Data | nfft Even/Odd | Length of S and w | Range of w  |
|-------------------------|---------------|-------------------|-------------|
| Real-valued             | Odd           | $(nfft + 1) / 2$  | $[0, \pi)$  |
| Complex-valued          | Even or odd   | nfft              | $[0, 2\pi)$ |

$[S, f] = pmusic(x, p, nfft, fs)$  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 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

$[S, f] = pmusic(\dots, 'corr')$  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  $2 \times p(1)$  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 below for related information on this syntax.

---

**Note** The arguments  $nwin$  and  $noverlap$  are ignored when you include the string 'corr' in the syntax.

---

$[...] = pmusic(..., freqrange)$  specifies the range of frequency values to include in  $f$  or  $w$ . This syntax is useful when  $x$  is real.  $freqrange$  can be either:

- 'onesided' — returns the one-sided PSD of a real input signal,  $x$ . If  $nfft$  is even,  $Pxx$  has length  $nfft/2 + 1$  and is computed over the interval  $[0, \pi]$ . If  $nfft$  is odd, the length of  $Pxx$  is  $(nfft + 1)/2$  and the frequency interval is  $[0, \pi]$ . When you specify  $fs$ , the intervals are  $[0, fs/2)$  and  $[0, fs/2]$  for even and odd length  $nfft$  respectively.
- 'twosided' — returns the two-sided PSD for either real or complex input,  $x$ . In this case,  $Pxx$  has length  $nfft$  and is computed over the interval  $[0, 2\pi)$ . When you specify  $fs$ , the frequency interval is  $[0, fs)$ .
- 'centered' — returns the centered two-sided PSD for either real or complex input,  $x$ . In this case,  $Pxx$  has length  $nfft$  and is computed over the interval  $(-\pi, \pi]$  for even length  $nfft$  and  $(-\pi, \pi)$  for odd length  $nfft$ . When you specify  $fs$ , the frequency intervals are  $(-fs/2, fs/2]$  and  $(-fs/2, fs/2)$  for even and odd length  $nfft$  respectively.

---

**Note** You can put the string arguments  $freqrange$  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.

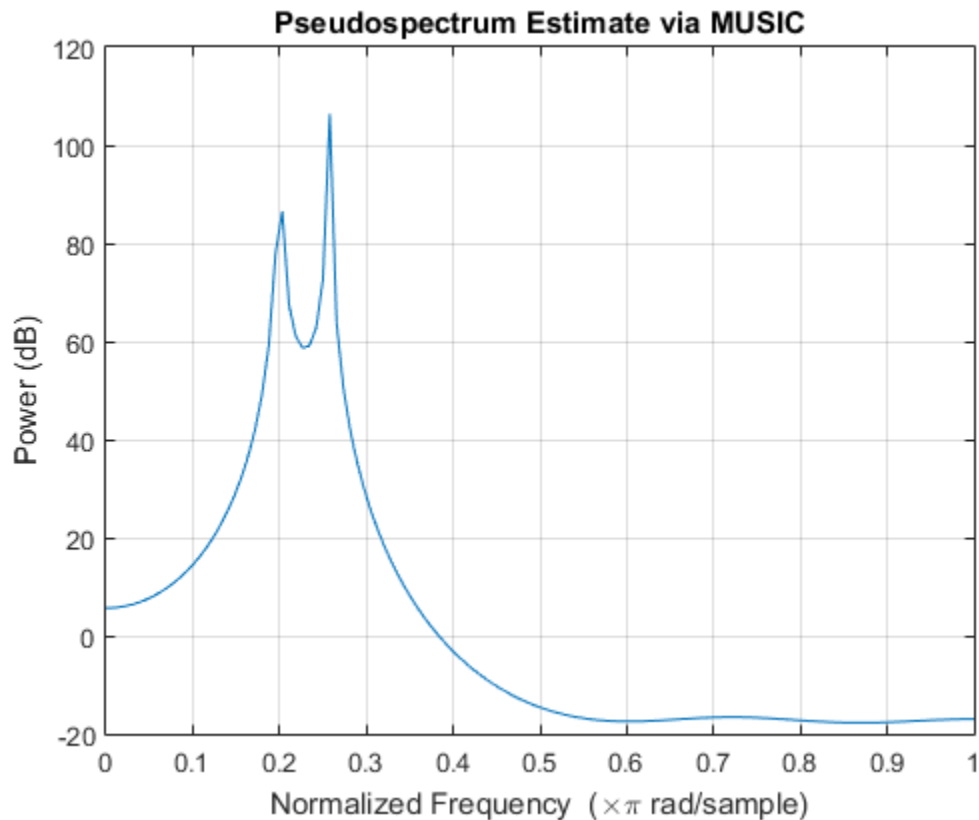
`pmusic(...)` with no output arguments plots the pseudospectrum in the current figure window.

## Examples

### **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.

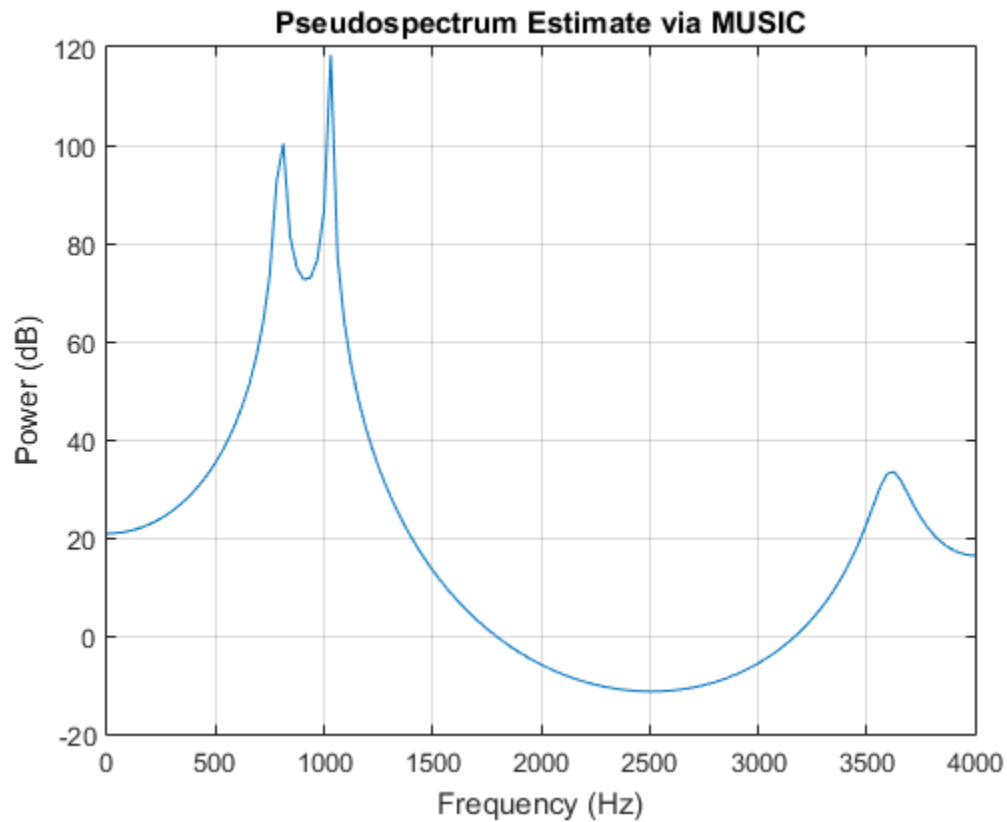
```
n = 0:199;
x = cos(0.257*pi*n) + sin(0.2*pi*n) + 0.01*randn(size(n));
pmusic(x,4)      % Set p to 4 because there are two real inputs
```



### Specifying Sampling Frequency and Subspace Dimensions

This example analyzes the same signal vector,  $x$ , with an eigenvalue cutoff of 10% above the minimum. Setting  $p(1) = \text{Inf}$  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:

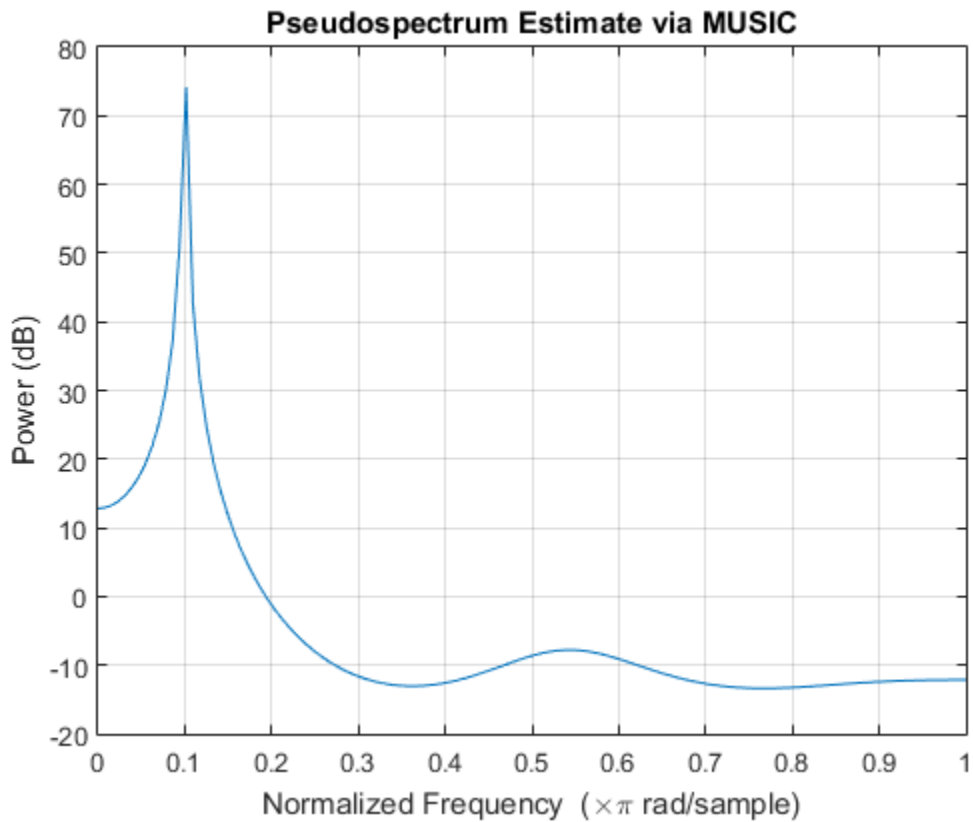
```
rng default
n = 0:199;
x = cos(0.257*pi*n) + sin(0.2*pi*n) + 0.01*randn(size(n));
[P,f] = pmusic(x,[Inf,1.1],[],8000,7); % Window length = 7
plot(f,20*log10(abs(P)))
xlabel 'Frequency (Hz)', ylabel 'Power (dB)'
title 'Pseudospectrum Estimate via MUSIC', grid on
```



### 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);  
pmusic(R,4, 'corr')
```

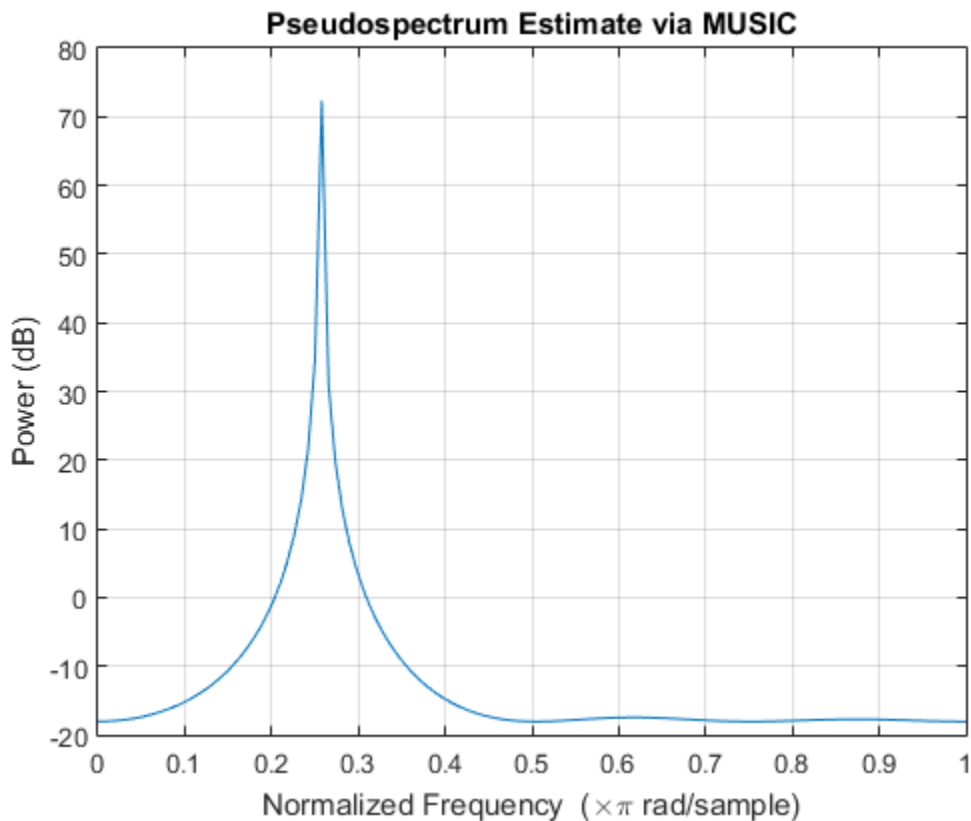


### Entering a Signal Data Matrix Generated from corrmtx

Enter a signal data matrix,  $X_m$ , generated from data using corrmtx.

```
n = 0:699;
x = cos(0.257*pi*(n)) + 0.1*randn(size(n));
Xm = corrmtx(x,7,'mod');
pmusic(Xm,2)
```

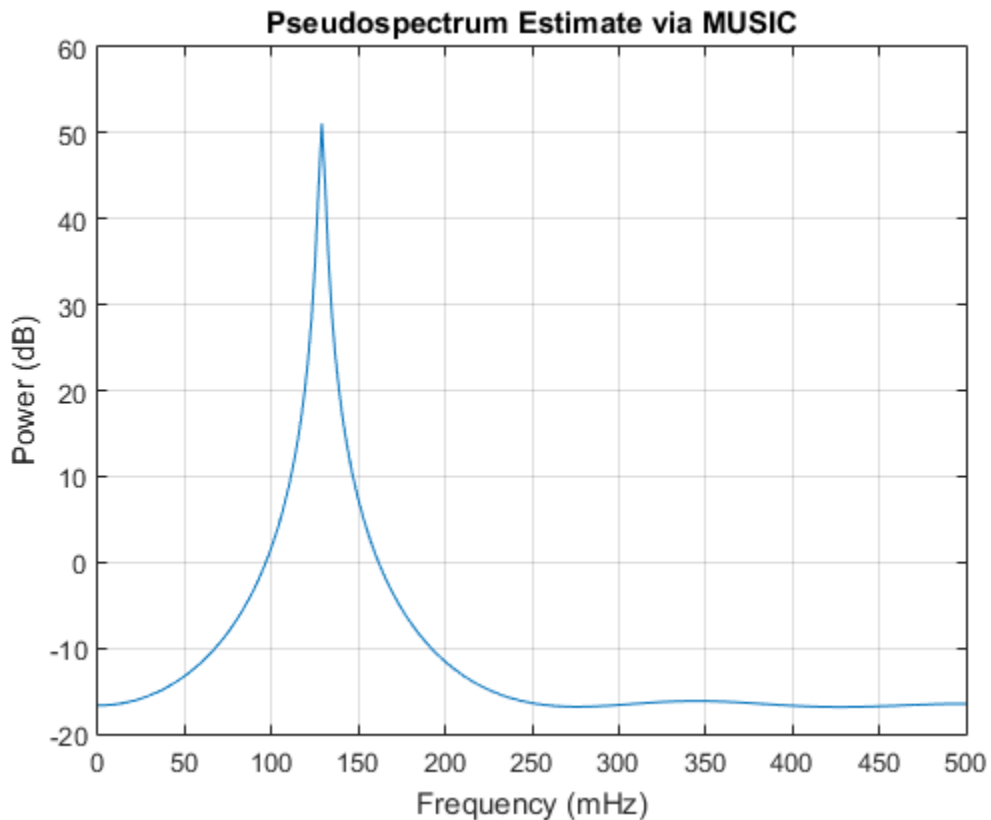




### Using Windowing to Create the Effect of a Signal Data Matrix

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.

```
n = 0:699;
x = cos(0.257*pi*(n)) + 0.1*randn(size(n));
[PP,ff] = pmusic(x,2,512,[],7,0);
pmusic(x,2,512,[],7,0)
```



## More About

### Tips

In the process of estimating the pseudospectrum, `pmusic` computes the noise and signal subspaces from the estimated eigenvectors  $v_j$  and eigenvalues  $\lambda_j$  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                     | <code>nwin</code> is specified as a scalar integer.  | <code>nwin</code>          |
| Row or column vector                     | <code>nwin</code> is specified as a vector.  | <code>length(nwin)</code>  |
| Row or column vector                     | <code>nwin</code> is not specified.  | $2 \times p(1)$            |
| $l$ -by- $m$ matrix                      | If <code>nwin</code> is specified as a scalar, it is not used. If <code>nwin</code> is specified as a vector, <code>length(nwin)</code> must equal $m$ . | $m$                        |
| $m$ -by- $m$ nonnegative definite matrix | The string ' <code>corr</code> ' is specified and <code>nwin</code> 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.

### Algorithms

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

$$P_{\text{MUSIC}}(f) = \frac{1}{e^H(f) \left( \sum_{k=p+1}^N v_k v_k^H \right) e(f)} = \frac{1}{\sum_{k=p+1}^N |v_k^H e(f)|^2}$$

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

$$v_k^H e(f)$$

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. Lawrence. *Digital Spectral Analysis*. Englewood Cliffs, NJ: Prentice-Hall, 1987, pp.373–378.
- [2] Schmidt, R. O. “Multiple Emitter Location and Signal Parameter Estimation.” *IEEE Transactions on Antennas and Propagation*. Vol.AP-34, March, 1986, pp.276–280.
- [3] Stoica, Petre, and Randolph L. Moses. *Spectral Analysis of Signals*. Upper Saddle River, NJ: Prentice Hall, 2005.

## See Also

corrmtx | pmtm | prony | pwelch | pburg | peig | periodogram | rooteig | rootmusic

# poly2ac

Convert prediction filter polynomial to autocorrelation sequence

## Syntax

```
r = poly2ac(a,efinal)
```

## Description

`r = poly2ac(a,efinal)` returns the autocorrelation vector, `r`, corresponding to the autoregressive prediction filter polynomial, `a`, and the final prediction error, `efinal`. `r` is approximately equal to the autocorrelation of the output of a prediction filter with coefficients `a`. If `a(1)` is not equal to 1, `poly2ac` normalizes the prediction filter polynomial by `a(1)`. `a(1)` cannot be 0.

## Examples

### Autocorrelation Sequence from Prediction Filter

Given a prediction filter polynomial, `a`, and a final prediction error, `efinal`, find the autocorrelation sequence.

```
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
```

## More About

### Tips

You can apply this function to both real and complex polynomials.

## References

- [1] Kay, Steven M. *Modern Spectral Estimation*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

### See Also

ac2poly | rc2ac | poly2rc

# poly2lsf

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

### Generate Line Spectral Frequencies

Given a vector, `a`, of prediction filter coefficients, generate the corresponding line spectral frequencies.

```
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, John R., John G. Proakis, and John H. L. Hansen. *Discrete-Time Processing of Speech Signals*. New York: Macmillan, 1993.

[2] Rabiner, Lawrence R., and Ronald W. Schafer. *Digital Processing of Speech Signals*. Englewood Cliffs, NJ: Prentice-Hall, 1978.

**See Also**

lsf2poly



# poly2rc

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`.

## Examples

### Find Reflection Coefficients from Prediction Filter Polynomial

Given a prediction filter polynomial, `a`, and a final prediction error, `efinal`, determine the reflection coefficients of the corresponding lattice structure and the zero-lag autocorrelation.

```
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  $\text{abs}(k(i)) \approx 1$  for any  $i$ , finding the reflection coefficients is an ill-conditioned problem. `poly2rc` returns some NaNs and provides a warning message in those cases.

## More About

### Tips

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)
```

### Algorithms

`poly2rc` implements this recursive relationship:

$$k(n) = a_n(n)$$
$$a_{n-1}(m) = \frac{a_n(m) - k(n)a_n(n-m)}{1 - k(n)^2}, \quad m = 1, 2, \dots, n-1$$

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, Steven M. *Modern Spectral Estimation*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

## See Also

ac2rc | latc2tf | latcfilt | poly2ac | rc2poly | tf2latc

## polyscale

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.

### More About

#### Tips

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`

# polystab

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. `a` is a vector of polynomial coefficients, normally in the  $z$ -domain:

$$A(z) = a(1) + a(2)z^{-1} + \dots + a(m+1)z^{-m}.$$

## Examples

### Convert Linear-Phase Filter to Minimum-Phase

Use the window method to design a 25th-order FIR filter with normalized cutoff frequency  $0.4\pi$  rad/sample. Verify that it has linear phase but not minimum phase.

```
h = fir1(25,0.4);
h_linphase = islinphase(h)
h_minphase = isminphase(h)
```

```
h_linphase =
```

```
1
```

```
h_minphase =
```

```
0
```

Use `polystab` to convert the linear-phase filter into a minimum-phase filter. Plot the phase responses of the filters.

```
hmin = polystab(h)*norm(h)/norm(polystab(h));
```

```
hmin_linphase = islinphase(hmin)
```

```
hmin_minphase = isminphase(hmin)
```

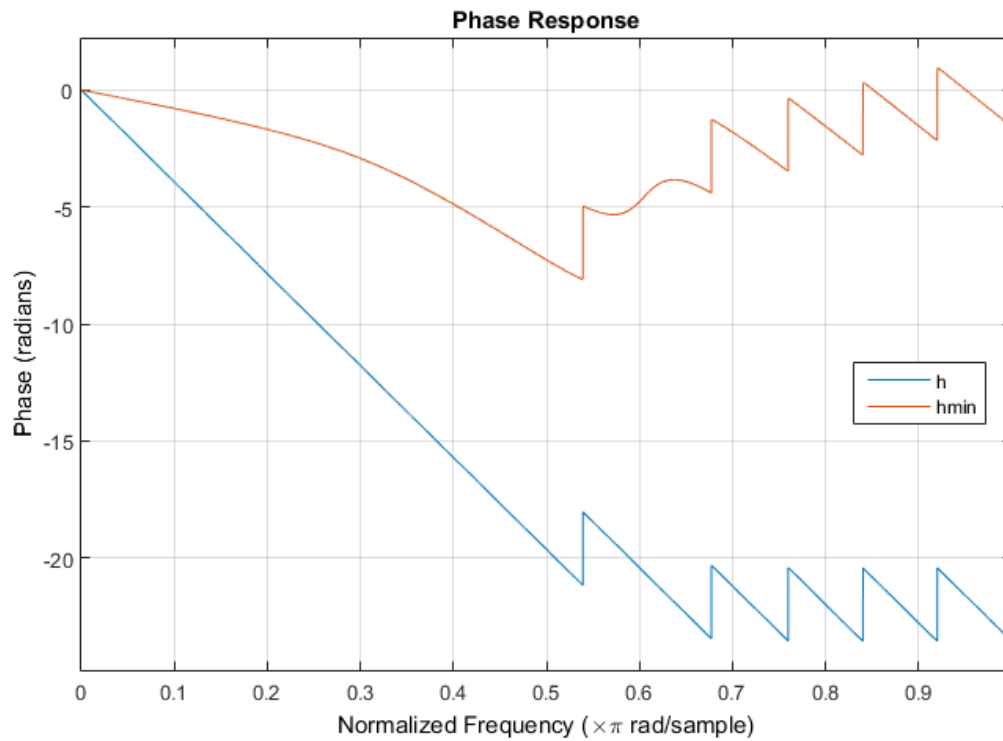
```
hfvt = fvtool(h,1,hmin,1,'Analysis','phase');  
legend(hfvt,'h','hmin')
```

```
hmin_linphase =
```

```
0
```

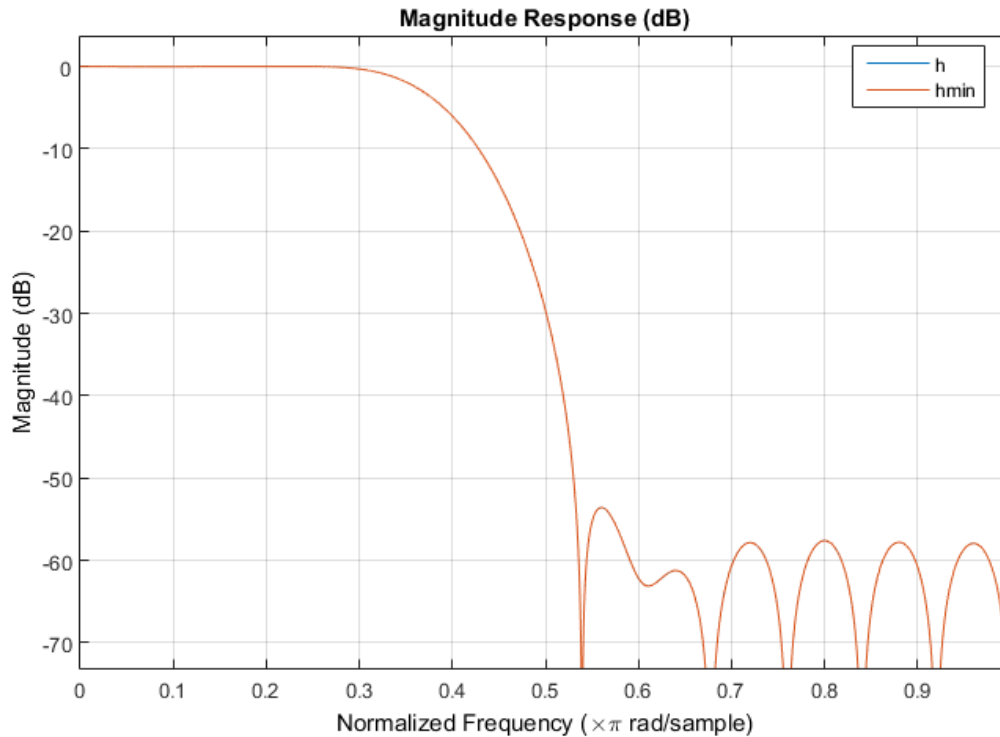
```
hmin_minphase =
```

```
1
```



Verify that the two filters have identical magnitude responses.

```
hfvt = fvtool(h,1,hmin,1);  
legend(hfvt,'h','hmin')
```



## More About

### Algorithms

`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

Convert power to decibels

## Syntax

```
ydb = pow2db(y)
```

## Description

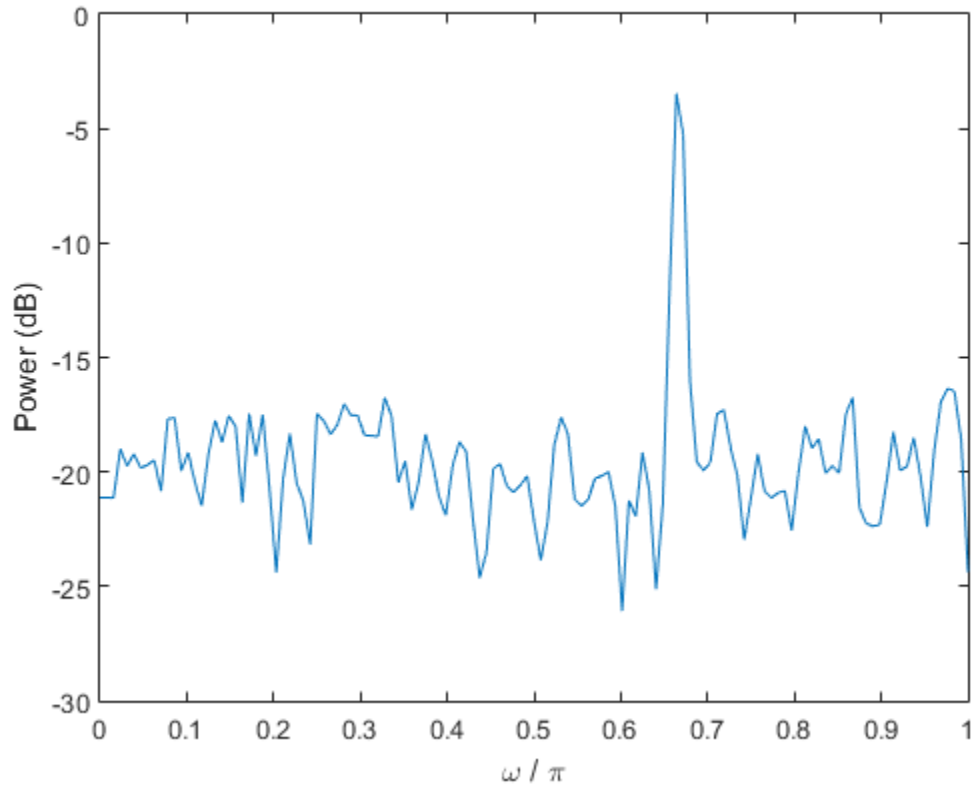
`ydb = pow2db(y)` expresses in decibels (dB) the power measurements specified in `y`. The relationship between power and decibels is  $ydb = 10 \log_{10}(y)$ .

## Examples

### Power Spectrum of a Noisy Sinusoid

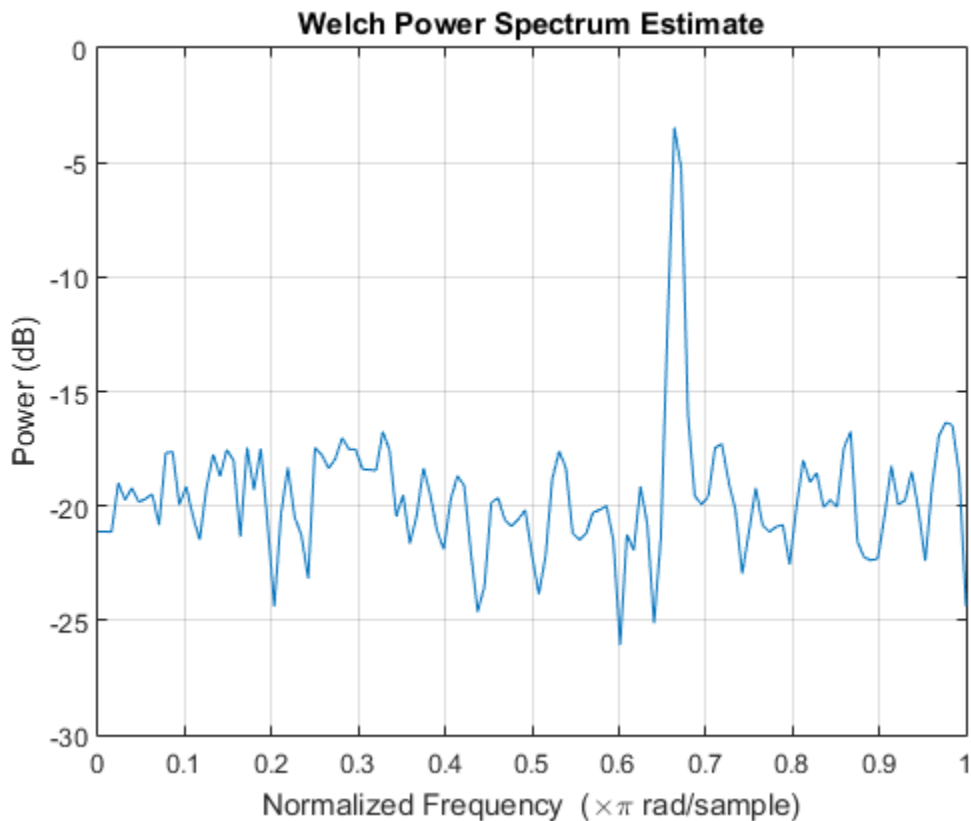
Generate 1024 samples of a noisy sinusoid having a normalized frequency of  $2\pi/3$  rad/sample. Estimate the power spectrum of the signal using `pwelch`. Express the estimate in decibels and plot it.

```
n = 0:1024-1;  
x = cos(2*pi*n/3) + randn(size(n));  
  
[pxx,w] = pwelch(x, 'power');  
  
dB = pow2db(pxx);  
  
plot(w/pi, dB)  
xlabel('\omega / \pi')  
ylabel('Power (dB)')
```



Repeat the computation using `pwelch` without output arguments.

```
pwelch(x, 'power')
```



## Input Arguments

### **y** — Input array

scalar | vector | matrix | N-D array

Input array, specified as a scalar, vector, matrix, or N-D array. When  $y$  is nonscalar, `pow2db` is an element-wise operation.

Data Types: `single` | `double`

## Output Arguments

### **ydb** — Power measurements in decibels

scalar | vector | matrix | N-D array

Power measurements in decibels, returned as a scalar, vector, matrix, or N-D array of the same size as `y`.

### **See Also**

`db` | `db2mag` | `db2pow` | `mag2db`

# powerbw

Power bandwidth

## Syntax

```
bw = powerbw(x)
```

```
bw = powerbw(x, fs)
```

```
bw = powerbw(pxx, f)
```

```
bw = powerbw(sxx, f, rbw)
```

```
bw = powerbw( ____, freqrange, r)
```

```
[bw, flo, fhi, power] = powerbw( ____, )
```

```
powerbw( ____, )
```

## Description

`bw = powerbw(x)` returns the 3-dB (half-power) bandwidth, `bw`, of the input signal, `x`.

`bw = powerbw(x, fs)` returns the 3-dB bandwidth in terms of the sample rate, `fs`.

`bw = powerbw(pxx, f)` returns the 3-dB bandwidth of the power spectral density (PSD) estimate, `pxx`. The frequencies, `f`, correspond to the estimates in `pxx`.

`bw = powerbw(sxx, f, rbw)` computes the 3-dB bandwidth of the power spectrum estimate, `sxx`. The frequencies, `f`, correspond to the estimates in `sxx`. `rbw` is the resolution bandwidth used to integrate each power estimate.

`bw = powerbw( ____, freqrange, r)` specifies the frequency interval over which to compute the reference level, using any of the input arguments from previous syntaxes. `freqrange` must lie within the target band.

If you also specify `r`, the function computes the difference in frequency between the points where the spectrum drops below the reference level by `r` dB or reaches an endpoint.

`[bw, flo, fhi, power] = powerbw( ___ )` also returns the lower and upper bounds of the power bandwidth and the power within those bounds.

`powerbw( ___ )` with no output arguments plots the PSD or power spectrum in the current figure window and annotates the bandwidth.

## Examples

### 3-dB Bandwidth of Chirps

Generate 1024 samples of a chirp sampled at 1024 kHz. The chirp has an initial frequency of 50 kHz and reaches 100 kHz at the end of the sampling. Add white Gaussian noise such that the signal-to-noise ratio is 40 dB. Reset the random number generator for reproducible results.

```
nSamp = 1024;
Fs = 1024e3;
SNR = 40;
rng default

t = (0:nSamp-1)'/Fs;

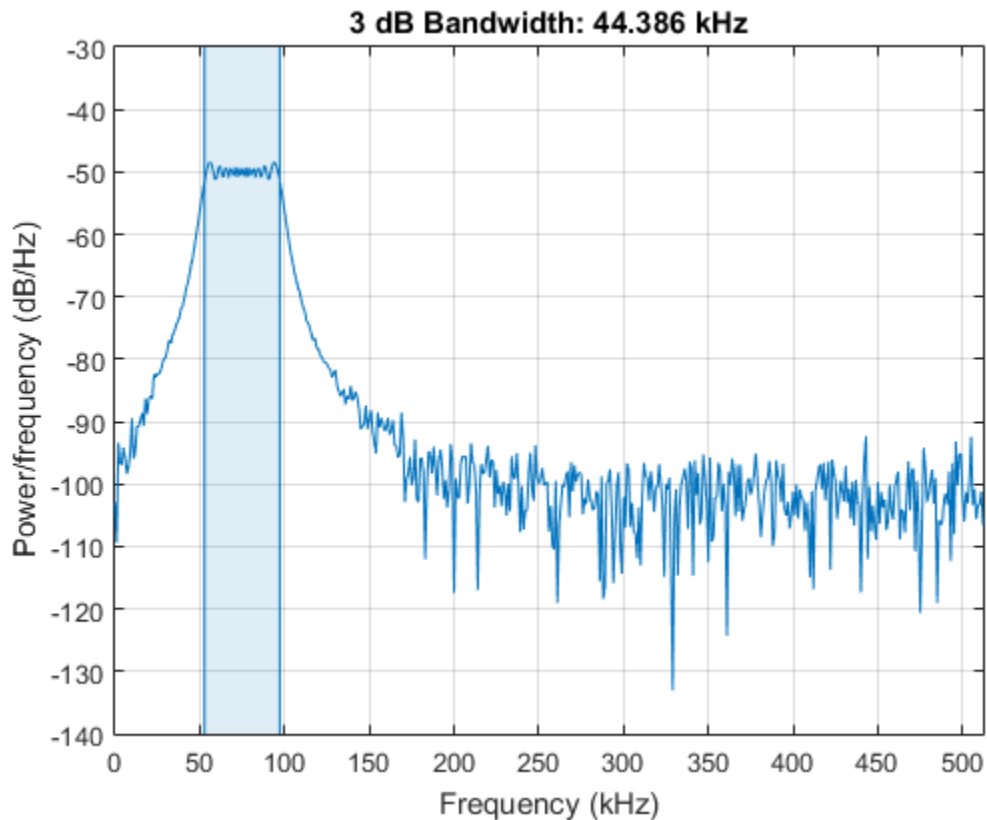
x = chirp(t,50e3,nSamp/Fs,100e3);
x = x+randn(size(x))*std(x)/db2mag(SNR);
```

Estimate the 3-dB bandwidth of the signal and annotate it on a plot of the power spectral density (PSD).

```
powerbw(x,Fs)
```

```
ans =

    4.4386e+04
```



Generate another chirp. Specify an initial frequency of 200 kHz, a final frequency of 300 kHz, and an amplitude that is twice that of the first signal. Add white Gaussian noise.

```
x2 = 2*chirp(t,200e3,nSamp/Fs,300e3);
x2 = x2+randn(size(x2))*std(x2)/db2mag(SNR);
```

Concatenate the chirps to produce a two-channel signal. Estimate the 3-dB bandwidth of each channel.

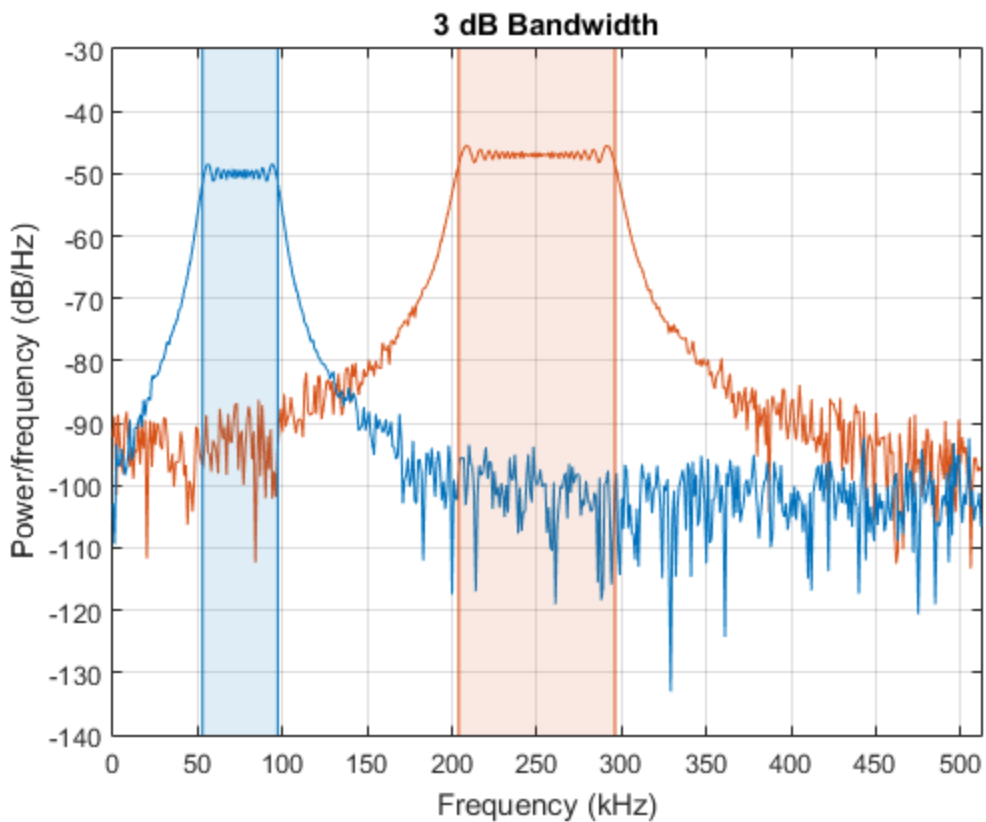
```
y = powerbw([x x2],Fs)
```

```
y =
```

```
1.0e+04 *
4.4386 9.2208
```

Annotate the 3-dB bandwidths of the two channels on a plot of the PSDs.

```
powerbw([x x2],Fs);
```



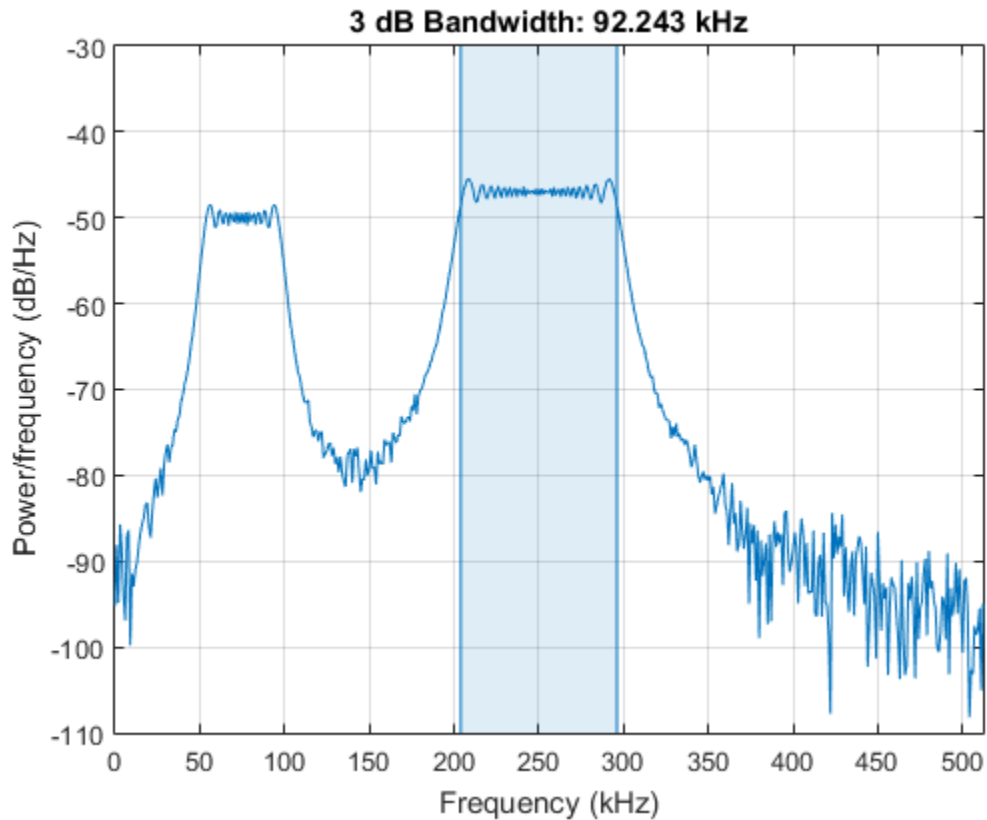
Add the two channels to form a new signal. Plot the PSD and annotate the 3-dB bandwidth.

```
powerbw(x+x2,Fs)
```



ans =

9.2243e+04



### 3-dB Bandwidth of Sinusoids

Generate 1024 samples of a 100.123 kHz sinusoid sampled at 1024 kHz. Add white Gaussian noise such that the signal-to-noise ratio is 40 dB. Reset the random number generator for reproducible results.

```
nSamp = 1024;
Fs = 1024e3;
SNR = 40;
```

```
rng default
```

```
t = (0:nSamp-1)'/Fs;
```

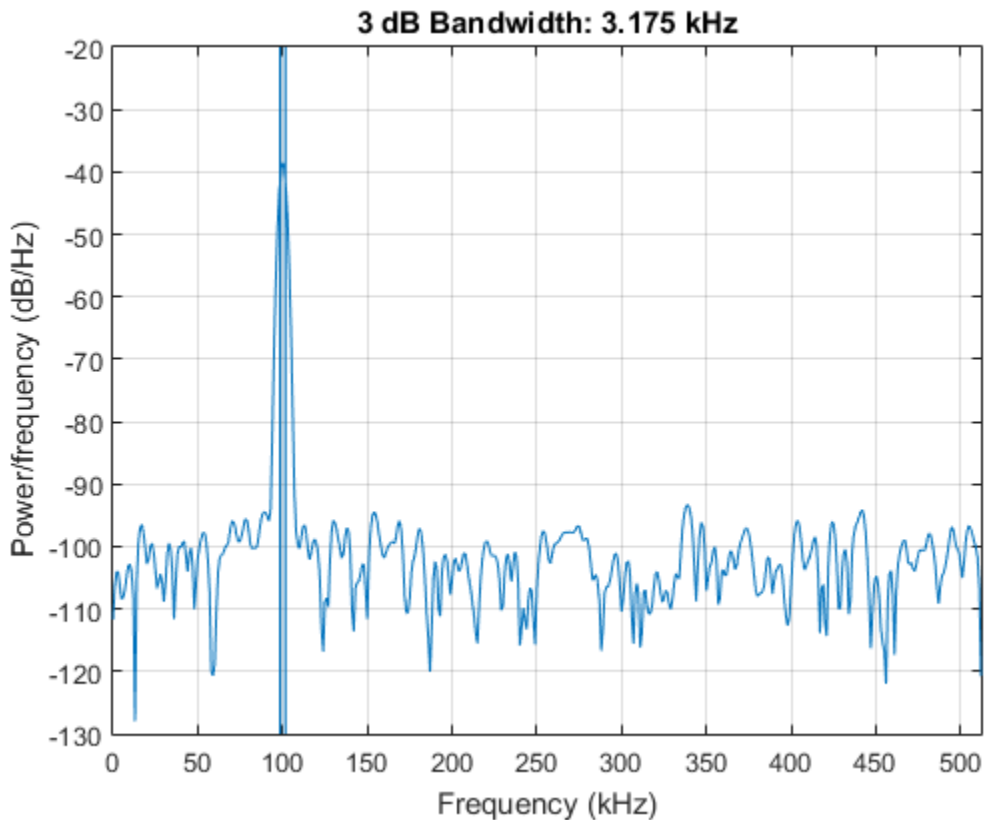
```
x = sin(2*pi*t*100.123e3);
```

```
x = x + randn(size(x))*std(x)/db2mag(SNR);
```

Use the `periodogram` function to compute the power spectral density (PSD) of the signal. Specify a Kaiser window with the same length as the signal and a shape factor of 38. Estimate the 3-dB bandwidth of the signal and annotate it on a plot of the PSD.

```
[Pxx,f] = periodogram(x,kaiser(nSamp,38),[],Fs);
```

```
powerbw(Pxx,f);
```



Generate another sinusoid, this one with a frequency of 257.321 kHz and an amplitude that is twice that of the first sinusoid. Add white Gaussian noise.

```
x2 = 2*sin(2*pi*t*257.321e3);  
x2 = x2 + randn(size(x2))*std(x2)/db2mag(SNR);
```

Concatenate the sinusoids to produce a two-channel signal. Estimate the PSD of each channel and use the result to determine the 3-dB bandwidth.

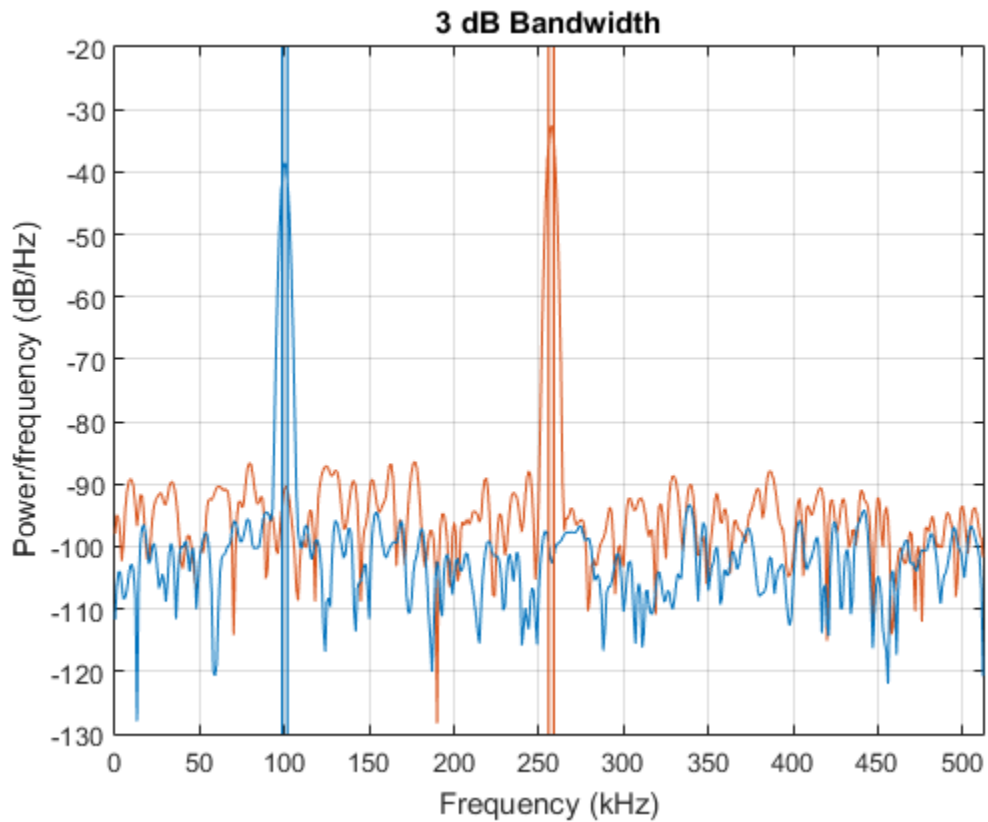
```
[Pyy,f] = periodogram([x x2],kaiser(nSamp,38),[],Fs);  
y = powerbw(Pyy,f)
```

```
y =
```

```
1.0e+03 *  
3.1753    3.3015
```

Annotate the 3-dB bandwidths of the two channels on a plot of the PSDs.

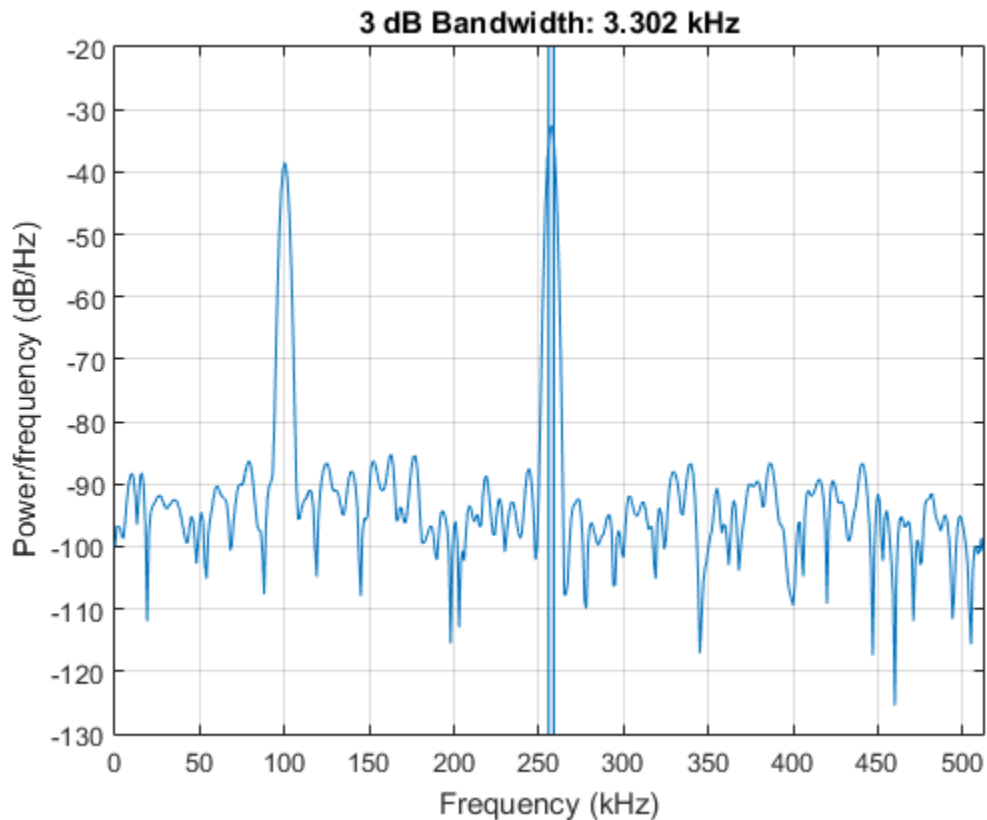
```
powerbw(Pyy,f);
```



Add the two channels to form a new signal. Estimate the PSD and annotate the 3-dB bandwidth.

```
[Pzz,f] = periodogram(x+x2,kaiser(nSamp,38),[],Fs);
```

```
powerbw(Pzz,f);
```



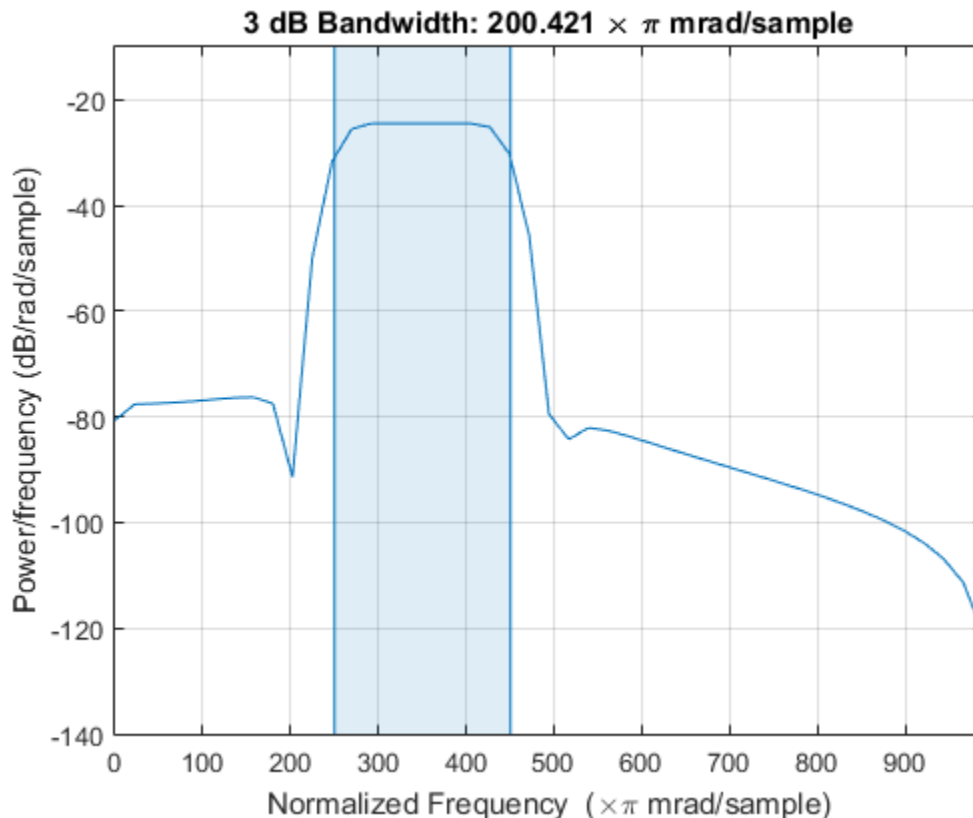
### Bandwidth of Bandlimited Signals

Generate a signal whose PSD resembles the frequency response of an 88th-order bandpass FIR filter with normalized cutoff frequencies  $0.25\pi$  rad/sample and  $0.45\pi$  rad/sample.

```
d = fir1(88,[0.25 0.45]);
```

Compute the 3-dB occupied bandwidth of the signal. Specify as a reference level the average power in the band between  $0.2\pi$  rad/sample and  $0.6\pi$  rad/sample. Plot the PSD and annotate the bandwidth.

```
powerbw(d,[],[0.2 0.6]*pi,3);
```



Output the bandwidth, its lower and upper bounds, and the band power. Specifying a sample rate of  $2\pi$  is equivalent to leaving the rate unset.

```
[bw,flo,fhi,power] = powerbw(d,2*pi,[0.2 0.6]*pi);

fprintf('bw = %.3f*pi, flo = %.3f*pi, fhi = %.3f*pi \n', ...
        [bw flo fhi]/pi)
fprintf('power = %.1f%% of total',power/bandpower(d)*100)

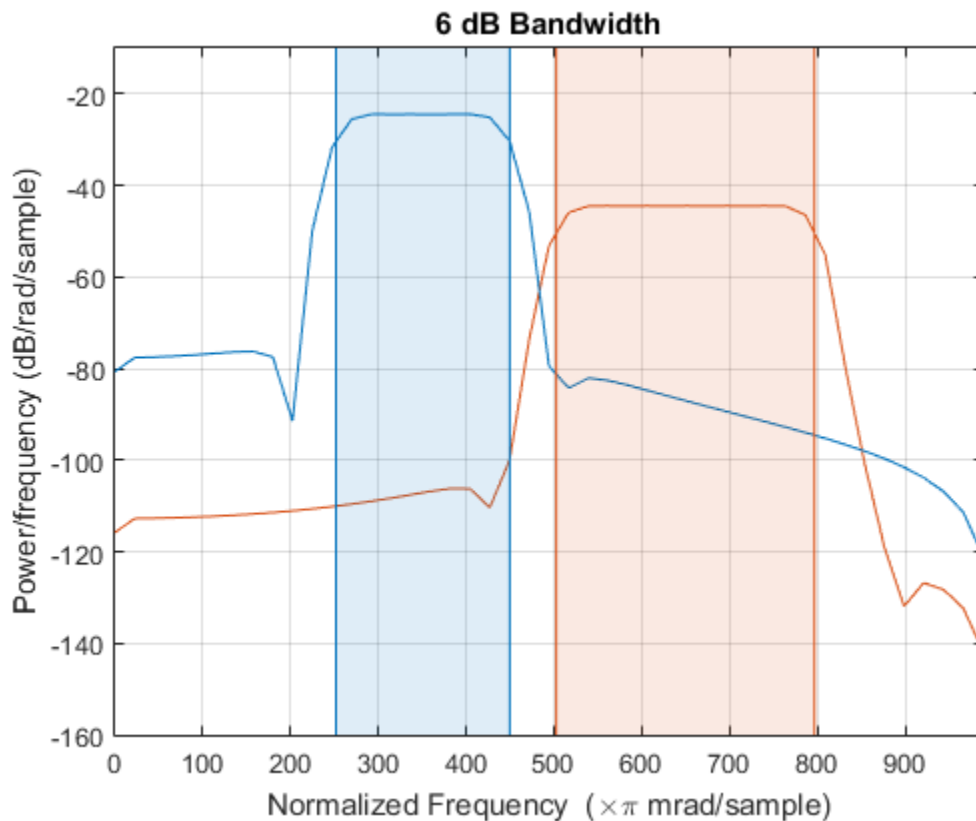
bw = 0.200*pi, flo = 0.250*pi, fhi = 0.450*pi
power = 96.9% of total
```

Add a second channel with normalized cutoff frequencies  $0.5\pi$  rad/sample and  $0.8\pi$  rad/sample and an amplitude that is one-tenth that of the first channel.

```
d = [d;fir1(88,[0.5 0.8])/10]';
```

Compute the 6-dB bandwidth of the two-channel signal. Specify as a reference level the maximum power level of the spectrum.

```
powerbw(d,[],[],6);
```



Output the 6-dB bandwidth of each channel and the lower and upper bounds.

```
[bw,flo,fhi] = powerbw(d,[],[],6);
bds = [bw;flo;fhi];
```

```
fprintf('One: bw = %.3f*pi, flo = %.3f*pi, fhi = %.3f*pi \n',bds(:,1)/pi)
fprintf('Two: bw = %.3f*pi, flo = %.3f*pi, fhi = %.3f*pi \n',bds(:,2)/pi)
```

One: `bw = 0.198*pi, flo = 0.252*pi, fhi = 0.450*pi`  
Two: `bw = 0.294*pi, flo = 0.503*pi, fhi = 0.797*pi`

## Input Arguments

### **x** — Input signal

vector | matrix

Input signal, specified as a vector or matrix. If `x` is a vector, it is treated as a single channel. If `x` is a matrix, then `powerbw` computes the power bandwidth independently for each column. `x` must be finite-valued.

Example: `cos(pi/4*(0:159))+randn(1,160)` is a single-channel row-vector signal.

Example: `cos(pi./[4;2]*(0:159))'+randn(160,2)` is a two-channel signal.

Data Types: `single` | `double`

### **fs** — Sample rate

positive real scalar

Sample rate, specified as a positive real scalar. The sample rate is the number of samples per unit time. If the time is measured in seconds, then the sample rate is in hertz.

Data Types: `single` | `double`

### **pxx** — Power spectral density

vector | matrix

Power spectral density (PSD) estimate, specified as a vector or matrix. If `pxx` is a one-sided estimate, then it must correspond to a real signal. If `pxx` is a matrix, then `powerbw` computes the bandwidth of each column of `pxx` independently.

Example: `periodogram(cos(pi./[4;2]*(0:159))'+randn(160,2))` is the periodogram PSD estimate of a two-channel sinusoid embedded in white Gaussian noise.

Data Types: `single` | `double`

### **f** — Frequencies

vector

Frequencies, specified as a vector. If the first element of `f` is 0, then `powerbw` assumes that the spectrum is a one-sided spectrum of a real signal. In other words, the function doubles the power value in the zero-frequency bin as it seeks the 3-dB point.



Data Types: `single` | `double`

### **sxx — Power spectrum estimate**

vector | matrix

Power spectrum estimate, specified as a vector or matrix. If `sxx` is a matrix, then `obw` computes the bandwidth of each column of `sxx` independently.

Example: `periodogram(cos(pi./[4;2]*(0:159))'+randn(160,2),'power')` is the periodogram power spectrum estimate of a two-channel sinusoid embedded in white Gaussian noise.

Data Types: `single` | `double`

### **rbw — Resolution bandwidth**

positive scalar

Resolution bandwidth, specified as a positive scalar. The resolution bandwidth is the product of two values: the frequency resolution of the discrete Fourier transform and the equivalent noise bandwidth of the window used to compute the PSD.

Data Types: `single` | `double`

### **freqrange — Frequency range**

two-element vector

Frequency range, specified as a two-element vector of real values. If you specify `freqrange`, then the reference level is the average power level in the reference band. If you do not specify `freqrange`, then the reference level is the maximum power level of the spectrum.

Data Types: `single` | `double`

### **r — Power level drop**

$10 \log_{10}2$  (default) | positive real scalar

Power level drop, specified as a positive real scalar expressed in dB.

Data Types: `single` | `double`

## **Output Arguments**

### **bw — Power bandwidth**

scalar | vector

Power bandwidth, returned as a scalar or vector.

- If you specify a sample rate, then `bw` has the same units as `fs`.
- If you do not specify a sample rate, then `bw` has units of rad/sample.

### **f1o, f1i — Bandwidth frequency bounds**

scalars | vectors

Bandwidth frequency bounds, returned as scalars.

### **power — Power stored in bandwidth**

scalar | vector

Power stored in bandwidth, returned as a scalar or vector.

## **More About**

### **Algorithms**

To determine the 3-dB bandwidth, `powerbw` computes a periodogram power spectrum estimate using a rectangular window and takes the maximum of the estimate as a reference level. The bandwidth is the difference in frequency between the points where the spectrum drops at least 3 dB below the reference level. If the signal reaches one of its endpoints before dropping by 3 dB, then `powerbw` uses the endpoint to compute the difference.

### **See Also**

`bandpower` | `obw` | `periodogram` | `plomb` | `pwelch`

**Introduced in R2015a**

## prony

Prony method for filter design

## Syntax

[Num,Den] = prony(impulse\_resp,num\_ord,denom\_ord)

## Description

[Num,Den] = prony(impulse\_resp,num\_ord,denom\_ord) returns the numerator Num and denominator Den coefficients for a causal rational system function with impulse response impulse\_resp. The system function has numerator order num\_ord and denominator order denom\_ord. The lengths of Num and Den are num\_ord+1 and denom\_ord+1. If the length of impulse\_resp is less than the largest order (num\_ord or denom\_ord), impulse\_resp is padded with zeros. Enter 0 in num\_ord for an all-pole system function. For an all-zero system function, enter a 0 for denom\_ord.

## Definitions

The *system function* is the z-transform of the impulse response  $h[n]$ :

$$H(z) = \sum_{n=-\infty}^{\infty} h[n]z^{-n}$$

A *rational system function* is a ratio of polynomials in  $z^{-1}$ . By convention the numerator polynomial is  $B(z)$  and the denominator is  $A(z)$ . The following equation describes a causal rational system function of numerator order num\_ord  $q$  and denominator order denom\_ord  $p$ :

$$H(z) = \frac{\sum_{k=0}^q b[k]z^{-k}}{1 + \sum_{l=1}^p a[l]z^{-l}}$$

where  $a[0]=1$ .

## Examples

### Filter Responses via Prony's Method

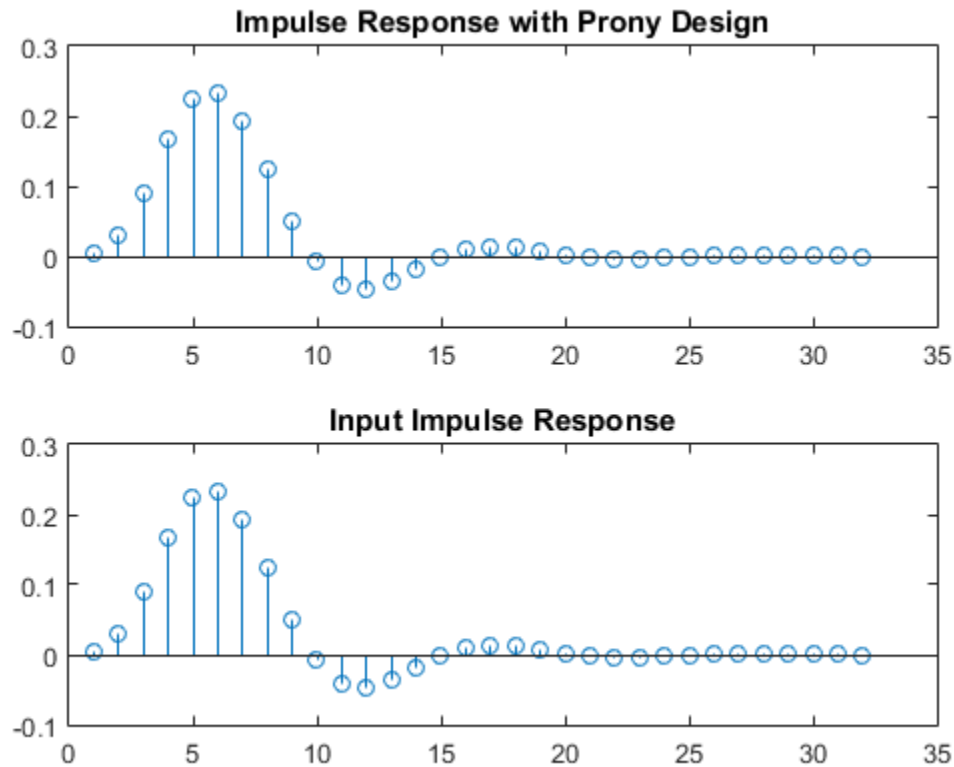
Fit a 4th-order IIR model to the impulse response of a lowpass filter. Plot the original and Prony-designed impulse responses.

```
d = designfilt('lowpassiir','NumeratorOrder',4,'DenominatorOrder',4, ...
              'HalfPowerFrequency',0.2,'DesignMethod','butter');

impulse_resp = filter(d,[1 zeros(1,31)]);
denom_order = 4;
num_order = 4;
[Num,Den] = prony(impulse_resp,num_order,denom_order);

subplot(2,1,1)
stem(impz(Num,Den,length(impulse_resp)))
title 'Impulse Response with Prony Design'

subplot(2,1,2)
stem(impulse_resp)
title 'Input Impulse Response'
```

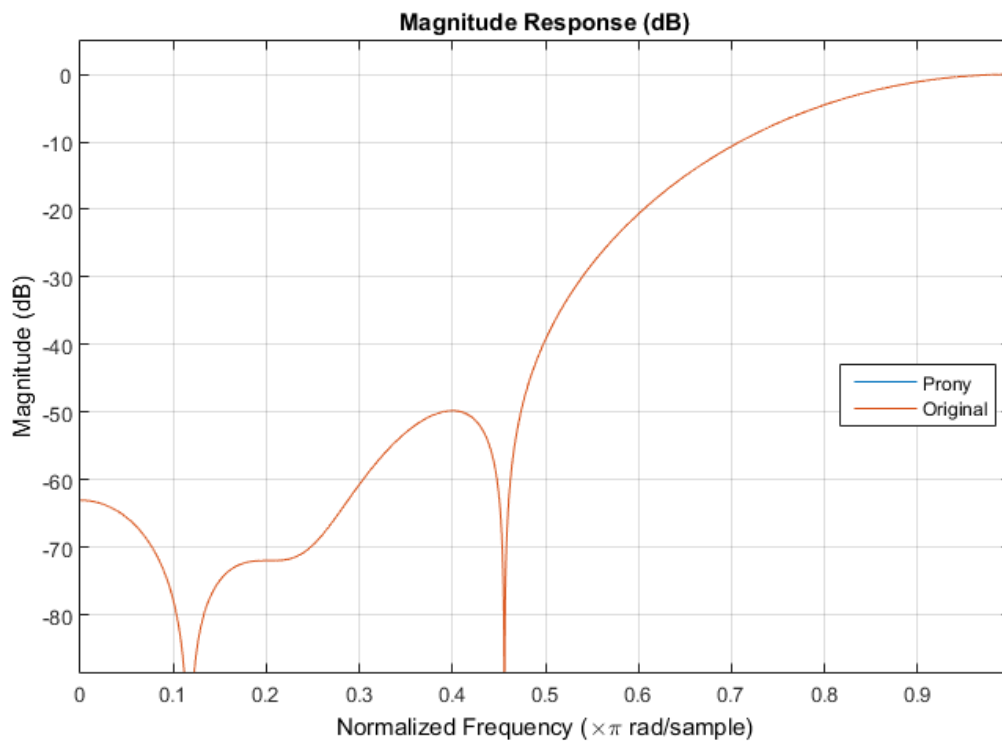


Fit a 10th-order FIR model to the impulse response of a highpass filter. Plot the original and Prony-designed frequency responses.

```
d = designfilt('highpassfir', 'FilterOrder', 10, 'CutoffFrequency', .8);

impulse_resp = filter(d,[1 zeros(1,31)]);
num_order = 10;
denom_order = 0;
[Num,Den] = prony(impulse_resp,num_order,denom_order);

fvt = fvtool(Num,Den,d);
legend(fvt,'Prony','Original')
```



## More About

- “Parametric Modeling”

## References

Parks, Thomas W., and C. Sidney Burrus. *Digital Filter Design*. New York: John Wiley & Sons, 1987, pp 226–228.

## See Also

`designfilt` | `lpc` | `impz` | `levinson`

# pulseperiod

Period of bilevel pulse

## Syntax

```
P = pulseperiod(X)
P = pulseperiod(X,FS)
P = pulseperiod(X,T)
[P,INITCROSS] = pulseperiod(...)
[P,INITCROSS,FINALCROSS] = pulseperiod(...)
[P,INITCROSS,FINALCROSS,NEXTCROSS] = pulseperiod(...)
[P,INITCROSS,FINALCROSS,NEXTCROSS,MIDLEV] = pulseperiod(...)
[P,INITCROSS,FINALCROSS,NEXTCROSS,MIDLEV] = pulseperiod(...,
Name,Value)
pulseperiod(...)
```

## Description

`P = pulseperiod(X)` returns a vector, `P`, containing the difference between the mid-reference level instants of the initial transition of each positive-polarity pulse and the next positive-going transition in the bilevel waveform, `X`. If `pulseperiod` does not find two positive-polarity transitions, `P` is empty. To determine the transitions for each pulse, `pulseperiod` estimates the state levels of the input waveform by a histogram method and identifies all regions which cross the upper-state boundary of the low state and the lower-state boundary of the high state. The low-state and high-state boundaries are expressed as the state level plus or minus a multiple of the difference between the state levels. See “State-Level Tolerances” on page 1-1182. Because `pulseperiod` uses interpolation to determine the mid-reference level instants, `P` may contain values that do not correspond to sampling instants of the bilevel waveform, `X`.

`P = pulseperiod(X,FS)` specifies the sampling rate in hertz as a positive scalar. The first sample instant in `X` corresponds to `t=0`. Because `pulseperiod` uses interpolation to determine the mid-reference level instants, `P` may contain values that do not correspond to sampling instants of the bilevel waveform, `X`.

`P = pulseperiod(X,T)` specifies the sampling instants in a vector equal in length to `X`. Because `pulseperiod` uses interpolation to determine the mid-reference level

instants, *P* may contain values that do not correspond to sampling instants of the bilevel waveform, *X*.

[*P*, *INITCROSS*] = `pulseperiod(...)` returns the mid-reference level instants of the first transition of each pulse.

[*P*, *INITCROSS*, *FINALCROSS*] = `pulseperiod(...)` returns the mid-reference level instants of the final transition of each pulse.

[*P*, *INITCROSS*, *FINALCROSS*, *NEXTCROSS*] = `pulseperiod(...)` returns the mid-reference level instants of next detected transition after each pulse.

[*P*, *INITCROSS*, *FINALCROSS*, *NEXTCROSS*, *MIDLEV*] = `pulseperiod(...)` returns the mid-reference level, *MIDLEV*.

[*P*, *INITCROSS*, *FINALCROSS*, *NEXTCROSS*, *MIDLEV*] = `pulseperiod(..., Name, Value)` returns the pulse periods with additional options specified by one or more *Name, Value* pair arguments.

`pulseperiod(...)` plots the signal and darkens every other identified pulse. It marks the location of the mid crossings, and their associated reference level. The state levels and their associated lower and upper boundaries (adjustable by the *Name, Value* pair with name 'Tolerance') are also plotted.

## Input Arguments

### **X**

Bilevel waveform. If the waveform, *X*, does not contain at least two transitions, `pulseperiod` outputs an empty matrix.

### **FS**

Sample rate in hertz.

### **T**

Vector of sample instants. The length of *T* must equal the length of the bilevel waveform, *X*.



## Name-Value Pair Arguments

### 'MidPercentReferenceLevel'

Mid-reference level as a percentage of the waveform amplitude.

**Default:** 50

### 'Polarity'

Pulse polarity. Specify the polarity as 'positive' or 'negative'. If you specify 'positive', pulseperiod looks for pulses whose initial transition is positive-going (positive polarity). If you specify 'negative', pulseperiod looks for pulses whose initial transition is negative-going (negative polarity).

**Default:** 'positive'

### 'StateLevels'

Low- and high-state levels. StateLevels is a 1-by-2 real-valued vector. The first element is the low-state level. The second element is the high-state level. If you do not specify low and high-state levels, pulseperiod estimates the state levels from the input waveform using the histogram method.

### 'Tolerance'

Tolerance levels (lower and upper state boundaries) expressed as a percentage. See “State-Level Tolerances” on page 1-1182.

**Default:** 2

## Output Arguments

### P

Pulse period in seconds. The pulse period is defined as the time between the mid-reference level instants of two consecutive transitions.

### INITCROSS

Mid-reference level instant of initial transition.

**FINALCROSS**

Mid-reference level instant of final transition.

**NEXTCROSS**

Mid-reference level instant of the first pulse transition after the final transition of the preceding pulse.

**MIDLEV**

Waveform value that corresponds to the mid-reference level.

## Examples

**Pulse Period of Bilevel Waveform**

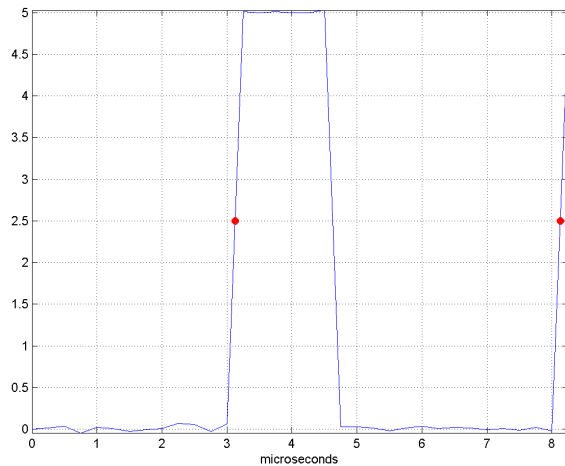
Compute the pulse period of a bilevel waveform with two positive-polarity transitions. The sampling rate is 4 MHz.

```
load('pulseex.mat', 'x', 't');  
p = pulseperiod(x, t);
```

**Determine Mid-Reference Level Instants of Pulse Period**

Determine the mid-reference level instants, which define the pulse period for a bilevel waveform. Mark the mid-reference level instants on a plot of the data.

```
load('pulseex.mat', 'x', 't');  
[p,initcross,~,nextcross,midlev] = pulseperiod(x,t);  
fprintf('Pulse period is %2.3f microseconds \n',p*1e6);  
plot(t.*1e6,x); hold on;  
grid on; axis tight; xlabel('microseconds');  
plot(initcross.*1e6,midlev,'ro','markerfacecolor',[1 0 0]);  
plot(nextcross.*1e6,midlev,'ro','markerfacecolor',[1 0 0]);
```



## More About

### Mid-Reference Level

The mid-reference level in a bilevel waveform with low-state level,  $S_1$ , and high-state level,  $S_2$ , is

$$S_1 + \frac{1}{2}(S_2 - S_1)$$

### Mid-Reference Level Instant

Let  $y_{50\%}$  denote the mid-reference level.

Let  $t_{50\%_-}$  and  $t_{50\%_+}$  denote the two consecutive sampling instants corresponding to the waveform values nearest in value to  $y_{50\%}$ .

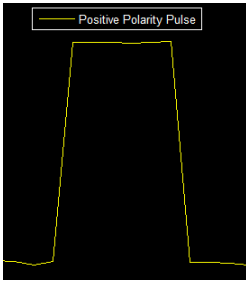
Let  $y_{50\%_-}$  and  $y_{50\%_+}$  denote the waveform values at  $t_{50\%_-}$  and  $t_{50\%_+}$ .

The mid-reference level instant is

$$t_{50\%} = t_{50\%_0} + \left( \frac{t_{50\%_+} - t_{50\%_-}}{y_{50\%_+} - y_{50\%_-}} \right) (y_{50\%_+} - y_{50\%_0})$$

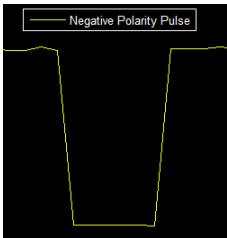
### Pulse Polarity

If the initial transition of a pulse is positive-going, the pulse has positive polarity. The following figure shows a positive-polarity pulse.



Equivalently, a positive-polarity (positive-going) pulse has a terminating state more positive than the originating state.

If the initial transition of a pulse is negative-going, the pulse has negative polarity. The following figure shows a negative-polarity pulse.



Equivalently, a negative-polarity (negative-going) pulse has an originating state more positive than the terminating state.

### State-Level Tolerances

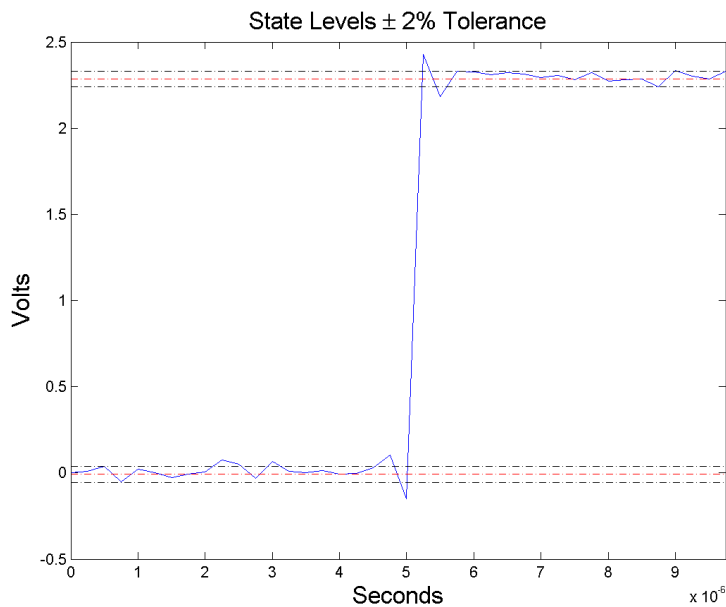
Each state level can have associated lower- and upper-state boundaries. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference

between the high state and low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the  $\alpha\%$  tolerance region for the low state is defined as

$$S_1 \pm \frac{\alpha}{100}(S_2 - S_1)$$

where  $S_1$  is the low-state level and  $S_2$  is the high-state level. Replace the first term in the equation with  $S_2$  to obtain the  $\alpha\%$  tolerance region for the high state.

The following figure illustrates lower and upper 2% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The red dashed lines indicate the estimated state levels.



## References

- [1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003.

**See Also**

dutycycle | pulsedsep | pulsewidth | statelevels

# pulsesep

Separation between bilevel waveform pulses

## Syntax

```
S = pulsesep(X)
S = pulsesep(X,FS)
S = pulsesep(X,T)
[S,INITCROSS] = pulsesep(...)
[S,INITCROSS,FINALCROSS] = pulsesep(...)
[S,INITCROSS,FINALCROSS,NEXTCROSS] = pulsesep(...)
[S,INITCROSS,FINALCROSS,NEXTCROSS,MIDLEV] = pulsesep(...)
[S,INITCROSS,FINALCROSS,NEXTCROSS,MIDLEV] = pulsesep(...,Name,Value)
pulsesep(...)
```

## Description

`S = pulsesep(X)` returns the differences, `S`, between the mid-reference level instants of the final negative-going transitions of every positive-polarity pulse and the next positive-going transition. `X` is a bilevel waveform. To determine the transitions that compose each pulse, `pulsesep` estimates the state levels of `X` by a histogram method. `pulsesep` identifies all regions that cross the upper-state boundary of the low state and the lower-state boundary of the high state. The low-state and high-state boundaries are expressed as the state level plus or minus a multiple of the difference between the state levels. See “State-Level Tolerances” on page 1-1190. Because `pulsesep` uses interpolation to determine the mid-reference level instants, `S` may contain values that do not correspond to sampling instants of the bilevel waveform, `X`.

`S = pulsesep(X,FS)` specifies the sampling rate, `FS`, in Hz as a positive scalar. The first time instant corresponds to `t=0`. Because `pulsesep` uses interpolation to determine the mid-reference level instants, `S` may contain values that do not correspond to sampling instants of the bilevel waveform, `X`.

`S = pulsesep(X,T)` specifies the sampling instants, `T`, in a vector equal in length to `X`. Because `pulsesep` uses interpolation to determine the mid-reference level instants, `S`

may contain values that do not correspond to sampling instants of the bilevel waveform, **X**.

`[S, INITCROSS] = pulsesep(...)` returns the mid-reference level instants, **INITCROSS**, of the first positive-polarity transitions.

`[S, INITCROSS, FINALCROSS] = pulsesep(...)` returns the mid-reference level instants, **FINALCROSS**, of the final transition of each pulse.

`[S, INITCROSS, FINALCROSS, NEXTCROSS] = pulsesep(...)` returns the mid-reference level instants, **NEXTCROSS**, of the next detected transition after each pulse.

`[S, INITCROSS, FINALCROSS, NEXTCROSS, MIDLEV] = pulsesep(...)` returns the mid-reference level, **MIDLEV**.

`[S, INITCROSS, FINALCROSS, NEXTCROSS, MIDLEV] = pulsesep(..., Name, Value)` returns the pulse separations with additional options specified by one or more **Name, Value** pair arguments.

`pulsesep(...)` plots the signal and darkens the regions between each pulse where pulse separation is computed. It marks the location of the mid crossings, and their associated reference level. The state levels and their associated lower and upper boundaries (adjustable by the **Name, Value** pair with name '**Tolerance**') are also plotted.

## Input Arguments

### **X**

Bilevel waveform. If the waveform, **X**, does not contain at least two transitions, `pulsesep` outputs an empty matrix.

### **FS**

Sample rate in hertz.

### **T**

Vector of sample instants. The length of **T** must equal the length of the bilevel waveform, **X**.



## Name-Value Pair Arguments

### 'MidPercentReferenceLevel'

Mid-reference level as a percentage of the waveform amplitude.

**Default:** 50

### 'Polarity'

Pulse polarity. Specify the polarity as 'positive' or 'negative'. If you specify 'positive', `pulsesep` looks for pulses with positive-going (positive polarity) initial transitions. If you specify 'negative', `pulsesep` looks for pulses with negative-going (negative polarity) initial transitions. See “Pulse Polarity” on page 1-1190.

**Default:** 'positive'

### 'StateLevels'

Low- and high-state levels. `StateLevels` is a 1-by-2 real-valued vector. The first element is the low-state level. The second element is the high-state level. If you do not specify low- and high-state levels, `pulsesep` estimates the state levels from the input waveform using the histogram method.

### 'Tolerance'

Tolerance levels (lower- and upper-state boundaries) expressed as a percentage. See “State-Level Tolerances” on page 1-1190.

**Default:** 2

## Output Arguments

### S

Pulse separations in seconds. The *pulse separation* is defined as the time between the mid-reference level instants of the final transition of one pulse and the initial transition of the next pulse. See “Pulse Separation” on page 1-1191.

### INITCROSS

Mid-reference level instants of initial transition.

**FINALCROSS**

Mid-reference level instants of final transition.

**NEXTCROSS**

Mid-reference level instants of the initial transition after the final transition of the preceding pulse.

**MIDLEV**

Waveform value that corresponds to the mid-reference level.

## Examples

**Pulse Separation in Bilevel Waveform**

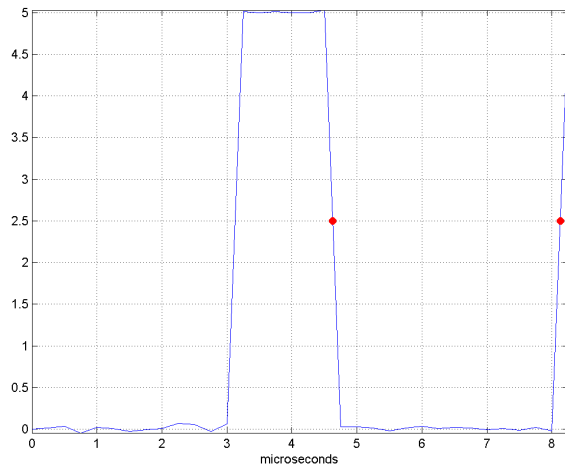
Compute the pulse separation in a bilevel waveform with two positive-polarity transitions. The sampling rate is 4 MHz.

```
load('pulseex.mat', 'x', 't');  
s = pulsesep(x, t);
```

**Determine Mid-Reference Level Instants Defining Pulse Separation**

Determine the mid-reference level instants, which define the pulse separation for a bilevel waveform. Mark the mid-reference level instants on a plot of the data.

```
load('pulseex.mat', 'x', 't');  
[s,~,finalcross,nextcross,midlev] = pulsesep(x,t);  
fprintf('Pulse separation is %2.3f microseconds \n',s*1e6);  
plot(t.*1e6,x); hold on;  
grid on; axis tight; xlabel('microseconds');  
plot(finalcross.*1e6,midlev,'ro','markerfacecolor',[1 0 0]);  
plot(nextcross.*1e6,midlev,'ro','markerfacecolor',[1 0 0]);
```



## More About

### Mid-Reference Level

The mid-reference level in a bilevel waveform with low-state level,  $S_1$ , and high-state level,  $S_2$ , is

$$S_1 + \frac{1}{2}(S_2 - S_1)$$

### Mid-Reference Level Instant

Let  $y_{50\%}$  denote the mid-reference level.

Let  $t_{50\%}$  and  $t_{50\%+}$  denote the two consecutive sampling instants corresponding to the waveform values nearest in value to  $y_{50\%}$ .

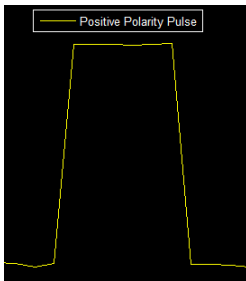
Let  $y_{50\%}$  and  $y_{50\%+}$  denote the waveform values at  $t_{50\%}$  and  $t_{50\%+}$ .

The mid-reference level instant is

$$t_{50\%} = t_{50\%_0} + \left( \frac{t_{50\%_+} - t_{50\%_-}}{y_{50\%_+} - y_{50\%_-}} \right) (y_{50\%_+} - y_{50\%_0})$$

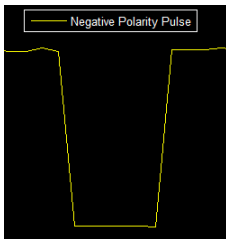
## Pulse Polarity

If the pulse has an initial positive-going transition, the pulse has positive polarity. The following figure shows a positive-polarity pulse.



Equivalently, a positive-polarity (positive-going) pulse has a terminating state more positive than the originating state.

If the pulse has an initial negative-going transition, the pulse has negative polarity. The following figure shows a negative-polarity pulse.



Equivalently, a negative-polarity (negative-going) pulse has an originating state more positive than the terminating state.

## State-Level Tolerances

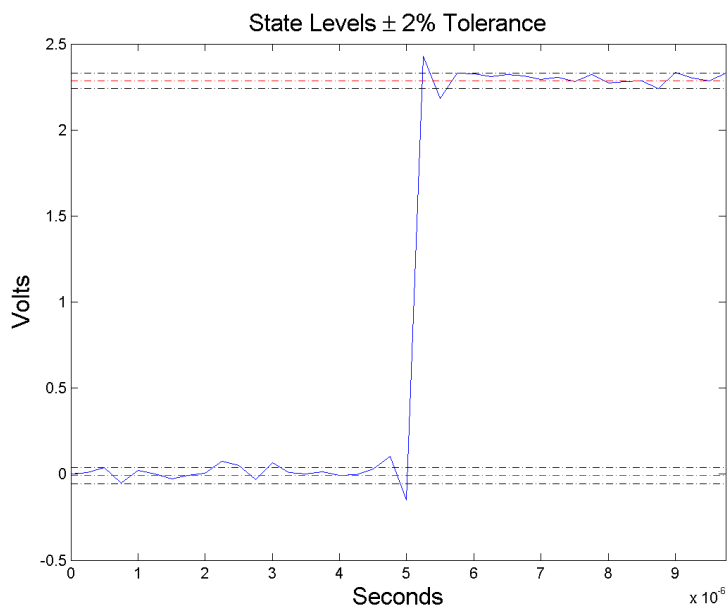
Each state level can have an associated lower- and upper-state boundary. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference

between the high state and low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the  $\alpha\%$  tolerance region for the low state is defined as

$$S_1 \pm \frac{\alpha}{100}(S_2 - S_1)$$

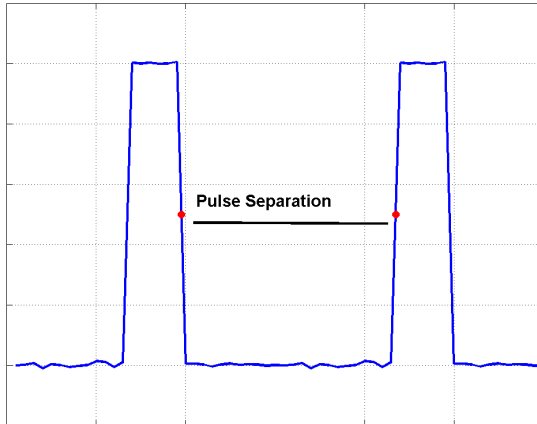
where  $S_1$  is the low-state level and  $S_2$  is the high-state level. Replace the first term in the equation with  $S_2$  to obtain the  $\alpha\%$  tolerance region for the high state.

The following figure illustrates lower and upper 2% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The red dashed lines indicate the estimated state levels.



### Pulse Separation

Pulse separation is the time difference between the mid-reference level instant of the final transition of one pulse and the mid-reference level instant of the initial transition of the next pulse. The following figure illustrates pulse separation.



## References

- [1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003.

## See Also

`dutycycle` | `pulseperiod` | `pulsewidth` | `statelevels`

# **pulsewidth**

Bilevel waveform pulse width

## **Syntax**

```
W = pulsewidth(X)
W = pulsewidth(X,FS)
W = pulsewidth(X,T)
[W,INITCROSS] = pulsewidth(...)
[W,INITCROSS,FINALCROSS] = pulsewidth(...)
[W,INITCROSS,FINALCROSS,MIDLEV] = pulsewidth(...)
W = pulsewidth(...,Name,Value)
pulsewidth(...)
```

## **Description**

`W = pulsewidth(X)` returns a vector, `W`, containing the time differences between the mid-reference level instants of the initial and final transitions of each positive-polarity pulse in the bilevel waveform, `X`. To determine the transitions, `pulsewidth` estimates the low- and high-state levels of `X` by a histogram method. `pulsewidth` identifies all regions that cross the upper-state boundary of the low state and the lower-state boundary of the high state. The low-state and high-state boundaries are expressed as the state level plus or minus a multiple of the difference between the state levels. See “State-Level Tolerances” on page 1-1197. Because `pulsewidth` uses interpolation to determine the mid-reference level instants, `W` may contain values that do not correspond to sampling instants of the bilevel waveform, `X`.

`W = pulsewidth(X,FS)` specifies the sample rate, `FS`, in hertz as a positive scalar. The first sample in the waveform corresponds to `t=0`. Because `pulsewidth` uses interpolation to determine the mid-reference level instants, `W` may contain values that do not correspond to sampling instants of the bilevel waveform, `X`.

`W = pulsewidth(X,T)` specifies the sample instants, `T`, as a vector with the same number of elements as `X`. Because `pulsewidth` uses interpolation to determine the mid-reference level instants, `W` may contain values that do not correspond to sampling instants of the bilevel waveform, `X`.

`[W,INITCROSS] = pulswidth(...)` returns a column vector, `INITCROSS`, whose elements correspond to the mid-reference level instants of the initial transition of each pulse.

`[W,INITCROSS,FINALCROSS] = pulswidth(...)` returns a column vector, `FINALCROSS`, whose elements correspond to the mid-reference level instants of the final transition of each pulse.

`[W,INITCROSS,FINALCROSS,MIDLEV] = pulswidth(...)` returns the waveform value, `MIDLEV`, which corresponds to the mid-reference level.

`W = pulswidth(...,Name,Value)` returns the pulse widths with additional options specified by one or more `Name,Value` pair arguments.

`pulswidth(...)` plots the signal and darkens the regions of each pulse where pulse width is computed. It marks the location of the mid crossings, and their associated reference level. The state levels and their associated lower and upper boundaries (adjustable by the `Name,Value` pair with name `'Tolerance'`) are also plotted.

## Input Arguments

### **X**

Bilevel waveform. `X` is a real-valued row or column vector.

### **FS**

Sample rate in hertz.

### **T**

Vector of sample instants. The length of `T` must equal the length of the bilevel waveform, `X`.

## Name-Value Pair Arguments

### **'MidPercentReferenceLevel'**

Mid-reference level as percentage of the waveform amplitude. See “Mid-Reference Level” on page 1-1196.

**Default:** 50



**'Polarity'**

Pulse polarity. Specify the polarity as 'positive' or 'negative'. If you specify 'positive', pulsewidth looks for pulses with positive-going (positive polarity) initial transitions. If you specify 'negative', pulsewidth looks for pulses with negative-going (negative polarity) initial transitions. See “Pulse Polarity” on page 1-1196.

**Default:** 'positive'

**'StateLevels'**

Low- and high-state levels. StateLevels is a 1-by-2 real-valued vector. The first element is the low-state level. The second element is the high-state level. If you do not specify low- and high-state levels, pulsewidth estimates the state levels from the input waveform using the histogram method.

**'Tolerance'**

Tolerance levels (lower and upper state boundaries) expressed as a percentage. See “State-Level Tolerances” on page 1-1197.

**Default:** 2

## Output Arguments

**W**

Pulse widths in seconds. The pulse width is the time difference between the initial and final transitions of a pulse. The times of the initial and final transitions are referred to as *transition occurrence instants* in [1].

**INITCROSS**

Mid-reference level instants of the initial transition

**FINALCROSS**

Mid-reference level instants of the final transition

**MIDLEV**

Waveform value corresponding to the mid-reference level

## Definitions

### Mid-Reference Level

The mid-reference level in a bilevel waveform with low-state level,  $S_1$ , and high-state level,  $S_2$ , is

$$S_1 + \frac{1}{2}(S_2 - S_1)$$

### Mid-Reference Level Instant

Let  $y_{50\%}$  denote the mid-reference level.

Let  $t_{50\%_-}$  and  $t_{50\%_+}$  denote the two consecutive sampling instants corresponding to the waveform values nearest in value to  $y_{50\%}$ .

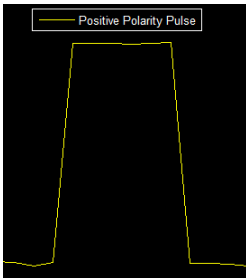
Let  $y_{50\%_-}$  and  $y_{50\%_+}$  denote the waveform values at  $t_{50\%_-}$  and  $t_{50\%_+}$ .

The mid-reference level instant is

$$t_{50\%} = t_{50\%_-} + \left( \frac{t_{50\%_+} - t_{50\%_-}}{y_{50\%_+} - y_{50\%_-}} \right) (y_{50\%_+} - y_{50\%_-})$$

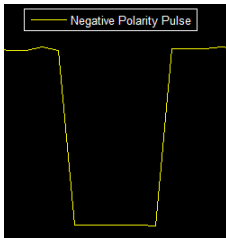
### Pulse Polarity

If the pulse has a positive-going initial transition, the pulse has positive polarity. The following figure shows a positive-polarity pulse.



Equivalently, a positive-polarity (positive-going) pulse has a terminating state more positive than the originating state.

If the pulse has a negative-going initial transition, the pulse has negative polarity. The following figure shows a negative-polarity pulse.



Equivalently, a negative-polarity (negative-going) pulse has a originating state more positive than the terminating state.

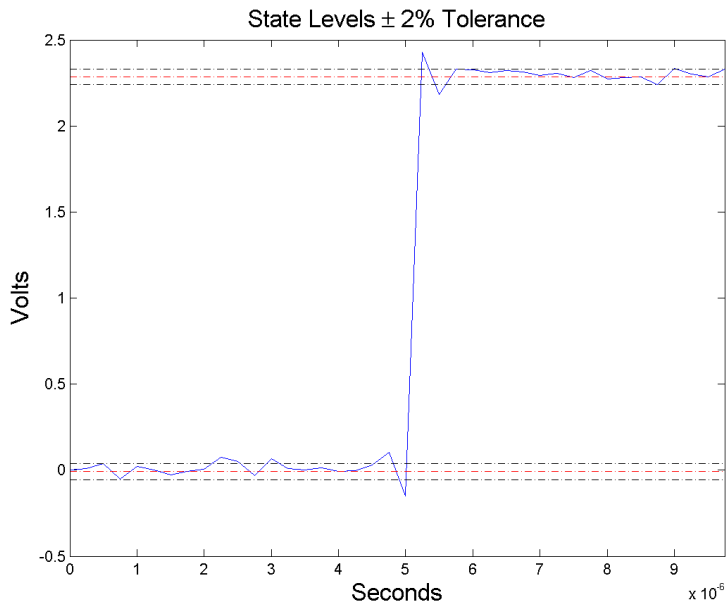
## State-Level Tolerances

Each state level can have an associated lower- and upper-state boundary. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference between the high state and low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the  $\alpha\%$  tolerance region for the low state is defined as

$$S_1 \pm \frac{\alpha}{100}(S_2 - S_1)$$

where  $S_1$  is the low-state level and  $S_2$  is the high-state level. Replace the first term in the equation with  $S_2$  to obtain the  $\alpha\%$  tolerance region for the high state.

The following figure illustrates lower and upper 2% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The red dashed lines indicate the estimated state levels.



## Examples

### Pulse Width of Bilevel Waveform

Compute the pulse width of a bilevel waveform sampled at 4 MHz.

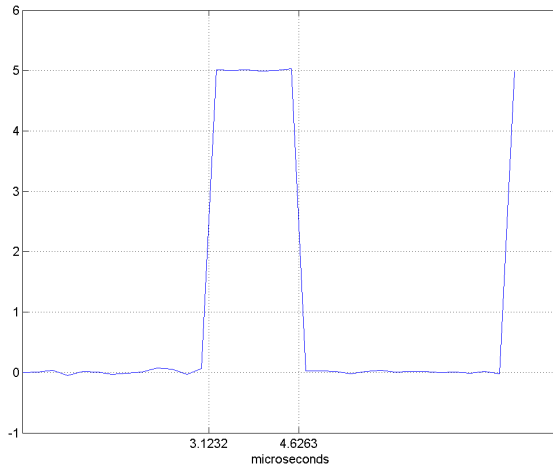
```
load('pulseex.mat', 'x', 't');
w = pulsewidth(x, t);
plot(t,x); grid on;
```

### First and Second Transition Times for Bilevel Waveform

Compute the initial and final transition occurrences for a bilevel waveform sampled at 4 MHz. Plot the result annotated with the transition occurrences.

```
load('pulseex.mat', 'x', 't');
fs = 4e6;
[w,initcross,finalcross] = pulsewidth(x,fs);
plot(t.*1e6,x);
```

```
set(gca,'xtick',[initcross*1e6 finalcross*1e6]);  
grid on;  
xlabel('microseconds');
```



### Specify State Levels for Bilevel Waveform

Specify the state levels for the bilevel waveform instead of estimating the levels from the data. Use the 'StateLevels' name-value pair to enter the low-state level as 0 and the high-state level as 5.

```
load('pulseex.mat', 'x', 't');  
[w,initcross,finalcross] = pulsewidth(x,fs,'StateLevels',[0 5]);
```

## References

- [1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003.

### See Also

dutycycle | pulseperiod | pulsesep | statelevels

# pulstran

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.

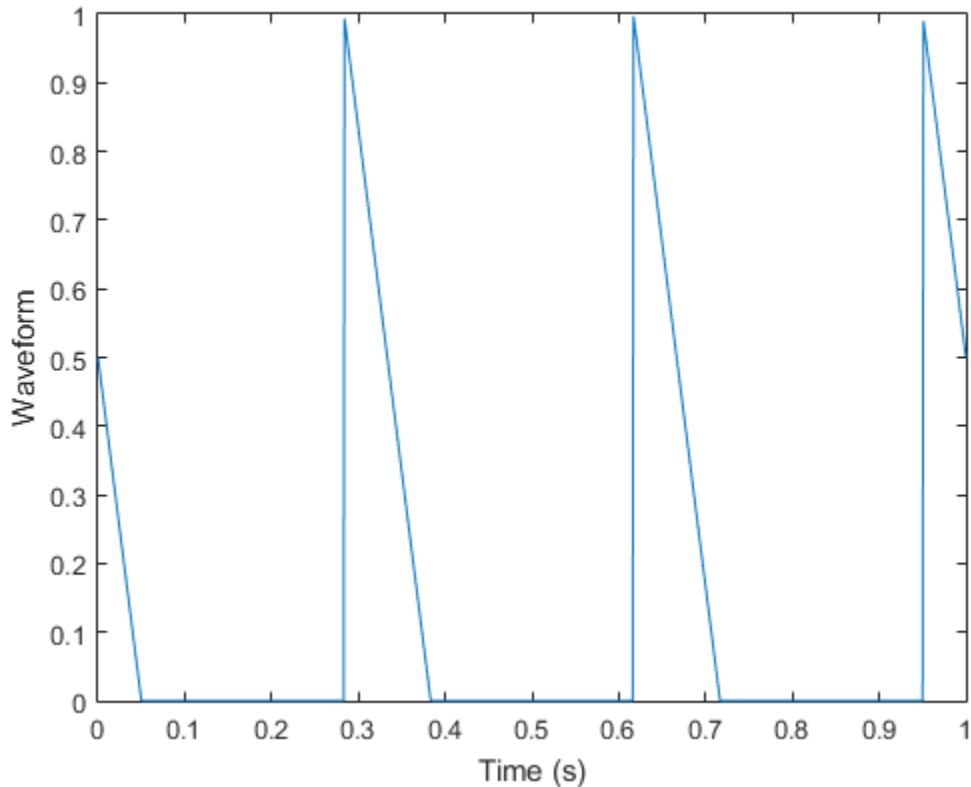
## Examples

### Asymmetric Sawtooth Waveform

This example generates an asymmetric sawtooth waveform with a repetition frequency of 3 Hz and a sawtooth width of 0.1 s. The signal length is 1 s and the sample rate is 1 kHz.

```
t = 0 : 1/1e3 : 1;          % 1 kHz sample freq for 1 s
d = 0 : 1/3 : 1;          % 3 Hz repetition frequency
y = pulstran(t,d,'tripuls',0.1,-1);

plot(t,y)
xlabel 'Time (s)', ylabel Waveform
```

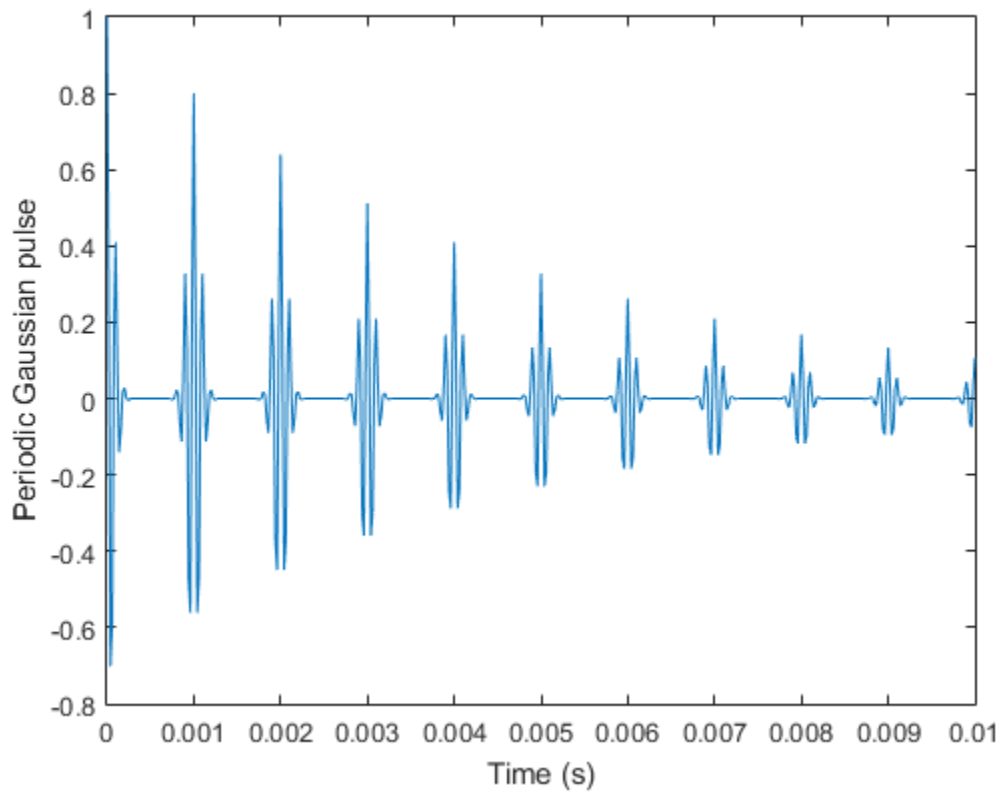


### Periodic Gaussian Pulse

This example generates a periodic Gaussian pulse signal at 10 kHz with 50% bandwidth. The pulse repetition frequency is 1 kHz, the sample rate is 50 kHz, and the pulse train length is 10 ms. 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)  
xlabel 'Time (s)', ylabel 'Periodic Gaussian pulse'
```



**See Also**

chirp | cos | diric | gauspuls | rectpuls | sawtooth | sin | sinc | square | tripuls

# pwelch

Welch's power spectral density estimate

## Syntax

```
pxx = pwelch(x)
pxx = pwelch(x,window)
pxx = pwelch(x,window,noverlap)
pxx = pwelch(x,window,noverlap,nfft)

[pxx,w] = pwelch( ___ )
[pxx,f] = pwelch( ___ ,fs)

[pxx,w] = pwelch(x,window,noverlap,w)
[pxx,f] = pwelch(x,window,noverlap,f,fs)

[ ___ ] = pwelch(x,window, ___ ,freqrange)
[ ___ ] = pwelch(x,window, ___ ,spectrumtype)
[ ___ ] = pwelch(x,window, ___ ,trace)

[ ___ ,pxxc] = pwelch( ___ ,'ConfidenceLevel',probability)

pwelch( ___ )
```

## Description

`pxx = pwelch(x)` returns the power spectral density (PSD) estimate, `pxx`, of the input signal, `x`, found using Welch's overlapped segment averaging estimator. When `x` is a vector, it is treated as a single channel. When `x` is a matrix, the PSD is computed independently for each column and stored in the corresponding column of `pxx`. If `x` is real-valued, `pxx` is a one-sided PSD estimate. If `x` is complex-valued, `pxx` is a two-sided PSD estimate. By default, `x` is divided into the longest possible sections to obtain as close to but not exceed 8 segments with 50% overlap. Each section is windowed with a Hamming window. The modified periodograms are averaged to obtain the PSD estimate. If you cannot divide the length of `x` exactly into an integer number of sections with 50% overlap, `x` is truncated accordingly.

`pxx = pwelch(x,window)` uses the input vector or integer, `window`, to divide the signal into sections. If `window` is a vector, `pwelch` divides the signal into sections equal in length to the length of `window`. The modified periodograms are computed using the signal sections multiplied by the vector, `window`. If `window` is an integer, the signal is divided into sections of length `window`. The modified periodograms are computed using a Hamming window of length `window`.

`pxx = pwelch(x,window,noverlap)` uses `noverlap` samples of overlap from section to section. `noverlap` must be a positive integer smaller than `window` if `window` is an integer. `noverlap` must be a positive integer less than the length of `window` if `window` is a vector. If you do not specify `noverlap`, or specify `noverlap` as empty, the default number of overlapped samples is 50% of the window length.

`pxx = pwelch(x,window,noverlap,nfft)` specifies the number of discrete Fourier transform (DFT) points to use in the PSD estimate. The default `nfft` is the greater of 256 or the next power of 2 greater than the length of the segments.

`[pxx,w] = pwelch(____)` returns the normalized frequency vector, `w`. If `pxx` is a one-sided PSD estimate, `w` spans the interval  $[0,\pi]$  if `nfft` is even and  $[0,\pi)$  if `nfft` is odd. If `pxx` is a two-sided PSD estimate, `w` spans the interval  $[0,2\pi)$ .

`[pxx,f] = pwelch(____,fs)` returns a frequency vector, `f`, in cycles per unit time. The sampling frequency, `fs`, is the number of samples per unit time. If the unit of time is seconds, then `f` is in cycles/sec (Hz). For real-valued signals, `f` spans the interval  $[0,fs/2]$  when `nfft` is even and  $[0,fs/2)$  when `nfft` is odd. For complex-valued signals, `f` spans the interval  $[0,fs)$ .

`[pxx,w] = pwelch(x,window,noverlap,w)` returns the two-sided Welch PSD estimates at the normalized frequencies specified in the vector, `w`. The vector, `w`, must contain at least 2 elements.

`[pxx,f] = pwelch(x,window,noverlap,f,fs)` returns the two-sided Welch PSD estimates at the frequencies specified in the vector, `f`. The vector, `f`, must contain at least 2 elements. The frequencies in `f` are in cycles per unit time. The sampling frequency, `fs`, is the number of samples per unit time. If the unit of time is seconds, then `f` is in cycles/sec (Hz).

`[____] = pwelch(x,window,____,freqrange)` returns the Welch PSD estimate over the frequency range specified by `freqrange`. Valid options for `freqrange` are: `'onesided'`, `'twosided'`, or `'centered'`.

[ \_\_\_ ] = `pwelch(x,window, ___, spectrumtype)` returns the PSD estimate if `spectrumtype` is specified as `'psd'` and returns the power spectrum if `spectrumtype` is specified as `'power'`.

[ \_\_\_ ] = `pwelch(x,window, ___, trace)` returns the maximum-hold spectrum estimate if `trace` is specified as `'maxhold'` and returns the minimum-hold spectrum estimate if `trace` is specified as `'minhold'`.

[ \_\_\_, pxxc ] = `pwelch( ___, 'ConfidenceLevel', probability)` returns the probability  $\times$  100% confidence intervals for the PSD estimate in `pxxc`.

`pwelch( ___ )` with no output arguments plots the Welch PSD estimate in the current figure window.

## Examples

### Welch Estimate Using Default Inputs

Obtain the Welch PSD estimate of an input signal consisting of a discrete-time sinusoid with an angular frequency of  $\pi/4$  rad/sample with additive  $N(0, 1)$  white noise.

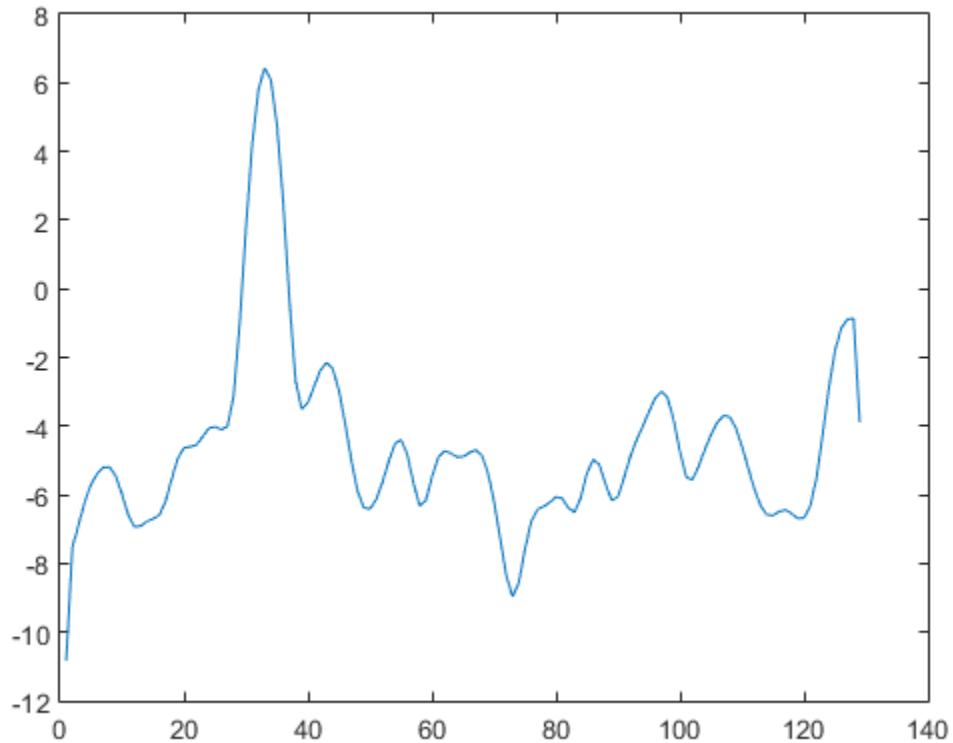
Create a sine wave with an angular frequency of  $\pi/4$  rad/sample with additive  $N(0, 1)$  white noise. Reset the random number generator for reproducible results. The signal is 320 samples in length.

```
rng default
```

```
n = 0:319;  
x = cos(pi/4*n)+randn(size(n));
```

Obtain the Welch PSD estimate using the default Hamming window and DFT length. The default segment length is 71 samples and the DFT length is the 256 points yielding a frequency resolution of  $2\pi/256$  rad/sample. Because the signal is real-valued, the periodogram is one-sided and there are  $256/2+1$  points.

```
pxx = pwelch(x);  
plot(10*log10(pxx))
```



### Welch Estimate Using Specified Segment Length

Obtain the Welch PSD estimate of an input signal consisting of a discrete-time sinusoid with an angular frequency of  $\pi/4$  rad/sample with additive  $N(0, 1)$  white noise.

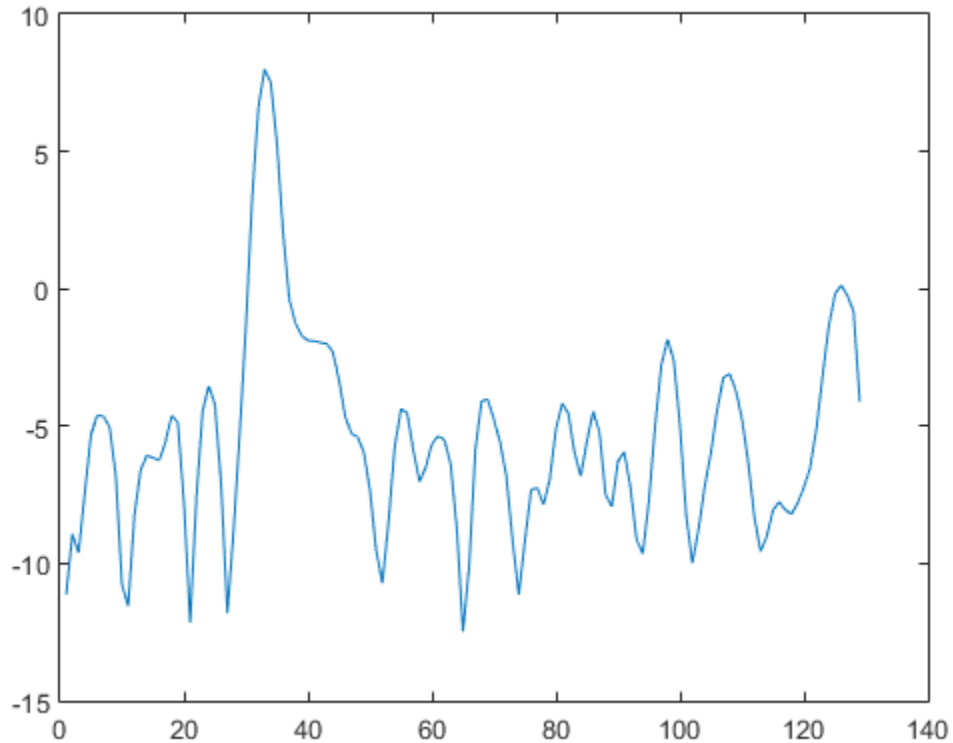
Create a sine wave with an angular frequency of  $\pi/4$  rad/sample with additive  $N(0, 1)$  white noise. Reset the random number generator for reproducible results. The signal is 320 samples in length.

```
rng default
```

```
n = 0:319;  
x = cos(pi/4*n)+randn(size(n));
```

Obtain the Welch PSD estimate dividing the signal into segments 100 samples in length. The signal segments are multiplied by a Hamming window 100 samples in length. The number of overlapped samples is 25. The DFT length is 256 points, yielding a frequency resolution of  $2\pi/256$  rad/sample. Because the signal is real-valued, the PSD estimate is one-sided and there are  $256/2+1$  points.

```
segmentLength = 100;  
noverlap = 25;  
pxx = pwelch(x,segmentLength,noverlap);  
  
plot(10*log10(pxx))
```



### Welch Estimate Specifying Segment Overlap

Obtain the Welch PSD estimate of an input signal consisting of a discrete-time sinusoid with an angular frequency of  $\pi/4$  rad/sample with additive  $N(0, 1)$  white noise.

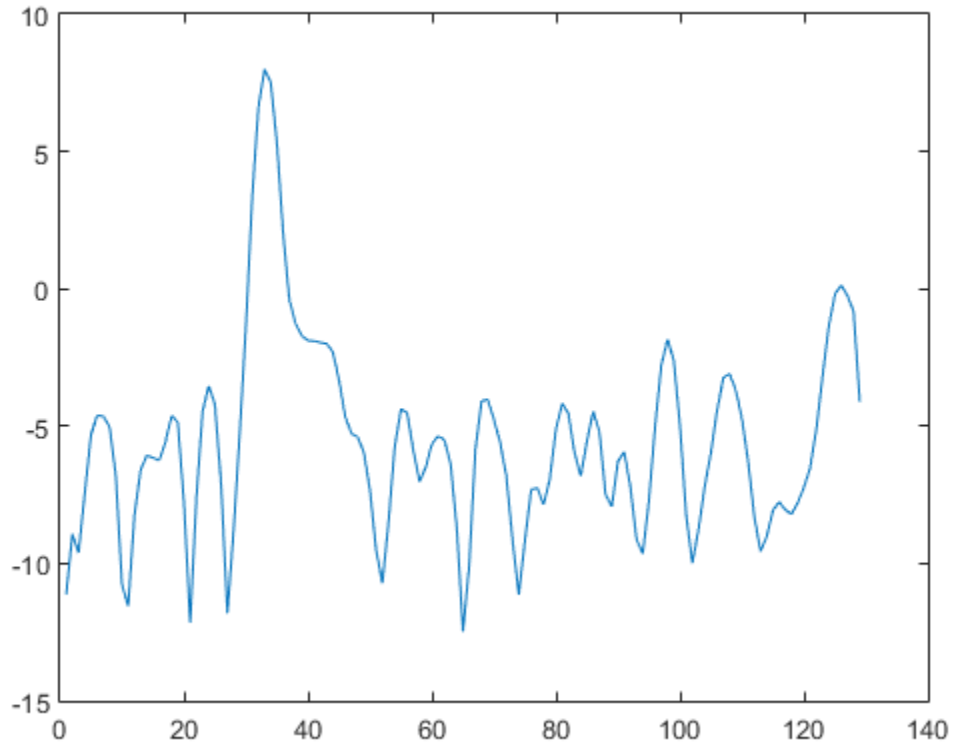
Create a sine wave with an angular frequency of  $\pi/4$  rad/sample with additive  $N(0, 1)$  white noise. Reset the random number generator for reproducible results. The signal is 320 samples in length.

```
rng default  
  
n = 0:319;  
x = cos(pi/4*n)+randn(size(n));
```

Obtain the Welch PSD estimate dividing the signal into segments 100 samples in length. The signal segments are multiplied by a Hamming window 100 samples in length. The number of overlapped samples is 25. The DFT length is 256 points yielding a frequency resolution of  $2\pi/256$  rad/sample. Because the signal is real-valued, the PSD estimate is one-sided and there are  $256/2+1$  points.

```
segmentLength = 100;  
noverlap = 25;  
pxx = pwelch(x,segmentLength,noverlap);  
  
plot(10*log10(pxx))
```





### Welch Estimate Using Specified DFT Length

Obtain the Welch PSD estimate of an input signal consisting of a discrete-time sinusoid with an angular frequency of  $\pi/4$  rad/sample with additive  $N(0, 1)$  white noise.

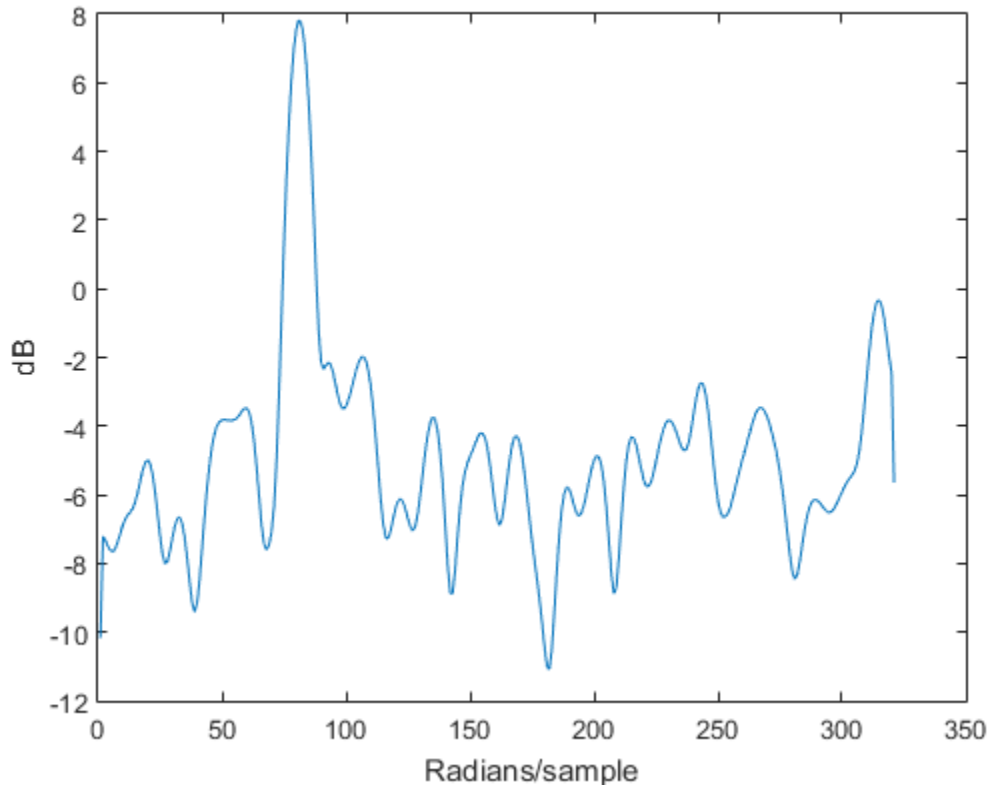
Create a sine wave with an angular frequency of  $\pi/4$  rad/sample with additive  $N(0, 1)$  white noise. Reset the random number generator for reproducible results. The signal is 320 samples in length.

```
rng default  
  
n = 0:319;  
x = cos(pi/4*n)+randn(size(n));
```

Obtain the Welch PSD estimate dividing the signal into segments 100 samples in length. Use the default overlap of 50%. Specify the DFT length to be 640 points so that the frequency of  $\pi/4$  rad/sample corresponds to a DFT bin (bin 81). Because the signal is real-valued, the PSD estimate is one-sided and there are  $640/2+1$  points.

```
segmentLength = 100;
nfft = 640;
pxx = pwelch(x,segmentLength,[],nfft);

plot(10*log10(pxx));
xlabel('Radians/sample')
ylabel('dB')
```



### Welch PSD Estimate of Signal with Frequency in Hertz

Create a signal consisting of a 100 Hz sinusoid in additive  $N(0, 1)$  white noise. Reset the random number generator for reproducible results. The sample rate is 1 kHz and the signal is 5 seconds in duration.

```
rng default
```

```
fs = 1000;
t = 0:1/fs:5-1/fs;
x = cos(2*pi*100*t)+randn(size(t));
```

Obtain Welch's overlapped segment averaging PSD estimate of the preceding signal. Use a segment length of 500 samples with 300 overlapped samples. Use 500 DFT points

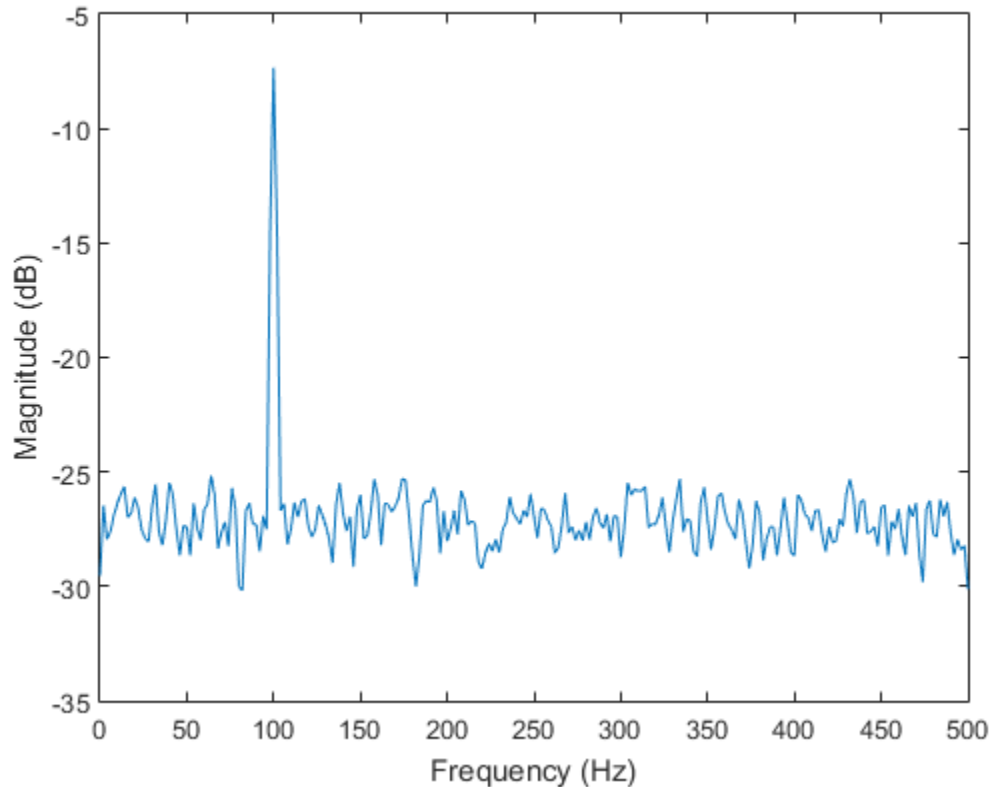
so that 100 Hz falls directly on a DFT bin. Input the sample rate to output a vector of frequencies in Hz. Plot the result.

```
[pxx,f] = pwelch(x,500,300,500,fs);
```

```
plot(f,10*log10(pxx))
```

```
xlabel('Frequency (Hz)')
```

```
ylabel('Magnitude (dB)')
```



### DC-Centered Power Spectrum

Create a signal consisting of a 100 Hz sinusoid in additive  $N(0, 1/4)$  white noise. Reset the random number generator for reproducible results. The sample rate is 1 kHz and the signal is 5 seconds in duration.

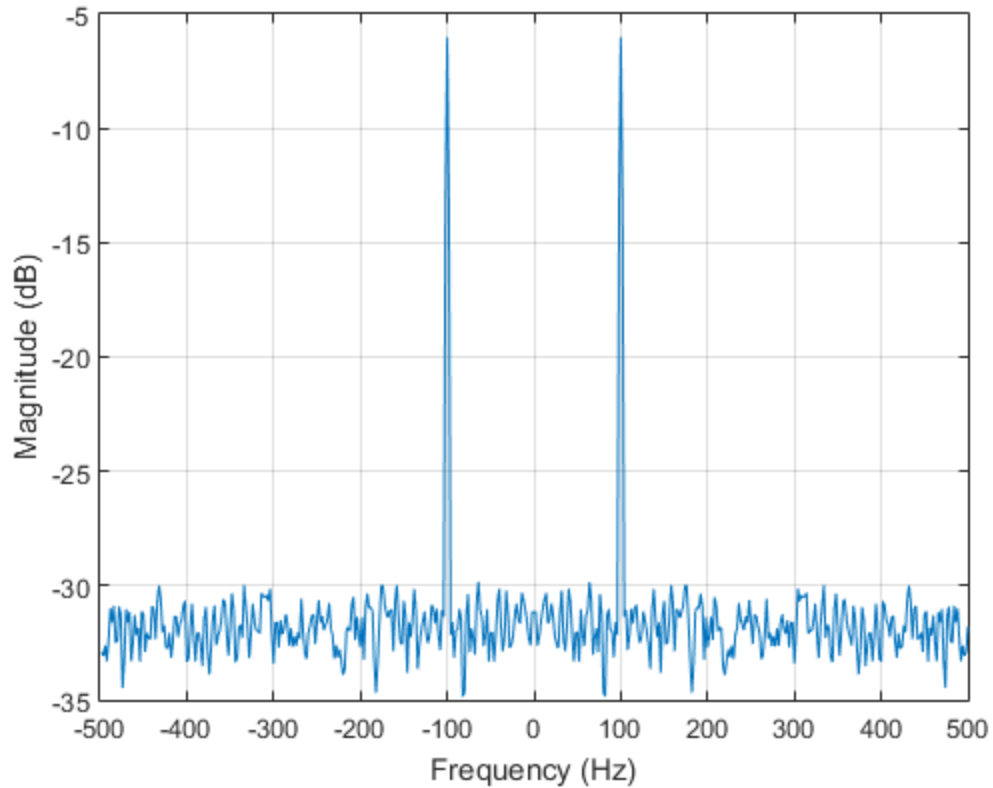
```
rng default

fs = 1000;
t = 0:1/fs:5-1/fs;

noisevar = 1/4;
x = cos(2*pi*100*t)+sqrt(noisevar)*randn(size(t));
```

Obtain the DC-centered power spectrum using Welch's method. Use a segment length of 500 samples with 300 overlapped samples and a DFT length of 500 points. Plot the result.

```
[pxx,f] = pwelch(x,500,300,500,fs,'centered','power');  
  
plot(f,10*log10(pxx))  
xlabel('Frequency (Hz)')  
ylabel('Magnitude (dB)')  
grid
```



You see that the power at -100 and 100 Hz is close to the expected power of 1/4 for a real-valued sine wave with an amplitude of 1. The deviation from 1/4 is due to the effect of the additive noise.

### Maximum-Hold and Minimum-Hold Spectra

Create a signal consisting of three noisy sinusoids and a chirp, sampled at 200 kHz for 0.1 second. The frequencies of the sinusoids are 1 kHz, 10 kHz, and 20 kHz. The sinusoids have different amplitudes and noise levels. The noiseless chirp has a frequency that starts at 20 kHz and increases linearly to 30 kHz during the sampling.

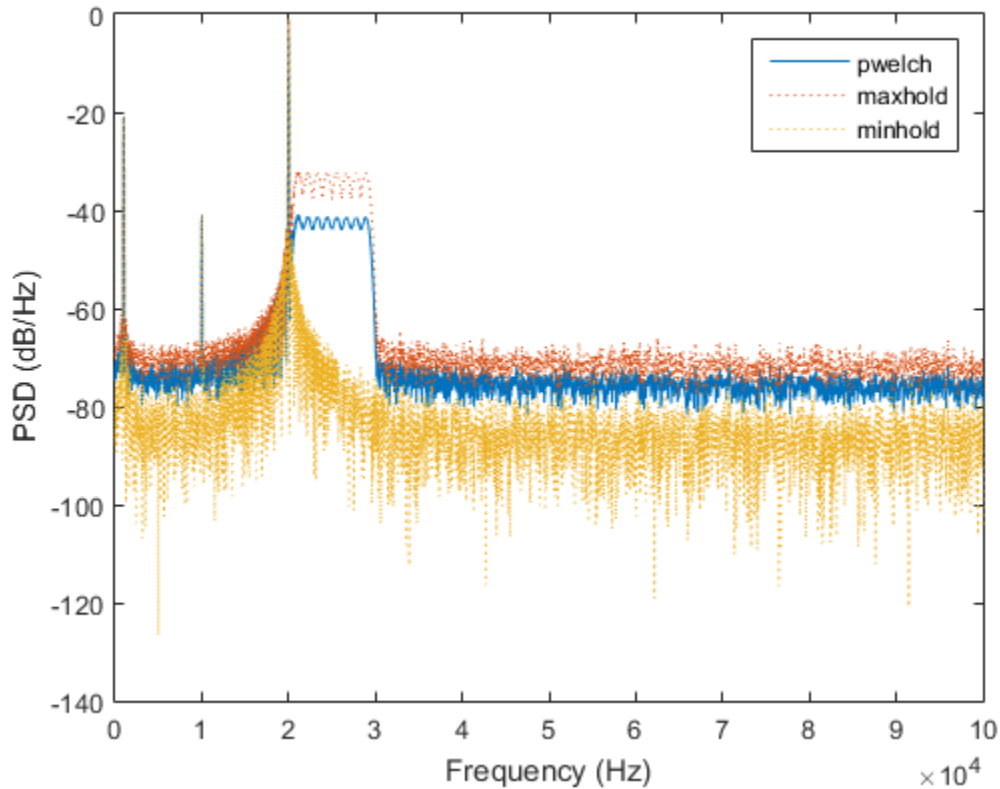
```
Fs = 200e3;
Fc = [1 10 20]'*1e3;
Ns = 0.1*Fs;

t = (0:Ns-1)/Fs;
x = [1 1/10 10]*sin(2*pi*Fc*t)+[1/200 1/2000 1/20]*randn(3,Ns);
x = x+chirp(t,20e3,t(end),30e3);
```

Compute the Welch PSD estimate and the maximum-hold and minimum-hold spectra of the signal. Plot the results.

```
[pxx,f] = pwelch(x,[],[],[],Fs);
pmax = pwelch(x,[],[],[],Fs,'maxhold');
pmin = pwelch(x,[],[],[],Fs,'minhold');

plot(f,pow2db(pxx))
hold on
plot(f,pow2db([pmax pmin]),':')
hold off
xlabel('Frequency (Hz)')
ylabel('PSD (dB/Hz)')
legend('pwelch','maxhold','minhold')
```

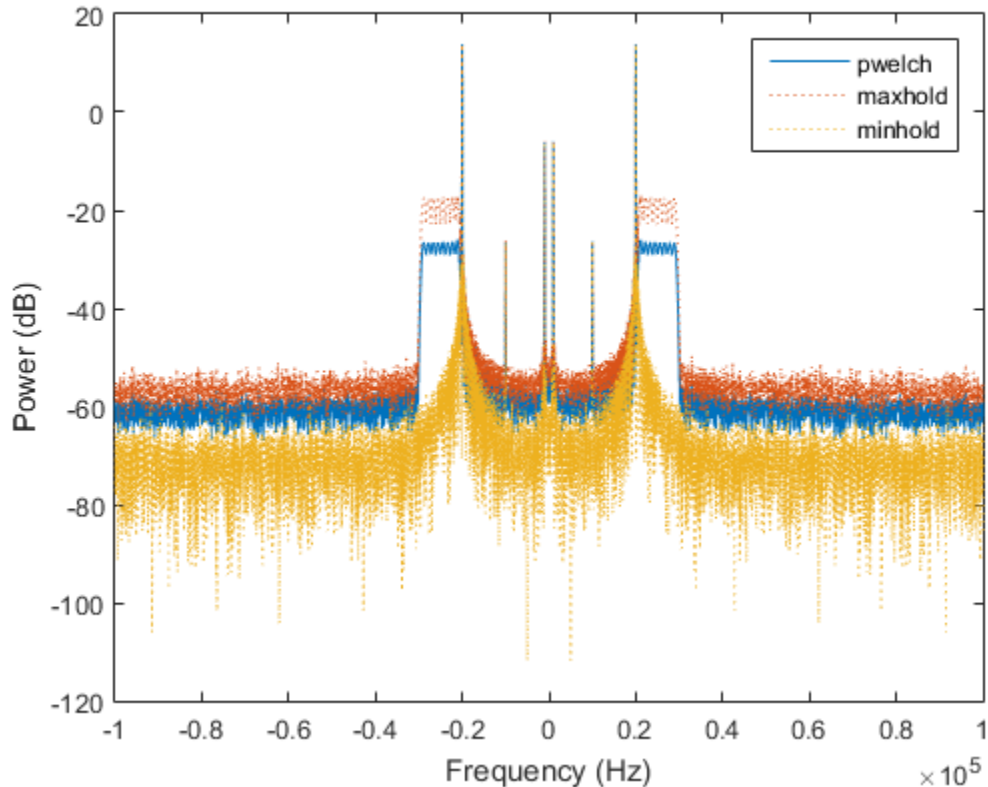


Repeat the procedure, this time computing centered power spectrum estimates.

```
[pxx,f] = pwelch(x,[],[],[],Fs,'centered','power');
pmax = pwelch(x,[],[],[],Fs,'maxhold','centered','power');
pmin = pwelch(x,[],[],[],Fs,'minhold','centered','power');

plot(f,pow2db(pxx))
hold on
plot(f,pow2db([pmax pmin]),':')
hold off
xlabel('Frequency (Hz)')
ylabel('Power (dB)')
legend('pwelch','maxhold','minhold')
```





### Upper and Lower 95%-Confidence Bounds

This example illustrates the use of confidence bounds with Welch's overlapped segment averaging (WOSA) PSD estimate. While not a necessary condition for statistical significance, frequencies in Welch's estimate where the lower confidence bound exceeds the upper confidence bound for surrounding PSD estimates clearly indicate significant oscillations in the time series.

Create a signal consisting of the superposition of 100 Hz and 150 Hz sine waves in additive white  $N(0, 1)$  noise. The amplitude of the two sine waves is 1. The sample rate is 1 kHz. Reset the random number generator for reproducible results.

```
rng default
```

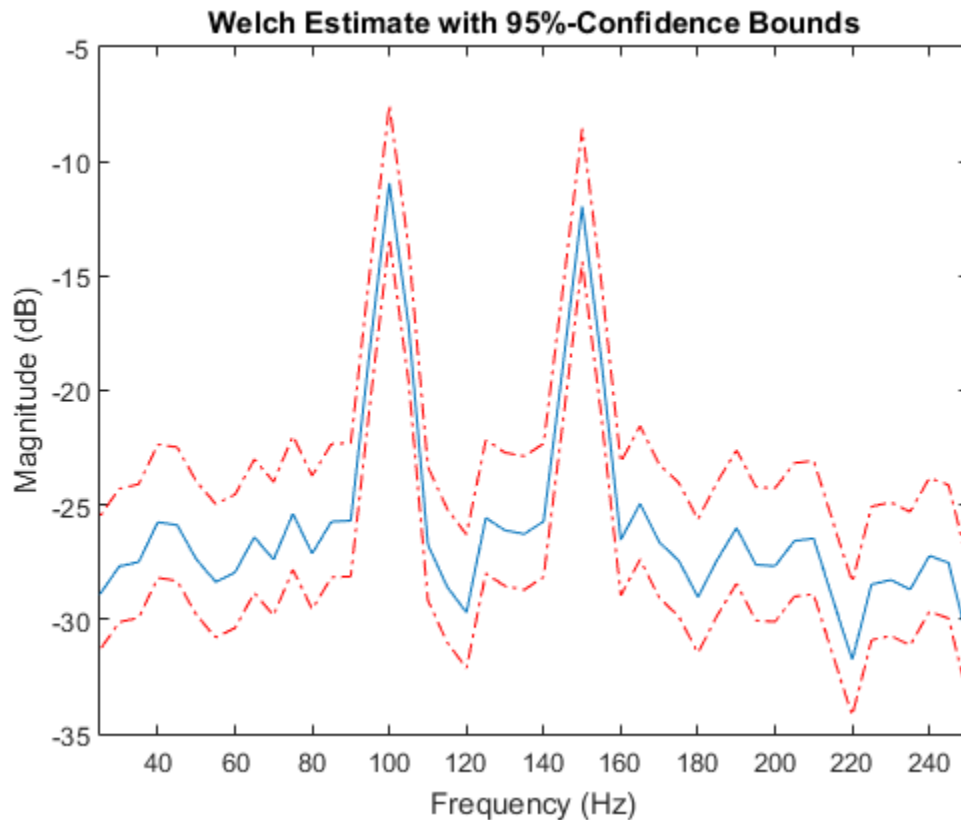
```
t = 0:0.001:1-0.001;
fs = 1000;
x = cos(2*pi*100*t)+sin(2*pi*150*t)+randn(size(t));
```

Obtain the WOSA estimate with 95%-confidence bounds. Set the segment length equal to 200 and overlap the segments by 50% (100 samples). Plot the WOSA PSD estimate along with the confidence interval and zoom in on the frequency region of interest near 100 and 150 Hz.

```
L = 200;
noverlap = 100;
[pxx,f,pxxc] = pwelch(x,hamming(L),noverlap,200,fs,...
    'ConfidenceLevel',0.95);

plot(f,10*log10(pxx))
hold on
plot(f,10*log10(pxxc),'r-.')

xlim([25 250])
xlabel('Frequency (Hz)')
ylabel('Magnitude (dB)');
title('Welch Estimate with 95%-Confidence Bounds');
```



At 100 and 150 Hz, the lower confidence bound exceeds the upper confidence bounds for surrounding PSD estimates.

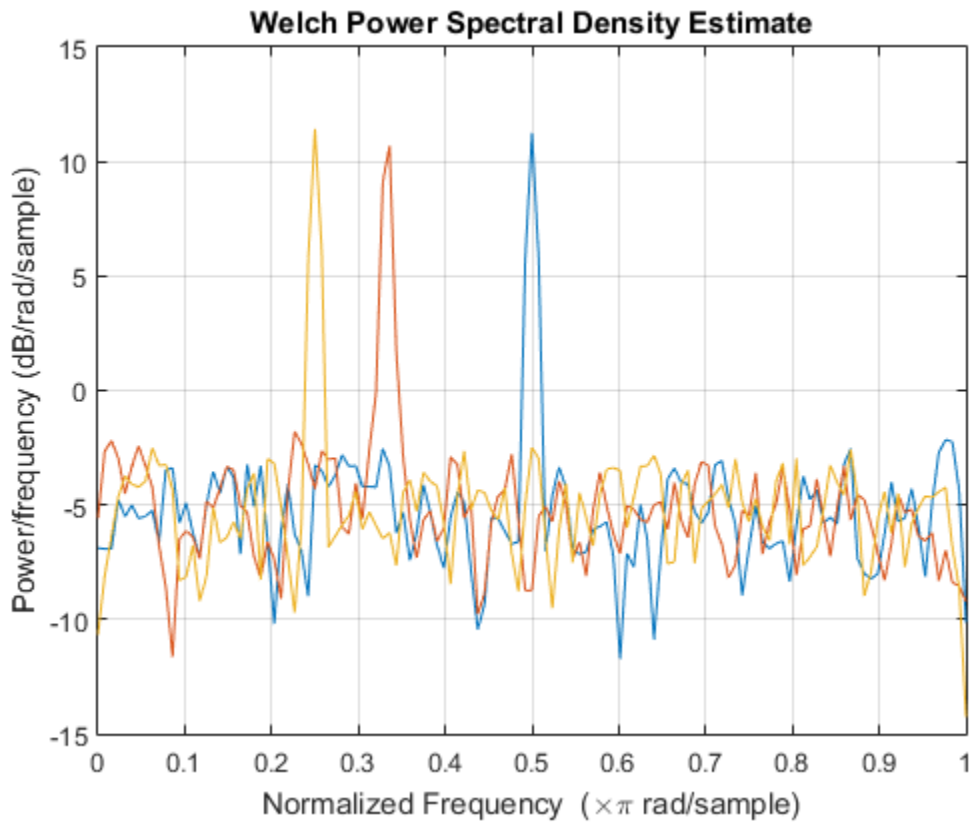
### Welch PSD Estimate of a Multichannel Signal

Generate 1024 samples of a multichannel signal consisting of three sinusoids in additive  $N(0, 1)$  white Gaussian noise. The sinusoids' frequencies are  $\pi/2$ ,  $\pi/3$ , and  $\pi/4$  rad/sample. Estimate the PSD of the signal using Welch's method and plot it.

```
N = 1024;
n = 0:N-1;
```

```
w = pi./[2;3;4];
```

```
x = cos(w*n)' + randn(length(n),3);  
pwelch(x)
```



- “Bias and Variability in the Periodogram”

## Input Arguments

**x** — Input signal  
vector | matrix

Input signal, specified as a row or column vector, or as a matrix. If  $x$  is a matrix, then its columns are treated as independent channels.

Example: `cos(pi/4*(0:159))+randn(1,160)` is a single-channel row-vector signal.

Example: `cos(pi./[4;2]*(0:159))'+randn(160,2)` is a two-channel signal.

Data Types: `single` | `double`

Complex Number Support: Yes

### **window** — Window

`integer` | `vector` | `[]`

Window, specified as a row or column vector or an integer. If `window` is a vector, `pwelch` divides  $x$  into overlapping sections of length equal to the length of `window`, and then multiplies each signal section with the vector specified in `window`. If `window` is an integer, `pwelch` is divided into sections of length equal to the integer value, and a Hamming window of equal length is used. If the length of  $x$  cannot be divided exactly into an integer number of sections with `noverlap` number of overlapping samples,  $x$  is truncated accordingly. If you specify `window` as empty, the default Hamming window is used to obtain eight sections of  $x$  with `noverlap` overlapping samples.

Data Types: `single` | `double`

### **noverlap** — Number of overlapped samples

`positive integer` | `[]`

Number of overlapped samples, specified as a positive integer smaller than the length of `window`. If you omit `noverlap` or specify `noverlap` as empty, a value is used to obtain 50% overlap between segments.

### **nfft** — Number of DFT points

`max(256,2^nextpow2(length(window)))` (default) | `integer` | `[]`

Number of DFT points, specified as a positive integer. For a real-valued input signal,  $x$ , the PSD estimate, `pxx` has length  $(nfft/2 + 1)$  if `nfft` is even, and  $(nfft + 1)/2$  if `nfft` is odd. For a complex-valued input signal,  $x$ , the PSD estimate always has length `nfft`. If `nfft` is specified as empty, the default `nfft` is used.

If `nfft` is greater than the segment length, the data is zero-padded. If `nfft` is less than the segment length, the segment is wrapped using `datawrap` to make the length equal to `nfft`.

Data Types: `single` | `double`

**fs — Sampling frequency**

positive scalar

Sampling frequency, specified as a positive scalar. The sampling frequency is the number of samples per unit time. If the unit of time is seconds, the sampling frequency has units of hertz.

**w — Normalized frequencies**

vector

Normalized frequencies, specified as a row or column vector with at least 2 elements. Normalized frequencies are in rad/sample.

Example: `w = [pi/4 pi/2]`

Data Types: double

**f — Cyclical frequencies**

vector

Cyclical frequencies, specified as a row or column vector with at least 2 elements. The frequencies are in cycles per unit time. The unit time is specified by the sampling frequency, `fs`. If `fs` has units of samples/second, then `f` has units of Hz.

Example: `fs = 1000; f = [100 200]`

Data Types: double

**freqrange — Frequency range for PSD estimate**

'onesided' | 'twosided' | 'centered'

Frequency range for the PSD estimate, specified as a one of 'onesided', 'twosided', or 'centered'. The default is 'onesided' for real-valued signals and 'twosided' for complex-valued signals. The frequency ranges corresponding to each option are

- 'onesided' — returns the one-sided PSD estimate of a real-valued input signal, `x`. If `nfft` is even, `pxx` has length  $nfft/2 + 1$  and is computed over the interval  $[0, \pi]$  rad/sample. If `nfft` is odd, the length of `pxx` is  $(nfft + 1)/2$  and the interval is  $[0, \pi]$  rad/sample. When `fs` is optionally specified, the corresponding intervals are  $[0, fs/2]$  cycles/unit time and  $[0, fs/2)$  cycles/unit time for even and odd length `nfft` respectively.
- 'twosided' — returns the two-sided PSD estimate for either the real-valued or complex-valued input, `x`. In this case, `pxx` has length `nfft` and is computed over the interval  $[0, 2\pi]$  rad/sample. When `fs` is optionally specified, the interval is  $[0, fs)$  cycles/unit time.

- **'centered'** — returns the centered two-sided PSD estimate for either the real-valued or complex-valued input,  $x$ . In this case,  $pxx$  has length  $nfft$  and is computed over the interval  $(-\pi, \pi]$  rad/sample for even length  $nfft$  and  $(-\pi, \pi)$  rad/sample for odd length  $nfft$ . When  $fs$  is optionally specified, the corresponding intervals are  $(-fs/2, fs/2]$  cycles/unit time and  $(-fs/2, fs/2)$  cycles/unit time for even and odd length  $nfft$  respectively.

Data Types: char

### **spectrumtype — Power spectrum scaling**

'psd' (default) | 'power'

Power spectrum scaling, specified as one of 'psd' or 'power'. Omitting the `spectrumtype`, or specifying 'psd', returns the power spectral density. Specifying 'power' scales each estimate of the PSD by the equivalent noise bandwidth of the window. Use the 'power' option to obtain an estimate of the power at each frequency.

Data Types: char

### **trace — Trace mode**

'mean' (default) | 'maxhold' | 'minhold'

Trace mode, specified as one of 'mean', 'maxhold', or 'minhold'. The default is 'mean'.

- **'mean'** — returns the Welch spectrum estimate of each input channel. `pwelch` computes the Welch spectrum estimate at each frequency bin by averaging the power spectrum estimates of all the segments.
- **'maxhold'** — returns the maximum-hold spectrum of each input channel. `pwelch` computes the maximum-hold spectrum at each frequency bin by keeping the maximum value among the power spectrum estimates of all the segments.
- **'minhold'** — returns the minimum-hold spectrum of each input channel. `pwelch` computes the minimum-hold spectrum at each frequency bin by keeping the minimum value among the power spectrum estimates of all the segments.

### **probability — Confidence interval for PSD estimate**

0.95 (default) | scalar in the range (0,1)

Coverage probability for the true PSD, specified as a scalar in the range (0,1). The output, `pxxc`, contains the lower and upper bounds of the  $\text{probability} \times 100\%$  interval estimate for the true PSD.

## Output Arguments

### **pxx** — PSD estimate

vector | matrix

PSD estimate, returned as a real-valued, nonnegative column vector or matrix. Each column of `pxx` is the PSD estimate of the corresponding column of `x`. The units of the PSD estimate are in squared magnitude units of the time series data per unit frequency. For example, if the input data is in volts, the PSD estimate is in units of squared volts per unit frequency. For a time series in volts, if you assume a resistance of  $1 \Omega$  and specify the sampling frequency in hertz, the PSD estimate is in watts per hertz.

Data Types: `single` | `double`

### **w** — Normalized frequencies

vector

Normalized frequencies, returned as a real-valued column vector. If `pxx` is a one-sided PSD estimate, `w` spans the interval  $[0, \pi]$  if `nfft` is even and  $[0, \pi)$  if `nfft` is odd. If `pxx` is a two-sided PSD estimate, `w` spans the interval  $[0, 2\pi)$ . For a DC-centered PSD estimate, `w` spans the interval  $(-\pi, \pi]$  rad/sample for even length `nfft` and  $(-\pi, \pi)$  radians/sample for odd length `nfft`.

Data Types: `double`

### **f** — Cyclical frequencies

vector

Cyclical frequencies, returned as a real-valued column vector. For a one-sided PSD estimate, `f` spans the interval  $[0, fs/2]$  when `nfft` is even and  $[0, fs/2)$  when `nfft` is odd. For a two-sided PSD estimate, `f` spans the interval  $[0, fs)$ . For a DC-centered PSD estimate, `f` spans the interval  $(-fs/2, fs/2]$  cycles/unit time for even length `nfft` and  $(-fs/2, fs/2)$  cycles/unit time for odd length `nfft`.

Data Types: `double`

### **pxxc** — Confidence bounds

matrix

Confidence bounds, returned as a matrix with real-valued elements. The row size of the matrix is equal to the length of the PSD estimate, `pxx`. `pxxc` has twice as many columns as `pxx`. Odd-numbered columns contain the lower bounds of the confidence intervals, and



even-numbered columns contain the upper bounds. Thus,  $\text{pxxc}(m, 2*n - 1)$  is the lower confidence bound and  $\text{pxxc}(m, 2*n)$  is the upper confidence bound corresponding to the estimate  $\text{pxx}(m, n)$ . The coverage probability of the confidence intervals is determined by the value of the probability input.

Data Types: `single` | `double`

## More About

### Welch's Overlapped Segment Averaging Spectral Estimation

The periodogram is not a consistent estimator of the true power spectral density of a wide-sense stationary process. Welch's technique to reduce the variance of the periodogram breaks the time series into segments, usually overlapping. Welch's method computes a modified periodogram for each segment and then averages these estimates to produce the estimate of the power spectral density. Because the process is wide-sense stationary and Welch's method uses PSD estimates of different segments of the time series, the modified periodograms represent approximately uncorrelated estimates of the true PSD and averaging reduces the variability.

The segments are typically multiplied by a window function, such as a Hamming window, so that Welch's method amounts to averaging modified periodograms. Because the segments usually overlap, data values at the beginning and end of the segment tapered by the window in one segment, occur away from the ends of adjacent segments. This guards against the loss of information caused by windowing.

- "Spectral Analysis"

### See Also

`periodogram` | `pmtm`

# pyulear

Autoregressive power spectral density estimate — Yule-Walker method

## Syntax

```
pxx = pyulear(x,order)
pxx = pyulear(x,order,nfft)

[pxx,w] = pyulear( ___ )
[pxx,f] = pyulear( ___ ,fs)

[pxx,w] = pyulear(x,order,w)
[pxx,f] = pyulear(x,order,f,fs)

[ ___ ] = pyulear(x,order, ___ ,freqrange)

[ ___ ,pxxc] = pyulear( ___ , 'ConfidenceLevel',probability)

pyulear( ___ )
```

## Description

`pxx = pyulear(x,order)` returns the power spectral density estimate, `pxx`, of a discrete-time signal, `x`, found using the Yule-Walker method. When `x` is a vector, it is treated as a single channel. When `x` is a matrix, the PSD is computed independently for each column and stored in the corresponding column of `pxx`. `pxx` is the distribution of power per unit frequency. The frequency is expressed in units of rad/sample. `order` is the order of the autoregressive (AR) model used to produce the PSD estimate.

`pxx = pyulear(x,order,nfft)` uses `nfft` points in the discrete Fourier transform (DFT). For real `x`, `pxx` has length  $(nfft/2 + 1)$  if `nfft` is even, and  $(nfft + 1)/2$  if `nfft` is odd. For complex-valued `x`, `pxx` always has length `nfft`. `pyulear` uses a default DFT length of 256.

`[pxx,w] = pyulear( ___ )` returns the vector of normalized angular frequencies, `w`, at which the PSD is estimated. `w` has units of rad/sample. For real-valued signals, `w`

spans the interval  $[0,\pi]$  when `nfft` is even and  $[0,\pi)$  when `nfft` is odd. For complex-valued signals, `w` always spans the interval  $[0,2\pi)$ .

`[pxx,f] = pyulear( ____, fs)` returns a frequency vector, `f`, in cycles per unit time. The sampling frequency, `fs`, is the number of samples per unit time. If the unit of time is seconds, then `f` is in cycles/second (Hz). For real-valued signals, `f` spans the interval  $[0,fs/2]$  when `nfft` is even and  $[0,fs/2)$  when `nfft` is odd. For complex-valued signals, `f` spans the interval  $[0,fs)$ .

`[pxx,w] = pyulear(x,order,w)` returns the two-sided AR PSD estimates at the normalized frequencies specified in the vector, `w`. The vector, `w`, must contain at least two elements.

`[pxx,f] = pyulear(x,order,f,fs)` returns the two-sided AR PSD estimates at the frequencies specified in the vector, `f`. The vector, `f`, must contain at least two elements. The frequencies in `f` are in cycles per unit time. The sampling frequency, `fs`, is the number of samples per unit time. If the unit of time is seconds, then `f` is in cycles/second (Hz).

`[ ____ ] = pyulear(x,order, ____, freqrange)` returns the AR PSD estimate over the frequency range specified by `freqrange`. Valid options for `freqrange` are: `'onesided'`, `'twosided'`, or `'centered'`.

`[ ____, pxxc] = pyulear( ____, 'ConfidenceLevel', probability)` returns the `probability` × 100% confidence intervals for the PSD estimate in `pxxc`.

`pyulear( ____ )` with no output arguments plots the AR PSD estimate in dB per unit frequency in the current figure window.

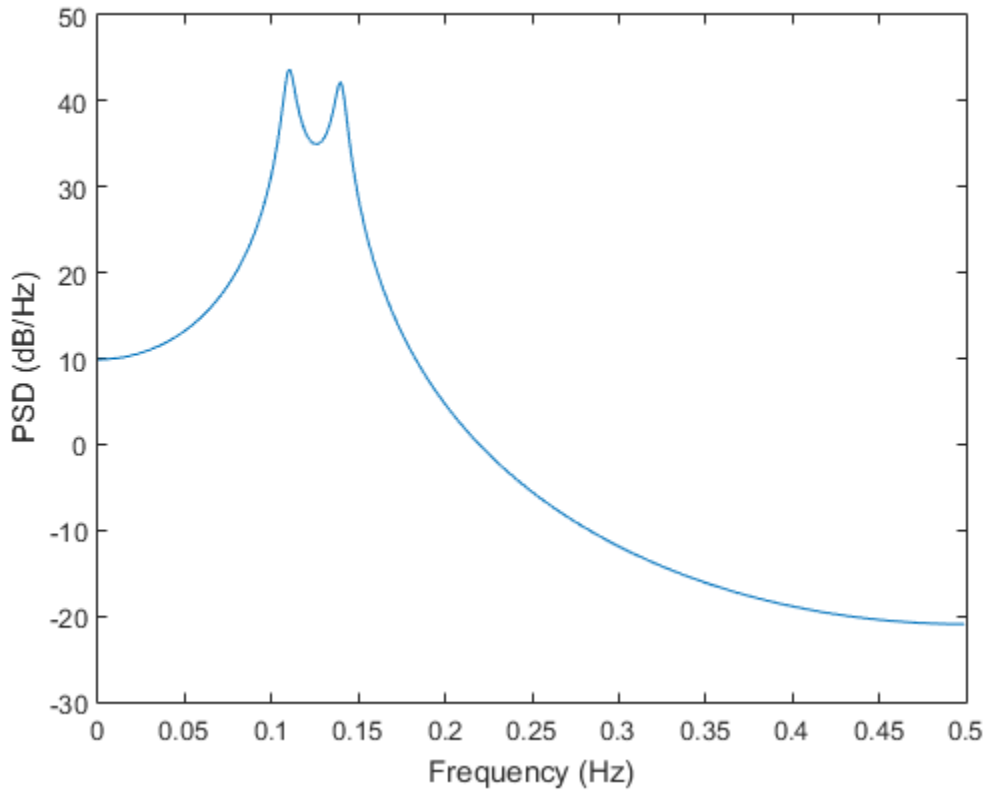
## Examples

### Yule-Walker PSD Estimate of an AR(4) Process

Create a realization of an AR(4) wide-sense stationary random process. Estimate the PSD using the Yule-Walker method. Compare the PSD estimate based on a single realization to the true PSD of the random process.

Create an AR(4) system function. Obtain the frequency response and plot the PSD of the system.

```
A = [1 -2.7607 3.8106 -2.6535 0.9238];  
[H,F] = freqz(1,A,[],1);  
plot(F,20*log10(abs(H))  
  
xlabel('Frequency (Hz)')  
ylabel('PSD (dB/Hz)')
```



Create a realization of the AR(4) random process. Set the random number generator to the default settings for reproducible results. The realization is 1000 samples in length. Assume a sampling frequency of 1 Hz. Use `pyulear` to estimate the PSD for a 4th-order process. Compare the PSD estimate with the true PSD.

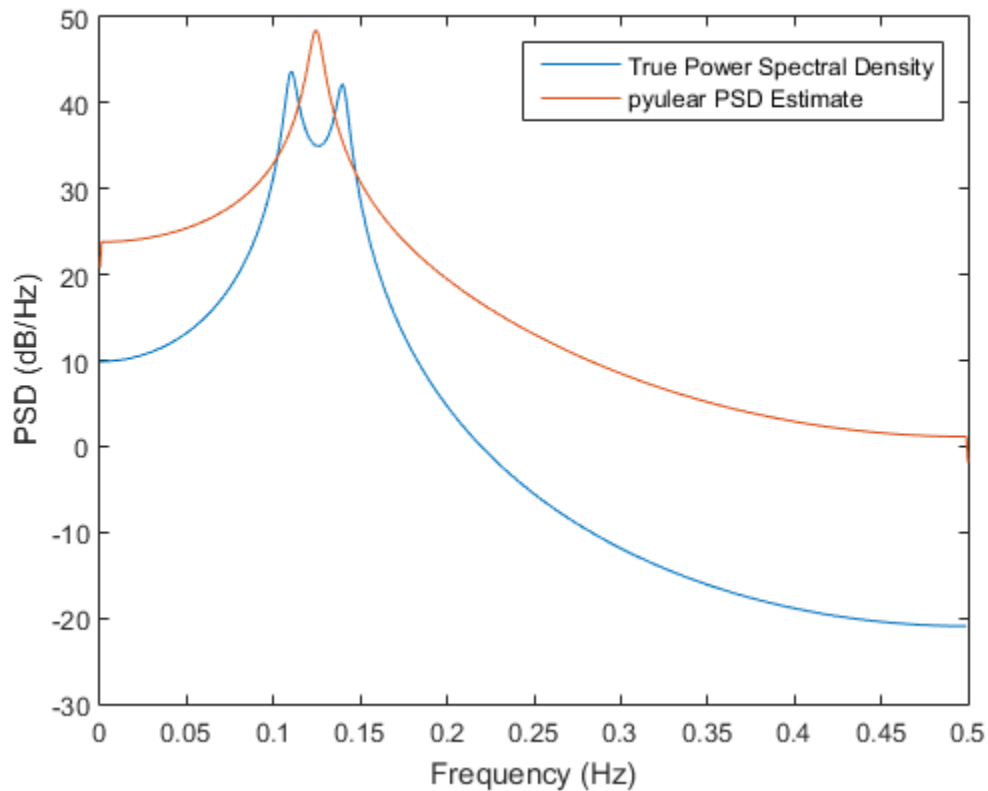
```
rng default
```

```

x = randn(1000,1);
y = filter(1,A,x);
[Pxx,F] = pyulear(y,4,1024,1);

hold on
plot(F,10*log10(Pxx))
legend('True Power Spectral Density','pyulear PSD Estimate')

```



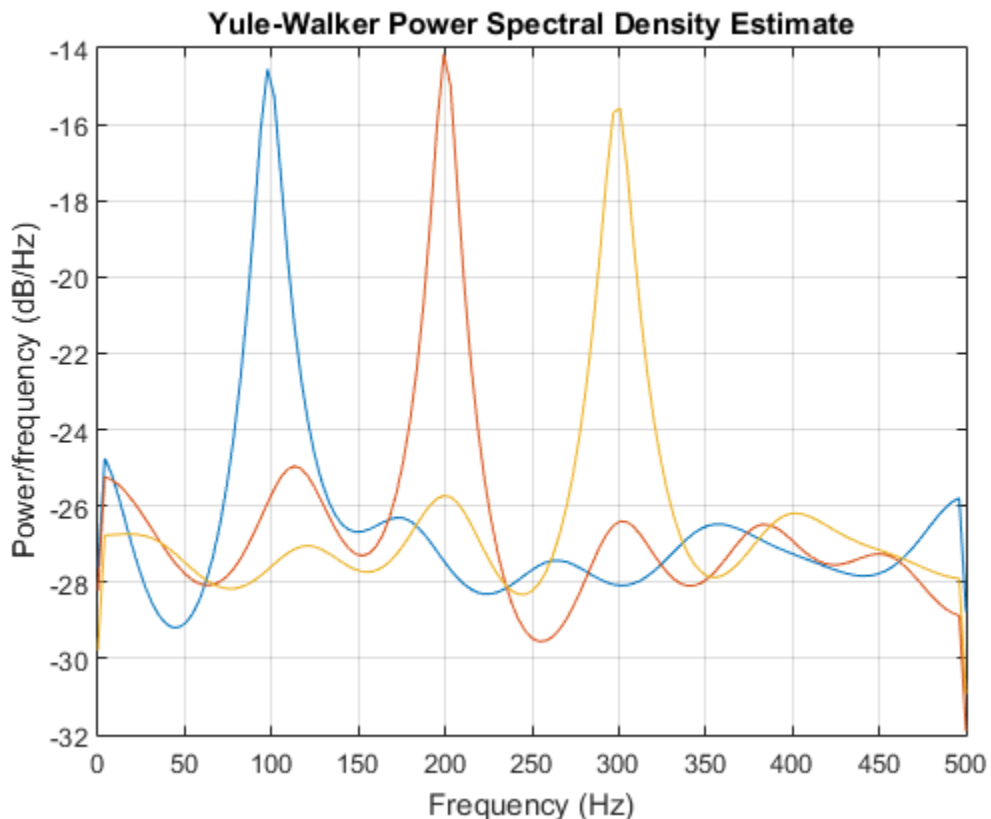
### Yule-Walker PSD Estimate of a Multichannel Signal

Create a multichannel signal consisting of three sinusoids in additive  $N(0,1)$  white Gaussian noise. The sinusoids' frequencies are 100 Hz, 200 Hz, and 300 Hz. The sampling frequency is 1 kHz, and the signal has a duration of 1 s.

```
Fs = 1000;  
t = 0:1/Fs:1-1/Fs;  
f = [100;200;300];  
x = cos(2*pi*f*t)' + randn(length(t),3);
```

Estimate the PSD of the signal using the Yule-Walker method with a 12th-order autoregressive model. Use the default DFT length. Plot the estimate.

```
morder = 12;  
pyulear(x,morder,[],Fs)
```



## Input Arguments

### **x** — Input signal

vector | matrix

Input signal, specified as a row or column vector, or as a matrix. If  $x$  is a matrix, then its columns are treated as independent channels.

Example: `cos(pi/4*(0:159))+randn(1,160)` is a single-channel row-vector signal.

Example: `cos(pi./[4;2]*(0:159))'+randn(160,2)` is a two-channel signal.

Data Types: `single` | `double`

Complex Number Support: Yes

**order** — Order of autoregressive model

positive integer

Order of the autoregressive model, specified as a positive integer.

Data Types: double

**nfft** — Number of DFT points

256 (default) | integer | []

Number of DFT points, specified as a positive integer. For a real-valued input signal,  $x$ , the PSD estimate,  $pxx$  has length  $(nfft/2+1)$  if  $nfft$  is even, and  $(nfft+1)/2$  if  $nfft$  is odd. For a complex-valued input signal,  $x$ , the PSD estimate always has length  $nfft$ . If  $nfft$  is specified as empty, the default  $nfft$  is used.

Data Types: single | double

**fs** — Sampling frequency

positive scalar

Sampling frequency, specified as a positive scalar. The sampling frequency is the number of samples per unit time. If the unit of time is seconds, the sampling frequency has units of hertz.

**w** — Normalized frequencies

vector

Normalized frequencies, specified as a row or column vector with at least 2 elements. Normalized frequencies are in rad/sample.

Example:  $w = [\pi/4 \ \pi/2]$

Data Types: double

**f** — Cyclical frequencies

vector

Cyclical frequencies, specified as a row or column vector with at least 2 elements. The frequencies are in cycles per unit time. The unit time is specified by the sampling frequency,  $fs$ . If  $fs$  has units of samples/second, then  $f$  has units of Hz.

Example:  $fs = 1000; f = [100 \ 200]$



Data Types: double

### **freqrange** — Frequency range for PSD estimate

'onesided' | 'twosided' | 'centered'

Frequency range for the PSD estimate, specified as a one of 'onesided', 'twosided', or 'centered'. The default is 'onesided' for real-valued signals and 'twosided' for complex-valued signals. The frequency ranges corresponding to each option are

- 'onesided' — returns the one-sided PSD estimate of a real-valued input signal,  $x$ . If  $nfft$  is even,  $pxx$  has length  $nfft/2 + 1$  and is computed over the interval  $[0, \pi]$  rad/sample. If  $nfft$  is odd, the length of  $pxx$  is  $(nfft + 1)/2$  and the interval is  $[0, \pi]$  rad/sample. When  $fs$  is optionally specified, the corresponding intervals are  $[0, fs/2]$  cycles/unit time and  $[0, fs/2)$  cycles/unit time for even and odd length  $nfft$  respectively.
- 'twosided' — returns the two-sided PSD estimate for either the real-valued or complex-valued input,  $x$ . In this case,  $pxx$  has length  $nfft$  and is computed over the interval  $[0, 2\pi)$  rad/sample. When  $fs$  is optionally specified, the interval is  $[0, fs)$  cycles/unit time.
- 'centered' — returns the centered two-sided PSD estimate for either the real-valued or complex-valued input,  $x$ . In this case,  $pxx$  has length  $nfft$  and is computed over the interval  $(-\pi, \pi]$  rad/sample for even length  $nfft$  and  $(-\pi, \pi)$  rad/sample for odd length  $nfft$ . When  $fs$  is optionally specified, the corresponding intervals are  $(-fs/2, fs/2]$  cycles/unit time and  $(-fs/2, fs/2)$  cycles/unit time for even and odd length  $nfft$  respectively.

Data Types: char

### **probability** — Confidence interval for PSD estimate

0.95 (default) | scalar in the range (0,1)

Coverage probability for the true PSD, specified as a scalar in the range (0,1). The output,  $pxxc$ , contains the lower and upper bounds of the probability  $\times 100\%$  interval estimate for the true PSD.

## Output Arguments

### **pxx** — PSD estimate

vector | matrix

PSD estimate, returned as a real-valued, nonnegative column vector or matrix. Each column of `pxx` is the PSD estimate of the corresponding column of `x`. The units of the PSD estimate are in squared magnitude units of the time series data per unit frequency. For example, if the input data is in volts, the PSD estimate is in units of squared volts per unit frequency. For a time series in volts, if you assume a resistance of  $1 \Omega$  and specify the sampling frequency in hertz, the PSD estimate is in watts per hertz.

Data Types: `single` | `double`

### **w** — Normalized frequencies

vector

Normalized frequencies, returned as a real-valued column vector. If `pxx` is a one-sided PSD estimate, `w` spans the interval  $[0, \pi]$  if `nfft` is even and  $[0, \pi)$  if `nfft` is odd. If `pxx` is a two-sided PSD estimate, `w` spans the interval  $[0, 2\pi)$ . For a DC-centered PSD estimate, `f` spans the interval  $(-\pi, \pi]$  rad/sample for even length `nfft` and  $(-\pi, \pi)$  radians/sample for odd length `nfft`.

Data Types: `double`

### **f** — Cyclical frequencies

vector

Cyclical frequencies, returned as a real-valued column vector. For a one-sided PSD estimate, `f` spans the interval  $[0, fs/2]$  when `nfft` is even and  $[0, fs/2)$  when `nfft` is odd. For a two-sided PSD estimate, `f` spans the interval  $[0, fs)$ . For a DC-centered PSD estimate, `f` spans the interval  $(-fs/2, fs/2]$  cycles/unit time for even length `nfft` and  $(-fs/2, fs/2)$  cycles/unit time for odd length `nfft`.

Data Types: `double`

### **pxxc** — Confidence bounds

matrix

Confidence bounds, returned as a matrix with real-valued elements. The row size of the matrix is equal to the length of the PSD estimate, `pxx`. `pxxc` has twice as many columns as `pxx`. Odd-numbered columns contain the lower bounds of the confidence intervals, and even-numbered columns contain the upper bounds. Thus, `pxxc(m, 2*n - 1)` is the lower confidence bound and `pxxc(m, 2*n)` is the upper confidence bound corresponding to the estimate `pxx(m, n)`. The coverage probability of the confidence intervals is determined by the value of the probability input.

Data Types: `single` | `double`

## **See Also**

pburg | pcov | pmcov

# realizemdl

Simulink subsystem block for filter

## Syntax

```
realizemdl(FiltObject)
realizemdl(FiltObject,propertyname1,propertyvalue1,...)
```

## Description

`realizemdl(FiltObject)` generates a model of the filter object `FiltObject` in a Simulink subsystem block using sum, gain, and delay blocks from Simulink. The properties and values of `FiltObject` define the resulting subsystem block parameters.

`realizemdl` requires Simulink. To accurately realize models of quantized filters, use Fixed-Point Designer.

`realizemdl(FiltObject,propertyname1,propertyvalue1,...)` generates the model for `FiltObject` with the associated `propertyname/propertyvalue` pairs, and any other values you set in `FiltObject`.

---

**Note** Subsystem filter blocks that you use `realizemdl` to create support sample-based input and output only. You cannot input or output frame-based signals with the block.

---

Using the optional `propertyname/propertyvalue` pairs lets you control more fully the way the block subsystem model gets built, such as where the block goes, what the name is, or how to optimize the block structure. Valid properties and values for `realizemdl` are listed in this table, with the default value noted and descriptions of what the properties do.

| Property Name | Property Values                                      | Description  |
|---------------|--|--|
| Destination   | 'current' (default) or 'new' or <i>Subsystemname</i> | Specify whether to add the block to your current Simulink model or create a new model to contain the block. If you |

| Property Name    | Property Values         | Description   |
|------------------|-------------------------|---|
|                  |                         | provide the name of a current subsystem in <i>subsystemname</i> , <i>realizemdl</i> adds the new block to the specified subsystem.  |
| Blockname        | 'filter' (default)      | Provides the name for the new subsystem block. By default the block is named 'filter'. To enter a name for the block, use the <i>propertyvalue</i> set to a string ' <i>blockname</i> '.  |
| MapCoeffstoPorts | 'off' (default) or 'on' | Specify whether to map the coefficients of the filter to the ports of the block.  |
| MapStates        | 'off' (default) or 'on' | Specifies whether to apply the current filter states to the realized model. This lets you save states from a filter object you may have used or configured in a specific way. The default setting of 'off' means the states are not transferred to the model. Setting the property to 'on' preserves the current filter states in the realized model. |
| OverwriteBlock   | 'off' or 'on'           | Specify whether to overwrite an existing block with the same name or create a new block.  |
| OptimizeZeros    | 'off' (default) or 'on' | Specify whether to remove zero-gain blocks.   |
| OptimizeOnes     | 'off' (default) or 'on' | Specify whether to replace unity-gain blocks with direct connections.   |

| Property Name       | Property Values  | Description   |
|---------------------|--|---|
| OptimizeNegOnes     | 'off' (default) or 'on'  | Specify whether to replace negative unity-gain blocks with a sign change at the nearest sum block.  |
| OptimizeDelayChains | 'off' (default) or 'on'  | Specify whether to replace cascaded chains of delay blocks with a single integer delay block to provide an equivalent delay.  |
| CoeffNames          | { 'Num' } (default FIR), { 'Num', 'Den' } (default direct form IIR), { 'Num', 'Den', 'g' } (default IIR SOS), { 'K' } (default form lattice) | Specify the coefficient variable names as string variables in a cell array. <code>MapCoeffsToPorts</code> must be set to 'on' for this property to apply.   |
| InputProcessing     | 'columnsaschannels' (default), 'elementsaschannels', or 'inherited'  | Specify frame-based ('columnsaschannels') or sample-based ('elementsaschannels') processing.<br><br>The Inherited (this choice will be removed - see release notes) option will be removed in a future release. |
| RateOption          | 'enforcesinglerate' (default) or 'allowmultirate'  | Specify how the block adjusts the rate at the output to accommodate the reduced number of samples. This parameter applies only when <code>InputProcessing</code> is 'columnsaschannels'.                        |

## Examples

Realize Simulink model of lowpass Butterworth filter:

```
Hd = fdesign.lowpass('N,F3dB',4,0.25);  
d = design(Hd,'butter');  
realizemdl(d);
```

Realize Simulink model with coefficients mapped to ports:

```
Hd = fdesign.lowpass('N,F3dB',4,0.25);  
d = design(Hd,'butter');  
%Realize Simulink model and export coefficients  
realizemdl(d,'MapCoeffsToPorts','on');
```

In this case, the filter is an IIR filter with a direct form II second-order sections structure. Setting `MapCoeffstoPorts` to `'on'` exports the numerator coefficients, the denominator coefficients, and the gains to the MATLAB workspace using the default variable names `Num`, `Den`, and `g`. Each column of `Num` and `Den` represents one second-order section. You can modify the filter coefficients directly in the MATLAB workspace providing tunability to the realized Simulink model.

## See Also

`block` | `design` | `fdesign`

## rc2ac

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

#### Compute Autocorrelation Sequence

Determine the autocorrelation sequence that corresponds to a given vector, `k`, of reflection coefficients and an initial zero-lag autocorrelation given by `r0`.

```
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, Steven M. *Modern Spectral Estimation*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

## See Also

ac2rc | poly2ac | rc2poly

## rc2is

Convert reflection coefficients to inverse sine parameters

### Syntax

```
isin = rc2is(k)
```

### Description

`isin = rc2is(k)` returns a vector of inverse sine parameters, `isin`, from a vector of reflection coefficients, `k`.

### Examples

#### Compute Inverse Sine Parameters

Define a vector, `k`, of reflection coefficients and determine the corresponding inverse sine parameters.

```
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, John R., John G. Proakis, and John H. L. Hansen. *Discrete-Time Processing of Speech Signals*. New York: Macmillan, 1993.

### See Also

`is2rc`

# rc2lar

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

### Log Area Ratio Parameters

Define a vector, `k`, of reflection coefficients and compute the log area ratio parameters.

```
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, John R., John G. Proakis, and John H. L. Hansen. *Discrete-Time Processing of Speech Signals*. New York: Macmillan, 1993.

## See Also

lar2rc

## rc2poly

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

#### Equivalent Prediction Filter Representation

Consider a lattice IIR filter given by a set of reflection coefficients. Find its equivalent prediction filter representation.

```
k = [0.3090 0.9800 0.0031 0.0082 -0.0082];
```

```
a = rc2poly(k)
```

```
a =
```

```
1.0000    0.6148    0.9899    0.0000    0.0032   -0.0082
```

## More About

### Algorithms

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**,  $\mathbf{a} = [\mathbf{a} + \mathbf{a}(i-1:-1:1)*\mathbf{k}(i) \ \mathbf{k}(i)]$ .

- 3 Implements  $\mathbf{a} = [1 \ \mathbf{a}]$ .

## References

[1] Kay, Steven M. *Modern Spectral Estimation*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

### See Also

ac2poly | latc2tf | latcfilt | poly2rc | rc2ac | rc2is | rc2lar | tf2latc

## rceps

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.

### More About

#### Algorithms

`rceps` is an 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. Schaffer, *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

# rcosdesign

Raised cosine FIR pulse-shaping filter design

## Syntax

```
b = rcosdesign(beta,span,sps)
b = rcosdesign(beta,span,sps,shape)
```

## Description

`b = rcosdesign(beta,span,sps)` returns the coefficients, `b`, that correspond to a square-root raised cosine FIR filter with rolloff factor specified by `beta`. The filter is truncated to `span` symbols, and each symbol period contains `sps` samples. The order of the filter, `sps*span`, must be even. The filter energy is 1.

`b = rcosdesign(beta,span,sps,shape)` returns a square-root raised cosine filter when you set `shape` to `'sqrt'` and a normal raised cosine FIR filter when you set `shape` to `'normal'`.

## Examples

### Design a Square-Root Raised Cosine Filter

Specify a rolloff factor of 0.25. Truncate the filter to 6 symbols and represent each symbol with 4 samples. Verify that `'sqrt'` is the default value of the `shape` parameter.

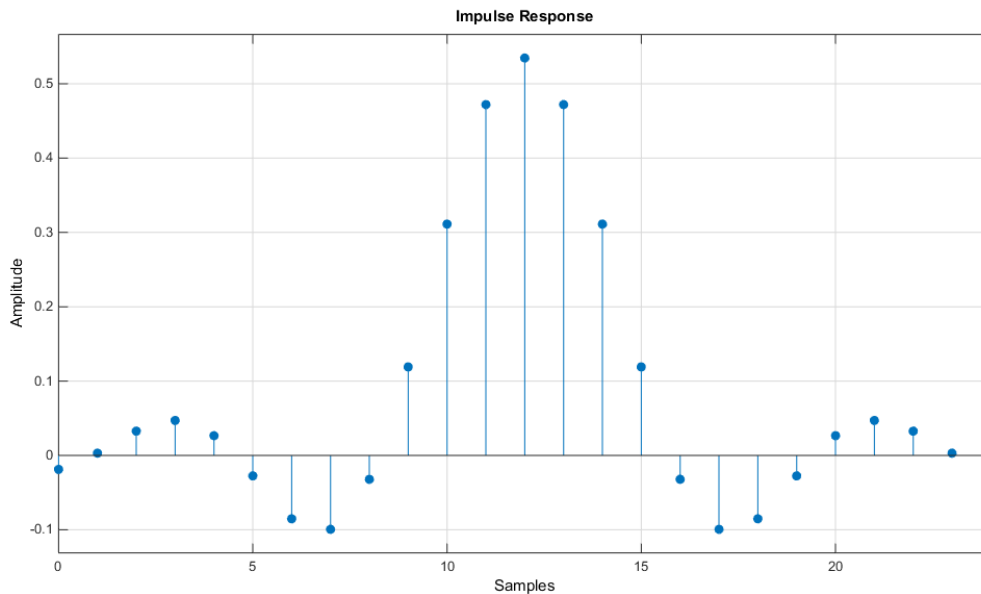
```
h = rcosdesign(0.25,6,4);
mx = max(abs(h-rcosdesign(0.25,6,4,'sqrt')))
```

`fvtool(h,'Analysis','impulse')`

```
mx =
```

```
0
```



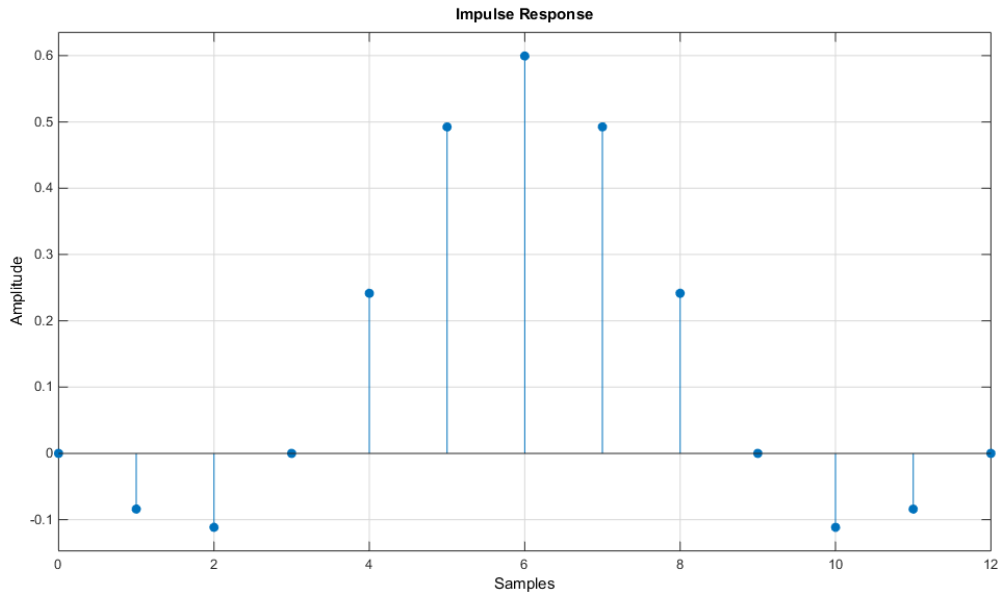


### Impulse Responses of Normal and Square-Root Raised Cosine Filters

Compare a normal raised cosine filter with a square-root cosine filter. An ideal (infinite-length) normal raised cosine pulse-shaping filter is equivalent to two ideal square-root raised cosine filters in cascade. Thus, the impulse response of an FIR normal filter should resemble that of a square-root filter convolved with itself.

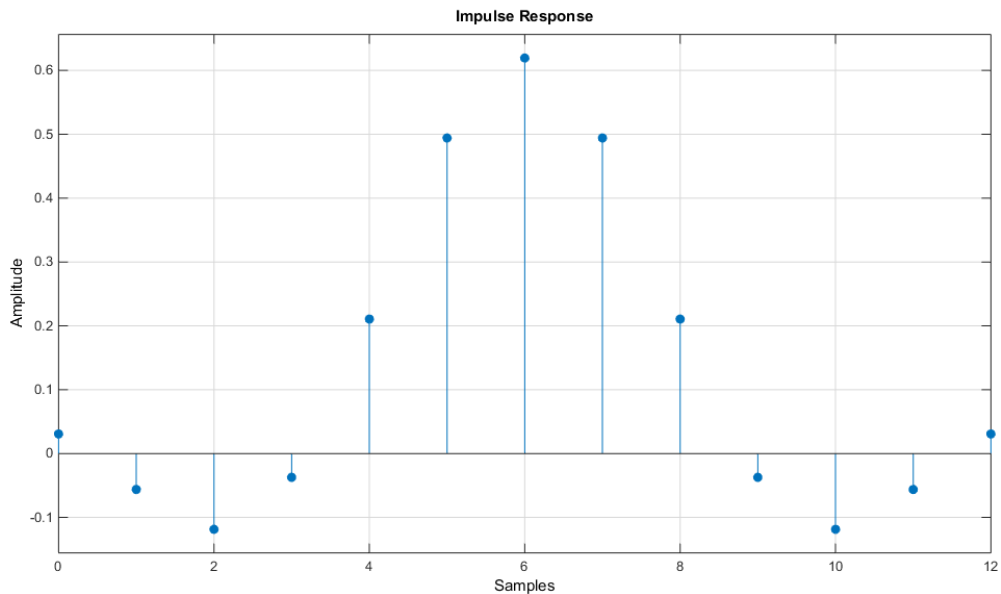
Create a normal raised cosine filter with rolloff 0.25. Specify that this filter span 4 symbols with 3 samples per symbol.

```
rf = 0.25;
span = 4;
sps = 3;
h1 = rcosdesign(rf,span,sps,'normal');
fvtool(h1,'impulse');
```



The normal filter has zero crossings at integer multiples of `sps`. It thus satisfies Nyquist's criterion for zero intersymbol interference. The square-root filter, however, does not.

```
h2 = rcosdesign(rf,span,sps,'sqrt');  
fvtool(h2,'impulse');
```

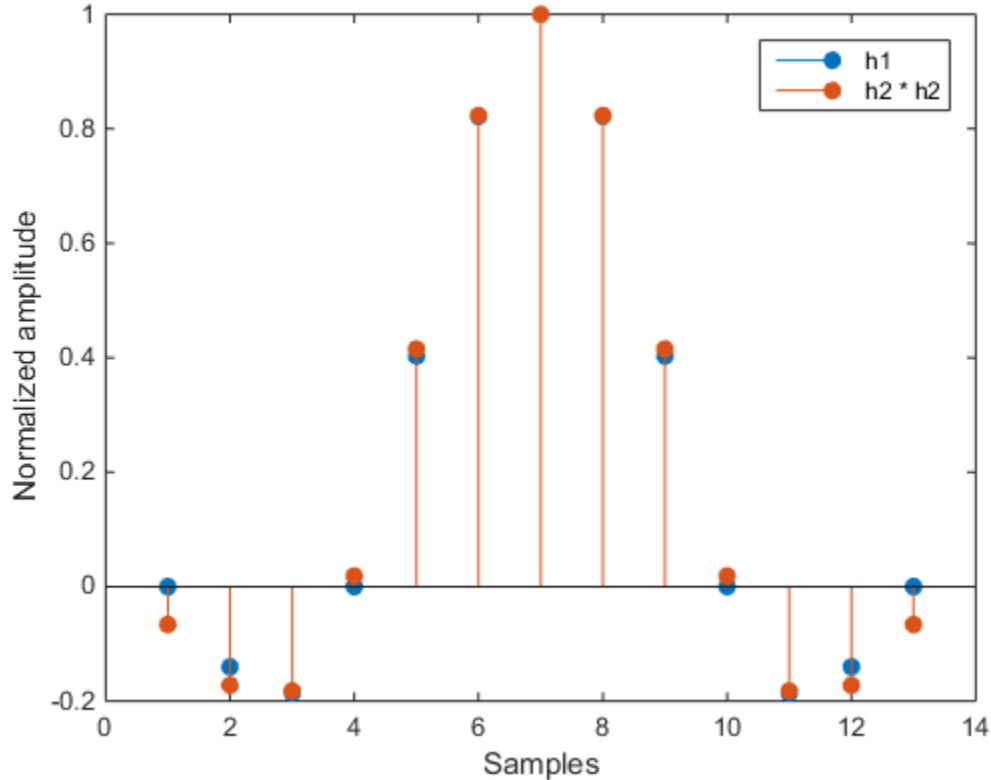


Convolve the square-root filter with itself. Truncate the impulse response outward from the maximum so it has the same length as  $h_1$ . Normalize the response using the maximum. Then, compare the convolved square-root filter to the normal filter.

```

h3 = conv(h2,h2);
p2 = ceil(length(h3)/2);
m2 = ceil(p2-length(h1)/2);
M2 = floor(p2+length(h1)/2);
ct = h3(m2:M2);
stem([h1/max(abs(h1));ct/max(abs(ct))],'filled')
xlabel('Samples'),ylabel('Normalized amplitude')
legend('h1','h2 * h2')

```



The convolved response does not coincide with the normal filter because of its finite length. Increase `span` to obtain closer agreement between the responses and better compliance with the Nyquist criterion.

### Pass a Signal through a Raised Cosine Filter

This example shows how to pass a signal through a square-root, raised cosine filter.

Specify the filter parameters.

```

rolloff = 0.25;    % Rolloff factor
span = 6;         % Filter span in symbols
sps = 4;          % Samples per symbol

```

Generate the square-root, raised cosine filter coefficients.

```
b = rcosdesign(rolloff, span, sps);
```

Create a vector of bipolar data.

```
d = 2*randi([0 1], 100, 1) - 1;
```

Upsample and filter the data for pulse shaping.

```
x = upfirdn(d, b, sps);
```

Add noise.

```
r = x + randn(size(x))*0.01;
```

Filter and downsample the received signal for matched filtering.

```
y = upfirdn(r, b, 1, sps);
```

### Interpolate and Decimate Using an RRC Filter

This example shows how to interpolate and decimate signals using square-root, raised cosine filters designed with the `rcosdesign` function. This example requires the Communications System Toolbox software.

Define the square-root, raised cosine filter parameters. Define the signal constellation parameters.

```
rolloff = 0.25; % Filter rolloff
span = 6;      % Filter span
sps = 4;       % Samples per symbol
M = 4;         % Size of the signal constellation
k = log2(M);   % Number of bits per symbol
```

Generate the coefficients of the square-root, raised cosine filter using the `rcosdesign` function.

```
rrcFilter = rcosdesign(rolloff, span, sps);
```

Generate 10,000 data symbols using the `randi` function.

```
data = randi([0 M-1], 10000, 1);
```

Apply PSK modulation to the data symbols. Because the constellation size is 4, the modulation type is QPSK.

```
modData = pskmod(data, M, pi/4);
```

Using the `upfirdn` function, upsample and filter the input data.

```
txSig = upfirdn(modData, rrcFilter, sps);
```

Convert the  $E_b/N_0$  to SNR and then pass the signal through an AWGN channel.

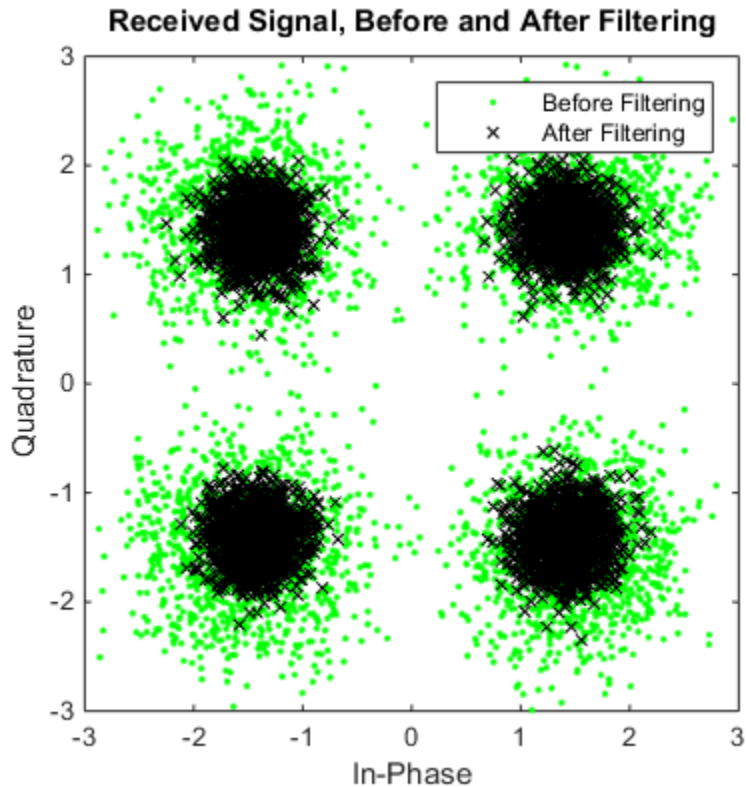
```
EbNo = 7;  
snr = EbNo + 10*log10(k) - 10*log10(sps);  
rxSig = txSig + awgn(txSig, snr, 'measured');
```

Filter and downsample the received signal. Remove a portion of the signal to compensate for the filter delay.

```
rxFilt = upfirdn(rxSig, rrcFilter, 1, sps);  
rxFilt = rxFilt(span+1:end-span);
```

Create a scatter plot of the modulated data using the first 5,000 symbols.

```
h = scatterplot(sqrt(sps)* ...  
    rxSig(1:sps*5000), ...  
    sps,0, 'g. ');  
hold on;  
scatterplot(rxFilt(1:5000),1,0, 'kx',h);  
title('Received Signal, Before and After Filtering');  
legend('Before Filtering','After Filtering');  
axis([-3 3 -3 3]); % Set axis ranges  
hold off;
```



## Input Arguments

### **beta** — Rolloff factor

real nonnegative scalar

Rolloff factor, specified as a real nonnegative scalar not greater than 1. The rolloff factor determines the excess bandwidth of the filter. Zero rolloff corresponds to a brick-wall filter and unit rolloff to a pure raised cosine.

Data Types: `double` | `single`

**span — Number of symbols**

positive scalar

Number of symbols, specified as a positive integer scalar.

Data Types: `double` | `single`

**sps — Samples per symbol**

positive integer scalar

Number of samples per symbol (oversampling factor), specified as a positive integer scalar.

Data Types: `double` | `single`

**shape — Shape of the raised cosine window**

'sqrt' (default) | 'normal'

Shape of the raised cosine window, specified as a string. Valid entries for shape are 'normal' and 'sqrt'.

Data Types: `char`

## Output Arguments

**b — FIR filter coefficients**

row vector

Raised cosine filter coefficients, returned as a row vector.

Data Types: `double` | `single`

## More About

### Tips

- If you have a license for Communications System Toolbox™ software, you can perform multirate raised cosine filtering with streaming behavior. To do so, use the System object filters, `comm.RaisedCosineTransmitFilter` and `comm.RaisedCosineReceiveFilter`.



## References

- [1] Tranter, William H., K. Sam Shanmugan, Theodore S. Rappaport, and Kurt L. Kosbar. *Principles of Communication Systems Simulation with Wireless Applications*. Upper Saddle River, NJ: Prentice Hall, 2004.

## See Also

gaussdesign

## rectpuls

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`

# rectwin

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.

## More About

### Algorithms

```
w = ones(L,1);
```

## References

[1] Oppenheim, A.V., and R.W. Schaffer. *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

Resample uniform or nonuniform data to new fixed rate

## 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, ___ )

y = resample(x,tx)
y = resample(x,tx,fs)
y = resample(x,tx,fs,p,q)
y = resample(x,tx,___,method)
[y,ty] = resample(x,tx,___ )
[y,ty,b] = resample(x,tx,___ )
```

## Description

`y = resample(x,p,q)` resamples the input sequence, `x`, at `p/q` times the original sample rate. If `x` is a matrix, then `resample` treats each column of `x` as an independent channel. `resample` applies an antialiasing FIR lowpass filter to `x` and compensates for the delay introduced by the filter.

`y = resample(x,p,q,n)` uses an antialiasing filter of order  $2 \times n \times \max(p,q)$ .

`y = resample(x,p,q,n,beta)` specifies the shape parameter of the Kaiser window used to design the lowpass filter.

`y = resample(x,p,q,b)` filters `x` using the filter coefficients specified in `b`.

`[y,b] = resample(x,p,q, ___ )` also returns the coefficients of the filter applied to `x` during the resampling.

`y = resample(x,tx)` resamples the values, `x`, of a signal sampled at the instants specified in vector `tx`. The function interpolates `x` linearly onto a vector of uniformly

spaced instants with the same endpoints and number of samples as `tx`. NaNs are treated as missing data and are ignored.

`y = resample(x,tx,fs)` uses a polyphase antialiasing filter to resample the signal at the uniform sample rate specified in `fs`.

`y = resample(x,tx,fs,p,q)` interpolates the input signal to a uniform grid with a sample spacing of  $(p/q)/fs$  and then filters the result to upsample it by `p` and downsample it by `q`. For best results, ensure that  $fs \times q/p$  is at least twice as large as the highest frequency component of `x`.

`y = resample(x,tx, __, method)` specifies the interpolation method along with any of the arguments from previous syntaxes in this group. The interpolation method can be 'linear', 'pchip', or 'spline'.

`[y,ty] = resample(x,tx, __)` returns in `ty` the instants that correspond to the resampled signal.

`[y,ty,b] = resample(x,tx, __)` returns in `b` the coefficients of the antialiasing filter.

---

**Note:** If `x` is not slowly varying, consider using `interp1` with the 'pchip' interpolation method.

---

## Examples

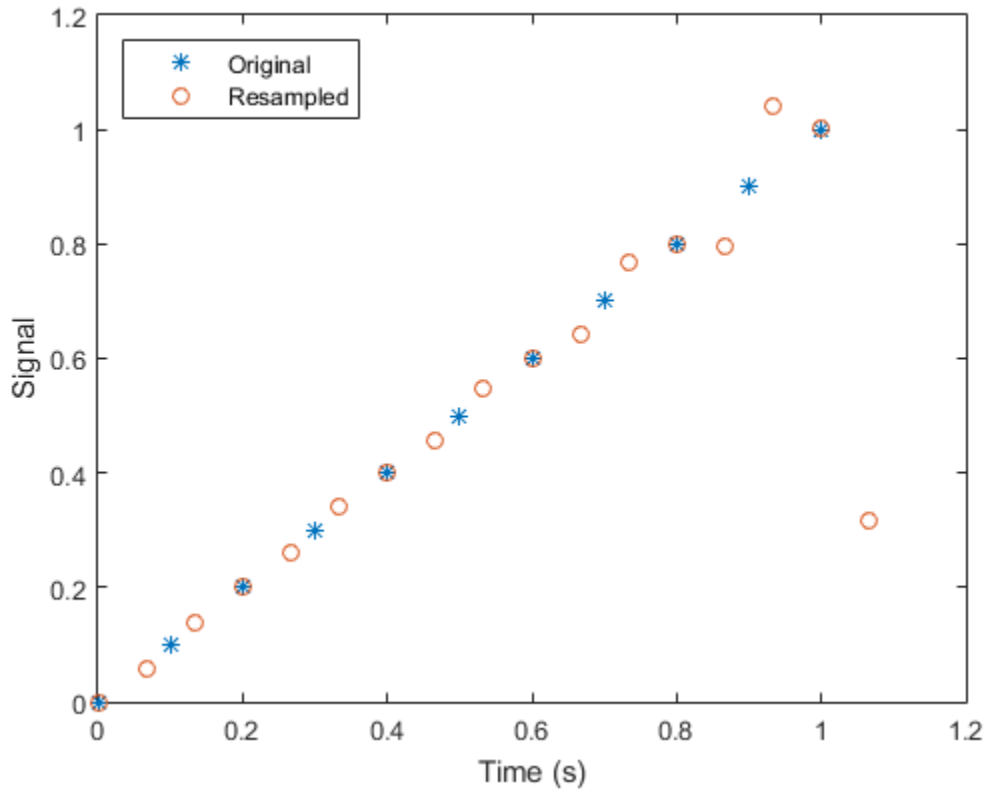
### Resample Linear Sequence

Resample a simple linear sequence at 3/2 the original rate of 10 Hz. Plot the original and resampled signals on the same figure.

```
fs = 10;
t1 = 0:1/fs:1;
x = t1;
y = resample(x,3,2);
t2 = (0:(length(y)-1))*2/(3*fs);

plot(t1,x,'*',t2,y,'o')
xlabel('Time (s)')
ylabel('Signal')
legend('Original','Resampled', ...
```

```
'Location', 'NorthWest')
```



When filtering, `resample` assumes that the input sequence, `x`, is zero before and after the samples it is given. Large deviations from zero at the endpoints of `x` can result in unexpected values for `y`.

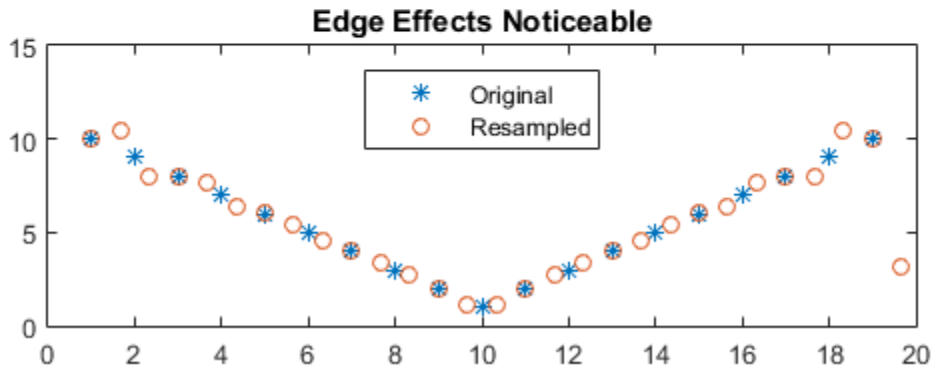
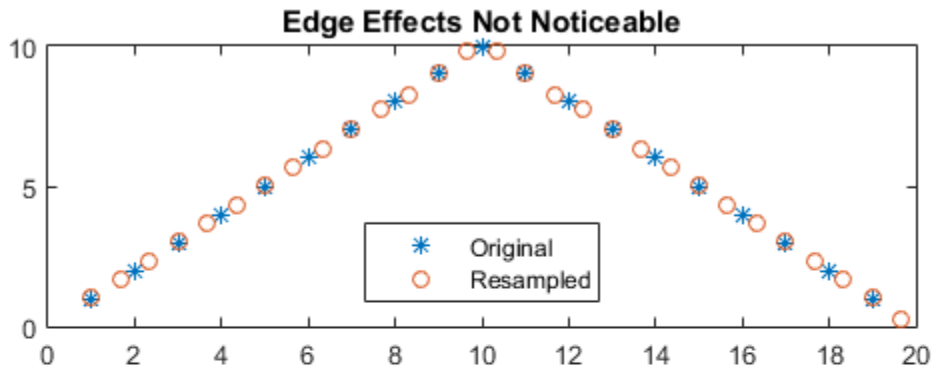
Show these deviations by resampling a triangular sequence and a vertically shifted version of the sequence with nonzero endpoints.

```
x = [1:10 9:-1:1;
     10:-1:1 2:10]';
y = resample(x,3,2);

subplot(2,1,1)
```

```
plot(1:19,x(:,1),'*', (0:28)*2/3 + 1,y(:,1),'o')
title('Edge Effects Not Noticeable')
legend('Original', 'Resampled', ...
       'Location', 'South')

subplot(2,1,2)
plot(1:19,x(:,2),'*', (0:28)*2/3 + 1,y(:,2),'o')
title('Edge Effects Noticeable')
legend('Original', 'Resampled', ...
       'Location', 'North')
```



### Resample Using Kaiser Window

Construct a sinusoidal signal. Specify a sample rate such that 16 samples correspond to exactly one signal period. Draw a stem plot of the signal. Overlay a stairstep graph for sample-and-hold visualization.

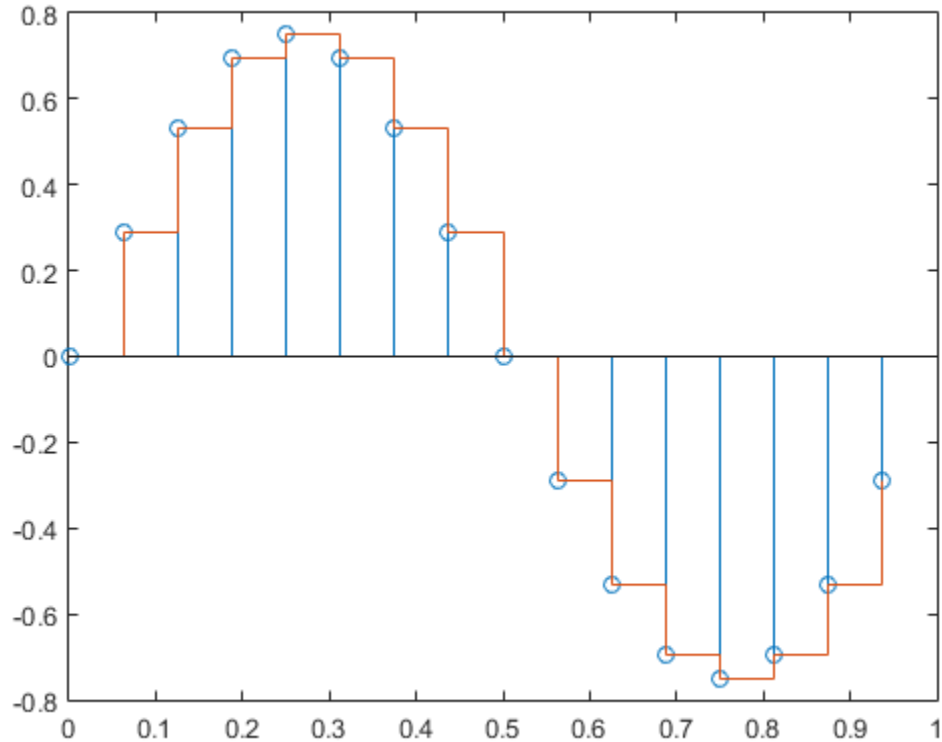
```
fs = 16;
t = 0:1/fs:1-1/fs;

x = 0.75*sin(2*pi*t);

stem(t,x)
hold on
stairs(t,x)
```



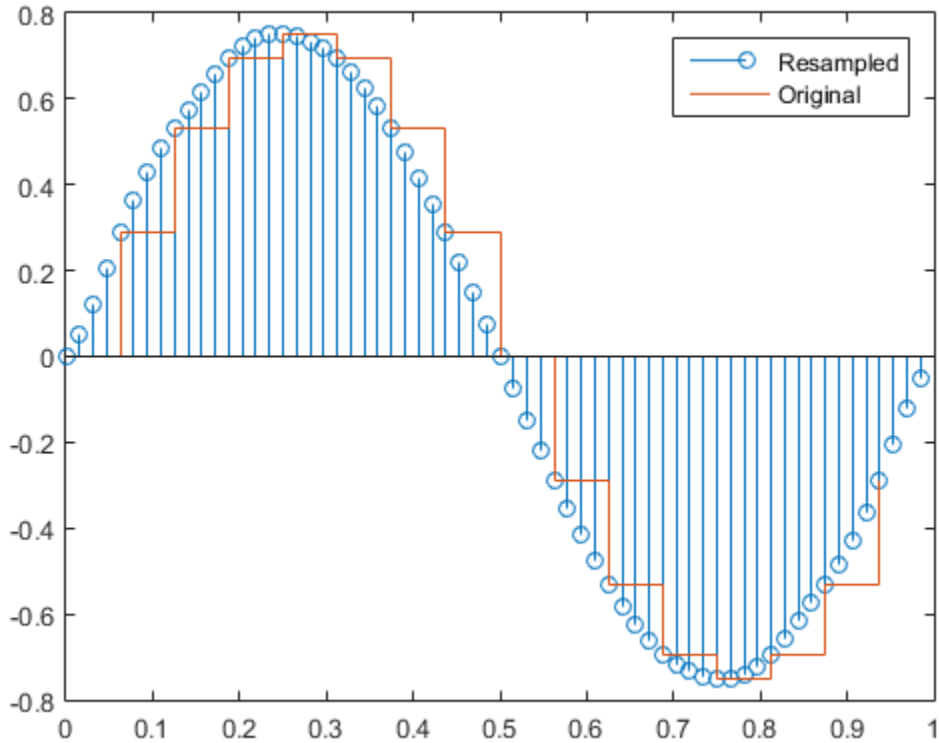
```
hold off
```



Use `resample` to upsample the signal by a factor of four. Use the default settings. Plot the result alongside the original signal.

```
ups = 4;  
dns = 1;  
  
fu = fs*ups;  
tu = 0:1/fu:1-1/fu;  
  
y = resample(x,ups,dns);
```

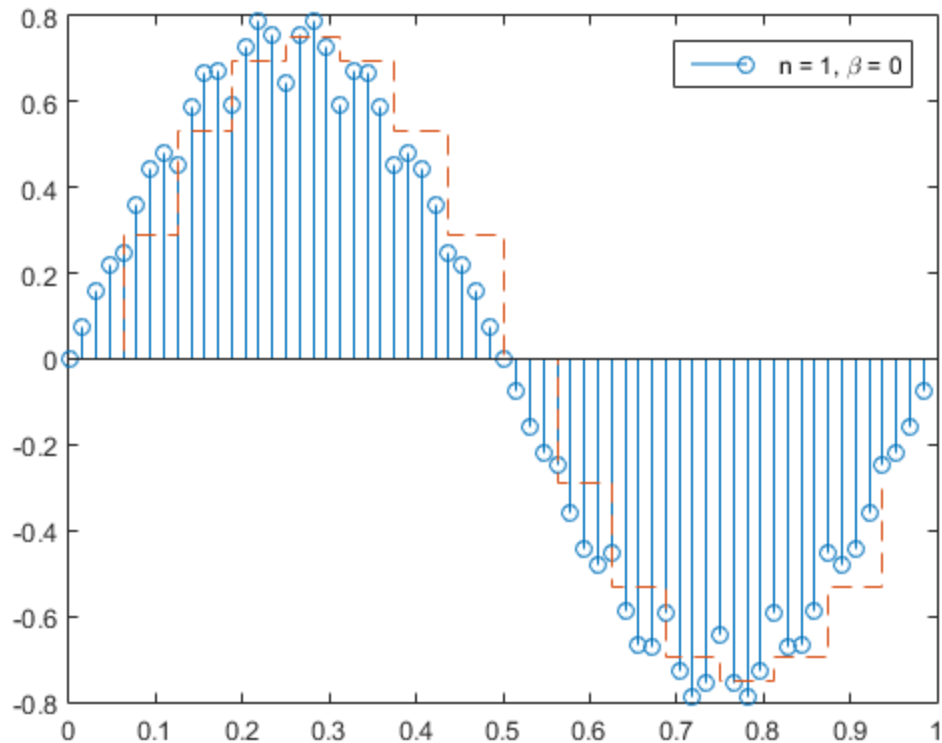
```
stem(tu,y)
hold on
stairs(t,x)
hold off
legend('Resampled','Original')
```



Repeat the calculation. Specify  $n = 1$  so that the antialiasing filter is of order  $2 \times 1 \times 4 = 8$ . Specify a shape parameter  $\beta = 0$  for the Kaiser window. Output the filter as well as the resampled signal.

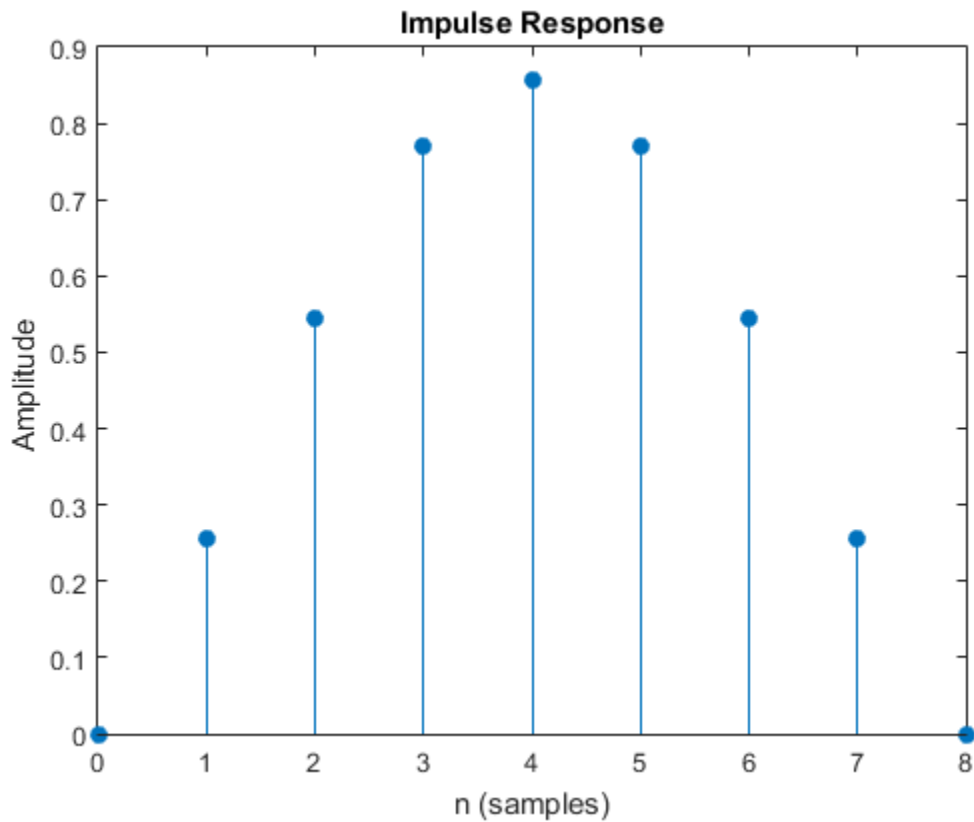
```
n = 1;
beta = 0;
```

```
[y,b] = resample(x,ups,dns,n,beta);  
  
stem(tu,y)  
hold on  
stairs(t,x,'--')  
hold off  
legend('n = 1, \beta = 0')
```



Verify that the filter has the expected order by plotting its impulse response.

```
impz(b)
```

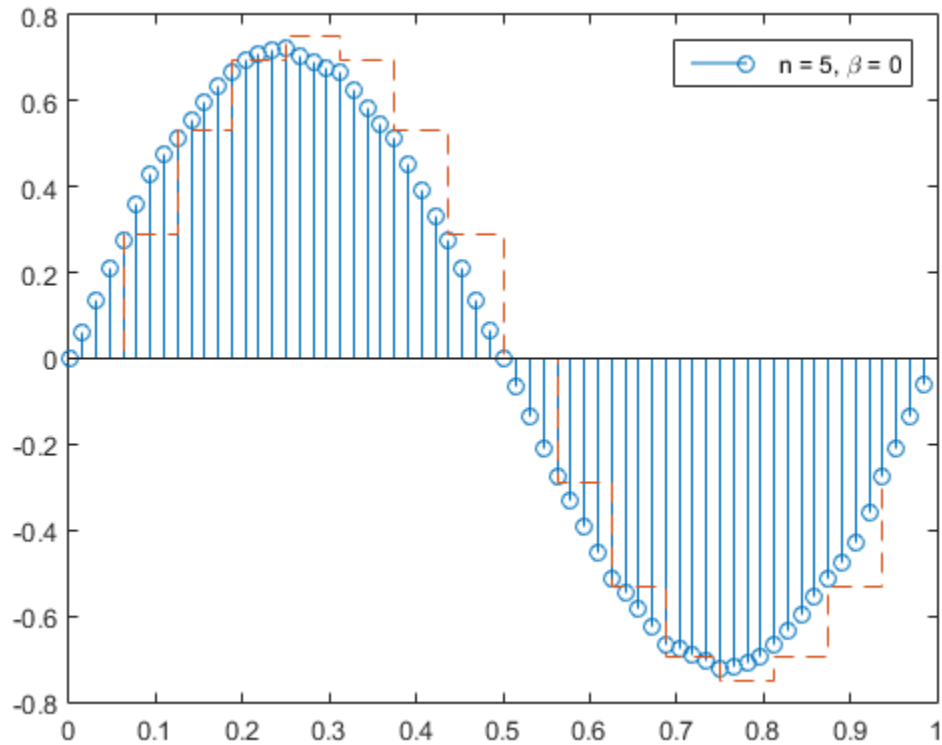


Increase  $n$  to 5 and leave  $\beta = 0$ . Verify that the filter is of order 40. Plot the resampled signal.

```
n = 5;
[y,b] = resample(x,ups,dns,n,beta);
fo = filtord(b)

stem(tu,y)
hold on
stairs(t,x,'--')
hold off
legend('n = 5, \beta = 0')
```

```
fo =
    40
```



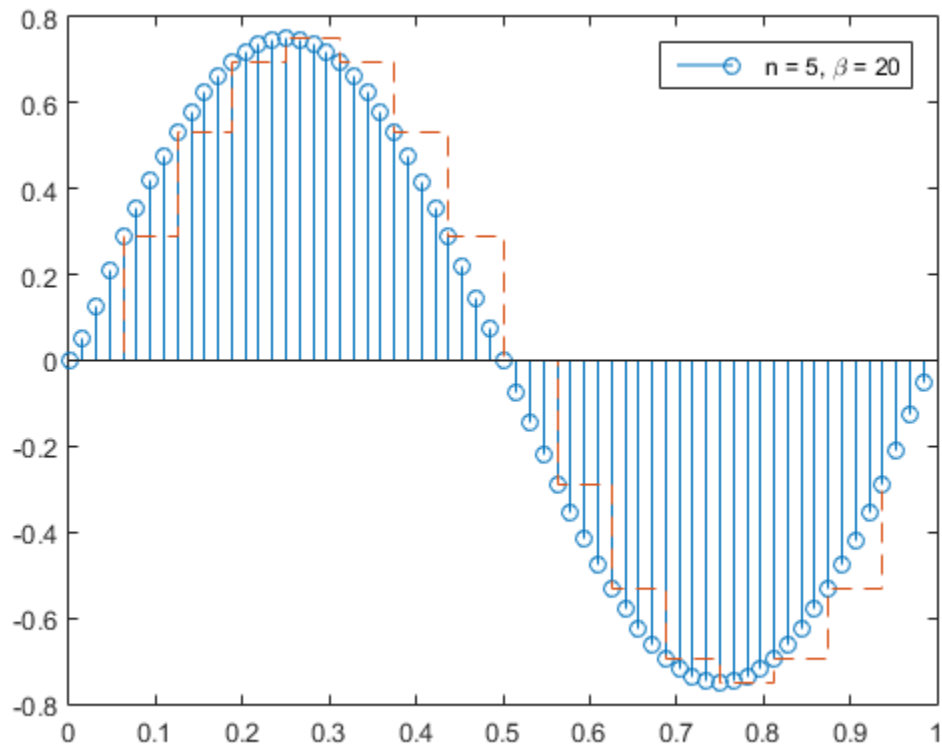
Leave the filter order at  $2 \times 5 \times 4 = 40$  and increase the shape parameter to  $\beta = 20$ .

```
beta = 20;
```

```
y = resample(x,ups,dns,n,beta);
```

```
stem(tu,y)
```

```
hold on
stairs(t,x,'--')
hold off
legend('n = 5, \beta = 20')
```



Decrease the filter order back to  $2 \times 1 \times 4 = 8$  and leave  $\beta = 20$ .

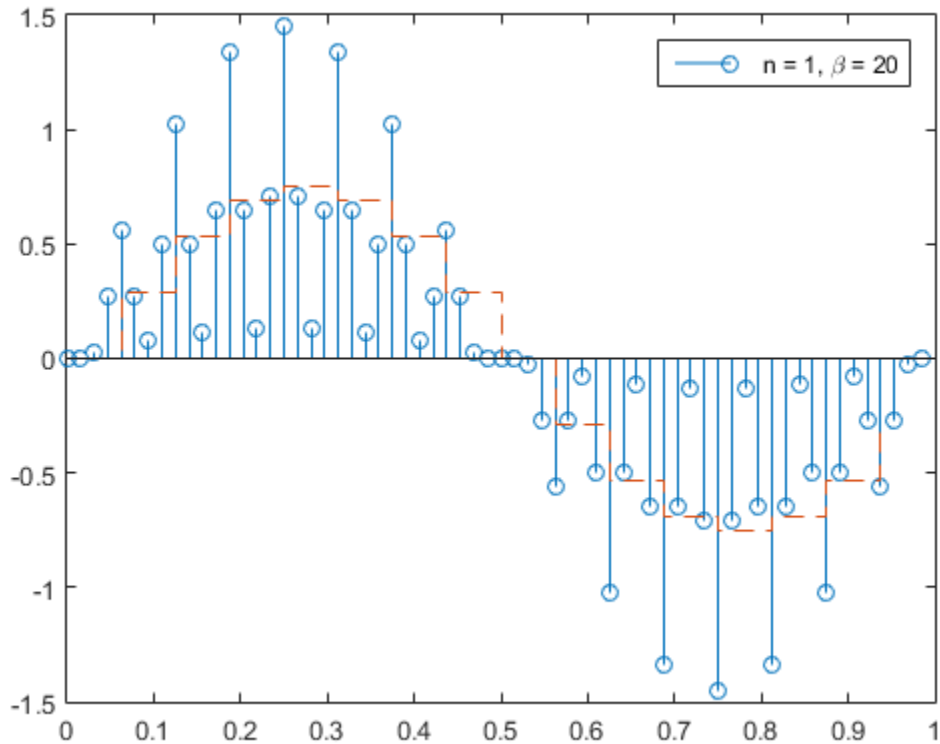
```
n = 1;
```

```
[y,b] = resample(x,ups,dns,n,beta);
```

```
stem(tu,y)
```

```
hold on
```

```
stairs(t,x,'--')  
hold off  
legend('n = 1, \beta = 20')
```



### Resample a Sinusoid

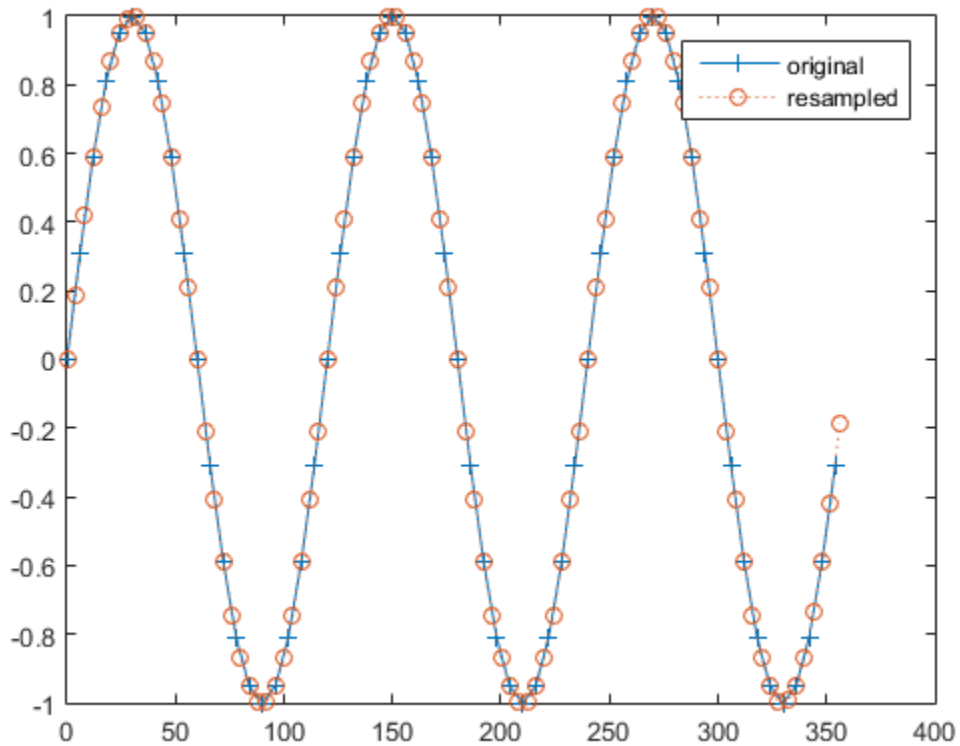
Generate 60 samples of a sinusoid and resample it at  $3/2$  the original rate. Display the original and resampled signals.

```
tx = 0:6:360-3;  
x = sin(2*pi*tx/120);
```

```
ty = 0:4:360-2;
```

```
[y by] = resample(x,3,2);

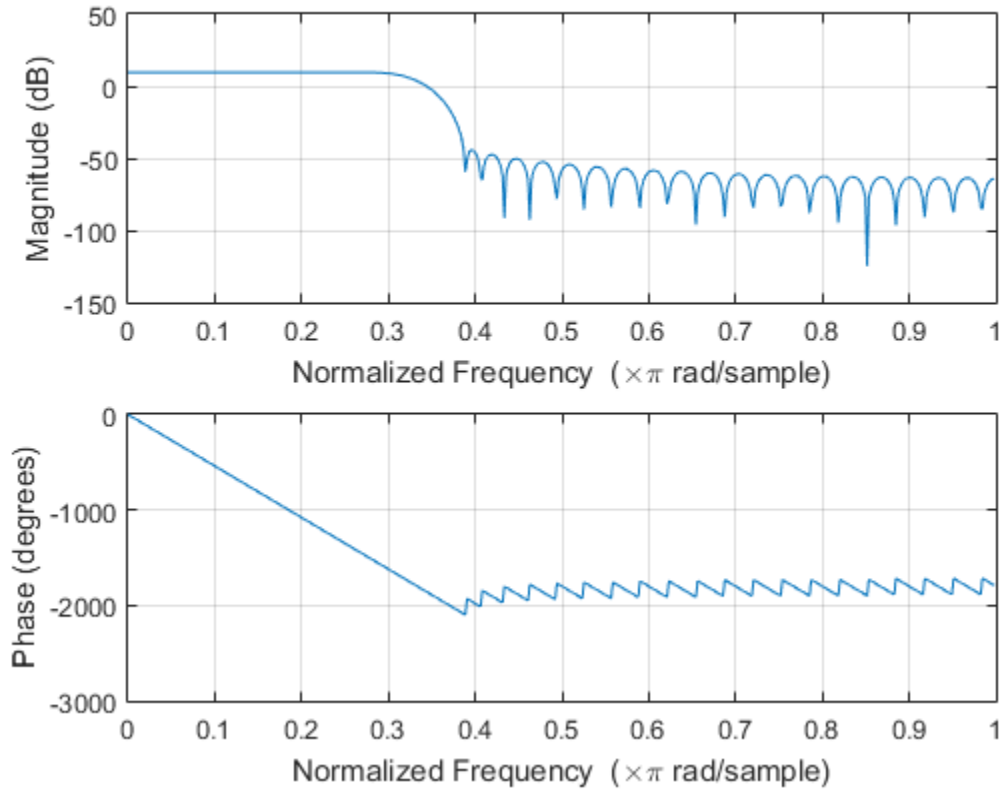
plot(tx,x,'+-',ty,y,'o:')
legend('original','resampled')
```



Plot the frequency response of the anti-aliasing filter.

```
freqz(by)
```

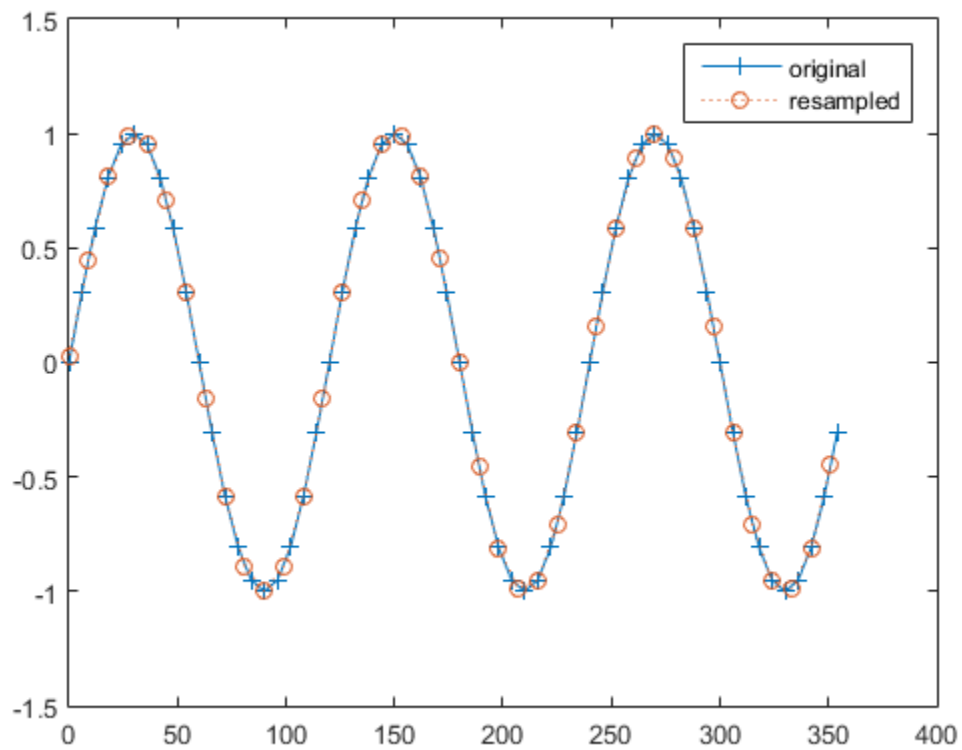




Resample the signal at  $2/3$  the original rate. Display the original signal and its resampling.

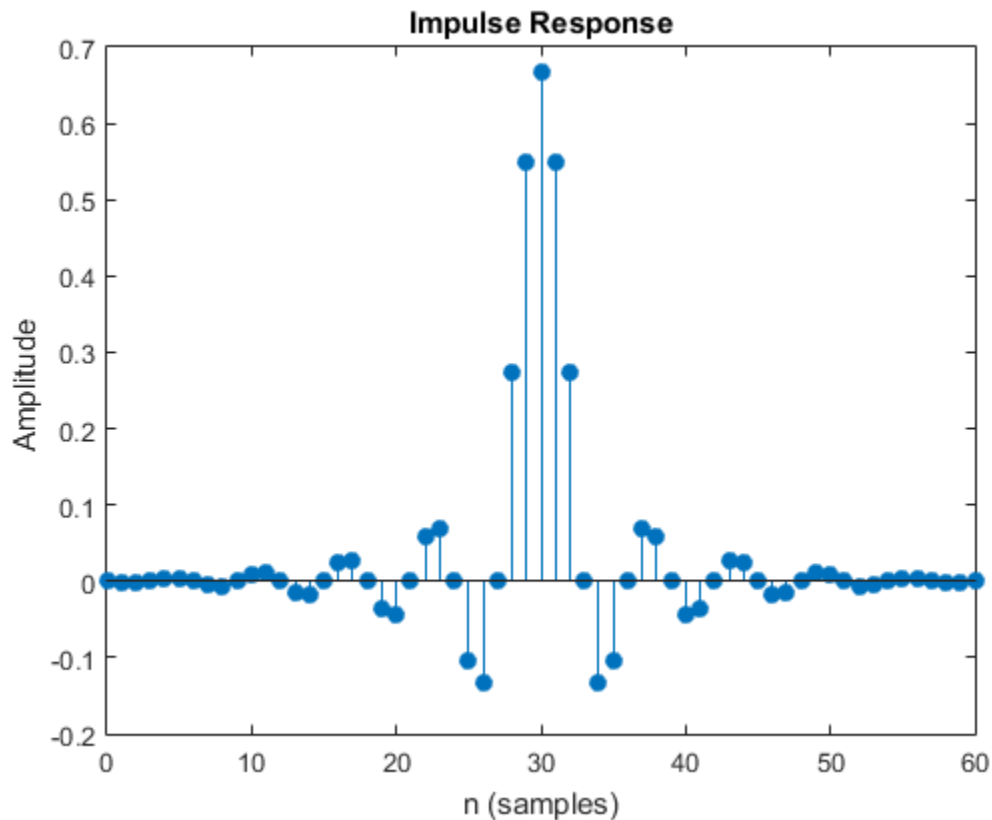
```
tz = 0:9:360-9;
[z bz] = resample(x,2,3);

plot(tx,x,'+ ',tz,z,'o:')
legend('original','resampled')
```



Plot the impulse response of the new lowpass filter.

```
impz(bz)
```



### Resample a Nonuniformly Sampled Data Set

Use the data recorded by Galileo Galilei in 1610 to determine the orbital period of Callisto, the outermost of Jupiter's four largest satellites.

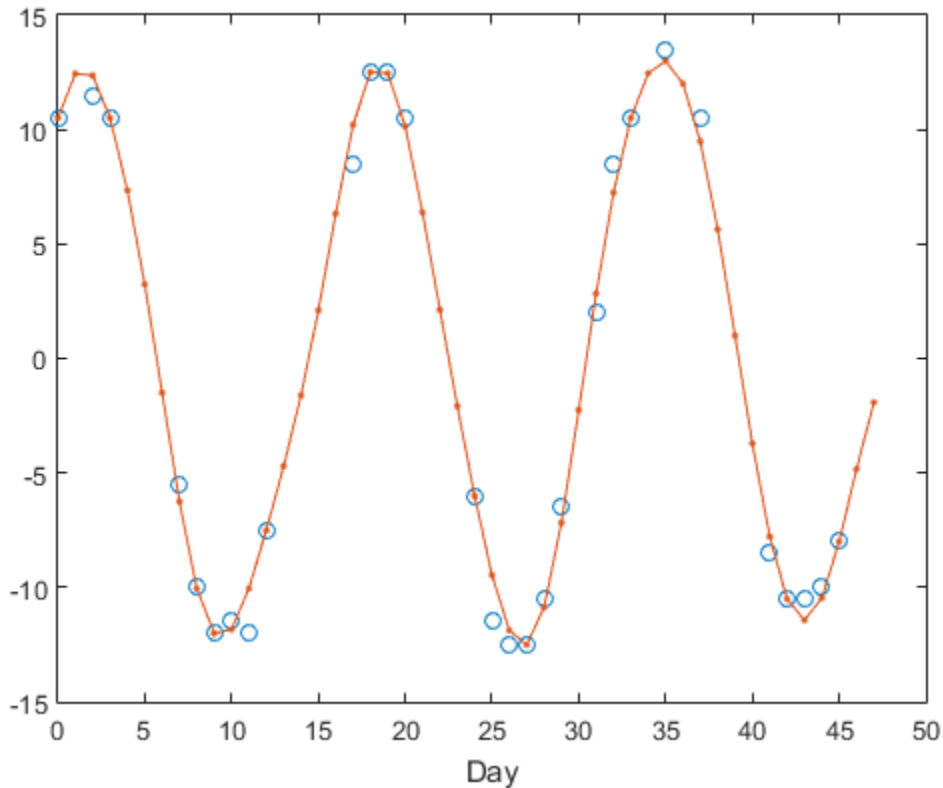
Galileo observed the satellites' motion for six weeks. The observations have several gaps because Jupiter was not visible on cloudy nights.

```
t = [0 2 3 7 8 9 10 11 12 17 18 19 20 24 25 26 27 28 29 31 32 33 35 37 ...
     41 42 43 44 45]';
```

```
yg = [10.5 11.5 10.5 -5.5 -10.0 -12.0 -11.5 -12.0 -7.5 8.5 12.5 12.5 ...
      10.5 -6.0 -11.5 -12.5 -12.5 -10.5 -6.5 2.0 8.5 10.5 13.5 10.5 -8.5 ...
      -10.5 -10.5 -10.0 -8.0]';
```

Resample the data onto a regular grid using a sample rate of one observation per day. Use a moderate upsampling factor of 3 to avoid overfitting.

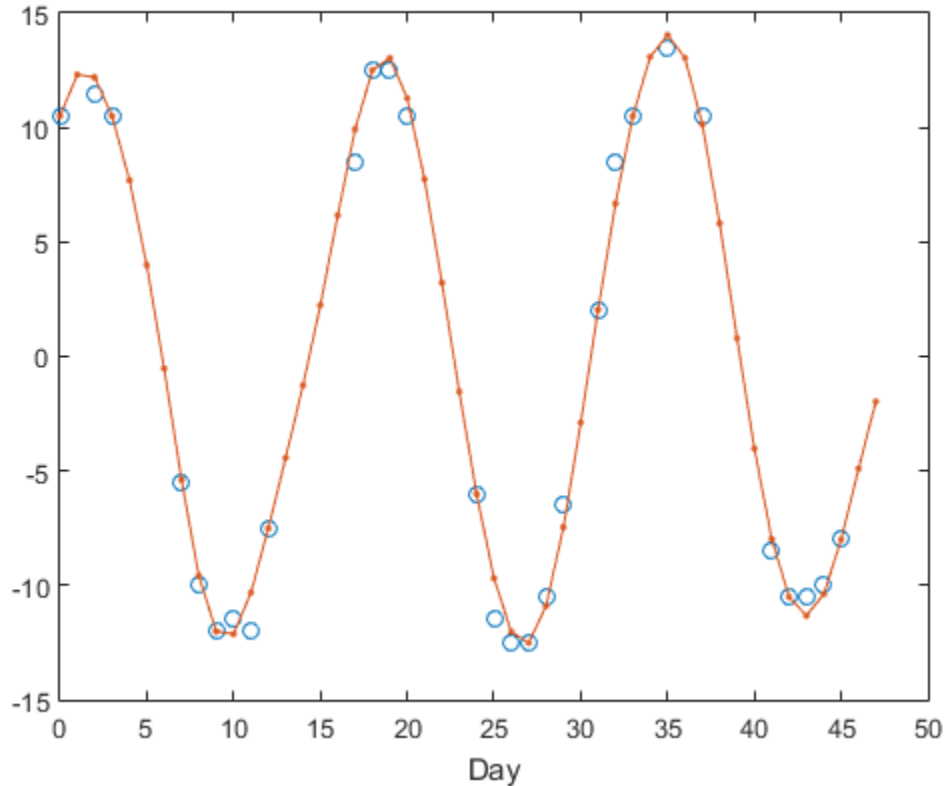
```
fs = 1;
[y,ty] = resample(yg,t,fs,3,1);
Plot the data and the resampled signal.
plot(t,yg,'o',ty,y,'.-')
```



Repeat the procedure using spline interpolation.

```
[ys,tys] = resample(yg,t,fs,3,1,'spline');
```

```
plot(t,yg,'o',tys,ys,'.-')
xlabel('Day')
```



Compute the periodogram power spectrum estimate of the uniformly spaced, linearly interpolated data. The signal peaks at the inverse of the orbital period.

```
[pxx,f] = periodogram(y,[],[],fs,'power');
[pk,i0] = max(pxx);
```

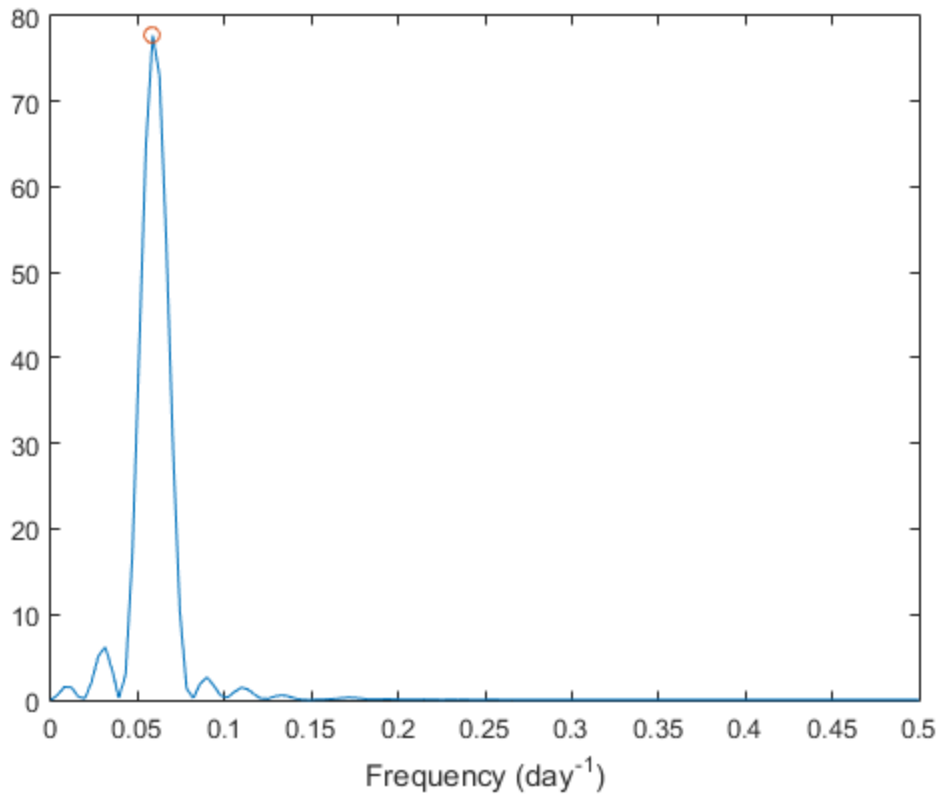
```
f0 = f(i0);
T0 = 1/f0
```

```
plot(f,pxx,f0,pk,'o')
```

```
xlabel('Frequency (day-1)')
```

```
T0 =
```

```
17.0667
```



- “Reconstruct a Signal from Irregularly Sampled Data”

## Input Arguments

### **x** — Input signal

vector | matrix

Input signal, specified as a vector or matrix. If  $x$  is a matrix, then its columns are treated as independent channels.  $x$  can contain NaNs. NaNs are treated as missing data and are excluded from the resampling.

Example: `cos(pi/4*(0:159))+randn(1,160)` is a single-channel row-vector signal.

Example: `cos(pi./[4;2]*(0:159))'+randn(160,2)` is a two-channel signal.

Data Types: `single` | `double`

### **p, q** — Resampling factors

positive integers

Resampling factors, specified as positive integers.

Data Types: `single` | `double`

### **n** — Neighbor term number

10 (default) | positive integer

Neighbor term number, specified as a positive integer. If  $n = 0$ , `resample` performs nearest-neighbor interpolation. The length of the antialiasing FIR filter is proportional to  $n$ . Larger values of  $n$  provide better accuracy at the expense of more computation time.

Data Types: `single` | `double`

### **beta** — Shape parameter of Kaiser window

5 (default) | positive real scalar

Shape parameter of Kaiser window, specified as a positive real scalar. Increasing  $\beta$  widens the mainlobe of the window used to design the antialiasing filter and decreases the amplitude of the window's sidelobes.

Data Types: `single` | `double`

### **b** — FIR filter coefficients

vector

FIR filter coefficients, specified as a vector. By default, `resample` designs the filter using `firls` with a Kaiser window. When compensating for the delay, `resample` assumes `b` has odd length and linear phase.

Example: `fir1(4,0.5)` specifies a 4th-order lowpass filter with normalized cutoff frequency  $0.5\pi$  rad/sample.

Data Types: `single` | `double`

**tx — Time instants**

nonnegative real vector

Time instants, specified as a nonnegative real vector. `tx` must increase monotonically but need not be uniformly spaced. `tx` can contain NaNs. NaNs are treated as missing data and are excluded from the resampling.

Data Types: `single` | `double`

**fs — Sample rate**

positive scalar

Sample rate, specified as a positive scalar. The sample rate is the number of samples per unit time. If the unit of time is seconds, then the sample rate is in Hz.

Data Types: `single` | `double`

**method — Interpolation method**

'linear' (default) | 'pchip' | 'spline'

Interpolation method, specified as one of 'linear', 'pchip', or 'spline':

- 'linear' — Linear interpolation.
- 'pchip' — Shape-preserving piecewise cubic interpolation.
- 'spline' — Spline interpolation using not-a-knot end conditions.

See the `interp1` reference page for more information.

Data Types: `char`

## Output Arguments

**y — Resampled signal**

vector | matrix



Resampled signal, returned as a vector or matrix. If  $x$  is a signal of length  $N$  and you specify  $p,q$ , then  $y$  is of length  $[N \times p/q]$ .

**b — FIR filter coefficients**

vector

FIR filter coefficients, returned as a vector.

Data Types: `single` | `double`

**ty — Output instants**

nonnegative real vector

Output instants, returned as a nonnegative real vector.

Data Types: `single` | `double`

## More About

### Algorithms

`resample` performs an FIR design using `firls`, normalizes the result to account for the processing gain of the window, and then implements a rate change using `upfirdn`.

### See Also

`decimate` | `downsample` | `firls` | `interp` | `interp1` | `intfilt` | `kaiser` | `mfilt` | `spline` | `upfirdn` | `upsample`

## residuez

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  $z$ .

$$B(z) = b_0 + b_1 z^{-1} + b_2 z^{-2} + \dots + b_m z^{-m}$$

$$A(z) = a_0 + a_1 z^{-1} + a_2 z^{-2} + \dots + a_n z^{-n}$$

If there are no multiple roots and  $a > n - 1$ ,

$$\frac{B(z)}{A(z)} = \frac{r(1)}{1 - p(1)z^{-1}} + \dots + \frac{r(n)}{1 - p(n)z^{-1}} + k(1) + k(2)z^{-1} + \dots + k(m - n + 1)z^{-(m-n)}$$

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

$$n = \text{length}(a) - 1 = \text{length}(r) = \text{length}(p)$$

The direct term coefficient vector **k** is empty if  $\text{length}(b)$  is less than  $\text{length}(a)$ ; otherwise:

$$\text{length}(k) = \text{length}(b) - \text{length}(a) + 1$$

If  $p(j) = \dots = p(j+s-1)$  is a pole of multiplicity **s**, then the expansion includes terms of the form

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

`[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`.

## More About

### Algorithms

`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  $\text{length}(b) > \text{length}(a) - 1$ .
- The poles using `p = roots(a)`.
- Any repeated poles, reordering the poles according to their multiplicities.
- The residue for each nonrepeating pole  $p_j$  by multiplying  $b(z)/a(z)$  by  $1/(1 - p_j z^{-1})$  and evaluating the resulting rational function at  $z = p_j$ .

- 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.

$$\frac{1}{1 - p_j z^{-1}}, \frac{1}{(1 - p_j z^{-1})^2}, \dots, \frac{1}{(1 - p_j z^{-1})^{s_j}}$$

The vector  $h$  and matrices  $S1$  and  $S2$  have  $n + \text{tra}$  rows, where  $n$  is the total number of roots and the internal parameter  $\text{tra}$ , set to 1 by default, determines the degree of over-determination of the system of equations.

## References

- [1] Oppenheim, Alan V., Ronald W. Schaffer, and John R. Buck. *Discrete-Time Signal Processing*. 2nd Ed. Upper Saddle River, NJ: Prentice Hall, 1999.

## See Also

convmtx | roots | ss2tf | tf2ss | tf2zp | tf2zpk | zp2ss | deconv | poly | prony | residue

## risetime

Rise time of positive-going bilevel waveform transitions

### Syntax

```
R = risetime(X)
R = risetime(X,FS)
R = risetime(X,T)
[R,LT,UT] = risetime(...)
[R,LT,UT,LL,UL] = risetime(...)
[...] = risetime(...,Name,Value)
risetime(...)
```

### Description

`R = risetime(X)` returns a vector, `R`, containing the time each transition of the input bilevel waveform, `X`, takes to cross from the 10% to 90% reference levels. To determine the transitions, `risetime` estimates the state levels of the input waveform by a histogram method. `risetime` identifies all regions that cross the upper-state boundary of the low state and the lower-state boundary of the high state. The low-state and high-state boundaries are expressed as the state level plus or minus a multiple of the difference between the state levels. See “State-Level Tolerances” on page 1-1292. Because `risetime` uses interpolation, `R` may contain values that do not correspond to sampling instants of the bilevel waveform, `X`.

`R = risetime(X,FS)` specifies the sampling frequency in hertz. The sampling frequency determines the sample instants corresponding to the elements in `X`. The first sample instant in `X` corresponds to  $t=0$ . Because `risetime` uses interpolation, `R` may contain values that do not correspond to sampling instants of the bilevel waveform, `X`.

`R = risetime(X,T)` specifies the sample instants, `T`, as a vector with the same number of elements as `X`.

`[R,LT,UT] = risetime(...)` returns vectors, `LT` and `UT`, whose elements correspond to the time instants where `X` crosses the lower- and upper-percent reference levels.

`[R,LT,UT,LL,UL] = risetime(...)` returns the levels, LL and UL, that correspond to the lower- and upper-percent reference levels.

`[...] = risetime(...,Name,Value)` returns the rise times with additional options specified by one or more Name,Value pair arguments.

`risetime(...)` plots the signal and darkens the regions of each transition where rise time is computed. The plot marks the lower and upper crossings and the associated reference levels. The state levels and the corresponding associated lower- and upper-state boundaries are also plotted.

## Input Arguments

### **X**

Bilevel waveform. X is a real-valued row or column vector.

### **FS**

Sample rate in hertz.

### **T**

Vector of sample instants. The length of T must equal the length of the bilevel waveform, X.

## Name-Value Pair Arguments

### **'PercentReferenceLevels'**

Reference levels as a percentage of the waveform amplitude. The low-state level is defined to be 0 percent. The high-state level is defined to be 100 percent. The value of 'PercentReferenceLevels' is a two-element real row vector whose elements correspond to the lower and upper percent reference levels.

**Default:** [10 90]

### **'StateLevels'**

Low- and high-state levels. Specifies the levels to use for the low- and high-state levels as a 2-element real row vector. The first element is the low-state level. The second element is the high-state level.

**'Tolerance'**

Tolerance levels (lower- and upper-state boundaries) expressed as a percentage. See “State-Level Tolerances” on page 1-1292.

**Default:** 2

## Output Arguments

**R**

Rise times. R is a vector containing the duration of each positive-going transition. If you specify the sampling rate, FS, or the sampling instants, T, rise times are in seconds. If you do not specify a sampling rate, or sampling instants, rise times are in samples.

**LT**

Instants when positive-going transition crosses the lower-reference level. By default, the lower reference level is the 10% reference level. The upper reference level is the 90% reference level. You can change the default reference levels by specifying the 'PercentReferenceLevels' name-value pair.

**UT**

Instants when positive-going transition crosses the upper-reference level. By default, the lower reference level is the 10% reference level. The upper reference level is the 90% reference level. You can change the default reference levels by specifying the 'PercentReferenceLevels' name-value pair.

**LL**

Lower reference level in waveform amplitude units. LL is a vector containing the waveform value corresponding to the lower reference level in each positive-going transition. By default, the lower reference level is the 10% reference level. You can change the default reference levels by specifying the 'PercentReferenceLevels' name-value pair.

**UL**

Upper reference level in waveform amplitude units. LL is a vector containing the waveform value corresponding to the upper reference level in each positive-going

transition. By default, the upper reference level is the 90% reference level. You can change the default reference levels by specifying the 'PercentReferenceLevels' name-value pair.

## Examples

### Rise Time in a Bilevel Waveform

Determine the rise time in samples for a 2.3 V clock waveform.

Load the 2.3 V clock data. Determine the rise time in samples. Use the default [10 90] percent reference levels.

```
load('transitionex.mat','x');  
R = risetime(x);
```

The rise time is less than 1, indicating that the transition occurred in a fraction of a sample.

### Rise Time with 20% and 80% Reference Levels

Determine the rise time in a 2.3 V clock waveform sampled at 4 MHz. Compute the rise time using the 20% and 80% reference levels.

Load the 2.3 V clock data with sampling instants. Plot the waveform.

```
load('transitionex.mat','x','t');  
plot(t,x);
```

Determine the rise time using the 20% and 80% reference levels.

```
R = risetime(x,'PercentReferenceLevels',[20 80]);
```

### Rise Time, Reference-Level Instants, and Reference Levels

Determine the rise time, reference-level instants, and reference levels in a 2.3 V clock waveform sampled at 4 MHz.

Load the 2.3 V clock waveform along with the sampling instants.

```
load('transitionex.mat','x','t');
```

Determine the rise time, reference-level instants, and reference levels.



```
[R,LT,UT,LL,UL] = risetime(x,t);
```

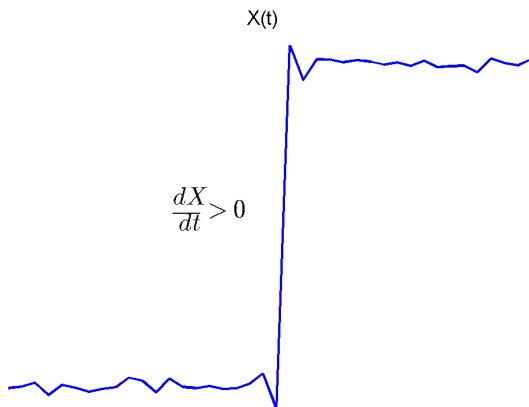
Plot the waveform in microseconds with the lower- and upper-reference levels and reference-level instants. Show that the rise time is the difference between the upper- and lower-reference level instants.

```
plot(t.*1e6,x);
xlabel('microseconds'); ylabel('Volts');
hold on; grid on;
plot(LT.*1e6,LL,'ro','markerfacecolor',[1 0 0]);
plot(UT.*1e6,UL,'ro','markerfacecolor',[1 0 0]);
fprintf('Rise time is %1.4f microseconds.\n',(UT-LT)*1e6)
```

## More About

### Positive-Going Transition

A *positive-going transition* in a bilevel waveform is a transition from the low-state level to the high-state level. A positive-polarity (positive-going) pulse has a terminating state more positive than the originating state. If the waveform is differentiable in the neighborhood of the transition, an equivalent definition is a transition with a positive first derivative. The following figure shows a positive-going transition.



In the preceding figure, the amplitude values of the waveform do not appear because a positive-going transition does not depend on the actual waveform values. A positive-going transition is defined by the direction of the transition.

### Percent Reference Levels

If  $S_1$  is the low state,  $S_2$  is the high state, and  $U$  is the *upper*-percent reference level. The waveform value corresponding to the upper percent reference level is

$$S_1 + \frac{U}{100}(S_2 - S_1)$$

If  $L$  is the *lower*-percent reference level, the waveform value corresponding to the lower percent reference level is

$$S_1 + \frac{L}{100}(S_2 - S_1)$$

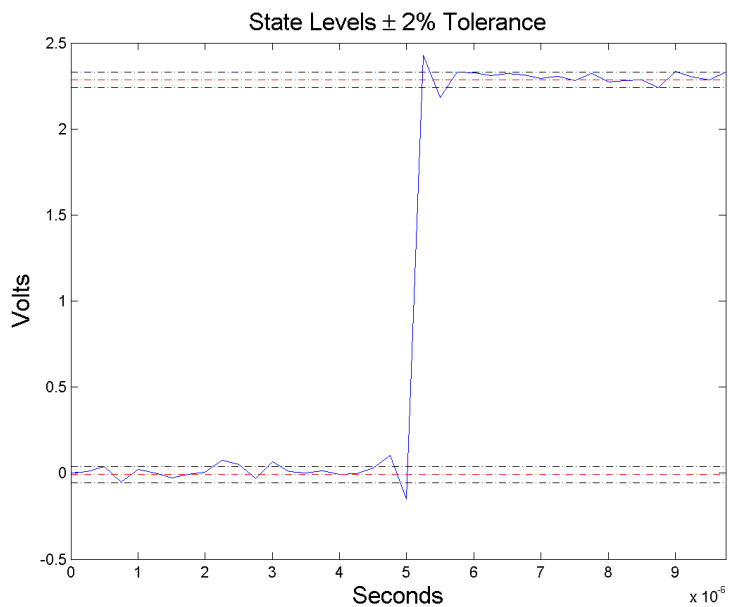
### State-Level Tolerances

Each state level can have associated lower- and upper-state boundaries. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference between the high state and low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the  $\alpha\%$  tolerance region for the low state is defined as

$$S_1 \pm \frac{\alpha}{100}(S_2 - S_1)$$

where  $S_1$  is the low-state level and  $S_2$  is the high-state level. Replace the first term in the equation with  $S_2$  to obtain the  $\alpha\%$  tolerance region for the high state.

The following figure illustrates lower and upper 2% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The red dashed lines indicate the estimated state levels.



## References

- [1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003, pp. 15–17.

## See Also

falltime | slewrate | statelevels

# rlevinson

Reverse Levinson-Durbin recursion

## Syntax

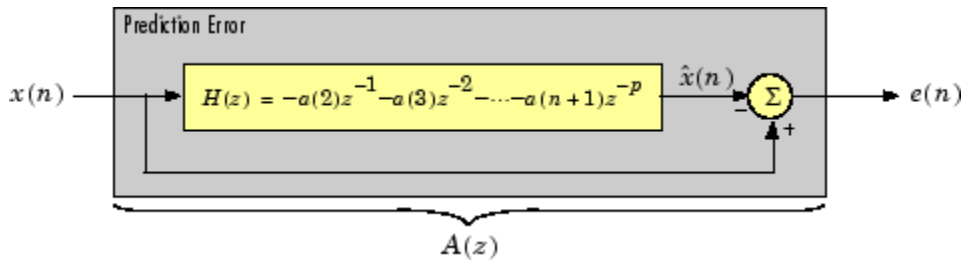
```
r = rlevinson(a,efinal)
[r,u] = rlevinson(a,efinal)
[r,u,k] = rlevinson(a,efinal)
[r,u,k,e] = rlevinson(a,efinal)
```

## Description

The reverse Levinson-Durbin recursion implements the step-down algorithm for solving the following symmetric Toeplitz system of linear equations for  $r$ , where  $r = [r(1) \ Lr(p+1)]$  and  $r(i)^*$  denotes the complex conjugate of  $r(i)$ .

$$\begin{bmatrix} r(1) & r(2)^* & \dots & r(p)^* \\ r(2) & r(1) & \dots & r(p-1)^* \\ \vdots & \ddots & \ddots & \vdots \\ r(p) & \dots & r(2) & r(1) \end{bmatrix} \begin{bmatrix} a(2) \\ a(3) \\ \vdots \\ a(p+1) \end{bmatrix} = \begin{bmatrix} -r(2) \\ -r(3) \\ \vdots \\ -r(p+1) \end{bmatrix}$$

`r = rlevinson(a,efinal)` solves the above system of equations for  $r$  given vector  $a$ , where  $a = [1 \ a(2) \ L \ a(p+1)]$ . 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  $\mathbf{a}$  represents the polynomial coefficients of this prediction error filter in descending powers of  $z$ .

$$A(z) = 1 + a(2)z^{-1} + \dots + a(n+1)z^{-p}$$

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

$$R = \begin{bmatrix} r(1) & r(2)^* & \dots & r(p)^* \\ r(2) & r(1) & \dots & r(p-1)^* \\ \vdots & \ddots & \ddots & \vdots \\ r(p) & \dots & r(2) & r(1) \end{bmatrix}$$

and  $E$  is a diagonal matrix with elements returned in output  $\mathbf{e}$  (see below). This decomposition permits the efficient evaluation of the inverse of the autocorrelation matrix,  $R^{-1}$ .

Output matrix  $\mathbf{u}$  contains the prediction filter polynomial,  $\mathbf{a}$ , 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  $j$ th coefficient of the  $i$ th 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  $\mathbf{a}$ .

`[r,u,k] = rlevinson(a,efinal)` returns a vector  $\mathbf{k}$  of length  $(p+1)$  containing the reflection coefficients. The reflection coefficients are the conjugates of the values in the first row of  $\mathbf{u}$ .

$$\mathbf{k} = \text{conj}(u(1,2:\text{end}))$$

`[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:  $\mathbf{e}(1)$  is the prediction error from the first-order model,  $\mathbf{e}(2)$  is the prediction error from the second-order model, and so on.

These prediction error values form the diagonal of the matrix  $\mathbf{E}$  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

## rms

Root-mean-square level

### Syntax

$Y = \text{rms}(X)$

$Y = \text{rms}(X, \text{DIM})$

### Description

$Y = \text{rms}(X)$  returns the root-mean-square (RMS) level of the input,  $X$ . If  $X$  is a row or column vector,  $Y$  is a real-valued scalar. For matrices,  $Y$  contains the RMS levels computed along the first nonsingleton dimension. For example, if  $X$  is an  $N$ -by- $M$  matrix with  $N > 1$ ,  $Y$  is a 1-by- $M$  row vector containing the RMS levels of the columns of  $X$ .

$Y = \text{rms}(X, \text{DIM})$  computes the RMS level of  $X$  along the dimension,  $\text{DIM}$ .

### Input Arguments

#### $X$

Real or complex-valued input vector or matrix. By default, `rms` acts along the first nonsingleton dimension of  $X$ .

#### $\text{DIM}$

Dimension for RMS levels. The optional  $\text{DIM}$  input argument specifies the dimension along which to compute the RMS levels.

**Default:** First nonsingleton dimension

## Output Arguments

**Y**

Root-mean-square level. For vectors, Y is a real-valued scalar. For matrices, Y contains the RMS levels computed along the specified dimension DIM. By default, DIM is the first nonsingleton dimension.

## Examples

### RMS Level of Sinusoid

Compute the RMS level of a 100-Hz sinusoid sampled at 1 kHz.

```
t = 0:0.001:1-0.001;  
X = cos(2*pi*100*t);  
Y = rms(X);
```

### RMS Levels of 2-D Matrix

Create a matrix where each column is a 100-Hz sinusoid sampled at 1 kHz with a different amplitude. The amplitude is equal to the column index.

Compute the RMS levels of the columns.

```
t = 0:0.001:1-0.001;  
x = cos(2*pi*100*t)';  
X = repmat(x,1,4);  
amp = 1:4;  
amp = repmat(amp,1e3,1);  
X = X.*amp;  
Y = rms(X);
```

### RMS Levels of 2-D Matrix Along Specified Dimension

Create a matrix where each row is a 100-Hz sinusoid sampled at 1 kHz with a different amplitude. The amplitude is equal to the row index.

Compute the RMS levels of the rows specifying the dimension equal to 2 with the DIM argument.

```
t = 0:0.001:1-0.001;
```



```
x = cos(2*pi*100*t);
X = repmat(x,4,1);
amp = (1:4)';
amp = repmat(amp,1,1e3);
X = X.*amp;
Y = rms(X,2);
```

## More About

### Root-Mean-Square Level

The root-mean-square level of a vector,  $X$ , is

$$X_{\text{RMS}} = \sqrt{\frac{1}{N} \sum_{n=1}^N |X_n|^2}$$

with the summation performed along the specified dimension.

## References

[1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Std 181, 2003.

### See Also

mean | peak2rms | std

## rooteig

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<sup>2</sup>. 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  $\lambda_{\min}$ , the smallest estimated eigenvalue of the signal's correlation matrix. Eigenvalues below the threshold  $\lambda_{\min} * p(2)$  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

### Frequency Content of Complex Exponentials

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. Reset the random number generator for reproducible results.

```
rng default
n = 0:99;
s = exp(1i*pi/2*n)+2*exp(1i*pi/4*n)+exp(1i*pi/3*n)+randn(1,100);

X = corrmatrix(s,12,'mod');
[W,P] = rooteig(X,3)
```

W =

```
0.7883
1.5674
1.0429
```

P =

4.1748  
1.0572  
1.2419

## More About

### Algorithms

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` | `rootmusic`

# rootmusic

Root MUSIC algorithm

## Syntax

```
W = rootmusic(X,P)
[W,POW] = rootmusic(X,P)
[F, POW] = rootmusic(...,Fs)
[W,POW] = rootmusic(...,'corr')
```

## Description

`W = rootmusic(X,P)` returns the frequencies in radians/sample for the  $P$  complex exponentials (sinusoids) that make up the signal  $X$ .

The input  $X$  is specified either as:

- A row or column vector representing one realization of 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

`[W,POW] = rootmusic(X,P)` returns the estimated absolute value squared amplitudes of the sinusoids at the frequencies  $W$ .

The second input argument,  $P$  is the number of complex sinusoids in  $X$ . You can specify  $P$  as either:

- A positive 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  $\lambda_{\min}$ , the smallest estimated eigenvalue of the signal's correlation matrix. Eigenvalues below the threshold  $\lambda_{\min} * P(2)$  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

$$\text{length}(POW) = 0.5 * \text{length}(W)$$

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. You can place the `'corr'` string anywhere after the  $P$  input argument.

---

**Note** You can use the output of `corrmtx` to generate such an array  $X$ .

---

## Examples

### Sinusoid Amplitudes

Estimate the amplitudes for two sinusoids in noise. The separation between the sinusoids is less than the resolution of the periodogram,  $2\pi/N$  radians/sample. Use the autocorrelation matrix as the input to `rootmusic`.

```
rng default
n = (0:99)';
freqs = [pi/4 pi/4+0.06];

s = 2*exp(1j*freqs(1)*n)+1.5*exp(1j*freqs(2)*n)+ ...
    0.5*randn(100,1)+1j*0.5*randn(100,1);

[~,R] = corrmtx(s,12,'mod');
[W,P] = rootmusic(R,2,'corr')
```

$W =$

0.7946

```
0.8917
```

```
P =
```

```
4.1535  
0.7797
```

## Diagnostics

If the input signal, `x` is real and an odd number of sinusoids, `p` is specified, the following error message is displayed:

```
Real signals require an even number p of complex sinusoids.
```

## More About

### Algorithms

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.

### See Also

```
corrmtx | peig | pmusic | rooteig
```

## **rssq**

Root-sum-of-squares level

### **Syntax**

$Y = \text{rssq}(X)$

$Y = \text{rssq}(X, \text{DIM})$

### **Description**

$Y = \text{rssq}(X)$  returns the root-sum-of-squares (RSS) level,  $Y$ , of the input,  $X$ . If  $X$  is a row or column vector,  $Y$  is a real-valued scalar. For matrices,  $Y$  contains the RSS levels computed along the first nonsingleton dimension. For example, if  $Y$  is an  $N$ -by- $M$  matrix with  $N > 1$ ,  $Y$  is a 1-by- $M$  row vector containing the RSS levels of the columns of  $Y$ .

$Y = \text{rssq}(X, \text{DIM})$  computes the RSS level of  $X$  along the dimension,  $\text{DIM}$ .

### **Input Arguments**

#### **$X$**

Real- or complex-valued input vector or matrix. By default, **rssq** acts along the first nonsingleton dimension of  $X$ .

#### **$\text{DIM}$**

Dimension for root-sum-of-squares (RSS) level. The optional  $\text{DIM}$  input argument specifies the dimension along which to compute the RSS level.

**Default:** First nonsingleton dimension



## Output Arguments

Y

Root-sum-of-squares level. For vectors, Y is a real-valued scalar. For matrices, Y contains the RSS levels computed along the specified dimension, DIM. By default, DIM is the first nonsingleton dimension.

## Examples

### RSS Level of Sinusoid

Compute the RSS level of a 100-Hz sinusoid sampled at 1 kHz.

```
t = 0:0.001:1-0.001;  
X = cos(2*pi*100*t);  
Y = rssq(X);
```

### RSS Level of 2-D Matrix

Create a matrix where each column is a 100-Hz sinusoid sampled at 1 kHz with a different amplitude. The amplitude is equal to the column index.

Compute the RSS level of the columns.

```
t = 0:0.001:1-0.001;  
x = cos(2*pi*100*t)';  
X = repmat(x,1,4);  
amp = 1:4;  
amp = repmat(amp,1e3,1);  
X = X.*amp;  
Y = rssq(X);
```

### RSS Level of 2-D Matrix Along Specified Dimension

Create a matrix where each row is a 100-Hz sinusoid sampled at 1 kHz with a different amplitude. The amplitude is equal to the row index.

Compute the RSS level of the rows specifying the dimension equal to 2 with the DIM argument.

```
t = 0:0.001:1-0.001;
```

```
x = cos(2*pi*100*t);  
X = repmat(x,4,1);  
amp = (1:4)';  
amp = repmat(amp,1,1e3);  
X = X.*amp;  
Y = rssq(X,2);
```

## More About

### Root-Sum-of-Squares Level

The root-sum-of-squares (RSS) level of a vector,  $X$ , is

$$X_{\text{RSS}} = \sqrt{\sum_{n=1}^N |X_n|^2}$$

with the summation performed along the specified dimension. The RSS is also referred to as the  $l^2$  norm.

## References

- [1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003.

# sawtooth

Sawtooth or triangle wave

## Syntax

```
sawtooth(t)
sawtooth(t,width)
```

## Description

`sawtooth(t)` generates a sawtooth wave with period  $2\pi$  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  $2\pi$  and to increase linearly with time with a slope of  $1/\pi$  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  $2\pi$  at which the maximum occurs. The function increases from  $-1$  to  $1$  on the interval 0 to  $2\pi \times \text{width}$ , then decreases linearly from  $1$  to  $-1$  on the interval  $2\pi \times \text{width}$  to  $2\pi$ . Thus a parameter of 0.5 specifies a standard triangle wave, symmetric about time instant  $\pi$  with peak-to-peak amplitude of 1. `sawtooth(t,1)` is equivalent to `sawtooth(t)`.

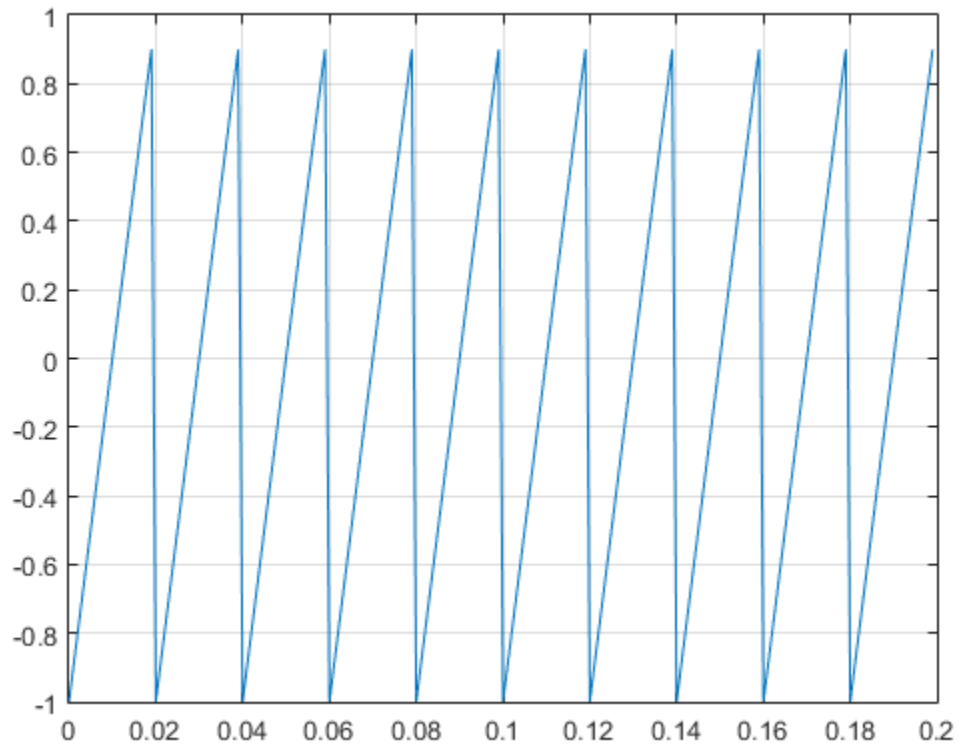
## Examples

### 50 Hz Sawtooth Waveform

Generate 10 periods of a sawtooth wave with a fundamental frequency of 50 Hz. The sample rate is 1 kHz.

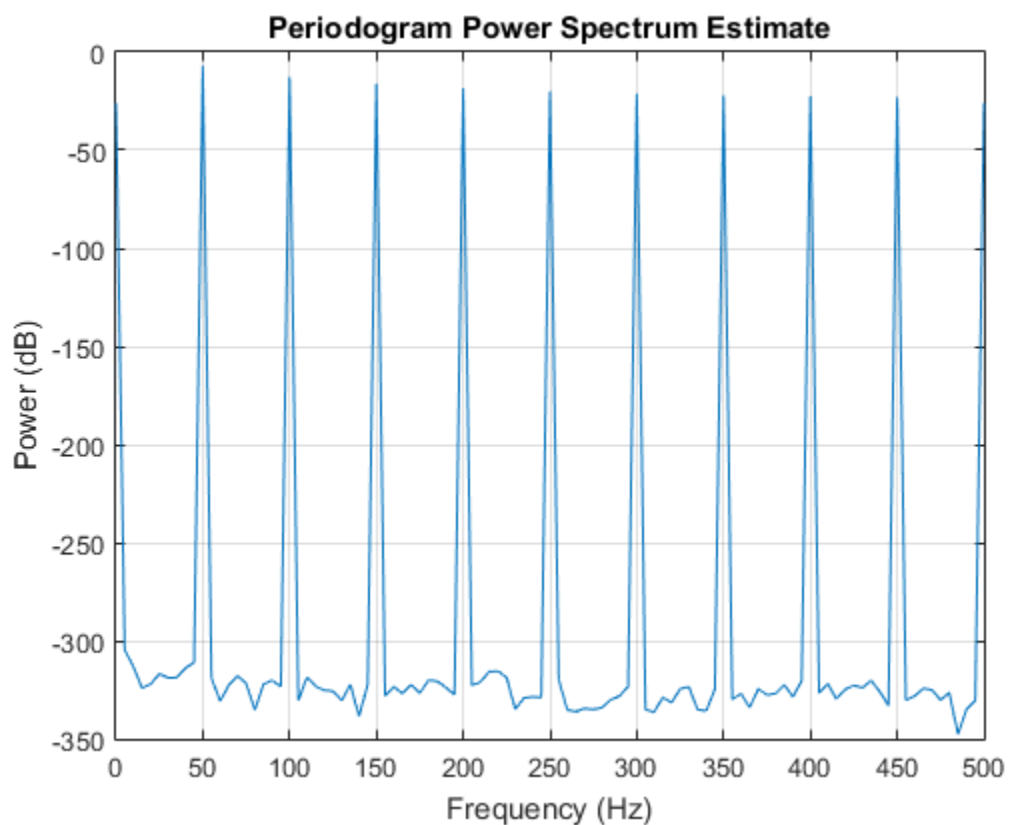
```
T = 10*(1/50);
Fs = 1000;
dt = 1/Fs;
t = 0:dt:T-dt;
x = sawtooth(2*pi*50*t);
```

```
plot(t,x)  
grid on
```



Plot the power spectrum.

```
periodogram(x,[],length(x),Fs,'power')
```

**See Also**

`chirp` | `cos` | `diric` | `gauspuls` | `pulstran` | `rectpuls` | `sin` | `sinc` | `square` | `tripuls`

## schurrc

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

#### Reflection Coefficients of Speech Autocorrelation Sequence

Create an autocorrelation sequence from the MATLAB® speech signal contained in `mtlb.mat`. Use the Schur algorithm to compute the reflection coefficients of a lattice prediction filter for the 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, John G., and Dimitris G. Manolakis. *Digital Signal Processing: Principles, Algorithms, and Applications*. 3rd Edition. Upper Saddle River, NJ: Prentice-Hall, 1996, pp.868–873.

## See Also

levinson

## settlingtime

Settling time for bilevel waveform

### Syntax

```
S = settlingtime(X,D)
S = settlingtime(X,FS,D)
S = settlingtime(X,T,D)
[S,SLEV,SINST] = settlingtime(...)
[S,SLEV,SINST] = settlingtime(...,Name,Value)
settlingtime(...)
```

### Description

`S = settlingtime(X,D)` returns the time, `S`, from the mid-reference level instant to the time instant each transition enters and remains within a 2% tolerance region of the final state over the duration, `D`. `D` is a positive scalar. Because `settlingtime` uses interpolation to determine the mid-reference level instant, `S` may contain values that do not correspond to sampling instants. The length of `S` is equal to the number of detected transitions in the input signal, `X`. If for any transition, the level of the waveform does not remain within the lower and upper tolerance boundaries, the requested duration is not present, or an intervening transition is detected, `settlingtime` marks the corresponding element in `S` as `NaN`. See “Settle Seek Duration” on page 1-1321 for cases in which `settlingtime` returns a `NaN`. To determine the transitions, `settlingtime` estimates the state levels of the input waveform by a histogram method. `settlingtime` identifies all regions that cross the upper-state boundary of the low state and the lower-state boundary of the high state. The low-state and high-state boundaries are expressed as the state level plus or minus a multiple of the difference between the state levels. See “State-Level Tolerances” on page 1-1320.

`S = settlingtime(X,FS,D)` specifies the sampling rate for the bilevel waveform, `X` in hertz. The first sample instant in `X` is equal to `t=0`. Because `settlingtime` uses interpolation to determine the mid-reference level instant, `S` may contain values that do not correspond to sampling instants.

`S = settlingtime(X,T,D)` specifies the sample instants, `T`, as a vector with the same number of elements as `X`.



`[S,SLEV,SINST] = settlingtime(...)` returns vectors, `SLEV`, and `SINST`, whose elements correspond to the levels and sample instants of the settling points for each transition.

`[S,SLEV,SINST] = settlingtime(...,Name,Value)` returns the settling times, levels, and corresponding sample instants with additional options specified by one or more `Name,Value` pair arguments.

`settlingtime(...)` plots the signal and darkens the regions of each transition where settling time is computed. The plot marks the location of the settling time of each transition, the mid-crossings, and the associated reference levels. The plot also displays the state levels with the corresponding lower and upper tolerance boundaries.

## Input Arguments

### **X**

Bilevel waveform. `X` is a real-valued row or column vector.

### **D**

Settle-seek duration. `D` is a positive scalar, which defines the duration after the mid-reference level instant that `settlingtime` looks for a settling time. If no settling time occurs in `D` seconds after the mid-reference level instant, `settlingtime` returns a NaN. See “Settling Time” on page 1-1318 and “Settle Seek Duration” on page 1-1321.

### **FS**

Sample rate in hertz.

### **T**

Vector of sample instants. The length of `T` must equal the length of the bilevel waveform, `X`.

## Name-Value Pair Arguments

### **'MidPercentReferenceLevel'**

Mid-reference level as a percentage of the waveform amplitude. See “Mid-Reference Level” on page 1-1319.

**Default:** 50

**'StateLevels'**

Low and high-state levels. StateLevels is a 1-by-2 real-valued vector. The first element is the low-state level. The second element is the high-state level. If you do not specify low and high-state levels, `settlingtime` estimates the state levels from the input waveform using the histogram method.

**'Tolerance'**

Tolerance levels (lower and upper state boundaries) expressed as a percentage. See “State-Level Tolerances” on page 1-1320.

**Default:** 2

## Output Arguments

### **s**

The time from the mid-reference level instant to the time instant each transition enters and remains within a 2% tolerance region of the final state over duration, D.

### **SLEV**

Waveform values at the settling points.

### **SINST**

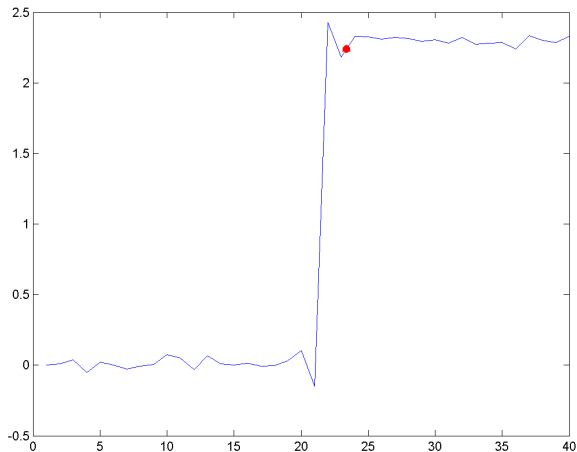
Time instants of the settling points.

## Examples

### **Determine Settling Point and Settling Level**

Determine the settling point and corresponding waveform value for a bilevel waveform. Plot the waveform and mark the settling point.

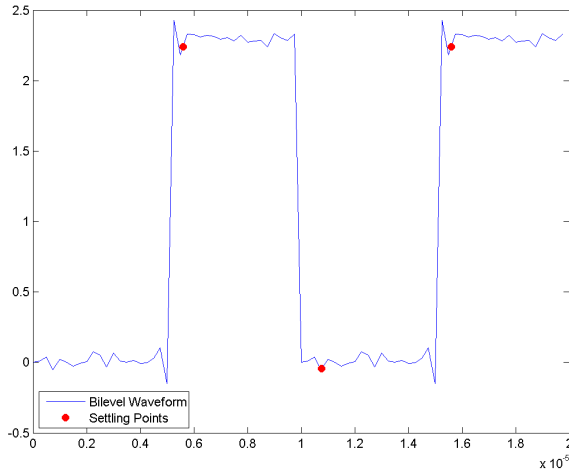
```
load('transitionex.mat', 'x');
[S,SLEV,SINST] = settlingtime(x,10);
plot(x); hold on;
plot(SINST,SLEV,'ro','markerfacecolor',[1 0 0]);
```



### Determine Settling Points for a Three-Transition Bilevel Waveform

Determine the settling points for a three-transition bilevel waveform. The data is sampled at 4 MHz. Use a one-microsecond settle-seek duration. Plot the settling points.

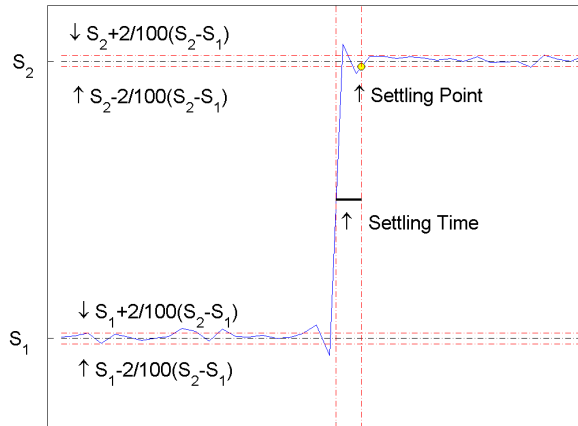
```
load('transitionex.mat', 'x');
y = [x; flip1r(x)];
fs = 4e6;
t = 0:1/fs:(length(y)*1/fs)-1/fs;
[S,SLEV,SINST] = settlingtime(y,fs,1e-6);
% equivalent to [S,SLEV,SINST] = settlingtime(y,t);
plot(t,y); hold on;
plot(SINST,SLEV,'ro','markerfacecolor',[1 0 0]);
legend('Bilevel Waveform','Settling Points','Location','SouthWest');
```



## More About

### Settling Time

The settling time is the time after the mid-reference level instant when the signal crosses into and remains in the 2%-tolerance region around the state level. The settling time is illustrated in the following figure. The low- and high-state levels are the dashed black lines. The 2% tolerances above and below the state levels are shown by the red dashed lines and the settling time is indicated by the yellow circle.



### Mid-Reference Level

The mid-reference level in a bilevel waveform with low-state level,  $S_1$ , and high-state level,  $S_2$ , is

$$S_1 + \frac{1}{2}(S_2 - S_1)$$

### Mid-Reference Level Instant

Let  $y_{50\%}$  denote the mid reference level.

Let  $t_{50\%_-}$  and  $t_{50\%_+}$  denote the two consecutive sampling instants corresponding to the waveform values nearest in value to  $y_{50\%}$ .

Let  $y_{50\%_-}$  and  $y_{50\%_+}$  denote the waveform values at  $t_{50\%_-}$  and  $t_{50\%_+}$ .

The mid-reference level instant is

$$t_{50\%} = t_{50\%_-} + \left( \frac{t_{50\%_+} - t_{50\%_-}}{y_{50\%_+} - y_{50\%_-}} \right) (y_{50\%_+} - y_{50\%_-})$$

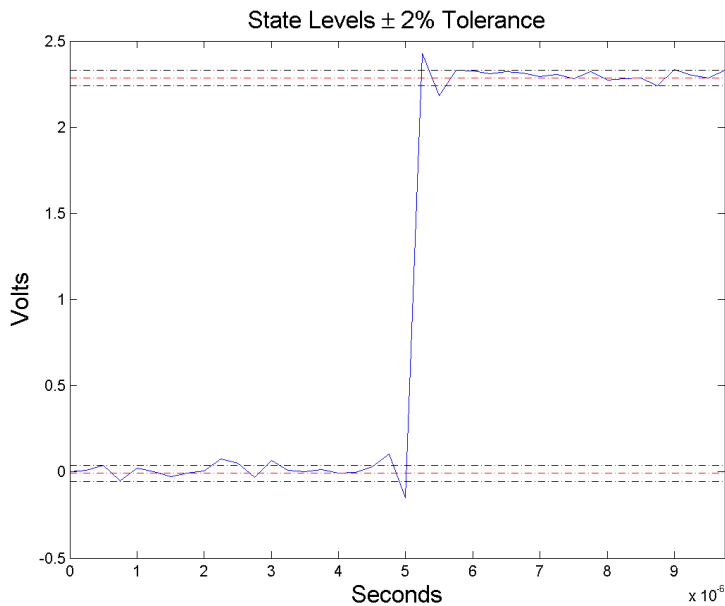
### State-Level Tolerances

Each state level can have associated lower- and upper-state boundaries. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference between the high state and low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the  $\alpha\%$  tolerance region for the low state is defined as

$$S_1 \pm \frac{\alpha}{100}(S_2 - S_1)$$

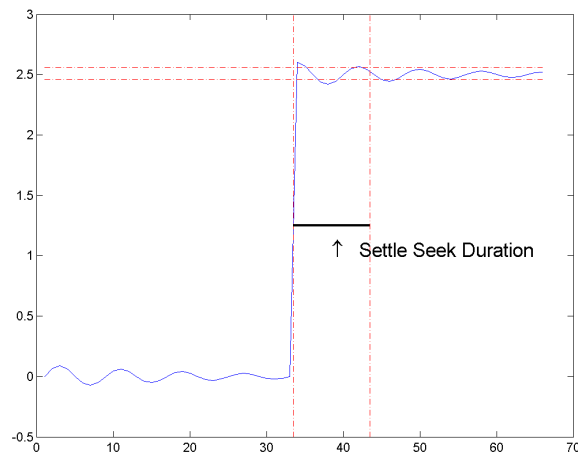
where  $S_1$  is the low-state level and  $S_2$  is the high-state level. Replace the first term in the equation with  $S_2$  to obtain the  $\alpha\%$  tolerance region for the high state.

The following figure illustrates lower and upper 2% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The estimated state levels are indicated by a dashed red line.



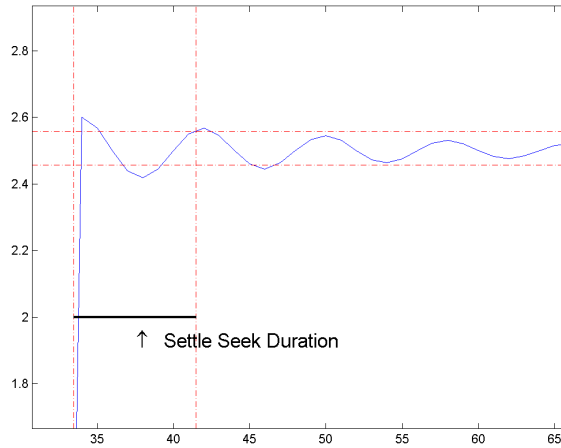
## Settle Seek Duration

The settle seek duration defines the interval of time after the mid-reference level instant that `settlingtime` looks for a settling point. If `settlingtime` does not find a settling point within the settle seek duration, `settlingtime` returns NaN for the settling time. The following figure illustrates a settle seek duration of 10 samples.



`settlingtime` may fail to find a settling point in the specified settle seek duration if any one of the following conditions occurs:

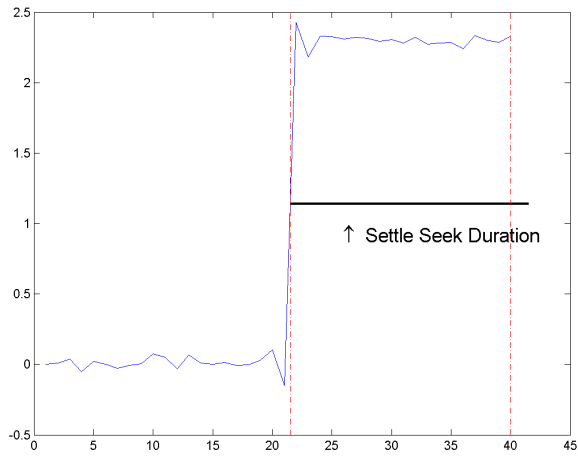
- The last waveform value in the settle seek interval is not within the upper- and lower-state boundaries determined by the specified tolerance. The following figure illustrates this condition for a settle seek duration of 8 samples and a 2% tolerance region.



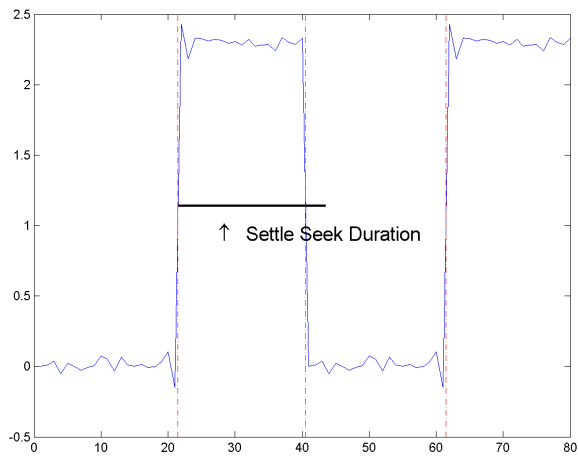
In the preceding figure, you see that the last sample in the settle seek interval exceeds the upper state boundary. In this example, reducing or increasing the settle seek duration can result in a valid settling time.

- There is an insufficient number of waveform samples for the specified settle seek duration. The following figure illustrates this condition for a settle seek duration of 20 samples. The settle seek duration extends beyond the final sample of the waveform.





- An intervening transition is detected before the end of the specified settle seek duration. The following figure illustrates this condition for a settle seek duration of 22 samples. An intervening transition is detected before the end of the 22-sample settle seek duration.



## References

- [1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003, pp. 23–24.

## See Also

falltime | midcross | pulsewidth | risetime | statelevels

# seqperiod

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)
```

$$p = \begin{matrix} & 2 & 1 & 4 & 3 \end{matrix}$$

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.

## setspecs

Specifications for filter specification object

### Syntax

```
setspecs(D,specvalue1,specvalue2,...)  
setspecs(D,Specification,specvalue1,specvalue2,...)  
setspecs(...Fs)  
setspecs(...,MAGUNITS)
```

### Description

`setspecs(D,specvalue1,specvalue2,...)` sets the specifications in filter specification object, `D`, in the same order they appear in the `Specification` property.

`setspecs(D,Specification,specvalue1,specvalue2,...)` changes the specifications for an existing filter specification object and sets values for the new `Specification` property.

`setspecs(...Fs)` specifies the sampling frequency, `Fs`, in Hz. The sampling frequency must be a scalar trailing all other specifications. Entering a sampling frequency causes all other frequency specifications to be in Hz.

`setspecs(...,MAGUNITS)` specifies the units for any magnitude specifications. `MAGUNITS` can be one of the following: `'linear'`, `'dB'`, or `'squared'`. The default is `'dB'`. The magnitude specifications are always converted and stored in dB regardless of how the units are specified.

Use `SET(D,'SPECIFICATION')` to get the list of all available specification types for the filter specification object, `D`.

### Examples

Construct a lowpass filter with specifications for the filter order and cutoff frequency (-6 dB). Use `setspecs` after construction to set the values of the filter order and cutoff frequency. Display the values in the MATLAB command window.

```
D = fdesign.lowpass('N,Fc');  
setspecs(D,10,0.2);  
D.FilterOrder  
D.Fcutoff
```

Construct a highpass filter with specifications for the numerator order, denominator order, and 3-dB frequency. Assume the sampling frequency is 1 kHz. Use `setspecs` to set the numerator and denominator orders to 6. Set the 3-dB frequency to 250 Hz. In order to use frequency specifications in Hz, specify the sampling frequency as a trailing scalar.

```
D = fdesign.highpass('Nb,Na,F3dB');  
setspecs(D,6,6,250,1000);
```

## See Also

`design` | `designmethods` | `designopts` | `fdesign`

# sfdr

Spurious free dynamic range

## Syntax

```

r = sfdr(x)
r = sfdr(x, fs)
r = sfdr(x, fs, msd)

r = sfdr(sxx, f, pwrflag)
r = sfdr(sxx, f, msd, pwrflag)

[r, spurpow, spurfreq] = sfdr( ___ )

sfdr( ___ )

```

## Description

`r = sfdr(x)` returns the spurious free dynamic range (SFDR),  $r$ , in dB of the real sinusoidal signal,  $x$ . `sfdr` computes the power spectrum using a modified periodogram and a Kaiser window with  $\beta = 38$ . The mean is subtracted from  $x$  before computing the power spectrum. The number of points used in the computation of the discrete Fourier transform (DFT) is the same as the length of the signal,  $x$ .

`r = sfdr(x, fs)` returns the SFDR of the time-domain input signal,  $x$ , when the sampling rate,  $fs$ , is specified. The default value of  $fs$  is 1 Hz.

`r = sfdr(x, fs, msd)` returns the SFDR considering only spurs that are separated from the fundamental (carrier) frequency by the minimum spur distance,  $msd$ , specified in cycles/unit time. The sampling frequency is  $fs$ . If the carrier frequency is  $F_c$ , then all spurs in the interval  $(F_c - msd, F_c + msd)$  are ignored.

`r = sfdr(sxx, f, pwrflag)` returns the SFDR of the one-sided power spectrum of a real-valued signal,  $sxx$ .  $f$  is the vector of frequencies corresponding to the power estimates in  $sxx$ . The first element of  $f$  must equal 0. The algorithm removes all the power that decreases monotonically away from the DC bin.

`r = sfdr(sxx, f, msd, pwrflag)` returns the SFDR considering only spurs that are separated from the fundamental (carrier) frequency by the minimum spur distance, `msd`. If the carrier frequency is `Fc`, then all spurs in the interval  $(F_c - \text{msd}, F_c + \text{msd})$  are ignored. When the input to `sfdr` is a power spectrum, specifying `msd` can prevent high sidelobe levels from being identified as spurs.

`[r, spurpow, spurfreq] = sfdr( ___ )` returns the power and frequency of the largest spur.

`sfdr( ___ )` with no output arguments plots the spectrum of the signal in the current figure window. It uses different colors to draw the fundamental component, the DC value, and the rest of the spectrum. It shades the SFDR and displays its value above the plot. It also labels the fundamental and the largest spur.

## Examples

### SFDR of Sinusoid

Obtain the SFDR for a 10 MHz tone with amplitude 1 sampled at 100 MHz. There is a spur at the 1st harmonic (20 MHz) with an amplitude of  $3.16 \times 10^{-4}$ .

```
deltat = 1e-8;  
fs = 1/deltat;  
t = 0:deltat:1e-5-deltat;  
x = cos(2*pi*10e6*t)+3.16e-4*cos(2*pi*20e6*t);  
r = sfdr(x,fs)
```

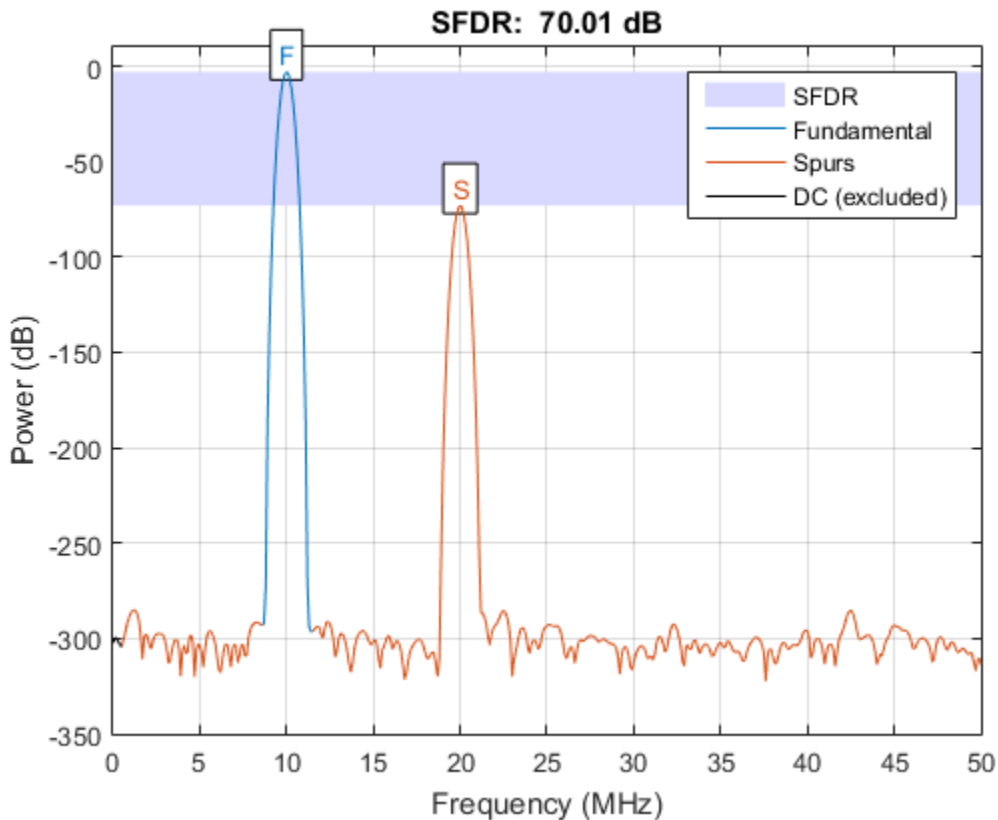
```
r =
```

```
70.0063
```

Display the spectrum of the signal. Annotate the fundamental, the DC value, the spur, and the SFDR.

```
sfdr(x,fs);
```





### Minimum Spur Distance

Obtain the SFDR for a 10 MHz tone with amplitude 1 sampled at 100 MHz. There is a spur at the 1st harmonic (20 MHz) with an amplitude of  $3.16 \times 10^{-4}$  and another spur at 25 MHz with an amplitude of  $10^{-5}$ . Skip the first harmonic by using a minimum spur distance of 11 MHz.

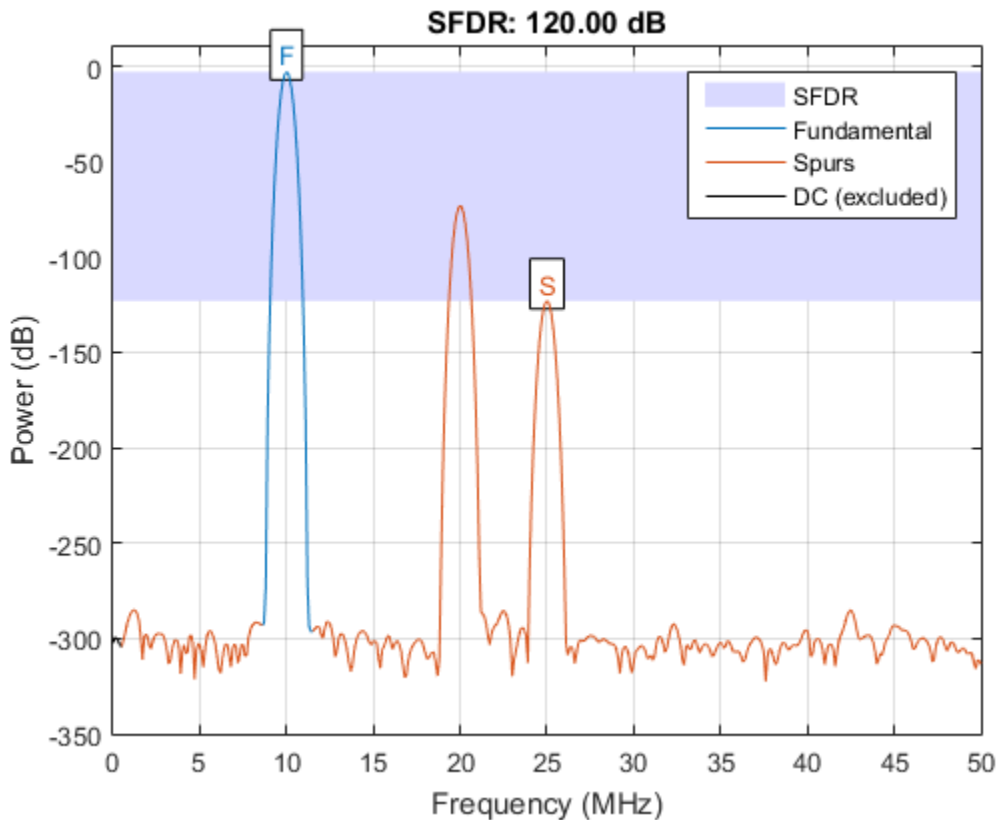
```
deltat = 1e-8;
fs = 1/deltat;
t = 0:deltat:1e-5-deltat;
x = cos(2*pi*10e6*t)+3.16e-4*cos(2*pi*20e6*t)+ ...
    0.1e-5*cos(2*pi*25e6*t);
r = sfdr(x,fs,11e6)
```

```
r =
```

```
120.0000
```

Display the spectrum of the signal. Annotate the fundamental, the DC value, the spurs, and the SFDR.

```
sfdr(x, fs, 11e6);
```



### SFDR from Periodogram

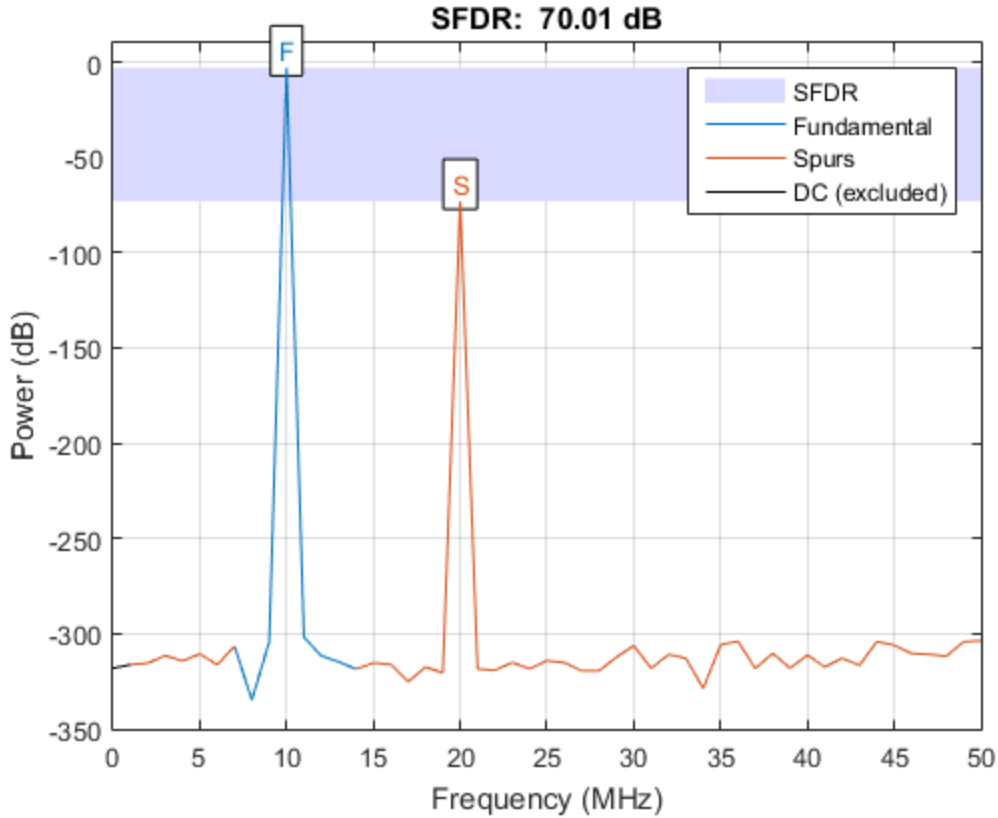
Obtain the power spectrum of a 10 MHz tone with amplitude 1 sampled at 100 MHz.

There is a spur at the 1st harmonic (20 MHz) with an amplitude of  $3.16 \times 10^{-4}$ . Use the one-sided power spectrum and a vector of corresponding frequencies in Hz to compute the SFDR.

```
deltat = 1e-8;
fs = 1/deltat;
t = 0:deltat:1e-6-deltat;
x = cos(2*pi*10e6*t)+3.16e-4*cos(2*pi*20e6*t);
[sxx,f] = periodogram(x,rectwin(length(x)),length(x),fs,'power');
r = sfdr(sxx,f,'power');
```

Display the spectrum of the signal. Annotate the fundamental, the DC value, the first spur, and the SFDR.

```
sfdr(sxx,f, 'power');
```



### Frequency and Power of Largest Spur

Determine the frequency in MHz for the largest spur. The input signal is a 10 MHz tone with amplitude 1 sampled at 100 MHz. There is a spur at the first harmonic (20 MHz) with an amplitude of  $3.16 \times 10^{-4}$ .

```
deltat = 1e-8;
t = 0:deltat:1e-6-deltat;
```

```
x = cos(2*pi*10e6*t)+3.16e-4*cos(2*pi*20e6*t);
[r,spurpow,spurfreq] = sfdr(x,1/deltat);
spur_MHz = spurfreq/1e6
```

```
spur_MHz =
    20
```

### SFDR from Time Series

Create a superposition of three sinusoids, with frequencies of 9.8, 14.7, and 19.6 kHz, in white Gaussian additive noise. The signal is sampled at 44.1 kHz. The 9.8 kHz sine wave has an amplitude of 1 volt, the 14.7 kHz wave has an amplitude of 100 microvolts, and the 19.6 kHz signal has amplitude 30 microvolts. The noise has 0 mean and a variance of 0.01 microvolt. Additionally, the signal has a DC shift of 0.1 volt.

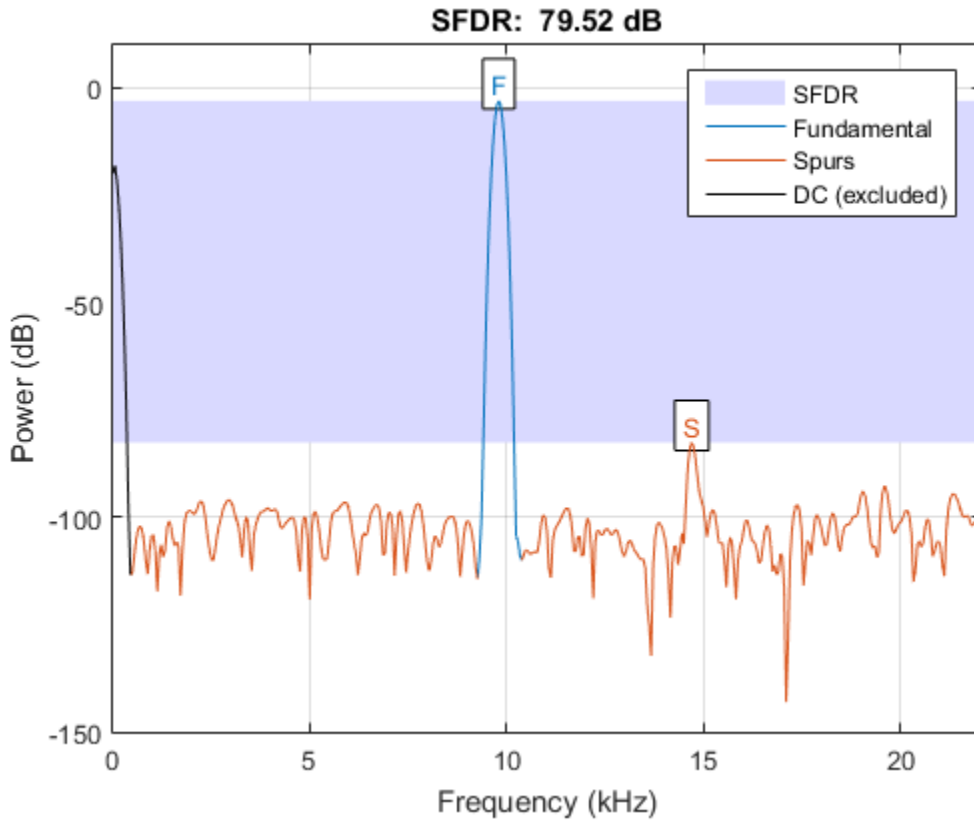
```
rng default

Fs = 44.1e3;
f1 = 9.8e3;
f2 = 14.7e3;
f3 = 19.6e3;
N = 900;

nT = (0:N-1)/Fs;
x = 0.1+sin(2*pi*f1*nT)+100e-6*sin(2*pi*f2*nT) ...
    +30e-6*sin(2*pi*f3*nT)+sqrt(1e-8)*randn(1,N);
```

Plot the spectrum and SFDR of the signal. Display its fundamental and its largest spur. The DC level is excluded from the SFDR computation.

```
sfdr(x,Fs);
```



- “Spurious-Free Dynamic Range (SFDR) Measurement”

## Input Arguments

**x** — Real-valued sinusoidal signal

row vector | column vector

Real-valued sinusoidal signal, specified as a row or column vector. The mean is subtracted from  $x$  prior to obtaining the power spectrum for SFDR computation.

Example:  $x = \cos(\pi/4*(0:79))+1e-4*\cos(\pi/2*(0:79));$

Data Types: double

**fs — Sampling rate**

1 (default) | positive scalar

Sampling rate of the signal in cycles/unit time, specified as a positive scalar. When the unit of time is seconds, fs is in Hz.

Data Types: double

**msd — Minimum spur distance**

0 (default) | positive scalar

Minimum number of discrete Fourier transform (DFT) bins to ignore in the SFDR computation, specified as a positive scalar. You can use this argument to ignore spurs or sidelobes that occur in close proximity to the fundamental frequency. For example, if the carrier frequency is  $F_c$ , then all spurs in the range  $(F_c - \text{msd}, F_c + \text{msd})$  are ignored.

Data Types: double

**sxx — One-sided power spectrum**

row or column vector of positive numbers

One-sided power spectrum to use in the SFDR computation, specified as row or column vector.

Data Types: double

**f — Vector of frequencies**

row or column vector of nonnegative numbers

Vector of frequencies corresponding to the power estimates in sxx, specified as a row or column vector.

**pwrflag — Power spectrum input flag**

'power'

Flag indicating that the input is a one-sided power spectrum, sxx, specified as the string 'power'.

## Output Arguments

**r — Spurious free dynamic range**

real-valued scalar

Spurious free dynamic range in dB, specified as a real-valued scalar. The spurious free dynamic range is the difference in dB between the power at the peak frequency and the power at the next largest frequency (spur). If the input is time series data, the power estimates are obtained from a modified periodogram using a Hamming window. The length of the DFT used in the periodogram is equal to the length of the input signal, `x`. If you want to use a different power spectrum as the basis for the SFDR measurement, you can input your power spectrum using the `'power'` flag.

Data Types: `double`

**spurpow — power of largest spur**

real-valued scalar

Power in dB of the largest spur, specified as a real-valued scalar.

Data Types: `double`

**spurfreq — frequency of largest spur**

real-valued scalar

Frequency in Hz of the largest spur, specified as a real-valued scalar. If you do not supply the sampling frequency as an input argument, `sfdr` assumes a sampling frequency of 1 Hz.

Data Types: `double`

## More About

### Distortion Measurement Functions

The functions `thd`, `sfdr`, `sinad`, and `snr` measure the response of a weakly nonlinear system stimulated by a sinusoid.

When given time-domain input, `sfdr` performs a periodogram using a Kaiser window with large sidelobe attenuation. To find the fundamental frequency, the algorithm searches the periodogram for the largest nonzero spectral component. It then computes the central moment of all adjacent bins that decrease monotonically away from the maximum. To be detectable, the fundamental should be at least in the second frequency bin. If a harmonic lies within the monotonically decreasing region in the neighborhood of another, its power is considered to belong to the larger harmonic. This larger harmonic may or may not be the fundamental. The algorithm ignores all the power that decreases monotonically away from the DC bin.



sfdr fails if the fundamental is not the highest spectral component in the signal.

Ensure that the frequency components are far enough apart to accommodate for the sidelobe width of the Kaiser window. If this is not feasible, you can use the 'power' flag and compute a periodogram with a different window.

### **See Also**

bandpower | enbw | periodogram

# sgolay

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  $p^{\text{th}}$  order derivative, `xp`, of its middle value from:

```
xp((f+1)/2) = (factorial(p)) * g(:,p+1)' * x
```

## Examples

### Savitzky-Golay Smoothing of Noisy Sinusoid

Use `sgolay` to smooth a noisy sinusoid. Compute the resulting first and second derivatives.

```

N = 4;                % Order of polynomial fit
F = 21;              % Window length
[b,g] = sgolay(N,F); % Calculate S-G coefficients

dx = 0.2;
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,
    % Zeroth 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

Find the first and second derivatives using diff. Compare the 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', 'S-G Smoothed')
legend boxoff
ylabel('Sinusoid')

subplot(3,1,2)
plot([DiffD1', SG1'])
legend('Diff', 'S-G')
legend boxoff
ylabel('1st derivative')

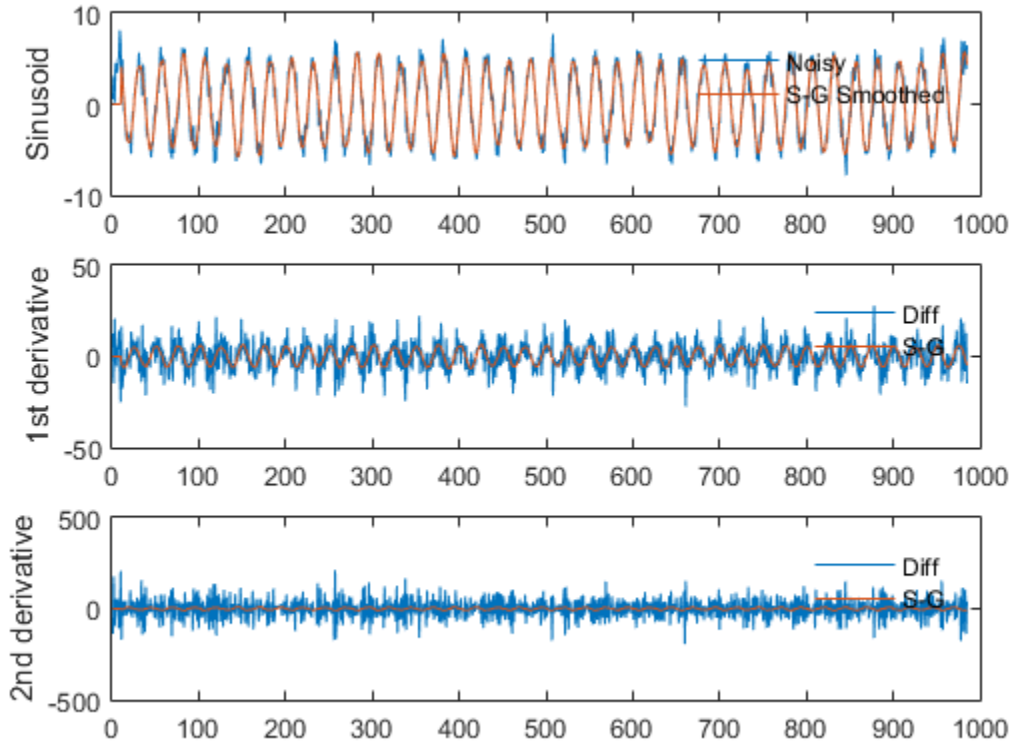
subplot(3,1,3)
plot([DiffD2', SG2'])

```

```

legend('Diff','S-G')
legend boxoff
ylabel('2nd derivative')

```

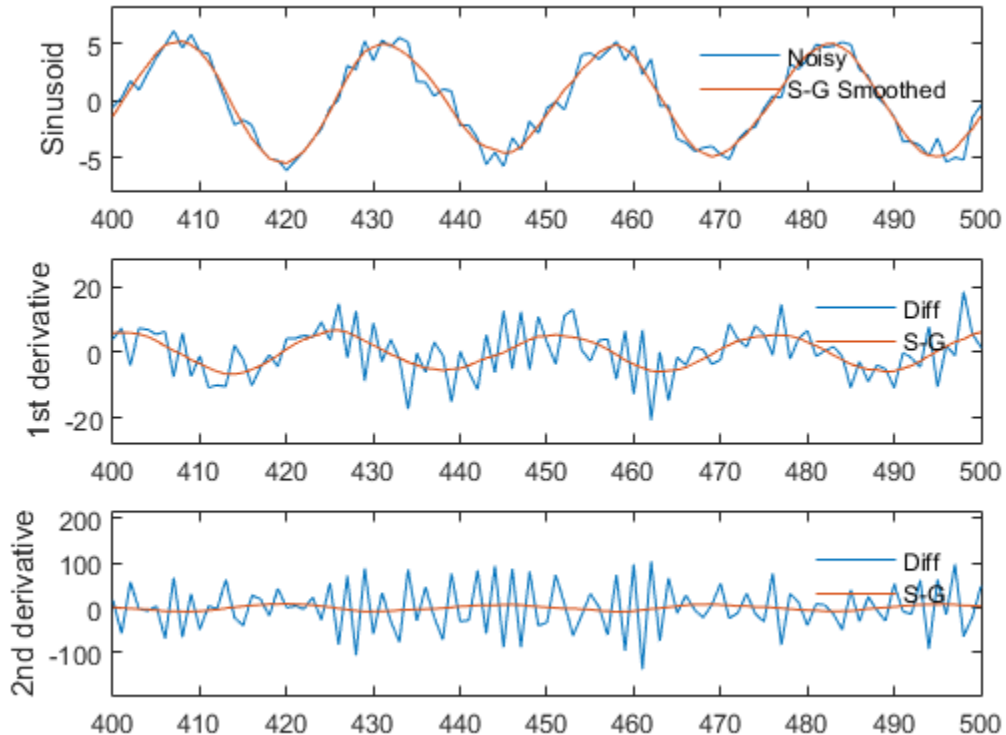


Zoom in to see more detail. Note how `diff` amplifies the noise and generates useless results.

```

for kj = 1:3
    subplot(3,1,kj)
    axis([400 500 -Inf Inf])
end

```



## More About

### Tips

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.

## References

- [1] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1996.

## See Also

`fir1` | `firls` | `filter` | `sgolayfilt`

# sgolayfilt

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.

## Examples

### Savitzky-Golay Filtering of a Speech Signal

Smooth the `mtlb` signal by applying a cubic Savitzky-Golay filter to data frames of length 41.

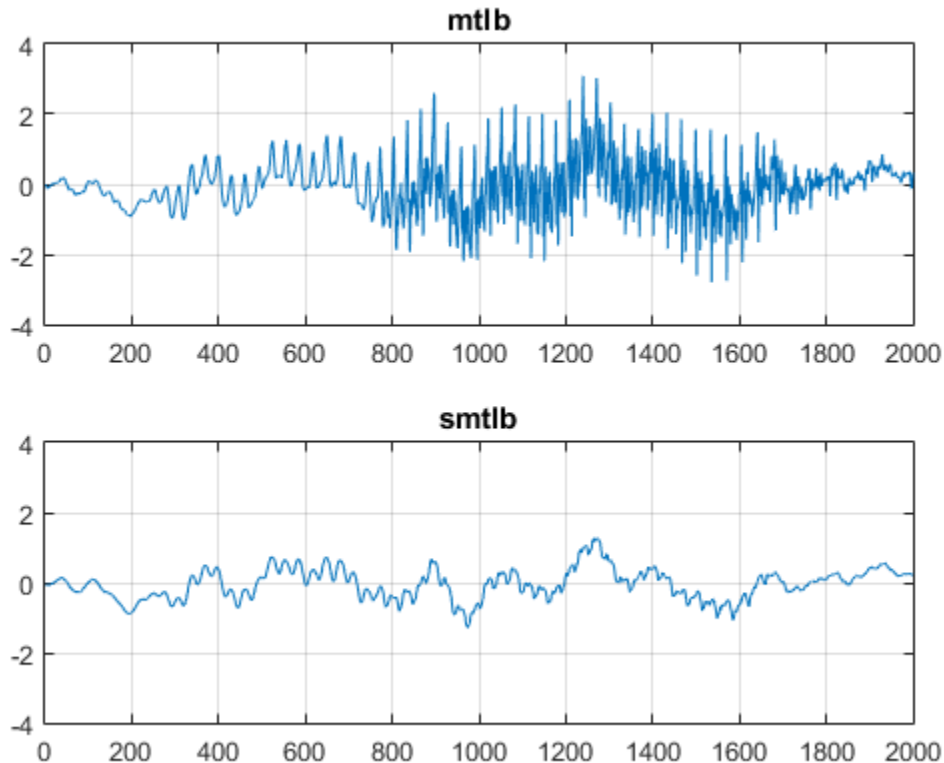
```
load mtlb
smtlb = sgolayfilt(mtlb,3,41);

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
```





## More About

### Tips

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.

## References

[1] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1996.

## See Also

`medfilt1` | `filter` | `sgolay` | `sosfilt`

# shiftdata

Shift data to operate on specified dimension

## Syntax

```
[x,perm,nshifts] = shiftdata(x,dim)
```

## Description

`[x,perm,nshifts] = shiftdata(x,dim)` shifts data `x` to permute dimension `dim` to the first column using the same permutation as the built-in `filter` function. The vector `perm` returns the permutation vector that is used.

If `dim` is missing or empty, then the first nonsingleton dimension is shifted to the first column, and the number of shifts is returned in `nshifts`.

`shiftdata` is meant to be used in tandem with `unshiftdata`, which shifts the data back to its original shape. These functions are useful for creating functions that work along a certain dimension, like `filter`, `goertzel`, `sgolayfilt`, and `sosfilt`.

## Examples

### Permute Dimensions of a Magic Square

This example shifts `x`, a 3-by-3 magic square, permuting dimension 2 to the first column. `unshiftdata` shifts `x` back to its original shape.

Create a 3-by-3 magic square.

```
x = magic(3)
```

```
x =
```

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

Shift the matrix `x` to work along the second dimension. The permutation vector, `perm`, and the number of shifts, `nshifts`, are returned along with the shifted matrix.

```
[x,perm,nshifts] = shiftdata(x,2)
```

```
x =
```

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

```
perm =
```

```
     2     1
```

```
nshifts =
```

```
     []
```

Shift the matrix back to its original shape.

```
y = unshiftdata(x,perm,nshifts)
```

```
y =
```

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

### **Rearrange Array to Operate on First Nonsingleton Dimension**

This example shows how `shiftdata` and `unshiftdata` work when you define `dim` as empty.

Define `x` as a row vector.

```
x = 1:5
```

```
x =  
    1    2    3    4    5
```

Define `dim` as empty to shift the first non-singleton dimension of `x` to the first column. `shiftdata` returns `x` as a column vector, along with `perm`, the permutation vector, and `nshifts`, the number of shifts.

```
[x,perm,nshifts] = shiftdata(x,[])
```

```
x =  
    1  
    2  
    3  
    4  
    5
```

```
perm =  
    []
```

```
nshifts =  
    1
```

Using `unshiftdata`, restore `x` to its original shape.

```
y = unshiftdata(x,perm,nshifts)
```

```
y =  
    1    2    3    4    5
```

## See Also

`permute` | `shiftdim` | `unshiftdata`

## sigwin

Signal processing window object

### Syntax

```
w = sigwin.window
```

### Description

---

**Note:** The use of `sigwin.window` is not recommended. Use the corresponding function instead. See “Windows” on page 1-1352 for the functional forms.

---

`w = sigwin.window` returns a window object, `w`, of type `window`. Each window type 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 specify a `window` type with `sigwin`.

---

### Windows

`window` for `sigwin` specifies the type of window. The following table lists the supported window functions with links to the corresponding class reference page for the window object.

| Window                           | Window Object                      | Corresponding Function      |
|----------------------------------|------------------------------------|-----------------------------|
| Modified Bartlett-Hanning Window | <code>sigwin.barthannwin</code>    | <code>barthannwin</code>    |
| Bartlett Window                  | <code>sigwin.bartlett</code>       | <code>bartlett</code>       |
| Blackman Window                  | <code>sigwin.blackman</code>       | <code>blackman</code>       |
| Blackman-Harris Window           | <code>sigwin.blackmanharris</code> | <code>blackmanharris</code> |
| Bohman Window                    | <code>sigwin.bohmanwin</code>      | <code>bohmanwin</code>      |

| Window   | Window Object                  | Corresponding Function  |
|--|--------------------------------|-------------------------|
| Dolph-Chebyshev Window                           | <code>sigwin.chebwin</code>    | <code>chebwin</code>    |
| Flat Top Window                                  | <code>sigwin.flattopwin</code> | <code>flattopwin</code> |
| Gaussian Window                                  | <code>sigwin.gausswin</code>   | <code>gausswin</code>   |
| Hamming Window                                   | <code>sigwin.hamming</code>    | <code>hamming</code>    |
| Hann (Hanning) Window                            | <code>sigwin.hann</code>       | <code>hann</code>       |
| Kaiser Window                                    | <code>sigwin.kaiser</code>     | <code>kaiser</code>     |
| Nuttall defined 4-term<br>Blackman-Harris Window | <code>sigwin.nuttallwin</code> | <code>nuttallwin</code> |
| Parzen Window                                    | <code>sigwin.parzenwin</code>  | <code>parzenwin</code>  |
| Rectangular Window                               | <code>sigwin.rectwin</code>    | <code>rectwin</code>    |
| Taylor Window                                    | <code>sigwin.taylorwin</code>  | <code>taylorwin</code>  |
| Triangular Window                                | <code>sigwin.triang</code>     | <code>triang</code>     |
| Tukey Window                                     | <code>sigwin.tukeywin</code>   | <code>tukeywin</code>   |

## 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  |
|-----------------------|--|
| <code>generate</code> | Returns a column vector of values representing the window.   |
| <code>info</code>     | Returns information about the window object.   |
| <code>winwrite</code> | Writes an ASCII file that contains window weights for a single window object or a vector of window objects. Default filename is <code>untitled.wf</code> .<br><br><code>winwrite(Hd, filename)</code> writes to a disk file named <code>filename</code> in the current |

| Method | Description   |
|--------|---|
|        | working directory. The <code>.wf</code> 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.

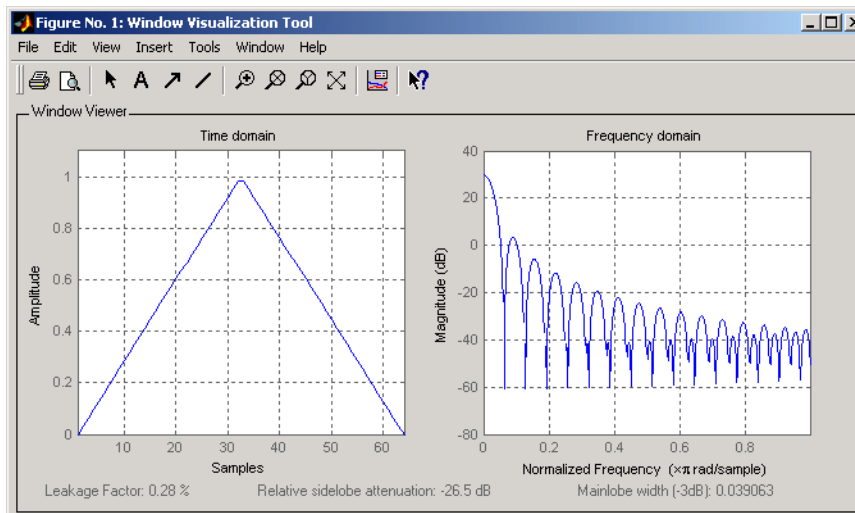
## Examples

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
wvtool(w)

w =
  Length: 64
  Name: 'Bartlett'
```





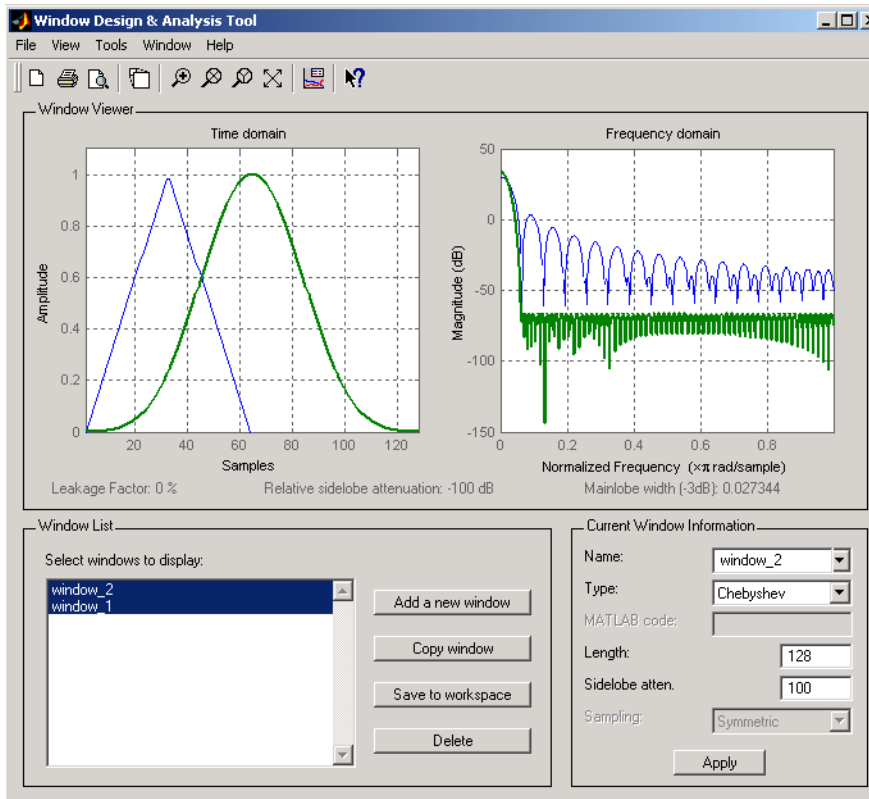
Create a 128-point Chebyshev window with 100 dB of sidelobe attenuation. (See `chebwin` for information on Chebyshev windows.) View the results of this and the above Bartlett window in the Window Design and Analysis Tool (`wintool`):

```
w1=sigwin.chebwin(128,100)
wintool(w,w1)
```

w1 =

```

      Length: 128
      Name: 'Chebyshev'
      SidelobeAtten: 100
```



To save the window values in a vector, use:

```
d = generate(w);
```

## See Also

window | wintool | wvtool

## sigwin.barthannwin class

**Package:** sigwin

Construct Bartlett-Hanning window object

### Description

---

**Note:** The use of `sigwin.barthannwin` is not recommended. Use `barthannwin` instead.

---

`sigwin.barthannwin` creates a handle to a Bartlett-Hanning window object for use in spectral analysis and FIR filtering by the window method. Object methods enable workspace import and ASCII file export of the window values.

The following equation defines a modified Bartlett-Hanning window of length  $N$ :

$$w(x) = 0.62 - 0.48|x| + 0.38 \cos 2\pi x, \quad -\frac{1}{2} \leq x \leq \frac{1}{2}$$

where  $x$  is an  $N$ -point linearly spaced vector over the interval  $[1/2, 1/2]$ .

### Construction

`H = sigwin.barthannwin` returns a modified Bartlett-Hanning window object `H` of length 64.

`H = sigwin.barthannwin(Length)` returns a modified Bartlett-Hanning window object `H` of length *Length*. *Length* requires a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

## Properties

### Length

Modified Bartlett-Hanning window length. The window length requires a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

## Methods

generate

Generates modified Bartlett-Hanning window

info

Display information about modified Bartlett-Hanning window object

winwrite

Save Bartlett window object values in ASCII file

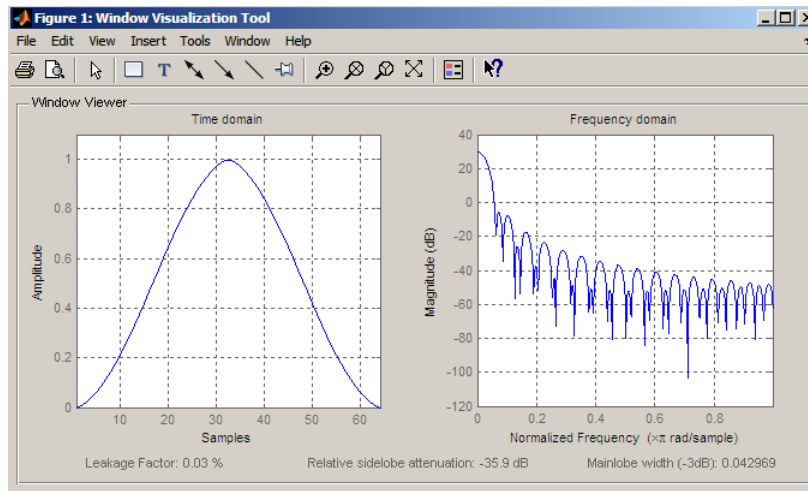
## Copy Semantics

Handle. To learn how copy semantics affect your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

## Examples

Default length  $N = 64$  modified Bartlett-Hanning window:

```
H = sigwin.barthannwin;  
wvtool(H)
```



Generate length  $N = 128$  modified Bartlett-Hanning window, return values, and write ASCII file with window values:

```
H = sigwin.barthannwin(128);
% Return window with generate
win = generate(H);
% Write ASCII file in current directory
% with window values
winwrite(H, 'barthannwin_128')
```

## References

Yeong, H. H., and Pearce, J. A. "A New Window and Comparison to Standard Windows." *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. 37, 1989, pp. 298–301.

## See Also

barthannwin | window | wvtool

## How To

- Class Attributes
- Property Attributes

## generate

**Class:** sigwin.barthannwin

**Package:** sigwin

Generates modified Bartlett-Hanning window

## Syntax

```
win = generate(H)
```

## Description

`win = generate(H)` returns the values of the modified Bartlett-Hanning window object `H` as a double-precision column vector.

## Examples

Extract values from modified Bartlett-Hanning window object:

```
H=sigwin.barthannwin(128);  
% Extract window values as column vector  
win=generate(H);
```

## info

**Class:** sigwin.barthannwin

**Package:** sigwin

Display information about modified Bartlett-Hanning window object

## Syntax

```
info(H)
info_win = info(H)
```

## Description

`info(H)` displays length information for the modified Bartlett-Hanning window object `H`.

`info_win = info(H)` returns length information for the modified Bartlett-Hanning window object `H` in the character array `info_win`.

## Examples

Return information about a modified Bartlett-Hanning window object:

```
H = sigwin.barthannwin(256);
info_win = info(H);
```

## winwrite

**Class:** sigwin.barthannwin

**Package:** sigwin

Save Bartlett window object values in ASCII file

## Syntax

```
winwrite(H)
winwrite(H, 'filename')
```

## Description

`winwrite(H)` opens a dialog box that enables you to export the values of the modified Bartlett-Hanning window object `H` to an ASCII file with filename extension `wf`.

`winwrite(H, 'filename')` saves the values of the modified Bartlett-Hanning window object `H` in the current folder as a column vector in the ASCII file `'filename'`. The filename extension is `wf`.

## Examples

Write modified Bartlett-Hanning window values to ASCII file:

```
H = sigwin.barthannwin;
% Open dialog box for ASCII file
winwrite(H);
```



# sigwin.bartlett class

**Package:** sigwin

Construct Bartlett window object

## Description

---

**Note:** The use of `sigwin.bartlett` is not recommended. Use `bartlett` instead.

---

`sigwin.bartlett` creates a handle to a Bartlett window object for use in spectral analysis and filtering by the window method. Object methods enable workspace import and ASCII file export of the window values.

For  $N$  even, the following equation defines the Bartlett window:

$$w(n) = \begin{cases} \frac{2n}{N-1} & 0 \leq n \leq N/2 - 1 \\ 2 - \frac{2n}{N-1} & N/2 \leq n \leq N-1 \end{cases}$$

For  $N$  odd, the equation for the Bartlett window is:

$$w(n) = \begin{cases} \frac{2n}{N-1} & 0 \leq n \leq (N-1)/2 \\ 2 - \frac{2n}{N-1} & (N-1)/2 + 1 \leq n \leq N-1 \end{cases}$$

## Construction

`H = sigwin.bartlett` returns a Bartlett window object `H` of length 64.

`H = sigwin.bartlett(Length)` returns a Bartlett window object `H` of length *Length*. *Length* must be a positive integer. Entering a positive noninteger value for *Length*

rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

## Properties

### Length

Bartlett window length. The length requires a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

## Methods

|          |  |
|----------|--|
| generate | Generates Bartlett window                        |
| info     | Display information about Bartlett window object |
| winwrite | Save Bartlett window object values in ASCII file |

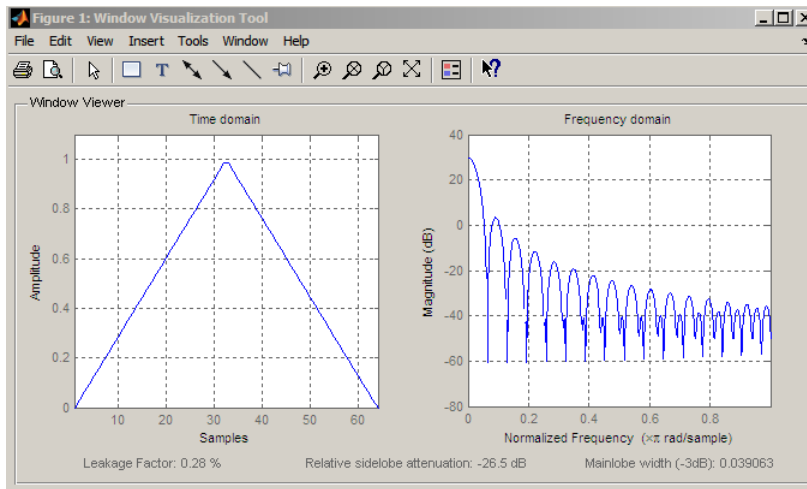
## Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

## Examples

Create default length  $N = 64$  Bartlett window:

```
H = sigwin.bartlett;  
wvtool(H)
```



Generate length  $N = 128$  Bartlett window, return values, and write an ASCII file with window values:

```
H = sigwin.bartlett(128);
% Return window with generate
win = generate(H);
% Write ASCII file in current directory
% with window values
winwrite(H, 'bartlett_128')
```

## References

Oppenheim, Alan V., and Ronald W. Schaffer. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1989.

## See Also

bartlett | window | wvtool

## How To

- Class Attributes
- Property Attributes

## generate

**Class:** sigwin.bartlett

**Package:** sigwin

Generates Bartlett window

## Syntax

```
win = generate(H)
```

## Description

`win = generate(H)` returns the values of the Bartlett window object `H` as a double-precision column vector.

## Examples

Extract values from Bartlett window object:

```
H = sigwin.bartlett(128);  
% Extract window values as column vector  
win=generate(H);
```

## info

**Class:** sigwin.bartlett

**Package:** sigwin

Display information about Bartlett window object

## Syntax

```
info(H)
info_win = info(H)
```

## Description

`info(H)` displays length information for the Bartlett window object `H`.

`info_win = info(H)` returns length information for the Bartlett window object `H` in the character array `info_win`.

## Examples

Return information about a Bartlett window object:

```
H = sigwin.bartlett(256);
info_win = info(H);
```

## winwrite

**Class:** sigwin.bartlett

**Package:** sigwin

Save Bartlett window object values in ASCII file

## Syntax

```
winwrite(H)
winwrite(H, 'filename')
```

## Description

`winwrite(H)` opens a dialog box that enables you to export the values of the Bartlett window object `H` to an ASCII file with filename extension `wf`.

`winwrite(H, 'filename')` saves the values of the Bartlett window object `H` in the current folder as a column vector in the ASCII file `'filename'`. The filename extension is `wf`.

## Examples

Write Bartlett window values to ASCII file:

```
H=sigwin.bartlett;
% Open dialog box for ASCII file
winwrite(H);
```

# sigwin.blackman class

**Package:** sigwin

Construct Blackman window object

## Description

---

**Note:** The use of `sigwin.blackman` is not recommended. Use `blackman` instead.

---

`sigwin.blackman` creates a handle to a Blackman window object for use in spectral analysis and FIR filtering by the window method. Object methods enable workspace import and ASCII file export of the window values.

The following equation defines the Blackman window of length  $N$ :

$$w(n) = 0.42 - 0.5 \cos \frac{2\pi n}{N-1} + 0.08 \cos \frac{4\pi n}{N-1}, \quad 0 \leq n \leq M-1$$

where  $M$  is  $N/2$  for  $N$  even and  $(N+1)/2$  for  $N$  odd.

In the symmetric case, the second half of the Blackman window  $M \leq n \leq N-1$  is obtained by flipping the first half around the midpoint. The symmetric option is the preferred method when using a Blackman window in FIR filter design.

The periodic Blackman window is constructed by extending the desired window length by one sample to  $N+1$ , constructing a symmetric window, and removing the last sample. The periodic version is the preferred method when using a Blackman window in spectral analysis because the discrete Fourier transform assumes periodic extension of the input vector.

## Construction

`H = sigwin.blackman` returns a Blackman window object `H` of length 64 with symmetric sampling.

`H = sigwin.blackman(Length)` returns a Blackman window object `H` of length *Length* with symmetric sampling. *Length* requires a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

`H = sigwin.blackman(Length,SamplingFlag)` returns a Blackman window object `H` with sampling *Sampling\_Flag*. The *Sampling\_Flag* can be either 'symmetric' or 'periodic'.

## Properties

### Length

Blackman window length. Must be a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

### SamplingFlag

'symmetric' is the default and forces exact symmetry between the first and second halves of the Blackman window. A symmetric window is preferred in FIR filter design by the window method.

'periodic' designs a symmetric Blackman window of length *Length*+1 and truncates the window to length *Length*. This design is preferred in spectral analysis where the window is treated as one period of a *Length*-point periodic sequence.

## Methods

|          |  |
|----------|--|
| generate | Generates Blackman window                        |
| info     | Display information about Blackman window object |
| winwrite | Save Blackman window in ASCII file               |



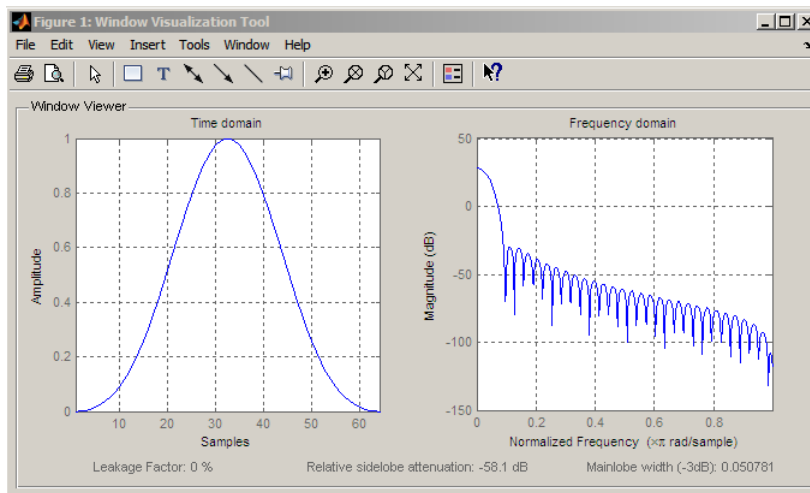
## Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

## Examples

Default length  $N = 64$  symmetric Blackman window:

```
H = sigwin.blackman;
wvtool(H)
```



Generate length  $N = 128$  periodic Blackman window, return values, and write ASCII file:

```
H = sigwin.blackman(128,'periodic');
% Return window with generate
win = generate(H);
% Write ASCII file in current directory
% with window values
winwrite(H,'blackman_128')
```

## References

Oppenheim, Alan V., and Ronald W. Schaffer. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1989.

## See Also

`window` | `wvtool` | `blackman`

## How To

- Class Attributes
- Property Attributes

## generate

**Class:** sigwin.blackman

**Package:** sigwin

Generates Blackman window

## Syntax

```
win = generate(H)
```

## Description

`win = generate(H)` returns the values of the Blackman window object `H` as a double-precision column vector.

## Examples

Extract values from Blackman window object:

```
H = sigwin.blackman(128);  
% Extract window values as column vector  
win = generate(H);
```

## info

**Class:** sigwin.blackman

**Package:** sigwin

Display information about Blackman window object

## Syntax

```
info(H)  
info_win = info(H)
```

## Description

`info(H)` displays length and sampling information about the Blackman window object `H`.

`info_win = info(H)` returns length and sampling information about the Blackman window object `H` in the character array `info_win`.

## Examples

Return information about a Blackman window object:

```
H = sigwin.blackman(256);  
info_win = info(H);
```

# winwrite

**Class:** sigwin.blackman

**Package:** sigwin

Save Blackman window in ASCII file

## Syntax

```
winwrite(H)
winwrite(H, 'filename')
```

## Description

`winwrite(H)` opens a dialog box that enables you to export the values of the Blackman window object `H` to an ASCII file with filename extension `wf`.

`winwrite(H, 'filename')` saves the values of the Blackman window object `H` in the current folder as a column vector in the ASCII file `'filename'` with filename extension `wf`.

## Examples

Write Blackman window values to ASCII file:

```
H=sigwin.blackman;
% Open dialog box for ASCII file
winwrite(H);
```

## sigwin.blackmanharris class

**Package:** sigwin

Construct Blackman-Harris window object

### Description

---

**Note:** The use of `sigwin.blackmanharris` is not recommended. Use `blackmanharris` instead.

---

`sigwin.blackmanharris` creates a handle to a Blackman-Harris window object for use in spectral analysis and FIR filtering by the window method. Object methods enable workspace import and ASCII file export of the window values.

The following equation defines the **symmetric** Blackman-Harris window of length  $N$ :

$$w(n) = a_0 - a_1 \cos\left(\frac{2\pi n}{N-1}\right) + a_2 \cos\left(\frac{4\pi n}{N-1}\right) - a_3 \cos\left(\frac{6\pi n}{N-1}\right), \quad 0 \leq n \leq N-1$$

The following equation defines the **periodic** Blackman-Harris window of length  $N$ :

$$w(n) = a_0 - a_1 \cos\frac{2\pi n}{N} + a_2 \cos\frac{4\pi n}{N} - a_3 \cos\frac{6\pi n}{N}, \quad 0 \leq n \leq N-1$$

The following table lists the coefficients:

| Coefficient | Value   |
|-------------|---------|
| $a_0$       | 0.35875 |
| $a_1$       | 0.48829 |
| $a_2$       | 0.14128 |
| $a_3$       | 0.01168 |

## Construction

`H = sigwin.blackmanharris` returns a Blackman-Harris window object `H` of length 64.

`H = sigwin.blackmanharris(Length)` returns a Blackman-Harris window object `H` of length `Length`. `Length` must be a positive integer. Entering a positive noninteger value for `Length` rounds the length to the nearest integer. Entering a 1 for `Length` results in a window with a single value of 1.

## Properties

### Length

Blackman-Harris window length. The window length requires a positive integer. Entering a positive noninteger value for `Length` rounds the length to the nearest integer. Entering a 1 for `Length` results in a window with a single value of 1.

### SamplingFlag

The type of window returned as one of `'symmetric'` or `'periodic'`. The default is `'symmetric'`. A symmetric window exhibits perfect symmetry between halves of the window. Setting the `SamplingFlag` property to `'periodic'` results in a N-periodic window. The equations for the Blackman-Harris window differ slightly based on the value of the `SamplingFlag` property. See “Description” on page 1-1376 for details.

## Methods

|                       |   |
|-----------------------|---|
| <code>generate</code> | Generates Blackman–Harris window                        |
| <code>info</code>     | Display information about Blackman–Harris window object |
| <code>winwrite</code> | Save Blackman–Harris window in ASCII file               |

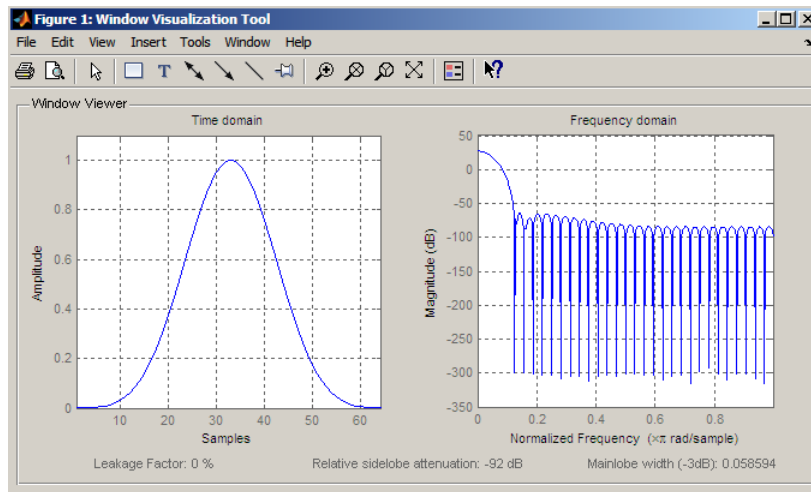
## Copy Semantics

Handle. To learn how copy semantics affect your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

## Examples

Default length  $N = 64$  Blackman-Harris window:

```
H = sigwin.blackmanharris;
wvtool(H)
```



Generate length  $N = 128$  periodic Blackman-Harris window, return values, and write ASCII file:

```
H = sigwin.blackmanharris(128);
H.SamplingFlag = 'periodic';
% Return window with generate
win = generate(H);
% Write ASCII file in current directory
% with window values
winwrite(H, 'blackmanharris_128')
```



## References

Harris, Fredric J. “On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform.” *Proceedings of the IEEE*. Vol. 66, January 1978, pp. 51–83.

## See Also

`blackmanharris` | `window` | `wvtool`

## How To

- Class Attributes
- Property Attributes

## generate

**Class:** sigwin.blackmanharris

**Package:** sigwin

Generates Blackman–Harris window

## Syntax

```
win = generate(H)
```

## Description

`win = generate(H)` returns the values of the Blackman–Harris window object `H` as a double-precision column vector.

## Examples

Extract values from Blackman–Harris window object:

```
H=sigwin.blackmanharris(128);  
% Extract window values as column vector  
win=generate(H);
```

---

## info

**Class:** sigwin.blackmanharris

**Package:** sigwin

Display information about Blackman–Harris window object

## Syntax

```
info(H)
info_win = info(H)
```

## Description

`info(H)` displays length information for the Blackman–Harris window object `H`.

`info_win = info(H)` returns length information for the Blackman–Harris window object `H` in the character array `info_win`.

## Examples

Return information about a Blackman–Harris window object:

```
H = sigwin.blackmanharris(256);
info_win = info(H);
```

## winwrite

**Class:** sigwin.blackmanharris

**Package:** sigwin

Save Blackman–Harris window in ASCII file

## Syntax

```
winwrite(H)
winwrite(H, 'filename')
```

## Description

`winwrite(H)` opens a dialog box that enables you to export the values of the Blackman–Harris window object `H` to an ASCII file with filename extension `wf`.

`winwrite(H, 'filename')` saves the values of the Blackman–Harris window object `H` in the current folder as a column vector in the ASCII file `'filename'` with filename extension `wf`.

## Examples

Write Blackman–Harris window values to ASCII file:

```
H=sigwin.blackmanharris;
% Open dialog box for ASCII file
winwrite(H);
```

# sigwin.bohmanwin class

**Package:** sigwin

Construct Bohman window object

## Description

---

**Note:** The use of `sigwin.bohmanwin` is not recommended. Use `bohmanwin` instead.

---

`sigwin.bohmanwin` creates a handle to a Bohman window object for use in spectral analysis and FIR filtering by the window method. Object methods enable workspace import and ASCII file export of the window values.

The following equation defines the Bohman window of length  $N$ :

$$w(x) = (1 - |x|) \cos(\pi |x|) + \frac{1}{\pi} \sin(\pi |x|), \quad -1 \leq x \leq 1$$

where  $x$  is a length  $N$  vector of linearly spaced values generated using `linspace`. The first and last elements of the Bohman window are forced to be identically zero.

## Construction

`H = sigwin.bohmanwin` returns a Bohman window object `H` of length 64.

`H = sigwin.bohmanwin(Length)` returns a Bohman window object `H` of length *Length*. *Length* is a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

## Properties

### Length

Bohman window length. Must be a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

## Methods

|          |  |
|----------|--|
| generate | Generates Bohman window                        |
| info     | Display information about Bohman window object |
| winwrite | Save Bohman window object values in ASCII file |

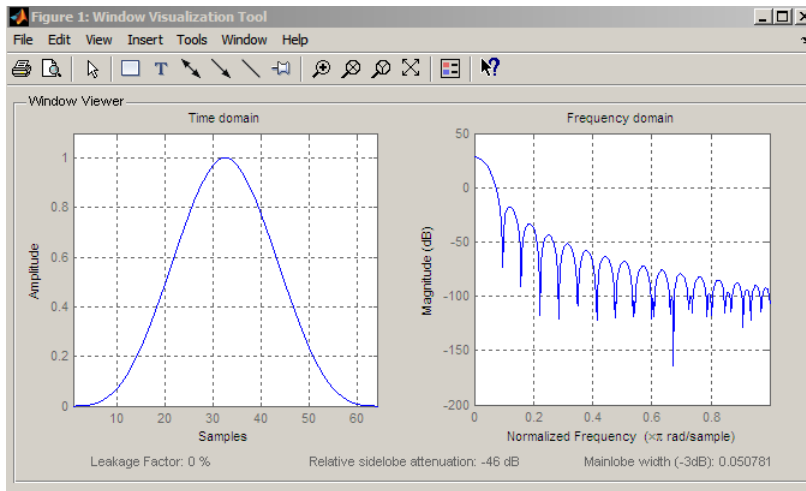
## Copy Semantics

Handle. To learn how copy semantics affect your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

## Examples

Default length  $N = 64$  Bohman window:

```
H = sigwin.bohmanwin;  
wvtool(H)
```



Generate length  $N = 128$  Bohman window, return values, and write ASCII file:

```
H = sigwin.bohmanwin(128);
% Return window with generate
win = generate(H);
% Write ASCII file in current directory
% with window values
winwrite(H, 'bohmanwin_128')
```

## References

Harris, Fredric J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform." *Proceedings of the IEEE*. Vol. 66, January 1978, pp. 51–83.

## See Also

bohmanwin | window | wvtool

## How To

- Class Attributes
- Property Attributes

## generate

**Class:** sigwin.bohmanwin

**Package:** sigwin

Generates Bohman window

## Syntax

```
win = generate(H)
```

## Description

`win = generate(H)` returns the values of the Bohman window object as a double-precision column vector.

## Examples

Extract values from Bohman window object:

```
H=sigwin.bohmanwin(128);  
% Extract window values as column vector  
win=generate(H);
```



## info

**Class:** sigwin.bohmanwin

**Package:** sigwin

Display information about Bohman window object

## Syntax

```
info(H)
info_win = info(H)
```

## Description

`info(H)` displays length information for the Bohman window object `H`.

`info_win = info(H)` returns length information for the Bohman window object `H` in the character array `info_win`.

## Examples

Return information for a Bohman window object:

```
H=sigwin.bohmanwin(256);
info_win=info(H);
```

## winwrite

**Class:** sigwin.bohmanwin

**Package:** sigwin

Save Bohman window object values in ASCII file

## Syntax

```
winwrite(H)
winwrite(H, 'filename')
```

## Description

`winwrite(H)` opens a dialog to export the values of the Bohman window object `H` to an ASCII file. The file extension `.wf` is automatically appended.

`winwrite(H, 'filename')` saves the values of the Bohman window object `H` in the current folder as a column vector in the ASCII file `'filename'`. The file extension `.wf` is automatically appended to `filename`.

## Examples

Write Bohman window values to ASCII file:

```
H=sigwin.bohmanwin;
% Open dialog box for ASCII file
winwrite(H);
```

## sigwin.chebwin class

**Package:** sigwin

Construct Dolph-Chebyshev window object

### Description

---

**Note:** The use of `sigwin.chebwin` is not recommended. Use `chebwin` instead.

---

`sigwin.chebwin` creates a handle to a Dolph-Chebyshev window object for use in spectral analysis and FIR filtering by the window method. Object methods enable workspace import and ASCII file export of the window values.

The Dolph-Chebyshev window is constructed in the frequency domain by taking samples of the window's Fourier transform:

$$\hat{W}(k) = (-1)^k \frac{\cos[N \cos^{-1}[\beta \cos(\pi k / N)]]}{\cosh[N \cosh^{-1}(\beta)]}, \quad 0 \leq k \leq N - 1$$

where

$$\beta = \cos[1 / N \cosh^{-1}(10^\alpha)]$$

$\alpha$  determines the level of the sidelobe attenuation. The level of the sidelobe attenuation is equal to  $-20\alpha$ . For example, 100 dB of attenuation results from setting  $\alpha = 5$

The discrete-time Dolph-Chebyshev window is obtained by taking the inverse DFT of  $\hat{W}(k)$  and scaling the result to have a peak value of 1.

### Construction

`H = sigwin.chebwin` returns a Dolph-Chebyshev window object `H` of length 64 with relative sidelobe attenuation of 100 dB.

`H = sigwin.chebwin(Length)` returns a Dolph-Chebyshev window object `H` of length *Length* with relative sidelobe attenuation of 100 dB. *Length* requires a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. A window length of 1 results in a window with a single value equal to 1.

`H = sigwin.chebwin(Length,SideLobeAtten)` returns a Dolph-Chebyshev window object with relative sidelobe attenuation of *atten\_param* dB.

## Properties

### Length

Dolph-Chebyshev window length.

### SideLobeAtten

The attenuation parameter in dB. The attenuation parameter is a positive real number that determines the relative sidelobe attenuation of the window.

## Methods

|                       |   |
|-----------------------|---|
| <code>generate</code> | Generates Dolph-Chebyshev window                        |
| <code>info</code>     | Display information about Dolph-Chebyshev window object |
| <code>winwrite</code> | Save Dolph-Chebyshev window object values in ASCII file |

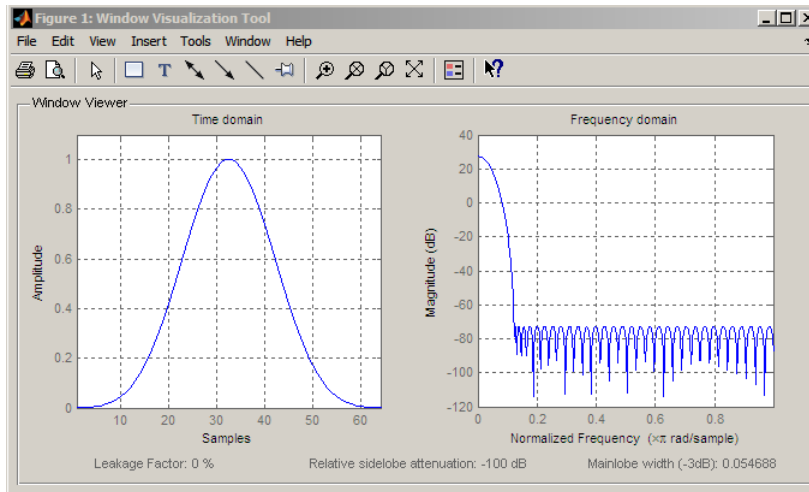
## Copy Semantics

Handle. To learn how copy semantics affect your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

## Examples

Default length  $N = 64$  Dolph-Chebyshev window with 100 dB relative sidelobe attenuation:

```
H = sigwin.chebwin;
wvtool(H)
```



Generate length  $N = 128$  Chebyshev window with 120 dB attenuation, return values, and write ASCII file:

```
H = sigwin.chebwin(128,120);
% Return window with generate
win = generate(H);
% Write ASCII file in current directory
% with window values
winwrite(H, 'chebwin_128_100')
```

## References

Harris, Fredric J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform." *Proceedings of the IEEE*. Vol. 66, January 1978, pp. 51–83.

## **See Also**

chebwin | window | wvtool

## **How To**

- Class Attributes
- Property Attributes

## generate

**Class:** sigwin.chebwin

**Package:** sigwin

Generates Dolph-Chebyshev window

## Syntax

```
win = generate(H)
```

## Description

`win = generate(H)` returns the values of the Dolph-Chebyshev window object `H` as a double-precision column vector.

## Examples

Extract values from Dolph-Chebyshev window object:

```
H=sigwin.chebwin(128);  
% Extract window values as column vector  
win=generate(H);
```

## info

**Class:** sigwin.chebwin

**Package:** sigwin

Display information about Dolph–Chebyshev window object

## Syntax

```
info(H)  
info_win = info(H)
```

## Description

`info(H)` displays length and relative sidelobe attenuation information for the Dolph-Chebyshev window object `H`.

`info_win = info(H)` returns length information for the Dolph-Chebyshev window object `H` in the character array `info_win`.

## Examples

Return information about a Dolph-Chebyshev window object:

```
H=sigwin.chebwin(256);  
info_win=info(H);
```



# winwrite

**Class:** sigwin.chebwin

**Package:** sigwin

Save Dolph-Chebyshev window object values in ASCII file

## Syntax

```
winwrite(H)
winwrite(H, 'filename')
```

## Description

`winwrite(H)` opens a dialog to export the values of the Dolph-Chebyshev window object `H` to an ASCII file. The file extension `.wf` is automatically appended.

`winwrite(H, 'filename')` saves the values of the Dolph-Chebyshev window object `H` in the current folder as a column vector in the ASCII file `'filename'`. The file extension `.wf` is automatically appended to `filename`.

## Examples

Write Dolph-Chebyshev window values to ASCII file:

```
H=sigwin.chebwin;
% Open dialog box for ASCII file
winwrite(H);
```

## sigwin.flattopwin class

**Package:** sigwin

Construct flat top window object

### Description

---

**Note:** The use of `sigwin.flattopwin` is not recommended. Use `flattopwin` instead.

---

`sigwin.flattopwin` creates a handle to a flat top window object for use in spectral analysis and FIR filtering by the window method. Object methods enable workspace import and ASCII file export of the window values.

### Construction

`H = sigwin.flattopwin` returns a flat top window object `H` of length 64 with symmetric sampling.

`H = sigwin.flattopwin(Length)` returns a flat top window object of length *Length* with symmetric sampling. *Length* must be a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

`H = sigwin.flattopwin(Length, SamplingFlag)` returns a flat top window object `H` of length *Length* with sampling *SamplingFlag*. The *SamplingFlag* can be either 'symmetric' or 'periodic'.

### Properties

#### Length

Flat top window length. Must be a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

## SamplingFlag

'symmetric' is the default and forces exact symmetry between the first and second halves of the flat top window. A symmetric window is preferred in FIR filter design.

'periodic' designs a symmetric flat top window of length  $Length+1$  and truncates the window to length  $Length$ . This design is preferred in spectral analysis where the window is treated as one period of a  $Length$ -point periodic sequence.

## Methods

|          |  |
|----------|--|
| generate | Generates flat top window                        |
| info     | Display information about flat top window object |
| winwrite | Save flat top window in ASCII file               |

## Definitions

The following equation defines the flat top window of length  $N$ :

$$w(n) = a_0 - a_1 \cos(2\pi n / (N - 1)) + a_2 \cos(4\pi n / (N - 1)) - a_3 \cos(6\pi n / (N - 1)) + a_4 \cos(8\pi n / (N - 1)),$$

where  $M$  is  $N/2$  for  $N$  even and  $(N+1)/2$  for  $N$  odd.

The second half of the symmetric flat top window  $M \leq n \leq N - 1$  is obtained by flipping the first half around the midpoint. The symmetric option is the preferred method when using a flat top window in FIR filter design by the window method.

The periodic flat top window is constructed by extending the desired window length by one sample, constructing a symmetric window, and removing the last sample. The periodic version is the preferred method when using a flat top window in spectral analysis because the discrete Fourier transform assumes periodic extension of the input vector.

The coefficients are listed in the following table:

| Coefficient | Value       |
|-------------|-------------|
| a0          | 0.21557895  |
| a1          | 0.41663158  |
| a2          | 0.277263158 |
| a3          | 0.083578947 |
| a4          | 0.006947368 |

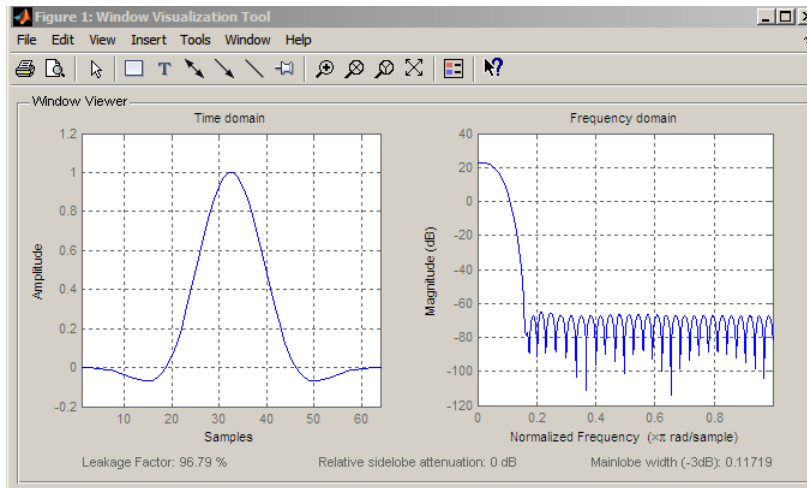
## Copy Semantics

Handle. To learn how copy semantics affect your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

## Examples

Default length  $N = 64$  symmetric flat top window:

```
H = sigwin.flattopwin;
wvtool(H)
```



Generate length  $N = 128$  periodic flat top window, return values, and write ASCII file:

```
H = sigwin.flattopwin(128,'periodic');  
% Return window with generate  
win = generate(H);  
% Write ascii file in current directory  
% with window values  
winwrite(H,'flattopwin_128')
```

## References

Oppenheim, Alan V., and Ronald W. Schaffer. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1989.

## See Also

[flattopwin](#) | [window](#) | [wvtool](#)

## How To

- [Class Attributes](#)
- [Property Attributes](#)

## generate

**Class:** sigwin.flattopwin

**Package:** sigwin

Generates flat top window

## Syntax

```
win = generate(H)
```

## Description

`win = generate(H)` returns the values of the flat top window object as a double-precision column vector.

## Examples

Extract values from flat top window object:

```
H=sigwin.flattopwin(128);  
% Extract window values as column vector  
win=generate(H);
```

## info

**Class:** sigwin.flattopwin

**Package:** sigwin

Display information about flat top window object

## Syntax

```
info(H)  
info_win = info(H)
```

## Description

`info(H)` displays length and sampling information for the flat top window object `H`.

`info_win = info(H)` returns length and sampling information for the flat top window object `H` in the character array `info_win`.

## Examples

Return information about a flat top window object:

```
H=sigwin.flattopwin(256);  
info_win=info(H);
```

## winwrite

**Class:** sigwin.flattopwin

**Package:** sigwin

Save flat top window in ASCII file

## Syntax

```
winwrite(H)
winwrite(H, 'filename')
```

## Description

`winwrite(H)` opens a dialog to export the flat top window values to an ASCII file. The file extension `.wf` is automatically appended.

`winwrite(H, 'filename')` saves the values of the flat top window object `H` in the current folder as a column vector in the ASCII file `'filename'`. The file extension `.wf` is automatically appended to `filename`.

## Examples

Write flat top window values to ASCII file:

```
H=sigwin.flattopwin;
% Open dialog for ASCII file
winwrite(H);
```



# sigwin.gausswin class

**Package:** sigwin

Construct Gaussian window object

## Description

---

**Note:** The use of `sigwin.gausswin` is not recommended. Use `gausswin` instead.

---

`sigwin.gausswin` creates a handle to a Gaussian window object for use in spectral analysis and FIR filtering by the window method. Object methods enable workspace import and ASCII file export of the window values.

The following equation defines the Gaussian window of length  $N$ :

$$w(x) = e^{-\frac{1}{2}(\alpha^2 x^2 / M^2)}, \quad -M \leq x \leq M$$

where  $M = (N - 1) / 2$  and  $x$  is a linearly spaced vector of length  $N$ .

Equating  $\alpha$  with the usual standard deviation of a Gaussian value,  $\sigma$ , note:

$$\alpha = \frac{(N - 1)}{2\sigma}$$

## Construction

`H = sigwin.gausswin` returns a Gaussian window object `H` of length 64 and dispersion parameter *alpha* of 2.5.

`H = sigwin.gausswin(Length)` returns a Gaussian window object `H` of length *Length* and dispersion parameter *alpha* of 2.5. *Length* requires a positive integer. Entering a

positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

`H = sigwin.gausswin(Length,Alpha)` returns a Gaussian window object with dispersion parameter *alpha*. *alpha* requires a nonnegative real number and is inversely proportional to the standard deviation of a Gaussian value.

## Properties

### Length

Gaussian window length. The window length requires a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

### Alpha

Width of Gaussian window. *Alpha* is inversely proportional to the standard deviation of a Gaussian. Larger values of *Alpha* produce Gaussian windows with inflection points closer to the peak value, or narrower windows. In the frequency domain, larger values of *Alpha* produce a Gaussian window with increased spread of the main lobe in frequency but decreased sidelobe energy.

## Methods

|          |  |
|----------|--|
| generate | Generates Gaussian window                        |
| info     | Display information about Gaussian window object |
| winwrite | Save Gaussian window in ASCII file               |

## Copy Semantics

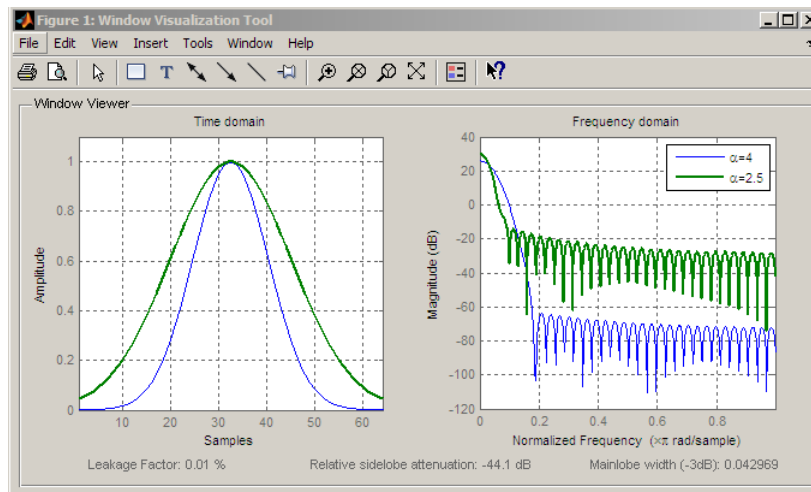
Handle. To learn how copy semantics affect your use of the class, see [Copying Objects in the MATLAB Programming Fundamentals documentation](#).

## Examples

Compare two Gaussian windows with different  $\alpha$  values:

```
H = sigwin.gausswin(64,4);
H1 = sigwin.gausswin(64,2.5);
% Plot comparison
fwvt = wvtool(H,H1);
legend(get(fwvt,'currentaxes'),'alpha=4','alpha=2.5');
```

The main lobe is wider for  $\alpha = 4$  but the window, with  $\alpha = 4$ , demonstrates reduced sidelobe energy.



## References

Harris, Fredric J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform." *Proceedings of the IEEE*. Vol. 66, January 1978, pp. 51–83.

## See Also

gausswin | window | wvtool

## How To

- Class Attributes

- Property Attributes

## generate

**Class:** sigwin.gausswin

**Package:** sigwin

Generates Gaussian window

## Syntax

```
win = generate(H)
```

## Description

`win = generate(H)` returns the values of the Gaussian window object `H` as a double-precision column vector.

## Examples

Extract values from Gaussian window object:

```
H=sigwin.gausswin(128,4);  
% Extract window values as column vector  
win=generate(H);
```

## info

**Class:** sigwin.gausswin

**Package:** sigwin

Display information about Gaussian window object

## Syntax

```
info(H)  
info_win = info(H)
```

## Description

`info(H)` displays length and dispersion information for the Gaussian window object `H`.

`info_win = info(H)` returns length and dispersion information for the Gaussian window object `H` in the character array `info_win`.

## Examples

Return information about a Gaussian window object:

```
H=sigwin.gausswin(256);  
info_win=info(H);
```

# winwrite

**Class:** sigwin.gausswin

**Package:** sigwin

Save Gaussian window in ASCII file

## Syntax

```
winwrite(H)
winwrite(H, 'filename')
```

## Description

`winwrite(H)` opens a dialog to export the values of Gaussian window object `H` to an ASCII file. The file extension `.wf` is automatically appended.

`winwrite(H, 'filename')` saves the values of the Gaussian window object `H` in the current folder as a column vector in the ASCII file `'filename'`. The file extension `.wf` is automatically appended to `filename`.

## Examples

Write Gaussian window values to ASCII file:

```
H=sigwin.gausswin;
% Open dialog for ASCII file
winwrite(H);
```

## sigwin.hamming class

**Package:** sigwin

Construct Hamming window object

### Description

---

**Note:** The use of `sigwin.hamming` is not recommended. Use `hamming` instead.

---

`sigwin.hamming` creates a handle to a Hamming window object for use in spectral analysis and FIR filtering by the `window` method. Object methods enable workspace import and ASCII file export of the window values.

The following equation defines the Hamming window of length  $N$ :

$$w(n) = 0.54 - 0.46 \cos \frac{2\pi n}{N-1}, \quad 0 \leq n \leq M-1$$

where  $M$  is  $N/2$  for  $N$  even and  $(N+1)/2$  for  $N$  odd.

The second half of the symmetric Hamming window  $M \leq n \leq N-1$  is obtained by flipping the first half around the midpoint. The symmetric option is the preferred method when using a Hamming window in FIR filter design.

The periodic Hamming window is constructed by extending the desired window length by one sample, constructing a symmetric window, and removing the last sample. The periodic version is the preferred method when using a Hamming window in spectral analysis because the discrete Fourier transform assumes periodic extension of the input vector.

### Construction

`H = sigwin.hamming` returns a symmetric Hamming window object `H` of length 64.



`H = sigwin.hamming(Length)` returns a symmetric Hamming window object with length *Length*. *Length* must be a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

`H = sigwin.hamming(Length, SamplingFlag)` returns a Hamming window with sampling *SamplingFlag*. The *SamplingFlag* can be either 'symmetric' or 'periodic'.

## Properties

### Length

Hamming window length. The window length must be a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

### SamplingFlag

'symmetric' is the default and forces exact symmetry between the first and second halves of the Hamming window. A symmetric window is preferred in FIR filter design by the window method.

'periodic' designs a symmetric Hamming window of length *Length*+1 and truncates the window to length *Length*. This design is preferred in spectral analysis where the window is treated as one period of a *Length*-point periodic sequence.

## Methods

generate

Generates Hamming window

info

Display information about Hamming window object

winwrite

Save Hamming window in ASCII file

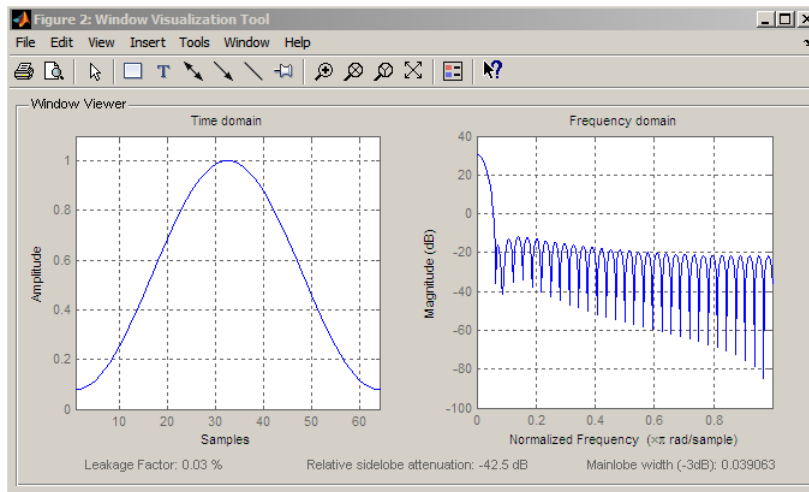
## Copy Semantics

Handle. To learn how copy semantics affect your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

## Examples

Default length  $N = 64$  symmetric Hamming window:

```
H = sigwin.hamming;
wvtool(H)
```



Generate a length  $N = 128$  periodic Hamming window, return the values, and write ASCII file:

```
H = sigwin.hamming(128,'periodic');
% Return window values with generate
win = generate(H);
% Write ASCII file in current directory
% with window values
winwrite(H,'hamming_128')
```

## References

Oppenheim, Alan V., and Ronald W. Schaffer. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1989.

## See Also

hamming | window | wvtool

## How To

- Class Attributes
- Property Attributes

## **generate**

**Class:** sigwin.hamming

**Package:** sigwin

Generates Hamming window

## **Syntax**

```
win = generate(H)
```

## **Description**

`win = generate(H)` returns the values of the Hamming window object as a double-precision column vector.

## **Examples**

Extract values from Hamming window object:

```
H=sigwin.hamming(128);  
% Extract window values as column vector  
win=generate(H);
```

## info

**Class:** sigwin.hamming

**Package:** sigwin

Display information about Hamming window object

## Syntax

```
info(H)  
info_win = info(H)
```

## Description

`info(H)` displays length and sampling information for the Hamming window object `H`.

`info_win = info(H)` returns length and sampling information for the Hamming window object `H` in the character array `info_win`.

## Examples

Return information about a Hamming window object:

```
H=sigwin.hamming(256);  
info_win=info(H);
```

## winwrite

**Class:** sigwin.hamming

**Package:** sigwin

Save Hamming window in ASCII file

## Syntax

```
winwrite(H)
winwrite(H, 'filename')
```

## Description

`winwrite(H)` opens a dialog to export the Hamming window values to an ASCII file. The file extension `.wf` is automatically appended.

`winwrite(H, 'filename')` saves the values of the Hamming window object `H` in the current folder as a column vector in the ASCII file `'filename'`. The file extension `.wf` is automatically appended to `filename`.

## Examples

Write Hamming window values to ASCII file:

```
H=sigwin.hamming;
% Open dialog box for ASCII file
winwrite(H);
```

# sigwin.hann class

**Package:** sigwin

Construct Hann (Hanning) window object

## Description

---

**Note:** The use of `sigwin.hann` is not recommended. Use `hann` instead.

---

`sigwin.hann` creates a handle to a Hann window object for use in spectral analysis and FIR filtering by the window method. Object methods enable workspace import and ASCII file export of the window values.

The symmetric Hann window of length  $N$  is defined as:

$$w(n) = \frac{1}{2} \left( 1 - \cos \frac{2\pi n}{N-1} \right), \quad 0 \leq n \leq M-1$$

where  $M$  is  $N/2$  for  $N$  even and  $(N+1)/2$  for  $N$  odd.

The second half of the symmetric Hann window  $M \leq n \leq N-1$  is obtained by flipping the first half around the midpoint. The symmetric option is the preferred method when using a Hann window in FIR filter design.

The periodic Hann window is constructed by extending the desired window length by one sample, constructing a symmetric window, and removing the last sample. The periodic version is the preferred method when using a Hann window in spectral analysis because the discrete Fourier transform assumes periodic extension of the input vector.

## Construction

`H = sigwin.hann` returns a symmetric Hann window object `H` of length 64.

`H = sigwin.hann(Length)` returns a symmetric Hann window object with length *Length*. *Length* requires a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

`H = sigwin.hann(Length, SamplingFlag)` returns a Hann window object with sampling *SamplingFlag*. The *SamplingFlag* can be either 'symmetric' or 'periodic'.

## Properties

### Length

Hann window length. Must be a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

### SamplingFlag

'symmetric' is the default and forces exact symmetry between the first and second halves of the Hann window. A symmetric window is preferred in FIR filter design by the window method.

'periodic' designs a symmetric Hann window of length *Length*+1 and truncates the window to length *Length*. This design is preferred in spectral analysis where the window is treated as one period of a *Length*-point periodic sequence.

## Methods

generate

Generates Hann window

info

Display information about Hann window object

winwrite

Save Hann window object values in ASCII file



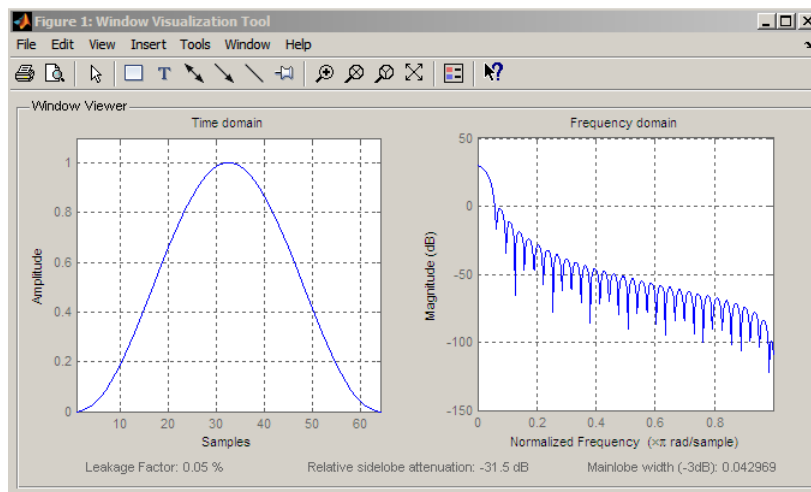
## Copy Semantics

Handle. To learn how copy semantics affect your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

## Examples

Default length  $N = 64$  symmetric Hann window:

```
H = sigwin.hann;
wvtool(H)
```



Generate length  $N=128$  periodic Hann window, return values, and write ASCII file:

```
H = sigwin.hann(128, 'periodic');
% Return window with generate
win = generate(H);
% Write ASCII file in current directory
% with window values
winwrite(H, 'hann_128')
```

## References

Oppenheim, Alan V., and Ronald W. Schaffer. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1989.

## See Also

`hann` | `window` | `wvtool`

## How To

- Class Attributes
- Property Attributes

## generate

**Class:** sigwin.hann

**Package:** sigwin

Generates Hann window

## Syntax

```
win = generate(H)
```

## Description

`win = generate(H)` returns the values of the Hann window object `H` as a double-precision column vector.

## Examples

Extract values from Hann window object:

```
H=sigwin.hann(128);  
% Extract window values as column vector  
win=generate(H);
```

## info

**Class:** sigwin.hann

**Package:** sigwin

Display information about Hann window object

## Syntax

```
info(H)  
info_win = info(H)
```

## Description

`info(H)` displays length and sampling information for the Hann window object `H`.

`info_win = info(H)` returns length and sampling information for the Hann window object `H` in the character array `info_win`.

## Examples

Return information about a Hann window object:

```
H=sigwin.hann(256);  
info_win=info(H);
```

# winwrite

**Class:** sigwin.hann

**Package:** sigwin

Save Hann window object values in ASCII file

## Syntax

```
winwrite(H)
winwrite(H, 'filename')
```

## Description

`winwrite(H)` opens a dialog to export the values of the Hann window object `H` to an ASCII file. The file extension `.wf` is automatically appended.

`winwrite(H, 'filename')` saves the values of the Hann window object `H` in the current folder as a column vector in the ASCII file `'filename'`. The file extension `.wf` is automatically appended to `filename`.

## Examples

Write Hann window values to ASCII file:

```
H=sigwin.hann;
% Open dialog box for ASCII file
winwrite(H);
```

## sigwin.kaiser class

**Package:** sigwin

Construct Kaiser window object

### Description

---

**Note:** The use of `sigwin.kaiser` is not recommended. Use `kaiser` instead.

---

`sigwin.kaiser` creates a handle to a Kaiser window object for use in spectral analysis and FIR filtering by the `window` method. Object methods enable workspace import and ASCII file export of the window values.

The following equation defines the Kaiser window of length  $N$ :

$$w(x) = I_0\left(\beta\sqrt{1 - \frac{4x^2}{(N-1)^2}}\right) / I_0(\beta), \quad -(N-1)/2 \leq x \leq (N-1)/2$$

where  $x$  is linearly spaced  $N$ -point vector and  $I_0()$  is the modified zero-th order Bessel function of the first kind.  $\beta$  is the attenuation parameter.

### Construction

`H = sigwin.kaiser` returns a Kaiser window object `H` of length 64 and attenuation parameter `beta` of 0.5.

`H = sigwin.kaiser(Length)` returns a Kaiser window object `H` of length *Length* and attenuation parameter `beta` of 0.5. *Length* requires a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

`H = sigwin.kaiser(Length,Beta)` returns a Kaiser window object with real-valued attenuation parameter `beta`.

## Properties

### Length

Kaiser window length. The window length requires a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

### Beta

Attenuation parameter. **Beta** requires a real number. Larger absolute values of **Beta** result in greater stopband attenuation, or equivalently greater attenuation between the main lobe and first side lobe.

## Methods

|          |  |
|----------|--|
| generate | Generates Kaiser window                        |
| info     | Display information about Kaiser window object |
| winwrite | Save Kaiser window in ASCII file               |

## Copy Semantics

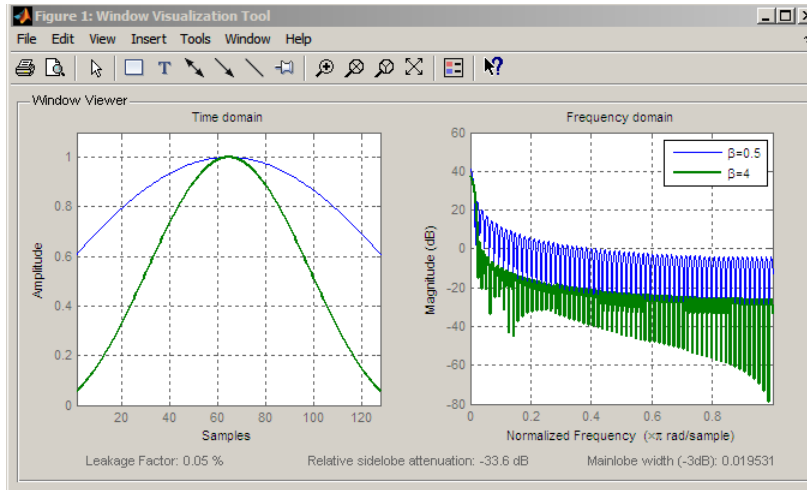
Handle. To learn how copy semantics affect your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

## Examples

Compare two Kaiser windows with different **Beta** values:

```
H = sigwin.kaiser(128,1.5);  
% Kaiser window with Beta=4.5  
H1 = sigwin.kaiser(128,4.5);
```

```
% Plot comparison
fwvt = wvtool(H,H1);
legend(get(fwvt, 'currentaxes'), '\beta=1.5', '\beta=4.5');
```



## References

Oppenheim, Alan V., and Ronald W. Schaffer. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1989.

## See Also

besseli | kaiser | window | wvtool |

## How To

- Class Attributes
- Property Attributes



# generate

**Class:** sigwin.kaiser

**Package:** sigwin

Generates Kaiser window

## Syntax

```
win = generate(H)
```

## Description

`win = generate(H)` returns the values of the Kaiser window object as a double-precision column vector.

## Examples

Extract values from Kaiser window object:

```
H=sigwin.kaiser(128,4);  
% Extract window values as column vector  
win=generate(H);
```

## info

**Class:** sigwin.kaiser

**Package:** sigwin

Display information about Kaiser window object

## Syntax

```
info(H)  
info_win = info(H)
```

## Description

`info(H)` displays length and attenuation information for the Kaiser window object `H`.

`info_win = info(H)` returns length and attenuation information for the Kaiser window object `H` in the character array `info_win`.

## Examples

Return information about a Kaiser window object:

```
H=sigwin.kaiser(256);  
info_win=info(H);
```

# winwrite

**Class:** sigwin.kaiser

**Package:** sigwin

Save Kaiser window in ASCII file

## Syntax

```
winwrite(H)
winwrite(H, 'filename')
```

## Description

`winwrite(H)` opens a dialog to export the Kaiser window values to an ASCII file. The file extension `.wf` is automatically appended.

`winwrite(H, 'filename')` saves the values of the Kaiser window object `H` in the current folder as a column vector in the ASCII file `'filename'`. The file extension `.wf` is automatically appended to `filename`.

## Examples

Write Kaiser window values to ASCII file:

```
H=sigwin.kaiser;
% Open dialog box for ASCII file
winwrite(H);
```

## sigwin.nuttallwin class

**Package:** sigwin

Construct Nuttall defined 4–term Blackman-Harris window object

### Description

---

**Note:** The use of `sigwin.nuttallwin` is not recommended. Use `nuttallwin` instead.

---

`sigwin.nuttallwin` creates a handle to a Nuttall defined 4–term Blackman-Harris window object for use in spectral analysis and FIR filtering by the `window` method. Object methods enable workspace import and ASCII file export of the window values.

### Construction

`H = sigwin.nuttallwin` returns a Nuttall defined 4–term Blackman-Harris window object `H` of length 64.

`H = sigwin.nuttallwin(Length)` returns a Nuttall defined 4–term Blackman-Harris window object `H` of length `Length`. Entering a positive noninteger value for `Length` rounds the length to the nearest integer. Entering a 1 for `Length` results in a window with a single value of 1. The `SamplingFlag` property defaults to `'symmetric'`.

### Properties

#### Length

Nuttall defined 4–term Blackman-Harris window length. The window length must be a positive integer. Entering a positive noninteger value for `Length` rounds the length to the nearest integer. Entering a 1 for `Length` results in a window with a single value of 1.

#### SamplingFlag

The type of window returned as one of `'symmetric'` or `'periodic'`. The default is `'symmetric'`. A symmetric window exhibits perfect symmetry between halves of the

window. Setting the `SamplingFlag` property to 'periodic' results in a N-periodic window. The equations for the Nuttall defined 4-term Blackman-Harris window differ slightly based on the value of the `SamplingFlag` property. See “Definitions” on page 1-1431 for details.

## Methods

|                       |  |
|-----------------------|--|
| <code>generate</code> | Generates Nuttall defined 4-term Blackman-Harris window                        |
| <code>info</code>     | Display information about Nuttall defined 4-term Blackman-Harris window object |
| <code>winwrite</code> | Save Nuttall defined 4-term Blackman-Harris window object values in ASCII file |

## Definitions

The following equation defines the symmetric Nuttall defined 4-term Blackman-Harris window of length  $N$ .

$$w(n) = a_0 - a_1 \cos\left(\frac{2\pi n}{N-1}\right) + a_2 \cos\left(\frac{4\pi n}{N-1}\right) - a_3 \cos\left(\frac{6\pi n}{N-1}\right), \quad 0 \leq n \leq N-1$$

The following equation defines the periodic Nuttall defined 4-term Blackman-Harris window of length  $N$ .

$$w(n) = a_0 - a_1 \cos\frac{2\pi n}{N} + a_2 \cos\frac{4\pi n}{N} - a_3 \cos\frac{6\pi n}{N}, \quad 0 \leq n \leq N-1$$

The following table lists the coefficients:

| Coefficient | Value     |
|-------------|-----------|
| $a_0$       | 0.3635819 |
| $a_1$       | 0.4891775 |

| Coefficient | Value     |
|-------------|-----------|
| a2          | 0.1365995 |
| a3          | 0.0106411 |

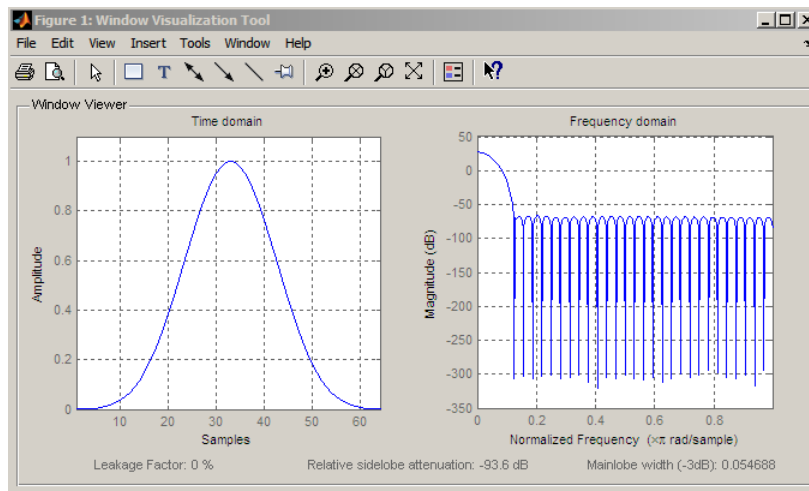
## Copy Semantics

Handle. To learn how copy semantics affect your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

## Examples

Construct a length  $N = 64$  symmetric Nuttall defined 4-term Blackman-Harris window:

```
H = sigwin.nuttallwin;
wvtool(H)
```



Generate a length  $N = 128$  periodic Nuttall defined 4-term Blackman-Harris window, return values, and write ASCII file:

```
H = sigwin.nuttallwin(128);
H.SamplingFlag = 'periodic';
```

```
% Return window with generate  
win = generate(H);  
% Write ASCII file in current directory  
% with window values  
winwrite(H,'nuttallwin_128')
```

## References

Nuttall, A. H. “Some Windows with Very Good Sidelobe Behavior.” *IEEE Transactions on Acoustics, Speech, and Signal Processing*. Vol. 29, 1981, pp. 84–91.

## See Also

nuttallwin | window | wvtool

## How To

- Class Attributes
- Property Attributes

## generate

**Class:** sigwin.nuttallwin

**Package:** sigwin

Generates Nuttall defined 4-term Blackman-Harris window

## Syntax

```
win = generate(H)
```

## Description

`win = generate(H)` returns the values of the Nuttall defined 4-term Blackman-Harris window object as a double-precision column vector.

## Examples

Extract values from Nuttall defined 4-term Blackman-Harris window object:

```
H=sigwin.nuttallwin(128);  
% Extract window values as column vector  
win=generate(H);
```



## info

**Class:** sigwin.nuttallwin

**Package:** sigwin

Display information about Nuttall defined 4-term Blackman-Harris window object

## Syntax

```
info(H)  
info_win = info(H)
```

## Description

`info(H)` displays length information about the Nuttall defined 4-term Blackman-Harris window object `H`.

`info_win = info(H)` returns length information about the Nuttall defined 4-term Blackman-Harris window object `H` in the character array `info_win`.

## Examples

Return information about Nuttall defined 4-term Blackman-Harris window object:

```
H=sigwin.nuttallwin(256);  
info_win=info(H);
```

## winwrite

**Class:** sigwin.nuttallwin

**Package:** sigwin

Save Nuttall defined 4-term Blackman-Harris window object values in ASCII file

## Syntax

```
winwrite(H)
winwrite(H, 'filename')
```

## Description

`winwrite(H)` opens a dialog to export the values of the Nuttall defined 4-term Blackman-Harris window object `H` to an ASCII file. The file extension `.wf` is automatically appended.

`winwrite(H, 'filename')` saves the values of the Nuttall defined 4-term Blackman-Harris window object `H` in the current folder as a column vector in the ASCII file `'filename'`. The file extension `.wf` is automatically appended to `filename`.

## Examples

Write Nuttall defined 4-term Blackman-Harris window values to ASCII file:

```
H=sigwin.nuttallwin;
% Open dialog box for ASCII file
winwrite(H);
```

# sigwin.parzenwin class

**Package:** sigwin

Construct Parzen window object

## Description

---

**Note:** The use of `sigwin.parzenwin` is not recommended. Use `parzenwin` instead.

---

`sigwin.parzenwin` creates a handle to a Parzen window object for use in spectral analysis and FIR filtering by the window method. Object methods enable workspace import and ASCII file export of the window values.

The following equation defines the  $N$ -point Parzen window over the interval

$$-\frac{(N-1)}{2} \leq n \leq \frac{(N-1)}{2} ;$$

$$w(n) = \begin{cases} 1 - 6\left(\frac{|n|}{N/2}\right)^2 + 6\left(\frac{|n|}{N/2}\right)^3, & 0 \leq |n| \leq (N-1)/4 \\ 2\left(1 - \frac{|n|}{N/2}\right)^3, & (N-1)/4 < |n| \leq (N-1)/2 \end{cases}$$

## Construction

`H = sigwin.parzenwin` returns a Parzen window object `H` of length 64.

`H = sigwin.parzenwin(Length)` returns a Parzen window object `H` of length *Length*. *Length* requires a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

## Properties

### Length

*Length* requires a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

## Methods

generate

Generate Parzen window

info

Display information about Parzen window object

winwrite

Save Parzen window in ASCII file

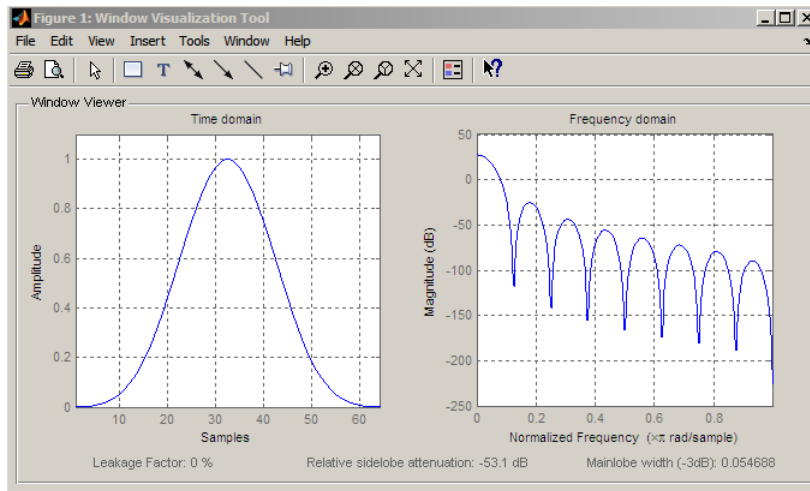
## Copy Semantics

Handle. To learn how copy semantics affect your use of the class, see [Copying Objects in the MATLAB Programming Fundamentals documentation](#).

## Examples

Default length  $N = 64$  Parzen window:

```
H = sigwin.parzenwin;  
wvtool(H)
```



Generate length  $N = 128$  Parzen window object, return values, and write ASCII file:

```
H = sigwin.parzenwin(128);
% Return window with generate
win = generate(H);
% Write ascii file in current directory
% with window values
winwrite(H, 'parzenwin_128')
```

## References

Harris, Fredric J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform." *Proceedings of the IEEE*. Vol. 66, January 1978, pp. 51–83.

## See Also

parzenwin | window | wvtool

## How To

- Class Attributes
- Property Attributes

## generate

**Class:** sigwin.parzenwin

**Package:** sigwin

Generate Parzen window

## Syntax

```
win = generate(H)
```

## Description

`win = generate(H)` returns the values of the Parzen window object as a double-precision column vector.

## Examples

Extract values from Parzen window object:

```
H=sigwin.parzenwin(128);  
% Extract window values as column vector  
win=generate(H);
```

## info

**Class:** sigwin.parzenwin

**Package:** sigwin

Display information about Parzen window object

## Syntax

```
info(H)  
info_win=info(H)
```

## Description

`info(H)` displays length information about the Parzen window object `H`.

`info_win=info(H)` returns length information about the Parzen window object `H` in the character array `info_win`.

## Examples

Return information about a Parzen window object:

```
% 256-point Parzen window  
H=sigwin.parzenwin(256);  
info_win=info(H);
```

## winwrite

**Class:** sigwin.parzenwin

**Package:** sigwin

Save Parzen window in ASCII file

## Syntax

```
winwrite(H)
winwrite(H, 'filename')
```

## Description

`winwrite(H)` opens a dialog to export the values of the Parzen window object `H` to an ASCII file. The file extension `.wf` is automatically appended.

`winwrite(H, 'filename')` saves the values of the Parzen window object `H` in the current folder as a column vector in the ASCII file `'filename'`. The file extension `.wf` is automatically appended to `filename`.

## Examples

Write Parzen window values to ASCII file:

```
H=sigwin.parzenwin;
% Open dialog box for ASCII file
winwrite(H);
```



# sigwin.rectwin class

**Package:** sigwin

Construct rectangular window object

## Description

---

**Note:** The use of `sigwin.rectwin` is not recommended. Use `rectwin` instead.

---

`sigwin.rectwin` creates a handle to a rectangular window object for use in spectral analysis and FIR filtering by the window method. Object methods enable workspace import and ASCII file export of the window values.

The following equation defines the rectangular window of length  $N$ :

$$w(n) = 1, \quad 0 \leq n \leq N - 1$$

## Construction

`H = sigwin.rectwin` returns a rectangular window object `H` of length 64.

`H = sigwin.rectwin(Length)` returns a rectangular window object `H` of length *Length*. *Length* requires a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

## Properties

### Length

Rectangular window length. The window length requires a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

## Methods

|          |   |
|----------|---|
| generate | Generates rectangular window                        |
| info     | Display information about rectangular window object |
| winwrite | Save rectangular window in ASCII file               |

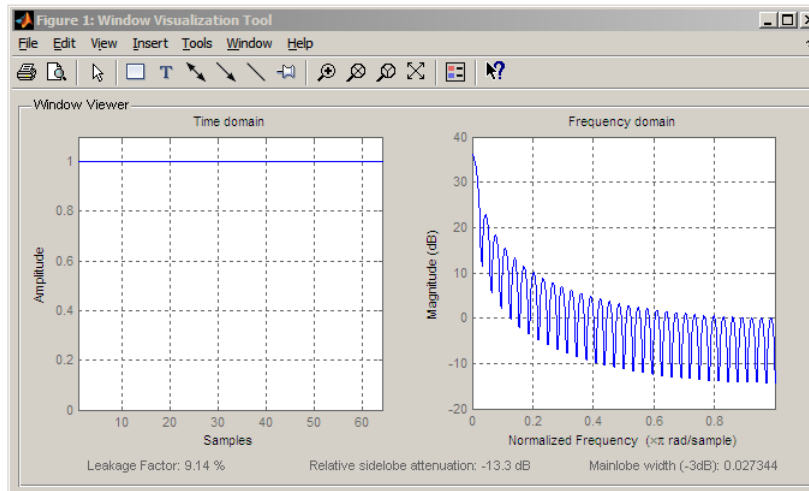
## Copy Semantics

Handle. To learn how copy semantics affect your use of the class, see [Copying Objects in the MATLAB Programming Fundamentals documentation](#).

## Examples

Create default length  $N = 64$  rectangular window:

```
H = sigwin.rectwin;
wvtool(H)
```



Generate length  $N = 128$  rectangular window, return values, and write ASCII file:

```
H = sigwin.rectwin(128);  
% Return window with generate  
win = generate(H);  
% Write ascii file in current directory  
% with window values  
winwrite(H, 'rectwin_128')
```

## References

Oppenheim, Alan V., and Ronald W. Schaffer. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1989.

## See Also

rectwin | window | wvtool

## How To

- Class Attributes
- Property Attributes

## generate

**Class:** sigwin.rectwin

**Package:** sigwin

Generates rectangular window

## Syntax

```
win = generate(H)
```

## Description

`win = generate(H)` returns the values of the rectangular window object `H` as a double-precision column vector.

## Examples

Extract values from rectangular window object:

```
H=sigwin.rectwin(128);  
% Extract window values as column vector  
win=generate(H);
```

## info

**Class:** sigwin.rectwin

**Package:** sigwin

Display information about rectangular window object

## Syntax

```
info(H)  
info_win = info(H)
```

## Description

`info(H)` displays length information for the rectangular window object `H`.

`info_win = info(H)` returns length information for the rectangular window object `H` in the character array `info_win`.

## Examples

Return information about a rectangular window object:

```
H=sigwin.rectangular(256);  
info_win=info(H);
```

## winwrite

**Class:** sigwin.rectwin

**Package:** sigwin

Save rectangular window in ASCII file

## Syntax

```
winwrite(H)
winwrite(H, 'filename')
```

## Description

`winwrite(H)` opens a dialog to export the values of the rectangular window object `H` to an ASCII file. The file extension `.wf` is automatically appended.

`winwrite(H, 'filename')` saves the values of the rectangular window object `H` in the current folder as a column vector in the ASCII file `'filename'`. The file extension `.wf` is automatically appended to `filename`.

## Examples

Write rectangular window values to ASCII file:

```
H=sigwin.rectwin;
% Open dialog box for ASCII file
winwrite(H);
```

# sigwin.taylorwin class

**Package:** sigwin

Construct Taylor window object

## Description

---

**Note:** The use of `sigwin.taylorwin` is not recommended. Use `taylorwin` instead.

---

`sigwin.taylorwin` creates a handle to a Taylor window object for use in spectral analysis and FIR filtering by the `window` method. Object methods enable workspace import and ASCII file export of the window values.

Taylor windows are similar to Dolph-Chebyshev windows. The Taylor window approximates the minimization of the main lobe width in the Dolph-Chebyshev window, but allows the sidelobe levels to decrease beyond a certain frequency. Taylor windows are typically used in radar applications, such as weighting synthetic aperture radar images and antenna design.

## Construction

`H = sigwin.taylorwin` returns a Taylor window object `H` of length 64, with a maximum sidelobe level of 30 dB and 4 constant-level sidelobes adjacent to the main lobe.

`H = sigwin.taylorwin(Length)` returns a Taylor window object `H` of length *Length* with a maximum sidelobe level of 30 dB and 4 constant-level sidelobes adjacent to the main lobe. *Length* must be a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

`H = sigwin.taylorwin(Length,Nbar)` returns a Taylor window object with *Nbar* nearly constant-level sidelobes adjacent to the main lobe. *Nbar* must be a positive integer.

`H = sigwin.taylorwin(Length,Nbar,SideLobeLevel)` returns a Taylor window object with a maximum sidelobe level *SideLobeLevel* dB below the main lobe level.

## Properties

### Length

Taylor window length. The window length must be a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

### Nbar

Number of nearly constant-level sidelobes. Must be a positive integer.

### SideLobeLevel

Maximum sidelobe level relative to the main lobe peak. The maximum sidelobe level is a nonnegative number which gives side lobes *SideLobeLevel* dB down from the main lobe peak.

## Methods

`generate`

Generates Taylor window

`info`

Display information about Taylor window object

`winwrite`

Save Taylor window object values in ASCII file

## Copy Semantics

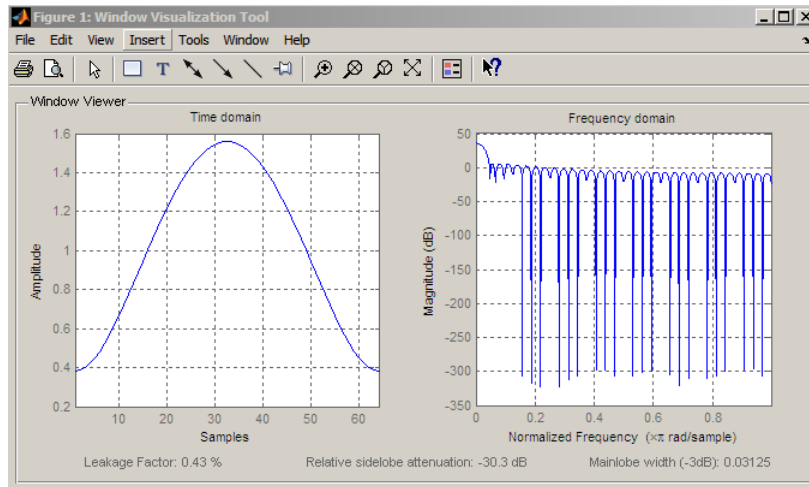
Handle. To learn how copy semantics affect your use of the class, see [Copying Objects in the MATLAB Programming Fundamentals documentation](#).



## Examples

Default length  $N = 64$  Taylor window:

```
H = sigwin.taylorwin;
wvtool(H)
```



Generate length  $N = 128$  Taylor window, return values, and write ASCII file with window values:

```
H = sigwin.taylorwin(128);
% Return window with generate
win = generate(H);
% Write ASCII file in current directory
% with window values
winwrite(H, 'taylorwin_128')
```

## References

Carrara, W. G., R. M. Majewski, and R. S. Goodman. *Spotlight Synthetic Aperture Radar: Signal Processing Algorithms*. Artech House Publishers, Boston, 1995, Appendix D.2.

## See Also

taylorwin | window | wvtool

## **How To**

- Class Attributes
- Property Attributes

## generate

**Class:** sigwin.taylorwin

**Package:** sigwin

Generates Taylor window

## Syntax

```
win = generate(H)
```

## Description

`win = generate(H)` returns the values of the Taylor window object `H` as a double-precision column vector.

## Examples

Extract values from Taylor window object:

```
H=sigwin.taylorwin(128);  
% Extract window values as column vector  
win=generate(H);
```

## info

**Class:** sigwin.taylorwin

**Package:** sigwin

Display information about Taylor window object

## Syntax

```
info(H)  
info_win = info(H)
```

## Description

`info(H)` displays length and sidelobe information for the Taylor window object `H`.

`info_win = info(H)` returns length and sidelobe information for the Taylor window object `H` in the character array `info_win`.

## Examples

Return information about a Taylor window object:

```
H=sigwin.taylorwin(256);  
info_win=info(H);
```

# winwrite

**Class:** sigwin.taylorwin

**Package:** sigwin

Save Taylor window object values in ASCII file

## Syntax

```
winwrite(H)
winwrite(H, 'filename')
```

## Description

`winwrite(H)` opens a dialog to export the values of the Taylor window object `H` to an ASCII file. The file extension `.wf` is automatically appended.

`winwrite(H, 'filename')` saves the values of the Taylor window object `H` in the current folder as a column vector in the ASCII file `'filename'`. The file extension `.wf` is automatically appended to `filename`.

## Examples

Write Taylor window values to ASCII file:

```
H=sigwin.taylorwin;
% Open dialog box for ASCII file
winwrite(H);
```

## sigwin.triang class

**Package:** sigwin

Construct triangular window object

### Description

---

**Note:** The use of `sigwin.triang` is not recommended. Use `triang` instead.

---

`sigwin.triang` is a triangular window object.

`sigwin.triang` creates a handle to a triangular window object for use in spectral analysis and FIR filtering by the `window` method. Object methods enable workspace import and ASCII file export of the window values.

For  $L$  odd, the triangular window is defined as:

$$w(n) = \begin{cases} \frac{2n}{L+1} & 1 \leq n \leq (L+1)/2 \\ 2 - \frac{2n}{L+1} & (L+1)/2 + 1 \leq n \leq L \end{cases}$$

For  $L$  even, the triangular window is defined as:

$$w(n) = \begin{cases} \frac{(2n-1)}{L} & 1 \leq n \leq L/2 \\ 2 - \frac{(2n-1)}{L} & L/2 + 1 \leq n \leq L \end{cases}$$

### Construction

`H = sigwin.triang` returns a triangular window object `H` of length 64.

`H = sigwin.triang(Length)` returns a triangular window object `H` of length *Length*. Entering a positive non-integer value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

## Properties

### Length

Triangular window length. The window length requires a positive integer. Entering a positive non-integer value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

## Methods

|                       |   |
|-----------------------|---|
| <code>generate</code> | Generates triangular window                 |
| <code>info</code>     | Display information about triangular window |
| <code>winwrite</code> | Save triangular window in ASCII file        |

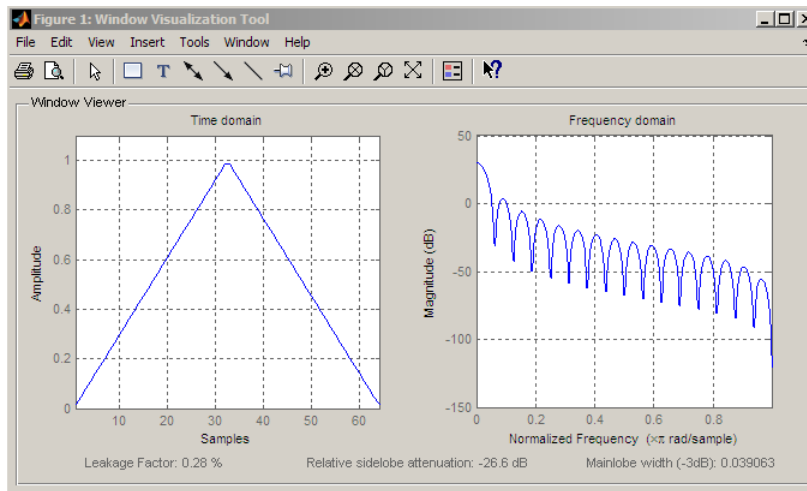
## Copy Semantics

Handle. To learn how copy semantics affect your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

## Examples

Default length `L = 64` triangular window:

```
H = sigwin.triang;  
wvtool(H)
```



Generate length  $L = 128$  triangular window, return values, and write ASCII file:

```
H = sigwin.triang(128);
% Return window with generate
win = generate(H);
% Write ascii file in current directory
% with window values
winwrite(H, 'triang_128')
```

## References

Oppenheim, Alan V., and Ronald W. Schaffer. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1989.

## See Also

triang | window | wvtool

## How To

- Class Attributes
- Property Attributes



## generate

**Class:** sigwin.triang

**Package:** sigwin

Generates triangular window

## Syntax

```
win = generate(H)
```

## Description

`win = generate(H)` returns the values of the triangular window object `H` as a double-precision column vector.

## Examples

Extract values from triangular window object:

```
H=sigwin.triang(128);  
% Extract window values as column vector  
win=generate(H);
```

## info

**Class:** sigwin.triang

**Package:** sigwin

Display information about triangular window

## Syntax

```
info(H)  
info_array = info(H)
```

## Description

`info(H)` displays length information for the triangular window object `H`.

`info_array = info(H)` returns length information for the triangular window object `H` in the character array `info_array`.

## Examples

Return information about a triangular window object:

```
H=sigwin.triangular(256);  
info_win=info(H);
```

# winwrite

**Class:** sigwin.triang

**Package:** sigwin

Save triangular window in ASCII file

## Syntax

```
winwrite(H)
winwrite(H, 'filename')
```

## Description

`winwrite(H)` opens a dialog to export the values of the triangular window object `H` to an ASCII file. The file extension `.wf` is automatically appended.

`winwrite(H, 'filename')` saves the values of the triangular window object `H` as a column vector in the ASCII file `'filename'` in the current folder. The file extension `.wf` is automatically appended to `filename`.

## Examples

Write triangular window values to ASCII file:

```
H=sigwin.triang;
% Open dialog box for ASCII file
winwrite(H);
```

## sigwin.tukeywin class

**Package:** sigwin

Construct Tukey window object

### Description

---

**Note:** The use of `sigwin.tukeywin` is not recommended. Use `tukeywin` instead.

---

`sigwin.tukeywin` creates a handle to a Tukey window object for use in spectral analysis and FIR filtering by the window method. Object methods enable workspace import and ASCII file export of the window values.

The following equation defines the  $N$ -point Tukey window:

$$w(x) = \begin{cases} \frac{1}{2} \{1 + \cos(\frac{2\pi}{\alpha}[x - \alpha/2])\} & 0 \leq x < \frac{\alpha}{2} \\ 1 & \frac{\alpha}{2} \leq x < 1 - \frac{\alpha}{2} \\ \frac{1}{2} \{1 + \cos(\frac{2\pi}{\alpha}[x - 1 + \alpha/2])\} & 1 - \frac{\alpha}{2} \leq x \leq 1 \end{cases}$$

where  $x$  is a  $N$ -point linearly spaced vector generated using `linspace`. The parameter  $\alpha$  is the ratio of cosine-tapered section length to the entire window length with  $0 \leq \alpha \leq 1$ . For example, setting  $\alpha=0.5$  produces a Tukey window where 1/2 of the entire window length consists of segments of a phase-shifted cosine with period  $2\alpha=1$ . If you specify  $\alpha \leq 0$ , an  $N$ -point rectangular window is returned. If you specify  $\alpha \geq 1$ , a von Hann window (`sigwin.hann`) is returned.

### Construction

`H = sigwin.tukeywin` returns a Tukey or cosine-tapered window object `H` of length 64 with *Alpha* parameter equal to 0.5.

`H = sigwin.tukeywin(Length)` returns a Tukey window object `H` of length *Length* with *Alpha* parameter equal to 0.5. *Length* requires a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer.

`H = sigwin.tukeywin(Length,Alpha)` returns a Tukey window object with the ratio of the tapered section length to the entire window length *Alpha*. *Alpha* defaults to 0.5. As *Alpha* approaches zero, the Tukey window approaches a rectangular window. As *Alpha* approaches one, the Tukey window approaches a Hann window.

## Properties

### Length

Tukey window length. The window length must be a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

### Alpha

The ratio of tapered window section to constant section. As a ratio, *Alpha* satisfies the inequality  $0 \leq \alpha \leq 1$ . As *Alpha* approaches zero, the Tukey window approaches a rectangular window. As *Alpha* approaches one, the Tukey window approaches a Hann window. Specifying *Alpha* less than zero or greater than one replaces *Alpha* with 0 and 1 respectively.

## Methods

generate

Generates Tukey window

info

Display information about Tukey window object

winwrite

Save Tukey window in ASCII file

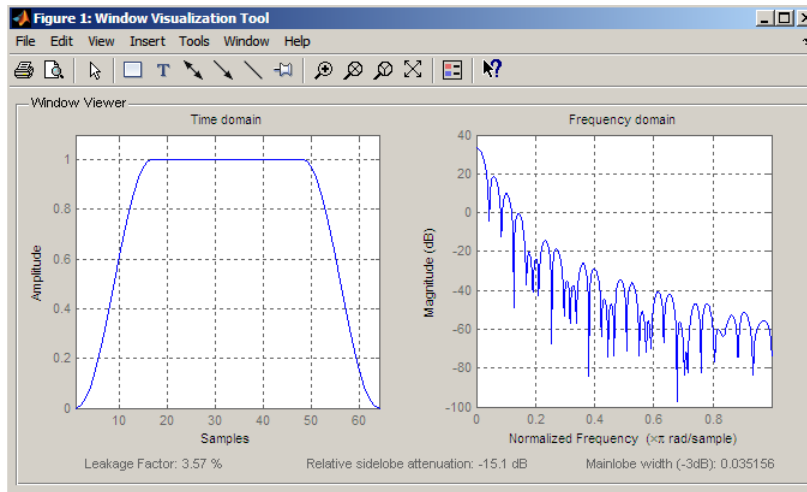
## Copy Semantics

Handle. To learn how copy semantics affect your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

## Examples

Default length  $N = 64$  Tukey window:

```
H=sigwin.tukeywin;
wvtool(H);
```



Generate length  $N = 128$  Tukey window, return values, and write ASCII file:

```
H=sigwin.tukeywin(128,1/4);
% Return window with generate
win=generate(H);
% Write ascii file in current directory
% with window values
winwrite(H,'tukeywin_128')
```

## References

- [1] Bloomfield, P. *Fourier Analysis of Time Series: An Introduction*. New York: Wiley-Interscience, 2000, p.69.

## See Also

tukeywin | window | wvtool

## How To

- Class Attributes
- Property Attributes

## generate

**Class:** sigwin.tukeywin

**Package:** sigwin

Generates Tukey window

## Syntax

```
win = generate(H)
```

## Description

`win = generate(H)` returns the values of the Tukey window object `H` as a double-precision column vector.

## Examples

Extract values from Tukey window object:

```
H=sigwin.tukeywin(128);  
% Extract window values as column vector  
win=generate(H);
```



# info

**Class:** sigwin.tukeywin

**Package:** sigwin

Display information about Tukey window object

## Syntax

```
info(H)  
info_win = info(H)
```

## Description

`info(H)` displays length and tapered-to-constant section ratio information for the Tukey window object `H`.

`info_win = info(H)` returns length and tapered-to-constant section ratio information for the Tukey window object `H` in the character array `info_win`.

## Examples

Return information about a Tukey window object:

```
H=sigwin.tukey(256);  
info_win=info(H);
```

## winwrite

**Class:** sigwin.tukeywin

**Package:** sigwin

Save Tukey window in ASCII file

## Syntax

```
winwrite(H)
winwrite(H, 'filename')
```

## Description

`winwrite(H)` opens a dialog to export the values of the Tukey window object to an ASCII file. The file extension `.wf` is automatically appended.

`winwrite(H, 'filename')` saves the values of the Tukey window object `H` in the current folder as a column vector in the ASCII file `'filename'`. The file extension `.wf` is automatically appended to `filename`.

## Examples

Write Tukey window values to ASCII file:

```
H=sigwin.tukeywin;
% Open dialog box for ASCII file
winwrite(H);
```

# sinad

Signal to noise and distortion ratio

## Syntax

```
r = sinad(x)
r = sinad(x,fs)
r = sinad(pxx,f,'psd')
r = sinad(sxx,f,rbw,'power')
[r,totdistpow] = sinad( ___ )
sinad( ___ )
```

## Description

`r = sinad(x)` returns the signal to noise and distortion ratio (SINAD) in dBc of the real-valued sinusoidal signal `x`. The SINAD is determined using a modified periodogram of the same length as the input signal. The modified periodogram uses a Kaiser window with  $\beta = 38$ .

`r = sinad(x,fs)` specifies the sampling frequency `fs` of the input signal `x`. If you do not specify `fs`, the sampling frequency defaults to 1.

`r = sinad(pxx,f,'psd')` specifies the input `pxx` as a one-sided power spectral density (PSD) estimate. `f` is a vector of frequencies corresponding to the PSD estimates in `pxx`.

`r = sinad(sxx,f,rbw,'power')` specifies the input as a one-sided power spectrum. `rbw` is the resolution bandwidth over which each power estimate is integrated.

`[r,totdistpow] = sinad( ___ )` returns the total noise and harmonic distortion power of the signal.

`sinad( ___ )` with no output arguments plots the spectrum of the signal in the current figure window and labels its fundamental component. It uses different colors to draw the fundamental component, the DC value, and the noise. The SINAD appears above the plot.

## Examples

### SINAD for Signal with One Harmonic or One Harmonic Plus Noise

Create two signals. Both signals have a fundamental frequency of  $\pi/4$  rad/sample with amplitude 1 and the first harmonic of frequency  $\pi/2$  rad/sample with amplitude 0.025. One of the signals additionally has additive white Gaussian noise with variance  $0.05^2$ .

Create the two signals. Set the random number generator to the default settings for reproducible results. Determine the SINAD for the signal without additive noise and compare the result to the theoretical SINAD.

```
n = 0:159;
x = cos(pi/4*n)+0.025*sin(pi/2*n);
rng default;
y = cos(pi/4*n)+0.025*sin(pi/2*n)+0.05*randn(size(n));
r = sinad(x)
powfund = 1;
powharm = 0.025^2;
thSINAD = 10*log10(powfund/powharm)
```

```
r =
    32.0412
thSINAD =
    32.0412
```

Determine the SINAD for the sinusoidal signal with additive noise. Show how including the theoretical variance of the additive noise approximates the SINAD.

```
r = sinad(y)
varnoise = 0.05^2;
thSINAD = 10*log10(powfund/(powharm+varnoise))
```

```
r =
    23.6793
thSINAD =
    25.0515
```

### SINAD for Signal with Sampling Rate

Create a signal with a fundamental frequency of 1 kHz and amplitude 1, sampled at 480 kHz. The signal additionally consists of the first harmonic with amplitude 0.02 and additive white Gaussian noise with variance  $0.01^2$ .

Determine the SINAD and compare the result with the theoretical SINAD.

```

fs = 48e4;
t = 0:1/fs:1-1/fs;
rng default;
x = cos(2*pi*1000*t)+0.02*sin(2*pi*2000*t)+0.01*randn(size(t));
r = sinad(x,fs)
powfund = 1;
powharm = 0.02^2;
varnoise = 0.01^2;
thSINAD = 10*log10(powfund/(powharm+varnoise*(1/fs)))

r =
    32.2059
thSINAD =
    33.9794

```

### SINAD from Periodogram

Create a signal with a fundamental frequency of 1 kHz and amplitude 1, sampled at 480 kHz. The signal additionally consists of the first harmonic with amplitude 0.02 and additive white Gaussian noise with standard deviation 0.01. Set the random number generator to the default settings for reproducible results.

Obtain the periodogram of the signal and use the periodogram as the input to `sinad`.

```

fs = 48e4;
t = 0:1/fs:1-1/fs;
rng default;
x = cos(2*pi*1000*t)+0.02*sin(2*pi*2000*t)+0.01*randn(size(t));
[pxx,f] = periodogram(x,rectwin(length(x)),length(x),fs);
r = sinad(pxx,f,'psd')

r =
    32.2109

```

### SINAD of Amplified Signal

Generate a sinusoid of frequency 2.5 kHz sampled at 50 kHz. Reset the random number generator. Add Gaussian white noise with standard deviation 0.00005 to the signal. Pass the result through a weakly nonlinear amplifier. Plot the SINAD.

```

rng default
fs = 5e4; f0 = 2.5e3;
N = 1024; t = (0:N-1)/fs;

```

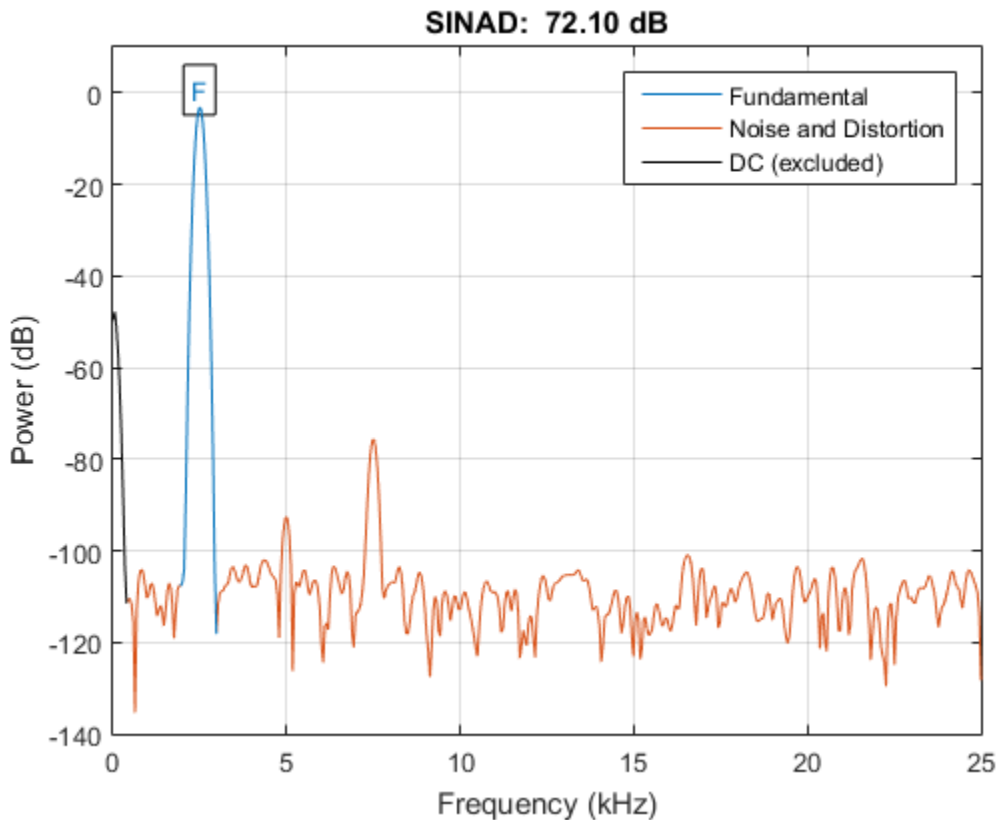
```

ct = cos(2*pi*f0*t);
cd = ct + 0.00005*randn(size(ct));

amp = [1e-5 5e-6 -1e-3 6e-5 1 25e-3];
sgn = polyval(amp,cd);

sinad(sgn,fs);

```



The plot shows the spectrum used to compute the ratio and the region treated as noise. The DC level and the fundamental are excluded from the noise computation. The fundamental is labeled.

- “Analyzing Harmonic Distortion”

## Input Arguments

### **x** — Real-valued sinusoidal input signal

vector

Real-valued sinusoidal input signal specified as a row or column vector.

Example: `cos(pi/4*(0:159))+cos(pi/2*(0:159))`

Data Types: `single` | `double`

### **fs** — Sampling frequency

positive scalar

Sampling frequency, specified as a positive scalar. The sampling frequency is the number of samples per unit time. If the unit of time is seconds, the sampling frequency has units of hertz.

### **pxx** — One-sided PSD estimate

vector

One-sided PSD estimate specified as a real-valued, nonnegative column vector.

Data Types: `single` | `double`

### **f** — Cyclical frequencies

vector

Cyclical frequencies corresponding to the one-sided PSD estimate, `pxx`, specified as a row or column vector. The first element of `f` must be 0.

Data Types: `double` | `single`

### **sxx** — Power spectrum

nonnegative real-valued row or column vector

Power spectrum specified as a real-valued nonnegative row or column vector.

### **rbw** — Resolution bandwidth

positive scalar

Resolution bandwidth specified as a positive scalar. The resolution bandwidth is the product of the frequency resolution of the discrete Fourier transform and the equivalent noise bandwidth of the window.

## Output Arguments

### **r** — Signal to noise and distortion ratio in dBc

real-valued scalar

Signal to noise and distortion ratio in dBc specified as a real-valued scalar.

### **totdistpow** — Total noise and harmonic distortion power of the signal

nonnegative scalar

Total noise and harmonic distortion power of the signal specified as a nonnegative scalar.

## More About

### Distortion Measurement Functions

The functions `thd`, `sfdR`, `sinad`, and `snr` measure the response of a weakly nonlinear system stimulated by a sinusoid.

When given time-domain input, `sinad` performs a periodogram using a Kaiser window with large sidelobe attenuation. To find the fundamental frequency, the algorithm searches the periodogram for the largest nonzero spectral component. It then computes the central moment of all adjacent bins that decrease monotonically away from the maximum. To be detectable, the fundamental should be at least in the second frequency bin. Higher harmonics are at integer multiples of the fundamental frequency. If a harmonic lies within the monotonically decreasing region in the neighborhood of another, its power is considered to belong to the larger harmonic. This larger harmonic may or may not be the fundamental.

The function estimates a noise level using the median power in the regions containing only noise and distortion. The DC component is excluded from the calculation. The noise at each point is the estimated level or the ordinate of the point, whichever is smaller. The noise is then subtracted from the values of the signal and the harmonics.

`sinad` fails if the fundamental is not the highest spectral component in the signal.

Ensure that the frequency components are far enough apart to accommodate for the sidelobe width of the Kaiser window. If this is not feasible, you can use the `'power'` flag and compute a periodogram with a different window.



## **See Also**

sfdr | snr | thd | toi

## sinc

Sinc function

### Syntax

```
y = sinc(x)
```

### Description

`y = sinc(x)` returns an array, `y`, whose elements are the sinc of the elements of the input, `x`. `y` is the same size as `x`.

### Input Arguments

**x** — Input array

scalar value | vector | matrix | N-D array

Input array, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. When `x` is nonscalar, `sinc` is an element-wise operation.

Data Types: `single` | `double`

Complex Number Support: Yes

### Output Arguments

**y** — Sinc of input

scalar value | vector | matrix | N-D array

Sinc of the input array, `x`, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as `x`.

## Examples

### Ideal Bandlimited Interpolation

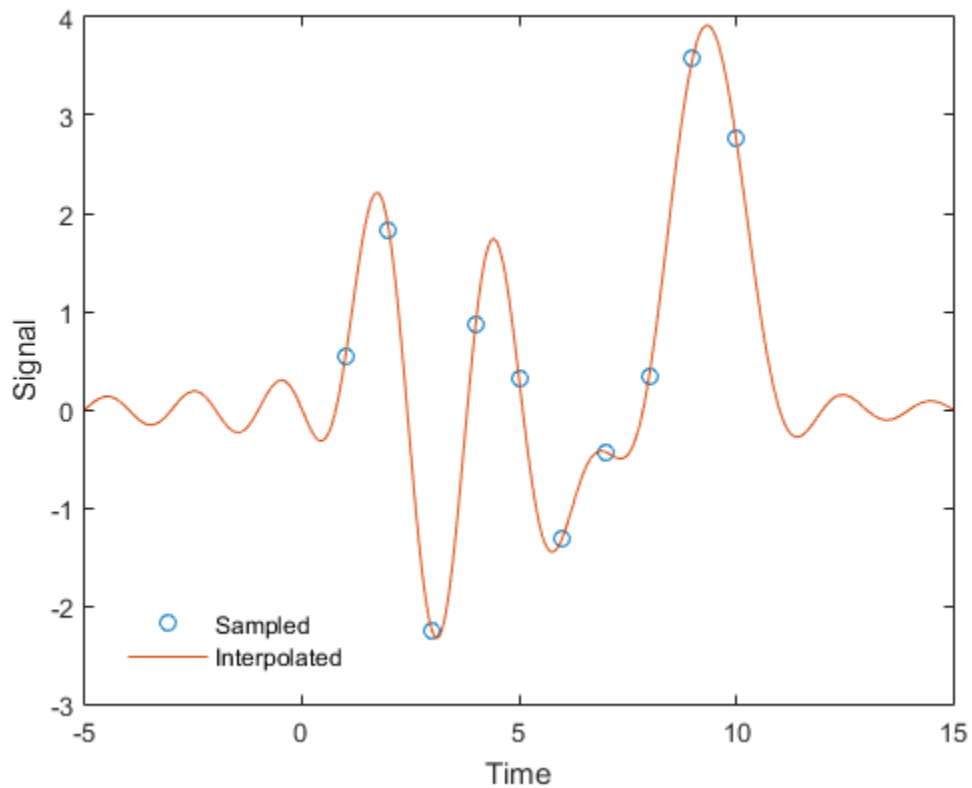
Perform ideal bandlimited interpolation of a random signal sampled at integer spacings.

Assume that the signal to interpolate,  $x$ , is 0 outside of the given time interval and has been sampled at the Nyquist frequency. Reset the random number generator for reproducibility.

```
rng default

t = 1:10;
x = randn(size(t))';
ts = linspace(-5,15,600);
[Ts,T] = ndgrid(ts,t);
y = sinc(Ts - T)*x;

plot(t,x,'o',ts,y)
xlabel Time, ylabel Signal
legend('Sampled','Interpolated','Location','SouthWest')
legend boxoff
```



## More About

### Sinc

The sinc function is defined by

$$\operatorname{sinc} t = \begin{cases} 1 & t = 0, \\ \frac{\sin \pi t}{\pi t} & t \neq 0. \end{cases}$$

This analytic expression corresponds to the continuous inverse Fourier transform of a rectangular pulse of width  $2\pi$  and height 1:

$$\text{sinc } t = \frac{1}{2\pi} \int_{-\pi}^{\pi} e^{j\omega t} d\omega.$$

The space of functions bandlimited in the frequency range  $\omega = (-\pi, \pi]$  is spanned by the countably infinite set of sinc functions shifted by integers. Thus, you can reconstruct any such bandlimited function  $g(t)$  from its samples at integer spacings:

$$g(t) = \sum_{n=-\infty}^{\infty} g(n) \text{sinc}(t - n).$$

### See Also

chirp | cos | diric | gausspuls | pulstran | rectpuls | sawtooth | sin | square | tripuls

# single

Cast coefficients of digital filter to single precision

## Syntax

```
f2 = single(f1)
```

## Description

`f2 = single(f1)` casts coefficients in a digital filter, `f1`, to single precision and returns a new digital filter, `f2`, that contains these coefficients. This is the only way that you can create single-precision `digitalFilter` objects.

## Examples

### Lowpass FIR Filter in Double and Single Precision

Use `designfilt` to design a 5th-order FIR lowpass filter. Specify a normalized passband frequency of  $0.2\pi$  rad/sample and a normalized stopband frequency of  $0.55\pi$  rad/sample. Cast the filter coefficients to single precision.

```
format long
d = designfilt('lowpassfir','FilterOrder',5, ...
              'PassbandFrequency',0.2,'StopbandFrequency', 0.55);
e = single(d);
classd = class(d.Coefficients)
classe = class(e.Coefficients)
```

```
classd =
```

```
double
```

```
classe =
```

```
single
```

---

## Input Arguments

### **f1 — Digital filter**

`digitalFilter` object

Digital filter, specified as a `digitalFilter` object. Use `designfilt` to generate `f1` based on frequency-response specifications.

Example: `d = designfilt('lowpassiir','FilterOrder',3,'HalfPowerFrequency',0.5)` specifies a third-order Butterworth filter with normalized 3-dB frequency  $0.5\pi$  rad/sample.

## Output Arguments

### **f2 — Single-precision digital filter**

`digitalFilter` object

Single-precision digital filter, returned as a `digitalFilter` object.

## See Also

`designfilt` | `digitalFilter` | `double` | `isdouble` | `issingle`

## slewrates

Slew rate of bilevel waveform

### Syntax

```
S = slewrates(X)
S = slewrates(X,Fs)
S = slewrates(X,T)
[S,LT,UT] = slewrates(...)
[S,LT,UT,LL,UL] = slewrates(...)
S = slewrates(...,Name,Value)
slewrates(...)
```

### Description

`S = slewrates(X)` returns the slew rate for all transitions found in the bilevel waveform, `X`. The slew rate is the slope of the line connecting the 10% and 90% reference levels. The sample instants of `X` are the indices of the vector. To determine the transitions, `slewrates` estimates the state levels of the input waveform by a histogram method. `slewrates` identifies all regions that cross the upper-state boundary of the low state and the lower-state boundary of the high state. The low-state and high-state boundaries are expressed as the state level plus or minus a multiple of the difference between the state levels. See “State-Level Tolerances” on page 1-1487.

`S = slewrates(X,Fs)` specifies the sample rate, `Fs`, in hertz. The first time instant in `X` corresponds to `t=0`.

`S = slewrates(X,T)` specifies the sample instants in the vector, `T`. The length of `T` must equal the length of `X`.

`[S,LT,UT] = slewrates(...)` returns the time instants when the waveform crosses the lower-percent reference level, `LT`, and upper-percent reference level, `UT`. If you do not specify lower- and upper-percent reference levels, the levels default to 10% and 90%.

`[S,LT,UT,LL,UL] = slewrates(...)` returns the waveform values that correspond to the lower-reference levels, `LL`, and upper-reference levels, `UL`.



`S = slewrates(...,Name,Value)` returns the slew rate for all transitions with additional options specified by one or more Name,Value pair arguments.

`slewrates(...)` plots the bilevel waveform and darkens the regions of each transition where the slew rate is computed. The plot marks the lower- and upper-reference level crossings and associated reference levels. The plot indicates the state levels and associated lower and upper tolerances.

## Input Arguments

### **X**

Bilevel waveform as a real-valued column or row vector. If the input waveform does not have at least one transition, `slewrates` returns an empty matrix.

### **Fs**

Sampling rate in hertz.

### **T**

Vector of sample instants. The length of T must equal the length of the bilevel waveform, X.

## Name-Value Pair Arguments

### **'PercentReferenceLevels'**

Percent reference levels. See “Percent Reference Levels” on page 1-1486 for a definition.

**Default:** [10,90]

### **'StateLevels'**

Low- and high-state levels. StateLevels is a 1-by-2 real-valued vector. The first element is the low-state level. The second element is the high-state level. If you do not specify low- and high-state levels, `slewrates` estimates the state levels from the input waveform using the histogram method.

### **'Tolerance'**

Tolerance levels (lower and upper state boundaries) expressed as a percentage. See “State-Level Tolerances” on page 1-1487.

**Default:** 2

## Output Arguments

### S

Slew rates as real-valued scalars. A positive slew rate indicates that the upper-percent reference level occurs later than the lower-percent reference level. A negative slew rate indicates that the upper-percent reference level occurs before the lower-percent reference level.

### LT

Time instants when signal crosses the lower percent reference level. If you do not specify the lower percent reference levels with the `'PercentReferenceLevels'` name-value pair, the lower percent reference level is 10%.

### UT

Time instants when signal crosses the upper-percent reference level. If you do not specify the upper-percent reference levels with the `'PercentReferenceLevels'` name-value pair, the upper-percent reference level is 90%.

### LL

Waveform values at the lower-reference level.

### UL

Waveform values at the upper-reference level.

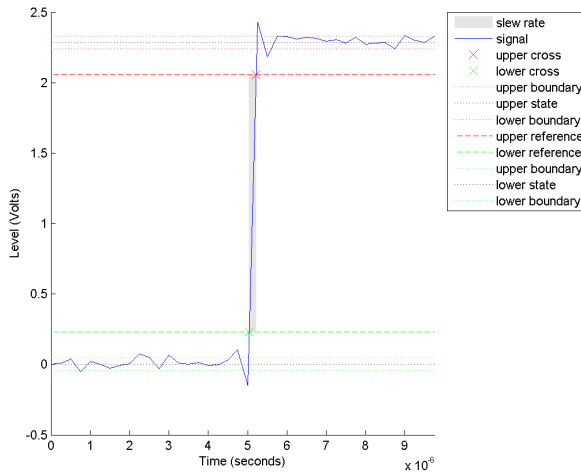
## Examples

### Slew Rate For One-Transition Waveform

Use `slewrates` with no output arguments to plot the slew rate information for a step waveform sampled at 4 MHz.

Load the `transitionex.mat` file and compute the slew rate.

```
load('transitionex.mat', 'x', 't');  
slewrates(x, t)
```



### Slew Rates for Three-Transition Waveform

Create a three-transition (two positive and one negative) bilevel waveform. Obtain the slew rates for the three transitions.

```
load('transitionex.mat', 'x');
y = [x ; flip1r(x)];
t = 0:1/4e6:(length(y)*(1/4e6))-1/4e6;
S = slewrates(y, t);
```

### Lower and Upper Transition Times

Return the lower- and upper-transition times for a three-transition waveform.

```
load('transitionex.mat', 'x');
y = [x ; flip1r(x)];
t = 0:1/4e6:(length(y)*(1/4e6))-1/4e6;
[S,LT,UT] = slewrates(y, t);
% or [S,LT,UT] = slewrates(y,4e6);
```

### Lower and Upper Reference Levels

Return the waveform values corresponding to the lower- and upper-reference levels for a three-transition waveform. Compute these values for the default 10% and 90% and for 20% and 80%.

```
load('transitionex.mat', 'x');
y = [x ; flip1r(x)];
t = 0:1/4e6:(length(y)*(1/4e6))-1/4e6;
[~,LT_1090,UT_1090,LL_1090,UL_1090] = slewrate(y, t);
[~,LT_2080,UT_2080,LL_2080,UL_2080] = slewrate(y, t,...
    'PercentReferenceLevels',[20 80]);
```

## More About

### Percent Reference Levels

If  $S_1$  is the low state,  $S_2$  is the high state, and  $U$  is the *upper*-percent reference level. The waveform value corresponding to the upper-percent reference level is

$$S_1 + \frac{U}{100}(S_2 - S_1)$$

If  $L$  is the *lower*-percent reference level, the waveform value corresponding to the lower percent reference level is

$$S_1 + \frac{L}{100}(S_2 - S_1)$$

### Slew Rate

The slew rate is the slope of a line connecting the upper- and lower-percent reference levels. Let  $t_L$  denote the time instant when the waveform crosses the lower reference level and  $t_U$  denote the time instant when the waveform crosses the upper percent reference level. Using the definitions for the upper and lower percent reference levels given in “Percent Reference Levels” on page 1-1486, the slew rate is

$$\frac{S_1 + \frac{U}{100}(S_2 - S_1) - \{S_1 + \frac{L}{100}(S_2 - S_1)\}}{t_U - t_L}$$

$$\frac{\frac{U-L}{100}(S_2 - S_1)}{t_U - t_L}$$

When  $t_L$  occurs earlier than  $t_U$ , the slew rate is positive. When  $t_U$  occurs earlier than  $t_L$ , the slew rate is negative.

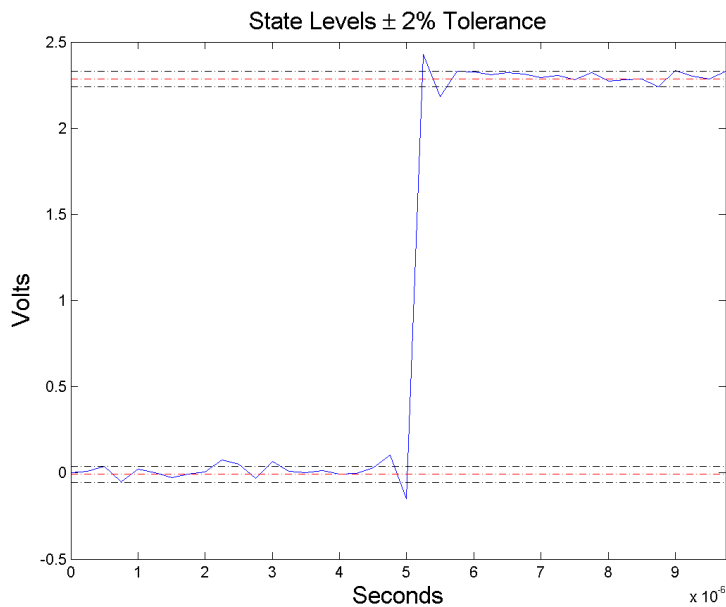
### State-Level Tolerances

Each state level can have associated lower- and upper-state boundaries. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference between the high state and low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the  $\alpha\%$  tolerance region for the low state is defined as

$$S_1 \pm \frac{\alpha}{100}(S_2 - S_1)$$

where  $S_1$  is the low-state level and  $S_2$  is the high-state level. Replace the first term in the equation with  $S_2$  to obtain the  $\alpha\%$  tolerance region for the high state.

The following figure illustrates lower and upper 2% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The red dashed lines indicate the estimated state levels.



## References

- [1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003.

## See Also

falltime | midcross | pulsewidth | risetime | settlingtime | statelevels

## snr

Signal-to-noise ratio

### Syntax

```
r = snr(x,y)
```

```
r = snr(x)
```

```
r = snr(x,fs,n)
```

```
r = snr(pxx,f,'psd')
```

```
r = snr(pxx,f,n,'psd')
```

```
r = snr(sxx,f,rbw,'power')
```

```
r = snr(sxx,f,rbw,n,'power')
```

```
[r,noisepow] = snr(____)
```

```
snr(____)
```

### Description

`r = snr(x,y)` returns the signal-to-noise ratio (SNR) in decibels of a signal, `x`, by computing the ratio of its summed squared magnitude to that of the noise, `y`. `y` must have the same dimensions as `x`. Use this form when the input signal is not necessarily sinusoidal and you have an estimate of the noise.

`r = snr(x)` returns the SNR in decibels relative to the carrier (dBc) of a real-valued sinusoidal input signal, `x`. The SNR is determined using a modified periodogram of the same length as the input. The modified periodogram uses a Kaiser window with  $\beta = 38$ . The result excludes the power of the first six harmonics, including the fundamental.

`r = snr(x,fs,n)` returns the SNR in dBc of a real sinusoidal input signal, `x`, sampled at a rate `fs`. The computation excludes the power contained in the lowest `n` harmonics, including the fundamental. The default value of `fs` is 1. The default value of `n` is 6.

`r = snr(pxx,f,'psd')` specifies the input `pxx` as a one-sided power spectral density (PSD) estimate. The argument `f` is a vector of the frequencies at which the estimates

of `pxx` occur. The computation of noise excludes the power of the first six harmonics, including the fundamental.

`r = snr(pxx,f,n,'psd')` specifies the number of harmonics, `n`, to exclude when computing the SNR. The default value of `n` is 6 and includes the fundamental.

`r = snr(sxx,f,rbw,'power')` specifies the input as a one-sided power spectrum, `sxx`, of a real signal. The input `rbw` is the resolution bandwidth over which each power estimate is integrated.

`r = snr(sxx,f,rbw,n,'power')` specifies the number of harmonics, `n`, to exclude when computing the SNR. The default value of `n` is 6 and includes the fundamental.

`[r,noisepow] = snr(____)` also returns the total noise power of the nonharmonic components of the signal.

`snr(____)` with no output arguments plots the spectrum of the signal in the current figure window and labels its main features. It uses different colors to draw the fundamental component, the DC value and the harmonics, and the noise. The SNR appears above the plot. This functionality works for all syntaxes listed above except `snr(x,y)`.

## Examples

### Signal-to-Noise Ratio for Rectangular Pulse with Gaussian Noise

Compute the signal-to-noise ratio (SNR) of a 20 ms rectangular pulse sampled for 2 s at 10 kHz in the presence of Gaussian noise. Set the random number generator to the default settings for reproducible results.

```
rng default
Tpulse = 20e-3;
Fs = 10e3;
t = -1:1/Fs:1;
x = rectpuls(t,Tpulse);
y = 0.00001*randn(size(x));
s = x + y;
pulseSNR = snr(x,s-x)

pulseSNR =
```



80.0818

### Compare SNR with THD and SINAD

Compute and compare the signal-to-noise ratio (SNR), the total harmonic distortion (THD), and the signal to noise and distortion ratio (SINAD) of a signal.

Create a sinusoidal signal sampled at 48 kHz. The signal has a fundamental of frequency 1 kHz and unit amplitude. It additionally contains a 2 kHz harmonic with half the amplitude and additive noise with variance  $0.1^2$ . Set the random number generator to the default settings for reproducible results.

```
rng default
fs = 48e3;
t = 0:1/fs:1-1/fs;
A = 1.0; powfund = A^2/2;
a = 0.4; powharm = a^2/2;
s = 0.1; varnoise = s^2;
x = A*cos(2*pi*1000*t) + ...
    a*sin(2*pi*2000*t) + s*randn(size(t));
```

Verify that SNR, THD, and SINAD agree with their definitions.

```
SNR = snr(x);
defSNR = 10*log10(powfund/varnoise);
SN = [SNR defSNR]

THD = thd(x);
defTHD = 10*log10(powharm/powfund);
TH = [THD defTHD]

SINAD = sinad(x);
defSINAD = 10*log10(powfund/(powharm+varnoise));
SI = [SINAD defSINAD]

SN =
    17.0178    16.9897
TH =
   -7.9546   -7.9588
SI =
```

```
7.4571    7.4473
```

### Noise Power

Compute the noise power in the sinusoid from the preceding example. Verify that it agrees with the definition. Set the random number generator to the default settings for reproducible results.

```
rng default
fs = 48e3;
t = 0:1/fs:1-1/fs;
A = 1.0; powfund = A^2/2;
a = 0.4; powharm = a^2/2;
s = 0.1; varnoise = s^2;
x = A*cos(2*pi*1000*t) + ...
    a*sin(2*pi*2000*t) + s*randn(size(t));
[SNR npow]=snr(x,fs);
[10*log10(powfund)-npow SNR]

ans =
    17.0281    17.0178
```

### Signal-to-Noise Ratio of a Sinusoid

Compute the SNR of a 2.5 kHz sinusoid sampled at 48 kHz. Add white noise with standard deviation 0.001. Set the random number generator to the default settings for reproducible results.

```
rng default
Fi = 2500; Fs = 48e3; N = 1024;
x = sin(2*pi*Fi/Fs*(1:N)) + 0.001*randn(1,N);
SNR = snr(x,Fs)

SNR =
    57.7103
```

### SNR of a Sinusoid Using the PSD

Obtain the periodogram power spectral density (PSD) estimate of a 2.5 kHz sinusoid sampled at 48 kHz. Add white noise with standard deviation 0.00001. Use this value as input to determine the SNR. Set the random number generator to the default settings for reproducible results.

```
rng default
```

```

Fi = 2500; Fs = 48e3; N = 1024;
x = sin(2*pi*Fi/Fs*(1:N)) + 0.00001*randn(1,N);
w = kaiser(numel(x),38);
[Pxx, F] = periodogram(x,w,numel(x),Fs);
SNR = snr(Pxx,F,'psd')

```

```

SNR =
    97.7446

```

### SNR of a Sinusoid Using the Power Spectrum

Compute the SNR of the sinusoid from the preceding example, using the power spectrum. Set the random number generator to the default settings for reproducible results.

```

rng default
Fi = 2500; Fs = 48e3; N = 1024;
x = sin(2*pi*Fi/Fs*(1:N)) + 0.00001*randn(1,N);
w = kaiser(numel(x),38);
[Sxx, F] = periodogram(x,w,numel(x),Fs,'power');
rbw = enbw(w,Fs);
SNR = snr(Sxx,F,rbw,'power')

```

```

SNR =
    97.7446

```

### SNR of Amplified Signal

Generate a sinusoid of frequency 2.5 kHz sampled at 50 kHz. Reset the random number generator. Add Gaussian white noise with standard deviation 0.00005 to the signal. Pass the result through a weakly nonlinear amplifier. Plot the SNR.

```

rng default

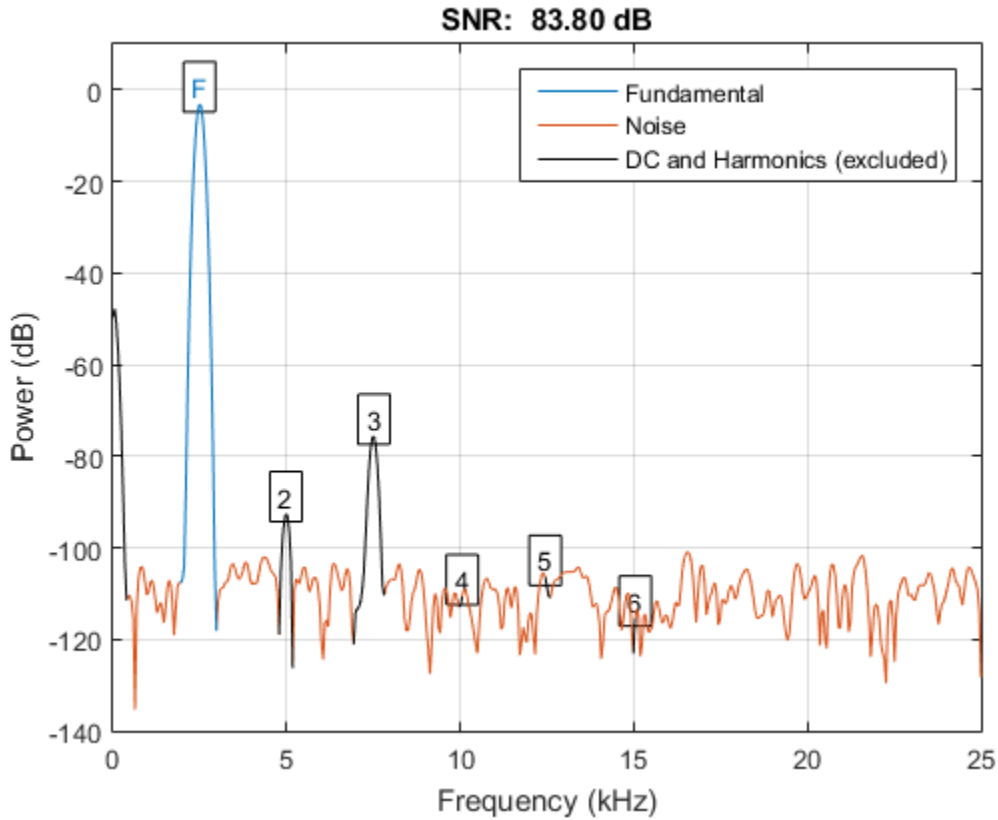
fs = 5e4;
f0 = 2.5e3;
N = 1024;
t = (0:N-1)/fs;

ct = cos(2*pi*f0*t);
cd = ct + 0.00005*randn(size(ct));

amp = [1e-5 5e-6 -1e-3 6e-5 1 25e-3];
sgn = polyval(amp,cd);

```

```
snr(sgn, fs);
```



The DC component and all harmonics, including the fundamental, are excluded from the noise measurement. The fundamental and harmonics are labeled.

- “Analyzing Harmonic Distortion”

## Input Arguments

**x** — Real-valued input signal  
real vector

Real-valued input signal, specified as a row or column vector.

Data Types: `double` | `single`

**y** — **Noise estimate**

real vector

Estimate of the noise in the input signal, specified as a real-valued row or column vector. It must have the same dimensions as `x`.

Data Types: `double` | `single`

**fs** — **Sampling frequency**

1 (default) | positive real scalar

Sampling frequency, specified as a positive scalar. The sampling frequency is the number of samples per unit time. If the unit of time is seconds, the sampling frequency has units of hertz.

Data Types: `double` | `single`

**n** — **Number of harmonics**

6 (default) | positive integer scalar

Number of harmonics to exclude from the SNR computation, specified as a positive integer scalar. The default value of `n` is 6.

**pxx** — **One-sided PSD estimate**

vector

One-sided power spectral density estimate, specified as a real-valued, nonnegative column vector.

Data Types: `double` | `single`

**f** — **Cyclical frequencies**

real-valued row or column vector

Cyclical frequencies of the one-sided PSD estimate, `pxx`, specified as a row or column vector. The first element of `f` must be 0.

Data Types: `double` | `single`

**sxx** — **Power spectrum**

nonnegative real-valued row or column vector

Power spectrum, specified as a real-valued nonnegative row or column vector.

Data Types: `double` | `single`

**rbw** — Resolution bandwidth

positive scalar

Resolution bandwidth, specified as a positive scalar. The resolution bandwidth is the product of the frequency resolution of the discrete Fourier transform and the equivalent noise bandwidth of the window.

Data Types: `double` | `single`

## Output Arguments

**r** — Signal-to-noise ratio

real-valued scalar

Signal-to-noise ratio, expressed in decibels relative to the carrier (dBc), returned as a real-valued scalar. The SNR is returned in decibels (dB) if the input signal is not sinusoidal.

Data Types: `double` | `single`

**noisepow** — Total noise power

real-valued scalar

Total noise power of the nonharmonic components of the input signal, returned as a real-valued scalar.

Data Types: `double` | `single`

## More About

### Distortion Measurement Functions

The functions `thd`, `sfdr`, `sinad`, and `snr` measure the response of a weakly nonlinear system stimulated by a sinusoid.

When given time-domain input, `snr` performs a periodogram using a Kaiser window with large sidelobe attenuation. To find the fundamental frequency, the algorithm searches

the periodogram for the largest nonzero spectral component. It then computes the central moment of all adjacent bins that decrease monotonically away from the maximum. To be detectable, the fundamental should be at least in the second frequency bin. Higher harmonics are at integer multiples of the fundamental frequency. If a harmonic lies within the monotonically decreasing region in the neighborhood of another, its power is considered to belong to the larger harmonic. This larger harmonic may or may not be the fundamental.

The function estimates a noise level using the median power in the regions containing only noise. The DC component is excluded from the calculation. The noise at each point is the estimated level or the ordinate of the point, whichever is smaller. The noise is then subtracted from the values of the signal and the harmonics.

snr fails if the fundamental is not the highest spectral component in the signal.

Ensure that the frequency components are far enough apart to accommodate for the sidelobe width of the Kaiser window. If this is not feasible, you can use the 'power' flag and compute a periodogram with a different window.

## **See Also**

sfdr | sinad | thd | toi

## sos2cell

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

$$m = [b_1 \ a_1; b_2 \ a_2; \dots; b_L \ a_L]$$

where both  $b_i$  and  $a_i$ , with  $i = 1, \dots, L$ , are 1-by-3 row vectors. The resulting `c` is a 1-by- $L$  cell array of cells of the form

$$c = \{ \{b_1 \ a_1\} \ {b_2 \ a_2\} \ \dots \ {b_L \ a_L\} \ }$$

`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

$$c = \{ \{g, 1\} \ {b_1, a_1\} \ {b_2, a_2\} \ \dots \ {b_L, a_L\} \ }$$

### Examples

#### Second-Order-Section Cell Array of Elliptic Filter

Generate a lowpass elliptic filter of order 4 with 0.5 dB of passband ripple and 20 dB of stopband attenuation. The passband edge is 0.6 times the Nyquist frequency. Convert the transfer function to a matrix of second-order sections.



```
[b,a] = ellip(4,0.5,20,0.6);  
m = tf2sos(b,a);
```

Use `sos2cell` to convert the 2-by-6 matrix produced by `tf2sos` into a 1-by-2 cell array, `c`, of cells. Display the second entry in the first cell of `c`. Verify that it contains the denominator coefficients of the first second-order section of `m`.

```
c = sos2cell(m);  
compare = [c{1}{2};m(1,4:6)]
```

```
compare =
```

```
    1.0000    0.1677    0.2575  
    1.0000    0.1677    0.2575
```

## See Also

`tf2sos` | `cell2sos`

## sos2ss

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 digital filter to an equivalent state-space representation.

`[A,B,C,D] = sos2ss(sos)` converts `sos`, a system expressed 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  $L \times 6$  matrix organized as

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

The entries of `sos` must be real for proper conversion to state space. The returned matrix `A` is of size  $2L \times 2L$ , `B` is a  $2L \times 1$  column vector, `C` is a  $1 \times 2L$  row vector, and `D` is a  $1 \times 1$  scalar.

$[A,B,C,D] = \text{sos2ss}(\text{sos},g)$  converts to state space a system SOS in second-order section form with gain  $g$ :

$$H(z) = g \prod_{k=1}^L H_k(z).$$

## Examples

### State-Space Representation of a Second-Order Section System

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,2)
```

A =

```
-10    0    10    1
  1     0     0     0
  0     1     0     0
  0     0     1     0
```

B =

```
1
0
0
0
```

C =

```
42    4   -32   -2
```

D =

- 4

## More About

### Algorithms

`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`

## sos2tf

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.

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

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

### Transfer Function Representation of a Second-Order Section System

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
```

## More About

### Algorithms

`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](#)

## sos2zp

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.

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

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

### Zeros, Poles, and Gain of a System

Compute the zeros, poles, 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 + 0.0000i  
-0.2808 + 0.0000i
```

p =

```
-1.0000  
 1.0000  
-9.8990  
-0.1010
```

k =

```
-2
```

## More About

### Algorithms

sos2zp finds the poles and zeros of each second-order section by repeatedly calling tf2zp.



**See Also**

sos2ss | sos2tf | ss2zp | tf2zp | tf2zpk | zp2sos

## sosfilt

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`.

---

**Note:** If either input to `sosfilt` is single precision, filtering is implemented using single-precision arithmetic. The output, `y`, is single precision.

---

`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.

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

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  $x$ .

The second order sections matrix, `sos`, the input signal,  $x$ , or both can be double or single precision. If at least one input is single precision, filtering is done with single precision arithmetic.

`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`

## spectrogram

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,w,t] = spectrogram( ___ )
[s,f,t] = spectrogram( ___ ,fs)

[s,w,t] = spectrogram(x,window,noverlap,w)
[s,f,t] = spectrogram(x,window,noverlap,f,fs)

[ ___ ,pxx] = spectrogram( ___ )

[ ___ ] = spectrogram( ___ ,freqrange)
[ ___ ] = spectrogram( ___ ,spectrumtype)

spectrogram( ___ )
spectrogram( ___ ,freqloc)
```

### Description

`s = spectrogram(x)` returns the short-time Fourier transform of the input signal, `x`. Each column of `s` contains an estimate of the short-term, time-localized frequency content of `x`.

`s = spectrogram(x,window)` uses `window` to divide the signal into sections and perform windowing.

`s = spectrogram(x,window,noverlap)` uses `noverlap` samples of overlap between adjoining sections.

`s = spectrogram(x,window,noverlap,nfft)` uses nfft sampling points to calculate the discrete Fourier transform.

`[s,w,t] = spectrogram( ___ )` returns a vector of normalized frequencies, `w`, and a vector of time instants, `t`, at which the spectrogram is computed. This syntax can include any combination of input arguments from previous syntaxes.

`[s,f,t] = spectrogram( ___ ,fs)` returns a vector of cyclical frequencies, `f`, expressed in terms of the sample rate, `fs`.

`[s,w,t] = spectrogram(x,window,noverlap,w)` returns the spectrogram at the normalized frequencies specified in `w`.

`[s,f,t] = spectrogram(x,window,noverlap,f,fs)` returns the spectrogram at the cyclical frequencies specified in `f`.

`[ ___ ,pxx] = spectrogram( ___ )` additionally returns a matrix, `pxx`, containing a power spectral density (PSD) estimate of each section.

`[ ___ ] = spectrogram( ___ ,freqrange)` returns the PSD estimate over the frequency range specified by `freqrange`. Valid options for `freqrange` are `'onesided'`, `'twosided'`, and `'centered'`.

`[ ___ ] = spectrogram( ___ ,spectrumtype)` returns the PSD estimate if `spectrumtype` is specified as `'psd'` and returns the power spectrum if `spectrumtype` is specified as `'power'`.

`spectrogram( ___ )` with no output arguments plots the spectrogram in the current figure window.

`spectrogram( ___ ,freqloc)` specifies the axis on which to plot the frequency.

## Examples

### Frequency Along x-Axis

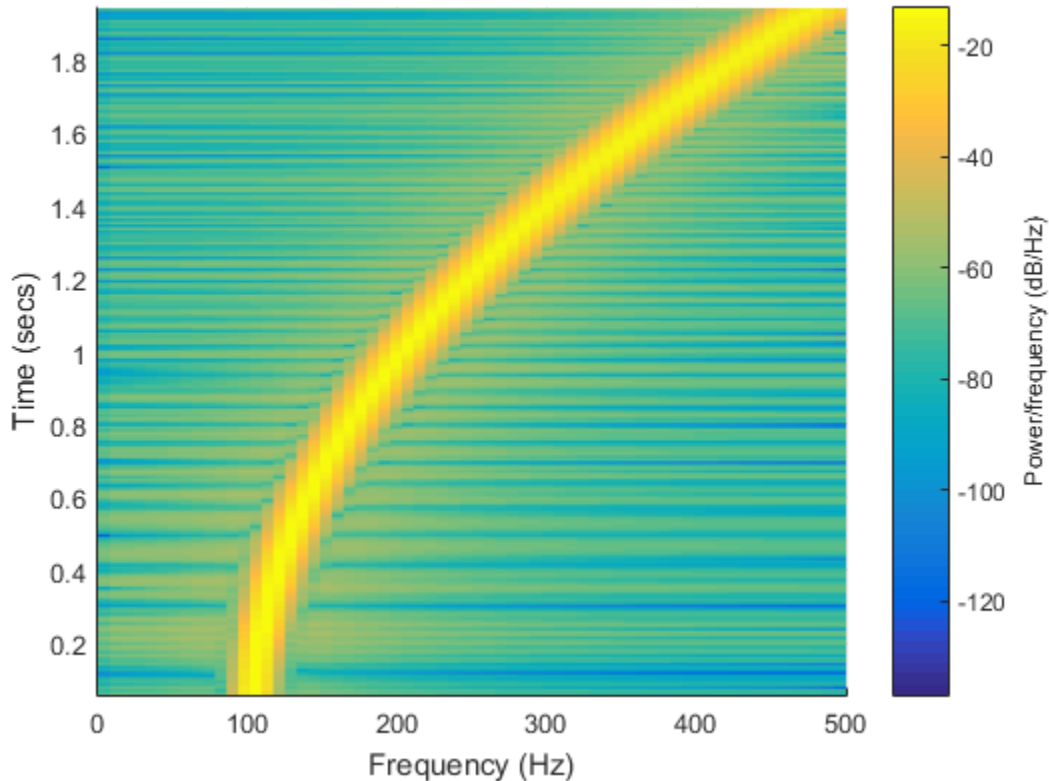
Generate a quadratic chirp, `x`, sampled at 1 kHz for 2 seconds. The frequency of the chirp is 100 Hz initially and crosses 200 Hz at `t = 1 s`.

```
t = 0:0.001:2;
```

```
x = chirp(t,100,1,200,'quadratic');
```

Compute and display the spectrogram of  $x$ . Divide the signal into sections of length 128, windowed with a Hamming window. Specify 120 samples of overlap between adjoining sections. Evaluate the spectrum at  $\lfloor 128/2 + 1 \rfloor = 65$  frequencies and  $\lfloor (\text{length}(x) - 120)/(128 - 120) \rfloor = 235$  time bins.

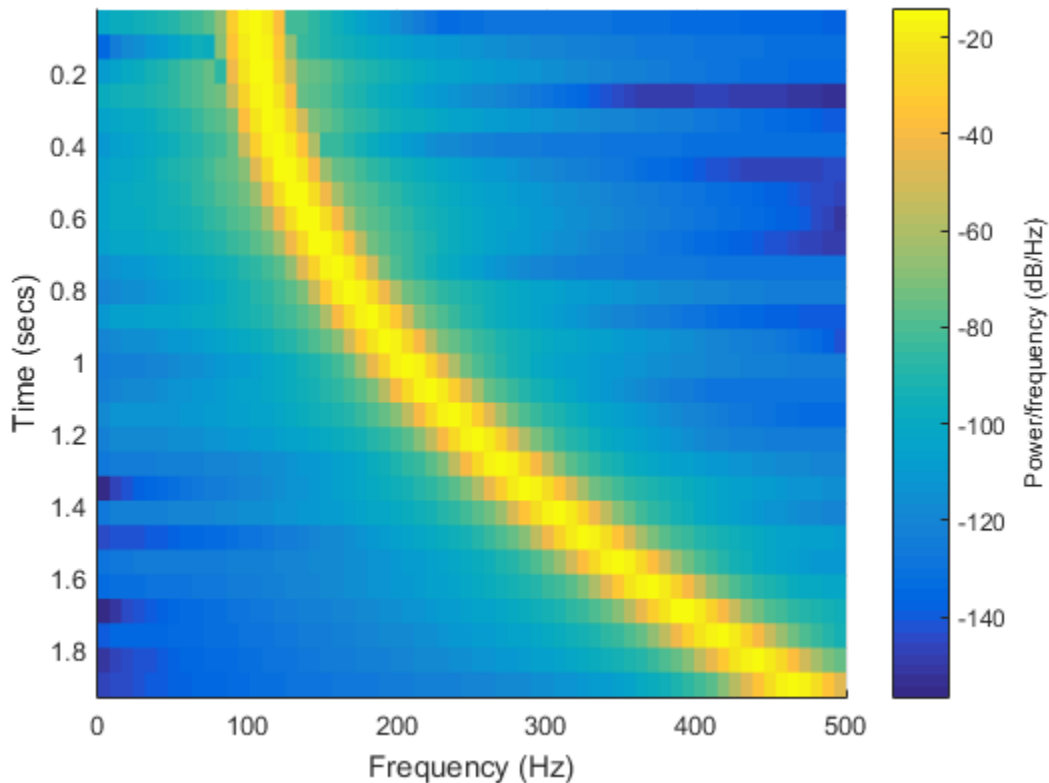
```
spectrogram(x,128,120,128,1e3)
```



Replace the Hamming window with a Blackman window. Decrease the overlap to 60 samples. Plot the time axis so that its values increase from top to bottom.

```
spectrogram(x,blackman(128),60,128,1e3)
ax = gca;
```

```
ax.YDir = 'reverse';
```



### Spectrogram and Instantaneous Frequency

Use the spectrogram to measure and track the instantaneous frequency of a signal.

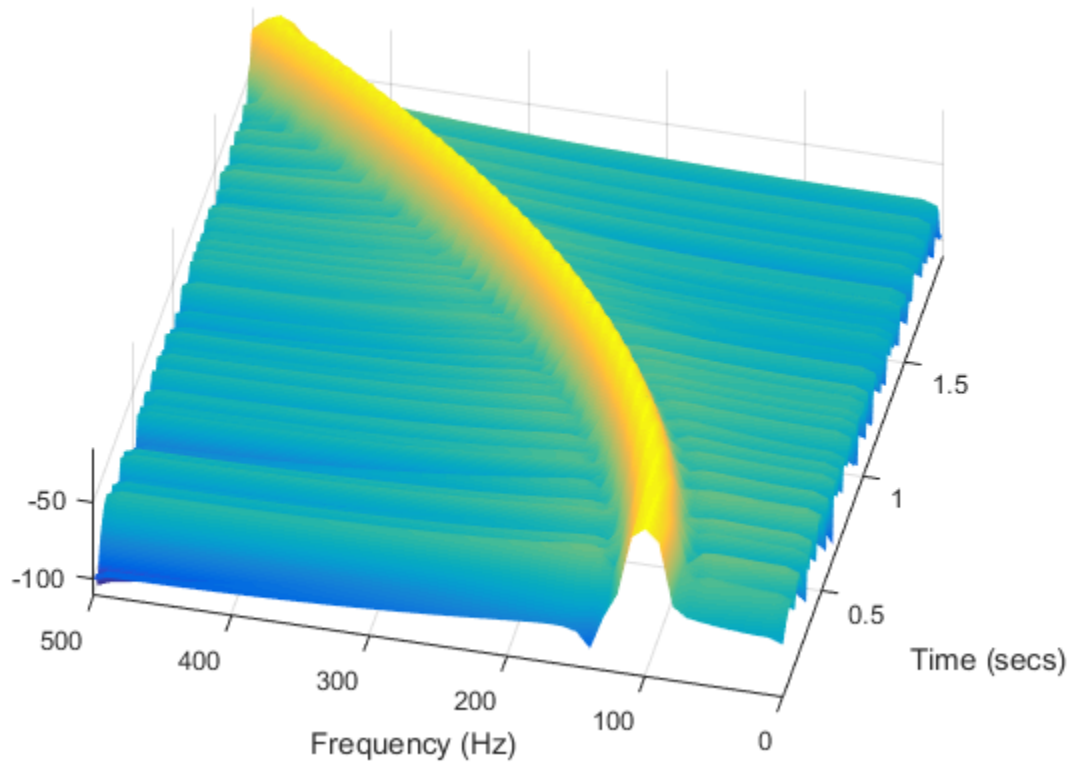
Generate a quadratic chirp sampled at 1 kHz for two seconds. Specify the chirp so its frequency is initially 100 Hz and increases to 200 Hz after one second.

```
Fs = 1000;  
t = 0:1/Fs:2-1/Fs;  
y = chirp(t,100,1,200,'quadratic');
```

Estimate the spectrum of the chirp using the short-time Fourier transform implemented in the spectrogram function. Divide the signal into sections of length 100, windowed

with a Hamming window. Specify 80 samples of overlap between adjoining sections and evaluate the spectrum at  $\lfloor 100/2 + 1 \rfloor = 51$  frequencies. Suppress the default color bar.

```
spectrogram(y,100,80,100,Fs,'yaxis')  
view(-77,72)  
shading interp  
colorbar off
```

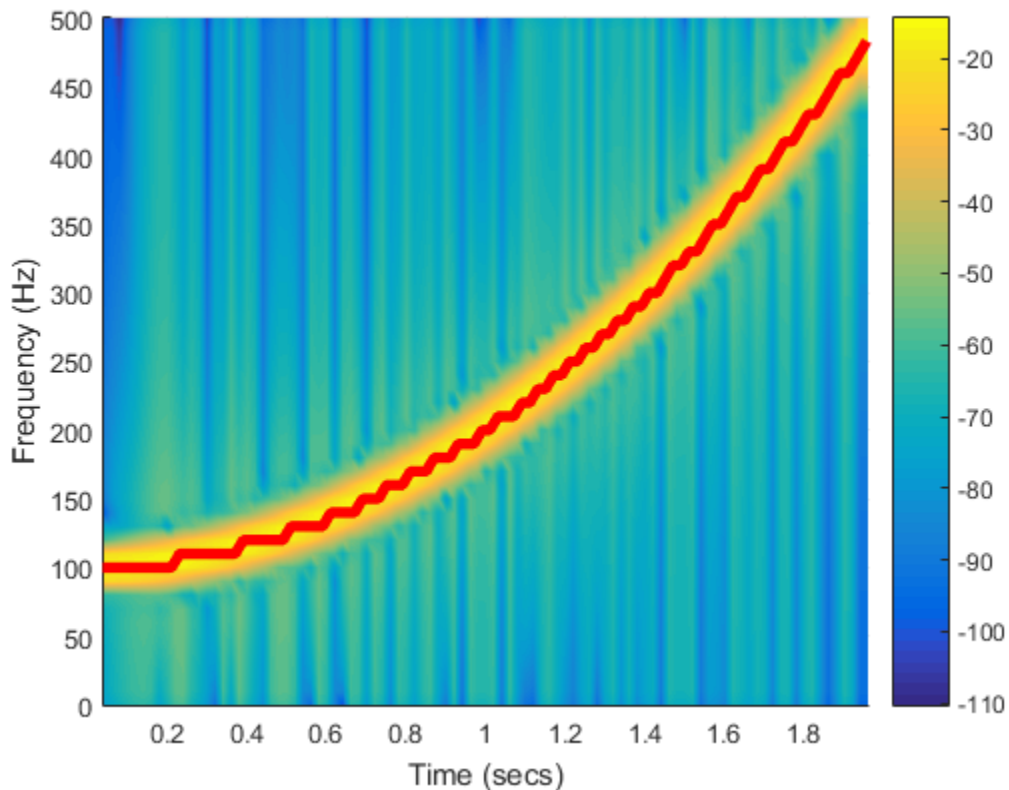


Track the chirp frequency by finding the maximum of the power spectral density at each of the  $\lfloor (2000 - 80)/(100 - 80) \rfloor = 96$  time points. View the spectrogram as a two-dimensional graphic. Restore the color bar.

```
[s,f,t,p] = spectrogram(y,100,80,100,Fs);
```



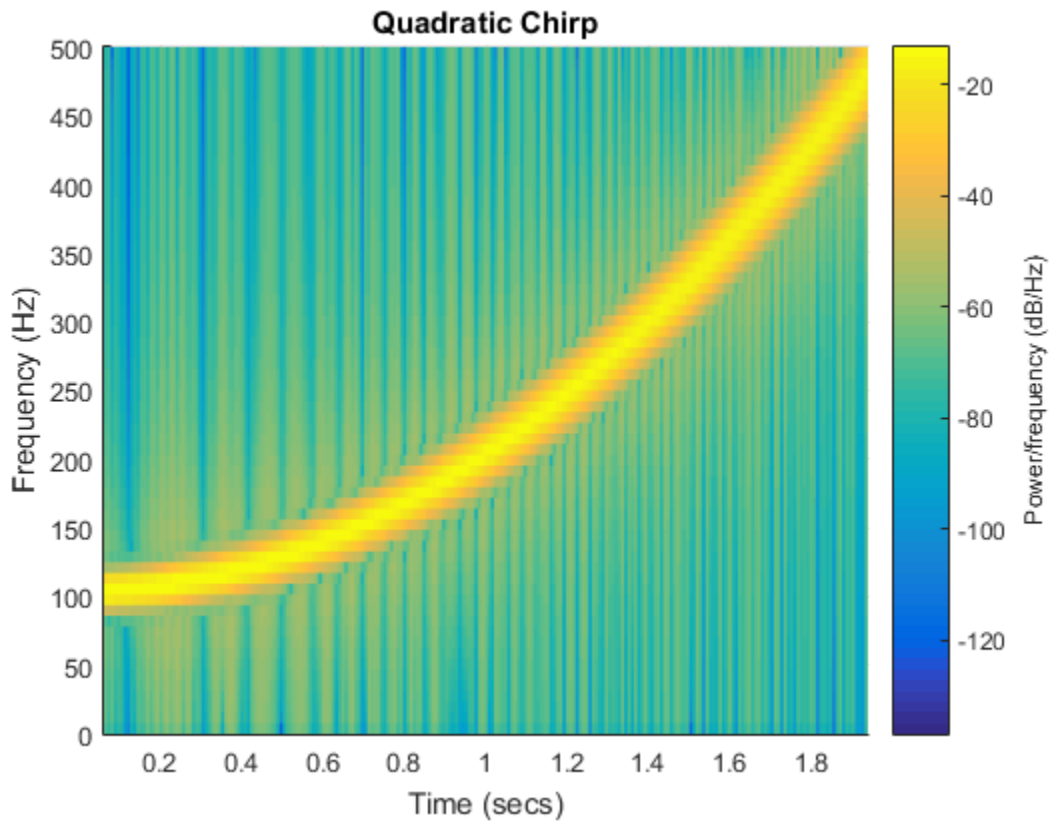
```
[q,nd] = max(10*log10(p));
hold on
plot3(t,f(nd),q,'r','linewidth',4)
hold off
colorbar
view(2)
```



### Power Spectral Densities of Chirps

Compute and display the PSD of each segment of a quadratic chirp that starts at 100 Hz and crosses 200 Hz at  $t = 1$  s. Specify a sample rate of 1 kHz, a segment length of 128 samples, and an overlap of 120 samples. Use 128 DFT points and the default Hamming window.

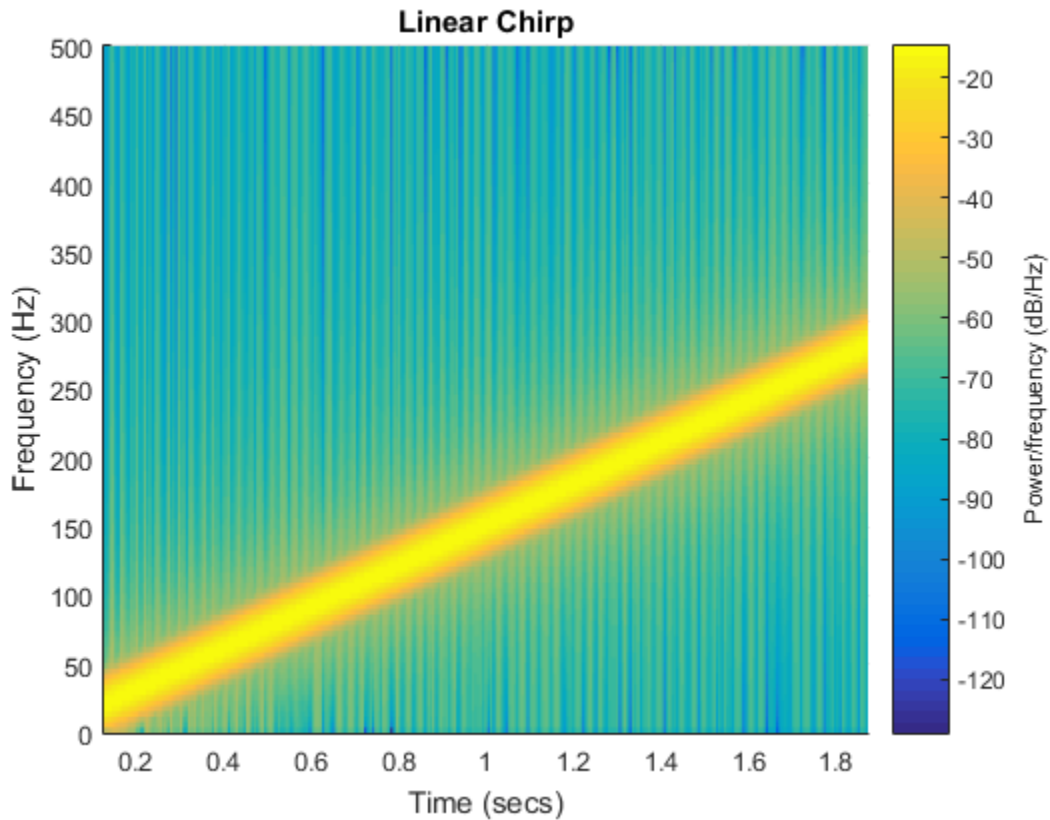
```
t = 0:0.001:2;  
x = chirp(t,100,1,200,'quadratic');  
  
spectrogram(x,128,120,128,1e3,'yaxis')  
title('Quadratic Chirp')
```



Compute and display the PSD of each segment of a linear chirp that starts at DC and crosses 150 Hz at  $t = 1$  s. Specify a sample rate of 1 kHz, a segment length of 256 samples, and an overlap of 250 samples. Use the default Hamming window and 256 DFT points.

```
t = 0:0.001:2;  
x = chirp(t,0,1,150);
```

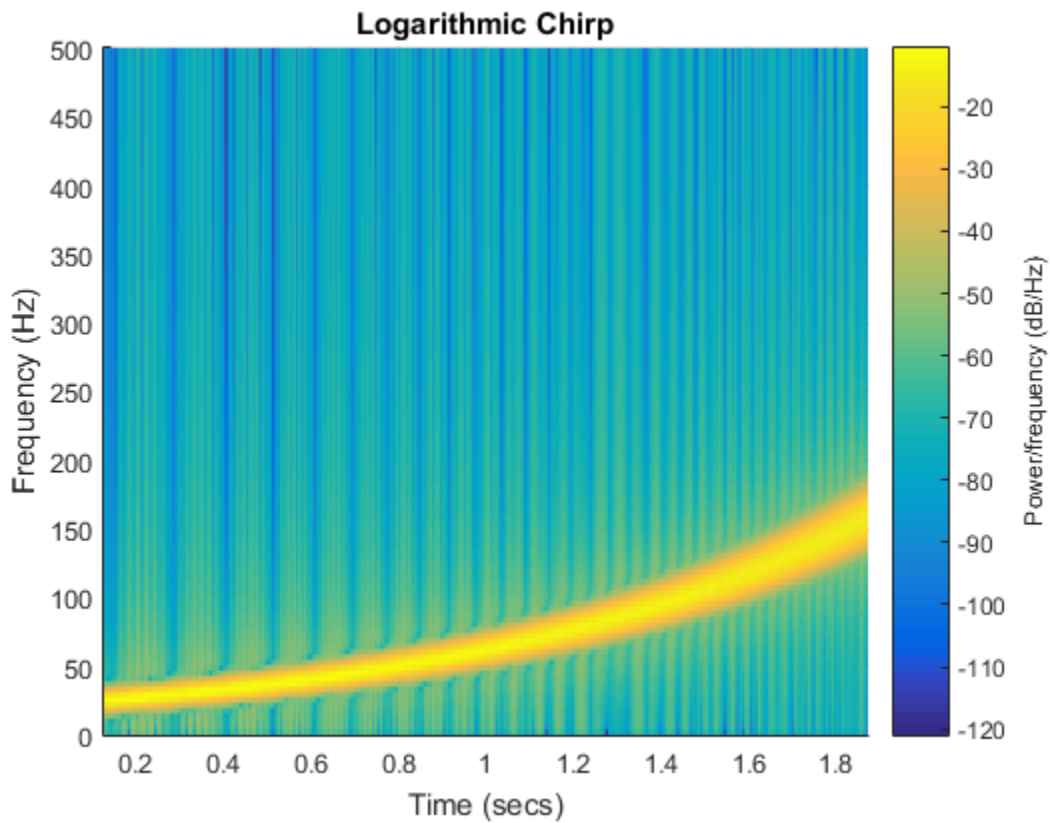
```
spectrogram(x,256,250,256,1e3,'yaxis')
title('Linear Chirp')
```



Compute and display the PSD of each segment of a logarithmic chirp sampled at 1 kHz that starts at 20 Hz and crosses 60 Hz at  $t = 1$  s. Specify a segment length of 256 samples and an overlap of 250 samples. Use the default Hamming window and 256 DFT points.

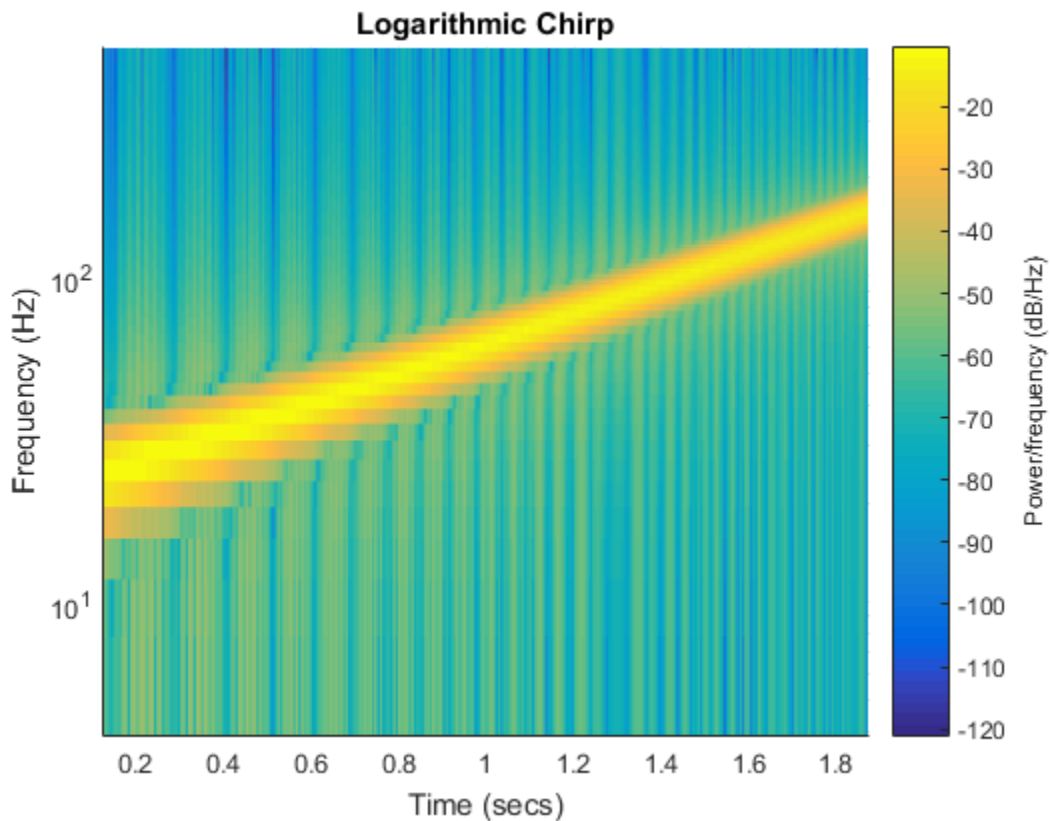
```
t = 0:0.001:2;
x = chirp(t,20,1,60,'logarithmic');

spectrogram(x,256,250,[],1e3,'yaxis')
title('Logarithmic Chirp')
```



Use a logarithmic scale for the frequency axis. The spectrogram becomes a line.

```
ax = gca;  
ax.YScale = 'log';
```

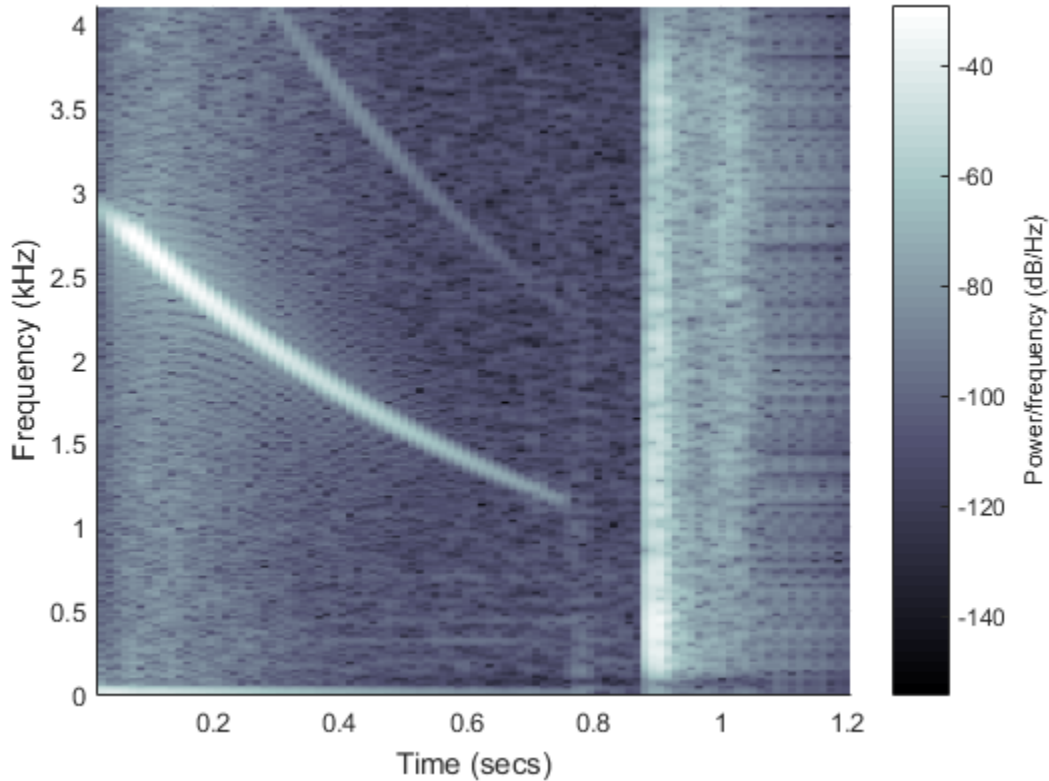


### Track Chirps in Audio Signal

Load an audio signal that contains two decreasing chirps and a wideband splatter sound. Compute the short-time Fourier transform. Divide the waveform into 400-sample segments with 300-sample overlap. Plot the spectrogram.

```
load splat
% To hear, type soundsc(y,Fs)
sg = 400;
ov = 300;
spectrogram(y,sg,ov,[],Fs,'yaxis')
```

colormap `bone`



Use the `spectrogram` function to output the power spectral density (PSD) of the signal.

```
[s,f,t,p] = spectrogram(y,sg,ov,[],Fs);
```

Track the two chirps using the `medfreq` function. To find the stronger, low-frequency chirp, restrict the search to frequencies above 100 Hz and to times before the start of the wideband sound.

```
f1 = f > 100;
```

```
t1 = t < 0.75;
```

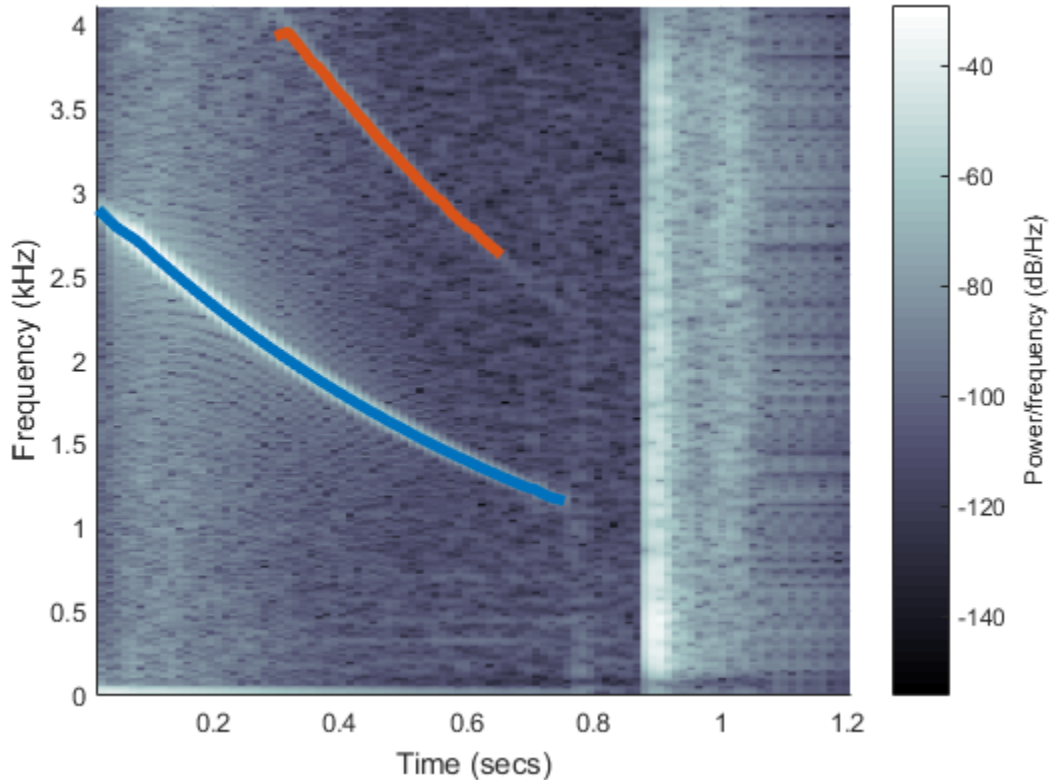
```
m1 = medfreq(p(f1,t1),f(f1));
```

To find the faint high-frequency chirp, restrict the search to frequencies above 2500 Hz and to times between 0.3 seconds and 0.65 seconds.

```
f2 = f > 2500;  
t2 = t > 0.3 & t < 0.65;  
  
m2 = medfreq(p(f2,t2),f(f2));
```

Overlay the result on the spectrogram. Divide the frequency values by 1000 to express them in kHz.

```
hold on  
plot(t(t1),m1/1000,'linewidth',4)  
plot(t(t2),m2/1000,'linewidth',4)  
hold off
```



### 3D Spectrogram Visualization

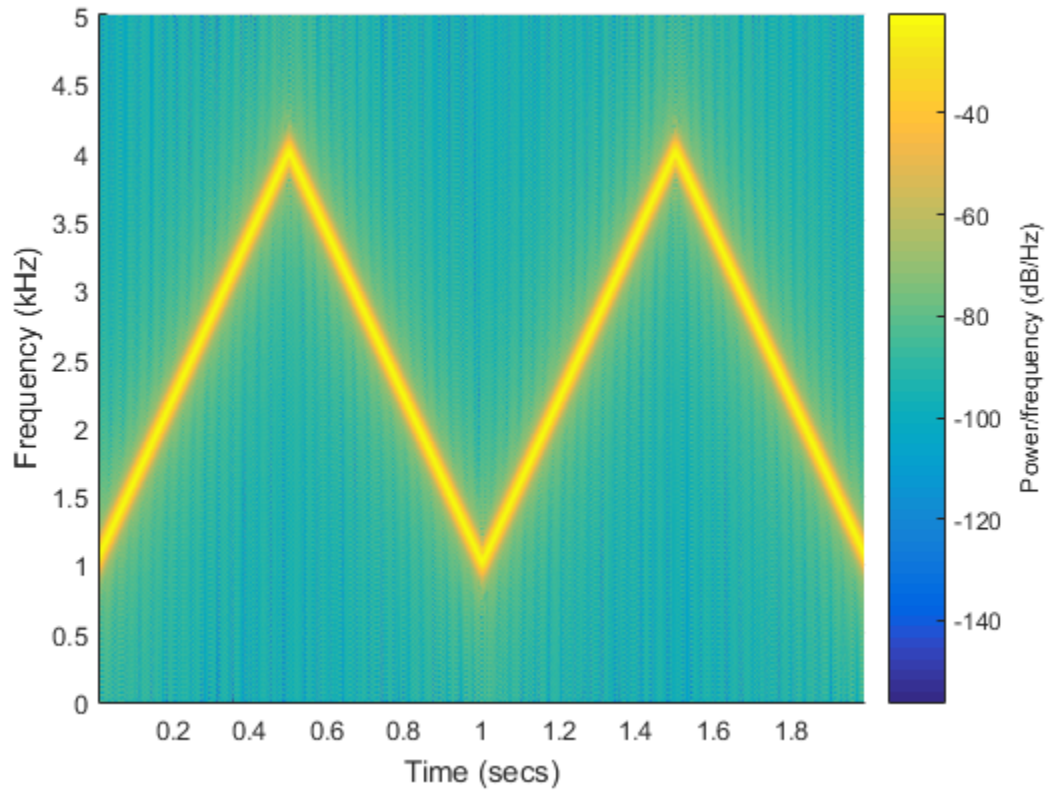
Generate two seconds of a signal sampled at 10 kHz. Specify the instantaneous frequency of the signal as a triangular function of time.

```
fs = 10e3;
t = 0:1/fs:2;
x1 = vco(sawtooth(2*pi*t,0.5),[0.1 0.4]*fs,fs);
```

Compute and plot the spectrogram of the signal. Use a Kaiser window of length 256 and shape parameter  $\beta = 5$ . Specify 220 samples of section-to-section overlap and 512 DFT points. Plot the frequency on the  $y$ -axis. Use the default colormap and view.

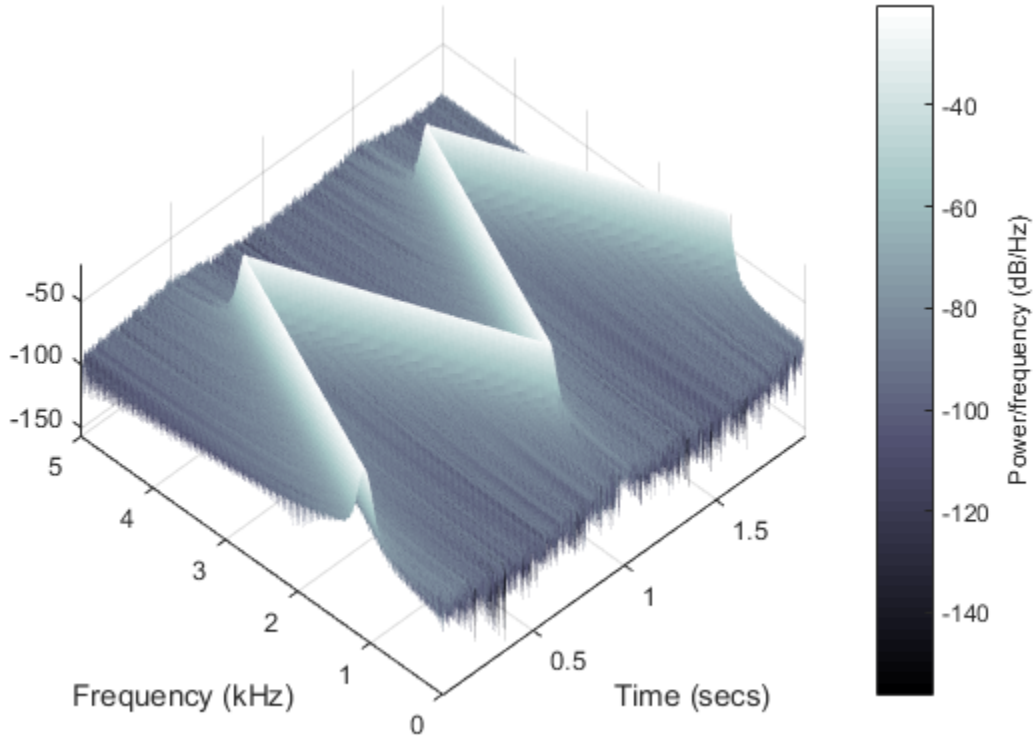
```
spectrogram(x1,kaiser(256,5),220,512,fs,'yaxis')
```





Change the view to display the spectrogram as a waterfall plot. Set the colormap to bone.

```
colormap bone  
view(-45,65)
```



- “Formant Estimation with LPC Coefficients”

## Input Arguments

### **x** — Input signal

vector

Input signal, specified as a row or column vector.

Example: `cos(pi/4*(0:159))+randn(1,160)` specifies a sinusoid embedded in white Gaussian noise.

Data Types: `single` | `double`

Complex Number Support: Yes

### **window** — Window

integer | vector | []

Window, specified as an integer or as a row or column vector. Use `window` to divide the signal into sections:

- If `window` is an integer, then `spectrogram` divides `x` into sections of length `window` and windows each section with a Hamming window of that length.
- If `window` is a vector, then `spectrogram` divides `x` into sections of the same length as the vector and windows each section using `window`.

If the length of `x` cannot be divided exactly into an integer number of sections with `noverlap` overlapping samples, then `x` is truncated accordingly.

If you specify `window` as empty, then `spectrogram` uses a Hamming window such that `x` is divided into eight sections with `noverlap` overlapping samples.

For a list of available windows, see “Windows”.

Example: `hann(N+1)` and `(1 - cos(2*pi*(0:N)'/N))/2` both specify a Hann window of length `N + 1`.

Data Types: `single` | `double`

### **noverlap** — Number of overlapped samples

positive integer | []

Number of overlapped samples, specified as a positive integer.

- If `window` is scalar, then `noverlap` must be smaller than `window`.
- If `window` is a vector, then `noverlap` must be smaller than the length of `window`.

If you specify `noverlap` as empty, then `spectrogram` uses a number that produces 50% overlap between sections. If the section length is unspecified, the function sets `noverlap` to  $\lfloor N_x/4.5 \rfloor$ , where  $N_x$  is the length of the input signal.

Data Types: `double` | `single`

### **nfft** — Number of DFT points

positive integer scalar | []

Number of DFT points, specified as a positive integer scalar. If you specify `nfft` as empty, then `spectrogram` sets the parameter to  $\max(256, 2^p)$ , where  $p = \lceil \log_2 N_x \rceil$  for an input signal of length  $N_x$ .

Data Types: `single` | `double`

### **w** — Normalized frequencies

vector

Normalized frequencies, specified as a vector. `w` must have at least two elements. Normalized frequencies are in rad/sample.

Example: `pi./[2 4]`

Data Types: `double` | `single`

### **f** — Cyclical frequencies

vector

Cyclical frequencies, specified as a vector. `f` must have at least two elements. The units of `f` are specified by the sample rate, `fs`.

Data Types: `double` | `single`

### **fs** — Sample rate

1 Hz (default) | positive scalar

Sample rate, specified as a positive scalar. The sample rate is the number of samples per unit time. If the unit of time is seconds, the sampling frequency is in Hz.

Data Types: `double` | `single`

### **freange** — Frequency range for PSD estimate

'onesided' | 'twosided' | 'centered'

Frequency range for the PSD estimate, specified as 'onesided', 'twosided', or 'centered'. For real-valued signals, the default is 'onesided'. For complex-valued signals, the default is 'twosided'.

- 'onesided' — returns the one-sided spectrogram of a real input signal. If `nfft` is even, then `pxx` has length  $nfft/2 + 1$  and is computed over the interval  $[0, \pi]$  rad/sample. If `nfft` is odd, then `pxx` has length  $(nfft + 1)/2$  and the interval is  $[0, \pi]$  rad/sample. If you specify `fs`, then the intervals are respectively  $[0, fs/2]$  cycles/unit time and  $[0, fs/2)$  cycles/unit time.

- `'twosided'` — returns the two-sided spectrogram of a real or complex signal. `pxx` has length `nfft` and is computed over the interval  $[0, 2\pi)$  rad/sample. If you specify `fs`, then the interval is  $[0, fs)$  cycles/unit time.
- `'centered'` — returns the centered two-sided spectrogram for a real or complex signal. `pxx` has length `nfft`. If `nfft` is even, then `pxx` is computed over the interval  $(-\pi, \pi]$  rad/sample. If `nfft` is odd, then `pxx` is computed over  $(-\pi, \pi)$  rad/sample. If you specify `fs`, then the intervals are respectively  $(-fs/2, fs/2]$  cycles/unit time and  $(-fs/2, fs/2)$  cycles/unit time.

Data Types: char

### **spectrumtype** — Power spectrum scaling

`'psd'` (default) | `'power'`

Power spectrum scaling, specified as `'psd'` or `'power'`.

- Omitting `spectrumtype`, or specifying `'psd'`, returns the power spectral density.
- Specifying `'power'` scales each estimate of the PSD by the equivalent noise bandwidth of the window. The result is an estimate of the power at each frequency.

Data Types: char

### **freqloc** — Frequency display axis

`'xaxis'` (default) | `'yaxis'`

Frequency display axis, specified as `'xaxis'` or `'yaxis'`.

- `'xaxis'` — displays frequency on the  $x$ -axis and time on the  $y$ -axis.
- `'yaxis'` — displays frequency on the  $y$ -axis and time on the  $x$ -axis.

This argument is ignored if you call `spectrogram` with output arguments.

Data Types: char

## Output Arguments

### **s** — Short-time Fourier transform

matrix

Short-time Fourier transform, returned as a matrix. Time increases across the columns of `s` and frequency increases down the rows, starting from zero.

- If  $x$  is a signal of length  $N_x$ , then  $s$  has  $k$  columns, where
  - $k = \lfloor (N_x - \text{noverlap}) / (\text{window} - \text{noverlap}) \rfloor$  if `window` is a scalar
  - $k = \lfloor (N_x - \text{noverlap}) / (\text{length}(\text{window}) - \text{noverlap}) \rfloor$  if `window` is a vector.
- If  $x$  is real and `nfft` is even, then  $s$  has  $(\text{nfft}/2 + 1)$  rows.
- If  $x$  is real and `nfft` is odd, then  $s$  has  $(\text{nfft} + 1)/2$  rows.
- If  $x$  is complex, then  $s$  has `nfft` rows.

Data Types: `double` | `single`

### **w** — Normalized frequencies

vector

Normalized frequencies, returned as a vector.  $w$  has a length equal to the number of rows of  $s$ .

Data Types: `double` | `single`

### **t** — Time instants

vector

Time instants, returned as a vector. The time values in  $t$  correspond to the midpoint of each section.

Data Types: `double` | `single`

### **f** — Cyclical frequencies

vector

Cyclical frequencies, returned as a vector.  $f$  has a length equal to the number of rows of  $s$ .

Data Types: `double` | `single`

### **pxx** — Power spectral density

matrix

Power spectral density (PSD), returned as a matrix.

- If  $x$  is real, then  $pxx$  contains the one-sided modified periodogram estimate of the PSD of each section.
- If  $x$  is complex, or if you specify a vector of frequencies, then  $pxx$  contains the two-sided modified periodogram estimate of the PSD of each section.

Data Types: `double` | `single`

## References

- [1] Oppenheim, Alan V., Ronald W. Schafer, and John R. Buck. *Discrete-Time Signal Processing*. 2nd Ed. Upper Saddle River, NJ: Prentice Hall, 1999.
- [2] Rabiner, Lawrence R., and Ronald W. Schafer. *Digital Processing of Speech Signals*. Englewood Cliffs, NJ: Prentice-Hall, 1978.

## See Also

`goertzel` | `periodogram` | `pwelch`

## spectrum

Spectral estimation

### Syntax

```
Hs = spectrum.estimate(input1, ...)
```

### Description

---

**Note:** The use of `spectrum.estimate` is not recommended. Use the corresponding function instead. See [Spectrum Estimation Methods](#) for the functional forms.

---

`Hs = spectrum.estimate(input1, ...)` returns a spectral estimation object `Hs` of type `estimate`. 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 `estimate` with `spectrum`.

---

#### Spectrum Estimation Methods

| <code>spectrum.estimate</code>    | Description         | Corresponding Function |
|-----------------------------------|---------------------|------------------------|
| <code>spectrum.burg</code>        | Burg                | <code>pburg</code>     |
| <code>spectrum.cov</code>         | Covariance          | <code>pcov</code>      |
| <code>spectrum.eigenvector</code> | Eigenvector         | <code>peig</code>      |
| <code>spectrum.mcov</code>        | Modified covariance | <code>pmcov</code>     |



| <b>spectrum.estmethod</b> | <b>Description</b>             | <b>Corresponding Function</b> |
|---------------------------|--------------------------------|-------------------------------|
| 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 1-1539).

---

## 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

| <b>Method</b> | <b>Description</b>  |
|---------------|---|
| msspectrum    | <p>Note that the <code>msspectrum</code> method is only available for the <code>periodogram</code> and <code>welch</code> spectrum estimation objects.</p> <p>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 <code>msspectrum</code>. Unlike the power spectral density (see <code>psd</code> 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</p> |

| Method                      | Description  |
|-----------------------------|--|
|                             | <p>in a frequency band. The units of the mean-squared spectrum are units of power.</p> <p><code>Hmss = msspectrum(Hs,X)</code> returns a mean-square spectrum object containing the mean-square (power) estimate of the discrete-time signal <code>X</code> using the spectrum object <code>Hs</code>. Default for real <code>X</code> is the 'onesided' Nyquist frequency range and for complex <code>X</code> the default is the 'twosided' Nyquist frequency range.</p> <p><code>Hmss</code> contains a vector of normalized frequencies <code>W</code>, at which the mean-square spectrum is estimated. For real signals, the range of <code>W</code> is <math>[0,\pi]</math> if the number of FFT points (<code>NFFT</code>) is even, and <math>[0,\pi)</math> if <code>NFFT</code> is odd. For complex signals, the range of <code>W</code> is <math>[0,2\pi)</math>. To estimate the spectrum on a vector of specific frequencies, see <code>FreqPoints</code> property below.</p> <p>The <code>msspectrum</code> method includes these properties, which you can set using this <code>msspectrum</code> method or via the <code>msspectrumopts</code> method. These properties are listed here and described in the <code>msspectrumopts</code> section below:</p> <p><code>SpectrumType</code> — 'onesided' or 'twosided'</p> <p><code>NormalizedFrequency</code> — normalizes frequency between 0 and 1</p> <p><code>Fs</code> — sampling frequency in Hz</p> <p><code>NFFT</code> — number of FFT points</p> <p><code>CenterDC</code> — shifts data and frequencies to center DC component</p> <p><code>FreqPoints</code> — 'All' or 'User Defined'</p> <p><code>FrequencyVector</code> — frequencies at which to compute spectrum</p> <p><code>ConfLevel</code> — confidence level to calculate the confidence interval. Value must be from 0 to 1.</p> <p>For example, <code>Hmss = msspectrum(Hs,X,'FreqPoints','User Defined', FreqVector,fvect)</code> returns a mean-square spectrum object where the spectrum is calculated only on the frequency points defined in the frequency vector, <code>fvect</code>.</p> <p><code>msspectrum(...)</code> with no output arguments plots the mean-square spectrum in dB.</p> |
| <code>msspectrumopts</code> | <code>Hopts = msspectrumopts(Hs)</code> returns an object that contains options for the spectrum object <code>Hs</code> .  |

| Method | Description  |
|--------|--|
|        | <p><code>Hopts = msspectrumopts(Hs,X)</code> returns an object with data-specific options and defaults.</p> <p>You can pass an <code>Hopts</code> options object as an argument to the <code>msspectrum</code> method. Any individual option you specify after the <code>Hopts</code> object overrides the value in <code>Hopts</code>. For example, <code>Hmss = msspectrum(Hs,X,Hopts, 'SpectrumType', 'twosided')</code> overrides the default <code>SpectrumType</code> value in <code>Hopts</code>.</p> <p>The following properties apply to both <code>msspectrumopts</code> and <code>msspectrum</code> methods.</p> <p><code>Hmss = msspectrum(..., 'SpectrumType', 'twosided')</code> returns the two-sided mean-square spectrum. The spectrum length (NFFT) is computed over <math>[0,2\pi)</math>, or if <code>Fs</code> is specified, <math>[0,Fs)</math>. Entering <code>'onesided'</code> returns the one-sided mean-square spectrum, which contains the total signal power in half the Nyquist range. Default is <code>'onesided'</code>.</p> <p><code>Hmss = msspectrum(Hs,X,'NormalizedFrequency',true)</code> returns a mean-square spectrum object with frequency values normalized between 0 and 1. Default is <code>true</code>.</p> <p><code>Hmss = msspectrum(Hs,X,'Fs',Fs)</code> returns a mean-square spectrum object computed as a function of frequency, where <code>Fs</code> is the sampling frequency in Hz. Note that you can set <code>Fs</code> only if <code>NormalizedFrequency</code> is set to <code>false</code>.</p> <p><code>Hmss = msspectrum(...,'NFFT',nfft)</code> specifies the number of FFT points to use. Valid values are a positive integer, <code>'Nextpow2'</code> or <code>'Auto'</code>. <code>'Nextpow2'</code> uses the next power of 2 greater than the input length or 256, whichever is greater. <code>'Auto'</code> uses the input length or 256, whichever is greater. Default is <code>'Nextpow2'</code>. Note that for <code>spectrum.welch</code>, <code>'Nextpow2'</code> and <code>'Auto'</code> are compared to the <code>SegmentLength</code> instead of the input length.</p> <p><code>Hmss = msspectrum(...,'Centerdc', true)</code> shifts the data and frequency values so that the DC component is at the center of the spectrum. Default is <code>false</code>.</p> |

| Method | Description  |
|--------|--|
|        | <p>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 <code>msspectrum</code> with a <code>FrequencyVector</code> property.</p> <pre>Hopts.FreqPoints = 'User Defined'</pre> <p>(Note that the default for <code>FreqPoints</code> is 'All', which causes <code>msspectrum</code> to use the NFFT property as described above.)</p> <p>Then, specify the frequency vector <code>F</code> to use.</p> <pre>Hopts.FrequencyVector = F</pre> <p>(Note that the default value for <code>FrequencyVector</code> is 'Auto'. In this case, the number of frequency points used follows the same rule as described for NFFT 'Auto' above.)</p> <p><code>Hmms = msspectrum(..., 'ConfLevel', p)</code> specifies the confidence level <code>p</code> for computing the confidence interval, which is an estimate of the error in the calculated mean-squared spectrum. The confidence level (<code>p</code>) is between 0 and 1. For example, <code>Hmss = msspectrum(Hs, X, 'ConfLevel', 0.95)</code> returns the 95% confidence interval.</p>  |
| psd    | <p>Note that <code>music</code> and <code>eigenvector</code> spectrum objects do not support the <code>psd</code> method. See the <code>pseudospectrum</code> method below.</p> <p>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 <code>msspectrum</code>, 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 <code>avgpower</code> method of <code>dspdata</code> for more information.</p> <p><code>Hpsd = psd (Hs, X)</code> returns a power spectral density object containing the power spectral density estimate of the discrete-time signal <code>X</code> using the spectrum object <code>Hs</code>. The PSD is the distribution of power per unit frequency. Default for real <code>X</code> is 'onesided' and for complex <code>X</code> is 'twosided'.</p> <p><code>Hpsd</code> contains a vector of normalized frequencies <code>W</code>, at which the PSD is estimated. For real signals, the range of <code>W</code> is <math>[0, \pi]</math> if the number of FFT points (NFFT) is even, and <math>[0, \pi)</math> if NFFT is odd. For complex signals, the range of <code>W</code> is <math>[0, 2\pi)</math>.</p> |

| Method  | Description  |
|---------|--|
|         | <p>The <code>psd</code> method includes these properties, which you can set using this <code>psd</code> method or via the <code>psdopts</code> method. These properties are listed here and described in the <code>psdopts</code> section below:</p> <p><code>SpectrumType</code> — 'onesided' or 'twosided'</p> <p><code>NormalizedFrequency</code> — normalizes frequency between 0 and 1</p> <p><code>Fs</code> — sampling frequency in Hz</p> <p><code>NFFT</code> — number of FFT points</p> <p><code>CenterDC</code> — shifts data and frequencies to center DC component</p> <p><code>FreqPoints</code> — 'All' or 'User Defined'</p> <p><code>FrequencyVector</code> — frequencies at which to compute spectrum</p> <p><code>ConfLevel</code> — confidence level to calculate the confidence interval. Value must be from 0 to 1.</p> <p>For example, <code>Hmss = psd(Hs,X,'FreqPoints','User Defined', FreqVector, fvect)</code> returns a PSD object where the spectrum is calculated only on the frequency points defined in the frequency vector, <code>fvect</code>.</p> <p><code>psd(...)</code> with no output arguments plots PSD in dB per unit frequency.</p> |
| psdopts | <p><code>Hopts = psdopts(Hs)</code> returns an object that contains options for the spectrum object <code>Hs</code>.</p> <p><code>Hopts = psdopts(Hs,X)</code> returns an object with data-specific options and defaults.</p> <p>You can pass an <code>Hopts</code> options object as an argument to the <code>psd</code> method. Any individual option you specify after the <code>Hopts</code> object overrides the value in <code>Hopts</code>. For example, <code>Hpsd = psd(Hs,X,Hopts,'SpectrumType','twosided')</code> overrides the <code>SpectrumType</code> value in <code>Hopts</code>.</p> <p>The following properties apply to both <code>psdmopts</code> and <code>psd</code> methods.</p> <p><code>Hpsd = psd(Hs,X,'SpectrumType','twosided')</code> returns the two-sided power spectral density of <code>X</code>. The spectrum length is <code>NFFT</code> and is computed over <math>[0,2\pi)</math> if <code>Fs</code> is not specified or <math>[0,Fs)</math> if <code>Fs</code> is specified. Entering 'onesided' returns the one-sided PSD, which contains the total signal power.</p>  |

| Method                             | Description  |
|------------------------------------|--|
|                                    | <p><code>Hmss = psd(Hs,X, 'NormalizedFrequency', true)</code> returns a power spectral density object with frequency values normalized between 0 and 1. Default is <code>true</code>.</p> <p><code>Hpsd = psd (... , 'Fs', Fs)</code> returns a power spectral density object computed as a function of frequency, where <code>Fs</code> is the sampling frequency in Hz.</p> <p><code>Hmss = psd(... , 'NFFT', nfft)</code> specifies the number of FFT points to use. Valid values are a positive integer, <code>'Nextpow2'</code> or <code>'Auto'</code>. <code>'Nextpow2'</code> uses the next power of 2 greater than the input length or 256, whichever is greater. <code>'Auto'</code> uses the input length or 256, whichever is greater. Default is <code>'Nextpow2'</code>. Note that for <code>spectrum.welch</code>, <code>'Nextpow2'</code> and <code>'Auto'</code> are compared to the <code>SegmentLength</code> instead of the input length.</p> <p><code>Hmss = psd (... , 'Centerdc', true)</code> shifts the data and frequency values so that the DC component is at the center of the spectrum. Default is <code>false</code>.</p> <p>To estimate the spectrum on a vector of specific frequencies, first set the number of frequency points to <code>'User Defined'</code>, which replaces the <code>NFFT</code> property of <code>psd</code> with a <code>FrequencyVector</code> property.<br/> <code>Hopts.FreqPoints = 'User Defined'</code><br/>                     (Note that the default for <code>FreqPoints</code> is <code>'All'</code> which causes <code>psd</code> to use the <code>NFFT</code> property as described above.)</p> <p><code>Hmms = psd(... , 'ConfLevel', p)</code> specifies the confidence level <code>p</code> for computing the confidence interval, which is an estimate of the error in the calculated PSD. The confidence level (<code>p</code>) is between 0 and 1. For example, <code>Hmss = psd(Hs,X, 'ConfLevel', 0.95)</code> returns the 95% confidence interval.</p> |
| <p><code>pseudospectrum</code></p> | <p>Note that this method is used for only <code>music</code> or <code>eigenvector</code> spectrum objects.</p> <p><code>Hps = pseudospectrum(Hs,X)</code> returns an object containing the pseudospectrum estimate of the discrete-time signal <code>X</code> using the spectrum object <code>Hs</code>. <code>Hs</code> must be a <code>music</code> or <code>eigenvector</code> object. Default for real <code>X</code> is <code>'half'</code> and for complex <code>X</code> is the <code>'whole'</code> Nyquist frequency range.</p>   |

| Method              | Description  |
|---------------------|--|
|                     | <p><code>Hps</code> contains a vector of normalized frequencies <math>W</math>, at which the pseudospectrum is estimated. For real signals, the range of <math>W</math> is <math>[0,\pi]</math> if the number of FFT points (NFFT) is even, and <math>[0,\pi)</math> if NFFT is odd. For complex signals, the range of <math>W</math> is <math>[0,2\pi)</math>.</p> <p>The <code>pseudospectrum</code> method includes these properties, which you can set using this <code>pseudospectrum</code> method or via the <code>pseudospectrumopts</code> method. These properties are described below:</p> <p><code>SpectrumRange</code> — 'half' or 'whole'</p> <p><code>NormalizedFrequency</code> — normalizes frequency between 0 and 1</p> <p><code>Fs</code> — sampling frequency in Hz</p> <p><code>NFFT</code> — number of FFT points</p> <p><code>CenterDC</code> — shifts data and frequencies to center DC component</p> <p><code>FreqPoints</code> — 'All' or 'User Defined'</p> <p><code>FrequencyVector</code> — frequencies at which to compute spectrum</p> <p>For example, <code>Hmss = psd(Hs,X,'FreqPoints','User Defined', FreqVector, fvect)</code> returns a PSD object where the spectrum is calculated only on the frequency points defined in the frequency vector, <code>fvect</code>.</p> <p><code>pseudospectrum(...)</code> with no output arguments plots the pseudospectrum in dB.</p> |
| pseudo-spectrumopts | <p><code>Hopts = pseudospectrumopts(Hs)</code> returns an object that contains options for the spectrum object <code>Hs</code>.</p> <p><code>Hopts = pseudospectrumopts(Hs,X)</code> returns an object with data-specific options and defaults. You can pass an <code>Hopts</code> options object as an argument to the <code>pseudospectrum</code> method. Any individual option you specify after the <code>Hopts</code> object overrides the value in <code>Hopts</code>. For example, <code>Hpspectrum = pseudospectrum(Hs,X, Hopts, 'SpectrumRange', 'whole')</code> overrides the <code>SpectrumRange</code> value in <code>Hopts</code>.</p> <p><code>Hmps = pseudospectrum(..., 'SpectrumRange', 'whole')</code> returns the pseudospectrum over the whole Nyquist range. The spectrum length is NFFT and is computed over <math>[0,2\pi)</math> if <code>Fs</code> is not specified or <math>[0,Fs)</math></p>  |

| Method   | Description   |
|----------|---|
|          | <p>if <code>Fs</code> is specified. Entering <code>'half'</code> returns the pseudospectrum calculated over half the Nyquist range.</p> <p><code>Hmss = pseudospectrum(Hs,X,'NormalizedFrequency',true)</code> returns a pseudospectrum object with frequency values normalized between 0 and 1. Default is <code>true</code>.</p> <p><code>Hps = pseudospectrum(Hs,X,'Fs',Fs)</code> returns a pseudospectrum object computed as a function of frequency, where <code>Fs</code> is the sampling frequency in Hz.</p> <p><code>Hps = pseudospectrum(...,'NFFT',nfft)</code> specifies the number of FFT points to use. Valid values are a positive integer, <code>'Nextpow2'</code> or <code>'Auto'</code>. <code>'Nextpow2'</code> uses the next power of 2 greater than the input length or 256, whichever is greater. <code>'Auto'</code> uses the input length or 256, whichever is greater. Default is <code>'Nextpow2'</code>.</p> <p><code>Hps = pseudospectrum(...,'Centerdc',true)</code> shifts the data and frequency values so that the DC component is at the center of the spectrum. The default value is <code>false</code>.</p> <p>To estimate the spectrum on a vector of specific frequencies, first set the number of frequency points to <code>'User Defined'</code>, which replaces the <code>NFFT</code> property of <code>pseudospectrum</code> with a <code>FrequencyVector</code> property.<br/> <code>Hopts.FreqPoints = 'User Defined'</code><br/>                     (Note that the default for <code>FreqPoints</code> is <code>'All'</code>, which causes <code>pseudospectrum</code> to use the <code>NFFT</code> property as described above.)</p> |
| powerest | <p>Note that <code>powerest</code> is available only for <code>music</code> and <code>eigenvector</code> spectrum objects.</p> <p><code>POW = powerest(Hs,X)</code> returns a vector <code>POW</code> containing estimates of the powers of the complex sinusoids in <code>X</code>. The input <code>X</code> can be a vector or a matrix. If it is a matrix it can be a data matrix, where <math>X^*X = R</math> or a correlation matrix <math>R</math>. The value the <code>InputType</code> property of <code>Hs</code> determines how <code>X</code> is interpreted. <code>Hs</code> must be a <code>music</code> or <code>eigenvector</code> spectrum object.</p> <p><code>[POW,W]=powerest(Hs,X)</code> returns <code>POW</code> and a vector <code>W</code> of the frequencies in rad/sample of the sinusoids in <code>X</code>.</p>   |



| Method | Description   |
|--------|---|
|        | <code>[POW,F]=powerest(Hs,X,Fs)</code> 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.

---

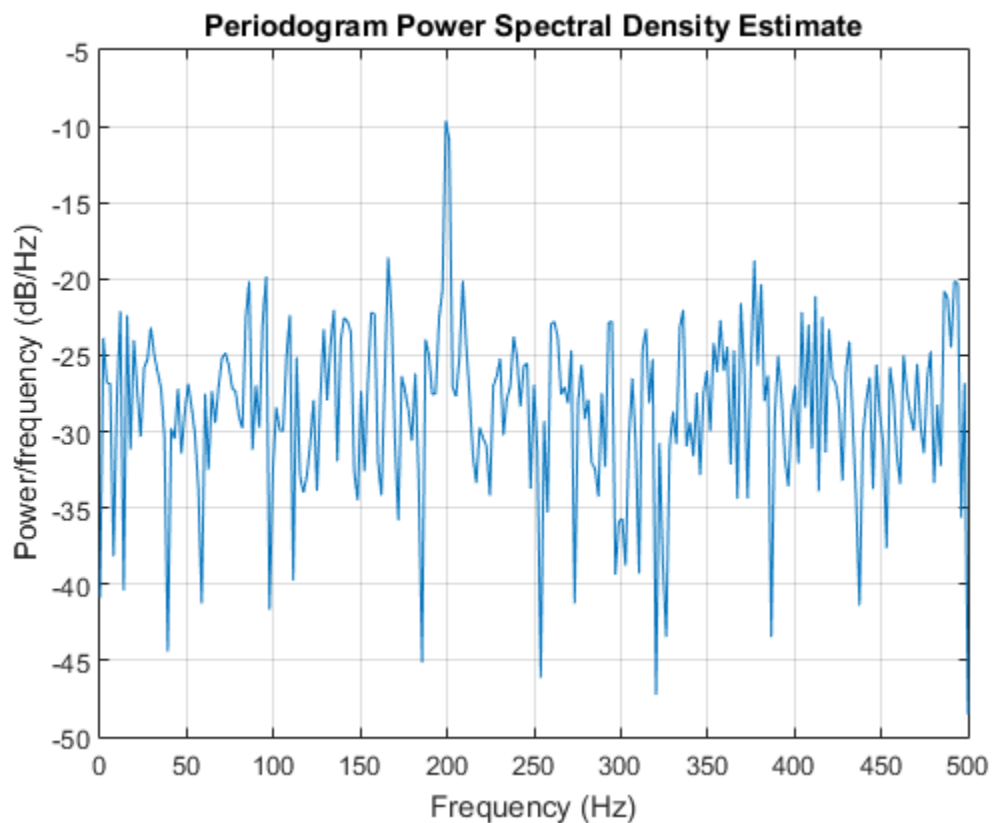
## Examples

### PSD via Periodogram

Generate a cosine of frequency 200 Hz sampled at 1 kHz for 300 ms. Add Gaussian white noise. View its power spectral density estimate generated with the `periodogram` algorithm.

```
Fs = 1000;  
t = 0:1/Fs:0.3;  
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.

# spectrum.burg

Burg spectrum

## Syntax

```
Hs = spectrum.burg
Hs = spectrum.burg(order)
```

## Description

---

**Note:** The use of `spectrum.burg` is not recommended. Use `pburg` instead.

---

`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 autoregressive (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 autoregressive model and view its power spectral density using the Burg algorithm.

```
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

`pburg` | `pcov` | `pmcov` | `pyulear`

## spectrum.cov

Covariance spectrum

### Syntax

```
Hs = spectrum.cov
Hs = spectrum.cov(order)
```

### Description

---

**Note:** The use of `spectrum.cov` is not recommended. Use `pcov` instead.

---

`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 autoregressive (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 autoregressive model and view its power spectral density using the covariance algorithm.

```
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

`pburg` | `pcov` | `pmcov` | `pyulear`

# spectrum.eigenvector

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)
Hs = spectrum.eigenvector(NSinusoids,SegmentLength,...OverlapPercent,WindowName)
```

## Description

---

**Note:** The use of `spectrum.eigenvector` is not recommended. Use `peig` instead.

---

`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   |
| WindowName     | 'Rectangular' | Window name string or 'User Defined' (see <code>window</code> for valid window names). For more information on each window, refer to its reference page. |

| Property Name     | Default Value | Description   |
|-------------------|---------------|---|
|                   |               | <p>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}.</p> <p>You can use <code>set</code> to change the value of the additional parameter or to define the MATLAB expression and parameters for a user-defined window (see <code>spectrum</code> for information on using <code>set</code>).</p> |
| SubspaceThreshold | 0             | <p>Threshold is the cutoff for signal and noise separation. The threshold is multiplied by <math>\lambda_{\min}</math>, the smallest estimated eigenvalue of the signal's correlation matrix. Eigenvalues below the threshold (<math>\lambda_{\min} * \text{threshold}</math>) are assigned to the noise subspace.</p>  |
| InputType         | 'Vector'      | <p>Type of input that will be used with this spectrum object. Valid values are 'Vector', 'DataMatrix' and 'CorrelationMatrix'.</p>  |

`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)`  
returns a spectrum object, `Hs`, with the specified subspace threshold.

`Hs = spectrum.eigenvector(NSinusoids,SegmentLength,...OverlapPercent,WindowName)`  
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.

```
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

`peig` | `pmusic`

## spectrum.mcov

Modified covariance spectrum

### Syntax

```
Hs = spectrum.mcov  
Hs = spectrum.mcov(order)
```

### Description

---

**Note:** The use of `spectrum.mcov` is not recommended. Use `pmcov` instead.

---

`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 autoregressive (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 autoregressive model and view its power spectral density using the modified covariance algorithm.

```
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

`pburg` | `pcov` | `pmcov` | `pyulear`

# spectrum.mtm

Thomson multitaper spectrum

## Syntax

```
Hs = spectrum.mtm
Hs = spectrum.mtm(TimeBW)
Hs = spectrum.mtm(DPSS,Concentrations)
Hs = spectrum.mtm(...,CombineMethod)
```

## Description

---

**Note:** The use of `spectrum.mtm` is not recommended. Use `pmtm` instead.

---

`Hs = spectrum.mtm` returns a default Thomson multitaper spectrum object, `HS` that defines the parameters for the Thomson 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.

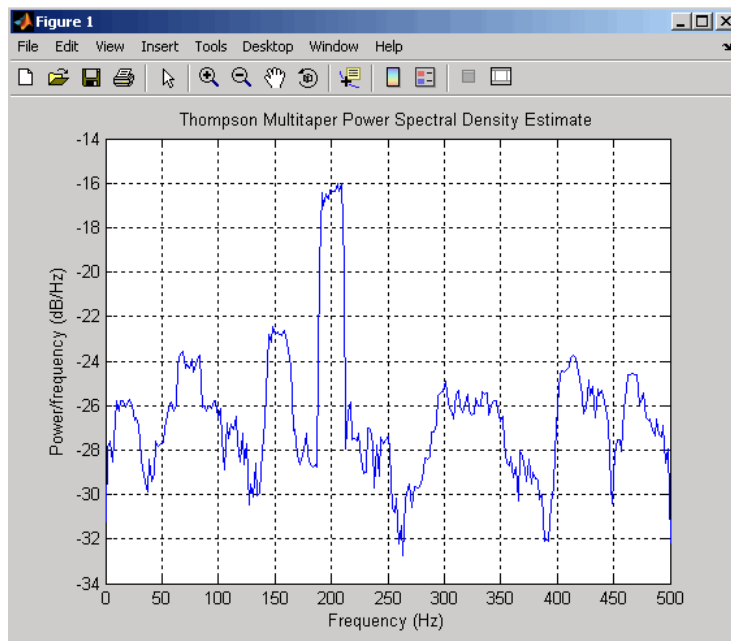
---

`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 Thomson 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

periodogram | pmtm | pwelch

## spectrum.music

Multiple signal classification spectrum

### Syntax

```
Hs = spectrum.music
Hs = spectrum.music(NSinusoids)
Hs = spectrum.music(NSinusoids,SegmentLength)
Hs = spectrum.music(NSinusoids,SegmentLength,...OverlapPercent)
Hs = spectrum.music(NSinusoids,SegmentLength,...OverlapPercent,WindowName)
Hs = spectrum.music(NSinusoids,SegmentLength,...OverlapPercent,WindowName,Subs)
Hs = spectrum.music(NSinusoids,SegmentLength,...OverlapPercent,WindowName,Subs)
```

### Description

---

**Note:** The use of `spectrum.music` is not recommended. Use `pmusic` instead.

---

`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  |
| WindowName     | 'Rectangular' | Window name string or 'User Defined' (see <code>window</code> for valid window names). For more |

| Property Name     | Default Value | Description   |
|-------------------|---------------|---|
|                   |               | <p>information on each window, refer to its reference page).</p> <p>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}.</p> <p>You can use <code>set</code> to change the value of the additional parameter or to define the MATLAB expression and parameters for a user-defined window (see <code>spectrum</code> for information on using <code>set</code>).</p> |
| SubspaceThreshold | 0             | <p>Threshold is the cutoff for signal and noise separation. The threshold is multiplied by <math>\lambda_{\min}</math>, the smallest estimated eigenvalue of the signal's correlation matrix. Eigenvalues below the threshold (<math>\lambda_{\min} * \text{threshold}</math>) are assigned to the noise subspace.</p>  |
| 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.music(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,Subs)`  
returns a spectrum object, `HS`, with the specified subspace threshold.

`Hs = spectrum.music(NSinusoids,SegmentLength,...OverlapPercent,WindowName,Substyp)`  
returns a spectrum object, `HS`, with the specified input type.

---

**Note** See `pmusic` for more information on the MUSIC algorithm.

---

## Examples

### MUSIC Pseudospectrum of a Sinusoidal Signal

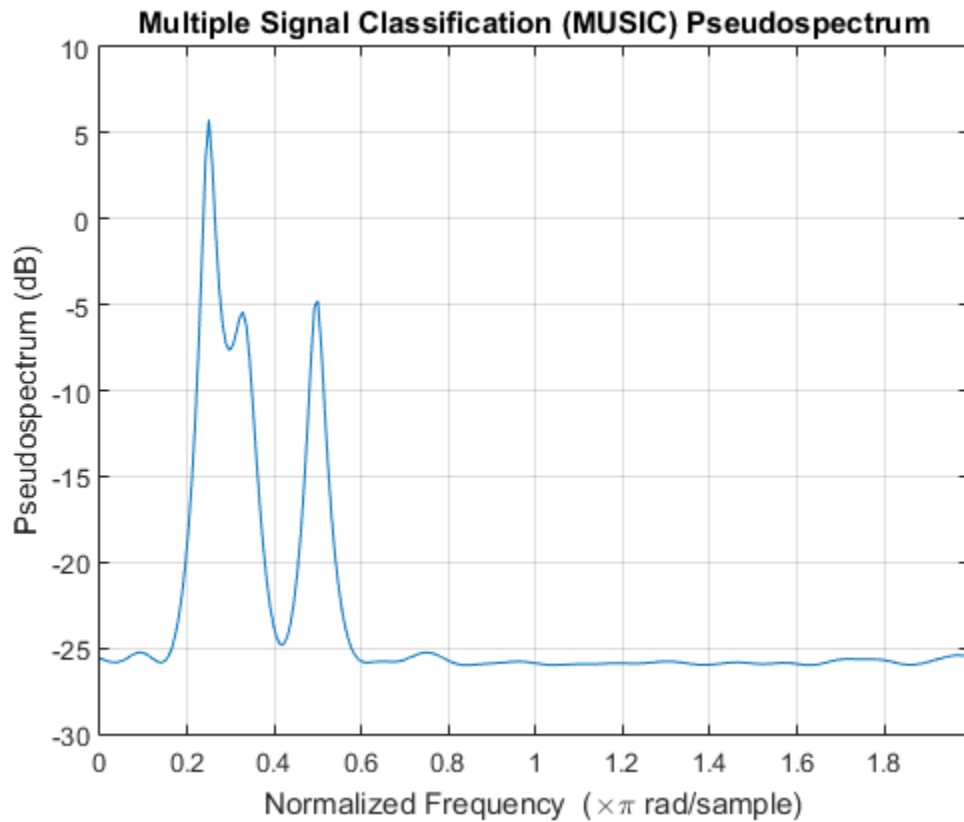
Define a complex signal with three sinusoids, add noise, and estimate its pseudospectrum using the MUSIC algorithm.

```
n = 0:99;
s = exp(1i*pi/2*n) + 2*exp(1i*pi/4*n) + exp(1i*pi/3*n) + randn(1,100);

Hs = spectrum.music(3,20);

pseudospectrum(Hs,s)
```





## References

- [1] Harris, Fredric. J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform." *Proceedings of the IEEE*. Vol. 66, January 1978, pp.51–83.

## See Also

peig | pmusic

## spectrum.periodogram

Periodogram spectrum

### Syntax

```
Hs = spectrum.periodogram
Hs = spectrum.periodogram(winname)
Hs = spectrum.periodogram({winname,winparameter})
```

### Description

---

**Note:** The use of `spectrum.periodogram` is not recommended. Use `periodogram` instead.

---

`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})`.

---

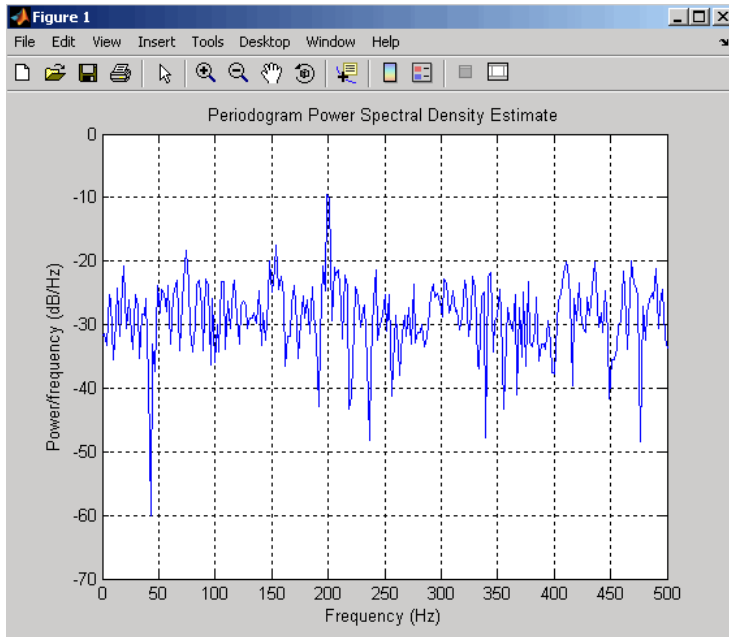
**Note** See `periodogram` for more information on the periodogram algorithm.

---

## 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, Fredric J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform." *Proceedings of the IEEE*. Vol. 66, January 1978, pp.51–83.

## See Also

[periodogram](#) | [pmtm](#) | [pwelch](#)

# spectrum.welch

Welch spectrum

## Syntax

```
Hs = spectrum.welch
Hs = spectrum.welch(WindowName)
Hs = spectrum.welch(WindowName,SegmentLength)
Hs = spectrum.welch(WindowName,SegmentLength,OverlapPercent)
```

## Description

---

**Note:** The use of `spectrum.welch` is not recommended. Use `pwelch` instead.

---

`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}  | 'Hamming',                 | Cell array containing the window name string or 'User Defined' and, if used for the particular window, an optional parameter value. (See <code>window</code> for valid window names and for more information on each window, refer to its reference page.)<br><br>You can use <code>set</code> to change the value of the additional parameter or to define the MATLAB expression and parameters for a user-defined window. (See |
| Cell array containing WindowName and optional window parameter | SamplingFlag:<br>symmetric |  |

| Property Name | Default Value                                | Description   |
|---------------|--|---|
|               |  | spectrum for information on using set.)   |
| WindowName    | 'Hamming',<br><br>SamplingFlag:<br>symmetric | <p>Valid windowname strings are:</p> <ul style="list-style-type: none"> <li>'Bartlett'</li> <li>'Bartlett-Hanning'</li> <li>'Blackman'</li> <li>'Blackman-Harris'</li> <li>'Bohman'</li> <li>'Chebyshev'</li> <li>'Flat Top'</li> <li>'Gaussian'</li> <li>'Hamming'</li> <li>'Hann'</li> <li>'Kaiser'</li> <li>'Nuttall'</li> <li>'Parzen'</li> <li>'Rectangular'</li> <li>'Triangular'</li> <li>'Tukey'</li> <li>'User Defined'</li> </ul> <p>Window names must be enclosed in single quotes, such as <code>spectrum.welch('tukey')</code> or <code>spectrum.welch({'tukey',0.7})</code>.</p> <p>See <code>window</code> and the corresponding window function page for window parameter information. You can use <code>set</code> to change the value of the additional window parameter or to define the MATLAB expression and parameters for a user-defined window (see <code>spectrum</code> for information on using <code>set</code>).</p> |

| Property Name  | Default Value | Description   |
|----------------|---------------|---|
| 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 produces 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.

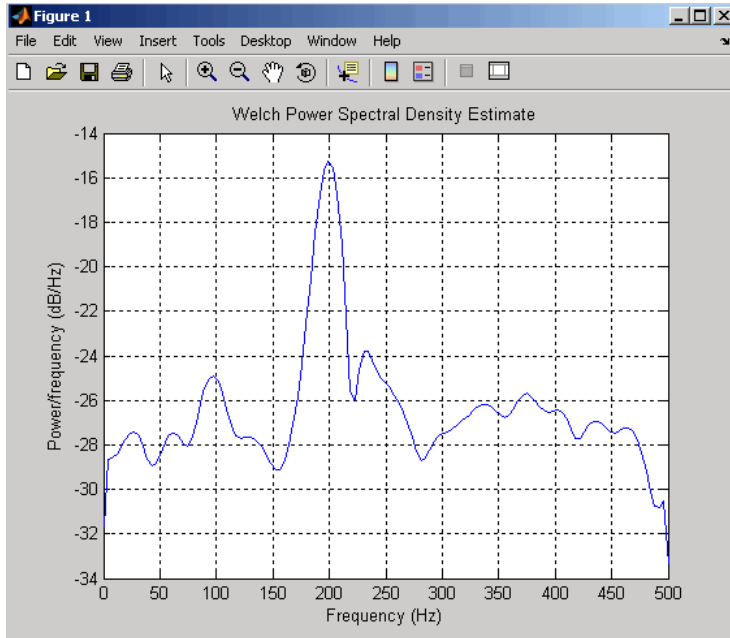
---

## 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 obsolete `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

periodogram | pmtm | pwelch

## spectrum.yulear

Yule-Walker spectrum object

### Syntax

```
Hs = spectrum.yulear  
Hs = spectrum.yulear(order)
```

### Description

---

**Note:** The use of `spectrum.yulear` is not recommended. Use `pyulear` instead.

---

`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 autoregressive (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 autoregressive model and view its spectral content using the Yule-Walker algorithm.

```
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**

pburg | pcov | pmcov | pyulear

## **sptool**

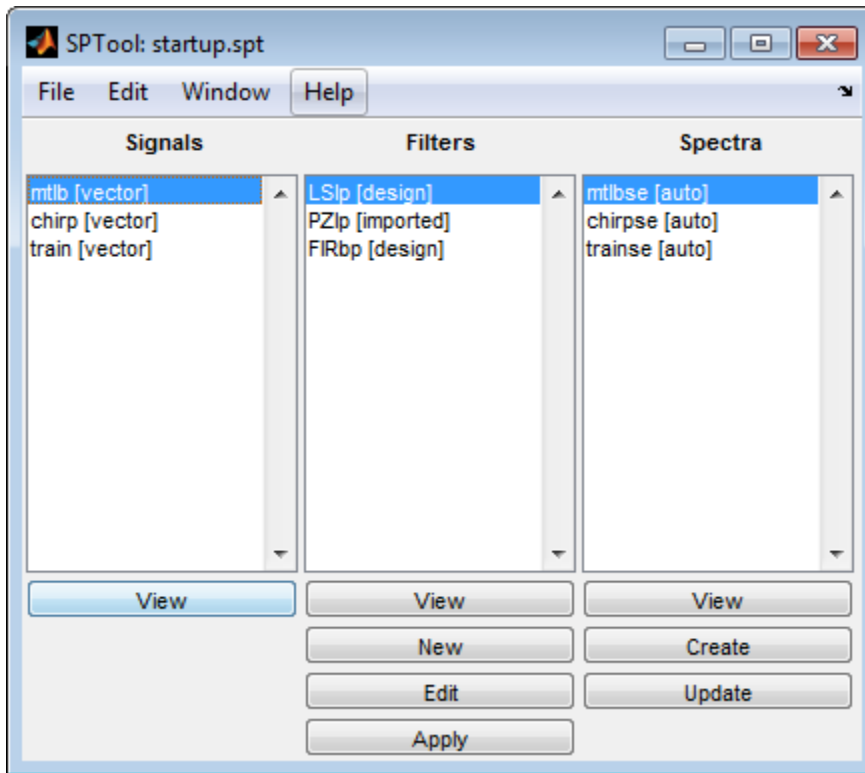
Open interactive digital signal processing tool

### **Syntax**

`sptool`

### **Description**

The command, `sptool`, opens SPTool, a suite of four tools: Signal Browser, Filter Design and Analysis Tool, FVTool, and Spectrum Viewer. These tools 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 suite opens.



Using SPTool, you can:

- Analyze signals listed in the **Signals** list box with the Signal Browser.
- Design or edit filters with the Filter Design and Analysis Tool (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 Design and Analysis Tool, and Spectrum Viewer.

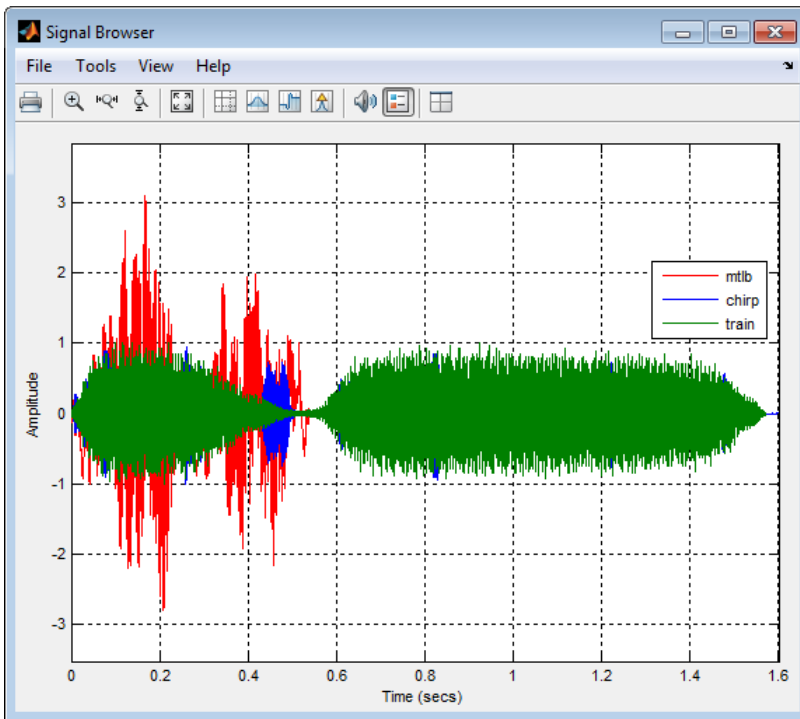
You can activate all four integrated signal processing tools from SPTool.

- “Signal Browser” on page 1-1568
- “Filter Design and Analysis Tool” on page 1-1600

- “Filter Visualization Tool” on page 1-1601
- “Spectrum Viewer” on page 1-1602

## Signal Browser

The Signal Browser, hereafter referred to as the scope, 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.



See the following sections for more information on the Signal Browser:

- “Displaying Multiple Signals” on page 1-1569
- “Signal Display” on page 1-1572
- “Toolbar” on page 1-1574

- “Measurements Panels” on page 1-1578
- “Visuals — Time Domain Options” on page 1-1592
- “Style Dialog Box” on page 1-1598

## Displaying Multiple Signals

### Multiple Signal Input

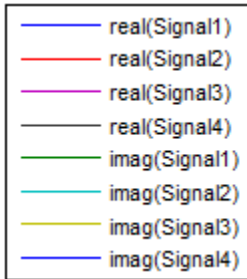
Select more than one signal in the **Signals** list box to show multiple signals within the same display or on separate displays. By default, the signals appear as different-colored lines on the same display. The signals can have different dimensions, sample rates, and data types. Each signal can be either real or complex valued.

### Multiple Signal Colors

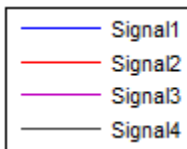
By default, Signal Browser has a white axes background and chooses line colors for each channel in a manner similar to the MATLAB `plot` function. Signal Browser considers each of the real and imaginary components of the input signals to be a channel, and assigns each channel a line color in the following order:



- 1 Blue
- 2 Dark Green
- 3 Red
- 4 Cyan
- 5 Purple
- 6 Dark Yellow
- 7 Black

If there are more than 7 channels, the scope repeats this order to assign line colors to the remaining channels. For example, if you select 4 complex-valued input signals, the following legend appears in the display.




If all the input signals are real-valued, Signal Browser skips the line colors that would be associated with their imaginary components. For example, if you select 4 real-valued input signals, the following legend appears in the display.



To manually modify any line color, select **View > Style** to open the Style dialog box. Next to **Properties for line**, select the signal name whose color you want to change. Then, next to **Line**, click the Line color button () and select any color from the palette. To change the axes background color, click the Axes background color button () and select any color from the palette.

## Multiple Displays

You can display multiple channels of data on different displays in the scope window. In the scope toolbar, select **View > Layout**, or select the Layout button ()

---

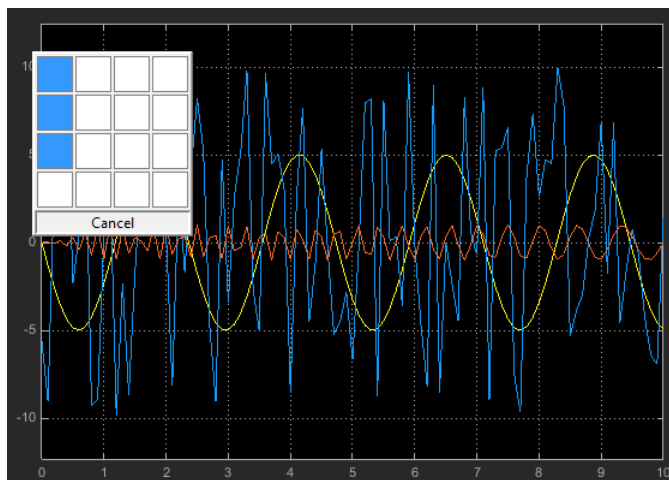
**Note:** The **Layout** menu item and button are not available when the scope is in snapshot mode.

---

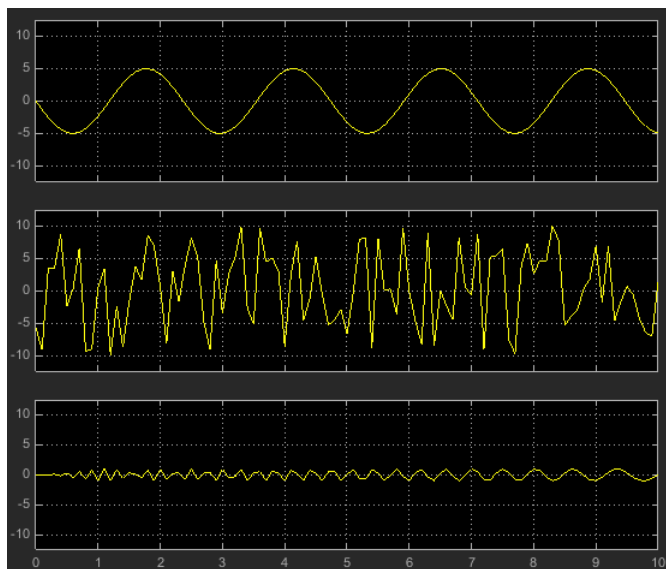
This feature allows you to tile the window into a number of separate displays, up to a grid of 4 rows and 4 columns. For example, if there are three inputs to the scope, you can



display the signals in separate displays by selecting row 3, column 1, as shown in the following figure.



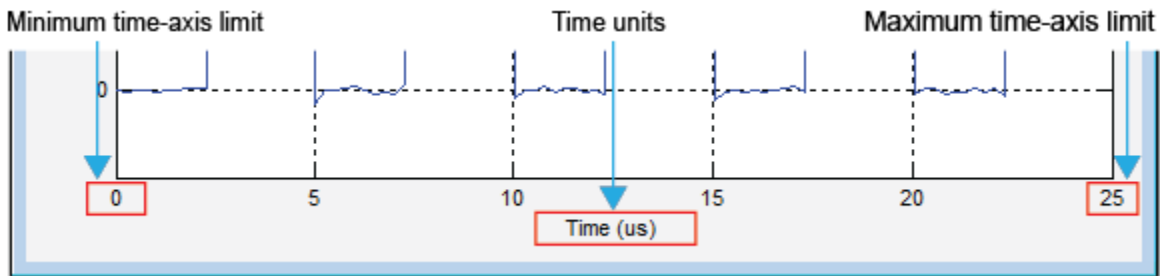
After you select row 3, column 1, the scope window is partitioned into three separate displays, as shown in the following figure.



When you use the Layout option to tile the window into multiple displays, the display highlighted in blue is referred to as the *active display*. The scope dialog boxes reference the active display.

## Signal Display

The Signal Browser uses the longest time length of all the input signals selected in the **Signals** list box for the time range. To communicate the array of times that corresponds to the current display, the scope uses the **Minimum time-axis limit**, **Time units**, and **Maximum time-axis limit** indicators on the scope window. The following figure highlights these aspects of the Signal Browser window.



- **Minimum time-axis limit** — The Signal Browser sets the minimum *time-axis* limit to 0.
- **Maximum time-axis limit** — The Signal Browser sets the maximum *time-axis* limit to the final time step of the longest input signal.
- **Time units** — The units used to describe the *time-axis*. The Signal Browser sets the time units using the value of the **Time Units** parameter on the **Main** tab of the Visuals:Time Domain Options dialog box. By default, this parameter is set to **Metric (based on Time Span)** and displays in metric units such as microseconds, milliseconds, minutes, days, etc. You can change the unit of measure to **Seconds** to always display the *time-axis* values in units of seconds. You can change it to **None** to suppress the display of units of measure on the *time-axis*. When you set this parameter to **None**, then the Signal Browser shows only the word **Time** on the *time-axis*.

To hide both the word **Time** and the values on the *time-axis*, set the **Show time-axis labels** parameter to **None**. To hide both the word **Time** and the values on the *time-axis* in all displays except the bottom ones in each column of displays, set

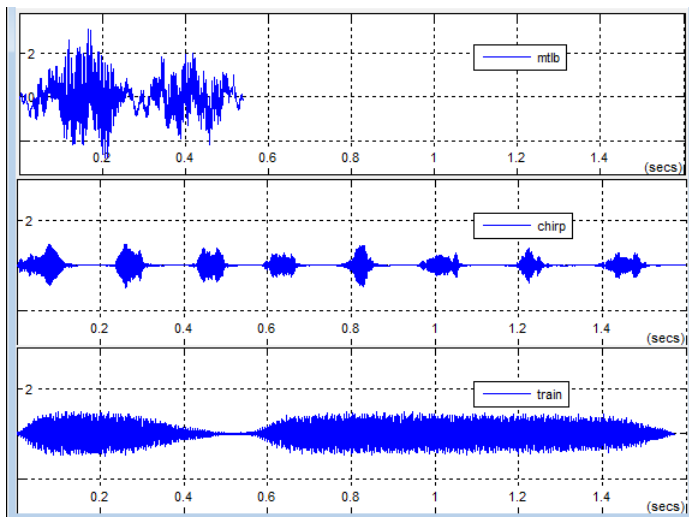
this parameter to **Bottom Displays Only**. This behavior differs from that of the Simulink Scope block, which always shows the values but never shows a label on the  $x$ -axis.

## Signal Names and Legend Strings

Signal Browser uses the names of the signals in the SPTool as the strings displayed in the legends. If you change the name of any selected signal in the **Signals** list box, its corresponding legend string in Signal Browser changes immediately. To change the name of any selected signal, from the SPTool menu, select **Edit > Name**. Signal Browser automatically updates the legend string to reflect the new signal name you entered. Similarly, if you modify any string in a legend in Signal Browser, SPTool updates the corresponding signal name in the **Signals** list box.

## Axes Maximization

You can specify whether to display the Signal Browser in maximized axes mode. In this mode, the axes are expanded to fill the entire display. In each display, there is no space to show titles or axis labels. The minimum and maximum *time*-axis limits are located at the far-left and far-right edges of the display. The values at the axis tick marks appear as grid lines on top of the axes. The following figure highlights how three displays appear in maximized axes mode in the Signal Browser window.



To enable or disable this mode, in the Signal Browser menu, select **View > Properties** to bring up the Visuals:Time Domain Options dialog box. In the **Main** pane, you can set the **Maximize axes** parameter to one of the following options:


- **Auto** — In this mode, the axes appear maximized in all displays only if the **Title** and **Y-Axis label** parameters are empty for every display. If you enter any value in any display for either of these parameters, the axes are not maximized.
- **On** — In this mode, the axes appear maximized in all displays. Any values entered into the **Title** and **Y-Axis label** parameters are hidden.
- **Off** — In this mode, none of the axes appear maximized.

See the “Visuals — Time Domain Options” on page 1-1592 section for more information.



## Toolbar




The Signal Browser toolbar contains the following buttons.


### Print Button


| Button   | Menu Location          | Shortcut Keys | Description   |
|--|------------------------|---------------|---|
|  | <b>File &gt; Print</b> | <b>Ctrl+P</b> | Print the current scope window. To print the current scope window to a figure rather than sending it to your printer, select <b>File &gt; Print to figure</b> . |

### Zoom and Axes Control Buttons



| Button  | Menu Location             | Shortcut Keys | Description  |
|---|---------------------------|---------------|--|
|  | <b>Tools &gt; Zoom In</b> | N/A           | When this tool is active, you can zoom in on the scope window. To do so, click in the center of your area of interest, or click and drag your cursor to draw a rectangular area of interest inside the scope window. |
|  | <b>Tools &gt; Zoom X</b>  | N/A           | You access the Zoom X button from the menu under the Zoom In icon. When this tool is active,   |



| Button  | Menu Location                         | Shortcut Keys | Description  |
|---|---------------------------------------|---------------|--|
|   |                                       |               | you can zoom in on the $x$ -axis. To do so, click inside the scope window, or click and drag your cursor along the $x$ -axis over your area of interest.   |
|  | <b>Tools &gt; Zoom Y</b>              | N/A           | You access the Zoom Y button from the menu under the Zoom In icon. When this tool is active, you can zoom in on the $y$ -axis. To do so, click inside the scope window, or click and drag your cursor along the $y$ -axis over your area of interest.  |
|  | <b>Tools &gt; Pan</b>                 | N/A           | You access the Pan button from the menu under the Zoom In icon. When this tool is active, you can pan on the scope window. To do so, click in the center of your area of interest and drag your cursor to the left, right, up, or down, to move the position of the display.   |
|  | <b>Tools &gt; Scale Y-Axis Limits</b> | <b>Ctrl+A</b> | <p>Click this button to scale the axes in the active scope window.</p> <p>Alternatively, you can enable automatic axes scaling by selecting one of the following options from the <b>Tools</b> menu:</p> <ul style="list-style-type: none"> <li>• <b>Automatically Scale Axes Limits</b> — When you select this option, the scope scales the axes as needed during simulation.</li> <li>• <b>Scale Axes Limits after 10 Updates</b> — When you select this option, the scope scales the axes after 10 updates. The scope does not scale the axes again during the simulation.</li> <li>• <b>Scale Axes Limits at Stop</b> — When you select this option, the scope scales the axes each time the simulation is stopped.</li> </ul> |

| Button  | Menu Location                                 | Shortcut Keys | Description   |
|---|---|---------------|---|
|  | <b>Tools &gt;<br/>Scale X-Axis<br/>Limits</b> | N/A           | <p>You access the Scale X-Axis Limits button from the menu under the current Axis Limits icon. Click this button to scale the axes in the X direction in the active scope window.</p> <p>Alternatively, you can enable automatic axes scaling by selecting one of the following options from the <b>Tools</b> menu:</p> <ul style="list-style-type: none"> <li>• <b>Automatically Scale Axes Limits</b> — When you select this option, the scope scales the axes as needed during simulation.</li> <li>• <b>Scale Axes Limits after 10 Updates</b> — When you select this option, the scope scales the axes after 10 updates. The scope does not scale the axes again during the simulation.</li> <li>• <b>Scale Axes Limits at Stop</b> — When you select this option, the scope scales the axes each time the simulation is stopped.</li> </ul> |




| Button  | Menu Location                                 | Shortcut Keys | Description   |
|---|---|---------------|---|
|  | <b>Tools &gt; Scale X &amp; Y Axes Limits</b> | N/A           | <p>You access the Scale X &amp; Y Axes Limits button from the menu under the current Axis Limits icon. Click this button to scale the axes in both the X and Y directions in the active scope window.</p> <p>Alternatively, you can enable automatic axes scaling by selecting one of the following options from the <b>Tools</b> menu:</p> <ul style="list-style-type: none"> <li>• <b>Automatically Scale Axes Limits</b> — When you select this option, the scope scales the axes as needed during simulation.</li> <li>• <b>Scale Axes Limits after 10 Updates</b> — When you select this option, the scope scales the axes after 10 updates. The scope does not scale the axes again during the simulation.</li> <li>• <b>Scale Axes Limits at Stop</b> — When you select this option, the scope scales the axes each time the simulation is stopped.</li> </ul> |

## Measurements Buttons

|   |   |     |   |
|---|---|-----|---|
|  | <b>Tools &gt; Measurements &gt; Cursor Measurements</b> | N/A | <p>Open or close the <b>Cursor Measurements</b> panel. This panel puts screen cursors on all the displays.</p> <p>See the section for more information.</p>   |
|  | <b>Tools &gt; Measurements &gt; Signal Statistics</b>   | N/A | <p>Open or close the <b>Signal Statistics</b> panel. This panel displays the maximum, minimum, peak-to-peak difference, mean, median, RMS values of a selected signal, and the times at which the maximum and minimum occur.</p> <p>See the section for more information.</p> |

|   |  |     |  |
|---|--|-----|--|
|  | <b>Tools &gt; Measurements &gt; Bilevel Measurements</b> | N/A | Open or close the <b>Bilevel Measurements</b> panel. This panel displays information about a selected signal's transitions, overshoots or undershoots, and cycles.<br><br>See the section for more information.  |
|  | <b>Tools &gt; Measurements &gt; Peak Finder</b>          | N/A | Open or close the <b>Peak Finder</b> panel. This panel displays maxima and the times at which they occur, allowing the settings for peak threshold, maximum number of peaks, and peak excursion to be modified.<br><br>See the section for more information. |

## Other Buttons

|   |  |     |   |
|---|--|-----|---|
|  | <b>Tools &gt; Play Selected Signal</b> | N/A | Play an audio signal. The function <code>soundsc</code> is used to play the signal.   |
|  | <b>View &gt; Show All Legends</b>      | N/A | Show a legend that matches each line style to a signal name in every display.   |
|  | <b>View &gt; Layout</b>                | N/A | Arrange the layout of displays in the Signal Browser. This feature allows you to tile your screen into a number of separate displays, up to a grid of 4 rows and 4 columns. You may find multiple displays useful when you select multiple input signals in SPTool. The default display is 1 row and 1 column. See the “Multiple Displays” on page 1-1570 section for more information. |

You can control whether this toolbar appears in the Signal Browser window. From the Signal Browser menu, select **View > Toolbar**.











## Measurements Panels



The Measurements panels are the five panels that appear at the right side of the Signal Browser. These panels are labeled **Trace selection**, **Cursor measurements**, **Signal statistics**, **Bilevel measurements**, and **Peak finder**.



## Measurements Panel Buttons

Each of the Measurements panels contains the following buttons that enable you to modify the appearance of the current panel.

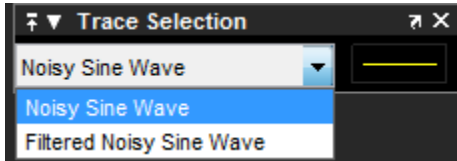
| Button  | Description  |
|---|--|
|    | Move the current panel to the top. When you are displaying more than one panel, this action moves the current panel above all the other panels.  |
|    | Collapse the current panel. When you first enable a panel, by default, it displays one or more of its panes. Click this button to hide all of its panes to conserve space. After you click this button, it becomes the expand button  .   |
|    | Expand the current panel. This button appears after you click the collapse button to hide the panes in the current panel. Click this button to display the panes in the current panel and show measurements again. After you click this button, it becomes the collapse button  again. |
|    | Undock the current panel. This button lets you move the current panel into a separate window that can be relocated anywhere on your screen. After you click this button, it becomes the dock button  in the new window.   |
|    | Dock the current panel. This button appears only after you click the undock button. Click this button to put the current panel back into the right side of the Scope window. After you click this button, it becomes the undock button  again.  |
|  | Close the current panel. This button lets you remove the current panel from the right side of the Scope window.  |

Some panels have their measurements separated by category into a number of panes. Click the pane expand button  to show each pane that is hidden in the current panel. Click the pane collapse button  to hide each pane that is shown in the current panel.

## Trace Selection Panel


When you use the scope to view multiple signals, the Trace Selection panel appears if you have more than one signal displayed and you click any of the other Measurements panels. The Measurements panels display information about only the signal chosen in

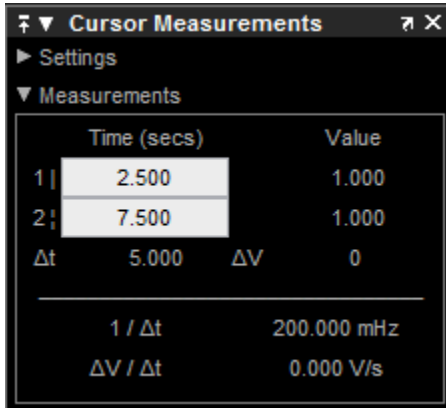
this panel. Choose the signal name for which you would like to display time domain measurements. See the following figure.



You can choose to hide or display the **Trace Selection** panel. In the Scope menu, select **Tools > Measurements > Trace Selection**.

## Cursor Measurements

The **Cursor Measurements** panel displays screen cursors. In the Scope menu, select **Tools > Measurements > Cursor Measurements**. Alternatively, in the Scope toolbar, click the Cursor Measurements  button.




You can use the mouse or the left and right arrow keys to move vertical or waveform cursors and the up and down arrow keys for horizontal cursors.

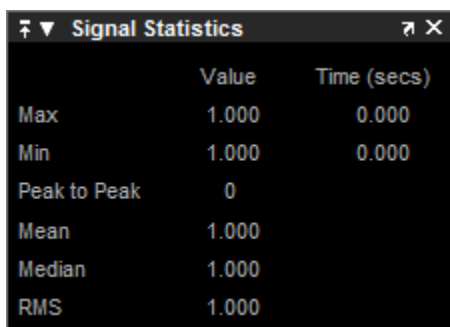
The **Measurements** pane shows the time and value measurements.

- **1 |**— Shows or enables you to modify the time or value at cursor number one, or both.
- **2 :**— Shows or enables you to modify the time or value at cursor number two, or both.

- $\Delta t$ — Shows the absolute value of the difference in the times between cursor number one and cursor number two.
- $\Delta V$ — Shows the absolute value of the difference in signal amplitudes between cursor number one and cursor number two.
- $1/\Delta t$ — Shows the rate, the reciprocal of the absolute value of the difference in the times between cursor number one and cursor number two.
- $\Delta V/\Delta t$ — Shows the slope, the ratio of the absolute value of the difference in signal amplitudes between cursors to the absolute value of the difference in the times between cursors.

## Signal Statistics Panel

The **Signal Statistics** panel displays the maximum, minimum, peak-to-peak difference, mean, median, and RMS values of a selected signal. It also shows the  $x$ -axis indices at which the maximum and minimum values occur. In the Scope menu, select **Tools > Measurements > Signal Statistics**. Alternatively, in the scope toolbar, click the Signal Statistics  button.



|              | Value | Time (secs) |
|--------------|-------|-------------|
| Max          | 1.000 | 0.000       |
| Min          | 1.000 | 0.000       |
| Peak to Peak | 0     |             |
| Mean         | 1.000 |             |
| Median       | 1.000 |             |
| RMS          | 1.000 |             |

The statistics shown are:

- **Max** — The maximum or largest value within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the MATLAB `max` function reference.
- **Min** — The minimum or smallest value within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the MATLAB `min` function reference.

- **Peak to Peak** — The difference between the maximum and minimum values within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `peak2peak` function reference.
- **Mean** — The average or mean of all the values within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the MATLAB `mean` function reference.
- **Median** — The median value within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the MATLAB `median` function reference.
- **RMS** — Shows the difference between the maximum and minimum values within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `rms` function reference.

When you use the zoom options in the Scope, the Signal Statistics measurements automatically adjust to the time range shown in the display. For example, you can zoom in on one pulse to make the **Signal Statistics** panel display information about only that particular pulse.

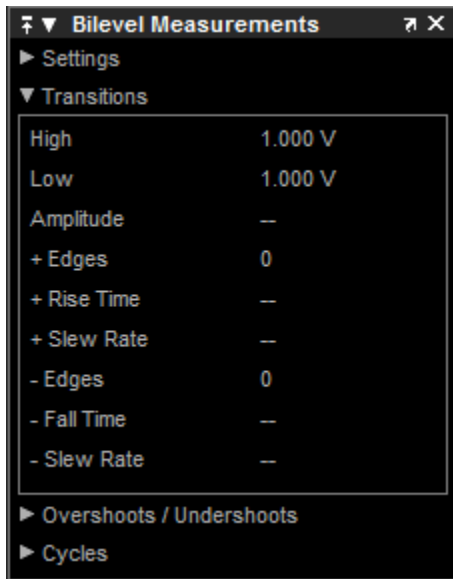
The Signal Statistics measurements are valid for any units of the input signal. The letter after the value associated with each measurement represents the appropriate International System of Units (SI) prefix, such as *m* for *milli-*. For example, if the input signal is measured in volts, an *m* next to a measurement value indicates that this value is in units of millivolts.

## Bilevel Measurements Panel

The **Bilevel Measurements** panel shows information about transitions, overshoots, undershoots, and cycles for a selected signal. You can choose to hide or display the **Bilevel Measurements** panel. In the scope menu, select **Tools > Measurements > Bilevel Measurements**. Alternatively, in the scope toolbar, you can select the Bilevel

Measurements  button.

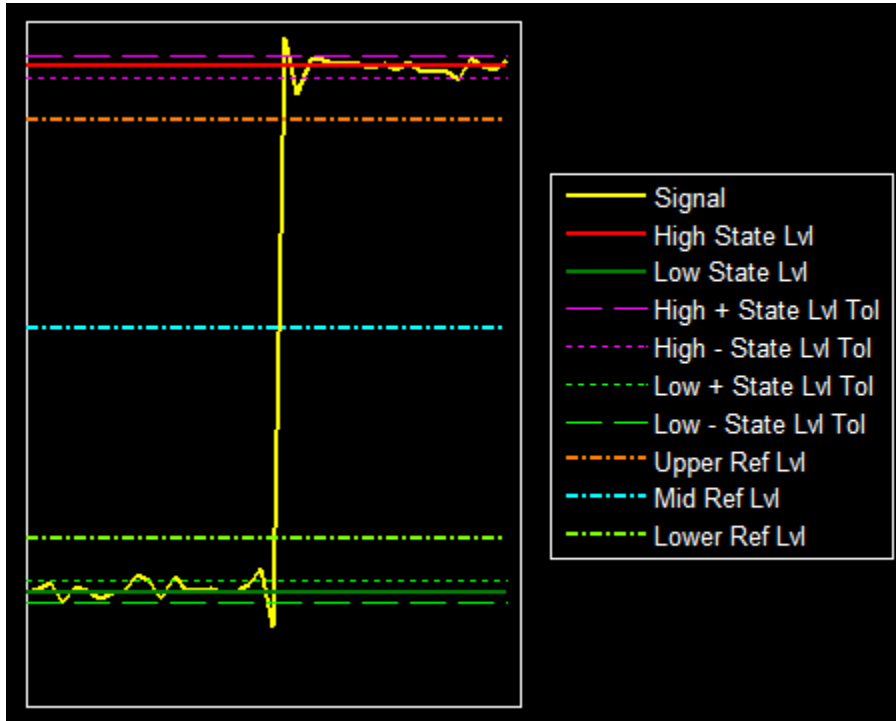
When you use the zoom options in the Scope, the bilevel measurements automatically adjust to the time range shown in the display. For example, you can zoom in on one rising edge to make the **Bilevel Measurements** panel display information about only that particular rising edge. This feature does not apply to the **High** and **Low** measurements.



The **Bilevel Measurements** panel is separated into four panes, labeled **Settings**, **Transitions**, **Overshoots / Undershoots**, and **Cycles**. You can expand each pane to see the available options.

The **Settings** pane enables you to modify the properties used to calculate various measurements involving transitions, overshoots, undershoots, and cycles. You can modify the high-state level, low-state level, state-level tolerance, upper-reference level, mid-reference level, and lower-reference level, as shown in the following figure.

## Bilevel Measurements Plot



- **Auto State Level** — When this check box is selected, the Bilevel measurements panel autodetects the high- and low- state levels of a bilevel waveform. For more information on the algorithm this option uses, see the Signal Processing Toolbox `statelevels` function reference. When this check box is cleared, you can enter in values for the high- and low- state levels manually.
  - **High** — Manually specify the value for a positive polarity or high-state level.
  - **Low** — Manually specify the value for a negative polarity or low-state level.
- **State Level Tolerance** — Tolerance within which the initial and final levels of each transition must be within their respective state levels. This value is expressed as a percentage of the difference between the high- and low-state levels.
- **Upper Ref Level** — Used to compute the end of the rise-time measurement or the start of the fall time measurement. This value is expressed as a percentage of the difference between the high- and low-state levels.

- **Mid Ref Level** — Used to determine when a transition occurs. This value is expressed as a percentage of the difference between the high- and low- state levels. The mid-reference level is shown as a horizontal line, and its corresponding mid-reference level instant is shown as a vertical line.
- **Lower Ref Level** — Used to compute the end of the fall-time measurement or the start of the rise-time measurement. This value is expressed as a percentage of the difference between the high- and low-state levels.
- **Settle Seek** — The duration after the mid-reference level instant when each transition occurs used for computing a valid settling time. This value is equivalent to the input parameter, `D`, which you can set when you run the `settlingtime` function. The settling time is displayed in the **Overshoots/Undershoots** pane.

The **Transitions** pane displays calculated measurements associated with the input signal changing between its two possible state level values, high and low. The Transition measurements assume that the amplitude of the input signal is in units of volts. Convert all input signals to volts for the Transition measurements to be valid.

A positive-going transition, or *rising edge*, in a bilevel waveform is a transition from the low-state level to the high-state level. A positive-going transition has a slope value greater than zero. Whenever there is a plus sign (+) next to a text label, this symbol refers to measurement associated with a rising edge, a transition from a low-state level to a high-state level.

A negative-going transition, or *falling edge*, in a bilevel waveform is a transition from the high-state level to the low-state level. A negative-going transition has a slope value less than zero. Whenever there is a minus sign (–) next to a text label, this symbol refers to measurement associated with a falling edge, a transition from a high-state level to a low-state level.

- **High** — The high-amplitude state level of the input signal over the duration of the **Time Span** parameter. You can set **Time Span** in the **Main** pane of the Visuals —Time Domain Properties dialog box. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `statelevels` function reference.
- **Low** — The low-amplitude state level of the input signal over the duration of the **Time Span** parameter. You can set **Time Span** in the **Main** pane of the Visuals —Time Domain Properties dialog box. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `statelevels` function reference.

- **Amplitude** — Difference in amplitude between the high-state level and the low-state level.
- **+ Edges** — Total number of positive-polarity, or rising, edges counted within the displayed portion of the input signal.
- **+ Rise Time** — Average amount of time required for each rising edge to cross from the lower-reference level to the upper-reference level. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `risetime` function reference.
- **+ Slew Rate** — Average slope of each rising-edge transition line within the upper- and lower-percent reference levels in the displayed portion of the input signal. The region in which the slew rate is calculated appears in gray.

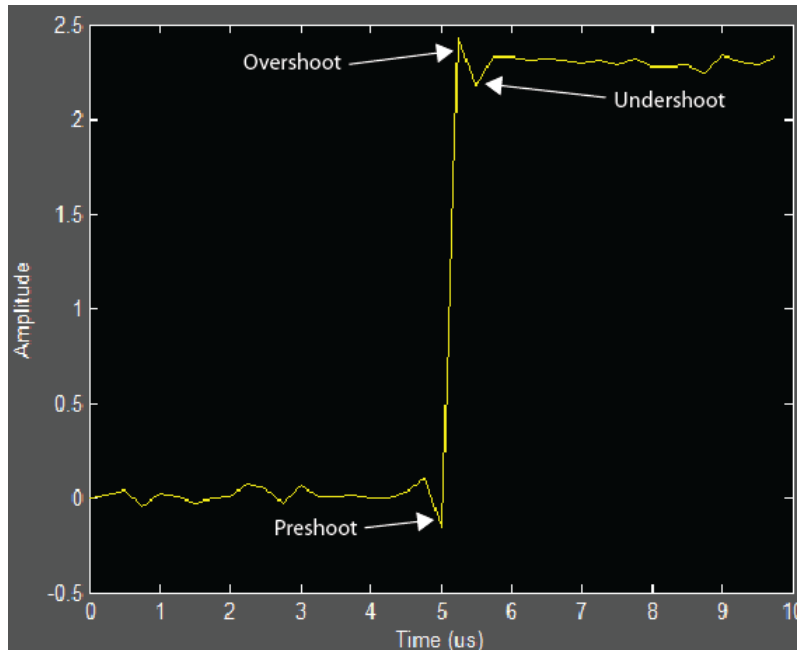
For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `slewrates` function reference.

- **- Edges** — Total number of negative-polarity or falling edges counted within the displayed portion of the input signal.
- **- Fall Time** — Average amount of time required for each falling edge to cross from the upper-reference level to the lower-reference level. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `falltime` function reference.
- **- Slew Rate** — Average slope of each falling edge transition line within the upper- and lower-percent reference levels in the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `slewrates` function reference.

The **Overshoots/Undershoots** pane displays calculated measurements involving the distortion and damping of the input signal. *Overshoot* and *undershoot* refer to the amount that a signal, respectively, exceeds and falls below its final steady-state value. *Preshoot* refers to the amount before a transition that a signal varies from its initial steady-state value. This figure shows preshoot, overshoot, and undershoot for a rising-edge transition.

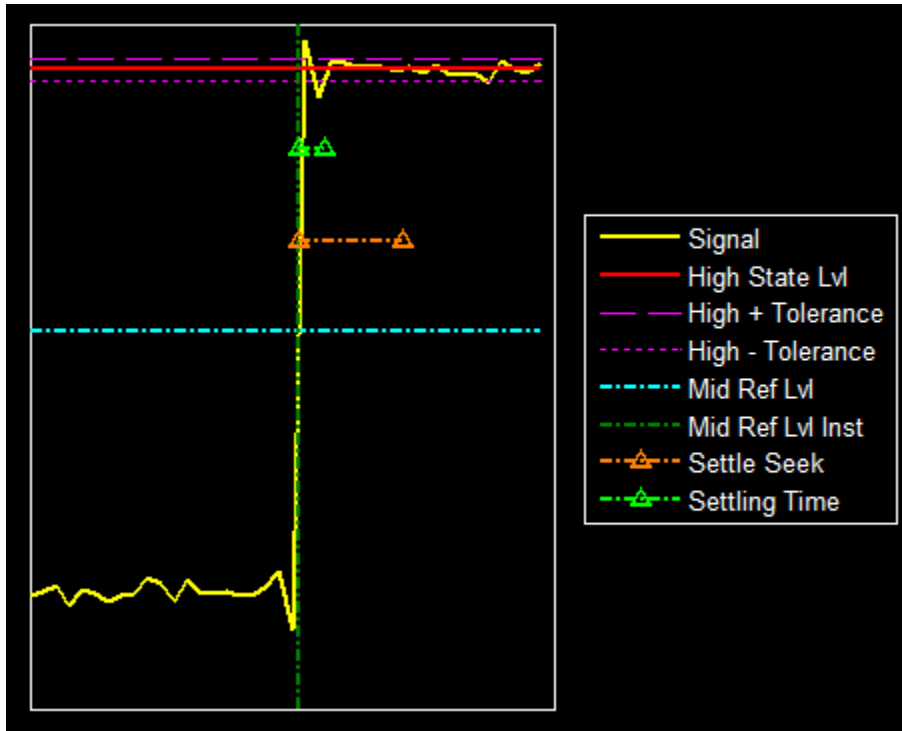


## Overshoot/Undershoot Plot



- **+ Preshoot** — Average lowest aberration in the region immediately preceding each rising transition.
- **+ Overshoot** — Average highest aberration in the region immediately following each rising transition. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `overshoot` function reference.
- **+ Undershoot** — Average lowest aberration in the region immediately following each rising transition. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `undershoot` function reference.
- **+ Settling Time** — Average time required for each rising edge to enter and remain within the tolerance of the high-state level for the remainder of the settle seek duration. The settling time is the time after the mid-reference level instant when the signal crosses into and remains in the tolerance region around the high-state level. This crossing is illustrated in the following figure.

## Settling Time Plot



You can modify the settle seek duration parameter in the **Settings** pane. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `settlingtime` function reference.

- – **Preshoot** — Average highest aberration in the region immediately preceding each falling transition.
- – **Overshoot** — Average highest aberration in the region immediately following each falling transition. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `overshoot` function reference.
- – **Undershoot** — Average lowest aberration in the region immediately following each falling transition. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `undershoot` function reference.

- – **Settling Time** — Average time required for each falling edge to enter and remain within the tolerance of the low-state level for the remainder of the settle seek duration. The settling time is the time after the mid-reference level instant when the signal crosses into and remains in the tolerance region around the low-state level. You can modify the settle seek duration parameter in the **Settings** pane. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `settlingtime` function reference.

The **Cycles** pane displays calculated measurements of to repetitions or trends in the displayed portion of the input signal.


- **Period** — Average duration between adjacent edges of identical polarity within the displayed portion of the input signal. The Bilevel measurements panel calculates period as follows. It takes the difference between the mid-reference level instants of the initial transition of each positive-polarity pulse and the next positive-going transition. These mid-reference level instants appear as red dots.

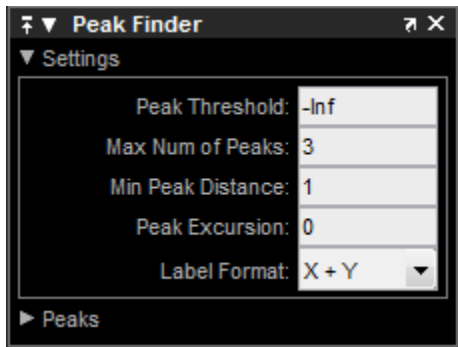
For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `pulseperiod` function reference.

- **Frequency** — Reciprocal of the average period. Whereas period is typically measured in some metric form of seconds, or seconds per cycle, frequency is typically measured in hertz or cycles per second.
- + **Pulses** — Number of positive-polarity pulses counted.
- + **Width** — Average duration between rising and falling edges of each positive-polarity pulse within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `pulsewidth` function reference.
- + **Duty Cycle** — Average ratio of pulse width to pulse period for each positive-polarity pulse within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `dutycycle` function reference.
- – **Pulses** — Number of negative-polarity pulses counted.
- – **Width** — Average duration between rising and falling edges of each negative-polarity pulse within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `pulsewidth` function reference.
- – **Duty Cycle** — Average ratio of pulse width to pulse period for each negative-polarity pulse within the displayed portion of the input signal. For more information

on the algorithm this measurement uses, see the Signal Processing Toolbox `dutycycle` function reference.

## Peak Finder Panel

The **Peak Finder** panel displays the maxima, showing the  $x$ -axis values at which they occur. This panel allows you to modify the settings for peak threshold, maximum number of peaks, and peak excursion. You can choose to hide or display the **Peak Finder** panel. In the scope menu, select **Tools > Measurements > Peak Finder**. Alternatively, in the scope toolbar, select the Peak Finder  button.



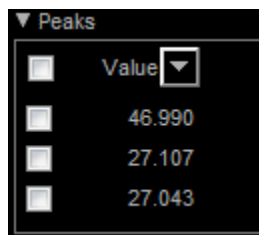
The **Peak finder** panel is separated into two panes, labeled **Settings** and **Peaks**. You can expand each pane to see the available options.

The **Settings** pane enables you to modify the parameters used to calculate the peak values within the displayed portion of the input signal. For more information on the algorithms this pane uses, see the Signal Processing Toolbox `findpeaks` function reference.

- **Peak Threshold** — The level above which peaks are detected. This setting is equivalent to the `MINPEAKHEIGHT` parameter, which you can set when you run the `findpeaks` function.
- **Max Num of Peaks** — The maximum number of peaks to show. The value you enter must be a scalar integer from 1 through 99. This setting is equivalent to the `NPEAKS` parameter, which you can set when you run the `findpeaks` function.
- **Min Peaks Distance** — The minimum number of samples between adjacent peaks. This setting is equivalent to the `MINPEAKDISTANCE` parameter, which you can set when you run the `findpeaks` function.


- **Peak Excursion** — The minimum height difference between a peak and its neighboring samples. The peak excursion setting is equivalent to the `THRESHOLD` parameter, which you can set when you run the `findpeaks` function.
- **Label Format** — The coordinates to display next to the calculated peak values on the plot. To see peak values, expand the **Peaks** pane and select the check boxes associated with individual peaks of interest. By default, both  $x$ -axis and  $y$ -axis values are displayed on the plot. Select which axes values you want to display next to each peak symbol on the display.
  - **X+Y** — Display both  $x$ -axis and  $y$ -axis values.
  - **X** — Display only  $x$ -axis values.
  - **Y** — Display only  $y$ -axis values.

The **Peaks** pane displays all of the largest calculated peak values. It also shows the coordinates at which the peaks occur, using the parameters you define in the **Settings** pane. You set the **Max Num of Peaks** parameter to specify the number of peaks shown in the list.



The numerical values displayed in the **Value** column are equivalent to the `pks` output argument returned when you run the `findpeaks` function. The numerical values displayed in the second column are similar to the `locs` output argument returned when you run the `findpeaks` function.

The Peak Finder displays the peak values in the **Peaks** pane. By default, the **Peak Finder** panel displays the largest calculated peak values in the **Peaks** pane in decreasing order of peak height. Use the sort descending button (▼) to rearrange the category and order by which Peak Finder displays peak values. Click this button again to sort the peaks in ascending order instead. When you do so, the arrow changes direction to become the sort ascending button (▲). A filled sort button indicates that the peak values are currently sorted in the direction of the button arrow. If the sort button is not filled

() , then the peak values are sorted in the opposite direction of the button arrow. The **Max Num of Peaks** parameter still controls the number of peaks listed.

Use the check boxes to control which peak values are shown on the display. By default, all check boxes are cleared and the **Peak Finder** panel hides all the peak values. To show all the peak values on the display, select the check box in the top-left corner of the **Peaks** pane. To hide all the peak values on the display, clear this check box. To show an individual peak, select the check box directly to the left of its **Value** listing. To hide an individual peak, clear the check box directly to the left of its **Value** listing.

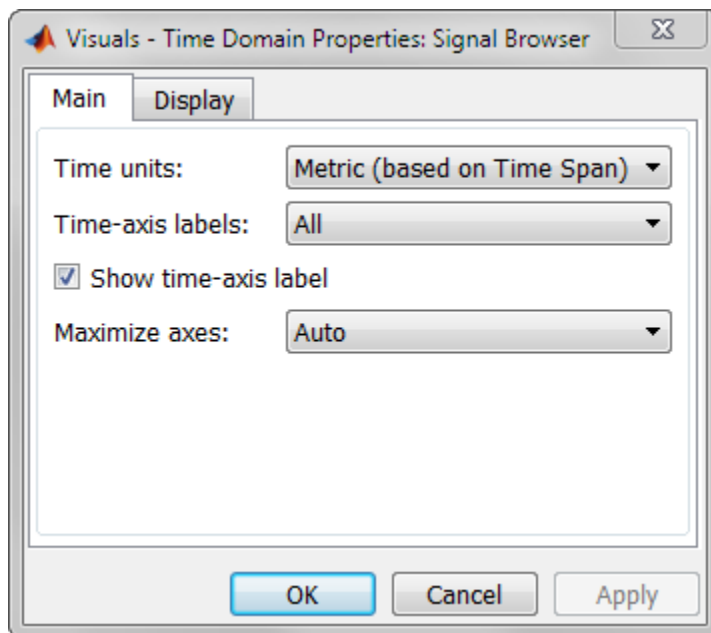
The Peaks are valid for any units of the input signal. The letter after the value associated with each measurement indicates the abbreviation for the appropriate International System of Units (SI) prefix, such as *m* for *milli-*. For example, if the input signal is measured in volts, an *m* next to a measurement value indicates that this value is in units of millivolts.

## Visuals — Time Domain Options

The Visuals — Time Domain Properties dialog box controls the visual configuration settings of the Signal Browser display. From the menu, select **View > Configuration Properties** to open this dialog box.

### Main Pane

The **Main** pane of the Visuals — Time Domain Properties dialog box appears as follows.



### Time units

Specify the units used to describe the *time*-axis. The default setting is **Metric**. You can select one of the following options.

- **Metric** — In this mode, the Signal Browser converts the times on the *time*-axis to some metric units such as milliseconds, microseconds, days, etc. The Signal Browser chooses the appropriate metric units, based on the minimum *time*-axis limit and the maximum *time*-axis limit of the window.
- **Seconds** — In this mode, the Signal Browser always displays the units on the *time*-axis as seconds.
- **None** — In this mode, the Signal Browser displays no units on the *time*-axis. The Signal Browser shows only the word **Time** on the *time*-axis.

### Time-axis labels

Specify how to display the time units used to describe the *time*-axis. The default setting is **All**. You can select one of the following options.

- **All** — In this mode, the *time*-axis labels appear in all displays.

- **None** — In this mode, the *time*-axis labels do not appear in the displays.
- **Bottom Displays Only** — In this mode, the *time*-axis labels appear only in the bottom row of the displays.

## Show time-axis label

Select to turn on *time*-axis label display.

## Maximize axes

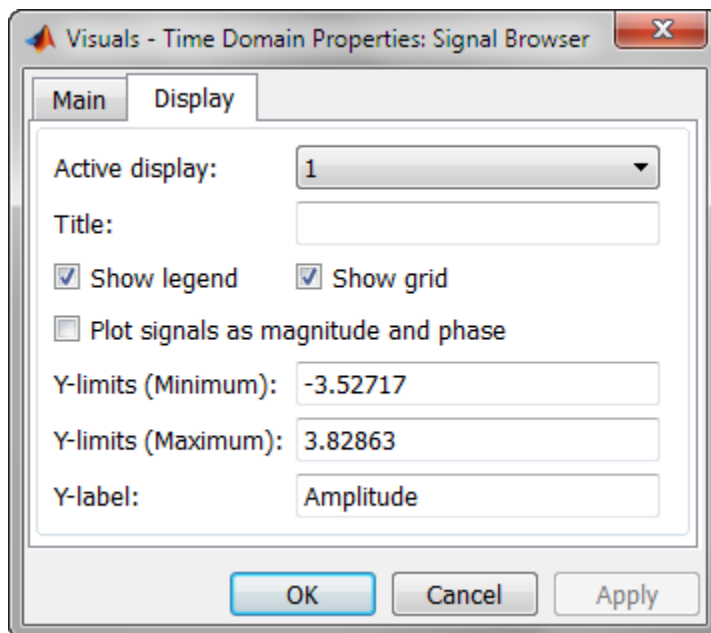
Specify whether to display the Signal Browser in maximized axes mode. In this mode, each of the axes is expanded to fit into the entire display. In each display, there is no space to show labels. Tick mark values are shown on top of the plotted data. The default setting is **Auto**. You can select one of the following options:

- **Auto** — In this mode, the axes appear maximized in all displays only if the **Title** and **Y-Axis label** parameters are empty for every display. If you enter any value in any display for either of these parameters, the axes are not maximized.
- **On** — In this mode, the axes appear maximized in all displays. Any values entered into the **Title** and **Y-Axis label** parameters are hidden.
- **Off** — In this mode, none of the axes appear maximized.

## Display Pane

The **Display** pane of the Visuals — Time Domain Properties dialog box appears as follows.





### Active display

Specify the active display as an integer to get and set relevant properties. The number of a display corresponds to its column-wise placement index. Set this property to control which display has its axes colors, line properties, marker properties, and visibility changed.

When you use the Layout option to tile the window into multiple displays, the display highlighted in blue is referred to as the *active display*. The default setting is 1.

### Title

Specify the active display title as a string. By default, the active display has no title.

### Show legend

Select this check box to show the legend in the display. The channel legend displays a name for each channel of each input signal. When the legend appears, you can place it anywhere inside of the scope window. To turn off the legend, clear the **Show legend** check box. This parameter applies only when the Spectrum **Type** is Power or Power density.

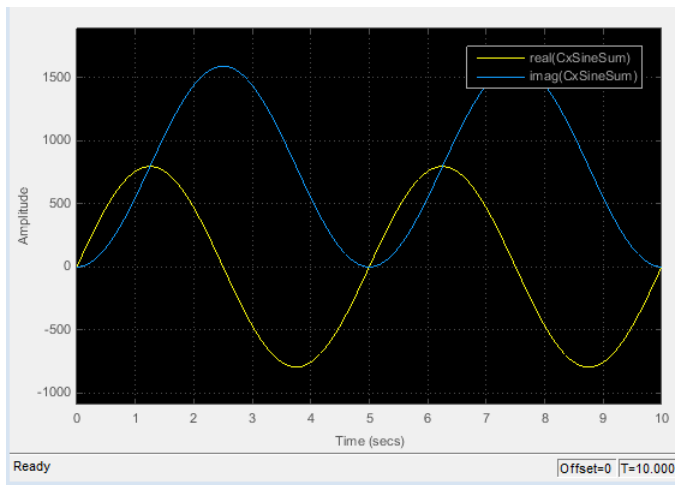
You can edit the name of any channel in the legend. To do so, double-click the current name, and enter a new channel name. By default, if the signal has multiple channels, the scope uses an index number to identify each channel of that signal. To change the appearance of any channel of any input signal in the scope window, from the scope menu, select **View > Style**.

## Show grid

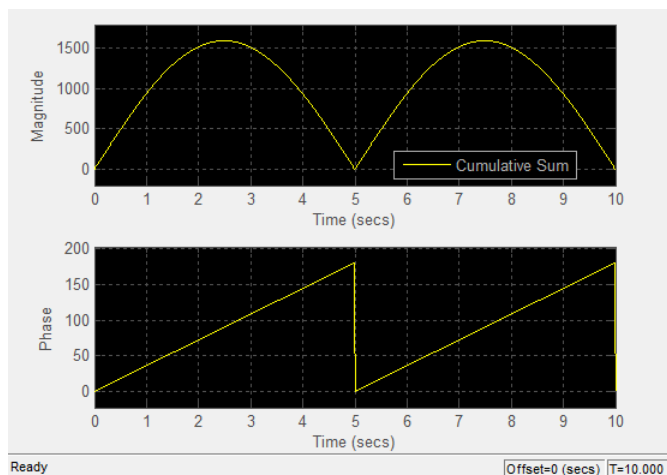
When you select this check box, a grid appears in the display of the scope figure. To hide the grid, clear this check box.

## Plot signals as magnitude and phase

When you select this check box, the scope splits the display into a magnitude plot and a phase plot. By default, this check box is cleared. If the input signal has complex values, the scope plots the real and imaginary portions on the same axes. These real and imaginary portions appear as different-colored lines on the same axes, as shown in the following figure.



Selecting this check box and clicking the **Apply** or **OK** button changes the display. The magnitude of the input signal appears on the top axes and its phase, in degrees, appears on the bottom axes. See the following figure.



This feature is useful for complex-valued input signals. If the input is a real-valued signal, selecting this check box returns the absolute value of the signal for the magnitude. The phase is 0 degrees for nonnegative input and 180 degrees for negative input.

### Y-limits (Minimum)

Specify the minimum value of the y-axis.

When you select the **Plot signal(s) as magnitude and phase** check box, the value of this property always applies to the magnitude plot on the top axes. The phase plot on the bottom axes is always limited to a minimum value of -180 degrees.

### Y-limits (Maximum)

Specify the maximum value of the y-axis.

When you select the **Plot signal(s) as magnitude and phase** check box, the value of this property always applies to the magnitude plot on the top axes. The phase plot on the bottom axes is always limited to a maximum value of 180 degrees.

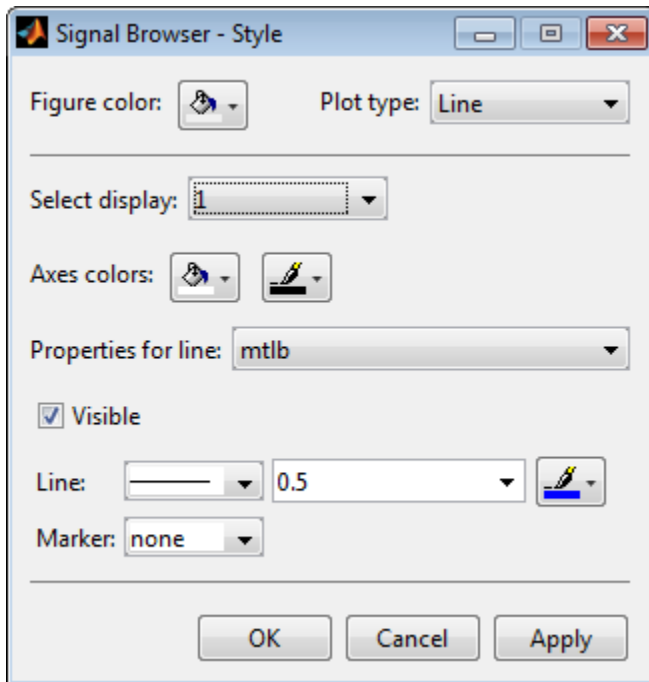
### Y-label

Specify as a string the text for the scope to display to the left of the y-axis.

This property becomes invisible when you select the **Plot signal(s) as magnitude and phase** check box. When you enable that property, the *y*-axis label always appears as **Magnitude** on the top axes and **Phase** on the bottom axes.

## Style Dialog Box

In the **Style** dialog box, you can customize the style of displays. You can change the color of the figure containing the displays, the background and foreground colors of display axes, and properties of lines in a display. From the Signal Browser menu, select **View > Style** to open this dialog box.



## Properties

The **Style** dialog box allows you to modify the following properties of the Signal Browser:

### Figure color

Specify the color that you want to apply to the background of the Signal Browser. By default, the figure color is gray.

### Plot type

Specify the type of plot to use. The default setting is **Line**. Valid values for **Plot type** are:

- **Line** — Displays input signal as lines connecting each of the sampled values. This approach is similar to the functionality of the MATLAB `line` or `plot` function.
- **Stairs** — Displays input signal as a *stairstep* graph. A stairstep graph is made up of only horizontal lines and vertical lines. Each horizontal line represents the signal value for a discrete sample period and is connected to two vertical lines. Each vertical line represents a change in values occurring at a sample. This approach is equivalent to the MATLAB `stairs` function. Stairstep graphs are useful for drawing time history graphs of digitally sampled data.

### Select display

Specify the active display as a number, where a display number corresponds to the index of the input signal. The number of a display corresponds to its column-wise placement index. The default setting is 1. Set this parameter to control which display should have its axes colors, line properties, marker properties, and visibility changed.

### Axes colors

Specify the color that you want to apply to the background of the axes for the active display.

### Properties for line

Specify the signal for which you want to modify the visibility, line properties, and marker properties.

### Visible

Specify whether the selected signal on the active display should be visible. If you clear this check box, the line disappears.

### Line

Specify the line style, line width, and line color for the selected signal on the active display.

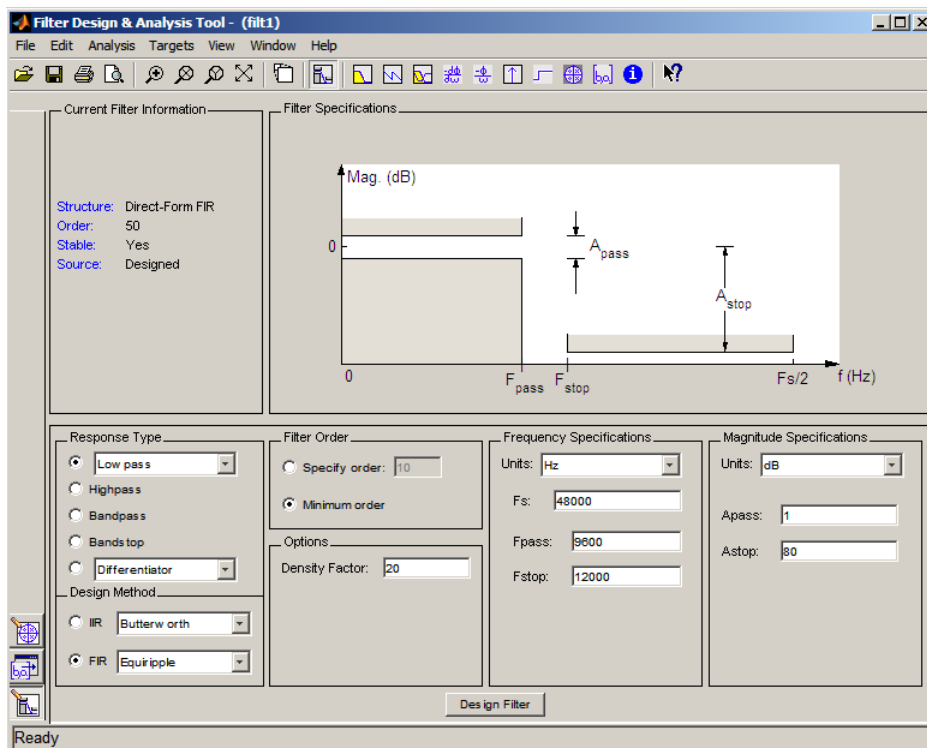
### Marker

Specify marks for the selected signal on the active display to show at data points. This parameter is similar to the **Marker** property for the MATLAB Handle Graphics plot objects. You can choose any of the marker symbols from the following table.

| Specifier | Marker Type                   |
|-----------|-------------------------------|
| none      | No marker (default)           |
| ○         | Circle                        |
| □         | Square                        |
| ×         | Cross                         |
| •         | Point                         |
| +         | Plus sign                     |
| *         | Asterisk                      |
| ◇         | Diamond                       |
| ▽         | Downward-pointing triangle    |
| △         | Upward-pointing triangle      |
| ◁         | Left-pointing triangle        |
| ▷         | Right-pointing triangle       |
| ☆         | Five-pointed star (pentagram) |
| ☆☆        | Six-pointed star (hexagram)   |


## Filter Design and Analysis Tool

The Filter Design and Analysis Tool `fdatool` allows you to design and edit FIR and IIR filters. To launch `fdatool`, press either the **New** button or the **Edit** button under the **Filters** list box in SPTool.



**Note** When you open FDATool from SPTool, a reduced version of FDATool that is compatible with SPTool opens.

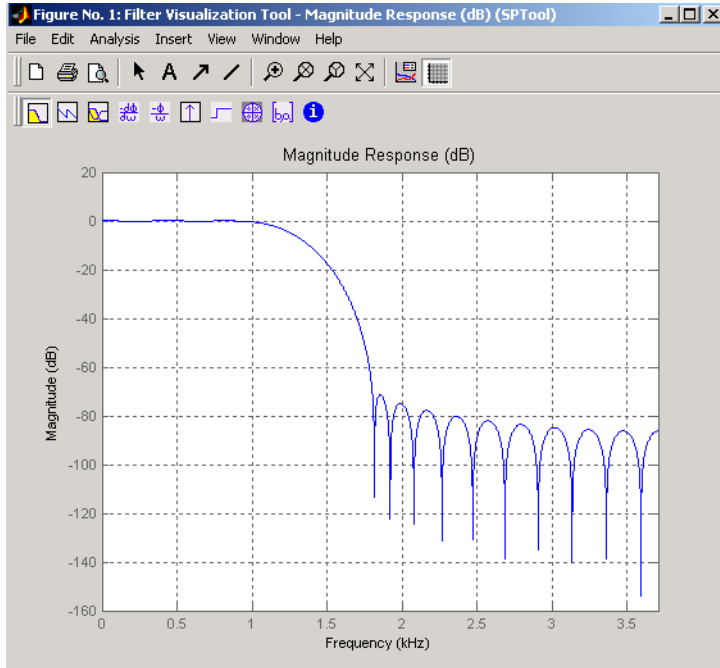
The Filter Design and Analysis Tool has a Pole/Zero Editor you can access by selecting

the  icon in the left column of FDATool.

## 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.

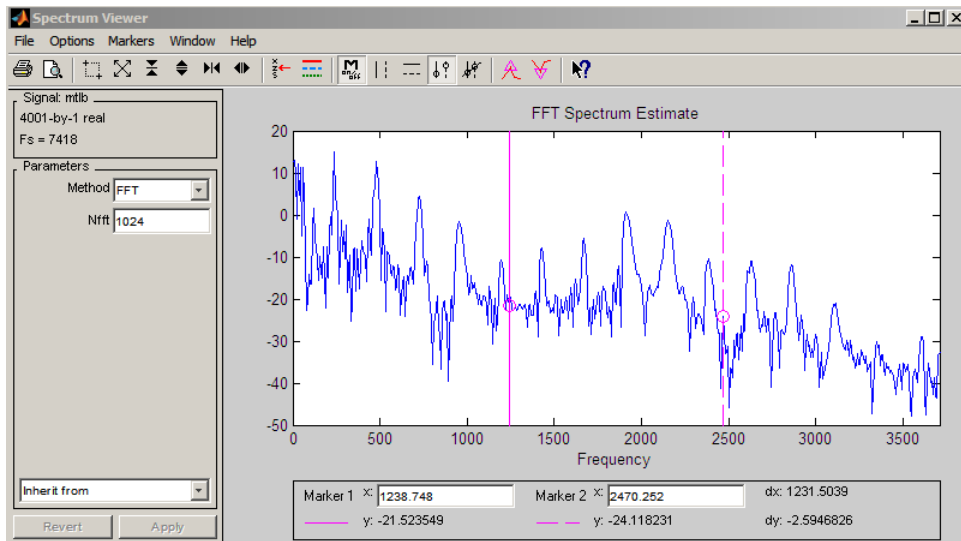


## Spectrum Viewer

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 to modify signal properties.

## See Also

fdatool | fvtool | findpeaks

## square

Square wave

### Syntax

```
x = square(t)
x = square(t,duty)
```

### Description

`x = square(t)` generates a square wave with period  $2\pi$  for the elements of time vector `t`. `square(t)` is similar to `sin(t)`, but creates a square wave with peaks of  $\pm 1$  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

## SS

Convert digital filter to state-space representation

## Syntax

```
[A,B,C,D] = ss(d)
```

## Description

`[A,B,C,D] = ss(d)` converts a digital filter, `d`, to its state-space representation.

The state-space representation of a filter is given by

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

where  $x$  is the state vector,  $u$  is the input vector, and  $y$  is the output vector.

## Examples

### State-Space Representation of a Lowpass IIR Filter

Design a lowpass IIR filter of order 6. Specify a normalized passband frequency of  $0.2\pi$  rad/sample. Compute the state-space representation of the filter.

```
d = designfilt('lowpassiir','FilterOrder',6,'PassbandFrequency',0.2);
[A,B,C,D] = ss(d)
```

A =

|        |         |        |         |        |         |
|--------|---------|--------|---------|--------|---------|
| 1.5640 | -0.9294 | 0      | 0       | 0      | 0       |
| 1.0000 | 0       | 0      | 0       | 0      | 0       |
| 0.1795 | 0.0036  | 1.6097 | -0.8112 | 0      | 0       |
| 0      | 0       | 1.0000 | 0       | 0      | 0       |
| 0.0020 | 0.0000  | 0.0408 | 0.0021  | 1.6956 | -0.7409 |

```
0 0 0 0 1.0000 0
```

B =

```
0.0913
0
0.0046
0
0.0001
0
```

C =

```
0.0020 0.0000 0.0408 0.0021 3.6956 0.2591
```

D =

```
5.2030e-05
```

## Input Arguments

### **d** — Digital filter

digitalFilter object

Digital filter, specified as a `digitalFilter` object. Use `designfilt` to generate a digital filter based on frequency-response specifications.

Example: `d = designfilt('lowpassiir','FilterOrder',3,'HalfPowerFrequency',0.5)` specifies a third-order Butterworth filter with normalized 3-dB frequency  $0.5\pi$  rad/sample.

## Output Arguments

### **A** — State matrix

matrix

---

State matrix, returned as a matrix.

Data Types: double

**B — Input-to-state matrix**

matrix

Input-to-state matrix, returned as a matrix.

Data Types: double

**C — State-to-output matrix**

matrix

State-to-output matrix, returned as a matrix.

Data Types: double

**D — Feedthrough matrix**

matrix

Feedthrough matrix, returned as a matrix.

Data Types: double

**See Also**

`designfilt` | `digitalFilter` | `tf` | `zpk`

## ss2sos

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

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

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

### Second-Order Section Form of a Filter

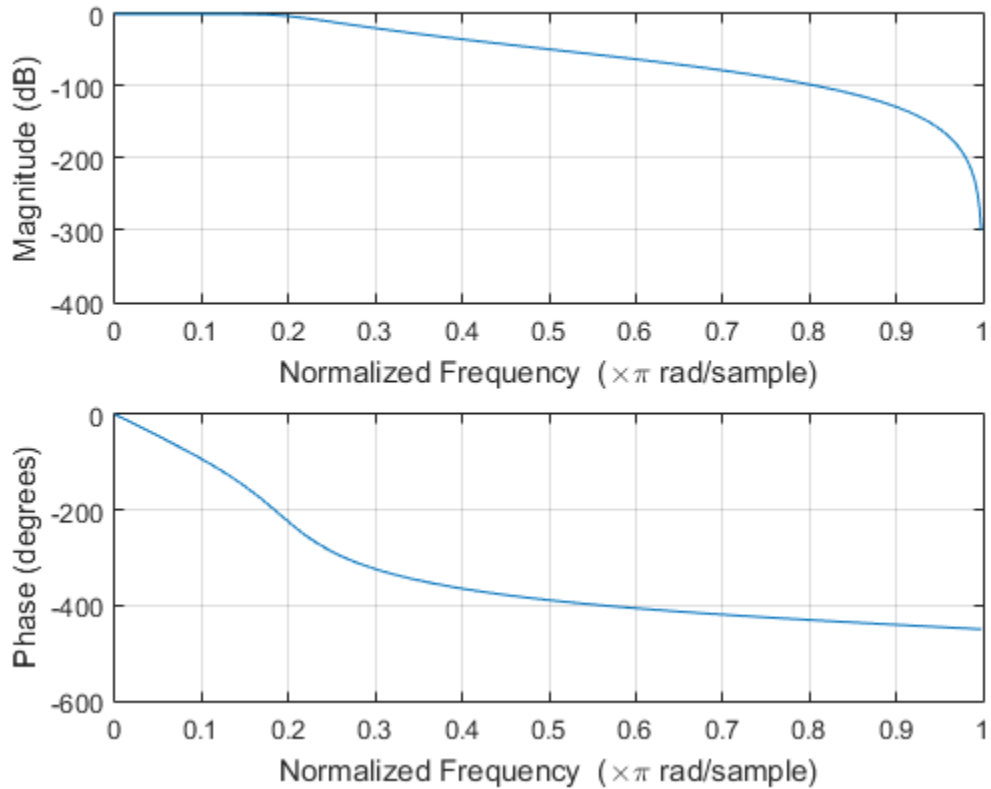
Design a 5th-order Butterworth lowpass filter using the `butter` function. Specify a cutoff frequency of  $0.2\pi$  rad/sample. Express the output in state-space form. Convert the state-space result to second-order sections. Visualize the frequency response of the filter.

```
[A,B,C,D] = butter(5,0.2);  
sos = ss2sos(A,B,C,D)  
freqz(sos)
```

```
sos =
```

```
    0.0013    0.0013         0    1.0000   -0.5095         0  
    1.0000    2.0012    1.0012    1.0000   -1.0966    0.3554  
    1.0000    1.9968    0.9968    1.0000   -1.3693    0.6926
```





## 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.

## More About

### Algorithms

`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

$$\|H\|_p = \left[ \frac{1}{2\pi} \int_0^{2\pi} |H(\omega)|^p d\omega \right]^{1/p}$$

where  $p$  can be either  $\infty$  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.

## References

- [1] Jackson, L. B. *Digital Filters and Signal Processing*. 3rd Ed. Boston: Kluwer Academic Publishers, 1996, chap. 11.
- [2] Mitra, S. K. *Digital Signal Processing: A Computer-Based Approach*. New York: McGraw-Hill, 1998, chap. 9.
- [3] Vaidyanathan, P. P. “Robust Digital Filter Structures.” *Handbook for Digital Signal Processing* (S. K. Mitra and J. F. Kaiser, eds.). New York: John Wiley & Sons, 1993, chap. 7.

## See Also

`cplxpair` | `sos2ss` | `ss2tf` | `ss2zp` | `tf2sos` | `zp2sos`

## ss2zp

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

$$\begin{aligned}\dot{x} &= Ax + Bu \\ y &= Cx + Du\end{aligned}$$

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

### Zeros, Poles, and Gain of a Discrete-Time System

Consider a discrete-time system defined by the transfer function

$$H(z) = \frac{2 + 3z^{-1}}{1 + 0.4z^{-1} + z^{-2}}.$$

Determine its zeros, poles, and gain directly from the transfer function. Pad the numerator with zeros so it has the same length as the denominator.

```
b = [2 3 0];
a = [1 0.4 1];
[z,p,k] = tf2zp(b,a)
```

z =

```
      0
-1.5000
```

p =

```
-0.2000 + 0.9798i
-0.2000 - 0.9798i
```

k =

```
2
```

Express the system in state-space form and determine the zeros, poles, and gain using `ss2zp`.

```
[A,B,C,D] = tf2ss(b,a);
[z,p,k] = ss2zp(A,B,C,D,1)
```

z =

```
      0  
-1.5000
```

p =

```
-0.2000 + 0.9798i  
-0.2000 - 0.9798i
```

k =

```
2
```

## More About

### Algorithms

`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*. Vol. 14, 1978, p. 557.

### See Also

`sos2zp` | `ss2sos` | `ss2tf` | `tf2zp` | `tf2zpk` | `zp2ss`

## statelevels

State-level estimation for bilevel waveform with histogram method

### Syntax

```
LEVELS = statelevels(X)  
LEVELS = statelevels(X,NBINS)  
LEVELS = statelevels(X,NBINS,METHOD)  
[LEVELS,HISTOGRAM] = statelevels(...)  
[LEVELS,HISTOGRAM,BINLEVELS] = statelevels(...)  
statelevels(...)
```

### Description

`LEVELS = statelevels(X)` estimates the low- and high-state levels in the bilevel waveform, `X`, using the histogram method. See “Algorithms” on page 1-1621.

`LEVELS = statelevels(X,NBINS)` specifies the number of bins to use in the histogram as a positive scalar. If unspecified, `NBINS` defaults to 100.

`LEVELS = statelevels(X,NBINS,METHOD)` estimates state levels using the mean or mode of the subhistograms. Valid entries for `METHOD` are 'mean' or 'mode'. `METHOD` defaults to 'mode'. See “Algorithms” on page 1-1621.

`[LEVELS,HISTOGRAM] = statelevels(...)` returns the histogram, `HISTOGRAM`, of the values in `X`.

`[LEVELS,HISTOGRAM,BINLEVELS] = statelevels(...)` returns the centers of the histogram bins.

`statelevels(...)` displays a plot of the signal and the corresponding computed histogram.

## Input Arguments

### **X**

Bilevel waveform. X is a real-valued row or column vector.

### **NBINS**

Number of histogram bins

**Default:** 100

### **METHOD**

State-level estimation method in the subhistograms. METHOD is a string indicating the statistic to use for the estimation of the low- and high-state levels. Valid entries for METHOD are 'mode' or 'mean'. See “Algorithms” on page 1-1621.

**Default:** 'mode'

## Output Arguments

### **LEVELS**

Levels of low and high states. LEVELS is a 1-by-2 row vector of state levels estimated by the histogram method. The first element of LEVELS is the low-state level. The second element of LEVELS is the high-state level.

### **HISTOGRAM**

Histogram counts (frequencies). HISTOGRAM is a column vector with NBINS elements containing the number of data values in each histogram bin.

### **BINLEVELS**

Histogram bin centers. BINLEVELS is a column vector containing the bin centers for the histogram counts in HISTOGRAM.



## Examples

### Display State Levels and Subhistograms

Estimate the low- and high-state levels of 2.3 V underdamped clock data. Plot the data with the estimated state levels and subhistograms.

```
load('clockex.mat', 'x');
statelevels(x);
```

### State Levels with 100 Bins and Modes of Subhistograms

Estimate the low and high-state levels of 2.3 V underdamped clock data sampled at 4 MHz.

Use the default number of bins and modes of the subhistograms to estimate the state levels. Plot the clock data with the lines indicating the estimated low and high-state levels.

```
load('clockex.mat', 'x', 't');
LEVELS = statelevels(x);
plot(t,x);
hold on;
plot(t,LEVELS(1).*ones(length(x)), 'r--');
plot(t,LEVELS(2).*ones(length(x)), 'r--');
```

### State Levels Using Means of Subhistograms

Estimate the low and high-state levels of 2.3 V underdamped clock data sampled at 4 MHz.

Use the default number of bins and means of the subhistograms to estimate the state levels. Plot the clock data with the lines indicating the estimated low and high-state levels.

```
load('clockex.mat', 'x', 't');
LEVELS = statelevels(x,1e3, 'mean');
```

### Histogram Counts and Histogram Bin Centers

Estimate the low- and high-state levels of 2.3 V underdamped clock data sampled at 4 MHz. Return the histogram counts and histogram bin centers used in the histogram method.

```
load('clockex.mat', 'x', 't');  
[LEVELS,HISTOGRAM,BINLEVELS] = statelevels(x);
```

## More About

### State

A particular level, which can be associated with an upper- and lower-state boundary. States are ordered from the most negative to the most positive. In a bilevel waveform, the most negative state is the low state. The most positive state is the high state.

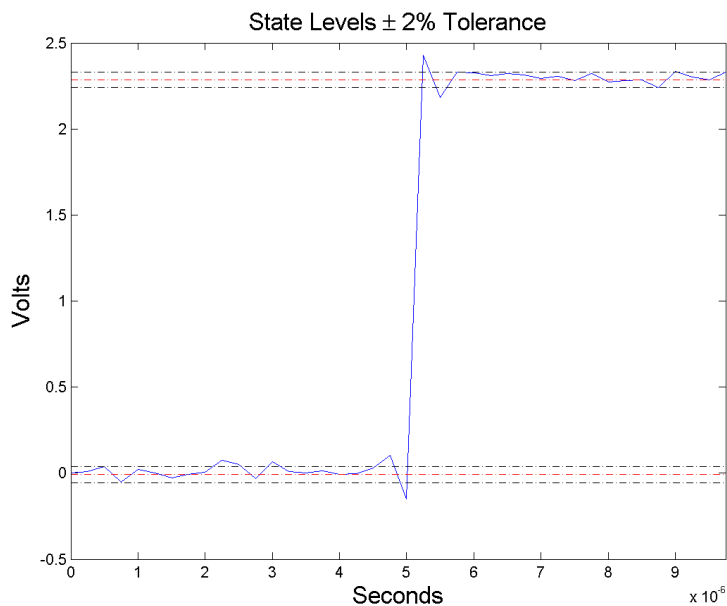
### State-Level Tolerances

Each state level can have associated lower- and upper-state boundaries. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference between the high state and low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the  $\alpha\%$  tolerance region for the low state is defined as

$$S_1 \pm \frac{\alpha}{100}(S_2 - S_1)$$

where  $S_1$  is the low-state level and  $S_2$  is the high-state level. Replace the first term in the equation with  $S_2$  to obtain the  $\alpha\%$  tolerance region for the high state.

The following figure illustrates lower and upper 2% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The red dashed lines indicate the estimated state levels.



## Algorithms

`statelevels` uses the histogram method to estimate the states of a bilevel waveform. The histogram method is described in [1]. To summarize the method:

- 1 Determine the maximum and minimum amplitudes and amplitude range of the data.
- 2 For the specified number of histogram bins, determine the bin width as the ratio of the amplitude range to the number of bins.
- 3 Sort the data values into the histogram bins.
- 4 Identify the lowest-indexed histogram bin,  $i_{low}$ , and highest-indexed histogram bin,  $i_{high}$ , with nonzero counts.
- 5 Divide the histogram into two subhistograms.

The indices of the lower histogram bins are  $i_{low} \leq i \leq 1/2(i_{high} - i_{low})$ .

The indices of the upper histogram bins are  $i_{low} + 1/2(i_{high} - i_{low}) \leq i \leq i_{high}$ .

- 6 Compute the state levels by determining the mode or mean of the lower and upper histograms.

## References

- [1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003, pp. 15–17.

## See Also

midcross | overshoot | risetime | undershoot

## stepz

Step response of digital filter

### Syntax

```
[h,t] = stepz(b,a)
[h,t] = stepz(sos)
[h,t] = stepz(d)
[h,t] = stepz(...,n)
[h,t] = stepz(...,n,fs)
stepz(...)
```

### Description

`[h,t] = stepz(b,a)` returns 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  $n = \text{length}(\mathbf{t})$  is computed automatically).

`[h,t] = stepz(sos)` returns the step response of the second order sections matrix, **sos**. **sos** is a  $K$ -by-6 matrix, where the number of sections,  $K$ , must be greater than or equal to 2. If the number of sections is less than 2, `stepz` considers the input to be the numerator vector, **b**. Each row of **sos** corresponds to the coefficients of a second-order (biquad) filter. The  $i$ th row of the **sos** matrix corresponds to `[bi(1) bi(2) bi(3) ai(1) ai(2) ai(3)]`.

`[h,t] = stepz(d)` returns the step response of the digital filter, **d**. Use `designfilt` to generate **d** based on frequency-response specifications.

`[h,t] = stepz(...,n)` computes the first  $n$  samples of the step response when  $n$  is an integer (**t** = `[0:n-1]'`). If  $n$  is a vector of integers, the step response is computed only at those integer values with 0 denoting the time origin.

`[h,t] = stepz(...,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(...)` with no output arguments plots the step response of the filter. If you input the filter coefficients or second order sections matrix, the current figure window is used. If you input a `digitalFilter`, the step response is displayed in `fvtool`.

---

**Note:** If the input to `stepz` is single precision, the step response is calculated using single-precision arithmetic. The output, `h`, is single precision.

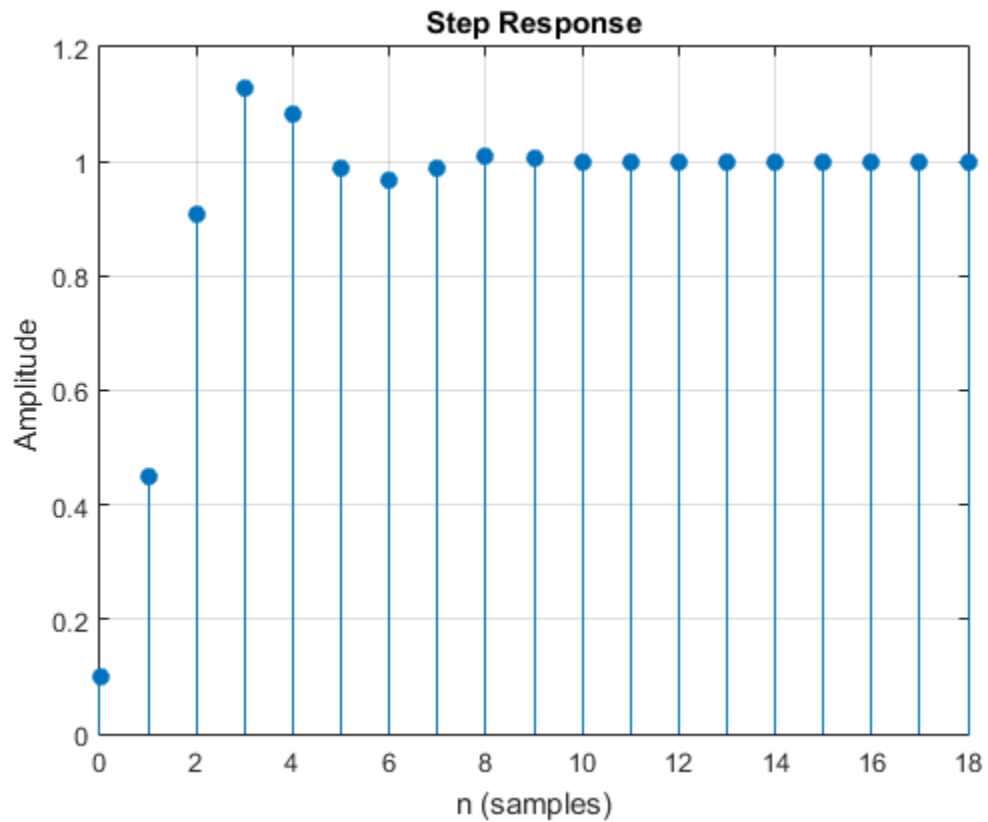
---

## Examples

### Step Response of a Butterworth Filter

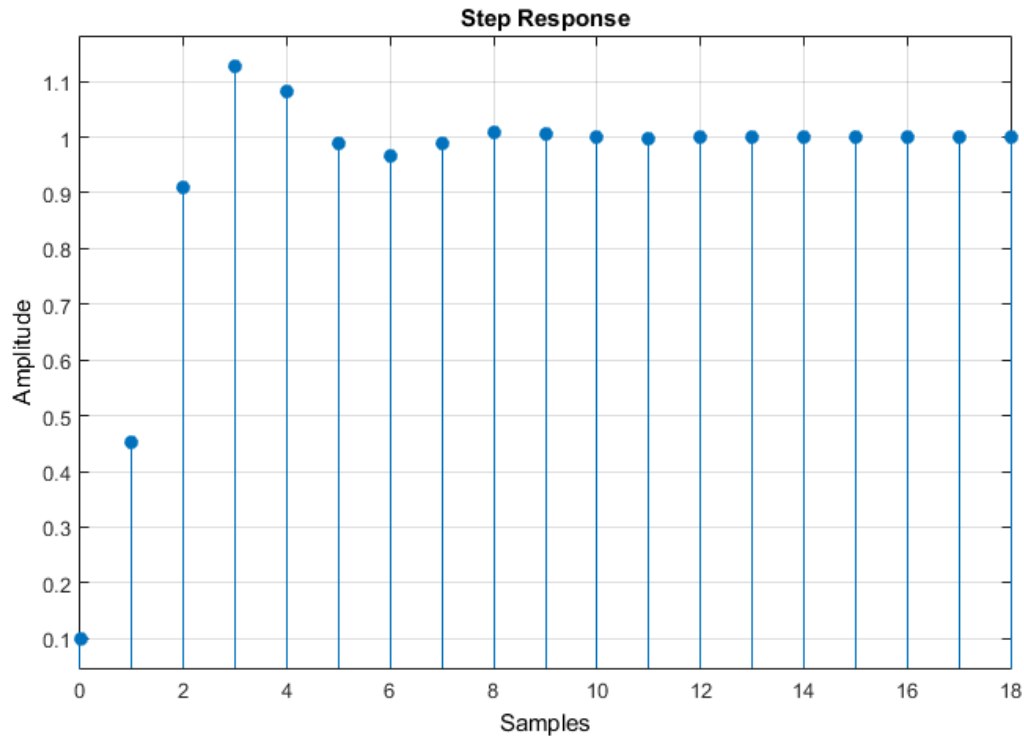
Create a third-order Butterworth filter with normalized half-power frequency  $0.4\pi$  rad/sample. Display its step response.

```
[b,a] = butter(3,0.4);  
stepz(b,a)  
grid
```



Create an identical filter using `designfilt` and display its step response using `fvttool`.

```
d = designfilt('lowpassiir', 'FilterOrder', 3, 'HalfPowerFrequency', 0.4);  
stepz(d)
```

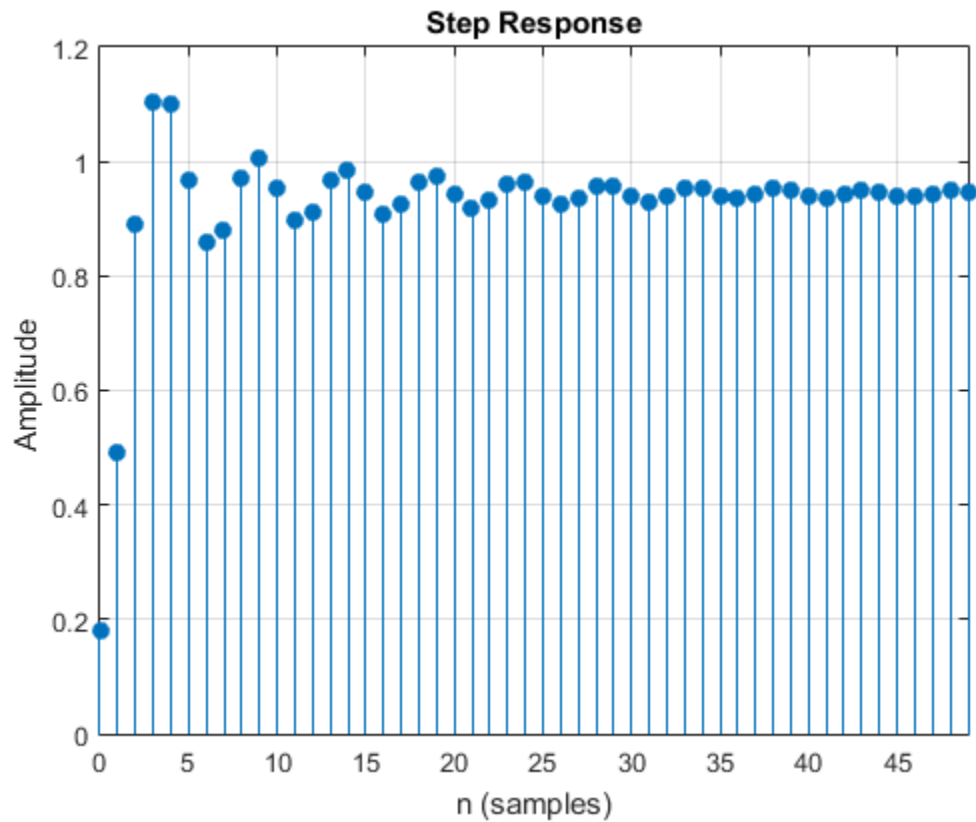


### Step Response of an Elliptic Filter

Design a fourth-order lowpass elliptic filter with normalized passband frequency  $0.4\pi$  rad/sample. Specify a passband ripple of 0.5 dB and a stopband attenuation of 20 dB. Plot the first 50 samples of the filter's step response.

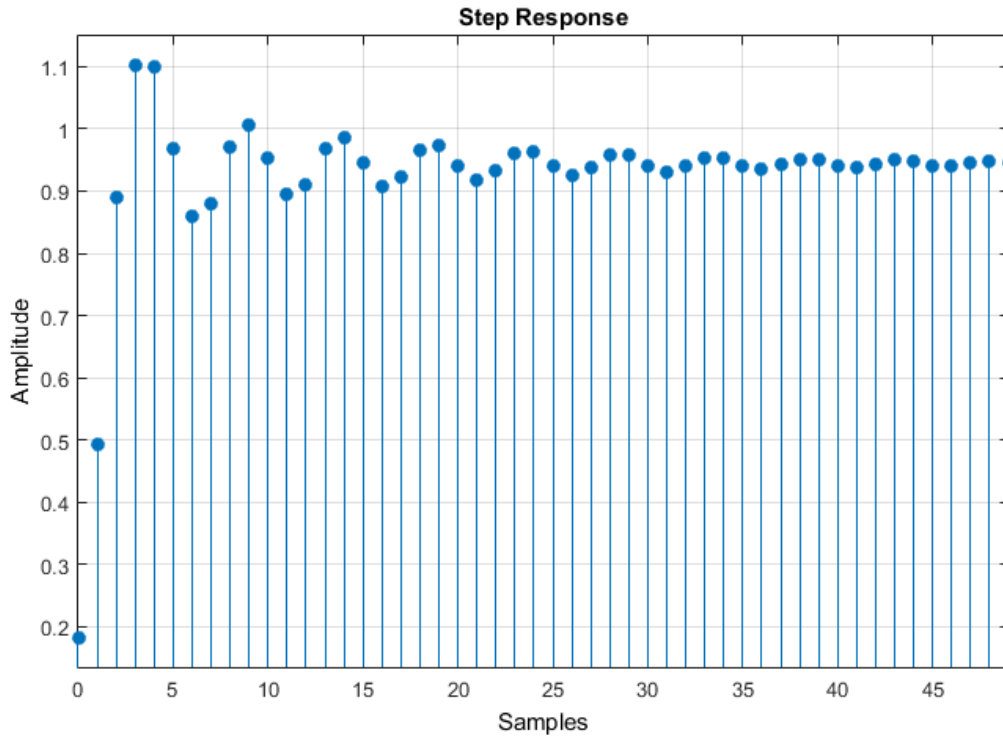
```
[b,a] = ellip(4,0.5,20,0.4);  
stepz(b,a,50)  
grid
```





Create the same filter using `designfilt` and display its step response using `fvtool`.

```
d = designfilt('lowpassiir', 'FilterOrder',4, 'PassbandFrequency',0.4, ...  
              'PassbandRipple',0.5, 'StopbandAttenuation',20, ...  
              'DesignMethod', 'ellip');  
stepz(d,50)
```



## More About

### Algorithms

`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  $5 \times 10^{-5}$  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 pole of largest nonunit amplitude is  $5 \times 10^{-5}$  times 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**

`designfilt` | `digitalFilter` | `freqz` | `grpdelay` | `impz` | `phasez` | `zplane`

## stmcb

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  $z$ .

$$H(z) = \frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \dots + b(nb+1)z^{-nb}}{a(1) + a(2)z^{-1} + \dots + a(na+1)z^{-na}}$$

## Examples

### Steiglitz-McBride Approximation of a Filter

Approximate the impulse response of an IIR filter with a system of lower order.

Specify a 6th-order Butterworth filter with normalized 3-dB frequency  $0.2\pi$  rad/sample.

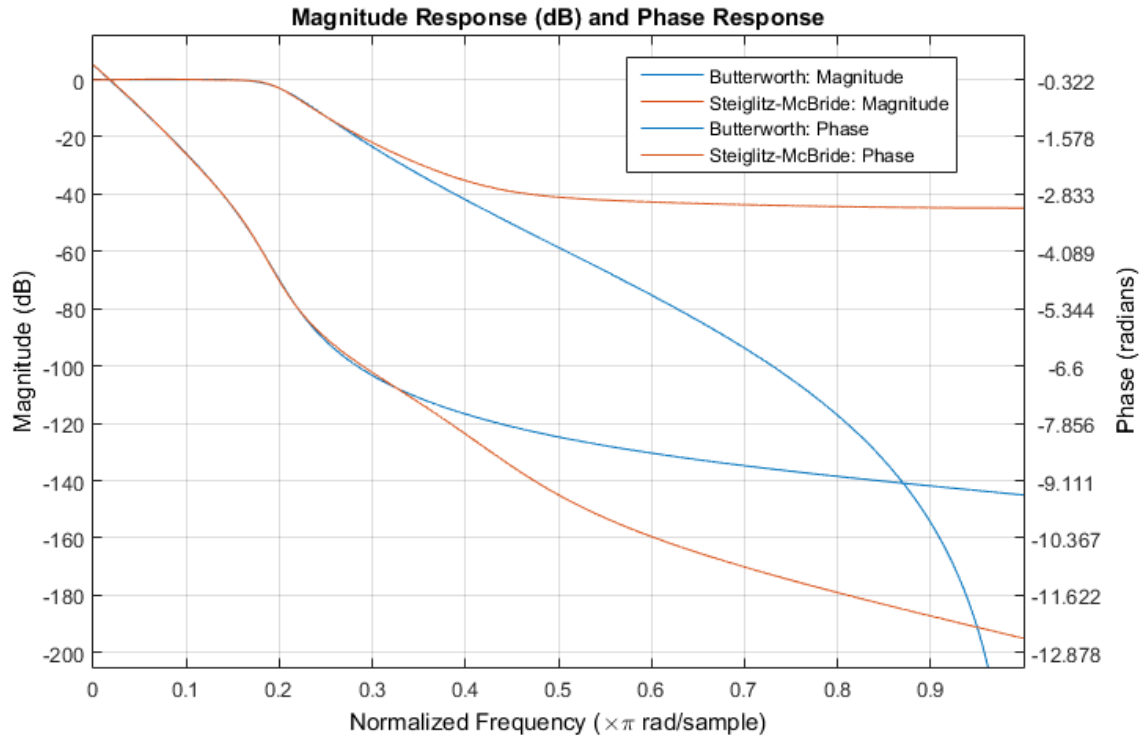
```
d = designfilt('lowpassiir', 'FilterOrder', 6, ...
    'HalfPowerFrequency', 0.2, 'DesignMethod', 'butter');
```

Use the Steiglitz-McBride iteration to approximate the filter with a 4th-order system.

```
h = impz(d);
[bb,aa] = stmcb(h,4,4);
```

Plot the frequency responses of the two systems.

```
hfvt = fvtool(d,bb,aa, 'Analysis', 'freq');
legend(hfvt, 'Butterworth', 'Steiglitz-McBride')
```



## 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.
```

## More About

### Algorithms

`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.

## References

- [1] Steiglitz, K., and L. E. McBride. "A Technique for the Identification of Linear Systems." *IEEE Transactions on Automatic Control*. Vol. AC-10, 1965, pp.461–464.
- [2] Ljung, Lennart. *System Identification: Theory for the User*. 2nd Edition. Upper Saddle River, NJ: Prentice Hall, 1999, p.354.

## See Also

levinson | lpc | aryule | prony

## strips

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.

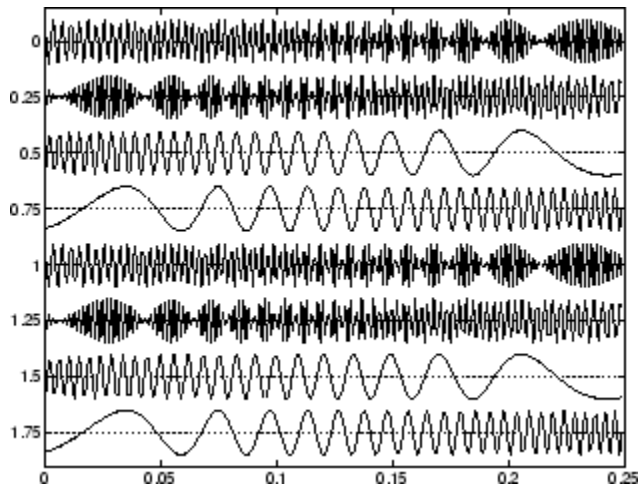
`strips` ignores the imaginary part of complex-valued `x`.

### 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

# taylorwin

Taylor window

## Syntax

```
w = taylorwin(n)
w = taylorwin(n,nbar)
w = taylorwin(n,nbar,sll)
```

## Description

Taylor windows are similar to Chebyshev windows. Whereas 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 the 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 aperture 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.

`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. Its default value is 4.

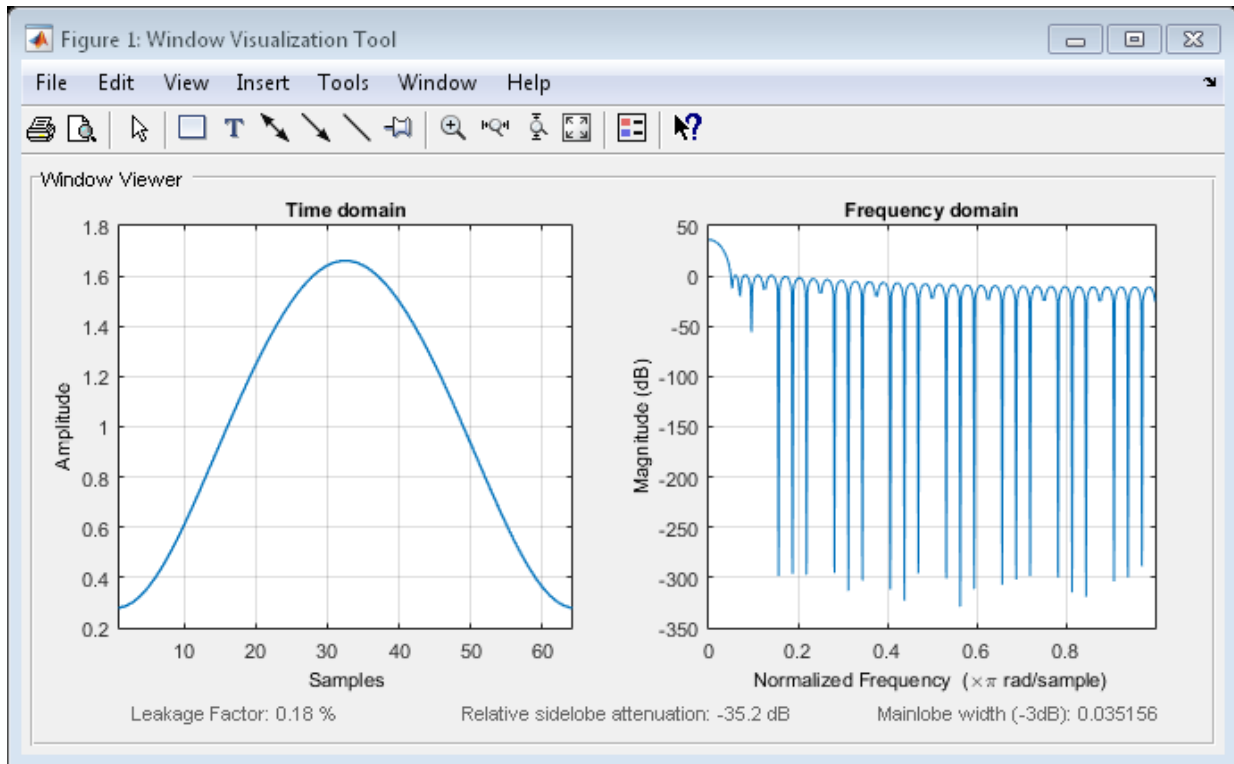
`w = taylorwin(n,nbar,sll)` returns an  $n$ -point Taylor window with a maximum sidelobe level of `sll` dB relative to the mainlobe peak. `sll` must be negative. Its default value is  $-30$ , which produces sidelobes with peaks 30 dB down from the mainlobe peak.

## Examples

### Taylor Window

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. Visualize the result with `wvtool`.

```
w = taylorwin(64,4,-35);
wvtool(w)
```



## References

- [1] Carrara, Walter G., Ronald M. Majewski, and Ron S. Goodman. *Spotlight Synthetic Aperture Radar: Signal Processing Algorithms*. Boston: Artech House, 1995, Appendix D.2.
- [2] Brookner, Eli. *Practical Phased Array Antenna Systems*. Boston: Artech House, 1991.

## tf

Convert digital filter to transfer function

### Syntax

```
[num,den] = tf(d)
```

### Description

`[num,den] = tf(d)` converts a digital filter, `d`, to numerator and denominator vectors.

### Examples

#### Highpass Filter Transfer Function

Design a highpass FIR filter of order 8 with passband frequency 75 kHz and passband ripple 0.2 dB. Specify a sample rate of 200 kHz. Compute the coefficients of the equivalent transfer function.

```
hpFilt = designfilt('highpassfir','FilterOrder',8, ...  
                  'PassbandFrequency',75e3,'PassbandRipple',0.2, ...  
                  'SampleRate',200e3);
```

```
[b,a] = tf(hpFilt)
```

```
b =  
    1.0e-03 *  
    0.0128    -0.1024    0.3583    -0.7166    0.8958    -0.7166    0.3583    -0.1024    0.0128  
a =  
    1.0000    5.7637    15.4847    25.1134    26.7644    19.1396    8.9554    2.5058    0.0000
```

### Input Arguments

#### **d** — Digital filter

digitalFilter object

Digital filter, specified as a `digitalFilter` object. Use `designfilt` to generate a digital filter based on frequency-response specifications.

Example: `d = designfilt('lowpassiir','FilterOrder',3,'HalfPowerFrequency',0.5)` specifies a third-order Butterworth filter with normalized 3-dB frequency  $0.5\pi$  rad/sample.

## Output Arguments

### **num** — Numerator coefficients

row vector

Numerator coefficients, returned as a row vector.

Data Types: `double`

### **den** — Denominator coefficients

row vector

Denominator coefficients, returned as a row vector.

Data Types: `double`

## See Also

`designfilt` | `digitalFilter` | `ss` | `zpk`

## tf2latc

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 | tf2sos | tf2ss | tf2zp | tf2zpk | latcfilt

## tf2sos

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.

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_1 + b_2z^{-1} + \dots + b_{n+1}z^{-n}}{a_1 + a_2z^{-1} + \dots + a_{m+1}z^{-m}}$$

sos is an  $L$ -by-6 matrix

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

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.

---



## Examples

### Second-Order Section Implementation of a Butterworth Filter

Design a Butterworth 4th-order lowpass filter using the function `butter`. Specify the cutoff frequency as half the Nyquist frequency. Implement the filter as second-order sections. Verify that the two representations are identical by comparing their numerators and denominators.

```
[nm,dn] = butter(4,0.5);
[ss,gn] = tf2sos(nm,dn);
numers = [conv(ss(1,1:3),ss(2,1:3))*gn;nm]
denoms = [conv(ss(1,4:6),ss(2,4:6)) ;dn]
```

```
numers =
```

```
    0.0940    0.3759    0.5639    0.3759    0.0940
    0.0940    0.3759    0.5639    0.3759    0.0940
```

```
denoms =
```

```
    1.0000   -0.0000    0.4860   -0.0000    0.0177
    1.0000   -0.0000    0.4860   -0.0000    0.0177
```

## More About

### Algorithms

`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

$$\|H\|_p = \left[ \frac{1}{2\pi} \int_0^{2\pi} |H(\omega)|^p d\omega \right]^{1/p}$$

where  $p$  can be either  $\infty$  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. Boston: Kluwer Academic Publishers, 1996, chap. 11.
- [2] Mitra, S. K. *Digital Signal Processing: A Computer-Based Approach*. New York: McGraw-Hill, 1998, chap. 9.
- [3] Vaidyanathan, P. P. "Robust Digital Filter Structures." *Handbook for Digital Signal Processing* (S. K. Mitra and J. F. Kaiser, eds.). New York: John Wiley & Sons, 1993, chap. 7.

## See Also

`cplxpair` | `sos2tf` | `ss2sos` | `tf2ss` | `tf2zp` | `tf2zpk` | `zp2sos`

## tf2ss

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

$$H(s) = \frac{B(s)}{A(s)} = \frac{b_1 s^{n-1} + \dots + b_{n-1} s + b_n}{a_1 s^{m-1} + \dots + a_{m-1} s + a_m} = C(sI - A)^{-1} B + D$$

in controller canonical form

$$\begin{aligned} \dot{x} &= Ax + Bu \\ y &= Cx + Du \end{aligned}$$

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.

**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.

---

## Examples

### State-Space Representation of Transfer Function

Consider the system described by the transfer function

$$H(s) = \frac{\begin{bmatrix} 2s + 3 \\ s^2 + 2s + 1 \end{bmatrix}}{s^2 + 0.4s + 1}.$$

Convert it to state-space form using `tf2ss`.

```
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

**See Also**

sos2ss | ss2tf | tf2sos | tf2zp | tf2zpk | zp2ss

## tf2zp

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

$$H(s) = \frac{B(s)}{A(s)} = \frac{b_1 s^{n-1} + \dots + b_{n-1} s + b_n}{a_1 s^{m-1} + \dots + a_{m-1} s + a_m}$$

you can use the output of `tf2zp` to produce the single-input, multi-output (SIMO) factored transfer function form

$$H(s) = \frac{Z(s)}{P(s)} = k \frac{(s - z_1)(s - z_2) \dots (s - z_m)}{(s - p_1)(s - p_2) \dots (s - p_n)}$$

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  $i$ th row of the matrix **b** represents the coefficients of the  $i$ th numerator polynomial (the  $i$ th 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

### Zeros, Poles, and Gain of a Continuous-Time System

Generate a system with the following transfer function.

$$H(s) = \frac{2s^2 + 3s}{s^2 + \frac{1}{\sqrt{2}}s + \frac{1}{4}} = \frac{2(s - 0)(s - (-\frac{3}{2}))}{(s - \frac{-1}{2\sqrt{2}}(1 - j))(s - \frac{-1}{2\sqrt{2}}(1 + j))}$$

Find the zeros, poles, and gain of the system. Use `eqtflength` to ensure the numerator and denominator have the same length. Plot the poles and zeros to verify that they are in the expected locations.

```
b = [2 3];
a = [1 1/sqrt(2) 1/4];
fvtool(b,a, 'polezero')
```

```
[b,a] = eqtflength(b,a);  
[z,p,k] = tf2zp(b,a)  
text(real(z)+.1,imag(z), 'Zero')  
text(real(p)+.1,imag(p), 'Pole')
```

z =

```
      0  
-1.5000
```

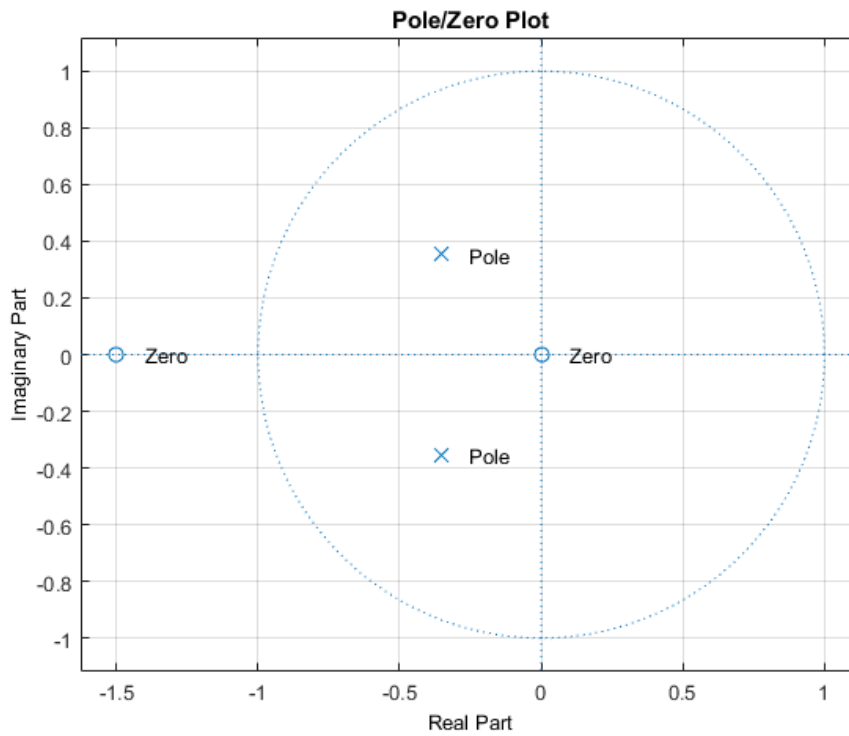
p =

```
-0.3536 + 0.3536i  
-0.3536 - 0.3536i
```

k =

```
2
```





### See Also

[sos2zp](#) | [ss2zp](#) | [tf2sos](#) | [tf2ss](#) | [tf2zpk](#) | [zp2tf](#)

## tf2zpk

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

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_1 + b_2z^{-1} \dots + b_{n-1}z^{-n} + b_nz^{-n-1}}{a_1 + a_2z^{-1} \dots + a_{m-1}z^{-m} + a_mz^{-m-1}}$$

you can use the output of `tf2zpk` to produce the single-input, multioutput (SIMO) factored transfer function form

$$H(z) = \frac{Z(z)}{P(z)} = k \frac{(z - z_1)(z - z_2) \dots (z - z_m)}{(z - p_1)(z - p_2) \dots (z - p_n)}$$

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  $i$ th row of the matrix **b** represents the coefficients of the  $i$ th numerator polynomial (the  $i$ th 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

### Poles, Zeros, and Gain of an IIR Filter

Design a 3rd-order Butterworth filter with normalized cutoff frequency  $0.4\pi$  rad/sample. Find the poles, zeros, and gain of the filter. Plot them to verify that they are where expected.

```
[b,a] = butter(3,.4);
fvtool(b,a,'polezero')
[z,p,k] = tf2zpk(b,a)
text(real(z)-0.1,imag(z)-0.1,'\bfZeros','color',[0 0.4 0])
text(real(p)-0.1,imag(p)-0.1,'\bfPoles','color',[0.6 0 0])
```

`z =`

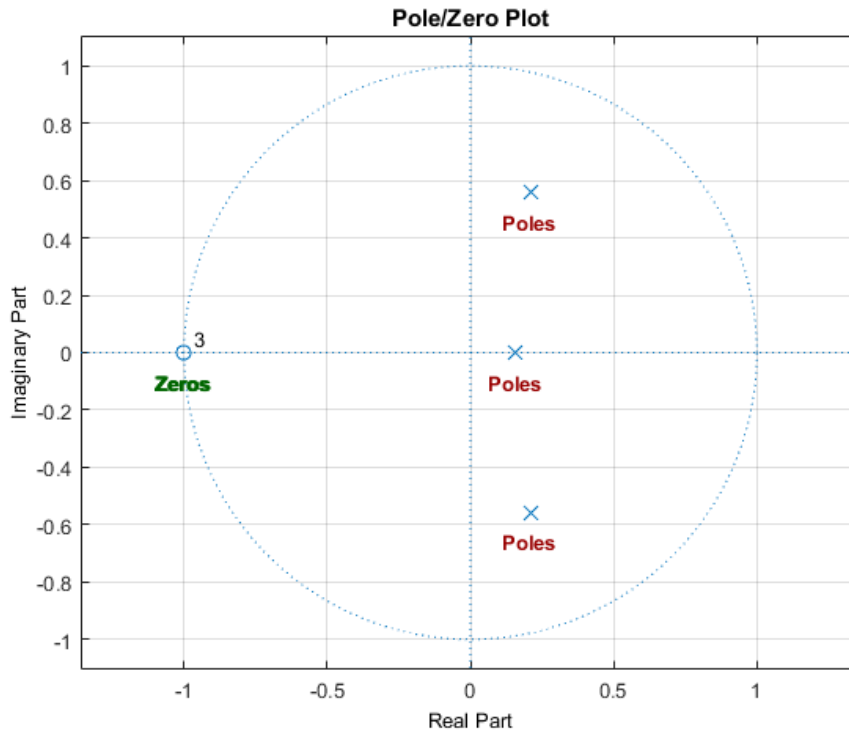
```
-1.0000 + 0.0000i
-1.0000 + 0.0000i
-1.0000 - 0.0000i
```

`p =`

```
0.2094 + 0.5582i
0.2094 - 0.5582i
0.1584 + 0.0000i
```

k =

0.0985



### See Also

[sos2zp](#) | [ss2zp](#) | [tf2sos](#) | [tf2ss](#) | [tf2zp](#) | [zp2tf](#)

# tfestimate

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,...,'twosided')
tfestimate(...)
```

## Description

`Txy = tfestimate(x,y)` finds a transfer function estimate, `Txy`, given an input signal, `x`, and an output signal, `y`.

The signals may be either vectors or two-dimensional matrices. If both are vectors, they must have the same length. If both are matrices, they must have the same size, and `tfestimate` operates columnwise: `Txy(:,n) = tfestimate(x(:,n),y(:,n))`. If one is a matrix and the other is a vector, then the vector is converted to a column vector and internally expanded so both inputs have the same number of columns.

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   |
|-------------------|---|---|
| <code>nfft</code> | FFT length which determines the frequencies at which the PSD is estimated | Maximum of 256 or the next power of 2 greater than the length of each section of <code>x</code> or <code>y</code> |

| Parameter             | Description  | Default Value  |
|-----------------------|--|--|
|                       | 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 $nfft$ .<br><br>If $nfft$ is greater than the signal length, the data is zero-padded. If $nfft$ is less than the signal length, the data segment is wrapped so that the length is equal to $nfft$ . |  |
| <code>fs</code>       | Sampling frequency   | 1  |
| <code>window</code>   | 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. |
| <code>noverlap</code> | 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`, then  $Txy$  uses a Hamming window of that length.

`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 (in rad/sample) at which the `tfestimate` 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  $[0, 2\pi)$ .

`[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,...,'twosided')` returns a transfer function estimate with frequencies that range over the entire interval from 0 to the sampling frequency,  $[0, fs)$ . Specifying `'onesided'` uses from 0 to the Nyquist frequency.

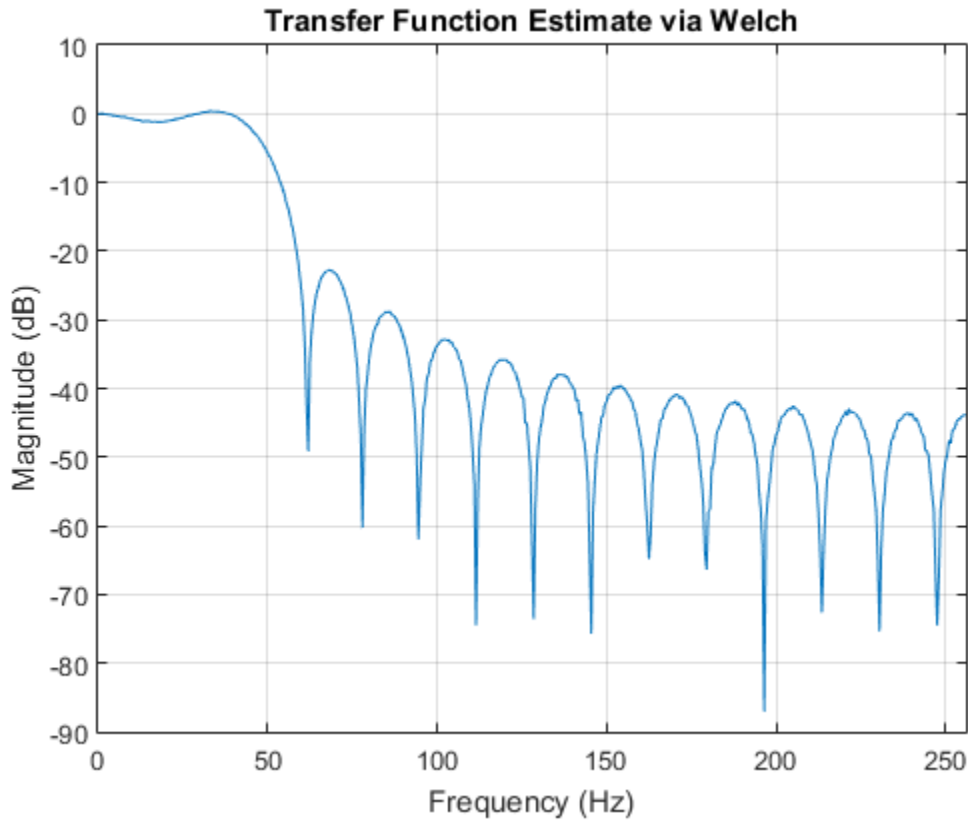
`tfestimate(...)` with no output arguments plots the transfer function estimate in the current figure window.

## Examples

### Transfer Function Between Two Sequences

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)
```



## More About

### Transfer Function

The relationship between the input  $x$  and output  $y$  is modeled by the linear, time-invariant transfer function  $T_{xy}$ . The *transfer function* is the quotient of  $P_{yx}$ , the cross power spectral density of  $x$  and  $y$ , and  $P_{xx}$ , the power spectral density of  $x$ :

$$T_{xy}(f) = \frac{P_{yx}(f)}{P_{xx}(f)}.$$



**Algorithms**

tfestimate uses Welch's averaged periodogram method. See pwelch for details.

**See Also**

cpsd | mscohere | pwelch | periodogram | spectrum

## thd

Total harmonic distortion

### Syntax

`r = thd(x)`

`r = thd(x, fs, n)`

`r = thd(pxx, f, 'psd')`

`r = thd(pxx, f, n, 'psd')`

`r = thd(sxx, f, rbw, 'power')`

`r = thd(sxx, f, rbw, n, 'power')`

`[r, harmpow, harmfreq] = thd( ___ )`

`thd( ___ )`

### Description

`r = thd(x)` returns the total harmonic distortion (THD) in dBc of the real-valued sinusoidal signal `x`. The total harmonic distortion is determined from the fundamental frequency and the first five harmonics using a modified periodogram of the same length as the input signal. The modified periodogram uses a Kaiser window with  $\beta = 38$ .

`r = thd(x, fs, n)` specifies the sampling rate `fs` and the number of harmonics (including the fundamental) to use in the THD calculation.

`r = thd(pxx, f, 'psd')` specifies the input `pxx` as a one-sided power spectral density (PSD) estimate. `f` is a vector of frequencies corresponding to the PSD estimates in `pxx`.

`r = thd(pxx, f, n, 'psd')` specifies the number of harmonics (including the fundamental) to use in the THD calculation.

`r = thd(sxx, f, rbw, 'power')` specifies the input as a one-sided power spectrum. `rbw` is the resolution bandwidth over which each power estimate is integrated.

`r = thd(sxx, f, rbw, n, 'power')` specifies the number of harmonics (including the fundamental) to use in the THD calculation.

`[r, harmpow, harmfreq] = thd( ___ )` returns the powers and frequencies of the harmonics (including the fundamental).

`thd( ___ )` with no output arguments plots the spectrum of the signal and annotates the harmonics in the current figure window. It uses different colors to draw the fundamental component, the harmonics, and the DC level and noise. The THD appears above the plot. The fundamental and harmonics are labeled. The DC term is excluded from the measurement and is not labeled.

## Examples

### Determine THD for a Signal with Two Harmonics

This example shows explicitly how to calculate the total harmonic distortion in dBc for a signal consisting of the fundamental and two harmonics. The explicit calculation is checked against the result returned by `thd`.

Create a signal sampled at 1 kHz. The signal consists of a 100 Hz fundamental with amplitude 2 and two harmonics at 200 and 300 Hz with amplitudes 0.01 and 0.005. Obtain the total harmonic distortion explicitly and using `thd`.

```
t = 0:0.001:1-0.001;
x = 2*cos(2*pi*100*t)+0.01*cos(2*pi*200*t)+0.005*cos(2*pi*300*t);
tharmdist = 10*log10((0.01^2+0.005^2)/2^2)
r = thd(x)
```

```
tharmdist =
    -45.0515
```

```
r =
    -45.0515
```

### Specify Number of Harmonics

Create a signal sampled at 1 kHz. The signal consists of a 100 Hz fundamental with amplitude 2 and three harmonics at 200, 300, and 400 Hz with amplitudes 0.01, 0.005, and 0.0025.

Set the number of harmonics to 3. This includes the fundamental. Accordingly, the power at 100, 200, and 300 Hz is used in the THD calculation.

```
t = 0:0.001:1-0.001;  
x = 2*cos(2*pi*100*t)+0.01*cos(2*pi*200*t)+ ...  
    0.005*cos(2*pi*300*t)+0.0025*sin(2*pi*400*t);  
r = thd(x,1000,3)
```

```
r =  
  
-45.0515
```

Specifying the number of harmonics equal to 3 ignores the power at 400 Hz in the THD calculation.

### **Specify Number of Harmonics (PSD Input)**

Create a signal sampled at 1 kHz. The signal consists of a 100 Hz fundamental with amplitude 2 and three harmonics at 200, 300, and 400 Hz with amplitudes 0.01, 0.005, and 0.0025.

Obtain the periodogram PSD estimate of the signal and use the PSD estimate as the input to thd. Set the number of harmonics to 3. This includes the fundamental. Accordingly, the power at 100, 200, and 300 Hz is used in the THD calculation.

```
t = 0:0.001:1-0.001;  
fs = 1000;  
x = 2*cos(2*pi*100*t)+0.01*cos(2*pi*200*t)+ ...  
    0.005*cos(2*pi*300*t)+0.0025*sin(2*pi*400*t);  
[pxx,f] = periodogram(x,rectwin(length(x)),length(x),fs);  
r = thd(pxx,f,3,'psd')
```

```
r =  
  
-45.0515
```

### **THD from Power Spectrum**

Determine the THD by inputting the power spectrum obtained with a Hamming window and the resolution bandwidth of the window.

Create a signal sampled at 10 kHz. The signal consists of a 100 Hz fundamental with amplitude 2 and three odd-numbered harmonics at 300, 500, and 700 Hz with amplitudes 0.01, 0.005, and 0.0025. Specify the number of harmonics to 7. Determine the THD.

```
fs = 10000;
t = 0:1/fs:1-1/fs;
x = 2*cos(2*pi*100*t)+0.01*cos(2*pi*300*t)+ ...
    0.005*cos(2*pi*500*t)+0.0025*sin(2*pi*700*t);
[sxx,f] = periodogram(x,hamming(length(x)),length(x),fs,'power');
rbw = enbw(hamming(length(x)),fs);
r = thd(sxx,f,rbw,7,'power')
```

r =

```
-44.8396
```

### Harmonic Powers and Corresponding Frequencies

Create a signal sampled at 10 kHz. The signal consists of a 100 Hz fundamental with amplitude 2 and three odd-numbered harmonics at 300, 500, and 700 Hz with amplitudes 0.01, 0.005, and 0.0025. Specify the number of harmonics to 7. Determine the THD, the power at the harmonics, and the corresponding frequencies.

```
fs = 10000;
t = 0:1/fs:1-1/fs;
x = 2*cos(2*pi*100*t)+0.01*cos(2*pi*300*t)+ ...
    0.005*cos(2*pi*500*t)+0.0025*sin(2*pi*700*t);
[r,harpow,harmfreq] = thd(x,10000,7);
[harmfreq harmpow]
```

ans =

```
100.0000    3.0103
201.0000 -320.4988
300.0000  -43.0103
399.0000 -281.9553
500.0000  -49.0309
599.0000 -282.0726
700.0000  -55.0515
```

The powers at the even-numbered harmonics are on the order of  $-300$  dB, which corresponds to an amplitude of  $10^{-15}$ .

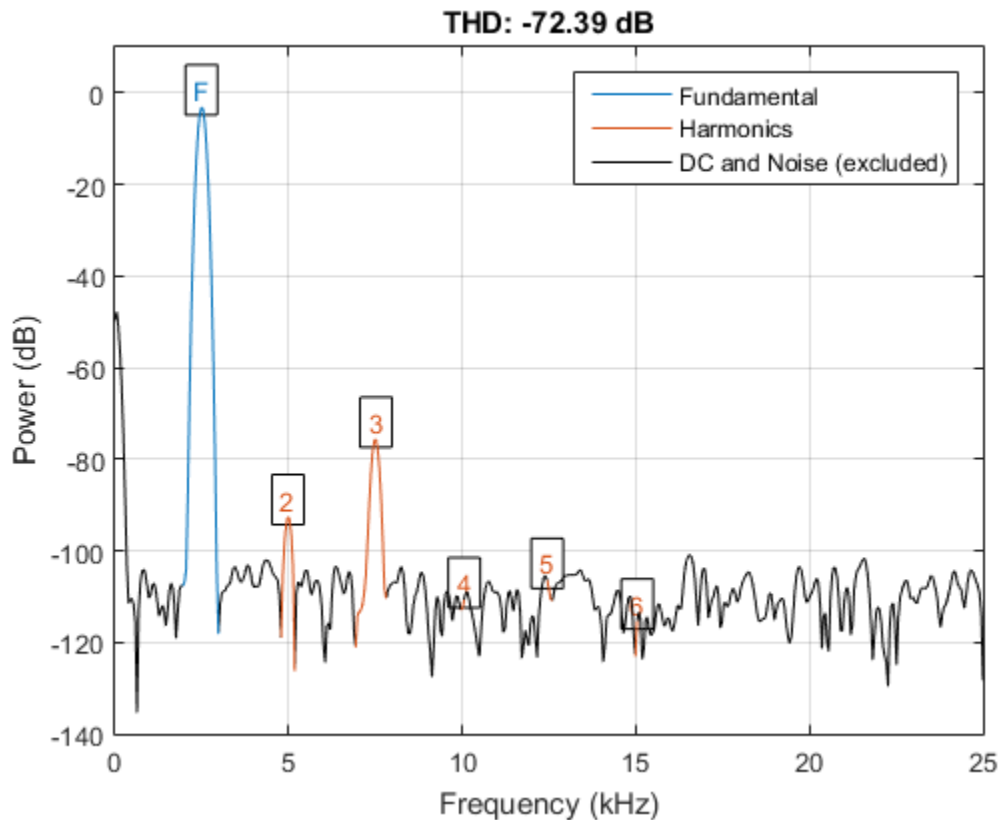
### **THD of an Amplified Signal**

Generate a sinusoid of frequency 2.5 kHz sampled at 50 kHz. Reset the random number generator. Add Gaussian white noise with standard deviation 0.00005 to the signal. Pass the result through a weakly nonlinear amplifier. Plot the THD.

```
rng default
fs = 5e4; f0 = 2.5e3;
N = 1024; t = (0:N-1)/fs;

ct = cos(2*pi*f0*t);
cd = ct + 0.00005*randn(size(ct));

amp = [1e-5 5e-6 -1e-3 6e-5 1 25e-3];
sgn = polyval(amp,cd);
thd(sgn,fs);
```



The plot shows the spectrum used to compute the ratio and the region treated as noise. The DC level is excluded from the computation. The fundamental and harmonics are labeled.

- “Analyzing Harmonic Distortion”

## Input Arguments

**x** — Real-valued sinusoidal input signal  
vector

Real-valued sinusoidal input signal specified as a row or column vector.

Example: `cos(pi/4*(0:159))+cos(pi/2*(0:159))`

Data Types: `single` | `double`

**fs — Sampling frequency**

positive scalar

Sampling frequency, specified as a positive scalar. The sampling frequency is the number of samples per unit time. If the unit of time is seconds, the sampling frequency has units of hertz.

**n — Number of harmonics**

positive integer

Number of harmonics specified as a positive integer.

**pxx — One-sided PSD estimate**

vector

One-sided PSD estimate specified as a real-valued, nonnegative column vector.

Data Types: `single` | `double`

**f — Cyclical frequencies**

vector

Cyclical frequencies corresponding to the one-sided PSD estimate, `pxx`, specified as a row or column vector. The first element of `f` must be 0.

Data Types: `double` | `single`

**sxx — Power spectrum**

nonnegative real-valued row or column vector

Power spectrum specified as a real-valued nonnegative row or column vector.

**rbw — Resolution bandwidth**

positive scalar

Resolution bandwidth specified as a positive scalar. The resolution bandwidth is the product of the frequency resolution of the discrete Fourier transform and the equivalent noise bandwidth of the window.



## Output Arguments

### **r** — Total harmonic distortion in dBc

real-valued scalar

Total harmonic distortion in dBc specified as a real-valued scalar.

### **harmpow** — Power of the harmonics

nonnegative scalar or vector

Power of the harmonics specified as a nonnegative scalar or vector. Whether `harmpow` is a scalar or a vector depends on the number of harmonics you specify as the input argument `n`.

### **harmfreq** — Frequencies of the harmonics

nonnegative scalar or vector

Frequencies of the harmonics specified as a nonnegative scalar or vector. Whether `harmfreq` is a scalar or a vector depends on the number of harmonics you specify as the input argument `n`.

## More About

### Distortion Measurement Functions

The functions `thd`, `sfdr`, `sinad`, and `snr` measure the response of a weakly nonlinear system stimulated by a sinusoid.

When given time-domain input, `thd` performs a periodogram using a Kaiser window with large sidelobe attenuation. To find the fundamental frequency, the algorithm searches the periodogram for the largest nonzero spectral component. It then computes the central moment of all adjacent bins that decrease monotonically away from the maximum. To be detectable, the fundamental should be at least in the second frequency bin. Higher harmonics are at integer multiples of the fundamental frequency. If a harmonic lies within the monotonically decreasing region in the neighborhood of another, its power is considered to belong to the larger harmonic. This larger harmonic may or may not be the fundamental.

`thd` fails if the fundamental is not the highest spectral component in the signal.

Ensure that the frequency components are far enough apart to accommodate for the sidelobe width of the Kaiser window. If this is not feasible, you can use the 'power' flag and compute a periodogram with a different window.

## **See Also**

`sfdr` | `sinad` | `snr` | `toi`

# toi

Third-order intercept point

## Syntax

```
oip3 = toi(x)  
oip3 = toi(x,fs)
```

```
oip3 = toi(pxx,f,'psd')  
oip3 = toi(sxx,f,rbw,'power')
```

```
[oip3,fundpow,fundfreq,imodpow,imodfreq] = toi(____)
```

```
toi(____)
```

## Description

`oip3 = toi(x)` returns the output third-order intercept (TOI) point, in decibels (dB), of a real sinusoidal two-tone input signal, `x`. The computation is performed over a periodogram of the same length as the input using a Kaiser window with  $\beta = 38$ .

`oip3 = toi(x,fs)` specifies the sampling rate, `fs`. The default value of `fs` is 1.

`oip3 = toi(pxx,f,'psd')` specifies the input as a one-sided power spectral density (PSD), `pxx`, of a real signal. `f` is a vector of frequencies that corresponds to the vector of `pxx` estimates.

`oip3 = toi(sxx,f,rbw,'power')` specifies the input as a one-sided power spectrum, `sxx`, of a real signal. `rbw` is the resolution bandwidth over which each power estimate is integrated.

`[oip3,fundpow,fundfreq,imodpow,imodfreq] = toi(____)` also returns the power, `fundpow`, and frequencies, `fundfreq`, of the two fundamental sinusoids. It also returns the power, `imodpow`, and frequencies, `imodfreq`, of the lower and upper intermodulation products. This syntax can use any of the input arguments in the preceding syntaxes.

`toi( ___ )` with no output arguments plots the spectrum of the signal and annotates the lower and upper fundamentals ( $f_1$ ,  $f_2$ ) and intermodulation products ( $2f_1 - f_2$ ,  $2f_2 - f_1$ ). Higher harmonics and intermodulation products are not labeled. The TOI appears above the plot.

## Examples

### Third-Order Intercept Point of a Two-Tone Nonlinear Signal with Noise

Create a two-tone sinusoid with frequencies  $f_1 = 5$  kHz and  $f_2 = 6$  kHz, sampled at 48 kHz. Make the signal nonlinear by feeding it to a polynomial. Add noise. Set the random number generator to the default settings for reproducible results. Compute the third-order intercept point. Verify that the intermodulation products occur at  $2f_2 - f_1 = 4$  kHz and  $2f_1 - f_2 = 7$  kHz.

```
rng default
fi1 = 5e3;
fi2 = 6e3;
Fs = 48e3;
N = 1000;
x = sin(2*pi*fi1/Fs*(1:N))+sin(2*pi*fi2/Fs*(1:N));
y = polyval([0.5e-3 1e-7 0.1 3e-3],x)+1e-5*randn(1,N);
[myTOI,Pfund,Ffund,Pim3,Fim3] = toi(y,Fs)
```

```
myTOI =
    1.3844
Pfund =
   -22.9133   -22.9132
Ffund =
    1.0e+03 *
    5.0000    6.0000
Pim3 =
   -71.4868   -71.5299
Fim3 =
    1.0e+03 *
    4.0002    6.9998
```

### Third-Order Intercept Point from Power Spectral Density

Create a two-tone sinusoid with frequencies 5 kHz and 6 kHz, sampled at 48 kHz. Make the signal nonlinear by evaluating a polynomial. Add noise. Set the random number generator to the default settings for reproducible results.

```
rng default
fi1 = 5e3;
fi2 = 6e3;
Fs = 48e3;
N = 1000;
x = sin(2*pi*fi1/Fs*(1:N))+sin(2*pi*fi2/Fs*(1:N));
y = polyval([0.5e-3 1e-7 0.1 3e-3],x)+1e-5*randn(1,N);
```

Evaluate the periodogram of the signal using a Kaiser window. Compute the TOI using the power spectral density. Plot the result.

```
w = kaiser(numel(y),38);

[Sxx, F] = periodogram(y,w,N,Fs,'psd');
[myTOI,Pfund,Ffund,Pim3,Fim3] = toi(Sxx,F,'psd')

toi(Sxx,F,'psd');
```

```
myTOI =
```

```
1.3843
```

```
Pfund =
```

```
-22.9133 -22.9132
```

```
Ffund =
```

```
1.0e+03 *
5.0000 6.0000
```

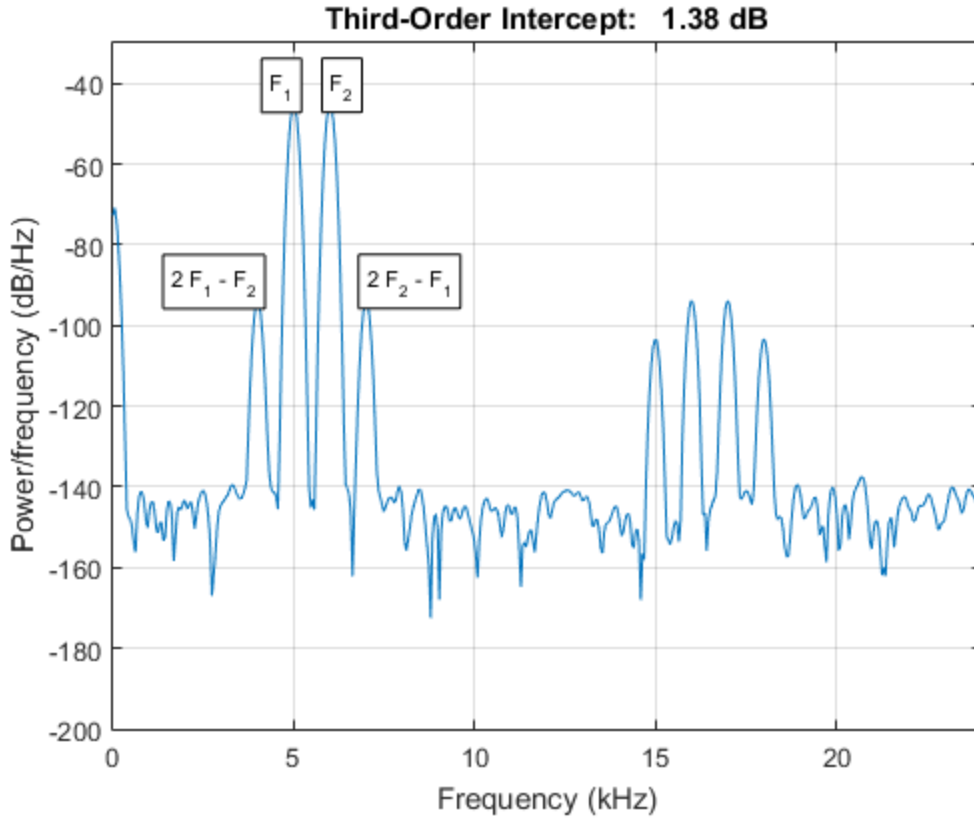
```
Pim3 =
```

```
-71.4868 -71.5299
```

```
Fim3 =
```

```
1.0e+03 *
```

4.0002    6.9998



### Third-Order Intercept Point from Power Spectrum

Create a two-tone sinusoid with frequencies 5 kHz and 6 kHz, sampled at 48 kHz. Make the signal nonlinear by evaluating a polynomial. Add noise. Set the random number generator to the default settings for reproducible results.

```
rng default

fi1 = 5e3;
fi2 = 6e3;
```

```
Fs = 48e3;
N = 1000;

x = sin(2*pi*fi1/Fs*(1:N))+sin(2*pi*fi2/Fs*(1:N));
y = polyval([0.5e-3 1e-7 0.1 3e-3],x)+1e-5*randn(1,N);
```

Evaluate the periodogram of the signal using a Kaiser window. Compute the TOI using the power spectrum. Plot the result.

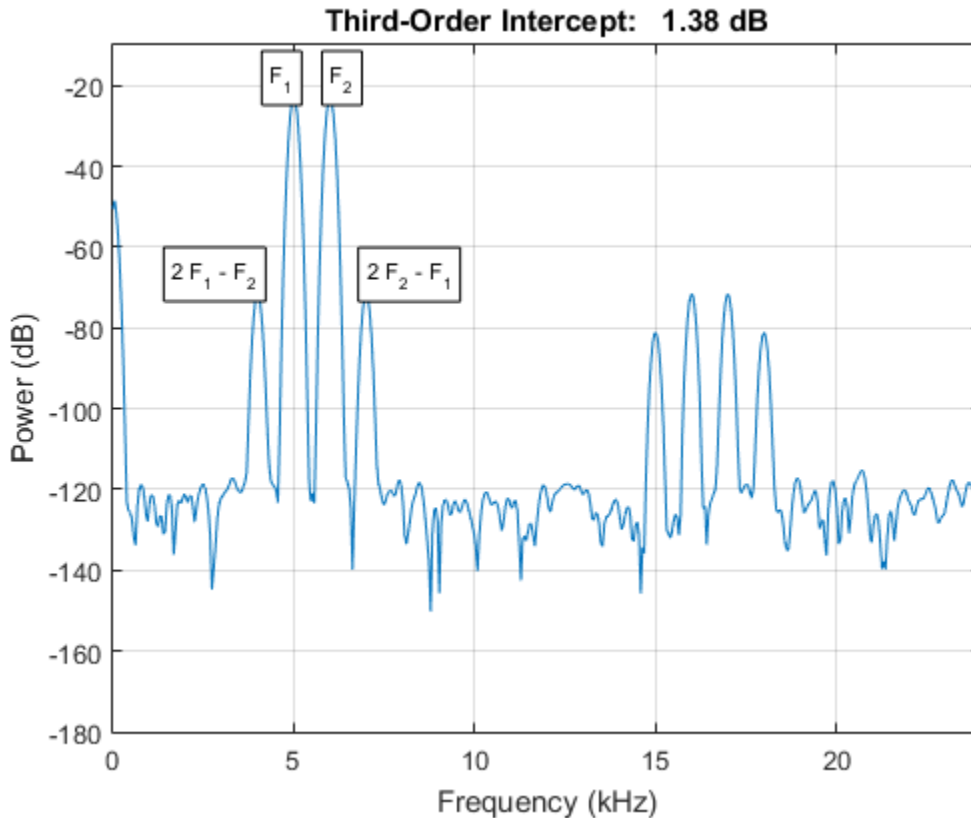
```
w = kaiser(numel(y),38);

[Sxx,F] = periodogram(y,w,N,Fs, 'power');

toi(Sxx,F,enbw(w,Fs), 'power')
```

```
ans =
```

```
1.3844
```



### Intermodulation Distortion Products

Generate 640 samples of a two-tone sinusoid with frequencies 5 Hz and 7 Hz, sampled at 32 Hz. Make the signal nonlinear by evaluating a polynomial. Add noise with standard deviation 0.01. Set the random number generator to the default settings for reproducible results. Compute the third-order intercept point. Verify that the intermodulation products occur at  $2f_2 - f_1 = 9$  Hz and  $2f_1 - f_2 = 3$  Hz.

```
rng default
x = sin(2*pi*5/32*(1:640))+cos(2*pi*7/32*(1:640));
q = x + 0.01*x.^3 + 1e-2*randn(size(x));
[myTOI,Pfund,Ffund,Pim3,Fim3] = toi(q,32)
```



```

myTOI =
    17.4230

Pfund =
    -2.8350    -2.8201

Ffund =
     5.0000     7.0001

Pim3 =
    -43.1362   -43.5211

Fim3 =
     3.0015     8.9744

```

### TOI Plot

Generate 640 samples of a two-tone sinusoid with frequencies 5 Hz and 7 Hz, sampled at 32 Hz. Make the signal nonlinear by evaluating a polynomial. Add noise with standard deviation 0.01. Set the random number generator to the default settings. Plot the spectrum of the signal. Display the fundamentals and the intermodulation products. Verify that the latter occur at 9 Hz and 3 Hz.

```

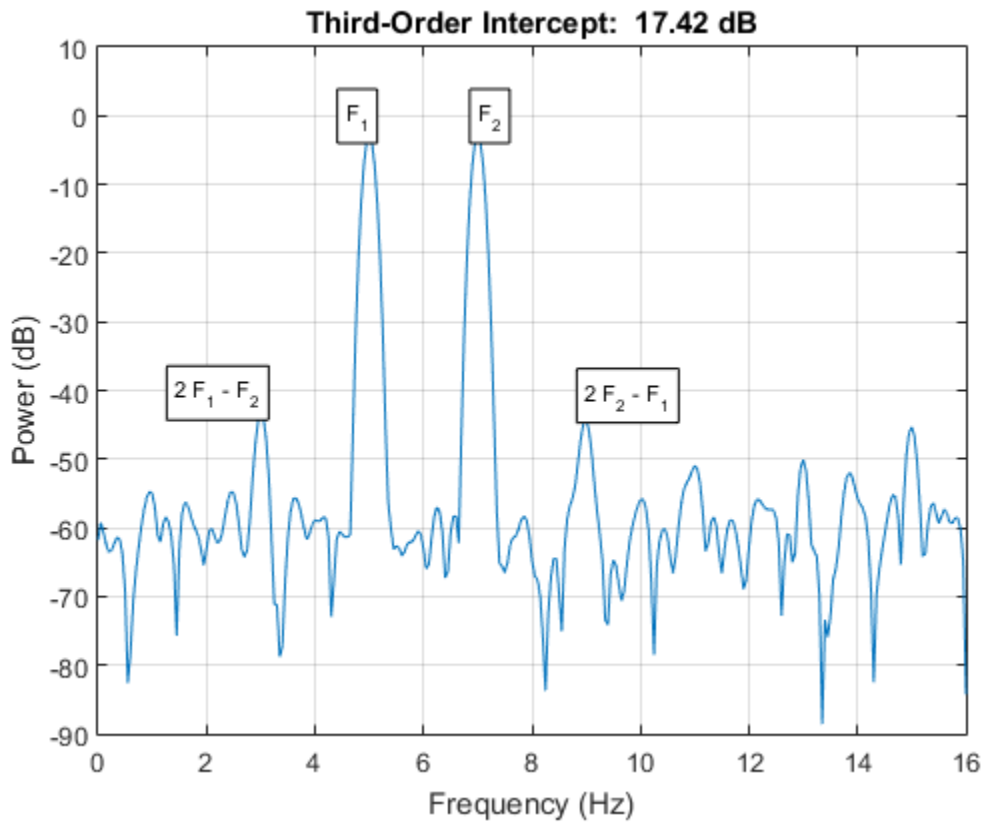
rng default
x = sin(2*pi*5/32*(1:640))+cos(2*pi*7/32*(1:640));
q = x + 0.01*x.^3 + 1e-2*randn(size(x));
toi(q,32)

```

```

ans =
    17.4230

```



## Input Arguments

**x** — Real-valued sinusoidal two-tone signal

vector

Real-valued sinusoidal two-tone signal, specified as a row or column vector.

Example: `polyval([0.01 0 1 0],sum(sin(2*pi*[5 7]'.*(1:640)/32))) + 0.01*randn([1 640])`

Data Types: double | single

**fs — Sampling frequency**

1 (default) | positive real scalar

Sampling frequency, specified as a positive real scalar. The sampling frequency is the number of samples per unit time. If the unit of time is seconds, the sampling frequency has units of hertz.

Data Types: double | single

**pxx — One-sided PSD estimate**

vector

One-sided power spectral density estimate, specified as a real-valued, nonnegative row or column vector.

Data Types: double | single

**f — Cyclical frequencies**

vector

Cyclical frequencies corresponding to the one-sided PSD estimate, pxx, specified as a row or column vector. The first element of f must be 0.

Data Types: double | single

**sxx — Power spectrum**

nonnegative real-valued row or column vector

Power spectrum, specified as a real-valued nonnegative row or column vector.

Data Types: double | single

**rbw — Resolution bandwidth**

positive scalar

Resolution bandwidth, specified as a positive scalar. The resolution bandwidth is the product of the frequency resolution of the discrete Fourier transform and the equivalent noise bandwidth of the window.

Data Types: double | single

## Output Arguments

### **oip3** — Third-order intercept point

scalar

Output third-order intercept point of a sinusoidal two-tone signal, returned as a real-valued scalar expressed in decibels. If the second primary tone is the second harmonic of the first primary tone, then the lower intermodulation product is at zero frequency. The function returns NaN in those cases.

Data Types: `double` | `single`

### **fundpow** — Power of fundamental sinusoids

two-element real row vector

Power contained in the two fundamental sinusoids of the input signal, returned as a real-valued two-element row vector.

Data Types: `double` | `single`

### **fundfreq** — Frequencies of fundamental sinusoids

two-element real row vector

Frequencies of the two fundamental sinusoids of the input signal, returned as a real-valued two-element row vector.

Data Types: `double` | `single`

### **imodpow** — Power of intermodulation products

two-element real row vector

Power contained in the lower and upper intermodulation products of the input signal, returned as a real-valued two-element row vector.

Data Types: `double` | `single`

### **imodfreq** — Frequencies of intermodulation products

two-element real row vector

Frequencies of the lower and upper intermodulation products of the input signal, returned as a real-valued two-element row vector.

Data Types: `double` | `single`

## References

- [1] Kundert, Kenneth S. “Accurate and Rapid Measurement of  $IP_2$  and  $IP_3$ .” May, 2002.  
<http://www.designers-guide.org/Analysis/intercept-point.pdf>.

## See Also

sfdr | sinad | snr | thd

## triang

Triangular window

### Syntax

```
w = triang(L)
```

### Description

`w = triang(L)` returns an L-point triangular window in the column vector, `w`.

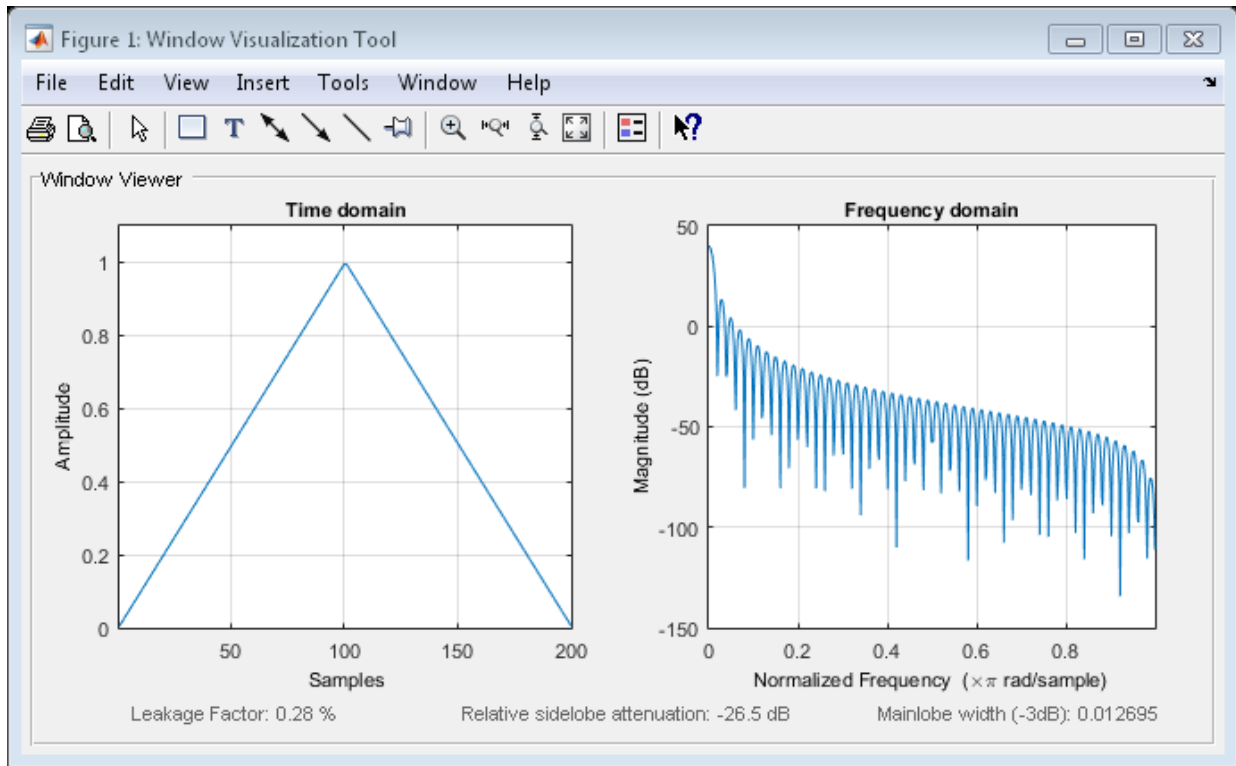
See “Definitions” on page 1-1681 for the equations that define the triangular window. The triangular window is very similar to a Bartlett window. The Bartlett window always ends with zeros at samples 1 and L, while the triangular window is nonzero at those points. For L odd, the center L-2 points of `triang(L-2)` are equivalent to `bartlett(L)`.

### Examples

#### Triangular Window

Create a 200-point triangular window. Display the result using `wvtool`.

```
L = 200;  
w = triang(L);  
wvtool(w)
```



## Definitions

The coefficients of a triangular window are the following.

For  $L$  odd:

$$w(n) = \begin{cases} \frac{2n}{L+1} & 1 \leq n \leq (L+1)/2 \\ 2 - \frac{2n}{L+1} & (L+1)/2 + 1 \leq n \leq L \end{cases}$$

For  $L$  even:

$$w(n) = \begin{cases} \frac{(2n-1)}{L} & 1 \leq n \leq L/2 \\ 2 - \frac{(2n-1)}{L} & L/2+1 \leq n \leq L \end{cases}$$

## References

- [1] Oppenheim, Alan V., Ronald W. Schaffer, and John R. Buck. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1999.

## See Also

barthannwin | bartlett | blackmanharris | bohmanwin | nuttallwin |  
parzenwin | rectwin | window | wintool | wvtool



# tripuls

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.

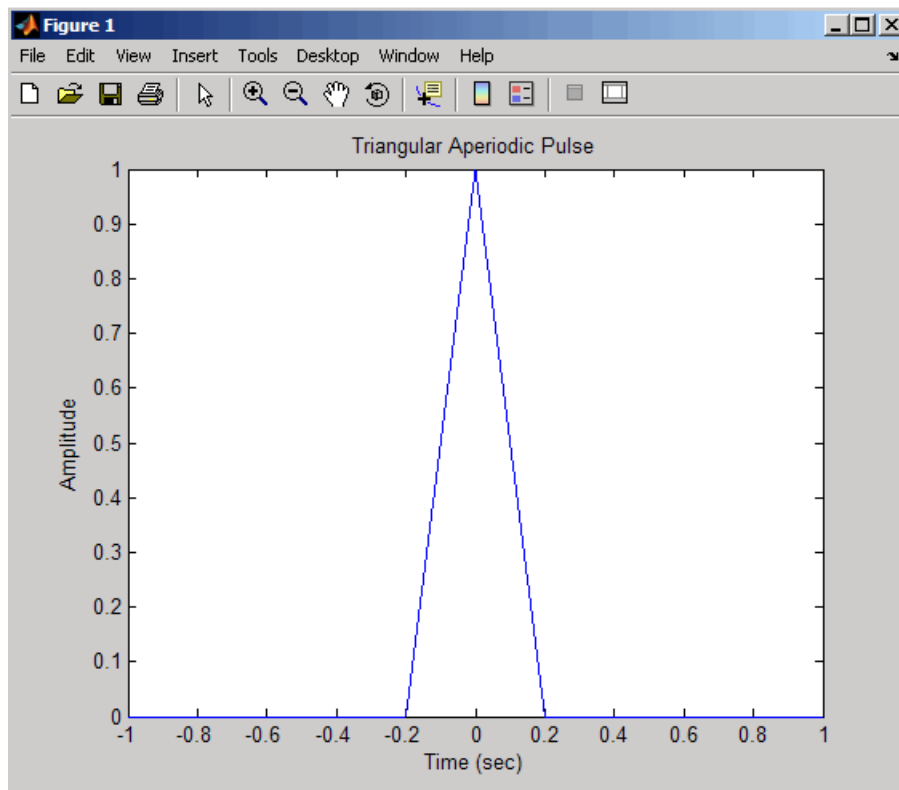
`y = tripuls(T,w)` generates a triangular pulse of width `w`.

`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

# tukeywin

Tukey (tapered cosine) window

## Syntax

```
w = tukeywin(L,r)
```

## Description

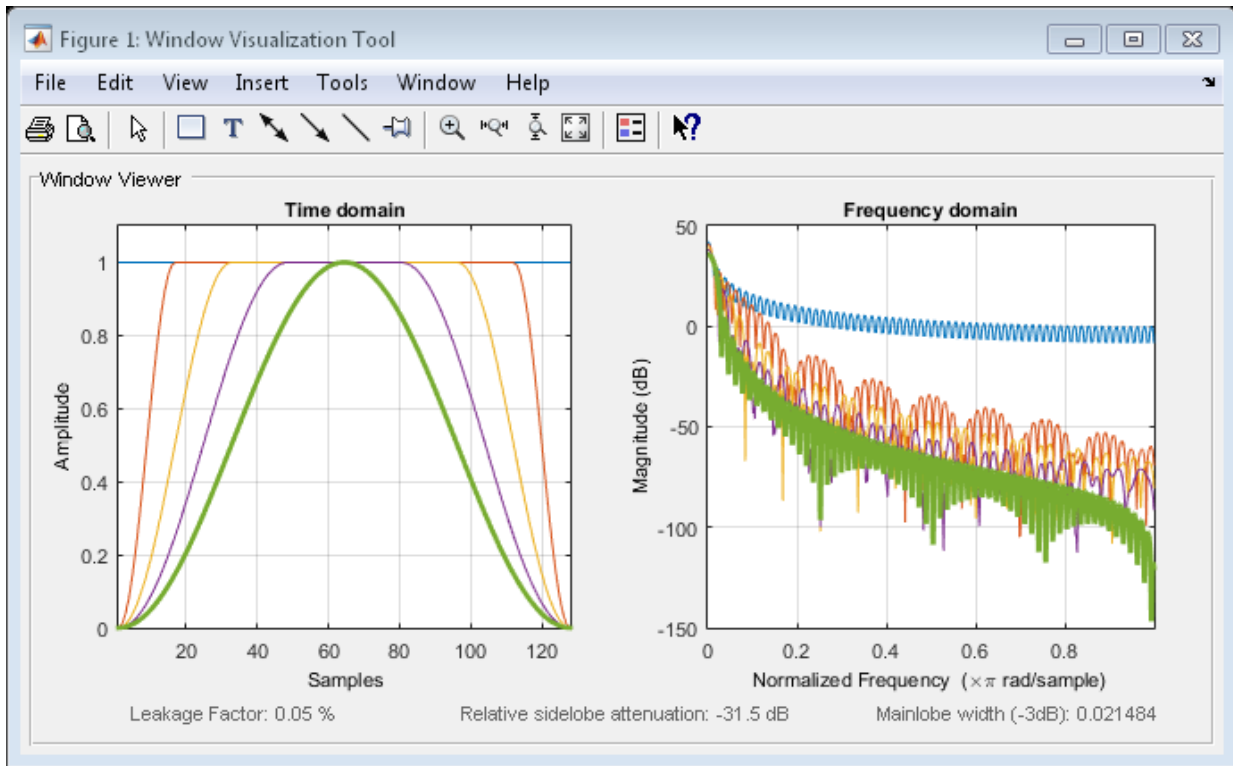
`w = tukeywin(L,r)` returns an L-point Tukey window in the column vector, `w`. A Tukey window is a rectangular window with the first and last  $r/2$  percent of the samples equal to parts of a cosine. See “Definitions” on page 1-1686 for the equation that defines the Tukey window. `r` is a real number between 0 and 1. If you input  $r \leq 0$ , you obtain a `rectwin` window. If you input  $r \geq 1$ , you obtain a `hann` window. `r` defaults to 0.5.

## Examples

### Tukey Windows

Compute 128-point Tukey windows with five different values of `r`, or "tapers." Display the results using `wvtool`.

```
L = 128;
t0 = tukeywin(L,0);           % Equivalent to a rectangular window
t25 = tukeywin(L,0.25);
t5 = tukeywin(L);            % r = 0.5
t75 = tukeywin(L,0.75);
t1 = tukeywin(L,1);          % Equivalent to a Hann window
wvtool(t0,t25,t5,t75,t1)
```



## Definitions

The following equation defines the  $L$ -point Tukey window:

$$w(x) = \begin{cases} \frac{1}{2} \{1 + \cos(\frac{2\pi}{r} [x - r/2])\}, & 0 \leq x < \frac{r}{2} \\ 1, & \frac{r}{2} \leq x < 1 - \frac{r}{2} \\ \frac{1}{2} \{1 + \cos(\frac{2\pi}{r} [x - 1 + r/2])\}, & 1 - \frac{r}{2} \leq x \leq 1 \end{cases}$$

where  $x$  is an  $L$ -point linearly spaced vector generated using `linspace`. The parameter  $r$  is the ratio of cosine-tapered section length to the entire window length with  $0 \leq r \leq 1$ .

For example, setting  $r = 0.5$  produces a Tukey window where  $1/2$  of the entire window length consists of segments of a phase-shifted cosine with period  $2r = 1$ . If you specify  $r \leq 0$ , an  $L$ -point rectangular window is returned. If you specify  $r \geq 1$ , an  $L$ -point von Hann window is returned.

## References

- [1] Bloomfield, P. *Fourier Analysis of Time Series: An Introduction*. New York: Wiley-Interscience, 2000.

## See Also

`chebwin` | `kaiser` | `window` | `gausswin` | `wintool` | `wvtool`

## udecode

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  $u = \text{mod}(u+2^{n/2}, 2^n) - (2^{n/2})$ ).
  - 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  $u = \text{mod}(u, 2^n)$ ).

## Examples

### Use udecode to Decode Integers

Create a signed eight-bit integer string. Decode with three bits.

```
u = int8([-1 1 2 -5]);
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 to 6.

```
ysatv = udecode(u,3,6)
```

```
ysatv =
```

```
-1.5000    1.5000    3.0000   -6.0000
```

The last input entry still saturates. Wrap the overflows.

```
ywrap = udecode(u,3,6,'wrap')
```

```
ywrap =  
    -1.5000    1.5000    3.0000    4.5000
```

Add more quantization levels.

```
yprec = udecode(u,5)
```

```
yprec =  
    -0.0625    0.0625    0.1250   -0.3125
```

## More About

### Algorithms

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] International Telecommunication Union. *General Aspects of Digital Transmission Systems: Vocabulary of Digital Transmission and Multiplexing, and Pulse Code Modulation (PCM) Terms*. ITU-T Recommendation G.701. March, 1993.

### See Also

uencode



# uencode

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.

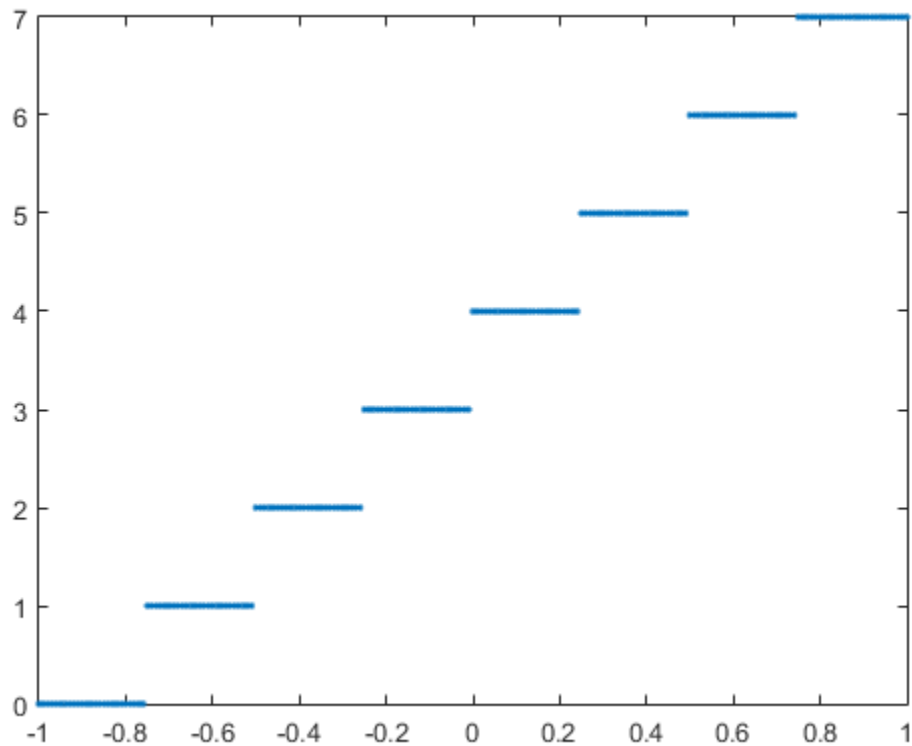
| <b>n</b> | <b>Unsigned Integer</b> | <b>Signed Integer</b> |
|----------|-------------------------|-----------------------|
| 2 to 8   | uint8                   | int8                  |
| 9 to 16  | uint16                  | int16                 |
| 17 to 32 | uint32                  | int32                 |

## Examples

### Map Floating-Point Scalars to Integers

Map floating-point scalars in  $[-1, 1]$  to `uint8` (unsigned) integers. Produce a staircase plot. The horizontal axis ranges from -1 to 1 and the vertical axis from 0 to 7 (i.e.,  $2^3 - 1$ ).

```
u = -1:0.01:1;  
y = uencode(u,3);  
plot(u,y, '.')
```



Look at saturation effects when you underspecify 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
```

Specify you want signed output.

```
u = -2:0.5:2;
```

```
y = uencode(u,5,2, 'signed')
```

```
y =
```

```
  -16  -12   -8   -4   0   4   8  12  15
```

## More About

### Algorithms

`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] International Telecommunication Union. *General Aspects of Digital Transmission Systems: Vocabulary of Digital Transmission and Multiplexing, and Pulse Code Modulation (PCM) Terms*. ITU-T Recommendation G.701. March, 1993.

### See Also

`udecode`

# unshiftdata

Inverse of shiftdata

## Syntax

```
y = unshiftdata(x,perm,nshifts)
```

## Description

`y = unshiftdata(x,perm,nshifts)` restores the orientation of the data that was shifted with `shiftdata`. The permutation vector is given by `perm`, and `nshifts` is the number of shifts that was returned from `shiftdata`.

`unshiftdata` is meant to be used in tandem with `shiftdata`. These functions are useful for creating functions that work along a certain dimension, like `filter`, `goertzel`, `sgolayfilt`, and `sosfilt`.

## Examples

### Permute Dimensions of a Magic Square

This example shifts `x`, a 3-by-3 magic square, permuting dimension 2 to the first column. `unshiftdata` shifts `x` back to its original shape.

Create a 3-by-3 magic square.

```
x = magic(3)
```

```
x =
```

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

Shift the matrix `x` to work along the second dimension. The permutation vector, `perm`, and the number of shifts, `nshifts`, are returned along with the shifted matrix.

```
[x,perm,nshifts] = shiftdata(x,2)
```

```
x =
```

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

```
perm =
```

```
     2     1
```

```
nshifts =
```

```
     []
```

Shift the matrix back to its original shape.

```
y = unshiftdata(x,perm,nshifts)
```

```
y =
```

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

### **Rearrange Array to Operate on First Nonsingleton Dimension**

This example shows how `shiftdata` and `unshiftdata` work when you define `dim` as empty.

Define `x` as a row vector.

```
x = 1:5
```

```
x =  
    1    2    3    4    5
```

Define `dim` as empty to shift the first non-singleton dimension of `x` to the first column. `shiftdata` returns `x` as a column vector, along with `perm`, the permutation vector, and `nshifts`, the number of shifts.

```
[x,perm,nshifts] = shiftdata(x,[])
```

```
x =  
    1  
    2  
    3  
    4  
    5
```

```
perm =  
    []
```

```
nshifts =  
    1
```

Using `unshiftdata`, restore `x` to its original shape.

```
y = unshiftdata(x,perm,nshifts)
```

```
y =  
    1    2    3    4    5
```

## See Also

`ipermute` | `shiftdata` | `shiftdim`

# upfirdn

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.



$y_{out} = \text{upfirdn}(x_{in}, h, p)$  specifies the integer upsampling factor  $p$ , where  $p$  has a default value of 1.

$y_{out} = \text{upfirdn}(x_{in}, h, p, q)$  specifies the integer downsampling factor  $q$ , where  $q$  has a default value of 1. The length of the output,  $y_{out}$ , is  $\text{ceil}(((\text{length}(x_{in}) - 1) * p + \text{length}(h)) / q)$

---

**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`.

---

## Examples

### Convert from DAT Rate to CD Sampling Rate

Change the sampling rate of a 1 kHz sinusoid by a factor of 147/160. This factor is used to convert from 48 kHz (DAT rate) to 44.1 kHz (CD sampling rate).

```

Fs = 48e3;                % Original sampling frequency-48kHz
L = 147;                  % Interpolation/decimation factors
M = 160;
N = 24*L;
h = fir1(N-1,1/M,kaiser(N,7.8562));
h = L*h;                  % Passband gain is L

n = 0:10239;              % 10240 samples, 0.213 seconds long
x = sin(2*pi*1e3/Fs*n);   % Original signal
y = upfirdn(x,h,L,M);     % 9430 samples, still 0.213 seconds

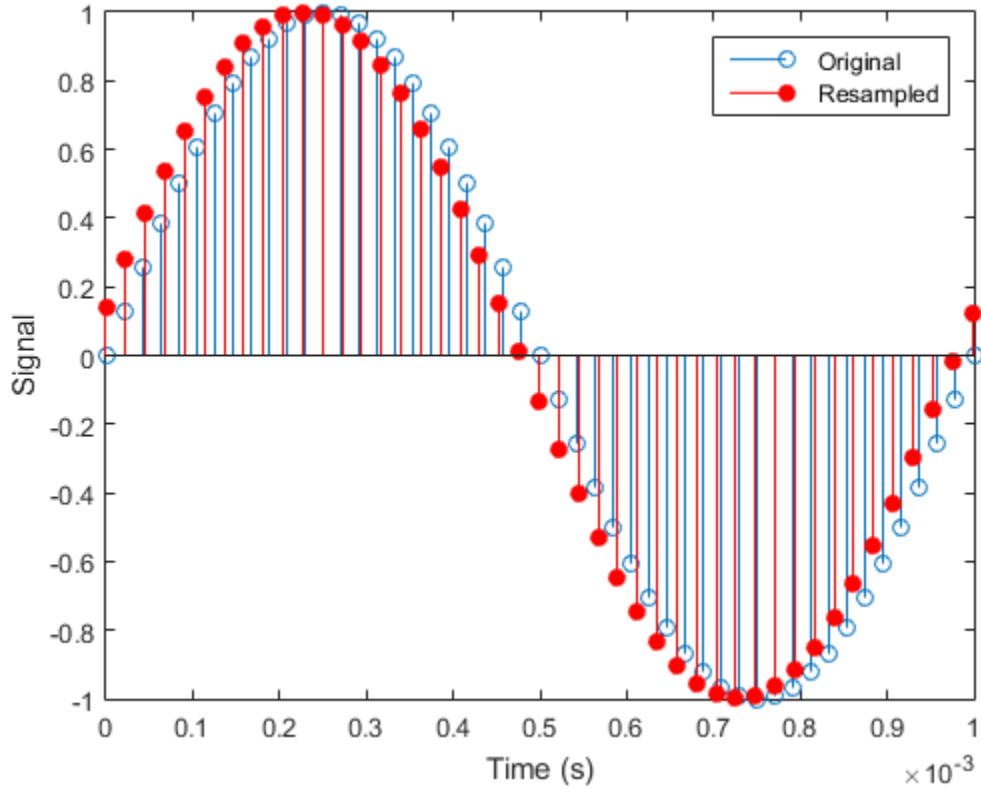
```

Plot the first millisecond of the original signal and overlay the resampled version.

```

stem(n(1:49)/Fs,x(1:49))
hold on
stem(n(1:45)/(Fs*L/M),y(13:57),'r','filled')
xlabel('Time (s)')
ylabel('Signal')
legend('Original','Resampled')

```



## 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.

## More About

### Tips

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 a 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 a 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 a matrix with the same number of columns as `xin` and `h`.

### Algorithms

`upfirdn` uses a polyphase interpolation structure. The number of multiply-add operations in the polyphase structure is approximately  $(L_h L_x - p L_x) / q$  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.

## References

- [1] Crochiere, R. E., and Lawrence R. Rabiner. *Multirate Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 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* (Digital Signal Processing Committee of the IEEE Acoustics, Speech, and Signal Processing Society, eds.). New York: IEEE Press, 1979, Programs 8.2-1–8.2-7.

**See Also**

`conv` | `decimate` | `filter` | `interp` | `downsample` | `intfilt` | `resample` | `upsample`

# upsample

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 Sampling Rates

Increase the sampling rate of a sequence by 3.

```
x = [1 2 3 4];
y = upsample(x,3)
```

```
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)
```

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)
```

y =

```
1 2  
0 0  
0 0  
3 4  
0 0  
0 0  
5 6  
0 0  
0 0
```

## See Also

[decimate](#) | [interp](#) | [downsample](#) | [interp1](#) | [resample](#) | [spline](#) | [upfirdn](#)

# undershoot

Undershoot metrics of bilevel waveform transitions

## Syntax

```
US = undershoot(X)
US = undershoot(X,FS)
US = undershoot(X,T)
[US,USLEV,USINST] = undershoot(...)
[...] = undershoot(...,Name,Value)
undershoot(...)
```

## Description

`US = undershoot(X)` returns the greatest deviations below the final state levels of each transition in the bilevel waveform, `X`. The undershoots, `US`, are expressed as a percentage of the difference between the state levels. See “Undershoot” on page 1-1708. The length of `US` corresponds to the number of transitions detected in the input signal. The sample instants in `X` correspond to the vector indices. To determine the transitions, `undershoot` estimates the state levels of the input waveform by a histogram method. `undershoot` identifies all regions that cross the upper-state boundary of the low state and the lower-state boundary of the high state. The low-state and high-state boundaries are expressed as the state level plus or minus a multiple of the difference between the state levels. See “State-Level Tolerances” on page 1-1709.

`US = undershoot(X,FS)` specifies the sampling frequency, `FS`, in hertz. The sampling frequency determines the sample instants corresponding to the elements in `X`. The first sample instant in `X` corresponds to  $t=0$ .

`US = undershoot(X,T)` specifies the sample instants, `T`, as a vector with the same number of elements as `X`.

`[US,USLEV,USINST] = undershoot(...)` returns the levels, `USLEV`, and sample instants, `USINST`, of the undershoots for each transition.

`[...] = undershoot(...,Name,Value)` returns the greatest deviations below the final state level with additional options specified by one or more `Name,Value` pair arguments.

`undershoot(...)` plots the bilevel waveform and marks the location of the undershoot of each transition as well as the lower- and upper reference-level instants and the associated reference levels. `undershoot` also plots the state levels and associated lower- and upper-state boundaries.

## Input Arguments

### **X**

Bilevel waveform. X is a real-valued row or column vector.

### **FS**

Sample rate in hertz.

### **T**

Vector of sample instants. The length of T must equal the length of the bilevel waveform, X.

## Name-Value Pair Arguments

### **'PercentReferenceLevels'**

Reference levels as a percentage of the waveform amplitude. The lower-state level is defined to be 0 percent. The upper-state level is defined to be 100 percent. The value of `'PercentReferenceLevels'` is a 2-element real row vector whose elements correspond to the lower and upper percent reference levels.

**Default:** [10 90]

### **'Region'**

Specify the region over which to perform the undershoot computation. Valid values for `'Region'` are `'Preshoot'` or `'Postshoot'`. If you specify `'Preshoot'`, the end of the pretransition aberration region is defined as the last instant when the signal exits the first state. If you specify `'Postshoot'`, the start of the posttransition aberration region is defined as the instant when the signal enters the second state.



**Default:** 'Postshoot'

**'SeekFactor'**

Aberration region duration. Specifies the duration of the region over which to compute the undershoot for each transition as a multiple of the corresponding transition duration. The edge of the waveform may be reached, or a complete intervening transition may be detected, before the duration aberration region duration elapses. In such cases, the duration is truncated to the edge of the waveform or the start of the intervening transition.

**Default:** 3

**'StateLevels'**

Lower- and upper-state levels. Specify the levels to use for the lower- and upper-state levels as a 2-element real row vector whose first and second elements correspond to the lower- and upper-state levels of the input waveform.

**'Tolerance'**

Specify the tolerance that the initial and final levels of each transition must be within the respective state levels. The 'Tolerance' value is a scalar expressing a percentage of the difference between the upper- and lower-state levels. See “State-Level Tolerances” on page 1-1709.

**Default:** 2

## Output Arguments

### US

Undershoots expressed as a percentage of the state levels. The undershoot percentages are computed based on the greatest deviation from the final state level in each transition. By default undershoots are computed for posttransition aberration regions. See “Undershoot” on page 1-1708.

### USLEV

Level of the pretransition or posttransition undershoot.

**USINST**

Sample instants of pretransition or posttransition undershoots. If you specify the sampling frequency or sampling instants, the undershoot instants are in seconds. If you do not specify the sampling frequency or sampling instants, the undershoot instants are the indices of the input vector.

**Definitions****Undershoot**

For a positive-going (positive-polarity) pulse, undershoot expressed as a percentage is

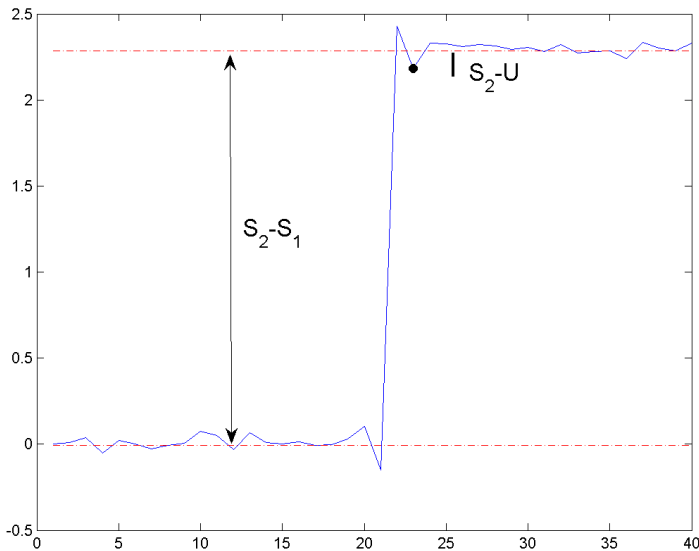
$$100 \frac{(S_2 - U)}{(S_2 - S_1)}$$

where  $U$  is the greatest deviation below the high-state level,  $S_2$  is the high state, and  $S_1$  is the low state.

For a negative-going (negative-polarity) pulse, undershoot expressed as a percentage is

$$100 \frac{(S_1 - U)}{(S_2 - S_1)}$$

The following figure illustrates the calculation of undershoot for a positive-going transition.



The red dashed lines indicate the estimated state levels. The double-sided black arrow depicts the difference between the high- and low-state levels. The solid black line indicates the difference between the high-state level and the undershoot value.

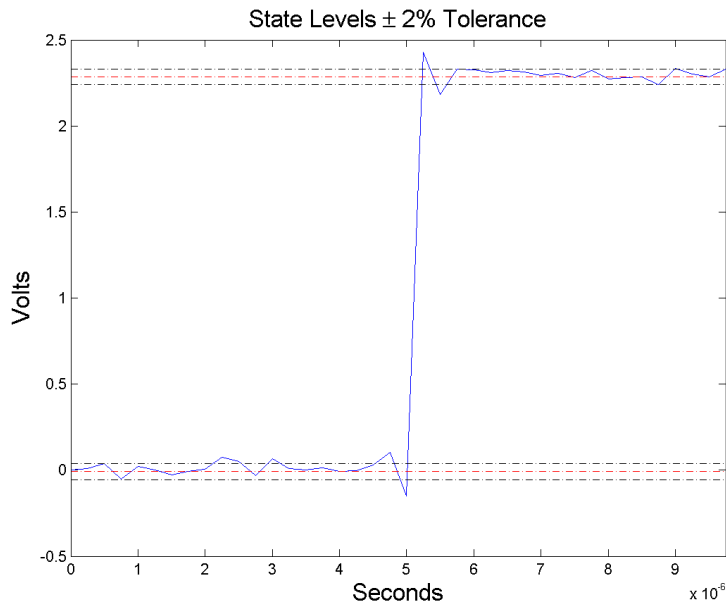
## State-Level Tolerances

Each state level can have associated lower- and upper-state boundaries. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference between the high state and low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the  $\alpha\%$  tolerance region for the low state is defined as

$$S_1 \pm \frac{\alpha}{100}(S_2 - S_1)$$

where  $S_1$  is the low-state level and  $S_2$  is the high-state level. Replace the first term in the equation with  $S_2$  to obtain the  $\alpha\%$  tolerance region for the high state.

The following figure illustrates lower and upper 2% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The red dashed lines indicate the estimated state levels.



## Examples

### Undershoot Percentage in Posttransition Aberration Region

Determine the maximum percent undershoot relative to the high-state level in a 2.3 V clock waveform.

Load the 2.3 V clock data. Plot the waveform. In this example, you see that the maximum undershoot in the posttransition region occurs near index 23.

```
load('transitionex.mat', 'x');
plot(x);
set(gca,'xtick',[1 5 12 19 23 30 40]);
grid on;
```

Determine the maximum percent undershoot.

```
us = undershoot(x);
```

### Undershoot Percentage, Levels, and Sample Instant in Posttransition Aberration Region

Determine the maximum percent undershoot relative to the high-state level, the level of the undershoot, and the sample instant in a 2.3 V clock waveform.

Load the 2.3 V clock data with sampling instants. Plot the waveform. The clock data is sampled at 4 MHz.

```
load('transitionex.mat', 'x','t');
plot(t,x);
```

Determine the maximum percent undershoot, the level of the undershoot in volts, and the sampling instant where the maximum undershoot occurs. Plot the result.

```
[us,uslev,usinst] = undershoot(x,t);
plot(t.*1e6,x); xlabel('Microseconds');
hold on; grid on;
plot(usinst*1e6,uslev,'ro','markerfacecolor',[1 0 0]);
```

### Undershoot Percentage, Levels, and Sample Instant in Pretransition Aberration Region

Determine the maximum percent undershoot relative to the low-state level, the level of the undershoot, and the sample instant in a 2.3 V clock waveform. Specify the 'Region' as 'Preshoot' to output pre-transition metrics.

Load the 2.3 V clock data with sampling instants. Plot the waveform. The clock data is sampled at 4 MHz.

```
load('transitionex.mat', 'x','t');
plot(t,x);
```

Determine the maximum percent undershoot, the level of the undershoot in volts, and the sampling instant where the maximum undershoot occurs. Plot the result.

```
load('transitionex.mat', 'x','t');
[us,uslev,usinst] = undershoot(x,t,'Region','Preshoot');
plot(t.*1e6,x); xlabel('Microseconds');
hold on; grid on;
```

```
plot(usinst*1e6,uslev,'ro','markerfacecolor',[1 0 0]);
```

## References

- [1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003, pp. 15–17.

## See Also

[overshoot](#) | [settlingtime](#) | [statelevels](#)

# validstructures

Structures for specification object with design method

## Syntax

```
filtstruct = validstructures(D)
C = validstructures(D,METHOD)
Cs = validstructures(D,...,'SystemObject',sysobjflag)
```

## Description

`filtstruct = validstructures(D)` returns a structure array containing all valid filter structures for the filter specification object, `D`, organized by design method. Each design method is a field in the structure array, `filtstruct`. The fields contain a cell array of strings.

`C = validstructures(D,METHOD)` returns the valid structures for the filter specification object, `D`, and the design method, `METHOD`, in a cell array of strings.

`Cs = validstructures(D,...,'SystemObject',sysobjflag)` returns the valid structures for designing a filter System object when *sysobjflag* is `true`. To use System objects, you must have the DSP System Toolbox product installed. When *sysobjflag* is `false`, the function returns valid structures for designing `dfilt` and `mfilt` objects, as described previously. Design methods and design options for filter System objects are not necessarily the same as those for `dfilt` and `mfilt` objects.

## Examples

Design a default lowpass filter specification object. Return all valid design methods and structures in a structure array. Display the fieldnames to see all valid design methods. Display the valid filter structures for the `equiripple` field.

```
D = fdesign.lowpass;
filtstruct = validstructures(D);
fieldnames(filtstruct)
```

`filtstruct.equiripple`

Create a highpass filter of order 50 with a 3-dB frequency of 0.2. Obtain the available structures for a Butterworth design.

```
D = fdesign.highpass('N,F3dB',50,0.2);  
C = validstructures(D,'butter');
```

If you have DSP System Toolbox software installed, use the `'SystemObject'`, `sysobjflag` syntax to return valid structures for a filter System object:

```
Cs = validstructures(D,'butter','SystemObject',true);
```

## See Also

`design` | `designmethods` | `designopts` | `fdesign`



## vco

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  $\pm 1$  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

### Spectrogram of Sawtooth Signal

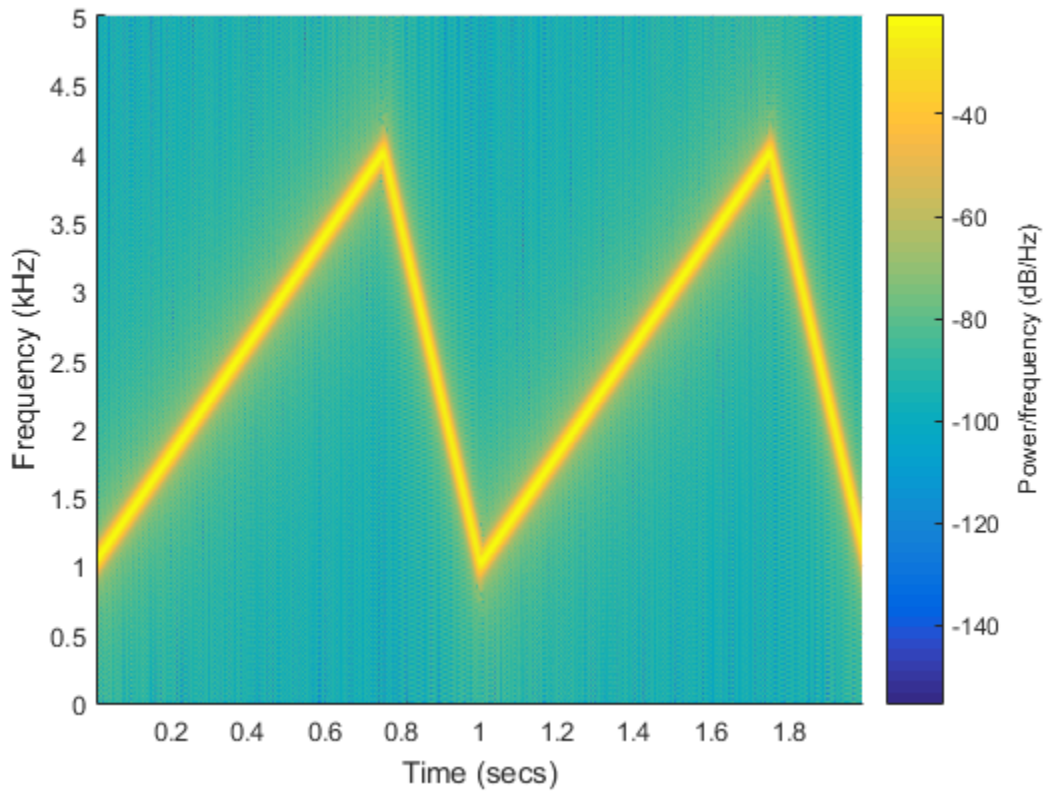
Generate two seconds of a signal sampled at 10 kHz whose instantaneous frequency is a triangle function of time.

```
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')
```



## Diagnostics

If any values of  $x$  lie outside  $[-1, 1]$ , `vco` gives the following error message:

```
X outside of range [-1,1].
```

## More About

### Algorithms

vco performs FM modulation using the `modulate` function.

### See Also

demod | modulate

# window

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.

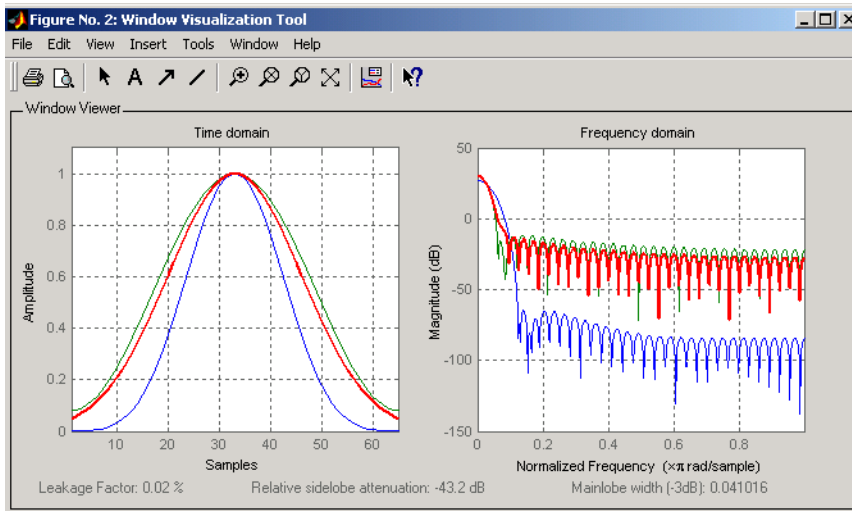
| Window                  | winopt Description  | winopt Value   |
|-------------------------|---|--|
| <code>blackman</code>   | sampling flag string  | 'periodic' or 'symmetric'                                  |
| <code>chebwin</code>    | sidelobe attenuation relative to mainlobe   | numeric  |
| <code>flattopwin</code> | sampling flag string  | 'periodic' or 'symmetric'                                  |
| <code>gausswin</code>   | alpha value (reciprocal of standard deviation)                                      | numeric  |
| <code>hamming</code>    | sampling flag string  | 'periodic' or 'symmetric'                                  |
| <code>hann</code>       | sampling flag string  | 'periodic' or 'symmetric'                                  |
| <code>kaiser</code>     | beta value  | numeric  |
| <code>taylorwin</code>  | 1. number of sidelobes<br>2. maximum sidelobe level in dB relative to mainlobe peak | 1. integer greater than or equal to 1<br>2. negative value |
| <code>tukeywin</code>   | ratio of taper to constant sections   | numeric  |

## Examples

Create Blackman Harris, Hamming, and Gaussian windows and plot them in the same WVTTool.

```
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

# window (filter design method)

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`.

`fcnhdl` 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.

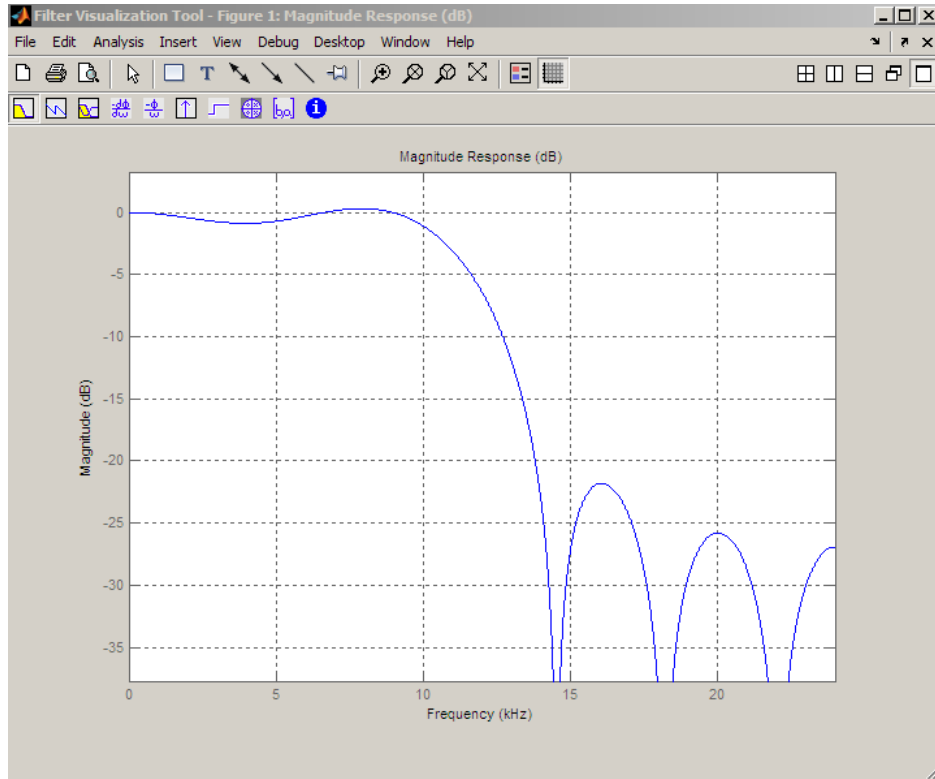
`h = window(d, win)` designs a filter using the vector you supply in `win`. The length of vector `win` must be the same as the impulse response of the filter, which is equal to the filter order plus one.

## Examples

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](#)



# wintool

Open Window Design and Analysis Tool

## Syntax

wintool

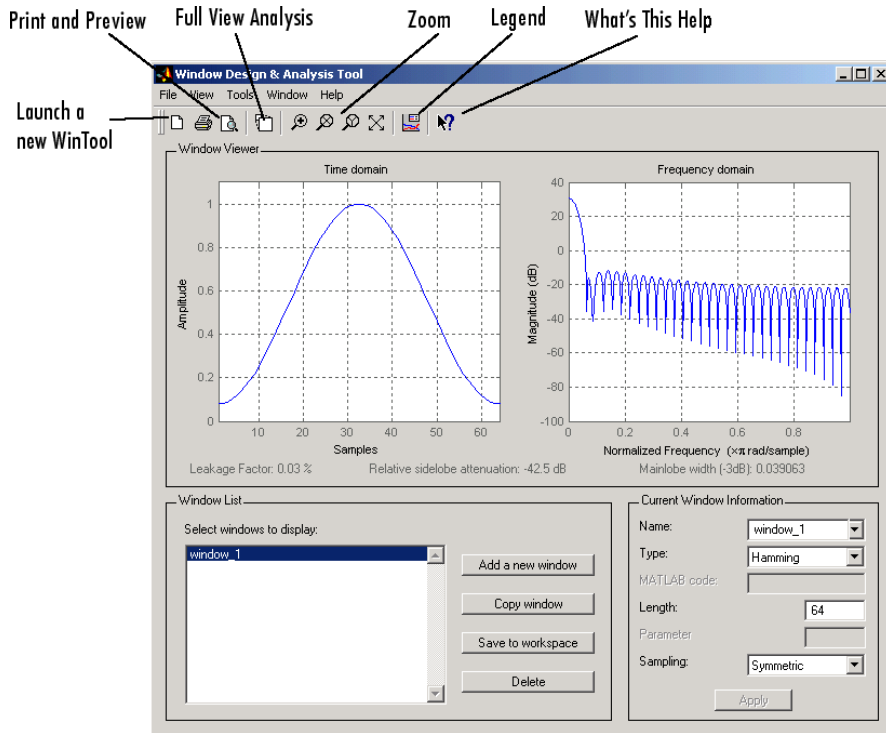
## Description

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.

---

**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:

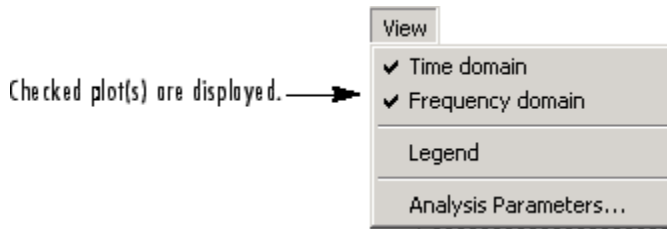
- **Export** — Exports window coefficient vectors to the MATLAB workspace, a text file, or a MAT-file.

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 set of window coefficients. 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               | s           |
| kHz              | ms          |
| MHz              | μs          |
| GHz              | ns          |

**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

## wvtool

Open Window Visualization Tool

### Syntax

```
wvtool(WindowVector)  
wvtool(WindowVector1,...,WindowVectorN)  
H = wvtool(...)
```

### Description

`wvtool(WindowVector)` opens the Window Visualization Tool (WVTool) with time and frequency domain plots of the window vector specified in `WindowVector`. `WindowVector` must be a real-valued row or column vector. By default, the frequency domain plot is the magnitude squared of the Fourier transform of the window vector in decibels (dB). You can generate window vectors for a number of common window functions using the Signal Processing Toolbox software. See `window` for a list of supported window functions.

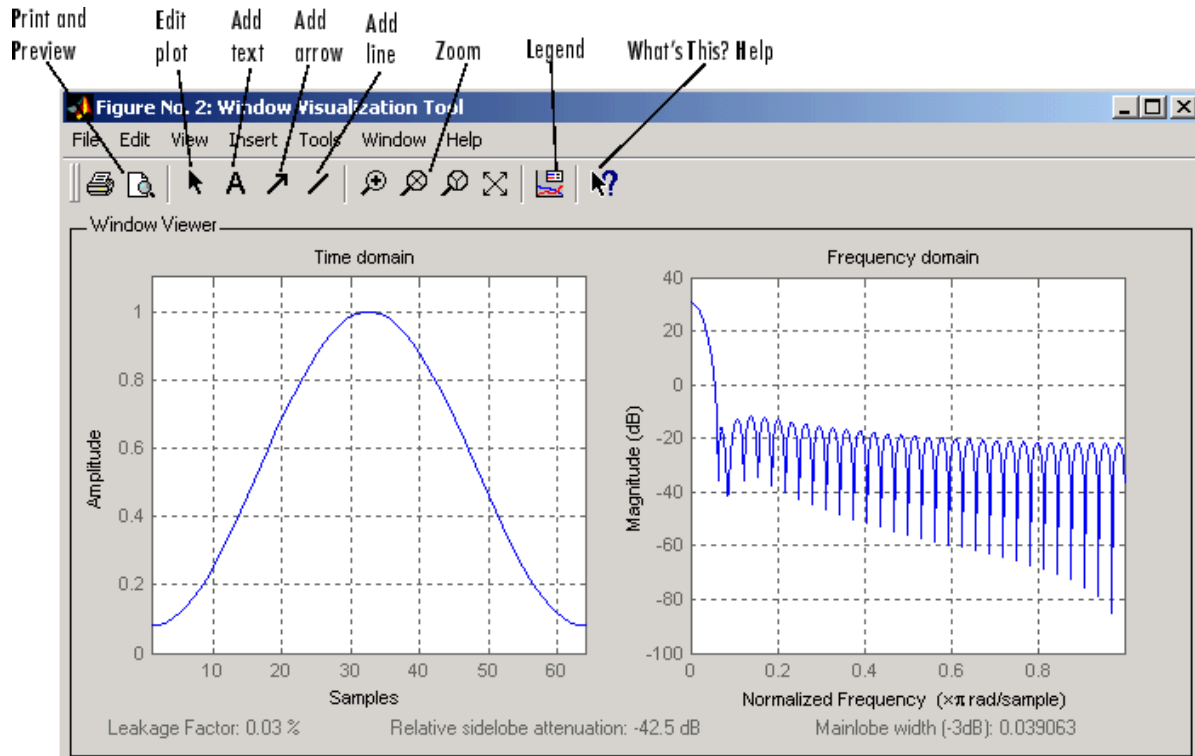
`wvtool(WindowVector1,...,WindowVectorN)` opens WVTool with time and frequency domain plots of the window vectors specified in `WindowVector1`, ..., `WindowVectorN`.

`H = wvtool(...)` returns the 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 menu items, wvtool contains these wvtool-specific menu commands:

**File** menu:

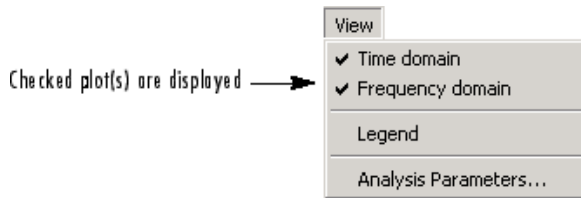
- **Export** — Exports the displayed plot(s) to a graphic file.

**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.

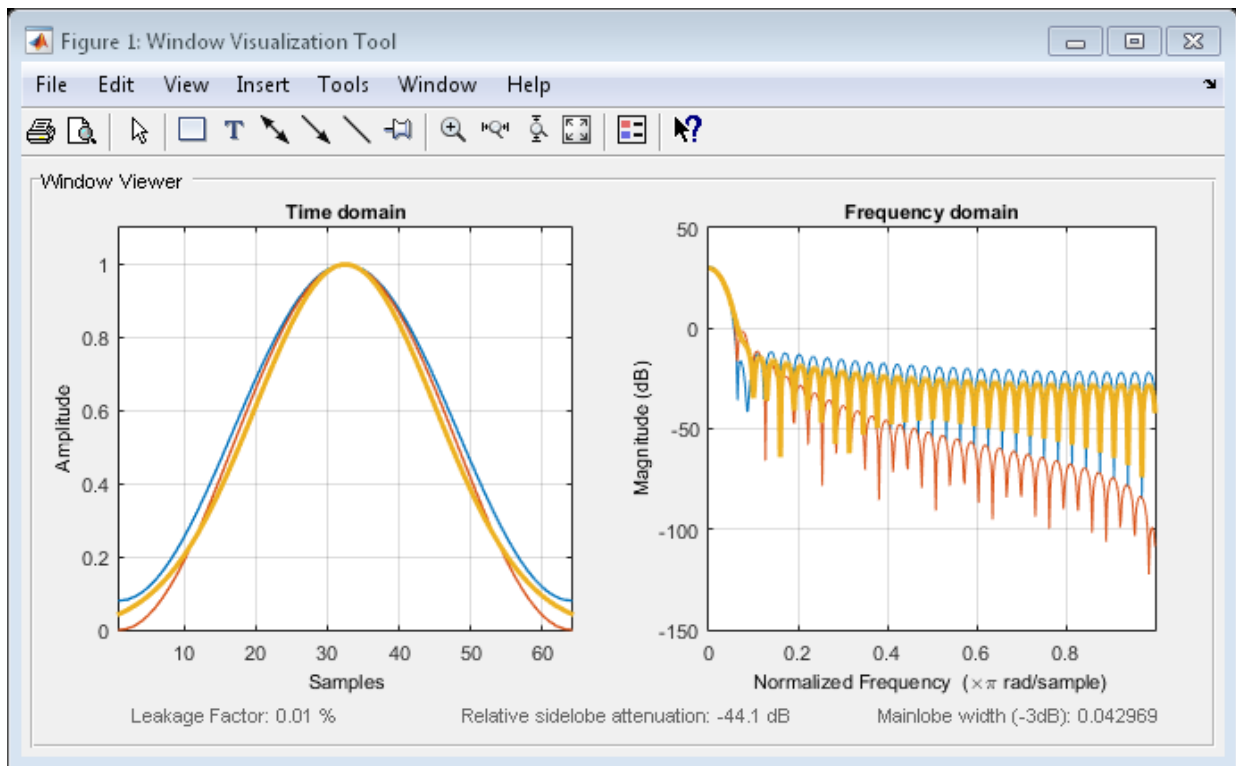


# Examples

## Display and Compare Windows

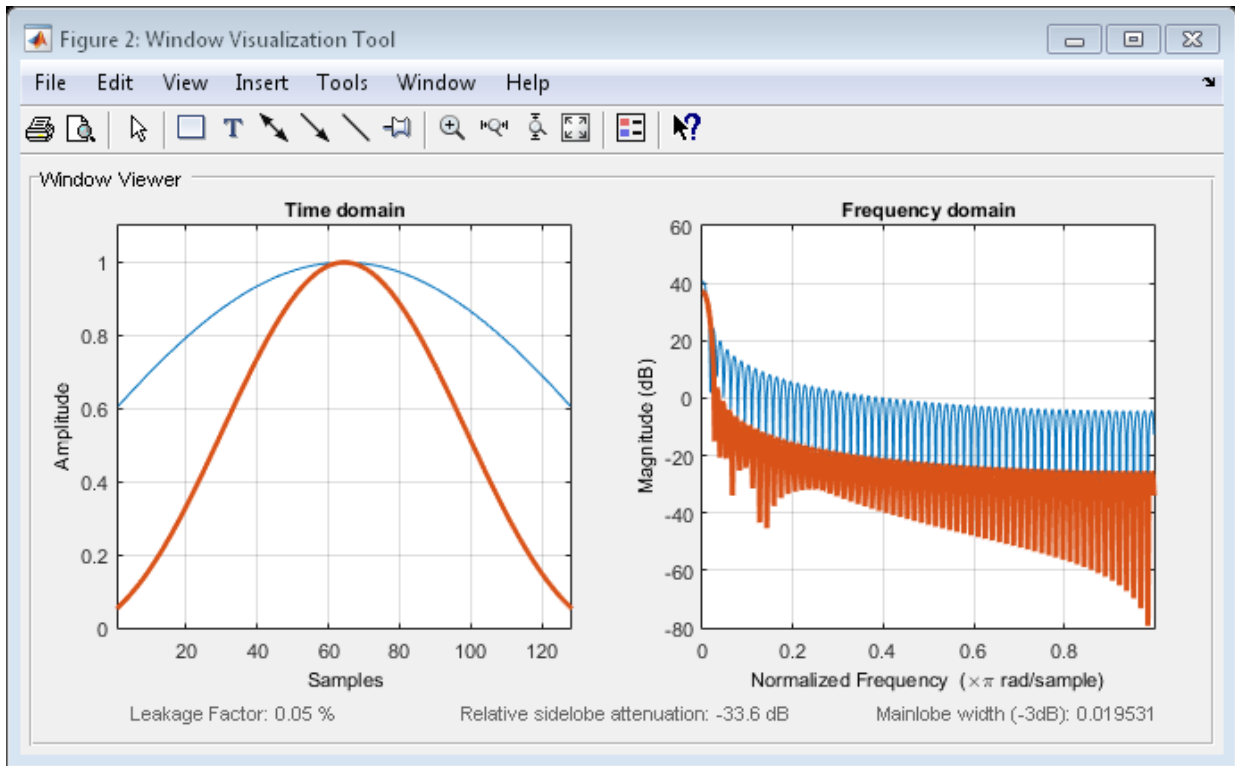
Use `wvtool` to display and compare 64-point Hamming, Hann, and Gaussian windows.

```
wvtool(hamming(64),hann(64),gausswin(64))
```



Compare 128-point Kaiser windows with different values of  $\beta$ .

```
wvtool(kaiser(128,1.5),kaiser(128,4.5))
```



## See Also

`fdatool` | `window` | `wintool`

## xcorr

Cross-correlation

### Syntax

```
r = xcorr(x,y)
```

```
r = xcorr(x)
```

```
r = xcorr( ____,maxlag)
```

```
r = xcorr( ____,scaleopt)
```

```
[r,lags] = xcorr( ____ )
```

### Description

`r = xcorr(x,y)` returns the cross-correlation of two discrete-time sequences, `x` and `y`. Cross-correlation measures the similarity between `x` and shifted (lagged) copies of `y` as a function of the lag. If `x` and `y` have different lengths, the function appends zeros at the end of the shorter vector so it has the same length,  $N$ , as the other.

`r = xcorr(x)` returns the autocorrelation sequence of `x`. If `x` is a matrix, then `r` is a matrix whose columns contain the autocorrelation and cross-correlation sequences for all combinations of the columns of `x`.

`r = xcorr( ____,maxlag)` limits the lag range from  $-\text{maxlag}$  to  $\text{maxlag}$ . This syntax accepts one or two input sequences. `maxlag` defaults to  $N - 1$ .

`r = xcorr( ____,scaleopt)` additionally specifies a normalization option for the cross-correlation or autocorrelation. Any option other than `'none'` (the default) requires `x` and `y` to have the same length.

`[r,lags] = xcorr( ____ )` also returns a vector with the lags at which the correlations are computed.

## Examples

### Delay Between Two Correlated Signals

Two sensors at different locations measure vibrations caused by a car as it crosses a bridge.

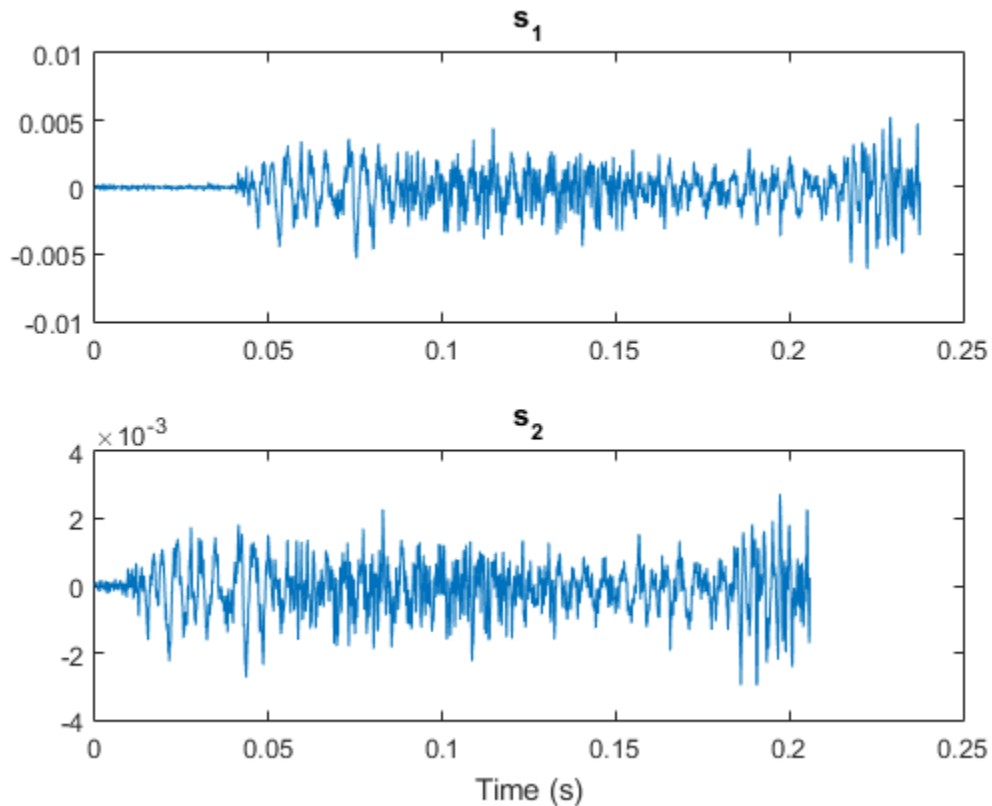
Load the signals and the sample rate,  $F_s = 11025$  Hz. Create time vectors and plot the signals. The signal from Sensor 2 arrives at an earlier time than the signal from Sensor 1.

```
load sensorData

t1 = (0:length(s1)-1)/Fs;
t2 = (0:length(s2)-1)/Fs;

subplot(2,1,1)
plot(t1,s1)
title('s_1')

subplot(2,1,2)
plot(t2,s2)
title('s_2')
xlabel('Time (s)')
```



The cross-correlation of the two measurements is maximum at a lag equal to the delay.

Plot the cross-correlation. Express the delay as a number of samples and in seconds.

```
[acor,lag] = xcorr(s2,s1);

[~,I] = max(abs(acor));
lagDiff = lag(I)
timeDiff = lagDiff/Fs

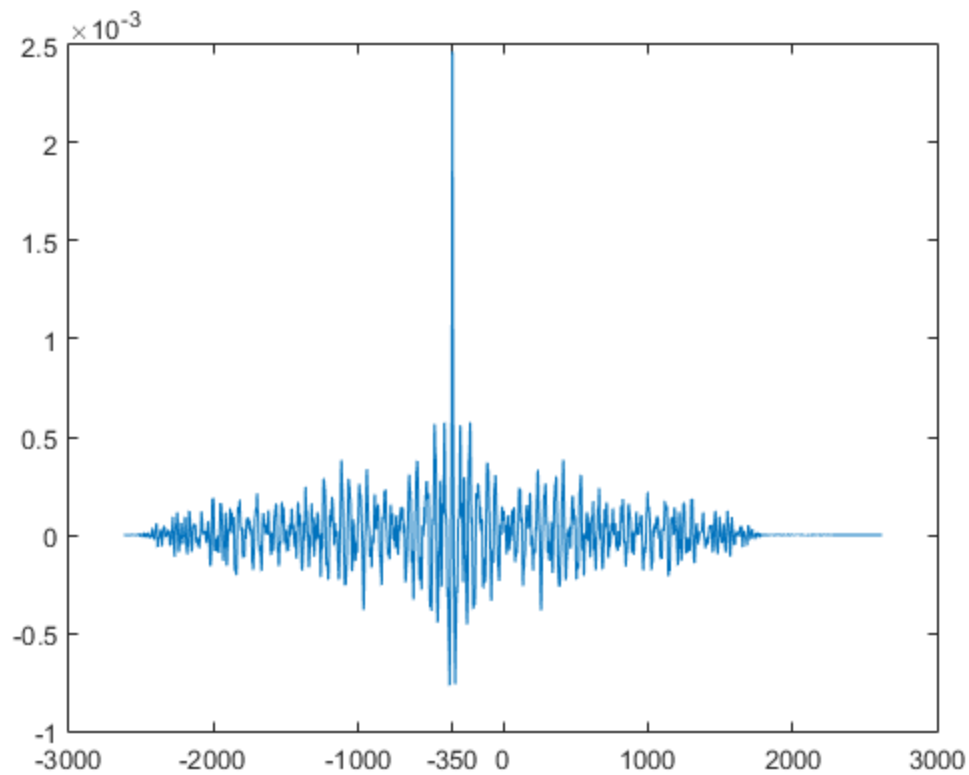
figure
plot(lag,acor)
a3 = gca;
a3.XTick = sort([-3000:1000:3000 lagDiff]);
```

```
lagDiff =
```

```
-350
```

```
timeDiff =
```

```
-0.0317
```

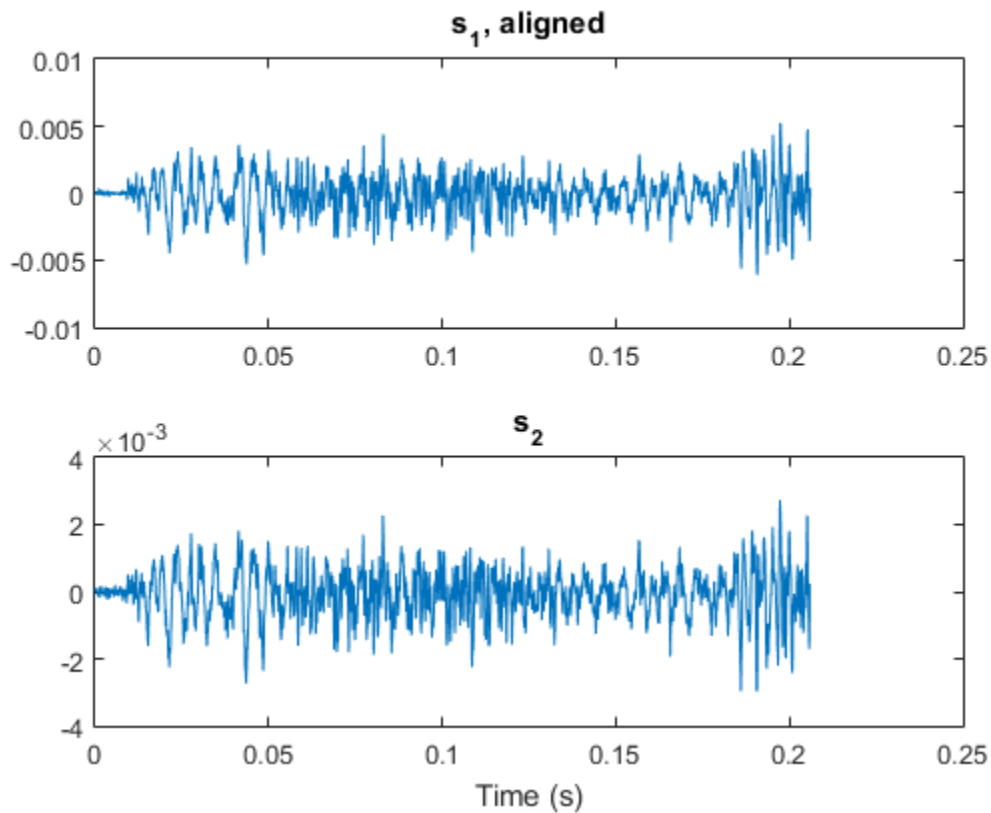


Align the two signals and replot them.

```
s1a1 = s1(-lagDiff:end);  
t1a1 = (0:length(s1a1)-1)/Fs;
```

```
subplot(2,1,1)
plot(t1a1,s1a1)
title('s_1, aligned')

subplot(2,1,2)
plot(t2,s2)
title('s_2')
xlabel('Time (s)')
```



### Echo Cancellation

A speech recording includes an echo caused by reflection off a wall. Use autocorrelation to filter it out.

In the recording, a person says the word MATLAB®. Load the data and the sample rate,  $F_s = 7418$  Hz.

```
load mtlb
```

```
% To hear, type soundsc(mtlb,Fs)
```

Model the echo by adding to the recording a copy of the signal delayed by  $\Delta$  samples and attenuated by a known factor  $\alpha$ :  $y(n) = x(n) + \alpha x(n - \Delta)$ . Specify a time lag of 0.23 s and an attenuation factor of 0.5.

```
timelag = 0.23;
delta = round(Fs*timelag);
alpha = 0.5;

orig = [mtlb;zeros(delta,1)];
echo = [zeros(delta,1);mtlb]*alpha;

mtEcho = orig + echo;
```

Plot the original, the echo, and the resulting signal.

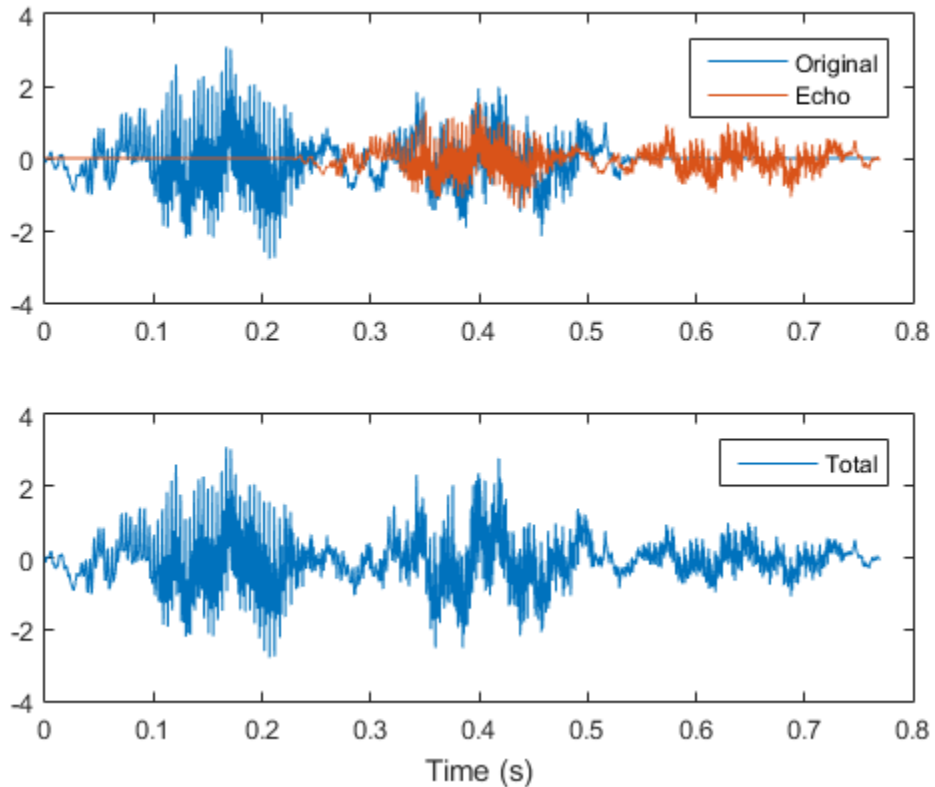
```
t = (0:length(mtEcho)-1)/Fs;

subplot(2,1,1)
plot(t,[orig echo])
legend('Original','Echo')

subplot(2,1,2)
plot(t,mtEcho)
legend('Total')
xlabel('Time (s)')
```

```
% To hear, type soundsc(mtEcho,Fs)
```



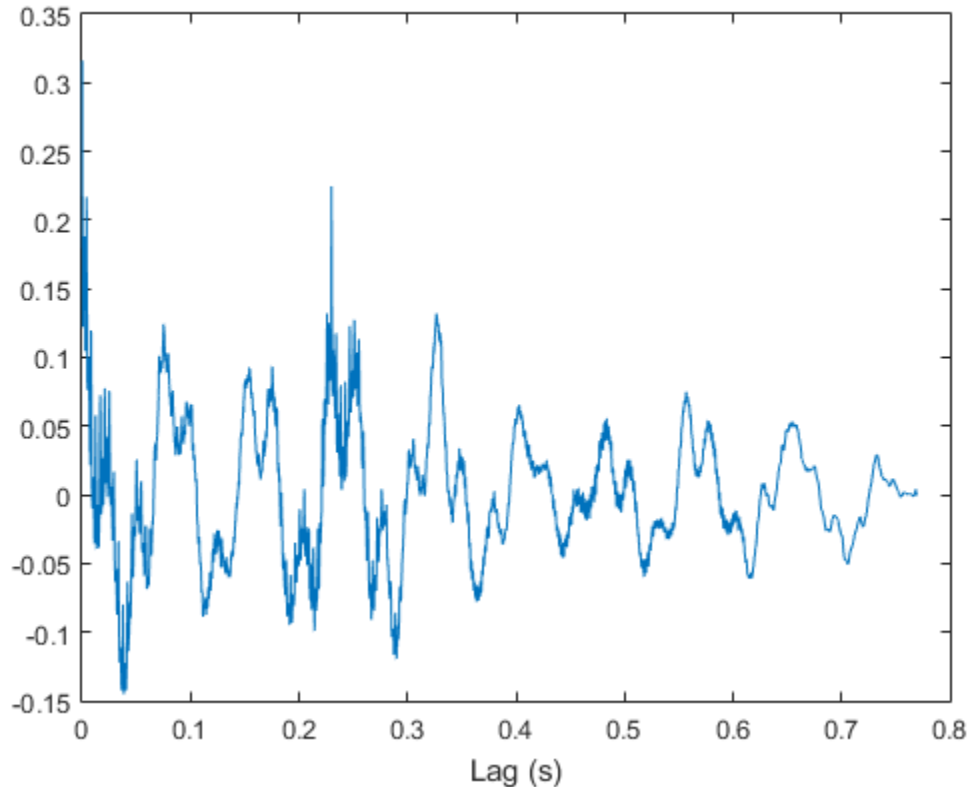


Compute an unbiased estimate of the signal autocorrelation. Select and plot the section that corresponds to lags greater than zero.

```
[Rmm,lags] = xcorr(mtEcho,'unbiased');
```

```
Rmm = Rmm(lags>0);  
lags = lags(lags>0);
```

```
figure  
plot(lags/Fs,Rmm)  
xlabel('Lag (s)')
```



The autocorrelation has a sharp peak at the lag at which the echo arrives. Cancel the echo by filtering the signal through an IIR system whose output,  $w$ , obeys  $w(n) + \alpha w(n - \Delta) = y(n)$ .

```
[~,d1] = findpeaks(Rmm,lags,'MinPeakHeight',0.22);
mtNew = filter(1,[1 zeros(1,d1-1) alpha],mtEcho);
```

Plot the filtered signal and compare to the original.

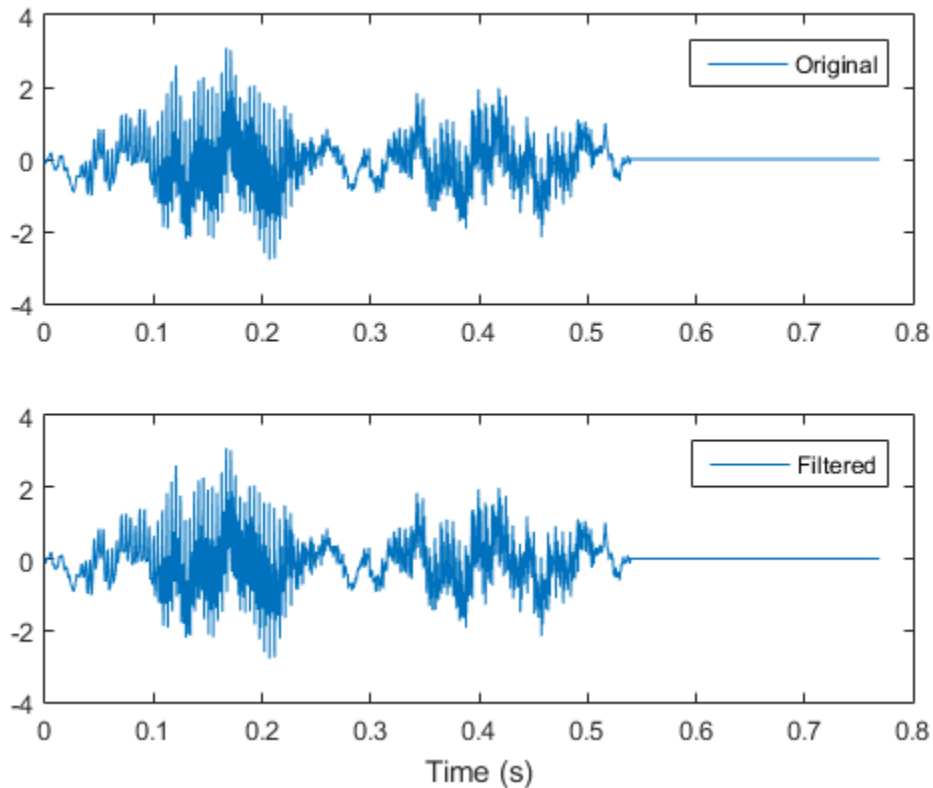
```
subplot(2,1,1)
plot(t,orig)
legend('Original')
```

```

subplot(2,1,2)
plot(t,mtNew)
legend('Filtered')
xlabel('Time (s)')

% To hear, type soundsc(mtNew,Fs)

```

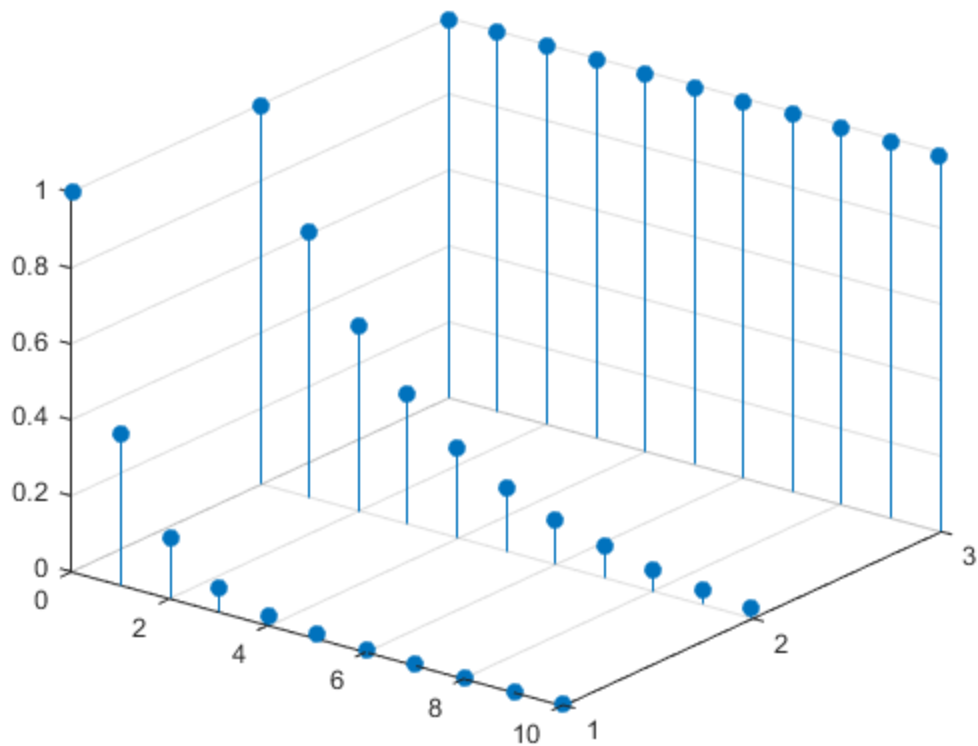


### Cross-Correlation with Multichannel Input

Generate three 11-sample exponential sequences given by  $0.4^n$ ,  $0.7^n$ , and  $0.999^n$ , with  $n \geq 0$ . Use `stem3` to plot the sequences side by side.

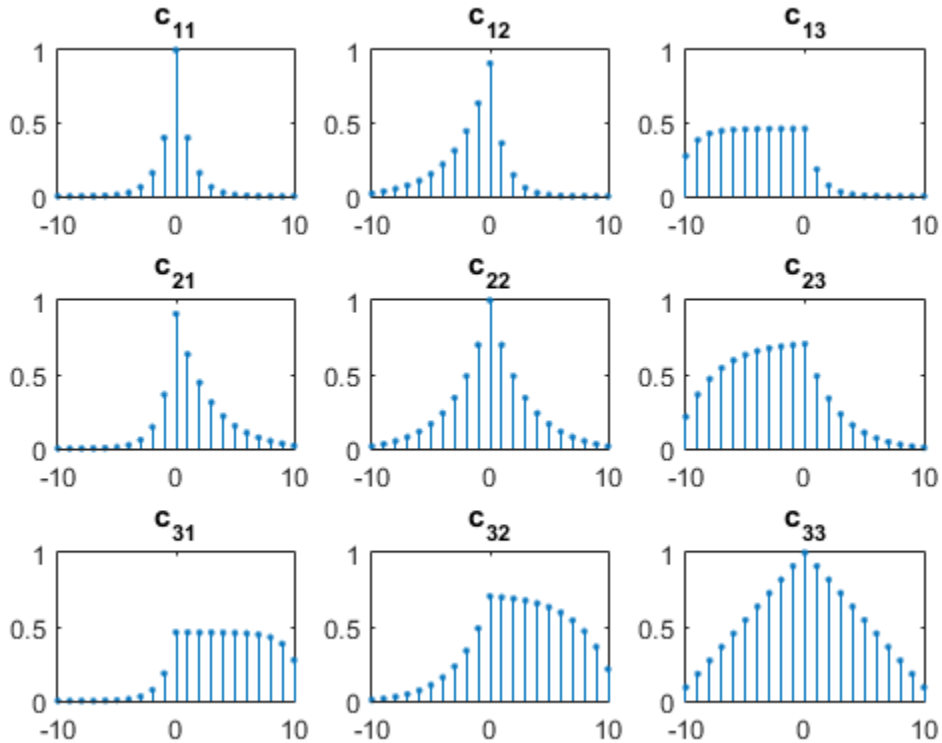
```
N = 11;
```

```
n = (0:N-1)';  
  
a = 0.4;  
b = 0.7;  
c = 0.999;  
  
xabc = [a.^n b.^n c.^n];  
  
stem3(n,1:3,xabc','filled')  
ax = gca;  
ax.YTick = 1:3;  
view(37.5,30)
```



Compute the autocorrelations and mutual cross-correlations of the sequences. Output the lags so you do not have to keep track of them. Normalize the result so the autocorrelations have unit value at zero lag.

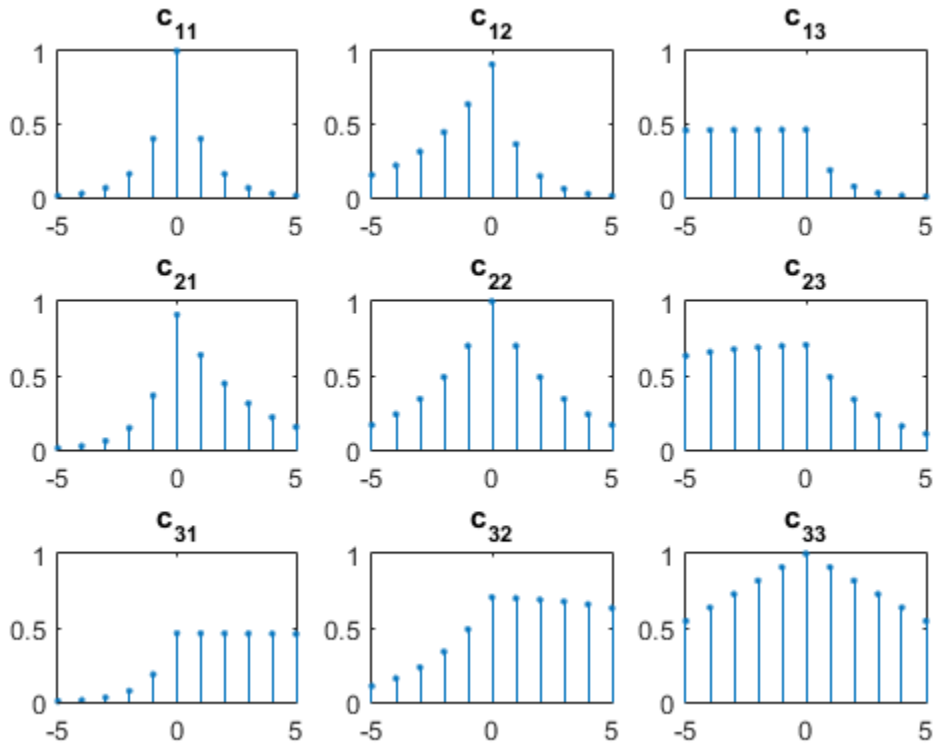
```
[cr,lgs] = xcorr(xabc, 'coeff');  
  
for row = 1:3  
    for col = 1:3  
        nm = 3*(row-1)+col;  
        subplot(3,3,nm)  
        stem(lgs,cr(:,nm),'.')  
        title(sprintf('c_{%d%d}',row,col))  
        ylim([0 1])  
    end  
end
```



Restrict the calculation to lags between  $-5$  and  $5$ .

```
[cr,lgs] = xcorr(xabc,5,'coeff');

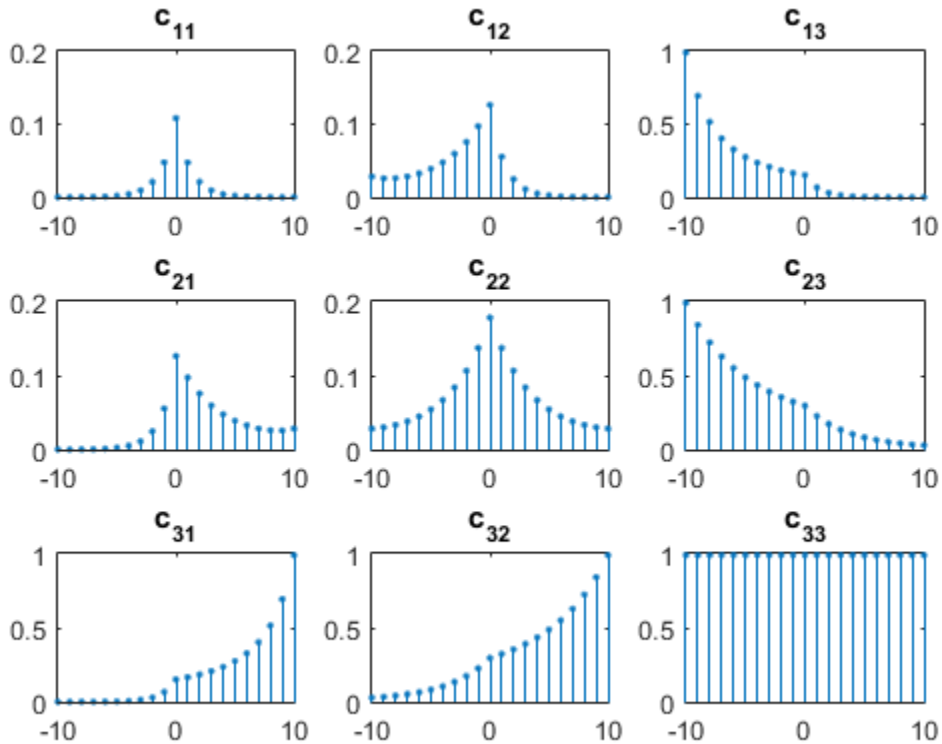
for row = 1:3
    for col = 1:3
        nm = 3*(row-1)+col;
        subplot(3,3,nm)
        stem(lgs,cr(:,nm),'.')
        title(sprintf('c_{%d%d}',row,col))
        ylim([0 1])
    end
end
```



Compute unbiased estimates of the autocorrelations and mutual cross-correlations. By default, the lags run between  $-(N-1)$  and  $N-1$ .

```
cu = xcorr(xabc, 'unbiased');

for row = 1:3
    for col = 1:3
        nm = 3*(row-1)+col;
        subplot(3,3,nm)
        stem(-(N-1):(N-1),cu(:,nm),'.')
        title(sprintf('c_{%d%d}',row,col))
    end
end
```



### Autocorrelation Function of Exponential Sequence

Compute the autocorrelation function of a 28-sample exponential sequence,  $x = 0.95^n$  for  $n \geq 0$ .

```
a = 0.95;
```

```
N = 28;
```

```
n = 0:N-1;
```

```
lags = -(N-1):(N-1);
```

```
x = a.^n;
```

```
c = xcorr(x);
```



Determine  $c$  analytically to check the correctness of the result. Use a larger sample rate to simulate a continuous situation. The autocorrelation function of the sequence  $x(n) = a^n$  for  $n \geq 0$ , with  $|a| < 1$ , is

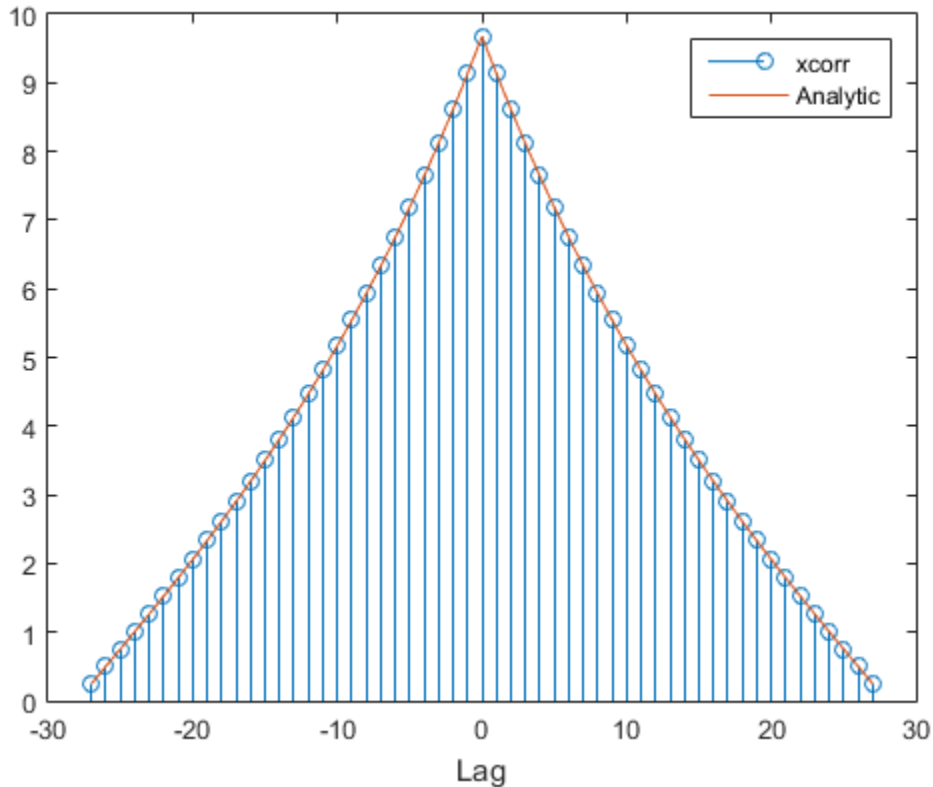
$$c(n) = \frac{1 - a^{2(N-|n|)}}{1 - a^2} \times a^{|n|}.$$

```
fs = 10;
nn = -(N-1):1/fs:(N-1);
cc = a.^abs(nn)/(1-a^2);

dd = (1-a.^(2*(N-abs(nn))))/(1-a^2).*a.^abs(nn);
```

Plot the sequences on the same figure.

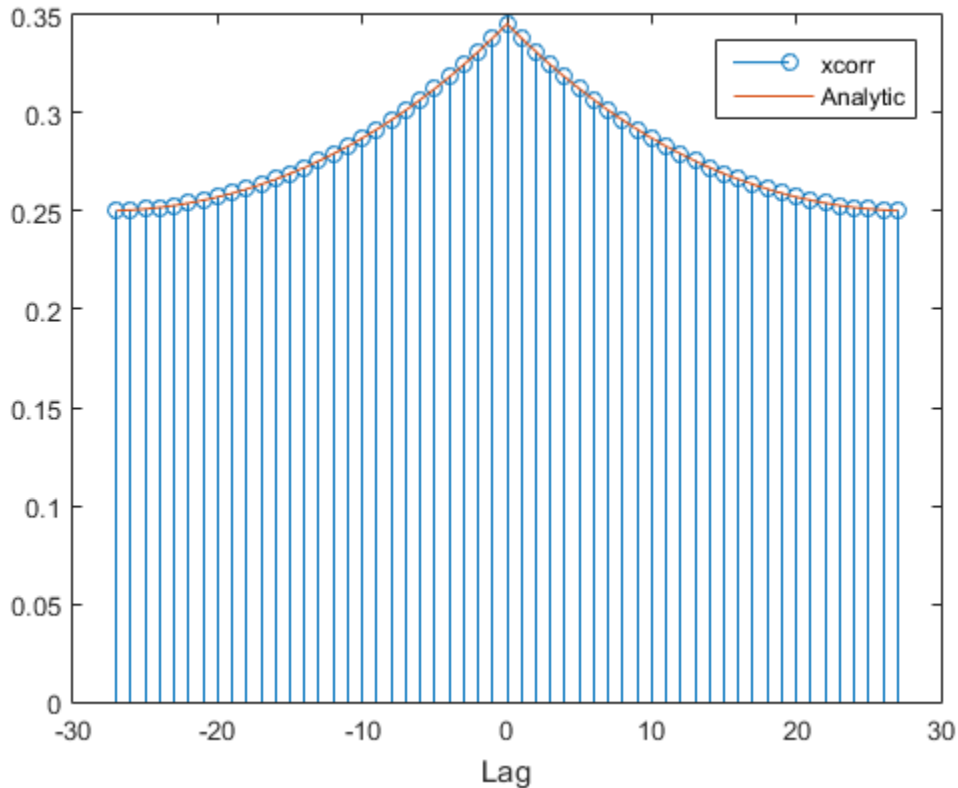
```
stem(lags,c);
hold on
plot(nn,dd)
xlabel('Lag')
legend('xcorr','Analytic')
hold off
```



Repeat the calculation, but now find an *unbiased* estimate of the autocorrelation. Verify that the unbiased estimate is given by  $c_u(n) = c(n)/(N - |n|)$ .

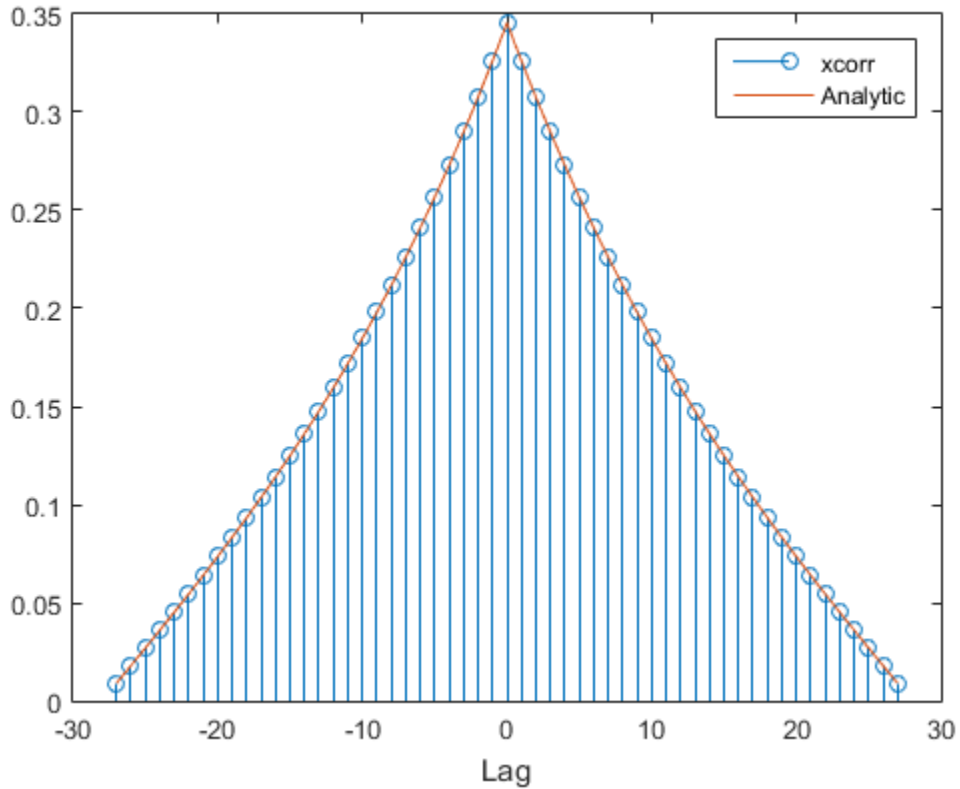
```
cu = xcorr(x, 'unbiased');
du = dd./(N-abs(nn));

stem(lags,cu);
hold on
plot(nn,du)
xlabel('Lag')
legend('xcorr','Analytic')
hold off
```



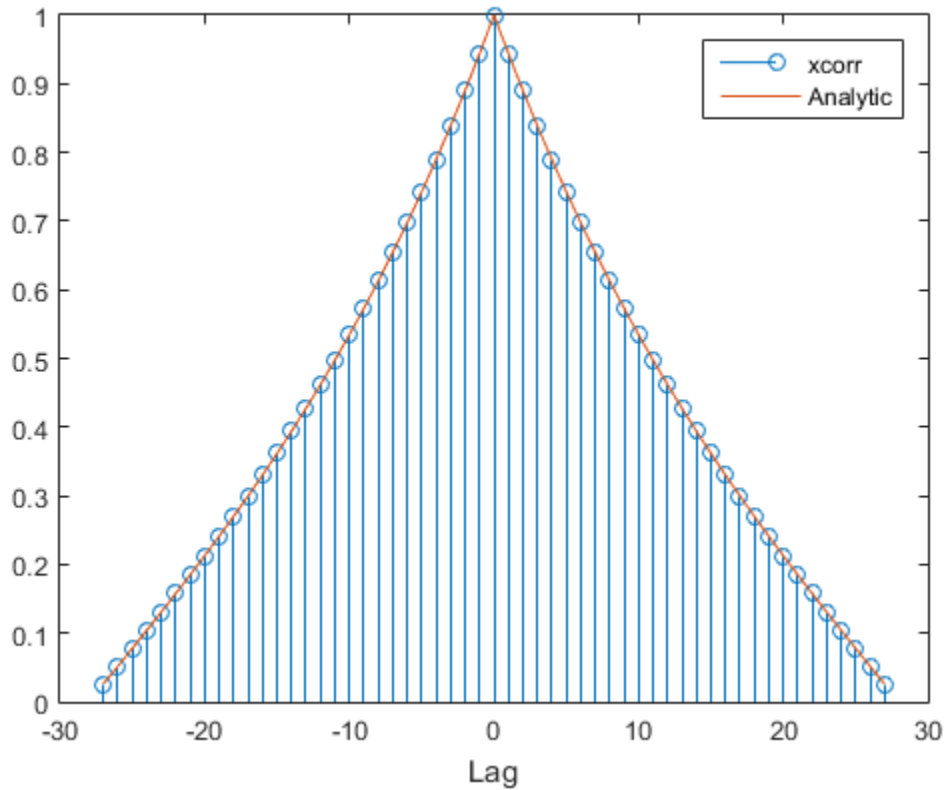
Repeat the calculation, but now find a *biased* estimate of the autocorrelation. Verify that the biased estimate is given by  $c_b(n) = c(n)/N$ .

```
cb = xcorr(x, 'biased');
db = dd/N;
stem(lags,cb);
hold on
plot(nn,db)
xlabel('Lag')
legend('xcorr','Analytic')
hold off
```



Find an estimate of the autocorrelation whose value at zero lag is unity.

```
cz = xcorr(x, 'coeff');  
  
dz = dd/max(dd);  
  
stem(lags,cz);  
hold on  
plot(nn,dz)  
xlabel('Lag')  
legend('xcorr','Analytic')  
hold off
```



### Cross-Correlation of Two Exponential Sequences

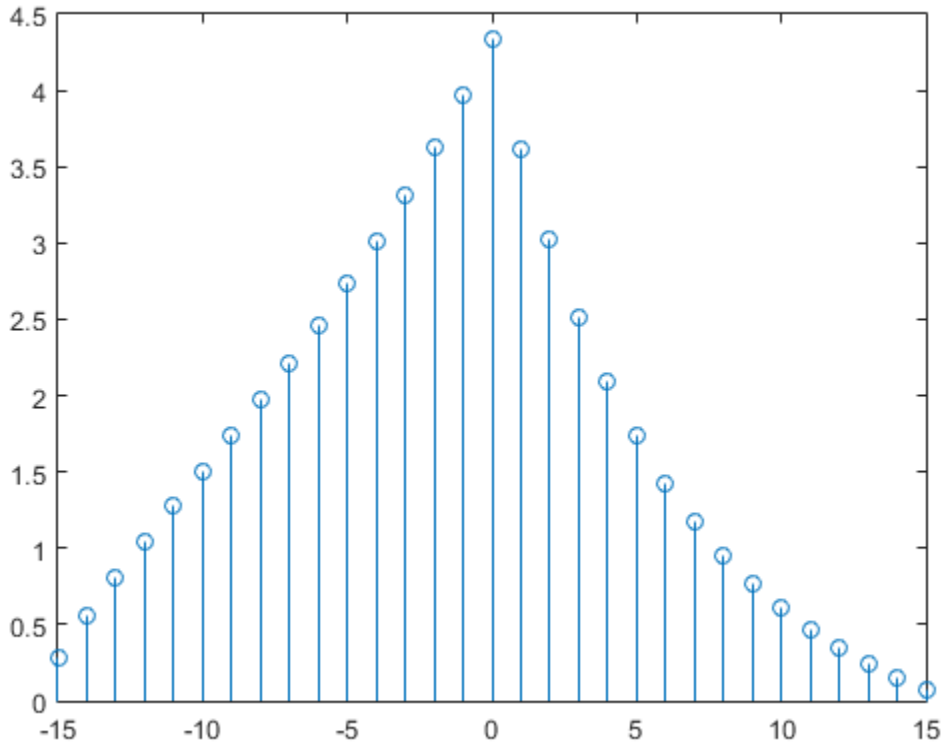
Compute and plot the cross-correlation of two 16-sample exponential sequences,  $x_a = 0.84^n$  and  $x_b = 0.92^n$ , with  $n \geq 0$ .

```
N = 16;  
n = 0:N-1;
```

```
a = 0.84;  
b = 0.92;
```

```
xa = a.^n;  
xb = b.^n;
```

```
r = xcorr(xa,xb);
stem(-(N-1):(N-1),r)
```



Determine  $c$  analytically to check the correctness of the result. Use a larger sample rate to simulate a continuous situation. The cross-correlation function of the sequences  $x_a(n) = a^n$  and  $x_b(n) = b^n$  for  $n \geq 0$ , with  $0 < a, b < 1$ , is

$$c_{ab}(n) = \frac{1 - (ab)^{N-|n|}}{1 - ab} \times \begin{cases} a^n, & n > 0, \\ 1, & n = 0, \\ b^{-n}, & n < 0. \end{cases}$$

```
fs = 10;
nn = -(N-1):1/fs:(N-1);
```

```

cn = (1 - (a*b).^(N-abs(nn)))/(1 - a*b) .* ...
      (a.^nn.*(nn>0) + (nn==0) + b.^-(nn).*(nn<0));

```

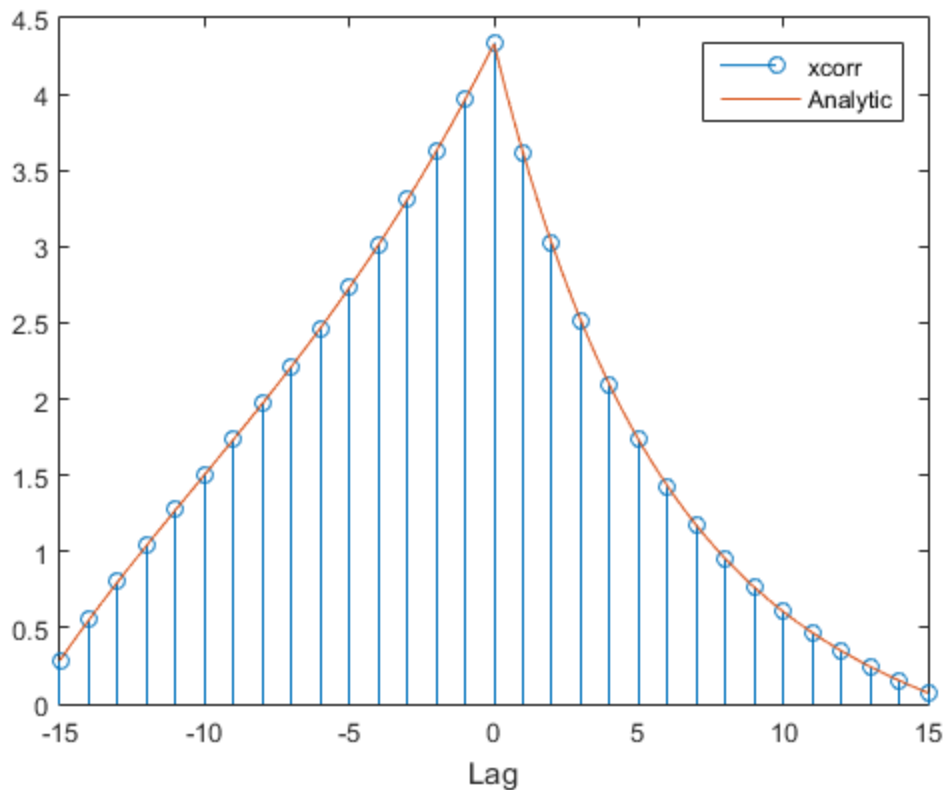
Plot the sequences on the same figure.

```

hold on
plot(nn,cn)

xlabel('Lag')
legend('xcorr','Analytic')

```



Verify that switching the order of the operands reverses the sequence.

figure

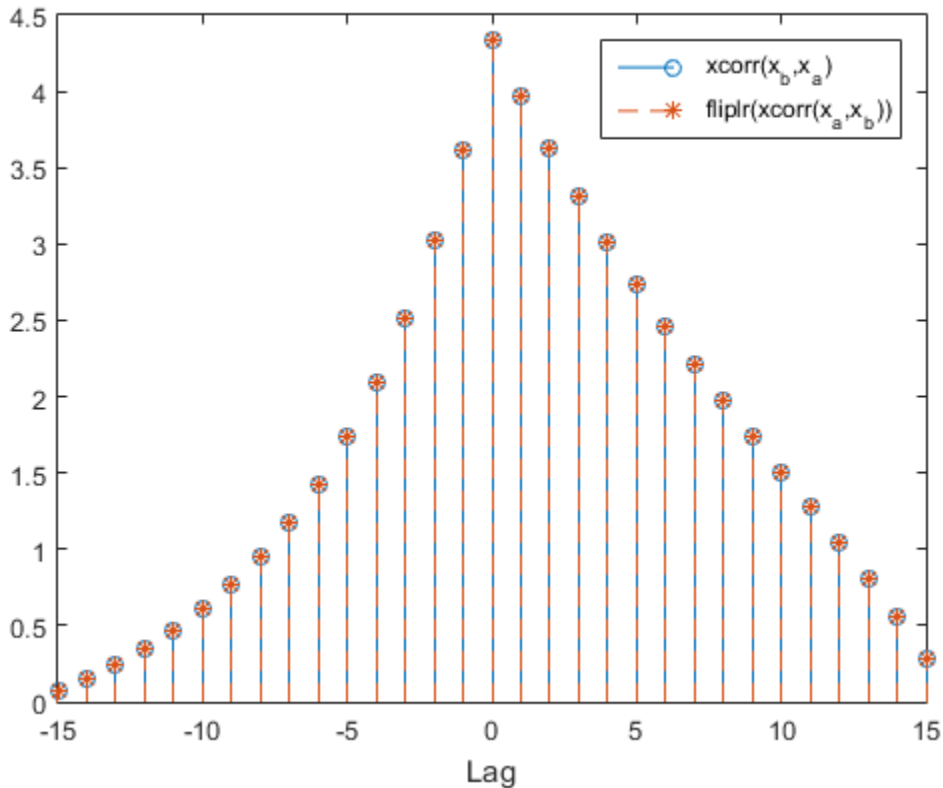
```

stem(-(N-1):(N-1),xcorr(xb,xa))

hold on
stem(-(N-1):(N-1),fliplr(r),'--*')

xlabel('Lag')
legend('xcorr(x_b,x_a)','fliplr(xcorr(x_a,x_b))')

```



Generate the 20-sample exponential sequence  $x_c = 0.77^n$ . Compute and plot its cross-correlations with  $x_a$  and  $x_b$ . Output the lags to make the plotting easier. `xcorr` appends zeros at the end of the shorter sequence to match the length of the longer one.

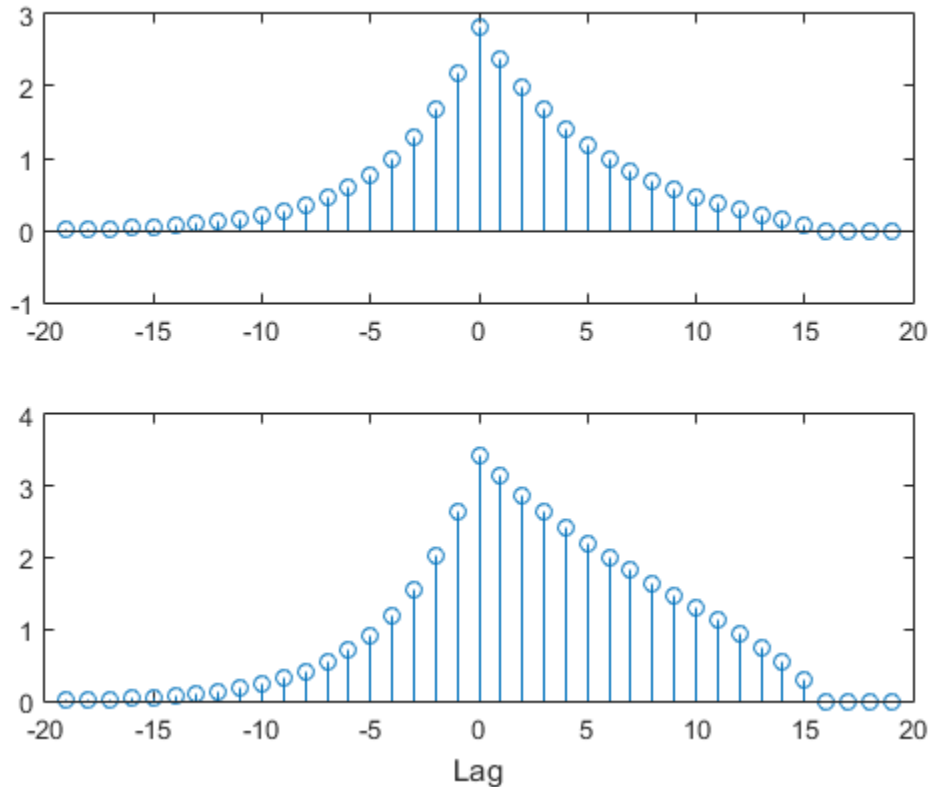
```
xc = 0.77.^(0:20-1);
```



```
[xca,la] = xcorr(xa,xc);  
[xcb,lb] = xcorr(xb,xc);
```

```
figure
```

```
subplot(2,1,1)  
stem(la,xca)  
subplot(2,1,2)  
stem(lb,xcb)  
xlabel('Lag')
```



### GPU Acceleration for Autocorrelation Sequence Estimation

This example requires Parallel Computing Toolbox software and a CUDA-enabled NVIDIA GPU with compute capability 1.3 or above. See GPU System Requirements for details.

Create a signal consisting of a 10 Hz sine wave in additive noise, sampled at 1 kHz. Use `gpuArray` to create a `gpuArray` object stored on your computer's GPU.

```
t = 0:0.001:10-0.001;
x = cos(2*pi*10*t) + randn(size(t));
X = gpuArray(x);
```

Compute the normalized autocorrelation sequence to lag 200.

```
[xc,lags] = xcorr(X,200,'coeff');
```

The output, `xc`, is a `gpuArray` object.

Use `gather` to transfer the data from the GPU to the MATLAB workspace as a double-precision vector.

```
xc = gather(xc);
```

- “Find a Signal in a Measurement”
- “Measuring Signal Similarities”
- “Accelerating Correlation with GPUs”

## Input Arguments

### **x** — Input array

vector | matrix | `gpuArray` object

Input array, specified as a vector, a matrix, or a `gpuArray` object.

See “GPU Computing” and GPU System Requirements for details on using `xcorr` with `gpuArray` objects.

Example: `sin(2*pi*(0:9)/10) + randn([1 10])/10` specifies a noisy sinusoid as a row vector.

Example: `sin(2*pi*[0.1;0.3]*(0:39))' + randn([40 2])/10` specifies a two-channel noisy sinusoid.

Example: `gpuArray(sin(2*pi*(0:9)/10) + randn([1 10])/10)` specifies a noisy sinusoid as a `gpuArray` object.

Data Types: `single` | `double`

Complex Number Support: Yes

### **y** — Input array

vector | `gpuArray` object

Input array, specified as a vector or a `gpuArray` object.

Data Types: `single` | `double`

Complex Number Support: Yes

**maxlag** — Maximum lag

integer scalar

Maximum lag, specified as an integer scalar. If you specify `maxlag`, the returned cross-correlation sequence ranges from  $-\text{maxlag}$  to  $\text{maxlag}$ . If you do not specify `maxlag`, the lag range equals  $2N - 1$ , where  $N$  is the greater of the lengths of  $x$  and  $y$ .

Data Types: `single` | `double`**scaleopt** — Normalization option

'none' (default) | 'biased' | 'unbiased' | 'coeff'

Normalization option, specified as one of these strings:

- 'none' — Raw, unscaled cross-correlation. This is the only allowed option when  $x$  and  $y$  have different lengths.
- 'biased' — Biased estimate of the cross-correlation:

$$\hat{R}_{xy,\text{biased}}(m) = \frac{1}{N} \hat{R}_{xy}(m).$$

- 'unbiased' — Unbiased estimate of the cross-correlation:

$$\hat{R}_{xy,\text{unbiased}}(m) = \frac{1}{N - |m|} \hat{R}_{xy}(m).$$

- 'coeff' — Normalizes the sequence so that the autocorrelations at zero lag equal 1.

Data Types: `char`

## Output Arguments

**r** — Cross-correlation or autocorrelation sequencevector | matrix | `gpuArray` objectCross-correlation sequence, returned as a vector, a matrix, or a `gpuArray` object.

If  $x$  is an  $M \times N$  signal matrix representing  $N$  channels in its columns, then `xcorr(x)` returns a  $(2M - 1) \times N^2$  matrix with the autocorrelations and mutual cross-correlations of the channels of  $x$ . If you specify `maxlag`, then  $r$  has size  $(2 \times \text{maxlag} - 1) \times N^2$ .

For example, if  $\mathbf{S}$  is a three-channel signal,  $\mathbf{S} = (x_1 \ x_2 \ x_3)$ , then the result of  $\mathbf{R} = \text{xcorr}(\mathbf{S})$  is organized as

$$\mathbf{R} = \begin{pmatrix} R_{x_1 x_1} & R_{x_1 x_2} & R_{x_1 x_3} & R_{x_2 x_1} & R_{x_2 x_2} & R_{x_2 x_3} & R_{x_3 x_1} & R_{x_3 x_2} & R_{x_3 x_3} \end{pmatrix}.$$

### lags — Lag indices

vector

Lag indices, returned as a vector.

## More About

### Cross-Correlation and Autocorrelation

The result of `xcorr` can be interpreted as an estimate of the correlation between two random sequences or as the deterministic correlation between two deterministic signals.

The true cross-correlation sequence of two jointly stationary random processes,  $x_n$  and  $y_n$ , is given by

$$R_{xy}(m) = E\{x_{n+m}y_n^*\} = E\{x_n y_{n-m}^*\},$$

where  $-\infty < n < \infty$ , the asterisk denotes complex conjugation, and  $E$  is the expected value operator. `xcorr` can only estimate the sequence because, in practice, only a finite segment of one realization of the infinite-length random process is available.

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,  $\mathbf{c}$ , has elements given by

$$c(m) = R_{xy}(m - N), \quad m = 1, 2, \dots, 2N - 1.$$

In general, the correlation function requires normalization to produce an accurate estimate. You can control the normalization of the correlation by using the input argument `scaleopt`.

## References

- [1] Buck, John R., Michael M. Daniel, and Andrew C. Singer. *Computer Explorations in Signals and Systems Using MATLAB*. 2nd Edition. Upper Saddle River, NJ: Prentice Hall, 2002.
- [2] Stoica, Petre, and Randolph Moses. *Spectral Analysis of Signals*. Upper Saddle River, NJ: Prentice Hall, 2005.

## See Also

`conv` | `corrcoef` | `cov` | `xcorr2` | `xcov`

## xcorr2

2-D cross-correlation

### Syntax

```
C = xcorr2(A,B)
A = xcorr2(A)
C = xcorr2(gpuArrayA,gpuArrayB)
```

### 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`.

`A = xcorr2(A)` is the autocorrelation matrix of input matrix **A**. It is identical to `xcorr2(A,A)`.

`C = xcorr2(gpuArrayA,gpuArrayB)` returns the cross-correlation of two matrices of class `gpuArray`. The output cross-correlation matrix, **C**, is also a `gpuArray` object. See “Establish Arrays on a GPU” for details on `gpuArray` objects. Using `xcorr2` with `gpuArray` objects requires Parallel Computing Toolbox software and a CUDA-enabled NVIDIA GPU with compute capability 1.3 or above. See <http://www.mathworks.com/products/parallel-computing/requirements.html> for details. See “GPU Acceleration for Cross-Correlation Matrix Computation” on page 1-1766 for an example of using a GPU to compute the cross-correlation.

## 2-D Cross-Correlation

The 2-D cross-correlation of an  $M$ -by- $N$  matrix  $X$  and a  $P$ -by- $Q$  matrix  $H$  is a matrix  $C$  of size  $M+P-1$  by  $N+Q-1$  given by

$$C(k,l) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} X(m,n) \bar{H}(m-k,n-l), \quad \begin{array}{l} -(P-1) \leq k \leq M-1, \\ -(Q-1) \leq l \leq N-1, \end{array}$$

where the bar over  $H$  denotes complex conjugation.

The output matrix,  $C(k,l)$ , has negative and positive row and column indices. A negative row index corresponds to an upward shift of the rows of  $H$ . A negative column index corresponds to a leftward shift of the columns of  $H$ . A positive row index corresponds to a downward shift of the rows of  $H$ . A positive column index corresponds to a rightward shift of the columns. To cast the indices in MATLAB form, simply add the size of  $H$ : the element  $C(k,l)$  corresponds to  $C(k+P, l+Q)$  in the workspace.

For example, consider the following 2-D cross-correlation:

```
X = ones(2,3);
H = [1 2; 3 4; 5 6]; % H is 3 by 2
C = xcorr2(X,H)
```

```
C =
     6     11     11     5
    10     18     18     8
     6     10     10     4
     2     3      3     1
```

The  $C(1,1)$  element in the output corresponds to  $C(1-3,1-2) = C(-2,-1)$  in the defining equation, which uses zero-based indexing. The  $C(1,1)$  element is computed by shifting  $H$  two rows upward and one column to the left. Accordingly, the only product in the cross-correlation sum is  $X(1,1) * H(3,2) = 6$ . Using the defining equation, you obtain

$$C(-2,-1) = \sum_{m=0}^1 \sum_{n=0}^2 X(m,n) \bar{H}(m+2,n+1) = X(0,0) \bar{H}(2,1) = 1 \times 6 = 6,$$

with all other terms in the double sum equal to zero.

## 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:



$$C_{\text{full,rows}} = I1_{\text{rows}} + I2_{\text{rows}} - 1 = 4 + 2 - 1 = 5$$

$$C_{\text{full,columns}} = I1_{\text{columns}} + I2_{\text{columns}} - 1 = 3 + 2 - 1 = 4$$

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

$$C_{\text{valid,columns}} = I1_{\text{columns}} - I2_{\text{columns}} + 1 = 2$$

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

$$I1 = \begin{bmatrix} 17 & 24 & 1 & 8 & 15 \\ 23 & 5 & 7 & 14 & 16 \\ 4 & 6 & 13 & 20 & 22 \\ 10 & 12 & 19 & 21 & 3 \\ 11 & 18 & 25 & 2 & 9 \end{bmatrix}$$

The second input matrix also represents an image and is defined as

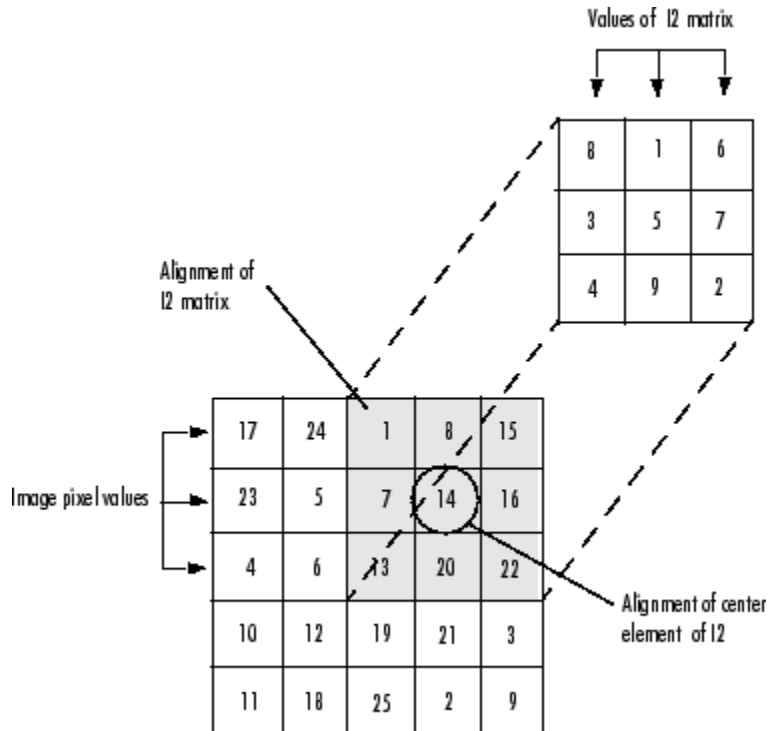
$$I2 = \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}$$

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

$$585/\text{sqrt}(\text{sum}(\text{dot}(I1p,I1p)) \cdot \text{sum}(\text{dot}(I2,I2))) = 0.8070$$

where  $I1p = [1 \ 8 \ 15; 7 \ 14 \ 16; 13 \ 20 \ 22]$ .

## Recovery of Template Shift with Cross-Correlation

Shift a template by a known amount and recover the shift using cross-correlation.

Create a template in an 11-by-11 matrix. Create a 22-by-22 matrix and shift the original template by 8 along the row dimension and 6 along the column dimension.

```

template = .2*ones(11);
template(6,3:9) = .6;
template(3:9,6) = .6;
offsetTemplate = .2*ones(22);
offset = [8 6];
offsetTemplate( (1:size(template,1))+offset(1),...
               (1:size(template,2))+offset(2) ) = template;

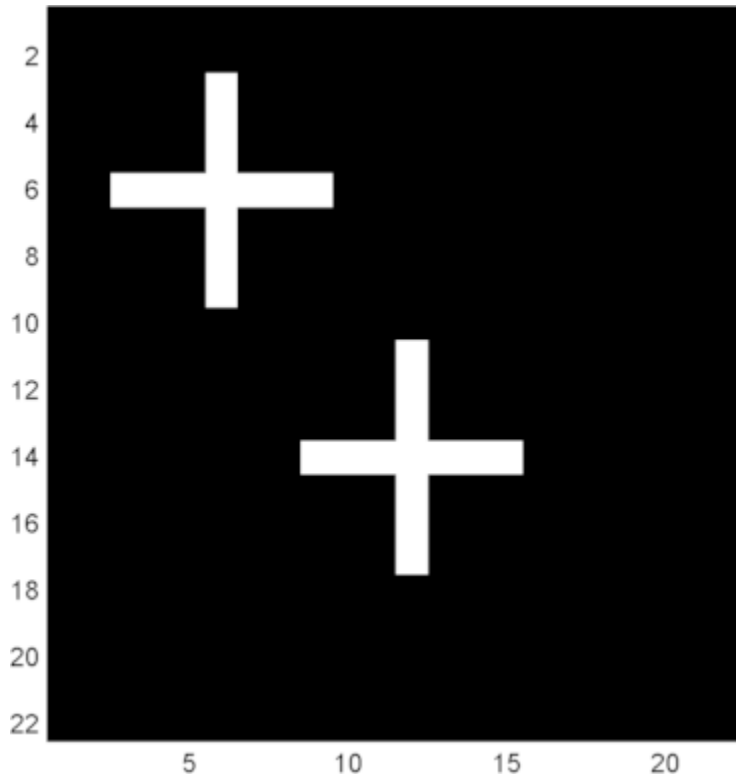
```

Plot the original and shifted templates.

```

imagesc(offsetTemplate); colormap gray;
hold on;
imagesc(template);

```



Cross-correlate the two matrices and find the maximum absolute value of the cross-correlation. Use the position of the maximum absolute value to determine the shift in the template. Check the result against the known shift.

```
cc = xcorr2(offsetTemplate,template);
[max_cc, imax] = max(abs(cc(:)));
[ypeak, xpeak] = ind2sub(size(cc),imax(1));
corr_offset = [ (ypeak-size(template,1)) (xpeak-size(template,2)) ];
isequal(corr_offset,offset)
```

The returned 1 indicates that the shift obtained the cross-correlation equals the known the template shift in both the row and column dimension.

## GPU Acceleration for Cross-Correlation Matrix Computation

The following example requires Parallel Computing Toolbox software and a CUDA-enabled NVIDIA GPU with compute capability 1.3 or above. See <http://www.mathworks.com/products/parallel-computing/requirements.html> for details.

Repeat the example “Recovery of Template Shift with Cross-Correlation” on page 1-1764. For convenience, the code to create the original and shifted templates is repeated.

```
template = .2*ones(11);
template(6,3:9) = .6;
template(3:9,6) = .6;
offsetTemplate = .2*ones(22);
offset = [8 6];
offsetTemplate( (1:size(template,1))+offset(1),...
               (1:size(template,2))+offset(2) ) = template;
```

Put the original and shifted template matrices on your GPU using `gpuArray` objects.

```
template = gpuArray(template);
offsetTemplate = gpuArray(offsetTemplate);
```

Compute the cross-correlation on the GPU.

```
cc = xcorr2(offsetTemplate,template);
```

Return the result to the MATLAB workspace using `gather`, use the maximum absolute value of the cross-correlation to determine the shift, and compare the result with the known shift.

```
cc = gather(cc);
[max_cc, imax] = max(abs(cc(:)));
[ypeak, xpeak] = ind2sub(size(cc),imax(1));
```

```
corr_offset = [ (ypeak-size(template,1)) (xpeak-size(template,2)) ];  
isequal(corr_offset,offset)
```

**See Also**

conv2 | filter2 | xcorr

## **xcov**

Cross-covariance

### **Syntax**

```
c = xcov(x,y)
c = xcov(x)
```

```
c = xcov( ____,maxlag)
c = xcov( ____,scaleopt)
```

```
[c,lags] = xcov( ____)
```

### **Description**

`c = xcov(x,y)` returns the cross-covariance of two discrete-time sequences, `x` and `y`. Cross-covariance measures the similarity between `x` and shifted (lagged) copies of `y` as a function of the lag. If `x` and `y` have different lengths, the function appends zeros at the end of the shorter vector so it has the same length as the other.

`c = xcov(x)` returns the autocovariance sequence of `x`. If `x` is a matrix, then `c` is a matrix whose columns contain the autocovariance and cross-covariance sequences for all combinations of the columns of `x`.

`c = xcov( ____,maxlag)` limits the lag range from  $-\text{maxlag}$  to  $\text{maxlag}$ . This syntax accepts one or two input sequences. `maxlag` defaults to  $N - 1$ .

`c = xcov( ____,scaleopt)` additionally specifies a normalization option for the cross-covariance or autocovariance. Any option other than `'none'` (the default) requires `x` and `y` to have the same length.

`[c,lags] = xcov( ____)` also outputs a vector with the lags at which the covariances are computed.

## Examples

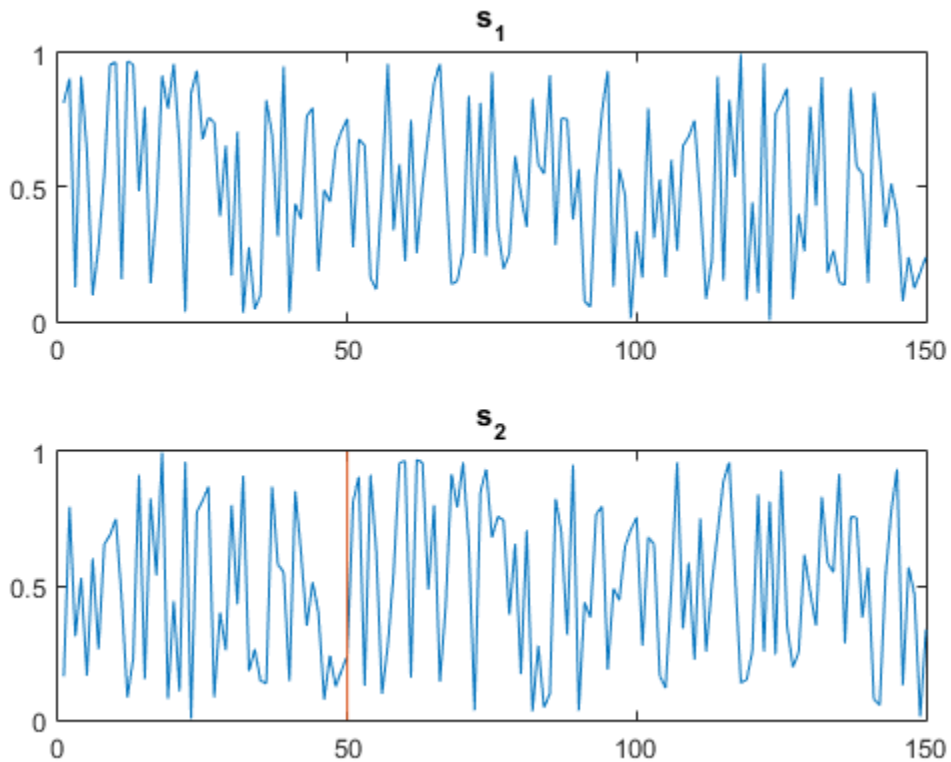
### Cross-Covariance of Two Shifted Signals

Create  $\mathbf{s}$ , a two-channel 150-sample signal consisting of  $\mathbf{s}_1$ , a uniform random sequence, and  $\mathbf{s}_2$ , a copy of  $\mathbf{s}_1$  shifted circularly by 50 samples. Reset the random number generator for reproducible results. Plot the sequences.

```
rng default
shft = 50;

s1 = rand(150,1);
s2 = circshift(s1,[shft 0]);
s = [s1 s2];

subplot(2,1,1)
plot(s1)
title('s_1')
subplot(2,1,2)
plot(s2)
title('s_2')
hold on
plot([shft shft],[0 1])
```

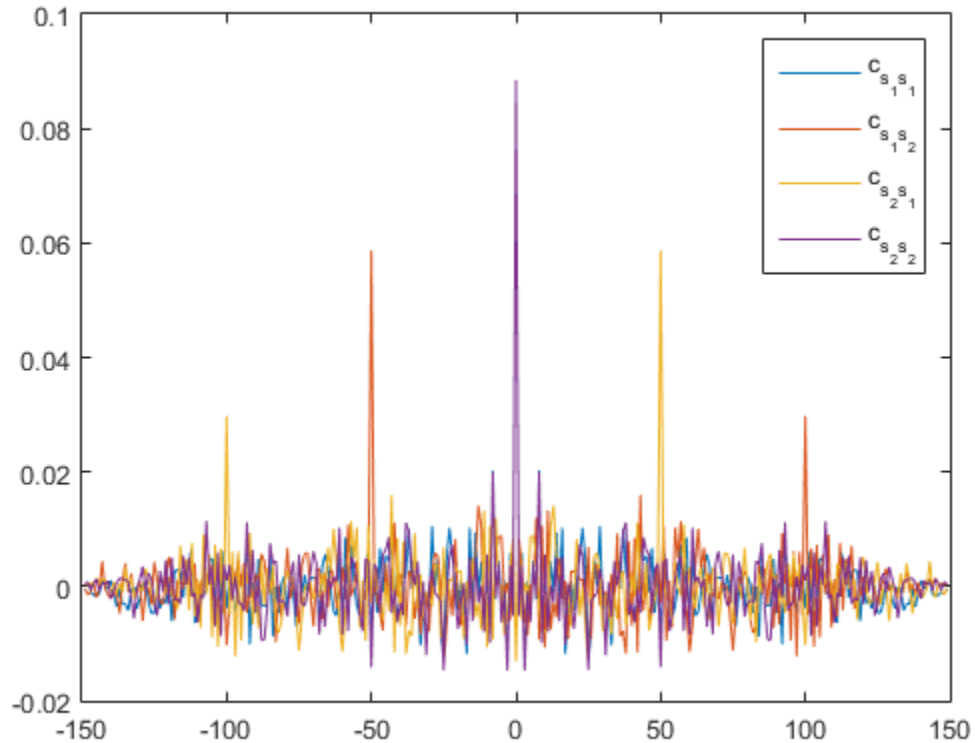


Compute biased estimates of the autocovariance and mutual cross-covariance sequences. The output array is organized as  $\mathbf{c} = (c_{s_1s_1} \ c_{s_1s_2} \ c_{s_2s_1} \ c_{s_2s_2})$ . Plot the result. The maxima at  $\mp 50$  and  $\pm 100$  are a result of the circular shift.

```
[c,lg] = xcov(s,'biased');

figure
plot(lg,c)
legend('c_{s_1s_1}','c_{s_1s_2}','c_{s_2s_1}','c_{s_2s_2}')
```

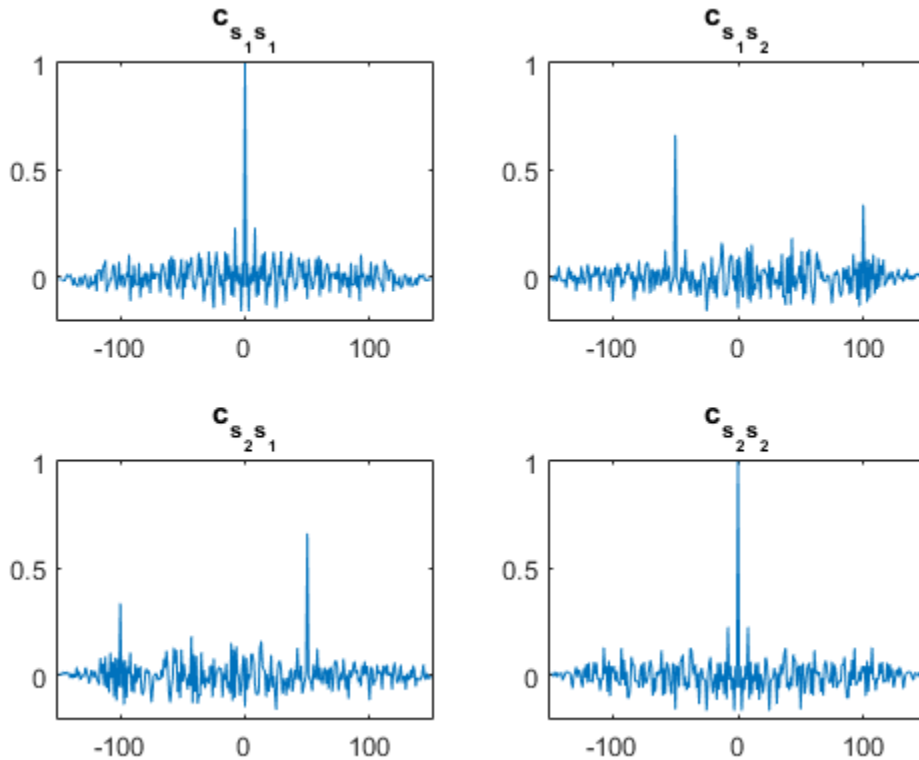




Change the normalization so that the cross-covariance and autocovariance sequences are unity at zero lag. Plot each sequence in its own subplot.

```
[c,lg] = xcov(s,'coeff');

for a = 1:2
    for b = 1:2
        nm = 2*(a-1)+b;
        subplot(2,2,nm)
        plot(lg,c(:,nm))
        title(sprintf('c_{s_%ds_%d}',a,b))
        axis([-150 150 -0.2 1])
    end
end
```



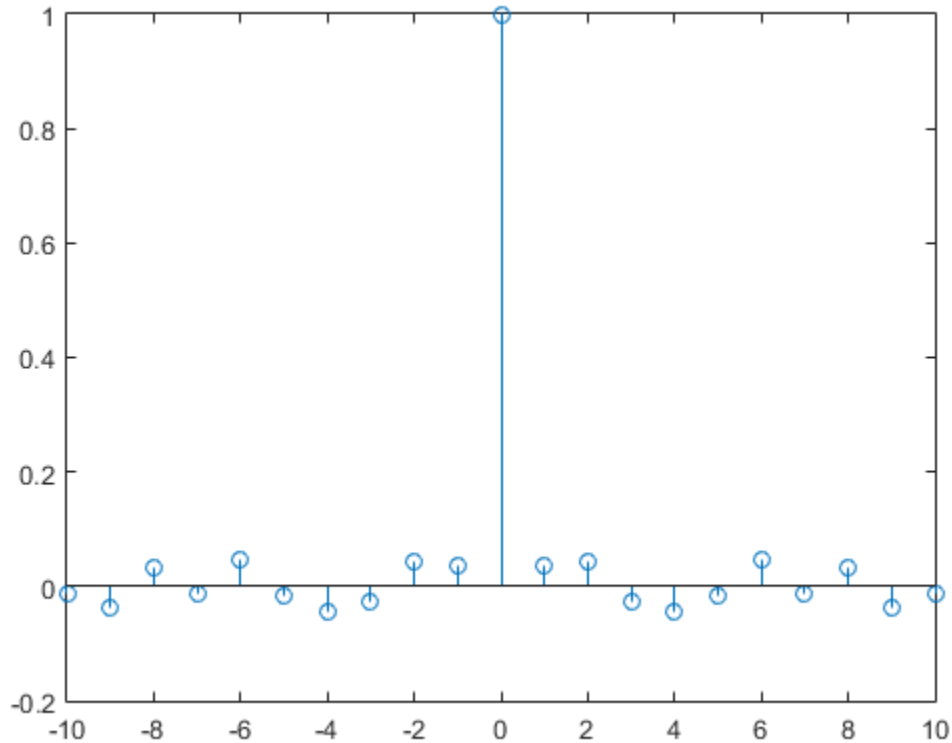
### Autocovariance of White Gaussian Noise

Display the estimated autocovariance of white Gaussian noise,  $c_{ww}(m)$ , for  $-10 \leq m \leq 10$ . Reset the random number generator for reproducible results. Normalize the sequence so that it is unity at zero lag.

```
rng default

ww = randn(1000,1);
[cov_ww,lags] = xcov(ww,10,'coeff');

stem(lags,cov_ww)
```



### GPU Acceleration for Autocovariance Sequence Estimation

This example requires Parallel Computing Toolbox software and a CUDA-enabled NVIDIA GPU with compute capability 1.3 or above. See GPU System Requirements for details.

Create a signal consisting of a 10 Hz sine wave in additive noise, sampled at 1 kHz. Use `gpuArray` to create a `gpuArray` object stored on your computer's GPU.

```
t = 0:0.001:10-0.001;  
x = cos(2*pi*10*t) + randn(size(t));  
X = gpuArray(x);
```

Compute the autocovariance sequence to lag 200.

```
[xc,lags] = xcov(X,200);
```

The output, `xc`, is a `gpuArray` object.

Use `gather` to transfer the data from the GPU to the MATLAB workspace as a double-precision vector.

```
xc = gather(xc);
```

- “Measuring Signal Similarities”
- “Accelerating Correlation with GPUs”

## Input Arguments

### **x** — Input array

vector | matrix | `gpuArray` object

Input array, specified as a vector, a matrix, or a `gpuArray` object.

See “GPU Computing” and GPU System Requirements for details on using `xcov` with `gpuArray` objects.

Example: `sin(2*pi*(0:9)/10) + randn([1 10])/10` specifies a noisy sinusoid as a row vector.

Example: `sin(2*pi*[0.1;0.3]*(0:39))' + randn([40 2])/10` specifies a two-channel noisy sinusoid.

Example: `gpuArray(sin(2*pi*(0:9)/10) + randn([1 10])/10)` specifies a noisy sinusoid as a `gpuArray` object.

Data Types: `single` | `double`

Complex Number Support: Yes

### **y** — Input array

vector | `gpuArray` object

Input array, specified as a vector or a `gpuArray` object.

Data Types: `single` | `double`

Complex Number Support: Yes

### **maxlag** — Maximum lag

integer scalar

Maximum lag, specified as an integer scalar. If you specify `maxlag`, the returned cross-covariance sequence ranges from  $-\text{maxlag}$  to  $\text{maxlag}$ . If you do not specify `maxlag`, the lag range equals  $2N - 1$ , where  $N$  is the greater of the lengths of  $x$  and  $y$ .

Data Types: `single` | `double`

### **scaleopt** — Normalization option

'none' (default) | 'biased' | 'unbiased' | 'coeff'

Normalization option, specified as one of these strings:

- 'none' — Raw, unscaled cross-covariance. This is the only allowed option when  $x$  and  $y$  have different lengths.
- 'biased' — Biased estimate of the cross-covariance.
- 'unbiased' — Unbiased estimate of the cross-covariance.
- 'coeff' — Normalizes the sequence so that the autocovariances at zero lag equal 1.

Data Types: `char`

## Output Arguments

### **c** — Cross-covariance or autocovariance sequence

vector | matrix | `gpuArray` object

Cross-covariance or autocovariance sequence, returned as a vector, a matrix, or a `gpuArray` object.

If  $x$  is an  $M \times N$  signal matrix representing  $N$  channels in its columns, then `xcov(x)` returns a  $(2M - 1) \times N^2$  matrix with the autocovariances and mutual cross-covariances of the channels of  $x$ . If you specify `maxlag`, then  $c$  has size  $(2 \times \text{maxlag} - 1) \times N^2$ .

For example, if  $S$  is a three-channel signal,  $S = (x_1 \ x_2 \ x_3)$ , then the result of  $C = \text{xcov}(S)$  is organized as

$$c = \begin{pmatrix} c_{x_1x_1} & c_{x_1x_2} & c_{x_1x_3} & c_{x_2x_1} & c_{x_2x_2} & c_{x_2x_3} & c_{x_3x_1} & c_{x_3x_2} & c_{x_3x_3} \end{pmatrix}.$$

### **lags** — Lag indices

vector

Lag indices, returned as a vector.

## More About

### Cross-Covariance and Autocovariance

`xcov` computes the mean of its inputs, subtracts the mean, and then calls `xcorr`. `xcov` does not check for any errors other than the correct number of input arguments. Instead, it relies on the error checking in `xcorr`.

The result of `xcov` can be interpreted as an estimate of the covariance between two random sequences or as the deterministic covariance between two deterministic signals.

The true cross-covariance sequence of two jointly stationary random processes,  $x_n$  and  $y_n$ , is the cross-correlation of mean-removed sequences,

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

where  $\mu_x$  and  $\mu_y$  are the mean values of the two stationary random processes, the asterisk denotes complex conjugation, and  $E$  is the expected value operator. `xcov` can only estimate the sequence because, in practice, only a finite segment of one realization of the infinite-length random process is available.

By default, `xcov` computes raw covariances with no normalization:

$$c_{xy}(m) = \begin{cases} \sum_{n=0}^{N-m-1} \left( x_{n+m} - \frac{1}{N} \sum_{i=0}^{N-1} x_i \right) \left( y_n^* - \frac{1}{N} \sum_{i=0}^{N-1} y_i^* \right), & m \geq 0, \\ c_{yx}^*(-m), & m < 0. \end{cases}$$

The output vector, `c`, has elements given by

$$c(m) = c_{xy}(m - N), \quad m = 1, \dots, 2N - 1.$$

The covariance function requires normalization to estimate the function properly. You can control the normalization of the correlation by using the input argument `scaopt`.

## References

- [1] Orfanidis, Sophocles J. *Optimum Signal Processing: An Introduction*. 2nd Edition. New York: McGraw-Hill, 1996.
- [2] Larsen, Jan. "Correlation Functions and Power Spectra." November, 2009. [http://www2.imm.dtu.dk/pubdb/views/edoc\\_download.php/4932/pdf/imm4932.pdf](http://www2.imm.dtu.dk/pubdb/views/edoc_download.php/4932/pdf/imm4932.pdf)

## See Also

conv | corrcoef | cov | xcorr | xcorr2

# yulewalk

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. All intermediate points must be 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  $z$ .

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

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.

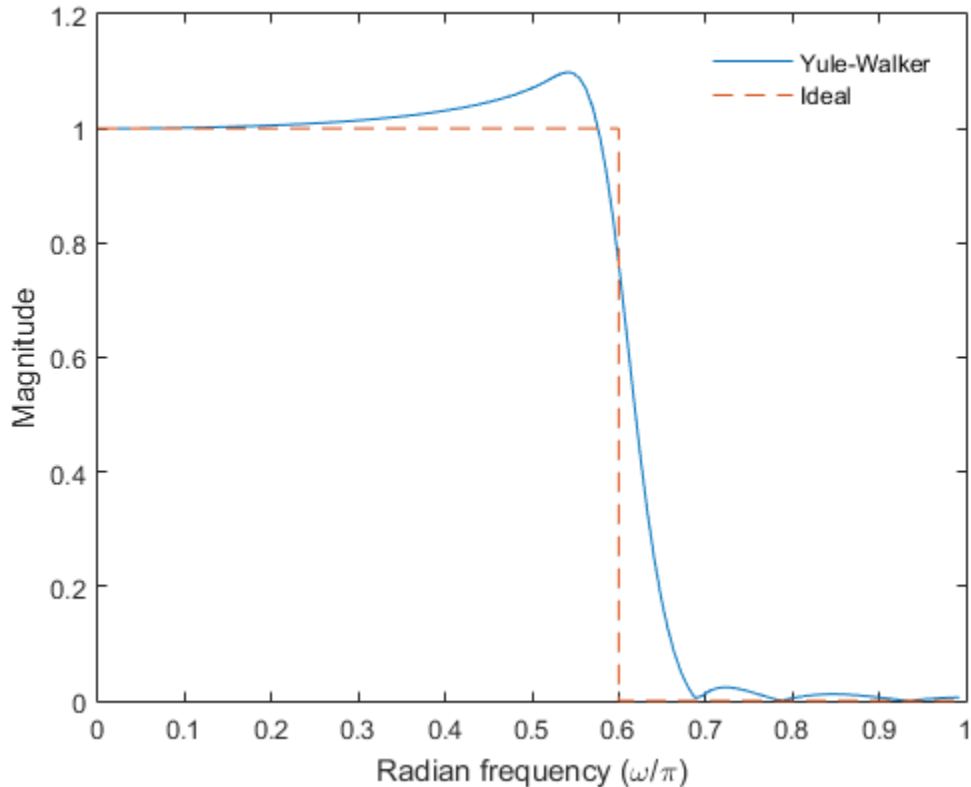


## Examples

### Yule-Walker Design of Lowpass Filter

Design an 8th-order lowpass filter with normalized cutoff frequency 0.6. Plot its frequency response and overlay the response of the corresponding ideal filter.

```
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(w/pi,abs(h),f,m,'--')  
xlabel 'Radian frequency (\omega/\pi)', ylabel Magnitude  
legend('Yule-Walker','Ideal'), legend boxoff
```



## More About

### Algorithms

`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 Boaz Porat. "The Modified Yule-Walker Method of ARMA Spectral Estimation." *IEEE Transactions on Aerospace Electronic Systems*. Vol. AES-20, Number 2, 1984, pp.158–173.

## See Also

butter | cheby1 | cheby2 | ellip | fir2 | firls | maxflat | firpm

## zerophase

Zero-phase response of digital filter

### Syntax

```
[Hr,w] = zerophase(b,a)
[Hr,w] = zerophase(sos)
[Hr,w] = zerophase(d)
[Hr,w] = zerophase(...,nfft)
[Hr,w] = zerophase(...,nfft,'whole')
[Hr,w] = zerophase(...,w)
[Hr,f] = zerophase(...,f,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,  $H_r(\omega)$ , is related to the frequency response,  $H(e^{j\omega})$ , by

$$H(e^{j\omega}) = H_r(\omega)e^{j\varphi(\omega)},$$

where  $\varphi(\omega)$  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(sos) returns the zero-phase response for the second order sections matrix, sos. sos is a  $K$ -by-6 matrix, where the number of sections,  $K$ , must be

greater than or equal to 2. If the number of sections is less than 2, **zerophase** considers the input to be the numerator vector, **b**. Each row of **sos** corresponds to the coefficients of a second order (biquad) filter. The *i*th row of the **sos** matrix corresponds to [**bi**(1) **bi**(2) **bi**(3) **ai**(1) **ai**(2) **ai**(3)].

[**Hr,w**] = **zerophase**(**d**) returns the zero-phase response for the digital filter, **d**. Use **designfilt** to generate **d** based on frequency-response specifications.

[**Hr,w**] = **zerophase**(...,**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. For best results, set **nfft** to a value greater than the filter order.

[**Hr,w**] = **zerophase**(...,**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**(...,**w**) returns the zero-phase response **Hr** and frequency vector **w** (radians/sample) at frequencies in vector **w**. The vector **w** must have at least two elements.

[**Hr,f**] = **zerophase**(...,**f**,**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. The vector **f** must have at least two elements.

[**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**(...) plots the zero-phase response versus frequency. If you input the filter coefficients or second order sections matrix, the current figure window is used. If you input a **digitalFilter**, the step response is displayed in **fvtool**.

---

**Note:** If the input to **zerophase** is single precision, the zero-phase response is calculated using single-precision arithmetic. The output, **Hr**, is single precision.

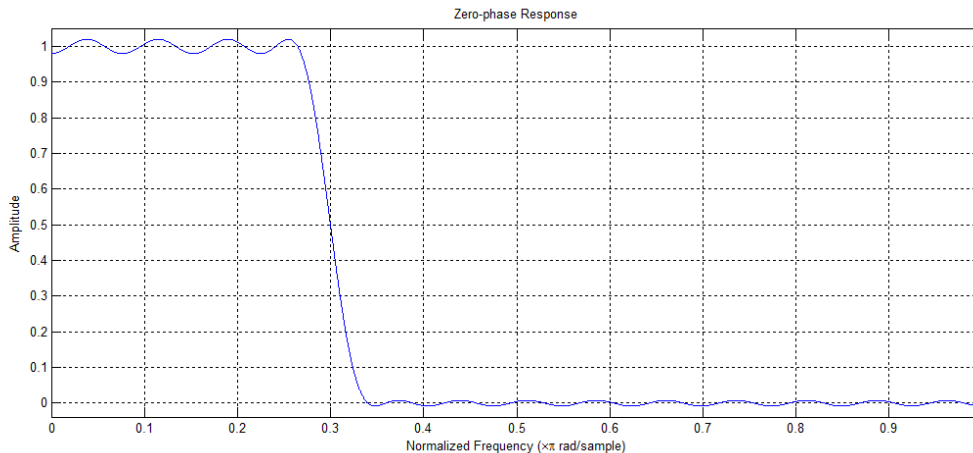
---

## Examples

### Zero-Phase Response of an FIR Filter

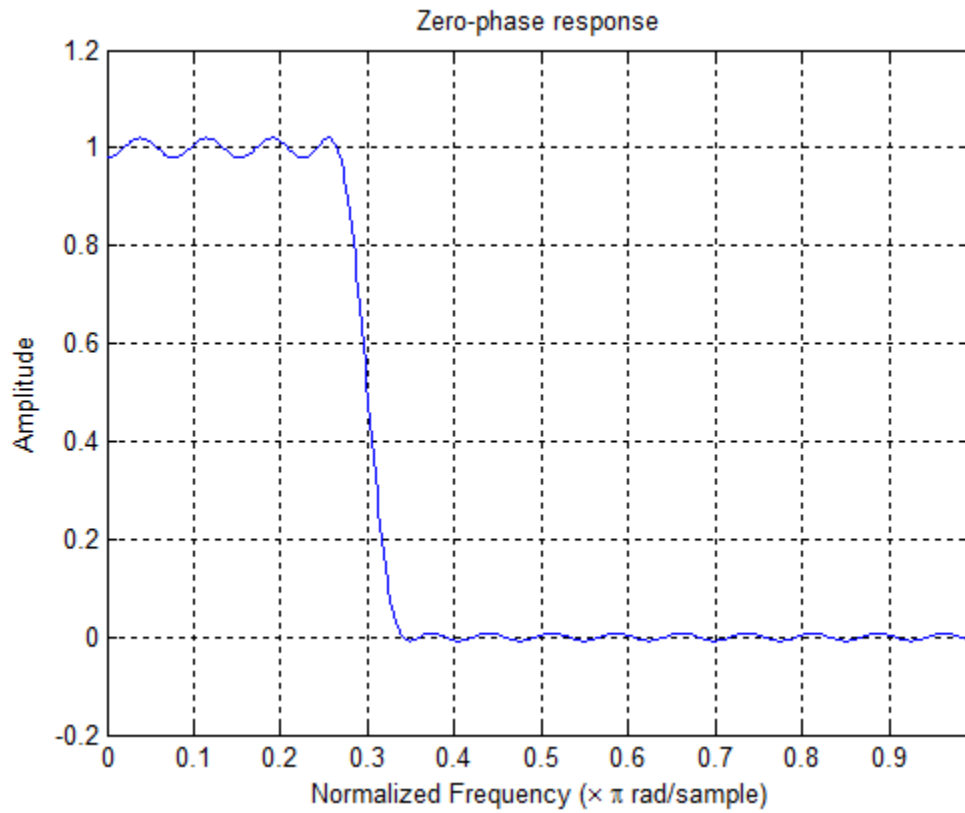
Use `designfilt` to design a 54th-order FIR filter with normalized cutoff frequency  $0.3\pi$  rad/sample, passband ripple 0.7 dB, and stopband attenuation 42 dB. Use the method of constrained least squares. Display the zero-phase response.

```
Nf = 54; Fc = 0.3; Ap = 0.7; As = 42;
d = designfilt('lowpassfir','FilterOrder',Nf,'CutoffFrequency',Fc, ...
    'PassbandRipple',Ap,'StopbandAttenuation',As,'DesignMethod','cls');
zerophase(d)
```



Design the same filter using `fircls1`. Keep in mind that `fircls1` uses linear units to measure the ripple and attenuation. Display the zero-phase response.

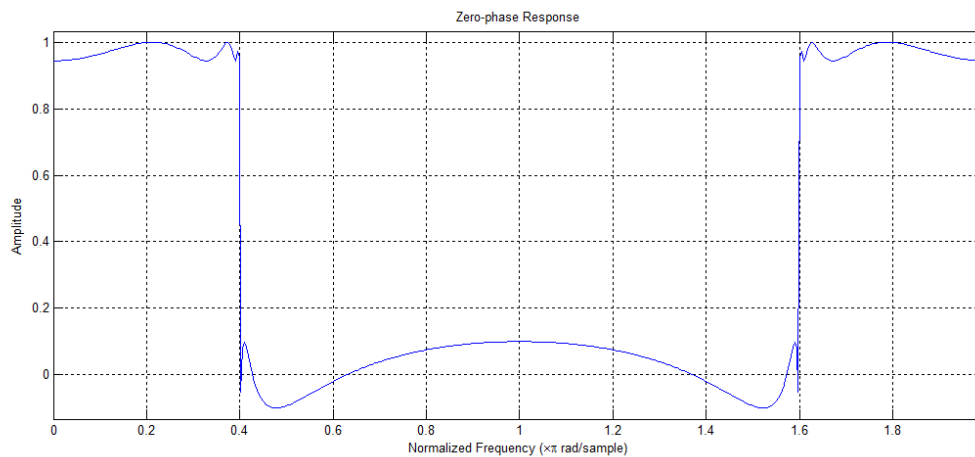
```
pAp = 10^(Ap/40); Apl = (pAp-1)/(pAp+1);
pAs = 10^(As/20); Asl = 1/pAs;
b = fircls1(Nf,Fc,Apl,Asl);
zerophase(b)
```



### Zero-Phase Response of an Elliptic Filter

Design a 10th-order elliptic lowpass IIR filter with normalized passband frequency  $0.4\pi$  rad/sample, passband ripple 0.5 dB, and stopband attenuation 20 dB. Display the zero-phase response of the filter on 512 frequency points around the whole unit circle

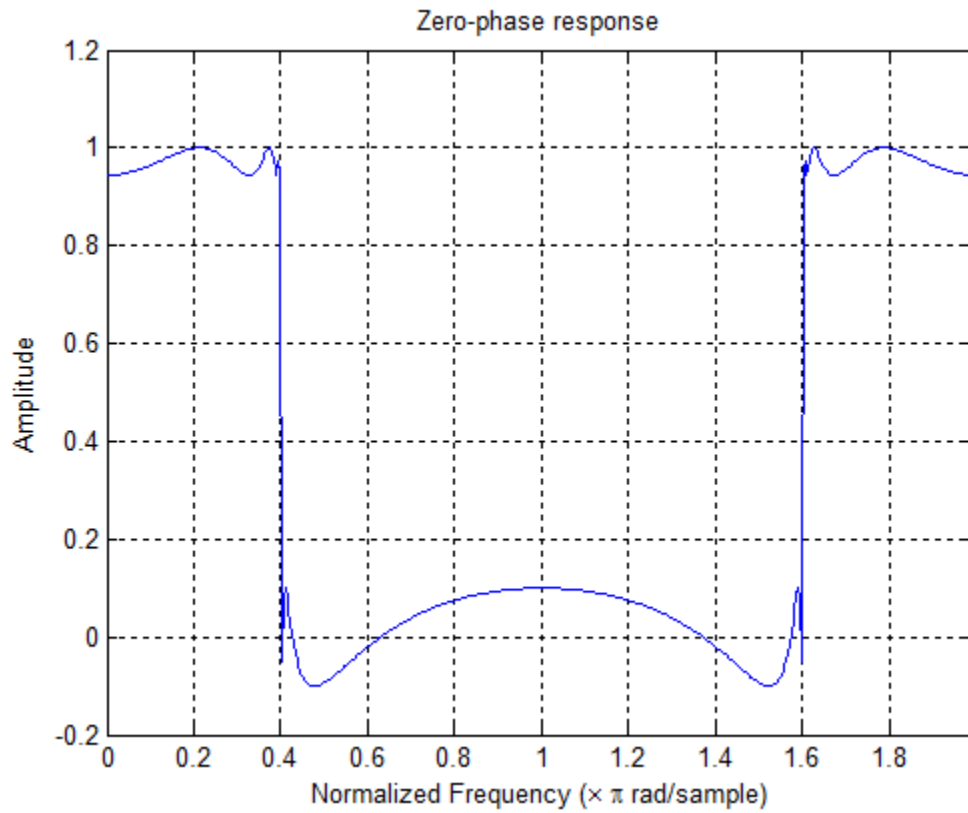
```
d = designfilt('lowpassiir','FilterOrder',10,'PassbandFrequency',0.4, ...
    'PassbandRipple',0.5,'StopbandAttenuation',20,'DesignMethod','ellip');
zerophase(d,512,'whole')
```



Create the same filter using `ellip`. Plot its zero-phase response.

```
[b,a] = ellip(10,0.5,20,0.4);  
zerophase(b,a,512,'whole')
```





### See Also

[designfilt](#) | [digitalFilter](#) | [freqs](#) | [freqz](#) | [fvtool](#) | [grpdelay](#) | [invfreqz](#) | [phasedelay](#) | [phasez](#)

## zp2sos

Convert zero-pole-gain filter parameters to second-order sections form

### Syntax

```
[sos,g] = zp2sos(z,p,k)
[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.

[sos,g] = zp2sos(z,p,k) creates 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

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

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

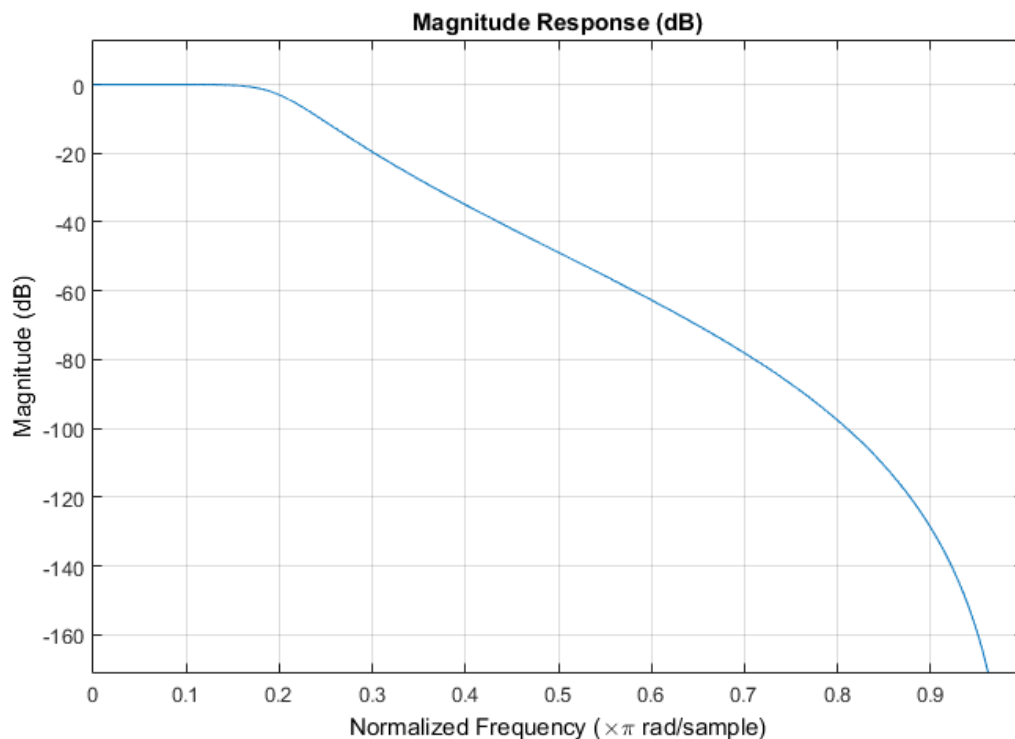
### Second-Order Sections from Zero-Pole-Gain Parameters

Design a 5th-order Butterworth lowpass filter using the function `butter` with output expressed in zero-pole-gain form. Specify the cutoff frequency to be one-fifth of the Nyquist frequency. Convert the result to second-order sections. Visualize the magnitude response.

```
[z,p,k] = butter(5,0.2);  
sos = zp2sos(z,p,k)  
fvtool(sos)
```

sos =

|        |        |        |        |         |        |
|--------|--------|--------|--------|---------|--------|
| 0.0013 | 0.0013 | 0      | 1.0000 | -0.5095 | 0      |
| 1.0000 | 2.0000 | 1.0000 | 1.0000 | -1.0966 | 0.3554 |
| 1.0000 | 2.0000 | 1.0000 | 1.0000 | -1.3693 | 0.6926 |



## More About

### Algorithms

`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

$$\|H\|_p = \left[ \frac{1}{2\pi} \int_0^{2\pi} |H(\omega)|^p d\omega \right]^{1/p}$$

where  $p$  can be either  $\infty$  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. Boston: Kluwer Academic Publishers, 1996, chap.11.
- [2] Mitra, S. K. *Digital Signal Processing: A Computer-Based Approach*. New York: McGraw-Hill, 1998, chap.9.
- [3] Vaidyanathan, P. P. "Robust Digital Filter Structures." *Handbook for Digital Signal Processing* (S. K. Mitra and J. F. Kaiser, eds.). New York: John Wiley & Sons, 1993, chap.7.

## See Also

`cplxpair` | `sos2zp` | `ss2sos` | `tf2sos` | `zp2ss` | `zp2tf` | `filternorm`

## zp2ss

Convert zero-pole-gain filter parameters to state-space form

### Syntax

$[A, B, C, D] = \text{zp2ss}(z, p, k)$

### Description

zp2ss converts a zero-pole-gain representation of a given system to an equivalent state-space representation.

$[A, B, C, D] = \text{zp2ss}(z, p, k)$  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.

## More About

### Algorithms

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



## zp2tf

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

$$\frac{B(s)}{A(s)} = \frac{b_1 s^{(n-1)} + \dots + b_{(n-1)} s + b_n}{a_1 s^{(m-1)} + \dots + a_{(m-1)} s + a_m}$$

given a system in factored transfer function form

$$H(s) = \frac{Z(s)}{P(s)} = k \frac{(s - z_1)(s - z_2) \dots (s - z_m)}{(s - p_1)(s - p_2) \dots (s - p_n)}$$

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.

## More About

### Algorithms

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`

# zpk

Convert digital filter to zero-pole-gain representation

## Syntax

```
[z,p,k] = zpk(d)
```

## Description

`[z,p,k] = zpk(d)` returns the zeros, poles, and gain corresponding to the digital filter, `d`, in vectors `z` and `p`, and scalar `k`, respectively.

## Examples

### Highpass Filter in Zero-Pole-Gain Form

Design a highpass FIR filter of order 8 with passband frequency 75 kHz and passband ripple 0.2 dB. Specify a sample rate of 200 kHz. Find the zeros, poles, and gain of the filter.

```
hpFilt = designfilt('highpassfir','FilterOrder',8, ...  
    'PassbandFrequency',75e3,'PassbandRipple',0.2, ...  
    'SampleRate',200e3);  
[z,p,k] = zpk(hpFilt)
```

```
z =
```

```
1  
1  
1  
1  
1  
1  
1  
1  
1
```

```
p =  
-0.6707 + 0.6896i  
-0.6707 - 0.6896i  
-0.6873 + 0.5670i  
-0.6873 - 0.5670i  
-0.7399 + 0.3792i  
-0.7399 - 0.3792i  
-0.7839 + 0.1344i  
-0.7839 - 0.1344i
```

```
k =  
1.2797e-05
```

## Input Arguments

### **d** — Digital filter

digitalFilter object

Digital filter, specified as a `digitalFilter` object. Use `designfilt` to generate a digital filter based on frequency-response specifications.

Example: `d = designfilt('lowpassiir','FilterOrder',3,'HalfPowerFrequency',0.5)` specifies a third-order Butterworth filter with normalized 3-dB frequency  $0.5\pi$  rad/sample.

## Output Arguments

### **z** — Zeros

column vector

Zeros of the filter, returned as a column vector.

Data Types: `double`

### **p** — Poles

column vector

Poles of the filter, returned as a column vector.

Data Types: double

**k – Gain**

real scalar

Gain of the filter, returned as a real scalar.

Data Types: double

**See Also**

`designfilt` | `digitalFilter` | `ss` | `tf`

## zplane

Zero-pole plot

### Syntax

```
zplane(z,p)
zplane(b,a)
zplane(d)
[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.

`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  $z^{-1}$ :

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

`zplane(d)` finds the zeros and poles of the transfer function represented by the digital filter, `d`. Use `designfilt` to generate `d` based on frequency-response specifications. The pole-zero plot is displayed in `fvtool`.

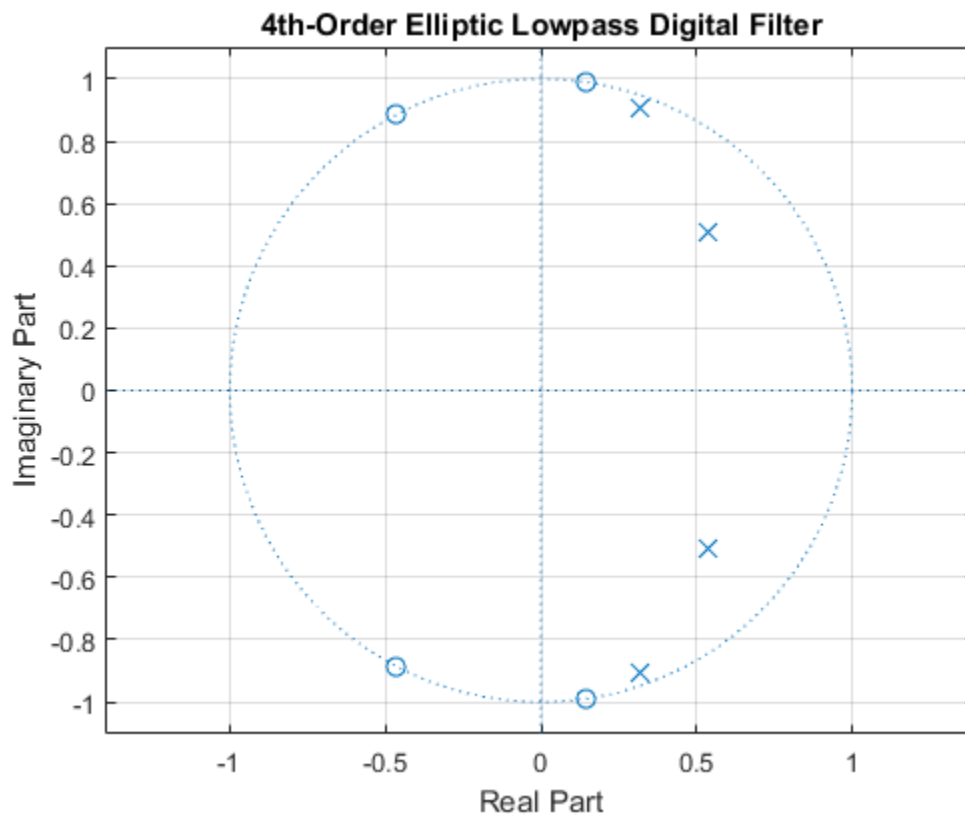
`[hz, hp, ht] = zplane(z,p)` returns vectors of handles to the zero lines, `hz`, and the pole lines, `hp`. `ht` is a vector of handles to the axes/unit circle line and to text objects, which are present when there are multiple zeros or poles. If there are no zeros or no poles, `hz` or `hp` is the empty matrix `[]`.

## Examples

### Poles and Zeros of an Elliptic Lowpass Filter

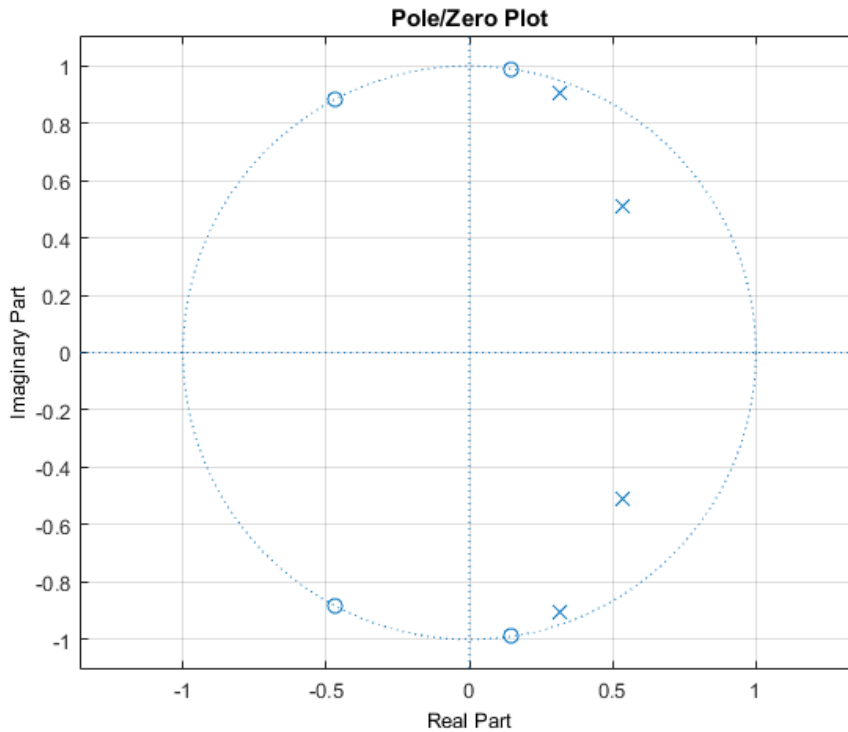
For data sampled at 1000 Hz, plot the poles and zeros of a 4th-order elliptic lowpass digital filter with cutoff frequency 200 Hz, 3 dB of ripple in the passband, and 30 dB of attenuation in the stopband.

```
[z,p,k] = ellip(4,3,30,200/500);  
zplane(z,p)  
grid  
title('4th-Order Elliptic Lowpass Digital Filter')
```



Create the same filter using `designfilt`. Use `fvtool` to plot its poles and zeros.

```
d = designfilt('lowpassiir', 'FilterOrder', 4, 'PassbandFrequency', 200, ...
              'PassbandRipple', 3, 'StopbandAttenuation', 30, ...
              'DesignMethod', 'ellip', 'SampleRate', 1000);
zplane(d)
```



## More About

### Tips

- You can override the automatic scaling of `zplane` using `axis([xmin xmax ymin ymax])`



after calling `zplane`. This is useful in cases where one or more of the zeros or poles have such a large magnitude that the others are grouped tightly around the origin and are hard to distinguish.

**See Also**

`designfilt` | `digitalFilter` | `freqz`

